Insper

Sistemas Hardware-Software

Aula 16 – Chamadas de sistema: Entrada/Saída

2018 – Engenharia

Igor Montagner, Fábio Ayres <igorsm1@insper.edu.br>

Linux do Zero

- Kernel
- Biblioteca padrão
  - Interface direta com o kernel
  - Funções de conveniência
- Ferramentas de usuário
  - ls, mkdir, rm, ln
  - bash ou outro shell interativo
- Sistema de Arquivos

Sistemas Operacionais

"software that controls the operation of a computer and directs the processing of programs (as by assigning storage space in memory and controlling input and output functions) ."

(Merriam Webster)

Sistemas Operacionais

Controla acesso a

- Memória
- Armazenamento
- Dispositivos

Para diversos programas de modo a garantir

- Isolamento
- Divisão de tempo de processamento
- Acesso concorrente aos dispositivos

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**. Interaje com o hardware por meio de **chamadas ao kernel** para obter

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

POSIX

1)The Portable Operating System Interface (POSIX) is a family of standards specified by the IEEE Computer Society for maintaining compatibility between operating systems. POSIX defines the **application programming interface (API)**, along with command line shells and utility interfaces, for software compatibility with variants of Unix and other operating systems

2) Wikipedia

POSIX

- O SO disponibiliza uma API para interagir com o hardware
- POSIX é uma API que pode ser implementada por vários sistemas diferentes
  - Linux
  - MacOS
  - Haiku
  - Windows
- Sistemas *POSIX compliant* são compatíveis em nível de código fonte
- Arquivo de cabeçalho <unistd.h>

