

Architettura dei Sistemi Operativi



Corso di Laurea in Ingegneria Informatica
Università degli Studi di Napoli Federico II
Anno Accademico 2024/2025, Canale San Giovanni

Sommario



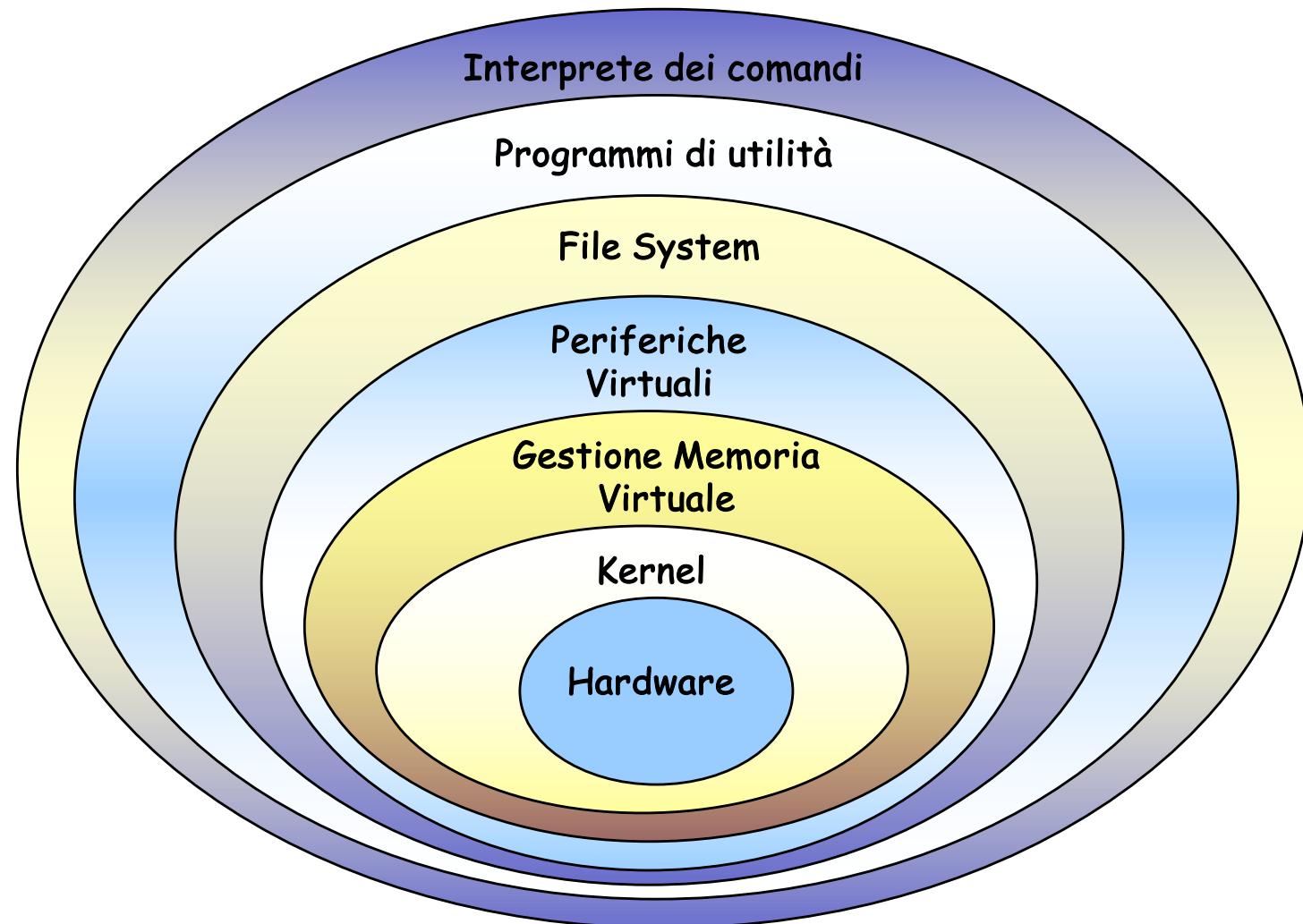
- Sommario:
 - Funzionalità di un SO
 - Architetture dei SO
- Riferimenti:
 - P. Ancilotti, M. Boari, A. Ciampolini, G. Lipari, "Sistemi Operativi", Mc-Graw-Hill (Cap.1, Par. 1.4)
 - www.ostep.org, Cap. 2 (sez. 2.1-2.5), Cap. 6 (sez. 6.1-6.2)

Funzionalità di un SO



- Virtualizzazione risorse hardware (fornitore di servizi)
 - File system
 - Processi
 - Periferiche "astratte"
 - etc...
- Gestione e Coordinamento
 - Meccanismi di protezione
 - Meccanismi per la comunicazione
 - Gestore Risorse

SO come macchina virtuale



Kernel



- Il **kernel** ("nucleo") è quella parte del SO che risiede in memoria principale
- Contiene le funzioni fondamentali del SO

Kernel



Al livello **kernel** (nucleo) la macchina virtuale realizzata dal SO

- possiede tante **CPU** quanti sono i processi (processori virtuali)
- non possiede meccanismi di **interruzione**
- possiede istruzioni di **sincronizzazione e scambio di messaggi** tra processi che operano sui processori virtuali



Esempio: virtualizzazione CPU

```
int main(int argc, char *argv[]) {  
  
    char *str = argv[1];          // 1° parametro a linea di comando  
  
    while(1) {  
  
        printf("%s\n", str);    // stampa il parametro  
        Spin(1);               // ciclo "a vuoto"  
  
    }  
}
```

```
$ make  
  
$ ./cpu A & ./cpu B & ./cpu C & ./cpu D &  
A  
B  
D  
C  
A  
B  
D  
C  
A  
...  
...
```



Livello gestione della memoria

Al livello **gestione della memoria**, la macchina virtuale realizzata dal SO

- consente di far riferimento a spazi di **indirizzi virtuali**
- garantisce la **protezione**
- consente in taluni casi di ignorare se il programma e/o i dati siano fisicamente residenti **in memoria centrale** o su **memoria di massa (disco)**



Esempio: virtualizzazione memoria

```
int main(int argc, char *argv[]) {  
  
    int *p = malloc(sizeof(int));  
    *p = atoi(argv[1]);           // converte 1° parametro in intero  
  
    printf("[PROCESSO %d] indirizzo di p=%d\n", getpid(), p);  
  
    while(1) { // ciclo "a vuoto"  
  
        Spin(1);  
        *p = *p + 1;  
        printf("[PROCESSO %d] valore di p=%d\n", getpid(), *p);  
    }  
}
```



Esempio: virtualizzazione memoria

```
int main(int argc, char *argv[]) {  
  
    int *p = malloc(sizeof(int));  
    *p = atoi(argv[1]);           // converte 1° parametro in intero  
  
    printf("[PROCESSO %d] indirizzo di p=%d\n", getpid(), p);  
  
while(1) { // ciclo "a vuoto"  
  
    Spin(1);  
    *p = *p + 1;  
    printf("[PROCESSO %d] valore  
}  
}
```

```
$ sudo bash -c "echo 0 > \  
/proc/sys/kernel/randomize_va_space"  
  
$ ./mem 0 & ./mem 0 &  
[1] 24113  
[2] 24114  
(24113) address pointed to by p: 0x200000  
(24114) address pointed to by p: 0x200000  
(24113) p: 1  
(24114) p: 1  
(24114) p: 2  
(24113) p: 2  
(24113) p: 3  
(24114) p: 3  
...  
...
```



Livello gestione periferiche

Al livello **gestione periferiche**, la macchina virtuale realizzata dal SO

- dispone di **periferiche dedicate** ai singoli processi
- maschera le caratteristiche **fisiche** delle periferiche
- gestisce parzialmente i **malfunzionamenti** delle periferiche



Livello file system

Al livello **file system**, la macchina virtuale realizzata dal S.O.

- offre **strutture logiche** (es. file e directory) per memorizzare blocchi di informazioni
- ne controlla gli **accessi**
- ne gestisce l'**organizzazione fisica** su memoria di massa

Esempio: virtualizzazione I/O



```
int main(int argc, char *argv[ ]) {  
  
    char buffer[20];  
    sprintf(buffer, "hello world\n"); // inizializza stringa  
  
    // fd "rappresenta" il file  
    int fd = open("/tmp/file", O_WRONLY | O_CREAT, 0644);  
  
    write(fd, buffer, strlen(buffer));  
  
    close(fd);  
}
```

```
$ ./io  
  
$ cat /tmp/file  
hello world
```

....due domande



-  Come è invocato un Sistema Operativo
(qual è il suo flusso di controllo)?
-  Come è fatto l'interno di un Sistema Operativo?



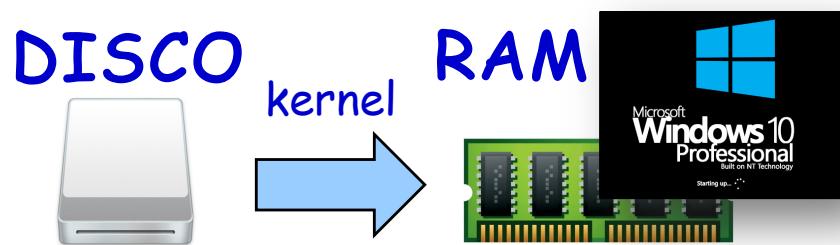
Sistemi Operativi Interrupt-Driven

- I sistemi operativi sono **guidati dalle interruzioni (sincrone ed asincrone)**
- Gran parte del nucleo viene eseguito come interrupt handler (**ISR**)
- Gli interrupt (sia hardware che software) guidano l'avvicendamento dei processi



Esecuzione di un sistema operativo

Avvio del computer (boot)



Applicazioni
(User Mode)



Eccezioni



System calls
(Interrupt sincroni)

