

Sistema de Arquivos no Unix

Introdução (2)

- Os principais sistemas de arquivos usados para a formatação de discos locais em Linux são o *ext2*, *ext3*, *ext4*, *reiser*, *xfs* e *ufs*, entre outros.
- Os diretórios de um sistema de arquivos no UNIX têm uma estrutura pré-definida, com poucas variações
 - */home* : raiz dos diretórios home dos usuários.
 - */boot* : arquivos de boot (kernel do sistema, etc)
 - */var* : arquivos variáveis, áreas de spool (impressão, e-mail, news), arquivos de log
 - */etc* : arquivos de configuração dos serviços
 - */usr* : aplicações voltadas aos usuários
 - */tmp* : arquivos temporários
 - */mnt* : montagem de diretórios compartilhados temporários
 - */bin* : aplicações de base para o sistema
 - */dev* : arquivos de acesso aos dispositivos físicos e conexões de rede
 - */lib* : bibliotecas básicas do sistema
 - */proc* : não é um diretório real em disco, mas a porta de acesso para estruturas do núcleo

Introdução (1)

- Estruturados na forma de uma árvore única, iniciando pelo diretório */*, que é chamado de "raiz".
- Há suporte para arquivos, diretórios e links (atalhos).
- Os arquivos podem ter qualquer nome, usando quaisquer caracteres, com distinção entre maiúsculas e minúsculas. Os nomes são normalmente limitados a 255 caracteres.
- O caractere separador de diretórios é o */* (barra).
- Arquivos e diretórios cujos nomes começam com *.* (ponto) são considerados "ocultos" e normalmente não aparecem nas listagens de diretórios.
- As extensões são normalmente usadas apenas para facilitar a vida do usuário, mas não são importantes para o sistema operacional, que não depende delas para identificar o conteúdo de um arquivo.
- Os arquivos e diretórios possuem permissões de acesso controláveis por seus proprietários

Tipos de Arquivos

- **Arquivos normais**
 - seqüências de bytes: texto, binário, executável, etc.
- **Diretórios**
 - lista de outros arquivos (nome do arquivo e *inode*)
- **Arquivos especiais (*dispositivos*)**
 - interface entre o sistema e dispositivos de entrada e saída
 - Dispositivos orientados a caractere ou a bloco
- **Links**
 - Simbólicas (soft): ponteiro para outro arquivo
 - Concretos (hard): atribue mais um nome ao mesmo arquivo (na mesma partição)
- **Sockets e Pipes**
 - usados para comunicação entre processos (mecanismo para programação)

Atributos de Arquivos (1)

- Além do nome do arquivo, temos:
 - Tipo de arquivo
 - Ex: regular, diretório, PIPE, links simbólicos, arquivos especiais representando dispositivos
 - **Número de *hard links* apontando p/ o arquivo**
 - Tamanho (bytes)
 - Device ID
 - Número do i-node
 - Dentro de um mesmo device, um i-node (arquivo) tem um número único
 - UIDs e GIDs do proprietário
 - Timestamps (último acesso, última modificação e última modificação de atributos)
 - Permissões e *mode flags*

5

Descritores de Arquivos (1)

- Sempre que um processo quiser ler ou escrever em um arquivo, ele deve primeiramente abrir o arquivo (*open*)
- Quando um processo abre um arquivo, o kernel precisa criar várias estruturas de dados para gerenciar seu uso
- Duas estruturas de dados são importantes nesse contexto
 - Tabela de Descritores de Arquivos (file descriptor table)
 - Tabela de Arquivos do Sistema (system file table).
- Cada processo possui sua própria tabela de descritores de arquivos
- Os descritores de arquivos usados pelo processo são índices (inteiros) das entradas nessa tabela local

7

Atributos de Arquivos (2)

- Quando um arquivo é criado
 - UID: herdado do *effective* UID do processo criador
 - GID: depende...
 - SVR3: herda o *effective* GID do processo criador
 - BSD: herda o GID do diretório pai
- Permissões
 - read, write, execute
 - Acessos divididos por categorias: owner, group, others
- *Mode flags*
 - Arquivos executáveis
 - *suid*: quando um usuário executa um arquivo, *effective* UID do processo correspondente é setado para o UDI do owner deste arquivo
 - *sticky*: o kernel mantém o programa na área de swap (usado p/ arquivos executados frequentemente)

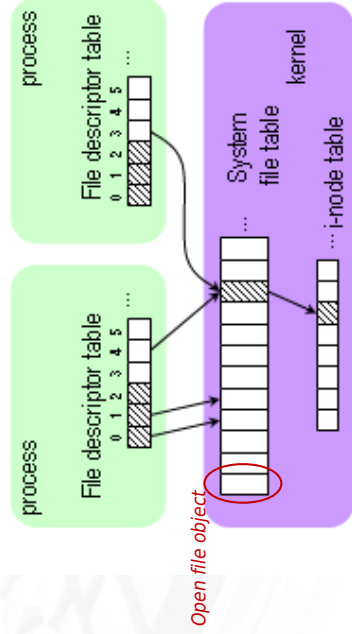
6

Descritores de Arquivos (2)

