System calls

Gaël Thomas Mathieu Bacou

Contents

Operating systems **1**
Operating systems (2/2) . . . 2
Testing the return value of system calls and functions . . . 2
How to check the return value of a function and handle errors? . 3
Displaying the cause of an error . . . 3
Generic error handling . . . 4
Debugger . . . 4

Stack frames **5**
Content of a *stack frame* . . . 5
Function call convention . . . 6
Buffer overflow . . . 8
Example . . . 8
Security vulnerabilities . . . 9
Stack overflow . . . 9
How to prevent buffer / stack overflow? . . . 10
Hardware privilege levels . . . 11
User/system interface . . . 12
User/system interface . . . 12
User/system interface . . . 12
`x86_32` . . . 13
`x86_64` . . . 13
ARM 64 bits . . . 13
`RISC-V` . . . 13

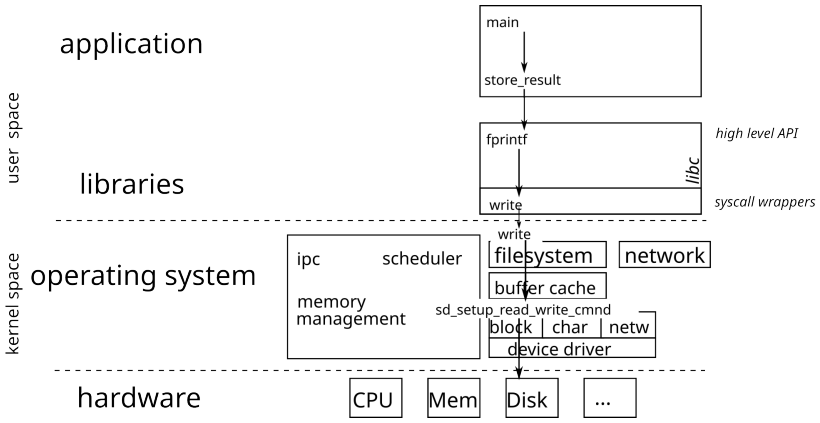
Bibliography **14**

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)



The operating system is responsible for operating various hardware. It, therefore, includes drivers capable of interacting with a particular material. The different drivers for the same type of peripheral offer the same interface, which allows the upper layers of the OS to use the hardware interchangeably.

The transition from user space to kernel space is done via a system call (*syscall*). The kernel processes the request for the application and returns a positive or zero integer on success, and `-1` on failure.

From the application point of view, system calls are exposed as functions (defined in `libc`) in charge of executing the system call.

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.

Testimony of a former ASR student: "Without insistence from [the CSC4508 teachers], it would not have jumped out to us so quickly that the problems (in the robotics championship) came from a lack of errors handling on a code that had not been carefully proofread".

How to check the return value of a function and handle errors?

The macro `void assert (scalar expression)` tests the expression passed in parameter and, if false, displays a message error and terminates the program (with the `abort ()` function):

```
struct stat buf;
int rc = stat(file, &buf);
assert(rc>=0);
// -> in case of an error, prints:
//   appli: appli.c:12: main: Assertion `rc>=0' failed.
//   Abandon
```

However, the macro should be used with caution because it is disabled when the program is compiled in optimized mode (with `gcc -O3` for example).

So it is better to test the return code, display a message describing the error, and possibly terminate the process.

```
struct stat buf;
int rc = stat(file, &buf);
if(rc < 0) {
  fprintf(stderr, "Error\n");
  exit(EXIT_FAILURE); // or abort();
}
```

Displaying the cause of an error

The `errno.h` file lists standard errors. The manual of each system call (see `man 2 function`), and of each function (`man 3 function`) indicates, in the `ERRORS` section, the different error codes that may be returned.

