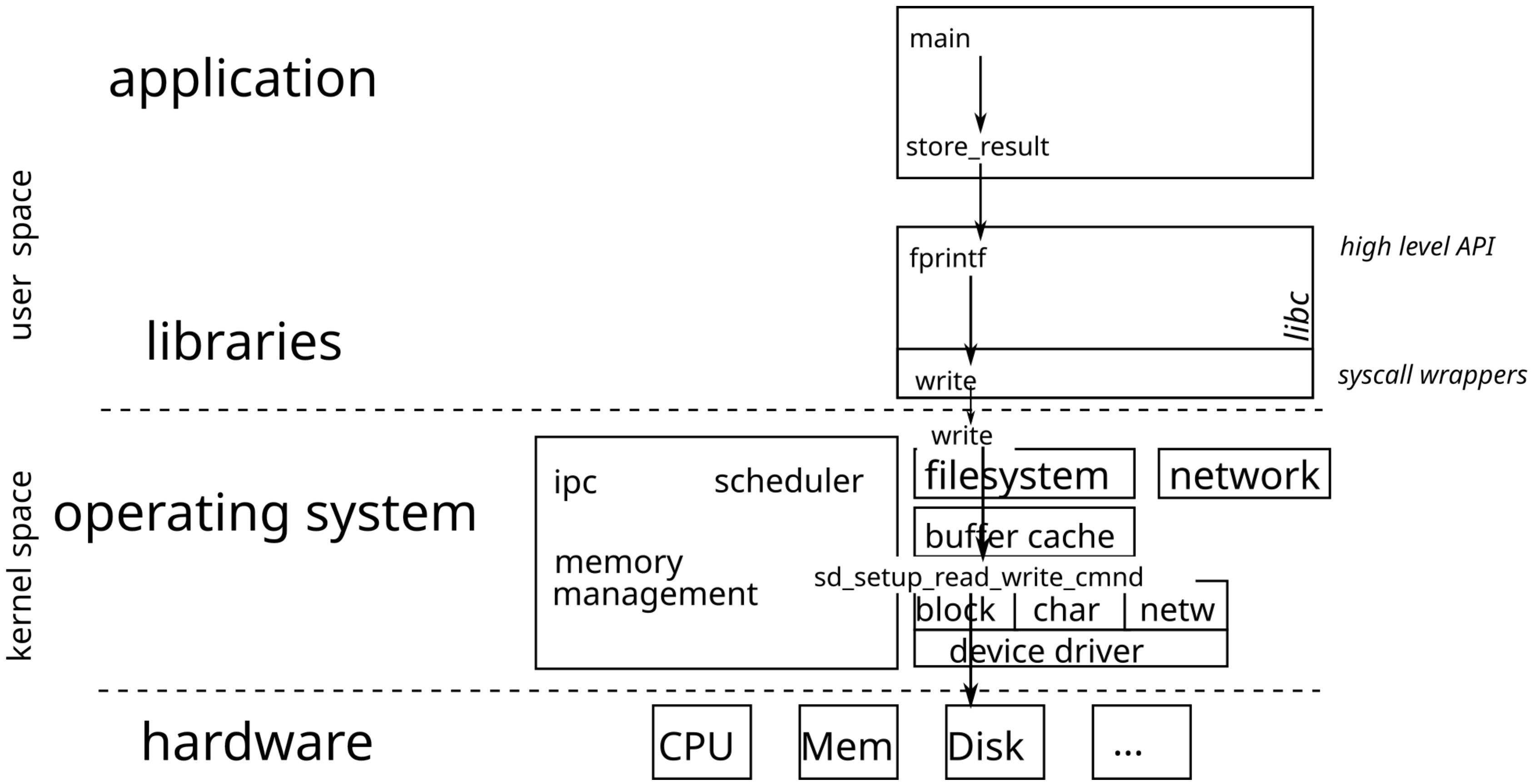
System calls

Gaël Thomas
Mathieu Bacou

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.

Operating systems (2/2)



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.

