

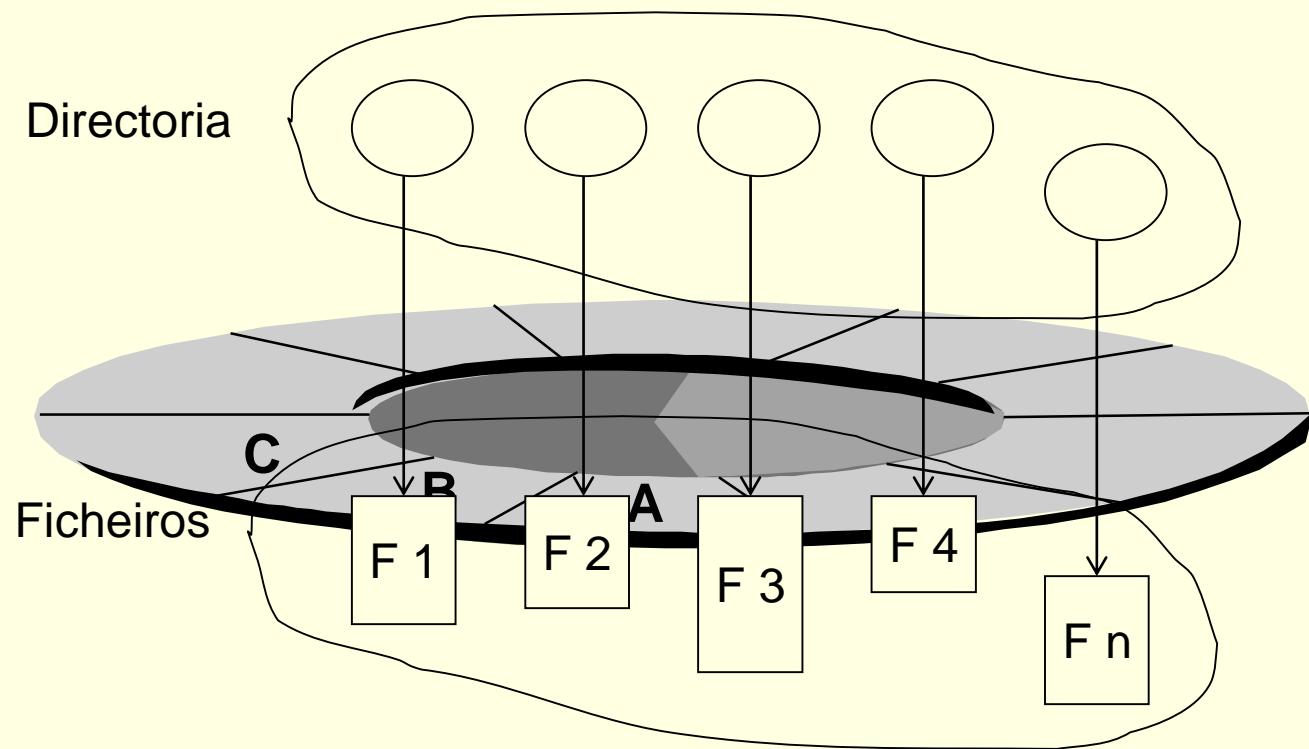
# *Fundamentos de Sistemas de Operação*

Unix Windows NT Netware Mac OS DOS/VMS Vax/VMS  
Linux Solaris HP/UX AIX UX Mach  
Chorus

*Gestão de Ficheiros*  
Sistema de Ficheiros em Disco –  
Organização e Gestão do Espaço (em disco) ...

