# Unidade 6 - Sistema de Ficheiros

# 6.1 Conceitos sobre Ficheiros

# Conceito de Ficheiro (1/1)

- existe uma variedade de dispositivos físicos de armazenamento persistente; o SO proporciona mecanismos de abstracção que simplificam a sua utilização
- *ficheiro*
  - colecção de informação inter–relacionada, com nome próprio
  - unidade lógica de armazenamento, independente do dispositivo
- tipicamente, os ficheiros albergam
  - programas
    - código fonte
    - código binário
  - dados
    - numéricos
    - alfabéticos / textuais (caracteres)
    - alfanuméricos
    - binários (imagem, vídeo, som, documentos, folhas de cálculo, ...)
- estrutura interna: sequência de bits, bytes, linhas, registos - depende da estrutura definida pelo criador; o conceito de ficheiro é muito genérico !

# Tipos de Ficheiros (1/1)

- o tipo de um ficheiro é um indicador da sua natureza, permitindo associar operações automáticas (.exe é executável, .png abre um visualizador, etc.)
  - em geral, o tipo é definido pela *extensão* no nome (.doc, .exe, etc.)
  - UNIX/Linux: a extensão é facultativa e não é vinculativa; o tipo de alguns ficheiros é definido por um *magic number* no seu início; o comando `file` retorna o tipo de um ficheiro (e.g. `file .bashrc`)
- tipos comuns de ficheiros regulares:

| file type | usual extension | function |
|---|---|---|
| executable | exe, com, bin or none | ready-to-run machine-language program |
| object | obj, o | compiled, machine language, not linked |
| source code | c, cc, java, perl, asm | source code in various languages |
| batch | bat, sh | commands to the command interpreter |
| markup | xml, html, tex | textual data, documents |
| word processor | xml, rtf, docx | various word-processor formats |

| file type | usual extension | function |
|---|---|---|
| library | lib, a, so, dll | libraries of routines for programmers |
| print or view | gif, pdf, jpg | ASCII or binary file in a format for printing or viewing |
| archive | rar, zip, tar | related files grouped into one file, sometimes compressed, for archiving or storage |
| multimedia | mpeg, mov, mp3, mp4, avi | binary file containing audio or A/V information |

- tipos especiais de ficheiros: diretorias, links (atalhos), pipes com nome e sockets UNIX (canais IPC), acesso a periféricos (character-oriented devices, block-oriented devices; em `/dev`), comunicação com o kernel (em `/proc`)

# Atributos dos Ficheiros (1/1)

- os atributos podem variar entre SOs, mas tipicamente incluem
  - *nome* - designação do ficheiro em formato legível (*conveniência*)
  - *id* - código numérico usado pelo SO para designar o ficheiro (*eficiência*)
  - *tipo* - apenas em sistemas que suportam diferentes tipos de ficheiros
  - *localização* - dispositivo e localização do ficheiro no dispositivo
  - *tamanho* - dimensão do ficheiro (e, possivelmente, dimensão máxima)
  - *protecção* - acessos permitidos (leitura, escrita, execução, e por quem)
  - *marcas temporais* - criação, últimos acessos (sem e com modificação)
  - *propriedade (posse)* - utilizador dono e grupo(s) com acesso permitido
  - *codificação, checksums, ...* - atributos extendidos (alguns SOs/SFs)
- consulta dos atributos em ambientes GUI: *mouse-righ-clik* num ficheiro + *properties*; consulta dos atributos em UNIX/Linux: `stat /path/to/file`
- os atributos de um ficheiro são mantidos na estrutura da diretoria que o contém, ou num bloco separado de metadados apontado pela diretoria

# Operações sobre Ficheiros (1/2)

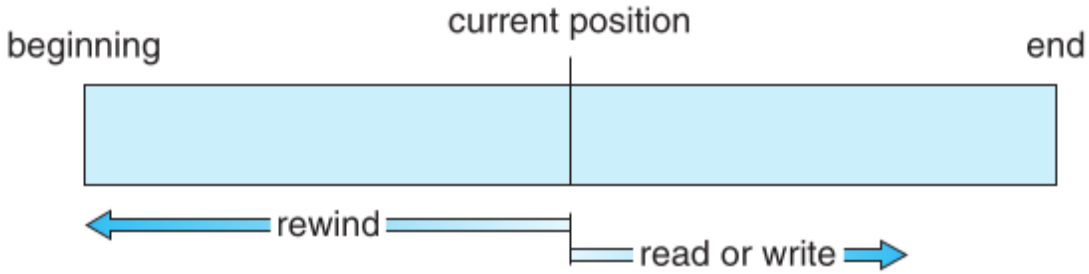
- qualquer SO tem de suportar um conjunto de **operações básicas sobre ficheiros**; o acesso a estas operações faz-se através de *syscalls* específicas
- **criar**: encontrar espaço para o ficheiro e criar um registo na sua diretoria
- **abrir**: encontrar o registo do ficheiro (com base no seu nome) e devolver uma *referência* (apontador) para ele (a usar pelo SO em próximas operações)
- **escrever**: usar a *referência* para sobrepor ou acrescentar dados, oriundos da memória; necessário saber a posição onde escrever, e poder alterar essa posição de forma arbitrária; e saber encontrar mais espaço se necessário
- **ler**: usar a *referência* para ler dados para a memória; necessário saber a posição de onde ler, e poder alterar essa posição de forma arbitrária
- **reposicionar**: poder alterar a posição de forma arbitária sem I/O envolvido; cada *referência* tem uma posição associada, partilhada pelas várias operações
- **remover**: remover o registo do ficheiro, da sua diretoria, e libertar o espaço consumido pelos seus dados; remoção efetiva após o último *link* ser removido
- **truncar**: libertar o espaço consumido pelos dados, sem remover o registo da diretoria; equivalente a remover e criar de novo o ficheiro (inicialmente vazio)

# Operações sobre Ficheiros (1/2)

- **outras operações são definidas à custa das operações elementares**; e.g, copiar = *criar* ficheiro novo vazio, *ler* conteúdo do antigo, *escrever* no novo
- outras operações: mover, criar *links* (atalhos), ler/modificar atributos, etc.
- a maioria das operações implica **encontrar a entrada do ficheiro na sua diretoria** (conceptualmente uma tabela), para aceder aos seus metadados (atributos) e aos seus dados (a localização destes é um desse atributos)
- a pesquisa repetida para cada operação, evita-se "abrindo" o ficheiro
  - a entrada na sua diretoria é copiada para uma tabela em memória (*)
  - operações subsequentes sobre o ficheiro utilizam a cópia em memória
  - essas operações identificam o ficheiro com base num índice da tabela
- quando o acesso ao ficheiro não é mais necessário, "fecha-se" o ficheiro
  - operação explícita (`close`), ou automática quando o processo acaba
  - escritas pendentes são realizadas, e liberta-se a entrada na tabela (*)
- UNIX: uma *tabela de descritores de ficheiros* (TDF) por processo, uma *tabela de ficheiros* (TF) global, uma *tabela de vnodes* (TV) global

# Métodos de Acesso (1/2)

- o acesso a um ficheiro pode ser feito de diferentes formas (padrões de acesso), dependendo do requisito da aplicação cliente e do suporte do SO
- em certos sistemas, o método de acesso é definido na criação do ficheiro
- *acesso sequencial*
  - dados acedidos por ordem, registo após registo
  - maioria das operações:
    - *read next*: ler próximo registo
    - *write next*: acrescentar novo registo no fim do ficheiro (*append*)
    - leitura e escrita avançam o *file pointer* automaticamente
  - *rewind*: reposicionar o *file pointer* no início
  - *skip +n*, *skip -n*: avançar/recuar *n* registos
  - padrão baseado no modelo de acesso a *fitas magnéticas* (*tapes*) (mas funciona perfeitamente em dispositivos de acesso aleatório (discos))



# Métodos de Acesso (2/2)

- *acesso direto* (ou *relativo*)
  - baseado no modelo de ficheiro assumido pelos *discos/SSDs*: ficheiro = sequência de registos/blocos numerados, todos da mesma dimensão
  - dados acessíveis aleatóriamente, com base no número do registo
  - o número do bloco é um parâmetro nas operações sobre ficheiros: *read n*, *write n*; alternativa: *position n*, *read next*, *write next*
  - necessário converter blocos lógicos (relativos) em blocos físicos (problema similar à conversão de páginas virtuais em frames físicas)

- outros métodos (assentes em acesso direto)
  - envolvem a construção e utilização de um *índice* do ficheiro
  - o índice pode ser desdobrado em vários níveis (similar à paginação);
  - pesquisa (sequencial, binária) do índice para localizar o bloco do disco contendo um certo registo, e acesso direto ao registo apontado

# 6.2 Conceitos sobre Diretorias

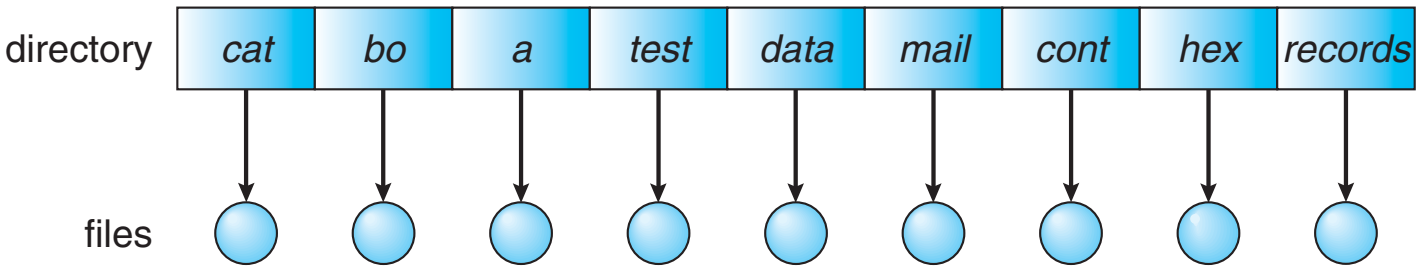
# Conceito e Operações

- **diretoria** = tabela que associa nomes de ficheiros aos seus atributos – mais precisamente, ao bloco do disco que os contém (File Control Block (FCB))
- esta "tabela" deve permitir a realização eficiente de várias operações que implicam a pesquisa das entradas dos ficheiros com base no seu nome
- **criar ficheiro**: implica criar/adicionar uma nova entrada na tabela (ou eventualmente reciclar uma entrada existente marcada como livre)
- **remover ficheiro**: eliminar (ou marcar como livre) a entrada na diretoria
- **listar diretoria**: listar os objetos (ficheiros ou sub-diretorias) e seus atributos
- **renomear ficheiro**: o nome do ficheiro é um atributo especial, mantido na sua diretoria (e não junto com os outros atributos, no FCB); renomear um ficheiro pode alterar a sua posição na tabela da diretoria, reorganizando-a
- **travessia do sistema de ficheiros**: visitar todas as diretorias e consultar os atributos de todos os ficheiros (buscar ficheiros, verificar consistência do SF)

# Funcionalidades Convenientes

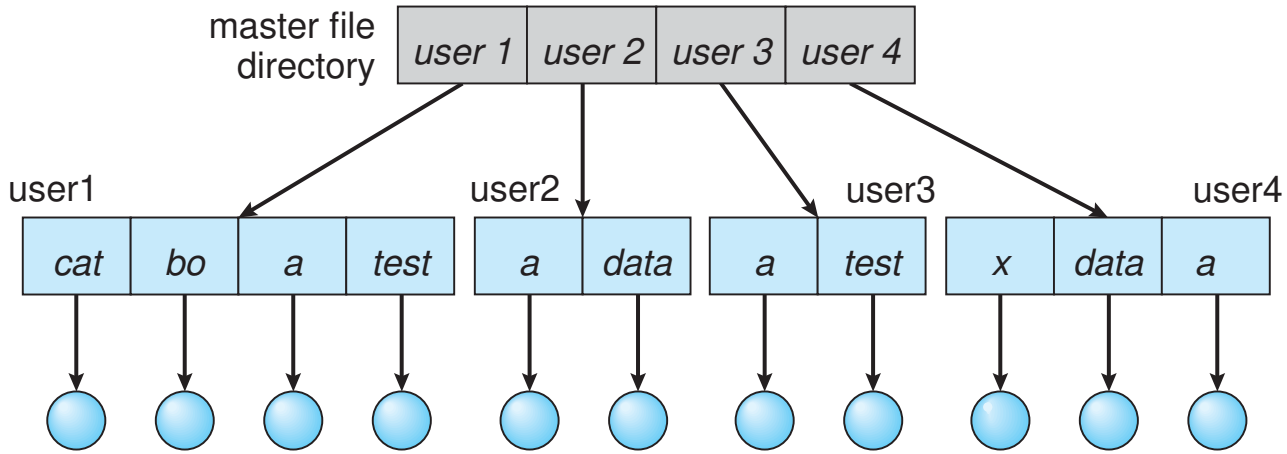
- para além da *eficiência* das operações anteriores, as diretorias (e o SF em geral) devem suportar funcionalidades *convenientes* ao utilizador:
  - poder atribuir nomes de forma flexível
    - mesmo nome usável em ficheiros diferentes (em sítios diferentes)
    - mesmo ficheiro com vários nomes (no mesmo sítio ou sítios diferentes)
    - extensão não-obrigatória no nome (e.g., `prog` em vez de `prog.exe`)
    - nomeação case-sensitive (fich $\neq$ FICH $\neq$ FiCh)
    - nomes compostos por caracteres especiais (espaços, etc.)
    - nomes grandes (Linux: até 255 caracteres; MSDOS: formato 8.3)
  - poder criar sub-diretorias, para *agrupamento* de objetos afins
- estas funcionalidades são suportadas de forma diferente, em diferentes estruturas adotada para as diretorias: 1 nível, 2 níveis, árvore, grafo

