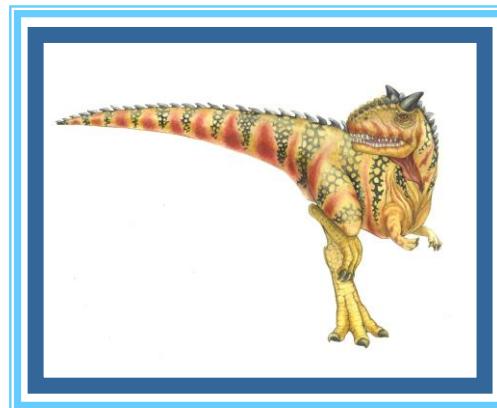


Struttura dei Sistemi Operativi

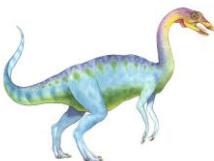




Obiettivi

- ❖ Identificare i servizi forniti da un sistema operativo
- ❖ Illustrare come si utilizzano le chiamate di sistema per ottenere servizi dal SO
- ❖ Confrontare strategie monolitiche, stratificate, microkernel, modulari e ibride per la progettazione di sistemi operativi
- ❖ Illustrare il processo di generazione di un sistema operativo
- ❖ Utilizzare strumenti di monitoraggio delle prestazioni

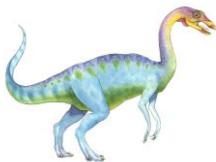




Sommario

- ❖ Servizi del sistema operativo
- ❖ Interfaccia utente
- ❖ Chiamate di sistema
- ❖ Programmi di sistema
- ❖ Linker e loader
- ❖ Progettazione e realizzazione
- ❖ Struttura del sistema operativo
- ❖ Generazione e avvio del sistema operativo
- ❖ Debugging del sistema operativo





Introduzione

- ❖ I sistemi operativi forniscono l'ambiente in cui si eseguono i programmi
 - Essendo organizzati secondo criteri che possono essere assai diversi (e dovendo "girare" su architetture molto differenziate), tale può essere anche la loro struttura interna
- ❖ La progettazione di un nuovo SO è un compito difficile
 - ⇒ il tipo di sistema desiderato definisce i criteri di scelta dei metodi e gli algoritmi implementati
- ❖ In fase di progettazione, il sistema operativo può essere definito/valutato in base a...
 - ...i servizi che esso dovrà fornire
 - ...l'interfaccia messa a disposizione di programmatore e utenti
 - ...la complessità di realizzazione



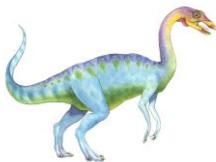


Servizi del sistema operativo – 1

❖ Servizi user-oriented

- **Interfaccia utente** — tutti gli attuali SO general-purpose sono dotati di un'interfaccia utente, a linea di comando (*Command Line Interface, CLI*) e grafica (*Graphic User Interface, GUI*), o touch-screen
- **Esecuzione di programmi** — capacità di caricare un programma in memoria ed eseguirlo, eventualmente rilevando, ed opportunamente gestendo, situazioni di errore
- **Operazioni di I/O** — il SO fornisce ai programmi utente i mezzi per effettuare l'I/O su file o periferica
- **Gestione del file system** — capacità dei programmi di creare, leggere, scrivere e cancellare file e muoversi nella struttura delle directory





Servizi del sistema operativo – 2

❖ Servizi user-oriented (cont.)

- **Comunicazioni** — scambio di informazioni fra processi in esecuzione sullo stesso elaboratore o su sistemi diversi connessi via rete
 - ▶ Le comunicazioni possono avvenire utilizzando *memoria condivisa* o *scambio di messaggi*
- **Rilevamento di errori** — il SO deve tenere il sistema di calcolo sotto controllo costante, per rilevare errori, che possono verificarsi nella CPU e nella memoria, nei dispositivi di I/O o durante l'esecuzione di programmi utente
 - ▶ Per ciascun tipo di errore, il SO deve prendere le opportune precauzioni per mantenere una modalità operativa corretta e consistente
 - ▶ I servizi di debugging possono facilitare notevolmente la programmazione e, in generale, l'interazione con il sistema di calcolo

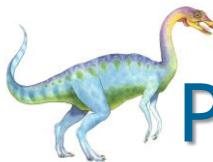




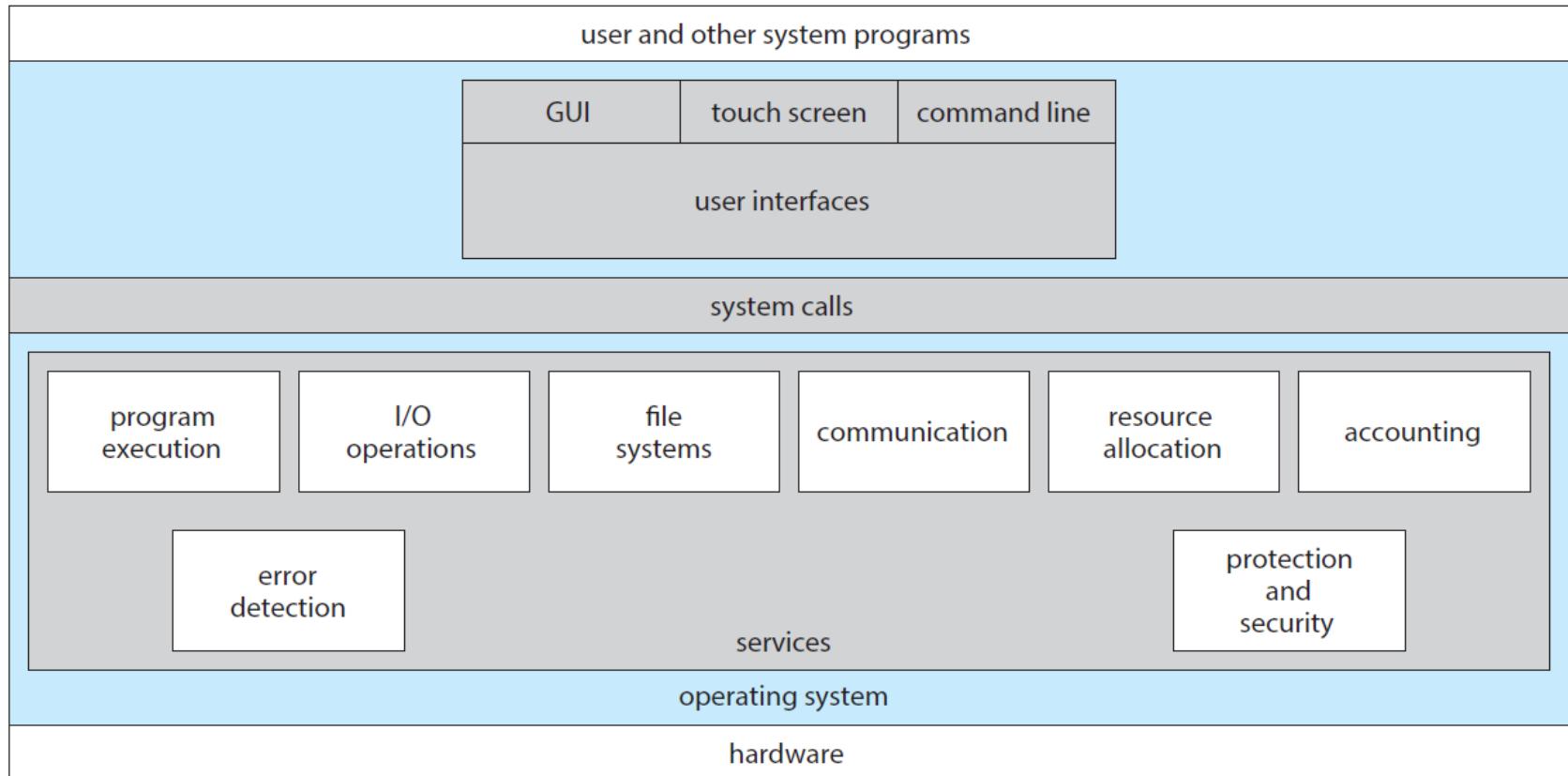
Servizi del sistema operativo – 3

- ❖ Esistono funzioni addizionali atte ad assicurare l'efficienza del sistema (non esplicitamente orientate all'utente)
 - **Allocazione di risorse** — quando più utenti o più processi vengono serviti in concorrenza, le risorse disponibili devono essere allocate equamente ad ognuno di essi
 - **Accounting e contabilizzazione dell'uso delle risorse** — tener traccia di quali utenti usano quali e quante risorse del sistema (utile per ottimizzare le prestazioni del sistema di calcolo)
 - **Protezione e sicurezza** — i possessori di informazioni memorizzate in un sistema multiutente o distribuito devono essere garantiti da accessi indesiderati ai propri dati; processi concorrenti non devono interferire fra loro
 - ▶ **Protezione:** assicurare che tutti gli accessi alle risorse di sistema siano regolamentati
 - ▶ **Sicurezza:** si basa sull'obbligo di identificazione tramite **password** e si estende alla difesa dei dispositivi di I/O esterni (es.: scheda di rete) da accessi illegali





Panoramica dei servizi del sistema operativo





Interfaccia utente CLI

- ❖ L'interfaccia utente a linea di comando permette di impartire direttamente comandi al SO (istruzioni di controllo)
 - Talvolta viene implementata nel kernel, altrimenti attraverso **programmi di sistema** (UNIX/Linux)
 - Può essere parzialmente personalizzabile, ovvero il SO può offrire più **shell**, più ambienti diversi, da cui l'utente può impartire le proprie istruzioni al sistema
 - ▶ Per UNIX/Linux, consultare:
<http://www.faqs.org/faqs/unix-faq/shell/shell-differences/>
 - La sua funzione è quella di interpretare ed eseguire le istruzioni impartite dall'utente (siano esse istruzioni built-in del SO o nomi di eseguibili utente) – **interprete dei comandi**





L'interprete dei comandi – 1

- ❖ I comandi ricevuti dall'interprete possono essere eseguiti secondo due modalità:
 - Se il codice relativo al comando è parte del codice dell'interprete, si effettua un salto all'opportuna sezione di codice
 - ⇒ Poiché ogni comando richiede il proprio segmento di codice (passaggio dei parametri e invocazione delle opportune chiamate di sistema), il numero dei comandi implementati determina le dimensioni dell'interprete
 - I comandi vengono implementati per mezzo di programmi di sistema
 - ⇒ I programmatori possono aggiungere nuovi comandi al sistema creando nuovi file con il nome appropriato (e memorizzandoli nelle directory opportune, es. **/usr/bin**)
 - ⇒ L'interprete dei comandi non viene modificato e può avere dimensioni ridotte





L'interprete dei comandi – 2

- ❖ In molti sistemi, solo un sottoinsieme delle funzionalità è disponibile via GUI e le funzioni meno comuni sono accessibili solo tramite linea di comando
- ❖ Le interfacce CLI:
 - Semplificano l'esecuzione di comandi ripetuti, perché sono programmabili
 - Un task eseguito di frequente e composto da più comandi può andare a costituire uno script
 - Gli shell script, molto comuni nei sistemi UNIX-like, non vengono compilati, ma interpretati dall'interfaccia a riga di comando