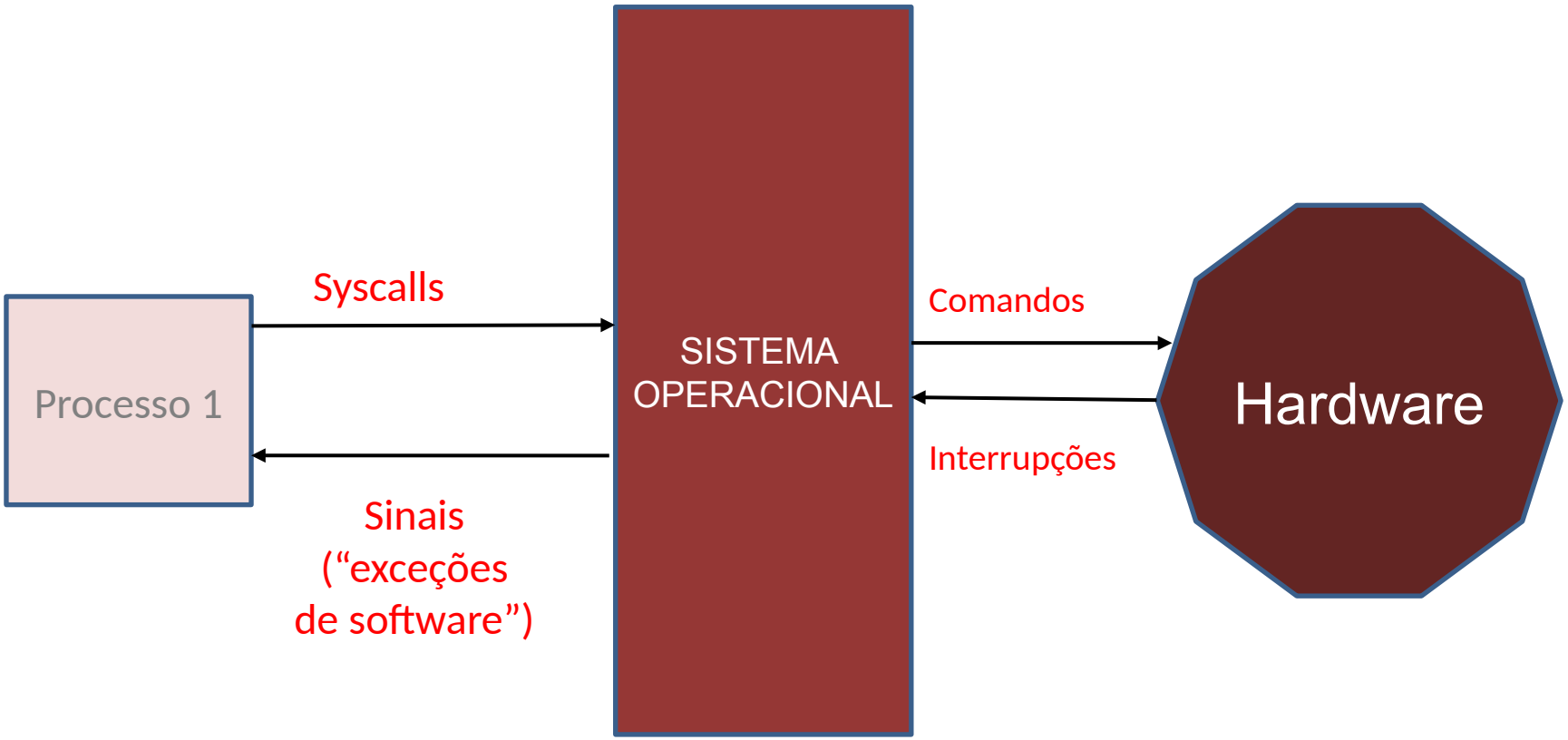
Windows API

- Windows também disponibiliza sua API própria
- WINE: implementação da API windows via POSIX:

> "Instead of simulating internal Windows logic like a virtual machine or emulator, Wine translates Windows API calls into POSIX calls on-the-fly"
> (winehq.org)

- Cygwin: implementação de API POSIX em cima da API windows. É necessário recompilar código para funcionar.
- Documentação: https://msdn.microsoft.com/en-us/library/windows/desktop/ff818516(v=vs.85).aspx

Interação do SO com seus processos



Insper

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)

Syscalls para arquivos (regulares)

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

Arquivos em Unix

- Um arquivo é uma sequência de m bytes: $B_0, B_1, \ldots, B_{m-1}$

- Todos os dispositivos de entrada/saída são representados como arquivos!
  - /dev

- O kernel disponibiliza algumas estruturas de dados para os usuários via sistema de arquivos!
  - /proc
  - /sys

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_CREATE` (cria se não existir)
  - `O_EXCL + O_CREATE` (se existir falha)
- `mode` indica as permissões de um arquivo criado usando open.

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)

1: standard output (stdout)

2: standard error (stderr)

Fechando um arquivo

Fechar um arquivo informa ao kernel que você já terminou de acessar o arquivo.

```
int fd;      /* file descriptor */
int retval; /* return value */

if ((retval = close(fd)) < 0) {
   perror("close");
   exit(1);
}
```

Cuidado: não feche um arquivo já fechado!

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.

Short counts

Quando short count pode ocorrer:

- EOF
- Lendo linhas do terminal
- Lendo e escrevendo em sockets

Short counts nunca ocorrem quando:

- Lendo de arquivos (exceto quando EOF)
- Escrevendo para arquivos

`read` e `write` são, por padrão, síncronas e bloqueiam se o arquivo não estiver pronto.

Atividade 1

Exercícios 1-4; Tempo total 30 minutos;

Arquivos em Unix

- Um arquivo é uma sequência de m bytes: $B_0, B_1, \ldots, B_{m-1}$

- Todos os dispositivos de entrada/saída são representados como arquivos!
  - /dev

- O kernel disponibiliza algumas estruturas de dados para os usuários via sistema de arquivos!
  - /proc
  - /sys

Tipos de arquivos

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

Tipos de arquivos

- Pipes(FIFOs)
  - Comunicação entre processos da mesma máquina
- Links simbólicos
  - Quase um “ponteiro” para outro arquivo!
- Dispositivos
  - Tipo bloco e caractere

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()