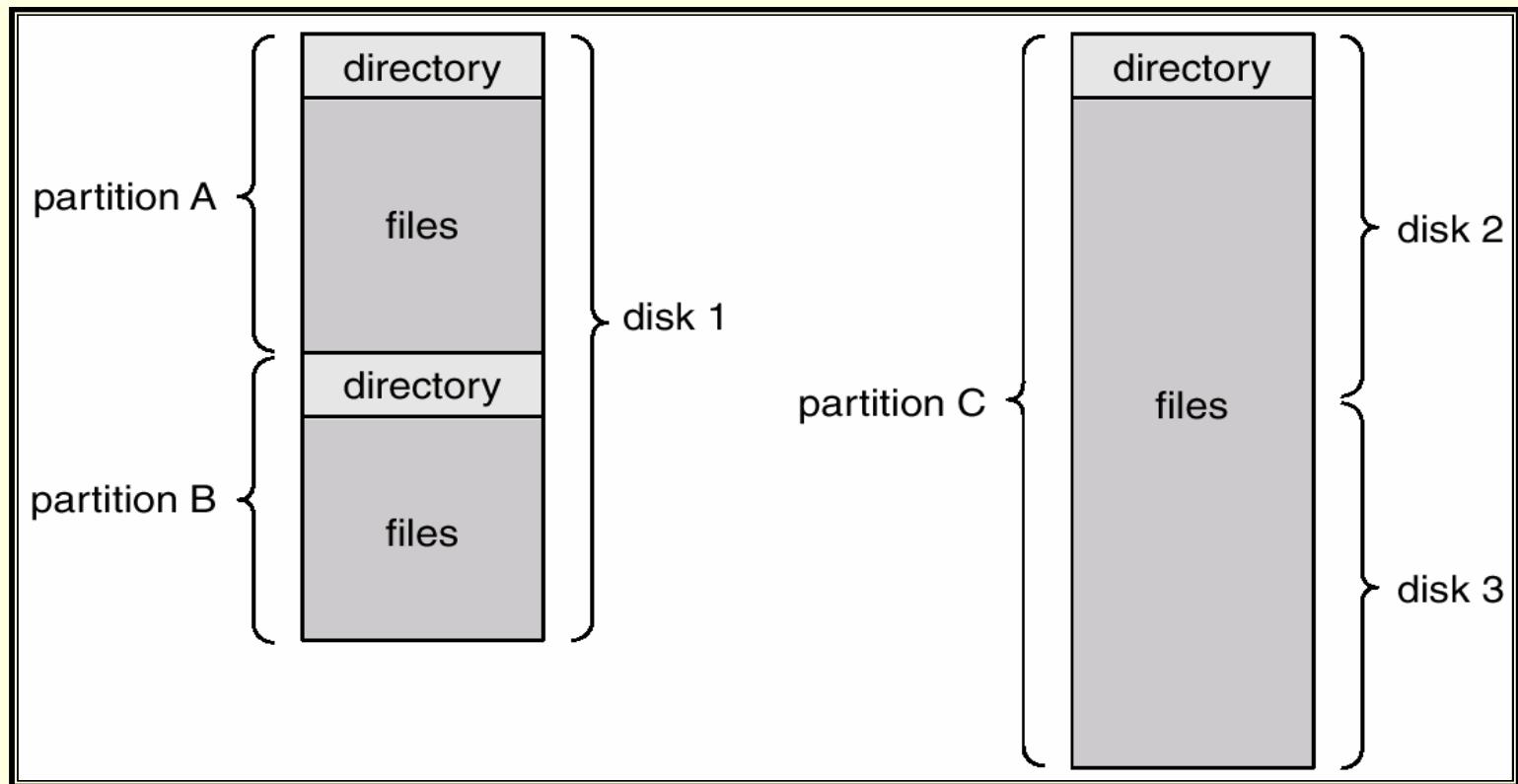
# O Sistema de Ficheiros em disco

- Como armazenar no disco as “directorias” e os “ficheiros”?



# O Sistema de Ficheiros em disco

- Um *layout simples*:

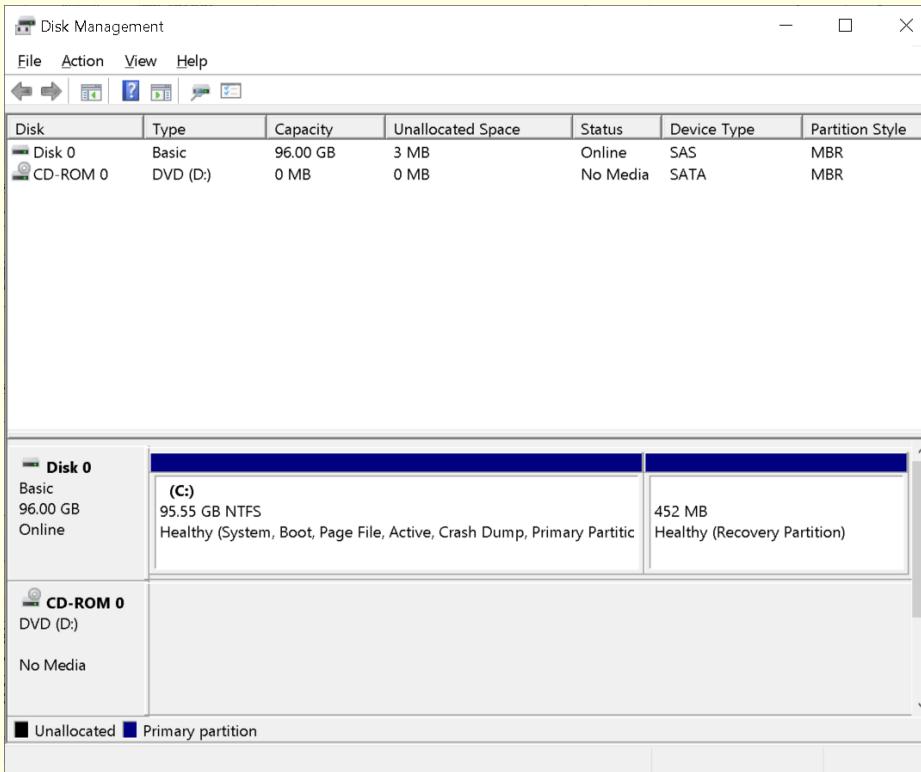


# Partições (1)

- O que é uma partição?
  - Uma partição é uma parte de um disco físico – formada por um conjunto de blocos contíguos;
  - As partições de um disco são descritas numa estrutura de dados – a Tabela de Partições (geralmente localizada no início do disco)
- Um sistema de ficheiros pode “assentar” as suas estruturas de dados
  - directamente sobre uma partição,
    - caso das partições A e B (fig. da página 3)
  - ou sobre um disco, ou volume lógico (e.g., um volume RAID)
    - caso da (incorrectamente chamada) partição C