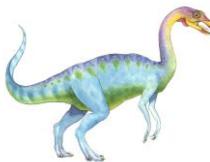




L'interprete dei comandi DOS

```
C:\>dir  
  
Volume in drive C is MS-DOS 5.0  
Volume Serial Number is 446B-2781  
Directory of C:\  
  
COMMAND.COM      47845 11-11-91  5:00a  
    1 file(s)     47845 bytes  
                  10280960 bytes free  
  
C:\>ver  
  
MS-DOS Version 5.00  
  
C:\>
```





L'interprete dei comandi di Solaris

Default

New Info Close Execute Bookmarks

Default Default

```
PBG-Mac-Pro:~ pbgs$ w
15:24 up 56 mins, 2 users, load averages: 1.51 1.53 1.65
USER TTY FROM LOGIN@ IDLE WHAT
pbgs console -
pbgs s000 -
PBG-Mac-Pro:~ pbgs$ iostat 5
          disk0      disk1      disk10     cpu    load average
KB/t tps MB/s KB/t tps MB/s KB/t tps MB/s us sy id 1m 5m 15m
33.75 343 11.30 64.31 14 0.88 39.67 0 0.02 11 5 84 1.51 1.53 1.65
  5.27 320 1.65  0.00 0 0.00  0.00 0 0.00  4 2 94 1.39 1.51 1.65
  4.28 329 1.37  0.00 0 0.00  0.00 0 0.00  5 3 92 1.44 1.51 1.65
AC
PBG-Mac-Pro:~ pbgs$ ls
Applications           Music           WebEx
Applications (Parallel) Pando Packages config.log
Desktop                Pictures         getsmartdata.txt
Documents              Public           imp
Downloads              Sites            log
Dropbox                Thumbs.db       panda-dist
Library                Virtual Machines prob.txt
Movies                 Volumes          scripts
PBG-Mac-Pro:~ pbgs$ pwd
/Users/pbgs
PBG-Mac-Pro:~ pbgs$ ping 192.168.1.1
PING 192.168.1.1 (192.168.1.1): 56 data bytes
64 bytes from 192.168.1.1: icmp_seq=0 ttl=64 time=2.257 ms
64 bytes from 192.168.1.1: icmp_seq=1 ttl=64 time=1.262 ms
AC
--- 192.168.1.1 ping statistics ---
2 packets transmitted, 2 packets received, 0.0% packet loss
round-trip min/avg/max/stddev = 1.262/1.760/2.257/0.498 ms
PBG-Mac-Pro:~ pbgs$ 
```

Mostra le statistiche per i dischi e la CPU (ogni 5 secondi)



Bourne shell, l'interprete dei comandi utilizzato da Solaris 10



L'interprete dei comandi di Mac OS

```
1. root@r6181-d5-us01:~ (ssh)
X root@r6181-d5-u... ● #1 X ssh #2 X root@r6181-d5-us01... #3

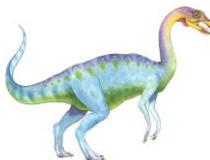
Last login: Thu Jul 14 08:47:01 on ttys002
iMacPro:~ pbgs$ ssh root@r6181-d5-us01
root@r6181-d5-us01's password:
Last login: Thu Jul 14 06:01:11 2016 from 172.16.16.162
[root@r6181-d5-us01 ~]# uptime
 06:57:48 up 16 days, 10:52,  3 users,  load
[root@r6181-d5-us01 ~]# df -kh
Filesystem      Size  Used Avail Use% Mounted on
/dev/mapper/vg_ks-lv_root
                  50G   19G   28G  41% /
tmpfs           127G  520K  127G   1% /tmp
/dev/sda1        477M   71M   381M  16% /boot
/dev/dssd0000    1.0T  480G  545G  47% /mnt/orangefs
tcp://192.168.150.1:3334/orangefs
                  12T  5.7T  6.4T  47% /mnt/orangefs
/dev/gpfs-test   23T  1.1T   22T   5% /mnt/gpfs
[root@r6181-d5-us01 ~]#
[root@r6181-d5-us01 ~]# ps aux | sort -nrk 3,3 | head -n 5
root      97653 11.2  6.6 42665344 17520636 ?  S<Ll Jul13 166:23 /usr/lpp/mmfs/bin/mmfsd
root      69849  6.6  0.0     0     0 ?  S  Jul12 181:54 [vpthread-1-1]
root      69850  6.4  0.0     0     0 ?  S  Jul12 177:42 [vpthread-1-2]
root      3829  3.0  0.0     0     0 ?  S  Jun27 730:04 [rp_thread 7:0]
root      3826  3.0  0.0     0     0 ?  S  Jun27 728:08 [rp_thread 6:0]
[root@r6181-d5-us01 ~]# ls -l /usr/lpp/mmfs/bin/mmfsd
-r-x----- 1 root root 20667161 Jun  3  2015 /usr/lpp/mmfs/bin/mmfsd
[root@r6181-d5-us01 ~]#
```

Sono visualizzati nell'ordine:

1. l'ora corrente
2. da quanto tempo il sistema è in funzione, espresso in giorni, ore e minuti
3. quanti sono gli utenti collegati al sistema
4. il carico medio del sistema nell'ultimo minuto
5. il carico medio negli ultimi cinque minuti
6. il carico medio negli ultimi quindici minuti

Bourne Again SHell (bash)





L'interprete dei comandi LINUX – 1

❖ Esempio

- A fronte del comando

```
$ rm file.txt
```

l'interprete cerca un file eseguibile chiamato `rm`, generalmente seguendo un percorso standard nel file system (`usr/bin`), lo carica in memoria e lo esegue con il parametro `file.txt`

- Esegue la cancellazione – *remove* – del file `file.txt`
- In alternativa...

```
$ rm -i file.txt
```

esegue la cancellazione solo dopo avere chiesto conferma all'utente





L'interprete dei comandi LINUX – 2

- ❖ **Esempio:** Un semplice script, salvato nel file **primo.sh**

```
cd  
mkdir d1  
chmod 444 d1  
cd d1
```

- Da linea di comando, si lancia

```
$ primo
```

ottenendo...

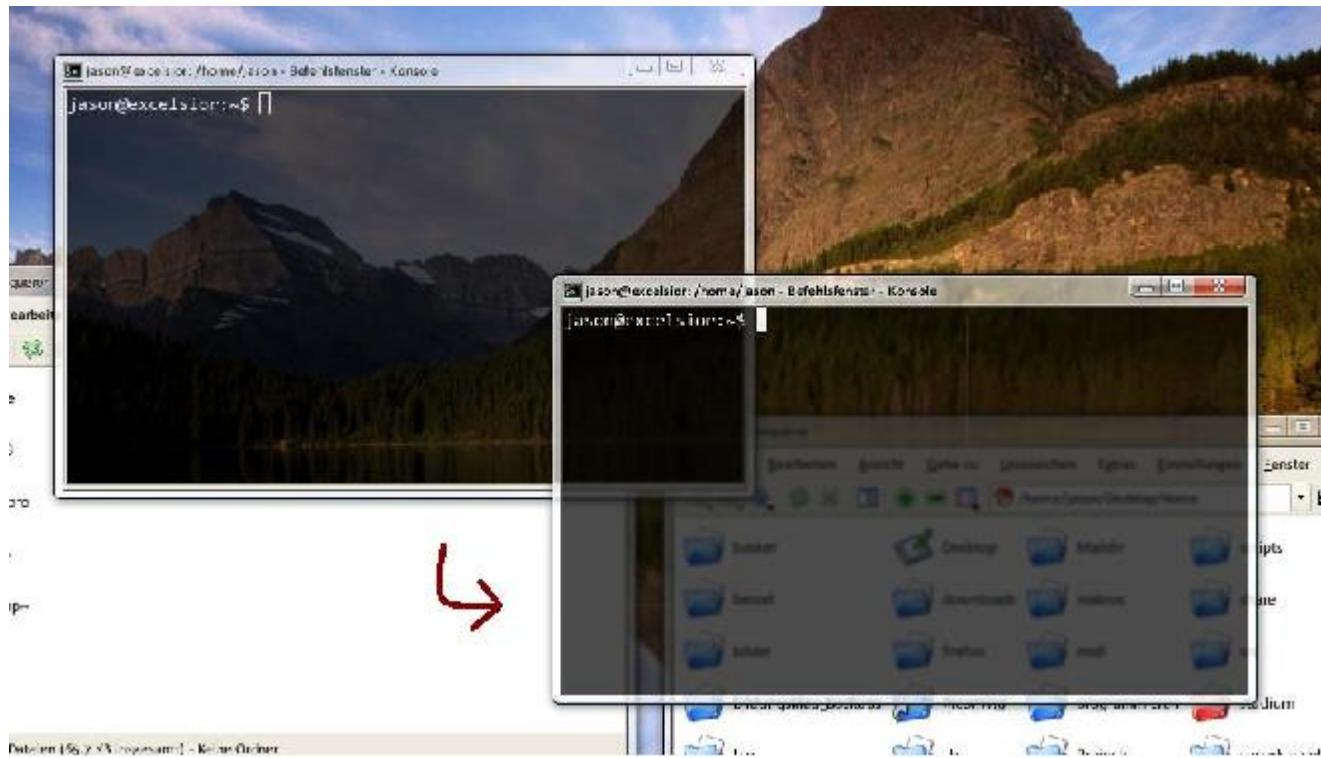
- ✖ **cd** cambia la directory corrente nella home dell'utente
- ✖ **mkdir d1** crea la directory **d1** nella directory corrente
- ✖ **chmod 444** imposta i permessi di **d1** a **r--r--r--**
- ✖ **cd d1** fallisce, provocando la visualizzazione del messaggio di errore

```
d1: Permission denied
```



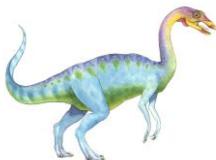


L'interprete dei comandi LINUX – 3



Bash shell è una shell testuale del progetto GNU, ma disponibile anche per alcuni sistemi Microsoft Windows





Interfaccia utente GUI – 1

- ❖ Interfaccia user-friendly che realizza la metafora della scrivania (**desktop**)
 - Interazione semplice tramite mouse, tastiera, monitor
 - Le **icone** rappresentano file, directory, programmi, etc.
 - I diversi tasti del mouse, posizionato su oggetti differenti, provocano diversi tipi di azione (forniscono informazioni sull'oggetto in questione, eseguono funzioni tipiche dell'oggetto, aprono directory – **folder**, o **cartelle**, nel gergo GUI)
 - Realizzate per la prima volta, all'inizio degli anni '70, dai laboratori di ricerca Xerox PARC di Palo Alto (computer Xerox Alto, 1973)





Interfaccia utente GUI – 2

Apple, Xerox e l'interfaccia grafica

Molti testi sulla nascita dell'interfaccia grafica riportano la storia di una Apple che approfittò del lavoro del centro di ricerca Xerox Palo Alto Research Center ([PARC](#)) e lo commercializzò con il Lisa e con il Macintosh, derubando Xerox ed i suoi ricercatori di gloria e profitti.

Si tratta di un falso mito: ad indagare meglio, ad esempio leggendo testi come ["Dealers of Lightning"](#), si scopre che Apple pagò per la possibilità di visitare i laboratori del PARC. In cambio della visione dei prototipi di GUI di Xerox e dei colloqui con ingegneri e sviluppatori, la Xerox ricevette in cambio l'opzione su un pacchetto di azioni di Apple, che all'epoca era in procinto quotarsi in borsa.