The error message associated with a value of `errno` can be obtained with `strerror ()` or `perror ()`:

```
struct stat buf;
int rc = stat(file, &buf);
if(rc < 0) {
  fprintf(stderr, "Error while accessing file '%s': %s\n", file, strerror());
  // -> message "Error while accessing file 'plop': No such file or directory"
  exit(EXIT_FAILURE);
}
```

or

```
struct stat buf;
int rc = stat(file, &buf);
if(rc < 0) {
  perror("Error while accessing file");
  // -> message: "Error while accessing file: No such file or directory"
  exit(EXIT_FAILURE);
}
```

Generic error handling

It is possible to define a macro displaying an error message and indicating where the error occurred. For example:

```
#define FATAL(errnum, ...) do {                                  \
    fprintf(stderr, "Error in %s:%d:\n", __FILE__, __LINE__); \
    fprintf(stderr, __VA_ARGS__);                                \
    fprintf(stderr, ": %s\n", strerror(errnum));                 \
    abort();                                                     \
  } while(0)

int main(int argc, char**argv) {
  char *file = argv[1];
  struct stat buf;
  int rc = stat(file, &buf);
  if(rc < 0) {
    FATAL(errno, "Cannot access file '%s'", file);
  }
  return EXIT_SUCCESS;
}
// affiche:
//  Error in fatal.c:21:
//  Cannot access file 'plop': No such file or directory
//  Abandon
```

Debugger

When a program calls the `abort ()` function in order to terminate the process, a `core dump` file (that describes the process when the error occured) can be generated in order to debug the program with `gdb`.

To activate the generation of a *core dump*, run the command `ulimit -c unlimited`. Therefore, the function `abort ()` generates a *core dump* which can be supplied to `gdb`:

```
$ ./fatal  plop
Error in fatal.c:21:
```

```
Cannot access file 'plop': No such file or directory
Abandon (core dumped)

$ gdb ./fatal core
GNU gdb (Debian 8.1-4+b1) 8.1
[...]
Reading symbols from ./fatal...(no debugging symbols found)...done.
[New LWP 11589]
Core was generated by `./fatal plop'.
Program terminated with signal SIGABRT, Aborted.
#0  __GI_raise (sig=sig@entry=6) at ../sysdeps/unix/sysv/linux/raise.c:50
50      ../sysdeps/unix/sysv/linux/raise.c: Aucun fichier ou dossier de ce type.
(gdb) bt
#0  __GI_raise (sig=sig@entry=6) at ../sysdeps/unix/sysv/linux/raise.c:50
#1  0x00007ffff7dfb535 in __GI_abort () at abort.c:79
#2  0x0000555555555232 in main (argc=2, argv=0x7fffffffdcd8) at fatal.c:21
```

On Linux distribution running systemd, the core dumps are managed by `coredumpctl`:

- `coredumpctl list` prints the list of all the available core dumps
- `coredumpctl info` display basic information (such as the command line that invoked the program, or the backtrace of the threads when the crash occured) about the last core dump.
- `coredumpctl debug` invokes a debugger (eg. `gdb`) on the last core dump

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`

Function call convention

Depending on the CPU architecture (and sometimes the compiler), the way of making a function call may vary.

`x86` 32 bits On 32-bit `x86` architectures, parameters are placed on the stack so that the first argument is located at address `ebp + 8`, the second at address `ebp + 12` (if the first argument is stored on 4 bytes), etc.

The return address (i.e. the address of the instruction to run after function) is stored on the stack at the address `ebp+4`.

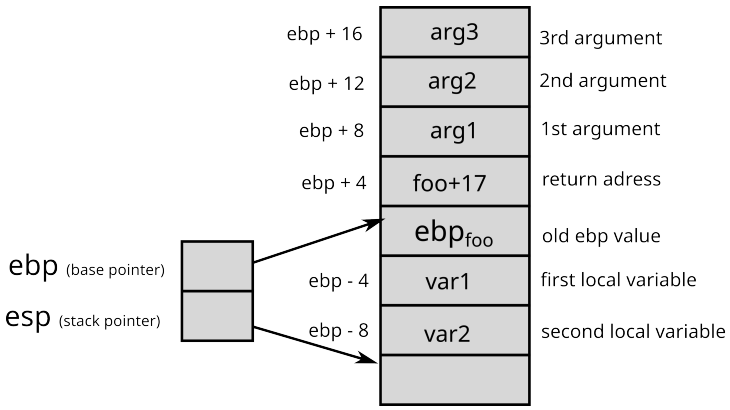


Figure 1: Stack frame on 32-bit `x86` architectures

`x86` 64 bits On 64-bit `x86` architectures, the parameters are passed via the `rdi`, `rsi`, `rdx`, `rcx`, `r8` and `r9` registers. If there are more than 6 parameters, the next parameters are placed on the stack.

