Princípios da Computação

The C programming language — a quick intro

Storing addresses: pointers

Pointer

- A **pointer** is a variable that **stores a memory address**.
- It *points* to the memory location of a data item…
- … rather than storing the data item.

Declaring a pointer

- The * identifies the variable as a pointer.

- The type of the pointed item is indicated before the asterisk.

  - It serves to tell the compiler how to treat the item in the memory location.

```
int * p1;      /* Pointer to an integer. */
float * p2;    /* Pointer to a float. */
```

Accessing the pointed item

- The * retrieves the pointed item.

```
int a = 7;

int * p = &a; /* p points to variable a */

printf("a: [%p] %d\n", &a, a);
printf("p: [%p] %p -> %d\n", &p, p, *p);
```

```
a: [0x20000] 7
p: [0x1999C] 0x20000 -> 7
```

Functions that modify the input arguments

- Functions use local copies of the received arguments.
    - Therefore, they are not able to change the original values of the function caller.
- However, if it receives the address of the item, the function can modify it!
    - The function can go directly to the source.

Functions that modify the input arguments

```
int a = 5;
int b = 10;

swap(&a, &b);

printf("a = %d\n", a);
printf("b = %d\n", b);
```

```
a = 10
b = 5
```

How is this possible???

Functions that modify the input arguments

```
void swap(int * p1, int * p2)
{
   int aux;

   aux = *p1;
   *p1 = *p2; /* Modifying item pointed by p1. */
   *p2 = aux; /* Modifying item pointed by p2. */
}
```

Pointer as an iterator

- Pointers can be incremented (and decremented).
- The pointer advances (or recedes) the size of the type being pointed to.
- Useful for iterating over arrays.

Pointer as an iterator

```
int v[5] = {1, 2, 3, 4, 5};
int * p = v;

for(int i = 0; i < 5; i++) {
   printf("%d\t", *p);
   p++; /* p advances 4 (sizeof int) bytes. */
}
```

```
1    2    3    4    5
```

System calls

isep Instituto Superior de Engenharia do Porto

System calls

- A system call is an interface through which user-level processes interact with the operating system kernel.

- System calls provide a way for applications and user-level programs to request services from the operating system kernel.

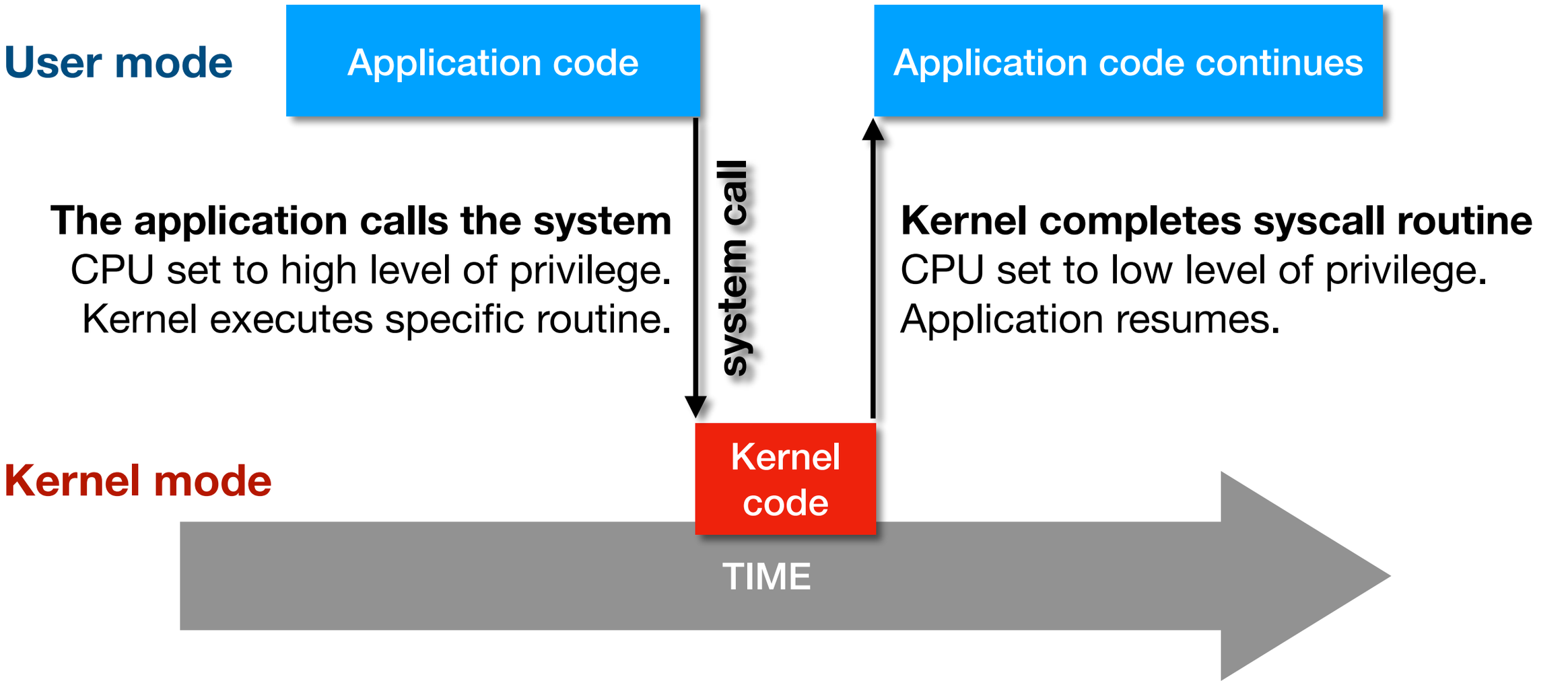
  - Hardware operations, creation of processes, interprocess communication and synchronisation, etc.