# Estrutura das Diretorias – 1 Nível



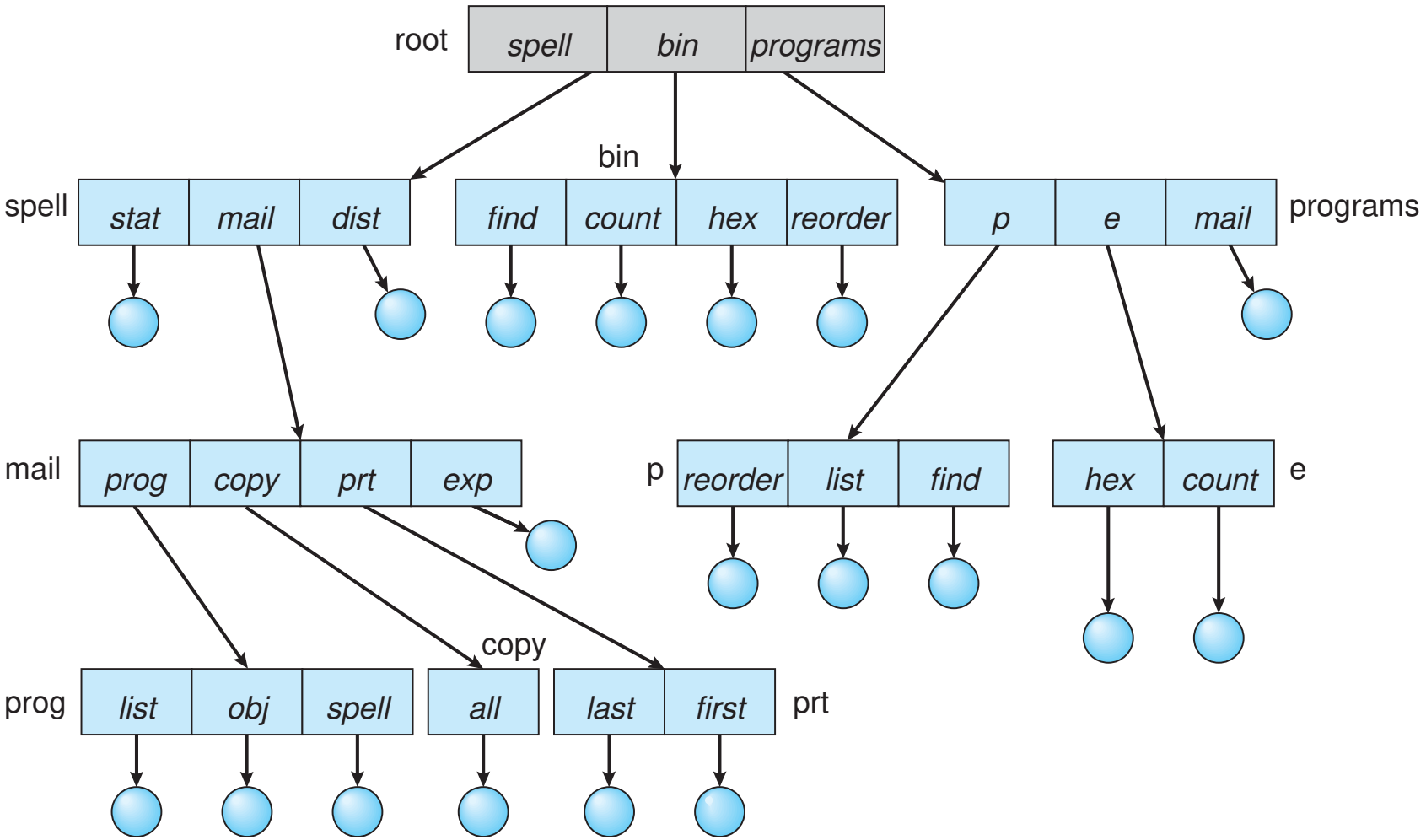

- uma só diretoria com todos os ficheiros
- inadequado para mais de um utilizador
- força a utilização de nomes diferentes
- não suporta agrupamento (subdiretorias)

# Estrutura das Diretorias – 2 Níveis




- uma diretoria para cada utilizador (e uma de sistema com utilitários)
- permite nomes iguais para ficheiros de utilizadores distintos
- agrupamento ainda limitado (não permite sub-diretorias de nível $>2$)
- caminho (*path*) para um objeto: `[/utilizador/]objeto`
- *search PATH*: diretorias de busca automática de executáveis
- pesquisa + eficiente: diretoria pessoal + pequena (menos ficheiros)
- ainda não suporta a partilha de ficheiros

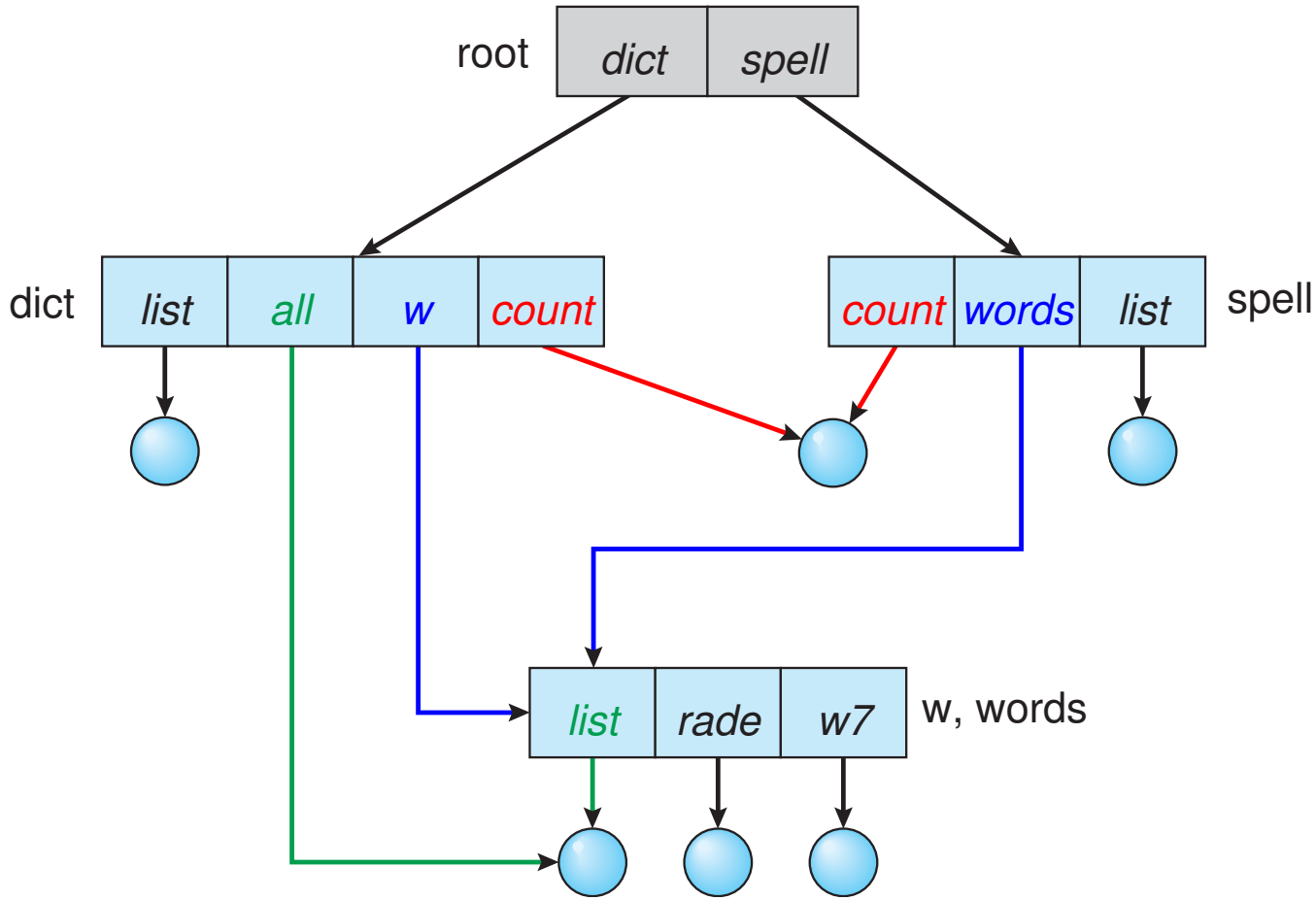
# Estrutura das Diretorias – Árvore com >2 Níveis



# Estrutura das Diretorias – Árvore com >2 Níveis

- generalização da estrutura de 2 níveis a n níveis
- conceito de *caminho absoluto* (/a/b/c) e *caminhos relativos* (../b/c)
- conceito de *diretoria corrente* / *diretoria de trabalho* ($PWD)
  (pois agora há várias possibilidades, além da diretoria pessoal)
  - os comandos que manipulam ficheiros assumem, por omissão, que as operações a realizar devem ser efetuadas na diretoria corrente
  - conceito de *home*: diretoria de trabalho *inicial* ($HOME)
- suporte natural do agrupamento (por sub–diretorias)
  - pesquisa de ficheiros mais eficiente: mais sub-diretorias, e mais pequenas, tornam mais rápida a pesquisa numa sub-diretoria
- ainda não suporta a partilha de ficheiros entre utilizadores, mas permite o acesso (especificando o caminho, ou mudando a diretoria corrente)
- alguns SFs suportam remoção recursiva de diretorias, mesmo não vazias
  - exemplo: comando `deltree` do MS-DOS e `rm -r` do UNIX

# Estrutura das Diretorias – Grafos Acíclicos

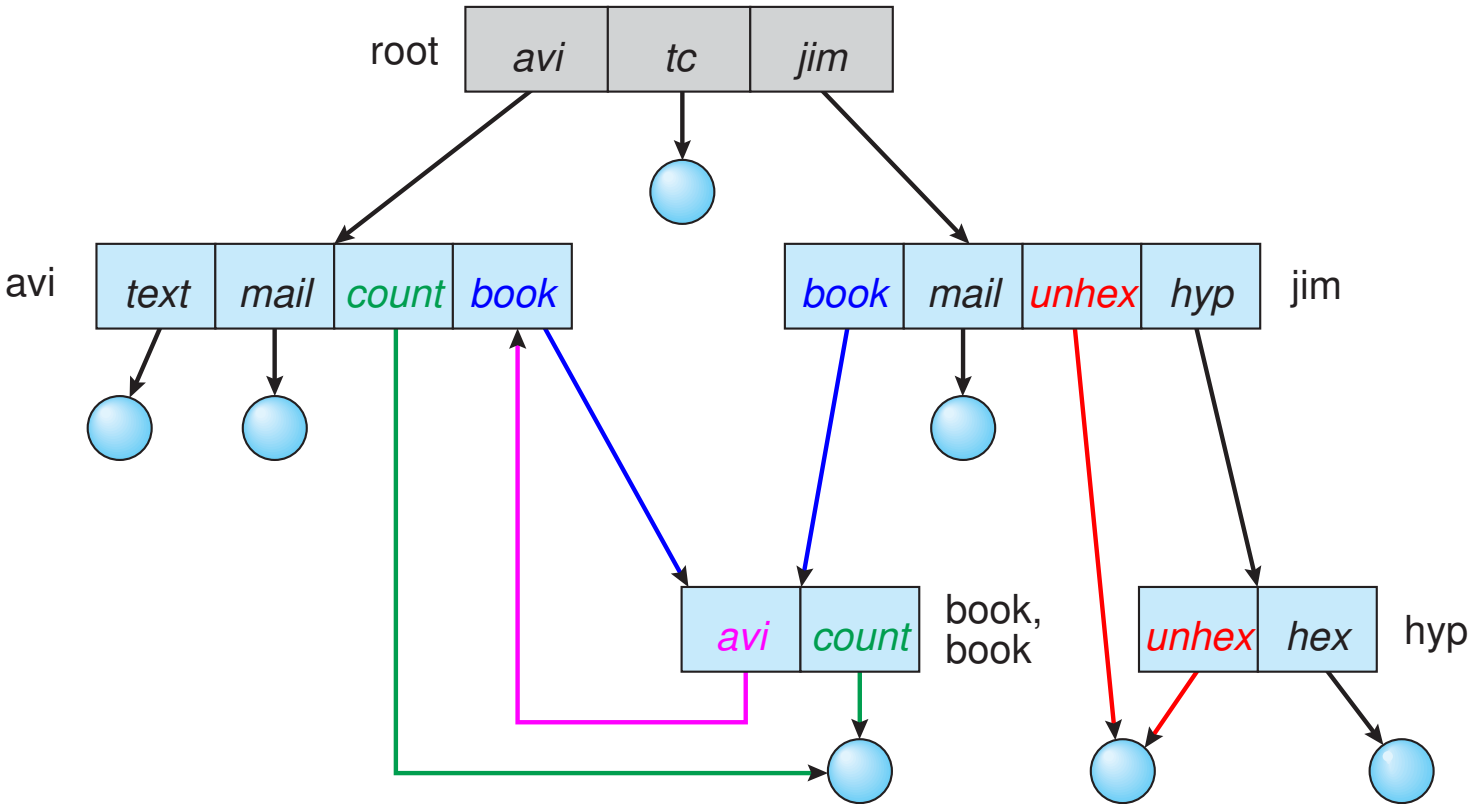



links:
/dict/count = /spell/count
/dict/w = /spell/words
/dict/all = /dict/w/list = /spell/words/list

# Estrutura das Diretorias – Grafos Acíclicos

- generalização da árvore, com suporte à partilha de ficheiros e diretorias
  - várias entradas (possivel/ em diretorias $\neq$s) são do mesmo objeto
  - possibilidade de *aliasing*: vários nomes para o mesmo objeto
  - fisicamente, existe uma só cópia do objeto (ficheiro ou diretoria)
  - exemplo: *links simbólicos* em UNIX/Linux (ln -s target linkname)
    - apontam para uma entrada alternativa, na mesma diretoria (ln -s a b $\Rightarrow$ ./b $\rightarrow$ ./a) ou noutra diretoria (ln -s /tmp/x y $\Rightarrow$ y $\rightarrow$ /tmp/x)
    - ignorados quando se efectua uma travessia do SF (para evitar ciclos)
- a remoção de objectos com aliases envolve cuidados especiais
  - libertar os aliases, mantendo o objecto original, não é problemático
  - libertar o objeto (e o espaço ocupado por este), e manter os aliases, deixa os aliases inválidos (e os aliases poderão vir a apontar para um objecto diferente quando o espaço libertado for reutilizado)
    - remover os aliases quando se remove o objecto (a entrada do objeto na sua diretoria tem a lista dos seus aliases; entrada de tamanho variável)
    - manter os aliases e gerar um erro quando se tentarem resolver (UNIX)
    - manter o objecto até que todas as referências para ele sejam eliminadas (necessita lista de aliases ou contador de referências - UNIX hard links)

# Estrutura das Diretorias – Grafos Genéricos




links:
/jim/unhex = /jim/hyp/unhex
/avi/book = /jim/book
/avi/book/count = /jim/book/count = /avi/count
/avi/book/avi = /avi (ciclo)

# Estrutura das Diretorias – Grafos Genéricos

- generalização dos grafos acíclicos, permitindo a existência de ciclos
- impacto dos ciclos no desempenho do sistema de ficheiros
  - uma pesquisa irá entrar em ciclo, quando envolve duas ou mais diretorias que se referenciam mutuamente (apontam uma para a outra)
  - solução 1: definir um limite para o número de (sub)diretorias visitadas (caindo-se num ciclo, o número de voltas será limitado, e não infinito)
  - solução 2: permitir *aliases* apenas para ficheiros, e não para diretorias
- impacto dos ciclos na remoção de objetos do sistema de ficheiros
  - um contador de referências pode ser $\neq$ zero e o objeto já não existir
    - exemplo: A é subdiretoria de PA (*); B é subdiretoria de A; dentro de B é criado um link para A (**); o contador de referência de A é 2 ((*)+(**)); remover em PA a entrada de A diminui o contador de referências de A, de 2 para 1 ((*) desaparece, (**) mantém-se); mas A é inacessível (deixou de existir), pois PA deixou de apontar para ela
  - solução 1: aplicação periódica de *garbage collection*
    - primeira travessia do SF, marcando todos os objectos acessíveis
    - nova travessia, marcando como livre o espaço dos objectos inacessíveis
  - solução2: não permitir ciclos quando se criarem aliases (UNIX)

