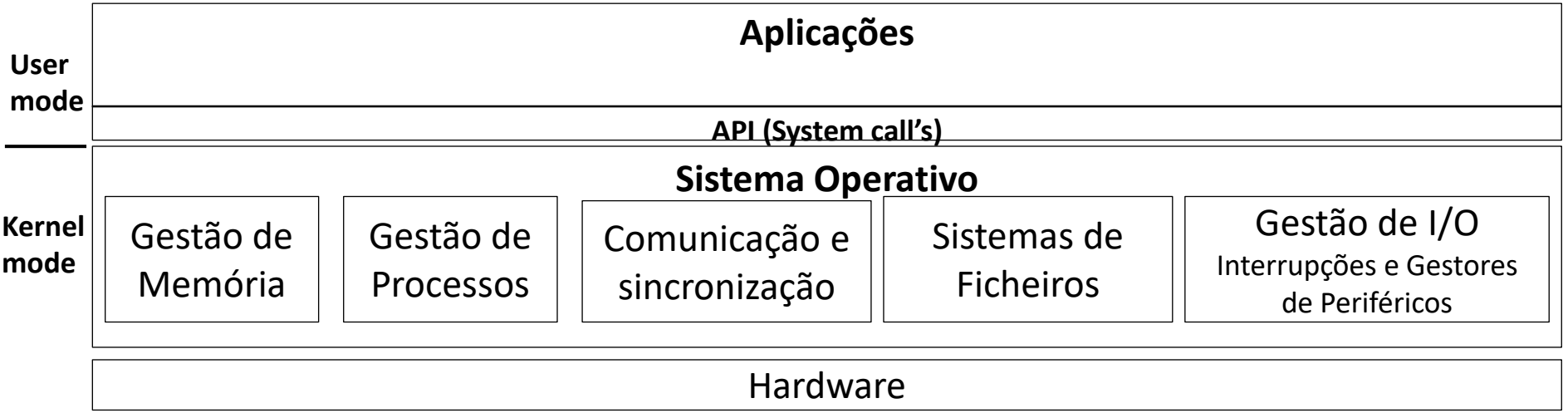
Sumário

- Sequência de chamada de serviços sistema
- Mecanismos hardware de suporte nas arquiteturas x86-32 e x86-64
- Objetivos de aprendizagem

Núcleo do sistema operativo

- Camada de software que abstrai o *hardware* das aplicações com o objectivo de:
  - Simplificar e uniformizar o acesso à máquina física, mapeando os recursos físicos em recursos lógicos (ou dito de outro modo, criando abstrações), acessíveis através de uma API (Aplication Program Interface).
  - Gerir de forma eficiente e segura a utilização dos recursos oferecidos pela máquina física (CPU, memória, dispositivos externos) entre todos os programas em execução - processos.

Arquitectura do Sistema Operativo (monolítica)



- Arquitectura monolítica
  - Organização em módulos
  - Estruturas de dados globais
  - Serviços do SO executam no contexto do processo chamador

Kernel mode vs user mode

- Em cada momento, o CPU encontra-se num determinado nível de privilégio.
- Os CPUs da família x86-32 (IA-32) e x86-64 (AMD64) suportam 4 níveis de privilégio, designados rings (0 .. 3).
- Os sistemas operativos Windows e Linux usam essencialmente os dois níveis extremos:
  - Ring 0 - **Kernel Mode** (*aka* **Supervisor Mode**) : nível de privilégio máximo, em que é possível executar qualquer instrução, aceder a todos os registos, a toda a memória e espaço de I/O

