Miguel Pimenta Monteiro

O Sistema Operativo Unix

Alguns aspectos da sua API

(*Application Programming Interface*)

1. Programas em C no Unix

1.1 Início e terminação de programas

1.1.1 Início

Quando se solicita ao S.O. a execução de um novo programa (serviço `exec()` no UNIX) este começa por executar uma rotina (no caso de programas em C) designada por *C startup*. Esta rotina é a responsável por chamar a função `main()` do programa, passando-lhe alguns parâmetros, se for caso disso, e por abrir e disponibilizar três "ficheiros" ao programa: os chamados *standard input*, *standard output* e *standard error*. O *standard input* fica normalmente associado ao teclado (excepto no caso de redireccionamento), enquanto que o *standard output* e o *standard error* ficam normalmente associados ao écran (também podem ser redireccionados).

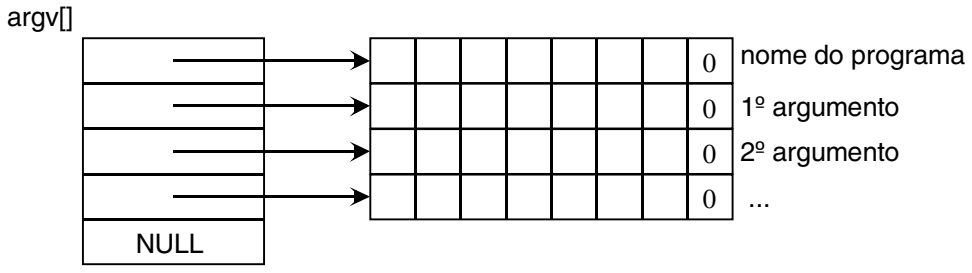
A função `main()` pode ser definida num programa em C de muitas formas:

```
int ou void main(void)
int ou void main(int argc)
int ou void main(int argc, char *argv[ ])
int ou void main(int argc, char *argv[ ], char *envp[ ])
```

Assim pode ser definida como procedimento (`void`) ou como função retornando um inteiro (`int`). Neste último caso o inteiro retornado é passado a quem chamou a função, ou seja à rotina de *C startup*, que por sua vez o passa ao S.O.

Quando se invoca um programa é possível passar-lhe parâmetros, que são um conjunto de 0 ou mais strings, separadas por espaço(s). Esses parâmetros podem depois ser acedidos em `main()` através dos argumentos `argc` e `argv`.

`argc` - número de argumentos passados, incluindo o próprio nome do programa.
`argv` - array de apontadores para string, apontando para os parâmetros passados ao programa. O array contém um número de elementos igual a `argc`+1. O primeiro elemento de `argv[]` aponta sempre para o nome do programa (geralmente incluindo todo o *path*). O último elemento de `argv[]` contém sempre o apontador nulo (valor `NULL`).



`envp` - array de apontadores para string, apontando para as variáveis de ambiente do programa. Todos os sistemas permitem a definição de variáveis de ambiente da forma `NOME=string;` Cada um dos elementos de `envp[]` aponta para uma string daquela forma (incluindo o `NOME=`). O array contém um número de elementos igual ao número de variáveis de ambiente + 1. O último elemento de `envp[]` contém sempre o apontador nulo (valor `NULL`). A estrutura de `envp[]` é semelhante à de `argv[]`.

Em rigor o parâmetro **envp** da função `main()` não está padronizado (não pertence à definição do ANSI C), mas é implementado em numerosos S.O.s incluindo o UNIX.

1.1.2 Terminação

Um programa em C termina quando a função `main()` retorna (usando `return` *expressão*, no caso de ter sido definida como `int`, e usando simplesmente `return` ou deixando chegar ao fim das instruções, no caso de ter sido definida como `void`). Outra possibilidade é chamar directamente funções terminadoras do programa que são:

```
#include <stdlib.h>

void exit(int status);
```

Termina imediatamente o programa, retornando para o sistema operativo o código de terminação **status**. Além disso executa uma libertação de todos os recursos alocados ao programa, fechando todos os ficheiros e guardando dados que ainda não tivessem sido transferidos para o disco.

```
#include <unistd.h>

void _exit(int status);
```

Termina imediatamente o programa, retornando para o sistema operativo o código de terminação **status**. Além disso executa uma libertação de todos os recursos alocados ao programa de forma rápida, podendo perder dados que ainda não tivessem sido transferidos para o disco.

Quando o programa termina pelo retorno da função `main()` o controlo passa para a rotina de *C startup* (que chamou `main()`); esta por sua vez acaba por chamar `exit()` e esta chama no seu final `_exit()`.

A função `exit()` pode executar, antes de terminar, uma série de rotinas (*handlers*) que tenham sido previamente registadas para execução no final do programa. Estas rotinas são executadas por ordem inversa do seu registo.

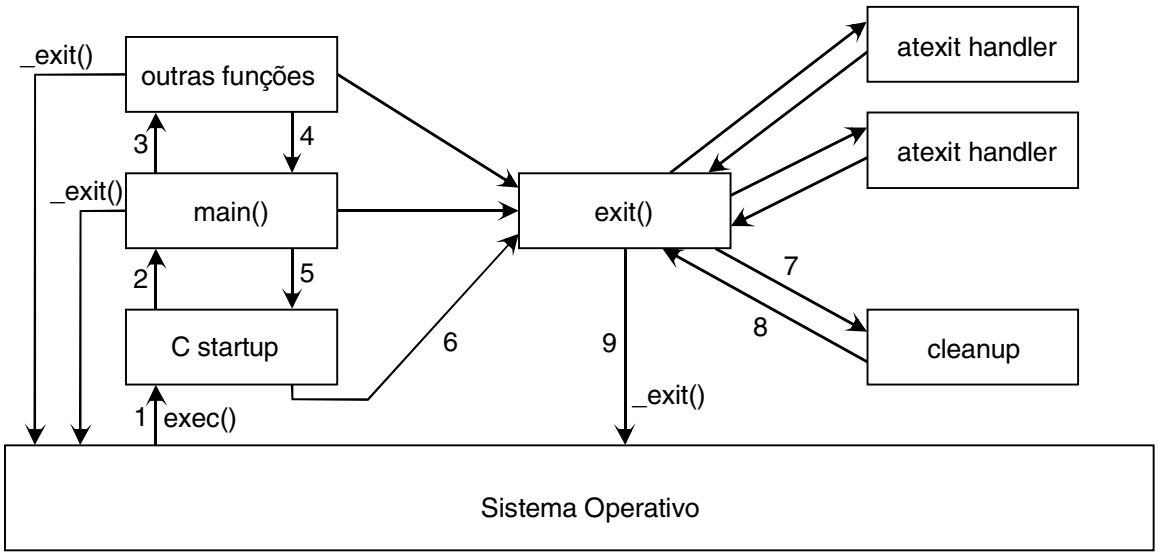
O registo destes *handlers* de terminação é feito por:

```
#include <stdlib.h>

int atexit(void (*func)(void));
```

Regista o *handler* de terminação **func** (função `void` sem parâmetros). Retorna 0 em caso de sucesso e um valor diferente de 0 em caso de erro.

Na figura seguinte pode ver-se um esquema dos processos de início e terminação de um programa em C.

As funções `exit()` e `_exit()` podem ser vistas como serviços do sistema operativo. De facto correspondem de perto a chamadas directas ao Sistema Operativo (p. ex. `ExitProcess()` e `TerminateProcess()` no caso do Windows NT).



1 .. 9 - Percurso mais frequente

1.2 Processamento dos erros

Grande parte dos serviços dos Sistemas Operativos retornam informação acerca do seu sucesso ou da ocorrência de algum erro que o impediu de executar o que lhe era pedido. No entanto, e em geral, essa informação apenas diz se ocorreu ou não um erro, sem especificar o tipo de erro ou a sua causa.

Para se extrair mais informação relativa ao último erro ocorrido é necessário utilizar outros mecanismos próprios de cada sistema operativo.

A maior parte dos serviços do Unix apenas indica se ocorreu ou não um erro, não especificando o tipo de erro. Esse tipo de erro é colocado, através de um código, numa variável global chamada **`errno`** e que é de tipo inteiro (`int`). No entanto essa variável só contém o código válido imediatamente após o retorno do serviço que causou o erro. Qualquer chamada a outro serviço ou função da biblioteca standard do C pode alterar o valor de **`errno`**. Associadas aos códigos possíveis colocados em **`errno`** existem também constantes simbólicas definidas no ficheiro de inclusão `errno.h`. Alguns exemplos dessas constantes são:

```
ENOMEM
EINVAL
ENOENT
```

No Unix é possível obter e imprimir uma descrição mais detalhada correspondente a cada código de erro. Para imprimir directamente essa descrição pode usar-se o serviço `perror()`:

`#include <stdio.h>` ou **`<stdlib.h>`**

`void perror(const char *string);`

Imprime na consola (em *`standard error`*) a string passada no argumento **`string`**, seguida do sinal de dois pontos (:), seguida da descrição correspondente ao código que nesse momento se encontra em `errno`. A linha é terminada com `'\n'`.

Para obter uma descrição colocada numa string deverá usar-se o serviço `strerror()`:

```
#include <string.h>

char *strerror(int errnum);
```

Retorna um apontador para string contendo uma descrição do erro cujo código foi passado no argumento **`errnum`**. A string retornada é alocada internamente, mas não precisa ser libertada explicitamente. Novas chamadas a `strerror()` reutilizam esse espaço.

Pode ver-se de seguida um exemplo da utilização de `perror()`:

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

void main(void)
{
  char *mycwd;

  if ((mycwd = (char *) malloc(129)) == NULL) { /* alocação de 129 bytes */
    perror("malloc:");
    exit(1);
  }
  if (getcwd(mycwd, 128) == NULL) {   /* obtenção do directório corrente */
    perror("getcwd:");
    exit(1);
  }
  printf("Current working directory: %s\n", mycwd);
  free(mycwd);
}
```

1.3 Medida de tempos de execução

Um programa pode medir o próprio tempo de execução e de utilização do processador através de um serviço do S.O. especialmente vocacionado para isso. Trata-se do serviço `times()`.

```
#include <sys/tymes.h>

clock_t times(struct tms *buf);
```

Preenche a estrutura cujo endereço se fornece em **`buf`** com informação acerca dos tempos de execução do processo.
Retorna o tempo actual do sistema (relógio), medido a partir do arranque.

O serviço anterior preenche uma estrutura que contém 4 campos e é definida em `<sys/times.h>` da seguinte forma:

```
struct tms {
  clock_t tms_utime;   /* tempo de CPU gasto em código do processo */
  clock_t tms_stime;   /* tempo de CPU gasto em código do sistema
                          chamado pelo processo */
  clock_t tms_cutime;  /* tempo de CPU dos filhos (código próprio) */
  clock_t tms_cstime;  /* tempo de CPU dos filhos (código do sistema) */
}
```

Todos os tempos são medidos em *clock ticks*. Cada sistema define o número de *ticks* por segundo, que pode ser consultado através do serviço `sysconf()`. Este último serviço (ver `man`) fornece o valor de inúmeras constantes dependentes da implementação. A que aqui nos interessa é designada por `_SC_CLK_TCK`.

O tempo retornado por `times()` é relativo a um instante arbitrário anterior (o tempo de arranque do sistema) sendo por isso necessário fazer uma chamada no início do programa e outra perto do final. O tempo total decorrido será a diferença entre essas duas medições.

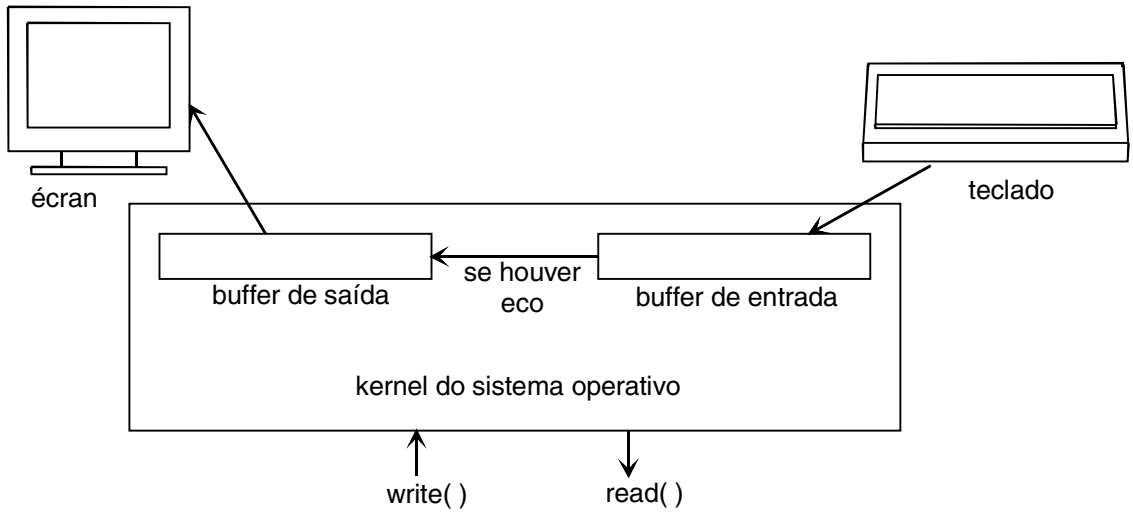
Apresenta-se de seguida um exemplo do método de medida de tempos.

```
#include <sys/times.h>
#include <unistd.h>
#include <stdio.h>

void main(void)
{
  clock_t start, end;
  struct tms t;
  long ticks;
  int k;

  start = times(&t);                          /* início da medição de tempo */
  ticks = sysconf(_SC_CLK_TCK);

  for (k=0; k<100000; k++)
    printf("Hello world!\n");

  printf("\nClock ticks per second: %ld\n", ticks);

  end = times(&t);                              /* fim da medição de tempo */

  printf("Clock:                %4.2f s\n", (double)(end-start)/ticks);
  printf("User time:            %4.2f s\n", (double)t.tms_utime/ticks);
  printf("System time:          %4.2f s\n", (double)t.tms_stime/ticks);
  printf("Children user time:   %4.2f s\n", (double)t.tms_cutime/ticks);
  printf("Children system time: %4.2f s\n", (double)t.tms_cstime/ticks);
}
```

2. Consola, ficheiros e directórios

2.1. Entrada/Saída na consola

A consola (teclado + écran) é vista geralmente nos S.O.'s como um ou mais ficheiros onde se pode ler ou escrever texto. Esses ficheiros são normalmente abertos pela rotina de *C startup*. A biblioteca standard do C inclui inúmeras funções de leitura e escrita directa nesses ficheiros (`printf()`, `scanf()`, `getchar()`, `putchar()`, `gets()`, `puts()`, ...).
No entanto podemos aceder também a esses periféricos através dos serviços dos S.O.'s.



Implementação típica da consola num sistema operativo

Em Unix os ficheiros são acedidos através de descritores, que são números inteiros, geralmente pequenos. A consola tem 3 componentes, conhecidos por *standard input* (teclado), *standard output* (écran) e *standard error* (também écran) que são representados por ficheiros com os descritores 0, 1 e 2, a que correspondem as constantes `STDIN_FILENO`, `STDOUT_FILENO` e `STDERR_FILENO`, definidas em `<unistd.h>`. No início da execução de todos os programas estes componentes encontram-se já abertos e prontos a utilizar.

Na API (*Application Programming Interface*) do Unix não existem funções específicas de leitura e escrita nos componentes da consola, sendo necessário usar as funções genéricas de leitura e escrita em ficheiros (descritas mais à frente), ou as já referidas funções da biblioteca standard do C.

No entanto os componentes da consola podem ser redireccionados para ficheiros em disco utilizando os símbolos da *shell* (`>`, `<`, `>>` e `<<`) na linha de comandos desta (por exemplo: **`ls -la > dir.txt`**), ou então programaticamente, utilizando o seguinte serviço:

```
#include <unistd.h>

int dup2(int filedes, int filedes2);
```

Identifica o ficheiro **`filedes`** com o descritor **`filedes2`**, ou seja copia a referência ao ficheiro identificado por **`filedes`** para a posição **`filedes2`**, de modo que quando se usa **`filedes2`** passa-se a referenciar o ficheiro associado a **`filedes`**. Se **`filedes2`** estiver aberto, é fechado antes da identificação. Retorna **`filedes2`** ou -1 se ocorrer um erro.

Assim, por exemplo, para redireccionar a saída standard (écran) para um outro ficheiro com descritor `fd` basta chamar:

```
dup2(fd, STDOUT_FILENO);
```

A partir deste momento qualquer escrita em `STDOUT_FILENO`, neste programa, passa a ser feita no ficheiro representado pelo descritor `fd`.

2.2. Características de funcionamento da consola

As consolas em Unix podem funcionar em dois modos distintos: canónico e primário (*raw*). No modo canónico existe uma série de caracteres especiais de entrada que são processados pela consola e não são transmitidos ao programa que a está a ler. São exemplos desses caracteres: <control-U> para apagar a linha actual, <control-H> para apagar o último carácter escrito, <control-S> para suspender a saída no écran e <control-Q> para a retomar. Muitos destes caracteres especiais podem ser alterados programaticamente. Além disso, em modo canónico, a entrada só é passada ao programa linha a linha (quando se tecla <return>) e não carácter a carácter. No modo primário não há qualquer processamento prévio dos caracteres teclados podendo estes, quaisquer que sejam, ser passados um a um ao programa que está a ler a consola.

Entre o funcionamento dos dois modos é possível programar, através de numerosas opções, algumas não padronizadas, muitos comportamentos diferentes em que alguns caracteres de entrada são processados pela própria consola e outros não.

As características das consolas em Unix podem ser lidas e alteradas programaticamente utilizando os serviços `tcgetattr()` e `tcsetattr()`:

```
#include <termios.h>

int tcgetattr(int filedes, struct termios *termptr);
int tcsetattr(int filedes, int opt, const struct termios *termptr);
```

O serviço `tcgetattr()` preenche uma estrutura `termios`, cujo endereço é passado em **`termptr`**, com as características do componente da consola cujo descritor é **`filedes`**. O serviço `tcsetattr()` modifica as características da consola **`filedes`** com os valores previamente colocados numa estrutura `termios`, cujo endereço é passado em **`termptr`**. O parâmetro **`opt`** indica quando a modificação irá ocorrer e pode ser uma das seguintes constantes definidas em `termios.h`:

- `TCSANOW` - A modificação é feita imediatamente
- `TCSADRAIN` - A modificação é feita depois de se esgotar o buffer de saída
- `TCSAFLUSH` - A modificação é feita depois de se esgotar o buffer de saída; quando isso acontece o buffer de entrada é esvaziado (*flushed*).

Retornam -1 no caso de erro.

A estrutura `termios` inclui a especificação de numerosas *flags* e opções. A sua definição (que se encontra em `termios.h`) é a seguinte:

```
struct termios {
    tcflag_t  c_iflag;          /* input flags */
    tcflag_t  c_oflag;          /* output flags */
    tcflag_t  c_cflag;          /* control flags */
    tcflag_t  c_lflag;          /* local flags */
```

```
    cc_t      c_cc[NCCS];          /* control characters */
}
```

Os quatros primeiros campos da estrutura `termios` são compostos por *flags* de 1 ou mais bits, em que cada uma delas é representada por um nome simbólico definido em `termios.h`. Quando pretendemos activar mais do que uma *flag* de um determinado campo simplesmente devemos efectuar o *or* bit a bit (`|`) entre os seus nomes simbólicos e atribuir o resultado ao campo a que pertencem, como se pode ver no seguinte exemplo:

```
struct termios tms;

tms.c_iflag = (IGNBRK | IGNCR | IGNPAR);
```

No entanto, e mais frequentemente, pretendemos apenas activar ou desactivar uma determinada *flag* de um dos campos de `termios` sem alterar as outras. Isso pode fazer-se com a operação *or* ou com a operação *and* (`&`) e a negação (`~`). Exemplos:

```
tcgetattr(STDIN_FILENO, &tms);  /* lê características actuais da consola */
tms.c_oflag |= (IXON | IXOFF);       /* activa Xon/Xoff */
tms.c_cflag &= ~(PARENB | CSTOPB);   /* desactiva paridade e 2 stop bits */
tcsetattr(STDIN_FILENO, TCSANOW, &tms); /* escreve novas características */
```

O último campo de `termios` é um array (`c_cc`) onde se definem os caracteres especiais que são pré-processados pela consola se esta estiver a funcionar no modo canónico. O tipo `cc_t` é habitualmente igual a `unsigned char`. Cada uma das posições do array é acedida por uma constante simbólica, também definida em `termios.h`, e corresponde a um determinado carácter especial. Por exemplo para definir o carácter que suspende o processo de *foreground* poderá usar-se:

```
tms.c_cc[VSUSP] = 26;     /* 26 - código de <control-Z> */
```

Na tabela seguinte podem ver-se os caracteres especiais e respectivas constantes simbólicas, definidos na norma POSIX.1.

Caracteres especiais	Constante de `c_cc[]`	Flag activadora		Valor típico
		Campo	Flag	
Fim de ficheiro	`VEOF`	`c_lflag`	`ICANON`	ctrl-D
Fim de linha	`VEOL`	`c_lflag`	`ICANON`	
Backspace	`VERASE`	`c_lflag`	`ICANON`	ctrl-H
Envia sinal de interrupção (SIGINT)	`VINTR`	`c_lflag`	`ISIG`	ctrl-C
Apaga a linha actual	`VKILL`	`c_lflag`	`ICANON`	ctrl-U
Envia sinal de quit (SIGQUIT)	`VQUIT`	`c_lflag`	`ISIG`	ctrl-\
Retoma a escrita no écran	`VSTART`	`c_iflag`	`IXON/IXOFF`	ctrl-Q
Para a escrita no écran	`VSTOP`	`c_iflag`	`IXON/IXOFF`	ctrl-S
Envia o sinal de suspenção (SIGTSTP)	`VSUSP`	`c_lflag`	`ISIG`	ctrl-Z

Listam-se agora as principais *flags* definidas na norma POSIX.1:

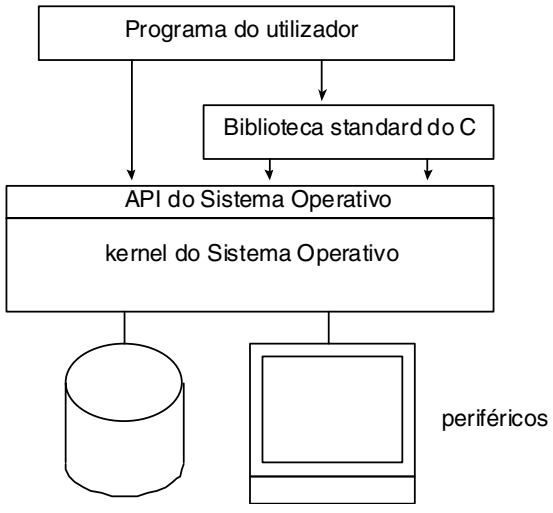
Campo	**Constante**	**Descrição**
`c_iflag`	BRKINT	Gera o sinal SIGINT na situação de BREAK
	ICRNL	Mapeia CR (return) em NL (newline) no buffer de entrada
	IGNBRK	Ignora situação de BREAK
	IGNCR	Ignora CR
	IGNPAR	Ignora caracteres com erro de paridade (terminais série)

	INLCR	Mapeia NL em CR no buffer de entrada
	INPCK	Verifica a paridade dos caracteres de entrada
	ISTRIP	Retira o 8º bit dos caracteres de entrada
	IXOFF	Permite parar/retomar o fluxo de entrada
	IXON	Permite parar/retomar o fluxo de saída
	PARMRK	Marca os caracteres com erro de paridade
`c_oflag`	OPOST	Executa pré-processamento de saída especificado noutras flags deste campo mas que são dependentes da implementação
`c_cflag`	CLOCAL	Ignora as linhas série de estado (RS-232)
	CREAD	Permite a leitura de caracteres (linha RD RS-232)
	CS5/6/7/8	5,6,7 ou 8 bits por carácter
	CSTOPB	2 stop bits (1 se desactivada)
	HUPCL	Desliga ligação quando do fecho do descriptor (RS-232, modem)
	PARENB	Permite o bit de paridade
	PARODD	Paridade ímpar (par se desactivada)
`c_lflag`	ECHO	Executa o eco dos caracteres recebidos para o buffer de saída
	ECHOE	Apaga os caracteres visualmente (carácter de backspace - VERASE)
	ECHOK	Apaga as linhas visualmente (carácter VKILL)
	ECHONL	Ecoa o carácter de NL mesmo se ECHO estiver desactivado
	ICANON	Modo canónico (processa caracteres especiais)
	IEXTEN	Processa caracteres especiais extra (dependentes da implementação)
	ISIG	Permite o envio de sinais a partir da entrada
	NOFLSH	Desactiva a operação de flush depois de interrupção ou quit
	TOSTOP	Envia o sinal SIGTTOU a um processo em background que tente escrever na consola.

Para mais pormenores sobre outras *flags* e características dependentes da implementação consultar o manual através do comando `man` do Unix.

2.3. Serviços de acesso a ficheiros

A biblioteca standard da linguagem C (*ANSI C library*) contém um conjunto rico de funções de acesso, criação, manipulação, leitura e escrita de ficheiros em disco e dos 3 componentes da consola. A utilização dessas funções torna os programas independentes do Sistema Operativo, uma vez que a biblioteca standard é a mesma para todos os S.O.'s. No entanto introduz uma camada extra entre o programa e o acesso aos objectos do S.O. (ficheiros, consola, etc) como se ilustra na figura seguinte.



A posição da biblioteca standard do C

Assim, para se conseguir um pouco mais de performance, é possível chamar directamente os serviços dos S.O.'s, pagando-se o preço dos programas assim desenvolvidos não poderem ser compilados, sem alterações, noutros Sistemas Operativos.

Apresentam-se de seguida alguns dos serviços referentes à criação, leitura e escrita de ficheiros no Sistema Operativo Unix, conforme especificados pela norma POSIX.1.

Utiliza-se o serviço `open()` para a abertura ou criação de novos ficheiros no disco. Este serviço retorna um descritor de ficheiro, que é um número inteiro pequeno. Esses descritores são depois usados como parâmetros nos outros serviços de acesso aos ficheiros. Os ficheiros em Unix não são estruturados, sendo, para o Sistema Operativo, considerados apenas como meras sequências de *bytes*. São os programas que têm de interpretar correctamente o seu conteúdo.

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

int open(const char * pathname, int oflag, …);  /* mode_t mode */
```