Addirittura i ricercatori di Xerox sulle prime si opposero a mostrare a Steve Jobs e ai suoi le idee sviluppate nel centro (nell'ordine: interfaccia grafica, stampante laser e ethernet) e lo fecero solo in seguito a pressione dei dirigenti.



Un dettaglio importante è che quella al PARC fu solo una dimostrazione e che non ci fu alcun passaggio di materiale o codice: il risultato fu indubbiamente di ispirare gli sviluppatori del Lisa e del Mac ma gli uomini di Apple reimplementarono, ripensarono e completarono le idee ancora acerbe di Xerox, in modo diverso ed originale.

Un caso esemplare fu quello di Bill Atkinson che, all'oscuro di come funzionasse il sistema di gestione della grafica su schermo del PARC e cercando di egualiarla, ne realizzò una versione superiore chiamata prima LisaGraf e poi [QuickDraw](#).



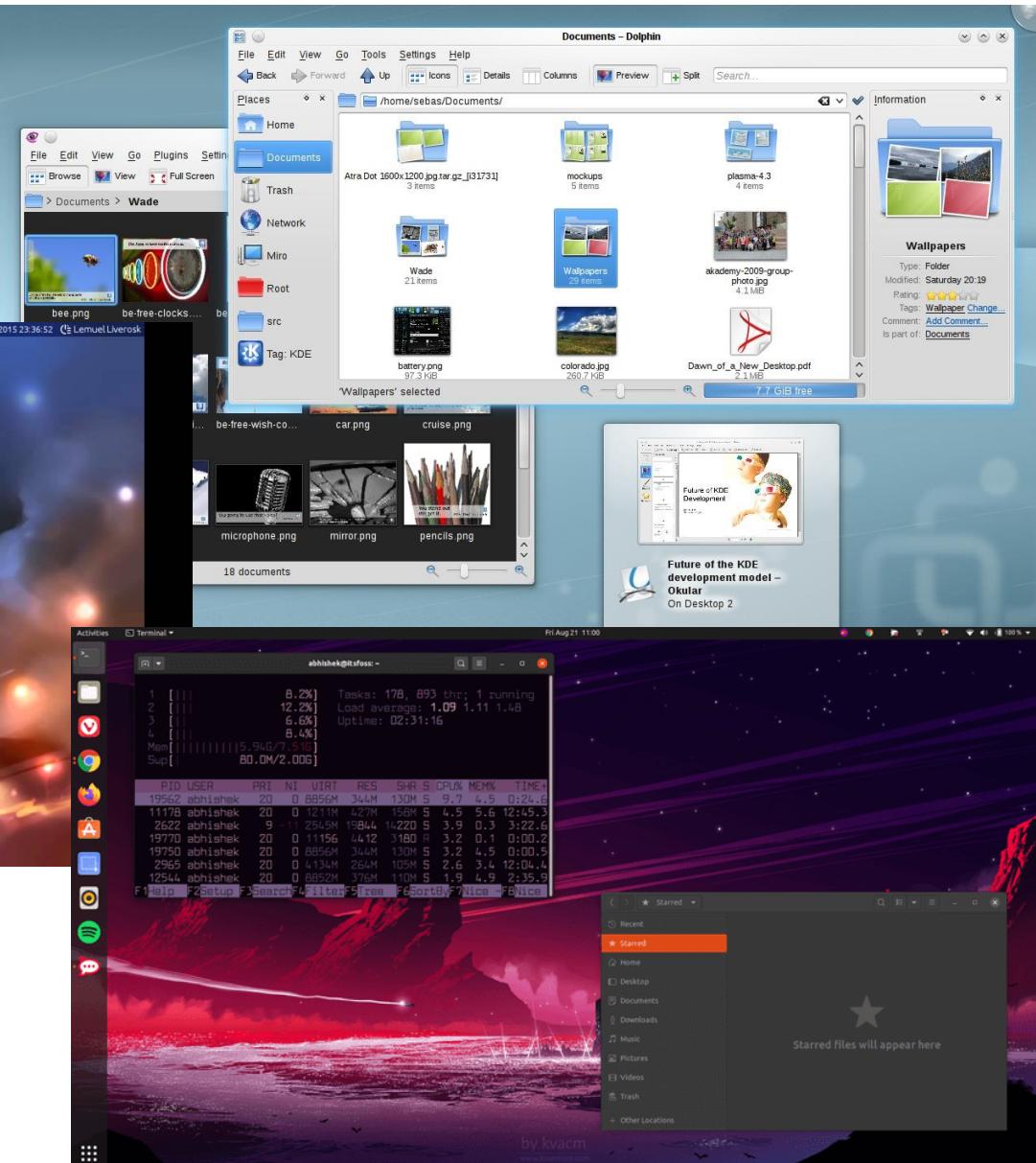
La perspicacia di Apple nel mettere in pratica le idee in nuce del PARC fu infine capita anche dagli stessi ricercatori del PARC: alcuni di loro, come Larry Tesler e Alan Kay, passarono a Cupertino perché frustrati dalla miopia di Xerox e lavorarono al Lisa ed a vari progetti di Apple nel corso degli anni '80.

Triste epilogo dei rapporti tra le due aziende è il ripensamento che ebbe Xerox nel dicembre 1989 quando provò a fare causa ad Apple sul copyright dell'interfaccia del Lisa e del Macintosh: l'anno dopo una corte statunitense pose fine ai procedimenti scagionando Apple.





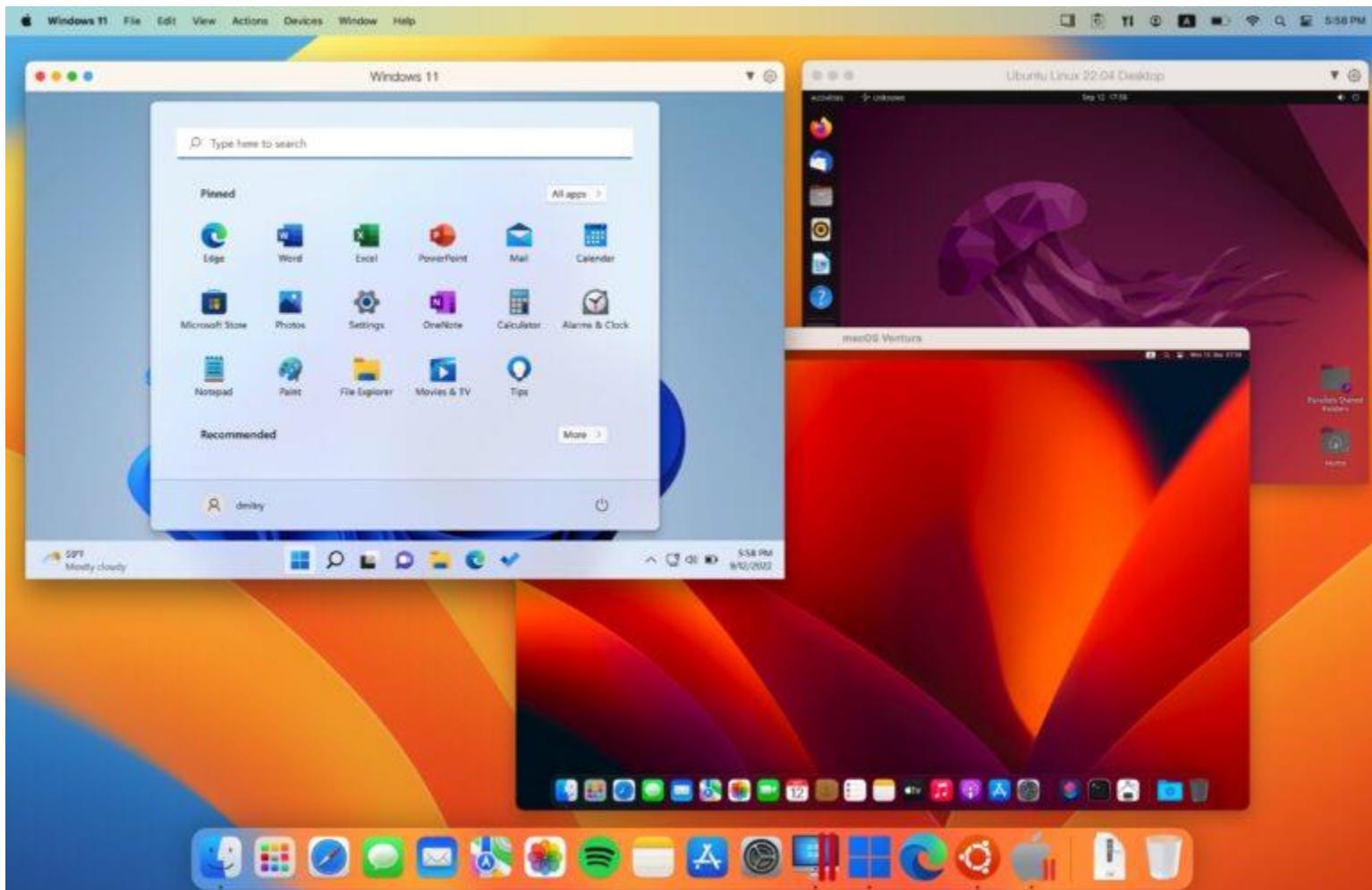
Interfaccia utente GUI – 3



Il desktop di GNU/Linux

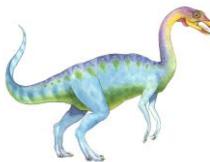


Interfaccia utente GUI – 4



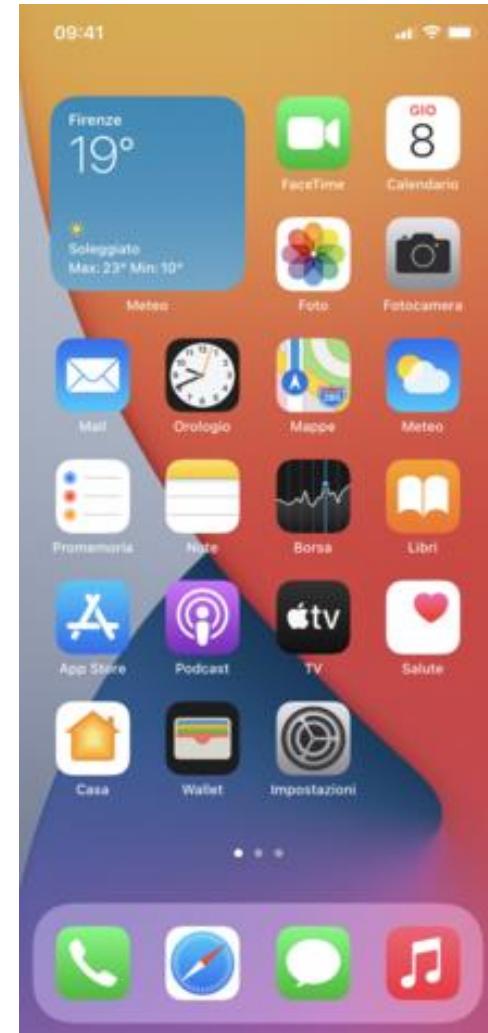
GUI di Mac OS (Ventura)





