



Escola de Tecnologias e Arquitectura
Departamento de Ciências e Tecnologias da Informação

Sistemas Operativos

Sebenta

João Baptista

Fernando Batista

Paulo Trezentos

8 de Setembro de 2018

Conteúdo

I	Comandos, Shell e Administração	1
1	Linha de comandos, comandos de utilizador	5
1.1	Ficheiros	5
1.2	Sistema de ficheiros	6
1.3	Ficheiros: nomes relativos e absolutos	7
1.4	Comandos relacionados com ficheiros e directorias	8
1.5	Elementos breves sobre a sintaxe dos comandos	10
1.6	Permissões	11
1.7	Ajuda	13
1.8	Expansão. "Wildcards"	13
1.9	Escapes	13
1.10	Redirecionamento	14
1.11	"Hello World"	15
2	Comandos de manipulação de texto	17
2.1	wc	17
2.2	head, tail	18
2.3	cut	18
2.4	sort, uniq	18
2.5	grep	19
2.6	sed	19
2.7	awk	21
2.8	Expressões regulares	22
2.9	Exercício	23
2.10	Outros exercícios	25
3	Programação em shell	27
3.1	Linha de comando	27
3.2	Elementos de programação	29
3.3	Comandos de apoio a scripts	38
3.4	Outros mecanismos. Complementos.	39
3.5	Exemplos	49
3.6	Elementos mais avançados	51
4	Tarefas de administração (Linux)	53
4.1	Gestão de contas	53
4.2	Agendamento de comandos (cron daemon)	55
4.3	Estrutura do sistema de ficheiros	57
4.4	Tarefas Comuns	58

II System Programming 61

5	Processos e sinais	63
5.1	Processos	63
5.2	Sinais	68
6	System V - IPCs	75
6.1	Filas de mensagens	75
6.2	Semáforos	85
6.3	Memória partilhada	92
6.4	Complementos	97
7	Input/Output	99
7.1	Funções de biblioteca	99
7.2	Chamadas do sistema para input/output (i/o)	100
7.3	IPC, pipes	103
7.4	Sistema de ficheiros	107
7.5	Terminal	107
8	Sockets	111
8.1	socket "unix"	111
8.2	socket "internet"	113
8.3	Exemplo com estruturas	115

III Anexos 119

9	Linguagem C	121
9.1	Introdução	121
9.2	Variáveis	121
9.3	Estruturas de controlo.	123
9.4	Expressões lógicas e condicionais	126
9.5	Funções	127
9.6	Arrays	128
9.7	Arrays de caracteres ("strings")	129
9.8	Leitura e escrita de caracteres	131
9.9	Ficheiros	133
9.10	Estruturas	136
9.11	Ficheiros binários	137
9.12	Ponteiros	140
9.13	Algumas situações com ponteiros	146
9.14	Exemplos	148
10	Editor de texto: vi	155
10.1	Elementos básicos	155
10.2	Mecanismos comuns de edição	156

Parte I

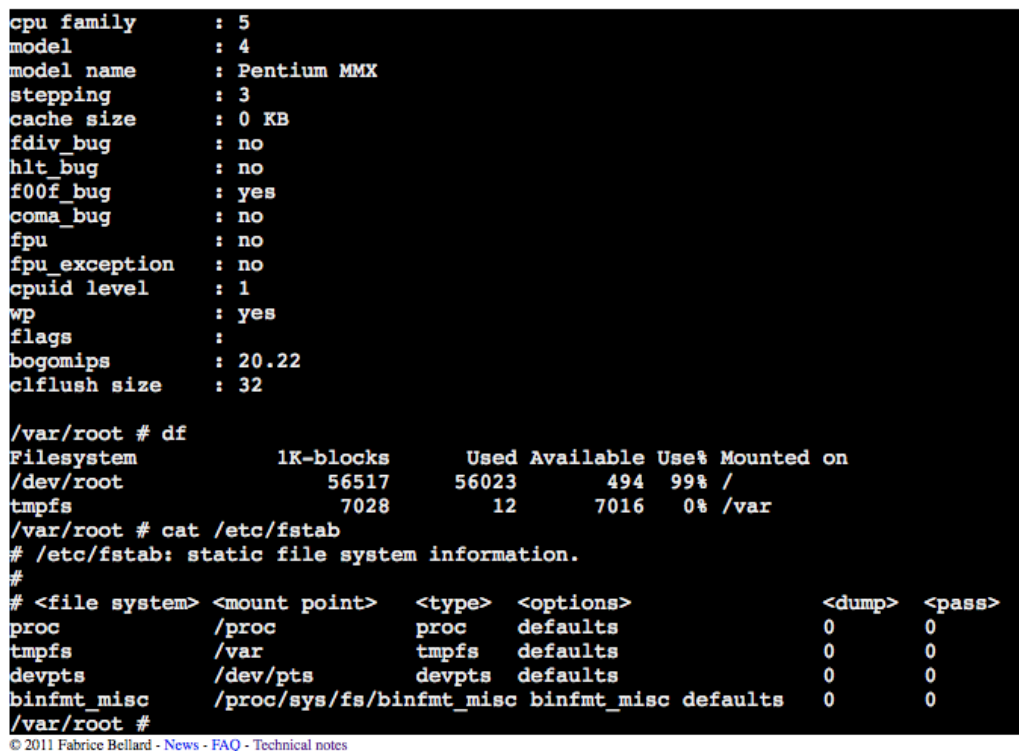
Comandos, Shell e Administração

Nota introdutória

Os elementos abordados nesta parte da sebenta podem ser testados na sua totalidade numa plataforma linux. Para os alunos do ISCTE-IUL está disponível o servidor tigre.iul.lab que pode ser usado dentro do ISCTE-IUL ou através de VPN caso se encontre fora do ISCTE-IUL. Aconselha-se a leitura do “Guia de Acesso ao Servidor Tigre” para uma rápida introdução à utilização do servidor.

Caso não tenha acesso a um servidor linux, pode testar ainda assim a grande maioria dos elementos, usando “emuladores” ou “servidores linux” disponíveis online. Estes sistemas online têm a particularidade de permitir a aprendizagem das ferramentas utilizando apenas um browser e são suficientemente versáteis. De seguida apresentam-se alguns desses sistemas.

- https://www.tutorialspoint.com/unix_terminal_online.php - Unix Terminal Online: easy interface to Linux operating system CentOS 7.
- <http://bellard.org/jslinux/> - Javascript PC Emulator. É uma boa solução para testar e compreender comandos. Permite a utilização comandos que normalmente estão apenas disponíveis no nível de super-utilizador. A sua principal limitação é ao nível da velocidade do processador, sendo pouco aconselhável para compilar e executar programas em C. O espaço disponível para trabalhar é limitado, contudo suficiente para executar praticamente tudo o que é mencionado nesta sebenta. A figura seguinte a apresenta um screenshot do sistema em funcionamento.



```
cpu family      : 5
model           : 4
model name      : Pentium MMX
stepping        : 3
cache size      : 0 KB
fdiv_bug        : no
hlt_bug         : no
f00f_bug        : yes
coma_bug        : no
fpu             : no
fpu_exception   : no
cpuid level     : 1
wp              : yes
flags           :
bogomips        : 20.22
clflush size    : 32

/var/root # df
Filesystem      1K-blocks    Used Available Use% Mounted on
/dev/root        56517      56023      494   99% /
tmpfs           7028         12      7016    0% /var
/var/root # cat /etc/fstab
# /etc/fstab: static file system information.
#
# <file system> <mount point> <type> <options>          <dump> <pass>
proc            /proc          proc    defaults            0       0
tmpfs           /var           tmpfs   defaults            0       0
devpts          /dev/pts       devpts  defaults            0       0
binfmt_misc     /proc/sys/fs/binfmt_misc binfmt_misc defaults            0       0
/var/root #
```

© 2011 Fabrice Bellard - [News](#) - [FAQ](#) - [Technical notes](#)

- <http://blinkenshell.org> - Free UNIX Shell Accounts
- <http://mobaxterm.mobatek.net>- Enhanced terminal for Windows with X11 server, tabbed SSH client, network tools and much more

1

Linha de comandos, comandos de utilizador

Teste o seguinte conjunto de comandos na linha de comandos e verifique os resultados obtidos.

```
w
whoami
pwd
cal
cal 2015
uname
uname -a
top    (usar a tecla 'q' para sair)
```

A generalidade dos comandos aceita opções e argumentos, como é o caso dos comandos *cal* e *uname*. As opções definem ou alteram o funcionamento normal do comando. Normalmente, escrevem-se a seguir ao comando, antecidos de sinal -. Por exemplo, a seguinte variante do comando *ls* é usada muitas vezes para listar os ficheiros:

```
ls -l
```

1.1 Ficheiros

Um ficheiro é caracterizado por um nome (presente num determinado directório), um conjunto de propriedades e um conteúdo. Para ver a *lista* de ficheiros que se encontram em determinado directório usa-se o comando:

```
ls
```

A maior parte das vezes para criar um novo ficheiro usamos um editor de texto. É normal numa instalação Linux encontrar vários editores de texto disponíveis, por exemplo o *vi* ou o *emacs*. O comando seguinte pode ser usado para criar um novo ficheiro de texto com o *vi*, sendo *novo.txt* o nome do ficheiro que se pretende criar.

```
vi novo.txt
```

Para criar um texto simples podemos fazer a seguinte sequência, depois de entrar no *vi*:

1. carregar em *i* passando o *vi* para o modo de edição de texto
2. escrever o texto
3. carregar na tecla *Esc*

4. escrever ZZ para sair do vi, gravando o conteúdo inserido.

Exercício 1. Use o vi para criar/alterar outros ficheiros de texto. O guia de utilização do vi, disponível no site, descreve os principais comandos e mecanismos de trabalho neste editor.

O comando cat permite ver o conteúdo de um ficheiro no ecrã. Por exemplo para ver o conteúdo do ficheiro novo.txt pode usar o comando:

```
cat novo.txt
```

Se o ficheiro for muito grande é mais prático usar, em alternativa, o comando more, que mostra o ficheiro página a página. Por exemplo o comando:

```
more /etc/passwd
```

mostra o ficheiro mencionado, começando por exibir apenas o conteúdo da "primeira página". Para mudar de página carrega na tecla de espaço. Se quiser sair carrega na tecla q (de "quit").

Exercício. Pode também experimentar executar: less /etc/passwd

1.2 Sistema de ficheiros

No Linux, o sistema de ficheiros encontra-se organizado numa árvore com directórios ("ramos") e ficheiros ("folhas"). O primeiro desses directórios representa-se por / e designa-se "raíz". Como se disse, o comando ls -l mostra o conteúdo de um directório. Por exemplo, o comando

```
ls -l /
```

mostra o conteúdo do directório / ("raíz") que será qualquer coisa do género:

```
drwxr-xr-x  2 root    root      4096 2004-01-13 16:48 bin
drwxr-xr-x  3 root    root      1024 2004-01-13 17:23 boot
drwxr-xr-x 29 root    root     61440 2004-09-18 11:26 dev
drwxr-xr-x 65 root    root      8192 2005-02-18 09:48 etc
drwxr-xr-x 28 root    root      4096 2005-02-04 15:28 home
drwxr-xr-x  7 root    root      4096 2003-09-09 18:37 lib
drwxr-xr-x  2 root    root      4096 2004-03-26 17:11 local
drwx----- 2 root    root     16384 2003-09-09 17:36 lost+found
drwxr-xr-x  5 root    root      4096 2003-03-18 17:49 media
drwxr-xr-x  2 root    root      4096 2003-03-18 17:49 mnt
drwxr-xr-x  4 root    root      4096 2003-09-11 10:19 net
drwxr-xr-x  8 root    root      4096 2003-09-09 18:10 opt
dr-xr-xr-x 77 root    root         0 2004-09-18 12:23 proc
drwx----- 16 root    root      4096 2005-02-18 09:48 root
drwxr-xr-x  3 root    root      8192 2003-11-01 16:36/sbin
drwxrwxrwt 31 root    root      8192 2005-02-28 13:45 tmp
drwxr-xr-x 12 root    root      4096 2003-09-09 17:47 usr
drwxr-xr-x 15 root    root      4096 2003-09-09 18:17 var
```

O ls -l assinala os directórios com um d na primeira coluna (os ficheiros normais são assinalados com -). Vemos assim que dentro deste directório / existem, por sua vez, vários outros directórios como o bin, o etc ou o home. Por exemplo, o comando seguinte mostra o conteúdo do directório /etc. Dentro deste existem por sua vez vários ficheiros e outros directórios, e assim sucessivamente formando a mencionada "árvore".

```
ls -l /etc
```

```

/
|-- etc
|   '-- passwd
|-- home
|   |-- a1000
|   |-- a1001
|   |   |-- notas.txt
|   |   '-- trabalho.java
|   '-- labso
|       |-- ac
|       |   '-- programa.mac
|       |-- geral.tab
|       '-- so
|           |-- ficheiro.txt
|           '-- outro_dir
|               '-- ficheiro.txt
|-- proc
|   '-- cpuinfo

```

Figura 1.3.1: Exemplo de uma estrutura de directórios.

1.3 Ficheiros: nomes relativos e absolutos

Considere o exemplo representado na Figura 1.3.1. Vemos que dentro do directório / existem, neste caso, mais três directórios: etc, home e proc. Por sua vez, dentro do directório home existem três outros directórios, com nomes a1000, a1001 e labso. Dentro deste último, existe um ficheiro de nome geral.tab e mais dois directórios chamados ac e so. E assim sucessivamente...

Um ficheiro é identificado pela sua posição na árvore de directórios, a partir da raiz. O nome completo de um ficheiro obtém-se traçando o caminho deste o directório / até ao ficheiro em causa. No traçado deste caminho os directórios percorridos são também separados por uma /. Assim, por exemplo, os seguintes nomes são válidos:

```

/home/a1001/trabalho.java
/home/labso/so/ficheiro.txt
/home/labso/so/outro_dir
/home/labso/so/outro_dir/ficheiro.txt

```

Note a diferença entre a primeira / e as seguintes. A primeira representa o nome do directório principal ("raiz"). As seguintes separam os directórios percorridos no percurso até ao ficheiro.

Na realidade, os nomes de ficheiros podem ser dados de duas maneiras:

- **absoluto**: partindo da / ("raiz")
- **relativo**: partindo do directório corrente actual.

O nome é **absoluto** (ou "completo") se começar por /; caso contrário é **relativo**. Para indicar um nome absoluto traça-se o caminho da raiz até ao ficheiro. Os exemplos anteriores são todos nomes absolutos. Para indicar um nome relativo traça-se, na mesma, o caminho até ao ficheiro, mas partindo do directório corrente. Para este efeito são válidos os seguintes elementos sintácticos:

- . directório corrente
- .. directório de cima (do nível anterior)

Exemplo. seja o directório corrente /home/labso/so. Nestas condições:

```
ficheiro.txt           m.q.   /home/labso/so/ficheiro.txt
outro_dir/ficheiro.txt m.q.   /home/labso/so/outro_dir/ficheiro.txt
../.. /a1001/trabalho.java m.q.   /home/a1001/trabalho.java
```

1.4 Comandos relacionados com ficheiros e directorias

Directório corrente

O directório corrente é o directório de trabalho em cada momento. Para saber qual o directório corrente actual usa o comando `pwd` (de "print working directory"). Por exemplo:

```
tigre:~> pwd
/home/a88888 //indica que o directório corrente actual é /home/a88888
```

Para mudarmos de directório corrente usa-se comando `cd` (de change directory). Por exemplo:

```
tigre:~> cd /home/labso // directório corrente passa a ser /home/labso
tigre:~> pwd           // confirmação com pwd
/home/labso
tigre:~> cd ..         // directório corrente passa a ser o "de cima"
tigre:~> pwd           // confirmação com pwd
/home
```

O directório corrente inicial quando entra no sistema, depois de fazer o login, designa-se directório de utilizador (ou "de trabalho").

Exercício. O directório de utilizador é indicado no ficheiro `/etc/passwd`. Localize o seu "username" neste ficheiro e o confirme o respectivo directório de utilizador.

Este directório é também, por vezes, relacionado com o conceito de "área de trabalho": é um directório que pertence ao utilizador e que o utilizador pode organizar internamente como entender (possivelmente criando uma sub-árvore de directórios) para guardar os seus ficheiros. O directório de utilizador é designado simbolicamente por `~`. Tal como o `..` e o `.`, este símbolo representa o nome de um directório e pode ser usado como tal na generalidade dos comandos. Assim, por exemplo:

```
cp /etc/passwd ~
```

copia o ficheiro `/etc/passwd` para o directório de utilizador, onde fica com o mesmo nome (`passwd`). Por omissão os comandos de manipulação de ficheiros operam no directório corrente. Por exemplo, fazendo a sequência:

```
cd /etc
ls -l
```

o `ls -l` lista o conteúdo do directório corrente, na circunstância o `/etc`. O mesmo resultado teria o comando

```
ls -l /etc
```

onde se explicita o mesmo directório.

Exercício. Qual o efeito de cada uma das seguintes variantes do `ls` ? Qual delas dá o mesmo resultado que, apenas: `ls -l`

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

Copiar, mover e apagar: `cp`, `mv` e `rm`

O comando `cp` (de "copy") permite copiar um ficheiro para outro. Em caso de sucesso a operação cria um novo ficheiro com nome diferente e conteúdo igual ao do ficheiro original. Por exemplo, admitindo que o directório de trabalho é `/home/labso`, o comando

```
cp geral.tab clone.tab
```

cria um novo ficheiro, com nome `clone.tab`, no mesmo directório do ficheiro original.

Exercício. Considere a árvore de ficheiros do ponto 1.3. Qual o efeito dos seguintes comandos:

```
cd /home/labso/so  
cp /etc/passwd .  
cp /etc/passwd ..  
cp /etc/passwd outro_dir/pass  
cp ficheiro.txt ..  
cp ../ac/programa.mac /home/labso
```

O comando `mv` (de "move") permite alterar o nome de um ficheiro. Por exemplo o comando:

```
mv programa.mac teste.mac
```

altera o nome de um ficheiro, de `programa.mac` para `teste.mac`, mantendo-se o ficheiro no mesmo directório. Já o comando:

```
mv programa.mac ..
```

movimenta o ficheiro de um directório para outro - neste caso, concretamente, para o directório de cima. Em qualquer dos casos o efeito é o mesmo é criado um novo ficheiro com outro nome (completo) e destruído o ficheiro original.

Exercício. Considere a árvore de ficheiros do ponto 1.3. Qual o efeito dos seguintes comandos:

```
cd /home/labso/so  
mv ficheiro.txt ../a1000/copia.txt  
mv ../ac/programa.mac ../ac/teste.mac
```

O comando `rm` (de "remove") permite eliminar um ficheiro. Por exemplo, o comando

```
rm programa.txt
```

apaga o ficheiro mencionado (o ficheiro com nome `programa.txt` no directório corrente).

links: `ln` / `ln -s`

O comando `ln` permite criar links entre ficheiros. O link é uma espécie de atalho ou “shortcut”. Na forma simples, `ln`, permite criar os chamados “hard link”.

```
ln ficheiro.txt copia.txt
```

A linha anterior cria um *hard link* `copia.txt` para o ficheiro original `ficheiro.txt`. Um *hard link* é parecido com uma cópia no sentido em que passam a existir dois nomes; mas apenas o nome é duplicado, o conteúdo é único e comum aos dois ficheiros. Se alterar o `ficheiro.txt` verá a mesma alteração reflectida no `copia.txt`. O comando `ls -l` que, como vimos, lista o conteúdo de um directório mostra também o número de “links” que cada ficheiro contém. Depois do comando anterior ambos os ficheiros (`ficheiro.txt` e `copia.txt`) ficarão com 2 links. Se apagar um dos nomes, por exemplo

```
rm copia.txt
```

não apaga o conteúdo do ficheiro, apenas um dos links. O número de links do `ficheiro.txt` diminuirá, em correspondência, de novo para 1.

A forma `ln -s` do mesmo comando cria os designados “soft link”. Um *soft link* é um tipo de ficheiro especial que “aponta” para um outro ficheiro. Por exemplo:

```
ln -s /etc/passwd soft
```

cria no directório corrente um ficheiro de nome `soft` para que “aponta” para o ficheiro `/etc/passwd`. Se vir agora a lista de ficheiros com `ls -l` fica clara a natureza especial do ficheiro `soft` e o ficheiro para que “aponta”. A criação de um “soft link” é idêntica à criação de um “atalho” (alias) no Windows.

Exercício. Crie um novo ficheiro e um “soft link” para ele. Qual o efeito no “soft link” se apagar o ficheiro original?

directórios

O comando `mkdir` (de “make directory”) permite criar um novo directório.

```
mkdir pasta
mkdir pasta/sub_pasta
```

Por exemplo os comandos anteriores criam um novo directório de nome `pasta` no directório corrente e um outro, de nome `sub_pasta`, dentro desse.

O comando `rmdir` apaga um directório que, para esse efeito, deve estar vazio. Para apagar um directório não vazio, utilizamos `rm -rf`. Este comando é muito perigoso e deve ser utilizado com cuidado.

1.5 Elementos breves sobre a sintaxe dos comandos

A generalidade dos comandos aceita opções e argumentos. As opções definem ou alteram o funcionamento normal do comando. Normalmente, escrevem-se a seguir ao comando, antecidos de sinal `-`. Por exemplo, a seguinte variante do comando `ls` é usada muitas vezes para listar os ficheiros:

```
ls -l
```

Os argumentos são os dados sobre o qual o comando trabalha. Por exemplo:

```
ls -l /etc
```

tem um argumento `/etc`. O comando lista o conteúdo de um directório - justamente o que é indicado no argumento.

Em muitos casos os argumentos podem ser identificados por omissão. Por exemplo

```
ls -l
```

não tem argumento explícito. Por omissão vale o directório corrente, ou seja dá o mesmo que:

```
ls -l .
```

A generalidade dos comandos tem variantes mais ou menos intuitivas. Por exemplo, muitos comandos aceitam vários ficheiros em lugar de apenas um. Assim, por exemplo:

```
cp ficheiro.txt copia.txt ..
```

`copia` dois ficheiros existentes no directório corrente para o directório de cima.

O comando `rm` tem algumas variantes de interesse. De um lado uma defensiva `rm -i` que pede uma confirmação antes de apagar o ficheiro. Ao contrário, a variante `rm -r`, útil mas particularmente perigosa, apaga recursivamente todos os ficheiros de um directório e dos subdirectórios nele contidos.

1.6 Permissões

As permissões de um ficheiro dividem-se em três grupos:

- *dono* (owner), *grupo* (group) ou *outros* (others)

Para cada um destes três grupos podem ser especificadas três operações:

- *r* (ler), *w* (escrever, alterar) ou *x* (executar).

O comando `ls -l` mostra as permissões de cada ficheiro nos 9 caracteres que se situam a seguir à primeira coluna (que como se mencionou anteriormente indica a natureza do "nome" presente no directório: por exemplo - para um ficheiro normal, `d` para um directório ou `l` para um "soft link"). Mostra também os nomes do dono e do grupo a que o ficheiro pertence.

Do conjunto dos 9 caracteres que representam as permissões o primeiro grupo de 3 caracteres indica as permissões que se aplicam ao dono do ficheiro. O segundo grupo indica as permissões que se aplicam aos utilizadores que pertencem ao grupo do ficheiro. E o terceiro grupo indica as permissões que se aplicam ao conjunto dos utilizadores. Assim, por exemplo, se tiver: `rw-r-----`, isso significa que ao dono do ficheiro aplicam-se as permissões `rw-`, ao grupo aplicam-se as permissões `r--` e ao conjunto de todos os utilizadores aplicam-se as permissões `---`.

```

rw-  r--  ---
  {   {   {
  user group others

```

Relativamente a ficheiros comuns as permissões indicam o direito de:

- r: ler (ver) o conteúdo do ficheiro;
- w: escrever (alterar) o conteúdo do ficheiro;
- x: executar o ficheiro como um programa.

Assim, por exemplo se tiver as permissões rw- isso significa que é permitido ler o ficheiro, escrever (alterar) o ficheiro mas não executá-lo. Se for r - - - pode-se apenas ler. Se for - - - não se pode fazer nenhuma das operações. Se o ficheiro for um "programa" poderá por exemplo, ter permissões r-x: pode-se ler e executar mas não alterar.

Relativamente a directórios as permissões indicam o direito de:

- r,x: poder "ler" (listar) o conteúdo do directório, ou seja, saber a lista dos ficheiros nele contidos;
- w: poder alterar ou seja criar/apagar nomes de ficheiros nesse directório;

Assim por exemplo, se um directório tiver permissões r-x pode-se "entrar" nele, fazer `ls -l` sobre ele mas não alterá-lo: ou seja não se podem criar ou apagar ficheiros nesse directório. Se tiver rwx podem-se fazer todas estas operações e se tiver - - - não se podem fazer quaisquer operações.

O dono de um ficheiro (ou directório) pode mudar as respectivas permissões usando o comando `chmod` (de "change mode"). Por exemplo, o comando seguinte muda as permissões do ficheiro `novo.txt` de acordo com o padrão indicado 744.

```
chmod 744 novo.txt
```

Este padrão indica as permissões através de 3 algarismos em octal, o primeiro para o dono, o segundo para o grupo e o terceiro para todos os utilizadores. O algarismo octal é formado fazendo corresponder a permissão r ("read") ao 4 a permissão w ("write") ao 2 e a permissão x ("execute") ao 1. Assim, por exemplo, r-x corresponde a 4+0+1 ou seja 5. Ao 6, ou seja, 4+2+0 corresponde rw-. Exemplos:

```
rw-rw---- → 760
rwxr-xr-x → 755
```

No exemplo anterior, com 744 estamos então a atribuir ao ficheiro as permissões:

```
rw-r--r--
```

Observação. Entretanto, o comando `chmod` tem outra forma sintáctica em que as permissões são indicadas através de letras. Por exemplo, o comando seguinte acrescenta a permissão x em todos os grupos. Veja a página do manual para mais pormenores.

```
chmod +x
```


1.7 Ajuda

Para saber mais pormenores sobre o funcionamento de cada comando pode consultar o manual do sistema usando o comando `man`. Por exemplo, o comando seguinte mostra a página do manual correspondente ao comando `ls`.

```
man ls
```

Exercício. Procure no manual uma variante do `ls`, que mostre os ficheiros por ordem cronológica inversa.

Por vezes, não sabemos o comando mas sabemos a operação que queremos efectuar. Por exemplo, se quisermos copiar um ficheiro, o comando “`man copy`” não devolveria a informação pretendida. Se precisarmos de pesquisar um comando pela sua descrição (por exemplo, `copy`), podemos fazer:

```
apropos copy
```

O `apropos` devolve a lista de comandos em cuja descrição esteja a palavra especificada. Um comando equivalente seria: `man -k copy`

1.8 Expansão. "Wildcards"

Um dos mecanismos mais usados da shell é a possibilidade de "apanhar" nomes de ficheiros usando o `"*"` ou, mais raramente, o `"?"`. De forma simples, o significado do `"*"` é "qualquer coisa". Mais exactamente, ao ler um nome com `*` a interpretação é: no lugar o `*` pode figurar qualquer sequência de caracteres, seja vazio ("nada"), um caractere qualquer, dois, etc. A utilização mais vulgar do `*` é no `ls`. Por exemplo,

```
ls -l *.txt
```

lista os nomes de ficheiro cuja nome adira ao padrão "qualquer coisa seguido de `.txt`" (ou seja, os ficheiros com extensão `.txt`). Mas pode ser usado em muitas outras circunstâncias como forma de apanhar um grupo de ficheiros. Por exemplo o comando seguinte cria um ficheiro `zip` com todos os `.txt` existentes na diretoria corrente.

```
zip textos.zip *.txt
```

1.9 Escapes

O `*` e o `?` são caracteres especiais para a shell. Como estes há outros, por exemplo `$`, `&` ou `>`. Se tivermos a ideia não muito feliz, de dar nomes a ficheiros contendo caracteres deste tipo podemos ter alguns problemas. Por exemplo, supondo que tínhamos a ideia não muito feliz de chamar a um ficheiro

```
***star***.txt
```

a utilização simples do nome do ficheiro em comandos, por exemplo

```
vi ***star***.txt
```

seria ambígua. Nesta situação teríamos que enquadrar o nome do ficheiro entre ' ':

```
vi '***star***.txt'
```

O mesmo aliás se usarmos nomes de ficheiros com espaços. Por exemplo se tivermos a ideia também não muito boa de chamar a um ficheiro

```
exemplo 1.txt
```

a utilização simples do nome do ficheiro, por exemplo:

```
vi exemplo 1.txt
```

é também ambígua e resolve-se com

```
vi 'exemplo 1.txt'
```

1.10 Redirecionamento

A maioria dos comandos mandam o resultado para o ecrã. Por exemplo, o resultado do comando seguinte, que é a lista de ficheiros existentes no directório corrente, vai para o ecrã.

```
ls -l
```

Se fizermos a seguinte variante nada aparece no ecrã. O resultado do comando vai para (é redirecionado) o ficheiro lista.txt

```
ls -l > lista.txt
```

Este é um exemplo de um mecanismo geral que a shell implementa e que permite redirecionar os canais *standard* de entrada e saída de um programa. A forma básica deste mecanismo é o normal redirecionamento da saída, por exemplo:

```
tigre:~> echo "data de hoje" > x    # saída para o ficheiro x
tigre:~> date >> x                  # acrescenta ao ficheiro x
tigre:~> echo "adeus" >> x          # volta a acrescentar
tigre:~> cat x
data de hoje
Sat, Oct 27, 2012  5:37:44 PM
adeus
```

Existe também o canal "standard de entrada" (*stdin*) que normalmente está associado ao teclado. Nos exemplos seguintes, i) é criado um ficheiro x; ii) o ficheiro x é usado como canal de entrada para o comando cat; iii) copia-se o conteúdo do ficheiro x, para um novo ficheiro y.

```
echo "teste" > x
cat < x
cat < x > y          ou          cat > y < x
```

Finalmente o `|` (designado "pipe") permite encadear comandos fazendo com que o standard output de um comando seja encaminhado para input do programa seguinte. O exemplo seguinte mostra o encadeamento de 3 comandos: o primeiro mostra o conteúdo do ficheiro `/etc/passwd`; o segundo filtra esse conteúdo mostrando apenas as linhas que contêm a palavra `root`; o terceiro conta o número de linhas resultantes.

```
cat /etc/passwd | grep "root" | wc -l
```

Normalmente a saída de um programa vai para o canal *standard de saída* (stdout) ou, em caso de erro, para o canal *standard de erros* (stderr). Um e outro podem ser redirecionados separadamente. Os descritores de ficheiro pré-definidos estão definidos na tabela seguinte:

dispositivo	descrição	descritor de ficheiro
stdin	entrada standard	0
stdout	saída standard	1
stderr	erro standard	2

No exemplo seguinte, a saída normal é redirecionada para o ficheiro `resultado.txt` e a de erros continua no ecrã. Se o ficheiro `x` não existir, a mensagem de erro vai para o ecrã;

```
cat x 1> resultado.txt # ou apenas: cat x > resultado.txt
```

No exemplo seguinte, a saída de erros é redirecionada para o ficheiro `erros.txt`. Os erros que aconteçam (por exemplo, se o ficheiro não existir) não aparecem no ecrã, vão todos para o ficheiro `erros.txt`

```
cat x 2> erros.txt
```

No exemplo seguinte, a saída de erros é redirecionada para `/dev/null`. Se o ficheiro existir, aparece no ecrã; caso contrário, não aparece nada no ecrã;

```
cat x 2> /dev/null
```

O exemplo seguinte, redirecciona a saída standard para `/dev/null` e a saída de erros para o mesmo sítio;

```
cat x > /dev/null 2>&1
```

Resta acrescentar que `/dev/null` é um ficheiro muito especial: representa um "poço sem fundo" (ou buraco negro) para onde se pode redireccionar qualquer output, que é descartado pelo Unix (e Linux);

1.11 "Hello World"

Considere o seguinte script, muito simples, que apenas escreve a mensagem "Hello World!" e "Adeus"

```
#!/bin/bash
echo "Hello world!"
echo -n "data e hora: "
date
echo "Adeus"
```

Para executar este script tem que:

1. escrever o programa num ficheiro de texto com extensão `.sh`; por exemplo `hello.sh`. Pode fazer isso em qualquer editor de texto como o `vi`, `emacs`, ou `kwrite`.
2. mudar as permissões do ficheiro

```
chmod +x hello.sh
```

3. executar o script, indicando que ele se encontra na diretório atual.

Tipicamente na consola (janela de comandos) seria feita a seguinte sequência:

```
vi hello.sh           # escrever o programa
chmod +x hello.sh     # mudar as permissões
./hello.sh            # executar o script
```

2

Comandos de manipulação de texto

Este capítulo apresenta alguns comandos que permitem processar e manipular ficheiros de texto. Estes comandos são de grande utilidade no dia-a-dia e são frequentemente utilizados no apoio a scripts.

Considere o ficheiro `f1.txt`, usado nos exemplos apresentados neste capítulo, com o seguinte conteúdo:

```
Manuel sporting 0001-1002-12345234234-11 sim 110.0
Pedro benfica 0010-0302-00005234234-22 não 120.0
Maria porto 0011-0333-00008989898-33 Sim 3333.5
Vanessa sporting 0100-0444-00004444444-33 sim 44.4
```

2.1 wc

Devolve indicações sobre a dimensão de um ficheiro: o número de linhas (`-l`), palavras (`-w`) e caracteres (`-c`). Por exemplo, a opção `-l` dá apenas o número de linhas. O exemplo seguinte conta letras, palavras e linhas num ficheiro (neste caso devolve: 4 20 198)

```
cat f1.txt | wc
```

O exemplo seguinte conta o número de utilizadores registados no sistema

```
cat /etc/passwd | wc -l
```

O exemplo seguinte conta o número de palavras numa frase (neste caso devolve 7)

```
echo "Batem batem levemente, como quem chama por mim" | wc -w
```

Pode também guardar o resultado numa variável e mostrá-lo mais tarde

```
letras=$(echo "Batem batem levemente" | wc -c)
echo "A frase tem $letras caracteres"

x=$(ls -l | wc -l)
echo "existem $x ficheiros na directoria actual"
```

Exercício 2. Comando que calcule quantos ficheiros existem no diretório `/etc`.