`pathname` – nome do ficheiro;
`oflag` – combinação de várias *flags* de abertura:
uma e só uma das três constantes seguintes deve estar presente:

- `O_RDONLY` – abertura para leitura
- `O_WRONLY` – abertura para escrita
- `O_RDWR` – abertura para leitura e escrita

Outras *flags*:

- `O_APPEND` – acrescenta ao fim do ficheiro
- `O_CREAT` – Cria o ficheiro se não existe; requer o parâmetro **`mode`**
- `O_EXCL` – Origina um erro se o ficheiro existir e se `O_CREAT` estiver presente
- `O_TRUNC` – Coloca o comprimento do ficheiro em 0, mesmo se já existir
- `O_SYNC` – as operações de escrita só retornam depois dos dados terem sido fisicamente transferidos para o disco

`mode` – permissões associadas ao ficheiro; só faz sentido para a criação de novos ficheiros; pode ser um *or* bit a bit (`|`) entre as seguintes constantes:
`S_IRUSR` - user read
`S_IWUSR` - user write
`S_IXUSR` - user execute
`S_IRGRP` - group read
`S_IWGRP` - group write
`S_IXGRP` - group execute
`S_IROTH` - others read
`S_IWOTH` - others write
`S_IXOTH` - others execute
A função retorna um descritor de ficheiros ou –1 no caso de erro.

Como se pode ver na descrição acima, quando se cria um novo ficheiro é necessário especificar as suas permissões no parâmetro **`mode`** de `open()`. Além disso, para se criar um novo ficheiro, é necessário possuir as permissões de *write* e *execute* no directório onde se pretende criá-lo. Se o ficheiro for efectivamente criado, as permissões com que fica podem não ser exactamente as que foram especificadas. Antes da criação, as permissões

especificadas são sujeitas à operação de *and* lógico com a negação da chamada máscara do utilizador (*user mask* ou simplemente *umask*). Essa máscara contém os bits de permissão que não poderão ser nunca colocados nos novos ficheiros ou directórios criados pelo utilizador. Essa máscara é geralmente definida no *script* de inicialização da *shell* de cada utilizador e um valor habitual para ela é o valor 0002 (octal), que proíbe a criação de ficheiros e directórios com a permissão de *write* para outros utilizadores (*others write*).

A máscara do utilizador pode ser modificada na *shell* com o comando `umask` ou programaticamente com o serviço do mesmo nome.

```
#include <sys/types.h>
#include <sys/stat.h>

mode_t umask(mode_t cmask);
```

Modifica a máscara de criação de ficheiros e directórios para o valor especificado em **`cmask`**; este valor pode ser construído pelo *or* lógico das constantes cujos nomes aparecem na descrição do parâmetro `mode` do serviço `open()`.
Retorna o valor anterior da máscara.

A leitura ou escrita em ficheiros faz-se directamente com os serviços `read()` e `write()`. Os *bytes* a ler ou escrever são tranferidos para, ou de, *buffers* do próprio programa, cujos endereços se passam a estes serviços. O kernel do S.O. pode também manter os seus próprios *buffers* para a transferência de informação para o disco. Os componentes da consola também podem ser lidos ou escritos através destes serviços.

A leitura ou escrita de dados faz-se sempre da ou para a posição actual do ficheiro. Após cada uma destas operações essa posição é automaticamente actualizada de modo a que a próxima leitura ou escrita se faça exactamente para a posição a seguir ao último byte lido ou escrito. Quando o ficheiro é aberto (com `open()`) esta posição é o início (posição 0) do ficheiro, excepto se a *flag* `O_APPEND` tiver sido especificada.

```
#include <unistd.h>

ssize_t read(int filedes, void *buffer, size_t nbytes);
ssize_t write(int filedes, const void *buffer, size_t nbytes);
```

`nbytes` indica o número de *bytes* a tranferir enquanto que **`buffer`** é o endereço do local que vai receber ou que já contém esses *bytes*.
As funções retornam o número de *bytes* efectivamente transferidos, que pode ser menor que o especificado. O valor –1 indica um erro, enquanto que o valor 0, numa leitura, indica que se atingiu o fim do ficheiro.

Quando se pretende ler da consola (`STDIN_FILENO`) e se utiliza o serviço `read()`, geralmente este só retorna quando se tiver entrado uma linha completa (a menos que a consola se encontre no modo primário). Se o tamanho do *buffer* de leitura for maior do que o número de *bytes* entrados na linha, o serviço `read()` retornará com apenas esses *bytes*, indicando no retorno o número lido. No caso contrário, ficarão alguns *bytes* pendentes no *buffer* interno da consola à espera de um próximo `read()`.

A posição do ficheiro de onde se vai fazer a próxima transferência de dados (leitura ou escrita) pode ser ajustada através do serviço `lseek()`. Assim é possível ler ou escrever partes descontínuas de ficheiros, ou iniciar a leitura ou escrita em qualquer posição do

ficheiro.

```
#include <sys/types.h>
#include <unistd.h>

off_t lseek(int filedes, off_t offset, int whence);
```

Desloca o apontador do ficheiro de **offset** bytes (pode ser negativo ou positivo) com uma origem especificada em **whence**:
SEEK_SET - início do ficheiro
SEEK_CUR - posição corrente do ficheiro
SEEK_END - final do ficheiro
Retorna -1 no caso de erro ou o valor da nova posição do ficheiro.

Se se quiser conhecer a posição actual do ficheiro, isso pode ser feito com a chamada:

```
pos = lseek(fd, 0, SEEK_CUR);
```

Quando não houver mais necessidade de manipular um ficheiro que se abriu anteriormente, este deve ser fechado quanto antes. Os ficheiros abertos fecham-se com o serviço `close()`.

```
#include <unistd.h>

int close(int filedes);
```

Retorna 0 no caso de sucesso e –1 no caso de erro.

Finalmente se quisermos apagar um ficheiro para o qual tenhamos essa permissão podemos utilizar o serviço `unlink()`.

```
#include <unistd.h>

int unlink(const char *pathname);
```

Apaga a referência no directório correspondente ao ficheiro indicado em **pathname** e decrementa a contagem de *links*.
Retorna 0 no caso de sucesso e –1 no caso de erro.

Para se poder apagar um ficheiro é necessário ter as permissões de escrita e execução no directório em que se encontra o ficheiro. Embora a referência ao ficheiro no directório especificado desapareça, o ficheiro não será fisicamente apagado se existirem outros *links* para ele. Os dados do ficheiro poderão ainda ser acedidos através desses *links*. Só quando a contagem de *links* se tornar 0 é que o ficheiro será efectivamente apagado.

Se se apagar (com `unlink()`) um ficheiro aberto, este só desaparecerá do disco quando for fechado.

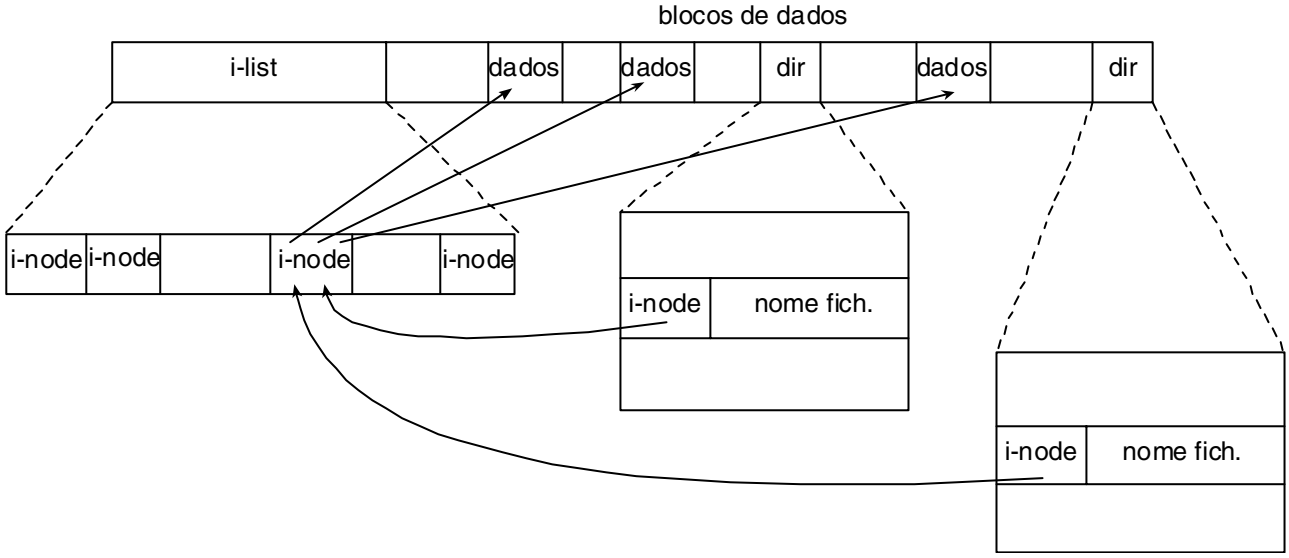
2.4. O sistema de ficheiros em Unix

Basicamente o sistema de ficheiros do Unix é constituído por duas partes distintas: uma lista de i-nodes e uma sequência de blocos de dados (ficheiros e directórios).

A lista de i-nodes contém uma série de pequenas estruturas designadas por i-nodes que contêm toda a informação associada a cada ficheiro ou directório, nomeadamente a data de criação e modificação, tamanho, dono, grupo, permissões, número de links e principalmente quais os blocos, e por que ordem, pertencem ao ficheiro ou directório associado. Se o ficheiro for muito extenso poderá haver necessidade de associar mais que 1 i-node a esse ficheiro (para listar todos os blocos que o compõem).

Os directórios são apenas sequências de entradas em que cada uma tem apenas o nome do ficheiro ou subdirectório contido e o número do i-node que contém o resto da informação.

É possível que um mesmo ficheiro físico (ou directório) esteja presente em vários directórios, constituindo-se assim um conjunto de (*hard*) links para esse ficheiro. Há também ficheiros especiais (*symbolic links*) cujo conteúdo referencia outro ficheiro.



O sistema de ficheiros

2.5. Acesso ao sistema de ficheiros

Já vimos que o sistema de ficheiros pode armazenar vários tipos de informação nos seus blocos de dados. Na breve exposição anterior podemos para já identificar 3 tipos: ficheiros normais, directórios e links simbólicos. No entanto, como veremos mais tarde existem outros, como: os FIFOs, os ficheiros que representam directamente periféricos de entrada/saída (*character special file* e *block special file*) e por vezes *sockets*, semáforos, zonas de memória partilhada e filas de mensagens. Todos estes tipos aparecem pelo menos listados num directório e podem ser nomeados através de um *pathname*.
É possível extrair muita informação, programaticamente, acerca de um objecto que apareça listado num directório, ou que já tenha sido aberto e seja representado por um descritor, através dos seguintes três serviços.

```
#include <sys/types.h>
#include <sys/stat.h>

int stat(const char *pathname, struct stat *buf);
int fstat(int filedes, struct stat *buf);
int lstat(const char *pathname, struct stat *buf);
```

Todos estes serviços preenchem uma estrutura `stat`, cujo endereço é passado em **`buf`**, com informação acerca de um nome que apareça num directório (especificado em **`pathname`**), ou através do seu descritor (**`filedes`**) se já estiver aberto.
O serviço `lstat()` difere de `stat()` apenas no facto de retornar informação acerca de um link simbólico, em vez do ficheiro que ele referencia (que é o caso de `stat()`).
Retornam 0 no caso de sucesso e –1 no caso de erro.

A estrutura `stat` contém numerosos campos e é algo dependente da implementação. Consultar o `man` ou a sua definição em `<sys/stat.h>`.
Alguns campos mais comuns são:

```
struct stat {
  mode_t st_mode;    /* tipo de ficheiro e permissões */
  ino_t st_ino;      /* número do i-node correspondente */
  nlink_t st_nlink;  /* contagem de links */
  uid_t st_uid;      /* identificador do dono */
  gid_t st_gid;      /* identificador do grupo do dono */
  off_t st_size;     /* tamanho em bytes (ficheiros normais) */
  time_t st_atime;   /* último acesso */
  time_t st_mtime;   /* última modificação */
  time_t st_ctime;   /* último mudança de estado */
  long st_blocks;    /* número de blocos alocados */
}
```

O tipo de ficheiro está codificado em alguns dos bits do campo `st_mode`, que também contém os bits de permissão nas 9 posições menos significativas. Felizmente em `<sys/stat.h>` definem-se uma série de macros para fazer o teste do tipo de ficheiro. A essas macros é necessário passar o campo `st_mode` devidamente preenchido, sendo avaliadas como `TRUE` ou `FALSE`.

As macros que geralmente estão aí definidas são as seguintes:

- `S_ISREG()` Testa se se trata de ficheiros regulares
- `S_ISDIR()` Testa se se trata de directórios
- `S_ISCHR()` Testa se se trata de dispositivos orientados ao carácter
- `S_ISBLK()` Testa se se trata de dispositivos orientados ao bloco
- `S_ISFIFO()` Testa se se trata de fifo's
- `S_ISLNK()` Testa se se trata de links simbólicos
- `S_ISSOCK()` Testa se se trata de sockets

No exemplo seguinte podemos ver um pequeno programa que indica o tipo de um ou mais nomes de ficheiros passados na linha de comando.

```
#include <sys/types.h>
#include <sys/stat.h>
#include <stdio.h>

int main(int argc, char *argv[])
{
  int k;
  struct stat buf;
  char *str;

  for (k=1; k<argc; k++) {
    printf("%s: is ", argv[k]);
```

```
    lstat(argv[k], &buf);          /* preenche buf (struct stat) */

    if      (S_ISREG(buf.st_mode))  str = "regular";
    else if (S_ISDIR(buf.st_mode))  str = "directory";
    else if (S_ISCHR(buf.st_mode))  str = "character special";
    else if (S_ISBLK(buf.st_mode))  str = "block special";
    else if (S_ISFIFO(buf.st_mode)) str = "fifo";
    else if (S_ISLNK(buf.st_mode))  str = "symbolic link";
    else if (S_ISSOCK(buf.st_mode)) str = "socket";
    else str = "unknown";

    printf("%s\n", str);
  }
  return 0;
}
```

Como se viu, os directórios do sistema de ficheiros do Unix não passam de ficheiros especiais. Existe uma série de serviços para criar, remover e ler directórios, assim como tornar um determinado directório o directório corrente. O directório corrente (também designado por *cwd-current working directory* ou directório por defeito, ou ainda directório de trabalho) é aquele onde são criados os novos ficheiros, ou abertos ficheiros existentes, se apenas for indicado o nome do ficheiro e não um *pathname*.

Nunca é possível escrever directamente num directório (no ficheiro especial que representa o directório). Só o sistema operativo o pode fazer quando se solicita a criação ou a remoção de ficheiros nesse directório.

A criação de directórios e a sua remoção pode fazer-se com os serviços `mkdir()` e `rmdir()`. Os directórios deverão ser criados com pelo menos uma das permissões de *execute* para permitir o acesso aos ficheiros aí armazenados (não esquecer a modificação feita pela máscara do utilizador). Para remover um directório este deverá estar completamente vazio.

```
#include <sys/types.h>
#include <sys/stat.h>

int mkdir(const char *pathname, mode_t mode);
```

Cria um novo directório especificado em **`pathname`** e com as permissões especificadas em **`mode`** (alteradas pela máscara do utilizador).

```
#include <unistd.h>

int rmdir(const char *pathname);
```

Remove o directório especificado em **`pathname`** se estiver vazio.
Ambos os serviços retornam 0 no caso de sucesso e –1 no caso de erro.

O acesso às entradas contidas num directórios faz-se através de um conjunto de 4 serviços, nomeadamente `opendir()`, `readdir()`, `rewinddir()` e `closedir()`. Para aceder ao conteúdo de um directório este tem de ser aberto com `opendir()`, retornando um apontador (`DIR *`) que é depois usado nos outros serviços. Sucessivas chamadas a `readdir()` vão retornando por ordem as várias entradas do directório, podendo-se voltar sempre à primeira entrada com `rewinddir()`. Quando já não for mais necessário deverá fechar-se o directório com `closedir()`.

```
#include <sys/types.h>
#include <dirent.h>

DIR *opendir(const char *pathname);
```

Abre o directório especificado em **pathname**. Retorna um apontador (`DIR *`) que representa o directório ou `NULL` no caso de erro.

```
struct dirent *readdir(DIR *dp);
```

Lê a próxima entrada do directório (começa na primeira) retornando um apontador para uma estrutura `dirent`. No caso de erro ou fim do directório retorna `NULL`.

```
void rewinddir(DIR *dp);
```

Faz com que a próxima leitura seja a da primeira entrada do directório.

```
int closedir(DIR *dp);
```

Fecha o directório. Retorna 0 no caso de sucesso e –1 no caso de erro.

A estrutura `dirent` descreve uma entrada de um directório contendo pelo menos campos com o nome do ficheiro e do i-node associado ao ficheiro. Esses dois campos obrigatórios são os que aparecem na definição seguinte:

```
struct dirent {
  ino_t d_ino;               /* número do i-node que descreve o ficheiro */
  char d_name[NAME_MAX+1];   /* string com o nome do ficheiro */
}
```

O directório corrente de um programa pode ser obtido com o serviço `getcwd()`.

```
#include <unistd.h>

char *getcwd(char *buf, size_t size);
```

Obtém o nome (*pathname*) do directório por defeito corrente. A esta função deve ser passado o endereço de um buffer (**buf**) que será preenchido com esse nome, e também o seu tamanho em *bytes* (**size**). O serviço retorna `buf` se este for preenchido, ou `NULL` se o tamanho for insuficiente.

Este directório corrente pode ser mudado programaticamente com o serviço `chdir()`.

```
#include <unistd.h>

int chdir(const char *pathname);
```

Muda o directório por defeito corrente para **pathname**.
Retorna 0 no caso de sucesso e -1 no caso de erro.

3. Criação e terminação de processos

Um processo é uma instância em execução de um programa. No sistema operativo Unix a única forma de se criar um novo processo (processo-filho) é através da invocação, por parte de um processo existente e em execução, do serviço `fork()`:

```
#include <sys/types.h>
#include <unistd.h>

pid_t fork(void);

Retorna:                    0 - para o processo filho
                pid do filho - para o processo pai
                           -1 - se houve erro e o serviço não foi executado
```

A função `fork()` é invocada uma vez, no processo-pai, mas retorna 2 vezes, uma no processo que a invocou e outra num novo processo agora criado, o processo-filho. Este serviço cria um novo processo que é uma cópia do processo que o invoca. Este novo processo-filho (assim como o pai) continua a executar as instruções que se seguem a `fork()`. O filho é uma cópia fiel do pai ficando com uma cópia do segmento de dados, heap e stack; no entanto o segmento de texto (código) é muitas vezes partilhado por ambos. Em geral, não se sabe quem continua a executar imediatamente após uma chamada a `fork()` (se é o pai ou o filho). Depende do algoritmo de escalonamento.

Todos os processos em Unix têm um identificador, geralmente designados por *pid* (*process identifier*). É sempre possível a um processo conhecer o seu próprio identificador e o do seu pai. Os identificadores dos processos em Unix são números inteiros (melhor, do tipo `pid_t` definido em `sys/types.h`) diferentes para cada processo. Os serviços a utilizar para conhecer *pid*'s (além do serviço `fork()`) são:

```
#include <sys/types.h>
#include <unistd.h>

pid_t getpid(void);            /* obtém o seu próprio pid */
pid_t getppid(void);           /* obtém o pid do pai */

Estas funções são sempre bem sucedidas.
```

No exemplo seguinte pode ver-se uma utilização destes três serviços:

```
#include    <sys/types.h>
#include    <unistd.h>
#include    <stdio.h>

int   glob = 6;       /* external variable in initialized data */

int main(void)
{
  int var;            /* automatic variable on the stack */
  pid_t pid;

  var = 88;
  printf("before fork\n");
  if ( (pid = fork()) < 0)
    fprintf(stderr, "fork error\n");
```

```
  else if (pid == 0) {                          /* ***child*** */
    glob++;       /* modify variables */
    var++;
  }
  else
    sleep(2);     /* ***parent***; try to guarantee that child ends first*/
  printf("pid = %d, glob = %d, var = %d\n", getpid(), glob, var);
  return 0;
}
```

Um resultado possível será (os *pid*'s serão concerteza outros):

before fork
pid = 430, glob = 7, var = 89 as variáveis do filho foram modificadas
pid = 429, glob = 6, var = 88 as do pai permanecem inalteradas

Um processo pode terminar normalmente ou anormalmente nas seguintes condições:
Normal:
- Executa `return` na função `main()`;
- Invoca directamente a função `exit()` da biblioteca do C;
- Invoca directamente o serviço do sistema `_exit()`.

Anormal:
- Invoca o função `abort()`;
- Recebe sinais de terminação gerados pelo próprio processo, ou por outro processo, ou ainda pelo Sistema Operativo.

