Programar com Ficheiros

Todos já Programaram com Ficheiros

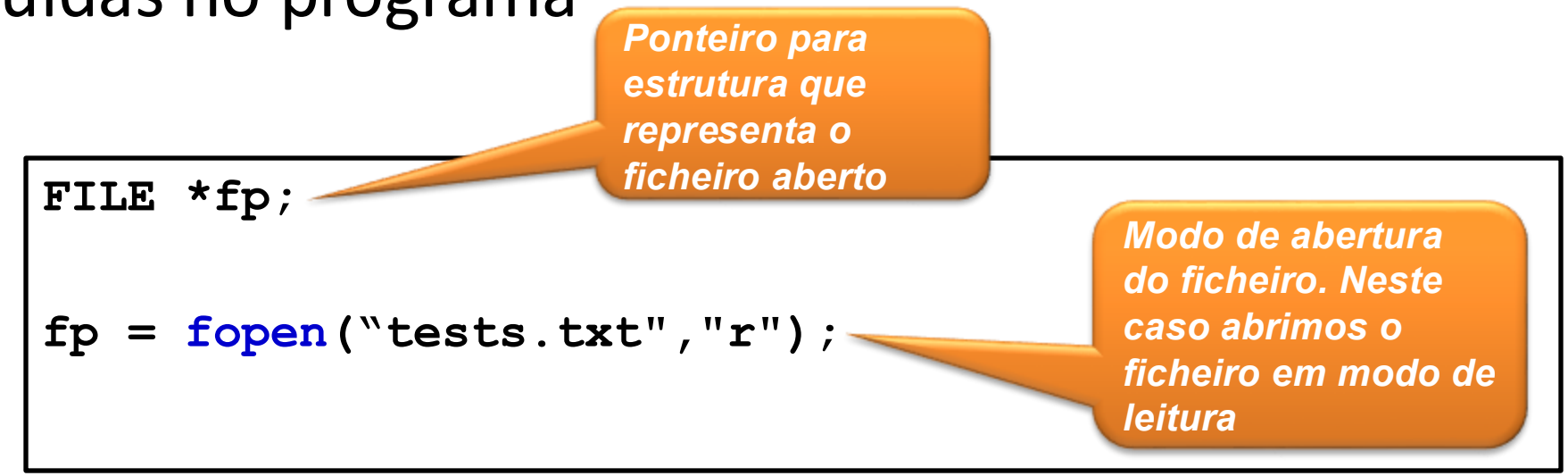
- Programar com ficheiros é dos aspetos mais básicos da aprendizagem de programação

- Nas linguagens de alto nível está incorporado no modelo

```
f = open("tests.txt", "r")
print(f.read())
```

Programar com ficheiros em C

- Em linguagens mais antigas é uma funcionalidade proporcionada por bibliotecas que têm de estar explicitamente incluídas no programa



Exemplo 1

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
  FILE *fp;

  fp = fopen("teste.txt", "r");
  if (fp == NULL) {
    printf("teste.txt: No such file or directory\n");
       exit(1);
  }

  return 0;
}
```

Fundamental incluir a biblioteca

Se não conseguir abrir, fp fica igual a NULL

Exemplo 2

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

int main()
{
  FILE *myfile; int i;
  float mydata[100];
  myfile = fopen("info.dat", "r");
  if (myfile== NULL) {
    perror("info.dat");
    exit(1);
  }
  for (i=0;i<100;i++)
     fscanf(myfile,"%f",&mydata[i]);

  fclose(myfile);
  return 0;
}
```

Lê um conjunto de 100 números reais (float) guardados num ficheiro

Funções Sistema

- Podemos dividir as funções relacionadas com o sistema de ficheiros em seis grupos:
  - Abertura, criação e fecho de ficheiros
  - Operações sobre ficheiros abertos
  - Operações complexas sobre ficheiros
  - Operações sobre diretórios
  - Operações de gestão dos sistemas de ficheiros.
  - Acesso a ficheiros mapeados em memória