# 6.3 Estrutura do Sistema de Ficheiros

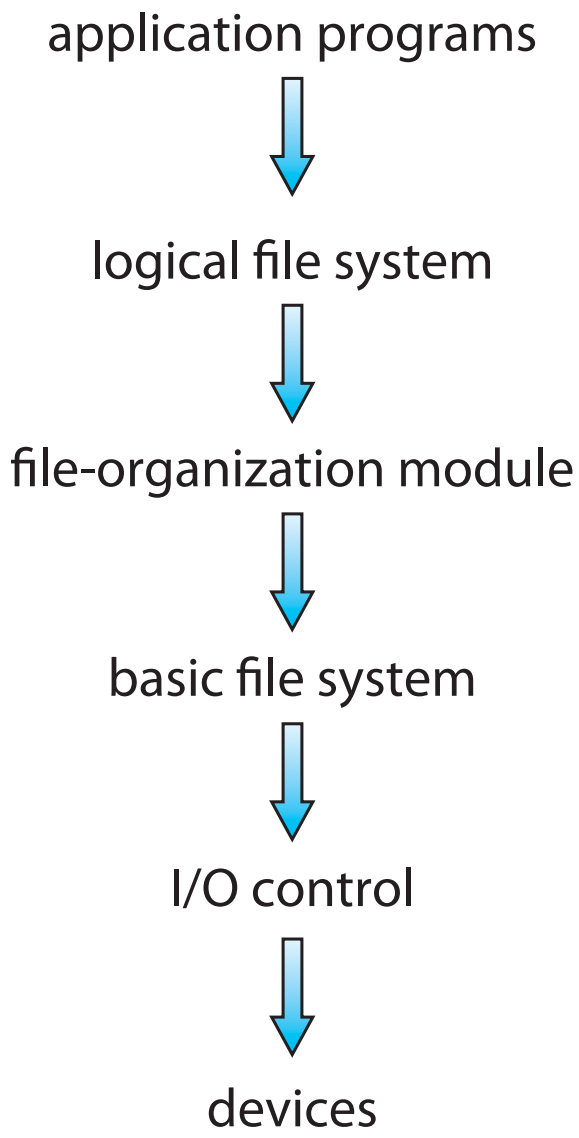
# Estrutura do Sistema de Ficheiros (1/3): Conceitos Básicos

- **sistema de ficheiros** (SF)
  - conjunto de algoritmos e estruturas de dados, usados pelo SO, para gerir o armazenamento e o acesso a dados em memória secundária
- dispositivos de **memória secundária**: discos rígidos e SSDs
  - oferecem um conjunto de blocos de tamanho fixo (512/4096 bytes)
  - cada **bloco físico** acessível diretamente (acesso aleatório) por uma coordenada única (coordenada linear (LBA) ou geométrica (CHS))
  - vários blocos são lidos/escritos de uma só vez da/para uma cache
- os **ficheiros** podem-se ver como uma **sequência de blocos lógicos** (bloco de metadados, e $\geq 1$ blocos de dados), de tamanho igual aos blocos físicos
- desenhar um sistema de ficheiros implica (com eficiência e conveniência):
  - definir o mapa da visão lógica dos ficheiros na visão física dos discos
  - definir as abstrações e operações oferecidas aos utilizadores (ficheiro, atributos, operações, métodos de acesso, estrutura das diretorias, ...)
- cada SO usam preferencialmente certos SFs: FAT e NTFS em Windows; HFS e APFS em MacOS; EXT* e XFS em Linux; UFS/ZFS em BSD; ...

# Estrutura do Sistema de Ficheiros (2/3): Organização em Camadas

**Camadas do Sistema de Ficheiros**:

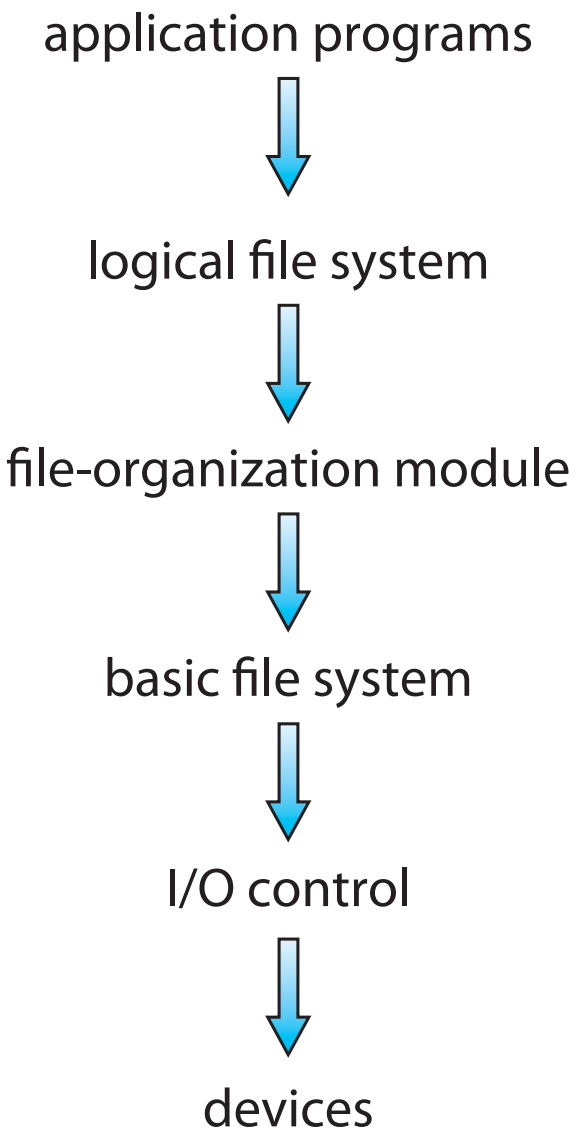
- **Programas**: acedem aos serviços do **SF lógico** via *syscalls*, otimizadas pela manutenção em *cache* de cópias dos metadados (vnodes), e utilização de várias tabelas (por processo ou globais – ver TDF e TFG)
- **Sistema de Ficheiros Lógico**: lida com *metadados*; **a)** gere a estrutura de *diretorias*, que mapeiam nomes de ficheiros em Blocos de Controlo de Ficheiros (FCBs) (contêm os *atributos* dos ficheiros, como propriedade, permissões e localização no disco dos seus blocos); **b)** suporta os mecanismos de segurança e proteção do SF
- **Módulo de Organização dos Ficheiros**: **a)** mapeia os blocos *lógicos* de um ficheiro em blocos *físicos lineares* (de coordenadas LBA) do disco, usando um *método de alocação*; **b)** faz a gestão dos blocos físicos livres



# Estrutura do Sistema de Ficheiros (3/3): Organização em Camadas

**Camadas do Sistemas de Ficheiros (cont.)**:

- **Sistema de Ficheiros Básico** (block IO subsystem): **a)** escalona os pedidos E/S; **b)** emite ao *device driver* do disco, comandos genéricos baseados em coordenadas LBA ("ler/escrever bloco 1000"); **c)** gere *buffers* e *caches* em RAM, com blocos do disco ((meta)dados)
- **Controlo E/S** (*device drivers* e *interrupt handlers*): **a)** transfere blocos entre a RAM e o disco; **b)** *device driver* traduz comandos do SF Básico em comandos de baixo nível do controlador do disco (padrões de bits/bytes escritos na memória do controlador)
- estes dois módulos (Controlo E/S e SF Básico) podem ser re-utilizados por diferentes SFs, sendo cada SF responsável por implementar os seus módulos específicos de Organização dos Ficheiros e SF Lógico (slide anterior)

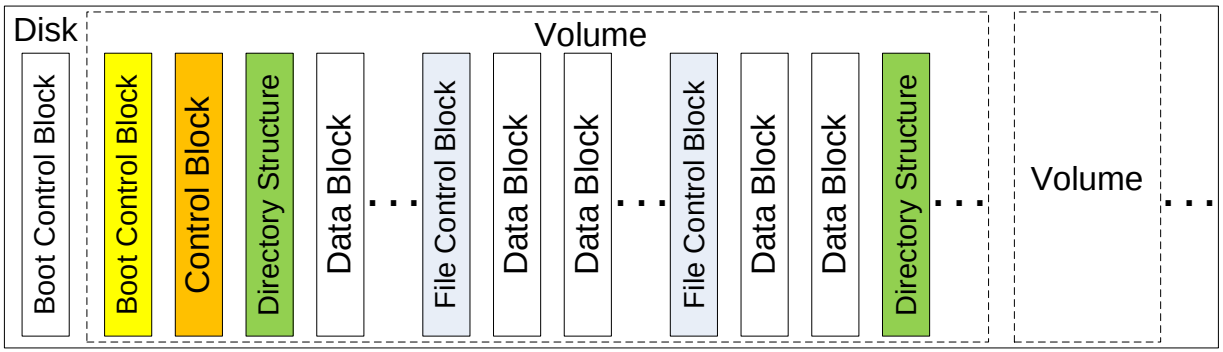


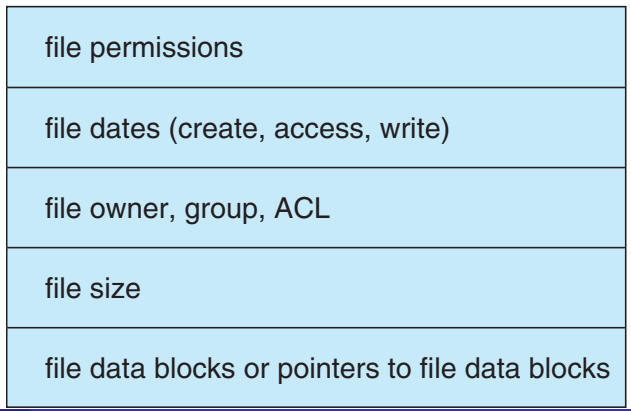
# 6.4 Implementação do Sistema de Ficheiros

# Implementação do SF (1/11): Estruturas de Dados

## Estruturas de Dados do SF no Disco – visão genérica

- **disk boot control block** (por disco): indica quais os volumes em que se divide o disco e qual tem o SO de arranque (volume ativo); MBR - *master boot record*
- **volume boot control block** (por volume): informação necessária para arrancar um SO instalado no volume; UFS - *boot block* ; NTFS - *partition boot sector*
- **volume control block** (por volume): informação sobre o volume/partição, como a) nº total de blocos, b) dimensão dos blocos, c) nº de blocos livres (e referências para eles), d) nº de FCBs livres (e referências para eles); UFS - *superblock*; NTFS - info. preservada no ficheiro *Master File Table*
- **estrutura de diretoria** (por diretoria): associa os nomes dos ficheiros e das sub-diretorias de uma diretoria, aos seus FCBs e estruturas; UFS - *directory i-node*
- **blocos de controlo dos ficheiros** (FCBs): um por cada ficheiro, com os seus atributos; UFS - *i-node*; NTFS - linha de uma BD relacional incluída no MFT

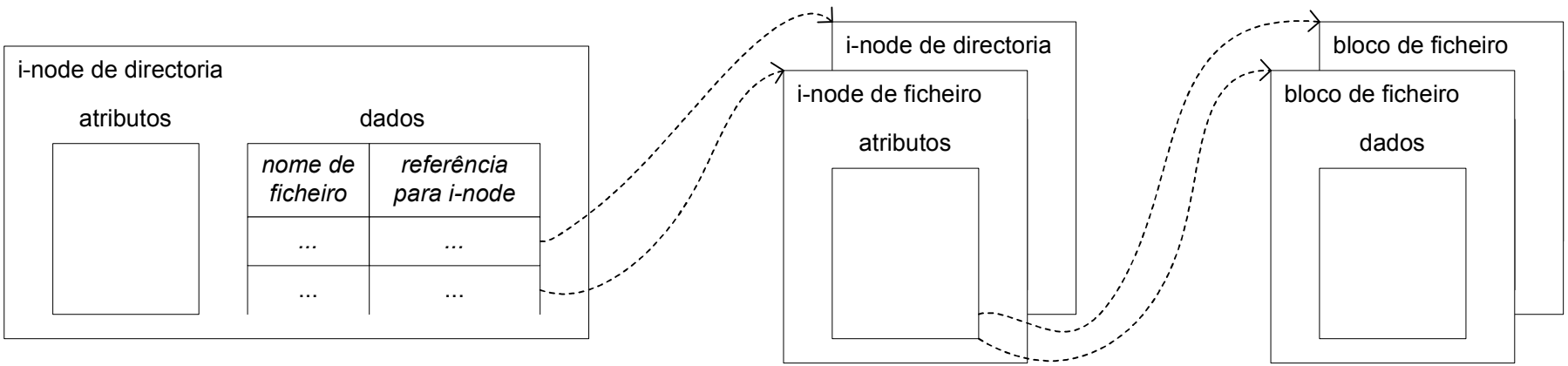


# Implementação do SF (2/11): Estruturas de Dados

## Estruturas de Dados do SF no Disco – UNIX/Linux

- estrutura e relações dos i-nodes



# Implementação do SF (3/11): Estruturas de Dados

## Estruturas de Dados do SF no Disco – UNIX/Linux

- **exemplo: blocos de controlo dos ficheiros em Linux (i-nodes)**
  - mostrar a coordenada LBA (índice linear físico) do i-node de um ficheiro

```
$ echo "hello" > myfile; ls -i myfile
39855407 file
```

  - mostrar os atributos guardados no i-node de um ficheiro

```
$ stat myfile
Size: 6               Blocks: 8          IO Block: 4096   regular file
Device: fd02h/64770d        Inode: 39855407    Links: 1
Access: (0644/-rw-r--r--)  Uid: (1000/rufino)   Gid: (1000/rufino)
Access: 2019-11-28 13:05:39.326295937 +0000
Modify: 2019-11-28 13:21:31.260255422 +0000
Change: 2019-11-28 13:21:31.260255422 +0000
```

  - mostrar os índices dos blocos de dados de um ficheiro

```
# hdparm --fibmap myfile
filesystem blocksize 4096, begins at LBA 0; assuming 512 byte sectors.
byte_offset  begin_LBA    end_LBA      sectors
0           1275716792    1275716799    8
```

