Ficha 6 – Ficheiros – API baixo nível

Tópicos abordados:

– Ficheiros – API baixo nível

– Exercícios

Duração prevista: 1 aula

1. Funções para manipulação de ficheiros

Os ficheiros permitem que um programa guarde os seus dados em memória persistente, possibilitando que estes se encontrem disponíveis mesmo após o reiniciar do computador. A forma como os dados são estruturados no dispositivo de armazenamento varia consoante o sistema de ficheiros associado ao mesmo. Exemplos de sistemas de ficheiros incluem o FAT, NTFS, EXT2/3/4, ZFS, etc.

Exercício 1

Execute o utilitário `df -T` num terminal. Quais os tipos de sistemas de ficheiros empregues na máquina em que é executado o utilitário?

Do ponto de vista do programador, um ficheiro não é mais do que uma estrutura de dados organizada de forma sequencial sem possuir um tamanho pré-definido. A linguagem C define uma API de baixo nível para manipulação do sistema de ficheiros. Esta API é bastante minimalista e consiste nas seguintes funções: ***open***, ***read***, ***write***, ***lseek***, ***close*** e ***stat***.

1.1. Descritores

Na API de baixo nível, um ficheiro é representado por um inteiro (descritor) e as operações de escrita e leitura são binárias. Na realidade, em UNIX, todos os periféricos do sistema (teclado, ecrã, impressora, ...) são representados através de descritores (um inteiro sequencial que representa um recurso do sistema operativo afeto a um processo).

Quando um programa é executado, o sistema operativo cria um processo para a sua execução. Na tabela de descritores desse processo são criados automaticamente 3 descritores:

- ***stdin*** – descritor da entrada de dados padrão (por omissão é o teclado). Representado pelo inteiro 0;
- ***stdout*** – saída de dados padrão (por omissão é o terminal). Representado pelo inteiro 1;
- ***stderr*** – saída de erros padrão (por omissão é o terminal). Representado pelo inteiro 2;

Por exemplo, a instrução:

```
printf("Mensagem para o stdout");
```

Consiste na seguinte invocação de baixo nível da API do C:

```
write(1, "Mensagem para o stdout", 22);
```

E a instrução:

```
scanf("%d", &a);
```

Inicia a seguinte invocação de baixo nível:

```
read(0, STRING, 1024);
```

Como se depreende a partir destas chamadas, a API de baixo nível apenas permite a leitura e escrita de dados binários como simples sequências de *bytes*. Para efetuar leituras ou escritas mais eficientes ou de informação textual é necessário recorrer à API de alto nível designada por standard I/O.

As funções da API de baixo nível de acesso a ficheiros correspondem quase todas a chamadas ao sistema. Deste modo, em sistemas UNIX, a respetiva documentação eletrónica encontra-se na secção 2 do sistema *man*. Assim, caso se pretende aceder à documentação *man* da função open da API de ficheiros, deve-se explicitar a secção 2, da seguinte forma:

`man 2 open`

2. API de baixo nível

A API de baixo nível para acesso e manipulação de ficheiros é composta pelas funções listadas na TABELA

open	(tenta) abrir um ficheiro, devolvendo um descritor para o ficheiro aberto
close	Encerra o acesso ao ficheiro através do descritor indicado
read	(tenta) efetuar a leitura de um n octetos do ficheiro
write	(tenta) proceder à escrita de n octetos para o ficheiro
lseek	(tenta) posicionar a posição corrente do ficheiro

Tabela 1: funções da API de baixo nível

2.1. Abertura e criação de ficheiro – open e creat

```
int open(const char *pathname, int flags);
int open(const char *pathname, int flags, mode_t mode);
int creat(const char *pathname, mode_t mode);
```