A função `abort()` destina-se a terminar o processo em condições de erro e pertence à biblioteca standard do C. Em Unix a função `abort()` envia ao próprio processo o sinal SIGABRT que tem como consequência terminar o processo. Esta terminação deve tentar fechar todos os ficheiros abertos.

```
#include <stdlib.h>

void abort(void);
```

Esta função nunca retorna.

O Unix mantém sempre uma relação pai-filho entre os processos. Se o pai de um processo terminar antes do filho, este fica momentaneamente orfão. Quando um processo termina, o SO percorre todos os seus processos activos e verifica se algum é filho do que terminou. Quando encontra algum nessas condições o seu pai passa a ser o processo 1 (que existe sempre no sistema). Assim os processos que ficam orfãos são adoptados pelo processo 1, ficando assim garantido que todos os processos têm um pai.

Um processo que termina não pode deixar o sistema até que o seu pai aceite o seu código de terminação (valor retornado por `main()` ou passado a `exit()` ou `_exit()`) , através da execução de uma chamada aos serviços `wait()` / `waitpid()`. Um processo que terminou, mas cujo pai ainda não executou um dos wait's passa ao estado de zombie. (Na saída do comando *ps* o estado destes processos aparece identificado como Z). Quando um processo que foi adoptado por *init* (processo 1) terminar, não se torna zombie, porque init executa um dos wait's para obter o seu código de terminação. Quando um processo passa ao estado de zombie a sua memória é libertada mas permanece no sistema alguma informação sobre o processo (geralmente o seu PCB - *process control block*).

Um pai pode esperar que um dos seus filhos termine e, então, aceitar o seu código de terminação, executando uma das funções `wait()`. Quando um processo termina (normalmente ou anormalmente) o kernel notifica o seu pai enviando-lhe um sinal (`SIGCHLD`). O pai pode ignorar o sinal ou instalar um *signal handler* para receber aquele sinal. Nesse handler poderá executar um dos wait's para obter o identificador (pid) do filho e o seu código de terminação.

```
# include <sys/types.h>
# include <wait.h>

pid_t wait(int *status);
pid_t waitpid(pid_t pid, int *status, int options);
```

Retornam: pid do processo - se OK
-1 - se houve erro

O argumento **`options`** de `waitpid` pode ser:
0 (zero) ou
or, bit a bit (operador |) das constantes
`WNOHANG` - que indica que `waitpid()` não bloqueia se o filho especificado por **`pid`** não estiver imediatamente disponível (terminado). Neste caso o valor de retorno é igual a 0.
`WUNTRACED` - que indica que, se a implementação suportar *job control*, o estado de terminação de qualquer filho especificado por **`pid`** que tenha terminado e cujo *status* ainda não tenha sido reportado desde que ele parou, é agora retornado.

(*job control* - permite iniciar múltiplos *jobs* (grupos de processos) a partir de um único terminal e controlar quais os *jobs* que podem aceder ao terminal e quais os *jobs* que são executados em background)

Um processo que invoque `wait()` ou `waitpid()` pode:
- bloquear - se nenhum dos seus filhos ainda não tiver terminado;
- retornar imediatamente com o código de terminação de um filho - se um filho tiver terminado e estiver à espera de retornar o seu código de terminação (filho zombie).
- retornar imediatamente com um erro - se não tiver filhos.

Diferenças entre `wait()` e `waitpid()`:
- serviço `wait()` pode bloquear o processo que o invoca até que um filho qualquer termine;
- serviço `waitpid()` tem uma opção que impede o bloqueio (útil quando se quer apenas obter o código de terminação do filho);
- `waitpid()` não espera que o 1º filho termine, tem um argumento para indicar o processo pelo qual se quer esperar.

O argumento **`status`** de `waitpid()` pode ser `NULL` ou apontar para um inteiro; no caso de ser ≠ `NULL` - o código de terminação do processo que terminou é guardado na posição indicada por **`status`**; no caso de ser = `NULL` - o código de terminação é ignorado.
O estado retornado por `wait()` / `waitpid()` na variável apontada por **`status`** tem certos bits que indicam se a terminação foi normal, o número de um sinal, se a terminação foi anormal, ou ainda se foi gerada uma *core file*. O estado de terminação pode ser examinado (os bits podem ser testados) usando macros, definidas em `<sys/wait.h>`. Os nomes destas macros começam por `WIF`.
O argumento **`pid`** de `waitpid()` pode ser:
`pid == -1` - espera por um filho qualquer (= `wait()`);
`pid > 0` - espera pelo filho com o pid indicado;

`pid == 0` - espera por um qualquer filho do mesmo *process group*;
`pid < -1` - espera por um qualquer filho cujo *process group ID* seja igual a `|pid|`.
`waitpid()` retorna um erro (valor de retorno = -1) se:
o processo especificado não existir;
o processo especificado não for filho do processo que o invocou;
o grupo de processos não existir.

Eis um exemplo:

```
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>

int main(void)
{
  pid_t pid, childpid;
  int status;

  printf ("I'm the parent proc. with pid %d\n", getpid());
  pid = fork();
  if (pid != 0) {  /* ***parent*** */
    printf ("I'm the parent proc. w/pid %d and ppid %d\n", getpid(),
            getppid());
    childpid = wait(&status);         /* wait for the child to terminate */
    printf("A child w/pid %d terminated w/EXIT CODE %d\n", childpid,
           status>>8);
  }
  else {  /* ***child*** */
    printf("I'm the child proc. w/ pid %d and ppid %d\n", getpid(),
           getppid());
    return 31;                                /*exit with a silly number*/
  }
  printf("pid %d terminates\n", getpid());
  return 0;
}
```

No ficheiro de inclusão `sys/wait.h` estão definidas algumas macros que podem ser usadas para testar o estado de terminação retornado por `wait()` ou `waitpid()`. São elas:
`WIFEXITED(status)` que é avaliada como verdadeira se o filho terminou normalmente.
Neste caso, a macro `WEXITSTATUS(status)` é avaliada como o código de terminação do filho (ou melhor os 8 bits menos significativos passados a `_exit()` / `exit()` ou o valor retornado por `main()`).
`WIFSIGNALED(status)` é avaliada como verdadeira se o filho terminou anormalmente, porque recebeu um sinal que não tratou. Neste caso a macro `WTERMSIG(status)` dá-nos o nº do sinal (não há maneira portável de obter o nome do sinal em vez do número).
`WCOREDUMP(status)` é avaliada como verdadeira se foi gerada uma *core file*.
Finalmente `WIFSTOPPED(status)` é avaliada como verdadeira se o filho estiver actualmente parado (*stopped*). O filho pode ser parado através de um sinal `SIGSTOP`, enviado por outro processo ou pelo sinal `SIGTSTP`, enviado a partir de um terminal (`CTRL-Z`). Neste caso, `WSTOPSIG(status)` dá-nos o nº do sinal que provocou a paragem do processo.

Outro exemplo:

```
#include <sys/types.h>
#include <unistd.h>
```

```
#include <sys/wait.h>
#include <stdio.h>

void pr_exit(int status);

void main(void)
{
  pid_t     pid;
  int status;

  if ((pid = fork()) < 0)
    fprintf(stderr,"fork error\n");
  else if (pid == 0) exit(7);                 /* child */

  if (wait(&status) != pid)
    fprintf(stderr,"wait error\n");
  pr_exit(status);               /* wait for child and print its status */

  if ((pid = fork()) < 0)
    fprintf(stderr,"fork error\n");
  else if (pid == 0) abort();                 /* child generates SIGABRT */

  if (wait(&status) != pid)
    fprintf(stderr,"wait error\n");
  pr_exit(status);               /* wait for child and print its status */

  if ((pid = fork()) < 0)
    fprintf(stderr,"fork error\n");
  else if (pid == 0)
    status /= 0;                /* child - divide by 0 generates SIGFPE */

  if (wait(&status) != pid)
    fprintf(stderr,"wait error\n");
  pr_exit(status);               /* wait for child and print its status */

  exit(0);
}

void pr_exit(int status)
{
  if (WIFEXITED(status))
    printf("normal termination, exit status = %d\n", WEXITSTATUS(status));
  else if (WIFSIGNALED(status))
    printf("abnormal termination, signal number = %d%s\n",
           WTERMSIG(status),
#ifdef WCOREDUMP               /* nem sempre está definida em sys/wait.h */
           WCOREDUMP(status) ? " (core file generated)" : "");
#else
           "");
#endif
  else if (WIFSTOPPED(status))
      printf("child stopped, signal number = %d\n", WSTOPSIG(status));
}
```

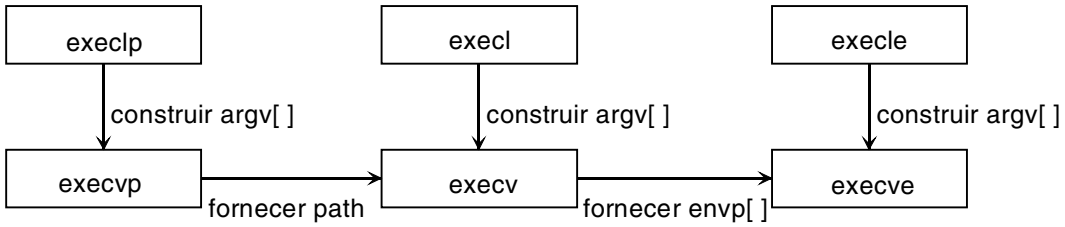
Como se viu atrás, a função `fork()` é a única no Unix capaz de criar um novo processo. Mas fá-lo duplicando o código do pai e não substituindo esse código por outro residente num ficheiro executável. Para esta última função existem no Unix seis serviços (designados genericamente por serviços `exec()`) que, embora não criando um novo processo, substituem totalmente a imagem em memória do processo que os invoca por uma imagem proveniente de um ficheiro executável, especificado na chamada.

Os seis serviços são:

```
#include <unistd.h>

int execl(const char *pathname, const char *arg0, … /* NULL */ );
int execv(const char *pathname, const char *argv[]);
int execle(const char *pathname, const char *arg0, … /* NULL,
           char * const envp[] */ );
int execve(const char *pathname, const char *argv[ ],
           char * const envp[]);
int execlp(const char *filename, const char *arg0, … /* NULL */ );
int execvp(const char *filename, const char *argv[]);
```

Se os serviços tiverem sucesso não retornam (o código que os chamou é substituído); retornam –1 se houver erro.

O parâmetro **`pathname`** (string) deve indicar o nome do ficheiro executável que se pretende venha a substituir o código do processo que chama o serviço. Este nome deve ser completo no sentido de incluir todo o *path* desde a raiz. Nos serviços que contêm um p no nome (2 últimos), basta indicar o nome do ficheiro executável para o parâmetro **`filename`**. O SO procurará esse ficheiro nos directórios especificados pela variável de ambiente PATH.
Os serviços que contêm um l (letra éle) no nome aceitam a seguir ao primeiro parâmetro, uma lista de strings, terminada por `NULL`, que constituirão os parâmetros de chamada do programa que agora se vai passar a executar (deve incluir-se, como primeiro parâmetro desta lista, o nome do programa). Nos serviços que contêm a letra v esses parâmetros de chamada devem ser previamente colocados num vector `argv[]` (ver capítulo 1). Finalmente os serviços que contêm a letra e, aceitam como último parâmetro um vector (`envp[]`) de apontadores para string com as variáveis de ambiente e suas definições (ver capítulo 1).



Alguns exemplos:

```
#include    <unistd.h>
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
execl("/bin/ls", "ls", "-l", NULL);
_____________________________________________________________________________
char   *env_init[] = {"USER=unknown", "PATH=/tmp", NULL};
 ...
execle("/users/stevens/bin/prog", "prog", "arg1", "arg2", NULL, env_init);
_____________________________________________________________________________
execlp("prog","prog","arg1",NULL);   /* o executável é procurado no PATH */
_____________________________________________________________________________
...
int main (int argc, char *argv[])
{
```

```
    if (fork() == 0) {
       execvp(argv[1], &argv[1]);
       fprintf(stderr, "Can't execute\n", argv[1]);
    }
}
```

Neste último pedaço de código mostra-se um pequeno programa que executa outro, cujo nome lhe é passado como 1º argumento. Se este programa se chamar run, a invocação run cc prog1.c executará cc prog1.c se cc for encontrado no PATH.

Como último serviço que referiremos temos o serviço `system()`. Este serviço permite executar um programa externo do interior do processo que o invoca. O processo que invoca o serviço fica bloqueado até que a execução pedida termine. Este serviço é assim equivalente à sequência `fork()`, `exec()`, `waitpid()`.

```
#include <stdlib.h>

int system(const char* cmdstring);
```

Retorna:
-1 - se `fork()` falhou ou se `waitpid()` retornou um erro
127 - se `exec()` falhou
código de terminação do programa invocado, no formato especificado por `waitpid()` se tudo correu bem.

Exemplo:

```
system("date > file");
```

4. *Threads* em Unix

Só recentemente a especificação POSIX da API do Unix estabeleceu uma interface standard para a criação e terminação de *threads*. Esta interface é geralmente implementada como uma biblioteca (libpthreads.a), podendo ou não ser directamente suportada pelo kernel do sistema operativo. Existem versões do Unix em que os *threads* são directamente implementados no kernel e escalonados independentemente (*kernel level threads*), enquanto que em outras versões o que o kernel vê são apenas processos, "existindo" os *threads* unicamente na biblioteca de suporte, e sendo escalonados para execução apenas na "fatia de tempo" dos respectivos processos (*user level threads*).

Quando um programa começa a executar na função `main()` constitui um novo processo e pode considerar-se que contém já um *thread*. Novos *threads* são criados através da chamada de um serviço e começam a executar numa função que faz parte do código do programa. O serviço de criação de novos *threads* é então:

```
#include <pthread.h>

int pthread_create(pthread_t *tid, const pthread_attr_t *attr,
                   void * (* start)(void *), void *arg);
```

Retorna 0 no caso de sucesso e um código de erro no caso contrário.
No caso de erro o código retornado pode ser passado directamente a `strerror()` ou a variável global `errno` poderá ser preenchida com esse valor antes de se chamar `perror()`.

Cada novo *thread* é representado por um identificador (*thread identifier* ou tid) de tipo `pthread_t`. É necessário passar o endereço de uma variável desse tipo, como primeiro parâmetro de `pthread_create()`, para receber o respectivo tid. O segundo parâmetro serve para indicar uma série de atributos e propriedades que o novo *thread* deverá ter. Se aqui se fornecer o valor `NULL`, o novo *thread* terá as propriedades por defeito, que são adequadas na maior parte dos casos. O terceiro parâmetro indica a função de início do *thread*. É uma função que deve existir com o seguinte protótipo:

```
void * nome_da_função(void *arg)
```

A função aceita um apontador genérico como parâmetro, que serve para passar qualquer informação, e retorna também um apontador genérico, em vez de um simples código de terminação. Se o valor de retorno for usado, é necessário que aponte para algo que não deixe de existir quando o *thread* termina. Finalmente o quarto parâmetro é o valor do apontador a ser passado à função de inicíco, como seu parâmetro.

Uma vez criado o novo *thread* ele passa a executar concorrentemente com o principal e com outros que porventura sejam criados.

Um *thread* termina quando a função de início, indicada quando da criação, retornar, ou quando o próprio *thread* invocar o serviço de terminação:

```
#include <pthread.h>

void pthread_exit(void *value_ptr);
```

onde **`value_ptr`** é o apontador que o *thread* deve ter como resultado.

Quando um *thread* termina pode retornar um apontador para uma área de memória que contenha qualquer tipo de resultado. Essa área de memória deve sobreviver o *thread*, ou seja, não pode ser nenhuma variável local, porque essas deixam de existir quando o *thread* termina.

Qualquer *thread* pode esperar que um dado *thread* termine, e ao mesmo tempo obter o seu valor de retorno, que é um apontador genérico (`void *`). Basta para isso usar o serviço:

```
#include <pthread.h>

int pthread_join(pthread_t thread, void **value_ptr);
```

onde **`thread`** é o identificador do *thread* (tid) que se pretende esperar, e **`value_ptr`** é o endereço de um apontador onde será colocado o resultado do *thread* que vai terminar. Se se passar para este parâmetro o valor `NULL`, o retorno do *thread* é ignorado.
O serviço retorna 0 no caso de sucesso e um código de erro no caso contrário.

Por defeito, quando um *thread* termina, o seu valor de retorno (o apontador genérico) não é destruído, ficando retido em memória até que algum outro *thread* execute um `pthread_join()` sobre ele. É possível criar *threads* que não são *joinable* e que por isso, quando terminam, libertam todos os seus recursos, incluindo o valor de retorno. No entanto não é possível esperar por esses *threads* com `pthread_join()`. Estes *threads* designam-se por *detached*, podendo ser já criados nessa situação (usando o parâmetro **`attr`** de `pthread_create()`), ou podendo ser colocados nessa situação após a criação, com o serviço:

```
#include <pthread.h>

int pthread_detach(pthread_t thread);
```

onde **`thread`** é o identificador do *thread* (tid) que se pretende tornar *detached*.
O serviço retorna 0 no caso de sucesso e um código de erro no caso contrário.

O normal será o próprio *thread* tornar-se ele próprio *detached*. Para isso necessita de conhecer o seu próprio tid. Um *thread* pode obter o seu tid com o serviço:

```
#include <pthread.h>

pthread_t pthread_self(void);
```

Retorna sempre o tid do *thread* que o invoca.

Segue-se um exemplo simples de demonstração do uso de *threads* em UNIX:

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

int global;

void *thr_func(void *arg);

int main(void)
{
```

```
  pthread_t tid;

  global = 20;
  printf("Thread principal: %d\n", global);
  pthread_create(&tid, NULL, thr_func, NULL);
  pthread_join(tid, NULL);
  printf("Thread principal: %d\n", global);
  return 0;
}

void *thr_func(void *arg)
{
    global = 40;
    printf("Novo thread: %d\n", global);
    return NULL;
}
```

Quando se cria um novo *thread* é possível especificar uma série de atributos e propriedades passando-os a `pthread_create()` através de uma variável do tipo `pthread_attr_t`. Essa variável terá de ser previamente inicializada com o serviço `pthread_attr_init()` e depois modificada através da chamada de serviços específicos para cada atributo.

Algumas dessas funções de construção da variável `pthread_attr_t` estão listadas a seguir:

```
#include <pthread.h>

int pthread_attr_init(pthread_attr_t *attr);
int pthread_attr_destroy(pthread_attr_t *attr);
int pthread_attr_setstacksize(pthread_attr_t *attr, int size);
int pthread_attr_getstacksize(pthread_attr_t *attr, int *size);
int pthread_attr_setstackaddr(pthread_attr_t *attr, int addr);
int pthread_attr_getstackaddr(pthread_attr_t *attr, int *addr);
int pthread_attr_setdetachstate(pthread_attr_t *attr, int state);
int pthread_attr_getdetachstate(pthread_attr_t *attr, int *state);
int pthread_attr_setscope(pthread_attr_t *attr, int scope);
int pthread_attr_getscope(pthread_attr_t *attr, int *scope);
int pthread_attr_setinheritsched(pthread_attr_t *attr, int sched);
int pthread_attr_getinheritsched(pthread_attr_t *attr, int *sched);
int pthread_attr_setschedpolicy(pthread_attr_t *attr, int policy);
int pthread_attr_getschedpolicy(pthread_attr_t *attr, int *policy);
int pthread_attr_setschedparam(pthread_attr_t *attr,
                               struct sched_param *param);
int pthread_attr_getschedparam(pthread_attr_t *attr,
                               struct sched_param *param);
```

Retornam 0 no caso de sucesso e um código de erro no caso contrário.

Por exemplo, para criar um *thread* já no estado de *detached* deveria usar-se:

```
pthread_attr_t attr;

…
pthread_attr_init(&attr);
pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_DETACHED);
pthread_create(&tid, &attr, …, …);
…
```

```
pthread_attr_destroy(&attr);
…
```

Para outros atributos e propriedades suportadas, consultar o manual (`man`) do Unix.

5. Comunicação entre processos - Sinais

5.1 Definição dos sinais

Os sinais são uma espécie de interrupção ao processo corrente. Podem ter diversas origens e são uma forma de tratar certos acontecimentos de carácter assíncrono.

Todos os sinais têm um nome simbólico que começa pelo prefixo `SIG`. Vêm listados no ficheiro de inclusão `<signal.h>`.

Possíveis origens: teclado - certas combinações de teclas dão origem a sinais; hardware - por exemplo, erros aritméticos como a divisão por 0; serviços do sistema operativo - `kill()`, `abort()`, `alarm()`, ...; comandos da shell - kill; software - certos eventos gerados no código dos processos dão origem a sinais, como erros de *pipes*.

Um processo pode programar a resposta a dar a cada tipo de sinal. As respostas permitidas são: ignorar o sinal, tratar o sinal com uma rotina do programador (*catch*), ou simplesmente deixar que o processo tenha a resposta por defeito (*default action*) para esse tipo de sinal.

Na tabela seguinte apresentam-se alguns sinais definidos nos Sistemas Operativos Unix:

Nome	Descrição	Origem	Acção por defeito
`SIGABRT`	Terminação anormal	`abort()`	Terminar
`SIGALRM`	Alarme	`alarm()`	Terminar
`SIGCHLD`	Filho terminou ou foi suspenso	S.O.	Ignorar
`SIGCONT`	Continuar processo suspenso	shell (fg, bg)	Continuar
`SIGFPE`	Excepção aritmética	hardware	Terminar
`SIGILL`	Instrução ilegal	hardware	Terminar
`SIGINT`	Interrupção	teclado (^C)	Terminar
`SIGKILL`	Terminação (*non catchable*)	S.O.	Terminar
`SIGPIPE`	Escrever num *pipe* sem leitor	S.O.	Terminar
`SIGQUIT`	Saída	teclado (^|)	Terminar
`SIGSEGV`	Referência a memória inválida	hardware	Terminar
`SIGSTOP`	Stop (*non catchable*)	S.O. (shell - stop)	Suspender
`SIGTERM`	Terminação	teclado (^U)	Terminar
`SIGTSTP`	Stop	teclado (^Y, ^Z)	Suspender
`SIGTTIN`	Leitura do teclado em *backgd*	S.O. (shell)	Suspender
`SIGTTOU`	Escrita no écran em *backgd*	S.O. (shell)	Suspender
`SIGUSR1`	Utilizador	de 1 proc. para outro	Terminar
`SIGUSR2`	Utilizador	idem	Terminar

5.2 Tratamento dos sinais

Quando se cria um processo novo a resposta a todos os sinais definidos no sistema é a acção por defeito. No entanto o processo, através da chamada de serviços do sistema, pode apanhar (*catch*) um determinado sinal e ter uma rotina (*handler*) para responder a esse sinal, ou simplesmente pode ignorá-lo (excepto para os sinais `SIGKILL` e `SIGSTOP`).

A instalação de um handler para o tratamento de um determinado sinal (ou a

determinação de o ignorar) pode ser feita através do serviço `signal()`.

Este serviço `signal()` faz também parte da biblioteca standard do C pelo que deve existir em todos os S.O.'s.