2.2 head, tail

Mostra as primeiras/últimas linhas. Caso não sejam indicadas opções assume que são as 10 primeiras/últimas linhas. Exemplos:

```
cat f1.txt | head
```

mostra as primeiras 10 linhas de um ficheiro. Neste caso mostra o ficheiro todo.

```
cat f1.txt | tail -2 > f2.txt
```

cria um ficheiro f2.txt com as 2 últimas linhas de f1.txt.

```
cat f1.txt | tail -n +2
```

mostra as linhas do ficheiro, começando na linha 2 e até ao final.

```
cat f1.txt | head -3 | tail -2
```

mostra as linhas 2 e 3 do ficheiro.

Os comandos seguintes operam no ficheiro /etc/passwd, devolvendo: 1) as primeiras 5 linhas; 2) as últimas 5 linhas; 3) as últimas linhas a partir da linha 10 até ao fim do ficheiro.

```
cat /etc/passwd | head -5  
cat /etc/passwd | tail -5  
cat /etc/passwd | tail -n +10
```

Exercício 3. Comando que permita obter as duas penúltimas linhas de um ficheiro.

2.3 cut

Apaga ou seleciona as colunas de um ficheiro

```
cat f1.txt | cut -d ' ' -f3
```

mostra apenas a coluna dos NIBs de cada utilizador

```
cat f1.txt | cut -d ' ' -f3 | cut -d '-' -f2,3,4
```

mostra apenas as colunas 3,4 e 5 dos NIBs (poderia também ser: -f2-4)

2.4 sort, uniq

O comando sort permite ordenar o output. O comando uniq permite remover duplicados se o conteúdo estiver ordenado. Muitas vezes, estes comandos operam em conjunto.

```
cat f1.txt | cut -d ' ' -f2 | sort | uniq
```

mostra a lista de equipas ordenada e sem repetição de nomes

2.5 grep

O comando `grep` permite procurar uma sequência nas linhas de um ficheiro. Na forma normal o comando mostra as linhas do ficheiro que contêm esse padrão. Podem-se usar expressões regulares (ver final do capítulo).

Opções mais utilizadas: `-i` ignora se é maiúscula ou minúscula; `-v` linhas onde não ocorre a sequência; `-n` juntamente com cada linha indica o nº de linha; `-c` conta as ocorrências, ou seja, o número de linhas onde ocorre a expressão.

```
cat f1.txt | grep sim
```

mostra as linhas que tenham a sequência "sim", neste caso mostra apenas as linhas "Manuel" e "Vanessa".

```
cat f1.txt | grep "^[A-U].... "
```

mostra todas as linhas que começam com letras maiúsculas de A a U e que após 4 letras tenham um espaço

```
cat f1.txt | grep -i "ort.* sim "
```

mostra todas as linhas com a sequência "ort", seguida de qualquer sequência de caracteres, seguida de " sim ".

```
cat f1.txt | grep -v -n "sporting"
```

mostra as linhas onde não ocorre "sporting", juntamente com o número de cada uma das linhas

```
grep -c "bash" /etc/passwd
```

mostra quantos utilizadores utilizam a bash

Exercício 4. Faça um script que:

- indique o número de ficheiros `.conf` existentes no directório `/etc`
- receba um argumento e indique cada um dos ficheiros `*.conf` onde o argumento aparece e quantas vezes

2.6 sed

O `sed` permite também utilizar expressões regulares e é especialmente importante quando se trata de scripts. Essencialmente, o `sed` lê uma entrada e produz uma saída transformada. A transformação a fazer é indicada num comando dado como argumento `sed`. Há duas transformações importantes:

- Substituir ocorrências de uma sequência no texto do ficheiro original.
- Filtrar parte das linhas do ficheiro original (exatamente o que faz o comando `grep`). Por exemplo, o seguinte comando mostra as linhas do ficheiro `/etc/passwd` que contêm a palavra `root`

```
sed -n '/root/p' /etc/passwd
```

A opção -n faz com que o sed reproduza apenas as linhas em que isso é pedido explicitamente; neste caso aparecem apenas as linhas seleccionadas pelo comando que tem a forma:

```
/expressão-regular/p
```

o efeito deste comando é escrever (o "p" é de print) as linhas que aderem à expressão-regular indicada; neste caso aderem à expressão regular as linhas contendo a palavra indicada.

permite fazer modificações aos dados que recebe, em particular permite fazer substituições.

```
cat f1.txt | sed 's/sim/SIM/g'
```

substitui a sequência "sim" por "SIM". O "g" no final indica para substituir todas as ocorrências na linha

```
cat f1.txt | sed 's/[sS]im/SIM/g'
```

substitui a sequência "sim" ou "Sim" por "SIM".

```
cat f1.txt | sed 's/ */ /g'
```

substitui múltiplas ocorrências de um espaço por um espaço

```
cat f1.txt | sed 's/^ */g; s/ *$/g'
```

Apaga os espaços no início e no fim da linha

```
cat f1.txt | sed 's/[0-9\ -]* / /'
```

Apaga o NIB. Substitui as sequências "espaço, zero ou mais caracteres 0..9 ou traços, espaço" por espaço

Exemplo 5. Utilização do sed para listar apenas as directorias

```
ls -l | sed -n '/^d/p'
```

Exemplo 6. Utilização do sed para listar apenas os ficheiros executáveis

```
ls -l | sed -n '/^...x..x..x/p'
```

Como foi acima dito, o sed pode ser usado para transformar o conteúdo de um ficheiro. Para esse efeito, o comando do sed tem a seguinte forma geral:

```
s/expressão-1/expressão-2/
```

e tem como efeito substituir a expressão-1 do texto original pela expressão-2. Nesta versão mais simples faria apenas uma substituição por linha, mas pode adicionar-se a opção (g)lobal para substituir todas as ocorrências de expressão-1 na mesma linha.

```
s/expressão-1/expressão-2/g
```

Exemplo 7. Utilização do sed para transformar o conteúdo de um ficheiro. Neste caso s/a/Aluno e tem como efeito substituir a letra a por Aluno. Note que o sed não altera o ficheiro, apenas o lê e mostra o seu conteúdo transformado.

```
sed 's/a/Aluno/' /etc/passwd
```

Exemplo 8. Eliminar a palavra "do" e substituir todos os "a" por "AA"

```
echo "Maria do Carmo Joaquina" | sed 's/do //g; s/a/A/g'
```


2.7 awk

O awk é um comando muito amplo que engloba, ele próprio, uma linguagem para programação orientada a padrões para processar ficheiros de texto. Vamos apenas ver algumas construções simples/típicas que se podem realizar.

Cada linha é considerada um conjunto de campos, que por omissão são separados por espaços ou tabs. O exemplo seguinte mostra a primeira e terceira colunas do resultado do comando `ls -l`, ou seja, as permissões e o dono de cada um dos ficheiros.

```
ls -l | awk ' {print $1, $3} '
```

O awk aceita um "comando" passado como argumento, indicado entre `{ }` e contém uma ou mais instruções, separadas por `;`. Neste caso a instrução é o `print` que "imprime" no ecrã. O que importa aqui é o facto de o awk, ao processar cada linha, dividir os campos (separados por espaços) em variáveis `$1` `$2` etc. Assim, ao escrever estas variáveis, estamos a escrever as colunas 1 e 3 do resultado do comando `ls -l`.

O exemplo anterior imprime todas as linhas. Uma variante que permite filtrar linhas através de uma expressão regular é um comando da forma:

```
/expressão-regular/ { comandos }
```

O exemplo seguinte mostra o mesmo conteúdo mas apenas para os directórios.

```
ls -l | awk '/^d/ { print $1 $3 }'
```

É possível associar um bloco de comandos a diferentes condições de selecção. O exemplo seguinte trata de forma diferente directórios e ficheiros comuns:

```
ls -l | awk '
  /^d/ { print "diretorio" $1 $3 }
  /^-/ { print "ficheiro normal" $2 }'
```

O comando seguinte mostra as colunas 1 e 2 do ficheiro `f1.txt`, que correspondem, neste caso, ao username e à equipa.

```
cat f1.txt | awk '{print $1, $2}'
```

Note que o comando anterior é equivalente a:

```
cat f1.txt | cut -d ' ' -f1,2.
```

A opção `-F` permite indicar separadores alternativos, aceitando expressões regulares. O comando seguinte considera que o separador é o traço e mostra as colunas 1 e 2 do NIB. Note que, neste exemplo, o primeiro campo vai até ao primeiro traço.

```
cat f1.txt | awk -F '-' '{print $3}'
```

O exemplo seguinte mostra o nome e descrição de cada utilizador (colunas 1 e 5 do ficheiro `/etc/passwd`).

```
cat /etc/passwd | awk -F ':' '{ print $1, $5}'
```

símbolo	significado especial nas expressões regulares
.	Um carater qualquer
[]	Um dos caracteres indicados na lista, podendo figurar intervalos. Exemplos: [abc] denota um caractere que seja a ou b ou c [a-z] denota uma letra minúscula [a-zA-Z] uma letra qualquer
^	Denota o início da linha. Exemplos: ^nome linhas que comecem com a palavra "nome"
[^...]	O simbolo ^ no início de uma lista denota todos os caracteres menos os indicados. Exemplos: [^abc] qualquer caractere que não seja o a ou b ou c [^a-b] qualquer caractere que não seja uma letra minúscula
\$	Denota o fim da linha Exemplo: :\$ seleciona linhas que terminem com ":"
\x	Escape de um caractere especial Exemplos: \\$ para representa o caractere \$ (deixa de representar o fim de linha) \. representa o símbolo . (deixa de representar qualquer caractere)
*	Repetição da sequência anterior 0 ou mais vezes. Exemplos: "[A-Z]*" detecta sequências de letras maiúsculas

Tabela 2.1: Resumo de símbolos utilizados em expressões regulares.

O exemplo seguinte considera que o separador pode ser tanto o espaço como o traço, mostrando: nome, último nº do NIB e a penúltima coluna. No exemplo, NF é uma variável que indica o nº de campos e, neste caso, corresponderá sempre a 8.

```
cat f1.txt | awk -F '[- ]' '{print NF, $1, $6, $(NF-1)}'
```

É ainda possível associar um bloco de comandos a uma situação especial, denotada pela marca BEGIN que ocorre antes da leitura de entrada. Isto é útil para fixar alguns parâmetros importantes antes de iniciar o processamento dos dados. O exemplo seguinte mostra, para cada linha, o número da linha e todo o seu conteúdo.

```
cat f1.txt | awk 'BEGIN{i=0} {printf("Linha %d : %s\n", ++i, $0)}'
```

2.8 Expressões regulares

Numa expressão-regular podem-se usar símbolos com significado especial que permitem bastante flexibilidade na localização de sequências de texto. A Tabela 2.1 resume essa simbologia e a Tabela 2.2 apresenta alguns exemplos de expressões regulares.

expressão	efeito
"Maria"	a palavra Maria
[Mm]aria	a palavra Maria ou a palavra maria
^A	localiza as linhas começadas pela letra A
a\$	linhas terminadas em a (contendo a sequência "a<fim-de-linha>")
[abc]	Contendo a letra a, b ou c
^[0-9]	Começadas por algarismo
^.\$	linha contendo apenas um caractere
^[a-zA-Z]\$	linha contendo apenas uma letra, maiúscula ou minúscula
Maria.*Reis	Palavra Maria, qualquer sequência de caracteres e Reis
"[A-Z][a-z]*"	palavras começadas com letras maiúsculas
"[0-9][0-9]*"	números inteiros com mais do que um dígito

Tabela 2.2: Algumas expressões regulares e o seu significado

2.9 Exercício

Considere um ficheiro com o seguinte conteúdo.

```

Numero Nome Equipa      NIB                      estatuto      saldo
-----
44444 Manuel sporting 0001-1002-12345234234-11 estudante 110.0
45323 Pedro benfica 0010-0302-00005234234-22 professor 120.3
43212 Maria porto 0011-0333-00008989898-33 estudante 3333.2
46444 Vanessa sporting 0100-0444-00004444444-33 estudante 44.5
51002 Marta braga 0100-0444-00004445555-23 estudante 1234.3
55002 rui sporting 0100-0444-00004445335-23 professor 2332.7
44000 David benfica 0100-0444-00002322335-23 estudante 123.8

```

2.9.1 Problemas a resolver com comandos simples

Para cada uma das seguintes alíneas, indique um comando que permita:

1. Obter o número de linhas do ficheiro
2. Mostrar as duas últimas linhas do ficheiro
3. Mostrar as linhas relativas a alunos com numero começado por 5 (cinquenta mil e ...)
4. Mostrar o nome de cada uma das pessoas
5. Mostrar o nome e o número
6. Mostrar o nome dos estudantes
7. Mostrar o nome dos estudantes do sporting. (Quem quiser pode fazer do benfica)
8. Mostrar o nome dos estudantes, ordenados por ordem alfabética
9. Mostrar a lista de equipas, ordenada e sem repetições

10. Contar o número de equipas diferentes que estão mencionadas
11. Criar um ficheiro a partir deste, no qual "rui" passa a estar com maiúscula
12. Criar um ficheiro com o nome e o número da conta (3º elemento do NIB)
13. Calcular o maior saldo
14. Ver o nome de quem tem maior saldo

2.9.2 Problemas a resolver com scripts

1. Faça um script que permita calcular o número da linha que no início da linha contém a sequência 55002.
2. Altere o script anterior de forma a que a sequência a procurar seja obtida como argumento, considerando também o caso da sequência poder não existir

2.10 Outros exercícios

Considere o seguinte excerto do ficheiro `/etc/passwd`:

```
a93615:x:1380:1006:Sandro Silva,,,:/home/a93615:/bin/bash
a38302:x:1381:1006:Sara Fernandes,,,:/home/a38302:/bin/bash
a93243:x:1384:1006:Soraia Pires,,,:/home/a93243:/bin/bash
a93162:x:1383:1006:Sergio Passos,,,:/home/a93162:/bin/bash
a64015:x:1382:1006:Antonio Junior,,,:/home/a64015:/bin/bash
a68804:x:1385:1006:Teresa Martins,,,:/home/a68804:/bin/bash
a61793:x:1386:1006:Tiago Moniz,,,:/home/a61793:/bin/bash
a62945:x:1389:1006:Tiago Monteiro,,,:/home/a62945:/bin/bash
a65746:x:1388:1006:Tiago Pedrinho,,,:/home/a65746:/bin/bash
a90037:x:1387:1006:Tomas Pedro,,,:/home/a90037:/bin/bash
a62713:x:1370:1006:Tomas Vendeirinho,,,:/home/a62713:/bin/bash
a93262:x:1371:1006:Tania Lam,,,:/home/a93262:/bin/bash
a88888:x:1014:1006:Aluno Joquim,,,:/home/a88888:/bin/bash
a88887:x:1013:1006:Aluno Rui,,,:/home/a88887:/bin/bash
a93134:x:1374:1006:Julio Pinto,,,:/home/a93134:/bin/bash
a65005:x:1373:1006:Tatiana Catan,,,:/home/a65005:/bin/bash
a55488:x:1372:1006:Lidia Goncalves,,,:/home/a55488:/bin/bash
a39988:x:1006:1006:Paulo Oliveira,,,:/home/a39988:/bin/bash
a61983:x:1375:1006:Pedro Afonso,,,:/home/a61983:/bin/bash
a93498:x:1376:1006:Nuno Antunes,,,:/home/a93498:/bin/bash
```

1. Exemplos de comandos que permitem extrair o nome dos alunos

```
cat /etc/passwd | cut -d ':' -f5 | cut -d ',' -f1
cat /etc/passwd | tr ',' ':' | cut -d ':' -f5
cat /etc/passwd | awk -F '[:,]' '{print $5}'
```

2. Comando que permite mostrar uma lista dos apelidos, sem repetição

```
cat /etc/passwd | awk -F '[:,]' '{print $5}' | cut -d ' ' -f2 | sort | uniq
```

3. Comando que permite mostrar os top 5 apelidos mais comuns

```
cat /etc/passwd | awk -F '[:,]' '{print $5}' | cut -d ' ' -f2 |
sort | uniq -c | sort -g -r | head -5
```

4. Comando que permite mostrar os nomes cujo primeiro nome termina numa vogal e o segundo começa com um "P"

```
cat /etc/passwd | awk -F '[:,]' '{print $5}' | grep "[aeiou] P"
```

5. Comando que permite mostrar os nomes cujo primeiro nome não termina numa vogal e o segundo começa com um "P"

```
cat /etc/passwd | awk -F '[:,]' '{print $5}' | grep "[^aeiou] P"
```


3

Programação em shell

3.1 Linha de comando

Esta secção foca alguns aspectos de utilização corrente da linha de comando.

3.1.1 Variáveis de ambiente

Parte das variáveis de ambiente suportam a configuração de alguns aspectos do funcionamento da própria shell. Por exemplo a variável `PS1` configura a prompt que a shell coloca no ecrã para pedir comandos. Por exemplo, o comando

```
PS1="diga> "
```

altera a prompt para uma coisa do género:

```
diga>
```

Uma das variáveis mais importantes da shell é a `PATH`. Esta variável contém uma lista de nomes de directórios, separados por ":" (dois pontos). Quando é dado um comando, a bash procura esse comando em cada um dos directórios dessa lista. Mais especificamente, há dois tipos de comandos: *internos* e *externos*. Comandos *internos* são os que a própria bash executa, dos quais `cd`, `pwd` ou `exit` são exemplos. Quando damos um comando *externo*, estamos na prática, a pedir à bash para executar um outro programa. A bash responde a este pedido, primeiro tentando localizar o programa, e depois fazendo uma de duas coisas:

1. se não conseguiu encontrá-lo dá um erro;
2. se conseguiu, executa o programa pedido. Mais tarde iremos estudar que, mais especificamente, a shell cria um processo filho (usando a primitiva `fork`) e executa nesse processo filho o programa pedido

Por exemplo, ao dar o comando `ls` estamos a pedir à bash para executar um programa chamado `ls` (sem dizer em que directório esse programa se encontra). A bash vai então procurar um ficheiro chamado `ls` em cada um dos directórios listados na variável `PATH`. Por exemplo, imagine que a variável `PATH` tem este conteúdo:

```
/home/programas:/etc/aquinaoesta:/bin:/home/maisprogramas
```

ao dar o comando `ls` a bash vai tentar executar:

```

/home/programas/ls      # não encontra
/etc/aquinaoesta/ls    # não encontra
/bin/ls                 # encontra e tenta executar este programa
/home/maisprogramas/ls # aqui nem chega a procurar porque já encontrou

```

É claro que a PATH só é relevante se o comando for dado com um nome simples, isto é, sem especificar o directório. Se ao dar o comando indicar o nome do ficheiro explícito a bash não precisa procurar. Por exemplo, se der o comando seguinte a bash tenta executar o programa que se encontre no local indicado.

```
/bin/ls
```

Acontece também que, por omissão, o directório corrente não está incluído na lista de directórios da PATH. Isto significa que pode ter um programa mesmo ali à mão no directório corrente e a bash não o encontra - pela simples razão de que não o procura lá. Por exemplo, se criar o programa `go.sh` no directorio actual e o tentar executar sem indicar o directorio vai obter algo do tipo:

```

go.sh          # a bash não localiza o programa no directório corrente
go.sh: not found

```

Há várias formas de resolver isto. Uma é indicar o nome do programa explicitamente.

```
./go.sh  # explicitar a localização do ficheiro (comando) a executar
```

Outra forma é resolver o problema de vez, adicionando o directório corrente à lista de directórios onde a bash procura comandos:

```
PATH=$PATH: . # juntar mais um elemento (o . ponto) à lista de directórios
```

3.1.2 subshell; exit

A shell é o programa interactivo que aceita e promove a execução dos nossos comandos. Há vários programas diferentes para este efeito (várias shell). A que vamos utilizar é a bash. Outras hipóteses são sh, csh, tcsh, zsh. Todos estes programas se encontram, normalmente, no directório `/bin`. Pode comprová-lo experimentando o comando:

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

Normalmente a shell é lançada, automaticamente, quando fazemos o login num terminal ou quando abrimos uma nova janela de comandos num ambiente gráfico; e termina quando damos o comando `exit`.

Exemplo: experimente o comando `ps`, para ver os processos em curso no terminal/janela de comandos. Um deles será a shell (muito provavelmente o único além do próprio comando `ps` em curso no momento). Podemos, entretanto, lançar novos processos shell executando o comando respectivo.

Exercício. experimente e interprete a seguinte sequência

```

/bin/bash      # lançar uma nova shell
ps             # deverá haver pelo menos dois processos shell em curso
exit
ps
exit          # oops...

```


3.2 Elementos de programação

A shell tem disponível um conjunto de mecanismos que normalmente fazem parte de uma linguagem de programação, como é o caso das variáveis, estruturas de controlo e funções.

3.2.1 scripts

Os comandos para a shell, que normalmente são executados interactivamente, podem também ser mandados executar através de um ficheiro.

Exemplo: escreva o seguinte conjunto de comandos num ficheiro `teste1`

```
#!/bin/bash
echo -n "Data: "
date "+%d de %B de %Y"
echo "LISTA DE UTILIZADORES"
who
```

Para executar estes comandos podemos agora mandar "executar o ficheiro". Normalmente será necessário, primeiro, atribuir a permissão `x` (execução) ao ficheiro. Trata-se de um ficheiro de texto e por isso, à partida, não é normal que tenha a permissão `x`. Podemos atribuí-la com

```
chmod +x teste1
```

ou, alternativamente,

```
chmod 755 teste1
```

Feito isso podemos então mandar executar o ficheiro com

```
./teste1
```

ou, se o directório corrente estiver na variável `$PATH`, simplesmente:

```
teste1
```

O efeito será idêntico ao que seria obtido pela execução dos comandos, um por um, na linha de comando.

3.2.2 Variáveis

Criação

Um script simples pode ser uma lista de comandos. Na realidade a shell inclui uma série de outros mecanismos que permitem configurar uma espécie de linguagem de programação. O primeiro desses mecanismos de programação é a possibilidade de criação de variáveis. Para definir uma nova variável utiliza-se um comando com a sintaxe (cuidado: não introduzir espaços):

```
nome_da_variavel=valor
```

Exemplo: Experimente a seguinte sequência de comandos:

```
x=Hello
echo $x
```

O primeiro cria uma variável da shell com nome *x* e conteúdo *Hello*; o segundo comando mostra a lista de variável, onde já deve aparecer a recém-criada variável *x*. Para ver o valor de uma variável pode-se usar o comando *echo*. Na sua forma mais simples este comando, *echo*, permite escrever um texto para o ecrã. Por exemplo:

```
echo Hello World
```

escreve no ecrã, o texto *Hello World*.

No texto a escrever pode-se incluir uma variável da shell. Para mencionar a variável escreve-se o nome antecedido de *\$*. Por exemplo:

```
echo $USER
echo Eu sou, portando, o utilizador $USER
echo Eu sou o utilizador $USER e estou a executar a shell $SHELL
```

Exemplo: experimente e interprete a seguinte sequência de comandos:

```
x=ABC
echo $x
echo x=$x
echo x dá x e \ $x dá $x
x=123
echo x foi modificado para $x
```

Manipulação de variáveis

As variáveis podem-se alterar (podem-se redefinir com outro valor). Por exemplo, experimente e interprete a seguinte sequência:

```
x=123
echo $x
x=456$x
echo $x
x=$x456
echo $x
```

O valor a atribuir a uma variável pode ser colocado entre *"* ou *'* (que, como habitualmente não ficarão a fazer parte do conteúdo). Este mecanismo é útil para incluir na variável caracteres que possam ter significado especial. Exemplo: o comando

```
x= ABC
```

não funciona para incluir espaços na variável e aliás dá erro. Pode fazer:

```
x=" ABC"
```

As variáveis da shell representam sequências de texto simples. Em geral o conteúdo de uma variável não é interpretado pela shell (ou seja, é usado literalmente).

Exemplo - experimente e interprete a seguinte sequência:

```
x=1
y=2
z=$x+$y
echo $z
```

Ocasionalmente, pode-se gerar ambiguidade na identificação do nome de uma variável. Nesse caso pode-se enquadrar no nome entre os caracteres { }. Exemplo, experimente e interprete a seguinte sequência:

```
x=abc
y=$xdef
echo $y
y=${x}def
echo $y
```

Uma variável pode ser destruída com o comando unset. A utilização de uma variável não definida não provoca qualquer erro - origina, simplesmente, uma cadeia de texto vazia. Exemplo, experimente e interprete a seguinte sequência:

```
x=ABC
set x                # deve aparecer na lista de variáveis
echo Valor de x = $x
unset x
set x                # já não deve aparecer na lista de variáveis
echo Valor de x = $x
```

Operações aritméticas

Muitas vezes é necessário fazer contas nos scripts em shell, tal como em qualquer linguagem de programação. O comando `$((...))` permite obter o resultado de expressões numéricas. Exemplo:

```
x=5
echo $(( $x+1 ))
```

Variáveis de ambiente

Identicamente, o script herda as variáveis de ambiente do processo original. Estas variáveis podem ser usadas e alteradas, apenas com efeitos durante a execução.

Considere a seguinte sequência:

```
export x=1
y=2
bsh_1c
echo "x=$x; y=$y; PATH=$PATH"
```

Elabore um script `bsh_1c` que mostre que:

- Pode usar e alterar as variáveis `x` e `PATH`. E a variável `y` ?
- Estas alterações não têm efeito no processo original;

As variáveis da shell podem-se confinar a um processo ou ser transmitidas aos processos descendentes. A estas últimas chamamos variáveis de ambiente. Para criar uma variável de ambiente utiliza-se o comando `export`, com a sintaxe:

```
export nome-da-variavel=valor
```

Por exemplo, o comando:

```
export v="Teste"
```

Cria uma variável de ambiente v.

Exemplo - experimente e interprete a seguinte sequência

```
x=ABC
export y=ABC
set      x e y figuram na lista de variáveis da shell em que foram criadas
/bin/bash abrir uma nova shell
set      apenas y figura na nova shell (processo filho) entretanto criada
exit     voltando à shell original...
set
```

As variáveis de ambiente são transmitidas aos processos descendentes por cópia. O processo pode modificar a variável recebida, mas essa modificação não se reflecte na variável existente no processo original.

Exemplo - experimente e interprete a seguinte sequência

```
export y=ABC
/bin/bash      abrir uma nova shell
echo $y
y=123
echo $y
exit          voltando à shell original
echo $y
```

3.2.3 Argumentos

Um dos mecanismos fundamentais para a construção de scripts são os argumentos passados na linha de comando. Estes argumentos são recebidos no script em variáveis especiais, designadas por:

```
$1 $2 $3 $4 ...
```

exemplo: construa o seguinte script (bsh_1d)

```
#!/bin/bash
echo "Primeiro argumento: $1"
echo "Segundo argumento: $2"
echo "Terceiro argumento: $3"
echo "Sétimo argumento: $7"
echo "Décimo segundo argumento: ${12}"
```

Experimente com:

```
bsh_1d a b c
bsh_1d 1 2 3 4 5 6 7 8 9 10 11 12 13
```

Exercício. Construa um script para verificar o significado das variáveis especiais \$0, \$# e \$*

A construção dos argumentos é feita após a substituição dos símbolos especiais pela shell; Verifique o valor dos argumentos nas seguintes situações:

```

bsh_1d ~
bsh_1d *
bsh_1d 0 meu nome de utilizador é $USER
bsh_1d "0 meu nome de utilizador é $USER"
bsh_1d $USER $NADA # $NADA não está definido
bsh_1d '$USER' $USER
bsh_1d '$USER $NADA $HOME'
bsh_1d "$USER $NADA $HOME"
bsh_1d 0 $NADA NAO EXISTE
bsh_1d 0 "$NADA" PODE EXISTIR

```

3.2.4 read

Outro dos mecanismos importantes para construir um script é o comando `read`, que permite ler interactivamente um valor para uma variável.

```

#!/bin/bash
echo "Diga qualquer coisa: "
read x
echo "Disse: $x"

```

3.2.5 Estruturas de controlo

if

Além de alinha comandos em sequência, a shell permite a utilização dos mecanismos de controlo habituais numa linguagem de programação, designadamente `if`'s e ciclos. Na sua forma básica o `if` permite escolher, para execução, um de dois grupos de comandos alternativos. A forma geral é:

```

if comando
then
    lista-comandos-1
else
    lista-comandos-2
fi

```

Exemplo:

```

if pwd
then
    echo "o pwd funciona"
else
    echo "difícilmente alguém verá isto"
fi

```

Exercício 9. faça um script `bsh_2d` que:

- recebe um argumento
- faz `chmod 700` do argumento
- se correu bem diz: "Comando executado com sucesso";
- caso contrário diz: "Comando falhou";

O comando `test` é especialmente vocacionado para a construção de condições de `if`. Trata-se de um comando que, na prática, serve para verificar uma condição produzindo um resultado adequado à utilização na estrutura do `if`. Exemplo:

```

echo -n "Username: "
read x
if test $x = $USER          # ou if [ $x = $USER ]
then
    echo "Acertou."
else
    echo "Falhou."
fi

```

Esta forma do comando test permite verificar se duas "strings" são iguais. Se forem, o resultado da execução do comando será "sucesso" (representado por 0). Caso contrário, será "insucesso" (representado pelo valor 1);

```

test "abc" = "xyz"          #ou [ "abc" = "xyz" ]
echo $?
[ "abc" = "xyz" ]           # ou test "abc" = "xyz"
echo $?
[ "abc" != "xyz" ]         # ou test "abc" != "xyz"
echo $?

```

O comando tem duas sintaxes alternativas: utilizar o comando test ou colocar os argumentos entre []; a partir daqui vamos sempre usar esta última; mas não se esqueça que [...] representa um comando test. As operações -z e -n permitem verificar, respectivamente, se uma string é vazia (tamanho 0) ou é não vazia (tamanho > 0). Exemplo:

```

#!/bin/bash
echo -n "Nome: "; read x
if [ -z "$x" ]; then
    echo "a string está vazia"
else
    if [ $x -eq $USER ] ; then
        echo "Acertou."
    else
        echo "Falhou."
    fi
fi

```

O exemplo anterior usa dois ifs encadeados, mas em alternativa pode ser usada a sintaxe if-elif-else-fi ilustrada no seguinte exemplo:

```

#!/bin/bash
echo -n "Nome: "; read x
if [ -z "$x" ]; then
    echo "(vazio)"
elif [ $x = $USER ]; then
    echo "Acertou"
else
    echo "Falhou."
fi

```

O mesmo comando permite comparar números inteiros; algumas das opções são exemplificadas no script seguinte:

```

#!/bin/bash
x=700
echo -n "diga um número: "; read n
if [ $n -lt $x ] ; then          # -lt : less than
    echo "Abaixo"

```

```

elif [ $n -gt $x ] ; then          # -gt : greater then
    echo "Acima"
else
    echo "Certo."
fi

```

(para uma lista completa dos operadores veja o manual do comando test)

Exercício 10. Faça outras versões do mesmo script usando as opções -eq e -gt; Faça um script que receba dois argumentos, ambos números inteiros, e escreva os mesmos dois números por ordem (e apenas um deles, se forem iguais);

Teste com ficheiros

Um outro grupo importante de opções do comando test serve para testar condições sobre ficheiros; Por exemplo, o seguinte script verifica se um dado ficheiro existe:

```

#!/bin/bash
if [ -f $1 ]; then
    echo "$1 existe"
else
    echo "$1 não existe"
fi

```

O seguinte script exemplifica algumas das opções mais do test para ficheiros;

```

if [ -z $1 ] ; then          # falta se não houver argumento
    echo "usage: $0 ficheiro"
    exit                    # terminar o script
fi

if [ -f $1 ] ; then; echo "$1 é um ficheiro normal"; fi
if [ -d $1 ] ; then; echo "$1 é um directório"; fi
if [ -r $1 ] ; then; echo "$1 tem permissão r"; fi
if [ -w $1 ] ; then; echo "$1 tem permissão w"; fi
if [ -x $1 ] ; then; echo "$1 tem permissão x"; fi

```

Case

A instrução case permite escolher um de vários blocos de comandos alternativos, o sua forma geral é a seguinte:

```

case $variavel in
caso1)
    lista-comandos1;;
caso2)
    lista-comandos2;;
...
esac

```

O seguinte exemplo ilustra a utilização do case:

```

#!/bin/bash
case $1 in
    benfica) echo "Lisboa";;

```

```
sporting) echo "Lisboa";;
porto) echo "Porto";;
boavista) echo "Porto";;
esac
```

No exemplo anterior as opções são valores unívocos. De uma forma geral, as opções podem ser um padrão; nesse caso, é seleccionada a primeira opção que adira à string de controlo (indicada na linha `case...in`). A construção do padrão admite mecanismos que são mais ou menos familiares:

| - alternativas (or)

* - qualquer cadeia de caracteres (0 ou mais)

? - um carácter qualquer

```
#!/bin/bash
case $1 in
  benfica|sporting) echo "Lisboa";;
  porto|boavista) echo "Porto";;
  *) echo "Outras cidades";;
esac
```

O padrão `*` captura todas as sequências, conseguindo-se assim uma forma de obter um "default";

Exercício 11. O que acontece se trocar a ordem das opções pondo o `*` em primeiro lugar ?