# Partições (2)

- Windows 10: partições vistas no Disk Manager (GUI)



# Partições (3)

- Windows 10: partições vistas no **diskpart** (CLI)

```
C:\WINDOWS\system32\diskpart.exe
Microsoft DiskPart version 10.0.18362.1
Copyright (C) Microsoft Corporation.
On computer: VMAHLER-MOBILE

DISKPART> list disk

Disk ###  Status     Size      Free      Dyn  Gpt
-----  -----
Disk 0    Online     96 GB    1024 KB

DISKPART> select disk 0

Disk 0 is now the selected disk.

DISKPART> list partition

Partition ###  Type      Size      Offset
-----  -----
Partition 1   Primary    95 GB    1024 KB
Partition 2   Recovery   452 MB    95 GB

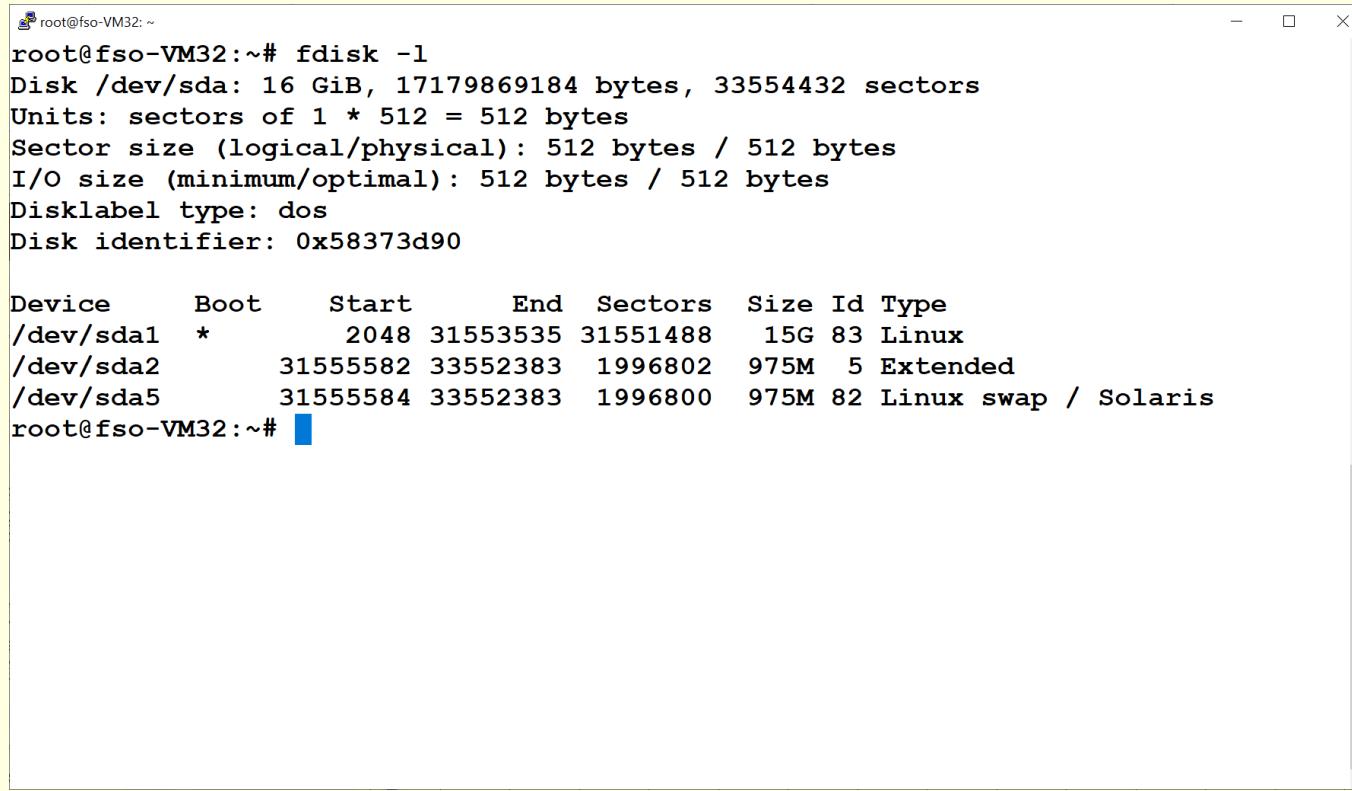
DISKPART> list volume

Volume ###  Ltr  Label        Fs  Type        Size  Status     Info
-----  --  --  -----  -----  -----  -----  -----
Volume 0    D            DVD-ROM    0 B  No Media
Volume 1    C            NTFS       Partition  95 GB  Healthy   System

DISKPART>
```

