Manipulação de Arquivos
Sistema de Arquivos UNIX

Taisy Silva Weber

UFRGS

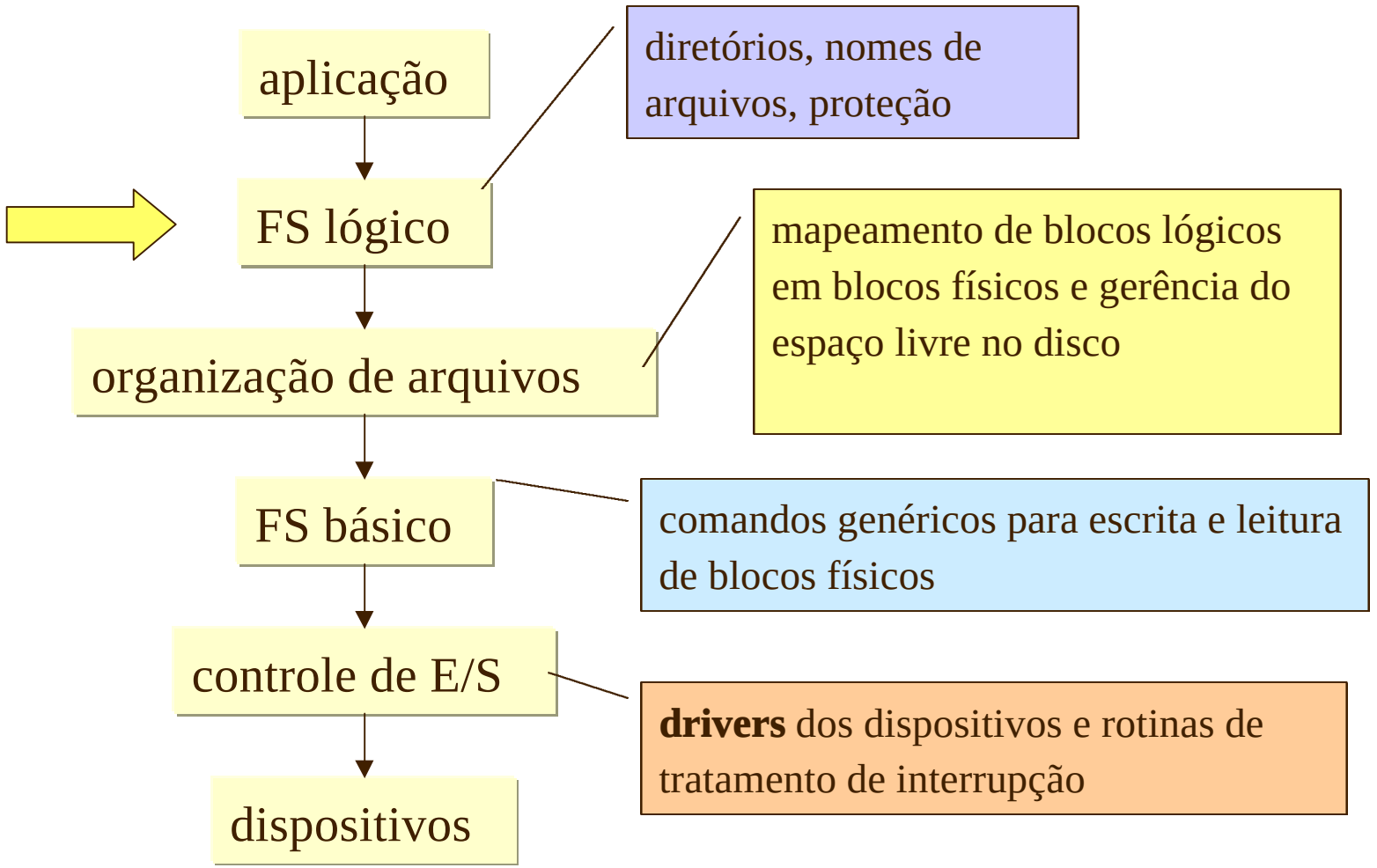
Trabalhando com arquivos

- ✓ conceitos básicos sobre FS
  - revisão — Operating System Concepts - Silberschatz & Galvin, 1998
  - sistema de arquivos UNIX
    - diretórios, arquivos e dispositivos (
- ✓ syscalls e funções
  - chamadas de sistema (write, read, open, etc.),
  - streams, chamadas a standard I/O (

  tratamento de exceções (

  chamadas de sistema avançadas (

Organização multinível



Sistema de arquivos

- componentes
  - arquivos
  - diretório
  - partições
- arquivo

organiza e fornece

disco lógico ou volume: conjunto de diretórios

coleção de informações relacionadas,

tipo abstrato de dado

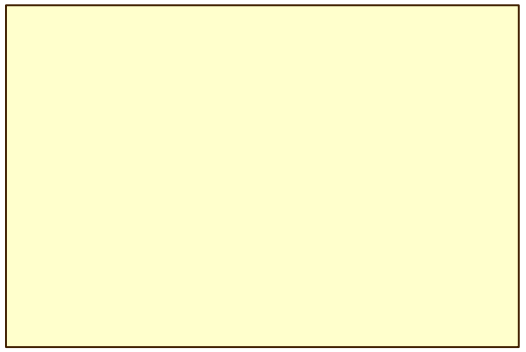
vamos tratar inicialmente de arquivos regulares

FS Linux

- Linux mantém o modelo de FS UNIX
  - um arquivo pode ser qualquer coisa capaz de manipular a saída e a entrada de um fluxo de

exemplos

- objetos armazenados em disco
- dados buscados pela rede
- driver de dispositivo
- canais de comunicação entre



Atributos de arquivos

- ✓ nome
- ✓ tipo
- ✓ localização
- ✓ tamanho
- ✓ proteção
- ✓ hora, data
- ✓ ...

pointer para um dispositivo e localização do arquivo nesse dispositivo

informação para controle de acesso: read, write, execute, ...

criação, última modificação,

dependente do sistema

Operações sobre arquivos

- ✓ criar — alocar espaço e criar uma entrada no diretório
- ✓ escrever — write pointer
- ✓ ler — read pointer

  (write pointer e read pointer: pode ser o mesmo ponteiro)
- ✓ reposicionar dentro do arquivo (
- ✓ remover — liberar o espaço e apagar a entrada no diretório
- ✓ truncar
- ✓ renomear

  (truncar e renomear: todos os atributos ficam iguais menos o tamanho (que fica zero))

... operações

- ✓ copiar
- ✓ abrir (open)
- ✓ fechar (close)

o SO mantém informação sobre arquivos abertos em uma **tabela**

pode ser usado

open file table

a abertura retorna um *índice* para a tabela (descritor de arquivo)

básicas:
open
close
read
write
ioctl

passa controles específicos para um driver de dispositivo

Tipos de arquivo **UNIX**

- ✓ extensão

  exe, com, bin, obj, asm, bat, txt, doc, lib, ps, devi, gif, arc, zip, ...

  extensões podem ser usadas mas não são reconhecidas

- ✓ números mágicos

  número no início de alguns arquivos identifica tipo

- ✓ tipos suportados

  arquivos regulares
  dispositivos de bloco e de caracter

Estrutura de arquivos

- ✓ formato especial determinado pelo SO
  - ✓ imposição de estrutura pelo SO

por exemplo para facilitar carga na memória e localização da primeira instrução a ser executada de um arquivo executável

  - ✓ quanto maior a quantidade de estruturas suportadas ⇨ mais complexo o SO

UNIX: uma única estrutura - seqüência de bytes (8 bits)

Blocos

- registros lógicos

UNIX: registro lógico é 1 byte

- blocos físicos

exemplos:
setor (512 bytes)
cluster no DOS ou
data block no UNIX

todas as operações de E/S de baixo nível são realizadas sobre um bloco (e não sobre frações de bloco)

os programas trabalham com funções de mais alto nível sobre registros lógicos

Diretório

- ✓ diretório — volume table of contents

guarda informações de todos os arquivos no volume (ou partição)

pode ser visto como uma **tabela de símbolos** que mapeia nomes de arquivos para as entradas da tabela

