Insper

Sistemas Hardware-Software

Aula 15 - Entrada e Saída

Engenharia
Fabio Lubacheski
Maciel C. Vidal
Igor Montagner
Fábio Ayres

Objetivos de hoje

- Compreender o mecanismo usado pelo Sistema Operacional para expor **recursos de hardware**
- Utilizar chamadas de sistema POSIX para **ler** e **escrever arquivos**
- Compreender **permissões** de **arquivos** em sistemas POSIX
- Experimentar uma situação de **concorrência** de recursos

Sistemas Operacionais

Kernel: software do sistema que gerencia

- Programas
- Memória
- Recursos do hardware

Roda com **privilégios totais** no hardware. Grosso modo, é um conjunto de handlers de interrupção.

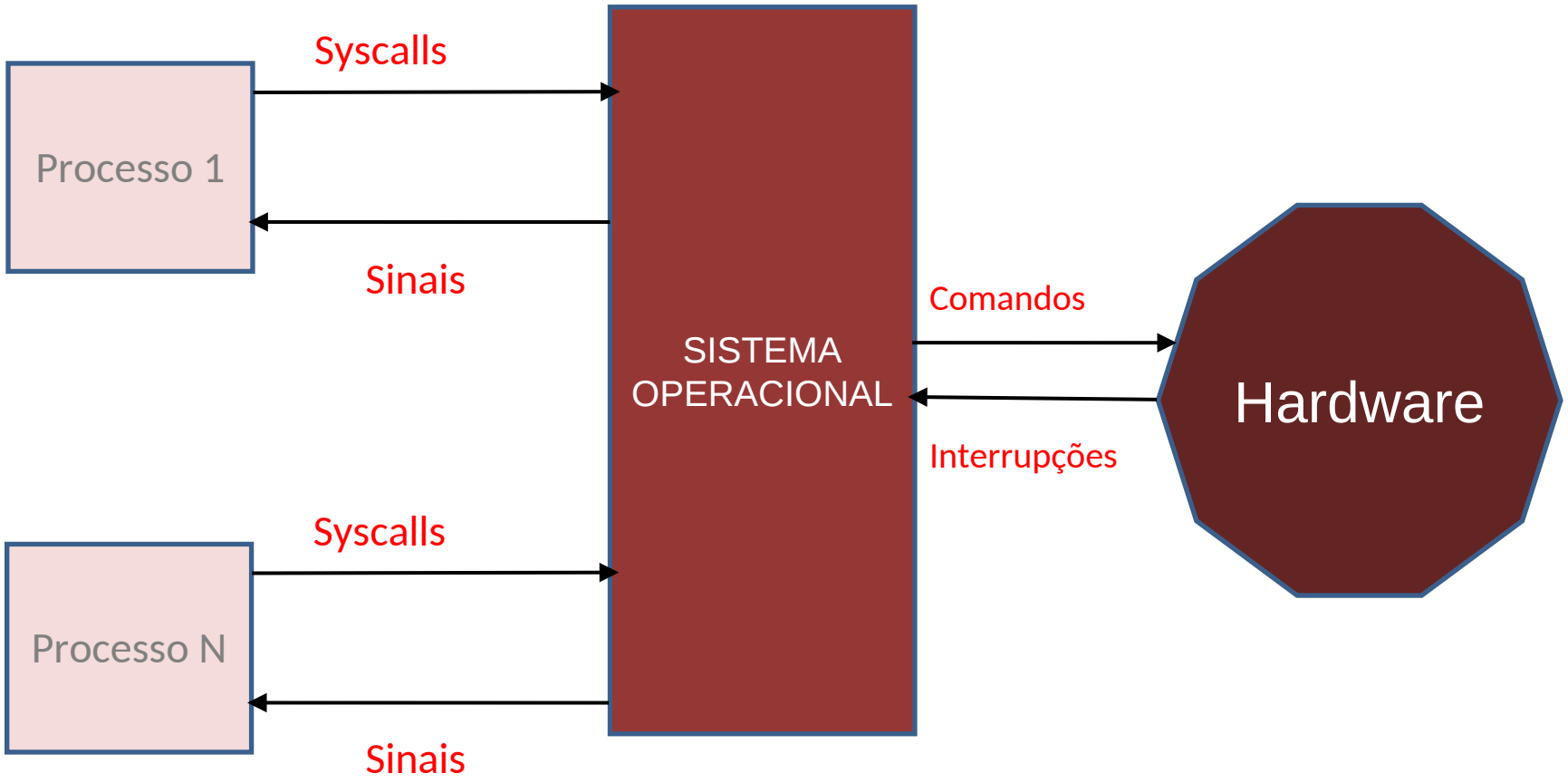
Sistemas Operacionais

Processo de usuário: qualquer programa sendo executado no computador. **A falha de um processo não afeta os outros.**