# Partições (4)

## Linux: partições vistas no **fdisk** (CLI)



```
root@fso-VM32:~# fdisk -l
Disk /dev/sda: 16 GiB, 17179869184 bytes, 33554432 sectors
Units: sectors of 1 * 512 = 512 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes
Disklabel type: dos
Disk identifier: 0x58373d90

Device      Boot   Start     End   Sectors   Size Id Type
/dev/sda1    *     2048 31553535 31551488   15G 83 Linux
/dev/sda2          31555582 33552383 1996802  975M  5 Extended
/dev/sda5          31555584 33552383 1996800  975M 82 Linux swap / Solaris
root@fso-VM32:~#
```

# Partições (5)

## □ Linux: partições vistas no **parted** (CLI)

```
root@fso-VM32:~# parted
GNU Parted 3.2
Using /dev/sda
Welcome to GNU Parted! Type 'help' to view a list of commands.
(parted) print
Model: VMware, VMware Virtual S (scsi)
Disk /dev/sda: 17,2GB
Sector size (logical/physical): 512B/512B
Partition Table: msdos
Disk Flags:

Number  Start   End     Size    Type      File system    Flags
 1      1049kB  16,2GB  16,2GB  primary   ext4          boot
 2      16,2GB  17,2GB  1022MB  extended
 5      16,2GB  17,2GB  1022MB  logical   linux-swap(v1)

(parted) █
```

# *Um Sistema de Ficheiros simples: I*

## □ Para cada partição:

- Uma estrutura de dimensão fixa (“tabela”) para a directoria
  - Que, por isso, apenas poderá conter um número limitado de entradas com informação sobre os ficheiros existentes nessa partição...
  - Cada entrada pode ser: nome, início, comprimento.
  - Para “informatizar” o SF há que decidir os tipos de dados...
    - `char nome[8]; unsigned int inicio, comprimento;`
- O restante espaço será disponibilizado para os ficheiros...
  - Como geri-lo?

# *Uma directoria simples:* exercício

- Considere um SF simples, só com uma directoria, e em que esta é armazenada num único bloco de disco:
  - Uma entrada é...

```
struct entrada {  
    char nome[8];  
    unsigned int inicio, comprimento;  
}
```
  - Implemente:
    - Um vector de struct entrada capaz de conter o máximo de entradas que podem ser armazenadas num único bloco (512 bytes).
      - Quantas entradas são suportadas? Há espaço desperdiçado?

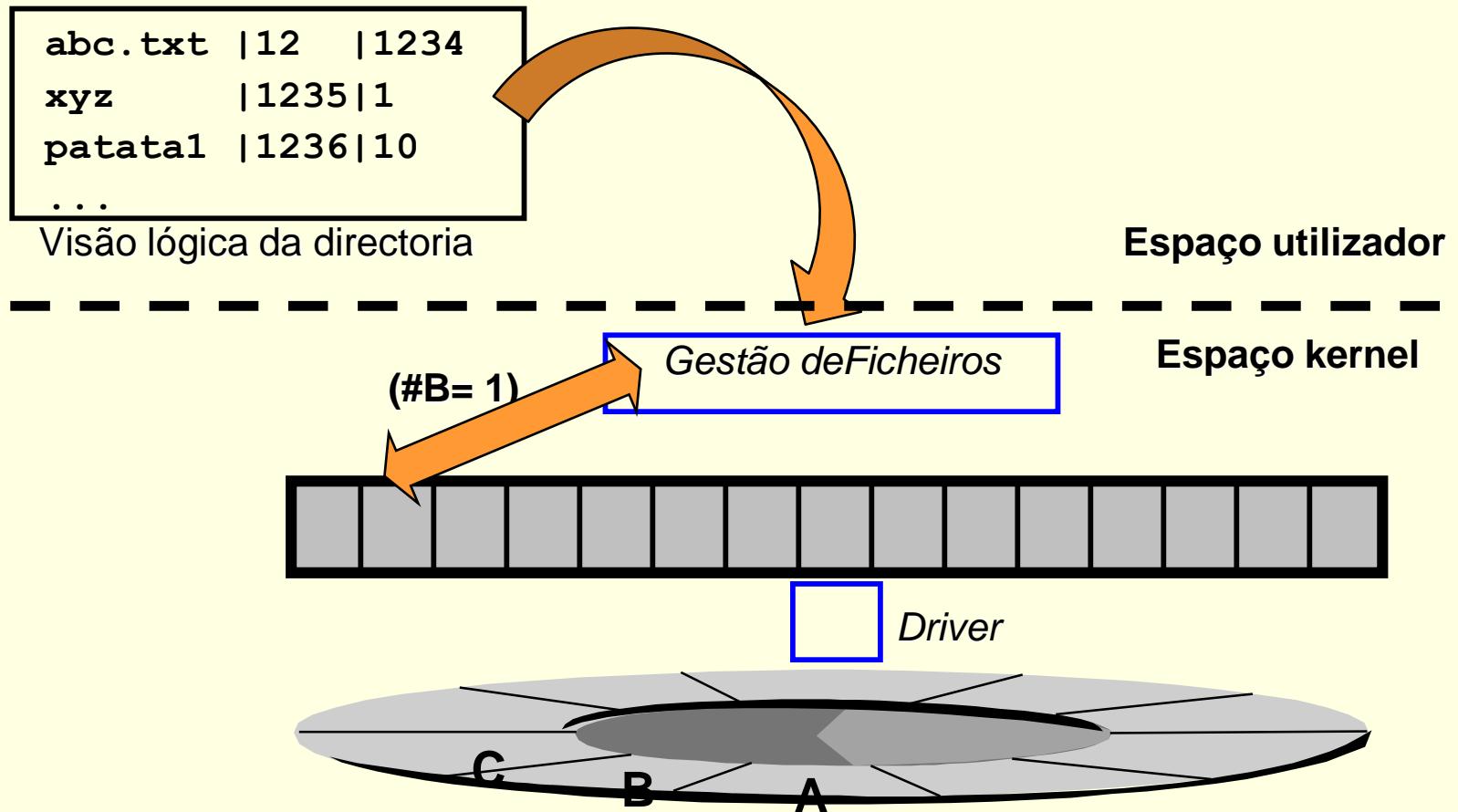
(continua)