- A Tabela de Descritores de Arquivos
 - Parte do espaço de endereçamento do processo (program area)
 - Representa um array de ponteiros indexado pelos “file descriptors”
 - Os ponteiros apontam para entradas da Tabela de Arquivos do Sistema, Tabela de Arquivos do Sistema
 - Encontra-se no espaço do kernel (kernel area)
 - Contém uma entrada para cada arquivo aberto do sistema
 - Cada entrada contém um ponteiro para a Tabela de i-nodes mantida na memória
 - Cada entrada ainda contém outras informações como o **offset** corrente do arquivo, e um **contador** do número de descritores de arquivos que estão usando (apontando para) esta entrada
 - Quando um arquivo é fechado, o contador é decrementado
 - A entrada é liberada quando o contador chega a 0
- Tabela de i-nodes na memória
 - Contém cópias de i-nodes que estão atualmente sendo acessados

8

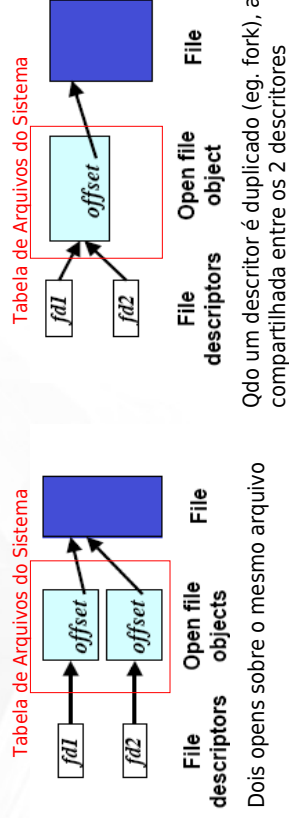
Descritores de Arquivos (3)



- Alguns descritores de arquivos têm um significado especial:
 - 0 é STDIN; 1 é STDOUT; 2 é STDERR

Descritores de Arquivos (5)

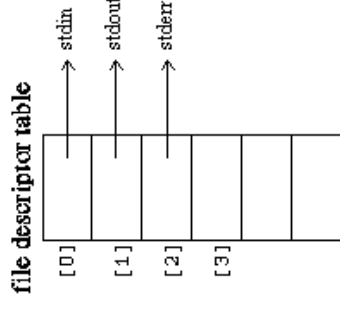
- Diferentes descritores de arquivos (pertencendo ou não a um mesmo processo) podem apontar para um mesmo arquivo aberto
- Um descritor aponta para um *Open file object* da Tabela de Arquivos do Sistema.
- Cada *Open file object* representa uma sessão independente de acesso a um arquivo.
- O *open file object* associado ao descritor contém o contexto desta sessão, como o modo em que o arquivo foi aberto e o *offset* em que a próxima leitura ou escrita deve ocorrer



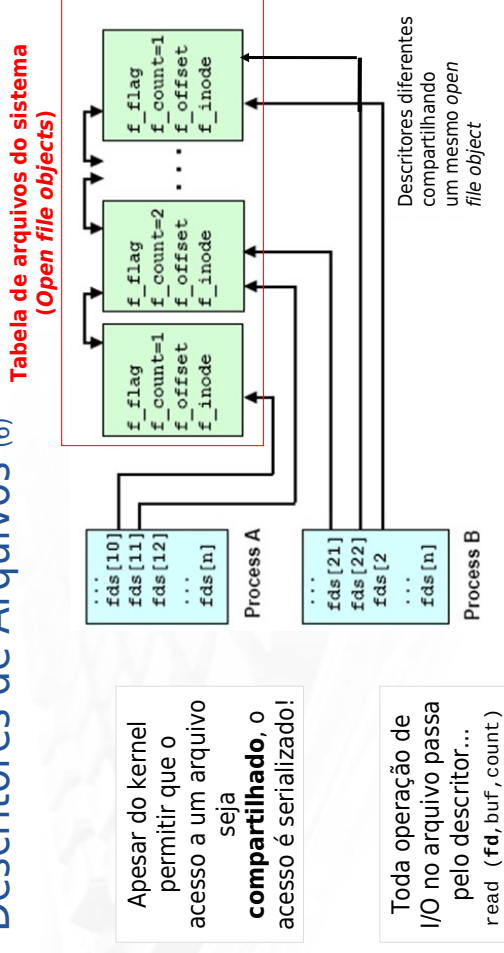
Quando um descritor é duplicado (eg. fork), a sessão é compartilhada entre os 2 descritores

Descritores de Arquivos (4)

- Cada processo sempre possui três descritores de arquivos pré-definidos, os chamados arquivos padrão, geralmente definidos no arquivo `unistd.h`
 - STDIN_FILENO (stream stdin)
 - Entrada padrão (default: teclado)
 - Usado por todas as funções de entrada de dados que não especificarem um descritor de arquivo.
 - STDOUT_FILENO (stream stdout)
 - Saída padrão (default: terminal)
 - Usado por todas as funções de saída de dados que não especificarem um descritor de arquivo
 - STDERR_FILENO (stream stderr)
 - Saída de erro (default: terminal)



Descritores de Arquivos (6)