O que Ganho/Perco com a Utilização das Funções Sistema

Prós:

- Em geral, são funções de mais baixo nível, logo permitem maior controlo
- Algumas operações sobre ficheiros só estão disponíveis através desta API

Contras:

- Normalmente, programa que usa `stdio` é mais simples e otimizado

Abertura, Criação e Fecho de Ficheiros

Operações	Genéricas	Linux	Descrição
Simples	Fd := Abrir (Nome, Modo) Fd := Criar (Nome, Proteção)	`int open(const char *path, int flags,mode_t mode)`	Abre um ficheiro Cria um novo ficheiro
	Fechar (Fd)	`int close(int fd)`	Fecha um ficheiro

- Abrir e Criar em Linux é a mesma função
- A função `open()` tem numerosos parâmetros

Abrir e Fechar ficheiros

- É mantida uma **Tabela de Ficheiros Abertos** por **processo**

Processo: instância de um programa em execução

- Abrir um ficheiro:
  - Pesquisar o diretório
  - Verificar se o processo tem permissões para o modo de acesso que pede
  - Copiar a meta-informação (o *inode)* para memória
  - Devolve um *file descriptor*, que é usado como referência para essa área de memória

- Fechar do ficheiro:
  - Liberta a memória que continha a meta-informação do ficheiro
  - Caso necessário, atualiza essa informação no sistema de memória secundária

Parametrização do Open

- O *open* tem numerosas parametrizações que permitem distinguir não só o modo de utilização, mas também o tipo de utilização dos ficheiros
- As mais usuais são intuitivas
- As mais complexas têm a ver com a otimização das operações de I/O

Table 4-3: Values for the *flags* argument of *open()*

Flag	Purpose
O_RDONLY	Open for reading only
O_WRONLY	Open for writing only
O_RDWR	Open for reading and writing
O_CLOEXEC	Set the close-on-exec flag (since Linux 2.6.23)
O_CREAT	Create file if it doesn't already exist
O_DIRECTORY	Fail if *pathname* is not a directory
O_EXCL	With O_CREAT: create file exclusively
O_LARGEFILE	Used on 32-bit systems to open large files
O_NOCTTY	Don't let *pathname* become the controlling terminal
O_NOFOLLOW	Don't dereference symbolic links
O_TRUNC	Truncate existing file to zero length
O_APPEND	Writes are always appended to end of file
O_ASYNC	Generate a signal when I/O is possible
O_DIRECT	File I/O bypasses buffer cache
O_DSYNC	Provide synchronized I/O data integrity (since Linux 2.6.33)
O_NOATIME	Don't update file last access time on *read()* (since Linux 2.6.8)
O_NONBLOCK	Open in nonblocking mode
O_SYNC	Make file writes synchronous

Privilégios de Acesso aos Ficheiros

Autenticação de Processos

- Cada processo corre em nome de um utilizador (UID)/grupo (GID)
- Atribuídos ao primeiro processo criado quando o utilizador se autentica (*log in*)
  - Obtidos do ficheiro `/etc/passwd` no momento do *login*
- Processos filho criados pelos processos desse utilizador herdam UID/GID

Controlo dos Direitos de Acesso

- O modelo de autorização mais frequente baseia-se numa Matriz de Direitos de Acesso

		Objectos	
Utilizadores	**1**	**2**	**3**
1	Ler	-	Escrever
2	-	Ler/Escrever/ Executar	-
3	-	-	Ler

- Colunas designam-se Listas de Direitos de Acesso (ACL)
- Linhas designam-se por Capacidades