Um script que aceita as opções `"-d"`, `"-u"` e dá um erro se não for usada nenhuma opção.

```
#!/bin/bash
if [ $# -ne 1 ]; then
  echo "Número incorreto de argumentos"
  exit 1
fi
case $1 in
  '-d') echo -n "Data atual:"
        date
        ;;
  '-c') echo "Candendário deste mês"
        cal
        ;;
  *) echo "não conheço essa opção";;
esac
```

Um outro exemplo... o que faz ?

```
#!/bin/bash
case $1 in
  *.txt) echo "ficheiro de texto";;
  *.C|*.c|*.h) echo "c/c++";;
  *) echo "outros ficheiros";;
esac
```

For

O comando `for` permite repetir um conjunto de comandos, para cada um dos elementos de uma lista. A sua sintaxe é


```

for variavel in lista
do
    lista-de-comandos
done

```

Exemplo:

```

for i in "Bom dia" boa tarde; do
    echo "i= $i"
done

```

Muitas vezes o `for` é utilizado para percorrer a lista de ficheiros de um directório. Para formação dessa lista aplicam-se os mecanismos habituais de expansão da shell:

```

#!/bin/bash
for i in *; do
    echo "entrada: $i"
done

```

Exemplo 12. Script que lista os ficheiros do directório atual (excluindo os directórios)

```

#!/bin/bash
for i in *; do
    if [ ! -d $i ] ; then
        ls -l $i
    fi
done

```

Exercício 13. altere para listar apenas os ficheiros de extensão `.c`, `.C` ou `.h`

Como vimos anteriormente, as variáveis especiais `$1`, `$2`, ... representam os argumentos do comando. Estas variáveis são adequadas para aceder aos argumentos um a um. Para aceder aos argumentos num ciclo são úteis as seguintes outras variáveis especiais:

`$*` - todos os argumentos

`$#` - o número de argumentos

```

#!/bin/bash
echo "foram dados $# argumentos, que são: "
for a in $* ; do
    echo "-> $a"
done

```

while

O ciclo `while` é um ciclo de condição: repete a execução de um bloco de comandos enquanto se verificar uma dada condição de controlo (ou até ela deixar de se verificar);

```

while [ condição ]
do
    lista-de-comandos
done

```

Exemplo:

```
#!/bin/bash
x=0
while [ $x != 700 ]; do
    echo -n "Adivinhe o numero: "; read x
done
```

Altere o exemplo anterior para fazer um pequeno jogo de aproximação: após cada tentativa o script deve ajudar dizendo se o valor certo é para "Cima" ou para "Baixo".

Na verdade a instrução `while` permite executar um conjunto de instruções repetidamente enquanto a execução de um comando não falhar. Isso permite-nos usá-la para fazer coisas muito úteis, como por exemplo, ler um ficheiro linha a linha. Por exemplo, o seguinte bloco de código lê e escreve as linhas introduzidas pelo utilizador (faça CTRL+D para parar de introduzir linhas).

```
while read linha; do
    echo "Linha: $linha"
done
```

O mesmo código pode ser facilmente adaptado para leitura de um ficheiro linha a linha. Exemplos:

```
while read linha; do
    echo "Linha: $linha"
done < ficheiro
```

ou

```
cat ficheiro | while read linha; do
    echo "Linha: $linha"
done
```

Uso do resultado da execução de comandos

Os delimitadores `' '` permitem usar o resultado obtido pela execução de um comando (escrito entre os dois delimitadores). Exemplo:

```
x='date'
echo $x
```

Este mecanismo é um precioso auxiliar para scripts. Por exemplo, pode ser utilizado para o backup de um ficheiro com a data actual. Alternativamente, poder-se-ia utilizar a forma `$(...)`, que também faz o mesmo efeito. O script anterior é equivalente a:

```
x=$(date)
echo $x
```

Uma forma de uso mais imediata poderia ser:

```
echo "A hora e data atual são: $(date)"
```

3.3 Comandos de apoio a scripts

Alguns comandos de apoio a scripts

3.3.1 expr

O comando `expr` (`/bin/expr`) contempla também algumas operações com strings; em particular é a forma mais fácil de fazer uma operação importante que é localizar uma string dentro de outra. Exemplo: o script seguinte lê uma string e indica em que posição se encontra a letra “a”:

```
#!/bin/bash
echo -n "String: "
read s
echo 'expr index $s a'
```

Na realidade o comando `expr` serve para fazer o cálculo de expressões simples, quer em texto quer em números inteiros, sendo portanto, também, um alternativa ou complemento ao cálculo numérico com `$(())`;

Ex: o seguinte script lê uma string, representando o nome completo de uma pessoa, e mostra apenas o primeiro nome próprio:

```
#!/bin/bash
echo -n "String: "
read s
n='expr index "$s" " "'
if [ $n -gt 0 ] ; then
    m='expr $n - 1'
    echo "Nome:" 'expr substr "$s" 1 $m'
fi
```

exercício (faça todos os cálculos com `expr`);

- altere para mostrar o primeiro e último nome;
- altere para mostrar o último apelido e depois todos os outros nomes;

3.3.2 basename

Há um conjunto de comandos que, embora possam ser usados interactivamente, são especialmente importantes no quadro da programação com a shell; é o caso, por exemplo, do comando `basename`; o comando extrai o nome base de um nome completo de ficheiro, retirando:

- o caminho até ao directório;
- a parte final (se for igual ao segundo argumento)

```
basename /etc/passwd    # passwd
basename teste.c .c     # teste
basename /etc/rc.1 .1   # rc
basename teste.c .x     # teste.c
```

3.4 Outros mecanismos. Complementos.

3.4.1 Expansão

Questão muito importante: quando fazemos um comando como o `ls -l *.txt` ou `zip docs *.doc` quem é que transforma o `*.txt` numa lista de nomes de ficheiros, a shell ou os comandos. Resposta: a shell. Por exemplo, ao fazermos

```
ls -l *.txt
```

a shell vai invocar o comando `ls` passando-lhe como argumentos todos os nomes de ficheiro que encontrar que adiram ao padrão `*.txt` (ou seja o nome de todos os ficheiros com extensão `.txt` que encontrar, neste caos no directório corrente).

Podemos, em caso de dúvida, fazer um programa nosso para o demonstrar. O seguinte programa (`eco.sh`) escreve no ecrã os argumentos que receber da linha de comando:

```
#!/bin/bash
num=1
for i in $*; do
    echo "$num : $i"
    num=$((num+1))
done
```

Experimente, por exemplo:

```
./eco.sh a b c
./eco.sh *.txt
```

MAIS AVANÇADO

Muitas vezes é preciso impedir a shell de seguir o procedimento normal de expansão de caracteres. Por exemplo, para escrever um `$` no ecrã é preciso indicar à shell para não interpretar o `$` como iniciador do nome de uma variável. Diz-se então que estamos a fazer o escape (do significado habitual) do caracter.

Uma das maneiras de escapar uma sequência é inseri-la entre `' '`. Por exemplo:

```
x=bifa
echo $x          # o $X é expandido
echo '$x'        # os ' ' impedem a expansão
```

O mecanismo de expansão deriva da presença de caracteres com significado especial para a shell. No caso anterior o significado especial é dado pelo `$`. As `' '` retiram o significado especial ao `$` e por isso `$x` fica a ser só, apenas e literalmente `$x`. Em rigor, as `' '` são necessárias apenas para lidar com o `$`. Ou seja, poderíamos obter o efeito desejado apenas com `echo '$x'`. Mas `'$x'` também funciona e é muito mais claro.

São também caracteres especiais da shell os seguintes:

```
* ? [ ] ' " \ $ ; & ( ) | ^ < > { }
```

que normalmente, para serem assumidos de forma literal, devem também ser escapados).

Há 3 elementos sintáticos que impedem a expansão: as `" "`, as `' '` e a `\`.

As aspas são as mais fracas: escapam apenas alguns caracteres. Não escapam, por exemplo, o `$` e, por isso, as variáveis colocadas entre aspas continuam a ser expandidas. Por exemplo:

```
echo "*** $HOME ***"
```

torna os `*` literais (escreve os asteriscos), mas mantém a expansão da variável `$HOME`

As `' '` escapam todos os caracteres e o `\` escapa o caracter seguinte (apenas 1) seja ele qual for. Por exemplo:

```
echo ' $HOME '
echo \$HOME
```

(para escrever `\` usa-se `\\`).

3.4.2 if / test / review

É importante perceber a subtilidade da "condição" do if. Numa linguagem de programação clássica a decisão do if é baseada numa condição. Na shell a decisão é baseada no resultado da execução de um comando.

Exemplo: considere o seguinte comando

```
chmod 700 x
```

Se o ficheiro x existir (e houver permissão) o comando é executado com sucesso; caso contrário, a operação não é executada e aparece uma mensagem de erro. Verifique o efeito no script seguinte (bsh_2b)

```
#!/bin/bash
echo "Ficheiro: "
read x
if chmod 700 $x
then
    echo "Comando executado!"
else
    echo "Como deve ter percebido, não correu bem."
fi
```

A variável especial \$? representa o resultado do último comando executado. Veja o valor de \$? após o comando

```
chmod 700 x
echo $?
```

(veja para o caso de sucesso e para o caso de insucesso na execução do comando).

Exercício 14. qual o problema com este script ?

```
#!/bin/bash
chmod 700 x
if $?
then
    echo "Ok"
else
    echo "Erro"
fi
```

3.4.3 exit; encadeamento de comandos

Interactivamente o comando exit serve para terminar uma shell; ex:

```
/bin/bash    # inicia nova shell
exit         # termina (volta à shell inicial)
```

num script tem o efeito correspondente, ie, termina a execução do próprio script; ex:

```
#!/bin/bash
date
exit         # o script termina aqui
echo "Bye."  # este comando não chega a ser executado
```

O `exit` pode ser utilizado para terminar um script, prematuramente, em condições anormais. Exemplo: o seguinte script (`mostra.sh`) mostra o conteúdo de um ficheiro, que é dado num argumento para o script; o primeiro passo é justamente verificar o argumento:

```
#!/bin/bash
if [ ! $# -eq 1 ] ; then
    echo "usage: $0 <file>"
    exit
fi
cat $1
```

Além da função de instrução de controlo, o `exit` tem outro efeito importante que é estabelecer o resultado – o exit status do comando; Exemplo: considere o seguinte script, que passaremos a referir com o nome `trivial`

```
#!/bin/bash
exit $1
```

verifique o efeito da seguinte sequência:

```
trivial 2
echo $? # o que dá ?
if [ trivial 0 ] ; then ; echo "funciona."; fi
```

O símbolo `?` representa o exit status do último comando a ser executado.

Considere a seguinte versão do exemplo `mostra.sh`:

```
#!/bin/bash
if [ ! $# -eq 1 ] ; then
    echo "usage: $0 <file>"
    exit 1
fi
cat $1
exit 0
```

- se não for dado um argumento o script termina com saída em de erro (diferente de 0);
- ao contrário, se tudo correr bem, termina com 0; note que o `exit 0` final não tem interesse de execução (não adianta fazer `exit` quando o script vai terminar "sozinho"); o interesse é apenas estabelecer o resultado de saída em situação normal - ou seja, 0;
- em vez de `exit 0` não seria melhor `exit $? ?`

Os elementos `&&` e `||` permitem fazer o encadeamento de comandos numa lógica parecida com a dos operadores homólogos da linguagem C.

A sequência seguinte executa os comandos por ordem enquanto derem "sucesso" (ou seja, termina a sequência se um deles falhar).

```
comando1 && comando2 && ...
```

A sequência seguinte executa os comandos por ordem até um deles ter sucesso (ou seja, executa os comandos por ordem enquanto falharem).

```
comando1 || comando2 || ....
```

Exemplo:

```
trivial 0 && echo "Ok."
trivial 1 && echo "NOP"
trivial 0 ||  echo "NOP"
trivial 1 ||  echo "Ok."
```

A seguinte versão do script comp verifica se é dado um argumento e se o ficheiro correspondente existe; caso uma das condições não se verifique o script termina com erro:

```
#!/bin/bash
if [ ! $# -eq 1 ] || [ ! -f $1 ] ; then
    echo "$0: invalid arguments"
    exit 1
fi
cc $1 -o 'basename $1 .c'
exit $?
```

Uma alternativa poderia ser:

```
#!/bin/bash
if [ $# -eq 1 ] && [ -f $1 ] ; then
    echo "Existe"
else
    echo "$0: invalid arguments"
    exit 1
fi
cc $1 -o 'basename $1 .c'
exit $?
```

Verificadas as duas condições é executado o comando echo "Existe" e o script segue depois do if; de contrário, termina com exit.

O comando seguinte muda as permissões ao programa e, em caso de sucesso, executa-o

```
chmod +x ./go.sh && ./go.sh
```

3.4.4 Funções

As funções são um elemento estruturante parecido com as funções das linguagens de programação clássicas; no caso dos scripts, enquadram um conjunto de comandos que são executados quando a função é invocada através do seu nome; ex:

```
#!/bin/bash
exemplo() {
    echo $FUNCNAME says hello
}

echo "chamar a função..."
exemplo
echo "repete..."
exemplo
```

As funções aceitam argumentos numa sintaxe muito semelhante à dos próprios argumentos dos scripts; exemplo:

```
#!/bin/bash
say () {
    echo "I say, " $1
}
```

```
say hello
say hello hello
say "helo hello hello"
```

O comando `return` termina a função, em determinado ponto, permitindo também formar um resultado de retorno; Exemplo:

```
#!/bin/bash
say () {
    if [ $# -eq 0 ] ; then
        echo "nothing to say"
        return 1
    fi
    echo "I say, " $*
    return 0
}

say hello
say hello hello
say "helo hello hello"
say
echo just say 'say'
```

As variáveis do script (as que são herdadas ou as que são criadas no próprio script) estão disponíveis na função; podem ser aí alteradas tal como podem ser criadas novas variáveis; as variáveis trabalhadas deste modo são todas "globais" (no sentido que o termo tem na programação clássica): existem podem ser criadas, alteradas e destruídas em todo o lado. Exemplo:

```
#!/bin/bash
f () {
    echo "$FUNCNAME : Y= $Y"
    X=77
    Y=88
    echo "$FUNCNAME : X= $X"
    echo "$FUNCNAME : Y= $Y"
}

Y=66
echo "Y= $Y"
f
echo "X= $X"
echo "Y= $Y"
```

Exemplo: no seguinte script é feita uma função `readline` que lê uma string:

```
#!/bin/bash
readline () {
    echo "readline..."
    msg=""
    if [ $# -gt 0 ] ; then
        msg="$*: "
    fi

    str=""
    while [ -z $str ] ; do
        echo -n $msg
        read str
    done
}
```



```

        done
        STR=str
    }

    readline "Teste"
    echo "Lido: " $STR

```

3.4.5 Arrays

Nos scripts podem-se usar variáveis indexadas; ex:

```

#!/bin/bash
a[0]="Hello"
a[3]="World"
echo "${a[0]} ${a[3]}"

```

note a utilização da sintaxe `${}` para isolar o nome das variáveis;

É raro haver interesse em usar variáveis indexadas, isoladamente, em vez de variáveis comuns. Normalmente o que se pretende é usar um conjunto de posições contíguas em processamentos iterativos (ou seja, o correspondente aos arrays nas linguagens de programação clássicas).

Exemplo 15. Gerar 6 números aleatórios:

```

#!/bin/bash
i=0
while [ $i -le 5 ] ; do
    num[i]=$RANDOM
    echo "Número : $i ${num[i]}"
    i=$(( $i + 1 ))
done

```

A variável `$RANDOM` fornece um número aleatório ("diferente") de cada vez que é usada; o número gerado situa-se entre 0 e 2^{15} (32767);

Um while deste género pode ser escrito de maneira mais familiar com a seguinte sintaxe alternativa, mais simpática para ciclos iterativos.

Exemplo 16. Script que gera 6 números aleatórios entre 1 e 20

```

#!/bin/bash
for (( i=0; i < 5; i++ )) ; do
    num[i]=$(( 1 + 20 * $RANDOM / 2**15 ))
done

```

Exemplo 17. Script gera 6 números aleatórios, entre 1 e 20, apresentando-os por ordem;

```

#!/bin/bash
for (( i=0; i < 5; i++ )) ; do
    num[i]=$(( 1 + 20 * $RANDOM / 2**15 ))
done

for (( i=0; i < 5; i++ )) ; do
    for (( j=0; j < 5; j++ )) ; do
        if [ ${num[i]} -lt ${num[j]} ] ; then
            x=${num[i]}

```

```

        num[i]=${num[j]}
        num[j]=$x
    fi
done
done

for (( i=0; i < 5; i++ )) ; do
    echo "Numero: $i ${num[i]}"
done

```

Exercício 18. Faça um script que gere uma aposta do totoloto, ie, 6 números diferentes, entre 1 e 49

3.4.6 Strings

length – permite obter o comprimento, ie o número de caracteres, de uma string;

Exemplo 19. Script que mostra o tamanho de uma string lida

```

#!/bin/bash
echo -n "String: "
read s
echo ${#s}

```

substring – permite extrair parte de uma string

Exemplo 20. Script que lê uma string e mostra parte dos caracteres lidos

```

#!/bin/bash
echo -n "String: "
read s
echo ${s:5}
echo ${s:5:3}
n=${#s}
if [ $(( n % 2 )) -eq 1 ] ; then
    m=$(( n / 2 ))
    echo ${s:$m:1}
fi

```

Exercício 21. Altere o exemplo anterior para mostrar também: a) o último caractere; b) a primeira metade da string.

substituição – permite substituir uma parte da string

Exemplo 22. Script que substitui a letra “a” pela letra “x” na string lida.

```

#!/bin/bash
echo -n "String: "
read s
s1=${s/a/x} ; echo $s1
s1=${s//a/y} ; echo $s1

```

Experimente uma entrada com várias letras a para verificar a diferença entre / e //;

3.4.7 Separação de palavras

Evidentemente que exercícios como o anterior são mais fáceis de realizar usando os mecanismos da shell que naturalmente separam palavras; Exemplo:

```
#!/bin/bash
echo -n "String: "
read s
p=""
for i in $s ; do
    if [ -z $p ] ; then
        p=$i; echo "Primeiro: $p"
    fi
done
echo "Ultimo: $i"
```

Estes métodos ganham ainda maior utilidade com a possibilidade de escolher o separador de palavras, que pode ser indicado à shell na variável IFS. Normalmente o separador é o espaço, mas pode-se alterar através desta variável. Exemplo: o seguinte script mostra a lista de directório da PATH, um por linha:

```
#!/bin/bash
IFS=:
for d in $PATH ; do
    echo $d
done
```

3.4.8 Exercícios

Acertar na soma

Exercício 23. Faça um script em bash que atribua números aleatórios às variáveis x e y, faça a sua soma e peça ao utilizador para adivinhar o resultado. Quando o utilizador acertar, o script deverá indicar o tempo que foi usado para fazer a conta em segundos.

```
#!/bin/bash
x=$(( $RANDOM / 100 ))
y=$(( $RANDOM / 100 ))
s=$((x+y))
di=$(date +%s)
echo -n "Qual a soma de $x com $y ? "
read g
while [ "$g" -ne $s ]; do
    echo -n "Tente de novo: "
    read g
done
df=$(date +%s)
t=$((df-di))
echo "Acertou em $t segundos"
```

Listar os ficheiros executáveis da directoria actual**Compilar todos os ficheiros .c numa dada directoria****Jogo da forca**

Considere o ficheiro `words.txt`, com o seguinte conteúdo

`words.txt`

```
barbatana
camelo
carro
caramelo
```

`./forca.sh`

```
#!/bin/bash

#readc: lê um character
readc() {
    echo -n "letra ( dispõe de $ntry tentativas ) : "
    read c
}

#marca : marca o character lido
marcac() {
    newd=""
    ok=0
    tryok=1
    for (( i=0; i < n ; i++)) ; do
        sx=${s:i:1}
        dx=${d:i:1}
        if [ $sx = $c ] || [ $dx != "-" ] ; then
            newd="$newd$sx"
        else
            newd="$newd-"
            ok=1
        fi
        if [ $sx = $c ] && [ $dx = "-" ] ; then
            tryok=0
        fi
    done
    d=$newd
    ntry=$(( $ntry - $tryok ))
}

#getword : obtém uma palavra do ficheiro
getword() {
    wn='cat words | wc -l'
    wx=$(( 1 + $wn * $RANDOM / 2**15 ))
    s='cat words | head -$wx | tail -1'
}

#main
#s=barbatana
getword

n=${#s}
```

```

d=""
for (( i=0; i < n ; i++)) ; do
    d="$d-"
done
echo $d

ok=1
ntry=7
while [ ! $ok -eq 0 ] && [ $ntry -gt 0 ] ; do
    readc
    marcac
    echo $d
done
if [ $ok -eq 0 ] ; then
    echo "GANHOU."
else
    echo "PERDEU."
fi

```

3.5 Exemplos

Testar ficheiros

```

#!/bin/bash
echo -n "Diga um nome: "
read n

if [ -x $n ]; then
    echo "$n é um ficheiro executável"
else
    echo "$n não é"
fi

```

Testar se um número está num intervalo

```

#!/bin/bash
x=15
if [ $x -gt 10 ] && [ $x -gt 20 ] ; then
    echo "Funcionou"
else
    echo "Falhou completamentwe"
fi

```

Classificar a idade em intervalos

```

#!/bin/bash
echo -n "diga o nome: "
read nome
echo -n "diga a idade: "
read idade
echo "Oh $nome, a sua idade é $idade"

if [ $idade -lt 25 ]; then
    echo "Muito novo"

```

```

elif [ $idade -lt 50 ]; then
    echo "normal"
else
    echo "A caminho da terceira idade"
fi

```

Testar parâmetros

```

#!/bin/bash
echo "nome do programa: $0"
echo "Argumento 1: $1"
echo "Argumento 2: $2"
echo "foram dados $# argumentos"

echo "Sobre $1, sei que está em:"
case $1 in
    boavista) echo "Porto";;
    sporting|benfica) echo "Lisboa";;
    iscte*) echo "Fica em Lisboa";;
    *) echo "desconheço a cidade";;
esac

```

Ler um ficheiro e escrevê-lo, uma linha de cada vez

```

#!/bin/bash
fich=$1
cat $fich | while read linha; do
    echo "Linha: $linha"
done

```

Utilização do *for* para percorrer elementos de uma lista

```

#!/bin/bash
contagem=1
for i in Maria 1 2 $1 "Pedro e Paulo" *.sh *.txt; do
    echo "Elemento $contagem: $i"
    if [ -f "$i" ]; then
        echo "Existe como ficheiro"
    fi
    contagem=$(( $contagem + 1 ))
done

```

Adivinhar um número

```

#!/bin/bash
# echo -n "Diga um numero: "
# read num
num=$RANDOM

tentativas=0
adivinha=0
while [ $adivinha -ne $num ]; do
    echo -n "Tente adivinhar : "
    read adivinha

    if [ $adivinha -gt $num ]; then

```

```

    echo "Muito grande"
elif [ $adivinha -lt $num ]; then
    echo "Muito pequeno"
else
    echo "Conseguiu"
fi
tentativas=$(( $tentativas + 1 ))
done
echo "Conseguiu com $tentativas tentativas"

```

3.6 Elementos mais avançados

3.6.1 diferença entre [[e [

O duplo parênteses reto "[[]]" é uma extensão da *bash* ao comando "[]". Implementa algumas melhorias que podem tornar-se uma mais-valia em scripts que apenas usam a *bash*. No entanto, deve ter-se em atenção que isto é uma extensão da *bash* e que se queremos manter os scripts compatíveis com a shell original deveremos continuar a usar "[]". Alguns exemplos:

- Na versão original, para evitar problemas com strings vazias, é necessário escrever

```
if [ -f "$ficheiro" ]
```

A versão [[não necessita disso. Basta escrever

```
if [[ -f $ficheiro ]]
```

- A nova versão permite também utilizar os operadores &&, || para testes de booleanos e permite comparar strings usando os operadores < e >. A versão original não pode fazer isso porque corresponde a um comando e os símbolos &&, ||, < e > não são passados para os comandos como argumentos.
- O operador =~ permite processar expressões regulares. Por exemplo, na versão original poder-se-á escrever

```
if [ "$resposta" = s -o "$resposta" = sim ]
```

Na versão [[basta escrever

```
if [[ $resposta =~ ^s(im)?$ ]]
```

ou simplesmente

```
if [[ $resposta =~ ^s* ]]
```


4

Tarefas de administração (Linux)

Os conteúdos deste capítulo foram adaptados a partir de materiais produzidos pelo colega João Pedro Oliveira (joao.p.oliveira@iscte-iul.pt) durante o ano 2010.

4.1 Gestão de contas

4.1.1 Conta de utilizador

Quando um sistema operativo é usado por muitos utilizadores, torna-se necessário diferenciar os utilizadores, por exemplo, para manter os seus ficheiros privados. Esta necessidade mantém-se mesmo que apenas um utilizador possa estar a usar o computador em simultâneo. Cada utilizador tem um portanto um username que é único e serve para entrar no sistema operativo (login).

Uma conta de utilizador é mais do que um simples username. Representa todos os ficheiros, recursos e informação relativa a um utilizador.

4.1.2 Criar uma conta

O kernel do sistema operativo trata cada utilizador como sendo um simplesmente um número. Cada utilizador tem assim associado um número único denominado user id (ou uid). O nome do utilizador, bem como informação adicional, é guardado num ficheiro separado. Para criar um utilizador, a maior parte das distribuições de Linux têm programas para isso. Nomeadamente adduser e useradd.

/etc/passwd

O ficheiro /etc/passwd é o ficheiro onde a informação principal associada a cada utilizador é guardada. Este ficheiro tem uma linha por utilizador, com campos separados por (:):

- username;
- campo onde a password foi guardada em tempos;
- user id (uid);
- group id (gid);
- nome textual do utilizador;
- directoria home;
- a shell de login.

Exemplo da informação de um utilizador:

```
labac:x:1000:1000:Arquitetura computadores:/home/labac:/bin/bash
labso:x:1001:1001:Utilizador labso:/home/labso:/bin/bash
```

A password de cada utilizador é encriptada e guardada num ficheiro à parte. Usualmente este ficheiro, `/etc/shadow`, apenas é acessível pelo administrador (root).

Ambiente inicial: `/etc/skel`

Quando um utilizador é criado, a directoria home é inicializada com os ficheiros existentes na directoria `/etc/skel`. Este é o sítio ideal onde o administrador deve colocar os ficheiros que deverão ser comuns a todos os novos utilizadores. O seu conteúdo deve ser mantido o mais pequeno possível, pois tornar-se-á difícil posteriormente actualizar o conteúdo dos diferentes utilizadores já criados.

Criar uma conta manualmente

Criar um utilizador manualmente envolve os seguintes passos:

- editar o ficheiro `/etc/passwd` com o comando `vipw` e adicionar uma nova linha para o utilizador. Este comando tem como vantagem bloquear o acesso ao ficheiro, de forma a que outros programas tentem actualizá-lo ao mesmo tempo. O campo da password deve ser inicializado com `*` (asterisco) de forma a ser impossível ao utilizador efectuar o login;
- de forma igual, editar o ficheiro `/etc/group` com o comando `vigr`, se for necessário criar um novo grupo;
- criar a directoria home do utilizador com o comando `mkdir`;
- copiar os ficheiros da directoria `/etc/skel` para a nova directoria home;
- mudar o dono do ficheiro e as permissões usando os comandos `chown` e `chmod`; a opção `-R` é útil para aplicar recursivamente a todos os ficheiros da directoria actual e respectivas subdirectorias:

```
cd /home/novoutilizador
chown -R username.group .
chmod -R go=u, go-w .
chmod -R go= .
```

- mudar a password do utilizador com o comando `passwd`.

Observação. estas operações devem ser realizadas como administrador, i.e., como root. Para tal pode usar-se o comando `su` para passar o utilizador actual para superuser. Nas distribuições mais recentes, o administrador costuma não ter login, pelo que alternativamente é apenas possível invocar comandos como sendo administrador, precedendo o comando a executar de `sudo`.

Por exemplo, para mudar as permissões dos ficheiros usar-se-ia o seguinte comando:

```
sudo chown -R username.group .
```

Observação. Os comandos `adduser` e `useradd` evitam os passos anteriores.

4.1.3 Remover uma conta

Remover um utilizador significa apagar toda a informação relativa ao mesmo. No entanto, os ficheiros de um utilizador podem encontrar-se espalhados pelo sistema de ficheiros. Isto é, os ficheiros de um utilizador não ficam limitados à directoria home. Por exemplo, o email é por vezes guardado na directoria `/var/mail` ou `/var/spool/mail`. É por isso necessário procurar os ficheiros do utilizador pelo sistema de ficheiros.

Existem dois comandos para remover utilizadores que fazem isso automaticamente. São eles o `deluser` e o `userdel`.

4.1.4 Configuração da conta (`.profile` e `.bashrc`)

Na directoria home de cada utilizador existem um conjunto de ficheiros com definições e/ou configurações especiais para cada utilizador. Usualmente o nome destes ficheiros começa por um ponto (`.`), o que significa que são ficheiros “escondidos” não visíveis através dos comandos `ls` ou `ls -l`. Para visualizar os ficheiros escondidos é necessário usar a opção `-a` do `ls`, por exemplo, executando o comando `ls -la`.

O ficheiro `.profile` é um script que é executado quando é feito o login do utilizador. Este script é executado depois do `/etc/profile`, que é suposto ter definições gerais para todos os utilizadores. Este ficheiro é usado por exemplo para definir variáveis de ambiente, definidas como já foi visto pelo comando `export`.

O ficheiro `.bashrc` é o ficheiro de configuração da `bash`. É executado cada vez que se inicia a `bash`.

4.2 Agendamento de comandos (cron daemon)

O processo `cron` é um processo especial que permite executar/agendar tarefas num determinado instante de tempo, que pode ser definido como um intervalo regular ou como uma determinada hora de um determinado dia. O `cron` baseia-se numa configuração que é guardada num ficheiro `crontab` (cron table). Este ficheiro indica os comandos que devem ser executados periodicamente num determinado horário.

Ao nível do sistema, os ficheiros de configuração dos diferentes utilizadores encontram-se dentro da directoria `/var/spool/cron/crontabs`. O seu conteúdo não deve ser alterado directamente, mas sim mediante o uso do comando `crontab`. Adicionalmente o processo `cron` lê o ficheiro `/etc/crontab`, que tem um formato diferente dos anteriores. Por exemplo, inclui o campo de utilizador para permitir executar um determinado comando/script com as permissões do respectivo utilizador. Todos os ficheiros de configuração da subdirectoria `/etc/cron.d` são também lidos; embora com um formato semelhante do `/etc/crontab`, estes não herdam por exemplo variáveis de ambiente, e o seu uso é mais restrito. Da mesma forma, todos os ficheiros que se encontram nas directorias `/etc/cron.daily`, `/etc/cron.weekly` e `/etc/cron.monthly` são lidos.

Uma vez ocorrida uma alteração a qualquer um dos ficheiros de configuração, não é necessário executar novamente o programa `cron`. Na verdade, este verifica de minuto a minuto se houve alguma alteração em algum ficheiro de configuração mediante a consulta do campo `modtime` do sistema de ficheiros.

O administrador deverá usar o ficheiro `/etc/crontab` e não a subdirectoria `/etc/cron.d`.

Os utilizadores podem também definir os seus comandos/scripts. Para isso deverão ter permissão no ficheiro `/etc/cron.allow` ou não estar listados no ficheiro `/etc/cron.deny`. Se estes ficheiros

não existirem, dependendo da configuração do sistema, apenas o administrador (root) poderá usar este comando.

Para mostrar o crontab do utilizador deverá usar-se o comando:

```
crontab -l
```

4.2.1 Formato do ficheiro

Um ficheiro *crontab* contém instruções para o programa *cron*. Os comentários são indicados por um #. Ou seja, tudo o que siga um # é ignorado.

Uma linha no ficheiro representa ou um comando ou uma definição de ambiente. Por exemplo

```
nome=valor
```

A string valor pode ser definida entre aspas para conter por exemplo espaços. Algumas variáveis são automaticamente definidas. A shell é definida como sendo */bin/sh* e LOGNAME e HOME são definidos a partir do ficheiro */etc/passwd* para o respectivo utilizador.

Os comandos são executados pelo cron quando o minuto, a hora, e o mês do ano forem iguais à data actual. Um comando é composto por 6 campos, sendo os primeiros 5 dedicados à data e hora e o sexto ao comando a executar propriamente dito. Os campos da hora e data são (por ordem):

minuto	hora	dia do mês	mês	dia da semana	comando
0-59	0-23	1-31	1-12	0-7	“um comando shell qualquer”

Um campo pode também ser um asterisco (*), que significa “primeiro-ultimo” valor. O campo pode também ser definido como um intervalo. Um intervalo é definido como sendo dois números separados por um hífen. Por exemplo “0-3” na hora, significa executar o comando às 0h, 1h, 2h e 3h. Pode especificar-se um conjunto discreto de valores, devendo estes ser definidos entre aspas e separados por vírgulas. No caso dos minutos, “10,15,20” significa executar o comando aos 10, 15 e 20 minutos da hora, respectivamente. Nos intervalos é possível também definir o passo da repetição usando o carácter /. Por exemplo, “1-9/2” é o mesmo que “1,3,5,7,9”, ou “*/2” no campo das horas significa de duas em duas horas.

