Parte 4

Acesso a ficheiros

O ciclo de trabalho com um ficheiro inicia-se com a sua *abertura* (e *criação*, se ainda não existe), que é seguida por uma sequência de operações de *leitura* e/ou de *escrita* (intercaladas com pedidos de reposicionamento, quando aplicável), e termina com o *fecho* do ficheiro.

A cada ficheiro aberto está associada uma *cabeça* (virtual) de leitura/escrita (L/E), colocada em alguma posição do ficheiro enquanto ele está aberto. Na abertura de um ficheiro, a sua cabeça de L/E é colocada na posição 0, correspondente ao primeiro *byte* do ficheiro.

Cada operação de leitura (ou escrita) faz avançar a posição da cabeça de L/E tantas posições quantos os *bytes* lidos (ou escritos). Se as operações sobre um ficheiro consistirem numa sequência de leituras ou numa sequência de escritas, temos um ficheiro com *acesso sequencial* (é o que acontece, em geral, com os ficheiros que só contêm texto, que são lidos ou escritos sequencialmente). É, no entanto, possível reposicionar a cabeça de L/E de um ficheiro e ter *acesso directo* a qualquer posição do ficheiro.

As funções seguintes, não sendo as únicas disponibilizadas pelo C, permitem executar as operações referidas acima.

`int open(char *filename, int flags, mode_t mode)`

A função `open` tenta abrir o ficheiro `filename` e devolve um inteiro não negativo, que será posteriormente usado no programa quando se quiser fazer alguma operação sobre este ficheiro. Este inteiro é normalmente designado como o *descritor do ficheiro* (*file descriptor*). Se ocorrer algum erro durante a sua execução, a função devolve o valor -1.
O argumento `flags` é uma combinação de *flags* que condicionam a operação da função. Algumas *flags* são

- `O_RDONLY` – a abertura é feita em modo de leitura
- `O_WRONLY` – a abertura é feita em modo de escrita
- `O_RDWR` – a abertura é feita em modo de leitura e escrita
- `O_CREAT` – o ficheiro é criado se ainda não existe
- `O_TRUNC` – se o ficheiro existe e é aberto `O_WRONLY` ou `O_RDWR`, o seu conteúdo é apagado

Uma das três primeiras *flags* tem de estar presente na chamada da função.
(A combinação de *flags* é feita através do operador `|`, que a calcula o *OU-bit-a-bit* dos seu operandos. Por exemplo, o valor de 0101_2 `|` 0011_2 é 0111_2.)

Para a utilização de um ficheiro com uma estrutura de dados dinâmica, uma combinação útil de *flags* é `O_RDWR | O_CREAT`. Com este valor para as *flags*, a função abre o ficheiro para leitura e escrita, se ele já existe, criando-o antes se ele ainda não existir.

Só é necessário incluir o terceiro argumento se as *flags* contiverem `O_CREAT`. Se o ficheiro tiver de ser criado, `mode` indica as permissões de acesso que lhe ficarão associadas. Quando o ficheiro já existe, as suas permissões mantêm-se.

O valor de `mode` resulta, tal como o de `flags`, da combinação de um ou mais valores através do operador `|`. Para que o utilizador que executa o programa possa ler e escrever o ficheiro criado, `mode` poderá ser a combinação `S_IRUSR | S_IWUSR`.

Para usar esta função, deverá incluir os ficheiros `sys/types.h`, `sys/stat.h` e `fcntl.h`.

(Encontra muito mais informação sobre `open` executando o comando `man 2 open`.)

`int close(int fd)`

Fecha o ficheiro associado a `fd`, que deverá ser o valor devolvido por `open` quando o ficheiro foi aberto. Todos os ficheiros abertos deverão ser fechados quando deixar de ser necessário aceder-lhes.
Para usar esta função, deverá incluir o ficheiro `unistd.h`.

`ssize_t read(int fd, void *address, size_t count)`

Esta função tenta ler `count` *bytes* do ficheiro associado a `fd`.
Os *bytes* efectivamente lidos serão colocados na zona de memória cujo endereço é indicado por `address`. Devolve o número de *bytes* transferidos ou -1, se ocorrer algum erro.

A função `read` devolve um inteiro não negativo diferente de `count` quando se tenta ler para além do fim do ficheiro. Se uma chamada a `open` que origina a criação do ficheiro for seguida de uma chamada a `read` (com `count` diferente de 0), esta última devolverá 0, porque o ficheiro está vazio. Esta sequência de operações pode ser usada para detectar que o ficheiro aberto é um ficheiro que não existia, como exemplifica o fragmento de código seguinte:

```
int fd;

fd = open(filename, O_CREAT | O_RDWR, S_IRUSR | S_IWUSR);

if (fd == -1)
```

```
  {
    perror("open");

    return <VALOR-QUE-INDICA-ERRO>;
  }

...

switch (read(fd, ...))
  {
    case -1:                    // ocorreu um erro na
leitura
      perror("read");

      ...       // limpezas

      close(fd);

      return <VALOR-QUE-INDICA-ERRO>;
    case 0:                     // o ficheiro está vazio
      ...       // inicializações

      /* fall through */
    default:                    // o ficheiro não está
vazio
      return <VALOR>;
  }
```

Para usar esta função, deverá incluir o ficheiro `unistd.h`.