```
#include <signal.h>

void (*signal(int signo, void (*handler)(int)))(int);
```

onde **`signo`** é o identificador do sinal a "apanhar" e **`handler`** o nome da função que será chamada sempre que o processo receber um sinal do tipo **`signo`**.

A função retorna o endereço do handler anterior para este sinal ou `SIG_ERR` no caso de erro.

A declaração do serviço `signal()` é um pouco complicada. Compreende-se melhor se se declarar primeiro o seguinte tipo:

```
typedef void sigfunc(int);
```

Esta declaração define um tipo chamado **`sigfunc`** que é uma função com um parâmetro inteiro retornando nada (um procedimento). Com o tipo **`sigfunc`** definido podemos agora declarar o serviço `signal()` como:

```
sigfunc *signal(int signo, sigfunc *handler);
```

ou seja `signal()` é uma função com dois parâmetros: um inteiro (identificador do sinal - **`signo`**) e um apontador para uma função - **`handler`**; `signal()` retorna um apontador para função do tipo **`sigfunc`**.

Assim os handlers de sinais deverão ser funções que não retornam nada com um parâmetro inteiro. Após a instalação de um handler para um determinado sinal, sempre que o processo recebe esse sinal o handler é chamado, com parâmetro de entrada igual ao identificador do sinal recebido.

Existem duas constantes que podem ser passadas no lugar do parâmetro **`handler`** de `signal()`:

- `SIG_IGN` - quando se pretende ignorar o sinal **`signo`**;
- `SIG_DFL` - quando se pretende reinstalar a acção por defeito do processo para o sinal **`signo`**.

5.3 Envio de sinais

Além das origens dos sinais já referidas atrás é possível um processo enviar um sinal qualquer a ele próprio ou a outro processo (desde que seja do mesmo utilizador). Os serviços utilizados para isso são:

```
#include <signal.h>

int raise(int signo);
```

Retorna 0 no caso de sucesso; outro valor no caso de erro.

O serviço `raise()` envia o sinal especificado em **`signo`** ao próprio processo que executa o serviço. A resposta depende da disposição que estiver em vigor (acção por defeito, *catch*, ou ignorar) para o sinal enviado.

O serviço que permite o envio de sinais entre processos diferentes é:

```
#include <sys/types.h>
#include <signal.h>

int kill(pid_t pid, int signo);

Retorna 0 no caso de sucesso; outro valor no caso de erro.
```

O processo de destino do sinal **`signo`** é especificado em **`pid`**. No caso de **`pid`** especificar o próprio processo a função `kill()` só retorna após o tratamento do sinal enviado pelo processo.

5.4 Outros serviços relacionados com sinais

O serviço `alarm()` enviará ao próprio processo o sinal `SIGALRM` após a passagem de **`seconds`** segundos. Se no momento da chamada estiver activa outra chamada prévia a `alarm()`, essa deixa de ter efeito sendo substituída pela nova. Para cancelar uma chamada prévia a `alarm()` sem colocar outra activa pode-se executar `alarm(0)`.

```
#include <unistd.h>

unsigned int alarm(unsigned int seconds);

Retorna 0 normalmente, ou o número de segundos que faltavam a uma possível chamada
prévia a alarm().
```

O serviço `pause()` suspende o processo que o chamou (deixa de executar). O processo só voltará a executar quando receber um sinal qualquer não ignorado. O serviço `pause()` só retornará se o handler do sinal recebido também retornar.

```
#include <unistd.h>

int pause(void);

Quando retorna, retorna o valor -1.
```

O serviço `abort()` envia o sinal `SIGABRT` ao próprio processo. O processo não deve ignorar este sinal. A acção por defeito deste sinal é terminar imediatamente o processo sem processar eventuais handlers `atexit()` existentes (ver capítulo 1).

```
#include <stdlib.h>

void abort(void);
```

O serviço `sleep()` suspende (bloqueia) temporariamente o processo que o chama, pelo intervalo de tempo especificado em **`seconds`**. A suspensão termina quando o intervalo de tempo se esgota ou quando o processo recebe qualquer sinal não ignorado e o respectivo

handler retornar.

```
#include <unistd.h>

unsigned int sleep(unsigned int seconds);
```

Retorna 0 normalmente, ou o número de segundos que faltam para completar o serviço.

Seguem-se alguns exemplos da utilização de sinais.

Exemplo 1 - processo que trata apenas os sinais `SIGUSR1` e `SIGUSR2`:

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

static void sig_usr(int);     /* o mesmo handler para os 2 sinais */

int main(void)
{
      if (signal(SIGUSR1, sig_usr) == SIG_ERR) {
            fprintf(stderr, "can't catch SIGUSR1\n");
            exit(1);
      }
      if (signal(SIGUSR2, sig_usr) == SIG_ERR) {
            fprintf(stderr, "can't catch SIGUSR2\n");
            exit(1);
      }
      for ( ; ; )
            pause();
}

static void sig_usr(int signo)    /* o argumento indica o sinal recebido */
{
      if (signo == SIGUSR1)
            printf("received SIGUSR1\n");
      else if (signo == SIGUSR2)
            printf("received SIGUSR2\n");
      else {
            fprintf(stderr, "received signal %d\n", signo);
            exit(2);
      }
      return;
}
```

Exemplo 2 - Estabelecimento de um alarme e respectivo handler

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

int alarmflag = 0;
void alarmhandler(int signo);

void main(void)
{
    signal(SIGALRM, alarmhandler);
    alarm(5);
    printf("Looping ...\n");
```

```
    while (!alarmflag)
        pause();
    printf("Ending ...\n");
}

void alarmhandler(int signo)
{
    printf("Alarm received ...\n");
    alarmflag = 1;
}
```

Exemplo 3 - Protecção contra `CTRL-C` (que gera o sinal `SIGINT`)

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

void main(void)
{
    void (*oldhandler)(int);

    printf("I can be Ctrl-C'ed\n");
    sleep(3);
    oldhandler = signal(SIGINT, SIG_IGN);
    printf("I'm protected from Ctrl-C now\n");
    sleep(3);
    signal(SIGINT, oldhandler);
    printf("I'm vulnerable again!\n");
    sleep(3);
    printf("Bye.\n");
}
```

Exemplo 4 - Aqui pretende-se um programa que lance um outro e espere um certo tempo para que o 2º termine. Caso isso não aconteça deverá terminá-lo de modo forçado.

Exemplo de linha de comando:
`limit n prog arg1 arg2 arg3`

`n` - nº de segundos a esperar
`prog` - programa a executar
`arg1, arg2, ..., argn` - argumentos de **`prog`**

```
#include <stdio.h>
#include <signal.h>
#include <unistd.h>
#include <sys/types.h>

int delay;
void childhandler(int signo);

void main(int argc, char *argv[])
{
    pid_t pid;

    signal(SIGCHLD, childhandler);   /* quando um processo filho termina */
    pid = fork();                    /* envia ao pai o sinal SIGCHLD */
    if (pid = 0)                     /* filho */
        execvp(argv[2], &argv[2]);
    else {                              /* pai */
        sscanf(argv[1], "%d", &delay);  /* transforma string em valor */
        sleep(delay);
```

```
        printf("Program %s exceeded limit of %d seconds!\n",
               argv[2], delay);
        kill(pid, SIGKILL);
    }
}

void childhandler(int signo)
{
    int status;
    pid_t pid;

    pid = wait(&status);
    printf("Child %d terminated within %d seconds.\n", pid, delay);
    exit(0);
}
```

5.5 Bloqueamento de sinais

A norma POSIX especifica outras formas mais complicadas de instalação de handlers para sinais e permite um outro mecanismo para o tratamento de sinais. Com os serviços POSIX é possível bloquear selectivamente a entrega de sinais a um processo. Se um determinado sinal estiver bloqueado (diferente de ignorado) quando for gerado, o S.O. não o envia para o processo, mas toma nota de que esse sinal está pendente. Logo que o processo desbloqueie esse sinal ele é imediatamente enviado ao processo (pelo S.O.) entrando em acção a disposição que estiver em vigor (acção por defeito, *catch* com handler, ou ignorar). Se vários sinais idênticos forem gerados para um processo, enquanto este os estiver a bloquear, geralmente o S.O. só toma nota de que um está pendente, perdendo-se os outros.

Para permitir especificar qual ou quais os sinais que devem ser bloqueados por um processo, este deve construir uma máscara. Esta máscara funciona como uma estrutura de dados do tipo "conjunto" que é possível esvaziar, preencher completamente com todos os sinais suportados, acrescentar ou retirar um sinal, ou ainda verificar se um dado sinal já se encontra na máscara.

Os serviços definidos em POSIX para estas operações são os seguintes e operam sobre uma máscara de tipo `sigset_t` definido no ficheiro de inclusão `signal.h`:

```
#include <signal.h>

int sigemptyset(sigset_t *set);
int sigfillset(sigset_t *set);
int sigaddset(sigset_t *set, int signo);
int sigdelset(sigset_t *set, int signo);
int sigismember(const sigset_t *set, int signo);
```

Todos estes serviços têm como primeiro argumento o endereço de uma máscara (**`set`**) do tipo `sigset_t`.
Os quatro primeiros retornam 0 se OK e -1 no caso de erro. O quinto retorna 1 se *true* e 0 se *false*.

O serviço `sigemptyset()` preenche a máscara como vazia (sem nenhum sinal), enquanto que `sigfillset()` preenche a máscara com todos os sinais suportados no sistema. O serviço `sigaddset()` acrescenta o sinal **`signo`** à máscara, emquanto que `sigdelset()`

retira o sinal **signo** à máscara. Finalmente `sigismember()` testa se o sinal **signo** pertence ou não à máscara.

Tendo construído uma máscara contendo os sinais que nos interessam podemos bloquear (ou desbloquear) esses sinais usando o serviço `sigprocmask()`.

```
#include <signal.h>

int sigprocmask(int how, const sigset_t *set, sigset_t *oset);
```

Se **oset** não for `NULL` então é preenchido com a máscara corrente do processo (contendo os sinais que estão bloqueados).
Se **set** não for `NULL` então a máscara corrente é modificada de acordo com o valor do parâmetro **how**:

- `SIG_BLOCK` - a nova máscara é a reunião da actual com a especificada em **set**, i.e. **set** contém sinais adicionais a bloquear (desde que ainda não estejam bloqueados);
- `SIG_UNBLOCK` - a nova máscara é a intersecção da actual com o complemento de **set**, i.e. **set** contém os sinais a desbloquear;
- `SIG_SETMASK` - a nova máscara passa a ser a especificada em **set**.

Retorna 0 se OK e -1 no caso de erro.

Para, por exemplo, bloquear os sinais `SIGINT` e `SIGQUIT`, podíamos escrever o seguinte código:

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

sigset_t sigset;

sigemptyset(&sigset);          /* comecemos com a máscara vazia */
sigaddset(&sigset, SIGINT);    /* acrescentar SIGINT */
sigaddset(&sigset, SIGQUIT);   /* acrescentar SIGQUIT */
if (sigprocmask(SIG_BLOCK, &sigset, NULL))
      perror("sigprocmask");
```

A norma POSIX estabelece um novo serviço para substituir `signal()`. Esse serviço chama-se `sigaction()` e além de permitir estabelecer uma disposição para o tratamento de um sinal, permite numerosas outras opções, entre as quais bloquear ao mesmo tempo outros sinais.

```
#include <signal.h>

int sigaction(int signo, const struct sigaction *act,
                         struct sigaction *oact);
```

O serviço `sigaction()` permite modificar e/ou examinar a acção associada a um sinal especificado em **signo**. Faz-se uma modificação se **act** for diferente de `NULL` e um exame se **oact** for diferente de `NULL`. O parâmetro **oact** é preenchido pelo serviço com as disposições actuais, enquanto que **act** contém as novas disposições.
O serviço retorna 0 se OK e -1 em caso de erro.

A definição da estrutura `sigaction` em `signal.h` é a que se apresenta a seguir (poderá conter mais campos):

```
struct sigaction {
    void (*sa_handler)(int);
    sigset_t sa_mask;
    int sa_flags;
}
```

O campo **sa_handler** contém o endereço de um handler (ou as constantes SIG_DFL ou SIG_IGN); o campo **sa_mask** contém uma máscara de sinais que são automaticamente bloqueados durante a execução do handler (juntamente com o sinal que desencadeou a execução do handler) unicamente se **sa_handler** contiver o endereço de uma função e não as constantes SIG_DFL ou SIG_IGN; o campo **sa_flags** poderá conter a especificação de comportamentos adicionais (que são dependentes da implementação - ver man).

No seguinte exemplo estabelece-se um handler para SIGINT (Control-C) usando o serviço sigaction():

```
    char msg[] = "Control - C pressed!\n";

    void catch_ctrl_c(int signo)
    {
        write(STDERR_FILENO, msg, strlen(msg));
    }
    ...
    struct sigaction act;
    ...
    act.sa_handler = catch_ctrl_c;
    sigemptyset(&act.sa_mask);
    act.sa_flags = 0;
    sigaction(SIGINT, &act, NULL);
    ...
```

O serviço pause() permite bloquear um processo até que este receba um sinal qualquer. No entanto se se pretender bloquear o processo até que seja recebido um sinal específico, uma forma de o fazer seria usar uma *flag* que seria colocada em 1 no handler desse sinal (sem ser modificada noutros handlers) e escrever o seguinte código:

```
    int flag = 0;
    ...
    while (flag == 0)
        pause();
    ...
```

No entanto este código tem um problema se o sinal que se pretende esperar surgir depois do teste da *flag* e antes da chamada a pause(). Para obviar este problema a norma POSIX especifica o seguinte serviço:

```
#include <signal.h>

int sigsuspend(const sigset_t *sigmask);
```

Quando retorna, retorna sempre o valor -1.

Este serviço põe em vigor a máscara especificada em **sigmask** e bloqueia o processo até este receber um sinal. Após a execução do handler e o retorno de sigsuspend() a máscara original é restaurada.

Exemplo:

Suponhamos que queríamos proteger uma região de código da ocorrência da combinação Control-C e, logo a seguir, esperar por uma dessas ocorrências. Poderíamos ser tentados a escrever o seguinte código:

```
sigset_t newmask, oldmask;
...
sigemptyset(&newmask);
sigaddset(&newmask, SIGINT);
sigprocmask(SIG_BLOCK, &newmask, &oldmask);

...     /* região protegida */

sigprocmask(SIG_SETMASK, &oldmask, NULL);
pause();
...
```

Este processo ficaria bloqueado se a ocorrência de Control-C aparecesse antes da chamada a `pause()`.

Uma versão correcta deste programa seria:

```
sigset_t newmask, oldmask;
...
sigemptyset(&newmask);
sigaddset(&newmask, SIGINT);
sigprocmask(SIG_BLOCK, &newmask, &oldmask);

...     /* região protegida */

sigsuspend(&oldmask);
sigprocmask(SIG_SETMASK, &oldmask, NULL);
...
```

Porque é necessário o último `sigprocmask()` ?

5.6 Interacção com outros serviços do Sistema Operativo

Dentro de um handler só é seguro chamar certas funções da API do S.O. Como os handlers podem ser chamados assincronamente em qualquer ponto do código de um processo os serviços aí chamados têm de ser reentrantes. A norma POSIX e cada fabricante de UNIX especificam quais são as funções que podem ser chamadas de forma segura do interior de um handler (essas funções dizem-se *async-signal safe*).

Certas funções da API do UNIX são classificadas como "*slow*" porque podem bloquear o processo que as chama durante intervalos de tempo longos (p. ex. operações de entrada/saída através de uma rede (*sockets*), através de *pipes*, abertura de um terminal ligado a um modem, certas operações de comunicação entre processos, etc). Esses serviços "*slow*" podem ser interrompidos quando da chegada de um sinal.
Quando isso sucede, e após a execução do handler do sinal que as interrompeu, essas funções retornam com a indicação de um erro específico (**`errno`** = `EINTR`). Para a correcta execução do programa que sofreu este tipo de interrupção é necessário tornar a chamar estes serviços. O código geralmente usado para isso, quando se chama um serviço "*slow*", é então:

```
while (n = read(fd_sock, buf, size), n==-1 && errno==EINTR);
if (n==-1)
    perror("read");
```

O ciclo `while` torna a chamar `read()` (num *socket* (rede)) se este serviço tiver sido interrompido pela chegada de um sinal.

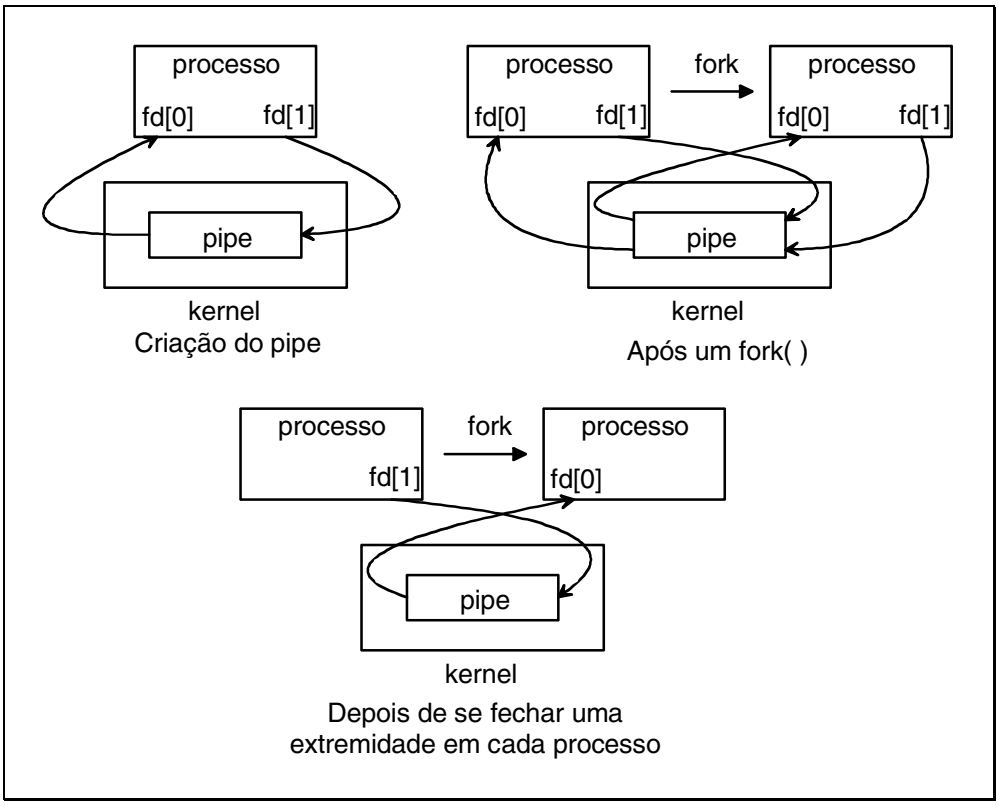
Cada versão do UNIX define quais são os seus serviços "*slow*". Algumas versões do UNIX permitem especificar no campo **`sa_flags`** da estrutura `sigaction` uma opção que torna a chamar automaticamente os serviços "*slow*" que tenham sido interrompidos por esse sinal. Quando existe, essa opção tem o nome de `SA_RESTART`.

6. Comunicação entre processos - *Pipes*

6.1 O que são *pipes*

Os *pipes* em UNIX constituem um canal de comunicação unidirecional entre processos com um ascendente comum (entre um pai e um seu descendente). Uma vez estabelecido o *pipe* entre os processos, um deles pode enviar "mensagens" (qualquer sequência de bytes) para o outro. O envio e recebimento destas "mensagens" é feito com os serviços normais de leitura e escrita em ficheiros - `read()` e `write()`. Os *pipes* possuem descritores semelhantes aos dos ficheiros.

Quando se cria um *pipe* o sistema retorna, para o processo que chamou o serviço de criação, dois descritores que representam o lado de escrita no *pipe* e o lado de leitura no *pipe*. Inicialmente esses descritores pertencem ambos a um processo. Quando esse processo lança posteriormente um filho este herdará esses descritores (herda todos os ficheiros abertos) ficando assim pronto o canal de comunicação entre os dois processos. Consoante o sentido de comunicação pretendido deverá fechar-se, em cada processo, um dos lados do *pipe* (ver a figura seguinte).



6.2 Criação e funcionamento de *pipes*

Assim, para criar um *pipe* num processo deve invocar-se o serviço:

```
#include <unistd.h>

int pipe(int filedes[2]);
```

Retorna 0 no caso de sucesso e -1 no caso de erro.

Deve passar-se ao serviço `pipe()` um array de 2 inteiros (**`filedes`**) que será preenchido pelo serviço com os valores dos descritores que representam os 2 lados do *pipe*. O descritor contido em `filedes[0]` está aberto para leitura - é o lado receptor do *pipe*. O descritor contido em `filedes[1]` está aberto para escrita - é o lado emissor do *pipe*.

Quando um dos lados do `pipe` está fechado e se tenta uma operação de leitura ou escrita do outro lado, aplicam-se as seguintes regras:

1. Se se tentar ler de um *pipe* cujo lado emissor tenha sido fechado, após se ter lido tudo o que porventura já tenha sido escrito, o serviço `read()` retornará o valor `0`, indicador de fim de ficheiro.

2. Se se tentar escrever num *pipe* cujo lado receptor já tenha sido fechado, gera-se o sinal `SIGPIPE` e o serviço `write()` retorna um erro (se `SIGPIPE` não terminar o processo, que é a sua acção por defeito).

Exemplo - estabelecimento de um *pipe* entre pai e filho

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <sys/types.h>

int main(void)
{
      int   n, fd[2];
      pid_t pid;
      char  line[MAXLINE];

      if (pipe(fd) < 0) {
            fprintf(stderr, "pipe error\n");
            exit(1);
      }
      if ( (pid = fork()) < 0) {
            fprintf(stderr, "fork error\n");
            exit(2);
      }
      else if (pid > 0) {           /* pai */
            close(fd[0]);           /* fecha lado receptor do pipe */
            write(fd[1], "hello world\n", 12);
      } else {                      /* filho */
            close(fd[1]);           /* fecha lado emissor do pipe */
            n = read(fd[0], line, MAXLINE);
            write(STDOUT_FILENO, line, n);
      }
      exit(0);
}
```

Outro exemplo - Usar um programa externo, p. ex. um pager (formatador de texto), para lhe enviar, através de um pipe redireccionado para a sua entrada standard, o texto a paginar.

```
#include    <unistd.h>
#include    <sys/wait.h>
#include    <stdio.h>

#define     PAGER "/usr/bin/more"                       /* programa pager */
```

```
int main(int argc, char *argv[])
{
      int   n, fd[2];
      pid_t pid;
      char  line[MAXLINE], *pager, *argv0;
      FILE  *fp;

      if (argc != 2) {
            printf("usage: prog filename\n");
            exit(0);
      }

      /* abertura do ficheiro de texto usando bibl. standard do C */

      if ( (fp = fopen(argv[1], "r")) == NULL) {
            fprintf(stderr, "can't open %s\n", argv[1]);
            exit(1);
      }
      if (pipe(fd) < 0) {
            fprintf(stderr, "pipe error\n");
            exit(2);
      }
      if ( (pid = fork()) < 0) {
            fprintf(stderr, "fork error\n");
            exit(3);
      }
      else if (pid > 0) {                              /* pai */
            close(fd[0]);                        /* fecha o lado receptor */

            /* copia argv[1] para o pipe */

            while (fgets(line, MAXLINE, fp) != NULL) {
                  n = strlen(line);
                  if (write(fd[1], line, n) != n) {
                        fprintf(stderr, "write error to pipe\n");
                        exit(4);
                  }
            }
            if (ferror(fp)) {
                  fprintf(stderr, "fgets error");
                  exit(5);
            }
            close(fd[1]);                  /* fecha lado emissor do pipe */

            /* espera pelo fim do filho */

            if (waitpid(pid, NULL, 0) < 0) {
                  fprintf(stderr, "waitpid error\n");
                  exit(6);
            }
            exit(0);                                 /* termina normalmente */
      } else {                     /* filho */
            close(fd[1]);                             /* fecha lado emissor */
            if (fd[0] != STDIN_FILENO) {        /* programação defensiva */

                  /* redirecciona fd[0] para entrada standard */

                  if (dup2(fd[0], STDIN_FILENO) != STDIN_FILENO) {
                        fprintf(stderr, "dup2 error to stdin\n");
                        exit(7);
```

```
                }
                close(fd[0]);          /* não é preciso depois do dup2()*/
        }
        pager = PAGER;                  /* obtém argumentos para execl()*/
        if ( (argv0 = strrchr(pager, '/')) != NULL)
                argv0++;                            /* após último slash */
        else
                argv0 = pager;                  /* não há slashes em pager */

        /* executa pager com a entrada redireccionada */

        if (execl(pager, argv0, NULL) < 0) {
                fprintf(stderr, "execl error for %s\n", pager);
                exit(8);
        }
    }
}
```

Constitui uma operação comum, quando se utilizam *pipes*, criar um *pipe* ligado à entrada ou saída standard de um outro processo que se lança na altura através de `fork()` - `exec()`. A fim de facilitar esse trabalho a biblioteca standard do C contém as seguintes funções.

```
#include <stdio.h>

FILE *popen(const char *cmdstring, const char *type);
```

Retorna apontar válido, ou `NULL` no caso de erro.

```
int pclose(FILE *fp);
```

Retorna código de terminação de cmdstring ou -1 no caso de erro.