`Arm` On `Arm` architectures, parameters are passed via registers (`x0` to `x7` on `Arm` 64 bits). The return address is also stored in a register.

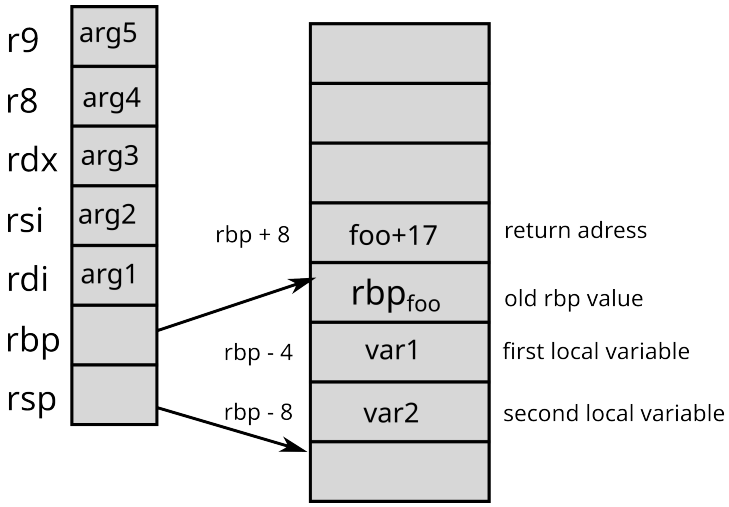


Figure 2: Stack frame on 64-bit `x86` architectures

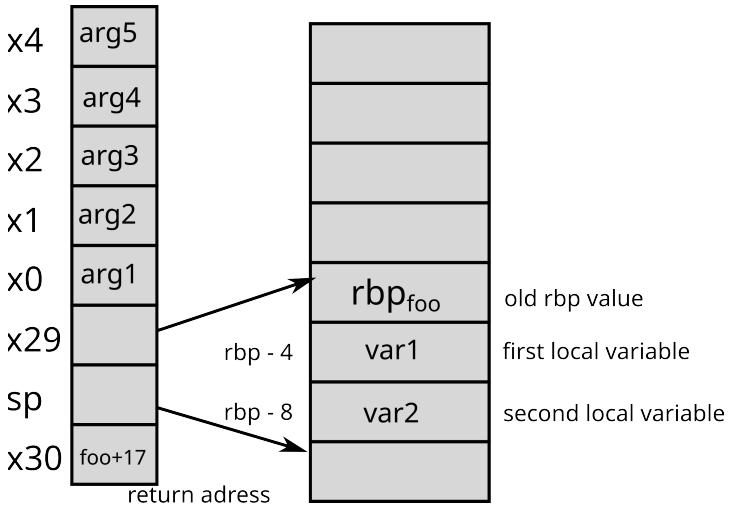


Figure 3: Stack frame on 64-bit `Arm` architectures

`RISC-V` On `RISC-V` architectures, parameters are passed via registers (`a0` to `a7`) like `Arm`. If there are more parameters, or their values do not fit in 64 bits registers, they are placed on the stack. The return address is also stored in a register. The address of the previous stack frame is not kept, the compiler issues an instruction to increment the stack pointer back to the previous stack frame. It means the preamble and epilogue of compiled functions are slightly more complicated [riasanovsky].

Buffer overflow

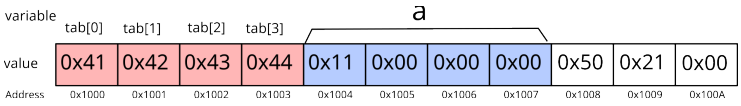
- (in French *dépassement de tampon*)
- Writing data outside the space allocated for a buffer
- Risk of overwriting other data
- Security vulnerability: overwriting data may change the behavior of the application

Here is an example of *buffer overflow*:

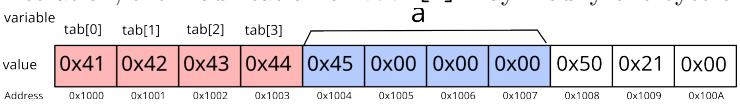
```c
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char**argv) {

  int N = 4;
  char tab[N];
  int a = 17;

  for(int i=0; i<=N ; i++) {
    tab[i] = 'a'+i;
  }

  printf("tab = {%c, %c, %c, %c}\n", tab[0], tab[1], tab[2], tab[3]);
  printf("a = %d\n", a);
  return 0;
}
```

Example

Here, the bug comes from the loop in charge of filling the array which iterates too many times (because of `<=`. After the first 4 iterations, here is the memory status:



During the fifth iteration, the modification of `tab [4]` may modify one byte of the variable `a`:



The variable `a` is therefore no longer equal to `17`, but `69` (or `0x45`).