cada entrada contém os atributos de um arquivo

em UNIX o diretório também é um arquivo

entretanto é necessário usar operações especiais para sua manipulação

opendir / readdir

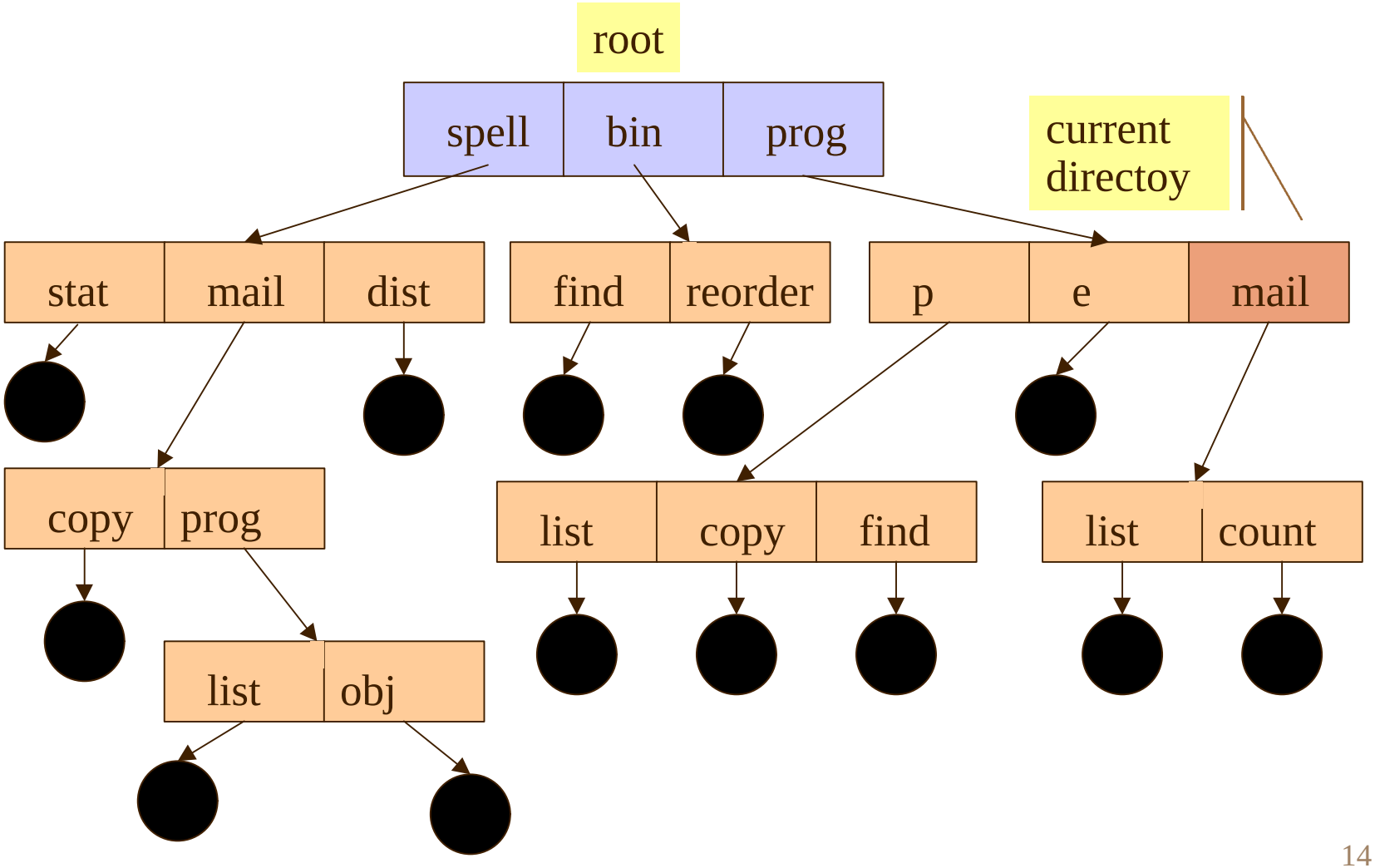
Operações sobre o diretório

- ✓ procurar por um arquivo
- ✓ criar um arquivo e adicionar ao diretório
- ✓ deletar um arquivo e remover do diretório
- ✓ listar o diretório
- ✓ renomear um arquivo
- ✓ percorrer o sistema de arquivos

para fazer backup por exemplo

Hierarquia

diretório em árvore



Subdiretórios

facilitam a localização dos arquivos

- subdiretórios:
  - permitem ao usuário estabelecer uma **organização lógica** para seus arquivos
  - subdiretórios são também **arquivos**

com formato específico estabelecido pelo sistema

- root (`/`):
  - topo da hierarquia
    - contém todos os arquivos do sistema em

`man hier` para descrição da hierarquia de diretórios no sistema

Diretório atual

current directory

- diretório em uso (no momento) `./`
  - referências a arquivo são buscadas no diretório atual
    diretório atual através de syscall
    - `cd` chama a syscall correspondente
- *path names*
  - absoluto — começa na raiz e segue até a folha
  - relativo — começa no

Operações sobre subdiretórios

- deletar subdiretório
  - só permitir quanto estiver vazio
  - deletar todo o seu conteúdo incluindo subdiretórios

`rmdir`

`rm` (remove)

- acessar arquivos de outros usuários
  - conhecer o caminho (**path)**

Diretório como grafo acíclico

- ✓ um arquivo (ou subdiretório) pode pertencer a dois subdiretórios diferentes

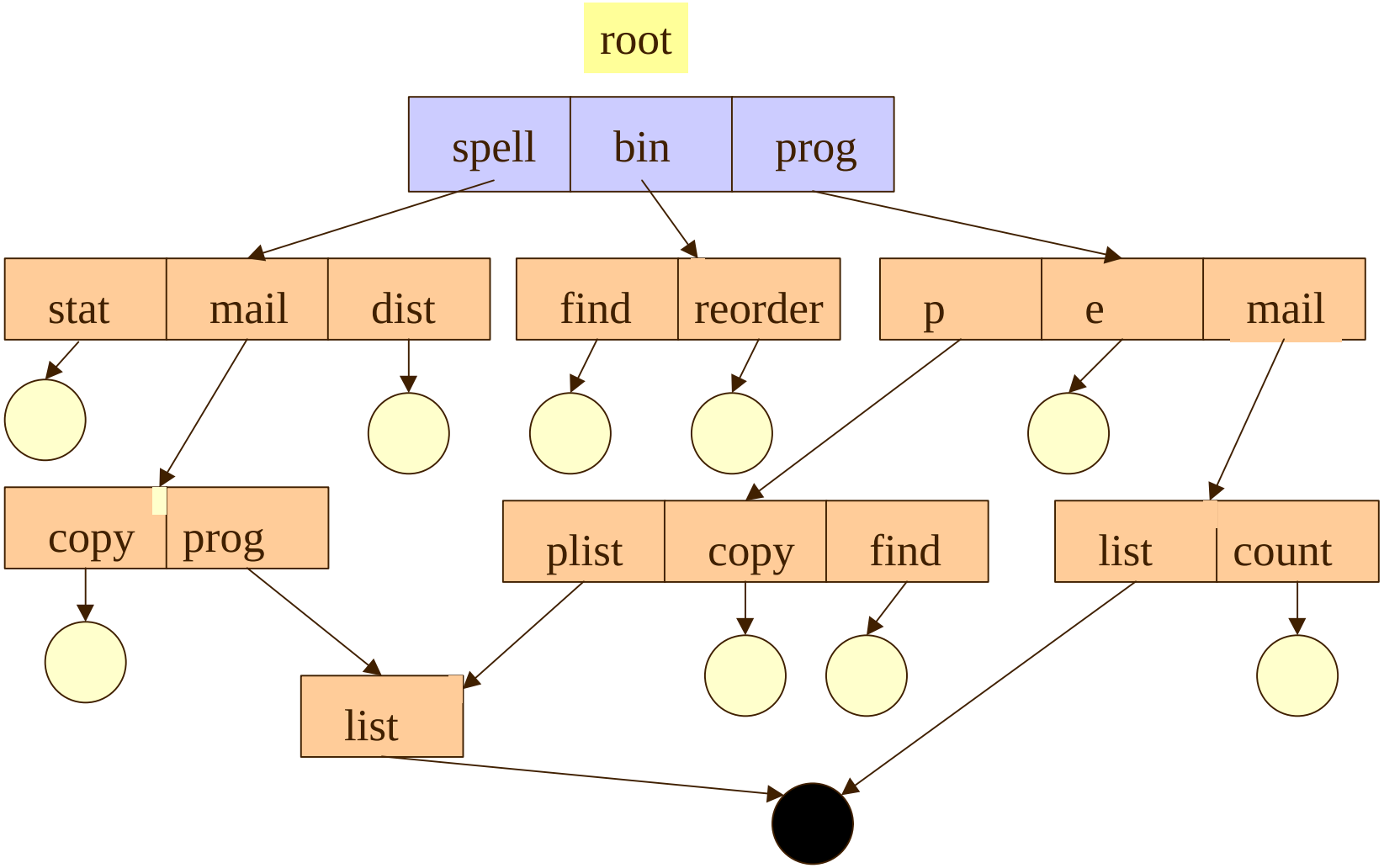
evtl de usuários diferentes

- ✓ **generalização** da estrutura em árvore
- ✓ implementação:
  - através de **links**
  - usando o comando `ln` na shell é possível fazer links para o mesmo arquivo de diretórios diferentes

ponteiro para o arquivo (path)

hardlinks e symlinks - serão vistos adiante

Diretório como grafo acíclico



Dispositivos

- ✓ dispositivos de bloco
- ✓ dispositivos de caracter

acesso direto ou seqüencial

- ✓ /dev
  - cada dispositivo é representado em `/dev` como um arquivo de dispositivo
    - quando se lê ou escreve em um arquivo de dispositivo, o dado vem de ou vai para o dispositivo que representa
  - normalmente arquivos de dispositivo existem embora o próprio dispositivo não esteja instalado
    - um arquivo, como /dev/sda, não significa que você realmente tenha um disco rígido de SCSI

arquivos em /dev

- ✓ nenhum programa especial é necessário para ter acesso a dispositivos
- ✓ por exemplo, enviar um arquivo à impressora:

```
$ cat filename > /dev/lp1
$
```