O parâmetro **`cmdstring`** indica o processo a lançar (juntamente com os seus argumentos) ao qual se vai ligar o *pipe*; o parâmetro **`type`** pode ser `"r"` ou `"w"`; se for `"r"` o *pipe* transmite do novo processo para aquele que chamou `popen()` (ou seja quem chamou `popen()` pode ler (`read()`) o *pipe*); se for `"w"` a transmissão faz-se em sentido contrário. A função `popen()` retorna um `FILE *` que representa o lado visível do pipe no processo que chama `popen()` como se fosse um ficheiro da biblioteca standard do C (leituras e escritas com `fread()` e `fwrite()` respectivamente).

Para fechar o *pipe* usa-se `pclose()` que também retorna o código de terminação do processo lançado por `popen()`.

Exemplo - O exemplo anterior escrito com `popen()` (sem verificação de erros)

```
#include    <stdio.h>

#define     PAGER       "/usr/bin/more"         /* programa pager */

int main(int argc, char *argv[])
{
      char  line[MAXLINE];
      FILE  *fpin, *fpout;

      fpin = fopen(argv[1], "r"))
```

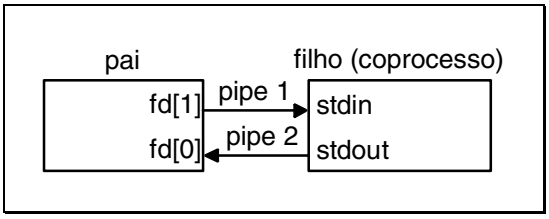
```
      fpout = popen(PAGER, "w"))
      while (fgets(line, MAXLINE, fpin) != NULL)
            fputs(line, fpout);
      pclose(fpout);
      return 0;
}
```

6.3 Coprocessos

Um filtro é um programa que lê informação da sua entrada standard, a modifica, e escreve essa informação modificada na sua saída standard.

Quando um outro programa envia informação para a entrada standard de um filtro e depois recolhe a resposta na sua saída standard, o filtro passa a chamar-se um coprocesso desse programa.

A forma usual de utilizar um coprocesso é lançá-lo como filho de um processo inicial e estabelecer ligações com as suas entrada e saída stantard através de dois *pipes*, como se mostra na figura seguinte.



O coprocesso nunca se apercebe da utilização dos pipes. Simplesmente lê a entrada standard e escreve a sua resposta na saída standard.
No exemplo seguinte temos o código de um coprocesso que lê dois inteiros da entrada standard e escreve a sua soma na saída standard.

```
#include <stdio.h>

int main(void)
{
      int   n, int1, int2;
      char  line[MAXLINE];

      while ( (n = read(STDIN_FILENO, line, MAXLINE)) > 0) {
            line[n] = 0;                    /* terminar linha lida com 0 */
            if (sscanf(line, "%d %d", &int1, &int2) == 2) {
                  sprintf(line, "%d\n", int1 + int2);
                  n = strlen(line);
                  if (write(STDOUT_FILENO, line, n) != n) {
                        fprintf(stderr, "write error"\n);
                        return 1;
                  }
            } else {
                  if (write(STDOUT_FILENO, "invalid args\n", 13) != 13) {
                        fprintf(stderr, "write error\n");
                        return 1;
                  }
            }
      }
      return 0;
}
```

Um programa que usa o anterior como coprocesso pode ser:

```
#include    <unistd.h>
#include    <signal.h>
#include    <sys/types.h>
#include    <stdio.h>

static void sig_pipe(int);                          /* our signal handler */

int main(void)
{
      int   n, fd1[2], fd2[2];
      pid_t pid;
      char  line[MAXLINE];

      signal(SIGPIPE, sig_pipe);
      pipe(fd1);
      pipe(fd2);
      pid = fork();
      if (pid > 0) {                                              /* parent */
            close(fd1[0]);
            close(fd2[1]);
            while (fgets(line, MAXLINE, stdin) != NULL) {
                  n = strlen(line);
                  write(fd1[1], line, n);
                  n = read(fd2[0], line, MAXLINE);
                  if (n == 0) {
                        printf("child closed pipe\n");
                        break;
                  }
                  line[n] = 0;
                  fputs(line, stdout);
            }
            return 0;
      } else {                                                     /* child */
            close(fd1[1]);
            close(fd2[0]);
            if (fd1[0] != STDIN_FILENO) {
                  dup2(fd1[0], STDIN_FILENO);
                  close(fd1[0]);
            }
            if (fd2[1] != STDOUT_FILENO) {
                  dup2(fd2[1], STDOUT_FILENO);
                  close(fd2[1]);
            }
            execl("./add2", "add2", NULL);
      }
}

static void sig_pipe(int signo)
{
      printf("SIGPIPE caught\n");
      exit (1);
}
```

6.4 *Pipes* com nome ou FIFOs

Os FIFOs são por vezes chamados *pipes* com nome e podem ser utilizados para estabelecer canais de comunicação entre processos não relacionados ao contrário dos

pipes, que exigem sempre um ascendente comum entre os processos que ligam.

Quando se cria um FIFO o seu nome aparece no directório especificado no sistema de ficheiros.

A criação de FIFOs faz-se com o seguinte serviço:

```
#include <sys/types.h>
#include <sys/stat.h>

int mkfifo(const char *pathname, mode_t mode);
```

Retorna 0 se houver sucesso e -1 no caso contrário.

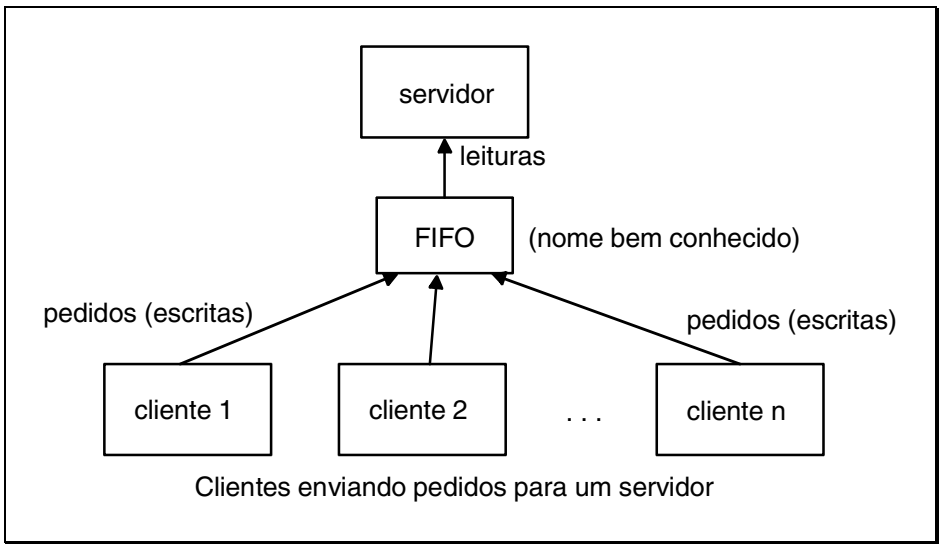
O parâmetro **`pathname`** indica o directório e o nome do FIFO a criar, enquanto que o argumento **`mode`** indica as permissões de acesso ao FIFO (ver serviço `open()` no capítulo 2).
Uma vez criado o FIFO é necessário abri-lo para leitura ou escrita, como se fosse um ficheiro, com o serviço `open()`. Um FIFO suporta múltiplos escritores, mas apenas um leitor.

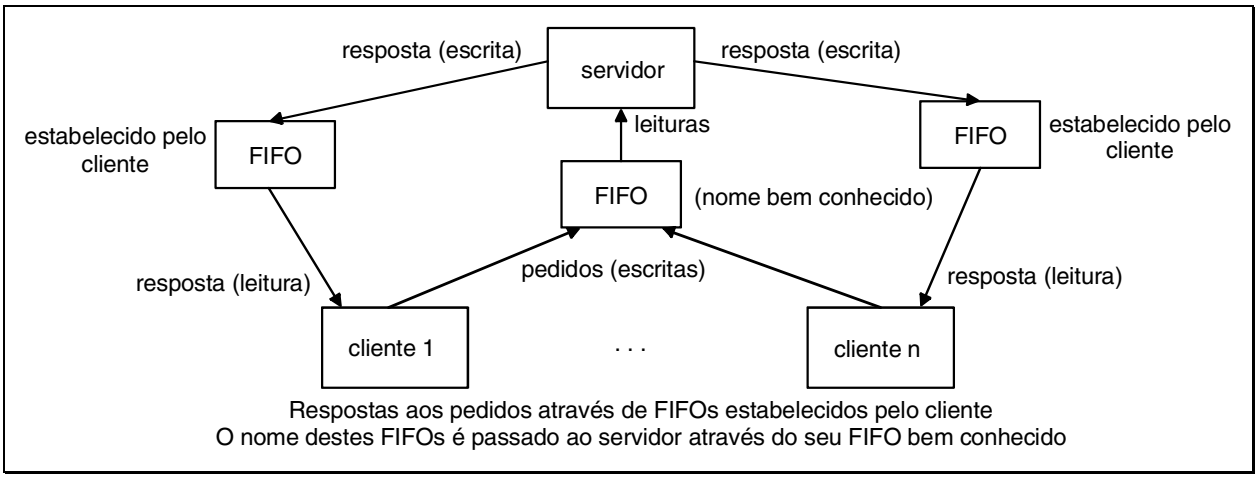
As leituras, escritas, fecho e eliminação de FIFOs fazem-se com os serviços correspondentes para ficheiros (`read()`, `write()`, `close()` e `unlink()`).

Quando se abre um FIFO a *flag* `O_NONBLOCK` de `open()` afecta a sua operação. Se a *flag* não for especificada a chamada a `open()` para leitura ou para escrita bloqueia até que um outro processo faça uma chamada a `open()` complementar (escrita ou leitura respectivamente). Quando a *flag* é especificada, uma chamada a `open()` para leitura retorna imediatamente com sucesso, enquanto que uma abertura para escrita retorna um erro se o FIFO não estiver já aberto para leitura por um outro processo.

O comportamento das leituras e escritas nos FIFOs (com `read()` e `write()`) é semelhante ao dos *pipes*.

As duas figuras seguintes ilustram algumas utilizações de FIFOs no funcionamento de sistemas de processos clientes-servidor.




resposta (escrita)
servidor
resposta (escrita)
estabelecido pelo cliente
FIFO
leituras
FIFO
estabelecido pelo cliente
FIFO
(nome bem conhecido)
resposta (leitura)
pedidos (escritas)
resposta (leitura)
cliente 1
. . .
cliente n
Respostas aos pedidos através de FIFOs estabelecidos pelo cliente
O nome destes FIFOs é passado ao servidor através do seu FIFO bem conhecido

7. Memória partilhada e sincronização em UNIX

7.1. Memória partilhada

Os sistemas Unix derivados do Unix System V, e outros compatíveis (quase todos), definem serviços que permitem partilhar entre vários processos um bloco de memória. Assim aquilo que um processo escreve numa determinada posição desse bloco de memória é imediatamente visto por outros processos que partilhem o mesmo bloco. O pedaço de memória física que constitui o bloco partilhado entra no espaço de endereçamento de cada processo que o partilha como um pedaço de memória lógica (cada processo que partilha o bloco de memória pode vê-lo com endereços lógicos diferentes).

Para partilhar um bloco de memória por vários processos é necessário que um deles crie o bloco partilhado, podendo depois os outros associá-lo ao respectivo espaço de endereçamento. Quando o bloco de memória partilhado não for mais necessário é muito importante libertá-lo explicitamente, uma vez que a simples terminação dos processos que o partilham não o liberta. Todos os sistemas suportam um número máximo de blocos de memória partilháveis. Se não forem convenientemente libertados, quando se atingir esse número máximo, nenhum processo poderá criar mais.

Criação e associação de blocos de memória partilhados:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

int shmget(key_t key, int size, int flag);
```

Retorna um valor positivo (identificador do bloco) no caso de sucesso ou -1 no caso de erro.

Para criar um bloco de memória partilhada usa-se o serviço acima, que deverá retornar um valor inteiro positivo conhecido como o idenficador do bloco de memória partilhada (*shmid* ou *shared memory identifier*); este identificador terá depois de ser usado nos outros serviços que dizem respeito à utilização da memória partilhada.

O valor **`key`** (geralmente um inteiro longo) pode ser a constante `IPC_PRIVATE` ou um valor arbitrário diferente dos já utilizados na criação de outros blocos de memória partilhada.

Quando se usa a constante `IPC_PRIVATE`, o bloco criado só pode ser utilizado em processos que sejam descendentes do processo que cria o bloco, e mesmo para isso é necessário passar-lhes de alguma maneira (p. ex. na linha de comando) o identificador retornado por `shmget()`. Os processos descendentes podem (e devem) usar esse identificador para aceder ao bloco de memória partilhada.

Quando se usa um valor específico para **`key`**, qualquer outro processo (incluindo os descendentes) pode partilhar o bloco (desde que o mesmo tenha permissões compatíveis). Para isso quem cria o bloco tem de incluir em **`flag`** a constante `IPC_CREAT`. Uma vez criado o bloco, outro processo que o queira usar tem também de chamar o serviço `shmget()`, especificando a mesma **`key`** (mas sem `IPC_CREAT` em **`flag`**) para obter o identificador *shmid*. Quando se usa uma **`key`** específica, conhecida de todos os processos que querem usar o bloco de memória partilhada, corre-se o risco, embora remoto (há

alguns biliões de *keys* diferentes), de que outros processos não relacionados já tenham utilizado essa **key** (quando se usa a constante IPC_PRIVATE é sempre criado um novo bloco com um identificador diferente). Para garantir que o sistema assinala um erro quando se está a usar uma **key** idêntica a um bloco já existente é necessário acrescentar (com *or* (|)) a constante IPC_EXCL a **flag**.

O parâmetro **size** especifica em bytes o tamanho do bloco de memória partilhada a criar.

O parâmetro **flag**, além de poder conter os valores IPC_CREAT e/ou IPC_EXCL também serve para especificar as permissões do bloco a criar no que diz respeito à leitura ou escrita por parte do *owner* (utilizador do processo que cria o bloco), *group* ou *others*. Para isso devem-se acrescentar (novamente com *or* (|)) respectivamente as constantes: SHM_R, SHM_W, SHM_R>>3, SHM_W>>3, SHM_R>>6 e SHM_W>>6.

Quando se pretende obter garantidamente uma **key** diferente associada a um determinado processo (proveniente de um ficheiro executável) pode usar-se o seguinte serviço:

```
#include <sys/ipc.h>

key_t ftok(char *pathname, int nr);
```

onde **pathname** será o nome do ficheiro executável que pretende a **key** (e respectivo *path*) e **nr** pode ser um valor entre 0 e 255, para prever diversas instâncias ou diversos blocos criados pelo processo.

Uma vez criado o bloco, e obtido o seu identicador por parte do processo que o quer utilizar, (ou tendo o mesmo sido passado a um filho quando se usa IPC_PRIVATE), é necessário agora mapear o bloco para o espaço de endereçamento do processo e obter um apontador para esse bloco (operação designada por *attach*). Para isso utiliza-se o serviço:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

void *shmat(int shmid, void *addr, int flag);

Retorna um apontador válido no caso de sucesso ou -1 no caso de erro.
```

O serviço retorna um apontador (genérico) para o início do bloco partilhado. O parâmetro **shmid** é o identificar do bloco, obtido em shmget(); o parâmetro **addr** poderá servir para sugerir um endereço lógico de mapeamento, no entanto geralmente usa-se para **addr** o valor 0, para deixar o sistema escolher esse endereço; por sua vez, o parâmetro **flag** pode ser 0 ou conter a constante SHM_RDONLY se se pretender apenas ler o bloco partilhado.

Quando um processo não necessitar de aceder mais ao bloco partilhado deve desassociá-lo do seu espaço de endereçamento com o serviço shmdt(), que significa *shared memory detach*.

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

int shmdt(void *addr);
```

addr é o apontador retornado por `shmat()`.

Retorna 0 no caso de sucesso ou -1 no caso de erro.

Notar que este serviço não destrói o bloco partilhado, nem sequer a terminação de todos os processos que o utilizaram, incluindo o que o criou. Para libertar totalmente um bloco de memória partilhado é necessário usar o serviço seguinte (com **cmd** igual a `IPC_RMID`):

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

int shmctl(int shmid, int cmd, struct shmid_ds *buf);
```

Retorna 0 no caso de sucesso ou -1 no caso de erro.

Além de se indicar o identificador do bloco (em **shmid**) é necessário especificar um "comando" (no parâmetro **cmd**) com a acção a executar, que pode ser:

- `IPC_RMID` - liberta o bloco, quando o último processo que tenha feito um `shmat()` execute o correspondente `shmdt()` ou termine. Após a execução deste comando já não é possível efectuar outras operações de `shmat()`, mesmo que o bloco ainda se mantenha em memória. Só um processo que pertença ao mesmo utilizador que criou o bloco pode executar este comando. O parâmetro **buf** deve ser `NULL` para este comando.
- `IPC_STAT` - Preenche a estrutura apontada por **buf** com informações acerca do bloco (permissões, *pid* do owner, instante de criação, número de associações, etc) (ver `man`).
- `IPC_SET` - Através dos campos `shm_perm.uid`, `shm_perm.gid` e `shm_perm.mode` da estrutura apontada por **buf** permite modificar as permissões, o dono e o grupo do bloco de memória. (ver `man`).

Exemplo 1: Criação e partilha de um bloco de memória entre pai e filho

```
...
int shmid;
int *pt1, *pt2;
pid_t pid;

shmid = shmget(IPC_PRIVATE, 1024, SHM_R | SHM_W);
pid = fork();
if (pid > 0) {
    pt1 = (int *) shmat(shmid, 0, 0);
    ...
    pt1[0] = 20;
    pt1[1] = 30;
    ...
    shmdt(pt1);
    waitpid(pid, ... );
    shmctl(shmid, IPC_RMID, NULL);
    exit(0);
}
else {
    pt2 = (int *) shmat(shmid, 0, 0);
```

```
        ...
        pt2[2] = pt2[0] * pt2[1];
        ...
        shmdt(pt2);
        exit(0);
    }
    ...
```

Exemplo 2: Criação e partilha de um bloco de memória entre quaisquer 2 processos

Processo 1:

```
    ...
    key_t key;
    int shmid;
    int *pt;

    key = ftok("proc1", 0);
    shmid = shmget(key, 1024, IPC_CREAT | IPC_EXCL | SHM_R | SHM_W);
    pt = (int *) shmat(shmid, 0, 0);
    ...
    pt[0] = 20;
    pt[1] = 30
    pt[100] = 0;
    ...
    while (pt[100] != 1) { }
    shmdt(pt);
    shmctl(shmid, IPC_RMID, NULL);
    exit(0);
    ...
```

Processo 2:

```
    ...
    key_t key;
    int shmid;
    int *pt;

    key = ftok("proc1", 0);              /* usa a mesma key */
    shmid = shmget(key, 0, 0);           /* não cria, apenas utiliza */
    pt = (int *) shmat(shmid, 0, 0);
    ...
    pt[2] = pt[0] * pt[1];
    ...
    pt[100] = 1;
    ...
    shmdt(pt);
    exit(0);
```

É claro que o acesso a uma mesma área de memória partilhada por parte de vários processos deverá ser sincronizada, utilizando semáforos ou mutexes.

Recentemente a norma POSIX introduziu um conjunto de novos serviços para a criação, utilização e destruição de blocos de memória partilhada. No entanto como essa norma é muito recente ainda são poucos os sistemas que a suportam.

Outra alternativa para a implementação de blocos de memória partilhada entre processos é o mapeamento de um ficheiro no espaço de endereçamento do processo. Para isso um ficheiro existente em disco é aberto com o serviço `open()`, obtendo-se assim um seu descritor. De seguida, utilizando o serviço `mmap()` (usar o `man` para uma descrição), é

possível mapear esse ficheiro em memória e acedê-lo usando apontadores. Vários processos independentes podem mapear o mesmo ficheiro nos seus espaços de endereçamento (com as permissões adequadas). A operação inversa executa-se com o serviço `munmap()`.

7.2. Mutexes

A norma POSIX que definiu a API de utilização dos *threads* em UNIX também definiu os objectos de sincronização denominados por mutexes e variáveis de condição. Os mutexes podem ser vistos como semáforos que só existem em 2 estados diferentes e servem fundamentalmente para garantir, de forma eficiente, a exclusão mútua de secções críticas de vários *threads* ou processos que executam concorrentemente. Quando um *thread* adquire (ou tranca (*locks*), na nomenclatura usada em Unix) um mutex, a tentativa de aquisição do mesmo mutex por parte de outro *thread* leva a que este fique bloqueado até que o primeiro *thread* liberte o mutex.

Assim, a protecção de uma secção crítica por parte de um *thread* ou processo deverá fazer-se usando as operações de aquisição e libertação (*lock* e *unlock*) de um mesmo mutex:

```
pthread_mutex_lock(&mutex);
    …  /* secção crítica */
pthread_mutex_unlock(&mutex);
```

Um mutex é simplesmente uma variável do tipo `pthread_mutex_t` definido no ficheiro de inclusão `pthread.h`. Antes de poder ser utilizado um mutex tem de ser inicializado. Pode fazer-se essa inicialização quando da sua declaração, usando uma constante pré-definida em `pthread.h` denominada `PTHREAD_MUTEX_INITIALIZER`:

```
pthread_mutex_t mut = PTHREAD_MUTEX_INITIALIZER;
```

Também é possível inicializar um mutex depois de declarado com o seguinte serviço:

```
#include <pthread.h>

int pthread_mutex_init(pthread_mutex_t *mptr,
                       const pthread_mutexattr_t *attr);
```

Retorna 0 se OK ou um valor positivo (com o código do erro) no caso de erro.

O parâmetro **`mptr`** é o endereço da variável que representa o mutex e que se pretende inicializar. O parâmetro **`attr`** permite especificar os atributos que o mutex irá ter. Para inicializar o mutex com os seus atributos por defeito (de forma equivalente à constante `PTHREAD_MUTEX_INITIALIZER`) podemos passar aqui o valor `NULL`.

Na maior parte dos sistemas, por defeito, os mutexes só podem ser utilizados em *threads* diferentes de um mesmo processo. Para utilizar mutexes em processos diferentes estes terão de residir em memória partilhada por esses processos e deverão ser inicializados de modo a que possam ser usados dessa forma. Nos sistemas que suportam este modo de funcionamento (sistemas que definem a constante `_POSIX_THREAD_PROCESS_SHARED` em `unistd.h`) a inicialização terá de ser feita como se mostra no seguinte exemplo:

```
#include <pthread.h>

pthread_mutex_t *mptr;
pthread_mutexattr_t mattr;

…
mptr = …        /* endereço em memória partilhada pelos vários processos */
pthread_mutexattr_init(&mattr);      /* inicializa variável de atributos */
pthread_mutexattr_setpshared(&mattr, PTHREAD_PROCESS_SHARED);
pthread_mutex_init(mptr, &mattr);
```

Uma vez inicializado podemos então operar sobre um mutex utilizando um dos seguintes serviços:

```
#include <pthread.h>

int pthread_mutex_lock(pthread_mutex_t *mptr);
int pthread_mutex_unlock(pthread_mutex_t *mptr);
int pthread_mutex_trylock(pthread_mutex_t *mptr);
```

mptr é o endereço da variável que representa o mutex.