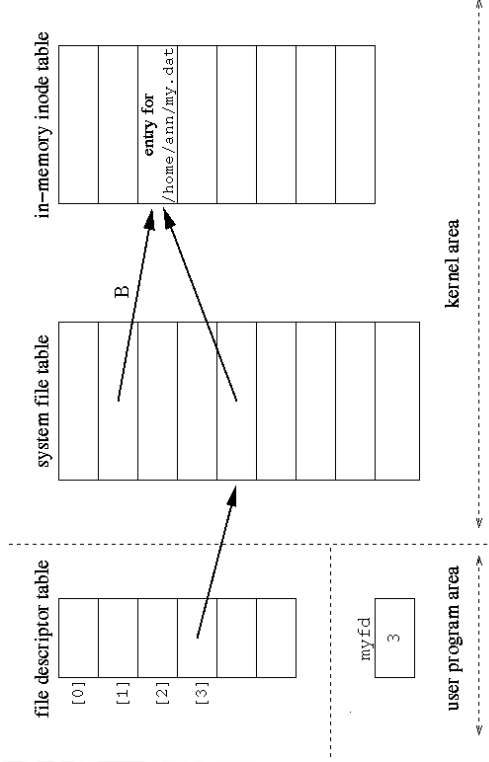
Apesar do kernel permitir que o acesso a um arquivo seja **compartilhado**, o acesso é serializado!

Toda operação de I/O no arquivo passa pelo descritor... read (fd, buf, count)

Descritores diferentes compartilhando um mesmo open file object

Descritores de Arquivos (7)

- Relação entre Tabela de descritores de arquivos, Tabela de arquivos abertos do sistema, Tabela de i-nodes na memória

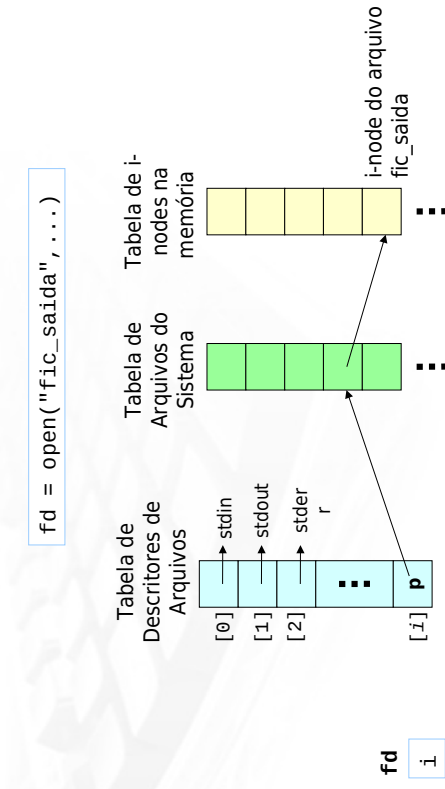


14

Redirecionamento de entrada/saída (1)

- Primitivas `dup2()`
 - `#include <unistd.h>`
 - `int dup2(int oldfd, int newfd);`
 - Valor de retorno: novo descritor de arquivo ou `-1` em caso de erro
- Esta primitiva cria uma cópia de um descritor de arquivo existente (`oldfd`) e fornece um novo descritor (`newfd`) tendo exatamente as mesmas características que aquele passado como argumento

Redirecionamento de entrada/saída (3)



16

Redirecionamento de entrada/saída (2)

- Este programa executa o comando shell 'ps', depois redireciona o resultado para o arquivo `fic_saida`

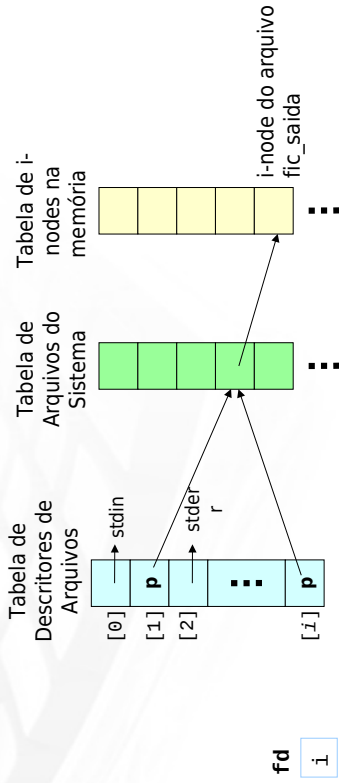
```
#include <errno.h>
#include <stdio.h>
#include <sys/types.h>
#include <fcntl.h>
#include <unistd.h>
#include <stdlib.h>

#define STDOUT 1

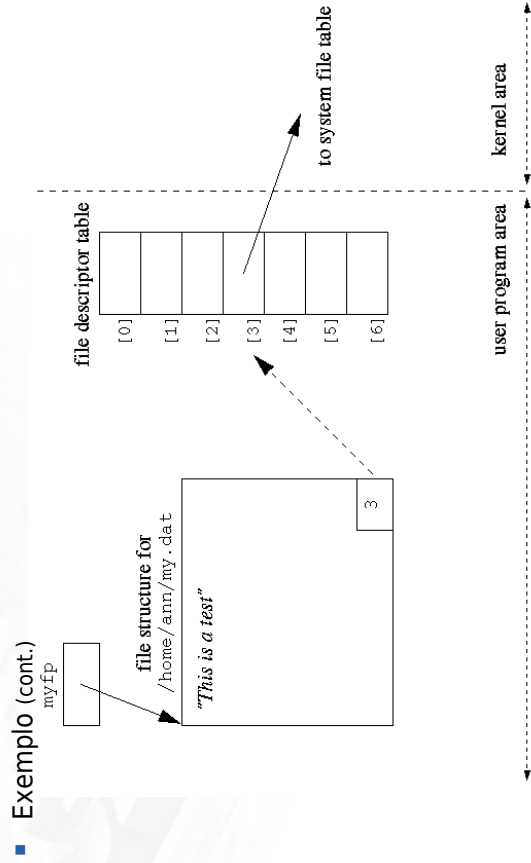
int main()
{
    int fd;
    /* associa fic_saida ao descritor fd */
    if ((fd = open("fic_saida", O_CREAT|O_WRONLY| O_TRUNC, 0666)) == -1) {
        perror("Error na abertura de fic_saida");
        exit(1);
    }
    dup2(fd, STDOUT); /* redireciona a saída padrao */
    system("ps"); /* executa o comando */
    exit(0);
}
```

