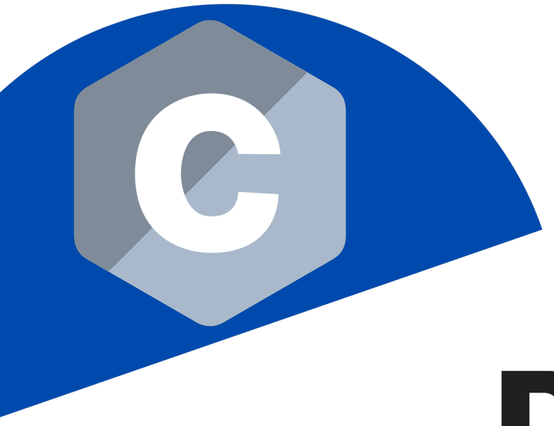


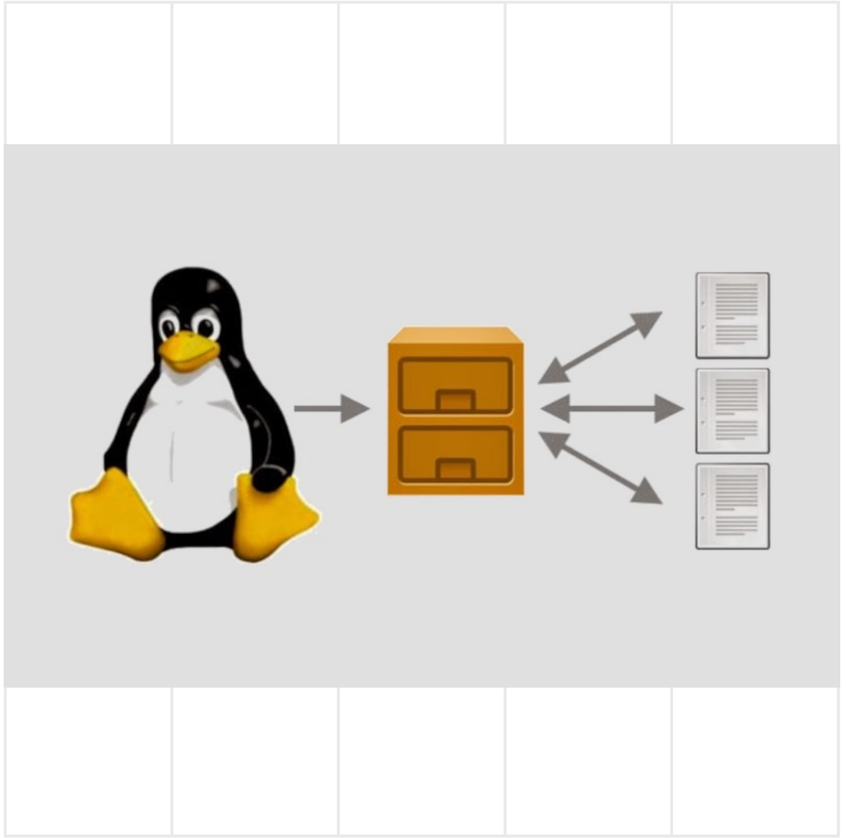
Processamento de Ficheiros

(usando a API do kernel – Parte 1)