Return-From-
Interrupt

**Kernel del
SISTEMA OPERATIVO
(Supervisor Mode)**

Idle loop



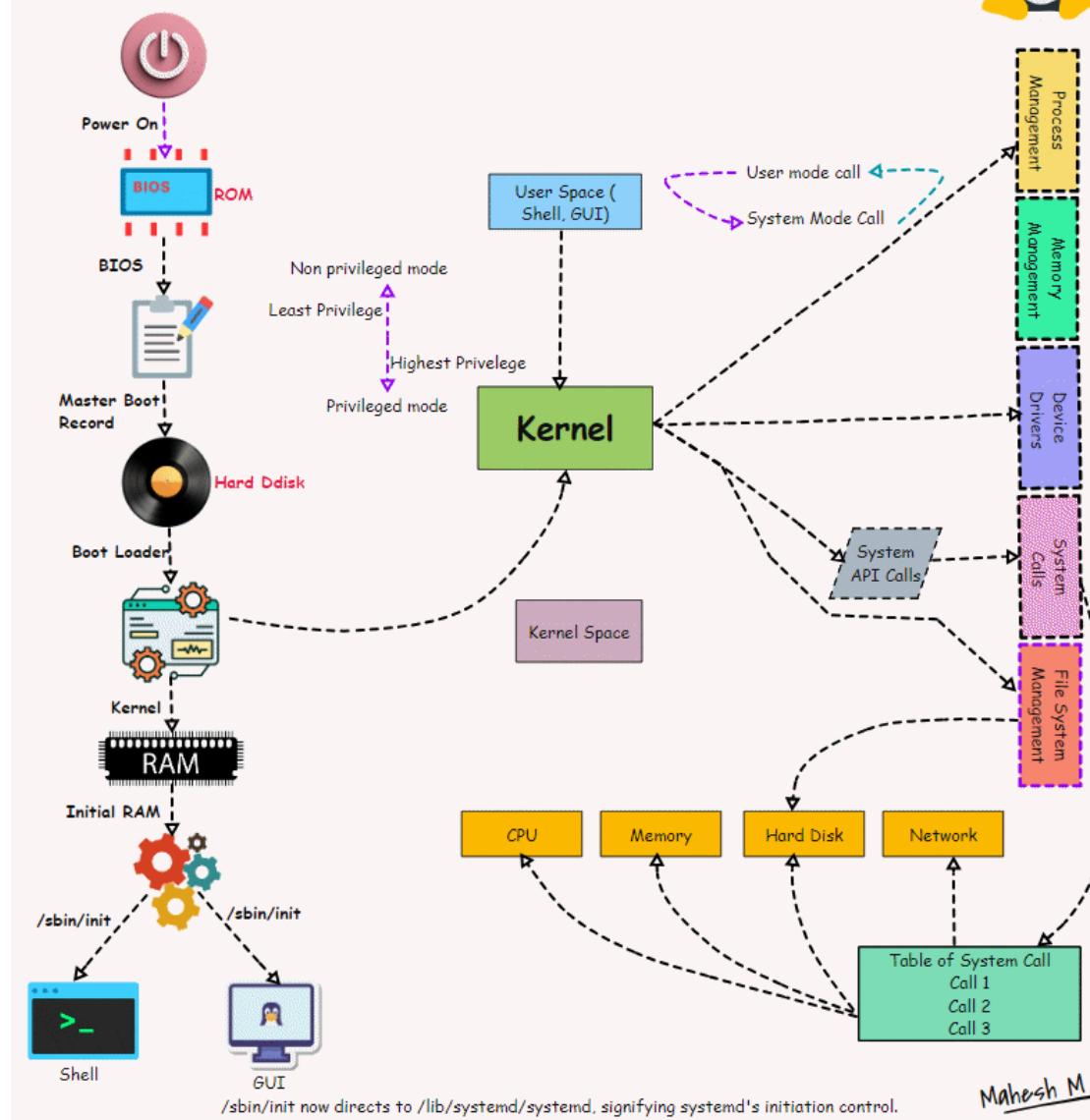
Interrupt
asincroni

Hardware



Esecuzione di un sistema operativo

HOW LINUX WORKS





System Call (richiesta di servizio)

- Per operare su una risorsa, i programmi devono fare una richiesta di un servizio (**system call**) al SO, es.
 - l'apertura di un file
 - la lettura dati da un file
 - la comunicazione con un altro processo
 - ...

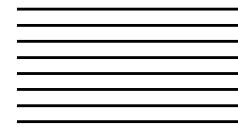
Una system call è **una richiesta al SO di eseguire privilegiate** per conto del programma chiamante

System Call (esecuzione)



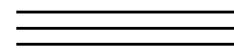
Programma
(user mode) SO
 (kernel mode)

main{



system
call

→ syscall () {



...
...
}



Una **system call** corrisponde alla attivazione di una parte del kernel a favore del **processo chiamante**, per espletare il servizio o per acquisire la risorsa richiesta

}



Categorie di System Call

- **Controllo dei processi.** Es. load, execute, allocate mem, free mem
- **Manipolazione dei file.** Es. create, delete, open, close, read
- **Gestione dei dispositivi.** Es. request device, read, write
- **Informazioni di sistema.** Es. get time, set time
- **Comunicazione.** Es. create connection, send, receive



Protezione delle risorse

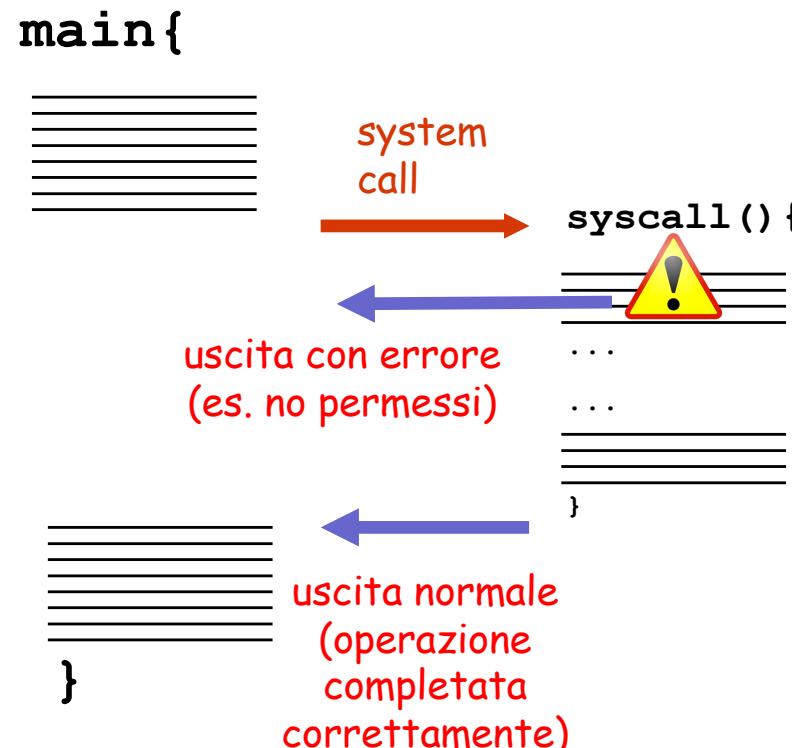
- Il codice del kernel è l'unico in esecuzione in **modalità supervisore**
- Le applicazioni (potenzialmente **buggate o malevole**) eseguono in **modalità utente**

Il kernel fa da intermediario nell'accesso alle risorse, effettuando **controlli di sicurezza**



Protezione delle risorse

Programma
(user mode) SO
 (kernel mode)

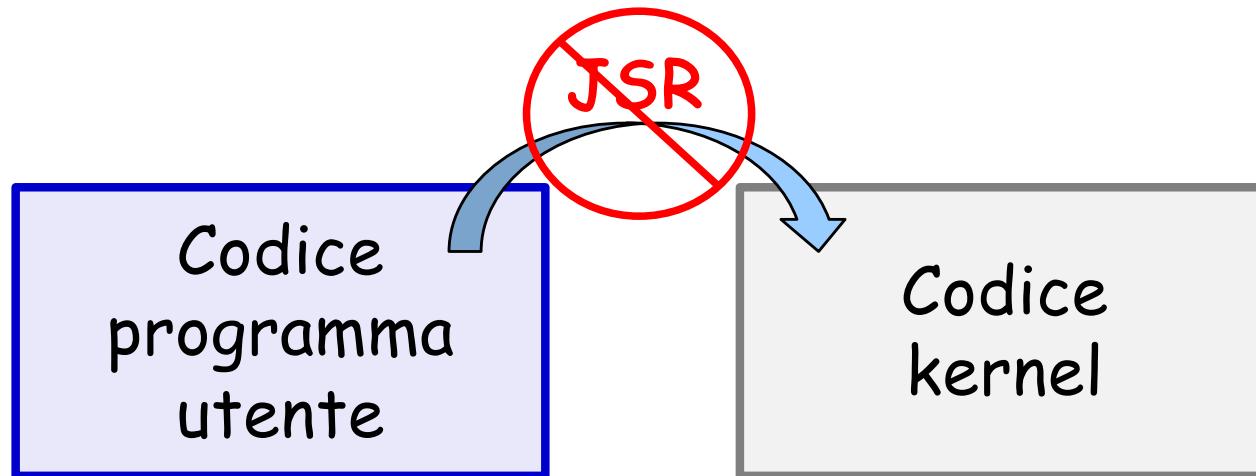


Il kernel può **terminare prematuramente** la system call se si accorge che il programma chiamante **non ha i permessi di accesso** alla risorsa richiesta



System Call (invocazione)

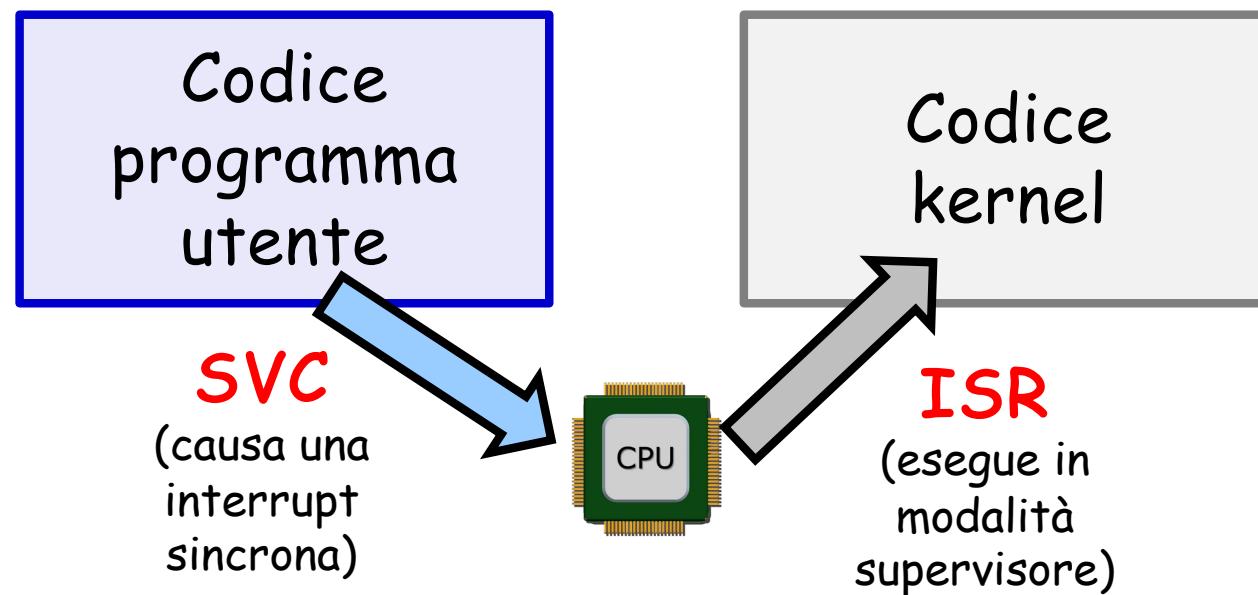
- Il programma utente **non può** invocare una system call mediante una **semplice istruzione di salto**
 - es. l'istruzione **JSR** per la chiamata di un sotto-programma
 - Il programma **utente non può, da solo, modificare il livello di privilegio** (sarebbe una contraddizione del principio della protezione!)