Retornam 0 se OK ou um código de erro positivo no caso contrário.

O serviço `pthread_mutex_lock()` adquire (tranca) o mutex se este estiver livre, ou bloqueia o *thread* (ou processo) que o executa se o mutex já pertencer a outro *thread* até que este o liberte (destranque).
O serviço `pthread_mutex_unlock()` liberta um mutex previamente adquirido. Em princípio esta operação deverá ser efectuada pelo *thread* que detém o mutex.
Por fim, o serviço `pthread_mutex_trylock()` tenta adquirir o mutex; se este estiver livre é adquirido; no caso contrário não há bloqueio e o serviço retorna o código de erro `EBUSY`.

Quando um determinado mutex não for mais necessário, os seus recursos podem ser libertados com o serviço:

```
#include <pthread.h>

int pthread_mutex_destroy(pthread_mutex_t *mptr);
```

Retorna 0 se OK, ou um código de erro positivo no caso contrário.

Exemplo:
Pretende-se um programa *multithreaded* que preencha um array comum com o máximo de 10000000 de entradas em que cada entrada deve ser preenchida com um valor igual ao seu índice. Devem ser criados vários *threads* concorrentes para executar esse preenchimento. Após o preenchimento, um último *thread* deverá verificar a correcção desse preenchimento. O número efectivo de posições do *array* a preencher e o número efectivo de *threads* devem ser passados como parâmetros ao programa.

Segue-se uma solução utilizando as funções `fill()` e `verify()` para os *threads* de preenchimento e verificação. Além disso toma-se nota do número de posições preenchidas por cada *thread* `fill()`. Os *threads* partilham globalmente o *array* a

preencher, a posição actual, e o valor de preenchimento actual (que por acaso é igual ao índice actual):

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>

#define MAXPOS 10000000                /* nr. max de posições */
#define MAXTHRS 100                    /* nr. max de threads */
#define min(a, b) (a)<(b)?(a):(b)

int npos;
pthread_mutex_t mut=PTHREAD_MUTEX_INITIALIZER;      /* mutex para a s.c. */
int buf[MAXPOS], pos=0, val=0;                   /* variáveis partilhadas */

void *fill(void *);
void *verify(void *);

int main(int argc, char *argv[])
{
  int k, nthr, count[MAXTHRS];                    /* array para contagens */
  pthread_t tidf[MAXTHRS], tidv;                  /* tid's dos threads */

  if (argc != 3) {
    printf("Usage: fillver <nr_pos> <nr_thrs>\n");
    return 1;
  }
  npos = min(atoi(argv[1]), MAXPOS);       /* nr. efectivo de posições */
  nthr = min(atoi(argv[2]), MAXTHRS);      /* nr. efectivo de threads */
  for (k=0; k<nthr; k++) {
    count[k] = 0;                          /* criação dos threads fill() */
    pthread_create(&tidf[k], NULL, fill, &count[k]);
  }
  for (k=0; k<nthr; k++) {
    pthread_join(tidf[k], NULL);          /* espera pelos threads fill() */
    printf("count[%d] = %d\n", k, count[k]);
  }
  pthread_create(&tidv, NULL, verify, NULL);
  pthread_join(tidv, NULL);               /* thread verificador */
  return 0;
}

void *fill(void *nr)
{
  while (1) {
    pthread_mutex_lock(&mut);
    if (pos >= npos) {
      pthread_mutex_unlock(&mut);
      return NULL;
    }
    buf[pos] = val;
    pos++; val++;
    pthread_mutex_unlock(&mut);
    *(int *)nr += 1;
  }
}

void *verify(void *arg)
{
  int k;
```

```
  for (k=0; k<npos; k++)
    if (buf[k] != k)          /* escreve se encontrar valores errados */
      printf("buf[%d] = %d\n", k, buf[k]);
  return NULL;
}
```

Se tudo correr bem o programa deverá apenas escrever o número de posições preenchidas por cada *thread* `fill()`.

Experimentar usando por exemplo:

```
fillver 1000000 5
fillver 5000000 5     e
fillver 10000000 5
```

7.3. Variáveis de condição

A utilização de variáveis de condição no Unix é adequada na seguinte situação:
Um determinado *thread* (ou processo) pretende aceder à sua secção crítica apenas quando uma determinada condição booleana (também chamada predicado) se verifica. Enquanto essa condição não se verificar o *thread* pretende ficar bloqueado sem consumir tempo de CPU.
A utilização exclusiva de mutexes não é adequada para esta situação.

Os serviços de variáveis de condição permitem programar esta situação de forma relativamente simples. Vamos supor que 2 *threads* partilham alguma informação e entre aquilo que é partilhado estão duas variáveis `x` e `y`. Um dos *threads* só pode manipular a informação partilhada se o valor de `x` for igual a `y`.

Usando um mutex, o acesso à informação partilhada (que inclui `x` e `y`) por parte do *thread* que necessita que `x` e `y` sejam iguais, poderia ser feito da seguinte forma:

```
while(1) {
    pthread_mutex_lock(&mut);
    if (x == y)
        break;
    pthread_mutex_unlock(&mut);
}
…… /* secção crítica */
pthread_mutex_unlock(&mut);
```

Ora esta solução pode consumir toda a fatia de tempo deste *thread* à espera da condição `(x==y)`, desperdiçando o CPU.

Usando variáveis de condição, este pedaço de código seria substituído por:

```
pthread_mutex_lock(&mut);
while (x != y)
    pthread_cond_wait(&var, &mut);
…… /* secção crítica */
pthread_mutex_unlock(&mut);
```

O que o serviço `pthread_cond_wait()` faz é bloquear este *thread* e ao mesmo tempo (de forma indivisível) libertar o mutex `mut`. Quando um outro *thread* sinalizar a variável de condição `var`, este *thread* é colocado pronto a executar; no entanto antes do serviço `pthread_cond_wait()` retornar terá de novamente adquirir o mutex `mut`. Assim, quando

`pthread_cond_wait()` retorna, o *thread* está garantidamente de posse do mutex `mut`.

Um outro *thread* que modifique de alguma maneira as variáveis `x` ou `y`, possibilitando assim que o estado do predicado que envolve `x` e `y` possa mudar, terá obrigatoriamente de sinalizar a variável de condição `var`, permitindo assim que um *thread* bloqueado em `var` possa novamente testar a condição. O código para fazer isso poderá ser:

```
pthread_mutex_lock(&mut);  /*quando o outro thread bloqueou, libertou mut*/
... ...  /* modifica o valor de x ou y ou ambos */
pthread_cond_signal(&var);                    /* sinaliza a variável var */
pthread_mutex_unlock(&mut);    /* permite que o outro thread adquira mut */
```

Notar que quando um *thread* sinaliza a variável de condição isso não significa que o predicado se satisfaça; daí a necessidade do ciclo `while()` no código que testa o predicado (ver atrás).

7.3.1. Como usar as variáveis de condição e seus serviços

Uma variável de condição é simplesmente uma variável declarada como pertencendo ao tipo `pthread_cond_t` que está definido em `pthread.h`. Da mesma forma que as variáveis que representam os mutexes, as variáveis de condição também têm de ser inicializadas antes de poderem ser utilizadas. A inicialização faz-se de forma em tudo semelhante à que já foi descrita para os mutexes; ou seja, uma variável de condição pode ser inicializada quando da sua declaração, usando a constante pré-definida `PTHREAD_COND_INITIALIZER`, ou então após a declaração, com o serviço:

```
#include <pthread.h>

int pthread_cond_init(pthread_cond_t *cvar,
                      const pthread_condattr_t *attr);
```

`cvar` - endereço da variável de condição a inicializar;
`attr` - endereço de uma variável de atributos; para inicializar com os atributos por defeito deverá usar-se aqui o valor `NULL`.

Retorna 0 se OK, ou um código de erro positivo no caso contrário.

Após a inicialização as variáveis de condição podem ser usadas através dos serviços descritos a seguir. Uma variável de condição tem sempre um mutex associado, como se viu nos exemplos acima. Esse mutex é utilizado no serviço `pthread_cond_wait()`, e por isso terá de lhe ser passado.

```
#include <pthread.h>

int pthread_cond_wait(pthread_cond_t *cvar, pthread_mutex_t *mptr);
int pthread_cond_signal(pthread_cond_t *cvar);
int pthread_cond_broadcast(pthread_cond_t *cvar);
int pthread_cond_destroy(pthread_cond_t *cvar);
```

`cvar` - endereço da variável de condição a inicializar;
`mptr` - endereço do mutex associado.

Retorna 0 se OK, ou um código de erro positivo no caso contrário.

O serviço `pthread_cond_broadcast()` desbloqueia todos os *threads* que nesse momento estão bloqueados na variável **`cvar`**, em vez de desbloquear apenas um *thread* como faz o serviço `pthread_cond_signal()`. No entanto como os *threads* desbloqueados terão de adquirir o mutex antes de prosseguirem, só um deles o poderá fazer. Os outros seguem-se-lhe à medida que o mutex for sendo libertado.

O serviço `pthread_cond_destroy()` deverá ser chamado quando não houver mais necessidade de utilizar a variável de condição por parte de nenhum dos *threads*.

Exemplo:

Thread 1:

```
#include <pthread.h>

pthread_cond_t cvar=PTHREAD_COND_INITIALIZER;
pthread_mutex_t mut=PTHREAD_MUTEX_INITIALIZER;
int x, y, …;      /* variáveis globais */

…
pthread_mutex_lock(&mut);
while (x != y)
    pthread_cond_wait(&cvar, &mut);
…
pthread_mutex_unlock(&mut);
…
pthread_cond_destroy(&cvar);
…
```

Thread 2:

```
…
pthread_mutex_lock(&mut)
x++;
… /* outras ops em vars comuns */
pthread_cond_signal(&var);
pthread_mutex_unlock(&mut);
…
```

7.4. Semáforos

Todos os sistemas Unix derivados do UNIX System V têm uma implementação de semáforos semelhante à que já vimos para a memória partilhada. A norma POSIX, muito recentemente definiu uma adenda também para a implementação de semáforos. No entanto, devido ao pouco tempo da sua existência, esta adenda ainda não foi implementada em alguns dos sistemas Unix actuais.

Passemos a descrever a implementação do UNIX System V. Nesta implementação é necessário criar os semáforos com um serviço próprio e obter um identificador (*semid*). Seguidamente é necessário inicializá-los com um valor positivo ou zero (geralmente pelo processo que os cria), e só depois é possível utilizá-los (usando as operações *wait*, *signal* e outras) através do respectivo identificador. Por fim é necessário libertar explicimente os semáforos criados. (Cada sistema suporta um número máximo de semáforos; quando se atinge esse número não é possível criar mais, sem libertar alguns).

A criação de um conjunto de semáforos faz-se usando o serviço:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

int semget(key_t key, int nsems, int flag);
```

Retorna um valor positivo (identificador do conjunto) no caso de sucesso ou -1 no caso de erro.

Os parâmetros **`key`** e **`flag`** têm exactamente o mesmo significado que foi já descrito para

o serviço `shmget()` (é apenas necessário substituir as constantes `SHM_R` e `SHM_W` por `SEM_R` e `SEM_A` (*alter*)). Este serviço cria um conjunto de semáforos contituído por um número de semáforos indicado em **`nsems`**, que deverá ser maior ou igual a 1. Retorna um identificador do conjunto que deverá ser utilizado nos outros serviços de manipulação dos semáforos.

Após a criação é necessários inicializar os semáforos que fazem parte do conjunto. Isso é feito com o serviço `semctl()`, que também executa outras acções:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

int semctl(int semid, int semnum, int cmd, union semun arg);
```

Retorna -1 no caso de erro; no caso de sucesso depende de **cmd**.

Para alguns dos "comandos" (que são especificados em **`cmd`**) deste serviço usa-se o parâmetro **`arg`** que é uma união entre 3 entidades, cada uma delas usada em "comandos" específicos. A união `semun` define-se então como:

```
union semun {
    int val;                    /* para o comando SETVAL */
    struct semid_ds *buf;       /* para os comandos IPC_STAT e IPC_SET */
    unsigned short *array;      /* para os comandos GETALL e SETALL */
};
```

O parâmetro **`semid`** especifica o identificador de um conjunto de semáforos obtido com `semget()`, enquanto que **`semnum`** espeficica qual o semáforo do conjunto, a que se refere o "comando" **`cmd`**. (**`semnum`** é um valor entre 0 e **`nsems`**`-1`). **`cmd`** pode ser um dos seguintes valores:

- `IPC_STAT` - Preenche a estrutura apontada por **`arg.buf`** com informações acerca do conjunto de semáforos (permissões, *pid* do owner, instante de criação, número de processos bloqueados, etc) (ver `man`).
- `IPC_SET` - Através dos campos `sem_perm.uid`, `sem_perm.gid` e `sem_perm.mode` da estrutura apontada por **`arg.buf`** permite modificar as permissões, o dono e o grupo do conjunto de semáforos. (ver `man`).
- `IPC_RMID` - Remove o conjunto de semáforos do sistema. Esta remoção é imediata. Os processos bloqueados em semáforos do conjunto removido ficam prontos a executar (mas a respectiva operação de *wait* retorna um erro). Este comando só pode ser executado por um processo cujo dono o seja também do conjunto de semáforos.
- `GETVAL` - Faz com que o serviço retorne o valor actual do semáforo **`semnum`**.
- `SETVAL` - Inicializa o semáforo indicado em **`semnum`** com o valor indicado em **`arg.val`**.
- `GETNCNT` - Faz com que o serviço retorne o número de processos bloqueados em operações de *wait* no semáforo **`semnum`**.
- `GETZCNT` - Faz com que o serviço retorne o número de processos bloqueados à espera que o semáforo **`semnum`** se torne 0.
- `GETALL` - Preenche o vector apontado por **`arg.array`** com os valores actuais de todos os semáforos do conjunto. O espaço de memória apontado por **`arg.array`** deverá ser suficiente para armazenar **`nsems`** valores *unsigned short*.
- `SETALL` - Inicializa todos os semáforos do conjunto com os valores indicados no vector **`arg.array`**. Este vector deverá conter pelo menos **`nsems`** valores *unsigned short*.

Os valores com que se inicializam os semáforos devem ser maiores ou iguais a 0.

As operações sobre os semáforos de um conjunto executam-se com o serviço `semop()`. É possível especificar várias operações numa só chamada. Todas as operações especificadas numa única chamada são executadas atomicamente. O serviço `semop()` define-se como:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

int semop(int semid, struct sembuf semoparray[ ], size_t nops);
```

Retorna 0 no caso de sucesso e -1 no caso de erro.

O parâmetro **`semid`** especifica o identificador do conjunto de semáforos. As operações a realizar sobre semáforos do conjunto, especificam-se no vector **`semoparray`**, cujo número de elementos se indica em **`nops`**. Cada operação é especificada num elemento diferente de **`semoparray`**. Cada elemento de **`semoparray`**, por sua vez, é uma estrutura do tipo `struct sembuf`, que se define como:

```
struct sembuf {
    unsigned short sem_num;     /* número do semáforo no conjunto */
    short sem_op;         /* tipo de operação a realizar sobre o semáforo */
    short sem_flg;              /* uma das flags IPC_NOWAIT e/ou SEM_UNDO */
};
```

Cada operação afecta apenas um semáforo do conjunto, que é especificado no campo `sem_num`; o campo `sem_flg` pode conter o valor `SEM_UNDO`, que indica que a operação especificada deve ser "desfeita" no caso do processo que a executa terminar sem ele próprio a desfazer, e/ou o valor `IPC_NOWAIT`, que indica que se a operação fosse bloquear o processo (p. ex. no caso de um *wait*), não o faz, retornando o serviço `semop()` um erro. O campo *sem_op* indica a operação a realizar através de um valor negativo, positivo ou zero:

- negativo - corresponde a uma operação de *wait* sobre o semáforo; o módulo do valor indicado é subtraído ao valor do semáforo; se o resultado for positivo ou 0 continua-se; se for negativo, o processo é bloqueado e não se mexe no valor do semáforo;
- positivo - corresponde a uma operação de *signal* sobre o semáforo; o valor indicado é somado ao valor do semáforo; o sistema verifica se algum dos processos bloqueados no semáforo pode prosseguir;
- zero - significa que este processo deseja esperar que o semáforo se torne 0; se o valor do semáforo for 0 prossegue-se imediatamente; se for maior do que 0 o processo bloqueia até que se torne 0.

Exemplo: Utilização de um semáforo

```
...
key_t key;
int semid;
union semun arg;
struct sembuf semopr;
...
key = ftok("myprog", 1);
```

```
semid = semget(key, 1, SEM_R | SEM_A);      /* cria 1 semáforo */
arg.val = 1;
semctl(semid, 0, SETVAL, arg);              /* inicializa semáforo com 1 */
semopr.sem_num = 0;
semopr.sem_op = -1;                         /* especifica wait */
semopr.sem_flg = SEM_UNDO;
semop(semid, &semopr, 1);                   /* executa wait */
.....                                       /* secção crítica */
semopr.sem_op = 1;                          /* especifica a op. signal */
semop(semid, &semopr, 1);                   /* executa op. signal */
...
semctl(semid, 0, IPC_RMID, arg);            /* liberta semáforo */
...
```

Muito importante:
Existem geralmente 2 comandos, que correm na *shell* do sistema operativo, que permitem listar e libertar semáforos e blocos de memória partilhada quando os processos que os criaram o não fizeram. São eles *ipcs* e *ipcrm* (ver `man`).

8. Sockets

8.1 Introdução

A API (*Application Programming Interface*) de sockets foi desenvolvida para permitir que duas aplicações distintas, correndo em máquinas diferentes, possam comunicar entre si através de um protocolo de rede. Para isso as duas máquinas deverão estar ligadas fisicamente através de uma rede suportada.
Embora os sockets tenham sido desenvolvidos de forma genérica e de modo a poderem suportar múltiplos protocolos de rede, iremos apenas tratar os mais comuns que são os protocolos internet.

Construídos por cima do protocolo base (IP – *internet protocol version 4* (endereços dos nós de 32 bits)) existem dois outros muito utilizados e com propriedades adicionais. São eles:

- TCP – *Transmission Control Protocol* – Trata-se de um protocolo de comunicação orientado à ligação e que contém mecanismos de alta fiabilidade na transmissão de informação nos dois sentidos de uma ligação. Uma vez estabelecida a ligação entre duas aplicações é possível trocar informação entre elas, nos dois sentidos (*full-duplex*), até que a ligação seja fechada.
- UDP – *User Datagram Protocol* – Neste protocolo as aplicações enviam mensagens para destinos bem definidos que terão de estar à escuta. É pois orientado à mensagem (*datagram*) e não fornece qualquer garantia de que a informação atinja o seu destino, nem que o faça pela ordem de envio. Compete à aplicação assegurar-se desses detalhes.

Nas secções que se seguem iremos tratar a utilização de sockets com estes dois protocolos.

8.2 Endereços e *byte order*

Os sockets em UNIX, como veremos mais tarde, são representados por descritores semelhantes aos descritores dos ficheiros, e uma vez estabelecida uma ligação são utilizados para ler e escrever informação. No entanto, antes de se poderem utilizar, é necessário conhecer e especificar o endereço da máquina com quem se pretende comunicar.

No protocolo IP (*version* 4) esse endereço é um número de 32 bits que é geralmente escrito através de 4 valores, correspondentes a cada *byte* (portanto de 0 a 255), separados por um ponto. Assim esses endereços vão de 0.0.0.0 a 255.255.255.255. Todas as máquinas ligadas em rede têm um endereço IP diferente.
Por outro lado uma mesma máquina pode manter simultanemente várias ligações activas com outras máquinas. Para distinguir essas ligações umas das outras, e para o Sistema Operativo poder distribuir a informação que chega pelas várias ligações activas, introduziu-se a noção de *porto* para identificar de forma inequívoca cada uma das ligações. Um porto, nos protocolos IP, é um número de 16 bits, o que significa que poderá haver 65536 portos diferentes numa mesma máquina. Assim o endereço de um socket (extremidade de uma ligação), nestes protocolos, consta de um endereço de 32 bits mais um porto de 16 bits.

O endereço completo de um socket é especificado através de uma estrutura (`struct sockaddr_in`) que contém campos para o endereço IP da máquina e para o porto. Esta

estrutura é um caso particular (para a família de endereços internet) de uma estrutura mais genérica (`struct sockaddr`) que é a que aparece como parâmetro nas definições dos serviços de sockets que necessitam de um endereço.

Assim a definição da estrutura **`sockaddr_in`** é a seguinte (definida em `<netinet/in.h>`):

```
struct sockaddr_in {
  sa_family_t     sin_family;
  in_port_t       sin_port;
  struct in_addr  sin_addr;
  char            sin_zero[8];
}
```

Os tipos `sa_family_t` e `in_port_t` são geralmente equivalentes a `unsigned short` (16 bits), enquanto que o tipo `struct in_addr` é definido como se segue, de acordo com as recomendações do último *draft* (versão 6.6 - ainda não aprovado) das normas Posix.1g, que tentam normalizar para todos os sistemas Unix a API de sockets:

```
struct in_addr {
  in_addr_t       s_addr;
}
```

O tipo `in_addr_t` é geralmente equivalente a `unsigned long` (32 bits). A estrutura **`in_addr`** foi durante muito tempo definida como uma união que permitia o acesso aos bytes individuais do endereço IP, mas actualmente considera-se desnecessário, mantendo-se no entanto, por questões de compatibilidade, o tipo do campo **`sin_addr`** da estrutura **`sockaddr_in`** como `struct in_addr`. O campo **`sin_zero`**, com o tamanho de 8 bytes (perfazendo um total de 16 bytes para a estrutura **`sockaddr_in`**) não é utilizado e deve ser sempre inicializado com 0.

A prática corrente, quando se utiliza uma estrutura **`sockaddr_in`** é começar por inicializá-la totalmente a zero, antes de preencher os outros campos:

```
struct sockaddr_in sin;
…
memset(&sin, 0, sizeof(sin));
…
```

A função `memset()` da biblioteca standard do C, é definida como:

```
#include <string.h>

void *memset(void *dest, int c, size_t count);
```

Inicializa **`count`** bytes da memória apontada por **`dest`** com o valor **`c`**. Retorna **`dest`**.

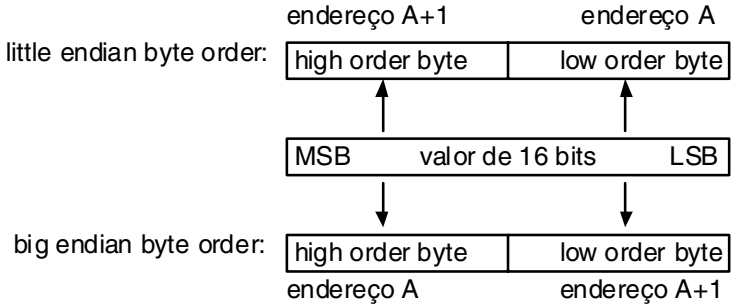
A estrutura mais genérica **`sockaddr`** que aparece nos serviços da API de sockets é simplesmente definida como:

```
struct sockaddr {
  sa_family_t     sa_family;
  char            sa_data[14];
}
```

mantendo o tamanho total de 16 bytes.