# *Uma directoria “simples”: exercício*

(continuação)

- Imagine que existe uma função para ler um bloco de disco, dado o seu número (naturalmente haverá uma para escrever...):  
`void readBlock(int num, void * blk)`
- Implemente:
  - Use a função para ler o bloco 1 do disco (bloco que contém a directoria) e carrega a informação no vector de entradas.
  - Faça uma função que percorre esse vector e imprime os nomes e comprimentos dos ficheiros, de uma forma parecida com o comando `dir` do Windows (ou `ls` do Linux).

# *Uma directoria “simples”:* exercício



# *Um Sistema de Ficheiros “simples”: II*

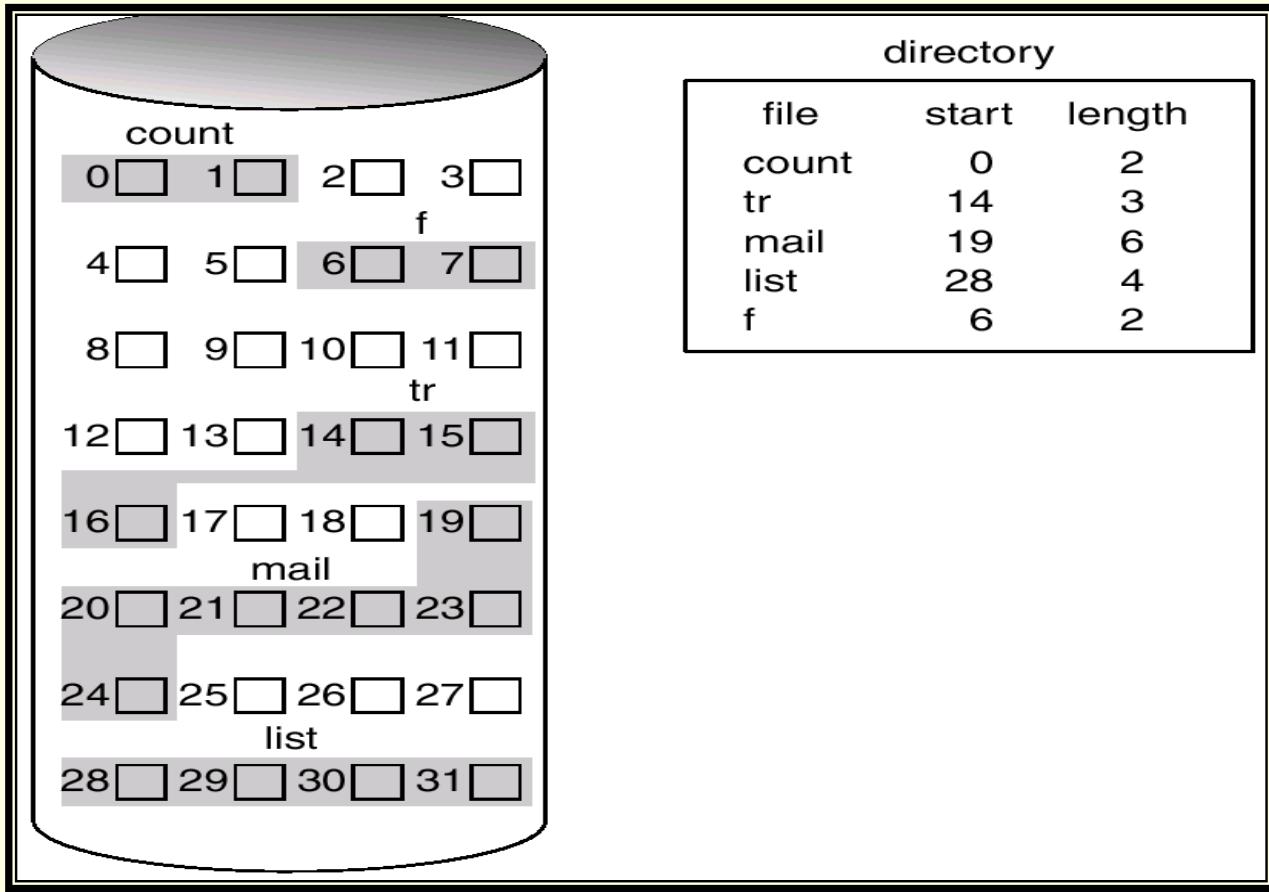
- Guardar os **Dados** de cada ficheiro:
  - Na estrutura, o `inicio` diz o #bloco onde começa um ficheiro, e tem-se o seu **comprimento** em bytes. Chega?
- Mas...
  - Quando queremos criar um ficheiro,
    - Para o guardar o nome, basta que haja uma entrada livre na directoria,
    - E para “ir buscar” o primeiro bloco? Como sabemos quais estão livres?
      - Subproblema a resolver: blocos livres vs. ocupados
  - Blocos pertencentes a um ficheiro:
    - Como indicá-los?