System Call (invocazione)

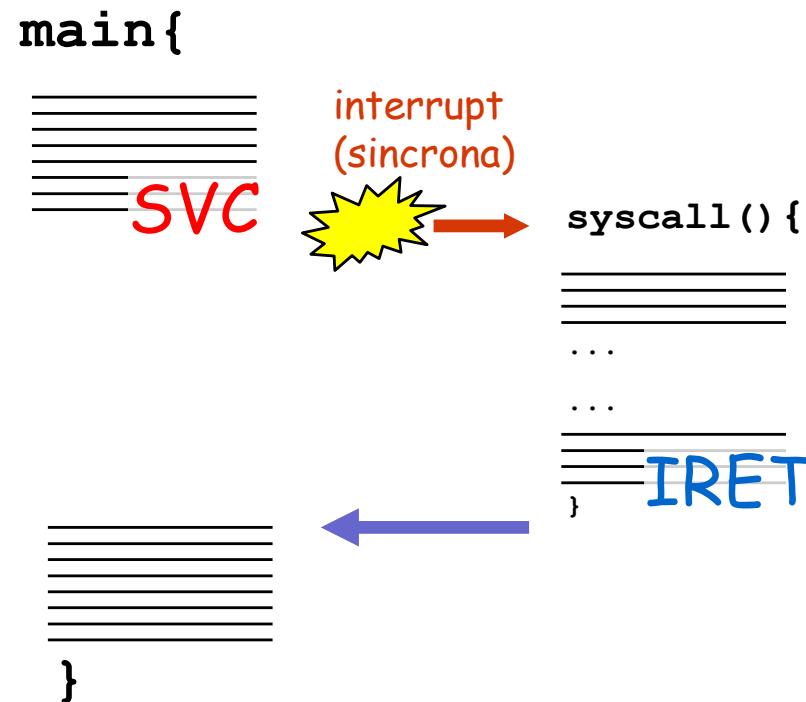
- Il programma è obbligato a utilizzare una istruzione speciale della CPU, spesso chiamata **SVC (supervisor call)**
- Innesca una **interrupt sincrona** e l'esecuzione di una ISR, che esegue le operazioni privilegiate



System Call (invocazione)



Programma
(user mode) SO
 (kernel mode)



L'istruzione macchina **SVC** innesca una **interrupt sincrona**.

In risposta alla interrupt, il SO esegue la **system call** e svolge una operazione per conto del programma utente (es. lettura dal disco).

Supervisor Call in Linux/x86



- **int 0x80** (vecchia istruzione, più semplice e lenta)
- **sysenter** (nuova istruzione, più veloce e complessa)



System Call in altre architetture

Arch/ABI	Instruction	System call #	Ret val	Ret val2	Error	Notes
alpha	callsys	v0	v0	a4	a3	1, 6
arc	trap0	r8	r0	–	–	
arm/OABI	swi NR	–	r0	–	–	2
arm/EABI	swi 0x0	r7	r0	r1	–	
arm64	svc #0	w8	x0	x1	–	
blackfin	excpt 0x0	P0	R0	–	–	
i386	int \$0x80	eax	eax	edx	–	
ia64	break 0x100000	r15	r8	r9	r10	1, 6
m68k	trap #0	d0	d0	–	–	
microblaze	brki r14,8	r12	r3	–	–	
mips	syscall	v0	v0	v1	a3	1, 6
nios2	trap	r2	r2	–	r7	
parisc	ble 0x100(%sr2, %r0)	r20	r28	–	–	
powerpc	sc	r0	r3	–	r0	1
powerpc64	sc	r0	r3	–	cr0.S0	1
riscv	ecall	a7	a0	a1	–	
s390	svc 0	r1	r2	r3	–	3
s390x	svc 0	r1	r2	r3	–	3
superh	trapa #31	r3	r0	r1	–	4, 6
sparc/32	t 0x10	g1	00	01	psr/csr	1, 6
sparc/64	t 0x6d	g1	00	01	psr/csr	1, 6
tile	swint1	R10	R00	–	R01	1
x86-64	syscall	rax	rax	rdx	–	5
x32	syscall	rax	rax	rdx	–	5
xtensa	syscall	a2	a2	–	–	

man syscall (<https://man7.org/linux/man-pages/man2/syscall.2.html>)



Application Binary Interface (ABI)

- Il **passaggio dei parametri di ingresso e uscita** alle system call deve seguire le regole imposte dal sistema operativo (**Application Binary Interface**)
- Linux su **x86 (32-bit)** adotta le seguenti convenzioni

Codice numerico della syscall: **EAX**

Parametri di ingresso:

- | | |
|--------|--------|
| 1. EAX | 4. ESI |
| 2. ECX | 5. EDI |
| 3. EDX | 6. EBP |

Valore di ritorno: **EAX**



Esempio di system call: write

```
$ man 2 write
```

WRITE(2)

Linux Programmer's Manual

WRITE(2)

NAME

write - write to a file descriptor

SYNOPSIS

```
#include <unistd.h>

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

DESCRIPTION

write() writes up to count bytes from the buffer starting at buf to the file referred to by the file descriptor fd.



Esempio di system call: write

```
1  #
2  # 32-bit system call numbers and entry vectors
3  #
4  # The format is:
5  # <number> <abi> <name> <entry point> <compat entry point>
6  #
7  # The __ia32_sys and __ia32_compat_sys stubs are created on-the-fly for
8  # sys_*() system calls and compat_sys_*() compat system calls if
9  # IA32_EMULATION is defined, and expect struct pt_regs *regs as their only
10 # parameter.
11 #
12 # The abi is always "i386" for this file.
13 #
14 0     i386    restart_syscall      sys_restart_syscall      __ia32_sys_restart_syscall
15 1     i386    exit                sys_exit              __ia32_sys_exit
16 2     i386    fork                sys_fork              __ia32_sys_fork
17 3     i386    read                sys_read              __ia32_sys_read
18 4     i386    write                sys_write             __ia32_sys_write
19 5     i386    open                sys_open              __ia32_compat_sys_open
20 6     i386    close                sys_close             __ia32_sys_close
```

La chiamata di sistema "write" corrisponde al **codice numerico 4**

I codici delle system call sono definiti nel file sorgente "**arch/x86/entry/syscalls/syscall_32.tbl**" del kernel Linux
(<https://git.io/JUPyB>)



Esempio di system call: write

```
mov $4,%eax          // EAX = codice della system call (4 = write)
mov $1,%ebx          // EBX = file destinazione (1 = il terminale)
mov $stringa,%ecx    // ECX = indirizzo della stringa ("hello")
mov $6,%edx          // EDX = lunghezza della stringa (6 char)
int $0x80
```

```
write(1, "hello", 6);
```



Esempio di system call: write

```
#include <unistd.h>
int main() {
    ssize_t result;
    char hello[] = "Hello world\n";
    size_t len = sizeof(hello);

    asm volatile (
        "mov $4,%eax;"           // EAX = 4 (NR_write)
        "mov $1,%ebx;"           // EBX = fd = 1
        "mov %1,%ecx;"           // ECX = %1 = hello
        "mov %2,%edx;"           // EDX = %2 = len
        "int $0x80;"             

        : "=g" (result)
        : "g" (hello), "g" (len)   // %1 = hello, %2 = len
        :"memory", "eax", "ebx", "ecx", "edx"
    );
    return 0;
}
```

syscall.c

Prerequisiti:

```
$ sudo apt-get -y install gcc-multilib g++-multilib
```

Compilare con:

```
$ gcc -m32 -O0 -Wall syscall.c -o syscall
```

https://github.com/rnatella/so_esempi



Esempio di system call: write

```
#include <unistd.h>
int main(){
    ssize_t result;
    char hello[] = "Hello world\n";
    size_t len = sizeof(hello);

    asm volatile (
        "mov $4,%eax;"           // EAX = 4 (NR_write)
        "mov $1,%ebx;"           // EBX = fd = 1
        "mov %1,%ecx;"           // ECX = %1 = hello
        "mov %2,%edx;"           // EDX = %2 = len
        "int$0x80"                // Call to kernel
        : "=g" (result)
        : "g" (hello), "g" (len)   // %1 = hello, %2 = len
        :"memory", "eax", "ebx", "ecx", "edx"
    );
    return 0;
}
```

syscall.c

Le funzioni di libreria standard del linguaggio C "nascondono" le chiamate di sistema

Prerequisiti:

```
$ sudo apt-get -y install gcc-multilib g++-multilib
```

Compilare con:

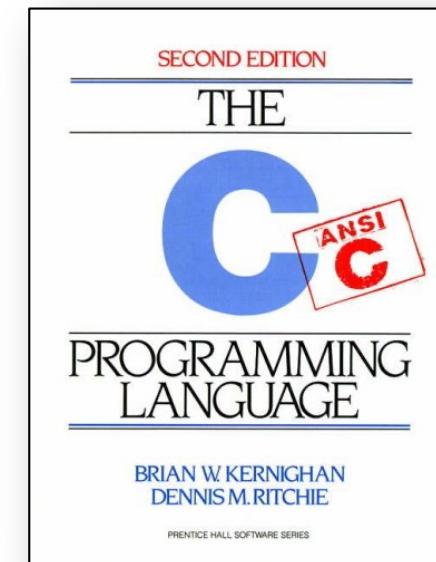
```
$ gcc -m32 -O0 -Wall syscall.c -o syscall
```

https://github.com/rnatella/so_esempi



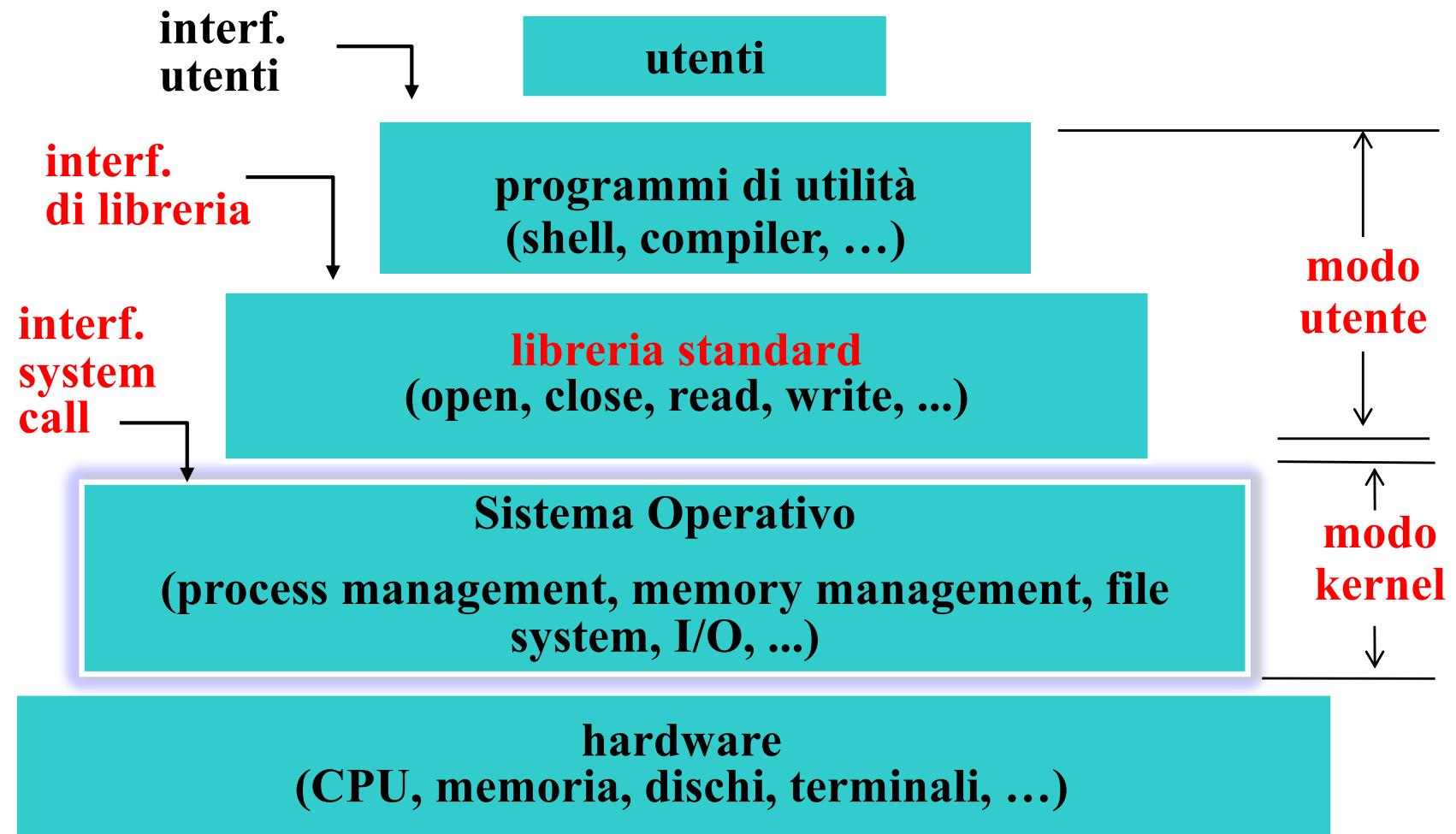
System Call tramite librerie

- Tipicamente, i linguaggi di programmazione forniscono **routine di interfaccia**, che trasformano **una tradizionale chiamata di procedura in una SVC**
- Facilitano la chiamata e lo scambio dei parametri
 - La **C Standard Library (libc)**, descritta nello standard ANSI C del linguaggio
 - In **C++**, il namespace **std** e la libreria **Boost**
 - Microsoft Windows fornisce la libreria e le API **Win32**

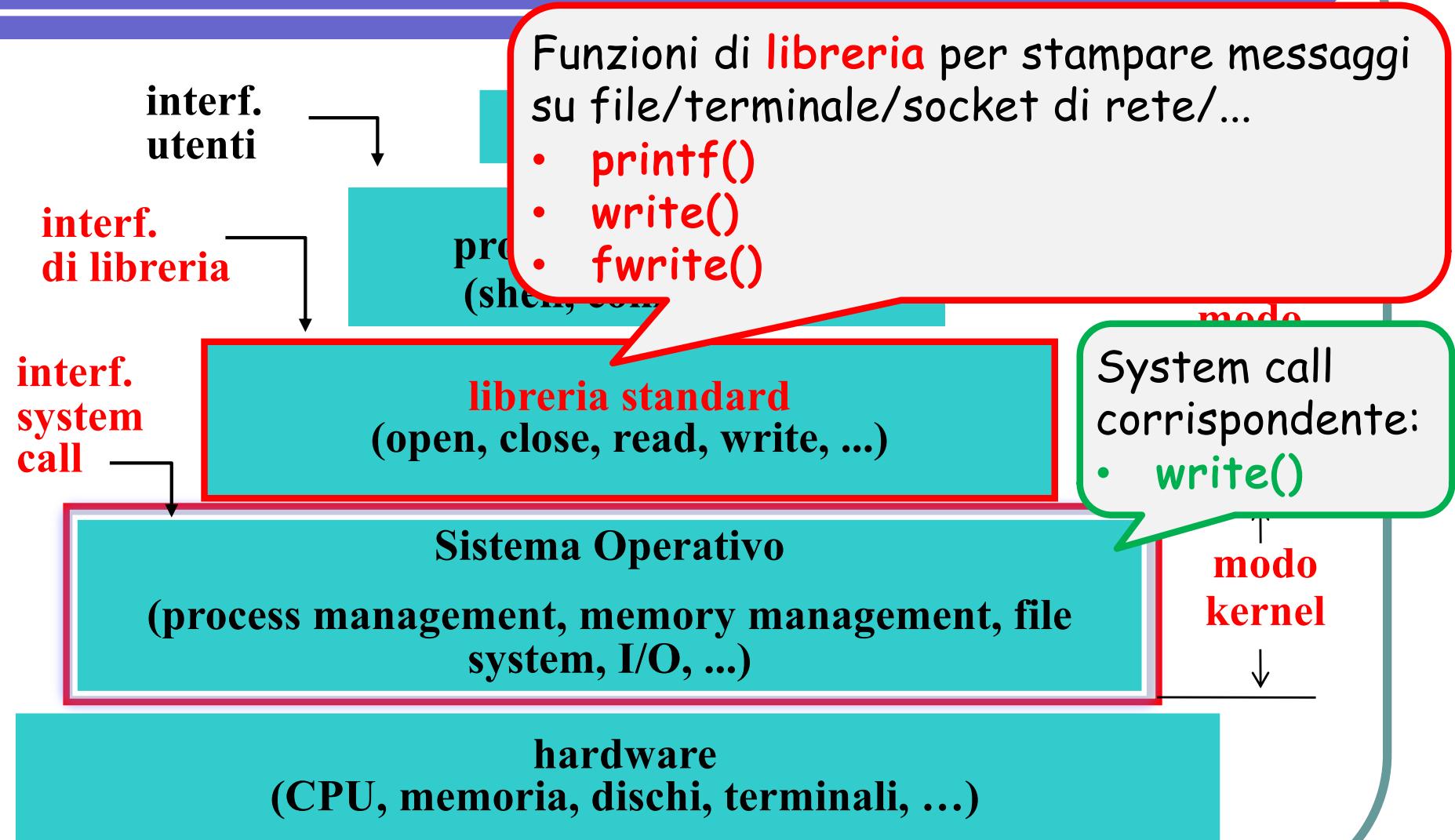




System Call tramite librerie



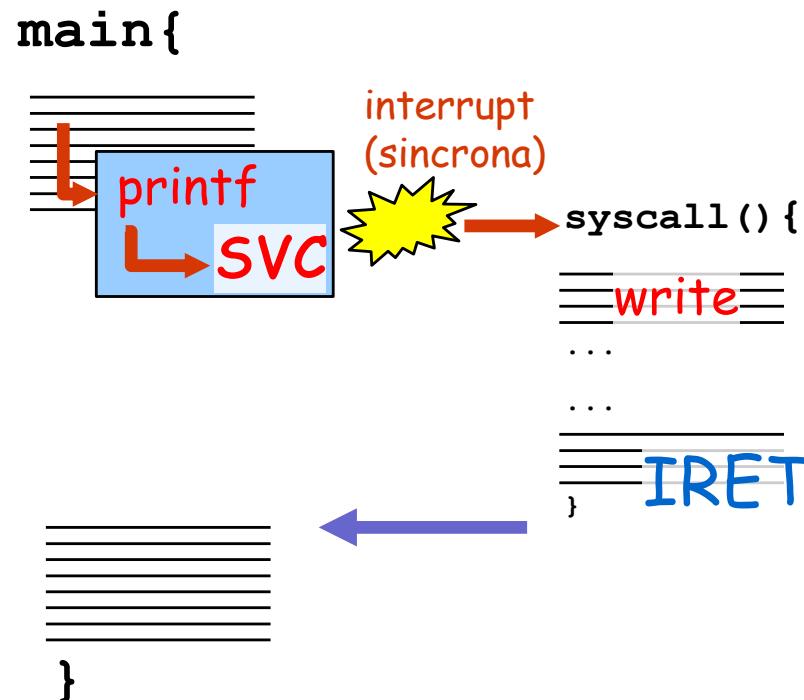
System Call tramite librerie





System Call tramite librerie

Programma
(user mode) SO
 (kernel mode)



Il programma "incorpora" al suo interno una **libreria** di funzioni (es. **printf**).

La funzione di libreria usa **SVC** per la chiamata di sistema (es. **write**).



System Call tramite librerie

- **write()** è sia il nome di una system call, sia di una funzione di libreria
- In UNIX, è utilizzata per scrivere dati su vari canali di I/O (file, rete, ...)
- "**Everything is a file**"

Architettura dei Sistemi Operativi



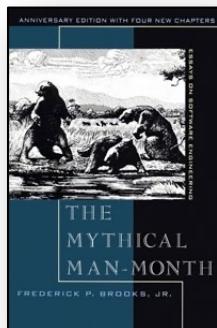
- Il SO è un software di **notevole complessità e dimensione**
- I primi sistemi operativi erano
 - costituiti da un **unico programma**
 - senza particolari suddivisioni
 - "raccolta" di funzioni
 - scritte in **linguaggio macchina**
 - a ognuna corrispondeva una system call





Architetture monolitiche

- Approccio **inadeguato per i complessi sistemi moderni**