Voltando à estrutura **`sockaddr_in`**, o campo **`sin_family`** indica a família de endereços daquele que vai ser especificado a seguir. Para os protocolos internet (IP version 4) esse campo deverá ter sempre o valor da constante **`AF_INET`** (*address family internet*).
O campo **`sin_port`** deverá conter o valor do porto, enquanto que o campo **`sin_addr`** conterá o valor do endereço IP de 32 bits.

Os processadores correntes representam os valores numéricos multibyte (codificados em binário) de forma diferente. Assim os processadores ditos *little endian* representam os valores numéricos multibyte (p.ex. valores de 16 ou 32 bits) colocando os bytes menos significativos nos endereços de memória mais baixos, enquanto que os processadores ditos *big endian* representam esses valores exactamente ao contrário (bytes menos significativos nos endereços mais elevados). Veja-se a figura seguinte para um valor de 16 bits.



Como os sockets permitem a comunicação entre máquinas diferentes que possivelmente podem representar a informação numérica de forma diferente é necessário um mecanismo para compatibilizar essa representação.
Assim, qualquer aplicação, antes de enviar informação numérica (as *strings* compostas por caracteres de 1 byte não têm esse problema), deverá convertê-la para uma representação bem definida a qual se designou por *network byte order* (é idêntica à dos processadores *big endian*). Da mesma forma, sempre que é recebida informação numérica através de um socket, esta deve ser convertida pela aplicação, do formato recebido (*network byte order*) para o formato usado no sistema (*host byte order*). Se todas as aplicações forem codificadas desta maneira garante-se a compatibilidade da informação em todos os sistemas.
A API de sockets contém uma série de serviços para efectuar essas conversões, que iremos ver a seguir. Os campos **`sin_port`** e **`sin_addr`** da estrutura **`sockaddr_in`** devem ser sempre preenchidos em *network byte order*.

Para converter valores numéricos de 16 e 32 bits da representação local para *network byte order* deveremos usar as funções `htons()` (*host to network short*) e `htonl()` (*host to network long*) respectivamente, enquanto que para efectuar as conversões contrárias se deverá usar `ntohs()` (*network to host short*) e `ntohl()` (*network to host long*).

```
#include <netinet/in.h>

uint16_t htons(uint16_t host16bitvalue);
uint32_t htonl(uint32_t host32bitvalue);
uint16_t ntohs(uint16_t net16bitvalue);
uint32_t ntohl(uint32_t net32bitvalue);
```

As duas primeiras retornam valores em *network byte order*. As duas últimas retornam valores em *host byte order*.

Geralmente quando se especificam os endereços IP das máquinas ligadas em rede usam-se strings em que o valor de cada byte do endereço é separado dos outros por um ponto. Por exemplo: "193.136.26.118"

A API de sockets contém também serviços para converter a representação numérica de 32 bits destes endereços na representação em string, mais habitual. Assim a função `inet_aton()` converte um endereço especificado numa string com os quatros valores separados por ponto para a representação binária de 32 bits já em *network byte order*, enquanto que a função `inet_ntoa()` faz o contrário.

```
#include <arpa/inet.h>

int inet_aton(const char *straddr, struct in_addr *addrptr);
```

Converte um endereço IP na forma de string e especificado em **`straddr`** para o formato binário de 32 bits em *network byte order*, preenchendo a estrutura **`in_addr`** apontada por **`addrptr`** com esse resultado. Retorna 1 se **`straddr`** for válido e 0 se ocorrer um erro. O parâmetro **`addrptr`** pode ser `NULL`. Neste caso o resultado não é produzido, havendo apenas uma validação da string **`straddr`**.

```
char *inet_ntoa(struct in_addr inaddr);
```

Converte o endereço IP especificado em **`inaddr`** em *network byte order* para uma representação em string (*dotted-decimal*). A string retornada reside em memória estática e é reutilizada numa próxima chamada.

Exemplo: preenchimento de uma estrutura **`sockaddr_in`** com o endereço "193.136.26.118" e porto 5001.

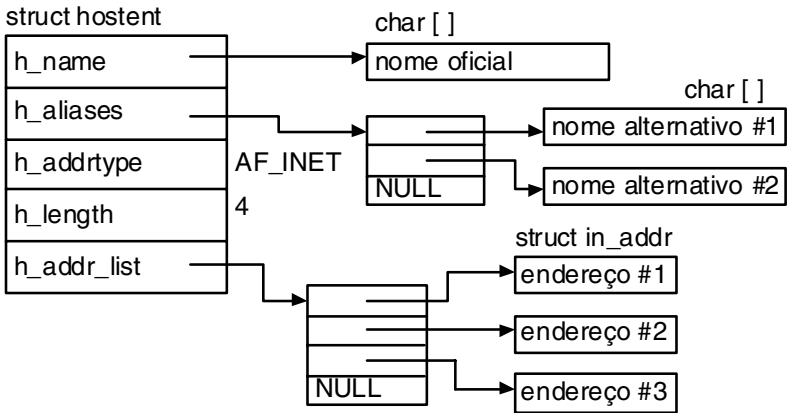
```
struct sockaddr_in sin;
…
memset(&sin, 0, sizeof(sin));
sin.sin_family = AF_INET;
sin.sin_port = htons(5001);
inet_aton("193.136.26.118", &sin.sin_addr);
…
```

8.3 DNS - Domain Name Services

A maior parte das vezes os endereços das máquinas com quem pretendemos comunicar são especificados através de nomes, uma vez que estes são muito mais fáceis de recordar do que os endereços numéricos. Esses nomes, como é sabido podem ter vários componentes separados por pontos.
No entanto a estrutura **`sockaddr_in`** só admite como endereços os valores numéricos de 32 bits (em *network byte order*). Para obter os endereços numéricos a partir dos nomes é necessário fazer uma consulta a um servidor DNS. É também possível obter o nome de uma máquina e seus *aliases* a partir do endereço numérico, basta para isso um novo tipo de consulta a um servidor DNS.
Estes dois tipos de consultas aos servidores DNS são desencadeados pelos serviços `gethostbyname()` e `gethostbyaddr()`. Qualquer um destes serviços retorna um apontador para uma estrutura, preenchida com a informação solicitada.
Essa estrutura tem o nome de **`hostent`**, é definida em `<netdb.h>`, e tem pelo menos os seguintes campos:

```
struct hostent {
  char *h_name;         /* nome oficial da máquina */
  char **h_aliases;     /* apontador para array de nomes alternativos */
  int h_addrtype;       /* tipo de endereço: AF_INET ou AF_INET6 */
  int h_length;         /* tamanho do endereço em bytes: 4 ou 16 */
  char **h_addr_list;   /* apontador para array de endereços da máquina */
}
```

O campo **h_addr_list** aponta para um array de apontadores para estruturas **in_addr** (se **h_addrtype** for AF_INET) contendo os endereços da máquina, no formato numérico e em *network byte order*. A lista contém mais do que um endereço se a máquina possuir mais do que uma placa de interface de rede (máquina *multihomed*) e assim mais do que um endereço. A figura seguinte ajuda a compreender esta estrutura.



Os serviços gethostbyname() e gethostbyaddr() são então definidos como:

```
#include <netdb.h>

struct hostent *gethostbyname(const char *hostname);
```

Consulta o servidor DNS a partir do nome de uma máquina especificado na string **hostname**.

```
struct hostent *gethostbyaddr(const char *addr, size_t len, int family);
```

Consulta o servidor DNS a partir de um endereço passado através do apontador **addr**. No caso de um endereço IP (v. 4 de 32 bits) ele deve ser passado numa estrutura **in_addr** em *network byte order*, **len** deve ser 4 e **family** deve ser AF_INET. Ambos os serviços retornam um apontador para uma estrutura **hostent** devidamente preenchida ou NULL no caso de erro.

Estes serviços, no caso de erro não afectam, como é habitual, a variável global **errno**, mas sim uma outra variável global definida em <netdb.h> e chamada **h_errno**. Pode obter-se uma string com a descrição do erro usando o serviço hstrerror():

```
#include <netdb.h>

char *hstrerror(int h_errno);
```

Retorna uma string com a descrição correspondente ao erro codificado em **h_errno**.

Exemplo: Imprimir a informação retornada pelo servidor DNS relativa a máquinas cujos nomes são passados como parâmetros ao programa.

```
#include <stdio.h>
#include <netdb.h>
#include <arpa/inet.h>

int main(int argc, char *argv[])
{
  char *ptr, **pptr;
  struct hostent *hptr;

  while (--argc > 0) {
    ptr = *++argv;
    if ( (hptr = gethostbyname(ptr)) == NULL ) {
      printf("gethostbyname error for host: %s: %s\n", ptr,
             hstrerror(h_errno));
      continue;
    }
    printf("Official hostname: %s\n", hptr->h_name);

    for (pptr = hptr->h_aliases; *pptr != NULL; pptr++)
      printf("\talias: %s\n", *pptr);

    if (hptr->h_addrtype == AF_INET)
      for (pptr = hptr->h_addr_list; *pptr != NULL; pptr++)
        printf("\taddress: %s\n", inet_ntoa(*(struct in_addr *)*pptr));
    else
      printf("Unknown address type\n");
  }
  return 0;
}
```

Se uma aplicação necessitar do nome da máquina onde executa poderá usar o serviço `gethostname()`:

```
#include <unistd.h>

int gethostname(char *name, size_t namelen);
```

Preenche o buffer **`name`** com tamanho máximo **`namelen`** com o nome da máquina actual. Retorna 0 se OK e -1 no caso de erro.

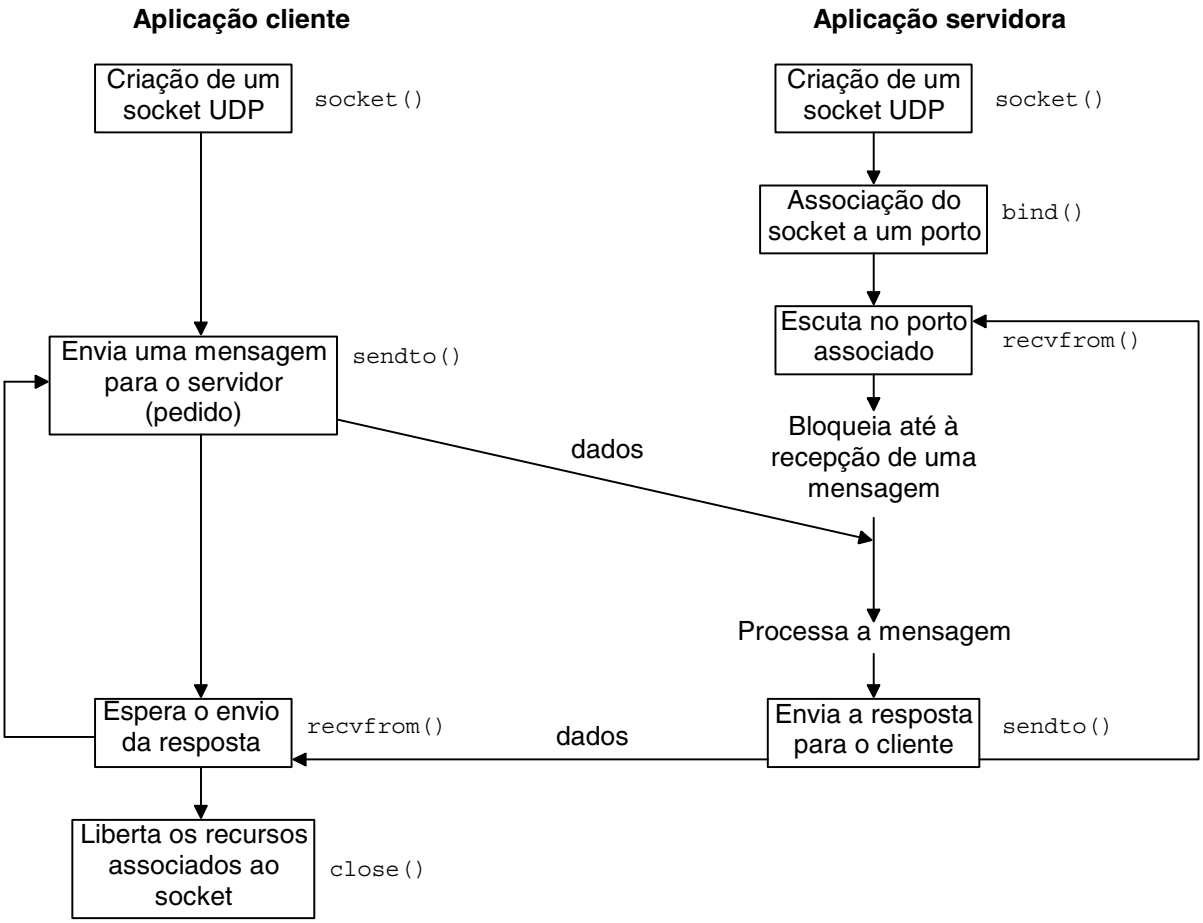
8.4 Comunicações com sockets UDP

Como já atrás se disse o protocolo UDP (*User Datagram Protocol*) é orientado à mensagem. Uma das aplicações envia uma mensagem para um endereço IP e porto conhecidos, tendo o outro lado da comunicação de estar à escuta nesse porto.

Apesar de não haver o estabelecimento de uma ligação duradoura entre as duas aplicações que utilizam o protocolo UDP é comum haver a troca de mais do que uma mensagem entre elas, sendo comum também adoptar-se um modelo servidor-cliente para a troca de mensagens. Assim a aplicação servidora coloca-se num estado de escuta num porto conhecido, ficando à espera que lhe chegue uma mensagem. Após a chegada duma mensagem é identificada a sua origem, e consoante o seu significado, segue-se normalmente uma resposta para o endereço e porto de origem, que entretanto deve ficar à escuta dessa resposta. Poderá haver troca de mensagens subsequentes entre as duas

aplicações.

Um esquema da arquitectura de aplicações clientes e servidor que utilizam sockets UDP para as suas comunicações pode então ver-se na figura seguinte.



Ambas as aplicações, cliente e servidora, têm de criar um socket, que irá representar o meio capaz de aceitar mensagens para envio e mensagens recebidas de outras aplicações. O socket é representado por um descritor que é em tudo semelhante aos descritores de ficheiros (um número inteiro e geralmente pequeno).

Quando da criação do socket há apenas necessidade de especificar algumas das suas propriedades. Outras propriedades necessárias são especificadas mais tarde, através de outros serviços, e quando já se conhece o descritor do socket.

Um socket é criado invocando o serviço `socket()` e é necessário especificar qual a família de protocolos a utilizar nas comunicações através desse socket, o tipo de socket, e para algumas famílias e tipos também o protocolo.

Obtido o descritor de um socket, este ainda não está pronto a comunicar. Dependendo do tipo e protocolo especificados na criação do mesmo, poderá ser necessário especificar outras propriedades, tais como endereços IP e portos associados, antes do mesmo estar apto a executar operações de recebimento e envio de informação.

Para alguns tipos de sockets essas operações executam-se com os mesmos serviços já utilizados para a leitura e escrita em ficheiros, ou seja, `read()` e `write()`.

Assim, temos a seguinte definição para o serviço `socket()`:

```
#include <sys/socket.h>

int socket(int family, int type, int protocol);
```

Para as redes internet e conjuntos de protocolos TCP/IP (v.4) a família a especificar no parâmetro **`family`** é representada pela constante `PF_INET` (*protocol family internet*).
O parâmetro **`type`** indica o tipo de socket que pretendemos criar. Os dois tipos que iremos ver são:

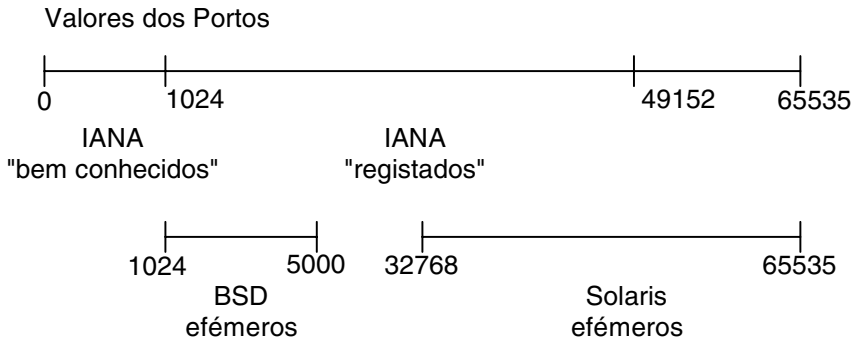
- sockets UDP, especificados através da constante `SOCK_DGRAM`
- sockets TCP, especificados através da constante `SOCK_STREAM`

Como estes tipos só usam um protocolo cada, o parâmetro **`protocol`** deve ser sempre 0. Retorna -1 em caso de erro, e um valor positivo, que é o descritor do socket, em caso de sucesso.

Todos os sockets após a sua criação têm de ser associados a um endereço IP e porto locais à máquina onde corre a aplicação. Em algumas circunstâncias essa associação é automática e feita pelo sistema operativo. Noutras circunstâncias tem de ser feita explicitamente pelo programador. Em regra, numa aplicação servidora, é necessário efectuar essa associação explicitamente antes de se passar à escuta de informação a ler. As aplicações clientes poderão, por vezes, deixar o sistema operativo escolher o porto associado.

A associação explícita de um endereço IP e porto locais a um socket é feita usando o serviço `bind()`. Se o servidor que cria o socket fornecer um serviço bem conhecido e standard (p. ex. ftp, telnet, smtp (*mail*), etc) o porto a associar é também standard e encontra-se provavelmente já definido pela IANA (*Internet Assigned Numbers Authority*). Estes portos têm um valor entre 0 e 1023 e geralmente só uma aplicação pertencente à *root* os pode especificar na chamada a `bind()`.

Quando é o sistema operativo a associar automaticamente um porto a um socket, escolherá um valor dito efémero, pertencente a uma gama que depende do sistema operativo. Por exemplo, nos sistemas derivados da arquitectura BSD essa gama vai de 1024 a 5000; no sistema Solaris essa gama vai de 32768 a 65535. A escolha do sistema pode recair em qualquer valor da gama dos portos efémeros, havendo a garantia de que o porto escolhido pelo sistema não está nesse momento a ser utilizado por nenhum outro socket. (Constitui um erro tentar associar um porto já utilizado por outro socket a um novo socket).



Quando pretendemos escrever um novo servidor para um serviço não standard escolhe-se geralmente para ele um porto fora da gama efémera e também fora da gama reservada

para serviços "bem conhecidos" (por exemplo pode-se escolher um valor acima de 5000). No entanto a IANA mantém uma lista de portos ditos "registados" na gama de 1024 a 49151, que embora não directamente controlados por ela, foram algum dia usados na implementação de algum servidor não "bem conhecido". Essa lista tem apenas um carácter informativo (por exemplo alguns servidores X usam portos na gama 6000 a 6063).

O serviço `bind()`, que associa um socket, através do seu descritor, a um endereço IP e porto local, tem então a seguinte definição:

```
#include <sys/socket.h>

int bind(int sockfd, const struct sockaddr *addr, socklen_t addrlen);
```

Associa ao socket **`sockfd`** um endereço especificado através de **`addr`**. O parâmetro **`addr`** deve ser um apontador para uma estrutura `sockaddr_in` (para endereços internet) devidamente preenchida com um endereço IP e porto (em *network byte order*). O parâmetro **`addrlen`** indica em bytes o tamanho da estrutura apontada por **`addr`**.
Retorna 0 se a associação ocorreu e -1 em caso de erro.

Geralmente o endereço IP especificado na estrutura apontada por **`addr`** é a constante `INADDR_IN`. Nas máquinas com apenas uma carta de rede esta constante corresponde sempre ao seu endereço. Nas máquinas com várias cartas (*multihomed*) a utilização desta constante permite que a comunicação através do socket se faça utilizando qualquer das cartas da máquina (quer no envio, quer na recepção). Nestas máquinas, se se especificar um endereço IP concreto, limita-se o envio e a recepção de informação à carta de rede correspondente.

Exemplo:

```
int sock;
struct sockaddr_in sadr;
…
sock = socket(PF_INET, SOCK_DGRAM, 0);      // criação do socket
memset(&sadr, 0, sizeof(sadr));
sadr.sin_family = AF_INET;
sadr.sin_addr.s_addr = htonl(INADDR_IN);    // conversão para net byte order
sadr.sin_port = htons(5001);                // especificação do porto 5001
bind(sock, (struct sockaddr *) &sadr, sizeof(sadr));
…
```

O envio e recepção de mensagens através de sockets UDP faz-se usando os serviços `sendto()` e `recvfrom()`. Estes serviços permitem que se especifique o endereço IP e porto de destino (no caso de `sendto()`) ou de onde provém a mensagem (no caso de `recvfrom()`).

Quando se chama `recvfrom()` é já obrigatório que o socket esteja associado a um endereço IP e porto locais. Além disso, esta chamada bloqueia o processo até que chegue uma mensagem destinada ao endereço IP e porto locais associados ao socket.

Quando se chama `sendto()` é necessário especificar o destino, e se o socket ainda não tiver um porto associado o sistema operativo escolherá automaticamente um porto efémero para o socket. Esse porto permanece associado ao socket até que este seja

fechado. O endereço IP e porto associados (de origem) são então enviados, juntamente com a mensagem, para o destino e recolhidos por `recvfrom()`.

As definições de `recvfrom()` e `sendto()` são então:

```
#include <sys/socket.h>

ssize_t recvfrom(int sockfd, void *buff, ssize_t nbytes, int flags,
                 struct sockaddr *from, socklen_t *addrlen);

ssize_t sendto(int sockfd, const void *buff, ssize_t nbytes, int flags,
               const struct sockaddr *to, socklen_t addrlen);
```

- **sockfd** indica o descritor do socket a ser utilizado na transferência da mensagem.
- **buff** é um apontador para um buffer de bytes a receber ou a enviar; no caso de `sendto()` o buffer deve já conter a informação a enviar.
- **nbytes** indica o tamanho máximo do buffer no caso de `recvfrom()` e o número de bytes a enviar no caso de `sendto()`.
- **flags** é um parâmetro que pode modificar o comportamento destes dois serviços; o comportamento por defeito obtém-se passando o valor 0.
- **from** e **to** são apontadores para uma estrutura **sockaddr_in** (endereços IP); no caso de `recvfrom()` essa estrutura (apontada por **from**) será preenchida com o endereço e porto de origem; no caso de `sendto()` a estrutura apontada por **to** deverá estar previamente preenchida com o endereço e porto de destino.
- **addrlen** indica o tamanho em bytes da estrutura apontada por **from** ou **to**; no caso de `recvfrom()` **addrlen** é um apontador para a variável que contém esse tamanho e deve ser previamente preenchida com o valor correcto; quando `recvfrom()` retorna, a variável apontada por **addrlen** pode ser modificada.

Se a aplicação que recebe a mensagem não necessitar de responder, e assim também não necessitar de conhecer a origem, os apontadores **from** e **addrlen** de `recvfrom()` podem ser `NULL`.

Ambos os serviços retornam o número de bytes efectivamente recebidos ou enviados, ou -1 no caso de erro.

Quando uma aplicação já não necessitar de efectuar mais comunicações através de um socket, deverá fechá-lo para libertar recursos. Em qualquer caso o sistema operativo fechará os sockets abertos quando o processo terminar. O fecho de um socket executa-se com o já conhecido serviço `close()`.

```
#include <unistd.h>

int close(int sockfd);
```

Retorna 0 no caso de sucesso e –1 no caso de erro.

8.4.1 Exemplo de servidor e cliente com sockets UDP

Apresenta-se de seguida um exemplo de servidor e cliente que comunicam através de sockets UDP. O exemplo implementa um servidor e cliente de *daytime*. O servidor fica à espera de receber uma mensagem (o conteúdo não interessa) e responde com a indicação da data e hora locais do sistema. O serviço de *daytime* é "bem conhecido" e tem porto

atribuído pela IANA (porto 13). No entanto esse porto só pode ser utilizado (no serviço `bind()`) pelo utilizador *root*. Assim iremos usar no exemplo um outro porto.