Interfaccia utente GUI – 5

- ❖ I device mobili con touch-screen richiedono nuovi tipi di interfaccia
 - Accesso senza il supporto del mouse (impossibile da usare o poco pratico)
 - Azioni ed operazioni di selezione realizzate tramite “gesti” (pressioni e strisciamenti delle dita)
 - Tastiera virtuale per l’immissione di testo
 - Comandi vocali



Springboard per iOS





Interfacce utente

- ❖ Molti sistemi operativi attuali includono interfacce sia CLI che GUI
 - **Microsoft Windows** principalmente basato su interfaccia grafica, ma dotato anche di una shell di comandi DOS-based (in Windows, tasto destro sul pulsante home → Windows PowerShell)
 - **Apple Mac OS** interagisce per mezzo della GUI “Aqua” (non una specifica interfaccia, ma una serie di regole legate alla sua realizzazione); inoltre, è dotato di un kernel UNIX e mette a disposizione diversi tipi di shell
 - **Solaris** è tipicamente CLI, con interfaccia GUI opzionale (Java Desktop, KDE)
 - **Linux** è modulare; si può scegliere tra GUI molto avanzate (KDE, GNOME, CDE, Unity,...) e la CLI





Chiamate di sistema

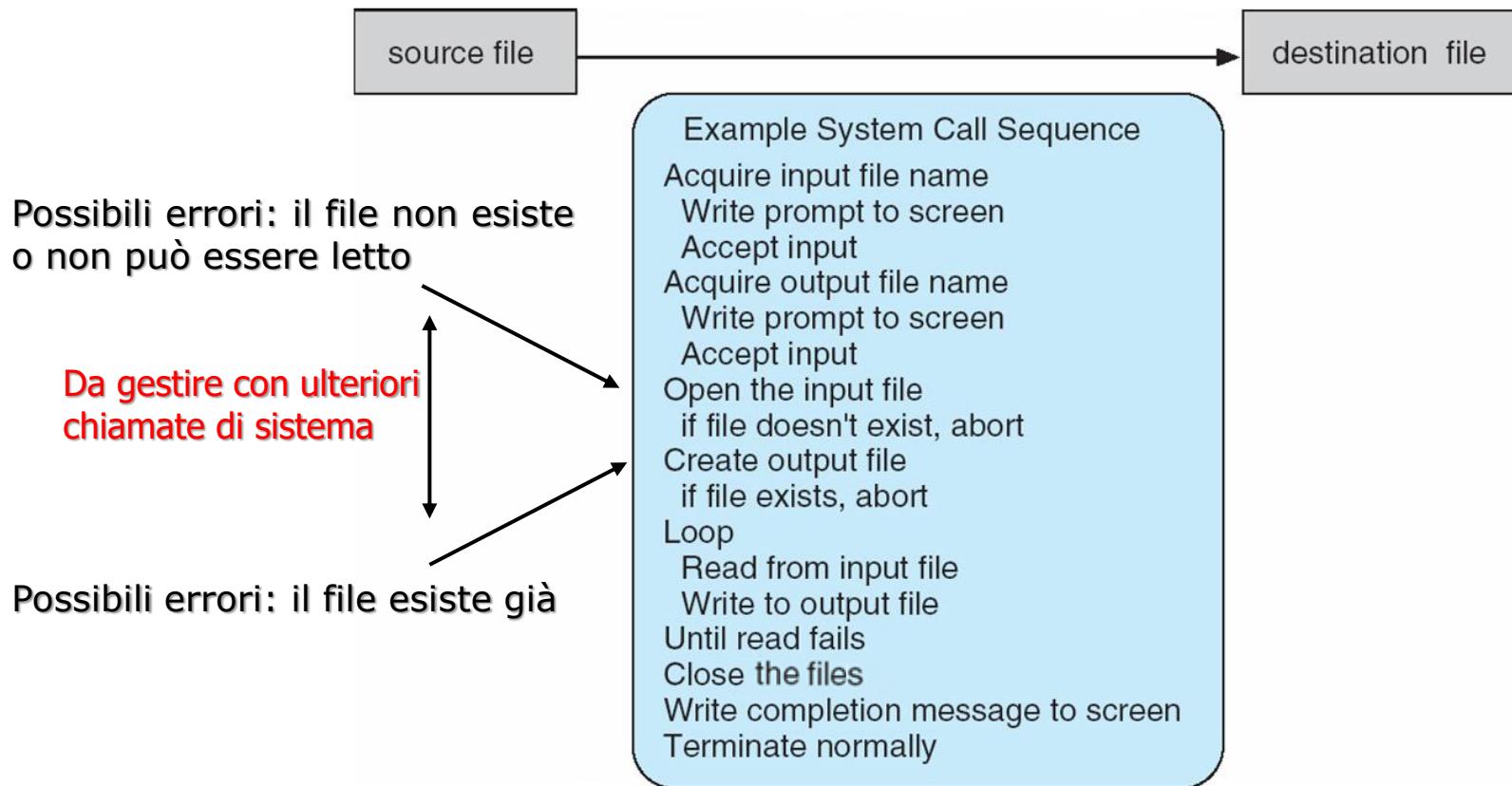
- ❖ Le chiamate al sistema forniscono l'interfaccia fra i processi e i servizi offerti dal SO
- ❖ Sono realizzate utilizzando linguaggi di alto livello (C o C++) e solo raramente l'assembly
- ❖ Normalmente, vengono richiamate dagli applicativi attraverso **API** (*Application Programming Interface*), piuttosto che per invocazione diretta
- ❖ Alcune API molto diffuse sono la Win64 API per Windows, la POSIX API per i sistemi POSIX-based (tutte le versioni di UNIX, Linux, e Mac OS), e la Java API per la Java Virtual Machine (JVM)
- ❖ **POSIX:** Portable Operating System Interface per UNIX





Esempio di chiamate di sistema

- ❖ Sequenza di chiamate di sistema per realizzare la copia di un file in un altro



⇒ Migliaia di chiamate di sistema al secondo!





Libreria standard del C come API

EXAMPLE OF STANDARD API

As an example of a standard API, consider the `read()` function that is available in UNIX and Linux systems. The API for this function is obtained from the `man` page by invoking the command

```
man read
```

on the command line. A description of this API appears below:

```
#include <unistd.h>

ssize_t      read(int fd, void *buf, size_t count)
```

return value function name parameters

A program that uses the `read()` function must include the `<unistd.h>` header file, as this file defines the `ssize_t` and `size_t` data types (among other things). The parameters passed to `read()` are as follows:

- `int fd`—the file descriptor to be read
- `void *buf`—a buffer where the data will be read into
- `size_t count`—the maximum number of bytes to be read into the buffer

On a successful read, the number of bytes read is returned. A return value of 0 indicates end of file. If an error occurs, `read()` returns `-1`.

The `<unistd.h>` file defines miscellaneous symbolic constants and types, and declares miscellaneous functions





Chiamate di sistema (cont.)

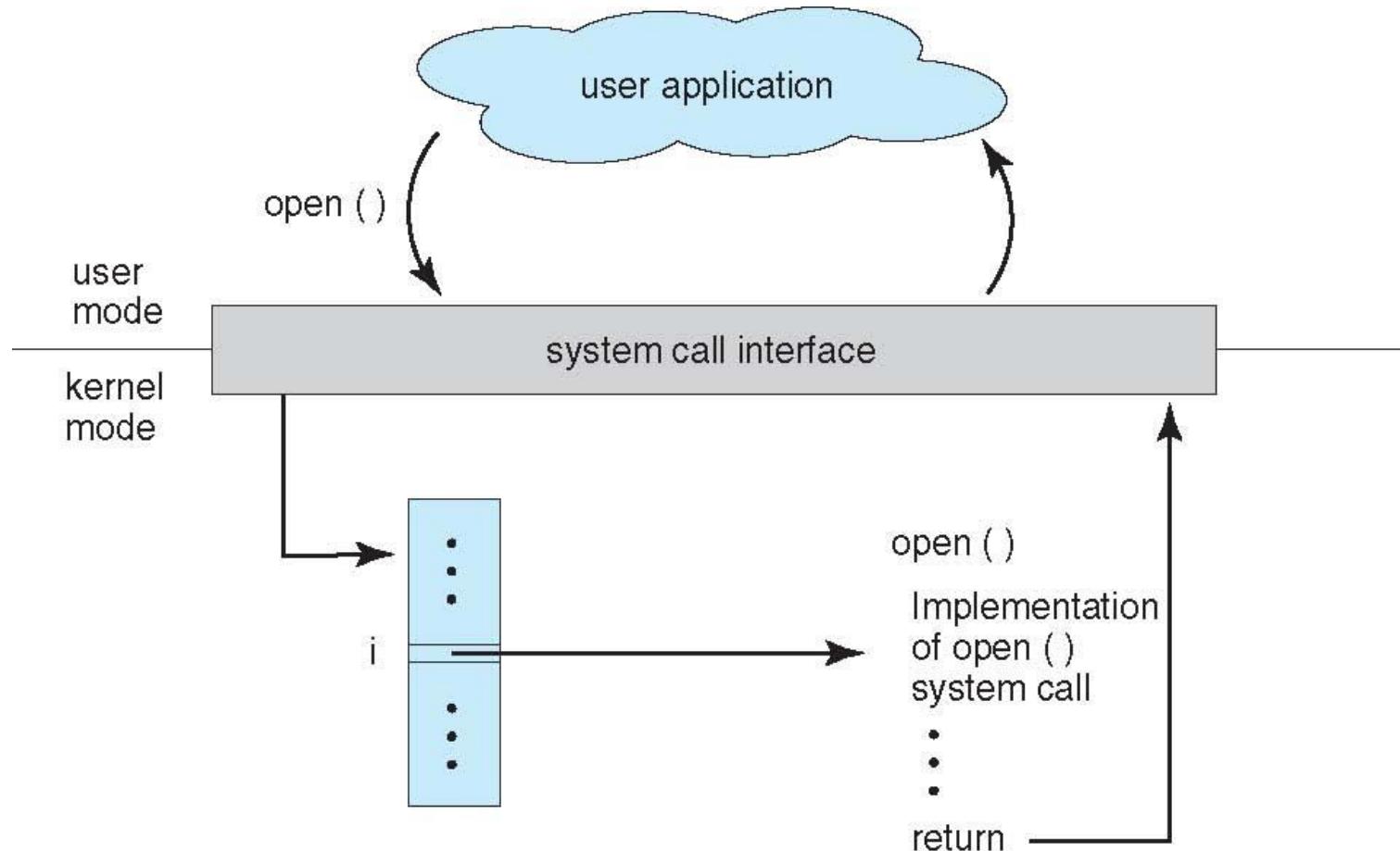
- ❖ Normalmente, a ciascuna *system call* è associato un numero
 - L'**interfaccia alle chiamate di sistema**, ovvero l'ambiente di esecuzione a run-time (RTE), mantiene una tabella indicizzata dal numero di system call, effettua la chiamata e ritorna al chiamante lo stato del sistema dopo l'esecuzione (ed eventuali valori restituiti)
- ❖ L'utente non deve conoscere i dettagli implementativi delle system call: deve conoscere la modalità di utilizzo dell'API (ed eventualmente il compito svolto dalle chiamate di sistema)
 - L'intermediazione della API garantisce la portabilità delle applicazioni
 - Molto spesso una system call viene chiamata tramite una funzione di libreria standard (ad esempio contenuta in `stdlibc`)





