7

I/O de baixo-nível em ficheiros

Sumário:

Referência bibliográfica:W. Stevens. Advanced Programming in the UNIX Environment. Addison-Wesley, 1992 (cap.3)

Introdução

Este capítulo. Objetivos:

- Descrição das funções **I/O** de baixo nível (system calls)
- Descrição das funções **sem entreposição** (*unbuffered* I/O) por contraste com as funções I/O com entreposição (*buffered* I/O) já estudadas em semestres anteriores. Veja-se capítulo anterior para uma breve revisão.
- Cada `read` ou `write` invoca uma chamada ao sistema no *kernel*.
- Mostrar como os ficheiros são partilhados entre vários processos e as estruturas de dados envolvidas no *kernel*.
- Usar MANUAL man para obter informação.
  Secção 1 (bash Shell commands)
  Secção 2 (System Calls)
  Secção 3 (C standard library)

Descritores de ficheiros

A entrada/saída de baixo-nível é feita sem entreposição (i.e. *unbuffered*). Isto significa que qualquer operação I/O a um ficheiro é feita diretamente. Equivalentemente, é escrito ou lido um bloco de bytes. Portanto, o ficheiro é considerado como um ficheiro binário (sem formatação), e não de texto (onde os bytes são interpretados em algum sistema de codificação, p.ex ASCII). Para o "kernel", todos os ficheiros abertos são identificados por descritores (inteiros não-negativo). Por exemplo, quando se abre um ficheiro ou se cria um novo, o kernel devolve um descritor desse ficheiro ao processo em causa. Cada processo tem, portanto, uma tabela (vetor) de descritores de ficheiros

Por convenção, o descritor 0 identifica a entrada estandardizada (standard input), o descritor 1 identifica a saída estandardizada (standard output) e o descritor 2 identifica o erro estandardizado (standard error). Estes descritores 0,1 e 2 podem ser substituídos pelas constantes simbólicas `STDIN_FILENO`, `STDOUT_FILENO` e `STDERR_FILENO`, respetivamente, em aplicações POSIX.

Funções I/O de baixo nível

Função open A função para abrir um ficheiro é a seguinte:

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

int open(char *filename, int access);
```

Retorna: descritor se OK, -1 em caso de erro

O argumento `access` descreve o tipo de acesso (ver <fcntl.h>); por exemplo, o valor O_RDONLY (open read only = abertura em modo apenas de leitura), O_APPEND, O_CREAT, O_EXCL, O_RDWR, O_WRONLY, etc. Estes acessos podem ser combinados através de operadores lógicos. Para mais informação, veja-se o manual *on-line* (man 2 open).

Função creat A função para criar um ficheiro é a seguinte:

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

int creat(const char *filename, int perms);
```

Retorna: descritor aberto para escrita se OK, -1 em caso de erro

O argumento `perms` contém as bandeiras (flags) de permissão. Para mais informação: (man 2 creat)

Permission flags

As constantes de permissão estão no ficheiro de cabeçalho <sys/stat.h>. Podem consultar o ficheiro stat.h no (por defeito) directório /usr/include/sys. As flags de permissão podem ser combinadas com o **operador binário** OR (|). Alguns dos constantes são dados em baixo.

```
S_IRUSR User read permission
S_IWUSR User write permission
S_IXUSR User execute permission
S_IRWXU User read, write and execute
```

```
S_IRGRP Group read permission
S_IWGRP Group write permission
S_IXUSR Group execute permission
S_IRWXG Group read, write and execute
```

```
S_IROTH Other read permission
S_IWOTH Other write permission
S_IXUSR Other execute permission
S_IRWXO Other read, write and execute
```

Os valores exatos podem ser vistos no ficheiro de inclusão: `cat /usr/inlcuude/linux/stat.h | grep S_I`

```
#define S_IRWXU 00700
#define S_IRUSR 00400
#define S_IWUSR 00200
#define S_IXUSR 00100
```

Uma utilização típica é de definir um constante, p.ex *#define COMMON_FILE_MODE (S_IRUSR | S_IWUSR)*, que represente a criação dum ficheiro com permissões de leitura e escrita para o próprio que será usado depois numa chamada a função creat. Por exemplo, creat(…. , COMMON_FILE_MODE);

Função close A função para fechar um ficheiro é a seguinte:

```
#include <unistd.h>

int close(int filedes);
```

Retorna: 0 se OK, -1 em caso de erro

Quando um processo termina, todos os ficheiros abertos são automaticamente fechados pelo *kernel*, não sendo necessário fechá-los expressamente. Para mais informação, veja-se o manual *on-line* (man close).