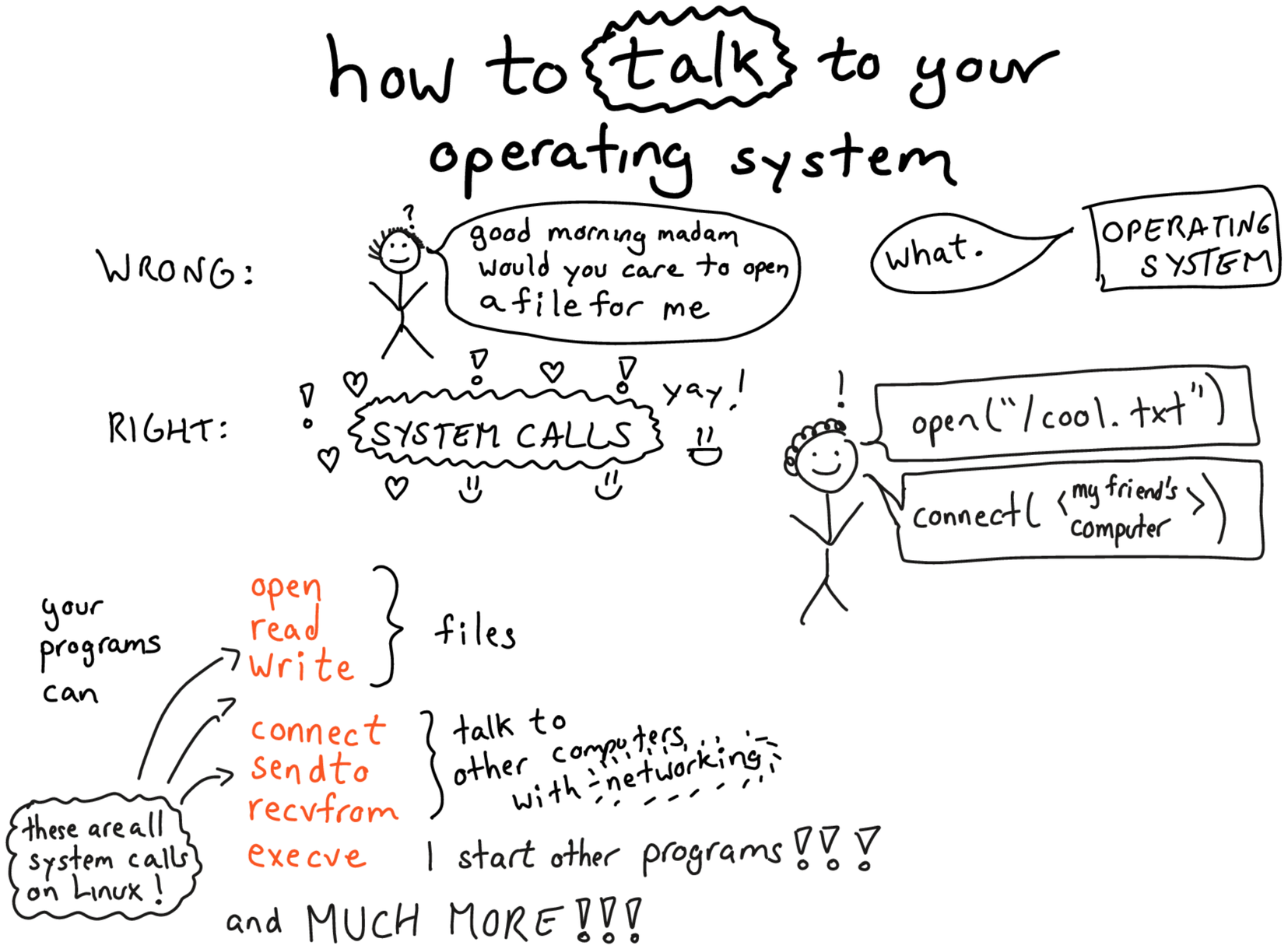
Bem-vindos à aula!