  - mas não é uma boa idéia ter várias pessoas enviando os arquivos para a impressora ao mesmo tempo
  - normalmente se usa `lpr`
    - este programa garante que só um arquivo está sendo impresso de cada vez

/dev

dispositivos aparecem como arquivos no diretório

- ✓ fácil ver quais arquivos de dispositivo existem
  - usar `ls`

a primeira coluna contém o tipo do arquivo
e suas permissões

```
$ ls -l /dev/cua0
crw-rw-rw-   1 root   uucp    5,  64 Nov 30  1993 /dev/cua0
$
```

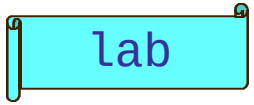
tipo do arquivo

  - `c` em `crw-rw-rw -`
    - c, um dispositivo de caracter.

      para arquivos regulares `

      para diretórios `

      para dispositivos de bloco `



3 arquivos em /dev

- ✓ 3 arquivos de dispositivos interessantes
  - ✓ /dev/console — único no sistema
    - console do sistema
    - mensagens de erro são enviadas para a console
  - ✓ /dev/tty — vários no sistema
    - alias para o terminal de controle de um processo (se
  - ✓ /dev/null — quando usado como entrada fornece EOF
    - toda a informação escrita é perdida

Mounting

montar um sistema de arquivo é análogo a abrir um arquivo

✓ mounting

- é dado um nome ao dispositivo

mount point

localização na estrutura de diretório para anexar o FS montado

é verificado se o dispositivo contém um FS válido

o FS é montado no *mount point*

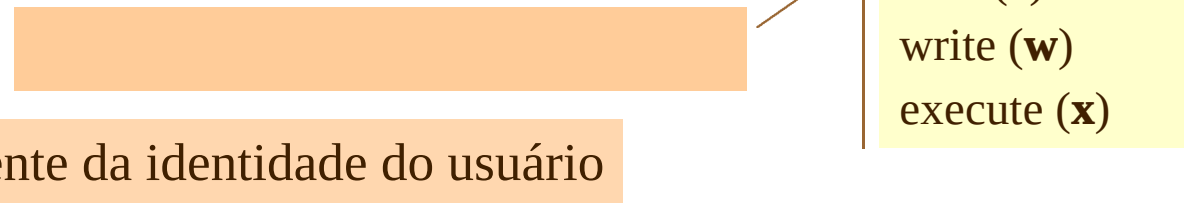
DOS e Windows não possuem *mounting*

você troca o disquete e o SO não percebe

Proteção de arquivos

em sistemas multiusuário

- **segurança** contra acesso impróprio
  - controle de acessos



- lista de acesso
  - associada a cada arquivo e/ou diretório
  - especifica o *user* e o tipo de acesso permitido
  - problema: criação e gerência da lista

Permissão de acesso

- lista de acesso para 3 tipos de usuários
  - owner — usuário que criou o arquivo
  - group
  - universe — todos os usuários do sistema

  group — usuários que compartilham o arquivo e necessitam de acessos similares
- exige rígido controle de *membership*

pelo administrador do sistema

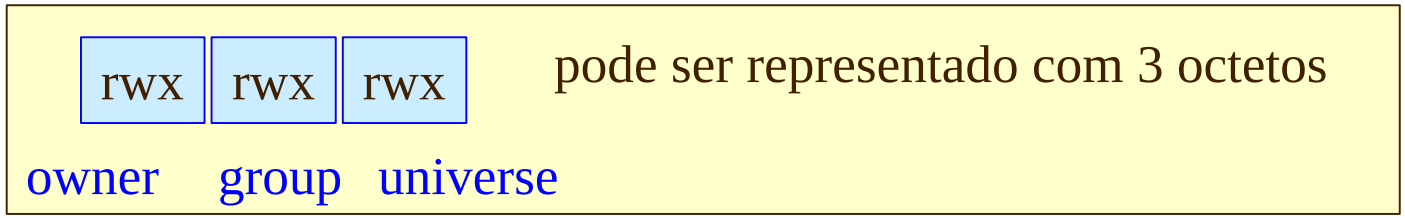
Permissões de acesso

UNIX: criação de grupo apenas por administrador

3 atributos para arquivo e diretórios: **rwx** para cada tipo de user (owner, group, universe)