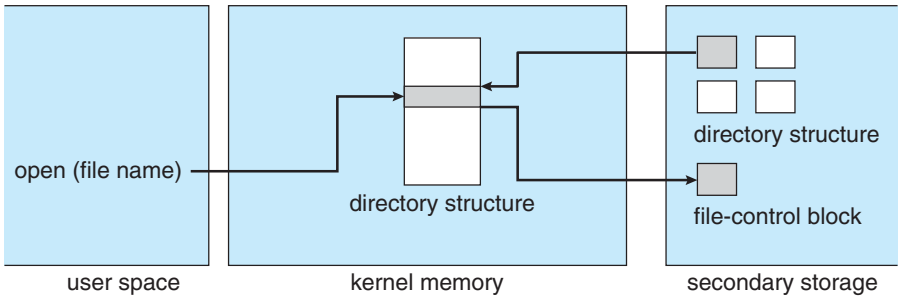
  - mostrar as extensões (grupos de 8 blocos de dados) de um ficheiro

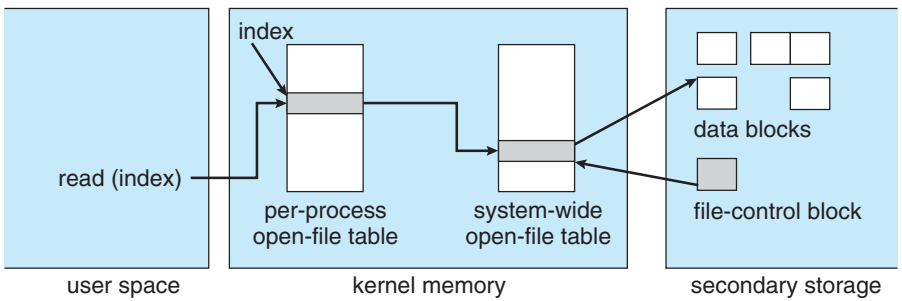
```
# debugfs -R "stat <39855407>" /dev/sda
EXTENTS:
(0):159464599
```

# Implementação do SF (4/11): Estruturas de Dados

## Estruturas de Dados do SF na Memória Principal – visão genérica

- **tabela de volumes montados**: uma só tabela, global; contém uma cópia dos *volume control blocks* dos volumes que, num dado instante, estão montados (i.e., associados a uma pasta (/) ou letra (C:) e portanto acessíveis ao SO)
- **cache de estruturas de diretorias**: uma só cache, global; contém cópias das *estruturas de diretorias* para as diretorias que estão a ser acedidas
- **cache de blocos de dados dos ficheiros**: uma só cache, global; contém cópias dos *blocos de dados* dos ficheiros que estão a ser acedidos
- **tabela global de ficheiros abertos** (TGFA): uma só tabela, global; contém uma cópia do **FCB** de cada ficheiro aberto no sistema, bem como outros atributos
- **tabela individual de ficheiros abertos** (TIFA): uma tabela por processo; tem referências para entradas na TGFA, para cada ficheiro aberto por um processo



# Implementação do SF (5/11): Estruturas de Dados

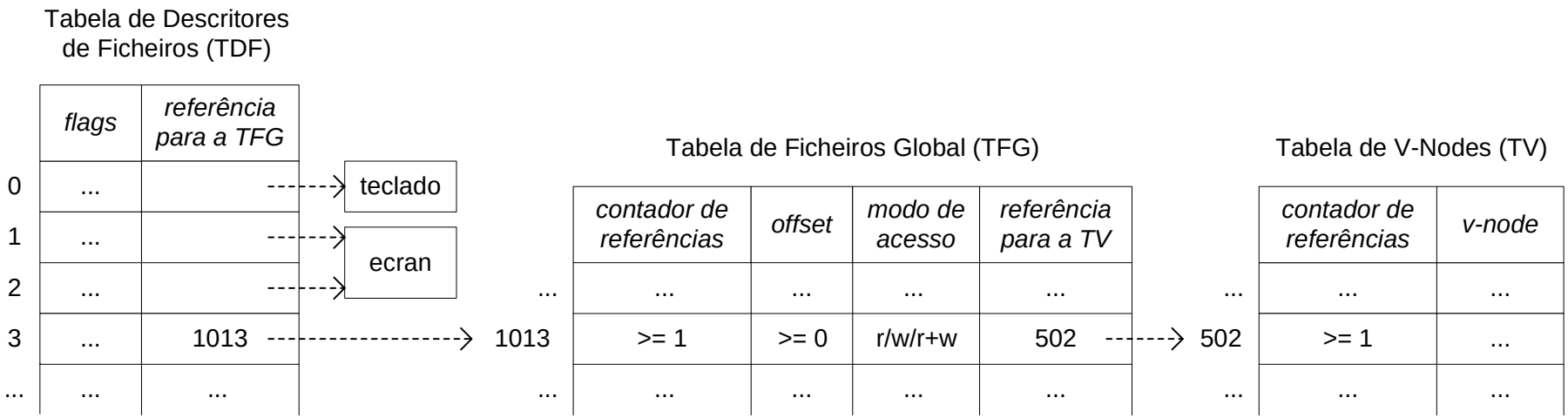
## Estruturas de Dados do SF na Memória Principal – UNIX/Linux

- as estruturas de diretorias, e os blocos de controlo de ficheiros, ocupam no disco blocos do mesmo tipo, designados por **i-nodes**
- **tabela de v-nodes (TV)**: uma só tabela, global; contém cópias (**v-nodes**) de i-nodes lidos do disco; cada i-node é copiado uma só vez para a TV sendo o v-node partilhado por todos os processos que acedem à diretoria/ficheiro respetiva(o)
- **tabela de ficheiros global (TFG)**: uma só tabela, global; por cada ficheiro aberto/criado por um processo, a TFG ganha uma entrada com atributos como *offset* e modo de acesso, e com uma referência para o v-node do ficheiro na TV
- **tabela de descritores de ficheiros (TDF)**: uma tabela por processo; os índices das entradas da TDF chama-se descritores; uma ou mais entradas da TDF referenciam entradas na TFG (em resultado de abertura/criação explícita de ficheiros, duplicação de entradas da TDF já ocupadas, ou herança de uma cópia da TDF do processo pai); vários descritores podem referenciar a mesma entrada na TFG
- cada entrada da TFG, e cada v-node da TV, têm um contador de referências; quando esse contador desce a 0, a entrada/v-node é libertado

# Implementação do SF (6/11): Estruturas de Dados

## Estruturas de Dados do SF na Memória Principal – UNIX/Linux

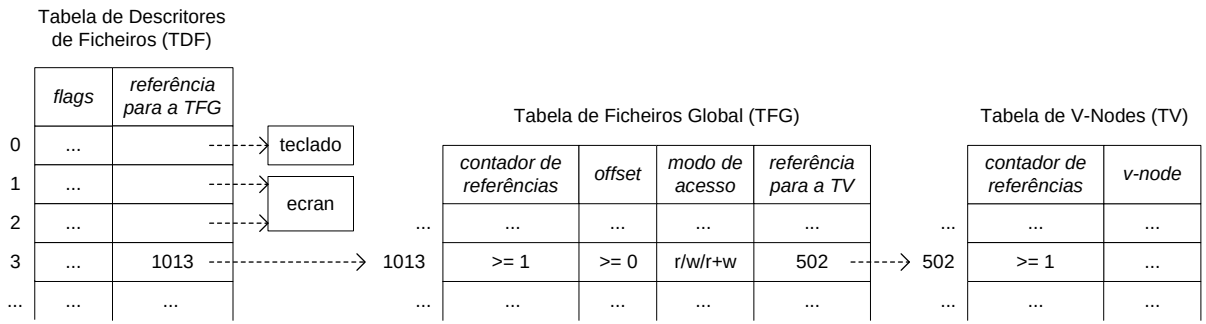
- estrutura das tabelas TDF, TFG e TV, e suas relações



# Implementação do SF (7/11): Estruturas de Dados

**Acesso a um Ficheiro "f" de uma Diretoria "d" em UNIX/Linux:**

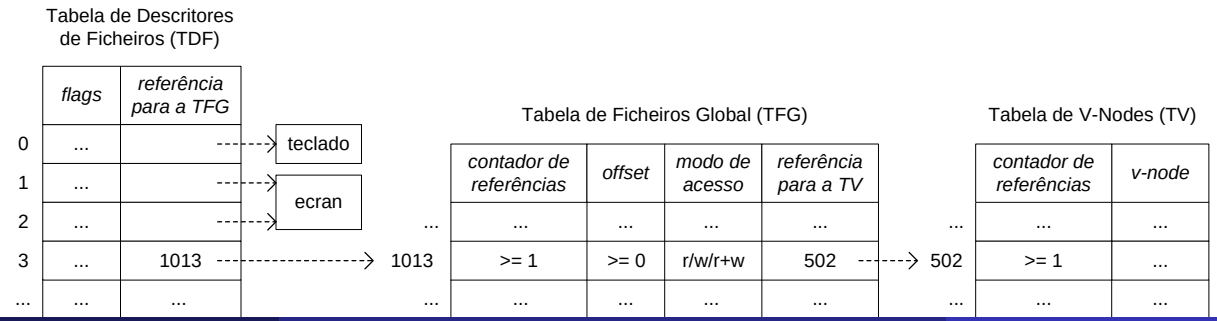
- através da primitiva `open`, o programa invoca o SF lógico, fornecendo "d\f"
- o SF lógico procura uma cópia do i-node da diretoria "d" em memória, na TV; se não a encontrar, lê o i-node do disco; para saber o bloco do disco que contém o i-node de "d", terá de consultar o v-node da diretoria ao qual "d" pertence
- com o i-node de "d" em memória, busca-se nele a entrada relativa a "f"; essa entrada terá uma referência (coordenada LBA) para o i-node de "f" no disco
- antes de ler o i-node de "f" do disco, procura-se (com base na sua coordenada LBA) na TV; se não se encontrar, o i-node de "f" é lido do disco para a TV
- com o i-node de "f" em memória, confronta-se a posse (dono e grupo) e as permissões (leitura, escrita e execução) de "f", com as do processo que lhe quer aceder; se forem compatíveis, a abertura continua; senão, termina com um erro



# Implementação do SF (8/11): Estruturas de Dados

**Acesso a um Ficheiro "f" de uma Diretoria "d" em UNIX/Linux (cont.):**

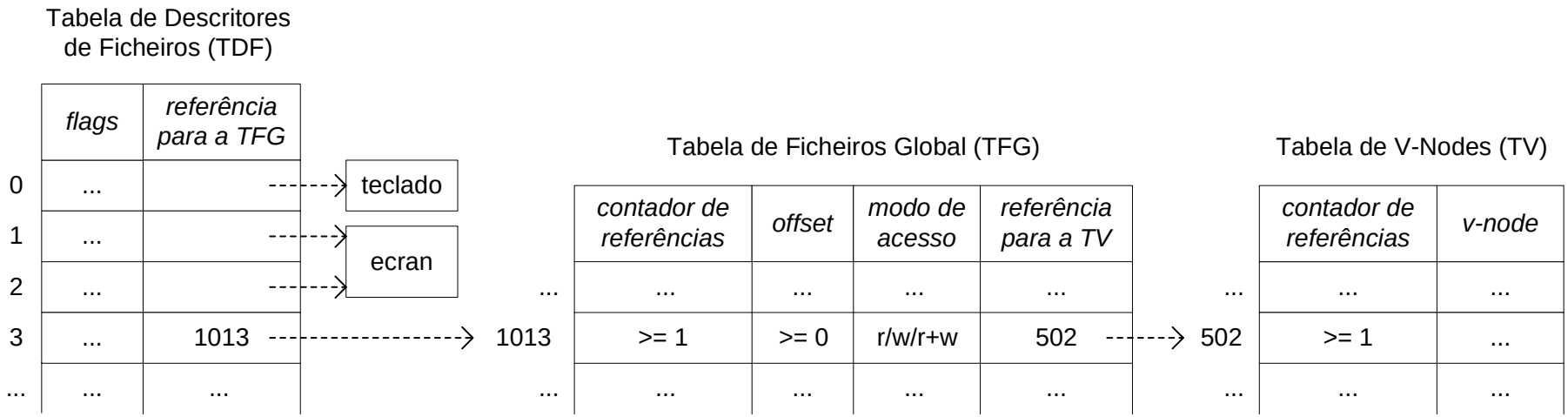
- reserva-se uma entrada livre na TFG e nela adiciona-se uma referência para o v-node de "f", sendo o contador de referências do v-node incrementado 1 unidade; dependendo do modo de abertura (normal ou append), o atributo offset da entrada na TFG é definido a 0 ou com o tamanho atual (em bytes) do ficheiro
- na TDF do processo que invocou `open`, é reservada a primeira entrada livre, e nesta entrada é colocada uma referência para a entrada reservada na TFG, sendo o contador de referências desta última incrementado 1 unidade; a abertura conclui pela devolução do índice (descritor) da TDF que foi usado, ao invocador
- à medida que se aceder (`read` ou `write`) ao ficheiro, será verificado se os blocos de dados pretendidos estão já em memória, ou se é preciso lê-los do disco; aceder a uma certa posição **o** (offset) do ficheiro implica determinar a que bloco lógico **n** a posição **o** pertence; sabendo **n** será depois preciso deduzir o bloco físico **m** respetivo, o que depende do método de alocação usado; sabido **m**, irá procurar-se uma cópia desse bloco em memória, sendo lido do disco se não se encontrar



# Implementação do SF (9/11): Estruturas de Dados

**Acesso a um Ficheiro "f" de uma Diretoria "d" em UNIX/Linux (cont.):**

- quando o processo fechar (`close`) o descritor na TDF, este fica livre e o contador de referências da entrada apontada na TFG é decrementado 1 unidade; se descer para 0, o contador de referências da entrada apontada na TV é decrementado 1 unidade; se um contador de referências descer a 0, a entrada respetiva é libertada
- quando o acesso é de escrita, o fecho do ficheiro garante a sincronização explícita dos blocos de dados, e do bloco de meta-dados (v-node) que foram alterados em memória, com as cópias originais no disco; notar que o SO despoletará também, periodicamente, essa sincronização, para garantir a robustez do Sist. de Ficheiros



# Implementação do SF (10/11): Diretorias

- em termos abstratos, uma **estrutura de diretoria** ou, mais simplesmente, uma **diretoria**, é uma tabela com duas colunas, dadas pelo par <nome de ficheiro, apontador para FCB> (para cada nome de ficheiro, indica qual o seu FCB)
- em termos concretos, essa tabela será armazenada em 1 ou mais blocos do disco
- o número máximo admissível desses blocos, juntamente com a dimensão de cada par, determina o número máximo de ficheiros de qualquer diretoria do SF