Função read Esta função permite ler a partir dum ficheiro aberto:

```
#include <unistd.h>

ssize_t read(int filedes, void *buff, size_t nbytes);
```

Retorna: número de bytes lidos, 0 se EOF, -1 em caso de erro

Se a leitura é feita com sucesso, a função devolve o número de bytes lidos. Se o fim do ficheiro é encontrado, a função devolve o valor 0 e em caso de erro o valor de -1. Há casos em que o número de bytes lidos poderá ser inferior à quantidade pedida em `nbytes`:

- Se o EOF é atingido antes de atingir o número de bytes pedidos. Por exemplo, se houver só mais 30 bytes para ler quando tinham sido solicitados 100, a função `read` só devolve aqueles 30 bytes.
- Leitura a partir dum terminal, só se lê normalmente uma linha de cada vez.
- Leitura a partir da rede, o tamanho do buffer de rede pode ser menor que a quantidade de bytes pretendida.
- Nalguns dispositivos baseados em registos (record-oriented devices), tais como os de fita magnética, só retornam um registo de x bytes de cada vez.

Função write Esta função permite escrever para um ficheiro aberto:

```
#include <unistd.h>
ssize_t write(int filedes, const void *buff, size_t nbytes);
```

Retorna: número de bytes escritos se OK, -1 em caso de erro

O valor devolvido pela função é usualmente igual ao valor do argumento `nbytes`; caso contrário, é porque ocorreu um erro, por exemplo devido a ter esgotado a capacidade física onde reside o ficheiro, o disco rígido por exemplo, ou o tamanho máximo admissível para um ficheiro dum dado processo. Para mais informação, veja-se o manual *on-line* (man 2 read ou man 2 write).

Exemplos e Exercícios

Exemplo 7.1:

O programa em baixo ilustra a criação dum ficheiro e a escrita neste ficheiro dum vector de dez inteiros.

Ler, Escrever, Compilar e Executar o programa.

```
/* ex71.c : compilar cc -Wall -c ex71.c -o ex71 */
#include <sys/stat.h>
#include <unistd.h>
#include <fcntl.h>
int main()
{
  int  fd, i, vec[10];

  for (i=0;i <10; i++)
          vec[i]=i+512;

  fd = creat("test.bin", S_IRUSR| S_IWUSR ) ;

  write(fd, vec , sizeof(int)*10);

  return (0);
}
```

Ver os detalhes do ficheiro, em particular o tamanho e tipo, criado com os comandos:

- **ls -l test.bin**
- **file test.bin**
- **cat test.bin** (para tentar visualizar o ficheiro)

Experimentar com o comando "Octal Dump", "**od test.bin".** Experimentar usar

- a opção **–i** do "octal dump" para visualizar grupos de 4 bytes como inteiros
- "od -s" para visual como "short ints" e
- "od -b" para para cada byte (nota por exemplo que 513 = 1 + 2*256)
- ou qualquer combinação, por exemplo "od -bi test.bin" ou "od -bc ex71.c | head +2"

Exercício 7.1:

(i) Escreva um programa (em C) para ler todo o conteúdo do ficheiro "test.bin" criado no exemplo 7.1 para um vector usando as funções de open() e read() e depois imprimir no ecrã os valores dos inteiros lidos (512 513 ….521 etc.) usando printf()

Vai precisar da função **open()** com opção O_RDONLY e a função **read()**

(ii) **Altere** o programa para usar "shorts" em vez de "ints" e ver o resultado !

(iii) **Experimente** a utilização do programa ***strace*** (system call trace) na execução dos dois programas.