Em caso de sucesso, ambas as funções devolvem um descritor para o ficheiro indicado por *pathname* (caminho absoluto ou relativo). O tipo de acesso (leitura, escrita, acréscimo, etc.) depende do que for passado no parâmetro **flags**. Qualquer uso da função **open** requer que seja passada pelo menos uma das seguintes *flags*: O_RDONLY (abertura somente para leitura), O_WRONLY (modo de abertura somente para escrita) ou O_RDWR (modo de abertura para leitura e escrita). Os identificadores O_RDONLY, O_WRONLY e O_RDWR correspondem a constantes do pré-processador. Em simultâneo com estas, poderemos utilizar outras como o O_CREAT (permite criar o ficheiro caso este não exista), O_APPEND (modo de abertura para acrescentar ao ficheiro), entre outras (consultar **man 2 open**).

O valor de retorno da função **open** é um número inteiro. Caso a operação tenha sido bem-sucedida é devolvido o denominado *descritor de ficheiro*. O descritor de ficheiro é um inteiro positivo que corresponde ao índice do ficheiro na tabela de ficheiros abertos do processo. Caso a tentativa de abertura do ficheiro tenha falhado, por exemplo, o ficheiro não existe, ou o processo não tem permissões para abrir o ficheiro, é devolvido o valor -1, sendo um apropriado código de erro atribuído à variável inteira **errno**.

2.1.1. Criação de um ficheiro

A versão da função **open** com três parâmetros é empregue quando se pretende criar o ficheiro, sendo nesse caso necessário especificar o parâmetro **mode** com as permissões que se pretendem atribuir ao ficheiro a criar. Em alternativa, pode usar-se a função **creat** que, internamente, corresponde à chamada à função **open** indicando **O_CREAT|O_WRONLY|O_TRUNC** para o parâmetro **flags**. Importa notar que o símbolo | (barra vertical) corresponde à operação de OR binário na linguagem C.

As permissões a serem indicadas através do parâmetro **mode** podem ser especificadas através de constantes do pré-processador existentes para o efeito. O uso de várias constantes para o parâmetro modo requer o uso do operador OR binário. Por exemplo, para indicar que se pretende um ficheiro com permissões RWX para o dono, RW para o grupo e sem permissões para os outros, o parâmetro **mode** deve ser especificado da seguinte forma:

```
mode = S_IRWXU | S_IRGRP | S_IWGRP
```

A lista completa das opções para o parâmetro **mode** encontra-se na página de documentação eletrónica das funções **open/creat**. Em alternativa, o parâmetro **mode** pode ser especificado através de um número em base octal. Por exemplo, as permissões RWX-RW---- são indicadas através do octal 0760. Note-se que o número deve iniciar-se com um zero à esquerda de modo a que seja interpretado como base octal.

Para mais informações, consulte **man 2 open**

2.2. Fecho de ficheiro - close

```
int close(int fd);
```

A função **close** encerra o descritor de ficheiro indicado por ***fd*** (*file descriptor*).

Para mais informações, consulte **man 2 close**

Exercício 2

Elabore o programa **open_file** que deve proceder à abertura para **leitura** do ficheiro cujo nome é especificado através do único parâmetro da linha de comando. Caso o número de parâmetros da linha de comando não corresponda ao solicitado, o programa deve enviar uma apropriada mensagem de erro para o canal de erro padrão e encerrar com o código de término 1.

Caso não seja possível a abertura do ficheiro, a aplicação deve terminar com o código de término 2, indicando o código de erro numérico, bem como a *string* associada ao código de erro.

Sugestão: função **strerror**

2.3. Operações de escrita - write

```
ssize_t write(int fd, const void *buf, size_t count);
```

A função **`write`** tenta escrever *count* octetos que estão na posição de memória apontada por ***`buf`*** para o ficheiro indicado pelo descrito ***`fd`***. Em caso de erro é devolvido -1, sendo atribuído pelo sistema o código de erro apropriado à situação à variável **`errno`**. Caso contrário, é devolvido o número de octetos que foram efetivamente escritos no ficheiro.