Exemplo: ACL em Unix

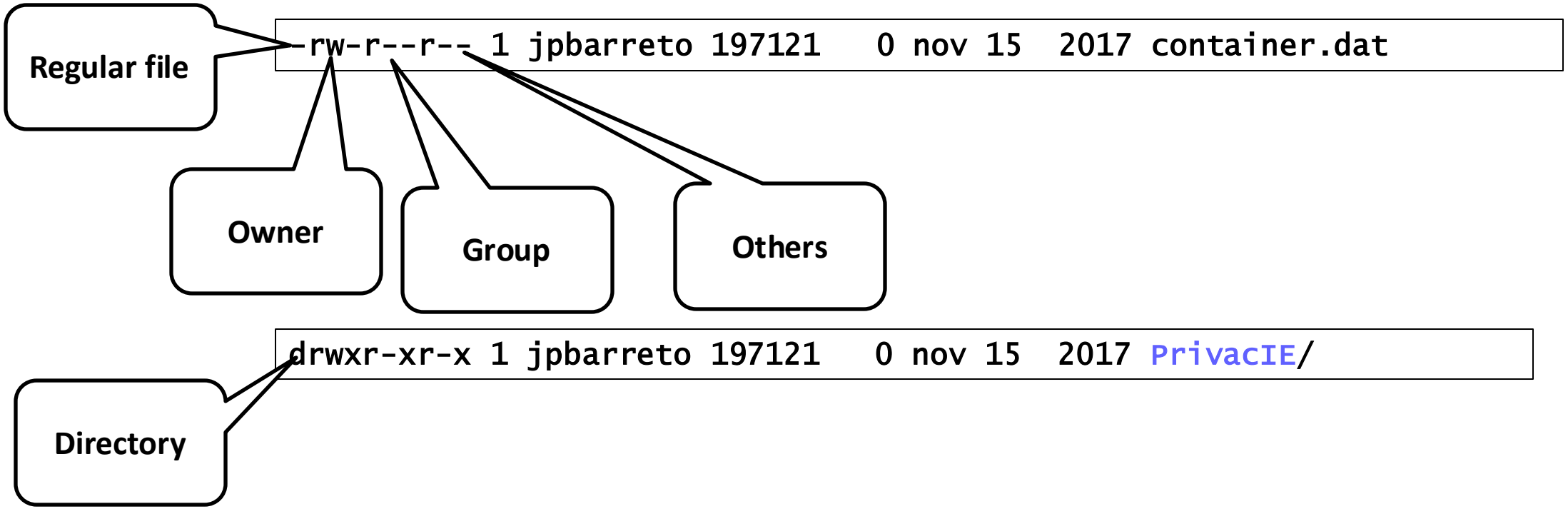
Nos SGF utiliza-se normalmente listas de controlo de acesso ACL guardadas na metadata dos ficheiros

```
$ ls -l
total 7
-rw-r--r-- 1 jpbarreto 197121 113 nov 20  2017 0SXW49AO.cookie
-rw-r--r-- 1 jpbarreto 197121   0 nov 15  2017 container.dat
-rw-r--r-- 1 jpbarreto 197121  91 jan 30  2018 deprecated.cookie
drwxr-xr-x 1 jpbarreto 197121   0 fev 20  2018 DNTException/
drwxr-xr-x 1 jpbarreto 197121   0 jul 15 12:04 ESE/
-rw-r--r-- 1 jpbarreto 197121 119 jan 25  2018 F0RK22C9.cookie
drwxr-xr-x 1 jpbarreto 197121   0 ago 16 23:36 Low/
-rw-r--r-- 1 jpbarreto 197121  95 nov 15  2017 N036J89L.cookie
-rw-r--r-- 1 jpbarreto 197121 470 dez  7  2017 OCPOPE1W.cookie
drwxr-xr-x 1 jpbarreto 197121   0 nov 15  2017 PrivacIE/
-rw-r--r-- 1 jpbarreto 197121 155 jan 26  2018 URPQ8UQ1.cookie
-rw-r--r-- 1 jpbarreto 197121 112 dez  7  2017 ZDCLU5P1.cookie
```