Security vulnerabilities

Buffer overflow bugs are potentially serious for the security of a system, because depending on an input (e.g. a string entered by the user), the bug may modify the behavior of the application (without necessarily *crashing* the program). In our example, if the variable `a` matches the username, the bug could allow attackers to pretend to be someone else (for example, an administrator)!

Buffer overflows are among the most common security vulnerabilities. To be convinced of this, just look for the vulnerability announcements that mention "buffer overflow" (around 780 faults in 2017)

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

Example Here is an example of *stack overflow*:

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

void foo(char* str) {
  char new_str[16];
  strcpy(new_str, str);
  printf("new_str = %s\n", new_str);
}

int main(int argc, char**argv) {

  foo(argv[1]);
  printf("Back in main()\n");
  return 0;
}
```

Here, the `foo` function does not check that `new_str` is large enough to hold `str`. So if `str` is too long, `strcpy` overflows and may overwrite the return address of `foo`.

Here is an example of execution leading to an *stack overflow*:

```
  $ gdb ./stack_overflow
  (gdb) r coucouAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: stack_overflow coucouAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
new_str = coucouAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA

Program received signal SIGSEGV, Segmentation fault.
0x000055555555518e in foo (str=0x7fffffffe03e "coucou", 'A' <repeats 83 times>) at stack_ove
9       }
(gdb) bt
#0  0x000055555555518e in foo (str=0x7fffffffe03e "coucou", 'A' <repeats 83 times>) at stack
#1  0x4141414141414141 in ?? ()
#2  0x4141414141414141 in ?? ()
#3  0x4141414141414141 in ?? ()
#4  0x4141414141414141 in ?? ()
#5  0x4141414141414141 in ?? ()
#6  0x4141414141414141 in ?? ()
#7  0x4141414141414141 in ?? ()
#8  0x4141414141414141 in ?? ()
#9  0x0000555555550041 in ?? ()
#10 0x0000000000000000 in ?? ()
(gdb)
```

Here, we observe that when exiting the `foo` function, the program tries to execute the instruction located at the address `0x4141414141414141` (`0x41` is the hexadecimal value of `'A'`), which generates an error.

We could exploit the bug by inserting in `argv [1]` the address of the function `void bar (int a, int b)` as well as its parameters [aleph1996].

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

Hardware privilege levels

The implementation of the two operating modes is dependent on the processor architecture.

`x86` On `x86`, there are four privilege levels called protection rings. Today, only two are used:

- the lowest one is the most privileged, this is the ring 0: kernel level
- the highest one is the least privileged, this is the ring 3: applications level