Servidor de *daytime*:

```
#include <stdio.h>
#include <string.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <netinet/in.h>
#include <time.h>

#define SERV_PORT 9877
#define MAXLINE 1024

void main(void)
{
  int sockfd;
  struct sockaddr_in servaddr, cliaddr;
  ssize_t n;
  socklen_t len;
  time_t ticks;
  char buff[MAXLINE];

  sockfd = socket(PF_INET, SOCK_DGRAM, 0);

  memset(&servaddr, 0, sizeof(servaddr));
  servaddr.sin_family = AF_INET;
  servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
  servaddr.sin_port = htons(SERV_PORT);
  bind(sockfd, (struct sockaddr *) &servaddr, sizeof(servaddr));

  for ( ; ; ) {
    len = sizeof(cliaddr);
    n = recvfrom(sockfd, buff, MAXLINE, 0, (struct sockaddr *) &cliaddr,
                 &len);
    printf("datagram from: %s - %d\n", inet_ntoa(cliaddr.sin_addr),
           ntohs(cliaddr.sin_port));
    ticks = time(NULL);
    sprintf(buff, "%.24s\r\n", ctime(&ticks));
    sendto(sockfd, buff, strlen(buff), 0, (struct sockaddr *) &cliaddr,
           len);
  }
}
```

Neste servidor cria-se um socket UDP, preenche-se uma estrutura **`sockaddr_in`** com o endereço e porto local (escolheu-se para este o valor 9877) e associa-se esta estrutura ao socket com `bind()`. Seguidamente entra-se num ciclo infinito onde se espera receber uma mensagem no porto associado. Quando esta é recebida imprime-se o endereço e porto de origem, determina-se o tempo actual (`time()`) e converte-se para um formato *ascii* conveniente (`ctime()`). Por fim envia-se esse tempo para o endereço e porto de origem, repetindo-se o ciclo.
No código anterior não se incluiram os testes de possíveis erros para tornar o código mais claro, mas numa aplicação séria todos esses possíveis erros devem ser testados.

Mostra-se a seguir um possível cliente que interroga este servidor. O porto é conhecido do cliente, mas este espera que o nome da máquina onde o servidor está a executar lhe seja fornecido na linha de comando.

Cliente de *daytime*:

```
#include <stdio.h>
#include <string.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <netinet/in.h>
#include <netdb.h>

#define SERV_PORT 9877
#define MAXLINE 1024

int main(int argc, char *argv[])
{
  int sockfd;
  struct sockaddr_in servaddr;
  ssize_t n;
  char buff[MAXLINE];
  struct hostent *hostp;

  if (argc != 2) {
    printf("Usage: daytime_c <hostname>\n");
    return 1;
  }

  hostp = gethostbyname(argv[1]);
  if (hostp == NULL) {
    printf("Host <%s> unknown!\n", argv[1]);
    return 1;
  }

  memset(&servaddr, 0, sizeof(servaddr));
  servaddr.sin_family = AF_INET;
  servaddr.sin_port = htons(SERV_PORT);
  servaddr.sin_addr = *(struct in_addr *)(hostp->h_addr_list[0]);

  printf("Asking time from: %s - %d\n", inet_ntoa(servaddr.sin_addr),
         ntohs(servaddr.sin_port));

  sockfd = socket(PF_INET, SOCK_DGRAM, 0);

  sendto(sockfd, "", 1, 0, (struct sockaddr *) &servaddr,
         sizeof(servaddr));
  n = recvfrom(sockfd, buff, MAXLINE-1, 0, NULL, NULL);
  close(sockfd);
  buff[n] = 0;
  printf("%s", buff);
  return 0;
}
```

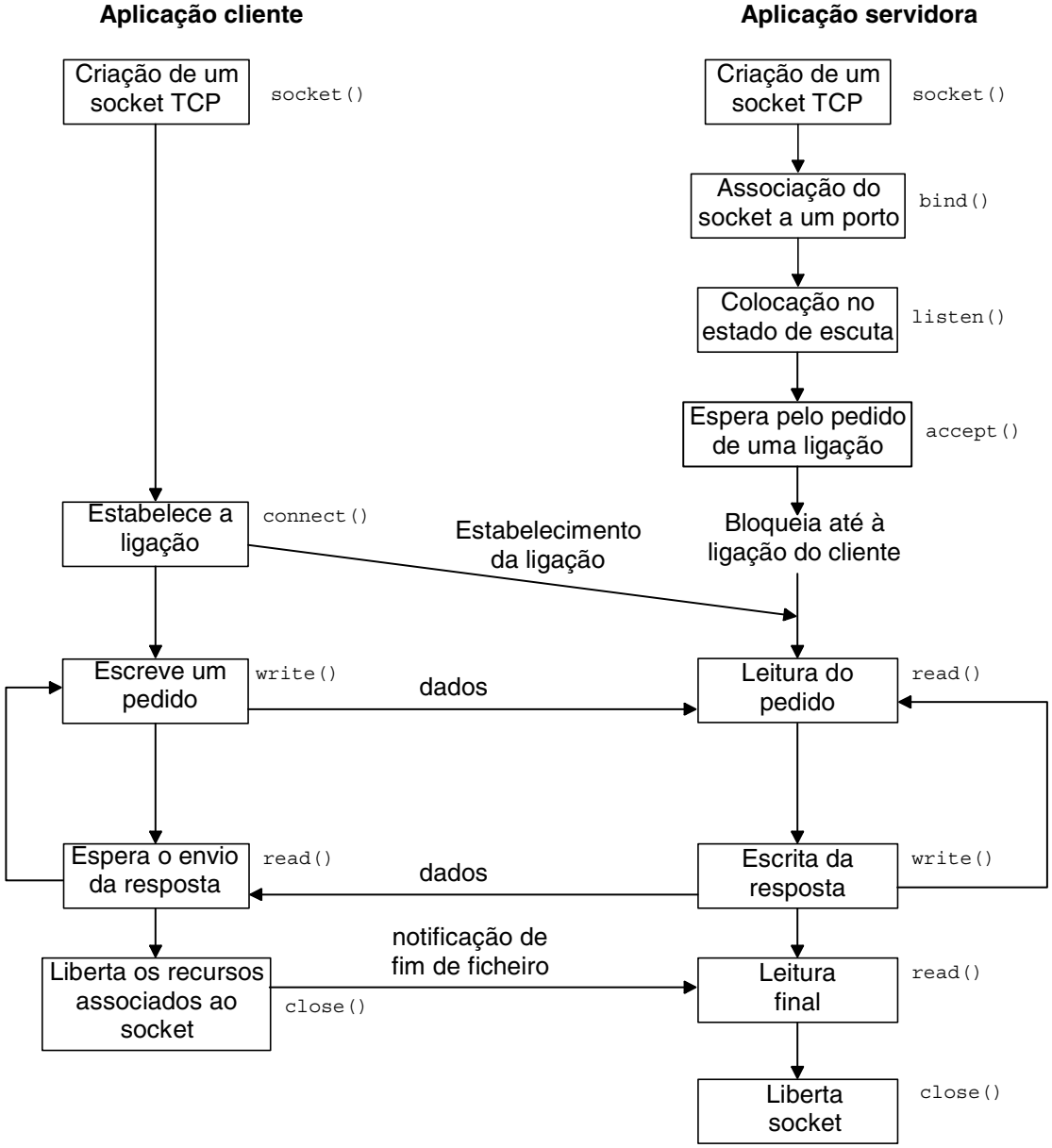
O cliente começa por interrogar os serviços de DNS para obter o endereço IP da máquina cujo nome lhe é passado na linha de comando (através do serviço `gethostbyname()`). Usando esse endereço e o porto conhecido do servidor, preenche uma estrutura **`sockaddr_in`**. Essa estrutura é depois usada para enviar uma mensagem de 1 byte (apenas o valor 0 que termina uma string vazia) para esse destino. Seguidamente espera-se a resposta do servidor e imprime-se o resultado. Quando se chama `sendto()` para enviar a mensagem de 1 byte para o servidor, o sistema operativo associa automaticamente (uma vez que este socket não teve um *bind*) um porto efémero local ao socket do cliente, e envia ao servidor o endereço IP de origem e esse porto efémero.

Reparar que a chamada a `recvfrom()` não toma nota do endereço e porto que vêm do servidor. O socket é fechado quando não se necessita mais dele.

8.5 Comunicações com sockets TCP

Quando clientes e servidores necessitam de trocar maiores volumes de informação ou quando se pretende uma comunicação inteiramente fiável dever-se-á usar o protocolo TCP e respectivos sockets. Este tipo de protocolo e comunicações são orientados à ligação, havendo necessidade de primeiro estabelecer esta ligação para depois se trocar informação e dados entre as duas aplicações ligadas.

Pode ver-se na figura seguinte o esquema geral das arquitecturas de um cliente e servidor TCP.



A primeira tarefa das aplicações que desejam comunicar através de sockets TCP é a criação de um socket deste tipo. Essa criação, como se viu, faz-se com o serviço

`socket()`, especificando como parâmetros as constantes `PF_INET` e `SOCK_STREAM` (ver mais atrás). O terceiro parâmetro é novamente 0.

Na aplicação servidora (aquela que fica à espera de uma ligação e pedidos por parte das aplicações clientes), depois da criação de um socket do tipo TCP, é também necessária a associação de um endereço IP e porto locais para o recebimento de pedidos de ligação. Essa associação faz-se, à semelhança dos sockets UDP, com o serviço `bind()`.

Depois da associação a um endereço e porto há que colocar o socket num estado capaz de receber pedidos de ligação provenientes de outras aplicações. Por defeito, após a sua criação, os sockets TCP apenas são capazes de originar pedidos através do serviço `connect()`, usado pelas aplicações clientes. Para poderem vir a receber esses pedidos ter-se-á de invocar o serviço `listen()`. Este serviço permite que o sistema operativo fique à escuta no porto associado ao socket, crie uma fila onde são colocados os pedidos de ligação pendentes ainda não completos, à medida que vão sendo solicitados pelos clientes, proceda ao *handshake* definido pelo protocolo TCP, e finalmente mova as ligações completamente estabelecidas para uma outra fila de ligações completas. No entanto o serviço não coloca o socket no estado capaz de receber e enviar dados.

```
#include <sys/socket.h>

int listen(int sockfd, int backlog);
```

Coloca o socket **`sockfd`** num estado capaz de vir a receber pedidos de ligação. O parâmetro **`backlog`** especifica de algum modo o tamanho das filas, estabelecidas pelo sistema operativo, onde vão sendo colocados os pedidos de ligação, provenientes dos clientes, incompletos ou já completados. O valor a especificar é interpretado de modo diferente consoante as implementações e depende da actividade prevista para o servidor. Um valor de 5 permite já uma actividade moderada.

Retorna 0 se não ocorrer nenhum erro e -1 no caso contrário.

Como se disse, o serviço `listen()` apenas permite a recepção de pedidos de ligação que vão sendo atendidos e enfileirados pelo sistema operativo. Este serviço não aceita definitivamente nenhuma ligação, nem sequer bloqueia o processo, retornando quase imediatamente.

Para que uma aplicação aceite uma ligação já completamente estabelecida, e assim possa comunicar através dela, é necessária a utilização de outro serviço: o serviço `accept()`. Este serviço retira da fila criada pelo sistema operativo a primeira ligação completa que aí se encontrar, cria um novo socket capaz de comunicar directamente com o cliente através da ligação estabelecida, e ainda fornece informação do endereço e porto de origem da ligação.
Se quando `accept()` for chamado não houver nenhum pedido de ligação completo, a chamada bloqueia até que isso aconteça. As comunicações deverão ser efectuadas no novo socket retornado pelo serviço.

Esta arquitectura do serviço `accept()` de retornar um novo socket para efectuar as comunicações com os clientes que vão pedindo as ligações com o servidor, presta-se, como iremos ver num exempo, a que o servidor possa ser implementado como uma aplicação multi-processo ou multi-*thread*.

```
#include <sys/socket.h>

int accept(int sockfd, struct sockaddr *cliaddr, socklen_t *addrlen);
```

Espera por que haja uma ligação completa na fila do sistema operativo associada a um socket em escuta. O socket em escuta é especificado no parâmetro **sockfd**. Quando surge essa ligação é criado e retornado um novo socket pronto a comunicar. O endereço e porto de origem são retornados através de uma estrutura **sockaddr_in** (para endereços IP) cujo endereço deve ser passado em **cliaddr**. O parâmetro **addrlen** é um apontador para um local onde deve, à partida, ser colocado o tamanho em bytes da estrutura passada em **cliaddr**. O serviço pode modificar esse tamanho.

Retorna um valor positivo (descritor do socket ligado), ou -1 no caso de erro.

O socket retornado em `accept()` está pronto a comunicar com o cliente. Isso é feito usando os serviços de leitura e escrita em ficheiros, `read()` e `write()`, descritos na secção 2.2 (página 11). Basta usar o descritor do socket, como se tratasse de um ficheiro. A comunicação é bufferizada podendo haver bloqueios quando os buffers estão vazios (caso da leitura) ou cheios (caso da escrita).

Quando o cliente fechar a ligação (fechando o respectivo socket), uma chamada a `read()` retornará o valor 0 (depois de esgotar a informação que ainda se encontre no buffer de leitura), que é o indicador de fim de ficheiro. Nessa altura o servidor deverá também fechar o socket correspondente. O serviço de fecho de um socket, como já se viu a propósito dos sockets UDP, é o serviço `close()`.

Resta apenas descrever como o cliente efectua um pedido de ligação TCP. Depois da criação de um socket TCP o pedido de ligação ao servidor pode ser imediatamente efectuado usando o serviço `connect()`. Se o socket do cliente não estiver associado a um porto local (o que é normal nos clientes) o sistema operativo fará essa associação neste momento escolhendo um porto efémero. Essa informação é enviada ao servidor juntamente com o pedido de ligação. No pedido de ligação deve ser especificado o endereço IP e porto do servidor. O serviço `connect()` bloqueará o cliente até que o servidor aceite (com `accept()`) a ligação.

```
#include <sys/socket.h>

int connect(int sockfd, const struct sockaddr *servaddr,socklen_t addrlen);
```

Efectua um pedido de ligação TCP do socket especificado em **sockfd** ao servidor cujo endereço IP e porto são especificados através de uma estrutura **sockadd_in** passada através do apontador **servaddr**. Em **addrlen** deve ser colocado o tamanho em bytes da estrutura passada em **servaddr**. O serviço retorna quando a ligação for aceite pelo servidor.

Retorna 0 se a ligação for aceite e -1 no caso de erro.

Quando `connect()` retorna com sucesso, o socket está pronto a comunicar com o servidor usando os serviços `read()` e `write()`. Quando o cliente não precisar de efectuar mais comunicações com o servidor deverá fechar o seu socket com `close()`. Isso permitirá ao servidor detectar essa situação, e por sua vez, fechar o seu lado da comunicação.

8.6 Exemplos com sockets TCP

Servidor sequencial de *daytime* usando sockets TCP:

```
#include <stdio.h>
#include <string.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <netinet/in.h>
#include <time.h>

#define SERV_PORT 9877
#define MAXLINE 1024

void main(void)
{
  int lsockfd, csockfd;
  struct sockaddr_in servaddr, cliaddr;
  socklen_t len;
  time_t ticks;
  char buff[MAXLINE];

  lsockfd = socket(PF_INET, SOCK_STREAM, 0);

  memset(&servaddr, 0, sizeof(servaddr));
  servaddr.sin_family = AF_INET;
  servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
  servaddr.sin_port = htons(SERV_PORT);

  bind(lsockfd, (struct sockaddr *) &servaddr, sizeof(servaddr));
  listen(lsockfd, 5);

  for ( ; ; ) {
    len = sizeof(cliaddr);
    csockfd = accept(lsockfd, (struct sockaddr *) &cliaddr, &len);
    printf("connection from: %s - %d\n", inet_ntoa(cliaddr.sin_addr),
           ntohs(cliaddr.sin_port));
    ticks = time(NULL);
    sprintf(buff, "%.24s\r\n", ctime(&ticks));
    write(csockfd, buff, strlen(buff));
    close(csockfd);
  }
}
```

Como se pode ver, neste exemplo aceita-se uma ligação de cada vez dentro do ciclo infinito do servidor. Assim que uma ligação é estabelecida envia-se imediatamente o tempo local sem necessidade de ler qualquer informação. Após este envio fecha-se o socket de comunicação (`csockfd`), aceitando-se de seguida outra ligação.

É possível tratar todos os pedidos que chegam, em paralelo, usando um servidor multiprocesso, como se pode ver no próximo exemplo.

Servidor multiprocesso de *daytime* usando sockets TCP (incorrecto):

```
#include <stdio.h>
#include <string.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <netinet/in.h>
```

```
#include <time.h>
#include <unistd.h>
#include <sys/types.h>

#define SERV_PORT 9877
#define MAXLINE 1024

void main(void)
{
  int lsockfd, csockfd;
  struct sockaddr_in servaddr, cliaddr;
  socklen_t len;
  pid_t pid;
  time_t ticks;
  char buff[MAXLINE];

  lsockfd = socket(PF_INET, SOCK_STREAM, 0);

  memset(&servaddr, 0, sizeof(servaddr));
  servaddr.sin_family = AF_INET;
  servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
  servaddr.sin_port = htons(SERV_PORT);

  bind(lsockfd, (struct sockaddr *) &servaddr, sizeof(servaddr));
  listen(lsockfd, 5);

  for ( ; ; ) {
    len = sizeof(cliaddr);
    csockfd = accept(lsockfd, (struct sockaddr *) &cliaddr, &len);

    if ( (pid = fork()) == 0) {
      close(lsockfd);
      printf("connection from: %s - %d\n", inet_ntoa(cliaddr.sin_addr),
             ntohs(cliaddr.sin_port));
      ticks = time(NULL);
      sprintf(buff, "%.24s\r\n", ctime(&ticks));
      write(csockfd, buff, strlen(buff));
      close(csockfd);
      exit(0);
    }

    close(csockfd);
  }
}
```

Quando se cria um novo processo através do serviço `fork()` todos os descritores de sockets e de ficheiros são duplicados nas tabelas do sistema operativo. Para eliminar essa duplicação é necessário fechar o socket de escuta nos processos filhos, assim como o socket de comunicação, retornado por `accept()`, no processo pai. É o que se faz no código acima.

O código do exemplo anterior, parecendo correcto, tem uma falha grave. Ao fim de algum tempo de execução e de servir alguns clientes bloquerá provavelmente todo o sistema.

O processo pai que compõe o servidor executa indefinidamente num ciclo infinito, enquanto que se vão criando processos filhos que ao fim de algum tempo terminam. No entanto como o seu código de terminação não é aceite pelo pai esses processos vão-se tornando *zombies*, continuando a ocupar a tabela de processos do sistema operativo. Quando o número máximo de processos for atingido o sistema será incapaz de criar novos

processos. Assim, é necessário aceitar o código de terminação dos filhos à medida que estes vão terminando, para libertar o sistema destes *zombies*.

Isso pode ser feito aproveitando o facto de que quando um processo termina este envia ao pai o sinal `SIGCHLD`. Por defeito este sinal é ignorado pelo pai, mas é possível instalar um handler para o tratar.

No entanto, o uso de sinais pode trazer mais complicações. Quando um processo está bloqueado numa chamada a um serviço de sockets e lhe chega um sinal, o serviço é interrompido e o handler do sinal é executado. Mas, quando o handler termina, o processo vê o serviço que estava bloqueado retornar com o erro `EINTR`. Há então que tornar a chamar o serviço que estava bloqueado.

Existe ainda uma outra particularidade que é preciso ter em conta neste cenário. Normalmente quando um handler de um sinal está a executar e chega outro sinal, este é descartado (os sinais não são enfileirados).

Assim, tomando em consideração tudo o que foi discutido, podemos corrigir o código anterior para o seguinte.

Servidor multiprocesso de *daytime* usando sockets TCP e sinais:

```
#include <stdio.h>
#include <string.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <netinet/in.h>
#include <time.h>
#include <unistd.h>
#include <signal.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <errno.h>

#define SERV_PORT 9877
#define MAXLINE 1024

void sig_chld(int);

void main(void)
{
  int lsockfd, csockfd;
  struct sockaddr_in servaddr, cliaddr;
  socklen_t len;
  pid_t pid;
  time_t ticks;
  char buff[MAXLINE];

  lsockfd = socket(PF_INET, SOCK_STREAM, 0);

  memset(&servaddr, 0, sizeof(servaddr));
  servaddr.sin_family = AF_INET;
  servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
  servaddr.sin_port = htons(SERV_PORT);

  bind(lsockfd, (struct sockaddr *) &servaddr, sizeof(servaddr));

  listen(lsockfd, 5);
```

```
  signal(SIGCHLD, sig_chld);

  for ( ; ; ) {
    len = sizeof(cliaddr);
    if ((csockfd = accept(lsockfd, (struct sockaddr *) &cliaddr, &len))< 0)
      if (errno == EINTR)
        continue;

    if ( (pid = fork()) == 0) {
      close(lsockfd);
      printf("connection from: %s - %d\n", inet_ntoa(cliaddr.sin_addr),
             ntohs(cliaddr.sin_port));
      ticks = time(NULL);
      sprintf(buff, "%.24s\r\n", ctime(&ticks));
      write(csockfd, buff, strlen(buff));
      close(csockfd);
      exit(0);
    }

    close(csockfd);
  }
}

void sig_chld(int signo)
{
  pid_t pid;
  int stat;

  while ((pid = waitpid(-1, &stat, WNOHANG)) > 0)
    printf("child %d terminated.\n", pid);
}
```

Finalmente veremos a seguir o código do cliente que necessita do nome do servidor na linha de comando.

Exemplo de cliente *daytime* usando sockets TCP :

```
#include <stdio.h>
#include <string.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <netinet/in.h>
#include <netdb.h>

#define SERV_PORT 9877
#define MAXLINE 1024

int main(int argc, char *argv[])
{
  int sockfd, n;
  struct sockaddr_in servaddr;
  char buff[MAXLINE];
  struct hostent *hostp;

  if (argc != 2) {
    printf("Usage: daytime_c <hostname>\n");
    return 1;
  }

  hostp = gethostbyname(argv[1]);
```

```
  if (hostp == NULL) {
    printf("Host <%s> unknown!\n", argv[1]);
    return 1;
  }

  memset(&servaddr, 0, sizeof(servaddr));
  servaddr.sin_family = AF_INET;
  servaddr.sin_port = htons(SERV_PORT);
  servaddr.sin_addr = *(struct in_addr *)(hostp->h_addr_list[0]);

  printf("Asking time from: %s - %d\n", inet_ntoa(servaddr.sin_addr),
         ntohs(servaddr.sin_port));

  sockfd = socket(PF_INET, SOCK_STREAM, 0);

  connect(sockfd, (struct sockaddr *) &servaddr, sizeof(servaddr));

  while ((n = read(sockfd, buff, MAXLINE-1)) > 0) {
    buff[n] = 0;
    printf("%s", buff);
  }
  close(sockfd);
  return 0;
}
```

Depois de estabelecida a ligação (quando `connect()` retorna) lêem-se todos os bytes disponíveis até que o servidor feche a ligação (nessa altura `read()` retorna 0). Dependendo do volume de informação, condições da rede, e tamanho dos buffers pode ser necessário efectuar várias chamadas a `read()`.

É de notar que nenhum dos exemplos apresentados efectua verificação de erros. Numa aplicação real essa verificação terá de ser efectuada em todas as chamadas que retornem informação de erro.