Sistemas Operacionais

Gestão de arquivos - sistemas de arquivos

Prof. Carlos Maziero

DInf UFPR, Curitiba PR

Fevereiro de 2025

Conteúdo

Gerência de arquivos

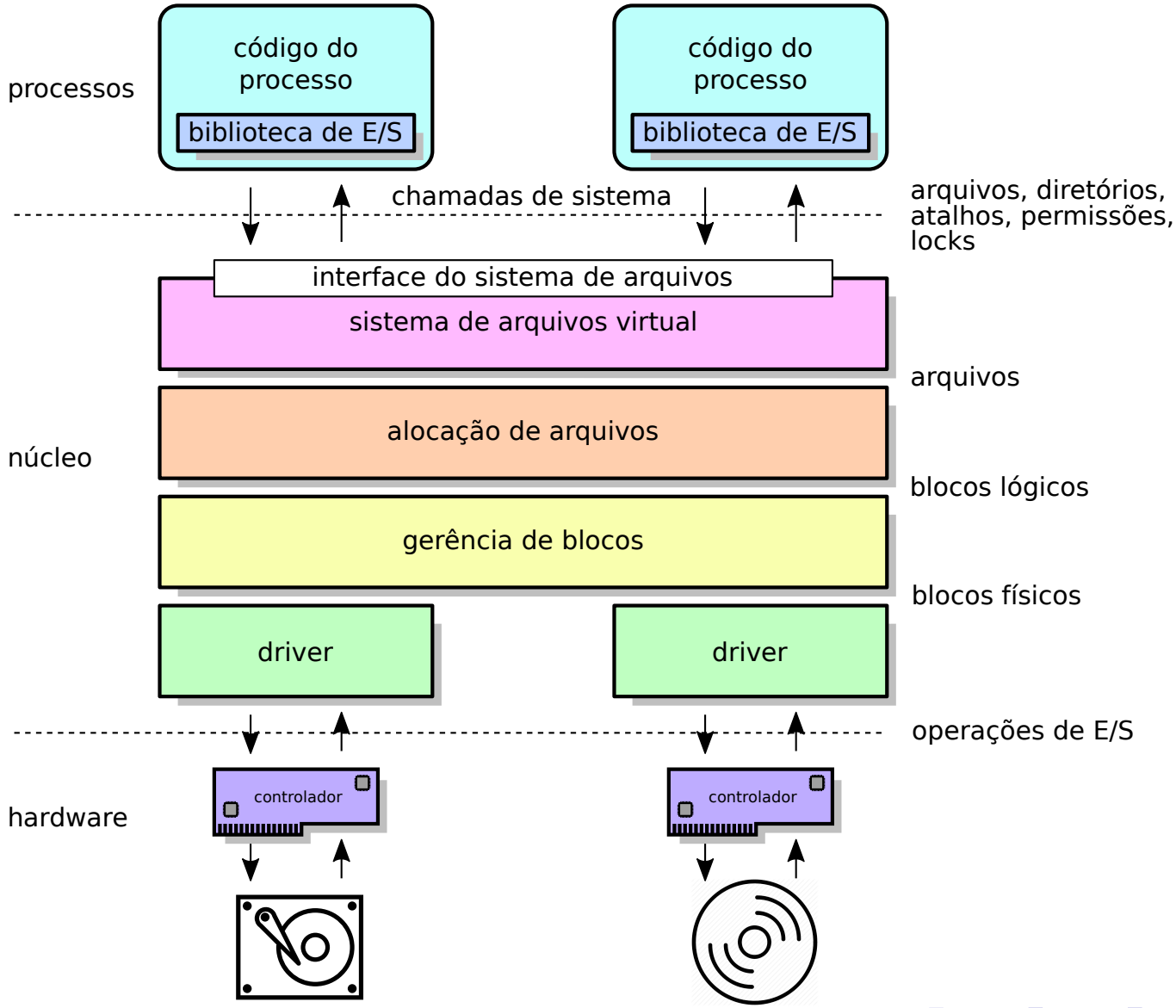
Funções da gerência de arquivos:

- Armazenar arquivos nos dispositivos de armazenamento
- Implementar diretórios e atalhos
- Implementar controle de acesso e travas
- Oferecer interfaces abstratas e padronizadas

Gerência de arquivos

- No hardware:
  - **Dispositivos**: armazenam os dados dos arquivos
  - **Controladores**: circuitos de controle e interface
- No núcleo:
  - **Drivers**: acessam os controladores para ler/escrever
  - **Gerência de blocos**: organiza os acessos aos blocos
  - **Alocação de arquivos**: aloca os arquivos nos blocos
  - **Virtual File System**: visão abstrata dos arquivos
  - **Chamadas de sistema**: interface de acesso ao VFS
- Na aplicação:
  - **Bibliotecas de E/S**: funções padronizadas de acesso

Gerência de arquivos



Organização do disco

Disco:

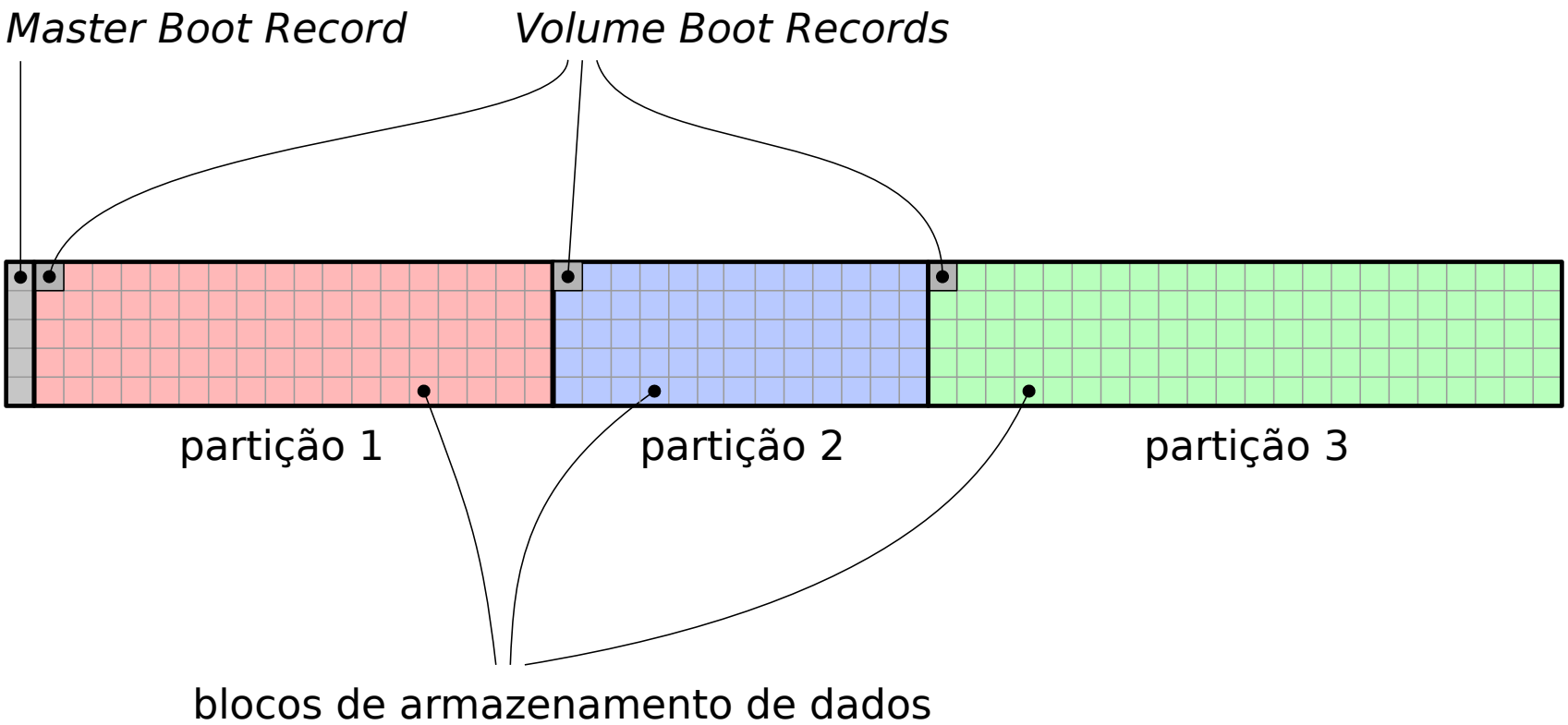
- Vetor de blocos com 512 bytes ou 4096 bytes
- Estruturado em **partições**
- MBR (*Master Boot Record*): tabela de partições + código