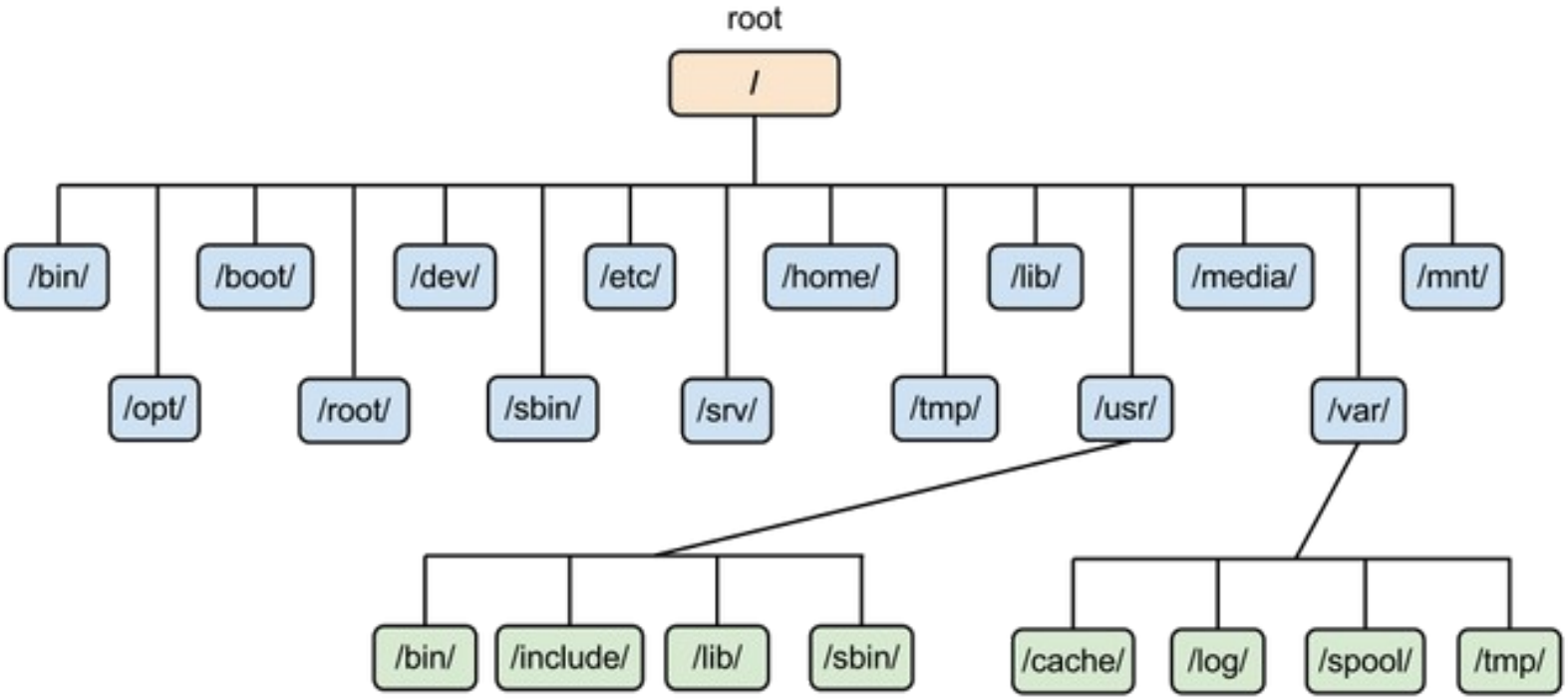
More Details

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/

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
```

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`

Permissões de arquivos

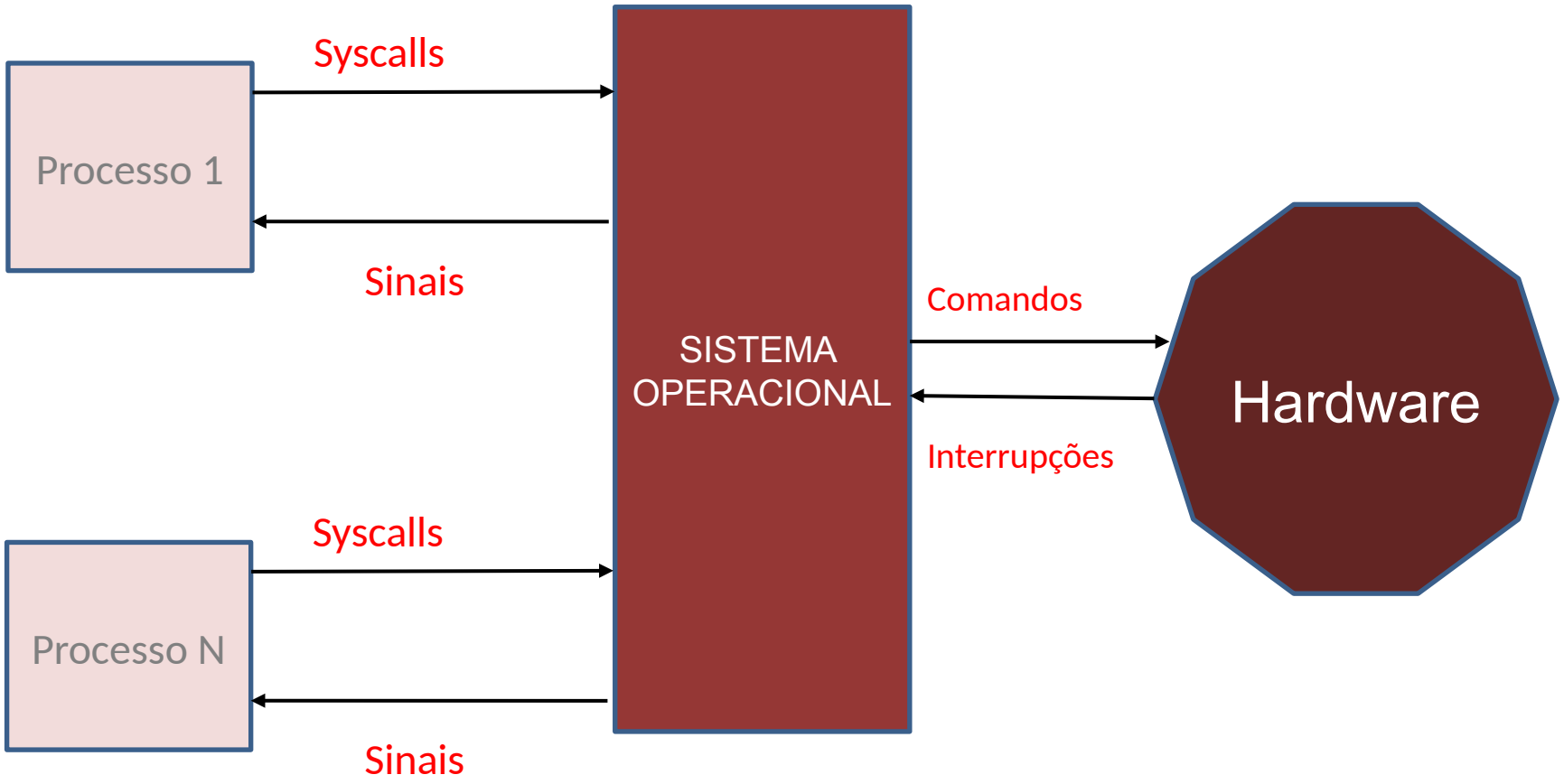
- Funcionam dentro da camada de Aplicação.
- Cada arquivo possui um usuário dono
- Permissões de leitura(4), escrita(2) e execução(1) para
  - Usuário dono do arquivo
  - Usuários no mesmo grupo de usuários do dono
  - Todo mundo
- Permissões codificadas usando números de 0 a 7

```
man
chmod
man
```

Atividade 2

- Parte 2 – Exercícios 1 - 4

Próxima aula



Insper

Insper

www.insper.edu.br