Two intermediate ones were used in a model where device drivers would run separately from the kernel.

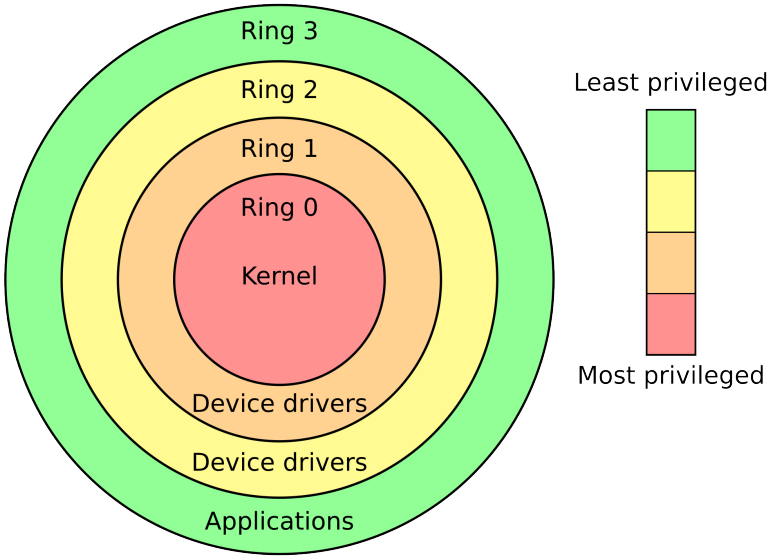


Figure 4: Privilege levels for `x86`. (c) Hertzsprung at English Wikipedia.

`RISC-V` Depending on the implemented platform, `RISC-V` uses up to three levels (sometimes called privilege modes).

From most privileged to less privileged:

- Machine mode (`M`): the level at which the firmware runs

- Supervisor mode (`S`) : the kernel level
- User mode (`U`) : the level for normal user applications

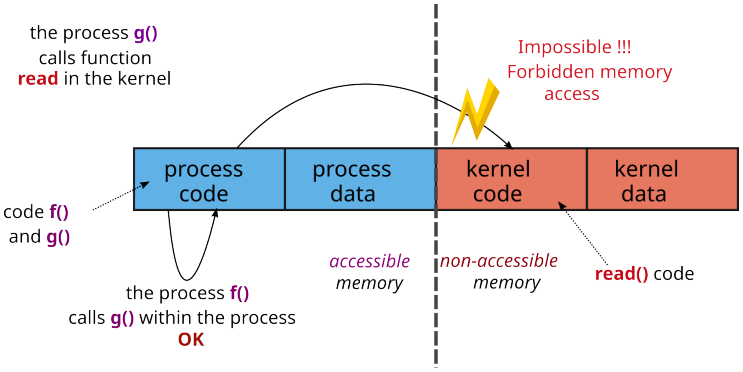
User/system interface

- 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

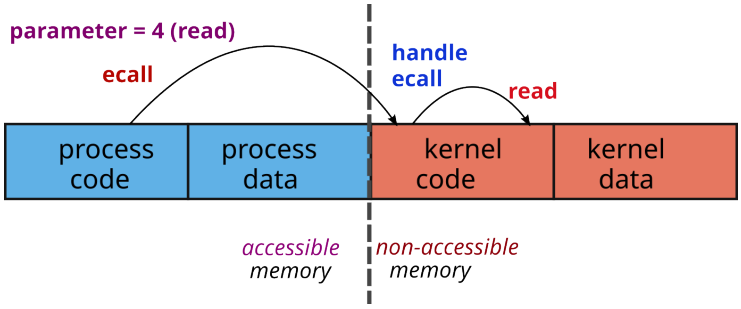
- Problem: how do you call a kernel function when you can't access its memory?



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



Depending on the type of processor, the way of making a system call may vary. The way to pass the parameters is a convention which can vary from one OS to another. For example, for Linux:

`x86_32`

- The parameters of the system call are stored in the `ebx` `ecx`, `edx`, `esi`, `edi`, and `ebp` registers;
- The syscall number is loaded into the `eax` register;
- Switching to kernel mode is done by generating the interrupt 128: `INT 0x80`;
- At the end of the system call, the return value is stored in the `eax` register.

`x86_64`

- The parameters of the system call are stored in the `rdi`, `rsi`, `rdx`, `rcx`, `r8`, and `r9` registers;
- The system call number is loaded into the `rax` register;
- Switching to kernel mode is done with the `syscall` instruction ;
- The return value of the system call is stored in the `rax` register.

ARM 64 bits

- The parameters of the system call are stored in the `x0` to `x5` registers;
- The system call number is loaded in the `x8` register;
- Switching to kernel mode is done with the `svc 0` instruction;
- The return value of the system call is stored in the `x0` register.

`RISC-V`

- The parameters of the system call are stored in the `a0` to `a5` registers;
- The system call number is loaded in the `a7` register;

- Switching to kernel mode is done with the `ecall` instructions
  - `ecall` is the generic instruction for a privilege level to call into the immediately-lower privilege level, i.e., from `U` to `S` and from `S` to `M`;
- The return value of the system call is stored in the `a0` register.

Bibliography

[riasanovsky] Riasanovsky, Nick. "Understanding RISC-V calling convention." EECS Department, University of California, Berkeley, Tech. Rep.[Online]. Available: https://inst.eecs.berkeley.edu/~cs61c/resources/RISCV_Calling_Convention.pdf

[aleph1996] Aleph, One. "Smashing the stack for fun and profit" Phrack #49, 1996. [Online]. Available: https://phrack.org/issues/49/1