Partição:

- Cada uma das áreas do disco
- Possui um VBR (*Volume Boot Record*) no início
- Organizada com um *filesystem* específico

Formatação: estruturas de dados para armazenar arquivos

Organização do disco



No Linux: comando `fdisk /dev/<disco>`

Montagem de volumes

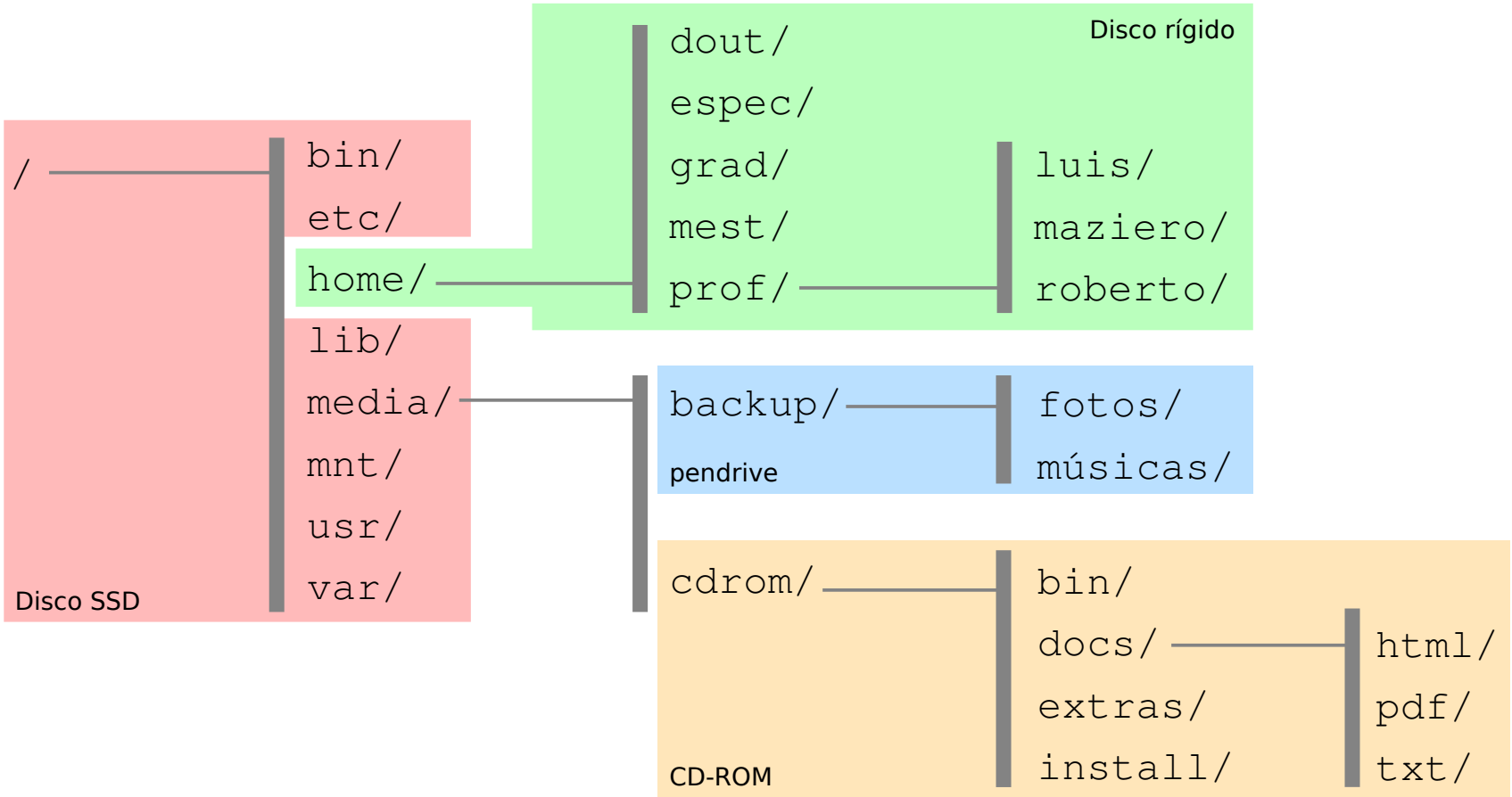
Montagem: preparar volume/partição para ser usado

1. Acessar a tabela de partições (MBR do dispositivo)
2. Acessar o VBR e ler dados do volume
3. escolher *ponto de montagem* (árvore ou floresta)
4. Criar estruturas de memória para representar o volume

Feito durante o *boot* para os dispositivos fixos

Frequente em mídias removíveis (pendrive, CD, etc)

Montagem de volumes em UNIX



No Linux: comandos `df` e `mount`

Blocos físicos e lógicos

- Discos usam blocos físicos de 512 bytes ou 4.096 bytes
- SOs usam *blocos lógicos* ou *clusters*
- Cada bloco lógico usa 2^n blocos físicos consecutivos
- Blocos lógicos de 4K a 32 KBytes são típicos
- Clusters oferecem:
  - ☺ mais desempenho de E/S
  - ☹ mais fragmentação
- Sistemas modernos implementam *sub-block allocation*

Políticas de caching

Caching de blocos de disco:

- Discos são dispositivos lentos!
- *Caching* melhora o desempenho
- Existe *caching* de leitura e de escrita

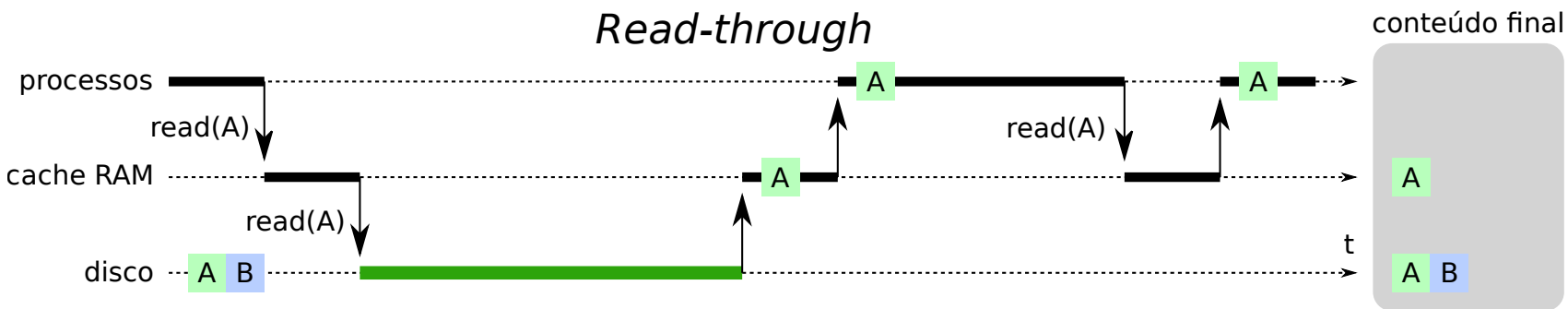
Políticas de *caching*:

- *Read-Through*
- *Read-Ahead*
- *Write-Through*
- *Write-Back*

Políticas de **gestão do cache**: LRU, segunda-chance, etc

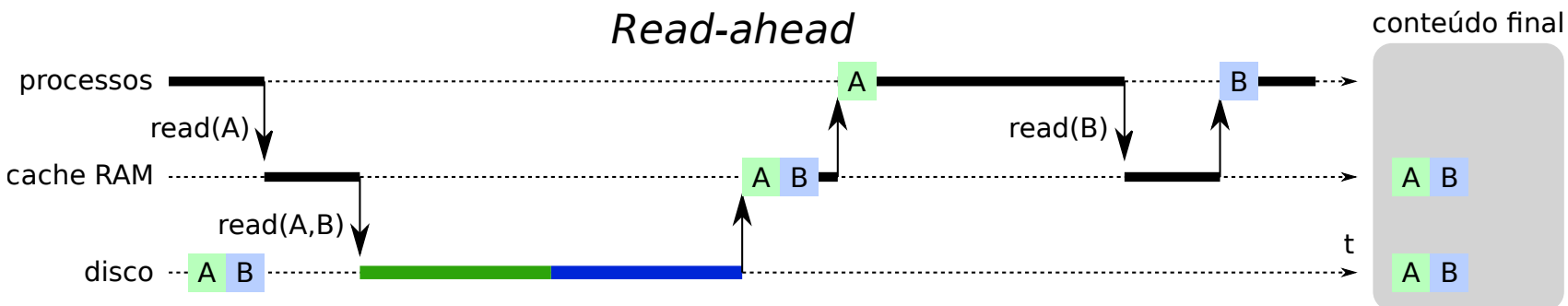
Política *Read-Through*

- O cache é consultado a cada leitura
- Se o bloco não estiver no cache, ele é lido do disco
- Blocos lidos são armazenados no cache