- na prática, implementar uma diretoria corresponde a resolver o problema: como mapear, de forma eficiente, uma tabela (que é uma sequência de pares) numa estrutura unidimensional (os blocos do disco, que são sequências de bytes)
- a forma mais simples de implementar a sequência de pares é um array de pares guardado no(s) bloco(s) do disco que suportam a diretoria; nesta abordagem, remover um par cria um "buraco" no array que pode ser preenchido pelo último par, e adicionar um par pode ser feito por acrescento no final do array

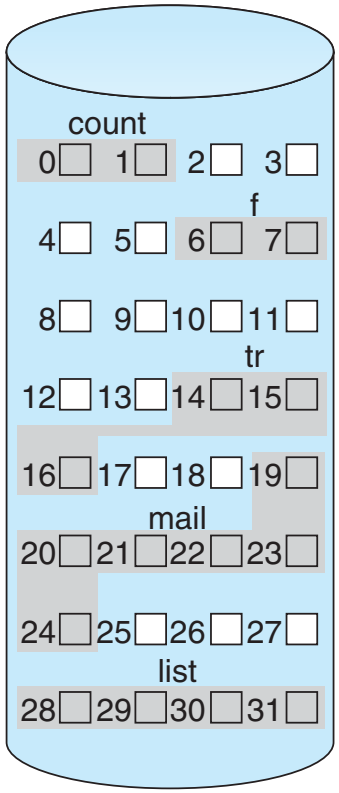
# Implementação do SF (11/11): Diretorias

- se os pares não forem mantidos ordenados, a pesquisa na diretoria será ineficiente; por outro lado a ordenação implica necessariamente uma sobrecarga temporal
- hipóteses de ordenação dos pares:
  - a) **ordenação sequencial** (permite pesquisa binária, mas obriga a mover parte do array para o manter ordenado; acaba por ser incomportável);
  - b) **ordenação com lista ligada embebida** (acrescenta a cada par um campo extra apontador; grande flexibilidade; pesquisa sequencial);
  - c) **ordenação com árvore binária embebida** (acrescenta a cada par três apontadores; grande flexibilidade; pesquisa binária)
- em qualquer caso, os pares marcados como livres após remoção do ficheiro respetivo, são organizados numa lista ligada, agilizando a sua reutilização

- alternativa à ordenação: os pares são mantidos num array não ordenado; uma **função de hash** associa o nome de um ficheiro a uma célula do array destinada ao par do ficheiro; a descoberta dessa célula é imediata; o problema é quando há *colisões* (a função de hash mapeia dois ficheiros diferentes na mesma célula ...)

# 6.5 Métodos de Alocação

# Métodos de Alocação (1/10): Alocação Contígua




directory

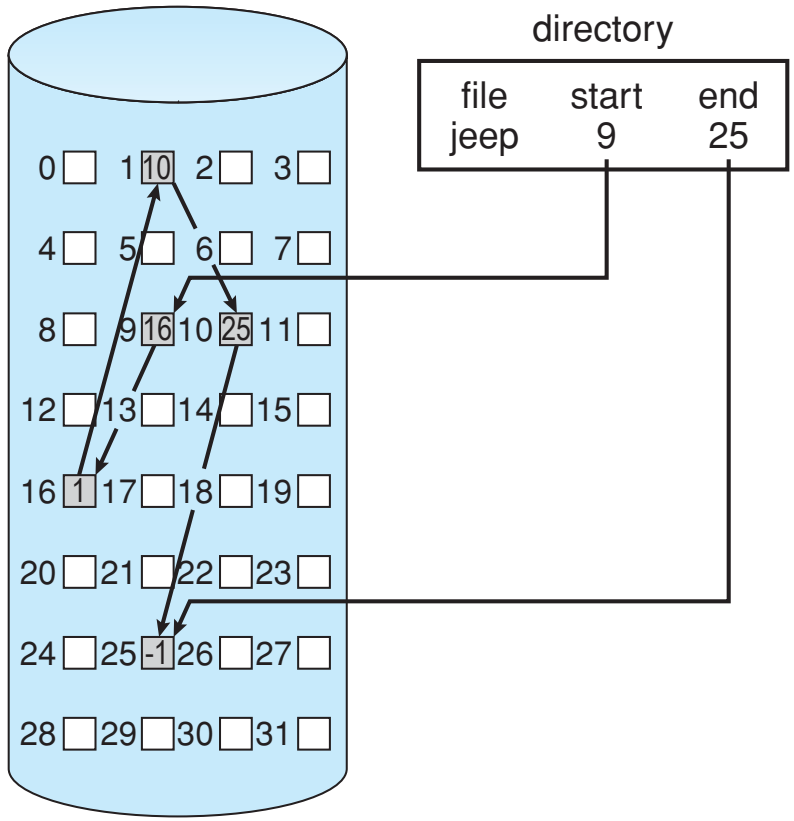
| file | start | length |
|---|---|---|
| count | 0 | 2 |
| tr | 14 | 3 |
| mail | 19 | 6 |
| list | 28 | 4 |
| f | 6 | 2 |

- cada ficheiro ocupa um conjunto de blocos contíguos no disco
- *desempenho*: acesso a bloco contíguos consome apenas um *disk seek*
- *simplicidade*: suficiente conhecer o 1º bloco físico ($b'$) e o nº total de blocos ($N$)

# Métodos de Alocação (2/10): Alocação Contígua (cont.)

- acesso sequencial:
  - recordar o último bloco físico acedido e aceder ao seguinte
- acesso aleatório:
  - aceder ao bloco lógico $b + i \Rightarrow$ ler o $i$'ésimo bloco físico a partir de $b'$
- encontrar espaço para um ficheiro com $N$ blocos lógicos
  - implica encontrar pelo menos $N$ blocos físicos livres adjacentes
  - usam-se estratégias semelhantes às da alocação contígua de RAM
    - *first-fit*: usar a 1ª sequência suficiente ($\geq$ N blocos) encontrada
    - *best-fit*: usar a sequência suficiente que minimiza o desperdício (nº de blocos livres adjacentes, posteriores ao bloco N do ficheiro)
    - *worst-fit*: usar a sequência suficiente que maximiza o desperdício
  - questões de fragmentação semelhantes às da alocação contígua de RAM
    - fragmentação interna: há espaço não usado no último bloco
    - fragmentação externa: demasiada dispersão dos blocos livres, que previne a sua alocação a ficheiros; corrigível com *compactação*
  - crescimento: alocar *extensões* (pequenas sequências de blocos livres)
  - pré–alocação: viável sabendo à partida o tamanho máximo do ficheiro

# Métodos de Alocação (3/10): Alocação Ligada



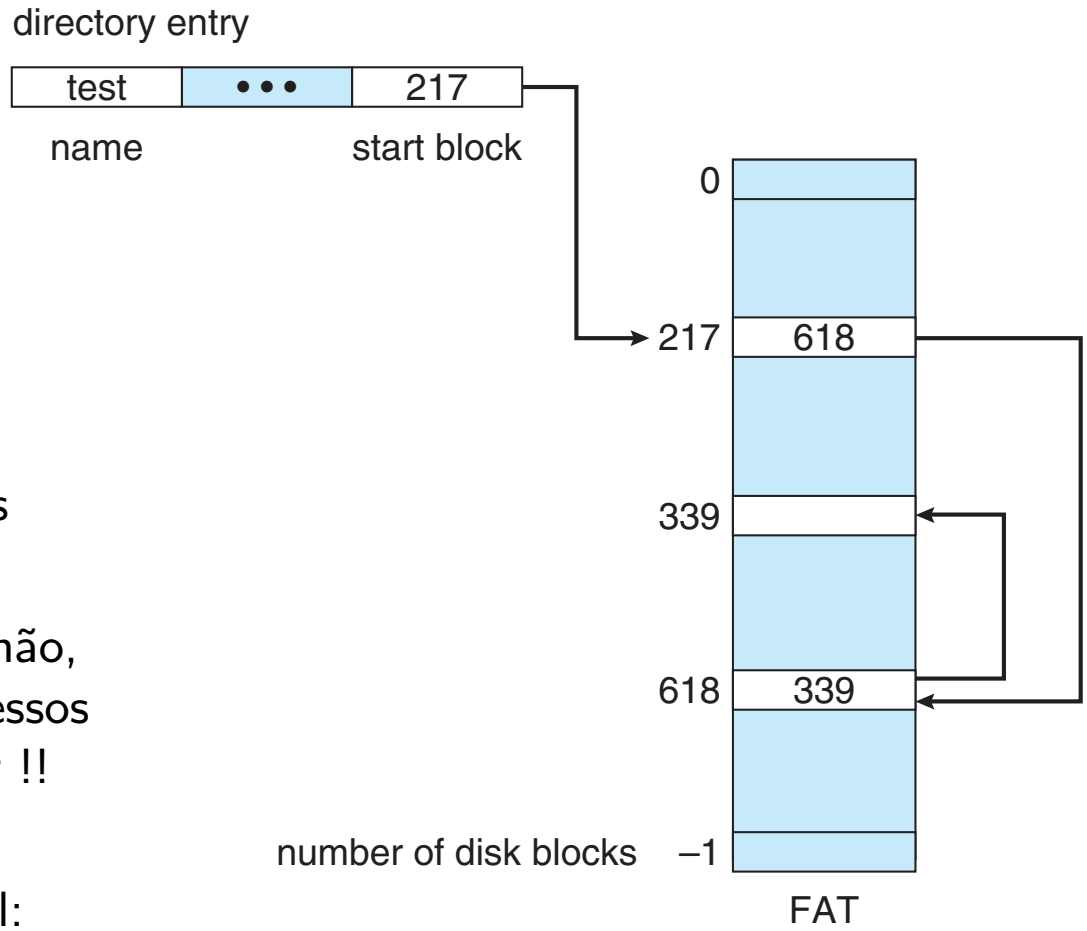

- cada ficheiro é uma lista ligada de blocos (possivelmente dispersos) em disco
- bloco = dados + apontador ($\Rightarrow$ um ficheiro consome mais espaço)
- *simplicidade*: suficiente conhecer o bloco físico inicial e final

# Métodos de Alocação (4/10): Alocação Ligada (cont.)

- o ficheiro pode crescer dinamicamente, enquanto houver blocos livres
- evita fragmentação externa (qualquer bloco livre satisfaz um pedido: first-fit)
- inadequado a acesso aleatório: percorrer a lista $\Rightarrow$ vários acessos ao disco
- lista ligada de *clusters* (grupo de vários blocos adjacentes)
  - $>$ desempenho do acesso
  - $<$ sobrecarga em apontadores
  - fragmentação interna no último *cluster*
- $<$ fiabilidade: listas ligadas são mais vulneráveis a corrupção; soluções:
  - listas duplamente ligadas
  - guardar nome do ficheiro e posição relativa em cada bloco
    $\Rightarrow$ $+$ sobrecarga

# Métodos de Alocação (5/10): Alocação Ligada – FAT

- FAT = *file allocation table*
- tabela no início do volume, com uma entrada por bloco (é um *volume control block*)
- o tamanho da FAT depende do número de blocos do volume
- a tabela contém listas embebidas
- a tabela é operada em *cache*; senão, seriam precisos pelo menos 2 acessos ao disco por cada bloco a aceder !!
- face à alocação ligada tradicional:
  - acesso aleatório melhorado
  - gestão do espaço livre facilitado



# Métodos de Alocação (6/10): Alocação Ligada – FAT (cont.)

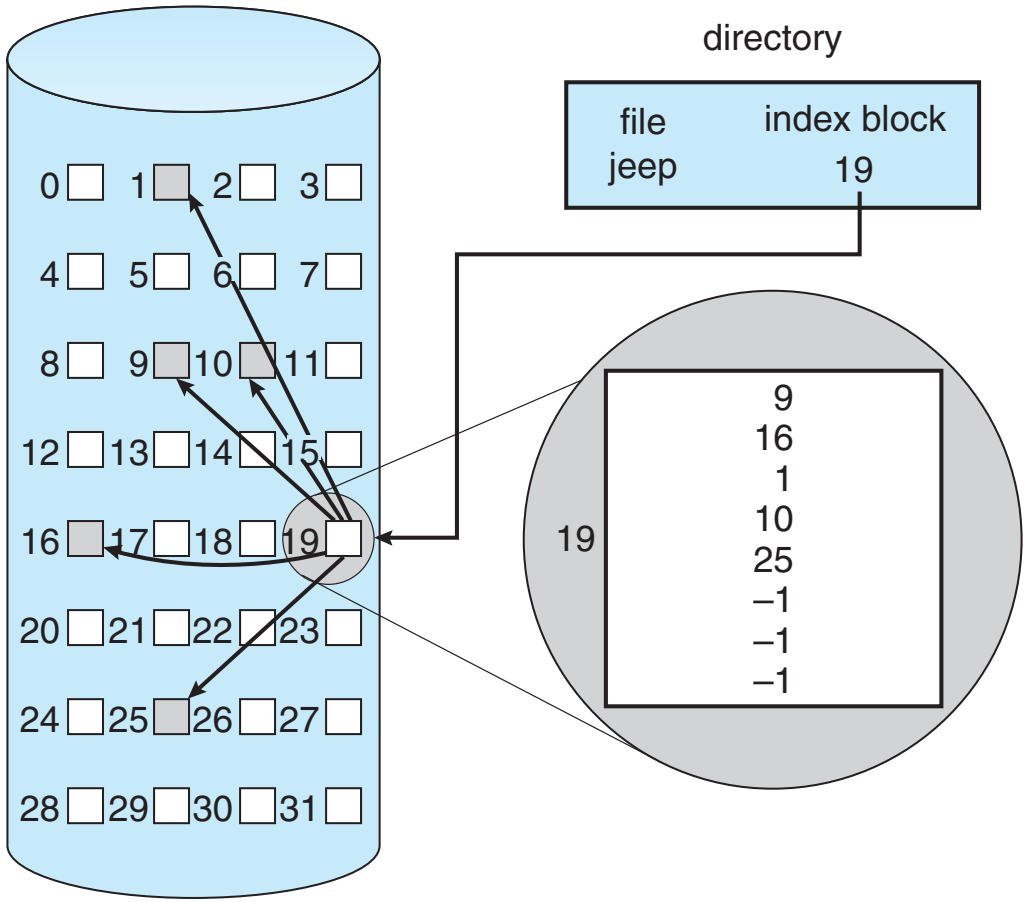
**exemplo de dimensionamento**:

- #BYTES(disco) = 1 Gbyte = $1 * 2^{30}$ bytes
- #BYTES(bloco) = 4 Kbytes = $4 * 2^{10}$ bytes
- #BYTES(entrada da FAT) = 32 bits = 4 bytes