total: nove (9) bits

`umask`



usar `ls -ls` para verificar as permissões de um dado arquivo

verificar comandos `chmod` e `umask`

`lab`

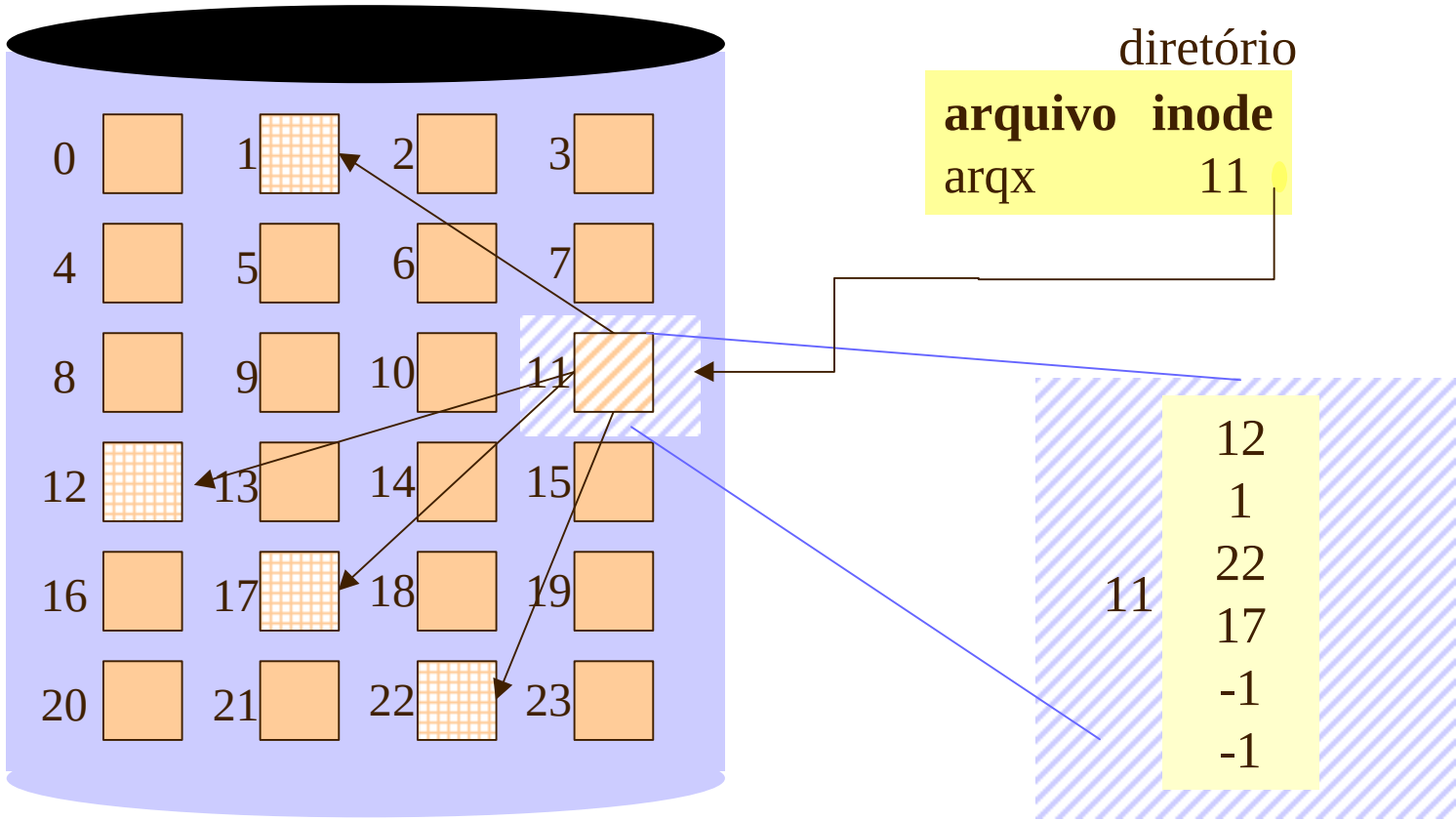
inode

index node ou **i**nformation node

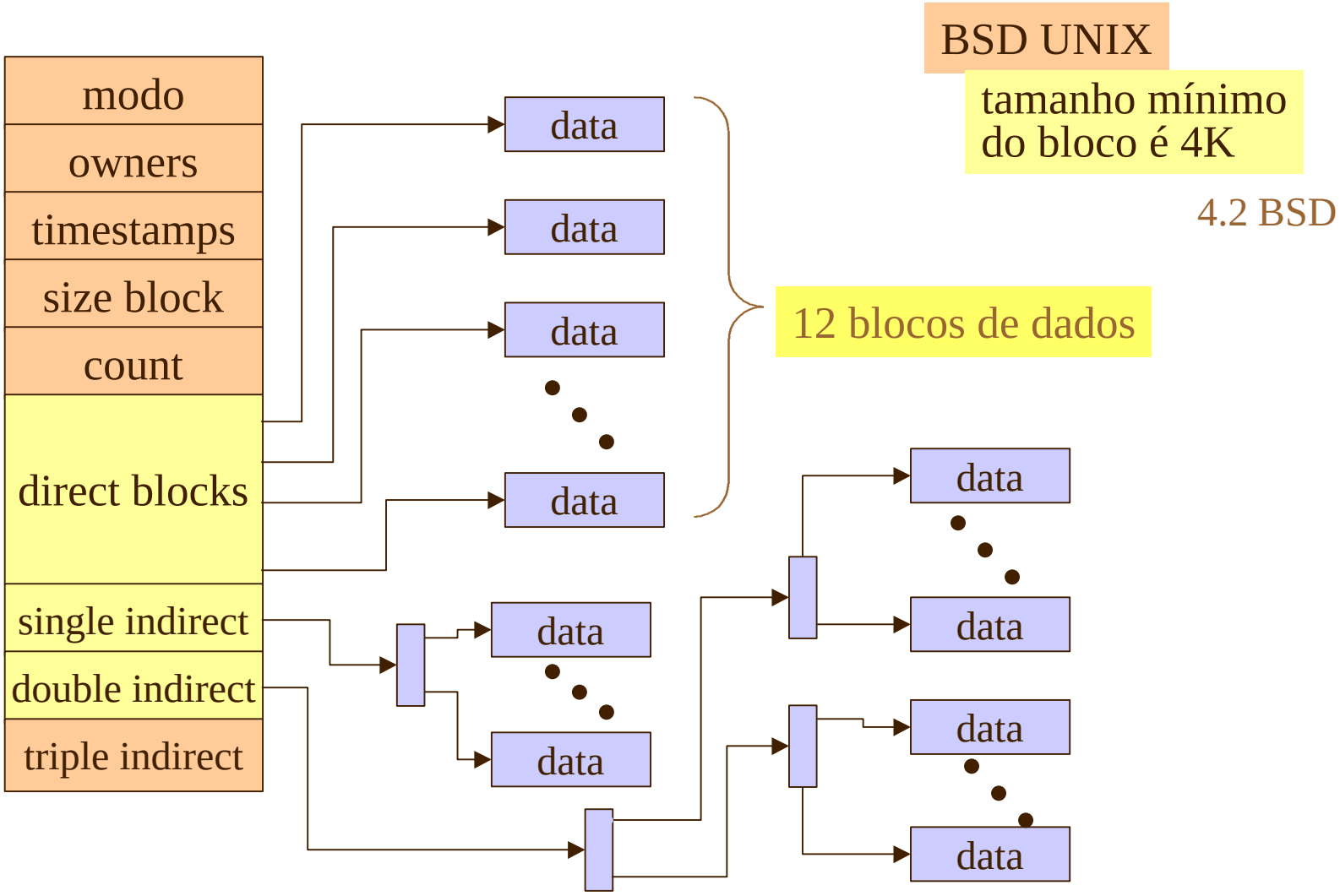
- ✓ bloco que contém ponteiros para todos os
  - localização dos blocos físicos (data
    acesso direto a blocos de um arquivo
  - mantém também outras informações sobre o arquivo
- ✓ cada arquivo tem o seu próprio *inode*
  - quando o arquivo é criado, todas as entradas do *index* contém ***nil***
- ✓ o inode é único por arquivo

verificar o número de inode de um arquivo
usando o comando `ln -i`

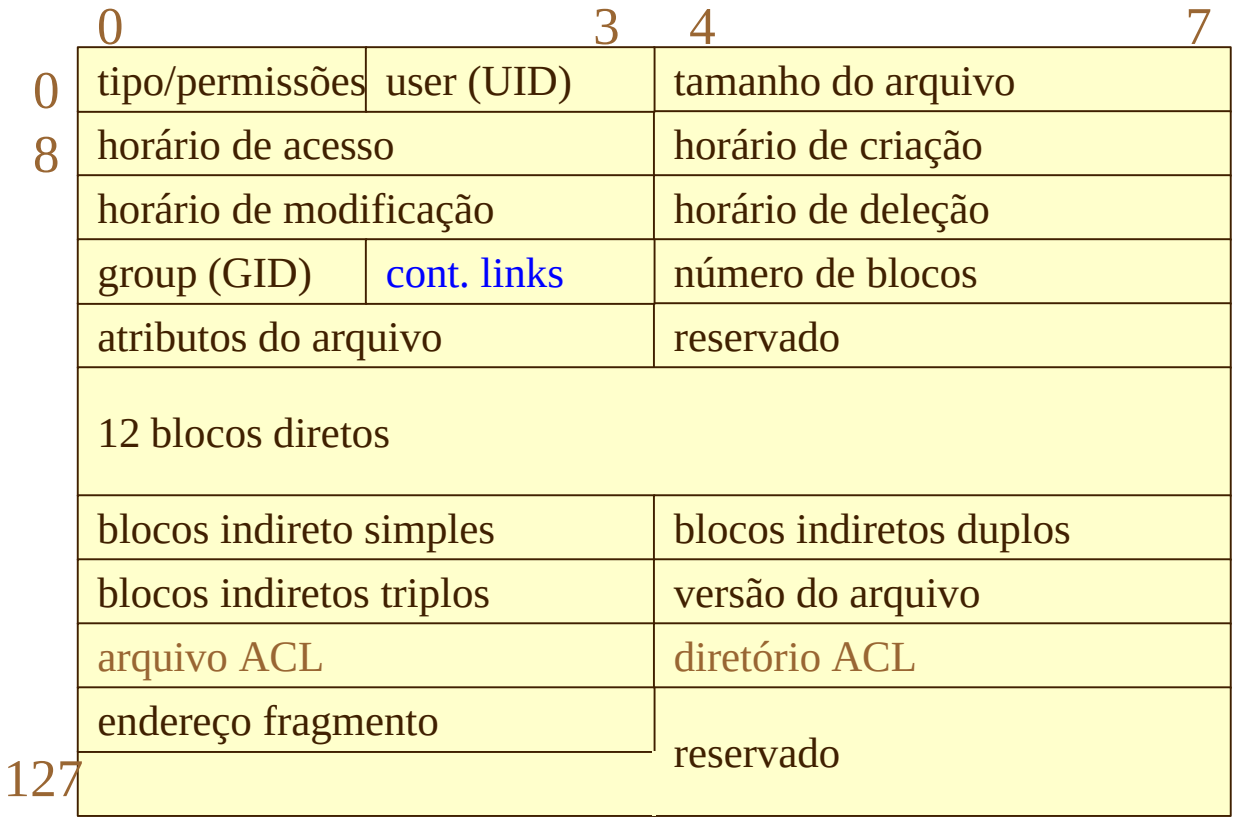
Alocação indexada



inode UNIX


modo
owners
timestamps
size block
count
direct blocks
single indirect
double indirect
triple indirect
data
data
data
data
data
data
data
data
data
data
BSD UNIX
tamanho mínimo
do bloco é 4K
4.2 BSD
12 blocos de dados

inode LINUX



ACL - acess control list

UNIX fs

- ✓ dois objetos principais
  - ✓ arquivos
  - ✓ diretórios
- ✓ data blocks
  - arquivos são compostos por
- ✓ arquivos e diretórios são representados por

diretório é apenas um caso especial de arquivo

conjunto de setores adjacentes

campo de tipo do inode
distingue entre arquivos e diretórios

Diretórios

- ✓ . e ..
  - primeiros dois nomes em cada diretório

link para o próprio diretório

link para o diretório pai

- ✓ *path name* e diretório atual

o usuário faz referência a um arquivo pelo seu **path name**
o sistema de arquivos usa o **inode**

- ✓ mount point

em qualquer diretório pode ser encontrado um **ponto de montagem** onde ocorre a mudança para outra estrutura de diretório

links

vários nomes podem representar o mesmo **inode**

- um arquivo pode ter vários nomes

um arquivo é representado univocamente pelo seu **inode**

depois de aberto, um arquivo é referenciado pelo seu **file descriptor** ou **stream**)

- hard links

entradas nos diretórios que apontam para um **inode**

- symbolic links

o campo de link identifica um

  - arquivos especiais
  - contém apenas o path name de um arquivo estabelecendo um link simbólico