- Lo **IBM OS-360**, SO batch multiprogrammato costituito da oltre un milione di istruzioni macchina, ha continuato a **produrre errori per molto tempo** durante tutto l'arco della sua vita



Brooks's law:
Adding manpower to a late software project makes it later
da: *The Mythical Man-Month*

Architettura dei Sistemi Operativi

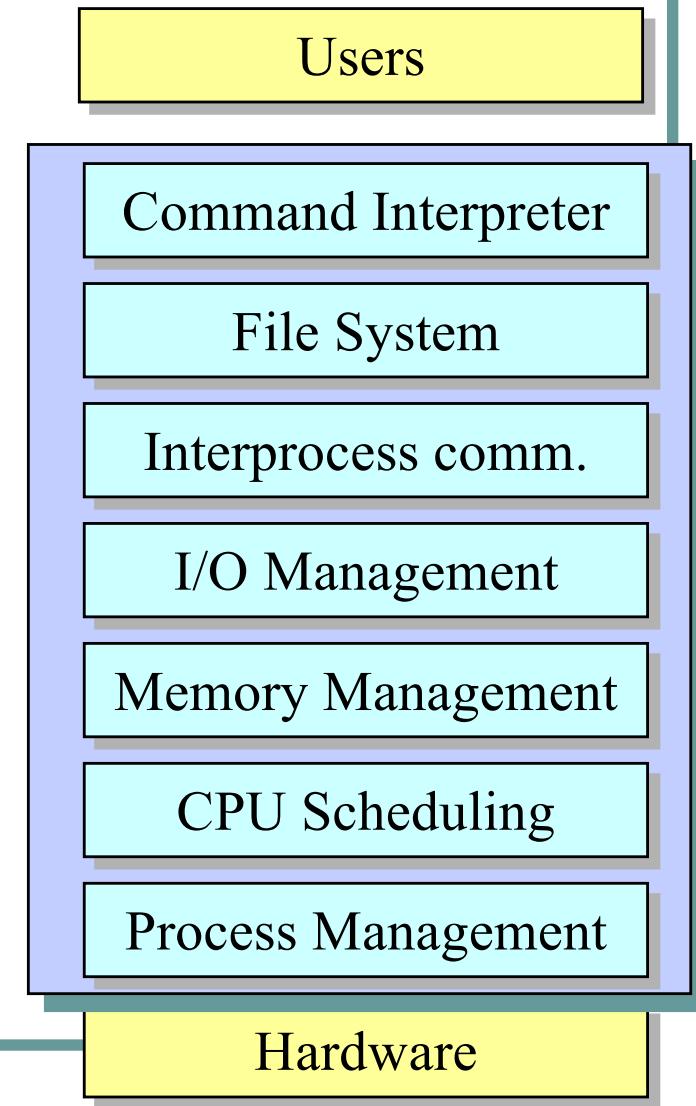


- La progettazione del SO richiede di applicare gli approcci propri della **ingegneria del software**, al fine di garantire:
 - Correttezza
 - Modularità
 - Facilità di manutenzione
 - Efficienza di esecuzione
 - Ecc.

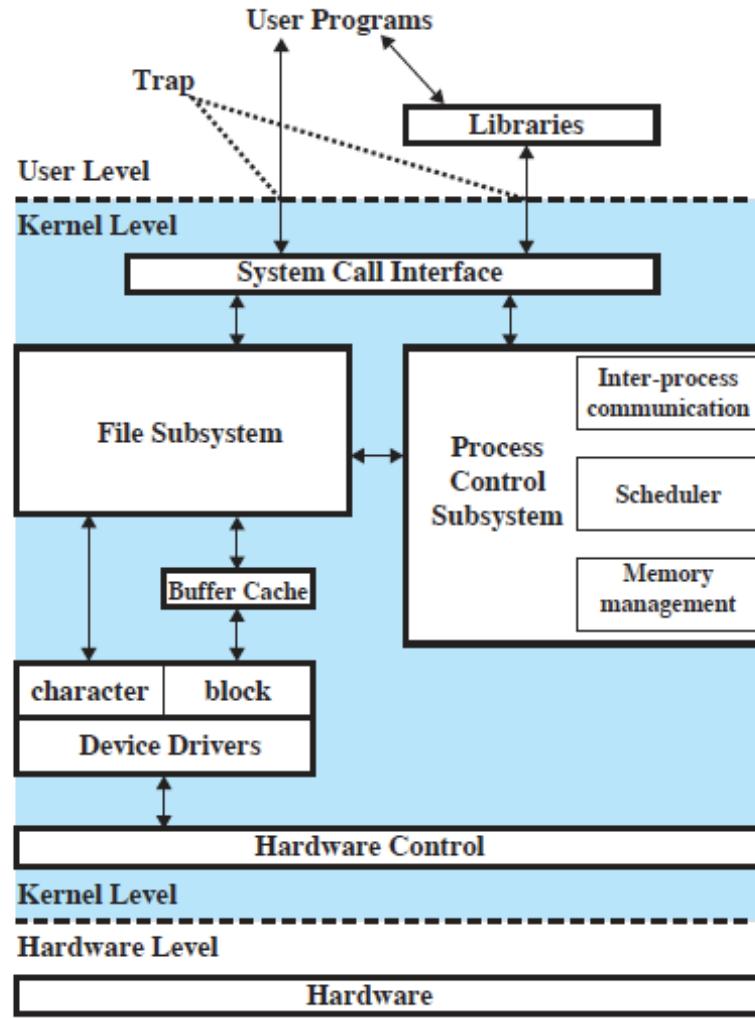
Architetture modulari



- I **SO modulari** suddividono il codice in più moduli:
 - **Interfaccia**: specifica le funzionalità offerte dal modulo
 - **Corpo**: l'implementazione del modulo
- Vantaggi
 - La modifica di un modulo ha **impatto ridotto** sul resto del SO
 - Si possono **caricare in memoria** i soli moduli necessari (es. device driver)

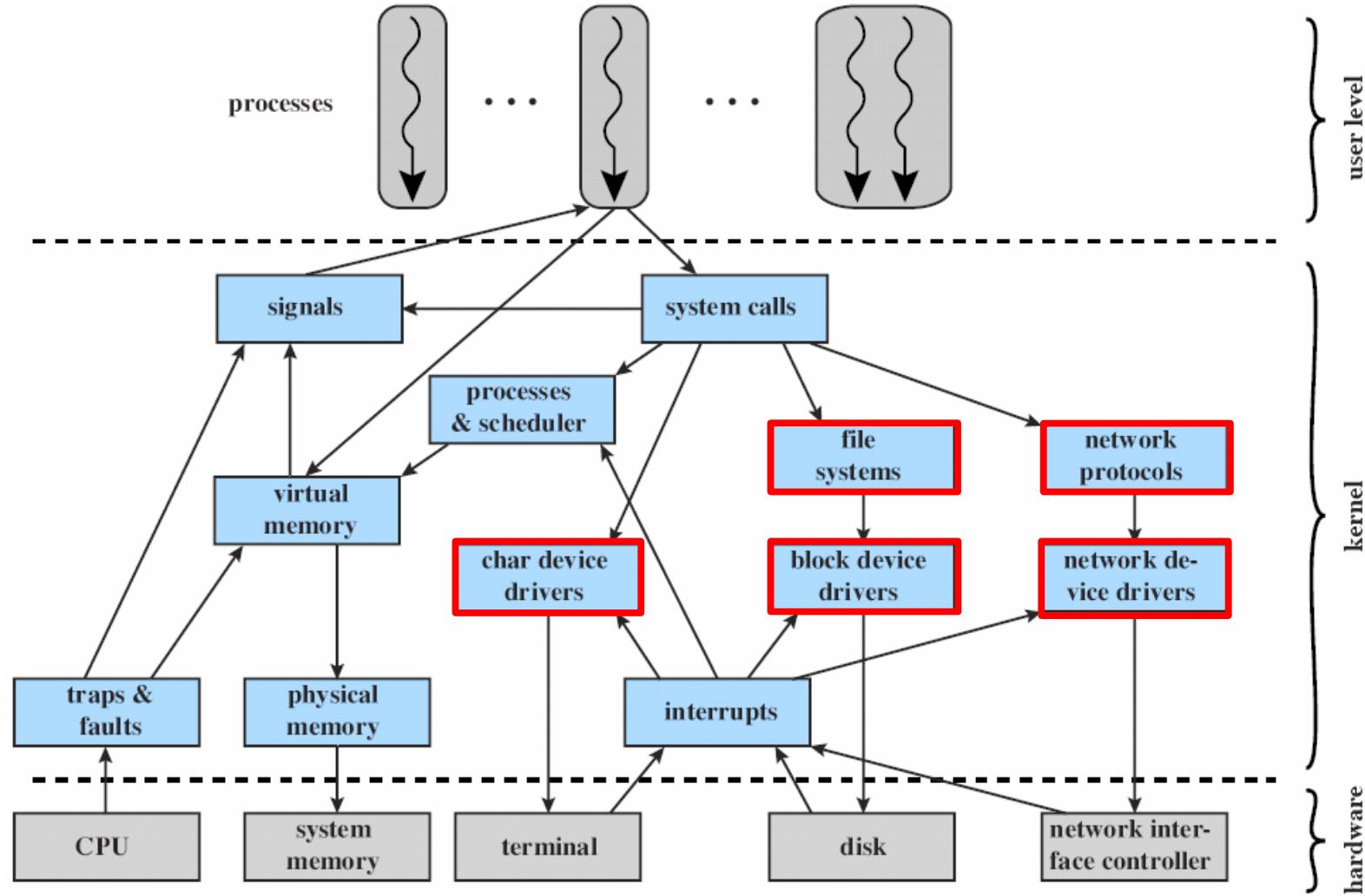


Esempi di architetture modulari: UNIX



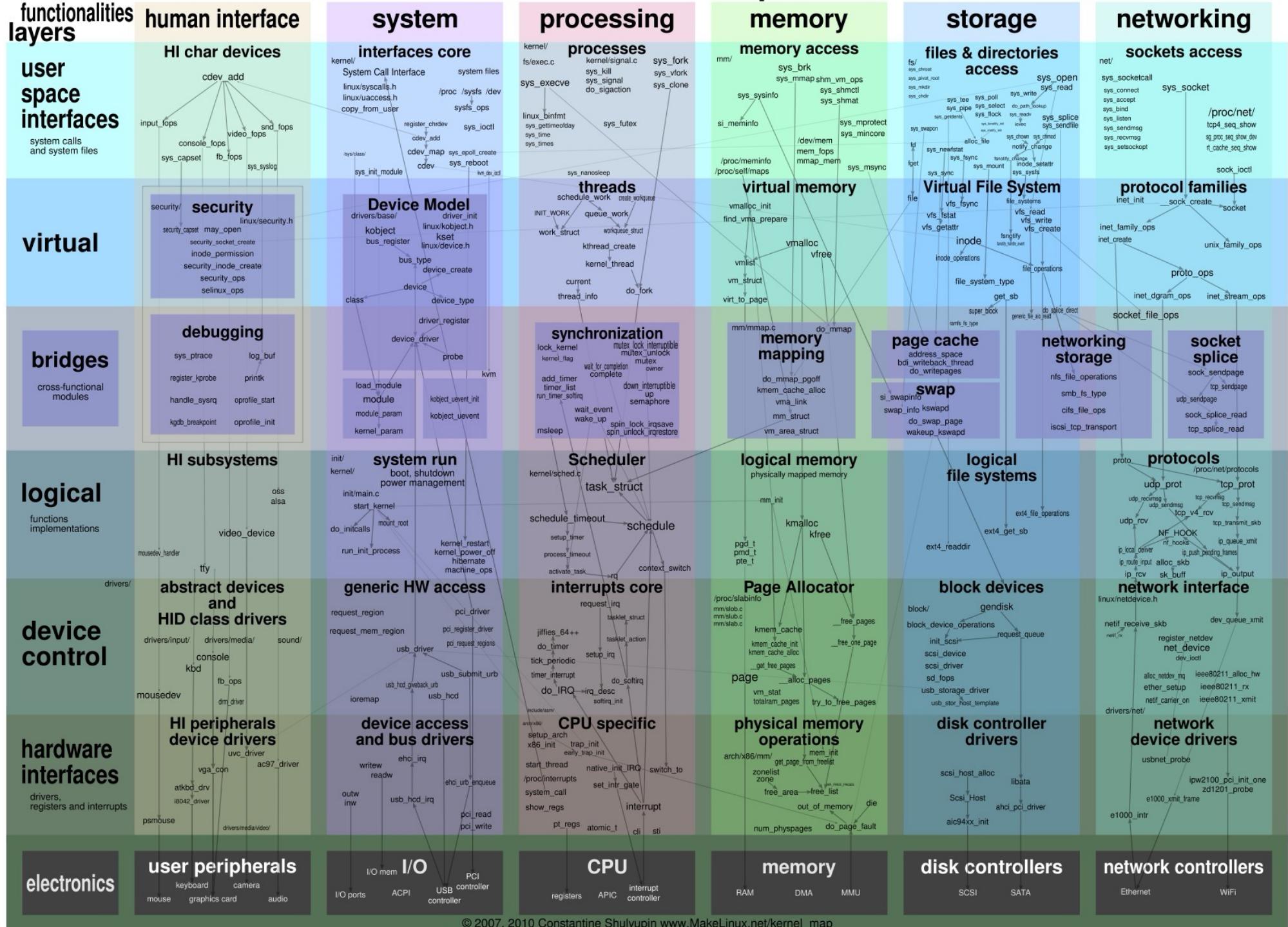


Esempi di architetture modulari: Linux



* **moduli caricabili** che possono essere collegati/scollegati dal kernel a tempo di esecuzione (runtime)

Linux kernel map





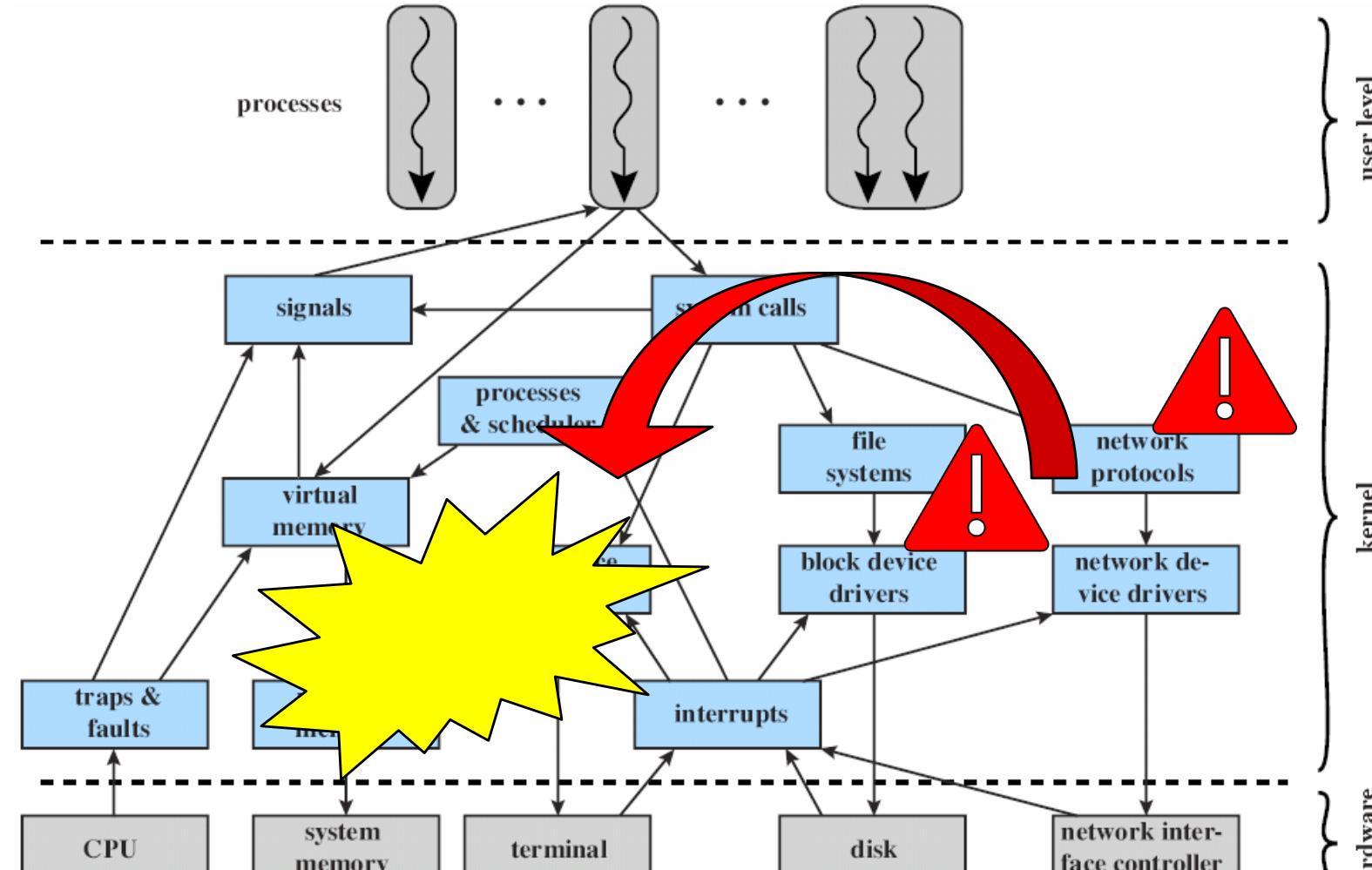
User-space vs. kernel-space

- Un aspetto rilevante è la suddivisione dei componenti del SO tra
 - **kernel-space**: eseguono in modo privilegiato
 - **user-space**: eseguono come fossero applicazioni (modo non-privilegiato)

Spostare i moduli **da kernel-space a user-space** favorisce **prestazioni, sicurezza, e modularità**



Moduli kernel-space: problematiche



Un bug o vulnerabilità in qualunque componente kernel può danneggiare l'intero sistema!