Exemplo. Exemplos de linhas num ficheiro *crontab*. A linha 1 executa o comando de 1 em 1 minuto; a linha 2 executa o comando às 0h30, 1h30, 2h30, etc.; a linha 3 executa o comando de 10 em 10 minutos.

```
* * * * * /home/manuel/atualiza-lista-de-pessoas
30 * * * * /home/manuel/faz-backup.sh
*/10 * * * * $HOME/verifica-ataques.sh
```

4.2.2 Definir um agendamento para um utilizador

Para definir um crontab para um determinado utilizador, pode primeiro criar-se um ficheiro com os comandos para o cron. Suponha que pretende escrever as horas de minuto a minuto no ficheiro *horas.txt* que está na sua *home directory*. Deve começar por criar um ficheiro, por exemplo *cron.txt*, com o seguinte conteúdo:

```
* * * * * date >> $HOME/horas.txt
```

De seguida deverá executar-se o comando, que ativará os comandos pelo cron:

```
crontab cron.txt
```

Para verificar que o comando foi bem sucedido pode verificar-se com

```
crontab -l
```

O exemplo anterior acrescenta ao ficheiro a hora atual de minuto a minuto. Pode confirmar que o cron está a funcionar visualizando o conteúdo do ficheiro vários minutos depois. Para apagar o crontab anterior pode usar-se o comando:

```
crontab -r
```

4.2.3 Exemplos

Exemplo de um crontab:

```
SHELL=/bin/bash
# não interessa de que é o crontab, manda mail para o "rui"
MAILTO=rui
#
# corre às 0:05 todos os dias
5 0 * * *      $HOME/bin/tarefa-diaria >> $HOME/tmp/out 2>&1
# corre às 14:15 no primeiro dia de cada mês -- manda mail para rui
15 14 1 * *    $HOME/bin/backup-mensal
# corre às 22h em dias de semana, só para chatear o manuel
0 22 * * 1-5    mail -s "são 10h" manuel%Manuel,%onde estás?%
23 0-23/2 * * * echo "corre às 0:23, 2:23, 4:23, ..., todos os dias"
5 4 * * sun     echo "corre às 4:05 todos os domingos (sunday)"
```

O ficheiro `/etc/crontab` tem uma estrutura idêntica, no entanto existe um campo adicional para o utilizador, a seguir aos campos da data e hora e o comando.

4.3 Estrutura do sistema de ficheiros

Esta secção descreve as partes mais importantes do sistema de ficheiros Linux, baseado no Filesystem Hierarchy Standard (FHS). Os sistemas de ficheiros encontram-se normalmente organizados sob a forma de uma árvore. As folhas, ou os nós da árvore, correspondem a directorias ou ficheiros. A árvore começa na raiz (root), a que corresponde a directoria `/`. A motivação para a divisão da hierarquia do sistema de ficheiros em diferentes ramos é apresentada de seguida. Mais informações em https://en.wikipedia.org/wiki/Filesystem_Hierarchy_Standard.

4.3.1 A directoria raiz (*root directory*)

A raiz do sistema de ficheiros contém várias outras directorias, comuns aos vários sistemas Unix-like, que agrupam os diferentes tipos de ficheiros, de acordo com a sua semântica:

<code>/bin</code>	Comandos necessários durante o processo de arranque do sistema operativo, que também possam ser usados pelos utilizadores (provavelmente após o processo de arranque).
-------------------	--

/sbin	Comandos que não se destinam aos utilizadores normais, mas sim ao utilizador root. Usualmente esta directoria não se encontra na PATH dos utilizadores normais. Como exemplo de comandos guardados nesta directoria temos: <i>ifconfig</i> – configura as interfaces de rede; <i>modprobe</i> – carrega módulos no kernel do sistema operativo; <i>init</i> – será explicado mais à frente.
/etc	Nesta directoria são guardados os ficheiros de configuração específicos da máquina onde o sistema operativo se encontra instalado. Contém os ficheiros de configuração do sistema operativo, bem como de diversos serviços (ex: servidor web, bases de dados, etc..).
/root	A directoria home do utilizador root.
/lib	Directoria onde estão guardadas as bibliotecas partilhadas (shared libraries) necessárias à execução dos programas.
/dev	Os ficheiros guardados nesta directoria servem como interface com diferentes dispositivos, por exemplo, teclado, discos rígidos, placas de rede etc... Não são ficheiros editáveis.
/tmp	Nesta directoria todos os utilizadores podem ler e/ou escrever. Esta serve para os diferentes programas escreverem em disco informação temporária. Nos sistemas Linux, normalmente os utilizadores têm um limite de espaço em disco para a directoria pessoal. A /tmp normalmente é limitada ao espaço da raiz /.
/boot	Nesta directoria guardam-se os ficheiros usados pelo bootstrap loader, por exemplo, LILO ou GRUB. As imagens do kernel são normalmente guardadas aqui. Esta directoria é normalmente guardada numa partição início do disco, ou eventualmente guardada separadamente noutro sistema de ficheiros.
/mnt, /media, /Volumes	Estas directorias servem para montar (mount) temporariamente dispositivos de armazenamento. Tipicamente, quando se liga uma pen ou se introduz um CDROM, é criada uma sub directoria para cada dispositivo. Esta tarefa pode ser feita automaticamente pelo sistema operativo, ou pelo administrador (root).
/proc, /usr, /var, /home	Estas directorias servem como pontos onde podem ser montados outros sistemas de ficheiros importantes (que veremos a seguir). A directoria /proc é um caso particular de uma directoria que não reside em nenhum disco.

4.4 Tarefas Comuns

4.4.1 Comprimir e descomprimir ficheiros

Existem variadas alternativas para comprimir ficheiros. As mais utilizadas em ambiente unix são: tar, gzip, bzip2, zip e unzip. O tar é provavelmente o mais utilizado, mas na sua forma mais simples apenas concatena ficheiros. Quando usado com outras as opções permite utilizar o gzip (-z) e bzip2 (-j). A tabela seguinte apresenta alguns exemplos de como compactar e descompactar ficheiros ou diretórios. Nos exemplos, a designação *ficheiros* corresponde a 1 ou mais nomes de ficheiros ou a diretórios:

extensões	compactar	descompactar
tar.gz, .tgz	tar zcvf <i>fich-compact.tar.gz</i> <i>ficheiros</i>	tar zxvf <i>fich-compact.tar.gz</i>
tar.bz2	tar jcvf <i>fich-compact.tar.bz2</i> <i>ficheiros</i>	tar jxvf <i>fich-compact.bz2</i>
.zip	zip -rp <i>fich-compact.zip</i> <i>ficheiros</i>	unzip <i>fich-compact.zip</i>

4.4.2 Enviar emails

Se o servidor estiver configurado, enviar emails a partir da linha de comando é um processo extremamente fácil e flexível. Pode, por exemplo, ser muito útil quando se gera um resultado e é necessário enviá-lo por mail.

Para verificar que o servidor está bem configurado, pode efetuar-se um primeiro teste com o comando seguinte, substituindo *endereco@email* pelo seu e-mail pessoal:

```
echo "Isto é um teste" | mail -s "Teste do mail" endereco@email
```

No exemplo anterior, o conteúdo do email será apenas “Isto é um teste”. Outro teste que se pode fazer é executar o comando seguinte, que ficará à espera que seja introduzido o conteúdo do mail através do teclado. O utilizador termina a sua mensagem pressionando Control+D.

```
mail -s "Teste do mail" endereco@email
```

Finalmente, se o utilizador pretende que o conteúdo do e-mail seja lido de um ficheiro, pode executar um comando semelhante ao seguinte.

```
mail -s "Teste do mail" endereco@email < mensagem.txt
```

Para verificar se o email foi enviado, pode utilizar o comando seguinte, que indica quais as mensagens que ainda estão em fila de espera para serem enviadas.

```
mailq
```

4.4.3 Comandos úteis

uname -a mostra informações sobre o sistema

lsuf mostra a lista de ficheiros abertos. A opção -u permite restringir a lista a um dado utilizador.

Exemplo: `lsuf -u maria`

dmesg mostra as mensagens de arranque do sistema

last mostra uma lista com os utilizadores que entraram no sistema recentemente

df -h mostra o espaço de armazenamento existente num formato mais agradável

Parte II

System Programming

5

Processos e sinais

5.1 Processos

Um programa em execução diz-se um *processo*. Cada processo é identificado pelo sistema operativo através de um número único - o *pid* (process identifier).

Experimente o seguinte programa:

```
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
main() {
    char s[80];
    printf ("PID=%d\n", getpid() );
    while ( 1 ) {
        printf ("Comando: ");
        fgets(s,80,stdin);
        s[strlen(s)-1]=0;
        if ( strcmp ( s, "exit" ) == 0 )
            exit(0);
    }
}
```

O programa começa por escrever no ecrã o seu identificador de processo; depois fica à espera que o utilizador introduza um comando. O único comando com efeito é o "exit", que faz o programa terminar. Tendo este programa em execução abra outra janela e execute o comando:

```
ps -u
```

Este comando permite ver a lista de todos processos pertencentes ao mesmo utilizador. Verifique que o programa aparece na lista (pode identificá-lo pelo *pid*).

Se puser este mesmo programa em execução em várias janelas terá vários processos diferentes (todos a executar o mesmo programa). Experimente e verifique com `ps -u`.

5.1.1 Chamadas ao sistema (system call)

O programa do ponto anterior usa a função `getpid()` para saber o seu *pid*.

A função `getpid()` origina uma chamada ao kernel do sistema operativo (system call). Da mesma forma, a função `exit()` origina uma chamada ao sistema operativo, neste caso pedindo ao sistema operativo para terminar o processo. Para saber mais pormenores sobre estas funções pode consultar o manual. Por exemplo:

```
man getpid
```

Um dos aspectos importantes do manual é a indicação dos "*includes*" que deve fazer para usar a função (pormenor que neste caso descurámos...).

Deve executar o comando `man nome_da_função` para saber quais os includes que deve fazer no seu programa em C no qual faça uso de uma qualquer função de sistema.

Estas funções de sistema operativo (às vezes chamadas também de "primitivas") são apresentadas na secção 2 dos manuais do Unix/Linux. Ocasionalmente o nome de uma função do sistema pode coincidir com o nome de um comando que esteja noutra secção do manual. Nesses casos pode ser preciso explicitar a secção do manual onde se pretende procurar. Por exemplo:

```
man 2 getpid
```

pede explicitamente o manual de `getpid` existente na secção 2.

Na realidade `getpid()` ou `exit()` são funções de biblioteca da linguagem C a que, com alguma liberdade perfeitamente aceitável, chamamos *system calls*. Mais rigorosamente, o que estas funções fazem é preparar a chamada ao sistema, a qual, especificamente, é feita através de uma interrupção de software ("trap") ao sistema operativo.

5.1.2 Hierarquia de processos

Quando abrimos uma janela "consola" no interface gráfico ou fazemos login num ecrã alfanumérico ficamos perante um programa que lê e executa os nossos comandos. Um programa deste tipo chama-se uma "shell". No Linux a variante de shell mais provável é a designada `bash` (programa `/bin/bash`).

O trabalho da shell é repetitivo. Basicamente, consiste em:

- pedir um comando (fazendo aparecer o "pronto">);
- ler e interpretar o comando;
- executar o comando (ou dar erro, se o comando for inválido)

e repetir, fazendo reaparecer o "pronto" para pedir um novo comando.

Os comandos podem ter efeito e duração variados. Por exemplo um `ls -l` ou um `ps` listam informação no ecrã e terminam de imediato. Um comando como o `vi` toma conta do ecrã e termina apenas quando o utilizador terminar a edição saindo do `vi`. Seja como for, no fim do comando, a shell reaparece para pedir outro comando.

Para executar um comando, a shell cria um novo processo. O novo processo fica ligado (diz-se que é um processo filho) do processo que o criou (que se diz o seu processo pai). Um processo pode saber o número do seu processo pai através da função `getppid()`. O seguinte programa exemplifica esta relação:

```
#include <stdio.h>
main() {
    printf ("PID = %d\n", getpid() );
    printf ("Processo pai = %d\n", getppid() );
}
```

O programa escreve no ecrã o seu número de processo e o número do seu processo pai. Sem surpresa, como pode verificar facilmente executando este programa e depois fazendo `ps`, o *pid* do processo pai é justamente o pid da shell que o criou.

5.1.3 Shell: comandos e programas

A generalidade dos comandos que damos à shell para executar são, na realidade, programas externos à própria shell. Nestes casos o que a shell faz é localizar o programa correspondente ao comando dado e criar um novo processo para o executar.

Por exemplo, o comando `ls` corresponde a um programa que se pode encontrar, tipicamente, no directório `/bin` (pode verificar fazendo `ls /bin/ls`). Assim, executar um comando como `ls -l` é, na realidade, executar o programa `/bin/ls`, passando-lhe `-l` como argumento.

No mesmo directório `/bin` pode também encontrar muitos outros comandos (programas) de uso comum: `cp`, `ps`, etc. Possivelmente encontrará também aí o próprio programa shell, num ficheiro `/bin/bash`. Se fizer o comando

```
bash
```

iniciará a execução de uma nova shell, que ficará "sobrepota" à anterior, passando a aceitar comandos na janela/terminal. Nestas circunstâncias, fazendo `ps`, verá aparecer dois processos `bash` (ou mais, se executar o comando `bash` várias vezes). Para terminar a shell dá-se o comando:

```
exit
```

Este é um exemplo de um comando interno, isto é, um comando executado pela própria `bash`.

5.1.4 Criação de processos: `fork()`

Para criar um novo processo usa-se a função `fork()`. Este função faz uma chamada ao sistema que cria um novo processo, duplicando o original. O seguinte exemplo simples ilustra a execução do `fork()`:

```
#include <stdio.h>
main() {
    printf ("Início\n" );
    fork();
    printf ("Fim\n" );
}
```

O programa começa por escrever "Início"; depois invoca o `fork()` que cria um novo processo clone do processo original. A partir daí passam a existir dois processos e ambos vão executar o segundo `printf`, escrevendo a mensagem "Fim" no ecrã.

O clone criado pelo `fork()` é, à partida, inteiramente igual ao original: o mesmo programa, com as mesmas variáveis, começando a ser executado a partir do ponto do `fork()`. Mas, obviamente, sendo outro processo, o `pid` é diferente do original. O exemplo seguinte ilustra essa diferença:

```
main() {
    printf ("Início PID=%d\n", getpid() );
    fork();
    printf ("Fim PID=%d\n", getpid() );
}
```

uma das mensagens "Fim" dá um `pid` igual ao original, outra dá um `pid` diferente - sendo esta última, obviamente, a que é escrita pelo novo processo, criado pelo `fork()`.

O novo processo criado pelo `fork()` diz-se um processo filho, sendo o processo pai o original. O exemplo seguinte ilustra esta relação:

```
main() {
    printf ("Início PID=%d filho de %d\n", getpid(), getppid() );
    fork();
    printf ("Fim PID=%d filho de %d\n", getpid(), getppid() );
}
```

A mensagem de fim escrita pelo programa original indica a shell como processo pai; a mensagem de fim escrita pelo novo processo indica o processo original como processo pai.

5.1.5 Mais sobre o fork(): processo pai e processo filho

Como se disse, o processo pai e processo filho originado por um `fork()` são inicialmente inteiramente semelhantes. Há apenas um pequeno, mas decisivo, pormenor distintivo: o valor que a própria função `fork()` devolve para cada um deles. Para o novo processo, o `fork()` devolve o valor 0; para o processo original devolve o PID do filho, ou seja, um valor diferente de zero.

Esta diferença permite distinguir o processo original e o processo filho depois do `fork()` e, com base nisso, traçar um caminho diferente para cada um deles. O exemplo seguinte ilustra essa técnica:

```
main() {
    printf ("Início\n" );
    int n = fork();
    if ( n == 0 )
        printf ("Eu sou o processo filho\n");
    printf ("Fim \n" );
}
```

Consideremos cada um dos dois processos depois do `fork()`. O processo filho segue para a condição do `if, n==0`, que dá Verdade; desta forma executa o `printf` que escreve no ecrã a mensagem "Eu sou..."; depois disso escreve a mensagem "Fim" e termina. O processo original segue também para a condição do `if` que, nesse caso, dá Falso; depois escreve a mensagem "Fim" e termina.

O código seguinte utiliza a primitiva `exit` para garantir que o processo filho fica confinado às instruções que se encontram dentro do `if`.

```
main() {
    printf ("Início\n" );
    int n = fork();
    if ( n == 0 ) {
        printf ("Eu sou o processo filho\n");
        exit(0);
    }
    printf ("Fim do pai\n" );
}
```

5.1.6 Execução de outros programas: exec

A chamada ao sistema `exec` permite a um processo em curso alterar o programa que está a executar. O seguinte exemplo ilustra o conceito:

```
#include <stdio.h>
main() {
    printf ("Executar outro programa...\n");
    execl( "/bin/ls", "ls", "-l", NULL );
    printf ("Não resultou!\n");
}
```

O programa começa por escrever a mensagem "Executar outro programa...". Depois faz um `exec`, pedindo ao sistema operativo para passar a executar o programa `/bin/ls`. Se a chamada tiver sucesso, aparece no ecrã o resultado de `"ls -l"`.

Atenção que não se trata de mandar executar o outro programa e voltar; trata-se de mudar, definitivamente, de um programa para outro. Neste caso, trata-se de mudar deste programa para o programa `ls`; se a mudança resultar, será executado o `ls` e a mensagem "Não resultou!" já não aparecerá.

A função `execl()` é uma das formas de fazer a chamada `exec` ao sistema operativo. Se executar o comando `man execl` poderá ver outras funções para o mesmo efeito, que variam na forma de disposição dos argumentos que configuram a chamada ao sistema.

5.1.7 Executar outro programa: `fork` e `exec`

A conjugação de `fork` e `exec` permite a um programa mandar executar um outro programa sem se "autodestruir" (como aconteceria de fizesse apenas um `exec`).

O seguinte exemplo ilustra a técnica. Ao fazer `fork()` encaminha o processo filho para um `exec`. Desta forma, o processo filho passa a executar um outro programa (no caso, `"ls -l"`) enquanto o programa original prossegue (neste caso para escrever a mensagem "Fim" e depois terminar). O `printf` que segue o `execl` nunca será executado, a não ser que o `execl` tenha falhado (porque o `/bin/ls` não existe, por exemplo). A instrução `exit` é fundamental para garantir que se algo correr mal o processo filho não passará também a executar as instruções do pai.

```
#include <stdio.h>
#include <unistd.h>
main() {
    int n;
    printf ("Início\n" );
    n = fork();
    if ( n == 0 )    {
        execl ( "/bin/ls", "ls", "-l", NULL );
        printf("correu mal...\n");
        exit(0);
    }
    printf ("Fim \n" );
}
```

5.1.8 `wait`

Consideremos ainda o exemplo anterior. Vamos supor que queremos garantir que a mensagem "Fim", escrita pelo processo original, apareça no ecrã depois do resultado do `ls` produzido pelo processo filho.

Uma vez criado o novo processo filho, ambos os processos competem no sistema por tempo de processamento. O sistema operativo executa os processos pela ordem e sequência que entende; não podemos, por isso, presumir o que quer que seja sobre qual dos processos é executado primeiro.

Há, entretanto, mecanismos do sistema operativo que nos permitem controlar a sequência dos processos. Um desses mecanismos é a primitiva `wait()`, que nos permite parar o processo original até que o processo filho termine. A utilização do `wait()` para esse efeito é ilustrada no seguinte exemplo:

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

```

main() {
    printf ("Início\n" );
    if ( fork() == 0 ) {
        execl ( "/bin/ls", "ls", "-l", NULL );
        exit(1);
    }
    wait(NULL);
    printf ("Fim \n" );
}

```

O `wait()` faz com que o processo fique bloqueado (parado) até que um processo filho termine. Assim, depois do `fork()`, enquanto o processo filho segue para o `execl` executando “ls -l”, o processo original vai para o `wait()`, onde fica parado até que o processo filho acabe – só depois passa o `wait()` e prossegue escrevendo a mensagem “Fim”.

Devemos também precaver a possibilidade de o `execl` falhar. Neste caso haverá poucas possibilidades, mas em situações mais gerais pode acontecer (se o programa invocado não existir, não tiver permissões de execução, etc.); se o `execl` falhar, o processo filho segue para `exit()` e termina sem mais efeitos.

5.1.9 Exemplo: uma pequena shell

O exemplo seguinte ilustra uma pequena shell capaz de receber e mandar executar um comando (os comandos têm que ser dados com nome completo, por exemplo `/bin/ls`)

```

#include <stdio.h>
#include <string.h>
#include <unistd.h>
main() {
    char s[80];
    while ( 1 ) {
        printf ("Comando> ");
        fgets ( s, 80, stdin );
        s[strlen(s)-1] = 0;
        if ( strcmp ( s, "exit" ) == 0 )
            exit(0);
        if ( fork() == 0 ) {
            execl ( s, "", NULL );
            printf ("%s: comando inválido.\n", s );
            exit(1);
        }
        wait(NULL);
    }
}

```

5.2 Sinais

O mecanismo mais rudimentar de comunicação entre processos é o envio de sinais. Vamos começar por ver este mecanismo através de um exemplo simples, conjugando dois processos:

- Processo 1: executa um programa que publica o seu PID e depois entra em espera;
- Processo 2: manda um sinal ao processo P1;

Os programas são os seguintes:

p1.c

```
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
main() {
    char s[100];
    printf ("PID=%d\n", getpid() );
    while ( 1 ) {
        printf ("> ");
        scanf("%s", s);
        if ( strcmp(s, "exit") == 0 ) exit(0);
    }
    printf ("Fim do Processo P1");
}
```

envia.c

```
#include <stdio.h>
#include <sys/types.h>
#include <signal.h>
main() {
    int n;
    printf ("Enviar sinal para o processo: ");
    scanf("%d", &n);
    kill ( n, SIGKILL );
}
```

Ponha o programa P1 em execução e anote o respectivo PID; indique esse mesmo PID no programa P2, enviando assim um sinal ao processo P1. De momento, o efeito visível do envio do sinal é a terminação do processo P1.

O SIGKILL é um dos sinais que podem ser enviados de um processo para outro. Além deste há diversos outros sinais, como se pode ver através do comando:

```
man 7 signal
```

Cada sinal é identificado por um número. Por exemplo, o sinal enviado no exemplo anterior tem o número 9.

O ficheiro signal.h contém a definição de uma mnemónica para cada um desses números. Por exemplo:

```
#define SIGKILL 9
```

Exercício. experimente o envio do sinal SIGUSR1 em vez de SIGKILL (de momento, o efeito não será muito diferente: na mesma termina o processo de destino).

5.2.1 Enviar um sinal

O comando kill tem o mesmo nome e um efeito semelhante ao da função kill: permite o envio de um sinal a um processo. A forma geral para este efeito é:

```
kill -sinal pid
```

Sendo `signal` o número do sinal a enviar e `pid` o número de processo ao qual o sinal vai ser enviado. Exemplo:

```
kill -9 1000
kill -16 1000
```

O primeiro envia o sinal SIGKILL ao processo 1000; o segundo envia ao mesmo processo o sinal SIGUSR1.

Apesar da função se chamar `kill`, nem sempre o envio de um sinal tem por consequência a "morte" do processo a quem o sinal é enviado. Dentro de certos limites o processo pode "capturar" o sinal que lhe é enviado e controlar a forma de resposta. Basicamente, há duas alternativas:

- Ignorar o sinal;
- Executar determinada função quando o sinal for recebido;

Ambas as acções são indicadas através da função `signal`:

```
signal ( sinal, handler)
```

A função tem dois argumentos: `signal` e `handler`. No primeiro indica-se o número do sinal que se pretende capturar. No segundo especifica-se a acção a tomar se e quando (algures no tempo) o sinal vier a ser recebido.

Para ignorar o sinal indica-se no argumento `handler` a constante `SIG_IGN`. Exemplo: modifique o programa anterior de forma a ignorar o sinal SIGUSR1:

`ignora.c`

```
#include <stdio.h>
#include <signal.h>
#include <stdlib.h>
main() {
    char s[100];
    signal ( SIGUSR1, SIG_IGN ); //ignorar o sinal SIGUSR1
    printf ("PID=%d\n", getpid() );
    while ( 1 ) {
        printf ("> ");
        gets(s);
        if ( !strcmp(s, "exit") ) exit(0);
    }
    printf ("Fim do Processo P1");
}
```

Experimente agora o envio do sinal SIGUSR1, através do programa `p2` ou do comando `kill`. O processo deverá ser insensível a este sinal.

5.2.2 Tratar um sinal

Em vez de ignorar o sinal, podemos indicar uma função para ser executada quando o sinal for recebido. Exemplo:

`trata.c`

```

#include <signal.h>
#include <string.h>
#include <stdlib.h>
#include <stdio.h>

void f_resposta(int sinal) {
    printf ("Recebi o sinal nº %d\n", sinal);
}

main() {
    char s[100];

    signal (SIGUSR1, f_resposta ); //armar o sinal...
    printf ("PID=%d\n", getpid() );
    while ( 1 ) {
        printf ("> ");
        gets(s);
        if ( !strcmp(s, "exit") ) exit(0);
    }
    printf ("Fim do Processo P1");
}

```

A chamada à função signal

```
signal (SIGUSR1, f_resposta );
```

Estabelece que, em resposta ao sinal SIGUSR1, deve ser executada a função f_resposta.

Experimente pôr o processo em execução e enviar-lhe o sinal SIGUSR1. Em resposta deverá aparecer no ecrã a mensagem escrita pela função f_resposta.

Note que a chamada o signal apenas indica a função a executar como resposta ao sinal (dizemos "arma o sinal"). A função indicada só será realmente executada se e quando o processo em causa receber um sinal.

Note ainda que o sinal permanece armado da mesma forma até nova chamada a signal. No caso do exemplo anterior, o processo responde com a mensagem indicada em f_resposta de cada vez que receber um sinal SIG_USR1. Este comportamento só poderia ser alterado com nova chamada a signal.

Exemplo: neste caso o processo responde uma única vez ao sinal:

```

f_resposta ( int sinal ) {
    printf ("SINAL Recebido. Desligar...\n");
    signal ( SIGUSR1, SIG_IGN );
}

```

5.2.3 alarm

A primitiva alarm permite a um processo mandar um sinal temporizado a si próprio. Por exemplo, a chamada:

```
alarm ( 5 )
```

faz com o próprio processo receba um sinal SIGALRM passado o número de segundos indicado no argumento. Exemplo:

```

#include <signal.h>
#include <string.h>
#include <stdlib.h>
#include <stdio.h>

int n;
void f(int sinal) {
    if ( sinal == SIGALRM ) {
        printf ("Contagem: %d\n", n--);
        if ( n > 0 )
            alarm(1);
    }
}
main() {
    char s[100];
    n = 5;
    signal ( SIGALRM, f );
    alarm (1);
    while ( 1 ) {
        printf ("> ");
        gets(s);
        if ( !strcmp( s, "exit" ) ) exit(0);
    }
}

```

O mecanismo de alarm é a base para a implementação de temporizações no processo. Por exemplo, para implementar um tempo de espera, pode-se lançar um *alarm* ficando o processo a fazer tempo até receber o respectivo sinal:

```

#include <signal.h>
#include <string.h>
#include <stdlib.h>
#include <stdio.h>

int n;
void f(int sinal) {
    if ( sinal == SIGALRM ) n = 1;
}
main() {
    // pausa de 5 segundos...
    printf ("esperar 5 segundos...\n");
    signal ( SIGALRM, f );
    alarm (5);
    while ( n == 0 ) ;
    printf ("ok\n");
}

```

Atenção: não é tempo real.

5.2.4 Espera ativa e passiva

O problema com a implementação anterior é que o processo enquanto espera, apesar de não estar a fazer nada de útil, está ainda assim a consumir recursos de máquina como um processo normal (e dos piores porque, não tendo i/o nem sequer bloqueia). Diz-se, neste caso que o processo está em *espera activa*: executando normalmente, instruções inúteis, à espera de determinado acontecimento. Em alternativa, o processo pode esperar na situação de bloqueado. Ou seja: lança o alarme

e depois pede ao sistema operativo que o bloqueie até receber o sinal. A vantagem é que, enquanto bloqueado, não consome recursos significativos, em particular o SO não o põe a executar.

Para um processo de bloquear chama a função `pause()`. A implementação alternativa pode então ser a seguinte:

```
{  
    // pausa de 5 segundos...  
    printf ("esperar 5 segundos...\n");  
    signal ( SIGALRM, f );  
    alarm (5);  
    while ( n == 0 )  
        pause();  
    printf ("ok\n");  
}
```

Este procedimento é semelhante ao que é implementado pela primitiva `sleep(sec)` que põe o processo em espera durante o número de segundos indicado no argumento.

Exercício: use o comando `ps` para ver diferenças entre as duas versões do processo.

6

System V - IPCs

Este capítulo apresenta os mecanismos de comunicação entre processos normalmente designado de "IPCs do Sistema V", designadamente filas de mensagens, semáforos e memória partilhada. As filas de mensagens permitem aos processos trocarem mensagens entre si. Os semáforos permitem a sincronização (coordenação de acções) entre processos. E a memória partilhada permite a partilha de dados (variáveis) entre processos, normalmente recorrendo à ajuda de semáforos para disciplinar o acesso simultâneo aos dados.

6.1 Filas de mensagens

Uma fila de mensagens pode ser descrita como lista, composta por várias mensagens, guardada num espaço de endereçamento pertencente ao núcleo do Sistema Operativo. Vários processos podem enviar mensagens para a fila e vários outros processos podem ler essas mensagens. Sempre que uma mensagem é lida, é removida da fila.

Cada mensagem pode ter associada a si um número inteiro longo não negativo, que se identifica como sendo o tipo da mensagem, que permite que um dado processo receptor possa esperar apenas mensagem de um determinado tipo. Permite ainda estabelecer esquemas baseados em prioridades nas filas de mensagens.

Quando um processo tenta escrever uma mensagem numa fila cheia, bloqueia até que exista espaço suficiente na fila. De modo semelhante, um processo bloqueia quando tenta ler mensagens de um dado tipo e não existe nenhuma mensagem desse tipo na fila de mensagens.

6.1.1 Criação de uma fila de mensagens

Considere o programa da Figura 6.1.1 que usa a função `msgget` para criar uma fila de mensagens (ou "mailbox"). A função `msgget` recebe dois argumentos. O primeiro é uma chave que tem de ser única para cada fila de mensagens criada no sistema. Neste exemplo usamos normalmente esta chave 1000; adiante voltaremos a este assunto.

No segundo argumento indicam-se duas coisas: `IPC_CREAT`, que pede para criar a fila de mensagens; e 0666 que indica as permissões da fila de mensagens.

As permissões têm uma leitura semelhante ao das permissões no sistema de ficheiros. Neste caso o (número octal) 666 indica o equivalente a `rw-` ou seja a permissão de "read" e "write" quer para o dono da fila de mensagens quer para todos os outros utilizadores.

Caso a fila de mensagens seja criada com sucesso a função `msgget` devolve um identificador que depois poderá ser usado para enviar ou receber mensagens através da mailbox. Em caso de erro devolve um número negativo; nesse caso o programa publica uma mensagem de erro e termina.

```

#include <stdio.h>
#include <stdlib.h>
#include <sys/ipc.h>
#include <sys/msg.h>
main() {
    int id = msgget ( 1000, IPC_CREAT | 0666 );

    if ( id < 0 ) {
        printf ("Erro na criação da fila de mensagens\n" );
        exit(1);
    }
    printf ("Criada fila de mensagens com id %d\n", id);
}

```

Figura 6.1.1: Criar uma fila de mensagens.

6.1.2 Comandos ipcs

A fila de mensagem é um recurso permanente mantido pelo kernel do sistema operativo. Uma vez criado permanece até se pedir ao sistema para o eliminar (ou até o sistema ser desligado).

Ou seja, caso o programa anterior tenha sucesso é criada uma fila de mensagens que fica no sistema. O comando `ipcs` permite listar as filas de mensagens existentes. Fazendo o comando

```
ipcs -q
```

poderá aparecer um output do semelhante a este:

```

----- Message Queue -----
key      msqid      owner      perms      bytes      msg
0x000003e8  0          jrg        666        0          0

```

A coluna `key` mostra obviamente a chave usada para criar a fila de mensagens (1000 no exemplo, nesta lista representado em hexadecimal). A coluna `msqid` mostra o id da fila, o mesmo que recebemos da função `msgget`.

O comando `ipcrm` permite eliminar uma fila de mensagens. Por exemplo, o comando

```
ipcrm -q 0
```

elimina a fila de mensagens com o id dado (neste caso 0). Se fizermos agora

```
ipcs -q
```

a lista aparecerá vazia. Se executarmos o programa outra vez a fila voltará a ser criada, com outro id. Fazendo o `ipcs` a nova fila poderá aparecer, por exemplo:

```

----- Message Queue -----
key      msqid      owner      perms      bytes      msg
0x000003e8  32768      jrg        666        0          0

```


6.1.3 Macro `exit_on_error`

Após uma chamada ao sistema devemos sempre verificar se correu bem ou se deu erro.

A generalidade das chamadas ao sistema devolvem um número negativo quando não têm sucesso.

Sendo assim, a seguir a uma chamada ao sistema faremos uma construção do género

```
n = chamada_ao_sistema ( ... );
if ( n < 0 ) {
    // erro: publicar uma mensagem de erro e sair
    printf ("... erro....");
    exit( ... );
}
// ! sucesso : continuar o programa
```

Para a mensagem de erro, em vez de um `printf` formado por nós, podemos usar a função `perror` que mostra uma mensagem de erro produzida pelo sistema operativo.

Traduzimos este arranjo na seguinte macro

```
#define exit_on_error(s,m) if ( s < 0 ) { perror(m); exit(1); }
```

que passaremos a usar normalmente após cada chamada ao sistema.