(Os tipos `size_t` e `ssize_t` correspondem, respectivamente, a inteiros sem e com sinal.)

`ssize_t write(int fd, void *address, size_t count)`

Escreve `count` *bytes*, localizados na memória a partir do endereço `address`, no ficheiro associado a `fd`. Devolve o número de *bytes* escritos ou -1, se ocorrer algum erro.
Para usar esta função, deverá incluir o ficheiro `unistd.h`.

`off_t lseek(int fd, off_t offset, int whence)`

Cada operação de leitura (ou escrita) lê (ou escreve) `count` *bytes* a partir da posição corrente da cabeça, e fá-la avançar tantas posições quantos os *bytes* lidos (ou escritos). Uma operação de leitura (ou escrita) subsequente operará a partir da nova posição corrente.
A função `lseek` permite reposicionar a cabeça de L/E de um ficheiro e torna possível aceder directamente a qualquer *byte* do ficheiro.

Enquanto que o primeiro argumento da função é o já conhecido descritor do ficheiro, o terceiro argumento indica-lhe como deve interpretar o segundo:

Valor de `whence`	Interpretação de `offset`	Nova posição
`SEEK_SET`	posição absoluta a partir do início do ficheiro	`offset`
`SEEK_CUR`	distância à posição corrente (pode ser negativa)	posição corrente + `offset`
`SEEK_END`	distância ao fim do ficheiro (pode ser positiva)	tamanho do ficheiro + `offset`

O valor devolvido é a nova posição da cabeça de L/E, ou -1 se ocorrer algum erro (por exemplo, se a posição resultante for negativa).
A tabela seguinte apresenta as posições da cabeça de L/E resultantes de uma sequência de chamadas a `lseek` com os argumentos mostrados, para um ficheiro com 100 *bytes*. (Repare que, sendo 0 a posição do primeiro *byte* do ficheiro, a posição 100 é a do *byte* a seguir ao último *byte* do ficheiro.)

`offset`	`whence`	Nova posição	
0	`SEEK_SET`	0	
0	`SEEK_CUR`	0	
10	`SEEK_CUR`	10	
30	`SEEK_CUR`	40	
-15	`SEEK_CUR`	25	
0	`SEEK_END`	100	a)
-20	`SEEK_END`	80	
20	`SEEK_END`	120	b)
50	`SEEK_SET`	50	
-60	`SEEK_CUR`	50	c)

Notas:

a. Quando `lseek` é chamada com `offset` 0 e `whence` igual a `SEEK_END`, a cabeça de L/E fica colocada imediatamente *a seguir* ao último *byte* do ficheiro.
b. A cabeça de L/E fica colocada 20 *bytes* para lá do fim do ficheiro. Se for executada uma operação de escrita com a cabeça nesta posição, o sistema preenche as 20 posições intermédias (da 100 à 119) com o *byte* 0. (O que acontecerá se for executada uma operação de escrita quando a cabeça de L/E está na posição imediatamente a seguir ao último *byte* do ficheiro?)
c. Esta chamada, que tenta colocar a cabeça de L/E *antes* da primeira posição do ficheiro, devolve -1 e não altera a posição corrente da cabeça.

Para usar esta função, deverá incluir os ficheiros `sys/types.h` e `unistd.h`.

`void perror(char *prefix)`

Todas as funções referidas acima sinalizam a ocorrência de um erro devolvendo -1, o que não permite distinguir entre os vários erros que podem ocorrer. Por exemplo, quando se tenta abrir um ficheiro para leitura e não se consegue, isso pode dever-se a não ter permissão para o ler (erro `EACCES`) ou a ele não existir (erro `ENOENT`).
As funções com este tipo de comportamento guardam, normalmente, o código do erro que ocorreu numa variável global chamada `errno`, definida numa biblioteca do sistema. A função `perror` consulta esta variável e escreve na *consola de erro* (`stderr`) do programa a mensagem correspondente.

Se `unreadable` for um ficheiro que não podemos ler e se não existir um ficheiro chamado `nosuchfile`, o código seguinte

```
fd = open("unreadable", O_RDONLY);
if (fd == -1)
  perror("open: unreadable");

fd = open("nosuchfile", O_RDONLY);
if (fd == -1)
  perror("open: nosuchfile");
```

produzirá na consola

```
open: unreadable: Permission denied
open: nosuchfile: No such file or directory
```

ou o equivalente noutra língua.

O protótipo desta função encontra-se no ficheiro `stdio.h`.