# *Atribuição contígua* (1)

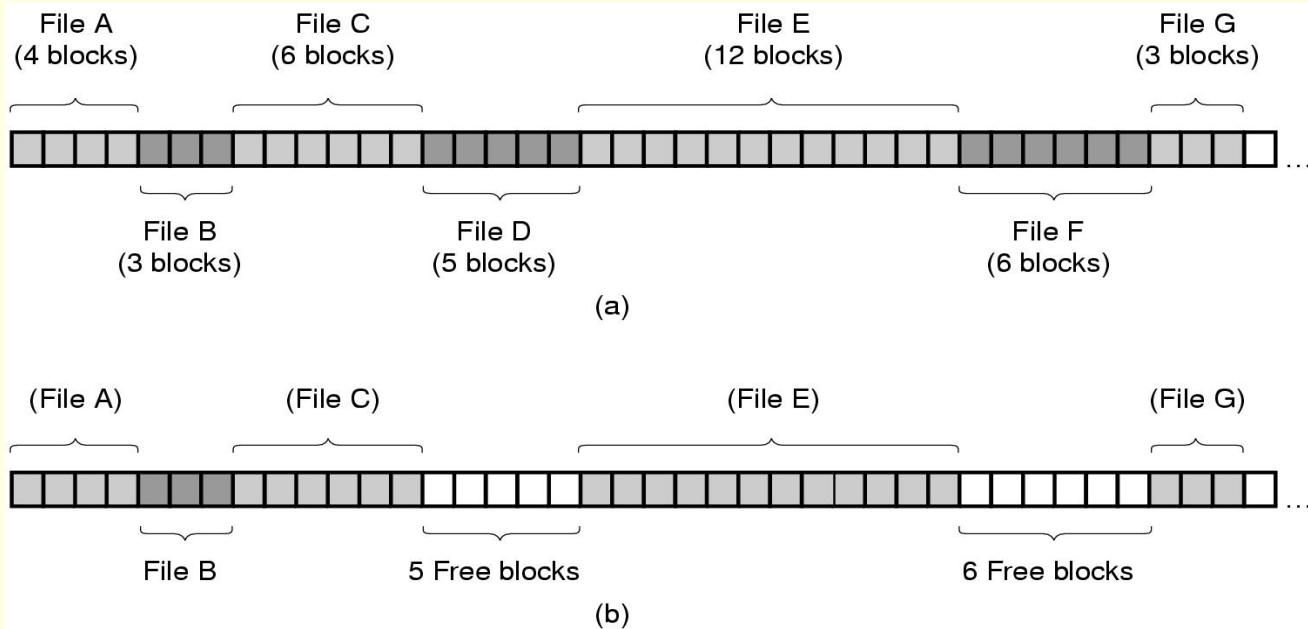
- Cada ficheiro ocupa um conjunto de blocos contíguos.
- Simples – só é necessário guardar o número do bloco inicial e o nº de blocos (melhor: o comprimento máximo, daí deduz-se o #blocos.  
**Nota:** e onde guardar o comprimento corrente do ficheiro?!)
- Vantagens:
  - Acesso directo facilmente suportado
  - Desempenho muito elevado em acesso sequencial
- Desvantagens:
  - Desperdício de espaço (fragmentação interna e externa!)
  - Quando o fich. é criado, tem de ser especificada a dimensão máxima
  - Dificuldade no crescimento dos ficheiros.