Redirecionamento de entrada/saída (4)

```
dup2(fd, STDOUT) ; /* redireciona a saída padrão */
```



Ponteiros de Arquivos e Buffering (2)



Ponteiros de Arquivos e Buffering (1)

- O padrão ISO C usa ponteiros de arquivos (**file pointers**) ao invés de descritores de arquivos diretamente para manipular I/O
- Um *file pointer* aponta para uma estrutura de dados chamada **FILE structure** no espaço de endereçamento do processo
 - Ela contém um buffer e um descritor de arquivo
- Exemplo:

```
FILE *myfp;

if ((myfp = fopen("/home/ann/my.dat", "w")) == NULL)
    perror("Failed to open /home/ann/my.dat");
else
    fprintf(myfp, "This is a test");
```

Ponteiros de Arquivos e Buffering (3)

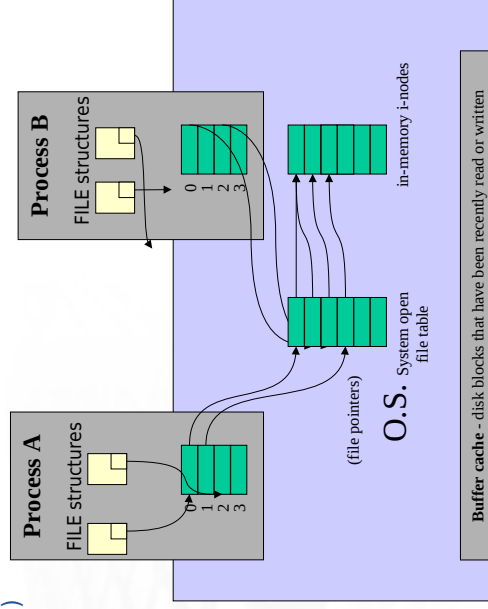
- I/O utilizando ponteiros de arquivos realizam leituras e escritas no buffer
- O buffer é preenchido e esvaziado quando necessário
 - Qdo o buffer é completamente preenchido, o subsistema de I/O realiza a escrita no disco (**e o buffer cache???**)
 - Uma escrita no buffer que o preencha parcialmente não causa uma escrita no disco.
- O tamanho do buffer pode variar.
- Se após uma escrita realizada na saída padrão o processo falha, o dado pode não aparecer no monitor
- A stderr não é bufferizado.
- É possível forçar a escrita no disco (fflush)

Buffer Cache (1)

- O sistema de arquivos mantém um buffer cache
- O buffer cache é armazenado em memória física (não paginada)
- Ele é usado para armazenar qualquer dado lido ou escrito em um block-device (e.g. disco, CD-ROM, DVD).
- Se um dado ã está no buffer cache:
 - O sistema aloca um buffer livre no buffer cache
 - Lê o dado do disco
 - Armazena o dado no buffer
- Se não houver nenhum buffer livre:
 - o sistema seleciona um buffer em uso
 - escreve seus dados no disco
 - marca o buffer como livre
 - aloca o buffer para o processo que solicitou

21

Buffer Cache (3)



23

Buffer Cache (2)

- Sincronizando buffers

```
#include <unistd.h>

int sync (void)
// Força que os buffers de todos os descritores do sistema sejam escritos
// em disco.

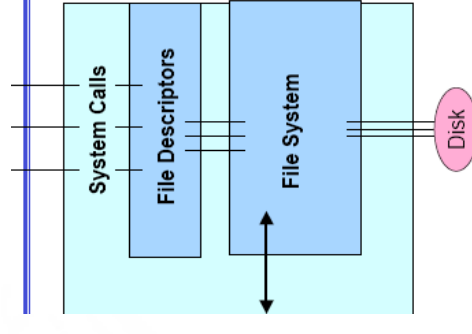
int fsync (int fildes)
// Aguarda até que todo o buffer de escrita associado a fildes seja escrito
// no disco. Isso inclui os dados e meta-dados do arquivo.

int fdatsync (int fildes)
// Aguarda até que todos os dados do arquivo tenham sido escritos em disco.
// Meta-dados não são considerados, portanto esta operação é mais rápida
// que fsync.
```