Relazioni API – System call – SO

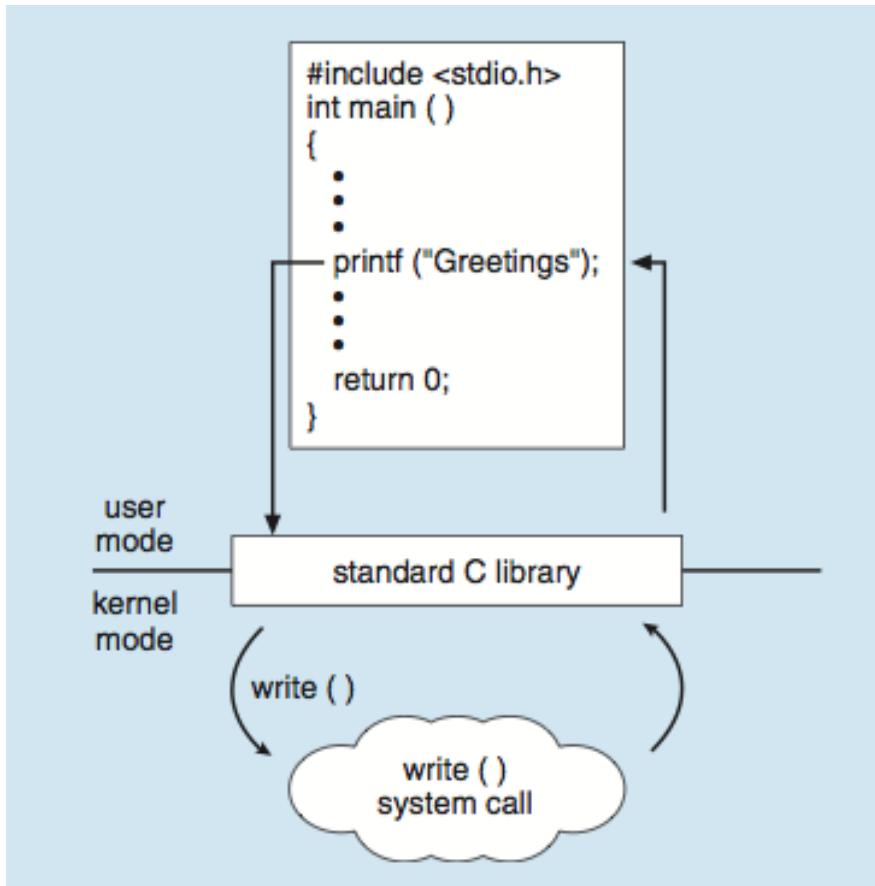
Gestione della chiamata di sistema `open()` invocata da un'applicazione utente





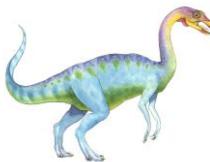
Esempi di syscall con `stdlibc` – 1

- ❖ Per Linux, la libreria standard del linguaggio C (il *run-time support system*) fornisce una parte della API



- Programma C che invoca la funzione di libreria per la stampa `printf()`
- La libreria C intercetta la funzione e invoca la system call `write()`
- La libreria riceve il valore restituito dalla chiamata al sistema e lo passa al programma utente





Esempi di syscall con stdlibc – 2

❖ Funzione C che copia il contenuto di un file in un altro

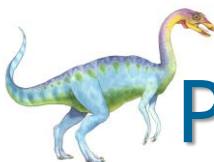
```
#include <stdio.h>
#include <stddef.h>
#define FAIL 0
#define SUCCESS 1

int copy_file(infile, outfile)
char *infile, *outfile;

{
    FILE *fp1, *fp2;
    if ((fp1 = fopen(infile, "rb")) == NULL)
        return FAIL;
    if ((fp2 = fopen(outfile, "wb")) == NULL)
    {
        fclose(fp1);
        return FAIL;
    }
    while (!feof(fp1))
        putc(getc(fp1), fp2);
    fclose(fp1);
    fclose(fp2);
    return SUCCESS;
}
```

- Per eseguire l'I/O, è necessario associare un flusso ad un file o a una periferica
 - ⇒ occorre dichiarare un puntatore alla struttura **FILE**
- La struttura **FILE**, definita in **stdio.h**, è costituita da campi che contengono informazioni quali il nome del file, la modalità di accesso, il puntatore al prossimo carattere nel flusso
- Entrambi i file vengono acceduti in modalità binaria
- La macro **getc()** legge il prossimo carattere dal flusso specificato e sposta l'indicatore di posizione del file avanti di un elemento ad ogni chiamata





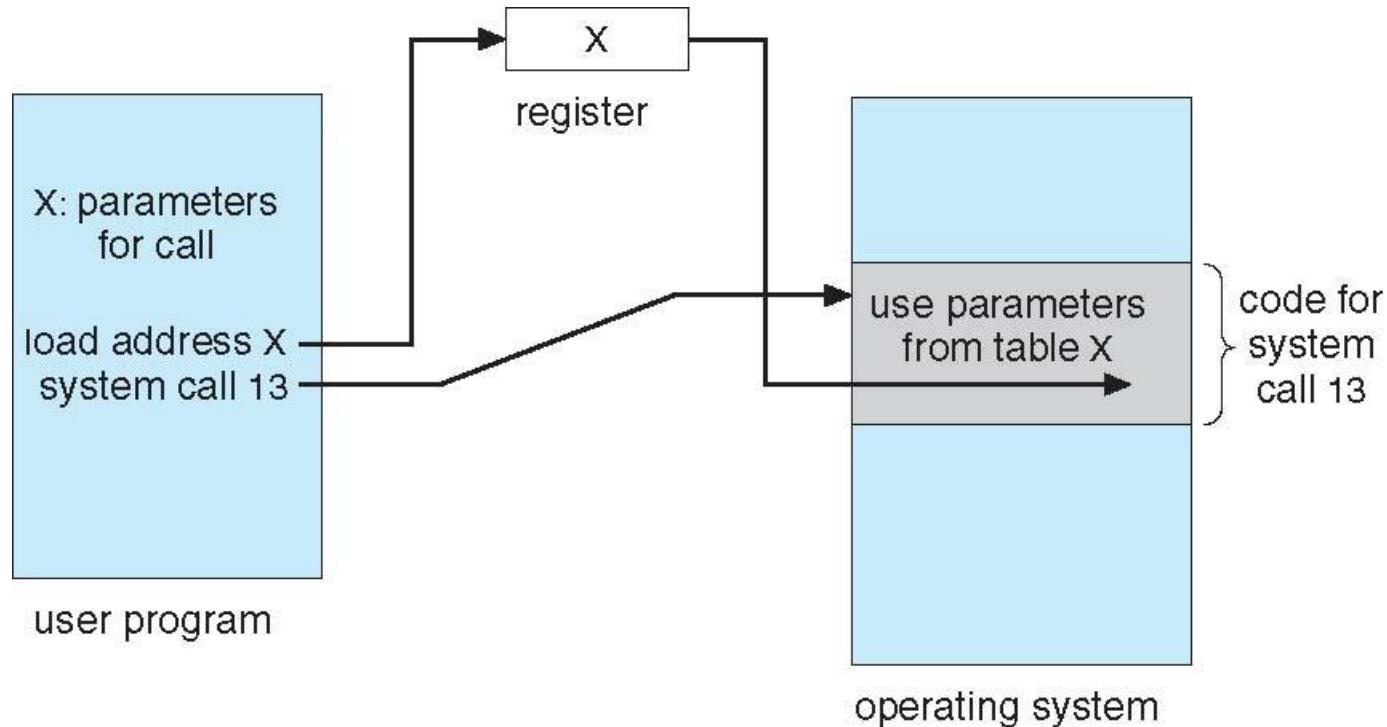
Passaggio di parametri alle system call

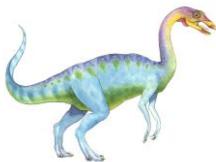
- ❖ Spesso l'informazione necessaria alla chiamata di sistema non si limita al solo nome (o numero di identificazione)
 - Il tipo e la quantità di informazione varia per chiamate diverse e diversi sistemi operativi
- ❖ Esistono tre metodi generali per passare parametri al SO
 - Il più semplice: passaggio di parametri nei registri
 - ▶ Talvolta, possono essere necessari più parametri dei registri presenti
 - Memorizzazione dei parametri in un blocco in memoria e passaggio dell'indirizzo del blocco come parametro in un registro
 - ▶ Approccio seguito da Linux (per > 5 parametri) e Solaris
 - *Push* dei parametri nello stack da parte del programma; il SO recupera i parametri con un *pop*
 - Gli ultimi due metodi non pongono limiti al numero ed alla lunghezza dei parametri passati





Passaggio di parametri per indirizzo





Tipi di chiamate di sistema – 1

- ❖ Controllo dei processi
- ❖ Gestione dei file
- ❖ Gestione dei dispositivi di I/O
- ❖ Gestione delle informazioni
- ❖ Comunicazione
- ❖ Protezione





Tipi di chiamate di sistema – 2

❖ Controllo dei processi

- Creazione e terminazione di un processo (**fork, exit**)
- Caricamento ed esecuzione (**exec/execve**)
- Lettura/modifica degli attributi di un processo (priorità, tempo massimo di esecuzione – **get/set process attributes**)
- Attesa per il tempo indicato o fino alla segnalazione di un evento (**wait/waitpid**)
- Assegnazione e rilascio di memoria (**alloc, free**)
- Invio di segnali (**signal, kill**)
- Dump della mappa di memoria in caso di errore
- Debugger ed esecuzione a passo singolo
- Gestione di lock per l'accesso a memoria condivisa

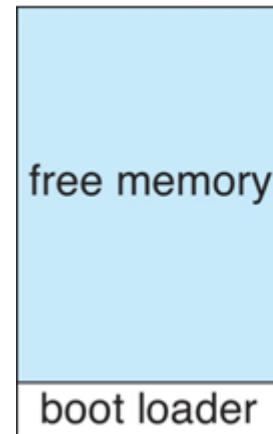




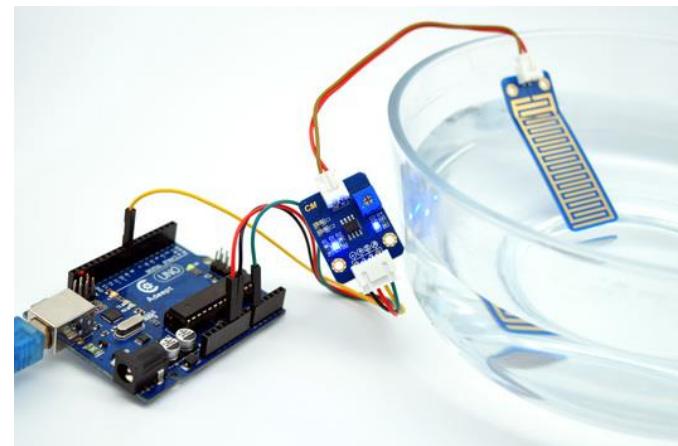
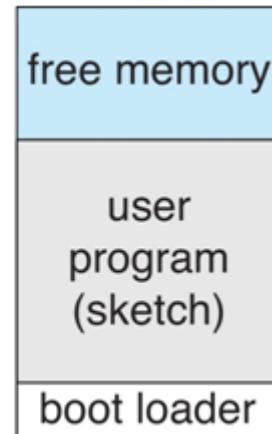
Controllo dei processi in Arduino