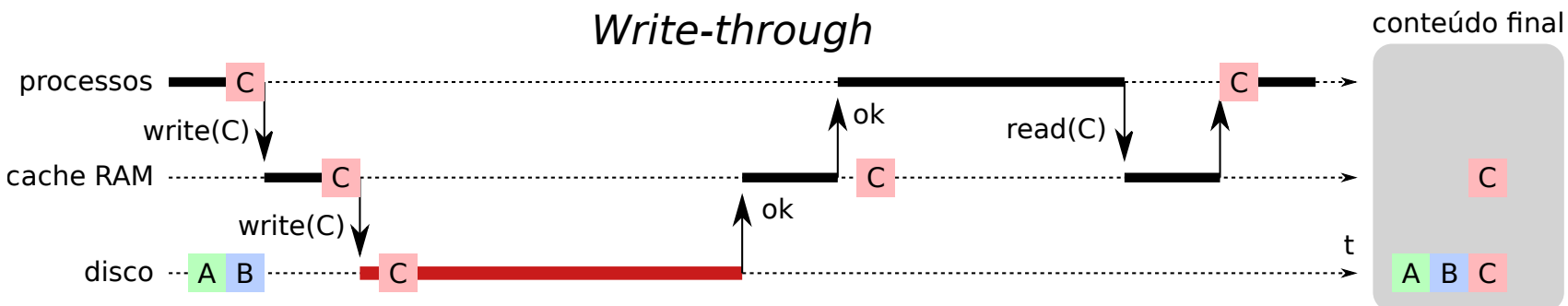
Política *Read-Ahead*

- Ao ler um bloco do disco, traz mais blocos que o requerido
- Blocos adicionais são lidos se o disco estiver ocioso
- Benéfica em acessos sequenciais e com boa localidade



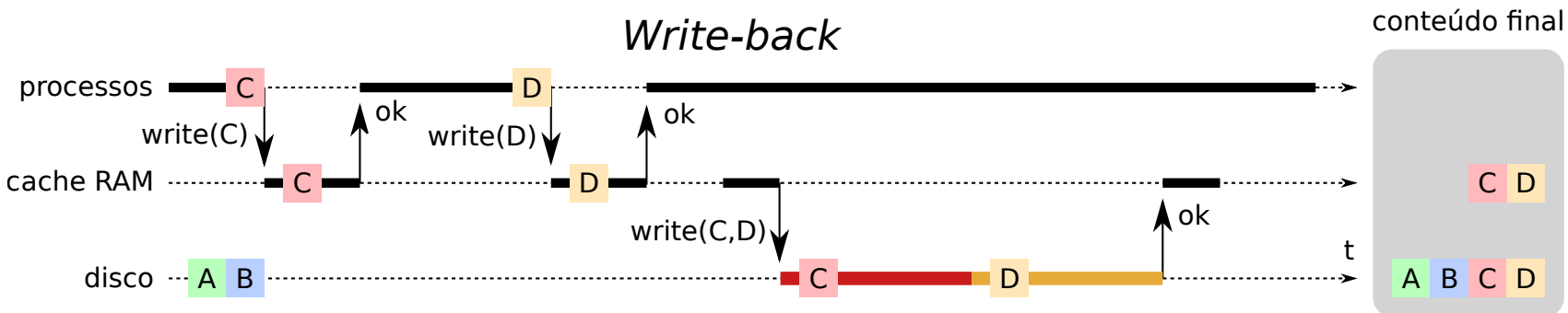
Política *Write-Through*

- As escritas são encaminhadas diretamente ao *driver*
- O processo solicitante é suspenso
- Uma cópia dos dados é mantida em cache para leitura
- Usual ao escrever metadados dos arquivos



Política *Write-back* ou *write-behind*

- As escritas são feitas só no cache
- O processo é liberado imediatamente
- A escrita efetiva no disco é feita mais tarde
- Melhora o desempenho de escrita
- Risco de perda de dados (queda de energia)



Alocação de arquivos

Um arquivo é definido por:

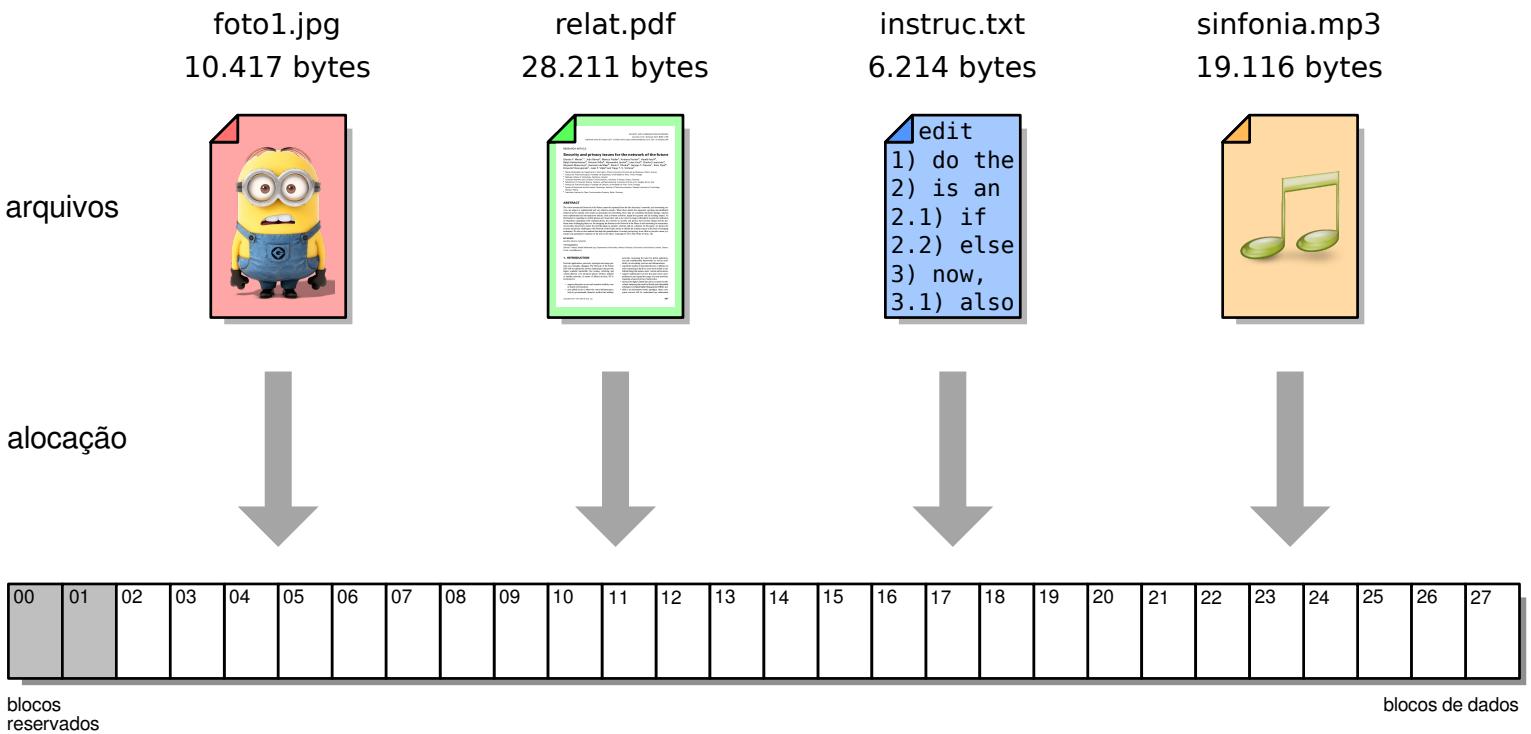
- Conteúdo: vetor de bytes
- Metadados:
  - Atributos: nome, data(s), permissões, etc.
  - Controles: localização dos dados no disco, etc.