- O comando é por exemplo : $ strace ./ex71
- Deverá tentar perceber porque o creat() devolve o descritor de ficheiro com valor de três (3)

Exemplo 7.2:
O programa seguinte ilustra a cópia da entrada estandardizada (teclado) para a saída estandardizada (ecrã). Compilar e Executar o programa.

```
/* ex72.c : compilar cc -Wall -c ex72.c -o ex72 */
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
#include <errno.h>
#define BUFFSIZE 128
void ioCopy (int IN, int OUT);
int main(){
    ioCopy (STDIN_FILENO, STDOUT_FILENO);   // 0 , 1
    return(0);
}
void ioCopy (int IN, int OUT)  //no error reporting
{
   int n;
   char buf[BUFFSIZE];
   while ( ( n = read (IN, buf, BUFFSIZE)) > 0)
   {
      if (write (OUT, buf, n) != n)
         perror("Erro de escrita!\n");
      // printf(stderr,"Leitura/Escrita de %d bytes\n",n);
   }
   if (n < 0)
     perror("Erro de leitura!\n");
}
```

- Compilar *cc –Wall exemplo72.c –o exemplo72*
- Executar usando o teclado como input *exemplo72 (lembrar: ctrl-d para EOF)*
- Executar usando redireccionamento *exemplo72 < exemplo72.c*

Exercício 7.2:

Baseado no programa anterior escreva um programa principal para copiar o conteúdo dum ficheiro para outro usando I/O de baixo nível. Os nomes dos dois ficheiros são fornecidos como argumentos do programa. A cópia do ficheiro é feita por blocos de 128 bytes.

Por exemplo ./exercicio72 exercicio72.c backup.c

Deverá verificar todos os casos de erro e produzir mensagens de error apropriado
Dica: Utilizar a função perror().

Um esboço do programa é dado em baixo

```
main( int argc, char *argv[] )
{
    int fdIn = open( argv[1]  ..);
    if erro ..
    int fdOut = creat( argv[2] ..… )
    if erro …
    ioCopy( fdIn, fdOut);
}
```

Função **lseek**

Esta função permite alterar a posição relativa (*offset*), a partir da qual são feitas outras operações sobre um ficheiro. Assim:

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

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

Retorna: novo *file offset* se OK, -1 em caso de erro

Qualquer ficheiro aberto tem um *offset* associado. Um *offset* é um inteiro não-negativo que mede o número de bytes a partir dum ponto do ficheiro. As operações de leitura e escrita num ficheiro são normalmente feitas no offset corrente do ficheiro, o que faz com que o offset seja incrementado pelo número de bytes lidos ou escritos. Por defeito, este offset é inicializado a 0 quando um ficheiro é aberto, a não ser que a opção O_APPEND seja especificada. A interpretação do *offset* depende do valor do argumento `whence`:

- Se `whence==SEEK_SET`, então o *offset* é igual ao número de bytes em `offset` contados a partir do início do ficheiro.
- Se `whence==SEEK_CUR`, então o *offset* é igual ao seu valor corrente acrescentado do valor em `offset`. O valor do argumento `offset` pode ser positivo ou negativo.
- Se `whence==SEEK_END`, então o *offset* é igual ao tamanho do ficheiro acrescentado do valor em `offset`. O valor do argumento `offset` pode ser positivo ou negativo.

Para mais informação, veja-se o manual *on-line* (man lseek).

Exemplo 7.3:

O programa abaixo ilustra a utilização da função lseek() para ler os últimos dois valores do vector escrito no exemplo 7.1

```
int main(void)
{
        int fd, vec[2];
        fd = open("test.bin", O_RDONLY ) ;
        lseek(fd, sizeof(int)*8,SEEK_SET)
        read(fd, &vec[0] , sizeof(int)*2);
        printf("Ultimos valores do vector %d %d\n", vec[0],vec[1]);
        return (0);
}
```

Exemplo 7.4:

O programa abaixo ilustra a utilização da função lseek() e como se pode criar um ficheiro com um *buraco*, isto é, sem dados pelo meio. De facto, o offset dum ficheiro pode ser maior do que o tamanho actual do ficheiro. Quaisquer bytes num ficheiro que não tenham sido escritos são lidos como 0.