Exemplo: ACL em Unix

Simplificação das ACL do Unix considerar três grupos

- Dono
- Grupo
- Restantes utilizadores



Controlo dos Direitos de Acesso

1. Processo pede para executar operação sobre objeto gerido pelo núcleo
2. Núcleo valida se na ACL do ficheiro o UID/GID correspondente ao processo tem direitos para executar aquela operação sobre aquele objeto
3. Se sim, núcleo executa operação; se não, retorna erro

Copy the file named argv[1] to a new file named in argv[2]

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

#define BUF_SIZE 1024

int
main(int argc, char *argv[])
{
    int inputFd, outputFd, openFlags;
    mode_t filePerms;
    ssize_t numRead;
    char buf[BUF_SIZE];

    if (argc != 3 || strcmp(argv[1], "--help") == 0)
        usageErr("%s old-file new-file\n", argv[0]);

    /* Open input and output files */

    inputFd = open(argv[1], O_RDONLY);
    if (inputFd == -1)
        errExit("opening file %s", argv[1]);

    openFlags = O_CREAT | O_WRONLY | O_TRUNC;
    filePerms = S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP |
                S_IROTH | S_IWOTH;      /* rw-rw-rw- */
    outputFd = open(argv[2], openFlags, filePerms);
    if (outputFd == -1)
        errExit("opening file %s", argv[2]);
```

Operações sobre Ficheiros Abertos

Operações	Genéricas	Linux
Ficheiros Abertos	Ler (Fd, Tampão, Bytes)	int read(int fd, void *buffer, size_t count)
	Escrever (Fd, Tampão, Bytes)	int write(int fd, void *buffer, size_t count)
	Posicionar (Fd, Posição)	int lseek(int fd, off_t offset, int origin)