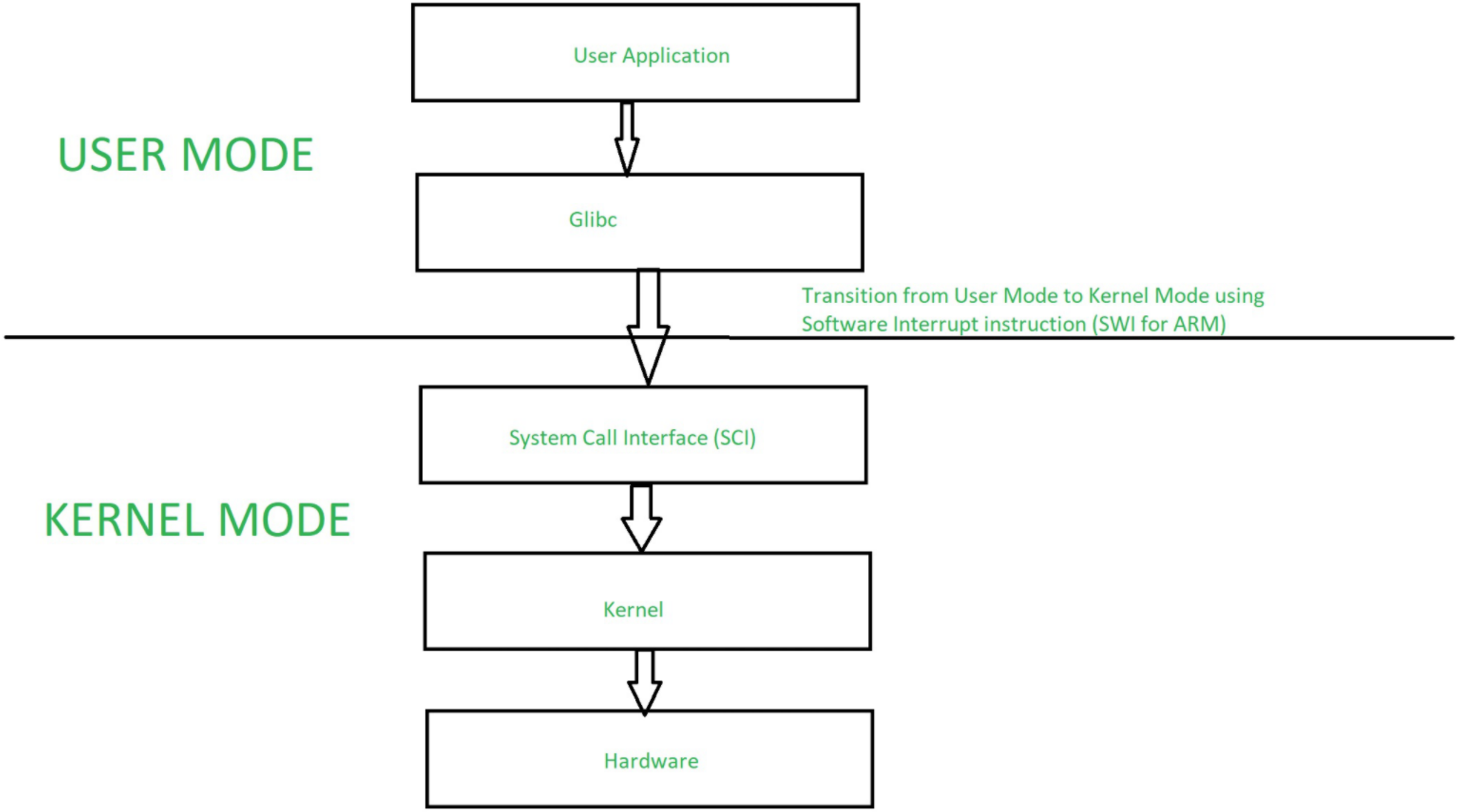
Agenda de Hoje

- Biblioteca Padrão (stdio.h) vs. Sistema Operativo (syscall)
- Processamento de ficheiros



stdio.h *vs.* system call




User Application
USER MODE
Glibc
Transition from User Mode to Kernel Mode using
Software Interrupt instruction (SWI for ARM)
System Call Interface (SCI)
KERNEL MODE
Kernel
Hardware

user space vs. kernel space

Julia Evans
@b0rk

drawings.jvns.ca

the Linux kernel has **millions** of lines of code

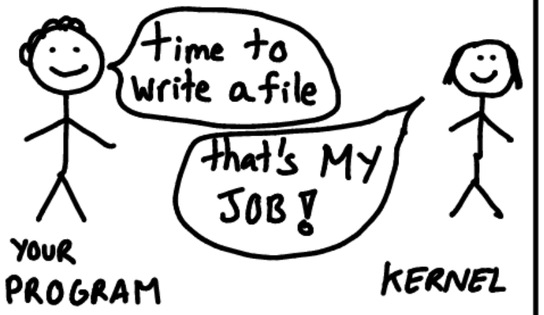
- read + write files
- decide which programs get to use the CPU
- make the keyboard work

when Linux kernel code runs, that's called

kernel space

when your program runs, that's

user space



time for a **context switch** to kernel space

your program switches back and forth

```
str = "my string"
x = x+2
file.write(str)   <- switch to kernel space
y = x+4
str = str * y     <- aand we're back to user space!
```

timing your process

```
$ time find /home
```

0.15 user — time spent in your process

0.73 system — time spent by the kernel doing work for your process

stdio.h *vs.* system call

stdio.h
Fornece uma camada de abstração que facilita a manipulação de ficheiros de forma mais intuitiva, usando `FILE *` para representar ficheiros, gerindo *buffers* automaticamente.