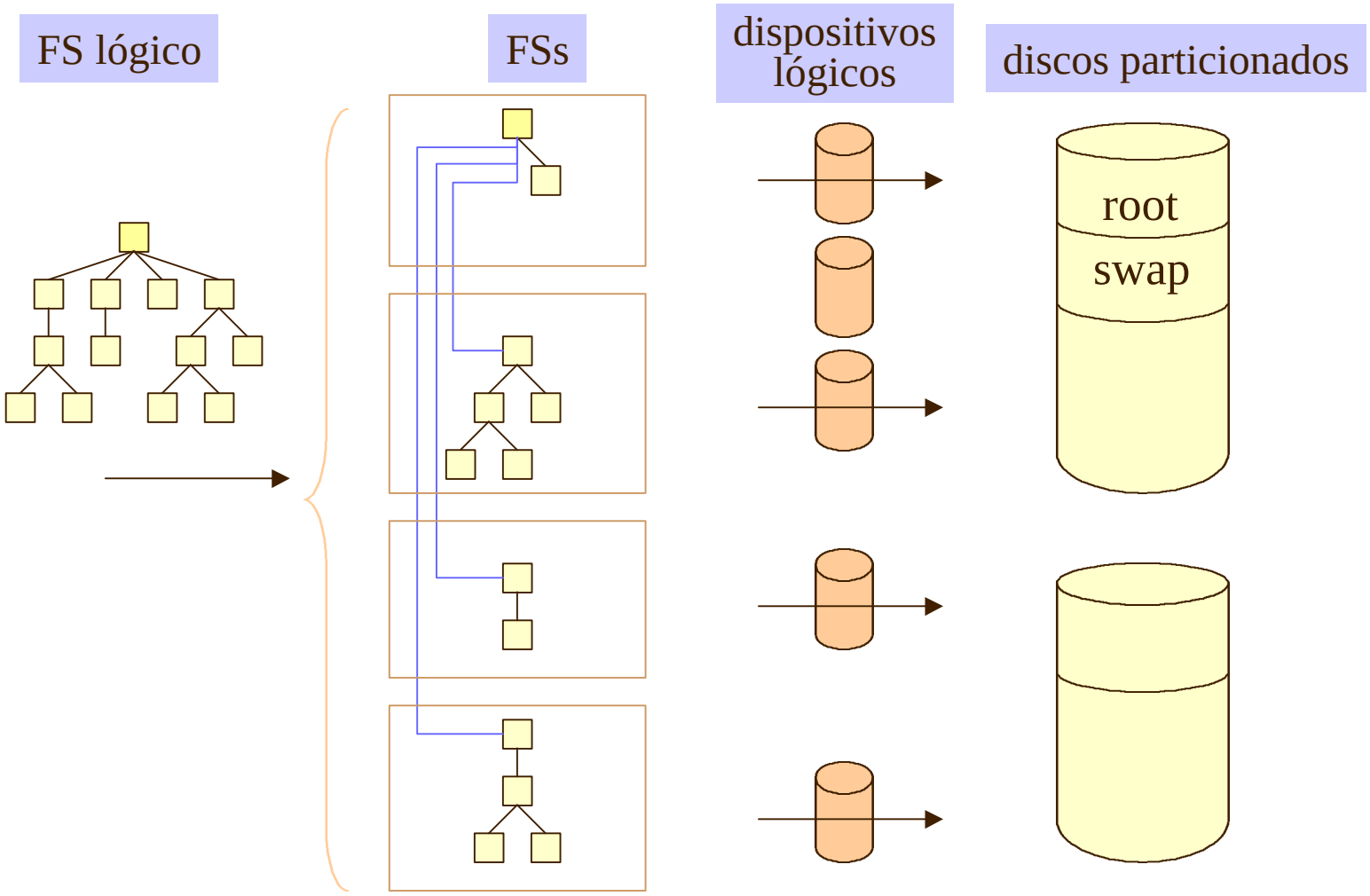
FS lógico versus FS físico

- FS lógico — o que o usuário conhece
- FS físico — dependente do dispositivo físico

um FS lógico pode ser composto por vários FS físicos

- dispositivos físicos — dispositivo de armazenamento
- dispositivos lógicos — partição de um dispositivo físico

FS lógico para dispositivo físico



Trabalhando com arquivos

✓ conceitos básicos sobre FS

- revisão
- sistema de arquivos UNIX
  - diretórios, arquivos e dispositivos (

✓ syscalls e funções

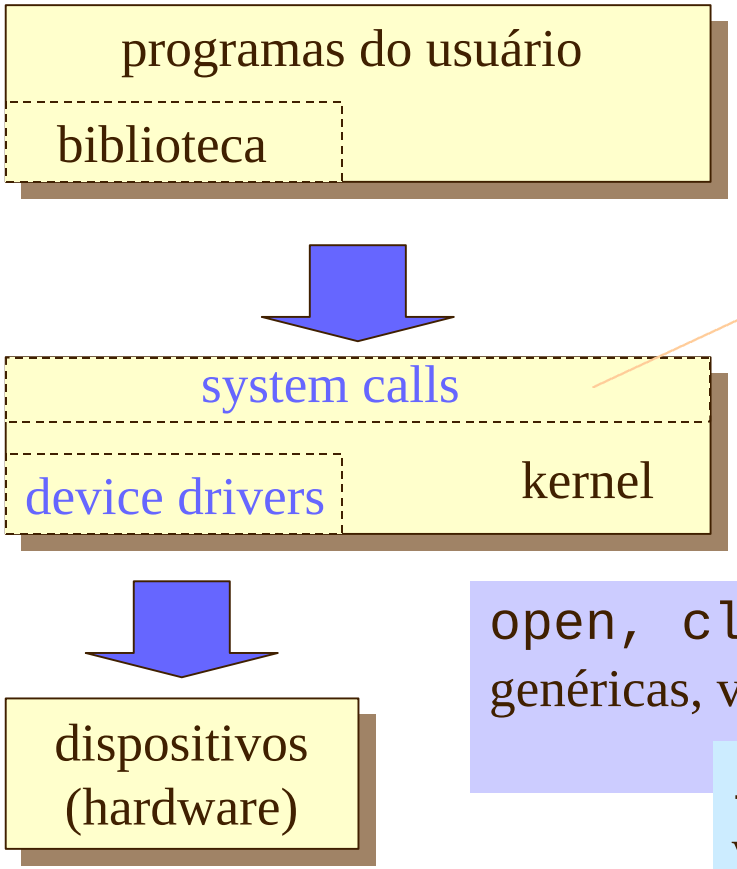
Matthew & Stones, cap 3

- chamadas de sistema (write, read, open, etc.),
- streams, chamadas a standard I/O (

tratamento de exceções (

chamadas de sistema avançadas (

System calls e drivers



espaço do usuário

open
close
read
write
ioctl

`open, close, read, write` são genéricas, valem para qualquer dispositivo

`ioctl` é específica, torna visível certas características peculiares do hardware

Syscalls

usar `man` pages section 2

- ✓ **read** ✓ ✓
- ✓ **write** ✓ ✓
- ✓ **open** ✓ ✓
- ✓ **close** ✓ ✓
- ✓ **ioctl** ✓ ✓

básicas

- ✓ trunc ✓
- ✓ lseek ✓
- ✓ dup ✓
- ✓ dup2 ✓

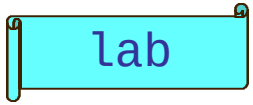
quase todas se encontram em `unist.h`

`lab`

Funções da biblioteca

- ✓ usar *system calls* diretamente pode ser ineficiente
  - complexidade de programação de baixo nível
    controle de tamanho de
- ✓ bibliotecas
  - uma forma eficiente, de mais alto nível
    geralmente bem documentadas
    include padrão

usar `man` pages section 3



Operações básicas

- ✓ operações básicas invocando system calls
  - mais baixo nível de operação
- ✓ na prática
  - ✓ o uso de syscalls e funções segue mais ou menos o mesmo padrão
    - arquivos precisam ser **abertos** antes de serem usados e precisam ser **fechados** no final
    - erros (exceções) são indicados de forma padrão
      - `errno`: variável global que indica erro

Descritores de arquivos

fd

- ✓ mapeamento de descritores para **inodes**
  - system call usa descritor como argumento
  - kernel usa descritor como índice na tabela de arquivos abertos do processo

    cada entrada aponta para uma estrutura, que aponta para o **inode** correspondente

