System calls

Gaël Thomas

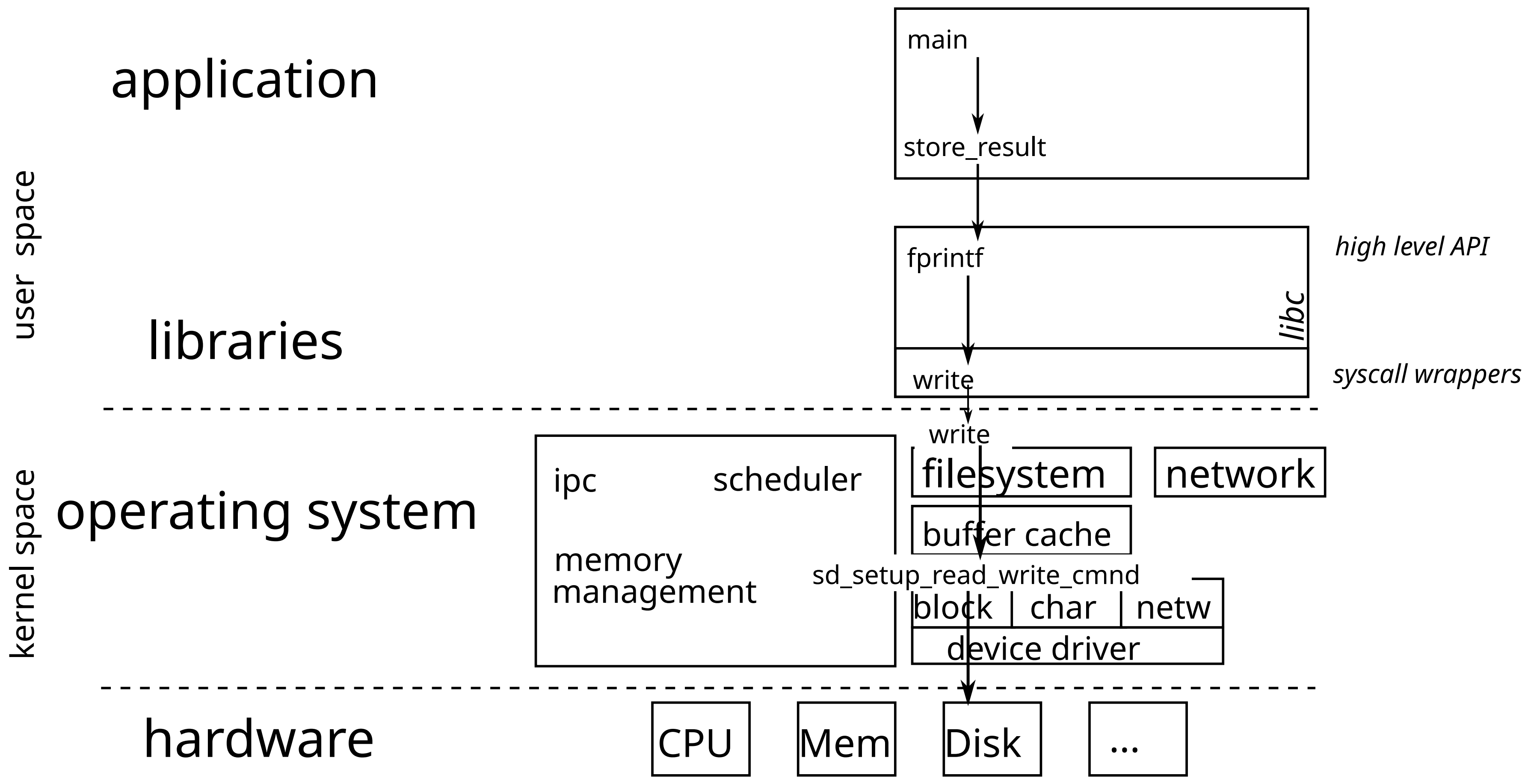
CSC4508 – Operating Systems

2022–2023

1 Operating systems

- Features
  - Offers a unified **programming interface** to the developer
  - Hides hardware implementation details
  - Allows you to run multiple **processes** on a **processor**
- Composition
  - A **library** called **kernel** (*noyau* in French)
    - Unified programming interface (open, fork, etc.)
    - Defined by specifications (System V, POSIX, Win32. . .)
  - + A **set of programs** allowing to interact with the core
    - `ls`, `cp`, `X`, `gnome`, etc.

1.1 Operating systems (2/2)



1.2 Testing the return value of system calls and functions

- You must **always** test the return value of a system call and deal with errors
  - Prevent the propagation of errors (the discovery of the error can take place much later)
    - see the *fail-fast* approach presented in CSC4102
- `errno`: external variable indicating the cause of the last error
  - The `ERRORS` section in a function manual describes the possible causes of error.