Faça algumas experiências usando estas funções, criando e abrindo ficheiros, e lendo e escrevendo texto e números de e para ficheiros.

Funções ISO C

As funções para acesso a ficheiros e as constantes descritas acima são definidas no *standard* POSIX para (a biblioteca d)o C.

O *standard* ISO que define a linguagem C (eg, C89 ou C99) também inclui funções para o mesmo efeito, que são apresentadas nesta secção. (Uma implementação que respeite um *standard* ISO do C não tem de incluir as funções anteriores, enquanto que uma implementação POSIX do C terá de incluir aquelas e as do ISO C.) No Linux, as funções acima correspondem a chamadas ao sistema e são usadas pelas funções ISO C.

Tal como as funções acima, em caso de erro, estas funções guardam o código correspondente em `errno`.

Para usar estas funções, deverá ser incluído o ficheiro `stdio.h`.

`FILE *fopen(char *path, char *mode)`

A função `fopen` abre o ficheiro indicado por `path` em modo `mode`, uma *string* que pode começar por

`r`

A abertura é feita em modo de leitura. É um erro o ficheiro não existir.

`w`

A abertura é feita em modo de escrita. Se o ficheiro existir, o seu conteúdo é apagado, senão é criado um ficheiro com o nome dado.

`a`

A abertura é feita em modo de escrita, só permitindo acrescentar dados ao ficheiro. Se o ficheiro não existe, é criado.

Se `mode` for `"r+"`, `"w+"` ou `"a+"`, aplicam-se as observações anteriores, mas o ficheiro é aberto para leitura e para escrita.

(`mode` pode também conter o carácter `b`, que indica tratar-se de um ficheiro binário, ie, não de texto. Esta indicação não tem qualquer efeito nos sistemas POSIX, como o Linux.)

O valor devolvido deverá ser usado em todas as funções abaixo (corresponde ao argumento `stream`), quando se pretender realizar alguma operação sobre o ficheiro. Em caso de erro na abertura/criação, o valor devolvido será `NULL`.

`int fclose(FILE *stream)`

Fecha o ficheiro associado a `stream`, depois de ter garantido que todas as alterações ao ficheiro foram realmente feitas.

`size_t fread(void *ptr, size_t size, size_t nmemb, FILE *stream)`

Basicamente equivalente a

`read(`*`fd`*`, ptr, size * nmemb).`

`size_t fwrite(void *ptr, size_t size, size_t nmemb, FILE *stream)`

Basicamente equivalente a

`write(`*`fd`*`, ptr, size * nmemb).`

`int fseek(FILE *stream, long offset, int whence)`

Equivalente a

`lseek(`*`fd`*`, offset, whence)`

mas devolve 0 (sucesso) ou -1 (erro).

`long ftell(FILE *stream)`

Serve para obter a posição corrente da cabeça de L/E e é equivalente a

`lseek(`*`fd`*`, 0, SEEK_CUR).`

`void rewind(FILE *stream)`

Equivalente a

`fseek(stream, 0, SEEK_SET).`

`int fflush(FILE *stream)`

É usada para forçar as alterações feitas ao ficheiro a serem efectivamente escritas em memória secundária (disco).

`int feof(FILE *stream)`

Indica se foi atingido o fim do ficheiro.

`int ferror(FILE *stream)`

Indica se ocorreu um erro na utilização do ficheiro

Em C, existem três nomes pré-definidos com tipo `FILE *`:

`stdin`

A *entrada normal* (*standard input*) de dados do programa, normalmente associada ao que é introduzido através do teclado.

`stdout`

A *saída normal* (*standard output*) do programa, destinada ao *output* normal do programa, normalmente associada ao terminal onde o programa é executado.

`stderr`

A *saída de erro* (*standard error*) do programa, destinada às mensagens de erro, normalmente associada ao terminal onde o programa é executado. (É usada, por exemplo, pela função `perror`.)

As funções `fread`, `fwrite` e `fseek` destinam-se, sobretudo, a ser usadas em ficheiro (binários) de acesso directo. O C possui, também, uma família de funções apropriadas para o uso em ficheiros de texto em que, tipicamente, o acesso é sequencial.

As primeiras são as generalizações das já conhecidas `printf` e `scanf`:

```
int fprintf(FILE *stream, char *format, ...)
int fscanf(FILE *stream, char *format, ...)
```

(De facto, `printf(...)` é o mesmo que `fprintf(stdout, ...)`, e `scanf(...)` é o mesmo que `fscanf(stdin, ...)`.)

Depois, há as funções `fgetc`/`getc` (leitura de um carácter), `getchar` (equivalente a `getc(stdin)`), `fgets` (leitura de uma linha), `fputc`/`putc` (escrita de um carácter), `putchar` (equivalente a `putc(c, stdout)`), `fputs`, (escrita de uma *string*) e `puts` (escrita de uma linha, equivalente a `fputs(string + "\n", stdout)`).