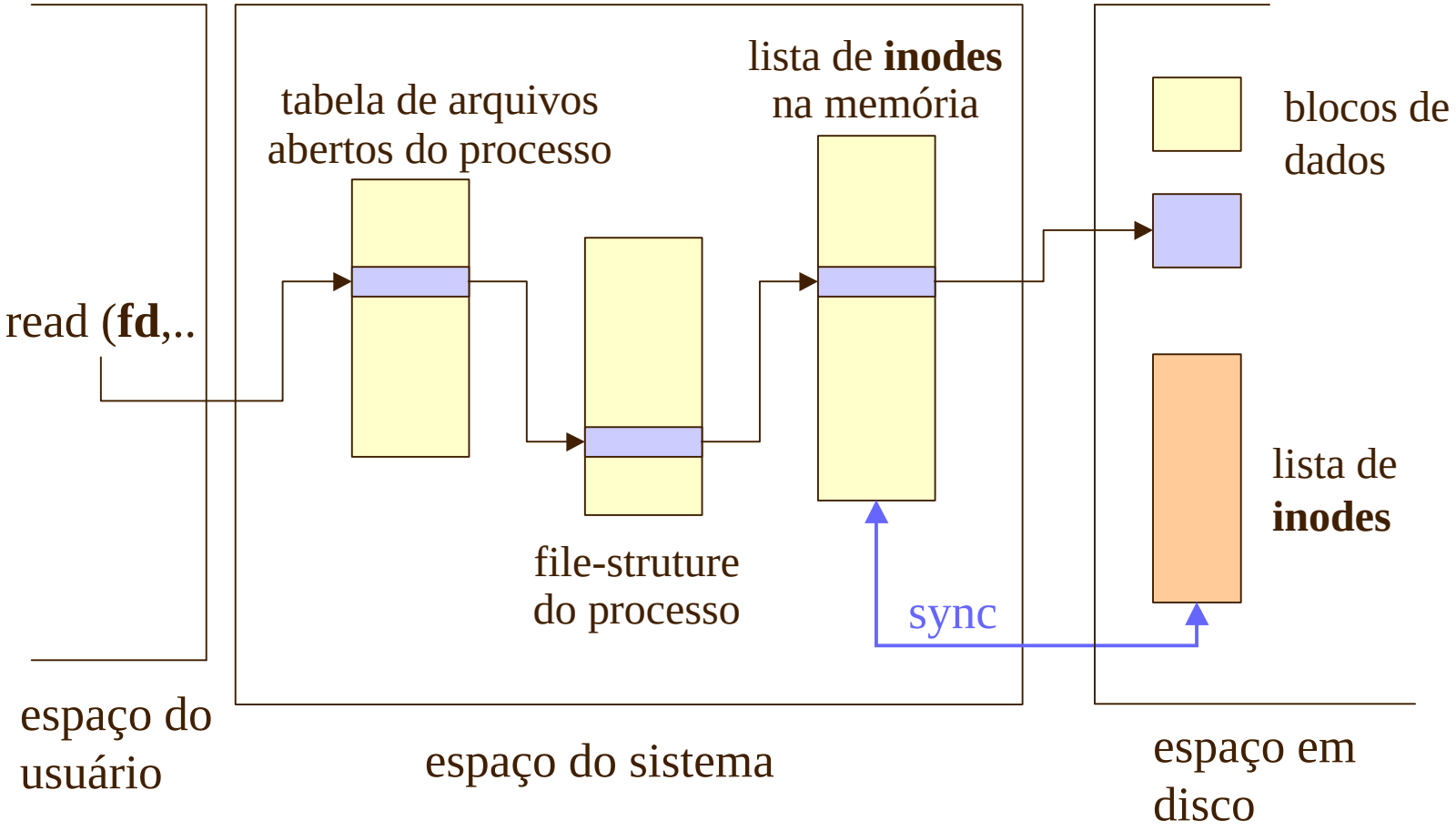
a tabela de arquivos abertos suporta um número fixo limitado de arquivos

  - quando um programa inicia, 3

    `fd0`, `fd1` e `fd2`

    stin, stout, stderr

Descritores & inodes



file-structure

- ✓ mantida pelo sistema

  uma estrutura por processo

  - não acessível aos aplicativos (programas no espaço

- ✓ guarda informações relativas ao processo que não podem ser mantidas no **inode**
  - por exemplo um arquivo aberto por dois processos
    - o **inode** é um só
    - cada processo usa um file-offset diferente para as operações de read e write sobre o arquivo

file-offset: posição relativa dentro do arquivo

Acesso a arquivos: write

- ✓ corresponde a syscall write

baixo nível

```
#include <unistd.h>

size_t write(int fd, const void *buf, size_t nbytes);
```

escreve os primeiros `nbytes` de `buf` no arquivo com descritor `fd`

retorna o número de bytes realmente escrito
se retornar 0 : nenhum dado foi escrito
se retornar -1 : erro está especificado na variável global `errno`

Lembrar que um programa inicia com 3 descritores já abertos. Outros arquivos devem ser abertos por `open`.

Exemplo: write simples

Matthew & Stones, cap 3

escreve uma mensagem em stout

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

int main()
{
    if ((write(1, "Here is some data\n", 18)) != 18)
       write(2, "write error on file descriptor 1\n",33);

    exit(0);
}
```

Note que se for escrito um número menor de bytes, isso não necessariamente é um erro. Um programa sério deve testar `errno`.

Para executar o programa o diretório atual deve estar no PATH ou o programa é invocado especificando o diretório

```
$ ./simple_write
```

lab

Acesso a arquivos: read

syscall read

```
#include <unistd.h>

size_t read(int fd, void *buf, size_t nbytes);
```

lê `nbytes` do arquivo com descritor `fd` na área de dados `buf`

retorna o número de bytes realmente lido
se retornar 0 : nenhum dado foi lido, EOF foi alcançado
se retornar -1 : erro está especificado na variável global `errno`

Exemplo: read simples

```
#include <unistd.h>

int main()
{
    char buffer[128];
    int nread;

    nread = read(0, buffer, 128);
    if (nread == -1)
        write(2, "A read error has occurred\n", 26);

    if ((write(1,buffer,nread)) != nread)
        write(2, "A write error has occurred\n",27);
```

lê uma mensagem de stin e escreve em stout

se a entrada possuir menos que 128 caracteres, o programa copia

lab

open

a abertura retorna um *índice* para uma tabela do SO (*open file table*)

índice = fd - descritor de arquivo

✓ syscall open

o fd é univoco para cada processo, mas dois processos que abriram o mesmo arquivo vão possuir cada um seu

no caso de escritas vai sobrepor a outra

dois processos, uma escrita

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

int open(const char *path, int oflags);
int open(const char *path, int oflags, mode_t mode);
```

} não necessários para Posix

nome do arquivo

O_CREAT

parâmetro `oflags` de open

✓ usado para especificar ações a serem tomadas

- or bit a bit
- modo de acesso e outros modos opcionais

usar `man` pages section 2

```
O_RDONLY  -  READ ONLY
O_WRONLY  -  WRITE ONLY
O_RDWR    -  READ WRITE
```

```
O_APPEND
O_TRUNC
O_CREAT
O_EXCL
```

com `O_CREAT` deve ser usada a forma de 3 `open`

`mode` está definido em `sys/stat.h`

Permissões iniciais para open

- `mode` com `O_CREAT`

`S_IRUSR` - read owner
`S_IWUSR` - write owner
`S_IXUSR` - execute owner
`S_IRGRP` - read group
`S_IWGRP` - write group
`S_IXGRP` - execute group
`S_IROTH` - read others
`S_IWOTH` - write others
`S_IXOTH` - execute others

read (**r**)
write (**w**)
execute (**x**)

owner
group
universe (others)

lembrar que esses flags são apenas um pedido de permissão na criação do arquivo; se a permissão será concedida ou não depende do valor de `umask` (*user mask - variável do sistema*)

exemplo:

```
open("arquivo",O_CREAT,S_IRUSR|S_IXOTH );
```

close

- syscall
  - termina a associação entre fd e o arquivo

```
#include <unistd.h>
int close(int fd);
```

o fd pode ser reusado

- o número de arquivos de um processo é limitado
  `limits.h` constante `OPEN_MAX`
- em sistemas POSIX no mínimo 16

ioctl

- ioctl - input/output control
- formato

```
#include <sys/ioctl.h>
int ioctl(int fd, int request, void * arg);
```

operação requerida

ponteiro para alguma coisa dependendo do tipo de arquivo e do tipo de operação

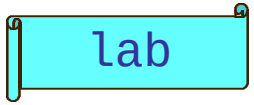
exemplo 1: cópia de arquivo

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

int main()
{
  char c;
  int in, out;

  in = open("file.in", O_RDONLY);
  out = open("file.out",O_WRONLY|O_CREAT,S_IRUSR|S_IWUSR);
  while(read(in,&c,1) == 1)
        write(out,&c,1);
```

copia um caracter de cada vez de

file.in é um arquivo existente
file.out não existe e será criado no programa

serão usados como descritores de arquivos



exemplo 2: cópia de arquivo

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

int main()
{
    char block[1024];
    int in, out;
    int nread;

    in = open("file.in", O_RDONLY);
    out = open("file.out", O_WRONLY|O_CREAT, S_IRUSR|S_IWUSR);
    while((nread = read(in,block,sizeof(block))) > 0)
        write(out,block,nread);
```

copia 1024 caracteres de cada vez

verificar a relação entre os tempos de execução dos dois programas de cópia

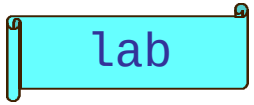
```
ls -ls file.in file.out
```

lab

Outras syscalls

lseek	- troca o *ptr*-posição de um *file descriptor*
fstat, stat, lstat	- obtém *status* do arquivo
dup	- duplica um fd (*file descriptor*)
dup2	- copia um fd (*file descriptor*) para outro

