File I/O
Low-level API

Patrício R. Domingues

Departamento de Eng Informática

ESTG/IPLeiria

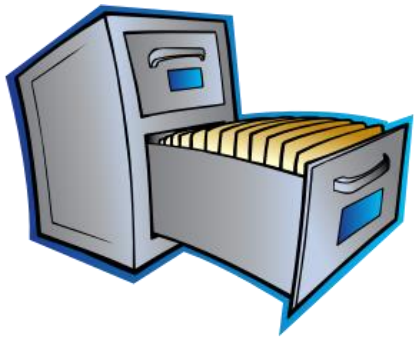
Introduction

- The C language provides two APIs to deal with files
  - Low-level API (*file IO*)
    - open, close, read, write
    - Use of integer descriptors
    - Most of these functions are system calls
  - High-level API (*buffered IO*)
    - fopen, fclose, fprintf,…
    - Stream-oriented
    - Use of FILE* pointers

- NEXT: What’s a file? >>

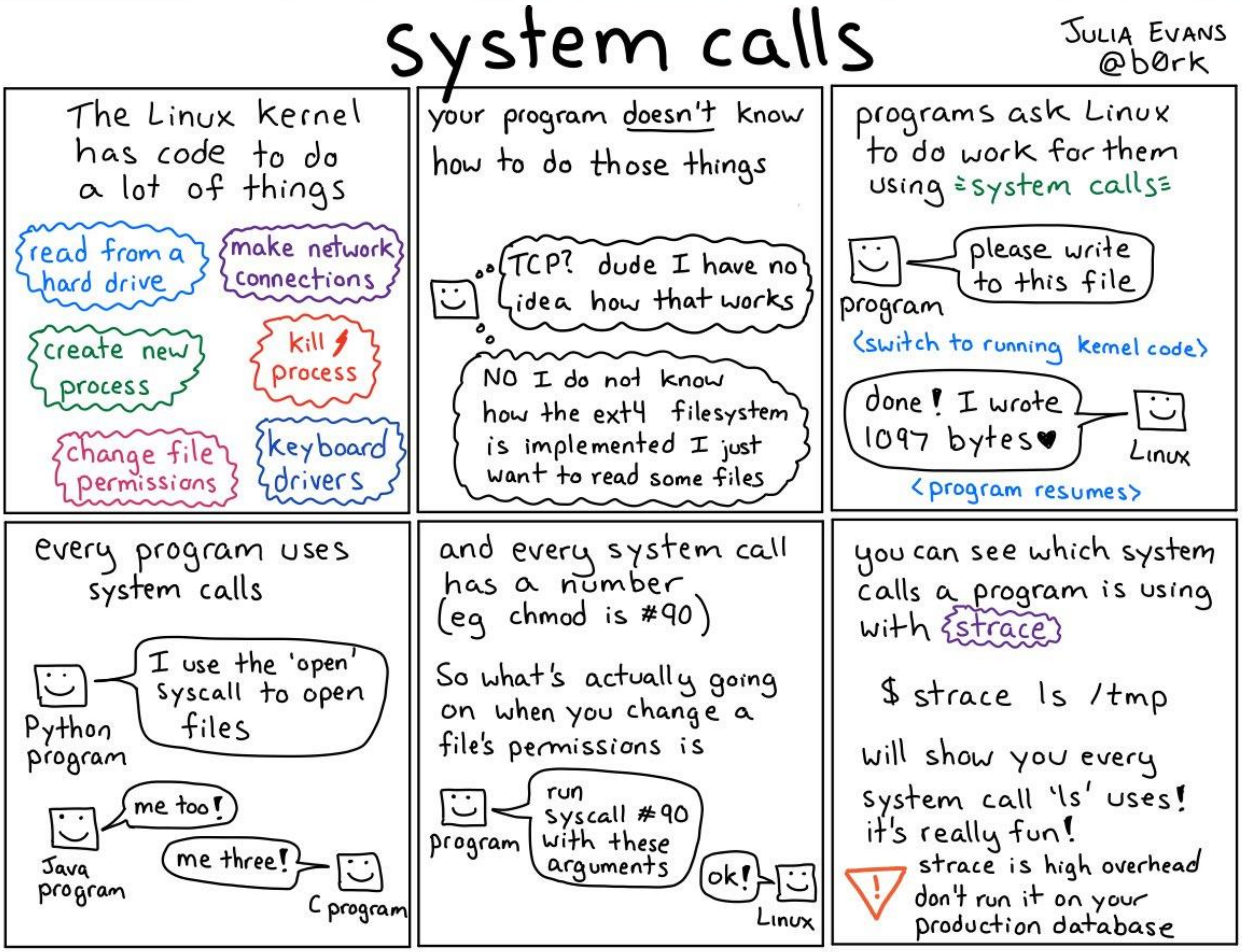
What’s a *file*?