22

Virtual File System - Motivação

- Existem muitos tipos distintos de Sistemas de Arquivos
 - Necessidade de interoperabilidade
- Interface padrão para acesso a arquivos
 - Programar orientado para um FS específico seria inviável
- Kernel precisa ser flexível para adição de novos sistemas de arquivos

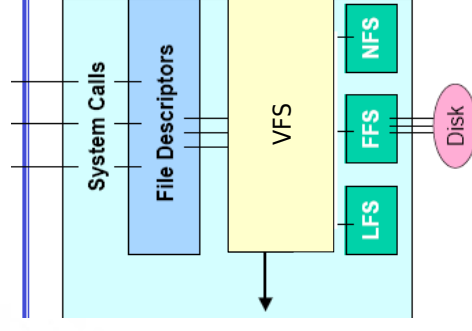


24

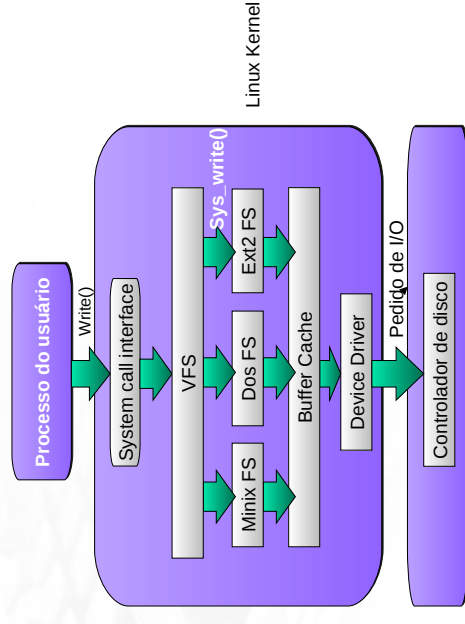
Virtual File System - Descrição

- Kernel implementa camada de abstração ao redor das interfaces FS de baixo-nível
- Esta camada define interfaces conceituais básicas que existem em qualquer FS
 - Common File Model
- A cada sistema de arquivos montado corresponde uma estrutura VFS no núcleo
- Mapeamento entre representação em VFS para o sistema de arquivos real

25

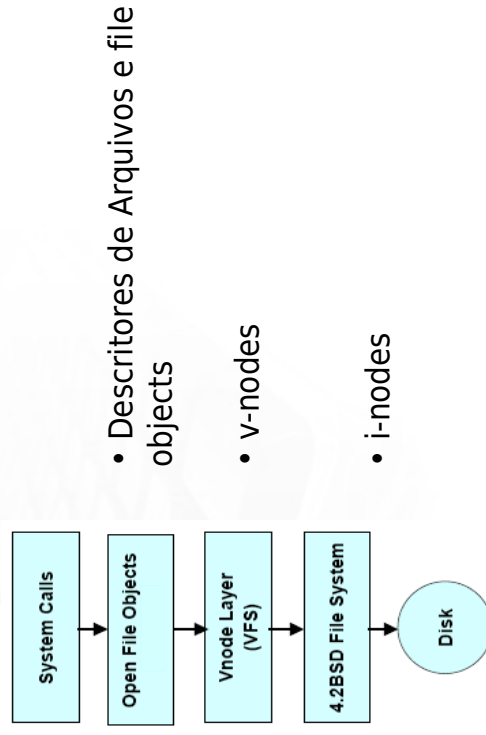


Virtual File System - Exemplo no Linux



26

Virtual File System - Estrutura de Dados



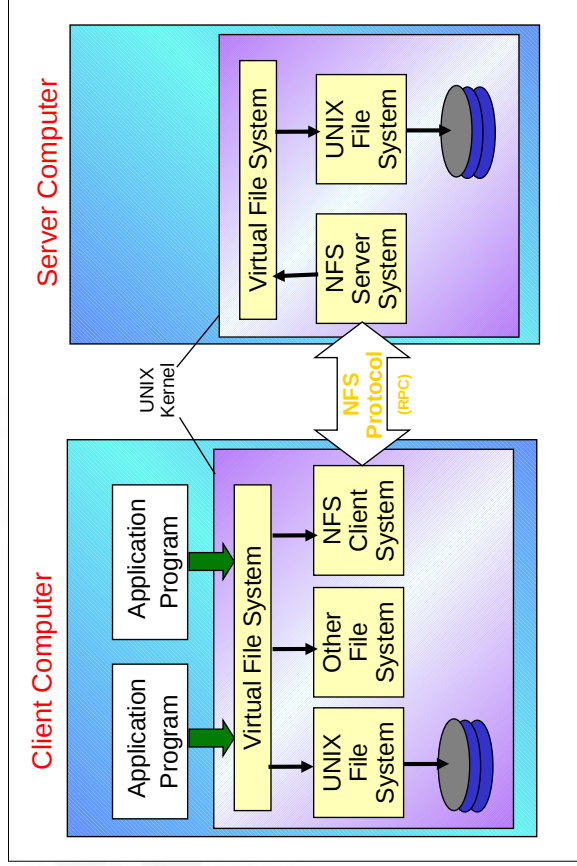
- Descritores de Arquivos e file objects
- v-nodes
- i-nodes

27

Sistema de Arquivos Distribuídos

- NFS (Network File System)
- Permite o acesso aos arquivos independentemente de sua localização física
 - O funcionamento do NFS é centrado no VFS
- Gerencia os *file servers* e a comunicação entre os *file servers*
- Coleção de clientes e servidores compartilham um sistema de arquivos
 - servidores exportam os diretórios
 - clientes, pela operação de montagem, ganham acesso aos arquivos.