Roda com **privilégios limitados**. Interage com o hardware por meio de **chamadas ao kernel** para obter

- Memória
- Acesso ao disco e outros periféricos
- Comunicar com outros processos

Sistemas Operacionais



Insper

SisCalls no Padrão POSIX

- Gerenciamento de usuários e grupos
- Manipulação de arquivos (incluindo permissões) e diretórios
- Criação de processos e carregamento de programas
- Comunicação entre processos
- Interação direta com hardware (via drivers)

Trabalhando com arquivos

- Durante a execução de um programa, seus dados ficam na memória. Quando o programa termina, ou o computador é desligado, os dados na memória desaparecem.
- Trabalhar com arquivos é muito parecido com trabalhar com livros. Para utilizar um livro, você tem que **abri-lo**. Quando você termina, você tem que **fechá-lo**.
- Enquanto o livro estiver **aberto**, você pode tanto lê-lo quanto escrever nele. Na maioria das vezes, você lê o livro em sua ordem natural, uma página após a outra.

Trabalhando com arquivos

- Em um **arquivo texto** os dados são organizados como uma sequência de caracteres dividida em linhas terminadas por um **caractere de fim de linha (\n)**.

- Em um programa a manipulação do arquivo passa por três etapas: **abertura do arquivo**, **leitura/escrita dos dados** e **fechamento do arquivo**.

Syscalls para arquivos

- Abrir e fechar arquivos: **open()** e **close()**
- Ler e escrever em arquivos: **read()** e **write()**
- Mudar posição corrente no arquivo: **lseek()**

Abrindo arquivos

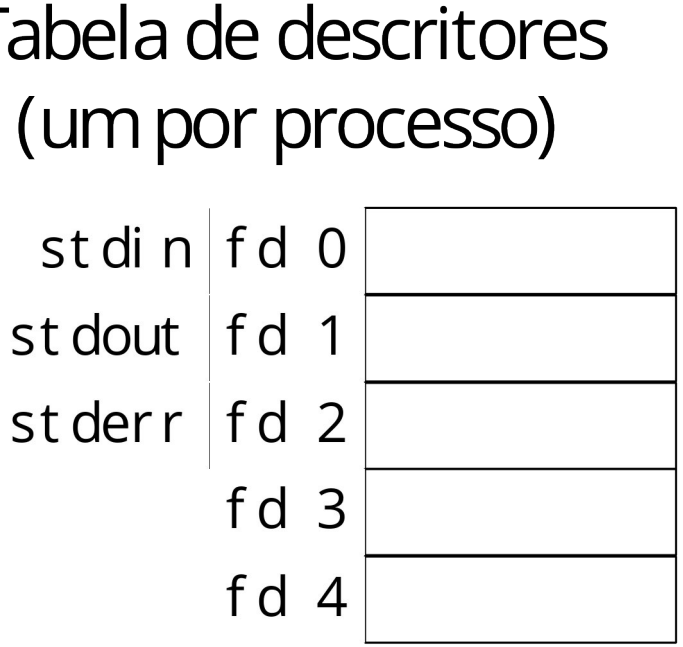
```
int open(const char *pathname, int flags,
                mode_t mode);
```

- Retorna um inteiro chamado ***file descriptor***.
- `flags` indicam opções de abertura de arquivo
  - `O_RDONLY, O_WRONLY, O_RDWR`
  - `O_CREAT` (cria se não existir)
  - `O_EXCL + O_CREAT` (se existir falha)
- `mode` indica as permissões de um arquivo criado usando open.

Descritores de Arquivos

- Um descritor de arquivo é um índice para um registro de uma tabela do descritor de arquivo do Kernel do Sistema Operacional.

- Cada processo possui sua própria tabela e cada entrada na tabela é acessada por um inteiro, um **int** em C.