procurar o significado e os formatos das funções correspondentes



Biblioteca padrão - stdio

- ✓ stdio.h
  - ✓ parte do ANSI standard C
    - não são syscalls (syscall pertencem ao SO)
  - ✓ interface versátil para syscall de baixo nível
    - funções *sofisticadas* para formatação de saída e manipulação de entrada
  - ✓ stream
    - equivalente ao fd
    - ponteiro para estrutura (FILE *)

um programa inicia com 3 **streams** já abertos

`stin, stout, stderr`

Funções da stdio

- ✓ fopen, fclose
- ✓ fread, fwrite
- ✓ fflush
- ✓ fseek
- ✓ fgetc, getc, getchar
- ✓ fput, putc, putchar
- ✓ fgets, gets
- ✓ printf, fprintf, sprintf
- ✓ scanf, fscanf, sscanf

usam buffer
> tomar cuidado pois os dados não são escritos diretamente no meio físico

quase todas começam com f

} entrada e saída formatada

fopen

`stdio.h`

- ✓ usada para arquivos e terminais
  - para controlar explicitamente dispositivos > usar syscalls de baixo nível

syscalls não possuem efeitos secundários
bufferização por

```
#include < stdio.h>
FILE *fopen(const char *filename, const char mode);
```

associa um *stream*
com o arquivo

especifica como o

`mode` em `fopen`

parâmetro deve ser

- ✓ “r” ou “rb”
- ✓ “w” ou “wb”
- ✓ “a” ou “ab”
- ✓ “r+” ou “rb+” ou “r+b”
- ✓ “w+” ou “wb+” ou “w+b”
- ✓ “a+” ou “ab+” ou “a+b”

b indica que o arquivo é binário (não texto) - na verdade UNIX não faz diferença entre esses tipos e trata tudo como binário

fclose

`stdio.h`

- ✓ fecha o **stream** especificado
  - causa a escrita de todos os dados ainda não escritos
    - quando um programa acaba normalmente todos **streams** abertos são fechados
    - o programador fica sem chance de testar erros indicados `fclose`
  - existe um limite para o número de **streams** abertos por um programa (`FOPEN_MAX` em `stdio.h`)

```
#include < stdio.h>
int fclose(FILE *stream);
```

fread

`stdio.h`

- ✓ lê registros de dados de um **stream** para um buffer

```
#include < stdio.h>
size_t fread(void *ptr, size_t size, size_t nitems,
                FILE *stream);
```

registro de tamanho **size**

número de registros **nitems**

retorna número de registros lidos

ptr - aponta para buffer

fread potencialmente não portáveis para máquinas diferentes

fwrite

`stdio.h`

- ✓ escreve registros de dados em um **stream** a partir de um buffer dado por **ptr**

registro de tamanho **size**

número de registros **nitems**

retorna o número de registros escritos com sucesso

ptr - aponta para buffer

```
#include < stdio.h>
size_t fwrite(const void *ptr, size_t size,
              size_t nitems, FILE *stream);
```

fwrite potencialmente não portáveis para máquinas diferentes

fflush & fseek

`stdio.h`

```
#include < stdio.h>
int fflush(FILE *stream);
```

- ✓ **fflush**: descarga (escrita) imediata do buffer
  - assegura commit de operações em disco
  - **fclose** realiza flush automático quando invocado

```
#include < stdio.h>
int fseek(FILE *stream, long int offset, int whence);
```

- ✓ **fseek**: estabelece a posição no *stream* para o próximo *read* ou *write* nesse *stream*
  - semelhante a lseek

fgetc, getc, getchar

`stdio.h`

- ✓ obtém o próximo caracter do **stream**
  - retorna EOF quando alcança fim de arquivo ou um

```
#include < stdio.h>

int fgetc(FILE *stream);
int getc(FILE *stream);
int getchar;
```

pode ser implementada como macro

lê o próximo caracter de `stdin`, *standard input*

fput, putc, putchar

`stdio.h`

- ✓ escreve o caracter **c** no **stream** de saída
  - retorna valor escrito, EOF ou um erro

```
#include < stdio.h>

int fputc(int c,FILE *stream);
int putc(int c FILE *stream);
int putchar(int c);
```

pode ser implementada como macro

escreve o caracter **c** em `stdout`, *standard output*

fgets & gets

`stdio.h`

- ✓ obtém um *string* do **stream** de entrada
  - o *string* é armazenado na posição indicada **s**
  - término:
    - nova linha é encontrada
    - n-1 caracteres foram transferidos
    - fim do arquivo é alcançado

    } o que ocorrer primeiro
  - retorno:
    - um ponteiro para s, ou *null pointer* quando alcança fim de arquivo ou um erro

```
#include < stdio.h>
char *fgets (char *s, int n, FILE *stream);
char *gets (char *s);
```

Funções da stdio: entrada e saída formatada

- ✓ fopen, fclose
- ✓ fread, fwrite
- ✓ fflush
- ✓ fseek
- ✓ fgetc, getc, getchar
- ✓ fput, putc, putchar
- ✓ fgets, gets
- ✓ printf, fprintf, sprintf
- ✓ scanf, fscanf, sscanf

`stdio.h`

entrada e saída formatada
procurar o significado e os formatos dessas funções

`lab`

Outro exemplo de cópia de arquivo

- ✓ copiar caracter a caracter usando

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

int main()
{
    int c;
    FILE *in, *out;

    in = fopen("file.in","r");
    out = fopen("file.out","w");

    while((c = fgetc(in)) != EOF)
        fputc(c,out);

    exit(0);
}
```

experimente esse programa e determine o seu tempo de execução

compare com o tempo de execução dos demais programas de cópia já feitos

lab

Syscalls para gerência de arquivos e diretórios

- ✓ read
- ✓ write
- ✓ open
- ✓ close
- ✓ ioctl
- ✓ trunc
- ✓ lseek
- ✓ dup
- ✓ dup2

- ✓ **chmod**
- ✓ **chown**
- ✓ **link**
- ✓ **unlink**
- ✓ **symlink**
- ✓ **mkdir**
- ✓ **rmdir**
- ✓ **chdir**
- ✓ **getwd**

alteração de permissões de acesso a arquivos

links

gerência de diretórios

- ✓ opendir
- ✓ readdir
- ✓ closedir
- ✓ telldir
- ✓ seekdir

chmod & chown : alteração de permissões

syscall

✓ chmod

geralmente apenas o *superuser* ou o *owner* podem alterar permissões

- troca as permissões de acesso de um arquivo ou

olhar open

```
#include <sys/stat.h>
int chmod(const char *path, mode_t mode);
```

✓ chown

- para o *superuser* trocar o *owner* de um arquivo

```
#include <unistd.h>
int chown(const char *path, uid_t ownwer,gid_t group);
```

user identification - obtida com `getuid` syscall
group identification- obtida com `getgid` syscall

Revisão do conceito de link

- ✓ um arquivo pode ter vários nomes

  um arquivo é representado univocamente pelo seu **inode**

  vários nomes podem representar o mesmo **inode**

- ✓ hard links
  - entradas nos diretórios que apontam para um **inode**
- ✓ symbolic links
  - arquivos especiais
  - contém apenas o *path name* de um arquivo estabelecendo um link simbólico

link & symlink: cria links

syscall

```
#include <unistd.h>

int link(const char *path1, const char *path2);
int symlink(const char *path1, const char *path2);
```

- ✓ link
  - cria um novo link para um arquivo existente - `path1`
  - nova entrada do diretório especificada `path2`
- ✓ symlink
  - cria um novo link simbólico para um arquivo existente - `path1`
  - um link simbólico não evita que um arquivo seja

unlink: remove links

```
#include <unistd.h>
int unlink(const char *path);
```

comando `rm` usa essa syscall

- ✓ unlink
  - remove a entrada de diretório de um arquivo
    decrementa o contador de links no
    é necessária permissão de ecute para o diretório

a) o contador de links chegar a zero e
b) nenhum processo possuir o arquivo aberto

(quando o último processo fechar o arquivo, ele será
blocos de dados serão

mkdir & rmdir

syscall

- mkdir
  - permissões semelhantes a `O_CREAT` em open

```
#include <sys/stat.h>
int mkdir(const char *path, mode_t mode);
```

- rmdir
  - remove diretórios apenas se estão vazios

```
#include <sys/stat.h>
int rmdir(const char *path);
```

chdir & getcwd

syscall

✓ chdir

change diretory

- semelhante ao comando cd

```
#include <unistd.h>
int chdir(const char *path);
```

✓ getcwd

get current working diretory

- determina o diretório de trabalho atual

```
#include <unistd.h>
char *getcwd(char *buf, size_t size);
```

tamanho de buf

retorna nome do diretório em buf

Syscalls para varrer diretórios

- ✓ read
- ✓ write
- ✓ open
- ✓ close
- ✓ ioctl
- ✓ trunc
- ✓ lseek
- ✓ dup
- ✓ dup2
- ✓ chmod
- ✓ chown
- ✓ link
- ✓ unlink
- ✓ symlink
- ✓ mkdir
- ✓ rmdir
- ✓ chdir
- ✓ getwd

scanning diretories

- ✓ **opendir**
- ✓ **readdir**
- ✓ **closedir**
- ✓ **telldir**
- ✓ **seekdir**

determinar os arquivos que residem em um dado diretório

um **diretório** poderia ser tratado como um arquivo convencional, mas funções específicas facilitam a manipulação das entradas do diretório e ajudam na

Varrendo diretórios

syscall

- ✓ opendir