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

#define FILE_MODE (S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH) //permissões típicos

char buf1[] = "abcdef", buf2[] = "ABCDEF";

int main()
{
        int fd;
        if ((fd = creat("file.hole", FILE_MODE)) < 0)
                printf("Erro na criacao de ficheiro!\n");

        if (write(fd, buf1, 10) != 10)
                fprintf(stderr,"Erro de escrita em buf1!\n");
        // offset now = 10

        if (lseek(fd, 40, SEEK_SET) == -1)
                fprintf(stderr,"Erro no posicionamento!\n");
        // offset now = 40

        if (write(fd, buf2, 10) != 10)
                fprintf(stderr,"Erro de escrita em buf2!\n");
        // offset now = 50

        return(0);
}
```

Compilar e Analisar os avisos do compilador: `bash$ **cc -Wall exemplo7-4.c**`

A execução do programa fornece o seguinte:

```
bash$ ./a.out
bash$ ls -l file.hole
-rw-r--r--    1 a15583   alunos         50 Mar 30 18:14 file.hole

bash$ cat file.hole
…
bash$ od -c file.hole
0000000   a   b   c   d   e   f  \0   A   B   C  \0  \0  \0  \0  \0  \0
0000020  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0
0000040  \0  \0  \0  \0  \0  \0  \0  \0   A   B   C   D   E   F  \0  \0
0000060  \0  \0
```

O comando **od** permite ver o conteúdo do ficheiro **file.hole**. A flag –c serve para escrever o conteúdo do ficheiro em caracteres. Podemos ver que 30 bytes não foram escritos no meio do ficheiro, sendo vistos a 0. Os 7 dígitos no início de cada linha do ouput do **od** é o offset em octal p.ex 0000020 -> byte position 16

Nota que neste caso as variáveis buf1 e buf2 estavam contiguas (lado ao lado) em memoria. Desta maneira a escrever 10 bytes a partir do buf1 também escreveremos 3 bytes (ABC) do buf2.
Isto, variáveis contiguas, pode não ser o caso quando execute o programa.

Exercícios

Exercício 7.3: Fácil

Escreva um programa para pedir o índice do inteiro a ler que está no ficheiro "test.bin" criado no exemplo 7.1 e depois imprimir no ecrã só o valor neste índice.
Deverá utilizar a função lseek() para posicionar o ficheiro no sitio correto e a função read() para ler apenas um inteiro (4 bytes) e a função printf() para imprimir o valor formatado no ecrã. Pode fazer em ciclo até que o utilizador introduza um indice negativo.

Exemplo de Execução

Introduza um índice : 1
Valor no índice 1 é 513
Introduza um índice : 4
Valor no índice 1 é 516
Introduza um índice : -1
Terminação

Exercício 7.4: Fácil

Escreva um programa que utilize I/O de baixo-nível que determine o número de linhas dum ficheiro de texto.

Exercício 7.5: Difícil

Escreva um programa que utilize I/O de baixo-nível para imprimir as últimas *n>0* linhas dum ficheiro de texto – numa maneira eficiente.

Uma solução possível implica saltar para o fim do ficheiro com lseek e a recuar em blocos de K bytes. A seguir ler os K bytes por ordem inversa para tentar identificar a posição da nsima '\n'

Exercício 7.6: Difícil

Escreva um programa eficiente que utilize I/O de baixo-nível para comparar dois ficheiros e que escreva no ecrã as linhas que são diferentes.

Uma solução possível é ler um bloco de Max_Linha bytes (255) de cada ficheiro para um buffer, identificar os fins de linha (e reposicionar o file offset com lseek) e imprimir para a ecrã as linhas se foram diferentes.

Faça uma solução (esta versão será muito mais simples) usando a biblioteca do alto nível, fgets(..)

Exercício 7.7:

Experimente a utilização do programa strace (system call trace) na execução dos programas.