```
int read(int fd, void *buffer, size_t count)
```

- O argumento `count` especifica o número máximo de *bytes* a ler.

- O argumento *buffer* indica o endereço de memória no qual os dados devem ser colocados. Este buffer deve ter pelo menos a dimensão indicada em `count`. **As chamadas do sistema não alocam memória para os dados que retornam**

- Uma chamada bem sucedida retorna o número de *bytes* realmente lidos, ou 0 se o fim do ficheiro for encontrado. Em caso de erro o retorno é -1

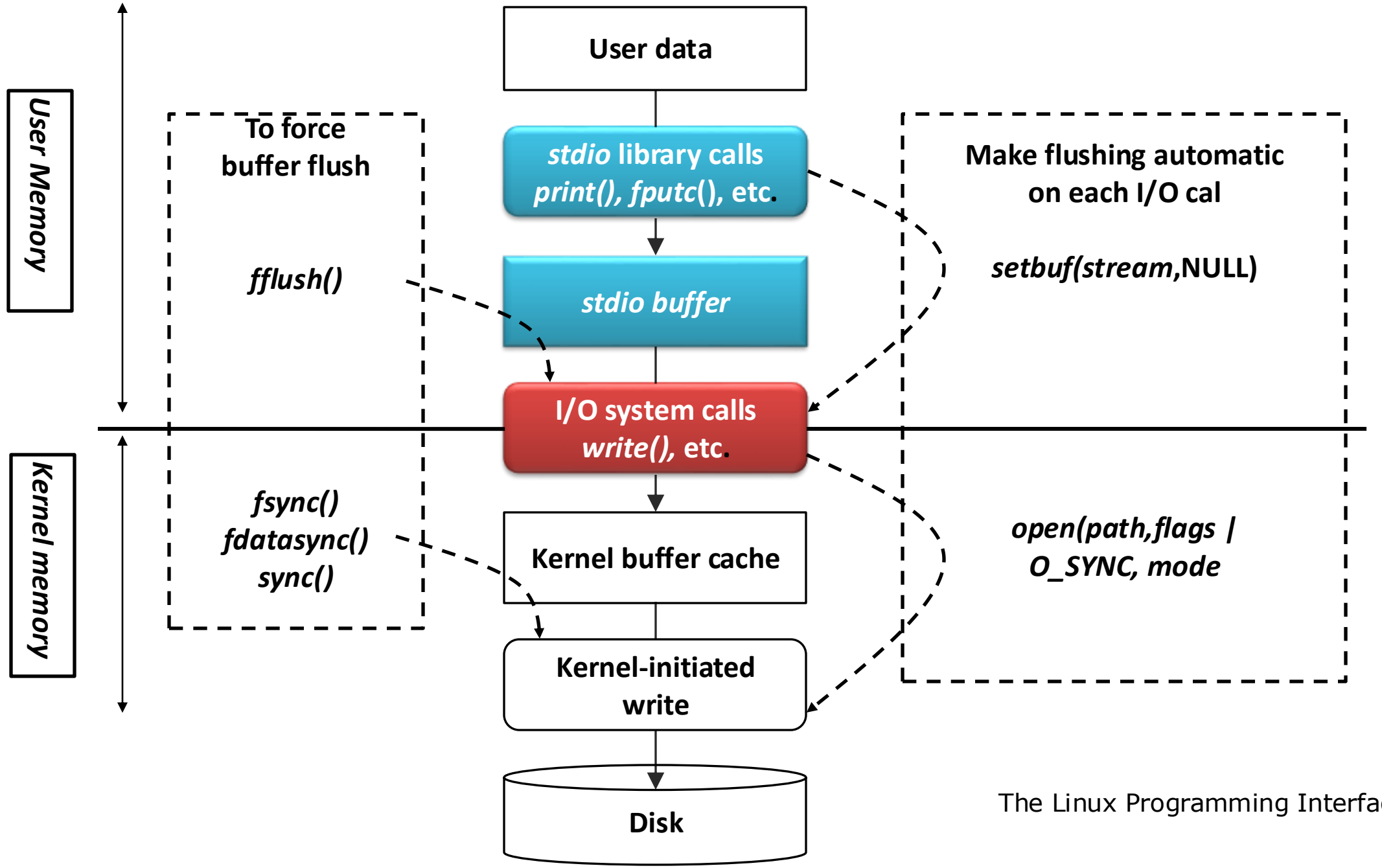
- Uma chamada a `read()` pode ler menos bytes que o solicitado. No caso de um ficheiro regular, a razão provável é que estávamos perto do fim do ficheiro

```
int write(int fd, void *buffer, size_t count)
```

- Os argumentos para `write()` são semelhantes aos de `read()`
  - `buffer` é o endereço dos dados a serem escritos;
  - `count` é o número de *bytes* a escrever a partir de `buffer`
  - `fd` é um *filedescriptor* referindo-se ao ficheiro para o qual os dados devem ser gravados.

- Se tiver sucesso, `write()` retorna o número de bytes realmente escritos, pode ser menos do que `count`.

- Um retorno bem sucedido de `write()` não garante que os dados foram transferidos para o disco, porque o núcleo do SO efectua o *buffering* de I/O para o disco, a fim de reduzir as transferências



The Linux Programming Interface

Copy the file named argv[1] to a new file named in argv[2]

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

#define BUF_SIZE 1024

int
main(int argc, char *argv[])
{
    int inputFd, outputFd, openFlags;
    mode_t filePerms;
    ssize_t numRead;
    char buf[BUF_SIZE];

    if (argc != 3 || strcmp(argv[1], "--help") == 0)
        usageErr("%s old-file new-file\n", argv[0]);

    /* Open input and output files */

    inputFd = open(argv[1], O_RDONLY);
    if (inputFd == -1)
        errExit("opening file %s", argv[1]);

    openFlags = O_CREAT | O_WRONLY | O_TRUNC;
    filePerms = S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP |
                S_IROTH | S_IWOTH;      /* rw-rw-rw- */
    outputFd = open(argv[2], openFlags, filePerms);
    if (outputFd == -1)
        errExit("opening file %s", argv[2]);
```

```c
 /* Transfer data until we encounter end of input or
an error */

    while((numRead = read(inputFd, buf, BUF_SIZE))
            > 0)
        if (write(outputFd, buf, numRead) !=
            numRead)
            fatal("couldn't write whole buffer");
    if (numRead == -1) errExit("read");

    if (close(inputFd) == -1)
        errExit("close input");
    if (close(outputFd) == -1)
        errExit("close output");

    exit(EXIT_SUCCESS);
}
```