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, if there are too many to fit in the registers
  - the return address of the function

Content of a *stack frame*

- A *stack frame* is defined by
  - the address of the top of the stack (the `sp` register)
  - a base address that indicates where the frame begins
    - on x86, it is kept in the `rbp` register
    - on RISC-V, the compiler keeps track of it when generating assembly
- Function entry:
  - decrement `sp` to make space to save registers, and for local variables
  - save registers
  - save `ra`
- Function exit:
  - restore saved registers
  - restore `ra`
  - increment `sp` back to its previous value
  - jump back to `ra`

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

Stack overflow

- Using a *buffer overflow* to change the program execution flow
- The return address of a function is on the stack -> possibility of *choosing* the code to be executed afterwards

How to prevent buffer / stack overflow?

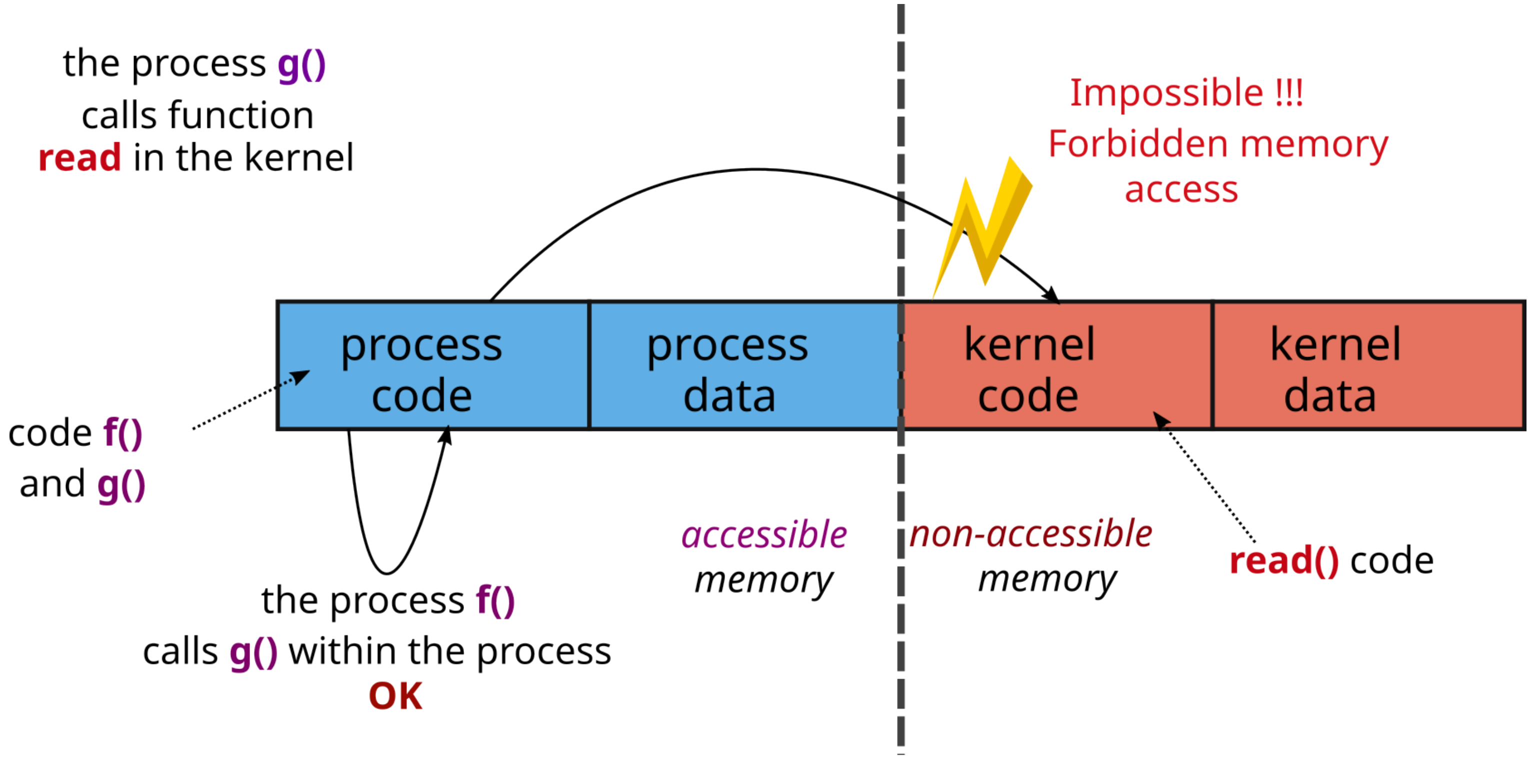
- Check the boundaries of buffers
  - done automatically in Java
  - not done in C / C ++ because it is too expensive
- Do not use the *unsafe* functions (`strcpy`, `gets` …)
  - Use their safe counterpart instead (`strncpy`, `fgets` …)
- Non-executable stack (enabled by default by Linux)
  - avoid the 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