E/S padrão

Todo processo criado por um shell Linux já vem com três arquivos abertos, e associados com o terminal:

0: standard input (stdin) – fd=0

1: standard output (stdout) - fd=1

2: standard error (stderr) - fd=2

Lendo/escrevendo em um arquivo

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

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

- Cada chamadas lê/escreve no máximo `count` bytes apontados por `buf` no arquivo `fd`.
- Ambas retornam o número de bytes lidos/escritos e -1 se houver erro.
- Se `read` retornar 0 acabou o arquivo.

Exemplo usando arquivo – ex1.slides.c

```c
int main () {
    int fd;
    char c;
    // abre o arquivo, retorna -1
    // se o arquivo não existir
    fd = open("entrada.txt", O_RDONLY);
    // le um caractere do arquivo
    read(fd,&c,1);

    printf("c = %c\n", c);
    // Fechar um arquivo informa ao kernel
    // que você já terminou de acessar
    // o arquivo. SEMPRE FAÇA ISSO.
    close(fd);
    return 0;
}
```

Exemplo usando arquivo – ex2.slides.c

```c
int main () {
    int fd1,fd2;
    char c;
    fd1 = open("entrada.txt", O_RDONLY);
    fd2 = open("entrada.txt", O_RDONLY);

    read(fd1,&c,1);
    printf("c1 = %c\n", c);

    read(fd2,&c,1);
    printf("c2 = %c\n", c);

    close(fd1);
    close(fd2);
    return 0;
}
```

Qual a saída nesse exemplo ?

Exemplo usando arquivo – ex3.slides.c

```c
int main () {
    int fd;
    char c;
    fd = open("entrada.txt", O_RDONLY);
    if(fork()==0){
        read(fd,&c,1);
        printf("c filho = %c\n", c);
    }else{
        read(fd,&c,1);
        printf("c pai = %c\n", c);
    }
    close(fd);
    return 0;
}
```

Qual a saída nesse exemplo ?

Explicando os exemplos

- No **ex2_slides.c** a saída será a letra **a** é impressa duas vezes, pois o processo abre o arquivo duas vezes, e cada chamada da função **open** gera um descritor (3 e 4) que tem acesso a um file position **distinto** na tabela de descritores.
- No **ex3_slides.c** o processo filho recebe uma cópia do descritor do processo pai e os dois processos apontam para mesma entrada na tabela de arquivos compartilhando, ou seja, os dois processos tem o mesmo **file position**, ou seja, quando um processo faz a leitura de um caractere o **file position** é atualizado e próximo processo irá ler na posição atualizada.

Atividade prática

Trabalhando com arquivos (40 minutos)

1. Usar funções básicas de tratamento de arquivos

Tipos de arquivos

- Arquivos regulares
    - Dados arbitrários
- Diretórios
    - Um índice para um grupo de arquivos
- Sockets
    - Para comunicar com outro processo em outra máquina

Arquivos regulares

Para o kernel, não existe diferença entre “arquivo texto” e “arquivo binário”: é tudo byte!

Arquivos texto: conceitualmente são uma sequência de linhas

Término de linhas:

- Linux e MacOS: *newline* ou *line feed* (‘\n’)
- Windows e protocolos Internet: *carriage return* seguido de *line feed* (‘\r\n’)

Diretórios

Um diretório é um array de links, mapeando um nome de arquivo a um arquivo

Todo diretório contém ao menos duas entradas:

- . (dot) é um link para si próprio
- .. (dot dot) é um link para o diretório pai na hierarquia de diretórios

Comandos: mkdir, ls, rmdir

Cada processo roda em um diretório corrente (current working directory – cwd), que pode ser alterado com chdir()