Blue Screen of Death (BSOD)



- Negli anni 90, il **SO Windows** era famigerato per la famosa **schermata blu** di errore
- Causato da errori in moduli in esecuzione in **kernel-space**



Blue Screen of Death (BSOD)

A defective CrowdStrike update sent computers around the globe into a reboot death spiral, taking down air travel, hospitals, banks, and more with it

All airport screens are the Windows
"Blue Screen of Death" (BSOD)



<https://www.wired.com/story/crowdstrike-outage-update-windows/>

Blue Screen of Death (BSOD)



...the root cause ... must be **related to a "kernel driver" update**

<https://www.wired.com/story/crowdstrike-outage-update-windows/>



Device drivers

- I **device drivers** si occupano della gestione dei singoli dispositivi (es. ISR)
- Sono tipicamente **moduli kernel** caricati in memoria su domanda
- Rappresentano la maggior parte del SO
 - Circa il **70%** delle linee di codice di Linux!
 - ... e anche una delle maggiori cause di **malfunzionamenti!**

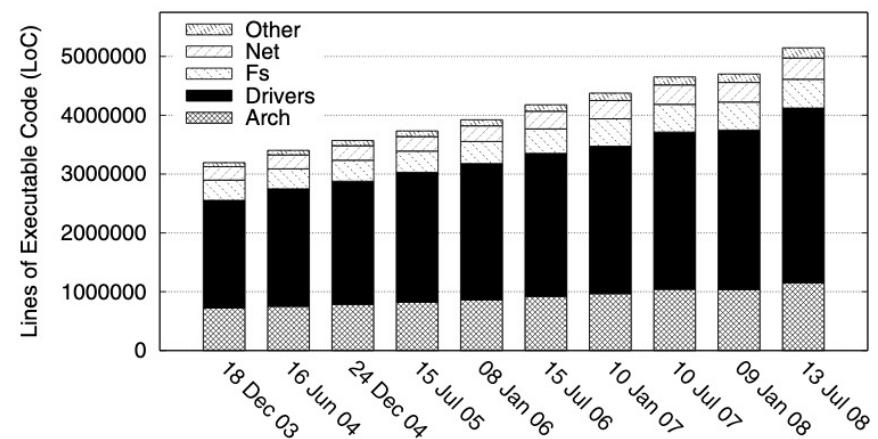
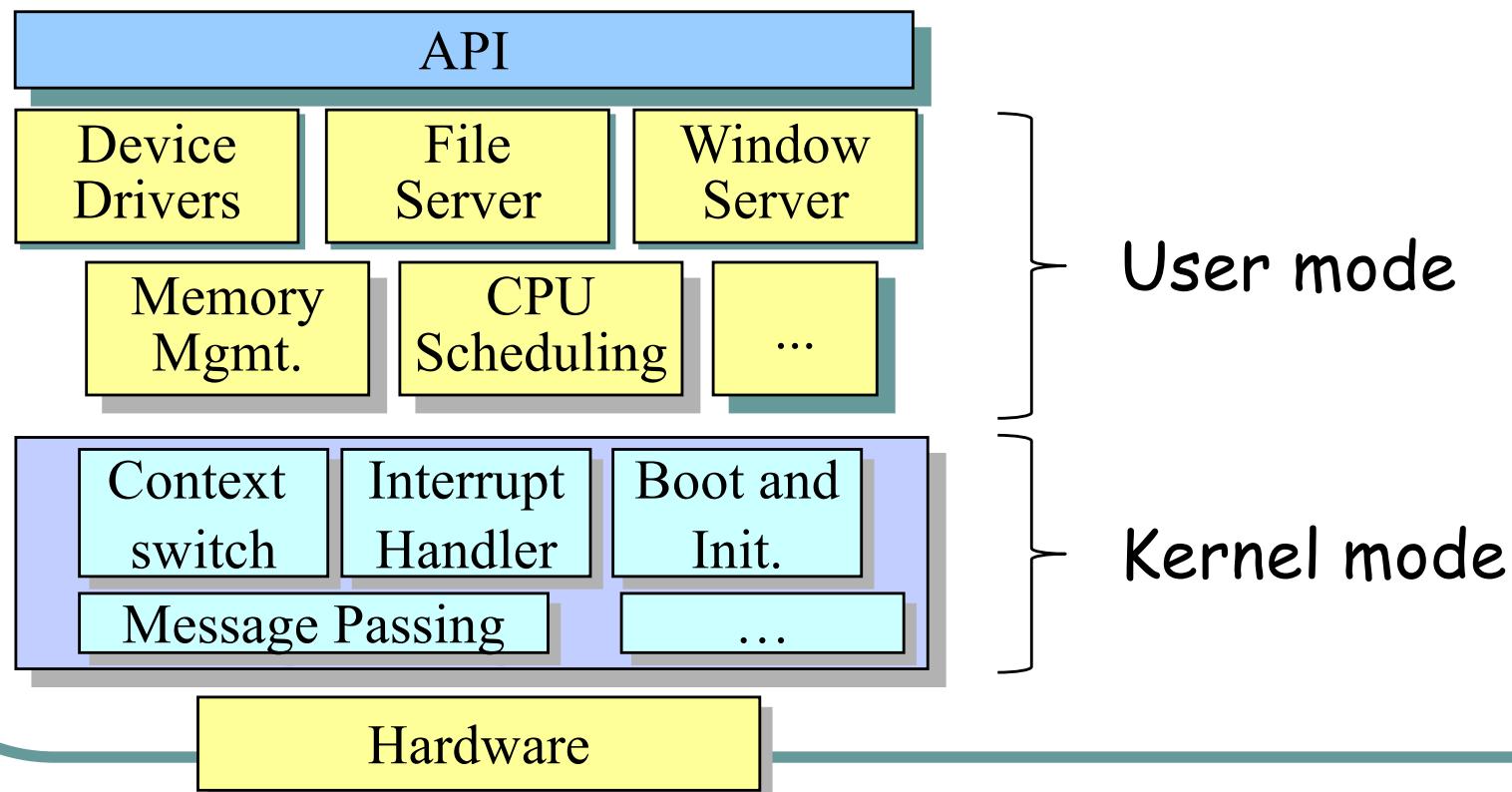


Figure 1: Growth of the Linux 2.6 kernel since its release.



Architetture a microkernel

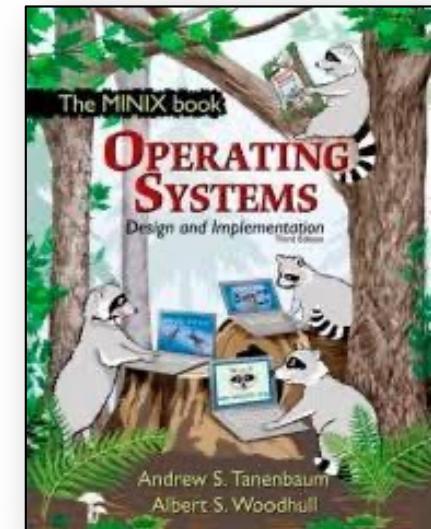
- Il **microkernel** implementa solo i **meccanismi** essenziali (es. comunicazione tra processi)
- Le **politiche di gestione** sono implementate all'esterno del kernel, in processi di sistema (user mode)





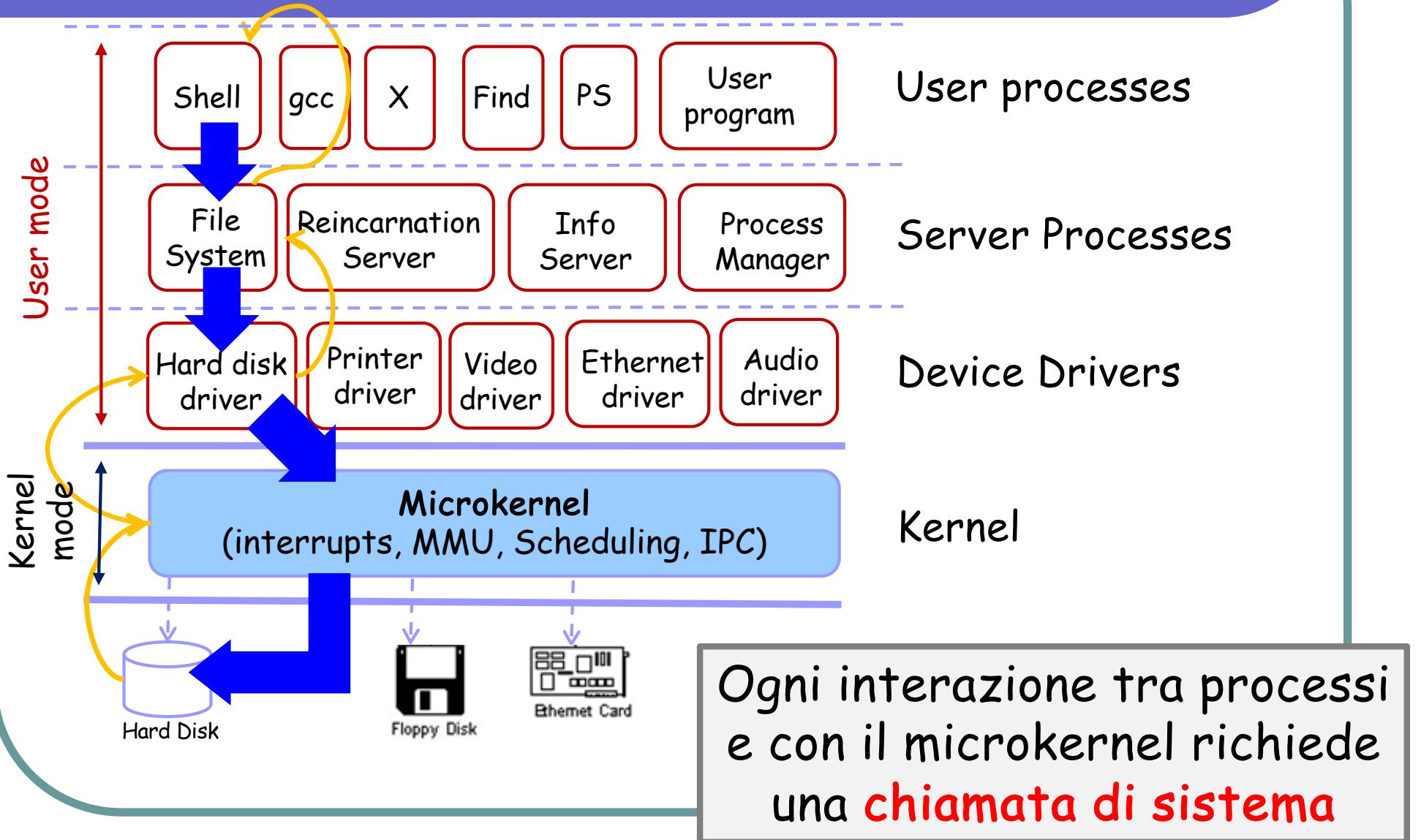
Esempio di microkernel: MINIX 3

- Le risorse sono gestite da processi di sistema, detti **server**
 - file server
 - terminal server
 - printer server
 - ...
- Quando un processo utente (**client**) deve usare una risorsa (es. un file), **invia ad un server una richiesta**