FCB – *File Control Block*:

- Um descritor para cada arquivo armazenado
- Contém os **metadados** do arquivo
- Também deve ser armazenado no disco
- **Diretório**: tabela de FCBs

Alocação de arquivos

- Dispositivos físicos são vetores de blocos
- Arquivos têm **conteúdo** e **metadados**
- Como armazenar os arquivos nos blocos do disco?



Alocação de arquivos

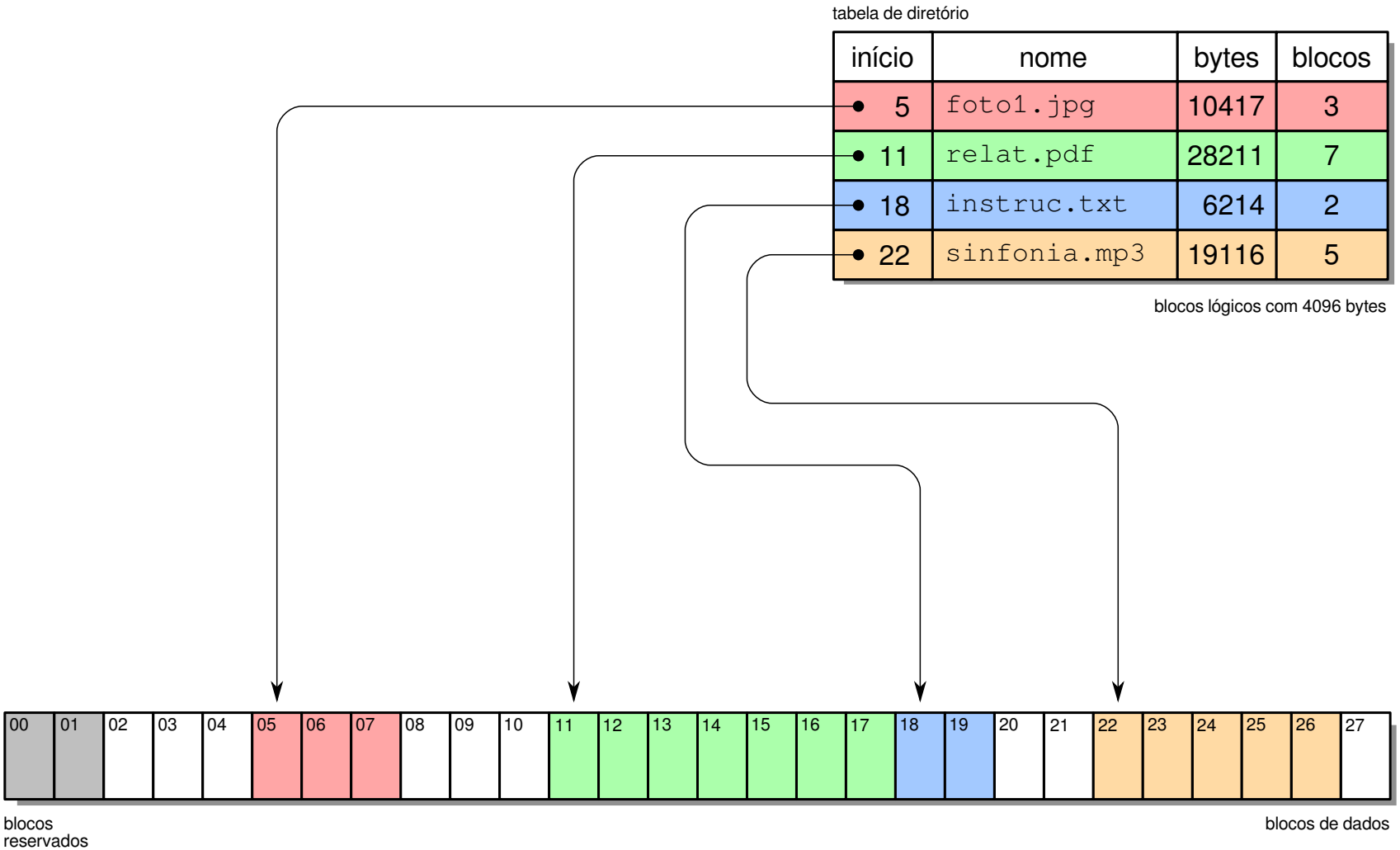
Estratégias de alocação:

- Alocação **contígua**
- Alocação **encadeada** simples e FAT
- Alocação **indexada** simples e multinível

Critérios de avaliação:

- **Rapidez** na leitura e escrita de arquivos
- **Robustez** em relação a erros no disco
- **Flexibilidade** na alocação e modificação de arquivos

Alocação contígua



Alocação contígua

Um arquivo é um grupo de **blocos consecutivos**:

- Acessos sequencial e direto aos dados são rápidos
- Boa robustez a falhas de disco
- Baixa flexibilidade (conhecer o tamanho final do arquivo)
- Forte risco de fragmentação externa
- Estratégia pouco usada
- Usada em CD-ROMs no padrão ISO-9660

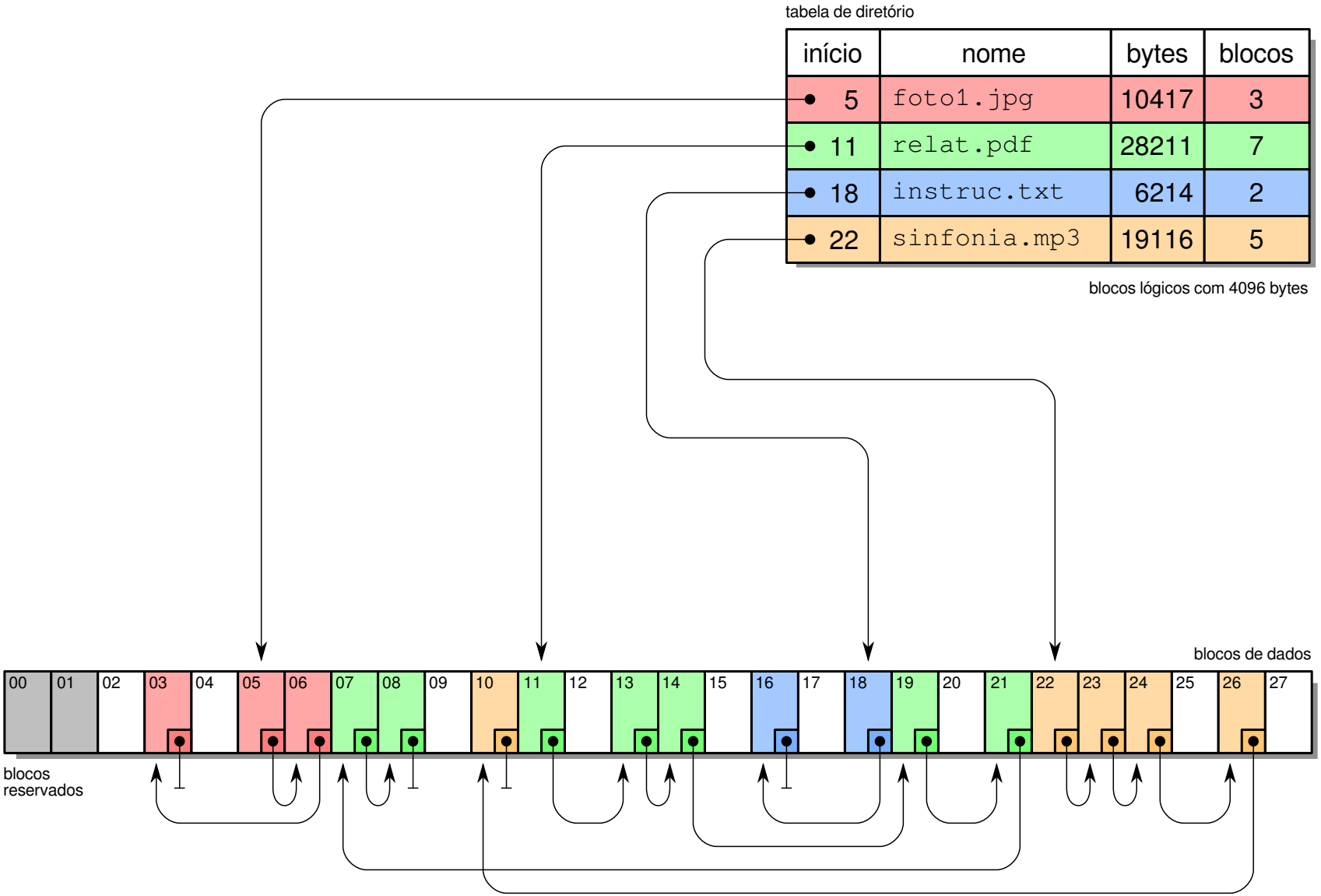
Alocação contígua - acesso direto

Entrada:
i: número do byte a localizar no arquivo
B: tamanho dos blocos lógicos, em bytes
b_0: número do bloco do disco onde o arquivo inicia