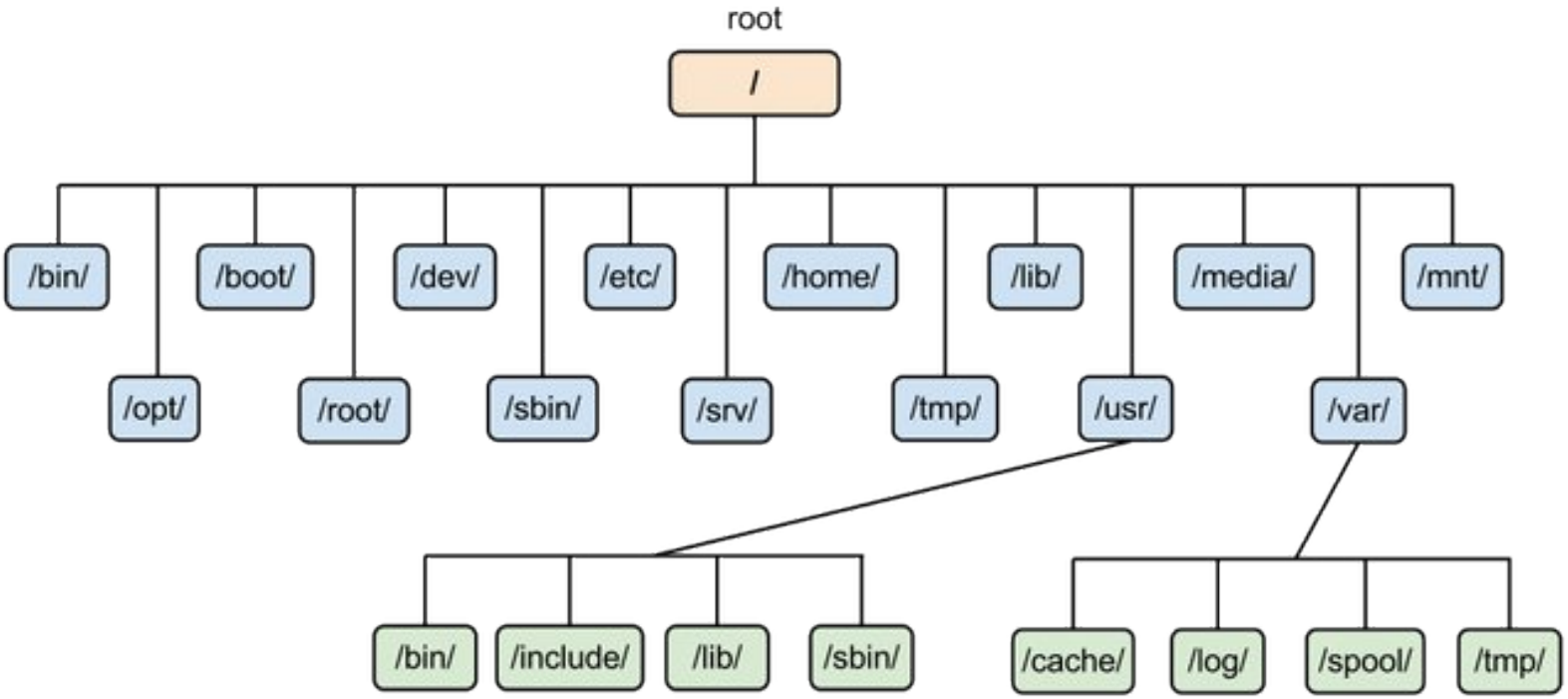
More Details

In a Google+ post by Rob Pike, A lesson in shortcuts, the more detailed rationale behind the dot files.

> Long ago, as the design of the Unix file system was being worked out, the entries . and .. appeared, to make navigation easier. I'm not sure but I believe .. went in during the Version 2 rewrite, when the file system became hierarchical (it had a very different structure early on). When one typed ls, however, these files appeared, so either Ken or Dennis added a simple test to the program. It was in assembler then, but the code in question was equivalent to something like this:
>
> `if (name[0] == '.') continue;`
>
> This statement was a little shorter than what it should have been, which is
>
> `if (strcmp(name, ".") == 0 || strcmp(name, "..") == 0) continue;` But hey, it was easy. Two things resulted. First, a bad precedent was set. A lot of other lazy programmers introduced bugs by making the same simplification. Actual files beginning with periods are often skipped when they should be counted. Second, and much worse, the idea of a "hidden" or "dot" file was created. As a consequence, more lazy programmers started dropping files into everyone's home directory. I don't have all that much stuff installed on the machine I'm using to type this, but my home directory has about a hundred dot files and I don't even know what most of them are or whether they're still needed. Every file name evaluation that goes through my home directory is slowed down by this accumulated sludge.
>
> I'm pretty sure the concept of a hidden file was an unintended consequence. It was certainly a mistake.

https://linux-audit.com/linux-history-how-dot-files-became-hidden-files/

Arquivos em Unix




```
man hier
```

Fonte: https://nepalisupport.wordpress.com/2016/06/29/linux-file-system-hierarchy/