Com esta macro o programa anterior ficaria:

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#define exit_on_error(s,m) if ( s < 0 ) { perror(m); exit(1); }

main() {
    int id = msgget ( 1000, IPC_CREAT | 0666 );
    exit_on_error ( id, "Criação da fila de mensagens");
    printf ("Criada fila de mensagens com id %d\n", id);
}
```

6.1.4 Header file

É possível definir um ficheiro que inclui todos os `#includes` e eventuais macros, de forma a não ter de escrever tudo isso num novo programa. Por exemplo, se escrever o ficheiro `definicoes.h` com o seguinte conteúdo

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#define exit_on_error(s,m) if ( s < 0 ) { perror(m); exit(1); }
```

o programa anterior pode ser re-escrito como

```
#include "definicoes.h"
main() {
    int id = msgget ( 1000, IPC_CREAT | 0666 );
    exit_on_error ( id, "Criação da fila de mensagens");
    printf ("Criada fila de mensagens com id %d\n", id);
}
```

6.1.5 Envio de uma mensagem

Considere o seguinte exemplo:

```
#include "definicoes.h"
#include <string.h>

typedef struct {
    long tipo;
    char texto[250];
} MsgStruct;

main() {
    int msg_id;
    int status;
    MsgStruct msg;

    // ligar à fila de mensagens
    msg_id = msgget ( 1000, 0600 | IPC_CREAT );
    exit_on_error (msg_id, "Criação/Ligação");

    // enviar uma mensagem
    msg.tipo = 1;
    strcpy( msg.texto, "Hello");
    status = msgsnd( msg_id, &msg, sizeof(msg.texto), 0);
    exit_on_error (status, "Envio");

    printf ("Mensagem enviada!\n");
}
```

Ligação

O primeiro passo para poder usar a fila de mensagens é ligar-se a essa fila; para isso usa-se, também, a função `msgget`. Digamos que a função `msgget` serve para duas coisas: para criar uma fila de mensagens ou para fazer a ligação a uma fila de mensagens previamente criada.

Na prática a chamada do exemplo serve para as duas coisas porque a opção `IPC_CREAT` faz com que a função crie a fila de mensagem caso não exista. Se quisermos apenas ligar à fila (e não criar a fila caso não exista) omitimos o `IPC_CREAT`:

```
msg_id = msgget( 1000, 0666 );
```

Neste caso a função daria erro se não existisse uma fila criada com chave 1000.

Estrutura de uma mensagem

Para se poder enviar uma mensagem é preciso criar uma estrutura como a que se ilustra no exemplo:

```
typedef struct {
    long tipo;
    char texto[250];
} MsgStruct;
```

A estrutura contém dois campos: o primeiro, do tipo `long`, é obrigatório. Podemos escolher o nome, mas é obrigatório que seja o primeiro e que seja do tipo `long`. Mais adiante veremos a utilidade.

O outro contém o espaço para o envio da mensagem propriamente dita, neste caso uma string contendo até 250 caracteres.

A mensagem a enviar é então constituída por uma variável do tipo `MsgStruct` que no exemplo é preenchida da seguinte forma:

```
msg.tipo = 1;
strcpy( msg.texto, "Hello");
```

Notar o tipo = 1 que é relevante para, posteriormente, se pode receber ("ir buscar") a mensagem.

Envio

O envio propriamente dito é feito chamando a função `msgsnd` ("message send"). O `msgsnd` recebe como argumentos o id da fila de mensagens, a variável com a mensagem (na realidade o endereço dessa variável, formado pelo operador `&`) e o tamanho da mensagem. Note que o tamanho é o `sizeof`, neste caso 250 (e não o `strlen`, que neste caso seria 5). No último argumento, de momento, pomos sempre 0.

A mensagem enviada fica na fila até alguém a receber ("ir buscar"). Podemos aliás um reflexo disso vendo o resultado do `ipcs -q` que agora dará alguma coisa do género:

```
----- Message Queue -----
key          msqid      owner          perms        bytes       msg
0x000003e8   32768         jrg           666         250         1
```

indicando que há uma mensagem em espera na fila.

Se executar o programa de novo mandará outra mensagem; ficarão, então, duas mensagens na fila à espera de serem levantadas.

6.1.6 Recepção de uma mensagem

Considere o seguinte exemplo:

```
#include "definicoes.h"
#include <string.h>

typedef struct {
    long tipo;
    char texto[250];
} MsgStruct;

main() {
    int msg_id;
    int status;
    MsgStruct msg;

    // ligar à fila de mensagens
    msg_id = msgget ( 1000, 0 );
    exit_on_error (msg_id, "Criação/Ligação");

    // receber uma mensagem (bloqueia se não houver)
    status = msgrcv( msg_id, &msg, sizeof(msg.texto), 1, 0);
    exit_on_error (status, "Recepção");

    printf ("MENSAGEM <%s>\n", msg.texto);
}
```

O programa usa a função `msgrcv` para receber (ir buscar) umas das mensagens que estejam na fila. Ao executar este programa deve aparecer o ecrã

```
MENSAGEM <Hello>
```

Os argumentos da função `msgrcv` são até certo ponto parecidos com os do `msgsnd`: primeiro o id da fila, depois uma variável com a estrutura da mensagem e o respectivo tamanho. A diferença é que, agora, a variável `msg` vai ser preenchida com a mensagem extraída da fila.

Há ainda um argumento importante, o último que deve aderir ao tipo da mensagem a recolher. Neste caso a mensagem foi enviada com o campo `tipo = 1` e, em correspondência, indicamos neste argumento que queremos receber uma mensagem enviada com o tipo 1.

6.1.7 Sincronização

A fila de mensagens é mais do que apenas um mecanismo que permite envio e recepção de mensagens. É também um mecanismo de sincronização, ou seja, um mecanismo que permite ordenar uma sequência de acções.

O ponto fundamental é este: o `msgrcv` bloqueia caso não haja nenhuma mensagem na fila. Quer dizer: se não há nenhuma mensagem na fila, o programa que faz `msgrcv` fica parado até que apareça uma nova mensagem - ou seja, até que outro programa faça um `msgsnd`.

Experiência: execute o programa do ponto 1.4 (aqui designado "enviar") numa janela e o programa do ponto 1.5 (aqui designado "Receber") noutra janela. Partindo da fila de mensagens vazia verifique a seguinte sequência

Janela 1	Janela 2	mgs na fila
		0
./Enviar		1
	./Receber	0
	./Receber -> bloqueia	0
./Enviar	-> desbloqueia	0
./Enviar		1
./Enviar		2
./Enviar		3
	./Receber	2
	./Receber	1
	./Receber	0
	./Receber -> bloqueia	

Assim, as filas de mensagens podem servir quer para diferentes programas (ou mais rigorosamente "processos") comunicarem entre si trocando mensagens, quer para sincronizarem as suas acções.

Imaginemos a arquitectura típica correspondente ao conceito "cliente"/"servidor". A ideia deste conceito é a cooperação entre dois tipos de processos: o "cliente" que submete pedidos; e o "servidor" que recebe e executa esses pedidos.

A ideia do "servidor" corresponde seguinte ciclo lógico:

```
repetir
```

```
esperar pedido
receber pedido
executar pedido
```

ou seja, um "ciclo infinito" em que o servidor está à espera de um pedido, quando recebe um pedido executa-o, e repete ficando à espera do próximo.

O "pedido" corresponde a uma mensagem colocada pelo cliente no fila de mensagens. O servidor está continuamente à espera que apareça uma mensagem; quando isso suceder executa-a e repete ficando à espera da seguinte. A ideia é traduzida no seguinte exemplo:

```
#include "definicoes.h"

typedef struct {
    long tipo;
    char texto[250];
} MsgStruct;

main() {
    //ligar à fila de mensagens
    int msg_id;
    msg_id = msgget ( 1000, 0666 | IPC_CREAT );
    exit_on_error (msg_id, "Criação/Ligação");

    int status;
    MsgStruct msg;

    while ( 1 ) {
        printf ("Esperar pedido...\n");

        status = msgrcv( msg_id, &msg, sizeof(msg.texto), 1, 0);
        exit_on_error (status, "Recepção");

        printf ("Recebido pedido <%s>. Executar!\n", msg.texto);
    }
}
```

Colocando este programa em execução ele ficará continuamente à espera do aparecimento de uma mensagem no fila de mensagens (o envio pode ser simulado com o programa do ponto 1.4) a que responderá com a mensagem ".... Executar!".

6.1.8 Estruturação das mensagens

Considere o seguinte exemplo:

```

#include "definicoes.h"

typedef struct {
    long tipo;
    struct {
        int num;
        char nome[100];
        int nota;
    } msg;
} MsgStruct;

main() {
    //ligar à fila de mensagens
    int msg_id;
    msg_id = msgget ( 1000, 0666 | IPC_CREAT );
    exit_on_error (msg_id, "Criação/Ligação");

    int status;
    MsgStruct m;

    while ( 1 ) {
        printf ("Esperar pedido...\n");

        status = msgrcv( msg_id, &m, sizeof(m.msg), 1, 0);
        exit_on_error (status, "Recepção");

        FILE *fp = fopen ("pauta.txt", "a" );
        fprintf(fp,"%d\n%s\n%d\n", m.msg.num, m.msg.nome, m.msg.nota);
        fclose(fp);
    }
}

```

Neste exemplo a mensagem tem um conteúdo estruturado, composto por um conjunto de campos. A estrutura `StructMsg` continua a incluir o primeiro campo obrigatório que define o tipo da mensagem; o conteúdo da mensagem é definido numa estrutura interna.

O exemplo represente um servidor que recebe mensagens, representando entradas de uma pauta, que vai descarregando para um ficheiro. O exemplo seguinte apresenta um programa que manda mensagens para o servidor:

```

#include "definicoes.h"
#include <string.h>

typedef struct {
    long tipo;
    struct {
        int num;
        char nome[100];
        int nota;
    } msg;
} MsgStruct;

int limpar_enter() {
    while ( getchar() != '\n' );
}

main() {
    //ligar à fila de mensagens
    int msg_id;
    msg_id = msgget ( 1000, 0666 | IPC_CREAT );
    exit_on_error (msg_id, "Criação/Ligação");

    int status;
    MsgStruct m;
    char s[100];
    m.tipo = 1;

    printf ("Numero: ");
    scanf ("%d", &(m.msg.num) );
    limpar_enter();

    printf ("Nome: ");
    fgets ( m.msg.nome, 100, stdin );
    m.msg.nome[ strlen(m.msg.nome) -1 ] = 0;

    printf ("Nota: ");
    fgets (s, 100, stdin );
    sscanf (s, "%d", &(m.msg.nota) );

    status = msgsnd( msg_id, &m, sizeof(m.msg), 0);
    exit_on_error (status, "Recepção");
}

```

6.1.9 Encaminhamento

O tipo das mensagens (que até agora deixámos sempre como 1) permite fazer o encaminhamento das mensagens (diferenciar emissores e destinatários).

Exemplo: pretendemos construir um modelo em que um servidor P1 recebe mensagens de diferentes clientes P2, P3, etc. e responde, individualmente, a cada um deles.

Para construir este modelo fazemos o seguinte plano baseado nos tipos:

- P2, P3, ... enviam a P1 mensagens do tipo 1;

no conteúdo da mensagem indicam o emissor;

esperam como resposta mensagens dos tipos 2, 3, ...;

- P1 espera mensagens do tipo 1;

responde com mensagens do tipo 2, 3, ... conforme o emissor indicado no conteúdo.

O programa seguinte ilustra o servidor P1:

```
#include "definicoes.h"
#include <string.h>

typedef struct {
    long tipo;
    struct {
        int emissor;
        char texto[250];
    } msg;
} MsgStruct;

main() {
    int status;
    MsgStruct m;
    int msg_id;

    msg_id = msgget ( 1000, 0666 | IPC_CREAT );
    exit_on_error (msg_id, "Criação/Ligação");

    while ( 1 ) {
        printf ("Esperar...\n");

        status = msgrcv( msg_id, &m, sizeof(m.msg), 1, 0);
        exit_on_error (status, "Recepção");

        m.tipo = m.msg.emissor;
        sprintf ( m.msg.texto, "Hello P%d\n", m.msg.emissor);

        printf ("Resposta: %s\n", m.msg.texto);

        status = msgsnd( msg_id, &m, sizeof(m.msg), 0);
        exit_on_error (status, "Envio");
    }
}
```

O programa seguinte apresenta o servidor P2. Os outros seriam semelhantes usando o tipo de mensagem 3, 4, ...

```
#include "definicoes.h"
#include <string.h>

main() {
    int status;
    MsgStruct m;
    int msg_id;

    msg_id = msgget ( 1000, 0666 | IPC_CREAT );
    exit_on_error (msg_id, "Criação/Ligação");

    m.tipo = 1;
    m.msg.emissor = 2;

    status = msgsnd( msg_id, &m, sizeof(m.msg), 0);
    exit_on_error (status, "Envio");

    status = msgrcv( msg_id, &m, sizeof(m.msg), 2, 0);
```



```

        exit_on_error (status, "Recepção");

        printf ("Resposta: %s\n", m.msg.texto);

    }

```

O cliente pode-se generalizar para usar como tipo de mensagem o próprio pid. Este é uma forma expedita de atribuir a cada processo um número genérico e seguramente não repetido.

```

#include "definicoes.h"
#include <string.h>

main() {
    int status;
    MsgStruct m;

    msg_id = msgget ( 1000, 0666 | IPC_CREAT );
    exit_on_error (msg_id, "Criação/Ligação");

    m.tipo = 1;
    m.msg.emissor = getpid();

    status = msgsnd( msg_id, &m, sizeof(m.msg), 0);
    exit_on_error (status, "Envio");

    status = msgrcv( msg_id, &m, sizeof(m.msg), getpid(), 0);
    exit_on_error (status, "Recepção");

    printf ("Resposta: %s\n", m.msg.texto);
}

```

6.2 Semáforos

Os semáforos são um dos mecanismos de comunicação entre processos, que permitem a sincronização de processos. Estes são habitualmente usados para garantir a exclusão mútua de processos em recursos partilhados.

As operações atómicas básicas com semáforos são:

- UP – Incrementa o valor do semáforo em uma unidade
- DOWN – Tenta decrementar o valor do semáforo uma unidade, causando um bloqueio do processo que o invoca se o resultado fosse dar negativo. O processo permanece bloqueado até que o valor do semáforo lhe permita efectuar o decremento. Ou seja, o valor de um semáforo é sempre positivo.
- ZERO - Um processo que efectua esta operação permanece bloqueado até que o valor do semáforo seja igual a zero

6.2.1 Criação e inicialização

Considere que o ficheiro `definicoes.h` contém

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#define exit_on_error(s,m) if ( s < 0 ) { perror(m); exit(1); }
```

O programa seguinte cria e inicializa um semáforo

```
#include "definicoes.h"

main() {
    int id = semget ( 1000, 1, IPC_CREAT | 0666 );
    exit_on_error (id, "Criação/Ligação");

    printf ("Criado semáforo com id %d\n", id);

    int status = semctl ( id, 0, SETVAL, 0);
    exit_on_error (status, "Inicialização");
}
```

Há algumas semelhanças de nomenclatura com as filas de mensagens. A criação do semáforo é feita pela função `semget`, que tem como primeiro argumento uma chave. Esta chave deve ser única; tal como nos exemplos anteriores usamos 1000 como chave.

O semáforo é também um recurso permanente gerido pelo sistema operativo. Para ver os semáforos criados podemos fazer o comando:

```
ipcs -s
```

A função `semctl` é usada no exemplo, para inicializar o semáforo. Neste caso o semáforo é inicializado com o valor 0 (o terceiro argumento da função).

6.2.2 Operações (up e down)

Os semáforos implementam o conceito apresentado nas aulas teóricas: um semáforo é uma variável, gerida pelo do sistema operativo, com valor ≥ 0 , sobre a qual são se podem realizar duas operações:

- UP(x) : aumenta 1 ao valor de x;
- DOWN(x): diminui 1 ao valor de x; bloqueia enquanto não puder fazê-lo.

O sublinhado é o ponto importante: um semáforo tem um valor ≥ 0 ; se o valor for 0 e um processo tentar fazer um down não pode; nessas condições fica bloqueado até poder fazê-lo (ou seja, até que outro processo faça um up!).

Os exemplos seguintes implementam funções UP e DOWN. O programa P1 faz um *down*. A operação é realizada pela função `semop`, que recebe como argumento a estrutura DOWN. O programa P2 faz um *up*. A operação é realizada pela função `semop`, que recebe como argumento a estrutura UP.

P1)

```
#include "definicoes.h"

struct sembuf UP = { 0, 1, 0 } ;
struct sembuf DOWN = { 0, -1, 0 };
main() {
    int sem_id;
```

```

    int status;
    sem_id = semget ( 1000, 1, 0600 );
    exit_on_error (sem_id, "Criação/Ligação");

    printf ("Esperar...\n");
    // DOWN() - para sair daqui alguém tem que fazer UP()
    status = semop ( sem_id, &DOWN, 1 );
    exit_on_error (status, "DOWN");

    printf ("Ok\n");
}

```

P2)

```

#include "definicoes.h"

struct sembuf UP = { 0, 1, 0 } ;
struct sembuf DOWN = { 0, -1, 0 } ;
main() {
    int sem_id;
    int status;

    //ligar ao semaforo
    sem_id = semget ( 1000, 1, 0600 );
    exit_on_error (sem_id, "Criação/Ligação");

    // UP()
    status = semop ( sem_id, &UP, 1 );
    exit_on_error (status, "UP");

    printf ("Ok\n");
}

```

A estrutura sembuf inclui os parâmetros necessários à execução da operação. Se fizer “man semop”, poderá verificar que a estrutura sembuf inclui três elementos: o primeiro corresponde ao número do semáforo; o segundo corresponde ao valor que se pretende adicionar ao semáforo; e o terceiro são apenas opções extra, raramente usadas e portanto normalmente matém o valor 0.

```

struct sembuf {
    ushort sem_num;        /* semaphore # */
    short  sem_op;         /* semaphore operation */
    short  sem_flg;        /* operation flags */
};

```

Experiência: execute o programa de inicialização do ponto 1.1, deixando portanto o semáforo com o valor 0. Se, de seguida, executar P1, este ficará bloqueado no down. Executando (noutra janela) o programa P2 este porá o semáforo a 1, permitindo assim a P1 desbloquear e prosseguir. O valor final do semáforo será 0.

Experiência: execute o programa de inicialização do ponto 1.1, deixando portanto o semáforo com o valor 0. De seguida executa 3 vezes o programa P2, deixando portanto o semáforo com valor 3. De seguida poderá executar P1 também 3 vezes; à 4ª vez P1 bloqueia.

6.2.3 Exemplo: cliente / servidor

No exemplo seguinte temos vários processos clientes sobem o valor do semáforo e um processo servidor que baixa o valor do semáforo. Verifique a execução de vários clientes e um servidor.

s1.c

```

#include "definicoes.h"

struct sembuf UP = { 0, 1, 0 } ;
struct sembuf DOWN = { 0, -1, 0 } ;

main() {
    int sem_id;
    int status, i = 1;
    sem_id = semget ( 1010, 1, 0666 | IPC_CREAT );
    exit_on_error (sem_id, "Criação/Ligação");

    while ( 1 ) {
        status = semop ( sem_id, &DOWN, 1 );
        exit_on_error (status, "DOWN");
        printf ("passagem %d\n", i++);
    }
}

```

c1.c

```

#include "definicoes.h"

struct sembuf UP = { 0, 1, 0 } ;
struct sembuf DOWN = { 0, -1, 0 } ;

main() {
    int sem_id;
    int status;
    sem_id = semget ( 1010, 1, 0666 | IPC_CREAT );
    exit_on_error (sem_id, "Problema de Criação/Ligação");

    status = semop ( sem_id, &UP, 1 );
    exit_on_error (status, "Problema com o UP");
}

```

6.2.4 Exemplo: zona de exclusão

Uma zona de exclusão é uma secção do programa com acessos condicionados a um processo de cada vez. Os exemplos seguintes simulam uma zona de exclusão. O programa de inicialização P1 cria um semáforo e inicializa-o com 1.

```

#include "definicoes.h"

main() {

    int id = semget ( 1000, 1, IPC_CREAT | 0666 );
    exit_on_error (id, "Criação/Ligação");

    printf ("Criado semáforo com id %d\n", id);

    int status = semctl ( id, 0, SETVAL, 1);
    exit_on_error (status, "Inicialização");
}

```

O programa P2 simula uma zona de exclusão.

```

#include "definicoes.h"

struct sembuf UP = { 0, 1, 0 } ;
struct sembuf DOWN = { 0, -1, 0 } ;

main() {
    int sem_id;
    int status;
    sem_id = semget ( 1000, 1, 0600 | IPC_CREAT );
    exit_on_error (sem_id, "Criação/Ligação");

    printf ("Entrada..\n");
    status = semop ( sem_id, &DOWN, 1 );
    exit_on_error (status, "DOWN");

    printf ("ENTER para continuar...");
    getchar();

    status = semop ( sem_id, &UP, 1 );
    exit_on_error (status, "DOWN");
    printf ("Saída\n");
}

```

Experiência: execute o programa de inicialização P1. Execute P2 em várias janelas; verifique apenas um dos processos entra, de cada vez, na zona de exclusão (simulada pelo "ENTER para continuar..").

6.2.5 Exemplo: "brancas e pretas"

O exemplo seguinte simula uma situação em dois processos executam à vez (como num jogo entre brancas e pretas).

O programa usa dois semáforos. O programa seguinte cria e inicializa os dois semáforos.

```

#include "definicoes.h"

main() {

    int id_brancas = semget ( 1001, 1, IPC_CREAT | 0666 );
    int id_pretas = semget ( 1002, 1, IPC_CREAT | 0666 );

    semctl ( id_brancas, 0, SETVAL, 1);
    semctl ( id_pretas, 0, SETVAL, 0);
}

```

O programa seguinte implementa o ciclo de jogo das brancas: esperam pela sua vez, jogam (simulado pelo "ENTER para continuar") e cedem a vez.

```

#include "definicoes.h"

void down ( id ) {
    struct sembuf DOWN = { 0, -1, 0 } ;
    int status = semop ( id, &DOWN, 1 );
    exit_on_error (status, "DOWN");
}

void up ( id ) {
    struct sembuf UP = { 0, 1, 0 } ;
    int status = semop ( id, &UP, 1 );
}

```

```

        exit_on_error (status, "UP");
    }

    main() {

        int id_branças = semget ( 1001, 1, IPC_CREAT | 0666 );
        int id_pretas = semget ( 1002, 1, IPC_CREAT | 0666 );

        while ( 1 ) {
            down ( id_branças );
            printf ("Jogam as brancas (ENTER para continuar)...");
            getchar();
            up ( id_pretas);
        }
    }
}

```

O programa seguinte implementa o ciclo de jogo das pretas.

```

#include "definicoes.h"

void down ( id ) {
    struct sembuf DOWN = { 0, -1, 0 } ;
    int status = semop ( id, &DOWN, 1 );
    exit_on_error (status, "DOWN");
}

void up ( id ) {
    struct sembuf UP = { 0, 1, 0 } ;
    int status = semop ( id, &UP, 1 );
    exit_on_error (status, "UP");
}

main() {
    int id_branças = semget ( 1001, 1, IPC_CREAT | 0666 );
    int id_pretas = semget ( 1002, 1, IPC_CREAT | 0666 );

    while ( 1 ) {
        down ( id_pretas );
        printf ("Jogam as pretas (ENTER para continuar)...");
        getchar();
        up ( id_branças);
    }
}

```

6.2.6 Arrays de semáforos

O programa seguinte usa a função semget para criar um "array de semáforos" (neste caso um array com dois semáforos).

```

#include "definicoes.h"

main() {
    int id = semget ( 1007, 2, IPC_CREAT | 0666 );
    exit_on_error (id, "Criação/Ligação");
}

```

```

    printf ("Criado semáforo com id %d\n", id);

    semctl ( id, 0, SETVAL, 1);
    semctl ( id, 1, SETVAL, 0);
}

```

A diferença para os exemplos anteriores é o segundo argumento do `semget` que agora pede que o recurso criado seja um array com 2 semáforos (em todos os exemplos anteriores tínhamos usado 1). Os dois semáforos criados são identificados como os elementos de um array; ou seja, o primeiro é identificado como semáforo nº 0 e o segundo é identificado como semáforo nº 1. A utilização do `semctl` já reflecte esta nomenclatura. O primeiro `setctl` inicializa o primeiro semáforo (o nº 0) com o valor 1. O segundo `semctl` inicializa o segundo semáforo (o nº 1) com o valor 0.

Os programas seguintes implementam uma nova versão do modelo "brancas e pretas" usando, desta vez, um array com dois semáforos (em vez de dois semáforos com chaves distintas, como na secção anterior).

O exemplo seguinte implementa o ciclo das brancas.

```

#include "definicoes.h"

void s_operacao ( int id, int idx, int v ) {
    struct sembuf s;
    s.sem_num = idx;
    s.sem_op = v;
    s.sem_flg = 0;

    int status = semop ( id, &s, 1 );
    exit_on_error (status, "DOWN");
}

main() {
    int id = semget ( 1007, 2, IPC_CREAT | 0666 );

    while ( 1 ) {
        s_operacao ( id, 0 , -1 );
        printf ("Jogam as brancas (ENTER para continuar)...");
        getchar();
        s_operacao ( id, 1 , 1 );
    }
}

```

Desta vez é usada uma função genérica `s_operacao` que recebe o id do semáforo, o número do índice e o valor a adicionar ao semáforo: 1 para up e -1 para down.

```

// Pretas
#include "definicoes.h"

main() {
    int id = semget ( 1007, 2, IPC_CREAT | 0666 );

    while ( 1 ) {
        s_operacao ( id, 0 , -1 );
        printf ("Jogam as brancas (ENTER para continuar)...");
        getchar();
        s_operacao ( id, 1 , 1 );
    }
}

```

6.3 Memória partilhada

Dois ou mais processos podem também efectuar trocas de informação se partilharem entre si um conjunto de posições de memória. Utilizando o mecanismo IPC da memória partilhada, pode-se definir um bloco de posições consecutivas de memória partilhado por vários processos.

Um processo pode escrever dados nesse bloco e outro processo pode lê-los. A vantagem da utilização de um esquema deste tipo é a rapidez na comunicação. A desvantagem é a inexistência de sincronização no acesso à memória.

Deste modo, a parte do código onde são efectuados acessos à memória partilhada, constitui geralmente uma região crítica, sendo portanto necessário assegurar a exclusão mútua de processos, utilizando por exemplo, os semáforos.

Considere que o ficheiro `definicoes.h` contém

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#define exit_on_error(s,m) if ( s < 0 ) { perror(m); exit(1); }
#define exit_on_null(s,m) if (s==NULL) { perror(m); exit(1); }
```

6.3.1 Criação

Considere o seguinte exemplo:

```
#include "definicoes.h"
main() {
    int id = shmget ( 1000, 100, IPC_CREAT | 0666 );
    exit_on_error (id, "Criação/Ligação");

    printf ("Criada memória partilhada com id %d\n", id);

    void *p = shmat ( id, 0, 0 );
    if ( p == NULL ) {
        printf ("Erro no attach\n");
        exit(1);
    }

    char *s = (char *) p;
    strcpy ( s, "Hello\n" );
}
```

A função `shmget` permite criar um bloco de memória partilhada. Quer dizer: estamos a pedir ao sistema operativo para disponibilizar um bloco de memória e manter o seu conteúdo de forma a poder ser usado por vários programas.

A exemplo dos IPCs anteriores podemos ver a lista de "memórias partilhadas" com o comando `ipcs`, desta vez com a opção `-m`. Também a exemplo dos IPCs anteriores, podemos destruir a memória partilhada com o comando `ipcrm -m`.

O objectivo final é poder usar a memória partilhada da mesma maneira que se usa a memória "normal" do programa, ou seja, sob a forma de variáveis. Para isso são precisas duas coisas:

- 1º) arranjar uma maneira de chegar à memória partilhada;
- 2º) definir o tipo de dados que se vão colocar nessa memória (`int`, `char` ... ???).

O primeiro passo designa-se attach e é realizado pela chamada à função `shmat`, que devolve um "ponteiro" para o bloco de memória partilhada. No exemplo o ponteiro devolvido por `shmat` é guardado na variável `p`.

A função `shmat` devolve um ponteiro que, tal como `p`, é do tipo `void *`. Este ponteiro indica a localização da memória partilhada mas não indica o tipo de dados que ela contém, e por isso não pode ser directamente usado para lhe aceder.

No passo seguinte, converte-se o ponteiro `void *` num ponteiro para um tipo de dados concreto, no caso `char *`. Isto quer dizer que, a partir daqui, podemos usar a shared memory através do ponteiro `p_char` tratando-a como contendo chars, ou seja como um array de chars.

No seguimento do exemplo, copia-se "Hello\n" para esse array, inicializando desta forma as primeiras 7 posições.

6.3.2 Outros exemplos de manipulação

Os exemplos seguintes manipulam a mesma memória de 100 bytes do exemplo anterior.

Exemplo 1

O programa seguinte escreve o conteúdo da memória presumindo que contém uma string.

```
#include "definicoes.h"
main() {
    int id = shmget ( 1000, 100, IPC_CREAT | 0666 );
    exit_on_error (id, "Criação/Ligação");

    printf ("Criada memória partilhada com id %d\n", id);

    void *p = shmat ( id, 0, 0 );
    exit_on_null (p, "Attach");

    char *s = (char *) p;
    printf ( "M: %s\n", s );
}
```

Executando este programa a seguir ao da secção anterior o output será:

```
M : Hello
```

Exemplo 2

O exemplo seguinte faz "dump" do conteúdo de cada uma das 100 posições de memória, consideradas ainda como caracteres, escrevendo o código ASCII de cada um desses caracteres.

dump.c

```
#include "definicoes.h"
main() {
    int id = shmget ( 1000, 100, IPC_CREAT | 0666 );
    exit_on_error (id, "Criação/Ligação");

    printf ("Criada memória partilhada com id %d\n", id);

    char *s = shmat ( id, 0, 0 );
    exit_on_null (s, "Attach");

    int i;
    for ( i = 0; i < 100; i++ ) {
```

```

        int c = s[i];
        printf ("%5d", c );
        if ( (i+1) % 10 == 0 ) printf ("\n");
    }
    printf ("\n");
}

```

Executando este programa a seguir ao da secção anterior o output será

```

72  101  108  108  111  10   0   0   0   0
0   0   0   0   0   0   0   0   0   0
0   0   0   0   0   0   0   0   0   0
0   0   0   0   0   0   0   0   0   0
0   0   0   0   0   0   0   0   0   0
0   0   0   0   0   0   0   0   0   0
0   0   0   0   0   0   0   0   0   0
0   0   0   0   0   0   0   0   0   0
0   0   0   0   0   0   0   0   0   0
0   0   0   0   0   0   0   0   0   0

```

onde se podem ver os caracteres resultantes da inicialização feita pelo programa na secção anterior: os caracteres Hello (código ASCII 72, ...) o '\n', código ASCII 10 e o terminador, código ASCII 0.

Experiência: execute o seguinte programa para inicializar a shared memory

```

#include "definicoes.h"
main() {
    int id = shmget ( 1000, 100, IPC_CREAT | 0666 );
    exit_on_error (id, "Criação/Ligação");

    char *s = shmat ( id, 0, 0 );
    exit_on_null (s, "Attach");

    int i;
    for ( i = 0; i < 50; i++ ) {
        s[i] = 'x';
    }
    strcpy ( s, "hello, world\n");;
}

```

Execute o programa anterior, "dump", para mostrar o conteúdo da shared memory e interprete o resultado.

6.3.3 Estruturar o conteúdo da memória (Caso 2)

Neste exemplo é criada uma memória partilhada para guardar 25 números inteiros.

```

#include "definicoes.h"
main() {
    int id = shmget ( 1001, 25*sizeof(int), IPC_CREAT | 0666 );
    exit_on_error (id, "Criação/Ligação");

    int *p = shmat ( id, 0, 0 );
    exit_on_null (p, "Attach");

    int i;
    for ( i = 0; i < 10; i++ ) {
        p[i] = i + 1;
    }
}

```

```
    }
}
```

A dimensão da memória pedida é agora `25*sizeof(int)`, ou seja, 25 vezes o tamanho (o número de bytes) ocupado por um inteiro. Admitindo que cada inteiro ocupa 4 bytes, estamos a falar, na mesma, de 100 bytes.

O resultado do attach é agora convertido para um tipo `int *`, permitindo assim usar a memória partilhada como um array de inteiros.

O exemplo seguinte mostra o conteúdo da memória partilhada, assumindo agora que a memória contém números inteiros.

```
#include "definicoes.h"
main() {
    int id = shmget ( 1001, 25*sizeof(int), IPC_CREAT | 0666 );
    exit_on_error (id, "Criação/Ligação");

    int *p = shmat ( id, 0, 0 );
    exit_on_null (p, "Attach");

    int i;
    for ( i = 0; i < 25; i++ ) {
        printf ("%5d", p[i] );
        if ( (i+1)%10 == 0 ) printf ("\n");
    }
    printf ("\n");
}
```

6.3.4 Estruturar o conteúdo da memória (Caso 2)

O exemplo seguinte cria uma memória partilhada para guardar dados do tipo definido pela estrutura `Aluno`.

```
#include "definicoes.h"

typedef struct {
    int num;
    char nome[100];
    int nota;
} Aluno;

int main() {
    int id = shmget ( 1002, 25*sizeof(Aluno), IPC_CREAT | 0666 );
    exit_on_error (id, "Criação/Ligação");

    Aluno *p = shmat ( id, 0, 0 );
    exit_on_null (p, "Attach");

    int i;
    for ( i = 0; i < 25; i++ ) {
        p[i].num = 0;
    }
}
```

O programa cria uma memória partilhada com o tamanho necessário para guardar 25 estruturas `Aluno`. O resultado do attach é convertido para o ponteiro `p` do tipo `Aluno *`. Através deste ponteiro podemos usar a memória como um array de 25 elementos do tipo `Aluno`.