2 Stack frames

- Each function call creates an *stack frame*
- A *stack frame* contains
  - local variables
  - a backup of the modified registers
  - the arguments of the function (specific to 32-bit x86 architectures)
  - the return address of the function (specific to x86 architectures)

2.1 Content of a *stack frame*

- A *stack frame* is defined by
  - a base address that indicates where the *frame* begins (the `rbp` register on `x86`)
  - the address of the top of the stack (the `rsp` register on `x86`)
- Function entry:
  - Save `rbp` (using `push rbp`)
  - Reset `rbp` (using `mov rbp, rsp`)
- Function exit:
  - Restore of the old `rbp` (`pop rbp`)
  - Jump to the return address (`ret`)

2.2 Buffer overflow

- (in French *dépassement de tampon*)
- Writing data outside the space allocated for a buffer
- Risk of overwriting other data
- Security vulnerability: overwriting data may change the behavior of the application

2.2.1 Stack overflow

- Using a *buffer overflow* to change the program execution flow
- The return address of a function is on the stack

$\Longrightarrow$ possibility of *choosing* the code to be executed afterwards

2.2.2 How to prevent buffer / stack overflow?

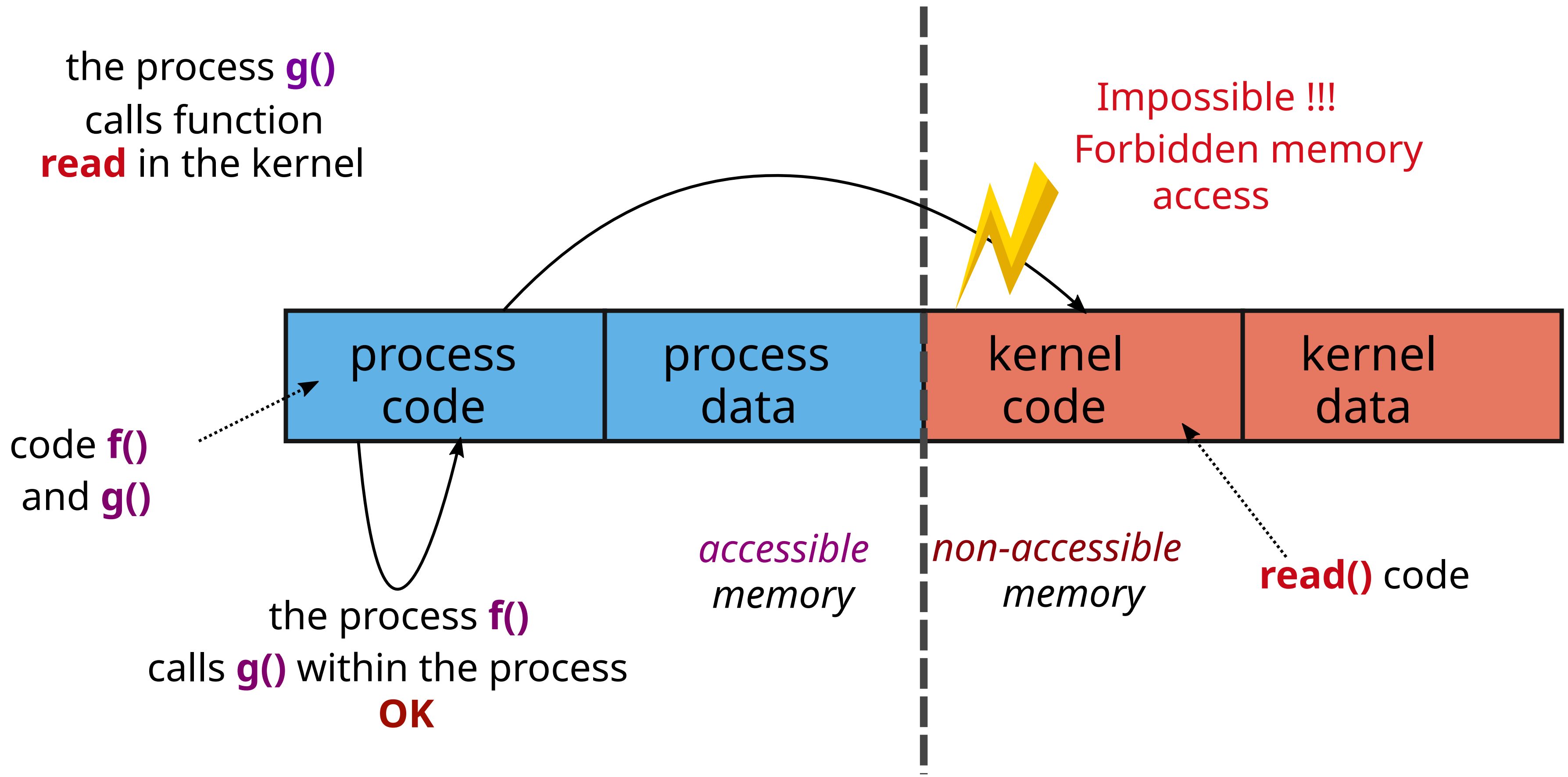
- Check the boundaries of buffers
  - done automatically in Java
  - not done in C / C ++ because it is too expensive
- Do not use the " *unsafe* " functions (`strcpy`, `gets` ...)
  - Use their safe counterpart instead (`strncpy`, `fgets` ...)
- Non-executable stack (enabled by default by Linux)
  - avoid the execution of an arbitrary code
- *Stack canaries*
  - A *canary* (a specific value) is placed on the stack when entering a function
  - If when exiting the function, the *canary* has been modified, there has been a *stack overflow*
  - Use the `-fstack-protector-all` option in gcc
- *Address space layout randomization* (ASLR) (enabled by default by Linux)
  - load the application code to a random address