Saída:
(b_i, o_i): bloco do disco e *offset* onde está o byte i

$b_i = b_0 + i \div B$ ▷ *divisão inteira*
$o_i = i \bmod B$ ▷ *resto da divisão*
return (b_i, o_i)

Alocação encadeada simples



Alocação encadeada simples

Um arquivo é uma lista encadeada de blocos:

- Bloco contém dados e o **número do próximo bloco**
- Mais flexibilidade na criação de arquivos
- Elimina a fragmentação externa
- Acesso sequencial é usualmente rápido
- Acesso direto é lento (percorrer a lista de blocos)
- Pouco robusto: blocos corrompidos "quebram" os arquivos

Alocação encadeada simples - acesso direto

Entrada:

P: tamanho dos ponteiros de blocos, em bytes

$b_{aux} = b_0$ ▷ *define bloco inicial do percurso*

$b = i \div (B - P)$ ▷ *calcula número de blocos a percorrer*

while $b > 0$ **do**

  $block = read\_block\ (b_{aux})$ ▷ *lê bloco do disco*

  b_{aux} = núm. próximo bloco (extraído de *block*)

  $b = b - 1$

end while

$b_i = b_{aux}$

$o_i = i \bmod (B - P)$

return(b_i, o_i)

Alocação encadeada FAT

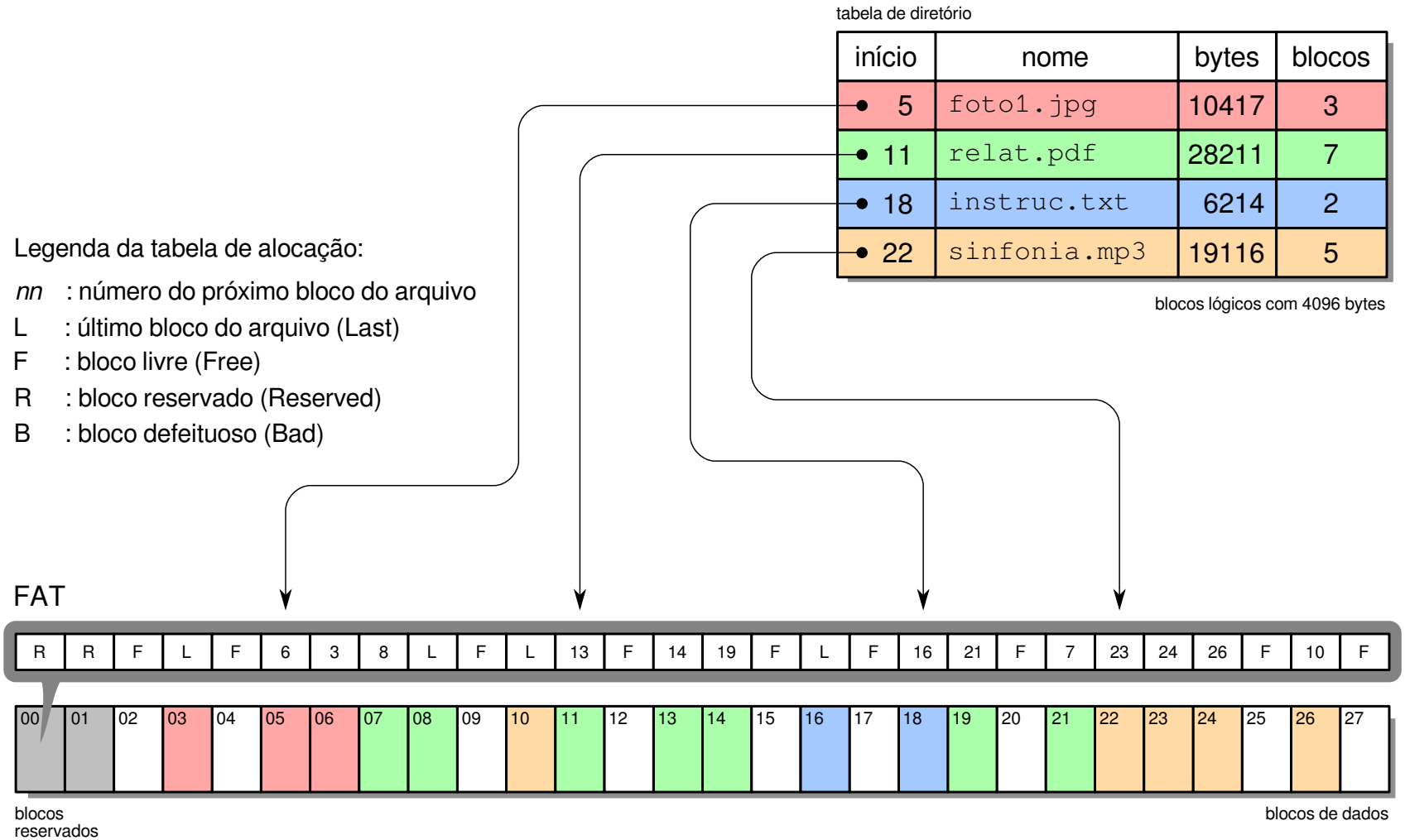
Armazenar ponteiros nos blocos é um problema:

- Diminui tamanho útil dos blocos
- Precisa ler blocos para percorrer lista

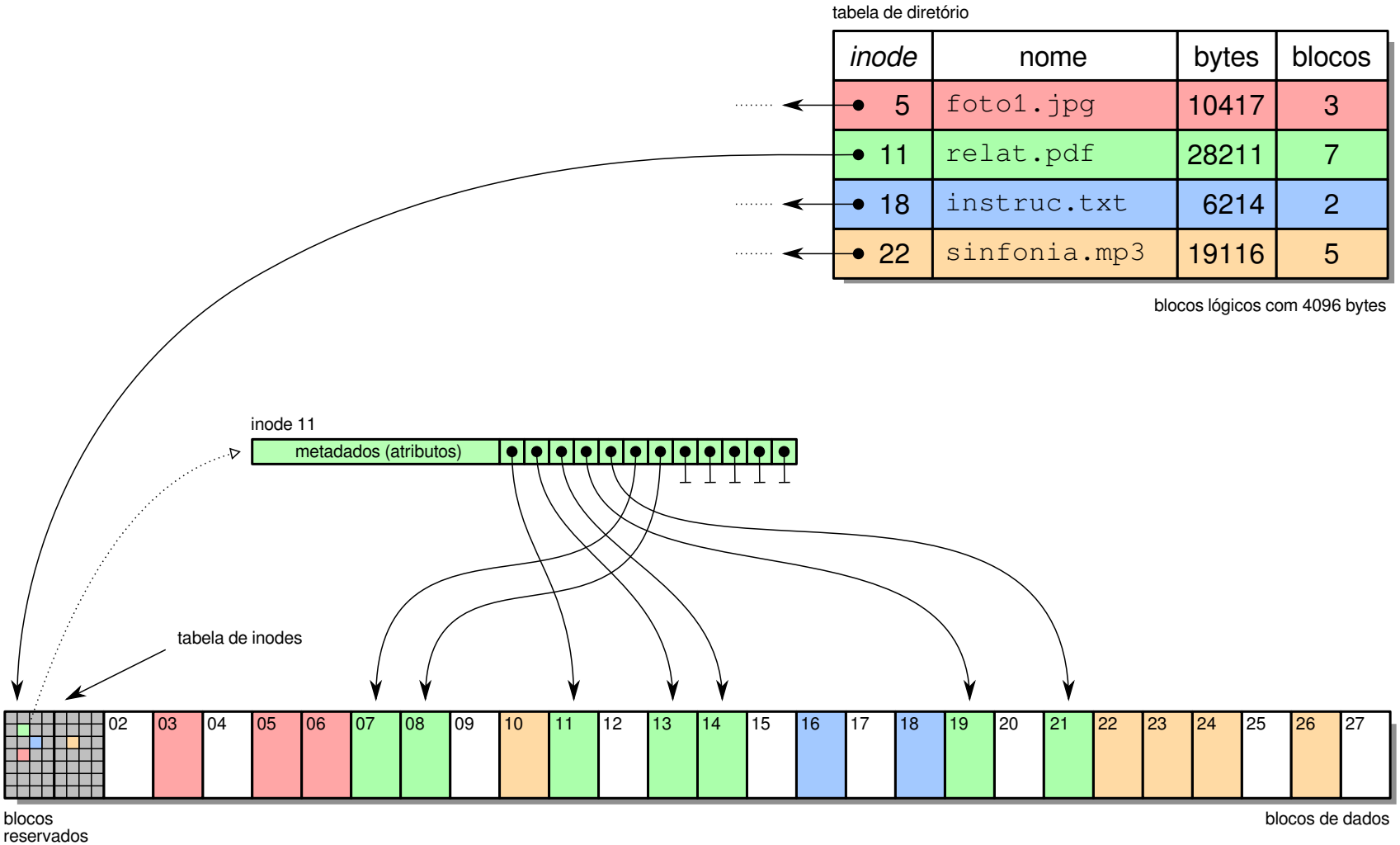
Solução: criar uma **tabela de ponteiros**:

- FAT - *File Allocation Table*
- Tabela de ponteiros armazenada nos blocos iniciais
- Mantida em cache na memória RAM
- Base dos sistemas FAT12, FAT16, FAT32, VFAT, ...

Alocação encadeada FAT



Alocação indexada simples



Alocação indexada simples

Ideia: um índice de blocos separado para cada arquivo

- *Index node* (*inode*): estrutura com índice e metadados
- Tabela de *inodes* mantida na área reservada do disco

Características:

- Rápida para acessos sequenciais e diretos
- Robusta para erros em blocos de dados
- *Inodes* são pontos frágeis
- Cópias da tabela de *inodes* espalhadas no disco

Alocação indexada multinível

Problema:

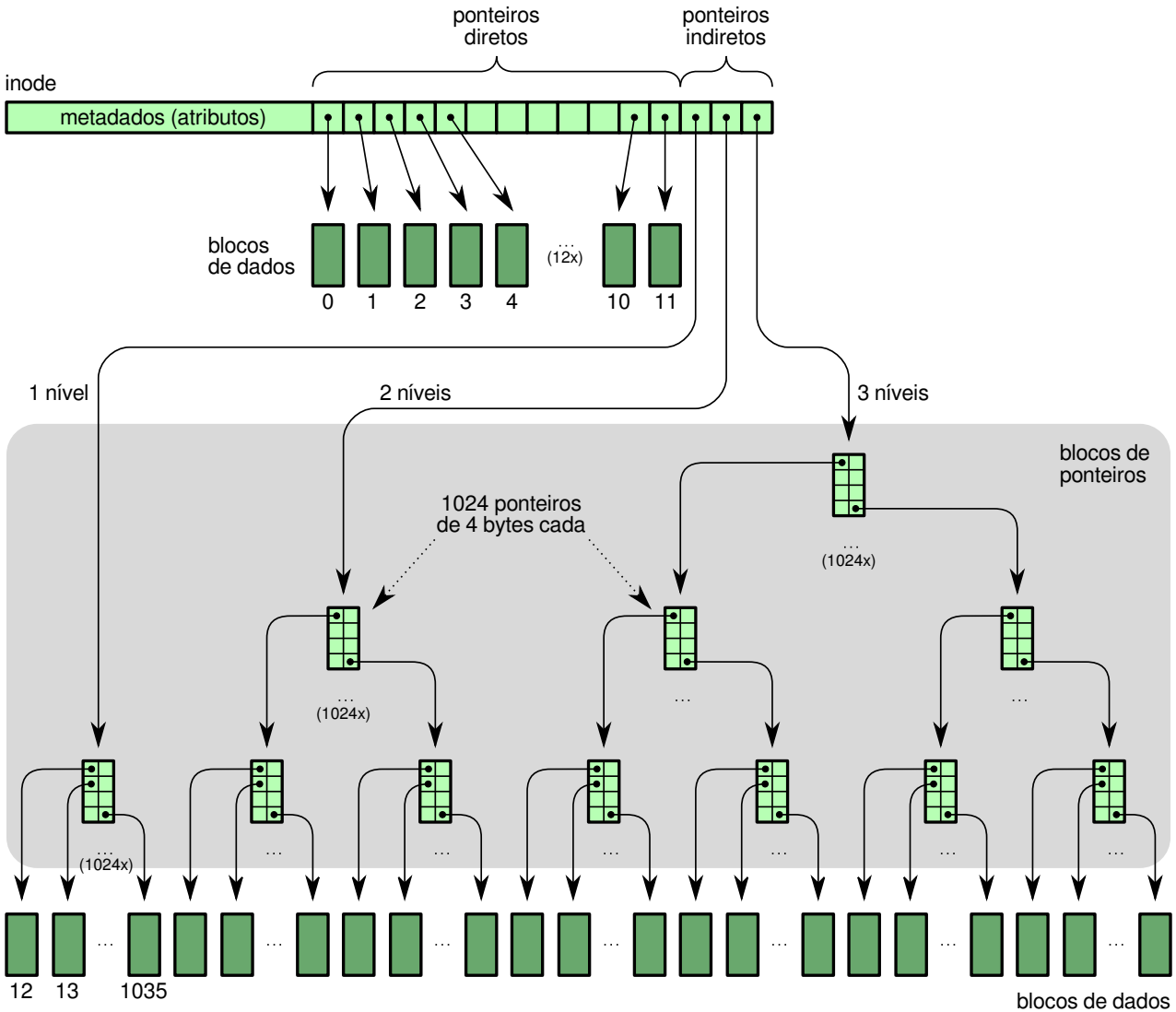
- *inode* tem tamanho fixo
- Número de ponteiros limita o tamanho do arquivo

Exemplo:

- *inode* com 64 ponteiros de blocos
- Blocos de 4 Kbytes
- Permite armazenar arquivos até $64 \times 4 = 256$ KBytes

Solução: Transformar o vetor de blocos em uma árvore

Alocação indexada multinível



Tamanho máximo de arquivo

Considerando um sistema indexado com:

- blocos lógicos de 4 Kbytes
- ponteiros de blocos com 4 bytes
- $\longrightarrow$ cada bloco de ponteiros contém 1.024 ponteiros

Cálculo do tamanho máximo de um arquivo:

$$
\begin{aligned}
max &= 4.096 \times 12 && \text{(ponteiros diretos)} \\
&+ 4.096 \times 1.024 && \text{(ponteiro 1-indireto)} \\
&+ 4.096 \times 1.024^2 && \text{(ponteiro 2-indireto)} \\
&+ 4.096 \times 1.024^3 && \text{(ponteiro 3-indireto)} \\
&= 4.402.345.721.856 \ bytes \\
max &\approx 4 \ Tbytes
\end{aligned}
$$

Alocação indexada multinível - acesso direto I

1: **Entrada:**
2: $ptr[0...14]$: vetor de ponteiros contido no *i-node*
3: $block[0...1023]$: bloco de ponteiros para outros blocos (1.024 ponteiros de 4 bytes)