User/system interface

- The kernel must *protect* itself 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

User/system interface

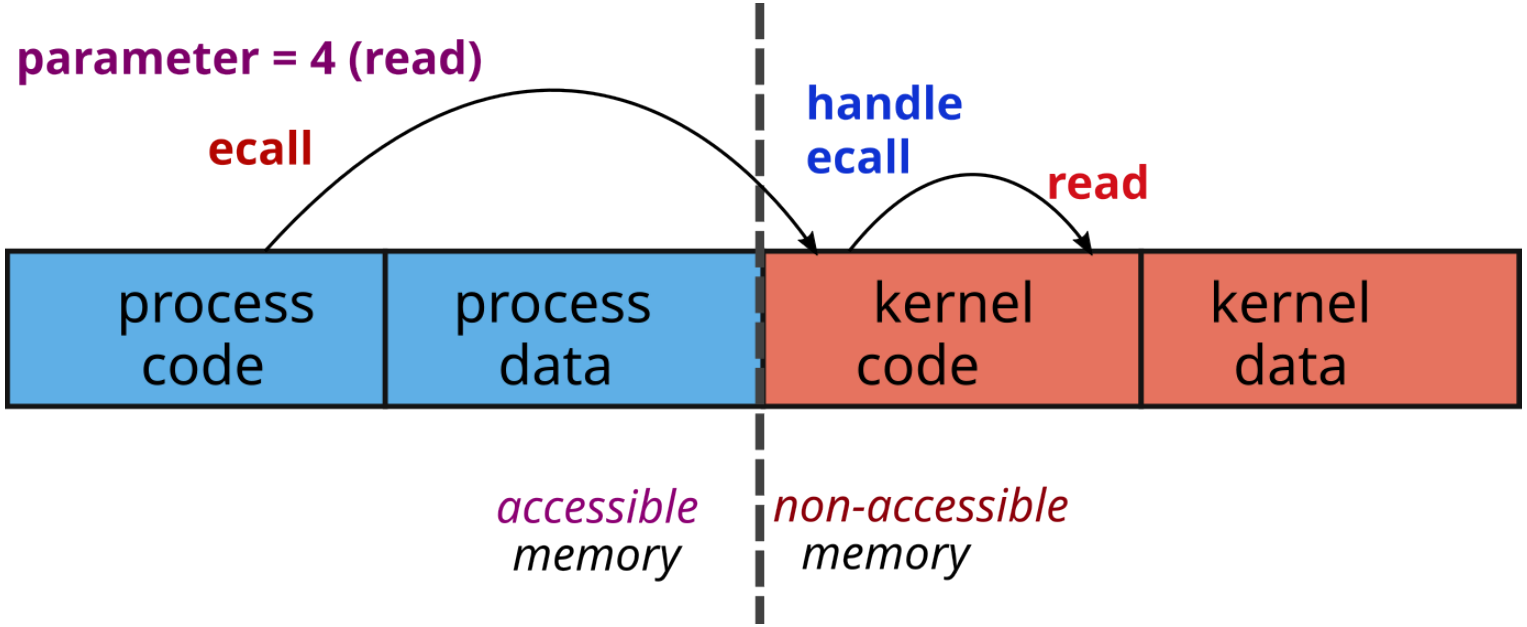
- Problem: how do you call a kernel function when you can’t access its memory?



the process **g()** calls function **read** in the kernel
Impossible !!! Forbidden memory access
process code
process data
kernel code
kernel data
code **f()** and **g()**
accessible memory
non-accessible memory
read() code
the process **f()** calls **g()** within the process
OK

User/system interface

- Solution: special processor instruction to call into system mode
  - The kernel associates the address of a `syscall` function to handle `ecall`
  - To call a kernel function
    1. The process gives the function number to call via a **parameter**
    2. The process executes the `ecall` instruction
    3. The processor changes mode and executes the `ecall` handler
    4. the handler uses the **parameter** to select the kernel function to be executed



Bibliography

Aleph, One. 1996. “Smashing the Stack for Fun and Profit.” *Phrack #49*.