Para mais informações, consulte **`man 2 write`**

Exercício 3

Elabore o programa **write_file** que deve proceder à abertura para escrita do ficheiro cujo nome é especificado através do único parâmetro da linha de comando. O programa deve sucessivamente:

i) Escrever os números de 1 a 10, em formato binário, usando representação inteira (tipo de dados *int*)

ii) Escrever os números de 1 a 10, em formato de *string* (um número por linha)

Após a criação do ficheiro, abra o ficheiro com um editor binário em modo hexadecimal (e.g., **`od -x nome_ficheiro`**). Que diferenças existem entre a parte que contém representação inteira e a parte que contém representação em *string*?

Exercício 4

Elabore o programa **append_file** que deve proceder à abertura para acréscimo (*append*) do ficheiro cujo nome é especificado através do único parâmetro da linha de comando. O programa deve acrescentar uma *string* com a data corrente no formato **YYYY.MM.DD** ao ficheiro.

Sugestão: função **strftime**

Dispositivo /dev/full

O Linux disponibiliza o dispositivo virtual `/dev/full` que, como o nome sugere, simula um dispositivo que está cheio. O `/dev/full` possibilite testar código que tenta escrever para um dispositivo que está cheio, na medida em que qualquer tentativa de escrita falha. Por sua vez, as operações de leitura (*read*) do `/dev/full` devolvem sempre o conteúdo '\0'. A documentação para o dispositivo `/dev/full` está disponível através da entrada *full* na secção 4 do manual (**`man 4 full`**)

Considere o seguinte caso de uso do dispositivo `/dev/full`, que tenta redirecionar a saída do comando `ps aux` para o dispositivo `/dev/full`.

```
ps aux > /dev/full
ps: write error: No space left on device
```

Figura 1 – Tentativa de redireccionamento da saída padrão do comando `ps aux` para o dispositivo `/dev/full`

Exercício 5

Elabore o programa **full** que deve tentar escrever os números inteiros 0, 1, 2 e 3 em formato *int* para o dispositivo `/dev/full`. O código deve tratar convenientemente as situações de erro, mostrando a mensagem de erro associado ao código de erro da variável `errno`.

2.4. Operações de leitura - read

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

A função **`read`** permite efetuar a leitura de até *count* octetos do ficheiro associado a *fd* para a zona de memória apontada por *buf*. A função devolve o número de octetos que

foram efetivamente lidos do ficheiro. Este valor pode ser inferior a *count*, quando, por exemplo, *count* é superior ao número de octetos que faltam ler do ficheiro. Caso estejamos no fim do ficheiro é devolvido o valor zero. Finalmente, em caso de erro, a função devolve -1, sendo atribuído à variável **`errno`** o código de erro do sistema.

Para mais informações, consulte **`man 2 read`**

2.5. lseek

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

A função **`lseek`** permite estabelecer a posição corrente do ficheiro associado ao descritor *`fd`*. O parâmetro *`offset`* corresponde a um deslocamento em octetos. A origem do deslocamento é indicada pelo parâmetro *`whence`*. Os valores possíveis para o parâmetro *`whence`* são:

- `SEEK_SET`: posiciona o descritor no byte *`offset`*
- `SEEK_CUR`: desloca o descritor *`offset`* octetos em relação à posição corrente. Caso *`offset`* seja um valor negativo, o deslocamento faz-se da posição corrente para o início do ficheiro.
- `SEEK_END`: altera a posição do descritor de ficheiro *`offset`* octetos em relação ao fim do ficheiro.

Caso a operação decorre com sucesso, a função `lseek` devolve a posição do descritor de ficheiro. Na ocorrência de um erro, é devolvido o valor -1, sendo atribuído à variável *`errno`* o código numérico da situação de erro.