    Qualquer erro na execução neste modo pode comprometer todo o sistema

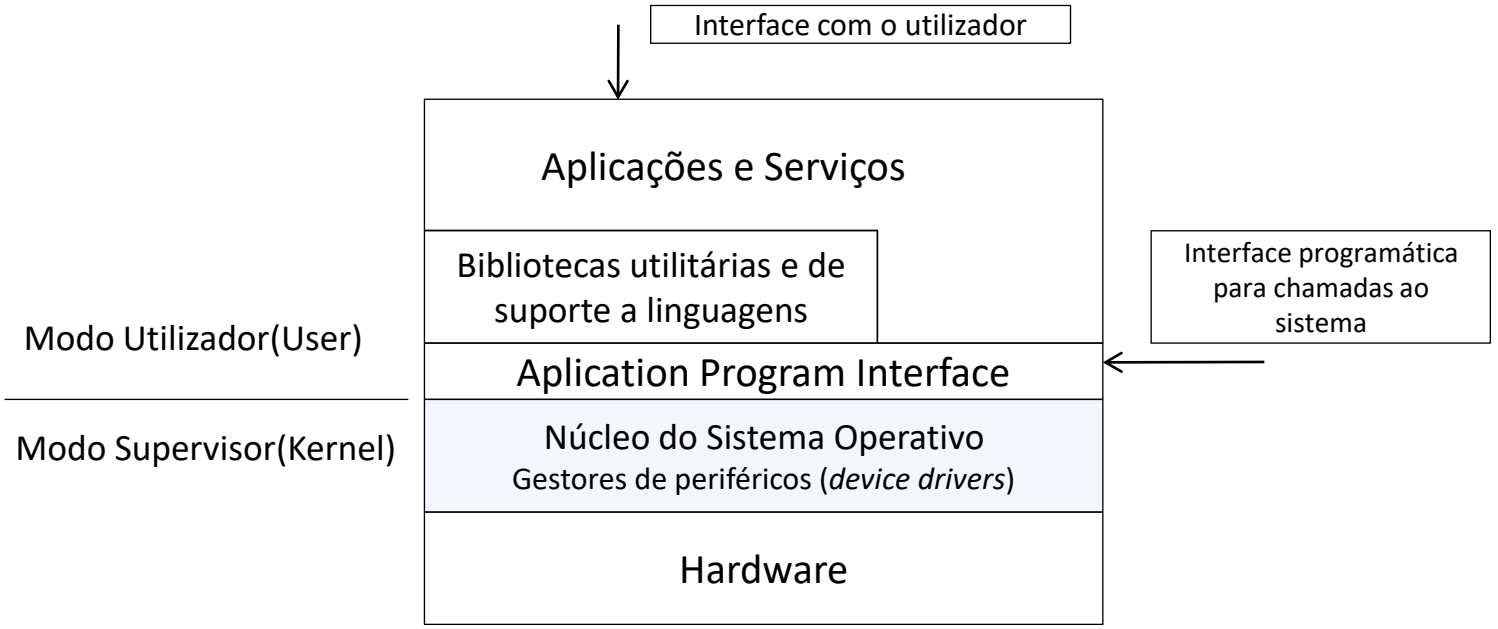
  - Ring 3 - **User Mode** : nível de privilégio mínimo, com algumas instruções, alguns registos e partes do espaço de endereçamento inacessíveis.