O exemplo seguinte lê dados e preenche o próximo Aluno livre na memória partilhada com esses dados.

```
#include "definicoes.h"

void limpar_enter() { while ( getc(stdin) != '\n' ); }

int main() {
    int id = shmget ( 1002, 25*sizeof(Aluno), IPC_CREAT | 0666 );
    exit_on_error (id, "Criação/Ligação");

    Aluno *p = shmat ( id, 0, 0 );
    exit_on_null (p, "Attach");

    Aluno a;

    printf ("Numero: ");
    scanf ("%d", &(a.num) );
    limpar_enter();

    printf ("Nome: ");
    fgets ( a.nome, 100, stdin );
    a.nome[ strlen(a.nome) -1 ] = 0;

    printf ("Nota: ");
    scanf ("%d", &(a.nota) );
    limpar_enter();

    int i;
    for ( i = 0; i < 25; i++ ) {
        if ( p[i].num == 0 ) {
            p[i] = a;
            break;
        }
    }
}
```

O exemplo seguinte lista no ecrã os alunos registados na memória partilhada.

```
#include "definicoes.h"

int main() {
    int id = shmget ( 1002, 25*sizeof(Aluno), IPC_CREAT | 0666 );
    exit_on_error (id, "Criação/Ligação");

    Aluno *p = shmat ( id, 0, 0 );
    exit_on_null (p, "Attach");

    int i;
    for ( i = 0; i < 25; i++ ) {
        if ( p[i].num == 0 ) continue;
        printf ("%7d %-40s %3d\n", p[i].num, p[i].nome, p[i].nota );
    }
}
```

6.4 Complementos

6.4.1 IPC_NOWAIT

Já antes vimos que um processo bloqueia quando tenta baixar um semáforo que está com o valor zero. Isso também acontece quando se tenta, por exemplo, ler uma mensagem de um determinado tipo e a fila de mensagens não tem mensagens desse tipo.

Por vezes, pode ser útil não bloquear o processo quando tenta fazer uma destas operações. Por exemplo, suponha que um processo quer simplesmente verificar se existem mensagens para si. Se não existirem, pretende-se que o processo continue a sua execução. Pode-se utilizar para isso o valor `IPC_NOWAIT`.

No caso das filas de mensagem, esse valor é colocado no último argumento da função `msgrcv`. Neste caso o `msgrcv` não bloqueia. Vai buscar uma mensagem e se houver recolhe, senão continua a sua execução.

```
// bloqueia se não houver mensagens
status = msgrcv( msg_id, &msg, sizeof(msg.texto), 1, 0);
// continua a execução, mesmo que não hajam mensagens
status = msgrcv( msg_id, &msg, sizeof(msg.texto), 1, IPC_NOWAIT);
```

No caso dos semáforos

```
// bloqueia, se não conseguir baixar o semáforo
status = semop ( sem_id, &DOWN, 0 );
// Não bloqueia
status = semop ( sem_id, &DOWN, IPC_NOWAIT );
```

6.4.2 ftok()

Nos exemplos anteriores, foi sempre usada uma chave fixa nas funções `msgget`, `semget` e `shmget`. No entanto, a função `ftok()` permite criar uma chave única muito mais interessante, que depende da localização de um dado ficheiro ou directório, bem como de um ID definido pelo utilizador.

O programa seguinte devolve uma chave diferente, cada vez que é executado em directórios diferentes, mas qualquer programa que seja executado no mesmo directório produzirá a mesma chave. Claro que em vez do directório actual "." se poderia usar por exemplo um caminho absoluto, por exemplo: `/home`. Nesse caso, a chave seria sempre fixa.

```
#include <sys/ipc.h>
#include <stdio.h>

main() {
    int chave;
    chave = ftok(".", 'a');
    printf("A chave em uso é: %d\n", chave);
}
```

Além disso, o ID definido pelo utilizador, neste caso a letra 'a', permite que possamos ter diferentes chaves para diferentes programas que se encontrem no mesmo directório, por exemplo: programas do projecto A e programas do projecto B.

O exemplo seguinte mostra um exemplo de utilização desta função, ilustrando o exemplo clássico do produtor/consumidor, em que vários processos consomem recursos e outros processos produzem esses recursos. Verifique que ambos os processos usam a mesma chave.

prod.c

```
#include <stdio.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#include <stdlib.h>
#define exit_on_error(s,m) if ( s < 0 ) { perror(m); exit(1); }

struct sembuf UP = { 0, 1, 0 } ;
struct sembuf DOWN = { 0, -1, 0 } ;

main() {
    int chave = ftok(".", 'a');
    int sem_id;
    int status, i = 1;
    sem_id = semget ( chave, 1, 0666 | IPC_CREAT );
    exit_on_error (sem_id, "Criação/Ligação");

    while (1) {
        sleep( getpid() % 5 ); // Simula o tempo de produção
        printf ("Acabei de produzir um recurso\n");
        status = semop ( sem_id, &UP, 1 );
        exit_on_error (status, "Problema com o UP");
    }
}
```

cons.c

```
#include <stdio.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#include <stdlib.h>
#define exit_on_error(s,m) if ( s < 0 ) { perror(m); exit(1); }

struct sembuf UP = { 0, 1, 0 } ;
struct sembuf DOWN = { 0, -1, 0 } ;

main() {
    int chave = ftok(".", 'a');
    int sem_id;
    int status, i = 1;
    sem_id = semget ( chave, 1, 0666 | IPC_CREAT );
    exit_on_error (sem_id, "Criação/Ligação");

    while (1) {
        printf ("Estou à espera de um recurso\n");
        status = semop ( sem_id, &DOWN, 1 );
        printf ("Consumi um recurso\n");
        exit_on_error (status, "Problema com o UP");
        sleep( getpid() % 5 );
    }
}
```

7

Input/Output

7.1 Funções de biblioteca

Esta secção faz uma revisão das funções de biblioteca mais comuns.

7.1.1 Leitura e escrita em ficheiros

Apresentam-se algumas das funções relacionadas com a leitura e escrita em ficheiros, que já foram abordadas em aulas anteriores.

- `fopen`, `fclose`
- `scanf`, `fscanf`, `gets`, `fgets`, `getc`, `fgetc`
- `printf`, `fprintf`, `puts`

7.1.2 `stdin`, `stdout` e `stderr`

Há partida um programa, quando inicia, já tem pelo menos três "ficheiros" abertos.

Um deles é designado por `stdin` (standard input) e representa, normalmente, o teclado. Nestas circunstâncias ler do `stdin` é ler do teclado. É isso que fazemos normalmente com

```
scanf ( "%d", .... )
```

que é o mesmo que

```
fscanf ( stdin, "%d", ... )
```

Outro é designado por `stdout` (standard output) e representa, normalmente, o ecrã. Nestas circunstâncias ao escrever para o `stdout` estamos a escrever para o ecrã. É isso que fazemos com

```
printf ( "...", ... )
```

que é o mesmo que

```
fprintf ( stdout, "...", ... );
```

O terceiro, designado por `stderr`, normalmente também corresponde ao ecrã, mas é um ficheiro especial para onde se enviam, normalmente, mensagens de erro e outras informações importantes para o controlo do programa.

Mas o `stdout` e o `stdin` não são necessariamente o ecrã e o teclado. São, mais geralmente, o sítio normal ("standard") de leitura e escrita do programa que podem ser o teclado e ecrã mas podem, também, ser redireccionados para outro lado, designadamente um ficheiro. É isto que acontece quando na shell damos um comando que redirecciona o input ou output. Por exemplo o comando

```
ls > fich
```

executa o comando `ls`, mas o seu output desta vez, em vez de ir para o ecrã vai para o ficheiro `fich`. Isto significa que agora o programa executa tendo o standard output direccionado para o ficheiro `fich`; desta forma quando fizer `printf (...)` ou `fprintf (stdout,...)` está, de facto, a escrever nesse ficheiro.

Mais adiante veremos como a shell pode fazer para implementar este mecanismo.

7.2 Chamadas do sistema para input/output (i/o)

Nesta secção vamos ver as chamadas ao sistema operativo que implementam as operações de input e output. As funções de biblioteca que vimos na secção anterior assentam sobre estas funções.

7.2.1 Leitura e Escrita

O programa seguinte exemplifica a leitura escrita usando funções `read` e `write`:

```
#include <stdio.h>
#include <string.h>
main() {
    char s[20];
    write ( 1, "Nome: ", sizeof("Nome: ") );
    bzero( s, sizeof(20) );
    read ( 0, s, sizeof(s) );
    char r[40];
    sprintf ( r, "Hello, %s\n", s );
    write ( 1, r, strlen(r) );
}
```

O primeiro argumento do `read` e do `write` representa o descritor do dispositivo de leitura ou escrita. No caso destas funções, este descritor é um número inteiro; os números 0, 1 e 2 representam os locais standard que na secção anterior, no âmbito das funções de biblioteca, designámos por `stdin`, `stdout` e `stderr`.

7.2.2 Open

O exemplo seguinte usa a função `open` para abrir um novo ficheiro, neste caso para escrita. Normalmente os descritores seguem uma numeração sequencial; os números 0, 1 e 2 estão associados aos canais `stdin`, `stdout` e `stderr`, normalmente direccionados para o teclado e ecrã. Assim, o novo ficheiro ganhará normalmente o número 3 (a seguir aos números 0, 1 e 2, que são abertos automaticamente).

```
// Exemplo 2.2.1
#include <stdio.h>
#include <string.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

typedef struct {
    int i;
    char s[20];
}
```



```

    int j;
} StructTeste;

main() {
    StructTeste t;
    t.i = 0x40414243;
    strcpy ( t.s, "Hello");
    t.j = 100;

    int n= open("fich.dat", O_CREAT|O_RDWR, S_IREAD| S_IWRITE );
    printf ( "descriptor %d\n", n );
    write ( n, &t, sizeof(t) );
    close ( n );
}

```

O exemplo seguinte lê o conteúdo do mesmo ficheiro e escreve no ecrã:

```

// Exemplo 2.2.2
#include <stdio.h>
#include <string.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

typedef struct {
    int i;
    char s[20];
    int j;
} StructTeste;

main() {

    StructTeste t;
    bzero ( &t, sizeof(t) );

    int n = open ( "fich.dat", O_RDONLY );
    read ( n, &t, sizeof(t) );
    close ( n );

    printf ( "%d %s %d\n", t.i, t.s, t.j );

}

```

Exemplos de leitura e escrita, no caso de uma pauta de alunos.

```

// escreve_pauta2.c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

typedef struct {
    int n;
    char nome[120];
} Aluno;

```

```

main() {
    char buff[1024];
    Aluno a;
    int n= open("pauta2.dat", O_CREAT|O_RDWR, S_IREAD| S_IWRITE );
    printf ( "descriptor: %d\n", n );

    bzero ( &a, sizeof(a) );

    write ( 1, "Numero: " , strlen("Numero: ") );
    read (0, buff, sizeof(buff) );
    a.n = atoi(buff);

    write ( 1, "Nome: " , strlen("Nome: ") );
    read (0, buff, sizeof(buff) );
    strncpy ( a.nome, buff, sizeof(a.nome) );

    lseek ( n, 0, SEEK_END );
    write ( n, &a, sizeof(a) );
    close ( n );
}

// mostra_pauta2.c

#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

typedef struct {
    int n;
    char nome[120];
} Aluno;

main() {
    Aluno a;
    bzero ( &a, sizeof(a) );
    int n = open ( "pauta2.dat", O_RDONLY );
    while ( ( read ( n, &a, sizeof(a) ) ) > 0 ) {
        fprintf ( stdout, "%d\t%s\n", a.n, a.nome );
    }
    close(n);
}

```

7.2.3 Posicionamento

7.2.4 Manipulação de descritores

Consideremos o seguinte comando que utiliza o “mostra”, descrito na secção 1.

```
./mostra teste.dat > fich.txt
```

A implementação deste mecanismo poderia ser a seguinte:

```
//Exemplo 2.2.4
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

main(int argc, char *argv[]) {
    int n = open ( "fich.txt", O_CREAT|O_RDWR, S_IREAD| S_IWRITE );
    int p = fork();
    if ( p == 0 ) {
        dup2 ( n, 1 );
        execl ( "./mostra", "mostra", argv[1], NULL );
        exit(0);
    }
    wait ( NULL );
}
```

A chamada à função `dup2()` faz com que o descritor 1 (isto é, o `stdout`) seja alterado para ficar igual ao `n` (ou seja o do ficheiro "fich.txt", aberto pelo `open`). O `execl` vai manter este novo ficheiro como descritor 1, de forma que o `mostra` vai arrancar tendo como descritor 1 (ou seja, como standard output) o ficheiro "fich.txt".

7.3 IPC, pipes

7.3.1 Half duplex

O seguinte exemplo ilustra o conceito de pipe:

```
//Exemplo 3.1.1
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>
main() {

    int fd[2];
    pipe ( fd );

    int n = fork();
    if ( n == 0 ) {
        close( fd[0] );
        int i;
        for ( i = 0; i < 5; i++ ) {
            write ( fd[1], "Hello\n", sizeof("Hello\n") );
        }
        close( fd[1] );
        printf ("Fim do processo filho\n");
        exit(0);
    }

    close ( fd[1] );
    char s[100];
    int i = 0;
    while ( ( n = read( fd[0], s, sizeof(s))) > 0 ) {
```

```

        write ( 0, s, n );
    }
    printf ("Fim do processo pai\n");
}

```

No exemplo anterior, depois de criar o pipe, usámos as funções `read` e `write` para ler e escrever. Podemos, em alternativa, reabrir os mesmos descritores e usar as funções mais comuns de leitura e escrita.

```

//Exemplo 3.1.2
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>
main() {

    int fd[2];
    pipe ( fd );

    int n = fork();
    if ( n == 0 ) {
        close( fd[0] );
        FILE *f = fdopen ( fd[1] , "w" );
        int i;
        for ( i = 0; i < 5; i++ ) {
            fprintf ( f, "Hello\n" );
        }
        fclose( f );
        printf ("Fim do processo filho\n");
        exit(0);
    }

    close ( fd[1] );
    char s[100];
    int i = 0;
    FILE *f = fdopen ( fd[0], "r" );
    while ( fgetc ( s, 100, f ) > 0 ) {
        printf ( "%d: %s", i++, s );
    }
    printf ("Fim do processo pai\n");
}

```

O pipe é um mecanismo de comunicação entre processos aparentemente com uma utilidade muito limitada uma vez que permite comunicação entre processos pai e filho (ou, pelo menos, a processos com um mesmo antepassado que tenha criado o pipe) e numa só direcção.

Mas este é um mecanismo muito usado nos sistemas Unix, em particular para implementação do conceito de pipe, que a shell proporciona. Consideremos o seguinte comando que é um exemplo típico

```
cat /etc/passwd | grep root
```

(outro exemplo: `./mostra teste.txt | /bin/more`)

A execução deste comando pela shell corresponderia um arranjo do género:

```

//Exemplo 3.1.3
#include <stdio.h>

```

```

#include <string.h>
#include <stdlib.h>
#include <unistd.h>

main() {

    int fd[2];
    pipe ( fd );

    int n = fork();
    if ( n == 0 ) {
        close ( fd[0] );
        dup2 ( fd[1], 1 );
        execl( "/bin/cat", "/bin/cat", "/etc/passwd", NULL );
        exit(1);
    }
    n = fork();
    if ( n == 0 ) {
        close ( fd[1] );
        dup2 ( fd[0], 0 );
        execl( "/usr/bin/grep", "/usr/bin/grep", "root", NULL );
        exit(1);
    }
    wait ( NULL );
}

```

7.3.2 FIFO (named pipes)

Os seguintes exemplos ilustram o funcionamento de um FIFO.

```

//Exemplo 3.2.1-A
#include <stdio.h>
main() {
    mkfifo ( "./fifo-1", 0666 );
    FILE *f = fopen ( "./fifo-1", "r" );

    char s[100];
    while ( fgets ( s, 100, f ) ) {
        printf ("%s", s );
    }
}

//Exemplo 3.2.1-B
#include <stdio.h>
main() {
    FILE *f = fopen ( "./fifo-1", "w" );

    char s[100];
    printf ("> ");
    while ( fgets( s, 100, stdin) != NULL ) {
        fprintf ( f, "%s", s ); fflush ( f );
        if ( strcmp ( s, "exit\n" ) == 0 ) break;
        printf ("> ");
    }
}

```

Os exemplos seguintes ilustram uma arquitectura cliente servidor baseada em pipes.

```

//Exemplo pipe-serv.c
#include <stdio.h>
#include <string.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

typedef struct { int pid; char texto[100]; } Msg;

main() {
    mkfifo ( "./fifo-1", 0666 );
    int p = open ( "./fifo-1", O_RDWR );

    Msg m, r;
    while ( 1 ) {
        printf ( "Aguardar...\n" );
        read ( p, &m, sizeof(m) );

        printf ( "responder a %d\n", m.pid );

        char pnome[100];
        sprintf ( pnome, "fifo-%d", m.pid );
        int pr = open ( pnome, O_RDWR );

        strcpy ( r.texto, "Hello, " );
        strcat ( r.texto, m.texto );
        write ( pr, &r, sizeof(r) );
    }
}

```

```

//Exemplo pipe-client.c
#include <stdio.h>
#include <string.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

typedef struct { int pid; char texto[100]; } Msg;

main() {
    int ps = open ( "./fifo-1", O_RDWR );
    char pnome[100];
    sprintf ( pnome, "fifo-%d", getpid() );
    printf ( "pnome=%s\n", pnome );

    mkfifo ( pnome, 0666 );
    int pc = open ( pnome, O_RDWR );

    Msg m, r;
    printf ( "Nome: " );
    fgets ( r.texto, 100, stdin );

    r.pid = getpid();
    write ( ps, &r, sizeof(r) );

    read ( pc, &m, sizeof(m) );
    printf ( "Resposta: %s\n", m.texto );
}

```

```

        unlink ( pnome );
    }

```

7.4 Sistema de ficheiros

7.5 Terminal

```

// t1.c
#include <stdio.h>
#include <string.h>
#include <termios.h>
#include <unistd.h>
main() {
    char u[80], p[80];

    fprintf ( stdout, "User: ");
    fgets( u, 80, stdin );

    fprintf ( stdout, "Password: ");

    struct termios t_reset, t;
    tcgetattr( 0, &t_reset);
    t = t_reset;
    t.c_lflag = t.c_lflag & ~ECHO;
    t.c_lflag = t.c_lflag | ECHONL;
    tcsetattr( 0, TCSANOW, &t);

    fgets ( p, 80, stdin );
    tcsetattr( 0, TCSANOW, &t_reset);
    p[strlen(p)-1] = 0;

    #ifdef DEBUG
        printf(stdout, "P:<%s>\n", p );
    #endif
}

```

Se pretender visualizar a password que foi previamente introduzida, deve compilar o programa com a opção `-DDEBUG`. Exemplo: `gcc -o t1 t1.c -DDEBUG`

```

// t2.c
#include <stdio.h>
#include <string.h>
#include <termios.h>
#include <unistd.h>
main() {
    char u[80], p[80];

    fprintf ( stdout, "User: ");
    fgets( u, 80, stdin );

    fprintf ( stdout, "Password: ");

    struct termios t_reset, t;

```

```

tcgetattr( 0, &t_reset);
t = t_reset;
t.c_lflag = t.c_lflag & ~ECHO;
t.c_lflag = t.c_lflag | ECHONL;
t.c_lflag = t.c_lflag & ~ICANON;
tcsetattr( 0, TCSANOW, &t);

int i = 0;
int c = getchar();
while ( c != '\n' ) {
    if ( c > 27 & c < 127 ) {
        p[i++] = c;
        putchar( '*' );
    }
    c = getchar();
}
p[i] = 0;
tcsetattr( 0, TCSANOW, &t_reset);

#ifdef DEBUG
    fprintf(stdout, "\nP:<%s>\n", p );
#endif
}

```

O programa acima mostra um asterisco por cada letra introduzida, contudo não permite eliminar caracteres com backspace. O próximo programa resolve esse problema.

```

// t3.c
#include <stdio.h>
#include <string.h>
#include <termios.h>
#include <unistd.h>
main() {
    char u[80], p[80];

    fprintf ( stdout, "User: ");
    fgets( u, 80, stdin );

    fprintf ( stdout, "Password: ");

    struct termios t_reset, t;
    tcgetattr( 0, &t_reset);
    t = t_reset;
    t.c_lflag = t.c_lflag & ~ECHO;
    t.c_lflag = t.c_lflag | ECHONL;
    t.c_lflag = t.c_lflag & ~ICANON;
    tcsetattr( 0, TCSANOW, &t);

    int BS = t.c_cc[VERASE];

    int i = 0;
    int c = getchar();
    while ( c != '\n' ) {

        if ( c == BS && i > 0 ) {
            fprintf ( stdout, "%c-%c", BS, BS );
            i--;
        }
    }
}

```



```
        if ( c > 27 & c < 127 ) {
            p[i++] = c;
            putchar( '*' );
        }
        c = getchar();
    }
    p[i] = 0;
    tcsetattr( 0, TCSANOW, &t_reset);

#ifdef DEBUG
    fprintf(stdout, "\nP:<%s>\n", p );
#endif
}
```


8

Sockets

8.1 socket "unix"

Um socket é uma porta através da qual dois processos podem comunicar lendo ou escrevendo para lá. É, assim, à partida, um mecanismo de comunicação porque permite enviar dados de um processo para outro (um escreve, o outro lê). Mas é também um mecanismo de sincronização na medida em que o processo que lê, estando à escuta, espera pelo que escreve para reagir.

O exemplo seguinte representa um "servidor", isto é, um programa que espera que algum outro lhe envie algum pedido. Neste caso espera "à escuta" num socket à espera que apareça lá alguma coisa (algum "pedido"). Quando isso acontecer recebe ("lê") e reage, neste caso respondendo ao processo que mandou o pedido.

Para isso o servidor executa uma série de passos:

1. criar um socket

neste caso criamos um socket do tipo AF_UNIX que permite a comunicação entre processos na mesma máquina;

2. ligar esse socket a um endereço (bind)

neste tipo de socket o endereço é representado por um ficheiro especial que o sistema operativo cria para o efeito; no exemplo o ficheiro chama-se "socket-1";

3. ficar à escuta (listen)

desta forma o programa fica à escuta até que apareça alguma coisa no socket (ou seja, até que um cliente escreva alguma coisa para esse socket; lá iremos quando tratarmos, mais adiante, do cliente);

4. aceitar uma ligação (accept)

quando aparecer alguma coisa no socket é feita uma ligação, ficando então o servidor em comunicação com o cliente (adiante veremos o mesmo do ponto de vista do cliente);

5. comunicar com o cliente

finalmente usa-se `recv` e `send` para receber dados do cliente e responder.

Estes passos estão expressos no exemplo seguinte:

```
//Exemplo 4.1.1-Serv
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <string.h>
#include <sys/types.h>
```

```

#include <sys/socket.h>
#include <sys/un.h>
#define exit_on_error(s,m) if ( s < 0 ) { perror(m); exit(1); }

main() {
    int s;
    s = socket ( AF_UNIX, SOCK_STREAM, 0 );
    exit_on_error ( s, "ao criar socket");

    struct sockaddr_un s_addr;
    s_addr.sun_family = AF_UNIX;
    strcpy ( s_addr.sun_path, "meu-socket");

    unlink ( s_addr.sun_path );

    int status;
    status = bind ( s, (struct sockaddr*)&s_addr, sizeof(s_addr) );
    exit_on_error ( status, "bind");

    status = listen ( s, 3 );
    exit_on_error ( status, "listen");

    while ( 1 ) {
        printf ("listening...\n");

        int s_cliente;
        struct sockaddr_un s_addr_c;

        int t = sizeof(s_addr_c);
        s_cliente = accept ( s, (struct sockaddr *)&s_addr_c, &t );
        exit_on_error ( s_cliente, "accept");

        char msg1[100], msg2[100];
        int n = recv ( s_cliente, msg1, sizeof(msg1), 0 );
        exit_on_error ( n, "recv");

        printf ("pedido de %s\n", msg1 );
        sprintf ( msg2, "Hello, %s", msg1);

        n = send ( s_cliente, msg2, sizeof(msg2), 0 );
        exit_on_error ( n, "send");
    }
}

```

Pondo o servidor em execução esse ficará parado no `listen()` esperando uma tentativa de ligação por parte de um cliente. Verifique que, entretanto, foi criado o ficheiro especial "socket-1".

O exemplo seguinte representa um cliente que executa os seguintes passos para se ligar o obter resposta do servidor:

1. criar um socket

neste caso criamos um socket do tipo `AF_UNIX` que permite a comunicação entre processos na mesma máquina;

2. fazer uma ligação desse socket ao endereço do servidor (`connect`)

o endereço é, como anteriormente, representado pelo ficheiro especial "meu-socket";

3. comunicar com o servidor

a partir daí usar send e recv para mandar e receber dados do servidor;

```
//Exemplo 4.1.1-Cliente
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <string.h>
#include <sys/socket.h>
#include <sys/types.h>
#include <sys/un.h>
#define exit_on_error(s,m) if ( s < 0 ) { perror(m); exit(1); }

main() {

    int s;
    s = socket ( AF_UNIX, SOCK_STREAM, 0 );
    exit_on_error ( s, "socket");

    struct sockaddr_un s_addr;
    s_addr.sun_family = AF_UNIX;
    strcpy ( s_addr.sun_path, "meu-socket");

    int status;
    status = connect( s, (struct sockaddr*)&s_addr, sizeof(s_addr) );
    exit_on_error ( status, "connect");

    char nome[100], msg2[100];
    printf ("Nome: ");
    fgets ( nome, 100, stdin );

    int n = send ( s, nome, sizeof(nome), 0 );
    exit_on_error ( n, "nome");

    n = recv ( s, msg2, sizeof(msg2), 0 );
    exit_on_error ( n, "recv");

    printf ("%s\n", msg2 );

}
```

8.2 socket "internet"

O socket AF_UNIX serve para comunicar entre processos no mesmo servidor. Outros tipos de socket servem para comunicar entre diferentes máquinas nomeadamente, se for o caso, pela internet.

O exemplo seguinte mostra o servidor a responder no endereço 127.0.0.1 e no porto XX. A alteração face à versão anterior é basicamente no tipo de socket, que agora passa a ser do tipo AF_INET e na ligação ao endereço. Agora o bind passa a ser feito para um endereço e um porto IP. Neste caso usamos o endereço 127.0.0.1 que representa, também, a máquina local.

```
// sockint-serv.c
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
```

```

#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/un.h>
#include <netinet/in.h>
#define exit_on_error(s,m) if ( s < 0 ) { perror(m); exit(1); }

main() {
    int s = socket ( PF_INET, SOCK_STREAM, 0 );
    exit_on_error ( s, "socket");

    struct sockaddr_in s_addr;
    s_addr.sin_family = AF_INET;
    s_addr.sin_addr.s_addr = inet_addr ( "127.0.0.1" );
    s_addr.sin_port = htons (5678);

    unlink ( s_addr.sin_addr ); // Elimina ligações existentes
    int status;
    status = bind ( s, (struct sockaddr*)&s_addr, sizeof(s_addr) );
    exit_on_error ( status, "bind");

    status = listen ( s, 3 );
    exit_on_error ( status, "listen");

    while ( 1 ) {
        printf ("listening...\n");
        int s_cliente;
        struct sockaddr_un s_addr_c;

        int t = sizeof(s_addr_c);
        s_cliente = accept ( s, (struct sockaddr *)&s_addr_c, &t );
        exit_on_error ( s_cliente, "accept");

        char msg1[1000], msg2[1000];
        int n = recv ( s_cliente, msg1, sizeof(msg1), 0 );
        exit_on_error ( n, "recv");

        printf ("pedido de %s\n", msg1 );

        sprintf(msg2, "Hello, %s", msg1);

        n = send ( s_cliente, msg2, sizeof(msg2), 0 );
        exit_on_error ( n, "send");
    }
}

```

Para lançar o servidor pode ser necessário entrar como root. Neste caso concreto, como o número do porto é superior a 1024 (5678) não será necessário.

No cliente deverão ser feitas alterações semelhantes.

```

// sockint-client.c
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>

```

```

#include <sys/un.h>
#include <netinet/in.h>
#define exit_on_error(s,m) if ( s < 0 ) { perror(m); exit(1); }

main() {
    int s = socket ( PF_INET, SOCK_STREAM, 0 );
    exit_on_error ( s, "socket");

    struct sockaddr_in s_addr;
    s_addr.sin_family = AF_INET;
    s_addr.sin_addr.s_addr = inet_addr ( "127.0.0.1" );
    s_addr.sin_port = htons(5678);

    int status;
    status = connect( s, (struct sockaddr*)&s_addr, sizeof(s_addr) );
    exit_on_error ( status, "connect");

    char nome[1000], msg2[1000];
    printf ("Nome: ");
    fgets ( nome, 1000, stdin );

    int n = send ( s, nome, sizeof(nome), 0 );
    exit_on_error ( n, "nome");

    n = recv ( s, msg2, sizeof(msg2), 0 );
    exit_on_error ( n, "recv");

    printf ("%s\n", msg2 );
}

```

8.3 Exemplo com estruturas

Os exemplos seguintes ilustram o caso de se mandar/ receber do servidor uma estrutura.

```

// sockstr-serv.c
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/un.h>
#include <netinet/in.h>
#define exit_on_error(s,m) if ( s < 0 ) { perror(m); exit(1); }

typedef struct {
    int i;
    char nome[100];
    int j;
} MsgCliente;

main() {
    int s = socket ( PF_INET, SOCK_STREAM, 0 );
    exit_on_error ( s, "socket");

    struct sockaddr_in s_addr;

```

```

s_addr.sin_family = AF_INET;
s_addr.sin_addr.s_addr = inet_addr ( "127.0.0.1" );
s_addr.sin_port = htons ( 5678);

int status;
status = bind ( s, (struct sockaddr*)&s_addr, sizeof(s_addr) );
exit_on_error ( status, "bind");

status = listen ( s, 3 );
exit_on_error ( status, "listen");
while ( 1 ) {

    printf ("listening...\n");

    int s_cliente;
    struct sockaddr_un s_addr_c;

    int t = sizeof(s_addr_c);
    s_cliente=accept ( s, (struct sockaddr *)&s_addr_c, &t );
    exit_on_error ( s_cliente, "accept");

    int n;
    MsgCliente m;
    n = recv ( s_cliente, &m, sizeof(m), 0 );
    exit_on_error ( n, "recv");

    char msg[100];
    sprintf ( msg, "%d - %s - %d\n", m.i, m.nome, m.j );
    n = send ( s_cliente, msg, sizeof(msg), 0 );
    exit_on_error ( n, "send");
    close(s_cliente);
}
}

// sockstr-cli.c
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/un.h>
#include <netinet/in.h>
#define exit_on_error(s,m) if ( s < 0 ) { perror(m); exit(1); }

typedef struct {
    int i;
    char nome[100];
    int j;
} MsgCliente;

main() {
    int s = socket ( PF_INET, SOCK_STREAM, 0 );
    exit_on_error ( s, "socket");

    struct sockaddr_in s_addr;
    s_addr.sin_family = AF_INET;

```



```

s_addr.sin_addr.s_addr = inet_addr ( "127.0.0.1" );
s_addr.sin_port = htons(5678);

int status;
status=connect( s, (struct sockaddr*)&s_addr, sizeof(s_addr) );
exit_on_error ( status, "connect");

MsgCliente m;
m.i = 1;
strcpy ( m.nome, "Ze");
m.j = 2;

int n = send ( s, &m, sizeof(m), 0 );
exit_on_error ( n, "nome");

char msg2[100];
n = recv ( s, msg2, sizeof(msg2), 0 );
exit_on_error ( n, "recv");

printf ("%s\n", msg2 );
}

```

Com um bocado de sorte poderemos fazer um servidor que passa a responder, também, ao browser. Experimente colocar o endereço: <http://127.0.0.1:5678/> no seu browser depois de executar o servidor seguinte.

```

// int-server.c
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/un.h>
#include <netinet/in.h>
#define exit_on_error(s,m) if ( s < 0 ) { perror(m); exit(1); }

main() {
    int s = socket ( PF_INET, SOCK_STREAM, 0 );
    exit_on_error ( s, "socket");

    struct sockaddr_in s_addr;
    s_addr.sin_family = AF_INET;
    //s_addr.sin_addr.s_addr = inet_addr ( "127.0.0.1" );
    s_addr.sin_addr.s_addr = INADDR_ANY;
    s_addr.sin_port = htons (5678);

    int status;
    status = bind ( s, (struct sockaddr*)&s_addr, sizeof(s_addr) );
    exit_on_error ( status, "bind");

    status = listen ( s, 3 );
    exit_on_error ( status, "listen");
    while ( 1 ) {

        printf ("listening...\n");

```

```
    int s_cliente;
    struct sockaddr_un s_addr_c;

    int t = sizeof(s_addr_c);
    s_cliente=accept ( s, (struct sockaddr *)&s_addr_c, &t );
    exit_on_error ( s_cliente, "accept");

    char msg[] = "hello, world\n";

    int n;
    n = send ( s_cliente, msg, strlen(msg), 0 );
    exit_on_error ( n, "send");
    close(s_cliente);
}
}
```

Parte III

Anexos

9

Linguagem C

9.1 Introdução

Considere o seguinte programa, muito simples, que apenas escreve a mensagem “Hello World!”.

```
#include <stdio.h>
main() {
    printf("Hello World!\n");
}
```

A primeira linha do programa é um "include". Esta directiva inclui o ficheiro indicado, neste caso o ficheiro `stdio.h`. Este `stdio.h` contém a definição de uma série de funções standard de input/output (I/O) normalmente usadas nos programas em C.

A execução de um programa em linguagem C começa na função `main`, que neste caso chama a função `printf` para escrever a mensagem "Hello World!\n" no ecrã. Para executar este programa tem que:

1. escrever o programa num ficheiro de texto com extensão `.c`; por exemplo `hello.c`. Pode fazer isso em qualquer editor de texto como o `vi`, o `kwrite` ou o `nano`.
2. compilar o programa com o comando
`gcc hello.c -o hello`
3. executar o programa invocando o ficheiro executável criado.