Syscalls (POSIX)
Interagem diretamente com o sistema operativo, sendo menos abstratas. Elas trabalham com `file descriptors`, que são inteiros representando ficheiros abertos no modo kernel, e não oferecem *buffering* automático.

Buffering

stdio.h
Utiliza buffers internamente, o que significa que armazena temporariamente os dados em memória antes de lê-los ou gravá-los no ficheiro. Isso permite um acesso mais eficiente em operações de leitura/escrita sequenciais, pois reduz o número de chamadas ao sistema, aumentando o desempenho.

Syscalls (POSIX)
Não faz uso de buffer internamente. O programador precisa lidar diretamente com os dados sem o benefício de armazenamento temporário, tornando o processo menos eficiente para operações repetitivas, mas fornecendo mais controle e precisão (o que é importante em operações de leitura/escrita em tempo real ou sincronizadas).

File Descriptors

Um **file descriptor** (ou descritor de ficheiro) é um identificador numérico que o sistema operativo atribui a cada ficheiro aberto, representando um "ponto de acesso" para operações de leitura, escrita e manipulação desse ficheiro. Em sistemas Unix e POSIX, como Linux e macOS, file descriptors são usados para gerenciar não apenas ficheiros, mas também outros recursos de entrada/saída, como **sockets**, **pipes**, e **dispositivos**.

Operações com File Descriptors

Usando um file descriptor, um programa pode:

- **Ler** de um ficheiro: utilizando funções como `read`, que lê diretamente do descritor.
- **Escrever** em um ficheiro: usando funções como `write`, que grava diretamente no descritor.
- **Mover-se** dentro do ficheiro: usando `lseek`, que permite alterar a posição de leitura/escrita dentro do ficheiro.
- **Fechar** o ficheiro: usando `close`, que libera o descritor de ficheiro e permite que o sistema o reatribua a outro recurso.

File Descriptors

No Unix e em sistemas compatíveis, os *file descriptors* padrão são:

- STDIN_FILENO (0) - Entrada padrão (*standard input*), geralmente o teclado.
- **STDOUT_FILENO** (1) - Saída padrão (*standard output*), geralmente a tela ou o console.
- STDERR_FILENO (2) - Saída de erro padrão (*standard error*), geralmente também a tela.

```
write(STDOUT_FILENO, buffer, nbytes);
```

Processamento de Ficheiros

mycat.c

```c
#include <stdio.h>
#include <stdlib.h>
#define BUFFER_SIZE 1024


int main (int argc, char* argv[]) {
     int fd = open(argv[1], O_RONLY);
     char buffer[BUFFER_SIZE];


     int nbytes = read(fd, buffer, BUFFER_SIZE);
     while (nbytes > 0) {
          write(STDOUT_FILENO, buffer, nbytes);
          nbytes = read(fd, buffer, BUFFER_SIZE);
     }

     close(fd);
     exit(EXIT_SUCCESS);
}
```

open()

Sintaxe

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

Parâmetros

`pathname`: caminho do ficheiro a ser aberto.
`flags`: especifica o modo de abertura do ficheiro.

Modos comuns de abertura

O_RDONLY: abre o ficheiro para leitura
O_WRONLY: abre o ficheiro para escrita
O_RDWR: abre o ficheiro para leitura e escrita

Retorno

Sucesso: Retorna um **file descriptor** (inteiro) que pode ser usado para operações subsequentes no ficheiro.
Falha: Retorna -1

```
int fd = open(file.txt, O_RDONLY);
if (fd == -1) {
     printf("error: cannotopen”);
     exit(EXIT_FAILURE);
}
```

read()

Sintaxe

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

Parâmetros

`fd:` file descriptor do ficheiro

`buf:` buffer onde os dados lidos serão armazenados. Esse buffer deve ser previamente alocado pelo programa para armazenar os dados lidos do ficheiro.