- Si utilizzano dei **meccanismi di comunicazione** forniti dal microkernel





Architetture a microkernel: Minix



La disputa Linux vs. Minix



LINUX is obsolete 91154 visualizzazioni



ast

a

I was in the U.S. for a couple of weeks, so I haven't commented much on LINUX (not that I would have said much had I been around), but for what it is worth, I have a couple of comments now.

As most of you know, for me MINIX is a hobby, something that I do in the evening when I get bored writing books and there are no major wars, revolutions, or senate hearings being televised live on CNN. My real job is a professor and researcher in the area of operating systems.

As a result of my occupation, I think I know a bit about where operating systems are going in the next decade or so. Two aspects stand out:

1. MICROKERNEL VS MONOLITHIC SYSTEM

Most older operating systems are monolithic, that is, the whole operating system is a single a.out file that runs in 'kernel mode.' This binary contains the process management, memory management, file system and the rest. Examples of such systems are UNIX, MS-DOS, VMS, MVS, OS/360, MULTICS, and many more.

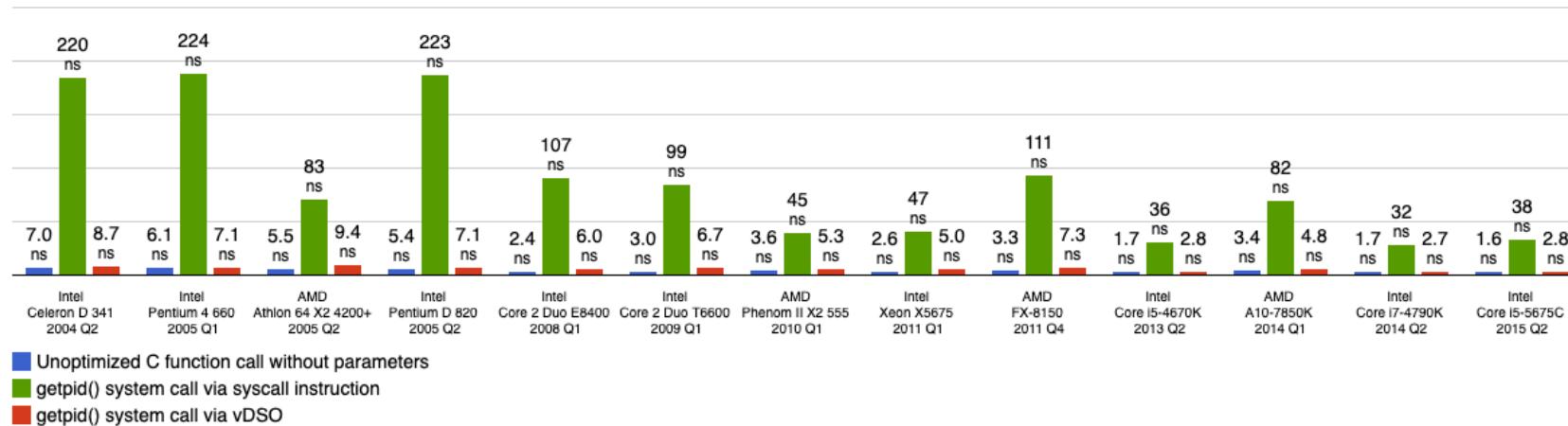


https://en.wikipedia.org/wiki/Tanenbaum%20vs%20Torvalds_debate
<https://groups.google.com/g/comp.os.minix/c/wlhw16QWltI>



User-space vs. kernel-space

- Le chiamate di sistema sono **significativamente più lente** di normali chiamate a funzione
- I moduli user-space hanno l'inconveniente di **incrementare** la quantità di chiamate di sistema





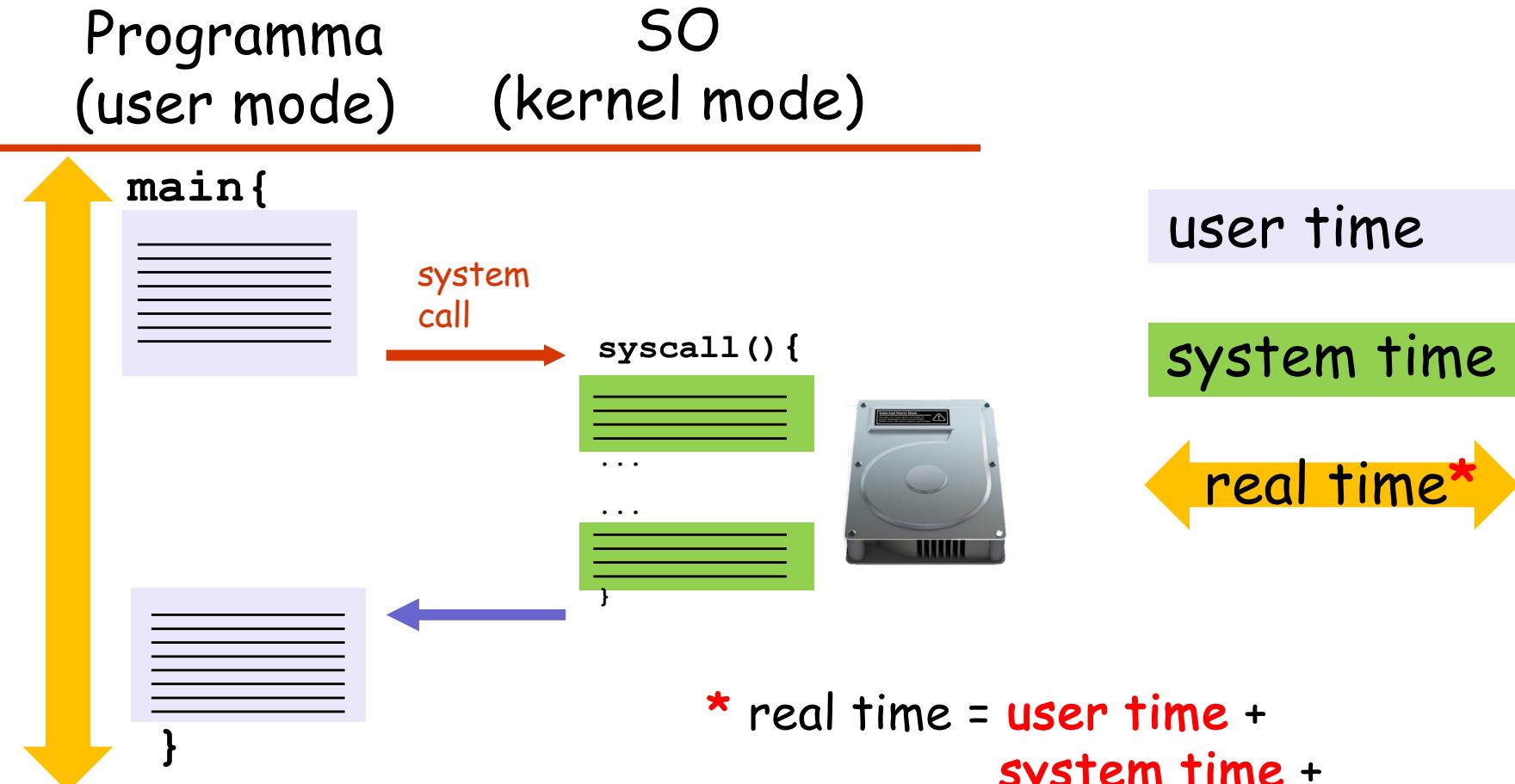
User/System Time in UNIX

- Il comando "**time**" di UNIX misura il tempo che un processo trascorre in **user-mode** e in **kernel-mode**