Tipicamente na consola (janela de comandos) seria feita a seguinte sequência:

```
vi hello.c          # escrever o programa
gcc hello.c -o hello
./hello             # executar o programa
```

9.2 Variáveis

Os tipos básicos mais comuns são ilustrados nas seguintes declarações de variáveis:

```
int a = 10;
float f = 12.5;
char c = 'x';
```

A estes acrescem algumas variantes de tamanho e de aritmética. Por exemplo:

```
short int si = 10;
long int l = 10;
double d = 12.5;
unsigned int i = 10;
```

O `char` é um tipo inteiro com tamanho de 1 byte. Os outros tipos têm tamanho dependente da máquina. O operador `sizeof()` dá o tamanho de um tipo. Por exemplo:

```
sizeof(int)    dá o tamanho do tipo int; por exemplo 4 (32 bits)
int i;
sizeof(i)      dá, identicamente, o tamanho do tipo a que i pertence
```

A sintaxe dos caracteres é consistente com o código ASCII.

Exemplo. Código ASCII

```
c='A' é o mesmo que c=65;
'A'+1 é 'B' ou seja 66;
se c == 'B' então c - 'A' dá 1.
```

As conversões entre tipos básicos são automáticas, sem necessidade de explicitar o cast.

Exemplo. Conversões entre tipos básicos

```
int i =0;
float f = 12.5;
i = f; → é o mesmo que i = (int) f;
```

O tipo de uma expressão deriva implicitamente do tipo dos operandos. Assim, por exemplo:

```
int i = 11;
i = 2*(i / 2); → atribui a i o valor 2*5 (resultado da divisão inteira de 10 por 2);
i = (float) i / 2; → atribui a i o valor (int) 2*2.5;
```

9.2.1 Escrever com printf

O `printf` permite escrever o valor de variáveis (em geral expressões) na forma ilustrada nos seguinte exemplo

```
#include <stdio.h>
int main() {
    int i=10;
    printf ("Para passar é preciso ter %d ou mais\n", i);
}
```

O `printf` escreve no ecrã sempre e só a mensagem indicada no primeiro argumento. Mas, nessa mensagem, o símbolo `%d` não é para ser escrito literalmente: é para ser substituído por um número inteiro. O número a usar vem do argumento seguinte do `printf`, neste caso o valor de `i`. Por cada símbolo `%d` inserido na mensagem é necessário colocar um argumento adicional no `printf`. Por exemplo:

```
printf("Um int ocupa %d bytes e um float %d bytes\n", sizeof(int),
      sizeof(float) );
```

O símbolo `%d` está associado ao tipo inteiro. Da mesma forma os símbolos `%c` e `%f` estão associados ao tipo `char` e `float` respectivamente.

```
int i = 10;
char c = 'Y';
float f = 12.7;
printf("Exemplo de \num int: %d \num char: %c \num float: %f\n", i, c, f );
```

Como é óbvio, os símbolos de formatação presentes têm que concordar em número e em tipo com o número e de argumentos adicionais que lhes vão fornecer os valores. Exemplos:

```
int i = 10;
char c = "Y";
float f = 12.7;
printf ("O código ASCII de %c é %d\n", c, c );
printf ("%f arredonda para %d\n", f, (int)(f+0.5) );
```

Maus exemplos:

```
printf ("O que é isto ? %d\n");           // falta um argumento
printf ("Valor=%d\n", i, j);             // um argumento a mais
printf ("Escrever a parte inteira %d\n", f ); // discordância de tipo
```

9.2.2 Leitura com scanf

A função homóloga do printf para leitura é o scanf. Um exemplo:

```
#include <stdio.h>
main() {
    int n;
    printf ("Diga um número: ");
    scanf ("%d", &n);
    printf ("O número seguinte é: %d\n", n + 1);
}
```

Neste exemplo é lido um número, através do scanf, para a variável n.

A especificação da leitura é dada pelo símbolo %d no primeiro argumento e indica, neste caso, que se pretende ler um inteiro. No segundo argumento indica-se a variável que vai receber o valor lido: &n. Note especialmente o símbolo & antes do n.

Outros exemplos:

```
scanf ("%d%d", &i, &j); // lê dois inteiros, o primeiro para i o segundo para j;
scanf ("%c", &c );      // lê um caractere para a variável c;
```

9.3 Estruturas de controlo.

9.3.1 if

O if escreve-se com a condição entre parêntesis e com um else opcional, controlando uma instrução ou um bloco de instruções (entre chavetas). Pode-se encadear uma sequência de "else if".

Exemplo. if simples

```
#include <stdio.h>
main() {
    int a, b, m;
    printf ("Diga dois números: ");
    scanf ("%d%d", &a, &b);
    m = a;
    if ( b > a )
        m = b;
    printf ("O maior é: %d\n", m);
}
```

Exemplo. *if* com *else if*

```
#include <stdio.h>
main() {
    int a, b, m;
    printf ("Diga dois números: ");
    scanf ("%d%d", &a, &b);
    if ( a > b )
        printf ("O maior é: %d\n", a);
    else if ( b > a )
        printf ("O maior é: %d\n", b);
    else
        printf ("São iguais\n");
}
```

Exemplo. novamente um *if* simples

```
#include <stdio.h>
main() {
    int a, b;
    printf ("Diga dois números: ");
    scanf ("%d%d", &a, &b);
    if ( b > a ) {
        int t = a;
        a = b;
        b = t;
    }
    printf ("O maior é: %d\n", a);
}
```

9.3.2 Ciclos: *while*, *for*

O ciclo *while* tem a condição à cabeça (pode executar 0 ou mais vezes). Lê-se: "enquanto a condição for verdadeira... executar".

Exemplo. ler 10 números e escreve o maior (com *while*)

```
#include <stdio.h>
int main() {
    int i, n, maior=0;
    i = 0;
    while ( i < 10 ) {
        printf ("Diga um número: ");
        scanf ("%d", &n);
        if ( n > maior ) maior = n;
        i++;
    }
    printf ("O maior é %d\n", maior);
}
```

O *for* é uma forma compacta do *while* em que escreve o controlo do ciclo (inicialização; condição; incremento) na mesma linha de cabeçalho do ciclo.

Exemplo 24. ler 10 números e escreve o maior (com *for*)


```
#include <stdio.h>
main() {
    int i, n, maior=0;
    for ( i = 0; i <10; i++ ) {
        printf ("Diga um número: ");
        scanf ("%d", &n);
        if ( n > maior ) maior = n;
    }
    printf ("O maior deles é %d\n", maior);
}
```

Exemplo 25. ler uma sequência de números terminada pelo número 0 e calcular a soma

```
#include <stdio.h>
main() {
    int n=1, soma=0;                // é preciso inicializar n
    while ( n != 0 ) {
        printf ("Diga um número: ");
        scanf ("%d", &n);
        soma += n;
    }
    printf ("A soma é %d\n", soma);
}
```

9.3.3 Ciclo do-while

Ocasionalmente pode ter interesse o ciclo do while (condição no fim), que executa 1 ou mais vezes.

Exemplo. ler uma sequência de números terminada pelo número 0 e calcular a soma

```
#include <stdio.h>
main() {
    int n, soma=0;
    do {
        printf ("Diga um número: ");
        scanf ("%d", &n);
        soma += n;
    } while ( n != 0 );
    printf ("A soma é %d\n", soma);
}
```

9.3.4 switch

A instrução switch permite decidir entre os vários valores de uma variável de um tipo enumerável (int, long, short ou char). A ideia consiste em descrever o que fazer para cada um dos valores que a variável pode assumir. No final, pode existir um caminho de execução por omissão, que apanha todos os outros valores.

O Exemplo seguinte devolve uma descrição para cada número inteiro (Nenhum, Um, Dois, Vários ou Muitos). Todos os casos importantes são listados e correspondem a uma acção, todos os outros casos são apanhados por default. A execução do programa prossegue sempre dentro do switch até ser encontrado um break.

```

switch(numero) {
    case 0 : printf("Nenhum\n"); break;
    case 1 : printf("Um\n"); break;
    case 2 : printf("Dois\n"); break;
    case 3 :
    case 4 :
    case 5 :
        printf("Vários\n");
        break;
    default :
        printf("Muitos\n");
        break;
}

```

9.3.5 break

A instrução `break` é usada para sair de um ciclo, ou da instrução `switch`. Ao encontrar a instrução `break`, a execução do programa continua na primeira instrução que segue o ciclo ou o `switch`.

No exemplo que se segue a variável `i` assume os valores 0 a 5. Quando `i` é 5 executa-se a instrução `break` que passa o controlo para o `printf`, imediatamente após o ciclo.

```

#include <stdio.h>
int main() {
    int i = 0;
    while (i < 10000) {
        if (i==5) break;
        printf("i: %d\n", i);
        i++;
    }
    printf("O Valor de i é %d\n", i);
}

```

9.4 Expressões lógicas e condicionais

9.4.1 Valores lógicos

O valor inteiro 0 é tomado como *false*. Qualquer valor não 0 é tomado como *true*.

Exemplo. detecta se numa sequência de 10 números aparece o 7.

```

#include <stdio.h>
main() {
    int aparece = 0;    // inicializa a false
    int i, n;
    for ( i = 0; i <10; i++ ) {
        printf ("Diga um número: ");
        scanf ("%d", &n);
        if ( n == 7)
            aparece = 1;
    }
    if ( aparece )
        printf ("O 7 aparece\n");
    else

```

```
printf ("0 7 não aparece\n");
}
```

Os operadores relacionais são:

==	!=	>	>=	<	<=
igual	diferente	maior	maior ou igual	menor	menor ou igual

Os operadores lógicos são

&&		!
and	or	not

O resultado de uma expressão condicional ou lógica é 0 (false) ou 1 (true). Exemplos:

```
1==2          dá 0 (falso)
1==2 || 1 == 1 dá 1 (true)
!0            dá 1
!1            dá 0
!5            dá 0
```

9.4.2 Cuidados e abreviações

Construções do gênero

```
if (a != 0) ou while ( a != 0 )
```

podem-se abreviar para

```
if ( a ) ou while ( a )
```

A primeira forma dá 1 (true) para valores de a diferentes de 0. A segunda forma usa a circunstância de todos os valores diferentes de 0 serem tomados como true. A construção seguinte configura um ciclo infinito.

```
while ( 1 ) {
    ...
}
```

Um dos erros mais chatos é pôr um = em vez de ==. Por exemplo `if(i=7)` em vez de `if(i==7)`. Neste exemplo: o efeito do `if(n=7)` seria atribuir a n o valor 7 (o sinal igual sozinho é uma atribuição!). Desta forma o resultado da "condição" do if seria também o valor 7, ou seja, *true*. Por consequência programa daria sempre "O 7 aparece".

9.5 Funções

A sintaxe de definição de uma função inclui o tipo, nome da função e lista de argumentos. Por exemplo:

```
#include <stdio.h>
main() {
    func2 ( 1, 2 );
}
```

```

int func2 ( int a, int b ) {
    printf ("Recebi argumentos %d e %d\n", a, b );
    return a + b;
}

```

A instrução `return` tem uma função dupla de controlo (termina a execução da função) e formação do valor de retorno. Se a função não tiver valor de retorno (se for um "procedimento") pode ser definida com o tipo `void`.

Exemplo. escrever *n* vezes um caractere dado

```

void espaco ( int a, char c ) {
    while ( a-- ) printf ("%c", c);
}

```

Dependendo da variedade de compilador de C que estiver a usar pode ter que ser obrigado a declarar as funções antes de usar. A sintaxe de declaração das dois exemplos anteriores seria:

```

int func(int, int);
void espaco(int, char);

```

9.6 Arrays

A sintaxe de declaração e inicialização de arrays é ilustrada nos seguintes exemplos:

```

int a[5];           // dimensão 5 (índices de 0 a 4);
int a[] = { 1, 2, 3}; // dimensão 3
int c[5] = { 1, 2 }; // dimensão 5; inicializa as 3 posições iniciais

```

A inicialização é a única operação onde se pode manipular o array como um conjunto; em todas as outras só se pode aceder a uma posição através de um índice. Os índices começam em 0.

Exemplo. ler 10 números e escrevê-los por ordem inversa da de leitura

```

#include <stdio.h>
#include <stdio.h>
main() {
    int i, a[10];
    printf ("Escreva 10 números: ");
    for ( i=0; i<=9; i++ )
        scanf ( "%d", &a[i] );
    printf ("%d\n", somaa ( a, 10 ) );
}
int somaa ( int a[], int n ) {
    int soma = 0;
    for ( n--; n>=0; n-- )
        soma += a[n];
    return soma;
}

```

A referência `&a[i]` significa `&(a[i])`, mas os parêntesis podem ser omitidos uma vez que `[]` tem precedência sobre `&`.

Para receber um array como argumento pode-se usar a forma `a[]`. A dimensão, sendo necessária, terá que ser passada num argumento adicional.

Exemplo. ler 10 números e somar

```
#include <stdio.h>
main() {
    int i, a[10];
    printf ("Escreva 10 números: ");
    for ( i=0; i<=9; i++ )
        scanf ( "%d", &a[i] );
    printf ("%d\n", somaa ( a, 10 ) );
}
int somaa ( int a[], int n ) {
    int soma = 0;
    for ( n--; n>=0; n-- )
        soma += a[n];
    return soma;
}
```

9.7 Arrays de caracteres ("strings")

9.7.1 Terminador

Os arrays de caracteres podem ser usados da mesma forma que qualquer outro array, mas efectivamente têm algumas características particulares. A inicialização de um array de caracteres pode ser feita da forma compacta ilustrada no seguinte exemplo:

```
char s[10] = { 'H', 'e', 'l', 'l', 'o' };
char s[10] = "Hello";
```

A segunda forma inicializa da mesma forma as 5 primeiras posições do array mas com uma diferença: inicializa também a posição seguinte com um caractere terminador da string; deste modo, com a segunda inicialização, o array fica com o seguinte conteúdo:

H	e	l	l	o	\0	?	?	?	?
---	---	---	---	---	----	---	---	---	---

O caractere terminador representa-se pelo símbolo '\0' e é o caractere com código ASCII 0.

Esta forma de terminação, embora convencional, é praticamente uma regra de programação em linguagem C. Todas as funções e mecanismos da linguagem a usam e pressupõem.

Os arrays de caracteres têm, como qualquer outro array, uma dimensão "física" relacionada com o espaço de memória que lhe está atribuído; neste exemplo são 10 posições (10 bytes). Além disso, têm a dimensão "efectiva" correspondente ao conteúdo com significado, que vai do início até ao terminador; neste caso a dimensão efectiva seria 5.

Com base nesta convenção pode-se usar um array de caracteres sem saber, à partida, a sua dimensão física; basta saber a dimensão real que está marcada, no próprio array, pelo terminador.

Por exemplo, para imprimir um dado array de caracteres s, independentemente do número de posições do array, pode-se aplicar o procedimento

```
for ( i=0; s[i] != '\0'; i++ ) {
    printf ("%c", s[i]);
}
```

ou seja imprimir todos os caracteres até ao '\0'

Este tipo de convenção é usado por todas as funções que mexem em arrays. É o caso do printf que permite imprimir um array de caracteres usando o formatador %s. Por exemplo:

```
char s[] = "Hello World";
printf ( "%s\n", s );
```

Note que, face a esta convenção, é preciso salvaguardar sempre uma posição do array para conter o '\0'. Assim, por exemplo, para guardar a palavra "Hello" é preciso um array de, pelo menos, 6 posições.

No exemplo anterior a dimensão física do array s é dada implicitamente pela inicialização. Já não deve espantar que a dimensão resultante neste exemplo seja 12: os 11 caracteres indicados mais uma posição para o '\0'.

9.7.2 Funções de manipulação de strings

Suportadas na mesma convenção de terminação há uma série de funções de biblioteca que facilitam a manipulação de arrays de caracteres.

A função `int strlen(char[])` devolve a dimensão efectiva do array indicado como argumento (ou seja, o número de caracteres deste o início até ao '\0'). Isto corresponde basicamente a um raciocínio do género:

```
int foo_strlen( char s[] ) {
    int n=0;
    while ( s[n] )
        n++;
    return n;
}
```

Outras funções são

`strcpy (char s[], char a[])` copia a para s;

`strcat (char s[] , char a[])` junta ("concatena") a a s;

`n = strcmp (char a[], char b[])` dá o resultado da comparação entre a e b;

O `strcpy` corresponde à noção de atribuição aplicada a strings; por exemplo dadas as declarações:

```
char a[] = "Hello";
char s[10];
```

a chamada

```
strcpy ( s, a );
```

copia o conteúdo de a para s, de forma que fica também s com "Hello". Evidentemente s fica, como a, terminado com '\0'. O `strcpy` corresponde ao raciocínio:

```
foo_strcpy (char s[], char a[]) {
    int i;
    for ( i = 0; a[i]; i++ ) {
        s[i] = a[i];
    }
    s[i] = 0;
}
```

O `strcat` permite juntar duas strings. Por exemplo dados:

```
char s[20] = "Hello";
char s[10] = "World";
```

a sequência de instruções

```
strcat ( s, " ");
strcat ( s, a );
```

forma em `s` a expressão "Hello World".

A função `strcmp` compara duas strings e devolve 0 caso sejam iguais ou outro valor caso sejam diferentes. Mais concretamente devolve a diferença entre os dois primeiros caracteres em que as duas strings difiram. Desta forma pode-se usar o resultado do `strcmp` quer para comparar quer para ordenar as strings.

Por exemplo, dados:

```
char a[10] = "Hello";
char b[10] = "World";
char c[10] = "Worx";
```

`n=strcmp(a, "Hello")` devolve 0 indicando que as duas strings são iguais;

`n=strcmp(a,b)` devolve um número negativo indicando que `a` é menor que `b`; a diferença resulta da comparação entre H e W;

`n=strcmp(c, b)` devolve um número positivo indicando que `c` é maior que `b`; a diferença resulta da comparação de x e l, os primeiros caracteres diferentes.

9.8 Leitura e escrita de caracteres

9.8.1 Ler e escrever (com strings)

O `printf` com `%s` escreve uma string no ecrã. O `printf` percorre os caracteres desde o início do array até ao terminador `'\0'`, não conhecendo a dimensão física do array.

Para ler e escrever strings há algumas funções adicionais. Por exemplo, no programa seguinte, a função `fgets` lê uma linha, ou seja todos os caracteres que apareçam até ser encontrado um fim de linha (`'\n'`).

```
#include <stdio.h>
main() {
    char s[100];
    printf ("Nome :");
    fgets ( s, 100, stdin );
    printf ("O seu nome é %s\n", s );
}
```

A função `fgets` deixa o caractere de fim de linha no array:

J	o	a	o	\n	\0	?	...	?	?
---	---	---	---	----	----	---	-----	---	---

ou seja, no caso da leitura com `fgets` o próprio `'\n'` fica no array.

Podemos eliminá-lo facilmente com a seguinte instrução:

```
s[strlen(s)-1] = '\0';
```

9.8.2 Ler caracteres

A função `fgetc(stdin)` lê um caractere.

Exemplo:

```
#include <stdio.h>
main() {
    char s[100], c;
    int i = 0;
    printf ("Nome :");
    while ( ( c = fgetc(stdin) ) != '\n' && i < 100 ) {
        s[i++] = c;
    }
    s[i] = 0;

    printf ( "O seu nome tem %d caracteres", strlen(s) );
    if ( strcmp( s, "João" ) != 0 )
        printf ( " e não é João");
    printf ( ".\n", s );
}
```

9.8.3 Disciplina de leitura

Há questões e cuidados a ter quando se mistura a leitura de números e strings

Em primeiro lugar é preciso perceber que a leitura é um mecanismo de consumo sequencial (por ordem) dos caracteres que vão sendo escritos.

O `scanf` com `%d` consome todos os separadores que se apresentem (espaços, fins de linha) até encontrar o primeiro dígito, depois consome todos os dígitos (parando, isto é, já não consumindo o primeiro não dígito).

O `fgets` consome todos os caracteres até encontrar um enter (inclusivé).

Exemplo: são dados os seguintes caracteres

```
123 abc
345
Hello
```

Aplicando a seguinte sequência de funções:

`scanf("%d",&x)` consome o espaço e os dígitos 123; deixa o espaço a seguir ao 3;
x fica com 123

`fgets(s, ...)` consome o espaço a seguir ao 3, os caracteres abc e o fim da linha;
s fica com " abc\n";

`fgets(s, ...)` consome 345 e o fim da linha;
s fica com " 345\n";

`scanf("%d",&x)` termina no H (não consome nada)
x fica como estava (não é alterado)

Exemplo: são dados os seguintes caracteres


```
123
456
Hello
```

Aplicando a seguinte sequência de funções:

```
scanf("%d", &x) consome os dígitos 123; deixa o enter a seguir ao 3;
    x fica com 123
```

```
scanf("%d", &x) consome o enter e os dígitos 456; deixa o enter a seguir ao 6;
    x fica com 456
```

```
fgets(s, ...) consome o enter;
    s fica com " \n";
```

```
fgets(s, ...) consome o Hello e o enter;
    s fica com "Hello \n";
```

9.9 Ficheiros

As funções de leitura e escrita em ficheiros são muito semelhantes às de leitura e escrita no ecrã. Antes de ler ou escrever é preciso abrir o ficheiro.

9.9.1 Escrever

O seguinte exemplo ilustra um exemplo tipo "Hello World" para ficheiro:

```
#include <stdio.h>
main() {
    FILE *f;
    f = fopen ("teste.txt", "w");
    fprintf (f, "Hello World\n" );
    fclose ( f );
}
```

A variável `f` (de um tipo que por enquanto não discutimos) representa um "canal" para acesso a um ficheiro. A função `fopen` abre o ficheiro para escrita (é dado o nome, neste caso "teste.txt" e a operação, neste caso "w" de write).

A função `fprintf` permite escrever num ficheiro. É parecida com o `printf`: tem um primeiro argumento indicando o ficheiro; depois é igual.

Nem sempre a abertura de um ficheiro resulta (permissões, ...).

Se a abertura falhar o `fopen` devolve `NULL`:

```
main() {
    FILE *f;
    f = fopen ("teste.txt", "w");
    if ( f == NULL ) {
        printf ( "não correu bem...\n" );
        exit ( 1 );
    }
    fprintf (f, "Hello World\n" );
    fclose ( f );
}
```

Exemplo. ler 10 números e escrevê-los num ficheiro:

```

1  #include <stdio.h>
2  main() {
3      FILE *f;
4      f = fopen ("numeros_1.txt", "w");
5      int i, n;
6      for ( i = 0; i < 10; i ++ ) {
7          printf ("Diga: ");
8          scanf ("%d", &n);
9          fprintf ( f, "%d", n);
10     }
11     fclose ( f );
12 }
```

Nesta primeira versão os números ficam todos colados na primeira linha (uma desgraça, quando se tentarem ler). Consideremos então a versão 2, substituindo a linha 9 por:

```
fprintf ( f, "%d ", n);
```

Nesta segunda versão ficam separados por espaços na primeira linha. Uma terceira versão ficando um em cada linha seria conseguida acrescentando “\n” no fprintf.

```
fprintf ( f, "%d\n", n);
```

Exemplo. ler várias linhas do ecrã e escrevê-las no ficheiro (termina quando se der "fim").

```

#include <stdio.h>
main() {
    FILE *f;
    char linha[80], i = 1;
    f = fopen ("texto.txt", "w");
    while ( 1 ) {
        printf ("%d: ", i++);
        fgets(linha, 80, stdin);
        if ( ! strcmp ( linha, "fim\n" ) ) break;
        fprintf ( f, "%s", linha );
    }
    fclose( f);
}
```

9.9.2 Ler

A leitura usa também funções semelhantes às já conhecidas:

`fscanf(f,"%d",&n)` ler um inteiro do ficheiro `f` para a variável `n`

`fgets(s,100,f)` ler uma linha do ficheiro `f` para o array `f`

O primeiro passo é ainda abrir o ficheiro, no caso com a operação "r" (de "read").

O seguinte exemplo tenta ler os números do ficheiro `numeros_1.txt`

```

#include <stdio.h>
main() {
    FILE *f;
    f = fopen ("numeros_1.txt", "r");
    if ( f == NULL ) {
        perror ("file");
        exit(1);
    }
    int i, n;
    printf ("Numeros:\n");
    for ( i = 0; i < 10; i ++ ) {
        fscanf (f, "%d", &n);
        printf ( "%d: %d\n", i+1, n);
    }
    printf ("\n");
    fclose ( f );
}

```

O resultado da leitura poderá não ser grande coisa: o ficheiro só tem um número que pode ser gigante e, nesse caso, aparece mal (mas isso é mais uma questão das cadeiras de AC).

A leitura dos ficheiros das outras versões, *numeros_2.txt* e *numeros_3.txt*, deverá correr melhorzinho.

9.9.3 Detetar o fim de ficheiro

Em qualquer dos casos a leitura de um ficheiro nunca se faz como no exemplo anterior presumindo que se encontram lá 10 números. Lê-se de um ficheiro o que lá estiver, terminando "no fim".

As funções de leitura assinalam esse fim. Por exemplo, o `scanf` assinala o fim de ficheiro devolvendo EOF ("end of file"):

```

#include <stdio.h>
main() {
    FILE *f;
    f = fopen ("numeros_1.txt", "r");
    if ( f == NULL ) {
        perror ("file");
        exit(1);
    }
    int n, i = 0;
    printf ("Numeros\n");
    while ( fscanf (f, "%d", &n) != EOF ) {
        printf ( "%d : %d", ++i, n);
    }
    printf ("\n");
    fclose ( f );
}

```

O `fgets` assinala o fim de ficheiro devolvendo NULL:

```

main() {
    FILE *f;
    char s[80];
    int i = 0;
    f = fopen ("texto.txt", "r");
    while ( fgets (s, 100, f) != NULL ) {
        printf ( "%d: %s", ++i, s );
    }
}

```

```

    }
    printf ("\n");
    fclose ( f );
}

```

9.9.4 Exercício

Pretende-se criar um editor para edição de ficheiros de texto. Suponha que o nome do ficheiro de texto é fixo: teste.txt. Faça um programa que apresente um menu com as opções: mostrar, acrescentar, apagar, duplicar e sair. A opção “mostrar” deverá mostrar o conteúdo do ficheiro. A opção acrescentar pede uma nova linha e acrescenta-a ao ficheiro. A opção apagar apaga uma linha, com base no número de linha introduzido pelo utilizador.

Melhore o exercício anterior de forma a:

- dar o nome do ficheiro no início (deixar de ser um nome de ficheiro fixo)
- acrescentar operações (ex: substituir o conteúdo de uma linha)
- acrescentar possibilidades (ex: mostrar o ficheiro pagina a página, procurar texto, etc)

9.10 Estruturas

Uma estrutura é uma variável composta de vários campos (ou "membros"), podendo cada um deles ser uma variável comum ou outra estrutura.

```

struct Aluno {
    int num;
    char nome[100];
    int nota;
} a, b, c;

```

Este exemplo representa uma estrutura correspondente a uma pauta de alunos; o aluno é representado por um número, nome e nota.

As variáveis a, b e c representam, cada uma, um aluno. Cada um delas tem, portanto, 3 campos, designados num (um inteiro), nome (um array de 100 caracteres) e nota (outro inteiro).

Para aceder a um campo da variável usa-se o . (ponto).

Assim, por exemplo:

a.num campo num do aluno a

b.nome campo nome do aluno b

Face àquelas declarações fazem sentido as seguintes construções:

```

a.num = 1001;
strcpy ( b.nome, "Joao" );
printf ( "%d - %s - %d\n", a.num, a.nome, a.nota );

```

9.10.1 Typedef

O typedef é uma instrução permite dar um novo nome a um tipo de dados. Considere o seguinte:

```
typedef int Inteiro;
typedef float Peso;
```

Desta forma poderíamos definir variáveis do tipo Inteiro e Peso.

```
Inteiro a = 10;
Peso meupeso = 120.5;
```

O typedef é especialmente útil para ajudar a definir tipos compostos, designadamente estruturas. Por exemplo:

```
typedef struct {
    int num;
    char nome[100];
    int nota;
} TipoAluno;
```

define um TipoAluno que permite, depois, fazer declarações equivalentes às anteriores:

TipoAluno a, b, c;

9.10.2 Arrays de estruturas

A seguinte declaração

```
TipoAluno a, pauta[20];
```

declara duas variáveis: a uma estrutura do tipo TipoAluno e um array pauta com 20 elementos do mesmo tipo. Face a estas declarações fazem sentido as seguintes construções:

```
pauta[1].num = 1070;
strcpy ( pauta[1].nome , "João" );
printf ("O nome do 3º aluno começa pela letra %c\n", pauta[2].nome[0]);
```

9.11 Ficheiros binários

Nos exemplos anteriores usámos apenas ficheiros de texto. Dizemos "ficheiro de texto" quando o conteúdo é constituído por códigos de caracteres ASCII e, portanto, o seu conteúdo é directamente visível por um utilizador; ou seja, pode-se editar directamente o conteúdo do ficheiro num editor de texto (vi, notepad, ...) ou executar o comando cat para ver o conteúdo no ficheiro no ecrã.

Ao contrário, se um ficheiro tiver um conteúdo não interpretável como caracteres ascii dizemos que é um ficheiro binário.

O seguinte exemplo cria um "ficheiro de texto", escrevendo lá um inteiro, um nome e outro inteiro.

```
// escreve1.c
#include <stdio.h>
main() {
    FILE *fp = fopen ( "teste.txt", "w" );
    fprintf ( fp, "%d%s%d\n", 0x40414243, "hello", 100 );
    fclose ( fp );
}
```

Uma vez executado o programa podemos ver o ficheiro teste.txt no ecrã ou no vi, por exemplo. De qualquer modo, para podermos inspeccionar cada um dos bytes do ficheiro com mais certeza vamos fazer o seguinte programa:

```
// mostra.c
#include <stdio.h>
int main( int argc, char *argv[]) {
    if ( argc < 2 ) return printf ( "usage: mostra ficheiro\n");

    FILE *fp = fopen ( argv[1], "r" );
    int i = 1 , c;

    while ( ( c = fgetc(fp)) != EOF ) {
        printf ("%3d(%c)\t", c, c > 27 ? c : '-' );
        if ( i++ %8 == 0) printf ("\n" );
    }
    printf ("\n" );
}
```

Este programa mostra cada um dos bytes do ficheiro em decimal e, se for o caso, com o caractere ascii correspondente. Para experimentar o programa fazemos:

```
> gcc mostra.c -o mostra
> ./mostra teste.txt
```

passando, portanto, o nome do ficheiro que se quer inspeccionar como argumento. Poderemos então ver, com facilidade, cada um dos bytes/caracteres do ficheiro.

Consideremos, agora, este outro exemplo:

```
// escreve2.c
#include <stdio.h>
#include <string.h>

typedef struct {
    int i;
    char s[20];
    int j;
} StructTeste;

main() {
    FILE *fp = fopen ( "teste.dat", "w" );
    StructTeste t;
    t.i = 0x40414243;
    strcpy ( t.s, "Hello");
    t.j = 100;
    fwrite ( &t, sizeof(t), 1, fp );
    fclose ( fp );
}
```

que escreve mais ou menos o mesmo conteúdo mas em binário. A diferença mais sensível é nos inteiros que agora, em vez de serem escritos em caracteres são escritos em binário (isto é em base 2, como foi visto na cadeira de AC).

Podemos, ainda assim, ver os bytes com o nosso programa mostra. Agora não é tão fácil mas ainda assim é possível interpretar o resultado. Os pontos mais salientes:

- por coincidência alguns dos números em binário correspondem a caracteres visíveis do código ascii; mas é coincidência...

- a string é ainda visível como tal porque a representação em binário é justamente, ainda, o código ascii; note-se em particular a presença do terminador a seguir ao "Hello"; do terminador para a frente é "lixo";

Ainda um pormenor: de alguma forma parece que os números binários estão ao contrário? Seria uma questão a aprofundar (Google: "little and big endian"...).

Os exemplos seguintes demonstram a utilização de um ficheiro para guardar uma pauta (ver Guia Linguagem C).

```
// regista_pauta.c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#define exit_on_null(p,m) if ( p == 0 ) { perror(m); exit(1); }

typedef struct {
    int n;
    char nome[120];
} Aluno;

void ler_enter(FILE *f) {
    while ( fgetc(f) != '\n' );
}

void tirar_enter ( char s[] ) {
    char c = s[ strlen(s) - 1 ];
    if ( c == '\n' )
        s[strlen(s)-1] = '\0';
}

main() {
    Aluno a;
    FILE *fp = fopen ( "pauta.dat", "a" );
    exit_on_null (fp, "Erro de abertura");

    printf("Num:");
    scanf("%d", &a.n ); ler_enter(stdin);

    printf("Nome:");
    fgets(a.nome, 120, stdin); tirar_enter(a.nome);

    if ( fwrite ( &a, sizeof(a), 1, fp ) > 0 ) {
        printf("Aluno registado: %d\t%s\n", a.n, a.nome );
    }
    fclose( fp );
}
```

mostra_pauta.c

```
// mostra_pauta.c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#define exit_on_null(p,m) if ( p == 0 ) { perror(m); exit(1); }

typedef struct {
```

```

    int n;
    char nome[120];
} Aluno;

main() {
    Aluno a;
    FILE *fp = fopen ( "pauta.dat", "r" );
    exit_on_null (fp, "Erro de abertura");
    while ( fread ( &a, sizeof(a), 1, fp ) > 0 ) {
        fprintf ( stdout, "%d\t%s\n", a.n, a.nome );
    }
    fclose( fp );
}

```

9.12 Ponteiros

9.12.1 Memória dinâmica

Considere o seguinte exemplo:

```

#include <stdio.h>
#include <stdlib.h>
main() {
    int a, *p;
    p = malloc ( sizeof(int) );
    a = 10;
    (*p) = 10;
    printf ( "%d;%d\n", a, (*p) );
}

```

A função `malloc()` permite ao programa arranjar mais memória. Neste caso arranjar `sizeof(int)` bytes, ou seja, o suficiente para guardar um número inteiro.

Ficamos assim no programa com duas variáveis `int`. A variável `a` que é declarada e fica disponível para utilização através do nome, como é normal. E essa outra que é criada com o `malloc()` e, a partir daí, fica também disponível para utilização, através de `(*p)`.