Insper

Permissões de arquivos

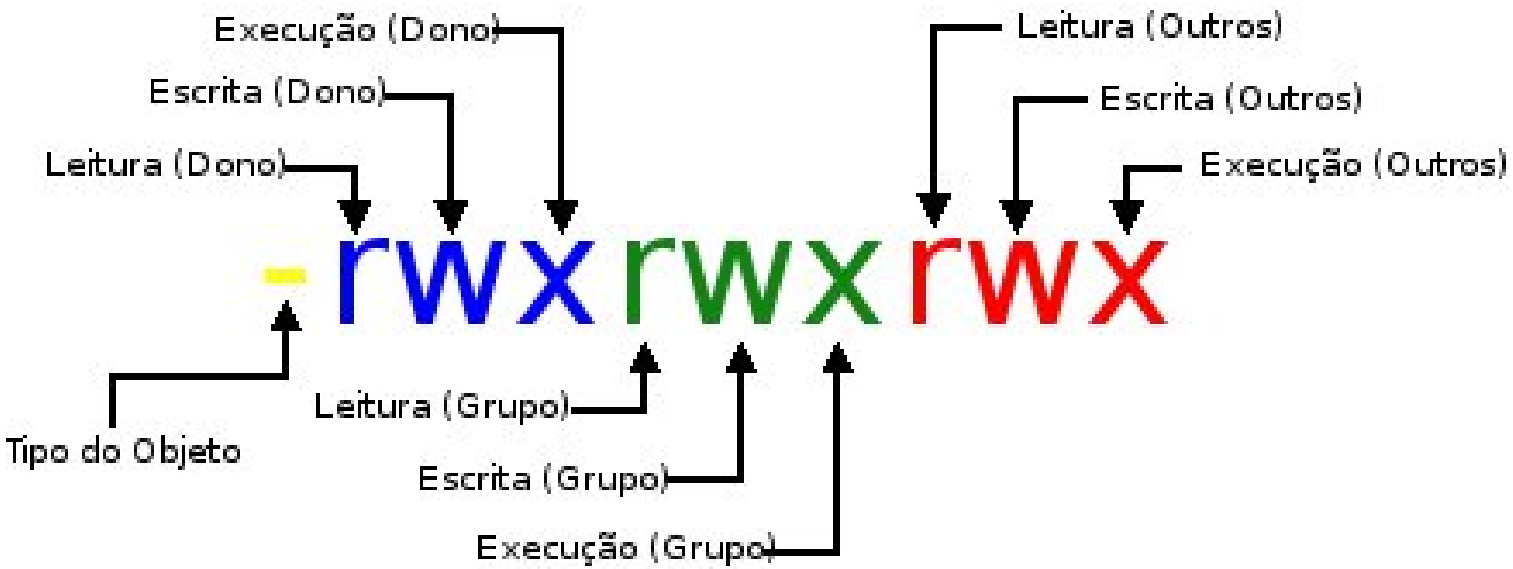
- Funcionam dentro da camada de Aplicação.
- Cada arquivo possui um usuário dono
- Permissões de leitura(4), escrita(2) e execução(1) para
  - Usuário dono do arquivo
  - Usuários no mesmo grupo de usuários do dono
  - Todo mundo
- Permissões codificadas usando números de 0 a 7

```
man chmod
man chown
```

Visualizando permissões no Linux

- Para visualizar a permissão de um diretório, podemos ir até o local onde esse diretório se encontra. E digitar ls -l (lista com detalhes os arquivos

```
$ ls -l entrada.txt
-rw-r--r-- 1 fabio fabio 2 out 23 09:34 entrada.txt
```



Metadados

```
/* Metadata returned by the stat and fstat functions */
struct stat {
    dev_t         st_dev;      /* Device */
    ino_t         st_ino;      /* inode */
    mode_t        st_mode;     /* Protection and file type */
    nlink_t       st_nlink;    /* Number of hard links */
    uid_t         st_uid;      /* User ID of owner */
    gid_t         st_gid;      /* Group ID of owner */
    dev_t         st_rdev;     /* Device type (if inode device) */
    off_t         st_size;     /* Total size, in bytes */
    unsigned long st_blksize;  /* Blocksize for filesystem I/O */
    unsigned long st_blocks;   /* Number of blocks allocated */
    time_t        st_atime;    /* Time of last access */
    time_t        st_mtime;    /* Time of last modification */
    time_t        st_ctime;    /* Time of last change */
};
```

Descrição detalhada em **`man 7 inode`** e **`man 2 stat`**

Atividade prática

Permissões e posse de arquivos (30 minutos)

1. Utilizar e entender as permissões de arquivo nas funções de E/S

Insper

www.insper.edu.br