Isto é muito parecido com os problemas de GM por partições fixas ☺

# Atribuição contígua (2)



# Atribuição contígua (3)

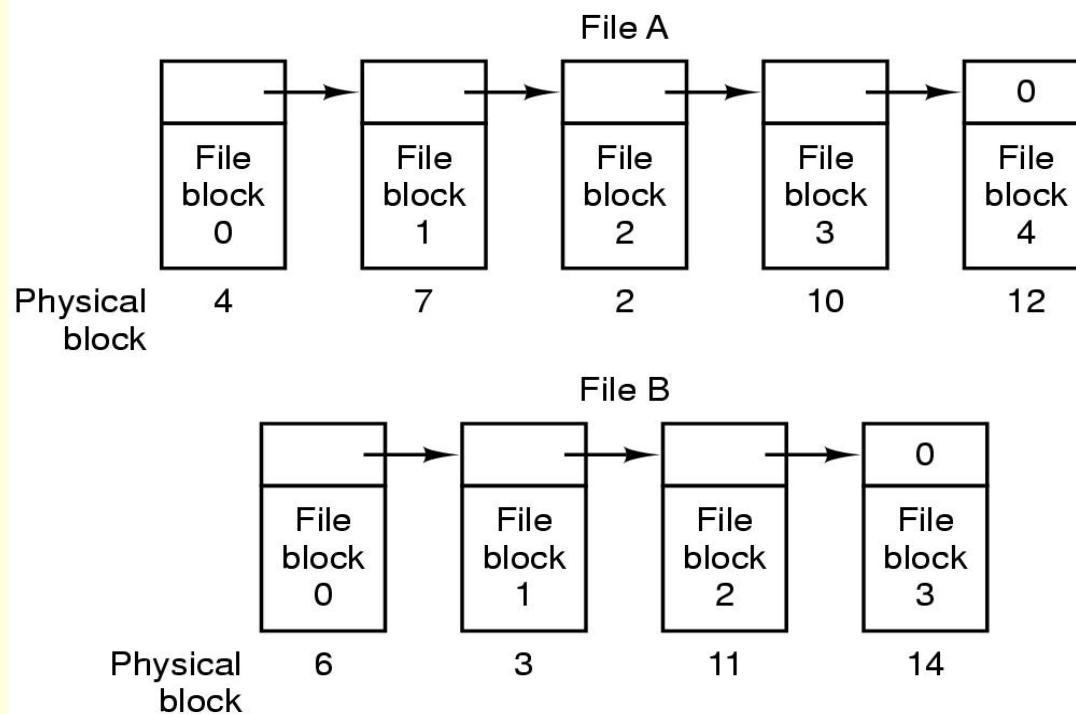


(a) Atribuição contígua de espaço para 7 ficheiros

(b) Estado do disco após a remoção dos ficheiros D e F

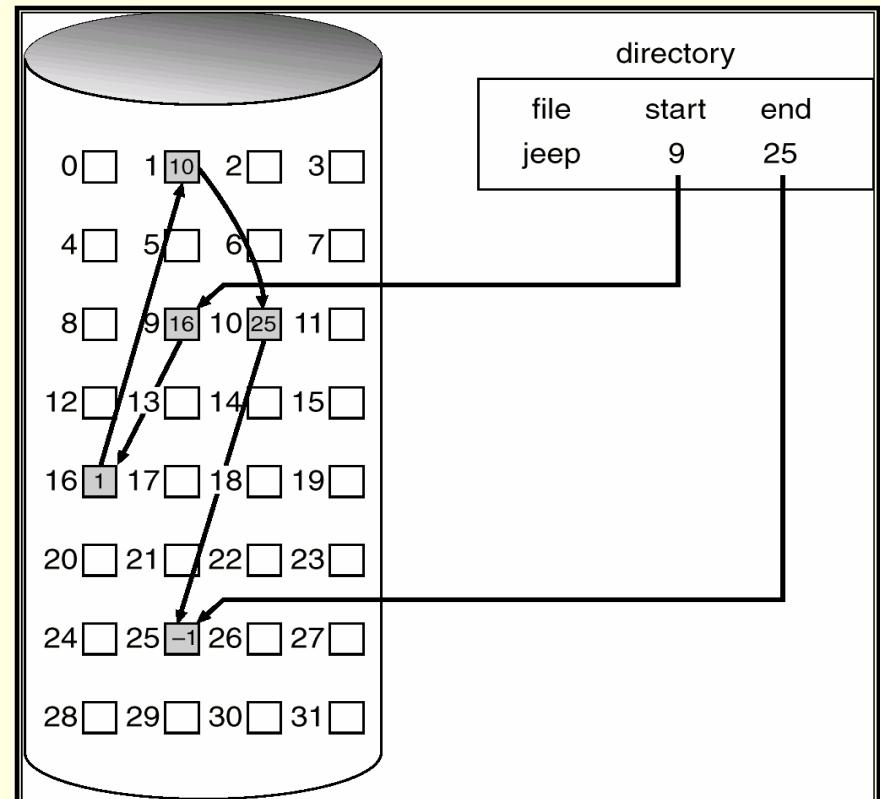
# Atribuição ligada (1)

- Cada bloco de um ficheiro ocupa uma qualquer posição.
- Aponta para o bloco seguinte desse mesmo ficheiro



# Atribuição ligada (2)

- Apontador para 1º bloco (e eventualmente último) de um ficheiro guardado na directória.