- Abstraction for keeping data in storage
  - Storage is often persistent
    - Survives reboot and power off
  - Unix uses the concept of “file” for other things
    - Pipes, sockets, etc.
- What’s file I/O?
  - Creation, deletion and use of files
- C has two levels of I/O
  - File I/O
  - Buffered I/O



escola superior de tecnologia e gestão
instituto politécnico de leiria

What's a system call?


system calls
JULIA EVANS @b0rk
The Linux kernel has code to do a lot of things
read from a hard drive
make network connections
create new process
kill process
change file permissions
keyboard drivers
your program doesn't know how to do those things
TCP? dude I have no idea how that works
NO I do not know how the ext4 filesystem is implemented I just want to read some files
programs ask Linux to do work for them using system calls
please write to this file
program
<switch to running kernel code>
done! I wrote 1097 bytes
Linux
<program resumes>
every program uses system calls
I use the 'open' syscall to open files
Python program
me too!
Java program
me three!
C program
and every system call has a number (eg chmod is #90)
So what's actually going on when you change a file's permissions is
run syscall #90 with these arguments
program
ok!
Linux
you can see which system calls a program is using with strace
$ strace ls /tmp
will show you every system call 'ls' uses! it's really fun!
strace is high overhead don't run it on your production database

Basic I/O (1)

- In basic I/O, the descriptor of a file is an ***integer***
  - **open** system call
    - maps the file given by name to a file descriptor
    - It returns the descriptor on success
  - The **open** system call has two "versions":
    - `int open(const char *name,int flags);`
    - `int open(const char *name,int flags, mode_t mode);`
      - **mode** is only used when O_CREAT flag is specified
      - **mode** provides the permissions to be assigned to file to be created
        - » Example: mode (expressed in octal) ➔ 0644 (rw-,r--,r--)
  - Documentation: `man 2 open`
    - NOTE: The section #2 of the manual is for system calls

Basic I/O (2)

- Creating a file is so common that there is a dedicated function:
  - `int creat(const char *name, mode_t mode);`
- Note:

```
creat(filename,0644);
```

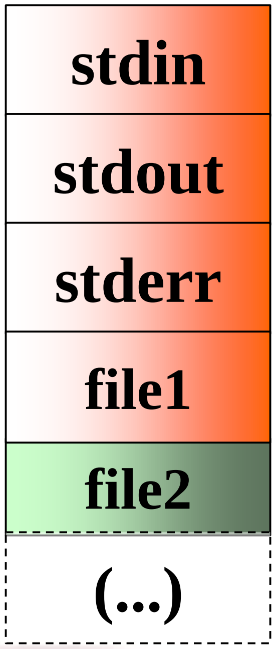
  is equal to...

```
open(filename,O_WRONLY|O_CREAT|O_TRUNC,0644);
```

- Return value for open and creat
  - -1 on error and global variable `errno` is set to the appropriate error code
  - Positive int if file open/`creat` is OK

The *int* descriptor

- Low-level IO uses an *int* descriptor
- The *int* descriptor is returned by open
  - **<u>`int`</u>** open(…)
- This *int* descriptor corresponds to the index of the opened file in the *file descriptor table*
  - There is one *file descriptor table* per process
  - The first three positions are:
  - 0: stdin; 1:stdout; 2:stderr



File descriptor table (one per process)

JULIA EVANS
@b0rk

file descriptors

Unix systems use integers to track open files

process
open foo.txt
okay! that's file #7 for you.

these integers are called file descriptors

lsof (list open files) will show you a process's open files

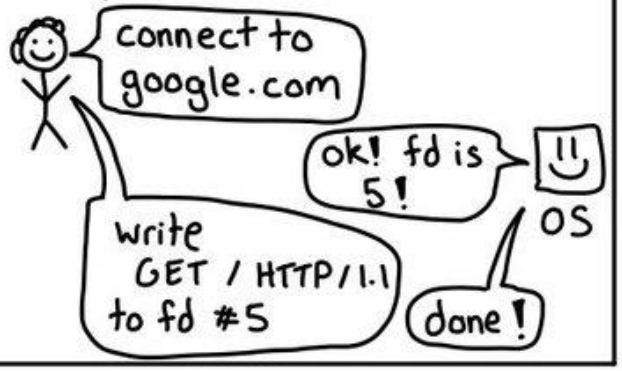
```
$ lsof -p 4242   ← PID we're interested in
FD  NAME
0   /dev/pts/tty1
1   /dev/pts/tty1
2   pipe:29174
3   /home/bork/awesome.txt
5   /tmp/
↑
```

FD is for file descriptor

file descriptors can refer to:

- → files on disk
- → pipes
- → sockets (network connections)
- → terminals (like xterm)
- → devices (your speaker! /dev/null!)
- → LOTS MORE (eventfd, inotify, signalfd, epoll, etc etc)

When you read or write to a file/pipe/network connection you do that using a file descriptor

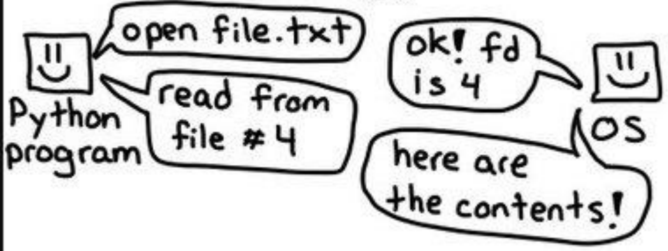


Let's see how some simple Python code works under the hood:

Python:

```
f = open("file.txt")
f.readlines()
```

Behind the scenes:



(almost) every process has 3 standard FDs

stdin → 0
stdout → 1
stderr → 2

"read from stdin"
means
"read from the file descriptor 0"
↗
could be a pipe or file or terminal

Reading from a file (1)

- `read` system call
  - `ssize_t read (int fd, void *buf, size_t len);`
- `fd`: file descriptor (obtained via `open` or `creat`)
- `buf`: points to a memory zone that will receive the bytes read from the file
- `len`: maximum number of bytes that caller wants to read from the file
  - `buf` must points to a memory zone that has, at least, `len` bytes

- Return value of `read` >>

Reading from a file (2)

- `read` Returns
  - -1 on error
  - Number of read bytes on success
  - 0 if the file is at EOF (*end-of-file)*
- The number of read bytes can be less than the parameter `len`.
  - Why?
    - We might be at / near the end of the file…

Non-blocking read

- Open can be called with the flag O_NONBLOCK
  - It means that I/O on the descriptor is non-blocking
  - Often used for *sockets*
- Using read over a non-blocking descriptor can yield the `EAGAIN errno` situation
  - It means that there is, currently, no data to read
  - Therefore, for non-blocking I/O this situation needs to be handled (it is not an error)
    - `read` call returns -1
    - `errno` is set by the system to `EAGAIN`
-

Writing to a file (1)

- The `write` system call

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

- **fd**: file descriptor
- **buf**: address of 1st byte to write to file pointed by fd
- **count**: number of bytes to write to file pointed by fd
- After the write operation, the file pointer is updated accordingly

- `write` possible return values >>

Writing to a file (2)

- The `write` system call returns a `ssize_t`
  - ssize_t aims to represent a size (positive value)
  - ssize_t also has negative values to represent errors
- `write` returns
  - Number of bytes writen to file pointed by fd
  - The number of bytes written to a file can be less than `count` (3rd parameter of `write`)
    - Example: the disk where the file is located is full...

escola superior de tecnologia e gestão
instituto politécnico de leiria

Writing to a file (3)

- How can we test code under a full device?
  - Linux has a special device file
    - /dev/full
  - The special file `/dev/full` always returns a “full” on write attempts
- Example

```
  ls > /dev/full
ls: write error: No space left on device
```

`write`: append mode

- A descriptor can be obtained in append mode (specifying `O_APPEND` in the flags of `open`)
  - It makes the `write` operation to be performed **always** at the end of the file
    - Even if the files is opened for writing (appending) by multiple processes
    - Useful for *logging operations*
  - Example:
    - Process A writes to file `general_log`
    - Process B also writes to file `general_log`
    - If process A and process B use O_APPEND, the system ensures that each `write` operation is done at the end

Synchronized I/O

- The operating system (OS) does not immediately write to the file (i.e., device where the file is located)
- How can the programmer force an immediate write operation?
  - `fsync` system call
    - `int fsync (int fd);`
  - The pending writes are commited to the storage
    - Note: the device (e.g., hard/SSD disk) might have *write caches* and thus defer write operations

Closing files

- Closing a file
  - Breaks the mapping between the descriptor and the file
  - `close` system call
    - `int close (int fd);`
  - It is considered wise to check the return value of `close`
    - A low level I/O error report about a previous call might get detected only when close is called

```
if( close(file_descriptor)== -1){
    fprintf(stderr,"error in close: %s (errno=%d)\n",
        strerror(errno), errno);
}
```

The `errno` variable (#1)

- Integer (global) variable: `int errno;`
- Usage of `errno` requires **#include <errno.h>**
- Set when an error occurs
  - System calls (open, etc.)
  - Some functions of the C library
- The value of errno points out what went wrong
- Its value is meaningful only when the return value of the call indicated an error
  - -1 from most system calls
  - -1 or NULL from most library functions)

The `errno` variable (#2)