- ❖ Piattaforma hardware costituita da un microcontrollore e da sensori
- ❖ I programmi (*sketch*) devono essere preventivamente scritti e compilati su PC
- ❖ Gli sketch vengono caricati da una flash attraverso la porta USB
- ❖ È il *boot loader* che si occupa di caricare/scaricare i programmi
- ❖ Single-task
- ❖ Nessun SO (nessuna interfaccia utente)

Alla startup del sistema



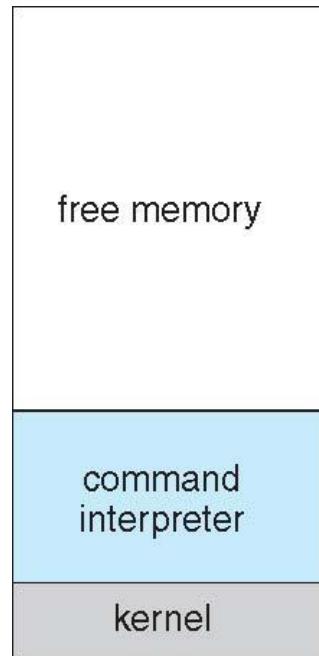
Durante l'esecuzione





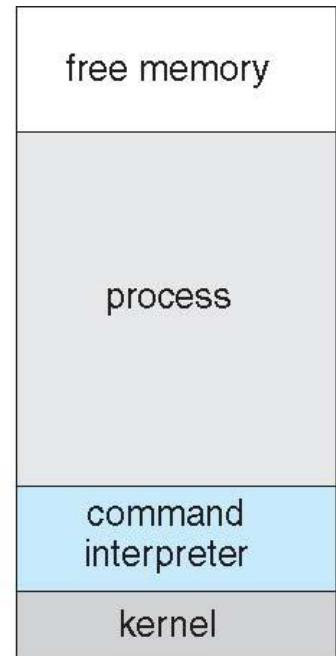
Controllo dei processi in MS-DOS

- ❖ Single-task
- ❖ La shell viene invocata al boot del sistema
- ❖ Metodo semplice di esecuzione dei processi
- ❖ Nessuna partizione della memoria
- ❖ Caricamento del processo in memoria a sovrascrivere lo stesso interprete (ma non il kernel)
- ❖ All'uscita dal programma → si ricarica la shell



(a)

Allo startup del sistema



(b)

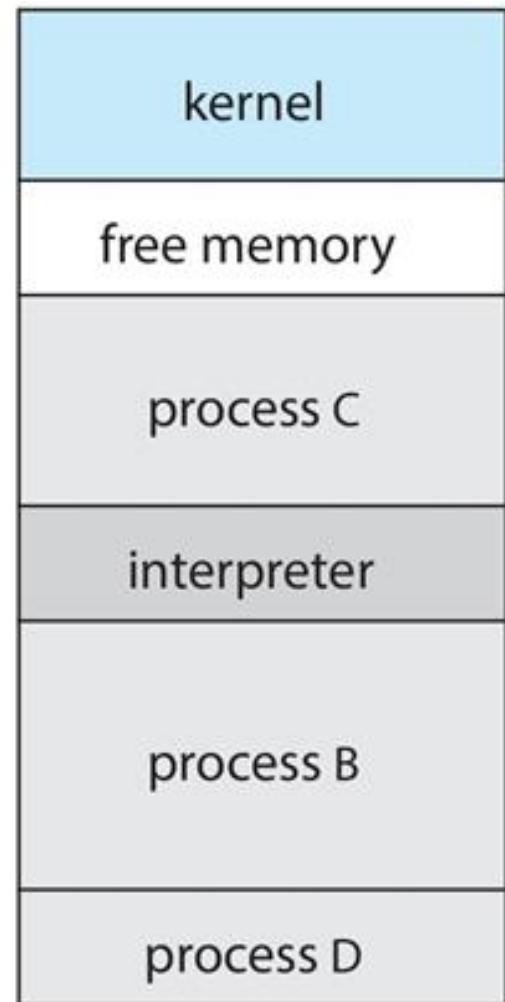
Durante l'esecuzione di un programma utente

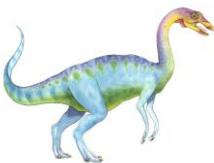




Controllo dei processi in FreeBSD

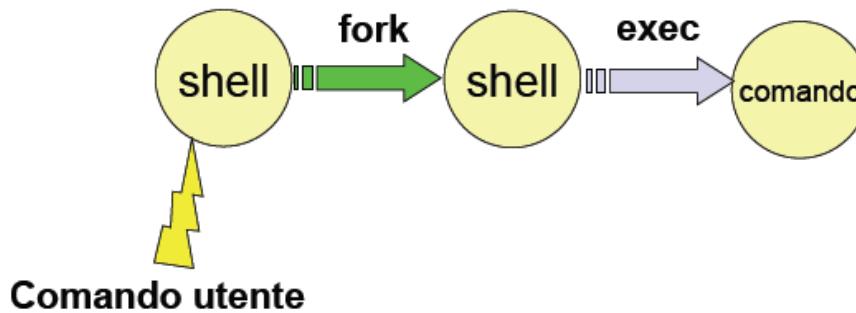
- ❖ È un sistema Unix open-source
 - ❖ Multitasking
 - ❖ User login → si invoca la shell scelta dall'utente
 - ❖ La shell esegue la system call **fork()** per creare nuovi processi
 - Si esegue la **exec()** per caricare il programma nel processo
 - La shell attende la terminazione del processo o continua ad eseguire i comandi utente
 - ❖ I processi terminano con:
 - codice = 0 – esecuzione corretta
 - codice > 0 – tipo di errore





Esempio 1

- ❖ Per ogni comando, la shell genera un processo figlio, una nuova shell, dedicato all'esecuzione del comando:



- ❖ Possibilità di due diversi comportamenti
 - Il padre si pone in attesa della terminazione del figlio (esecuzione in foreground)
→ `$ ls -l prog* > proginfo`
 - Il padre procede nell'esecuzione concorrentemente al figlio (esecuzione in background)
→ `$ ls -l prog* > proginfo &`





Esempio 2

- ❖ Cosa producono in stampa questi codici?

```
main ()  
{  
    val = 5;  
    if(fork())  
        wait(&val);  
    val++;  
    printf("%d\n", val);  
    return val;  
}
```

Restituisce 0 al processo figlio, il pid del figlio al processo padre

```
main ()  
{  
    val = 5;  
    if(fork())  
        wait(&val);  
    else  
        return val;  
    val++;  
    printf("%d\n", val);  
    return val;  
}
```

Il processo figlio incrementa il valore di val e lo stampa, quindi lo restituisce al padre (che era in attesa)

Il padre incrementa ancora val e lo stampa
⇒ 6, 7

Il processo figlio termina immediatamente restituendo il controllo al padre
Il padre incrementa val e lo stampa
⇒ 6

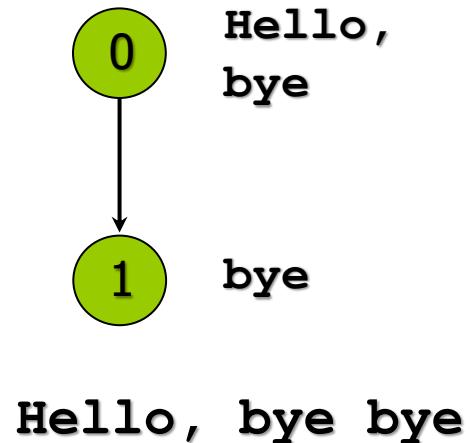




Esercizio 1

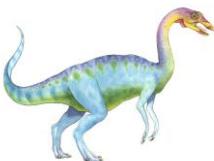
- ❖ Si consideri il codice seguente:

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
int main(void)
{
    printf("Hello, ");
    fork();
    printf("bye ");
    return 0;
}
```



Quale stampa produce la sua esecuzione? Fornire una chiara spiegazione alla risposta data.

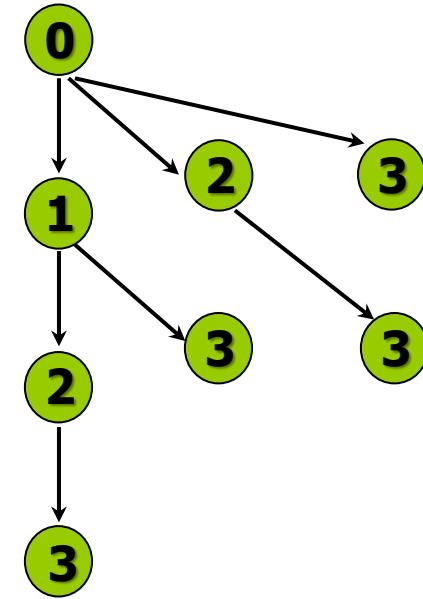




Esercizio 2

- ❖ Si consideri il seguente programma:

```
int main()
{
    pid_t pid;
    fork();
    fork();
    pid = fork();
    if (!pid)
        printf("Hello World\n");
}
```



Hello World
Stampato da tutti i nodi etichettati con 3

Si descriva l'albero dei processi creati e si dica quante volte viene stampata la stringa "Hello World".

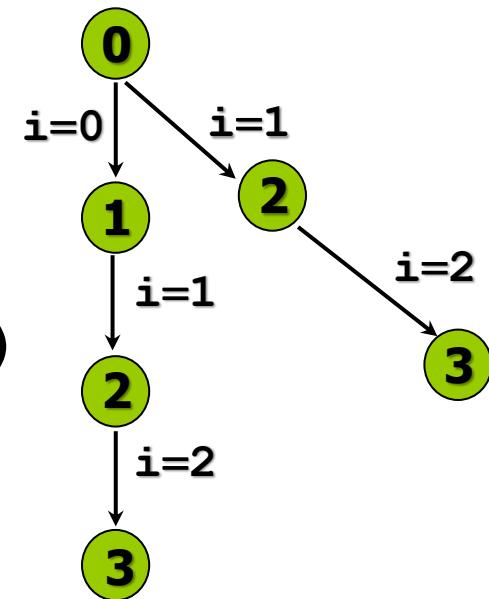




Esercizio 3

- ❖ Si stabilisca il numero di processi creati dal seguente codice, giustificando la risposta data con una descrizione dettagliata del codice stesso.