28

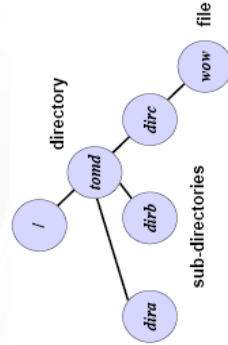
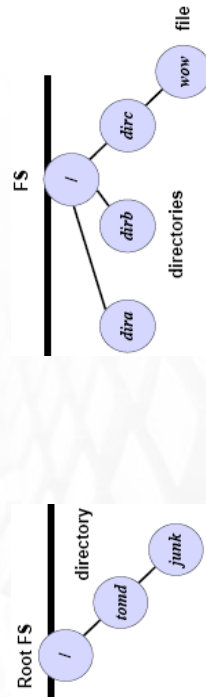


Montagem (Mounting) (1)

- O UNIX é um sistema que suporta “montagem”
- Existe um sistema de arquivos configurado para ser o *root file system*, e o seu diretório raiz se torna o diretório raiz do sistema
- Os demais sistemas de arquivos são integrados através do mecanismo de montagem
 - Cada sistema de arquivo é mapeado para um diretório
 - Este diretório é chamado de “mounting point”
- Com esse mecanismo cria-se um namespace comum
- Sistemas de arquivos removíveis (i.e. discos) ou remotos podem ser montados na árvore de arquivos como forma de integrar o sistema de arquivos
 - O diretório `/dev` contém os nomes de cada arquivo especial de dispositivo
 - Ex: `mount /dev/fd0 /mnt/floppy`

30

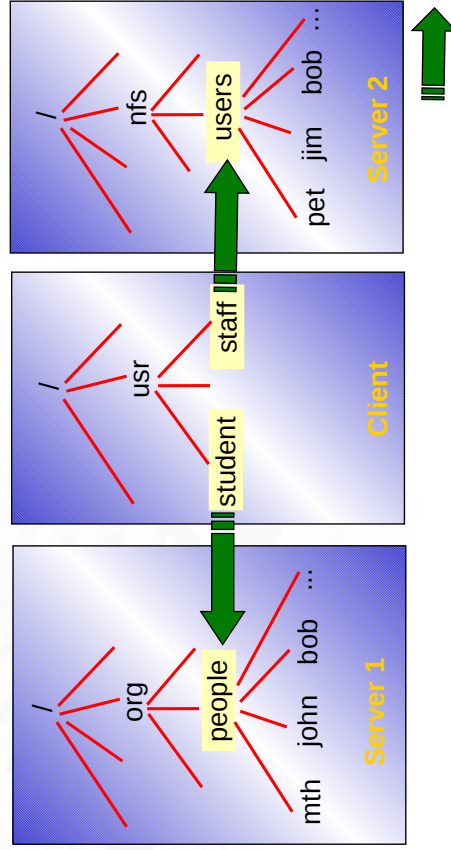
Montagem (Mounting) (2)



Qualquer acesso ao diretório “mounted-on” é traduzido em um acesso ao raiz do sistema de arquivos montado.

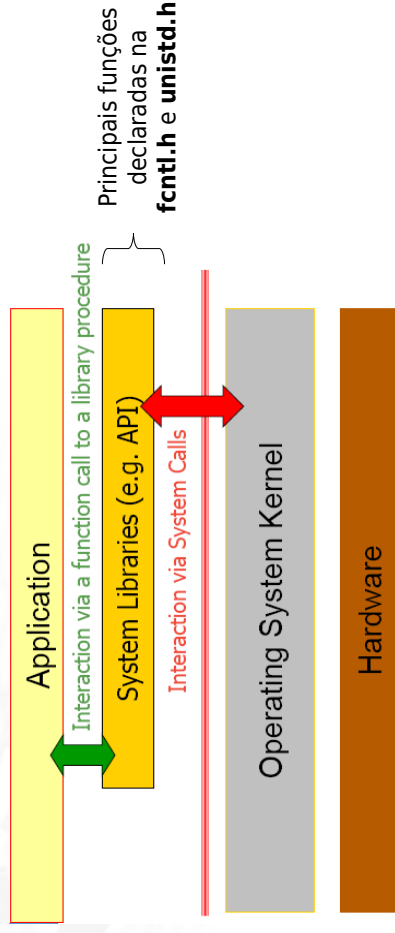
31

Montagem (Mounting) (3)



32

System Calls



33

System Calls – Manipulando Diretórios (2)