`count`: número máximo de bytes a serem lidos.

```
char buffer[BUFFER_SIZE];
int nbytes = read(fd, buffer,
                  BUFFER_SIZE);
```

Retorno

Sucesso: número de bytes efetivamente lidos

Falha: retorna **-1**

lseek()

Sintaxe

```
off_t lseek(int fd, off_t offset, int whence);
```

Parâmetros

`fd:` descritor do ficheiro

`offset:` número de bytes a mover o apontador de leitura/escrita.

`whence`: define o ponto de referência para o deslocamento (offset). Há 3 valores comuns:

- SEEK_SET: posição específica
- SEEK_CUR: posição atual do ficheiro
- SEEK_END: final do ficheiro

Retorno

Sucesso: retorna a nova posição (em bytes) a partir do início do ficheiro.

Falha: retorna -1 em caso de erro

- É usada para mover a posição de leitura/escrita dentro de um ficheiro aberto, manipulando diretamente o deslocamento (offset) dentro do ficheiro.

```
int newPos = lseek(fd, 2,SEEK_SET);
```

write()

Sintaxe

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

Parâmetros

`fd:` file descriptor do ficheiro onde se quer escrever os dados.

`buf:` buffer que contém os dados a serem escritos

`count`: número de bytes a serem escritos a partir do buffer

Retorno

Sucesso: retorna o número de bytes efetivamente escritos

Falha: retorna -1

```
const char *text = "Olá, mundo!";
int bytesWritten = write(fd, text, 12);
if (bytesWritten == -1) {
     perror("Erro ao escrever no
arquivo");
     close(fd);
     exit(EXIT_FAILURE);
}
```

close()

Sintaxe

```
int close(int fd);
```

Parâmetro

`fd:` file descriptor que se deseja fechar.

Retorno

Sucesso: retorna 0

Falha: retorna -1

- Em sistemas operacionais baseados em Unix, cada processo tem um número limitado de descritores de arquivo disponíveis.
- Não fechar um descritor após seu uso pode levar a um esgotamento de descritores disponíveis.

```
close(fd);
```

stdio.h *vs.* System Call

stdio.h	System Call Equivalente	Descrição
`fopen`	**`open`**	Abre um ficheiro. **`fopen`** retorna um apontador `FILE*`, enquanto **`open`** retorna um `file descriptor` (inteiro).
`fclose`	**`close`**	Fecha um ficheiro. **`fclose`** recebe um `FILE*`, enquanto **`close`** recebe um `file descriptor` (inteiro).
`fread`	**`read`**	Lê dados de um ficheiro. **`fread`** trabalha com buffers de alto nível, enquanto **`read`** é uma leitura de baixo nível.
`fwrite`	**`write`**	Escreve dados em um ficheiro. **`fwrite`** usa buffers internos, enquanto **`write`** grava diretamente no ficheiro.
`fseek`	**`lseek`**	Move o apontador de leitura/escrita para uma posição específica dentro do ficheiro.

stat()

O que é?
A system call `stat` é usada para obter informações detalhadas sobre um ficheiro ou diretório no sistema de ficheiros. Isso inclui dados como o tamanho do ficheiro, permissões, data da última modificação e outros. Essas informações são úteis em muitos contextos, como verificações de segurança, gestão de ficheiros e monitoramento de sistema

Sintaxe

```
int stat(const char *pathname, struct stat *statbuf);
```

Parâmetros
`pathname:` caminho para o ficheiro ou diretório
`statbuf:` apontador para uma estrutura do tipo `struct stat`, onde as informações sobre o ficheiro serão armazenadas. Após a chamada, essa estrutura conterá todos os dados relevantes sobre o ficheiro.