```
$ time ls
...
real 0m0.023s
user 0m0.004s
sys 0m0.008s
```



User/System Time in UNIX



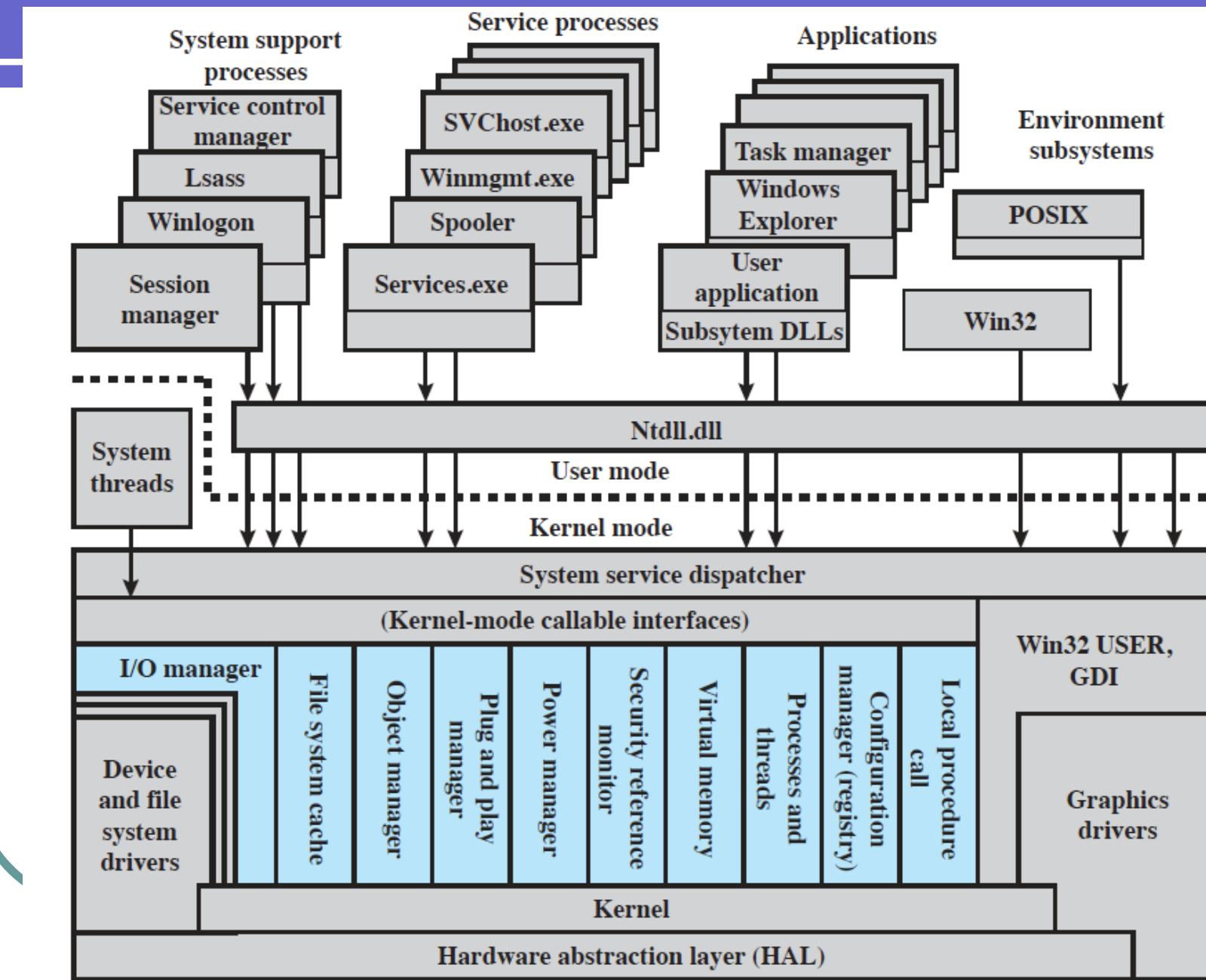
* real time = **user time** +
system time +
tempo in stato **sospeso**
(attesa I/O o CPU)



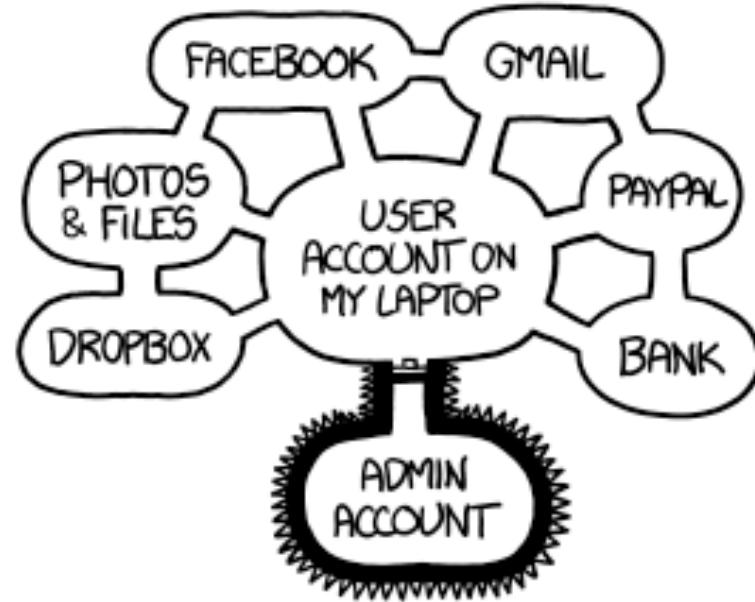
Windows Architecture

- Windows è basato su un'architettura ibrida
 - Non è puramente un'architettura modulare o microkernel
 - Molti componenti **non orientati alle performance** (gestione sessioni, autenticazione, servizi di sistema, ...) sono **spostati in user-space**

Windows Architecture



La sicurezza dei SO oggi...



<https://xkcd.com/1200/>

If someone **steals my laptop**, they can read my email, take my money, and impersonate me to my friends...

...but at least, **they can't install drivers** without my permission!

Quiz



1. Le system call sono chiamate mediante:

- Istruzioni di salto (JSR)
- Istruzioni che generano interrupt sincrone (SVC)

<https://forms.office.com/r/5zW4dwArjq>



Quiz



2. Una system call è una funzione che:

- è parte delle librerie di sistema, esegue in user mode
- è parte delle librerie di sistema, esegue in kernel mode
- è parte del kernel del SO, esegue in user mode
- è parte del kernel del SO, esegue in kernel mode

3. La funzione "printf":

- È una funzione di libreria
- È una chiamata di sistema

<https://forms.office.com/r/5zW4dwArjq>