```
#include <sys/types.h>
#include <dirent.h>
DIR *opendir(const char *name);
```

dirent - diretory entry

- ✓ readdir

```
#include <sys/types.h>
#include <dirent.h>
struct dirent *readdir(DIR *dirp);
```

- ✓ closedir

```
#include <sys/types.h>
#include <dirent.h>
int closedir(DIR *dirp);
```

dirp - diretory stream entry pointer

- ✓ telldir

```
#include <sys/types.h>
#include <dirent.h>
long int telldir(DIR *dirp);
```

- ✓ seekdir

```
#include <sys/types.h>
#include <dirent.h>
void seekdir(DIR *dirp, long int loc);
```

`loc` - posição obtida através de `telldir`

opendir & closedir

- opendir

```
DIR *opendir(const char *name);
```

retorna um pointer para uma estrutura DIR.
DIR deve ser usado para ler as entradas do diretório

- closedir

```
int closedir(DIR *dirp);
```

fecha um diretório e libera recursos associados a ele

readdir

syscall

✓ readdir

```
struct dirent *readdir(DIR *dirp);
```

retorna um pointer para uma estrutura relativa a entrada no diretório

se outros processos estiverem criando e arquivos no mesmo diretório ao mesmo tempo (`readdir` não garante conseguir listar todas as entradas nesse diretório

- uma entrada no diretório identifica um arquivo

`dirent` inclui:

- o inode do arquivo `ini_t d_ino`
- o nome do arquivo `char d_name[]`

telldir & seekdir

syscall

```
long int telldir(DIR *dirp);
```

- ✓ telldir

retorna um valor que indica a posição atual no diretório (

- ✓ seekdir

```
void seekdir(DIR *dirp, long int loc);
```

entry pointer no diretório desejado para a posição dada
deve ter sido obtido através de `teeldir`

Exemplo: imprime diretório

Programa cria uma lista simples das entradas do *diretório home atual*.
Existe um limite no nível de
limite de diretórios que podem estar abertos (*streams*) no sistema.

```
/*  We start with the appropriate headers
                    which prints out the current
```

lab

imprime diretório: parte inicial

```
void printdir(char *dir, int depth)
{
    DIR *dp;
    struct dirent *entry;
    struct stat statbuf;

    if((dp = opendir(dir)) == NULL) {
        fprintf(stderr,"cannot open directory: %s\n", dir);
        return;
    }
    chdir(dir);
```

verificação de erro

troca para o diretório dado como parâmetro

lab

imprime diretório: laço

```
while((entry = readdir(dp)) != NULL) {
        lstat(entry->d_name,&statbuf);
        if(S_ISDIR(statbuf.st_mode)) {
            /* Found a directory, but ignore . and .. */
            if(strcmp(".",entry->d_name) == 0 ||
                strcmp("..",entry->d_name) == 0)
                continue;
            printf("%*s%s/\n",depth,"",entry->d_name);
            /* Recurse at a new indent level */
            printdir(entry->d_name,depth+4);
        }
        else printf("%*s%s\n",depth,"",entry->d_name);
    }
    chdir("..");
    closedir(dp);
}
```

lab

imprime diretório: *main*

```
/*  Now we move onto the main function.  */

int main()
{
    printf("Directory scan of /home:\n");
    printdir("/home",0);
    printf("done.\n");

    exit(0);
}
```

lab

Erros

olhar `errno.h` para definição das constantes

- variable `errno`
  - um programa deve testar `errno` imediatamente depois da chamada de uma função que pode ter

exemplos

`EPERM` -	operação não permitida
`ENOENT` -	não encontrado tal arquivo ou diretório
`EIO` -	erro de E/S
`EBUSY` -	dispositivo ou recurso ocupado
`EINVAL` -	argumento inválido
`EMFILE` -	excesso de arquivos abertos

Funções para reportar erros

- sterror

```
#include <sttring.h>
char *sterror(int errnum);
```

mapeia um (número de) erro em um string descrevendo o erro

- perror

```
#include <stdio.h>
void perror(const char *s);
```

mapeia o erro atual, reportado em `errno`, em um string descrevendo o erro e imprime o erro na para erro padrão.

`s` é um string que será impresso antes da mensagem de erro

fcntl & mmap

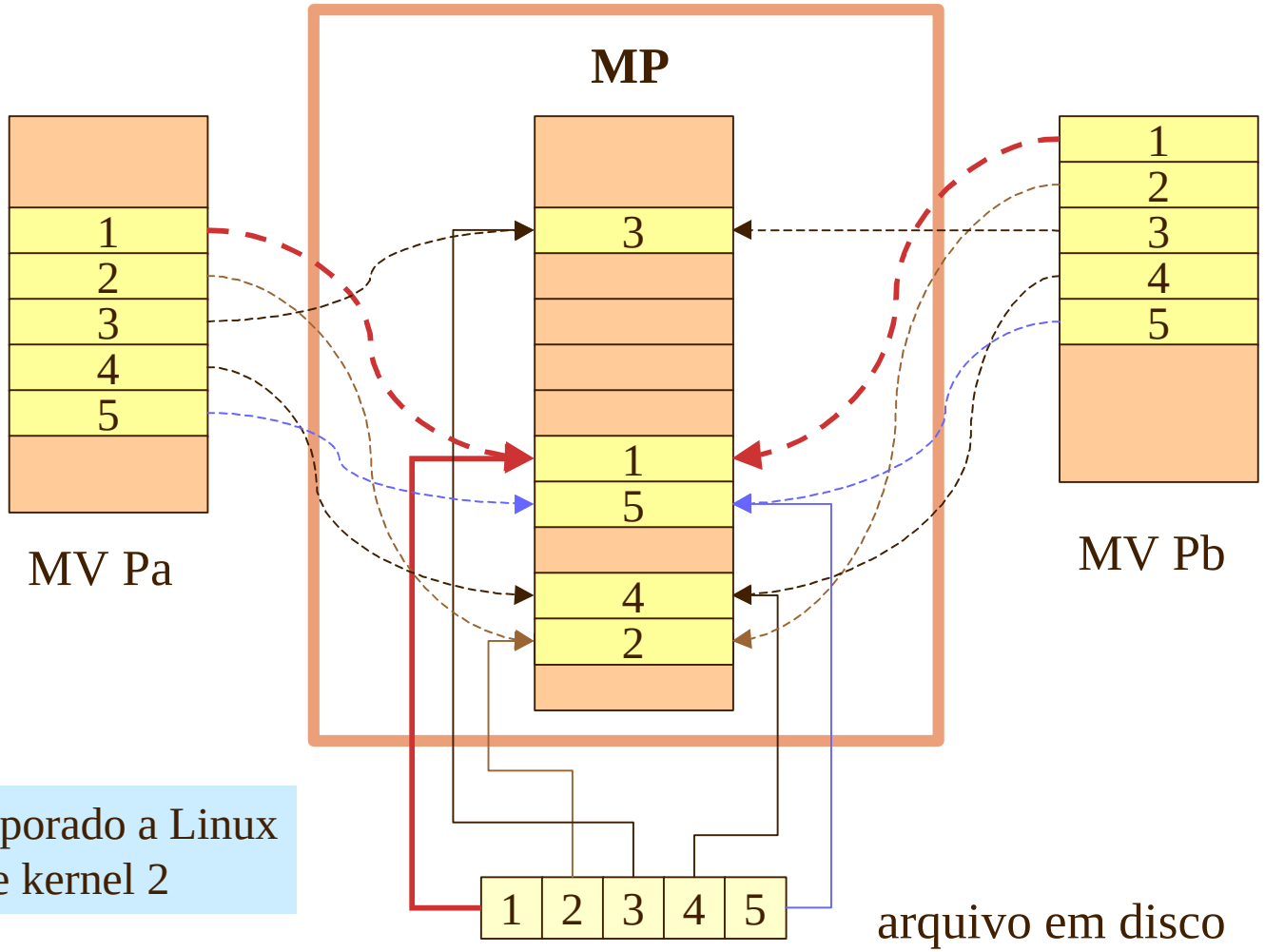
- ✓ funções raramente usadas
  - podem prover algumas facilidades para operações de

    : várias operações de controle

    - define um segmento de memória virtual que pode ser lido ou escrito por dois ou mais processos
    - permite que o conteúdo de um arquivo em disco pode ser manipulado como um array na memória
    - cria um ponteiro para uma região da
      permissões convenientes de acesso) associada ao conteúdo de um arquivo aberto

memory mapping

Operating System Concepts
Silberschatz & Galvin, 1998



incorporado a Linux
desde kernel 2

Fim ...

- ✓ de manipulação de arquivos Unix e Linux