```
#include <unistd.h>
int main(void) {
    int i;
    for (i = 0; i < 3; i++) {
        if (fork() && (i == 1))
        {
            break;
        }
    }
}
```





Esercizio 4

- ❖ Si consideri il seguente codice concorrente:

```
main() {  
    int a1=0, a2=0, a3=0;  
    a1=fork();  
    a2=fork();  
    if (a1 != 0) {  
        a3=fork();  
        printf("0");  
    }  
    return 0;  
}
```

Si disegni il relativo albero dei processi motivandone la costruzione. Si descriva inoltre l'output prodotto dal programma.





Tipi di chiamate di sistema – 3

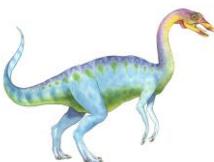
❖ Gestione dei file

- Creazione e cancellazione di file (`create`, `delete`)
- Apertura e chiusura di file (`open`, `close`)
- Lettura, scrittura e posizionamento (`read`, `write`, `seek`)
- Lettura/modifica degli attributi di un file (nome, tipo, codici di protezione, informazioni di contabilizzazione – `get/set file attributes`, `mv`, `chmod`, `chown`, `chgrp`)

❖ Gestione dei dispositivi di I/O

- Richiesta e rilascio di un dispositivo (`request`, `release`)
- Lettura, scrittura e posizionamento (`read`, `write`, `seek`)
- Lettura/modifica degli attributi di un dispositivo (`ioctl`)
- Connessione/disconnessione logica dei dispositivi





File e dispositivi

- ❖ Tutte le risorse controllate dal sistema operativo si possono concepire come **dispositivi**, alcuni dei quali sono fisici (come memorie non volatili e stampanti), mentre altri sono astratti o virtuali (i file, per esempio)
- ❖ La somiglianza fra file e dispositivi è infatti tale che molti SO, tra cui UNIX e Linux, ne forniscono una visione unificata
- ❖ In tal caso, l'insieme di system call è comune (eventualmente con differenze sui parametri)
- ❖ Linux rappresenta i dispositivi collegati al sistema come file nel file system e li colleziona nella directory **/dev**





Tipi di chiamate di sistema – 4

❖ Gestione delle informazioni

- Lettura/modifica dell'ora e della data (**time**, **date**)
- Informazioni sul sistema (**uname -r**)
- Lettura/modifica degli attributi di processi, file e dispositivi (**ps**, **getpid**)

❖ Comunicazione

- Apertura e chiusura di una connessione (**open connection**, **close connection**, **pipe**)
- Invio e ricezione di messaggi (**send**, **receive**)
- Informazioni sullo stato dei trasferimenti (**traceroute**)
- Inserimento ed esclusione di dispositivi remoti
- Condivisione della memoria (**shm_open**, **shmget**, **mmap**)

❖ Protezione

- Controllo di accesso alle risorse
- Lettura/modifica dei permessi di accesso (**chown**, **chmod**)
- Accreditamento degli utenti

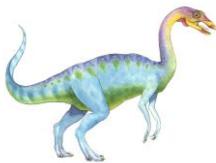




Esempi di chiamate di sistema in Windows e UNIX

	Windows	Unix
Process Control	CreateProcess() ExitProcess() WaitForSingleObject()	fork() exit() wait()
File Manipulation	CreateFile() ReadFile() WriteFile() CloseHandle()	open() read() write() close()
Device Manipulation	SetConsoleMode() ReadConsole() WriteConsole()	ioctl() read() write()
Information Maintenance	GetCurrentProcessID() SetTimer() Sleep()	getpid() alarm() sleep()
Communication	CreatePipe() CreateFileMapping() MapViewOfFile()	pipe() shmget() mmap()
Protection	SetFileSecurity() InitializeSecurityDescriptor() SetSecurityDescriptorGroup()	chmod() umask() chown()





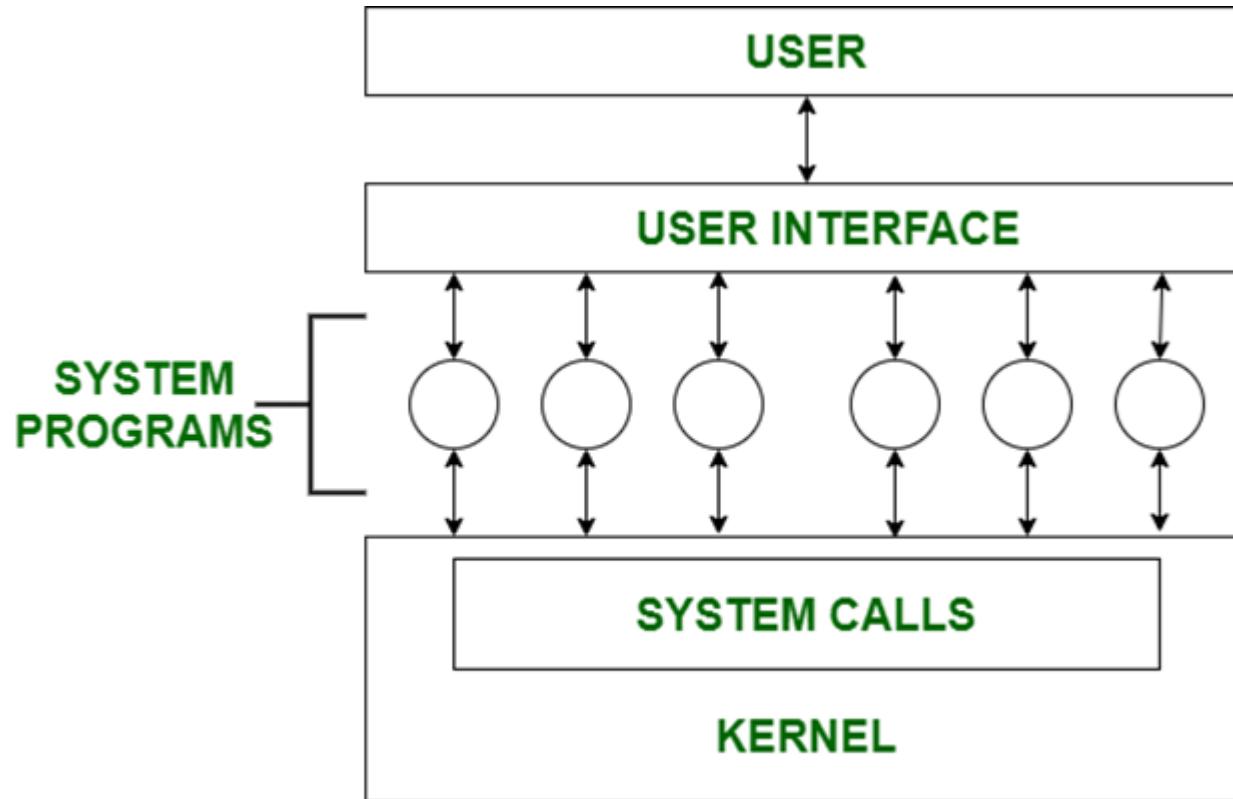
Programmi di sistema – 1

- ❖ I programmi di sistema forniscono un ambiente conveniente per lo sviluppo e l'esecuzione di programmi utente (semplici interfacce alle system call o programmi complessi)
- ❖ Esistono programmi di sistema per...
 - Gestione di file (es.: **cp**)
 - Informazioni di stato (es.: **du**, **who**)
 - Editing di file (es.: **vi**)
 - Supporto a linguaggi di programmazione (es.: **gcc**)
 - Caricamento e esecuzione di programmi (linker e loader)
 - Comunicazioni (es.: **telnet**, **rlogin**, **ftp**)
 - Supporto alla realizzazione di applicativi
- ❖ L'aspetto del SO per la maggioranza degli utenti è definito dai programmi di sistema, non dalle chiamate di sistema vere e proprie





Programmi di sistema – 2





Programmi di sistema – 3

- ❖ **Gestione di file** – per creare, cancellare, copiare, rinominare, listare, stampare e, genericamente, gestire le operazioni su file e directory
- ❖ **Informazioni di stato**
 - Per ottenere dal sistema informazioni tipo data, spazio di memoria disponibile, spazio disco, numero di utenti abilitati
 - Per ottenere informazioni sulle statistiche di utilizzo del sistema di calcolo (prestazioni, logging, etc.) e sul debugging
 - Per effettuare operazioni di formattazione e stampa dei dati
 - Per ottenere informazioni sulla configurazione del sistema (registry)





Programmi di sistema – 4

❖ **Modifica di file**

- Editori di testo, per creare e modificare file
- Comandi speciali per cercare informazioni all'interno di file o effettuare trasformazioni sul testo

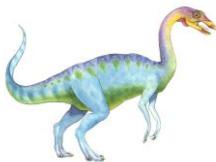
❖ **Supporto a linguaggi di programmazione** – assembler, compilatori e interpreti

❖ **Caricamento ed esecuzione di programmi** – linker, loader, debugger per linguaggio macchina e linguaggi di alto livello

❖ **Comunicazioni** – per creare connessioni virtuali tra processi, utenti e sistemi di elaborazione

- Permettono agli utenti lo scambio di messaggi video e via e-mail, la navigazione in Internet, il login remoto ed il trasferimento di file





Programmi di sistema – 5

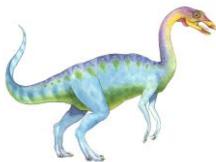
❖ Servizi background

- Lanciati durante la fase di boot
 - Taluni utili solo nella fase di startup del sistema
 - Altri in esecuzione dal boot allo shutdown
- Supportano servizi quali controllo del disco, scheduling dei processi, logging degli errori, stampa, connessioni di rete
- Vengono eseguiti in modalità utente
- Noti anche come **sottosistemi**, **daemon**

❖ Programmi applicativi

- Non fanno parte del sistema operativo
- Eseguiti dagli utenti
- Lanciati da linea di comando, dal click del mouse o da pressione sul touchscreen





Linker e loader – 1

- ❖ Codice sorgente compilato in file oggetto progettati per essere caricati in qualsiasi posizione di memoria fisica – **file oggetto rilocabili**
- ❖ Combinati dal **linker** in un singolo **file eseguibile binario** in cui vengono incluse anche le funzioni di libreria
- ❖ I programmi risiedono quindi nella memoria secondaria come eseguibili binari
- ❖ Devono essere caricati in memoria centrale dal **loader** per essere eseguiti
 - Durante la fase di rilocazione si assegnano gli indirizzi assoluti alle parti del programma e ai dati

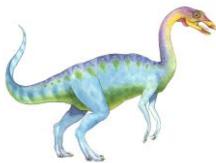




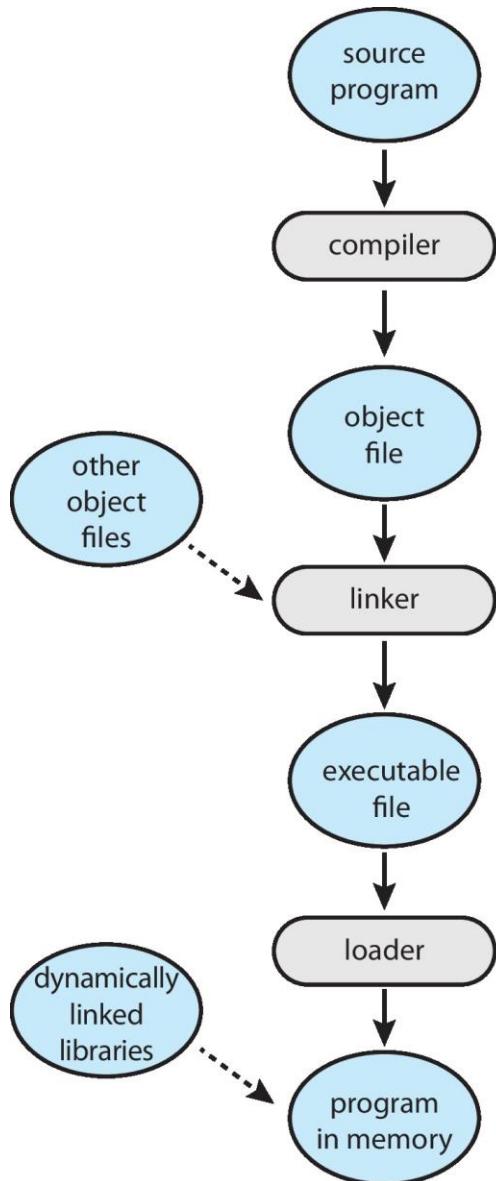
Linker e loader – 2

- ❖ I moderni sistemi general purpose non collegano le librerie ai file eseguibili