---

- #BLOCOS(disco) = #BYTES(disco)/#BYTES(bloco) =
  $= (1 * 2^{30})/(4 * 2^{10}) = 2^{18} = 256$ Kblocos
- #ENTRADAS(FAT) = #BLOCOS(disco) = 256 Kentradas
- #BYTES(FAT) = #BYTES(entrada da FAT) * #ENTRADAS(FAT) =
  $= (4 * 256K)$ bytes = 1024 Kbytes = 1 Mbyte = 1 * $2^{20}$ bytes
- #BLOCOS(FAT) = #BYTES(FAT)/#BYTES(bloco) =
  $= (1 * 2^{20})/(4 * 2^{10}) = 2^{8} = 256$ blocos
- #BLOCOS(úteis) = #BLOCOS(disco) – #BLOCOS(FAT) =
  = (256 K - 256) blocos
- max(#BYTES(ficheiro)) = #BLOCOS(úteis) * #BYTES(bloco) =
  = (256 K - 256) * 4 Kbytes

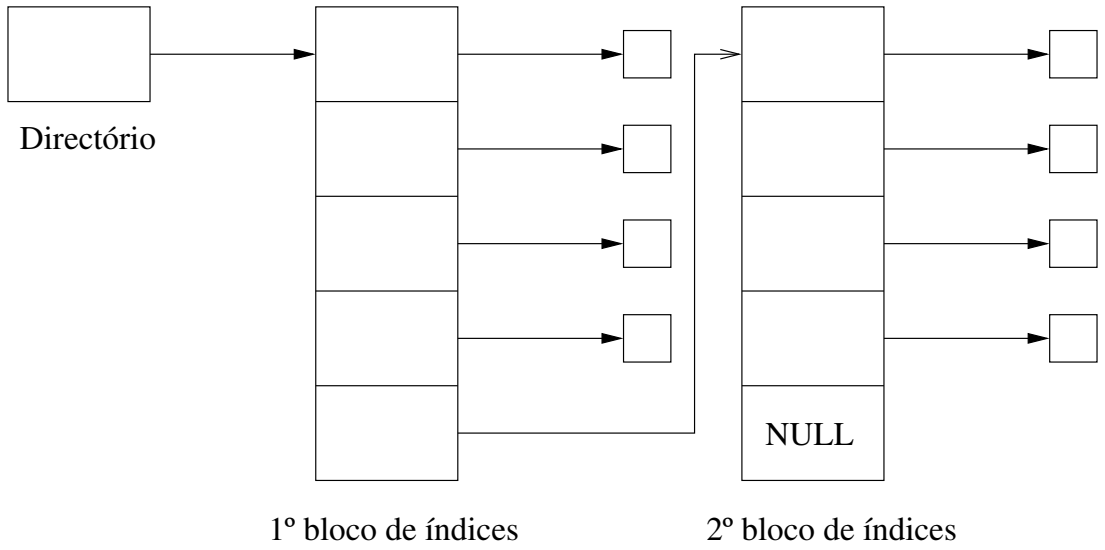
# Métodos de Alocação (7/10): Alocação Indexada




- cada entrada da diretoria aponta para um *bloco de índices*
- o *bloco de índices* concentra todos os apontadores para os blocos do ficheiro
- acesso direto ao i'ésimo bloco via i'ésimo apontador do *bloco de índices*

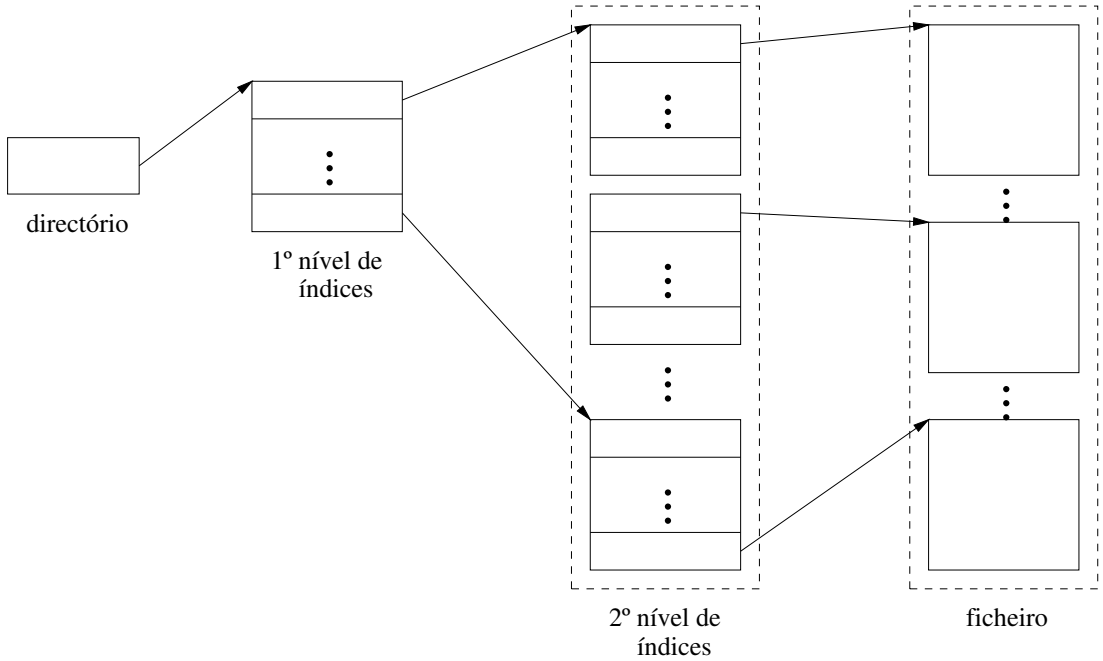
# Métodos de Alocação (8/10): Alocação Indexada (cont.)

- evita fragmentação externa (qualquer bloco livre satisfaz um pedido)
- $>$ sobrecarga com apontadores (*bloco de índices* parcialmente consumido)
  $\Rightarrow$ necessidade de definir cuidadosamente a dimensão do *bloco de índices*
- *desempenho*
  - é conveniente fazer *caching* do *bloco de índices*
  - demasiada dispersão dos blocos dos ficheiros prejudicará o desempenho
- suporte ao crescimento: *esquema ligado*



# Métodos de Alocação (9/10): Alocação Indexada (cont.)

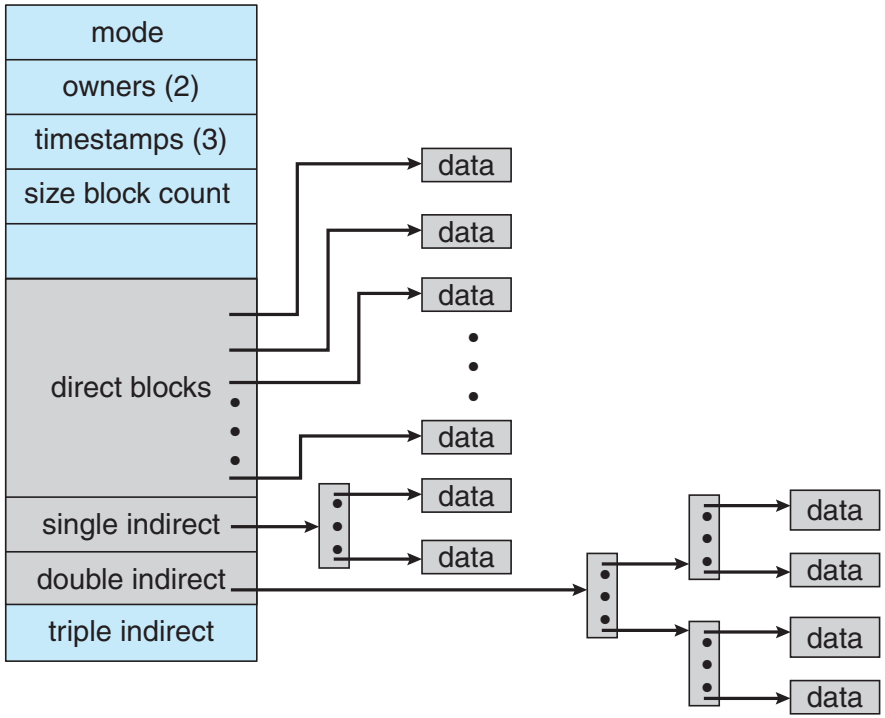
- suporte ao crescimento: *esquema multinível*




- **exemplo**: blocos de 4096 bytes (4 Kbytes), apontadores de 32 bits (4 bytes)
  - cada bloco suporta 4K / 4 = 1024 apontadores
  - com dois níveis de índices, têm-se 1024 * 1024 = $2^{20}$ apontadores
  - manter todos os índices em *cache* exigiria $(1 + 1024) * 4$ Kbytes $\approx$ 4 Mbytes
  - $2^{20}$ apontadores permitem referenciar $2^{20}$ blocos diferentes
  - o tamanho máximo de um ficheiro será $2^{20} * 4$ Kbytes = 4 Gbytes

# Métodos de Alocação (10/10): Alocação Indexada (cont.)

- suporte ao crescimento: *esquema combinado*



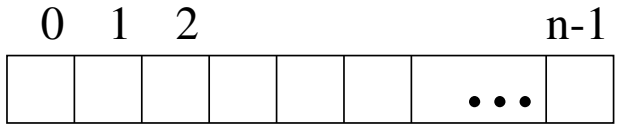

- **exemplo**: blocos de 4096 bytes (4 Kbytes), apontadores de 32 bits (4 bytes)
  - com 0 níveis de índices, há 12 apontadores (e ficheiros até 48 Kbytes)
  - com 1 nível de índices, há $2^{10}$ apontadores (e ficheiros até 4 Mbytes + 48 Kbytes)
  - com 2 níveis de índices, há $2^{20}$ apontadores (e ficheiros até 4 Gbytes + ... )
  - com 3 níveis de índices, há $2^{30}$ apontadores (e ficheiros até 4 Tbytes + ...)
  - máxima dimensão de um ficheiro: 48 Kbytes + 4 Mbytes + 4 Gbytes + 4 Tbytes

# 6.6 Gestão do Espaço Livre

# Gestão do Espaço Livre (1/5) : Vetor de Bits

- vetor de $n$ bits reflecte o estado (livre/ocupado) dos $n$ blocos do disco/partição:

$$vetor[i] = \begin{cases} 1 \Rightarrow \textit{bloco i está livre} \\ 0 \Rightarrow \textit{bloco i está ocupado} \end{cases}$$




- índice do 1º bloco livre:
  (nº de bits/palavra) x (nº de palavras a zero) + (deslocamento do 1º bit 1) - 1

- **exemplo** de pesquisa: $vetor = |00000000|00000000|00101111|...$
  índice do 1º bloco livre $= (8 \times 2) + 3 - 1 = 18$

- beneficia de instruções máquina adequadas (e.g., "deslocamento do 1º bit 1")

- $>$ eficiência exige o vetor de bits em RAM ($\Rightarrow$ sincronização periódica c/ disco)

- facilidade em obter ficheiros contíguos (encontrar sequências suficientes de 1s)

- a utilização de *clusters* permite vetores mais pequenos

# Gestão do Espaço Livre (2/5) : Vector de Bits (cont.)

**exemplo de dimensionamento**:

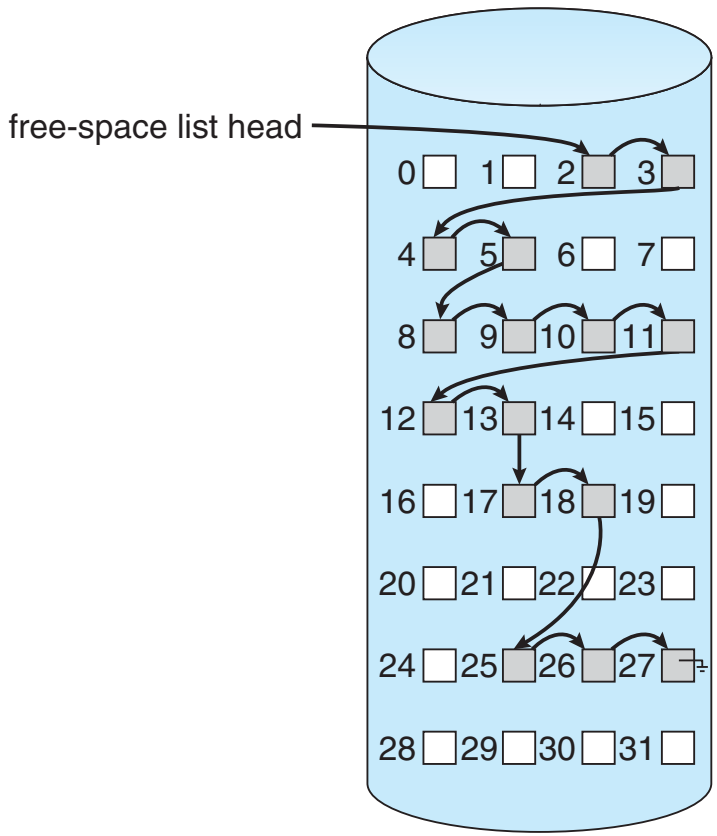
---

- #BYTES(disco) = 1 Gbyte = $2^{30}$ bytes
- #BYTES(bloco) = 4 Kbytes = $4 * 2^{10} = 2^{12}$ bytes

---

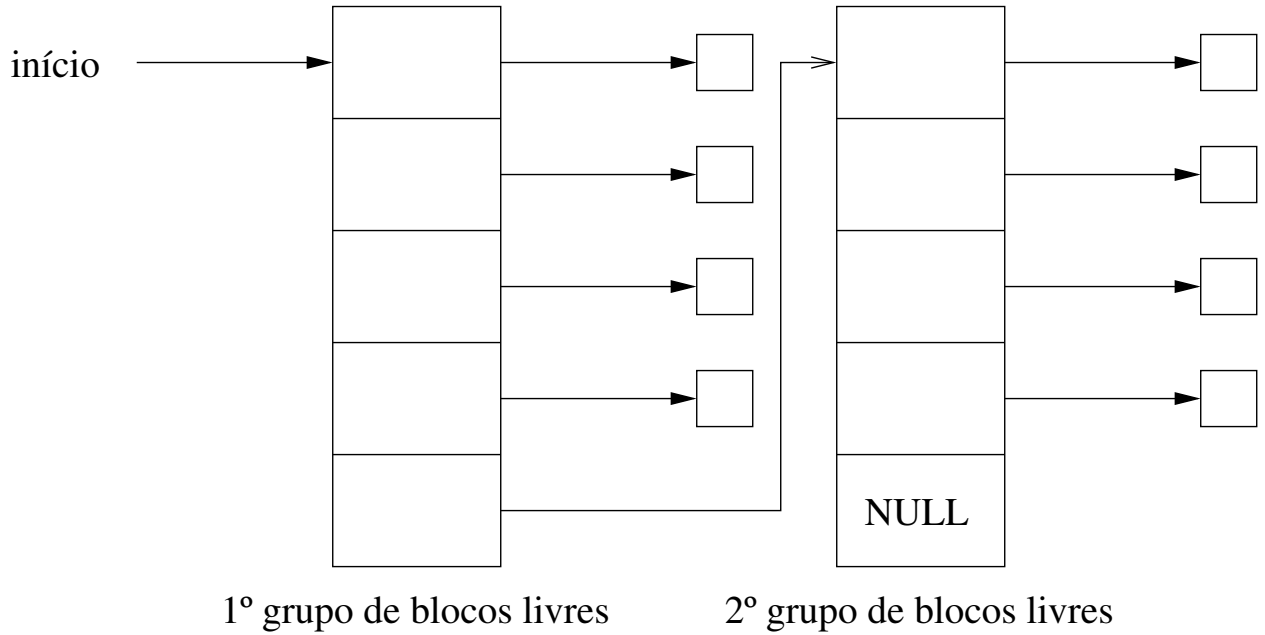
- #BLOCOS(disco) = #BYTES(disco)/#BYTES(bloco) =
  $= 2^{30}/2^{12} = 2^{18} = 256$ Kblocos
- #BITS(vetor) = #BLOCOS(disco) = 256 Kbits
- #BYTES(vetor) = #BITS(vetor)/8 =
  $= 2^{18}/2^{3} = 2^{15} = 32$ Kbytes
- #BLOCOS(vetor) = #BYTES(vetor)/#BYTES(bloco) =
  $= 2^{15}/2^{12} = 2^{3} = 8$ blocos