- Some of the possible values of errno (`man errno`)
  - EACCES ➔ Permission denied
  - ENFILE ➔ Too many open files in system

  (…)
- ENFILE
  - preprocessor constant
- How to get its numerical value?

```
#include <errno.h>
printf("ENFILE=%d\n", ENFILE);
```

- Result: `ENFILE=23`

The `errno` variable (#3)

- The symbolic constant gives the numerical value
  - Example: ENFILE → 23
- How to get a meaningful error message?
  - char *strerror(int errnum);

```
#include <errno.h>
#include <string.h>
printf("ENFILE=%d\n", ENFILE);
printf("Error string for ENFILE:'%s'\n",strerror(ENFILE));
```

- Result:

```
ENFILE=23
Error string for ENFILE:'Too many open files in system'
```

`seek` operations

- I/O occurs (usually) in a linear way within a file
  - Example
    - `read` from byte 0 to 256. Then, next byte read with be byte 257 and so on
    - The same goes on for `write` operations
- What about if a process needs to have a different *file position*?
- `lseek` system call
  - `off_t lseek (int fd, off_t pos, int origin);`

`lseek` system call >>

`lseek` (1)

- Prototype (`man 2 lseek`)
  - `off_t lseek (int fd, off_t pos, int origin);`
- **`pos`**: movement to be done to the file position
- **Origin:** can be
  - SEEK_CUR: from *current file position + pos*
    - Pos can be 0, >0 or <0
  - SEEK_END: from *end of file + pos*
    - Pos can be 0, >0 or <0
  - SEEK_SET: file position is set to *pos*
- return: current file position for descriptor **fd**

`lseek` (2)

- Wait, we can really do this?

```
int pos = 256;
lseek(descriptor, pos, SEEK_CUR);
```

- This means to set the file position 256 bytes ***beyond*** the end of the file…
- No effect if after we perform a read operation
  - EOF is returned
- But, next write operation will be performed at EOF+256
  - The will be a gap of 256 bytes in the file ("file hole")
  - Automatically padded by the OS (null byte is written in the "empty" space of the file)
    - sparse file

`lseek` (3)

- How can we get the current position of a file pointer?
  - We perform a lseek with SEEK_CUR and 0 (zero) offset

```
current_pos = lseek(descriptor,0, SEEK_CUR);
```

  - The current position of the file pointer is returned by lseek
    - Variable `current_pos` in the above example

close

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

- Close a file descriptor
- Every file opened by a program needs to be closed
- If a program fails to close a no longer used file descriptor
  - The descriptor of the file exists until the end of the process
- A program can run out of file descriptors

Example

example - too many open files

```
int open_file(const char *filename);
int open_file(const char *filename){
          return open(filename,O_RDONLY);
}
#define NUM_ELMS     (2048)
int main(void){
          char *filename_S = "a.txt";    // file “a.txt” must already exist
          int file_descriptors_V[NUM_ELMS];
          size_t i = 0;
          while( i < NUM_ELMS  ){
                    if( (i % 10) == 1 ){
                              printf(“i=%zu\n”, i);
                    }
                    file_descriptors_V[i] = open_file(filename_S);
                    if( file_descriptors_V[i] == -1 ){
                              fprintf(stderr,”ERR: can’t open file #%zu:’%s’\n”,
                                                  i, strerror(errno));
                              exit(1);
                    }
                    i++;
          }
          return 0;
}
```

Output

```
(...)
i=1001
i=1011
i=1021
ERR: can't open file #1021:'Too many open files'
```

unlink - delete file

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

- Deletes a file
  - In fact, it deletes the name
    - If the name is the last link to the file
      - and <u>**no**</u> process has the file open: **it deletes the file**
      - and at least one process has the file open: **it only deletes the file when all instances of the files are closed**

rename and **rmdir**

```
#include <stdio.h>
int rename(const char *oldpath, const char *newpath);
```

- Renames a file
  - "oldpath" to "newpath"

```
#include <unistd.h>
       int rmdir(const char *pathname);
```

- Removes a directory
  - The directory must be empty

Unix: `lsof` command

- Unix's command to "**Li**S**t** **O**pen **F**iles"
- Examples
  - lsof: list all open descriptors
  - sudo lsof +D /tmp
    - List all processes that have one (or more) file opened in /tmp
  - sudo lsof -u user
    - List all files/descriptors opened by user "user"
  - sudo lsof -i
    - List processes that have open ports (UDP or TCP)
- **Remember**: almost everything in unix is acessed through a descriptor

References

- *"File I/O", Chapter 2 - Linux System Programming,* Robert Love, 2nd Edition, O'Reilly, 2013
- man: *Unix Electronic Manual*
  - *`man 2 function_name`*
    - *`open, close, read, write, lseek,…`*
  - *`man -k word`*
  - *`man 1 lsof`*