- Exemplo: programa que exibe o diretório corrente

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

int main(void) {
    long maxpath;
    char *mycwdp;

    if ((maxpath = pathconf(".", _PC_PATH_MAX)) == -1) {
        perror("Failed to determine the pathname length");
        return 1;
    }
    if ((mycwdp = (char *) malloc(maxpath)) == NULL) {
        perror("Failed to allocate space for pathname");
        return 1;
    }
    if (getcwd(mycwdp, maxpath) == NULL) {
        perror("Failed to get current working directory");
        return 1;
    }
    printf("Current working directory: %s\n", mycwdp);
    return 0;
}
```

35

System Calls – Manipulando Diretórios (1)

- Navegação em diretórios

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

int chdir (const char *path)
// Retorna 0 ou -1 (ex: errno = EACCES ou ENOTDIR)

char *getcwd (char *buf, size_t size)
// Retorna path do dir. corrente

long fpathconf (int filedes, int name);
// Obtém valor da opção de conf. name do descritor de arquivo filedes

long pathconf (char *path, int name);
// Obtém valor da opção de conf. name do arquivo path
```

```
_PC_PATH_MAX: comprimento máximo de um caminho relativo de diretório
_PC_NAME_MAX: comprimento máximo de um nome de arquivo no diretório
_PC_MAX_CANON: comprimento máximo de uma linha de entrada formatada de terminal
```

34

System Calls – Manipulando Diretórios (3)

- Acesso a diretórios

```
#include <dirent.h>

DIR *opendir(const char *filename);
// Retorna um handle para um stream de diretório (seq. ordenada das entradas)

struct dirent *readdir(DIR *dirp);
// Lê uma entrada da stream de diretório, move a posição do stream p/
// a próxima entrada

void rewinddir(DIR *dirp);
// Reseta a posição do stream de diretório para o início

int closedir(DIR *dirp);
// fecha o stream de diretório associado a dirp
```

36

System Calls – Manipulando Diretórios (4)

- Exemplo:

programa que lista os arquivos em um diretório

```
#include <dirent.h>
#include <errno.h>
#include <stdio.h>

int main(int argc, char *argv[]) {
    struct dirent *direntp;
    DIR *dirp;

    if (argc != 2) {
        fprintf(stderr, "Usage: %s directory_name\n", argv[0]);
        return 1;
    }

    if ((dirp = opendir(argv[1])) == NULL) {
        perror("Failed to open directory");
        return 1;
    }

    while ((direntp = readdir(dirp)) != NULL)
        printf("%s\n", direntp->d_name);
    while ((closedir(dirp) == -1) && (errno == EINTR)) ;
    return 0;
}
```

37

System Calls – Informações sobre Arquivos (2)

- Determinando o tipo de arquivo
 - O campo `st_mode` codifica modo de acesso e tipo de arquivo
 - Pode-se usar macros para “interpretar” esta codificação

S_ISBLK(m)	block special file
S_ISCHR(m)	character special file
S_ISDIR(m)	directory
S_ISFIFO(m)	pipe or FIFO special file
S_ISLNK(m)	symbolic link
S_ISREG(m)	regular file
S_ISSOCK(m)	socket

39

System Calls – Informações sobre Arquivos (1)

- Funções que retornam informações sobre o arquivo especificado

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

int stat(const char *restrict path, struct stat *restrict buf);
int lstat(const char *restrict path, struct stat *restrict buf);
// Se path for um link simbólico, stat retorna info sobre o arquivo referenciado
// enquanto lstat retorna sobre o próprio arquivo que contém o link
int fstat(int fildes, struct stat *buf);
// Retorna info sobre o arquivo associado ao descritor fildes
```

Campos do struct stat definidos no padrão:

```
dev_t st_dev; /* device ID of device containing file */
mode_t st_mode; /* file mode */
nlink_t st_nlink; /* number of hard links */
uid_t st_uid; /* user ID of file */
gid_t st_gid; /* group ID of file */
off_t st_size; /* file size in bytes (regular files) */
time_t st_atime; /* path size (symbolic links) */
time_t st_mtime; /* time of last access */
time_t st_ctime; /* time of last data modification */
time_t st_mtime; /* time of last file status change */
```

System Calls – Informações sobre Arquivos (3)

- Exemplo: função para verificar se um arquivo é um diretório

```
#include <stdio.h>
#include <time.h>
#include <sys/stat.h>

int isdirectory(char *path) {
    struct stat statbuf;
    if (stat(path, &statbuf) == -1)
        return 0;
    else
        return S_ISDIR(statbuf.st_mode);
}
```

40

System Calls – Criando Links (1)

```
#include <sys/unistd.h>
int link (const char *path1, const char *path2);
// Cria um hard link (path2 -> path1)
int unlink (const char *path1, const char *path2);
// Apaga um hard link
```

Exemplo: criando um hard link