# Gestão do Espaço Livre (3/5) : Lista Ligada



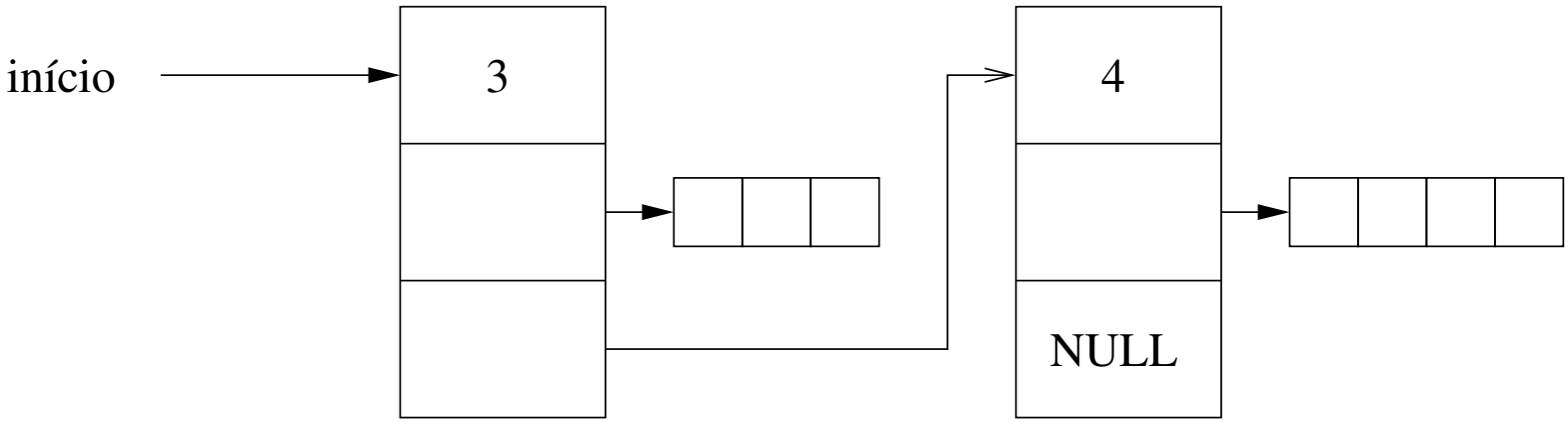

- o primeiro bloco da lista satisfaz um pedido unitário
- alocar blocos contíguos ⇒ pesquisar referências contíguas (pouco eficiente)
- numa FAT, uma lista ligada embebida une todas as entradas livres

# Gestão do Espaço Livre (4/5) : Agrupamento




- variante da lista ligada
- o 1º bloco livre armazena os índices de outros $n$ blocos livres
- desses $n$ blocos, apenas $n-1$ estão livres: o último guarda mais $n$ índices
- melhor desempenho (+ fácil obter, rapidamente, um grupo de blocos livres)

# Gestão do Espaço Livre (5/5) : Contagem




- é comum alocar/libertar vários blocos de uma só vez, talvez contíguos
- em vez de uma lista ligada de índices (endereços) de blocos livres ...
- ... cada nó da lista guarda um contador e o índice do 1º bloco da sequência
- esta lista ligada tende a ser menor que uma lista de todos os blocos livres

# 6.7 Eficiência e Desempenho

# Eficiência e Desempenho do Sistema de Ficheiros (1/3)

- dependente dos métodos de alocação e da implementação de diretorias
- influência dos métodos de alocação
  - UNIX: pré–alocação de *inodes* dispersos no disco ⇒ algum desperdício mas menor tempo de busca (blocos de dados próximos do inode)
  - BSD: variação da dimensão de um *cluster* para reduzir frag. interna
- influência dos dados mantidos nas entradas das diretorias ou nos *inodes*
  - manter o registo do último acesso (leitura e escrita, em separado) ⇒ ler o *inode* para memória, atualizar e gravar em disco (pouco eficiente)
  - o tamanho dos apontadores afeta não só o espaço consumido por eles, como ainda a dimensão máxima (o nº máximo de blocos) do ficheiro
    - **exemplo**: o IBM PC XT tinha 10 Mbytes de disco, sob MS-DOS com FAT de 12 bits (FAT12) e *clusters* de 8Kbytes; partição máxima de 32 Mbytes; à medida que as capacidades dos discos aumentavam, as partições continuavam limitadas a 32 Mbytes; posteriormente, introduziram-se apontadores de 16 bits (FAT16) e 32 bits (FAT32)

# Eficiência e Desempenho do Sistema de Ficheiros (2/3)

- pistas inteiras são lidas para a *cache do controlador*, "eliminando" a *latência rotacional*; os sectores pretendidos são seleccionados e entregues ao SO

- em geral, os SOs mantêm em RAM uma *cache* do disco, gerida sob LRU
  - nalguns casos, essa *cache* é dedicada apenas ao SF (*buffer cache*)
  - noutros, usa-se uma *page cache*, gerida pelo subsistema de Memória Virtual
  - o tipo de acesso ao ficheiro pode ter influência nos algoritmos de gestão da *cache*; por exemplo, quando o acesso é sequencial, aplicam-se as técnicas
    - *free–behind*: remover um bloco da cache mal o próximo seja lido
    - *read–ahead*: ler um bloco e os subsequentes, por antecipação

- escritas *síncronas* ou *assíncronas* exibem um desempenho bastante diferente
  - exemplo1: invocar `open` com a flag `O_SYNC` implica escritas síncronas
  - exemplo2: "montar" um SF com a opção `sync` (ignorada por alguns SFs)
  - com escritas síncronas, maximiza-se a consistência do SF, à custa de uma degradação do desempenho, pois não se tira partido da cache do SF

# Eficiência e Desempenho do Sistema de Ficheiros (3/3)

- as escritas são normal/ percebidas como mais rápidas que as leituras
  - as escritas são feitas, em 1a instância, na cache do SF; quando oportuno, o SO sincroniza a cache do SF com o disco, procurando minimizar os deslocamentos das cabeças do disco (*disk-head seeks*)
  - portanto, sob o ponto de vista do utilizador, os acessos de escrita ocorrem de forma assíncrona (i.e., são vistos como "instantâneos")
  - nas leituras, é mais provável ter que ir ao disco: uma primeira escrita de um bloco pode ser feita na cache do SF, ao passo que uma primeira leitura de um bloco tem de ser feita, obrigatoriamente, a partir do disco
  - donde, apesar de uma parte dos blocos poder ser lida por antecipação, a perceção é a de que a leitura é, em geral, mais síncrona que a escrita

- **RAM-disk**
  - zona da memória gerida como se de um disco se tratasse
  - transparente, de acesso muito rápido, sem persistência
  - adequado ao armazenamento de ficheiros temporários
  - diferente da *cache* do SF: o conteúdo da *cache* é controlado apenas pelo SO; o do RAM-disk é determinado pelas ações do utilizador

# 6.8 Recuperação e Backup

# Recuperação do Sistema de Ficheiros (1/2)

- cópias de várias estruturas (volume control blocks, estruturas de diretorias, FCBs, etc.) são mantidas em RAM, durante a operação normal do sistema
- falhas de corrente ou um *crash* do sistema podem impedir a sincronização da cópia em RAM dessas estruturas, com a sua instância principal em disco
- **verificação da consistência do SF**
  - no arranque do sistema, ou durante a operação do sistema (tipicamente apenas para volumes não "montados")
  - ferramentas: `fsck` (UNIX/Linux), `scandisk` (Windows)
  - técnicas:
    **a)** no início da modificação de metadados, é ativado um bit de estado, que só é desativado no fim da modificação; no arranque, a deteção de bits de estado ainda ativos desencadeia a verificação de consistência
    **b)** um FCB é atualizado antes dos respetivos blocos de dados

# Recuperação do Sistema de Ficheiros (2/2)

- a verificação de consistência poder ser insuficiente para reparar o SF (por vezes, é necessária intervenção humana, elevando o *downtime*)

- SFs **journaling / log-based transaction-oriented**
  - todas as alterações a realizar ao SF são registadas num *journal* / *log*
    - *transação*: conjunto indivisível de operações inter-relacionadas
  - as primitivas do SF podem retornar, mal o *journal* tenha sido atualizado
    - ou seja, a *transação* é considerada realizada (*commited*)
  - entretanto, as alterações são efetivamente propagadas através do SF
  - à medida que uma transação se desenrola, vai-se registando a sua progressão
  - quando uma transação termina com sucesso, é removida do *journal*
  - quando o sistema reinicia após falha de corrente ou crash
    - se o *journal* tiver transações por acabar, mas que foram já assumidas como realizadas (*commited*), então essas transações são finalizadas
    - se o *journal* tiver transações por acabar, mas que *não* foram assumidas como realizadas (*commited*), há que reverter quaisquer eventuais alterações que essas transações tenham feito ao SF

# Backup do Sistema de Ficheiros (1/2)

- **backup** (verbo):
  processo de **salvaguarda** de dados (pela criação de 1 ou + cópias de segurança), para permitir a sua **recuperação** (**restore**) na perda ou corrupção dos originais
- **backup** (nome): uma cópia de um conjunto de dados, ditos originais
- **razões para o desaparecimento dos dados originais**: desastres naturais, falhas de hardware, bugs de software, software malicioso, ação humana ((não-)intencional)
- **tipos de backups**
  - **total**: inclui uma cópia de todos os ficheiros selecionados para backup, independentemente de terem sido modificados ou não desde o último backup
  - **incremental**: apenas os ficheiros modificados após o último backup realizado
  - **diferencial**: inclui todas as modificações desde o último backup total
- **outras questões**
  - backups isolados/esporádicos versus regulares/pré–programados
  - reutilização dos média limitada pelo seu tempo de vida útil
  - acondicionamento seguro dos média
  - múltiplas cópias dos média (redundância)

# Backup do Sistema de Ficheiros (2/2)

**tipos de backups** (exemplo):

| | | | | |
|---|---|---|---|---|
| | A | A | **A'** | A' |
| | B | **B'** | B' | B' |
| | C | C | C | **C'** |
| | D | **D'** | D' | D' |
| | E | E | E | E |
| total | A,B,C,D,E | | | |
| incremental | | B',D' | A' | C' |
| diferencial | | B',D' | A',B',D' | A',B',D',C' |

# 6.9 Tópicos Suplementares

# Implementação do SF (1/3): Partições e Volumes

- um **sistema de computação** pode ter um ou vários **discos**
- um **disco** pode dividir-se em zero, uma ou várias **partições**
- **partições**: secções da área de armazenamento de um disco
  - **tabela de partições** de um disco: lista as partições desse disco
  - podem ser **raw** (sem nenhum SF) ou **cooked** (com um SF)
  - exemplos de partições raw: partições de *boot*, partições de *swap*, partições usadas por certas Bases de Dados (e.g., ORACLE), partições de suporte a RAID (e.g., com bitmaps RAID 1)
  - as designações **cooked** e **formatada** são equivalentes ...
- **volume**: área de armazenamento mais flexível que a partição
  - pode abarcar uma ou várias partições, de um ou mais discos
  - mas, na maioria dos casos, partição e volume confundem-se

# Implementação do SF (2/3): Arranque e Montagem

- **disco de arranque**: definido, na BIOS, como o disco a consultar para determinar o sistema operativo a executar (que pode estar nesse disco, ou noutro disco ...)

- **disk boot sector(s)** / sector(es) de arranque do disco:
  - sector zero (LBA = 0) e, eventualmente, mais alguns consecutivos
  - guarda(m) a tabela de partições do disco (em formato MBR ou GPT)
  - guardam o código do 1o estágio do arranque (1st stage boot-loader)

- **1st stage boot loader**: código alojado no(s) sector(es) de arranque do disco; identifica a **partição ativa** e executa o **2nd stage boot loader** aí alojado

- **volume boot sector(s)** / volume boot control block(s) / sector(es) de arranque da partição: alojam código do 2o estágio do arranque (2nd stage boot loader) e, por vezes, código de menus de arranque que não cabe no sector de arranque do disco

- **2nd stage boot loader**: código alojado no(s) sector(es) de arranque de partição; capaz de aceder a um SF; localiza o SO numa partição do disco e executa-o

# Implementação do SF (3/3): Arranque e Montagem

- **root partition** / partição raíz: contém as partes do SO que o 2nd stage boot loader executa (bem como outros ficheiros do SO); é uma partição cooked
- no arranque do SO, ocorre a **montagem automática** de outras partições
- **mount points** / pontos de montagem:
  - UNIX: diretorias
  - WINDOWS: letras e diretorias (versões + recentes)
- durante o arranque, e em função do tipo de SF das partições montadas, pode ser feita uma verificação da consistência dos SFs dessas partições
- após o arranque: é possível a **montagem manual** de outras partições, além da **montagem automática** de dispositivos *hot-plugable* (e.g, USB, SAS, ...); esta possibilidade inclui drives / volumes remotos, acessíveis via rede

# Exemplos de Sistemas de Ficheiros (1/5)

**UNIX**

- UFS (UNIX File System), baseado no FFS (Berkeley Fast File System)

**Linux**

- Ext2fs (Second Extended File System)
  - o SF original do Linux; considerado muito estável/fiável
  - sem suporte a journaling (adequado a discos pequenos)
- Ext3fs (Third Extended File System)
  - evolução do ext2 com suporte para journaling
- Ext4fs (Fourth Extended File System)
  - evolução do ext3 com suporte a ficheiros e discos de elevada dimensão (>2TB e >32TB, respetivamente)
  - (já) é o SF adotado, por omissão, nas principais distros
- ReiserFS (Hans Reiser File System)
  - um SF journaling, otimizado para ficheiros pequenos