  - The Linux kernel provides more than 300 syscalls.

User mode vs. kernel mode

- Applications execute in **user mode**.
    - CPU lowest level of privilege: no access to hardware, restricted access to memory, etc.
- Kernel executes in **kernel mode**.
    - CPU highest level of privilege: direct access to hardware, full access to memory, etc.

The concept


User mode
Application code
The application calls the system
CPU set to high level of privilege.
Kernel executes specific routine.
system call
Application code continues
Kernel completes syscall routine
CPU set to low level of privilege.
Application resumes.
Kernel mode
Kernel code
TIME

Using syscalls

- System calls are architecture-specific.

- The C library provides wrapper functions that establish an architecture-agnostic API:
  - Processes: `fork()`, `wait()`, `kill()`
  - Files: `open()`, `read()`, `write()`, `close()`
  - Etc.

Creating a new process: fork()

- The **`fork`** syscall requests the kernel to create a copy of the caller process.
- Both processes (father and child) will return from the fork syscall and continue afterwards. Fork returns:
  - 0 to the child, and
  - the process id (PID) of the child to the father.

Fork example

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
```

```
pid_t pid;

printf("I am the father.\n");
pid = fork();
if (pid == 0)
   printf("I am the child.\n");
else
   printf("I am the father... again.\n");
```

```
I am the father.
I am the father... again.
I am the child.
```

Waiting for a child process termination: wait()

- The **`wait`** syscall requests the kernel to suspend the execution of the calling process until a child terminates.
    - The kernel returns the PID and provides the exit status of the child.
- The calling process must have "forked" previously.

Wait example

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
```

```
pid_t pid;
int status;

printf("I am the father.\n");
pid = fork();
if (pid == 0) {
   printf("I am the child. Goodbye!\n");
   exit(22);      /* The child never escapes this block! */
}

pid = wait(&status)
printf("My child is gone. (status %d)\n", WEXITSTATUS(status));
```

Fork example

```
pid_t pid;
int status;

printf("I am the father.\n");
pid = fork();
if (pid == 0) {
   printf("I am the child. Goodbye!\n");
   exit(22);      /* The child never escapes this block! */
}

pid = wait(&status)
printf("My child is gone. (status %d)\n", WEXITSTATUS(status));
```

```
I am the father.
I am the child. Goodbye!
My child is gone. (status 22)
```

Loading and executing a new program: exec()

- The **`exec`** syscall requests the kernel to load a new program and run it in the caller process.

- The current program is replaced by the new program (and never returns to the first program).

- The C library provides a family of exec wrapper functions.

    - Check manual page: **`man 3 exec`**

Exec example

```
#include <stdio.h>
#include <unistd.h>

printf("I am the first program. Goodbye!\n");

execlp("ls", "ls", (char *)NULL);

printf("This line will not be printed...\n");
```

```
I am the first program. Goodbye!
APROG        Desktop     Documents   Downloads
PRCMP        WWW         PL11.pdf    TP11.pdf
```

A minimalistic shell (?)

```
pid_t pid;
int status;

printf("I am the father of 'ping'.\n\n");

pid = fork();
if (pid == 0) {
   execlp("ping", "ping", "-c3", "www.google.com", (char *)NULL);
}

pid = wait(&status)
printf("\nping is done: status %d\n", WEXITSTATUS(status));
```

A minimalistic shell (?)

```
I am the father of 'ping'.

PING www.google.com (142.250.184.4): 56 data bytes
64 bytes from 142.250.184.4: icmp_seq=0 ttl=60 time=191.597 ms
64 bytes from 142.250.184.4: icmp_seq=1 ttl=60 time=17.412 ms
64 bytes from 142.250.184.4: icmp_seq=2 ttl=60 time=18.758 ms

--- www.google.com ping statistics ---
3 packets transmitted, 3 packets received, 0.0% packet loss
round-trip min/avg/max/stddev = 17.412/75.922/191.597/81.796 ms

ping is done: status 0
```