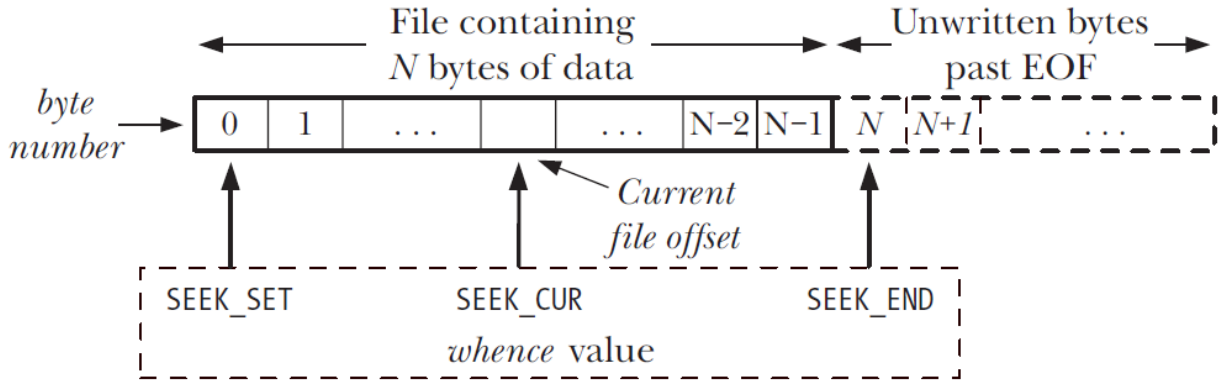
O Cursor

- Para qualquer ficheiro aberto, o núcleo mantem um cursor
- Este é o índice que indica a posição no ficheiro onde a próxima operação de `read()` ou `write()` se executará
- O cursor é expresso como o deslocamento em *bytes* a partir do inicio (o primeiro *byte* do ficheiro tem o deslocamento 0)

Avança automaticamente com cada *byte* lido ou escrito

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

- O cursor pode ser alterado explicitamente pelo programa
- O `offset` especifica um valor em *bytes*.
- O argumento `whence` indica o ponto base a partir do qual o deslocamento deve ser interpretado,



Comando do *Shell* `strace`

- Usar o comando `strace` para ver as chamadas sistema que os programas estão a executar
- Usem para ver a execução deste programa de demonstração

- The strace tool provides an awesome way to see what programs are up to. By running it, you can trace which system calls a program makes, see the arguments and return codes, and generally get a very good idea of what is going on.
- The tool also takes some arguments which can be quite useful.
- For example, -e trace=open,close,read,write only traces calls to those system calls and ignores all others.
- read the man pages and find out how to harness this wonderful tool.

Conclusões

- Neste módulo estivemos interessados na programação de ficheiros com base nas *system calls* disponibilizadas pelo Linux
- A *system call* `open()` é muito relevante pela elevado grau de parametrização e pela sua relação com o modelo de permissões do Linux que se aplica aos ficheiros e à generalidade dos objetos do núcleo
- As restantes *system calls* são muito semelhantes às que conheciam da utilização da `stdio`. Mas necessitam de alguma atenção para perceber os detalhes da semântica de execução e o significado dos parâmetros