```
#include <stdio.h>
#include <sys/stat.h>
...
if (link("/dirA/name1", "/dirB/name2") == -1)
    perror("Failed to make a new link in /dirB");
...
```

In /dirA/name1 /dirB/name2

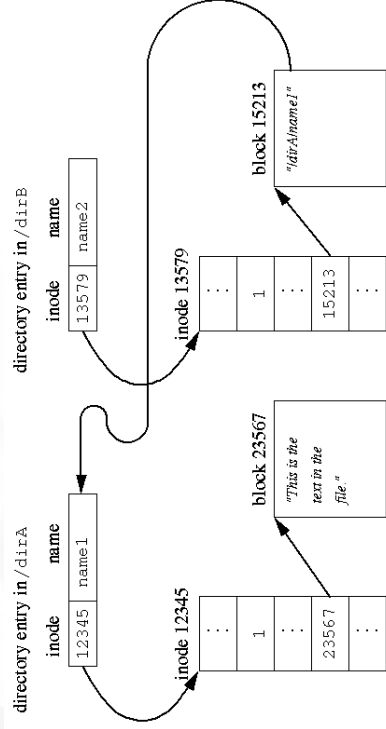
41

System Calls – Criando Links (3)

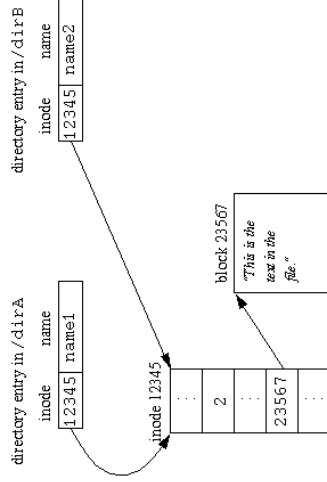
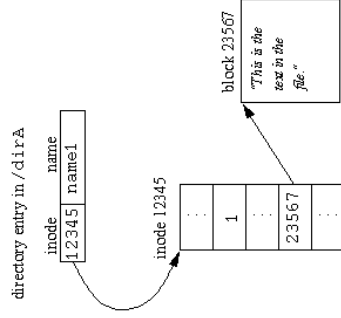
Criando link simbólicos

```
#include <sys/unistd.h>
int symlink (const char *path1, const char *path2);
// Cria um link simbólico (path2 -> path1)
```

In -s path1 path2



System Calls – Criando Links (2)



42

System Calls – Lendo e Escrevendo (1)

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

```
size_t
```

```
// Tipo de dado usado para representar o número de blocos a ler ou escrever em
// uma operação read ou write.
```

```
ssize_t read (int filedes, void *buffer, size_t size)
```

```
// Lê até size bytes do arquivo indicado pelo descritor filedes, armazenando o
// resultado em buffer. Retorna o número de bytes lidos, zero (EOF) ou -1 (erro)
```

```
ssize_t write (int filedes, const void *buffer, size_t size)
```

```
// Escreve até size bytes de dados contidos em buffer no arquivo indicado por
// filedes. Retorna o número de bytes escritos ou -1 (erro). Assim que a
// operação retorna, os dados estão disponíveis para leitura, mas não estão
// necessariamente no disco.
```

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

```
// Permite mudar a posição do ponteiro do arquivo indicado por filedes. O
// valor do deslocamento offset depende do modo de operação indicado por whence,
// que pode ser: relativo ao início do arquivo (SEEK_SET), à posição corrente
// (SEEK_CUR) ou ao final do arquivo (SEEK_END). Ver também lseek64.
```

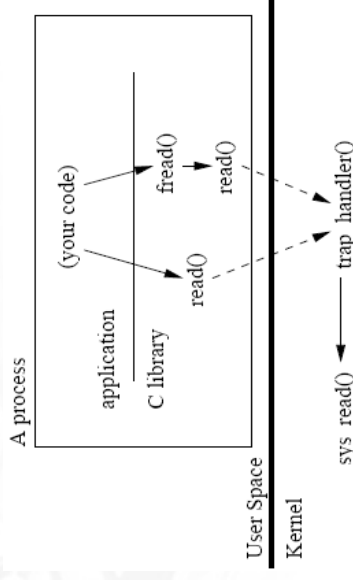
44

System Calls - { Lendo e Escrevendo (2)

Abrir um arquivo em modo de leitura, ler seus últimos 256 bytes e escrevê-los na saída padrão.

```
#int main (int argc, char *argv[])
{
    int fd ; /* file descriptor */
    char buffer[SIZE] ;
    ssize_t bytesLidos, bytesEscritos ;
    /* abrir arquivo em leitura */
    fd = open ("x", O_RDONLY) ;
    if ( fd < 0 )
    {
        perror ("Erro ao abrir o arquivo x") ;
        exit (1) ;
    }
    /* posicionar a SIZE bytes do final do arquivo */
    lseek (fd, -SIZE, SEEK_END) ;
    /* ler SIZE bytes do arquivo */
    bytesLidos = read (fd, &buffer, SIZE) ;
    if ( bytesLidos < 0 )
    {
        perror ("Erro na leitura de x") ;
        exit (1) ;
    }
    /* escrever os bytes lidos no terminal (stdout) */
    bytesEscritos = write (STDOUT_FILENO, &buffer, bytesLidos) ;
    if ( bytesEscritos < 0 )
    {
        perror ("Erro na escrita em stdout") ;
        exit (1) ;
    }
}
```

UNIX apresenta várias interfaces para manipular arquivos



Referências

- Vahalia, U. “Unix Internals – The New Frontiers”, 1ª. Edição, Editora Prentic-Hall, 1996.
 - Capítulo 8
- A. S. Tanenbaum, “Sistemas Operacionais Modernos”, 3a. Edição, Editora Prentice-Hall, 2010.
 - Seção 4.5
- Silberschatz A. G.; Galvin P. B.; Gagne G.; “Fundamentos de Sistemas Operacionais”, 6a. Edição, Editora LTC, 2004.
 - Seção 4.7