# Exemplos de Sistemas de Ficheiros (2/5)

**Linux** (cont.)

- JFS (Journaled File System, IBM)
  - originalmente desenvolvido pela IBM para o AIX
  - mais tarde portado para OS/2 e doado ao Linux
  - sofisticado; bom desempenho para ficheiros grandes
- XFS (Extents File System, SGI)
  - desenvolvido pela SGI para o IRIX, doado ao Linux
  - sofisticado, robusto, rápido e flexível
  - bom desempenho para ficheiros grandes
- Btrfs (B-Tree File System, da Oracle):
  - apontado como o sucessor do ext4
  - mais escalável, fiável e fácil de gerir

**Solaris**

- ZFS (Zettabyte File System, da Sun); suporta volumes e ficheiros ENORMES

**Mac OS**

- HFS (Hierarchical File System), HFS+ (HFS com journaling)

# Exemplos de Sistemas de Ficheiros (3/5)

**MS-DOS + Windows**

- FAT (File Allocation Table)
  - FAT: array com tantas células quantos os clusters do disco
  - FAT 12/16/32: conforme o número de bits por célula da FAT
  - FAT 12:
    - uso original: diskettes; limite teórico de partição: 32 MB; limite na prática: 15 MB, em disco rígido (MS-DOS 2.0)
  - FAT 16:
    - versão inicial: MS-DOS 3.0; partição ainda limitada a 32 MB; uso + eficiente do disco (mais clusters e mais pequenos); versão "final": MS-DOS 4.0; limite teórico de partição: 2 GB
  - FAT 32:
    - introduzida no Windows 95 OSR2; limite teórico de partição: 2TB; limite com utilitários Microsoft: 32 GB; ficheiros <= 4GB

# Exemplos de Sistemas de Ficheiros (4/5)

**MS-DOS + Windows**

- FAT (File Allocation Table) (cont.)
  - VFAT (Virtual FAT):
    - originalmente, em FAT os nomes são no formato 8.3
    - VFAT: extensão à FAT16/32 para suporte de nomes longos
  - FATX:
    - formato FAT específico para as consolas XBOX
  - exFAT (Extended FAT):
    - Windows XP (hotfix), Embeded CE 6.0, Vista SP1
    - escalabilidade superior a FAT32 (ficheiros e partições)
    - suporte e compatibilidade ainda bastante limitada
  - SFs FAT impõem menos stress em dispositivos FLASH
  - sem suporte a journaling (adequados a discos pequenos)

# Exemplos de Sistemas de Ficheiros (5/5)

**Windows**

- NTFS (NT File System)
  - o SF nativo dos Windows NT 3.x, 2000, XP, Vista, 7, Server
  - segurança: permissões (ACLs), encriptação
  - fiabilidade: journaling, remapping de clusters defeituosos
  - expansibilidade: suporta volumes até 256 Tb
  - flexibilidade: facilidade de redimensionamento
  - armazenamento otimizado de ficheiros pequenos

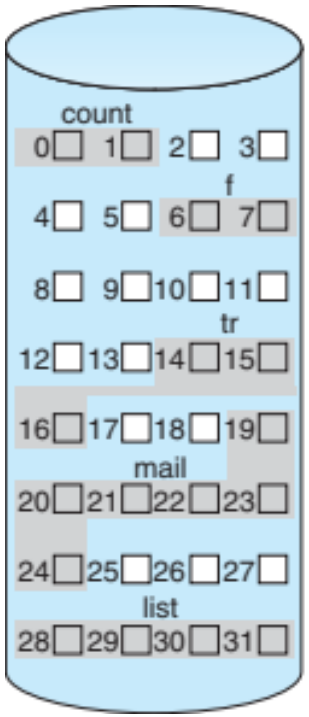
**SFs para suportes ópticos**

- CDFS / ISO9660
- UDF (Universal Disk Format)

## 6.10 Exercícios

# Exercício 6.1: Alocação Contígua

Considere a seguinte representação de um disco com 32 blocos de armazenamento, cada bloco de 512 bytes. Da representação faz parte uma diretoria com 5 ficheiros (assuma que o espaço ocupado pela diretoria não é relevante).




directory

| file | start | length |
|---|---|---|
| count | 0 | 2 |
| tr | 14 | 3 |
| mail | 19 | 6 |
| list | 28 | 4 |
| f | 6 | 2 |

**a)** Qual é a designação do método de alocação ilustrado pela representação fornecida ?

**b)** Qual o espaço consumido pelo ficheiro `tr`, medido em bytes ?

**c**) Qual é o bloco do disco que contém o byte 1033 do ficheiro `list` ?

**d**) Assumindo que uma sequência de blocos livres é representada por um par `<bloco_inicial, numero_de_blocos>`, apresente a lista de sequências de blocos livres do disco da figura, ordenada por ordem crescente do `bloco_inicial`.

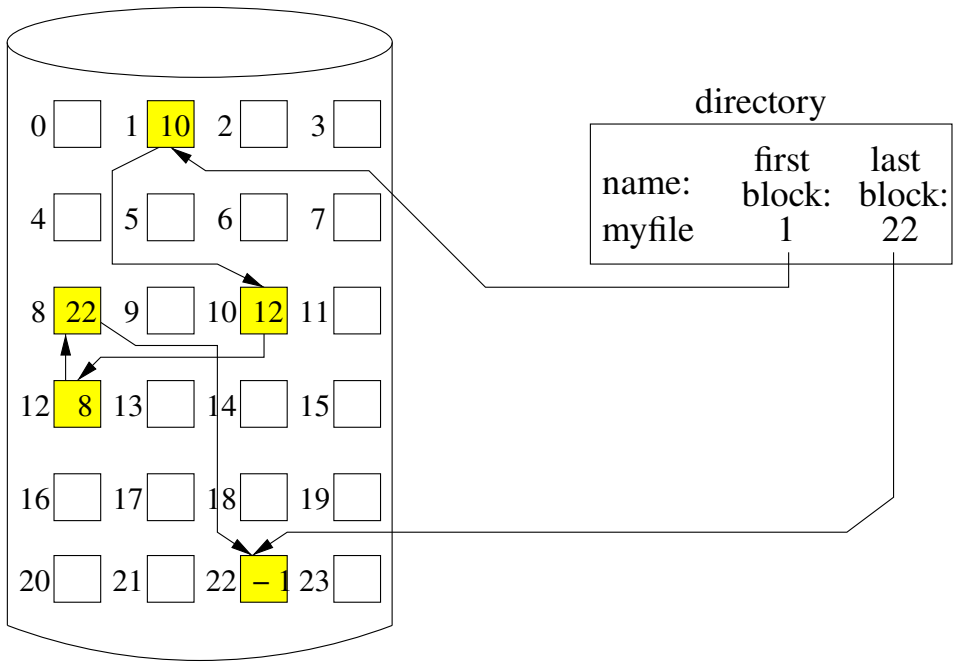
**e**) Considere a lista de sequências da alínea **d)**, e suponha que é necessário criar um novo ficheiro `fso` de 513 bytes de dados.

**e1**) Quais os blocos a atribuir ao ficheiro `fso` considerando cada uma das políticas *first-fit*, *best-fit* e *worst-fit* ?
**e2**) Apresente a nova lista de blocos livres resultante em cada uma das políticas referidas em **e1.**
**e3**) Qual é a medida da fragmentação interna (em bytes) do último bloco de dados do ficheiro `fso` ?

**f**) Forneça uma representação de tipo *FAT* que reflita o estado do disco tal como representado na figura acima.
**g**) Forneça uma representação de tipo *vetor de bits* que reflita o estado (livre/ocupado) dos blocos do disco da figura acima.

# Exercício 6.2: Alocação Ligada

Considere a seguinte representação de um disco, com 24 blocos de armazenamento, medindo cada bloco 4KBytes, e medindo cada índice de bloco 4 bytes. Da representação faz parte uma entrada numa certa diretoria, relativa a um ficheiro **`myfile`** (no contexto deste cenário, assume-se que o espaço ocupado pela diretoria em si é de zero bytes).




**a)** Qual é a designação do método de alocação ilustrado pela representação fornecida ?

**b)** Forneça uma representação equivalente à fornecida, mas em formato FAT.

**c)** Qual o espaço total consumido pelo ficheiro **`myfile`**, medido em bytes ?

**d)** Qual o espaço consumido pelos dados do ficheiro **`myfile`**, medido em bytes ?

**e)** Qual é o bloco do disco que contém o byte 4095 dos dados do ficheiro **`myfile`** ?

**f)** Apresente um vetor de bits que traduza o estado (livre/ocupado) dos blocos do disco.

**g**) Considerando que uma sequência de blocos livres é representada por um par de inteiros da forma `<bloco_inicial, numero_de_blocos>`, apresente as sequências de blocos livres do disco da figura, por ordem decrescente do `bloco_inicial` (auxílio: a primeira sequência é `<23,1>`, a segunda é `<13,9>`, e assim sucessivamente).

**h**) Considerando a lista de sequências da alínea **g**), suponha que é necessário criar um novo ficheiro **`xpto`** de 5000 bytes de dados. Assumindo que os blocos desse ficheiro devem ser contíguos e uma política de alocação *best-fit*:

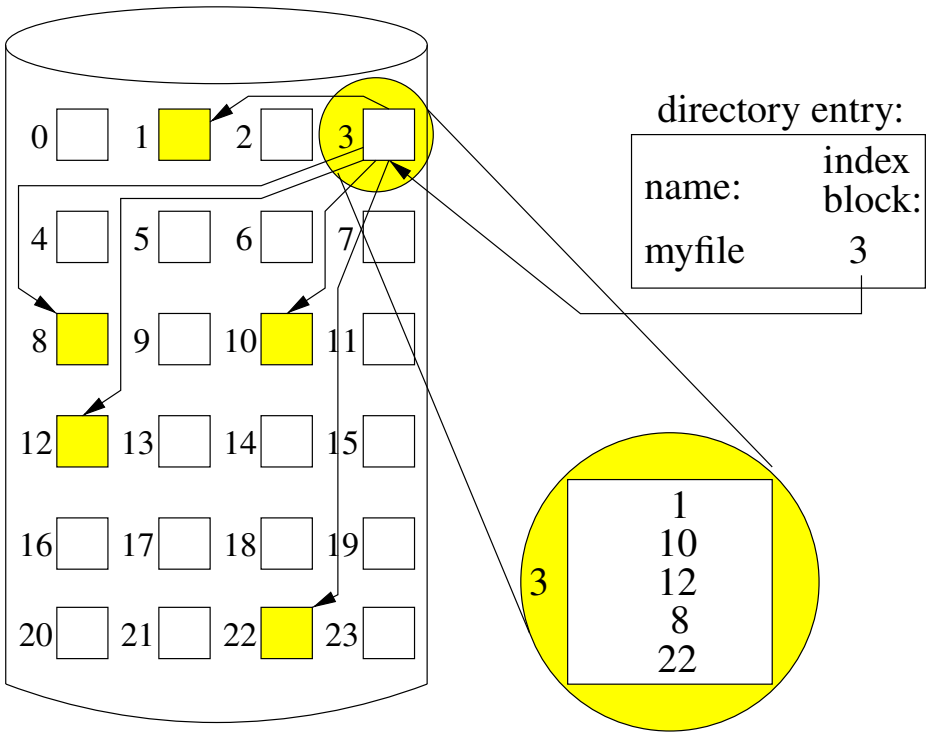
**h1**) Quais os blocos a atribuir ao ficheiro **`xpto`** ?

**h2**) Qual é a medida da fragmentação interna (em bytes) do último bloco de dados do ficheiro **`xpto`** ?

**h3**) Qual a nova sequência de blocos livres que resulta da criação do ficheiro **`xpto`** ?

# Exercício 6.3: Alocação Indexada

Considere a seguinte representação de um disco com 24 blocos de armazenamento, medindo cada bloco 512 bytes. Da representação faz parte uma entrada numa certa diretoria, relativa a um ficheiro **`myfile`**, bem com o respetivo bloco de indexação. No contexto deste cenário, assume-se que o espaço ocupado pela diretoria não é relevante.




**a)** Qual é a designação do método de alocação ilustrado pela representação fornecida ?

**b)** Qual o espaço consumido pelos **dados** do ficheiro **`myfile`**, medido em bytes ?

**c)** Qual o espaço **total** consumido pelo ficheiro **`myfile`**, medido em bytes ?

**d**) Qual é o bloco do disco que contém o byte 1033 dos dados do ficheiro **`myfile`** ?

**e**) Assumindo que uma sequência de blocos livres é representada por um par de números inteiros `<numero_de_blocos, bloco_inicial>`, apresente a lista de sequências de blocos livres do disco da figura, ordenada por ordem decrescente do `numero_de_blocos` (em caso de empate, desempate usando ordem decrescente do `bloco_inicial`).

**f**) Considerando a lista de sequências da alínea **e)**, suponha que é necessário criar um novo ficheiro **`myfile2`** de 513 bytes de dados. Assumindo que os blocos desse ficheiro devem ser contíguos e uma política de alocação *worst-fit*:

- **f1**) Quais os blocos a atribuir ao ficheiro **`myfile2`** ?
- **f2**) Forneça uma representação do bloco de indexação do ficheiro **`myfile2`**.
- **f3**) Qual é a medida da fragmentação interna (em bytes) do último bloco de dados do ficheiro **`myfile2`** ?
- **f4**) Qual a nova lista de sequências de blocos livres após a criação do ficheiro **`myfile2`** ?
- **f5**) Forneça uma representação de tipo FAT considerando apenas os blocos de dados do ficheiro **`myfile2`**.

# REFERÊNCIAS


- "Operating System Concepts, 10th Ed.", Silberschatz & Galvin, Addison-Wesley, 2018: Capítulo 14
- "Fundamentos de Sistemas Operacionais, 6a Ed.", Silberschatz, Galvin & Gagne, LTC, 2004: Capítulo 12
- Anatomy of the Linux file system, Tim Jones:
  `https://developer.ibm.com/tutorials/l-linux-filesystem/`
- Anatomy of the Linux virtual file system switch, Tim Jones:
  `https://web.archive.org/web/20110826183927/http://www.ibm.com/developerworks/linux/library/l-virtual-filesystem-switch/`
- Anatomy of Linux journaling file systems, Tim Jones:
  `https://web.archive.org/web/20080901001148/http://www.ibm.com/developerworks/linux/library/l-journaling-filesystems/`
- Anatomy of ext4, Tim Jones:
  `https://web.archive.org/web/20090223174947/http://www.ibm.com/developerworks/linux/library/l-anatomy-ext4`
- Page Cache, the Affair Between Memory and Files:
  `https://web.archive.org/web/20250801230222/https://manybutfinite.com/post/page-cache-the-affair-between-memory-and-files/`