4: $o_i = i \bmod B$
5: $pos = i \div B$
6: **if** $pos < 12$ **then** ▷ *usar ponteiros diretos*
7: $b_i = ptr[pos]$ ▷ *o ponteiro é o número do bloco* b_i
8: **else**
9: $pos = pos - 12$
10: **if** $pos < 1024$ **then** ▷ *usar ponteiro 1-indireto*
11: $block_1 = read\_block\ (ptr[12])$
12: $b_i = block_1[pos]$
13: **else**

Alocação indexada multinível - acesso direto II

14: $pos = pos - 1024$
15: **if** $pos < 1024^2$ **then** ▷ *usar ponteiro 2-indireto*
16: $block_1 = read\_block\ (ptr[13])$
17: $block_2 = read\_block\ (block_1[pos \div 1024])$
18: $b_i = block_2[pos \bmod 1024]$
19: **else** ▷ *usar ponteiro 3-indireto*
20: $pos = pos - 1024^2$
21: $block_1 = read\_block\ (ptr[14])$
22: $block_2 = read\_block\ (block_1[pos \div (1024^2)])$
23: $block_3 = read\_block\ (block_2[(pos \div 1024) \bmod 1024])$
24: $b_i = block_3[pos \bmod 1024]$
25: **end if**
26: **end if**
27: **end if**
28: **return**(b_i, o_i)

Estrutura do *Inode* do Ext4 (simplificado)

Offset	Size	Name	Description
0x00	2	`i_mode`	entry type and permissions
0x02	2	`i_uid`	user ID
0x04	4	`i_size_lo`	size (bytes)
0x08	4	`i_atime`	data access time
...	...	...	...
0x18	2	`i_gid`	group ID
0x1A	2	`i_links_count`	hard links counter
0x1C	2	`i_blocks_lo`	number of blocks
0x20	4	`i_flags`	several flag bits
...	...	...	...
0x28	60	`i_block[15]`	block map (pointers)
...	...	...	...

Alocação por extensões

Problema:

- Alocação indexada é ruim para arquivos muito grandes
- Muitos metadados: um ponteiro para cada bloco
- Ineficiência e custo de gestão

Observações:

- Alocação contígua é rápida mas inflexível
- Alocação contígua usa poucos metadados

Ideia: combinar alocação indexada e contígua

Alocação por extensões

Alocação por extensões:

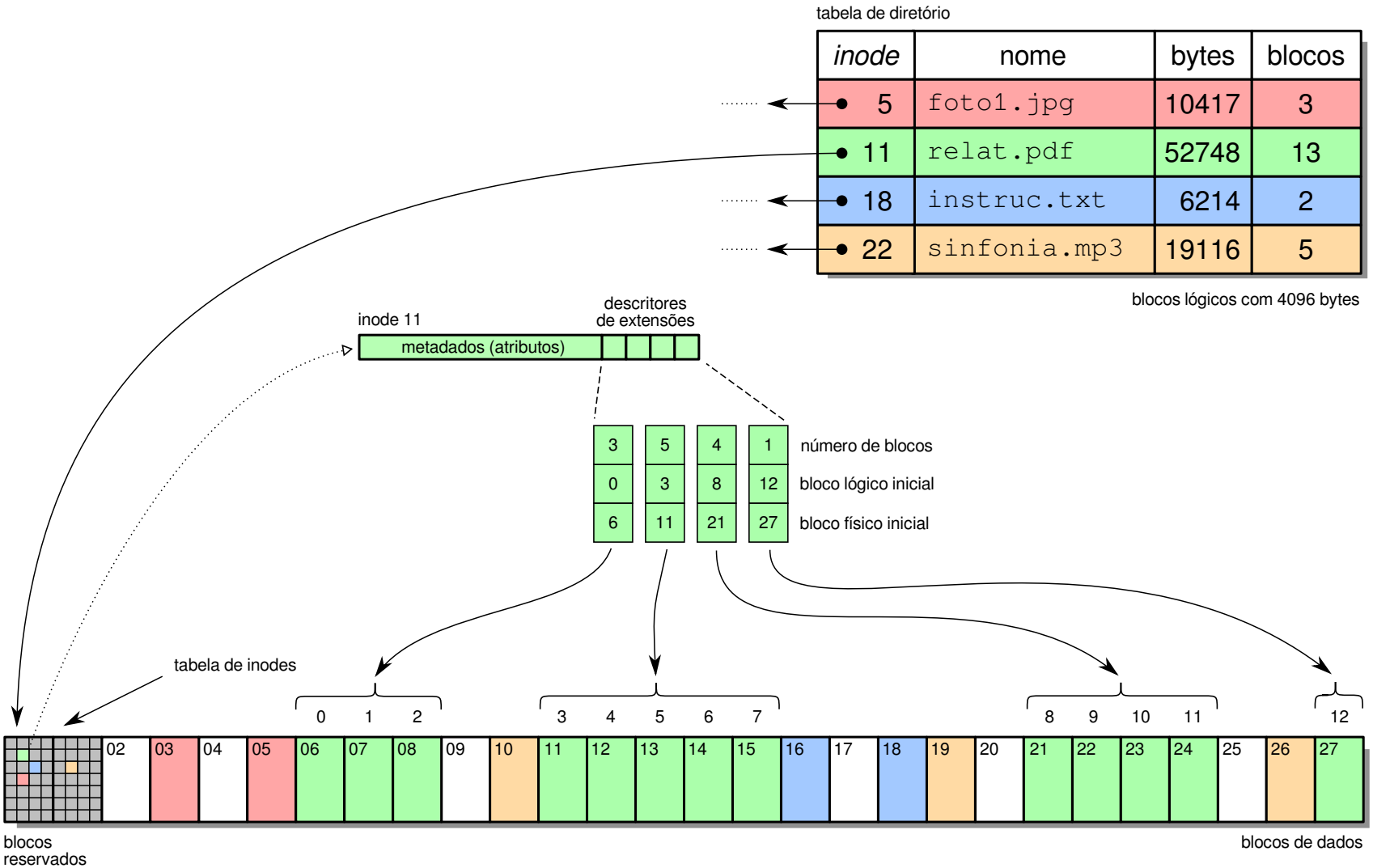
- Extensão: grupo de blocos contíguos no disco
- Arquivo é alocado em uma ou mais extensões

Cada extensão usa um pequeno *descritor de extensão*:

- Quantidade de blocos
- Número do bloco lógico inicial
- Número do bloco físico inicial

Cada *i-node* guarda poucos descritores de extensão e aponta para **árvore de descritores de extensão** em blocos adicionais

Alocação por extensões



Comparação entre estratégias

Estratégia	rapidez sequencial	rapidez aleatória	robusta	flexível	impõe limites
Contígua	✓	✓	✓		
Encadeada	✓			✓	✓ (3)
FAT	✓	✓	✓ (1)	✓	✓ (3)
Indexada	✓	✓	✓ (2)	✓	✓ (4)
Extensões	✓	✓	✓ (2)	✓	

1. Tabela FAT é ponto sensível (replicada)
2. Tabela de inodes é ponto sensível (replicada)
3. Tamanho dos ponteiros limita número de blocos
4. Limites no tamanho de arquivo e número de arquivos

Gestão do espaço livre

Registro dos blocos livres:

- Importante ao criar ou aumentar arquivos
- Atualizado a cada operação no disco

Estratégias:

- Mapa de bits na área reservada
- Listas/árvores de blocos livres
- Tabela de extensões livres
- FAT

Falhas e recuperação

Conteúdo de um sistema de arquivos:

Dados: conteúdo dos arquivos em si (dados do usuário, programas, bibliotecas, arquivos do sistema, etc)

Metadados: estrutura dos diretórios, alocação dos blocos de dados dos arquivos, atributos dos arquivos, etc.

Metadados definem uma imensa estrutura de dados no disco, atualizada a cada alteração no disco.

Falhas e recuperação

Alterações no sistema de arquivos envolvem várias operações.

Exemplo: criar o arquivo "`/home/user/novo.txt`":

1. Encontrar um *inode* livre e marcá-lo como ocupado no mapa de *inodes*
2. Preencher esse *inode* c/ os atributos de "`novo.txt`"
3. Localizar o *inode* de "`/home/user/`"
4. Incluir em "`/home/user/`" uma entrada com o nome e o *inode* de "`novo.txt`"

O disco só realiza uma operação por vez (sequencial).

Falhas e recuperação

Falha por *crash*: queda de energia ou travamento

Se ocorrer um crash ao criar o arquivo "`novo.txt`" ?