Retorno
Sucesso: retorna 0
Falha: retorna -1

stat()

Estrutura `struct stat`

A estrutura `struct stat` contém vários campos, cada um armazenando um tipo específico de informação sobre o ficheiro. Aqui estão alguns dos campos mais importantes:

- **`st_mode`**: Contém informações sobre as permissões do ficheiro e o tipo (arquivo regular, diretório, etc.).
- **`st_size`**: O tamanho do ficheiro em bytes.
- **`st_atime`**: O tempo do último acesso do ficheiro (última vez que o ficheiro foi lido).
- **`st_mtime`**: O tempo da última modificação do ficheiro (última vez que o conteúdo do ficheiro foi alterado).
- **`st_ctime`**: O tempo da última alteração do `inode` (metadados do ficheiro, como permissões).
- **`st_uid`** e **`st_gid`**: Identificadores do proprietário e do grupo do ficheiro, respectivamente.

stat()

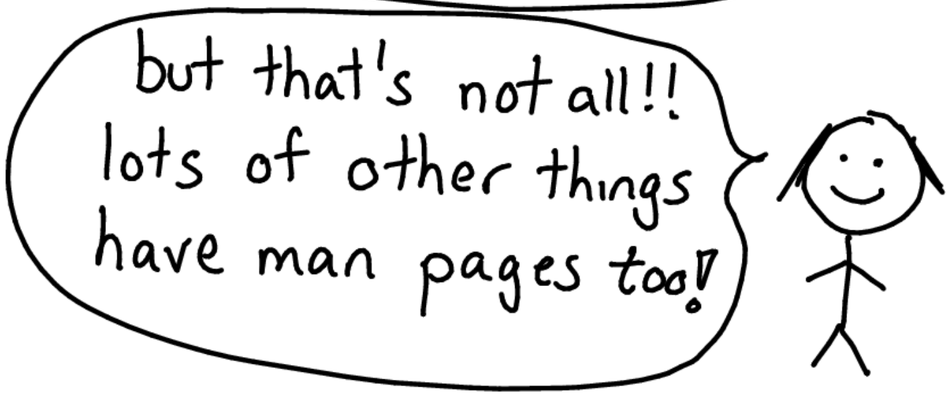
Por que usar `stat`?

- **Verificações de Segurança**: Antes de operar num ficheiro, pode verificar se ele existe e se tem as permissões necessárias para o aceder.
- **Gerenciamento de Ficheiros**: Ao construir aplicações que manipulam ficheiros, como editores de texto ou ferramentas de backup, é crucial saber sobre o estado atual dos ficheiros.
- **Monitorização de Sistemas**: Ferramentas de monitorização podem usar `stat` para acompanhar alterações em ficheiros e diretórios, ajudando na manutenção do sistema.

man pages = awesome

(sometimes. Quality may vary 😬)

Julia Evans
@b0rk



man pages are split up into 8 sections

① ② ③ ④ ⑤ ⑥ ⑦ ⑧

/usr/share/man/man5 has section 5 on my machine.

GREAT

① programs

$man grep
$man ls

② system calls

$man sendfile

③ C functions

$man 3 printf
$man fopen

④ devices

$man null
for /dev/null docs

⑤ file formats

$man sudoers
for /etc/sudoers

→ $man proc

⑥ games

(not very useful)
man sl is good if you have sl though

⑦ miscellaneous

$man 7 pipe
$man 7 symlink
(these are cool!)

⑧ sysadmin programs

$man apt
$man chroot

Julia Evans
@b0rk

unix permissions

drawings.jvns.ca

There are 3 things you can do to a file

read write execute

ls -l file.txt shows you permissions
Here's how to interpret the output:

```
rw-  rw-  r--  bork staff
```

- rw- : bork (user) can read & write
- rw- : staff (group) can read & write
- r-- : ANYONE can read

File permissions are 12 bits

setuid	setgid		user	group	all
0	0	0	110	110	100
		sticky	rwx	rwx	rwx

For files:
- r = can read
- w = can write
- x = can execute

For directories it's approximately:
- r = can list files
- w = can create files
- x = can cd into & modify files

110 in binary is 6

So rw- r-- r--
= 110 100 100
= 6 4 4

chmod 644 file.txt means change the permissions to:

rw- r-- r--

Simple!

setuid affects executables

$ls -l /bin/ping

rws r-x r-x root root

this means ping always runs as root

setgid does 3 different unrelated things for executables, directories, and regular files

Tente e Aprenda

Hora da Atividade

Ficha 4

E por hoje terminamos!