A primeira variável é criada pelos mecanismos normais previstos na linguagem, ou seja, a declaração. A segunda é criada pela chamada à função `malloc()` dizemos que é criada dinamicamente (ou que é "memória dinâmica").

O `p` tem um papel importante neste mecanismo de utilização de memória dinâmica. É através do `p` que se chega ao espaço de memória criado pelo `malloc()`. Ao fazer

```
malloc ( sizeof(int) );
```

estamos a invocar o `malloc()` que vai criar e dar ao nosso programa um novo espaço de memória.

Ao fazer

```
p = malloc ( sizeof(int) );
```

estamos a dizer que `p` fica a apontar para esse espaço. Ou seja será através de `p` que se vai poder aceder a esse espaço para usá-lo como variável. Daí dizer-se que `p` é um ponteiro (ou a apontador, se quiser).

E como é que se acede a esse espaço através de p. Obviamente, como a sintaxe do programa sugere, através de

```
(*p)
```

que dizemos ser "o que é apontado por p".

Em suma, a sequência

```
p = malloc ( sizeof(int) );
(*p) = 10;
```

- cria um espaço de memória com tamanho para um inteiro (ou seja uma variável int), ficando p a apontar para essa variável;
- nessa variável (a tal para que aponta) colocamos o valor 10.

9.12.2 Inicialização

Podemos experimentar este programa

```
#include <stdio.h>
#include <stdlib.h>
main() {
    int *p;
    *p = 10;
    printf ( "%d\n", *p );
}
```

O resultado será muito possivelmente um erro. Ou este, que não falha - dá mesmo erro:

```
#include <stdio.h>
#include <stdlib.h>
main() {
    int *p = NULL;
    *p = 10;
    printf ( "%d\n", *p );
}
```

No primeiro caso estamos a usar o que é "apontado por p" antes de ter posto p a apontar para algum lado; só por muita sorte p estaria a apontar para um sítio válido. No segundo caso estamos a inicializar p indicando que aponta para "lado nenhum"; é esse o sentido lógico da inicialização com p=NULL,

Ambos estão mal, claro. Só podemos usar o espaço para que p aponta depois de pôr p a apontar para um espaço que se possa usar (pois, é de lapalisse).

9.12.3 Memória dinâmica : arrays

Considere o seguinte exemplo:

```
#include <stdio.h>
#include <stdlib.h>
main() {
    int a, *p=NULL;
    p = malloc ( 10 * sizeof(int) );
    *p = 10;
```

```

    *(p+1) = 11;
    *(p+2) = *p + 2;
    printf ("%d %d %d\n", *p, *(p+1), *(p+2) );
}

```

Neste caso estamos a usar o malloc para arranjar espaço correspondente a 10 inteiros.

Em alternativa podemos usar a função calloc() que cria um conjunto de posições consecutivas que são, também, inicializadas com 0 (contrariamente ao malloc() que não inicializa).

```
p = calloc (sizeof(int), 10 );
```

A semelhança entre este array e o "array dinâmico" do ponto 13.4 é a origem do espaço de memória. Trata-se em qualquer dos casos de um espaço de memória semelhante: num caso e noutro um conjunto de posições consecutivas onde se podem guardar 10 inteiros.

9.12.4 Aritmética de ponteiros

Para usar estas 10 posições temos que ampliar a exploração do *p da forma ilustrada no exemplo. Assim, tal como

*p representa a posição de memória apontada por p,

*(p+1) representa a posição de memória seguinte.

Dizemos, "apontada por p+1". Desta forma podemos aceder a cada um das 10 posições de criadas pela chamada ao malloc(), desde a primeira, *p até à última *(p+9).

Na realidade, uma vez que p é uma variável, e portanto modificável, podemos fazer variar o próprio p para aceder às várias posições. Assim, por exemplo:

```

*p = 10;
p = p + 1;
*p = 11;

```

Coloca 10 numa posição de memória e 11 na posição seguinte.

Temos assim uma aritmética específica dos ponteiros, com um significado lógico. Dado um ponteiro p

p + i aponta para uma posição de memória i posições à frente, e

p - i aponta para uma posição de memória i posições para trás.

Na mesma lógica funcionam as operações relacionais. Por exemplo, dados dois ponteiros p1 e p2,

p1 == p2 se apontam para a mesma posição;

p1 > p2 se p1 aponta para uma posição à frente da apontada por p2.

O seguinte exemplo inicializa um conjunto de 10 posições de memória com números de 10 a 10 que, depois, depois escreve por ordem inversa no ecrã.

```

#include <stdio.h>
#include <stdlib.h>
main() {
    int i, *p=NULL;
    p = malloc ( 10* sizeof(int) );
    for ( i=0; i<10;i++) {
        *(p+i) = 10+i;
    }
    int *p1 = p + 9;
    for ( ; p1 >= p; p1-- ){
        printf ("%d\n", *p1 );
    }
}

```

9.12.5 Sintaxe de arrays

Na realidade os ponteiros podem ser usados com a mesma sintaxe dos arrays. O conceito

`*(p+i)`

a posição de memória apontada por `p+i`, ou seja, a posição que se encontra `i` casas à frente da posição inicial `p` é o mesmo expresso pela notação

`p[i]`

As duas sintaxes são intermutáveis.

O seguinte exemplo cria um conjunto de 10 posições de memória e inicializa-as com números de 10 a 19.

```

#include <stdio.h>
#include <stdlib.h>
main() {
    int i, *p=NULL;
    p = malloc ( sizeof(int), 10 );
    for ( i=0; i<10;i++) {
        p[i] = i + 10;
    }
}

```

9.12.6 Ponteiros e arrays

A semelhança entre ponteiros e arrays funciona também no sentido contrário: um array pode também ser manipulado com a sintaxe dos arrays.

Na realidade um array é um ponteiro, que aponta para a primeira posição do array). Assim, querendo, podemos aceder às suas diversas posições usando a sintaxe dos arrays, como acontece neste exemplo:

```

#include <stdio.h>
#include <stdlib.h>
main() {
    int a[10] = { 10, 11, 12, 13, 14, 15, 16, 17, 18, 19};
    int i;
    for ( i=0; i<10;i++) {

```

```

        printf ("%d\n", *(a+i) );
    }
}

```

Em suma, o `a` deste exemplo é da mesma natureza que o `p` dos exemplos anteriores. A diferença é que o `a` é um ponteiro constante; aponta sempre para a mesma posição, que é o início do array declarado. Já uma variável `p` declarada como nos exemplos anteriores

```
int *p;
```

sendo uma variável pode ser modificada - tendo neste caso por consequência apontar para posições diferentes. Assim, por exemplo, pode-se fazer:

```
int a[10];
p = a;
```

desta forma `p` com o mesmo valor que `a`, ou seja fica a apontar para a mesma posição que `a`. Desta forma `p[i]` passa a ser o mesmo que `a[i]`. Fazendo, por exemplo:

```
int a[10];
p = a+1;
```

`p` fica a apontar para a posição a seguir a `a`. Desta forma `p[i]` passa a ser o mesmo que `a[i+1]`. E, por exemplo, `p[-1]` será, na circunstância, o mesmo que `a[0]`.

Não é possível obviamente fazer:

```
a=p;
```

porque `a` é um ponteiro constante: aponta sempre para a posição onde começa o array `a`.

9.12.7 Ponteiros para estruturas

O seguinte exemplo cria uma variável dinâmica do tipo `Aluno`.

```

typedef struct {
    int num;
    char nome[100];
    int nota;
} TipoAluno;

main() {
    TipoAluno *p;
    p = malloc( sizeof(TipoAluno) );
    (*p).num = 100;
    strcpy ( (*p).nome, "Zé Carlos" );
    (*p).nota = 17;
}

```

Para o efeito é preciso dispor de um ponteiro para uma variável `TipoAluno`. É isso que é definido na declaração:

```
TipoAluno *p;
```

A função `malloc` é chamada neste caso para obter um espaço com o tamanho da estrutura; o ponteiro `p` fica a apontar para esse espaço. O `*p` denota, assim, uma variável do tipo `TipoAluno`. Esta variável é depois usada como uma estrutura normal, acedendo com o `.` a cada um dos seus campos.

Entretanto há nestas circunstâncias uma sintaxe alternativa que se pode utilizar, `p->nome` em vez de `(*p).num`

Assim, o programa ficaria

```
...
p = malloc( sizeof(TipoAluno) );
p->num = 100;
strcpy ( p->nome, "Zé Carlos" );
p->nota = 17;
```

9.12.8 Tipo dos ponteiros

Considere as seguintes declarações:

```
int *p_inteiro;
TipoAluno *p_aluno;
```

Trata-se de dois ponteiros, com tipos diferentes. O primeiro é um ponteiro para inteiro, ou seja é um ponteiro do tipo

```
int *
```

O segundo é um ponteiro para `TipoAluno`. É do tipo

```
TipoAluno *
```

As declarações têm aliás estas duas formas interessantes de serem lidas. Na perspectiva do tipo apontado podemos ler

```
int *p
```

salientando que o conjunto `(*p)` denota um inteiro. Na perspectiva do próprio apontador podemos ler

```
int *p;
```

que indica que `p` é um "apontador para inteiro".

O ponto importante é que um ponteiro aponta para um tipo de dados.

As funções `malloc()` devolvem um tipo

```
void *
```

que deve ser convertido, através de um `cast`, para um tipo específico. Por exemplo:

```
int *p = (int *) malloc ( 10 * sizeof(int) );
```

O `cast` de `void *` para outro tipo de ponteiro é automático; pode ser o omitido, como aconteceu nos exemplos anteriores.

9.13 Algumas situações com ponteiros

9.13.1 Ponteiros e endereços

Fisicamente um ponteiro contém um endereço de memória: o endereço da posição para que aponta. Por exemplo na instrução:

```
int *p = malloc ( sizeof(int) );
```

o `p` recebe o endereço da posição de memória que o `malloc` obteve para ser usada pelo programa. O operador `&` permite obter o endereço de uma variável comum. Por exemplo, dada a declaração:

```
int i;
```

a expressão `&i` denota o endereço de `i`.

Sendo `&i` um endereço pode ser atribuído a um apontador, que fica então a apontar para `i`:

```
int *p;
p = &i;
```

Nestas circunstâncias pode-se aceder a `i` através da `p`. Por exemplo: `*p = 10` ou `p[0] = 10`, fazem o mesmo que `i = 10`.

9.13.2 Passagem por referência

Na linguagem C os argumentos são sempre passados por valor. Ou seja, quando se invoca uma função são criadas variáveis locais da função para receber os argumentos.

```
main() {
    int x = 3;
    f ( x, 10 );
    ...
}

void f ( int a, int b ) {
    // a e b são variáveis locais...
    // neste caso a vai ser inicializada com o valor de x,
    // ou seja com 3; e b com 10;
    ...
}
```

Por consequência, nunca é possível a uma função alterar as variáveis passadas como argumento.

Em certas circunstâncias dá, de facto, jeito que uma função possa alterar variáveis passadas como argumento. Não sendo possível fazê-lo directamente pode-se obter o mesmo efeito passando como argumento não a variável mas o seu endereço.

```
main() {
    int x = 3;
    f ( &x );
    printf ("%d\n", x );
}

int f ( int *p ) {
    *p = 4;
}
```

A função `f()` recebe como argumento um ponteiro `int *`. Neste caso é passado como argumento `&x` ou seja o endereço de `x`, ou seja, o argumento `p` fica a apontar para `x`. Desta forma quando dentro da função se aceder a `*p` está-se a mexer na variável `x`.

Exemplo:

```
main() {
    int x = 3, y = 2;
    troca ( &x, &y);
    printf ("%d %d\n", x, y );
}

void troca ( int *pa, int *pb ) {
    int x = *pa;
    *pa = *pb;
    *pb = x;
}
```

Tratando-se de um array a situação é diferente. O array é um ponteiro (para a primeira posição do array). A partir dele a função pode aceder aos elementos do array, designadamente para os alterar se for o caso.

Exemplo: a seguinte função inicializa um array com o valor indicado no argumento.

```
main() {
    int a[10];
    init ( a, 10, 20);
}

void init ( int *p, int n, int v) {
    int i;
    for ( i=0; i<n; i++ ) {
        p[i] = v;
    }
}
```

9.13.3 Argumentos nas funções de leitura

Na instrução

```
scanf ("%d", &n);
```

o `scanf` recebe como argumento o endereço da variável `n`. Isto é indispensável para que a função possa alterar o valor de `n`, atribuindo-lhe o valor lido do teclado.

A situação é diferente quando se trata de um array como nos exemplos:

```
scanf ("%s", s);
fgets(s, ..., stdin);
```

A passagem do array com argumento permite às funções alterar o seu conteúdo.

9.13.4 Argumentos na linha de comando (`argc`, `argv`)

O seguinte exemplo escreve os argumentos recebidos na linha de comando.

```

#include <stdio.h>
#include <time.h>
int main() {
    struct tm st;
    time_t t = time ( NULL );
    st = * gmtime ( &t );
    printf ("Dia: %d-%02d-%02d\n", 1900+st.tm_year, st.tm_mon+1, st.tm_mday );
    printf ("Hora: %02d-%02d-%02d\n", st.tm_hour, st.tm_min, st.tm_sec );
}

```

Figura 9.14.1: Programa permite obter a data e hora atual

```

int main( int argc, char *argv[] ) {
    int i;
    for ( i=0; i<argc; i++ ) {
        printf ("%d-> %s\n", i, argv[i] );
    }
}

```

O `argv[]` é um array de ponteiros para `char` que veicula os argumentos recebidos na linha de comando. O `argc` indica quantos elementos existem nesse array.

Suponha que este programa existe num ficheiro teste e é executado dando o comando:

```
teste A B "1+2"
```

O primeiro elemento `argv[0]` contém sempre o nome do programa invocado. Os seguintes os argumentos adicionais dados na linha de comando, neste caso `argv[1]` com A, etc.

9.14 Exemplos

9.14.1 Obter a data e hora atual

O programa da Figura 9.14.1 permite obter a data e hora atual. Para obter mais informações sobre os elementos envolvidos poderá usar os comandos: `man 3 time` ou `man 3 gmtime`.

9.14.2 Gestão de uma Pauta

Versão 1: só o menu

O exemplo da Figura 9.14.2 implementa uma Pauta com as opções constantes de um menu. Há um pequeno problema: a função `nop()` tinha como intenção obrigar a uma paragem para o utilizador carregar em enter mas não resulta ("não espera"). A razão é simples: o `scanf` lê a opção mas deixa o enter por lêr; o `fgetc()` encontra esse enter e, por isso, não espera. Podemos ter o cuidado de limpar o resto da linha depois de ler a opção. Por exemplo com:

```

char dummy[100];
...
scanf ("%d", &opcao );
fgets (dummy, 100, stdin);
...

```



```

#include <stdio.h>
main() {
    int opcao;
    do {
        printf ("1. Inserir aluno\n" );
        printf ("2. Apagar aluno\n" );
        printf ("3. Inserir nota\n" );
        printf ("4. Lista de alunos\n" );
        printf ("5. Pauta por nome\n" );
        printf ("0. Sair \n" );
        scanf ("%d", &opcao );
        if ( opcao == 1 ) nop ( );
        if ( opcao == 2 ) nop ( );
        if ( opcao == 3 ) nop ( );
        if ( opcao == 4 ) nop ( );
    } while ( opcao != 0 );
}

int nop ( ) {
    printf ("Opção ainda em construção\n" );
    printf ("Carregue enter para continuar...");
    fgetc (stdin);
}

```

Figura 9.14.2: Gestão de uma Pauta: Versão 1

ou

```

scanf ("%d", &opcao );
while ( fgetc(stdin) != '\n' );
...

```

Versão 2: estrutura de dados

Vamos guardar os dados da pauta num array de estruturas, complementando com uma variável que indica o número de alunos (inicialmente 0).

```

typedef struct {
    int num;
    char nome[100];
    int nota;
} TipoAluno;

TipoAluno pauta[20];
int n_alunos = 0;

```

Estas variáveis vão ser partilhadas (ou seja vai ser globais) para conjunto de funções. Para que várias funções possam usar as mesmas variáveis vamos declará-las como estáticas (fora de qualquer função). Nesta versão 2 vamos implementar duas operações: a 1 para inserir um aluno e a 3 para listar (de forma a poder, de imediato, verificar se a inserção está a funcionar).

```

#include <stdio.h>
typedef struct {
    int num;

```

```

        char nome[100];
        int nota;
    } TipoAluno;

TipoAluno pauta[20];
int n_alunos = 0;

main() {
    int opcao;
    do {
        printf ("1. Inserir aluno\n" );
        printf ("2. Apagar aluno\n" );
        printf ("3. Inserir nota\n" );
        printf ("4. Lista de alunos\n" );
        printf ("5. Pauta por nome\n" );
        printf ("0. Sair \n" );
        scanf ("%d", &opcao );
        limpar_linha();
        if ( opcao == 1 ) inserir_aluno ();
        if ( opcao == 2 ) nop ();
        if ( opcao == 3 ) nop ();
        if ( opcao == 4 ) listar ();
        if ( opcao == 5 ) nop ();
    } while ( opcao != 0 );
}

int nop () { ... }

int limpar_linha() {
    while ( fgetc(stdin) != '\n');
}

int inserir_aluno () {
    int num;
    char nome[100];

    printf ("Nº aluno: " );
    scanf ("%d", &num );
    limpar_linha();           // senão o nome vai apanhar o enter
                             // e ficar vazio
    printf ("Nome: " );
    fgets(nome, 100, stdin);
    pauta[n_alunos].num = num;
    strcpy ( pauta[n_alunos].nome, nome );
    n_alunos++;
}

int listar () {
    int i;
    for ( i = 0; i < n_alunos; i++ ) {
        printf ("%5d %s\n", pauta[i].num, pauta[i].nome );
    }
}

```

Versão 3

Juntam-se nesta versão mais duas operações: a 3 lançar nota e a 5 para mostrar a pauta.

```

int lancar_nota ( ) {
    int i, num;
    printf ("Nº aluno: " );
    scanf ("%d", &num );
    limpar_linha();

    for ( i = 0; i < n_alunos; i++ ) {
        if (pauta[i].num == num) break;
    }
    if ( i == n_alunos ) {
        printf ("Aluno %d não consta\n", i );
        return;
    }

    printf ("Nota: " );
    scanf ("%d", &num );
    pauta[i].nota = num;
    limpar_linha();
}

int pauta() {
    int i;

    espacos( '=', 62 ); printf ("\n");
    printf ( "Nº    %-50s  Nota\n", "Nome");
    espacos ( '=', 62 ); printf ("\n");

    for ( i = 0; i < n_alunos; i++ ) {
        printf ( "%-5d %-50s %4d\n", pauta[i].num,
                                   pauta[i].nome, pauta[i].nota );
    }
    espacos ( '=', 62 ); printf ("\n");
}

int espacos ( char c, int n) {
    int i;
    for ( i = 1; i < n; i++ ) putchar (c);
}

```

Acrescentamos ainda uma função para ordenar a pauta, a chamar após a inserção de um novo aluno.

```

int ordenar_nome ( ) {
    int i, j;

    for ( i = 0; i < n_alunos; i++ ) {
        for ( j = i + 1; j < n_alunos; j++ ) {
            if ( strcmp( pauta[i].nome, pauta[j].nome ) > 0 ) {
                TipoAluno a;
                a = pauta[i];
                pauta[i] = pauta[j];
                pauta[j] = a;
            }
        }
    }
}

```

Versão 4 : guardar em ficheiro

Nesta versão vamos guardar os dados em ficheiro. O plano é: no início do programa lê-se do ficheiro para o array, onde depois se faz toda a manipulação dos dados; no fim do programa volta a gravar-se o array para ficheiro.

Para gravar usa-se a seguinte função

```
int gravar() {
    FILE *f;
    f = fopen ("pauta.txt", "w" );
    int i;
    for ( i = 0; i < n_alunos; i++ ) {
        fprintf ( f, "%d\n%s\n%d\n" , pauta[i].num, pauta[i].nome,
                pauta[i].nota );
    }
    fclose(f);
}
```

Fica, portanto, o número numa linha, o nome noutra e a nota ainda noutra (houve uma primeira ideia de pôr tudo na mesma linha mas isso não iria facilitar a leitura !).

Para ler usa-se a seguinte função:

```
int ler() {
    FILE *f;
    f = fopen ("pauta.txt", "r" );
    if ( f == NULL ) return;
    printf ("Carregar ficheiro...");
    int num, nota;
    char nome[100];

    while ( fscanf ( f, "%d", &num ) != EOF ) {
        fgets ( nome, 100, f );
        nome[strlen(nome)-1] = 0;
        fscanf ( f, "%d", &nota );

        pauta[n_alunos].num = num;
        strcpy ( pauta[n_alunos].nome , nome );
        pauta[n_alunos].nota = nota;
        n_alunos++;
    }
    fclose(f);
    printf ("\n");
}
```

A função ler é chamada no início do main e a função escrever é chamada no fim do main.

Versão 5: memória dinâmica

```
typedef struct _tipo_aluno {
    int num;
    char nome[100];
    int nota;
    struct _tipo_aluno *prox;
} TipoAluno;

TipoAluno *pauta = NULL
```

```

int inserir_aluno () {
    int num;
    char nome[100];

    printf ("Nº aluno: " );
    scanf ("%d", &num );
    limpar_linha();
    printf ("Nome: " );
    fgets( nome, 100, stdin);
    nome[strlen(nome)-1]='\0';

    TipoAluno *p;
    p = malloc ( sizeof(TipoAluno) );

    p->num = num;
    strcpy ( p->nome, nome );
    p->nota = -1;

    if ( pauta == NULL ) {
        pauta = p;
        p->prox = NULL;
    }
    else {
        TipoAluno *pa =pauta;
        pauta = p;
        p->prox = pa;
    }
}

int listar() {
    TipoAluno *p = pauta;
    while ( p ) {
        printf ("%d %s %d\n", p->num, p->nome, p->nota );
        p = p->prox;
    }
}

int inserir_aluno () {
    TipoAluno *p;
    p = malloc ( sizeof(TipoAluno) );

    printf ("Nº aluno: " );
    scanf ("%d", &(p->num) );

    limpar_linha();
    printf ("Nome: " );
    fgets( p->nome, 100, stdin );
    nome[strlen(p->nome)-1]='\0';

    p->nota = -1;
    p->prox=NULL;

    if ( pauta == NULL ) {
        pauta = p;
    }
    else {
        TipoAluno *pa = pauta;
        while ( pa->prox != NULL )

```

```

        pa = pa ->prox;
        pa->prox = p;
    }
}

int ler() {
    FILE *f;
    f = fopen ("pauta.txt", "r" );
    if ( f == NULL ) return;
    printf ("Carregar ficheiro...");

    int num, nota;
    char nome[100];

    TipoAluno *pa = pauta, *p;
    while ( fscanf ( f, "%d", &num ) != EOF ) {
        fgets ( nome, 100, f );
        nome[strlen(nome)-1] = 0;
        fscanf ( f, "%d", &nota );

        p = malloc( sizeof(TipoAluno) );
        p->num = num;
        strcpy ( p->nome, nome );
        p->nota = nota;
        p->prox = NULL;

        if ( pauta == NULL ) {
            pauta = p;
        }
        else
            pa->prox=p;
        pa=p;
    }
    fclose(f);
    printf ("\n");
}

```

10

Editor de texto: vi

Trabalhar remotamente no servidor implica, na maior parte dos casos, poder editar remotamente ficheiros. O Linux dispõe de um conjunto de editores em modo texto muito evoluídos. A título de exemplo podem referir-se o emacs, o vi, o joe, o nano, o pico, etc. Nesta disciplina será usado preferencialmente o vi (lê-se “vê i”).

10.1 Elementos básicos

10.1.1 Entrada

Para entrar no vi dá-se o comando:

```
vi ficheiro
```

Se o ficheiro indicado já existir aparece o respectivo conteúdo; senão o ecrã aparece vazio.

Observação. Pode acontecer que parte do ecrã apareça preenchida com o sinal ~. Este sinal denota uma linha vazia. O vi preenche com este símbolo as linhas que se situam entre o fim do conteúdo do ficheiro e o fundo do ecrã. Evidentemente que se o ficheiro estiver vazio todas as linhas serão preenchidas desta forma.

10.1.2 Edição de Texto

O vi funciona em dois modos: *comando* e *texto*. No modo *comando* os caracteres dados pelo utilizador são interpretados como comandos. No modo *texto* os caracteres dados pelo utilizador são tomados como texto a escrever no ficheiro.

Ao entrar o vi fica no modo *comando*, para passar ao modo *texto* pode-se dar o comando i (isto é escrever a letra i); outras hipóteses são a, o, I, etc. Para regressar ao modo comando usa-se a tecla <Esc>. No modo texto funcionam os mecanismos habituais de escrita: carrega em <enter> para mudar de linha, em <backspace> para apagar, etc.

Exercício. experimente a sequência tecla i (passar ao modo texto), escrever texto, tecla <ESC> (regressar ao modo comando) repetidas vezes. As setas funcionam para se movimentar no texto.

10.1.3 Conjunto de comandos

Este pode ser um conjunto mínimo de comandos para editar ficheiros com o vi:

i a	passar ao modo texto
j k h l	movimento no texto
x dd	apagar um caracter / apagar uma linha completa
J	juntar duas linhas
:wq ZZ	guardar o ficheiro e sair
q!	sair sem guardar

Os comandos **i a** e **A** todos passam ao modo texto, diferido apenas o local onde o cursor de inserção de texto fica posicionado: no caso do **i** é a posição anterior; no caso do **a** a posição seguinte; no caso do **A** a posição a seguir ao fim da linha. Outras hipóteses são **A**, **I**, **o**, **O**. Os comandos **j k h l** movimentam o cursor sobre o texto. Serão pouco usados porque em geral as setas funcionam para o mesmo efeito. As teclas **^b** e **^f** (quer dizer Control-f e Control-B) movimentam página a página (identicamente poucas usadas em desfavor de PgDown e PgUp).

O comando **J** apaga um fim de linha, ou seja, faz com que a linha seguinte se junte àquela onde está o cursor.

Exemplo. Ao escrever, por engano, dá um <enter> e corta uma linha a meio. Solução: vai ao modo comando, posiciona-se na primeira "metade" da linha e dá o comando **J** para lhe juntar a outra "metade".

O comando **:wq** guarda o ficheiro em edição e sai do vi. O comando **:q!** sai do vi sem guardar o ficheiro. Os comandos como este, iniciados com o caracter **:**, aparecem na última linha do ecrã do vi. Depois de **:** o cursor vai para a última linha e o resto do comando vai aparecendo nessa linha; para terminar o comando carrega em <enter>.

10.1.4 Sair e gravar

O comando **:w** guarda o ficheiro (apenas; não sai do vi). Para sair pode-se usar apenas **:q**. Mas, caso não tenha gravado as últimas alterações aparecerá uma mensagem de erro. Para forçar a saída sem guardar dá-se o comando **:q!**. Estes comandos terminam com <enter>.

10.2 Mecanismos comuns de edição

10.2.1 Repetição de comandos

Muitos comandos podem ser antecidos de um número e, nesse caso, serão repetidos o número de vezes indicado. Por exemplo **5x** repete 5 vezes o comando **x**, ou seja, apaga 5 caracteres (seguidos); da mesma forma o comando **5dd** apaga 5 linhas.

Exemplo. experimente escrever **5**, depois **i** para passar ao modo texto, escrever alguns caracteres e regressar ao modo comando com <esc>. O efeito é repetir a inserção 5 vezes.

10.2.2 Alteração de texto

O comando **R** permite alterar ("escrever por cima") um texto já existente. O comando passa o modo texto: começa a substituir o texto no ponto inicial e termina quando se der o <esc>, voltando ao modo comando. O comando **r** substitui apenas um caracter. Neste caso não vai para modo texto: depois de dado o caracter de substituição permanece no modo comando.

Exemplo. Imagine, por exemplo, que quer substituir um caracter por **x**. Escreve **rx**, ou seja o comando **r** e **x** o caracter de substituição. Note que continua em modo *comando*. Se fizer **3rx** substitui três caracteres por **x**.

O comando **cw** permite substituir uma palavra (os caracteres até ao próximo separador: espaço, enter, etc). Neste caso o vi assinala com o símbolo **\$** o âmbito da substituição: até esse ponto altera; a partir daí insere novo texto. O comando passa ao modo texto, terminando com **<esc>**.

Exemplo. Imagine, por exemplo, que tem num texto a palavra hipotótamo. Tendo o cursor da letra **p** e fazendo o comando **cw** vai substituir os caracteres potótamo. Se escrever **t<esc>** fica com a palavra **hit**; se escrever **lário<esc>** fica com a palavra **hilário**.

O comando **s** é útil na forma **ns** permitindo substituir o número de caracteres indicado. Por exemplo fazendo **3s** aparece o símbolo **\$** indicando a substituição de 3 caracteres; a partir daí é como no comando anterior.

10.2.3 Procura. Posicionamento

Para localizar determinado texto no conteúdo do ficheiro usa-se o comando **/texto**. Por exemplo **/main** procura a palavra main no ficheiro, ficando o cursor posicionado na primeira ocorrência que for encontrada (caso exista, evidentemente). O comando **/** aparece também na última linha: depois de escrita a **/** indica-se o texto a procurar terminando com **<enter>**.

Na sequência de uma primeira procura pode-se continuar com os comandos **n** que procura o mesmo texto seguindo para a frente no ficheiro ou **?** ou **N** que procuram o mesmo texto seguindo para trás no ficheiro.

Outras formas de posicionamento comuns são **:n** que posiciona o cursor na linha **n** do ficheiro, **g** que posiciona no início do ficheiro e **G** que posiciona no fim do ficheiro.

10.2.4 Substituição de texto

O comando **:%s/org/novo/g** permite substituir texto, indicando-se o texto original org a procurar no ficheiro e o texto novo que o irá substituir. Por exemplo **:%s/ola/ugue/g** substitui todas as ocorrências do texto "ola" pelo texto "ugue".

10.2.5 Copiar e colar

Se fizer **dd** a linha apagada não desaparece definitivamente; é guardada num buffer de serviço podendo ser recuperada. Para repôr a linha dá-se o comando **p** ou **P** (p repõe abaixo da linha corrente, P acima da linha corrente).

Observação. Um mecanismo simples de duplicar uma linha é **ddPP**: o **dd** apaga a linha salvaguardado-a no buffer de serviço; cada **P** repõe a linha uma vez. Esta seria digamos uma sequência "cut/paste/paste".

O comando **Y** copia uma linha para o buffer de serviço, mas sem a apagar. Assim, por exemplo, para copiar uma linha pode fazer com **Y** e **P** a sequência típica de "copy/paste": com **Y** copia a linha original; posiciona-se no sítio onde a pretende replicar; com **P** insere-a nessa posição. Se fizer a sequência **YP** duplica a linha no mesmo sítio.

O comando **nY** salvaguarda **n** linhas no buffer e é portanto útil para copiar várias linhas. Por exemplo, para copiar 5 linhas faz-se **5Y**, posiciona-se o cursor no local de destino e inserem-se as linhas salvaguardadas com **P**.

Além do buffer de serviço usado, por defeito pelo Y e P, podem ser usados outros buffer's para efeito semelhante. Os outros buffer's são designados pelo nome "x sendo x uma letra - por exemplo "a ou "b.

Os comandos de "copiar e colar" podem ser anteceditos do nome de um buffer específico e nesse caso farão a operação com esse buffer. Por exemplo: **"a5Y** salvaguarda 5 linha de texto no buffer **"a**. Estas linhas poderão mais tarde ser repostas com **"aP**.

10.2.6 Ficheiros

Para gravar o ficheiro em curso de edição usa-se o comando **:w**. Para sair do vi usa-se o comando **:q**. Os dois podem-se juntar na sequência **:wq**, já vista, que grava e sai.

O comando **:q** dá uma mensagem de erro e não resulta se houver alterações não gravadas. Para sair sem gravar usa-se a variante **:q!** que força a saída sem gravar.

No comando **:w** pode-se explicitar o nome do ficheiro (confirmando ou alterando o nome dado na entrada para o vi). Por exemplo o comando **:w poema.txt** salva a edição em curso no ficheiro **poema.txt**.

10.2.7 Troca de texto entre ficheiros

O comando **:r** permite importar o conteúdo de um ficheiro juntando-o à edição em curso. Por exemplo, o comando **:r /etc/passwd** insere na posição do cursor o conteúdo do ficheiro designado.

É possível também, com pouco esforço, manter dois ficheiros em edição "simultânea". Para abrir um segundo ficheiro dá-se o comando **:e** ficheiro (o comando só resulta depois de salvar as alterações ao original). Por exemplo dando o comando **:e /etc/passwd** passará a editar em simultâneo o ficheiro original e o ficheiro **/etc/passwd** (este último sem poder gravar as alterações, como é evidente). A partir daqui pode trocar entre os dois ficheiros em edição com o comando **:e#**.

Tendo os dois ficheiros abertos para edição pode também copiar blocos de texto entre eles. A ideia geral é a mesma que para fazer a cópia dentro do mesmo ficheiro: primeiro dá-se o comando **Y** para copiar depois o comando **P** para colar. Neste caso o processo será então: salvaguardar um conjunto de linhas num ficheiro; mudar de ficheiro com **:e**, posicionar o cursor no local de destino e inserir o texto com **P**.

10.2.8 Outros Comandos

O comando **u** faz *undo* (anula o efeito da última alteração). O comando **U** anula todas as alterações na linha. o comando **:redo** refaz o último *undo*

10.2.9 Personalização do vi

O vi dispõe de várias opções de personalização que incluem opções como mostrar os números das linhas, formatar automaticamente o texto etc. É possível cada utilizador alterar as definições gerais do sistema. Para isso basta editar o ficheiro **.vimrc**. Para definir uma determinada opção usa-se o comando **set**. Algumas opções interessantes são:

```
set number
```

mostra os números das linhas ao lado do código

```
set nonumber
```

não mostra os números das linhas ao lado do código

```
set syntax on
```

faz syntax highlight