- Entre as operações 1 e 2: um *inode* marcado como ocupado, mas não será usado nem referenciado.
- Entre as operações 2 e 4: um *inode* marcado como ocupado e preenchido, mas não aparece em diretório e não poderá ser localizado.

Resultado: sistema de arquivos pode ficar **inconsistente**.

Falhas e recuperação

Inconsistências podem levar a perda de dados:

- Perda de conteúdo de arquivos
- Perda de arquivos
- Perda de diretórios

Estratégias para minimizar as consequências das falhas:

- Escrita síncrona dos metadados
- Verificação do sistema de arquivos
- Sistemas de arquivos com *Journaling*
- Sistemas de arquivos com *Copy-on-Write*

Verificação do sistema de arquivos

Ao iniciar, percorrer as estruturas de dados do FS:

- Mapas de blocos e *inodes*
- Entradas dos *inodes*
- Tabelas de diretórios
- *Checksums* dos metadados

Verificar e corrigir inconsistências:

- Blocos ocupados, mas não apontados por inodes
- Blocos livres, mas apontados por inodes
- Blocos apontados por mais de um inode
- Inodes ocupados mas não referenciados em diretórios
- ...

Verificação do sistema de arquivos

A verificação:

- Deve ser feita na inicialização
- Ao retornar de um *crash* (como detectar?)
- Antes do sistema ser usado por aplicações
- Necessita ferramentas específicas (exemplo: "`fsck.ext4`")
- Processo lento (pode demorar horas)

Sistemas de arquivos com *Journaling*

Ideia:

- Registrar previamente as alterações nos metadados
- Registro prévio: antes de fazer as alterações
- Incluir dados suficientes para poder repeti-las
- Área de registro separada: *journal*

No caso de um crash:

- Ao reiniciar, consulta o journal
- Analisa as operações pendentes e completa-as

Similar à gestão de transações em BDs (*write-ahead logging*)

Sistemas de arquivos com *Journaling*

1. Antes de alterar o FS, registrar no *journal* as escritas realizadas; para criar "`/home/user/novo.txt`" serão necessárias estas operações (*inode* livre 317):
   1. Marcar o *inode* 317 como ocupado no mapa de *inodes*
   2. Preencher o *inode* 317 com os atributos de "`novo.txt`"
   3. Incluir no diretório "`/home/user/`" uma entrada com o nome "`novo.txt`" apontando para o *inode* 317
2. Realizar as operações no sistema de arquivos
3. Caso tenha sucesso, descartar o registro
4. Caso ocorra uma falha de *crash*, analisar o registro ao reiniciar, completar as operações pendentes e descartar o registro

Sistemas de arquivos com *Journaling*

Implementação do registro:

- Buffer circular de tamanho fixo
- Escritas sequenciais (bom desempenho)
- Escrita síncrona (minimizar risco ao journal)
- Somas de verificação (*checksums*) para consistência

Journaling é muito usado: NTFS (Windows), HFS (MacOS), Ext4 (Linux), JFS (IBM).

Geralmente só se aplica aos metadados (desempenho)

Sistemas de arquivos *Copy-on-Write*

Também denominados *Versioning File Systems*

Ao atualizar o conteúdo de um bloco:

- FS convencional: escreve o conteúdo na posição do próprio bloco, **sobrescrevendo** o conteúdo anterior
- FS *copy-on-write*: escreve o conteúdo em um bloco livre, **preservando** o conteúdo anterior

Conceito similar a "*Time Machine*" e GIT

Implementado no ZFS, BTRFS e ReFS (Windows)

Sistemas de arquivos *Copy-on-Write*

Funcionamento:

- Usar espaço livre para versões anteriores dos blocos
- Aplicável a dados e de metadados
- Construir “snapshots” dos estados anteriores do disco
- Retornar a um estado anterior em caso de crash
- Versões velhas podem ser removidas para abrir espaço

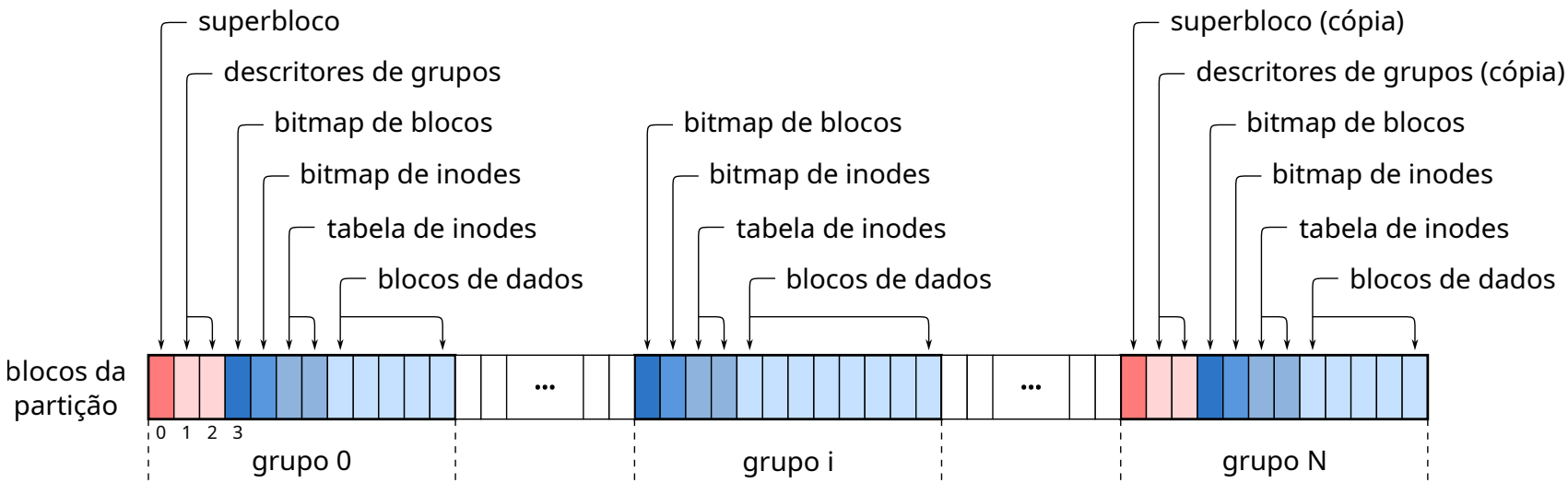
Snapshot:

- Conjunto de blocos com versão anterior do FS
- Usuário pode ver versões anteriores dos arquivos
- FS pode retornar a ele em caso de crash

O sistema Ext4

- Sistema de arquivos mais usado em Linux
- Deriva dos sistemas de arquivos anteriores Ext, Ext2 e Ext3
- Suporta partições com 64 ZBytes e arquivos com 16 TBytes
- Usa *journaling* e *checksum* de metadados
- Suporta quotas, controle de acesso avançado e criptografia
- Suporta alocação indexada multinível e alocação por extensões
- Gestão de espaço livre por bitmaps de blocos livres e bitmaps de *inodes* livres

Exemplo: o sistema Ext4



Blocos de 4 KBytes organizados em grupos de blocos contíguos

Grupo: geralmente 32.768 blocos ou 128 Mbytes

Exemplo: o sistema Ext4

O grupo 0:

- Superbloco (1 bloco): dados do FS (núm. blocos, grupos, *inodes*, etc); 1.024 bytes iniciais p/ código de boot.
- Tabela de descritores de grupos
- Bitmap de blocos (1 bloco)
- Bitmap de *inodes* (1 bloco)
- Tabela de *inodes* (alguns blocos)
- Blocos de dados (todos os demais blocos do grupo)

Os demais grupos: bitmaps de blocos e de *inodes*, tabela de *inodes*, blocos de dados.

Alguns grupos: cópias superbloco e descritores de grupos

Exemplo: o sistema Ext4

Inodes especiais no grupo 0:

Inode	**Uso**
0	não usado
1	lista de blocos defeituosos na partição
2	diretório raiz
3	controle de quota de uso por usuários
4	controle de quota de uso por grupos de usuários
5	inicializador (*Boot loader*)
6, 7, 9, 10	reservados
8	arquivo de registro (*journal*)
11	primeiro *inode* disponível para arquivos