3 User/system interface

- The kernel must *protect* from processes
  - To avoid bugs
  - To avoid attacks
- For this, the **processor** offers two operating modes
  - The ***system* mode**: access to all the memory and to all the processor instructions
  - The ***user* mode**: access only to the process memory and to a restricted set of instructions
    - In particular, no direct access to peripherals and instructions that manage the permissions associated with the memory

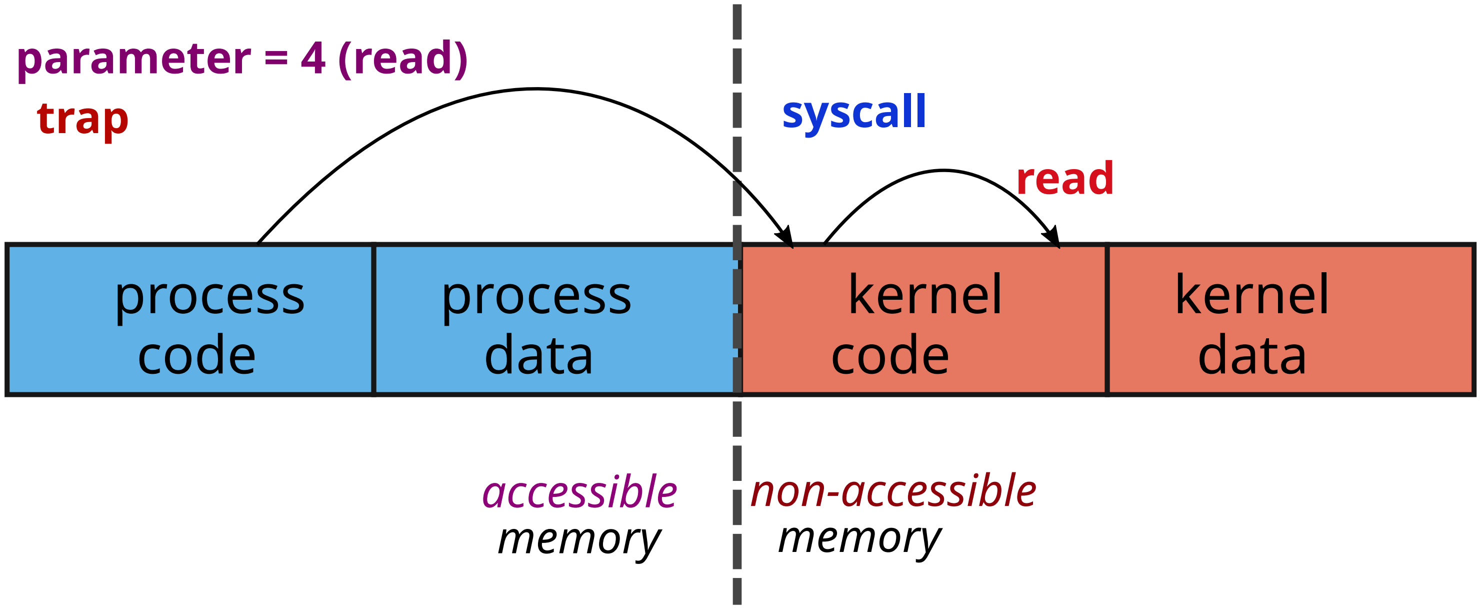
3.1 User/system interface

- Problem: how do you call a kernel function when you can't access its memory?



3.2 User/system interface

- Solution: special processor instruction called `trap`
  - The kernel associates the address of a `syscall` function to `trap`
  - To call a kernel function
    - The process gives the function number to call via a **parameter**
    - The process executes the `trap` instruction
    - The processor changes mode and executes the `syscall` instruction
    - syscall uses the **parameter** to select the kernel function to be executed



Bibliography