Para mais informações, consulte **`man 2 lseek`**

2.6. fsync

```
int fsync(int fd);
```

A função **`fsync`** indica ao sistema operativo que deve efetivar as operações que possam estar pendentes sobre o descritor `fd`, nomeadamente através da escrita de dados e meta dados que ainda estejam pendentes nas memórias temporárias (*buffer*) do sistema operativo.
Para mais informações, consulte **`man 2 fsync`**

3. Exercícios

1)

a) Implemente o utilitário `bhead` (*binary head*). Esse utilitário tem a seguinte sintaxe:

```
bhead   num_bytes   filename
```

O parâmetro `num_bytes` representa um número de octetos (valor positivo) e o parâmetro `filename` corresponde a um nome de ficheiro. O utilitário deve mostrar, em formato hexadecimal, os `num_bytes` primeiros octetos do ficheiro `filename`.
Deve ser desenvolvido um projeto com o respetivo `makefile` recorrendo ao *template* disponível no moodle da UC.
Exemplo de utilização do utilitário `bhead`:

```
./bhead 10 file.o
[#001] 7f
[#002] 45
[#003] 4c
[#004] 46
[#005] 01
[#006] 01
[#007] 01
[#008] 00
[#009] 00
[#010] 00
```

NOTA: o comportamento do utilitário `bhead` é similar à execução de:

```
od -t x1 -N num_bytes  filename
```

b) Com base no projeto `bhead`, crie a versão `bhead2`. Embora esta versão apresenta uma funcionalidade similar à versão anterior, os parâmetros da linha de comando devem serem especificados da seguinte forma:

--num *num_bytes* ou **-n** *num_bytes* ➔ parâmetro opcional (valor por omissão 10)

--filename *ficheiro* ou **-f** *ficheiro* ➔ parâmetro obrigatório

NOTA: O tratamento dos parâmetros da linha de comando deve ser implementado através do utilitário `gengetopt`.

2) O formato do ficheiro do utilitário gzip (extensão **.gz**) está descrito no RFC 1952 (https://www.ietf.org/rfc/rfc1952.txt). Pretende-se que implemente o utilitário **peekgzip** que deve implementar as seguintes funcionalidades:

i) Identificação que o ficheiro indicado é efetivamente um ficheiro no formato **gzip**. Para tal deve validar os dois primeiros octetos do ficheiro que devem ser, respetivamente, 31 (0x1f em hexadecimal) e 139 (0x8b em hexadecimal).

ii) Extração do campo de quatro octetos MTIME (modified time). Este campo está no formato UNIX time. O campo MTIME inicia-se no 5º octeto do ficheiro .gz

iii) Extração do tipo de sistema de ficheiro no qual foi criado o ficheiro **.gz**. O identificador de sistema operativo é guardado no 10º octeto do ficheiro, de acordo com a seguinte tabela.

```
  0 - FAT filesystem (MS-DOS, OS/2, NT/Win32)
  1 - Amiga
  2 - VMS (or OpenVMS)
  3 - Unix
  4 - VM/CMS
  5 - Atari TOS
  6 - HPFS filesystem (OS/2, NT)
  7 - Macintosh
  8 - Z-System
  9 - CP/M
 10 - TOPS-20
 11 - NTFS filesystem (NT)
 12 - QDOS
 13 - Acorn RISCOS
255 - unknown
```

O mapa dos primeiros 10 octetos de um ficheiro GZIP é o seguinte:

```
+---+---+---+---+---+---+---+---+---+---+
|ID1|ID2|CM |FLG|     MTIME     |XFL|OS | (more-->)
+---+---+---+---+---+---+---+---+---+---+
```

ID1 representa o 1º octeto identificador, ID2 o 2º octeto identificador. MTIME corresponde ao *modified time* em formato UNIX time. Finalmente, o 10º octeto identifica o tipo de sistema de ficheiros no qual foi criado o ficheiro .gzip.

Apresenta-se de seguida, um exemplo de saída da aplicação **peekgzip**:

```
peekgzip    test.gz

ID1=0x1f
ID2=0x8b
MTIME=1461435958
filesystem=3 (unix)
```

NOTA: A aplicação **peekgzip** deve ser desenvolvida em projeto recorrendo ao *template* disponibilizado no moodle da UC.