    Qualquer erro na execução neste modo apenas compromete a aplicação

Transições entre user mode e kernel mode

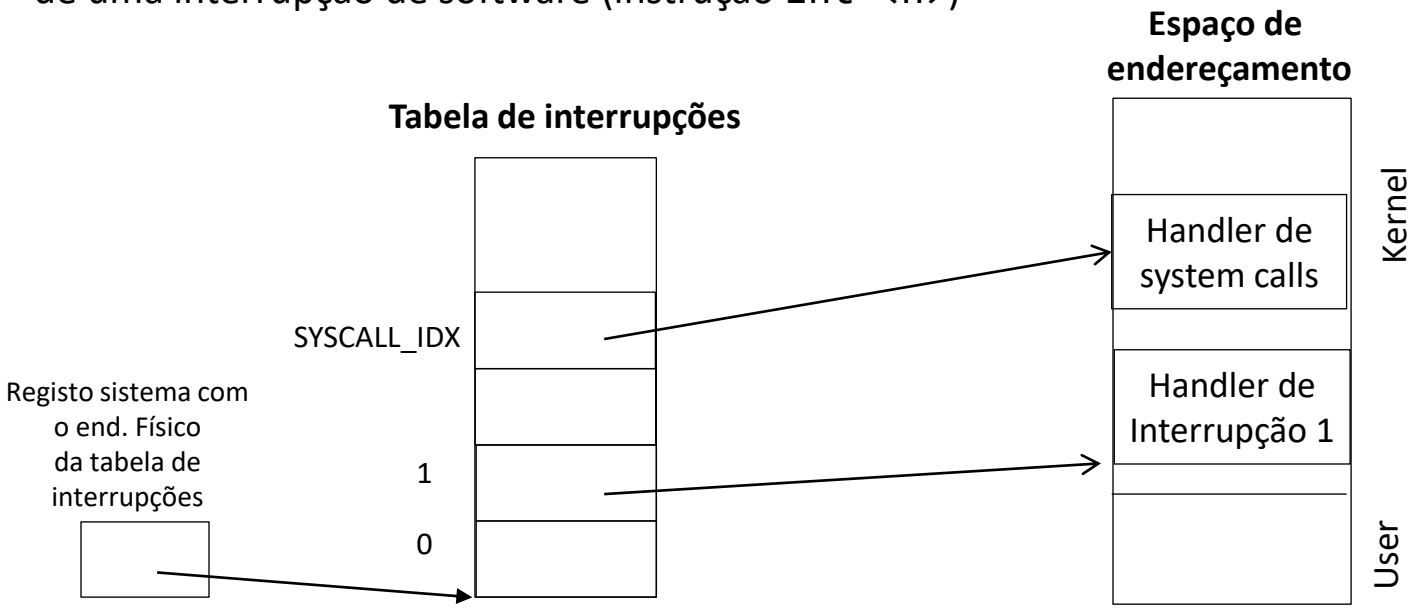
- O CPU arranca sempre em *kernel mode* (ring 0), executando-se o código de arranque do sistema.
- Quando as inicializações de sistema estão concluídas, é lançado o primeiro processo para execução em *user mode*. A partir desse momento a execução ocorre livremente em *user mode*, havendo três (únicos) motivos para regressar a *kernel mode*:
  - Atendimento de pedidos de interrupção em linhas de *hardware*
  - Tratamento de excepções do CPU (instruções ilegais, endereços de memória inacessíveis, divisão por 0, etc.)
  - Chamadas de sistema (*system calls*): invocação de operações do *kernel*

Camadas de Software



Visão geral (simplificada) do mecanismo de suporte a *system calls via tabela de interrupções*

- Nas arquitecturas que suportam interrupções vetorizadas existe uma tabela, construída pelo Sistema Operativo e consultada pelo *hardware,* onde se especifica o endereço da função *handler* para cada interrupção. Os *handlers* de interrupção correm em modo *kernel*. Na ocorrência de uma interrupção em **user mode** é provocada pelo *hardware* a comutação do privilégio de execução.
- Também os handlers de excepções (interrupções síncronas) são programados nesta tabela
- Nas versões iniciais da arqruitetura x86-32 uma entrada desta tabela era reservada para a chamada de serviços de sistema (***system calls***). A chamada era realizada através da execução de uma interrupção de software (instrução `int <n>`)



Excerto da listagem de id's de system calls em /usr/include/x86_64-linux-gnu/asm/unistd_64.h

```
#define __NR_read 0
#define __NR_write 1
#define __NR_open 2
#define __NR_close 3
#define __NR_stat 4
#define __NR_fstat 5
#define __NR_lstat 6
#define __NR_poll 7
#define __NR_lseek 8
#define __NR_mmap 9
#define __NR_mprotect 10
#define __NR_munmap 11
#define __NR_brk 12
#define __NR_rt_sigaction 13
#define __NR_rt_sigprocmask 14
#define __NR_rt_sigreturn 15
#define __NR_ioctl 16
```

```
#define __NR_pread64 17
#define __NR_pwrite64 18
#define __NR_readv 19
#define __NR_writev 20
#define __NR_access 21
#define __NR_pipe 22
#define __NR_select 23
#define __NR_sched_yield 24
#define __NR_mremap 25
#define __NR_msync 26
#define __NR_mincore 27
#define __NR_madvise 28
#define __NR_shmget 29
#define __NR_shmat 30
#define __NR_shmctl 31
#define __NR_dup 32
#define __NR_dup2 33
// …….
```

Chamada ao sistema em x86-64 (`syscall`/`sysret`)

- Registos usados no Linux gcc na passagem de argumentos a funções na convenção de chamada para a arquitetura x86-64:

  `rdi, rsi, rdx, rcx, r8, r9`

- Chamada de sistema na arquitetura x86-64 no Linux:

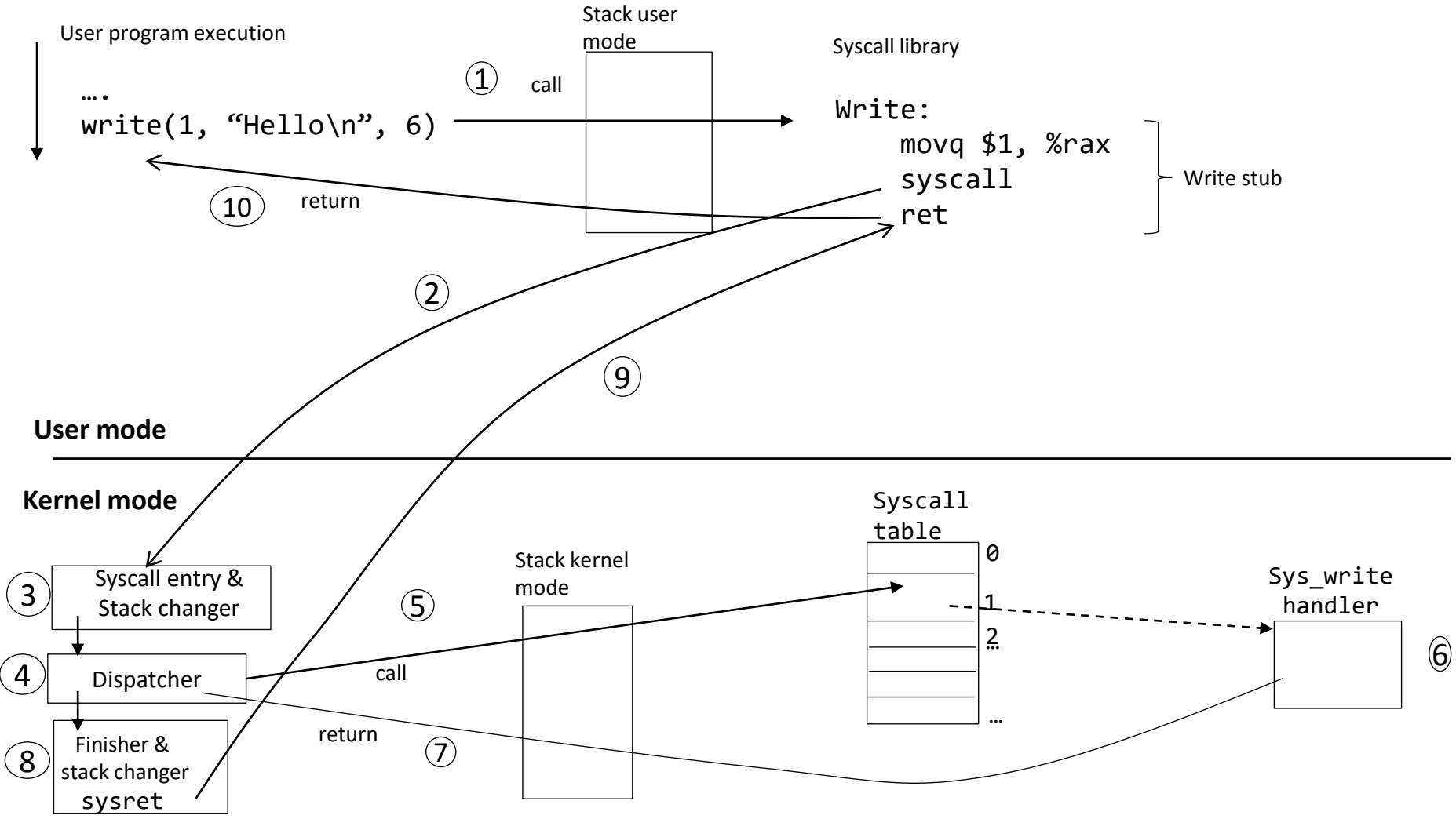
É usada uma forma mais eficiente de chamada (**syscall**/**sysret**) comparativamente com a solução de utilização do mecanismo de interrupções (via a instrução de interrupção por software - **int <n>**)

rax: identificador da operação de sistema

rdi, rsi, rdx, r10, r8, r9: argumentos da operação

1. Instrução **syscall** provoca entrada em *kernel mode*, para a rotina configurada previamente pelo sistema operativo em registos especiais (MSR – Model Specific Registers) do CPU.
2. Na execução em *kernel mode o stack de* **user mode** do processo é trocado para o *stack* específico de kernel. No SO há uma tabela de ponteiros para funções, indexada pelo valor recebido em rax. O respectivo serviço é invocado.
3. A saída do *kernel*, o *stack* de user mode é reposto e a comutação para modo user é realizada executando a instrução **sysret**, com o valor de retorno do serviço deixado em **rax**.