# *Atribuição ligada* (3)

## □ Vantagens:

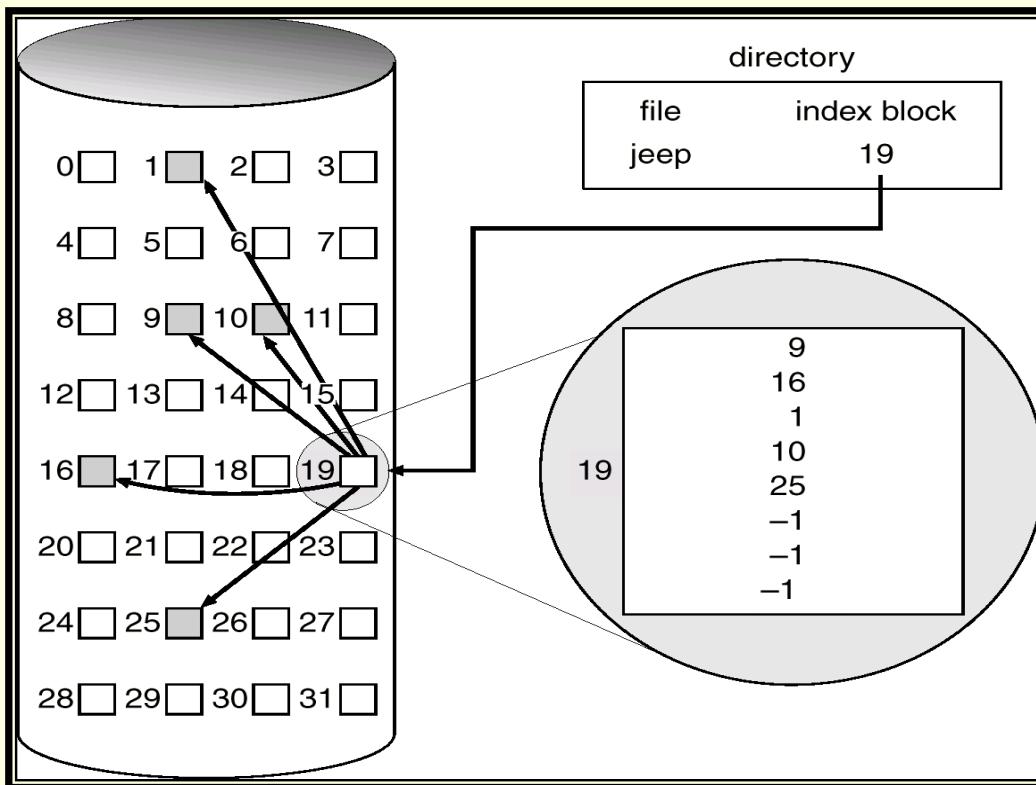
- Simples: só necessita de guardar o endereço inicial, e comprimento
- Pequeno desperdício de espaço (i.e., há fragmentação interna, mas não externa)

## □ Desvantagens:

- Acesso directo muito lento – para chegar ao bloco N é preciso ler os N-1 anteriores.
- Desperdício de espaço em apontadores, “fragmentação” dos dados (os dados de dois blocos só podem ser “agregados” em memória depois de “remover” espaço ocupado pelos apontadores)

# Atribuição indexada (1)

- Os apontadores para os blocos são guardados em blocos de índice.



O 1º bloco do fich. é o 9;  
o 2º é o 16;  
...  
o 5º é o 25. Não há mais!

# *Atribuição indexada (2)*

## □ Vantagens:

- Acesso directo rápido (máximo de 2 acessos, mas pode-se ter o índice em *cache*)
- Pequeno desperdício de espaço (não há fragmentação externa)

## □ Desvantagens:

- Espaço ocupado pela tabela de índices.
- Gestão da tabela de índices: tamanho fixo ou variável? Se variável, blocos de índice ligados entre si em lista ou por indexação “de ordem superior”?

# Atribuição: Casos reais

- Soluções híbridas, com o intuito aproveitar as vantagens de uma dada técnica e minorar as sua desvantagens...
  - FAT: MS-DOS, Windows 95, ...
    - Uma variante da atribuição ligada
  - ext2, 3, 4: Linux
    - Uma variante da atribuição indexada
- Outras melhorias
  - Usar “grupos de blocos” contíguos, em vez de um só bloco como unidade de atribuição; designados
    - clusters na FAT
    - extents no ext2