Passos na execução de system call



Exemplo de *chamada de sistema* (write)

As chamadas de sistema estão presentes na biblioteca de runtime do C
(na imagem usada em SO, na *shared library* `/lib/x86_64-linux-gnu/libc-2.31.so`)

A seguir apresenta-se o código associado ao *system call write:*

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

```
.text
.global mywrite

mywrite:

   movq $1, %rax
   syscall
   ret

.ends
```

Na versão (já antiga) 4.7 do *kernel* do Linux existem 337 serviços de sistema. Poderá ver a tabela em: https://blog.rchapman.org/posts/Linux_System_Call_Table_for_x86_64/

Detalhe do par SYSCALL/SYSRET - syscall (Opcional)

(retirado do doc. 64-ia-32-architectures-software-developer-manual da Intel)

No texto seguinte é feita referência a registos MSR (Model Specific Registers), que são registos de sistema utilizados pelo SO para armazenar informação necessária à gestão do Sistema Operativo (No caso, o endereço do *handler* do *system call* e dos enedereços de código e *stack* do kernel

SYSCALL

Invokes an OS system-call handler at privilege level 0. It does so by loading RIP from the (MSR register) IA32_LSTARMSR (after saving the address of the instruction following SYSCALL into RCX).
SYSCALL also saves RFLAGS into R11 and then masks RFLAGS using the IA32_FMASK MSR specifically, the processor clears in RFLAGS every bit corresponding to a bit that is set in the IA32_FMASK MSR.

SYSCALL loads the CS and SS selectors with values derived from bits 47:32 of the IA32_STAR MSR. The SYSCALL instruction does not save the stack pointer (RSP). If the OS system-call handler will change the stack pointer, it is the responsibility of software to save the previous value of the stack pointer. This might be done prior to executing SYSCALL, with software restoring the stack pointer with the instruction following SYSCALL (which will be executed after SYSRET). Alternatively, the OS system-call handler may save the stack pointer and restore it before executing SYSRET.

Detalhe do par SYSCALL/SYSRET - syscall (Opcional)

No texto seguinte é feita referência a registos MSR (Model Specific Registers), que são registos de sistema utilizados pelo SO para armazenar informação necessária à gestão do Sistema Operativo (No caso, o endereço do *handler* do *system call* e dos enedereços de código e *stack* do kernel

SYSRET

Is a companion instruction to the SYSCALL instruction. It returns from an OS system-call handler to user code at privilege level 3. It does so by loading RIP from RCX and loading RFLAGS from R11. SYSRET loads the CS and SS selectors with values derived from bits 63:48 of the IA32_STAR MSR.

The SYSRET instruction does not modify the stack pointer (ESP or RSP). For that reason, it is necessary for software to switch to the user stack. The OS may load the user stack pointer (if it was saved after SYSCALL) before executing SYSRET; alternatively, user code may load the stack pointer (if it was saved before SYSCALL) after receiving control from SYSRET.
If the OS loads the stack pointer before executing SYSRET, it must ensure that the handler of any interrupt or exception delivered between restoring the stack pointer and successful execution of SYSRET is not invoked with the user stack. It can do so using approaches such as the following: External interrupts. The OS can prevent an external interrupt from being delivered by clearing EFLAGS.IF before loading the user stack pointer.

Objectivos de aprendizagem

- Compreensão da necessidade de disponilização pelo hardware de pelo menos dois níveis de privilégio de execução de código
- Compreensão do mecanismo de despacho de interrupções/excepções e sua utilização na chamada de serviços.
- Compreensão da necessidade de comutação de stacks na transição para modo kernel
- Sensibilidade para o custo de uma chamada ao SO