 - Piuttosto, le **librerie collegate dinamicamente** (e.g., le **DLL** di Windows) vengono caricate secondo necessità e condivise da tutti i programmi che utilizzano la stessa versione della stessa libreria (presente in memoria in un'unica copia)
- ❖ I file oggetto e eseguibili hanno formati standard, perché il sistema operativo sappia come caricarli e avviarli
- ❖ Nei sistemi UNIX-like, il formato standard è l'**ELF**, per *Executable and Linkable Format*
 - Contiene il punto di inizio del programma, cioè l'indirizzo della prima istruzione da eseguire
- ❖ I sistemi Windows utilizzano il formato PE (Portable Executable), mentre MacOS usa file Mach-O





Linker e loader – 3

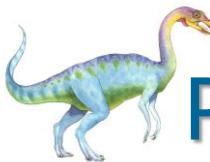


main.c
gcc -c main.c
generates
main.o
gcc -o main main.o -lm
generates
main
. /main

Opzione per linkare la
math library del C

Il file in formato ELF include
il codice macchina compilato
e una tabella dei simboli
contenente metadati relativi
a funzioni e variabili cui si fa
riferimento nel programma

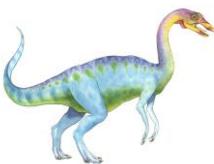




Perché le applicazioni sono SO-specific?

- ❖ Le applicazioni compilate su un sistema di solito non sono eseguibili su altri sistemi operativi
- ❖ Ogni sistema operativo offre le proprie chiamate di sistema uniche, i propri formati di file eseguibili, etc.
- ❖ Le applicazioni possono essere multi-SO se:
 - scritte in un linguaggio interpretato come Python, con interprete disponibile su più sistemi operativi
 - scritte in un linguaggio (come Java) che include una VM contenente l'app in esecuzione
 - scritte in un linguaggio standard (come C), ma compilate separatamente su ciascun SO
- ❖ **Application Binary Interface (ABI)** è l'equivalente di API, ma definisce in che modo i diversi componenti del codice binario possono interfacciarsi per un dato SO su una data architettura, CPU, etc.

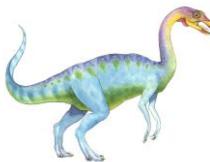




Riassumendo...

- ❖ I tipi di richieste di servizio al SO variano secondo il tipo di sistema ed il livello delle richieste stesse
- ❖ Il livello cui appartengono le chiamate di sistema deve offrire le funzioni di base (controllo di processi e memoria e gestione di file e dispositivi)
- ❖ Le richieste di livello superiore, soddisfatte dall'interprete dei comandi o dai programmi di sistema, vengono tradotte in una sequenza di chiamate al SO
- ❖ Oltre le categorie di richieste di servizio standard, gli errori nei programmi possono considerarsi richieste di servizio implicite





Progettazione e realizzazione di SO – 1

- ❖ Progettare il SO “perfetto” è un compito che non ammette soluzione, ma alcuni approcci implementativi si sono dimostrati comunque validi
- ❖ La struttura interna dei diversi SO può variare notevolmente...
 - ...in dipendenza dall'hardware
 - ...e dalle scelte progettuali che, a loro volta, dipendono dallo scopo del sistema operativo e ne influenzano i servizi offerti
- ❖ Richieste utente ed obiettivi del SO
 - **Richieste utente** – il SO deve essere di semplice utilizzo, facile da imparare, affidabile, sicuro e veloce
 - **Obiettivi del sistema** – il SO deve essere semplice da progettare, facile da realizzare e manutenere, flessibile, affidabile, error-free ed efficiente





Progettazione e realizzazione di SO – 2

- ❖ Per la progettazione e la realizzazione di un sistema operativo è fondamentale mantenere separati i due concetti di...
 - **Politiche:** Quali sono i compiti e i servizi che il SO dovrà svolgere/fornire? (Es.: scelta di un particolare algoritmo per lo scheduling della CPU)
 - **Meccanismi:** Come realizzarli? (Es.: timer)
- ❖ I meccanismi determinano “come fare qualcosa”, le politiche definiscono “cosa è necessario fare”
- ❖ **Esempio:** i problemi di assegnazione delle risorse impongono decisioni politiche, i meccanismi servono ad implementarne l’accesso controllato
 - La separazione fra politiche e meccanismi è fondamentale; si garantisce la massima flessibilità se le decisioni politiche subiscono cambiamenti nel corso del tempo





Progettazione e realizzazione di SO – 3

❖ Esempi

- **Windows e MacOS:** sia le politiche sia i meccanismi sono fissati a priori e cablati nel sistema; tutte le applicazioni hanno interfacce simili perché l'interfaccia stessa fa parte del kernel e delle librerie di sistema
- **UNIX/Linux:** Netta separazione fra meccanismi e politiche

❖ Progettare e realizzare un SO è un compito di **ingegneria del software** eminentemente creativo





Sviluppo software

- ❖ Tradizionalmente i SO venivano scritti in linguaggio **assembly**; successivamente, furono utilizzati linguaggi specifici per la programmazione di sistema, quali **Algol** e **PL/1**; attualmente vengono invece sviluppati in linguaggi di alto livello, particolarmente orientati al sistema: **C** o **C++**
- ❖ In realtà, normalmente, si utilizza un mix di linguaggi:
 - Componenti di basso livello sviluppate in **assembly** (es.: driver dei dispositivi)
 - Kernel in **C**
 - Programmi di sistema realizzati tramite, **C**, **C++**, e linguaggi di scripting, quali **PERL**, **Python**, **shell script**
- ❖ **Caratteristiche**
 - Veloci da codificare; codice compatto, di facile comprensione, messa a punto e manutenzione
 - Portabilità
 - Potenziale minor efficienza del codice **C** rispetto all'**assembly**



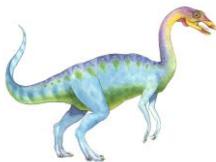


Struttura del sistema operativo

❖ Sistemi storici (e non solo): **monolitici**

- Le funzioni di gestione delle risorse sono tutte realizzate nel nucleo e l'intero sistema operativo tende a identificarsi col nucleo
- Anche se ogni funzione è separata dal resto, l'integrazione del codice è molto stretta e, poiché tutti i moduli operano nello stesso spazio di indirizzi, un bug in uno di essi può bloccare l'intero sistema
- Tuttavia, quando l'implementazione è sicura, la stretta integrazione interna dei componenti rende un buon kernel monolitico estremamente efficiente
- “Filosofia” simile a quando non si strutturano i programmi in sottoprogrammi e si inserisce tutto il codice nella funzione **main()**

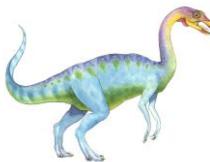




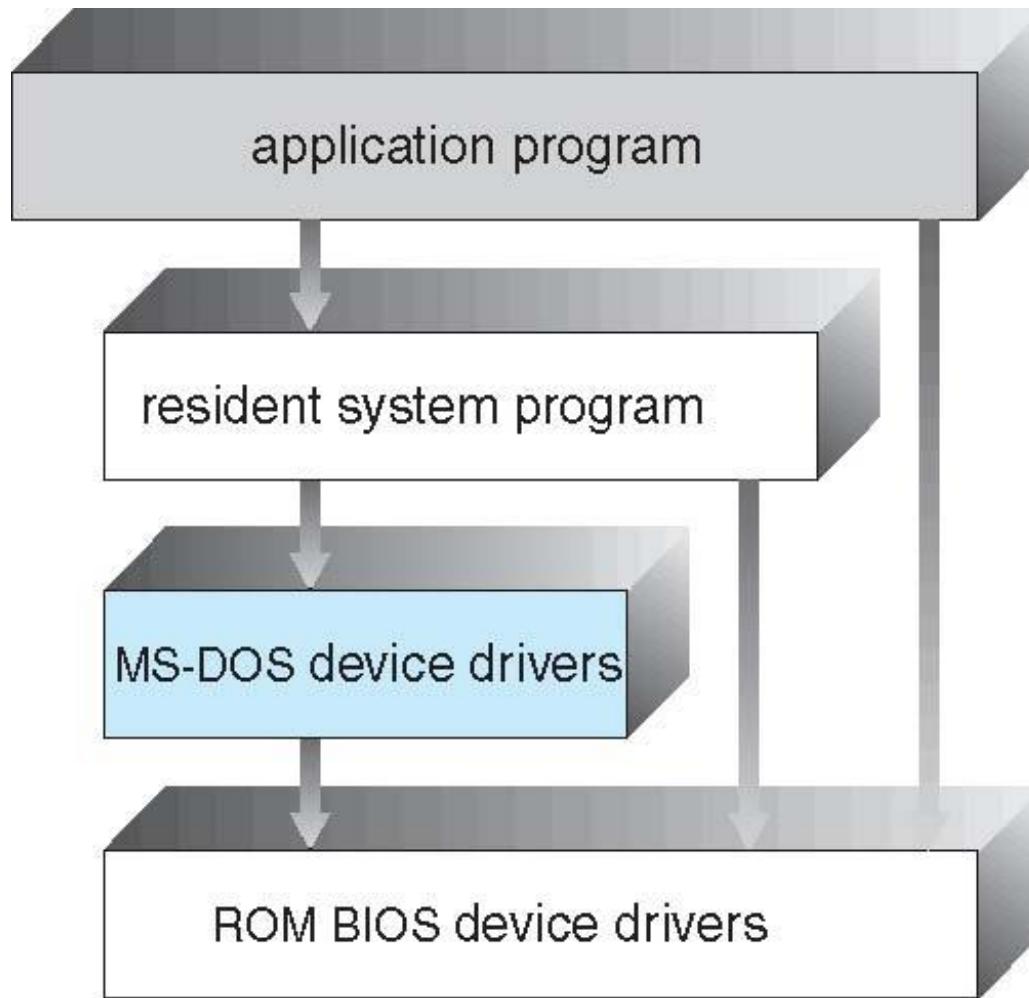
SO con struttura semplice

- ❖ **MS-DOS** — scritto per fornire il maggior numero di funzionalità utilizzando la minor quantità di spazio possibile
 - Non è suddiviso in moduli
 - Sebbene MS-DOS abbia una qualche struttura (a strati), le sue interfacce e livelli di funzionalità non sono ben separati
 - ▶ Le applicazioni accedono direttamente alle routine di sistema per l'I/O (ROM BIOS)
 - ▶ Vulnerabilità agli errori ed agli "attacchi" dei programmi utente
 - Intel 8088, per cui MS-DOS fu progettato, non offre duplice modo di funzionamento e protezione hardware
⇒ impossibile proteggere hardware/SO dai programmi utente





Struttura a strati di MS-DOS

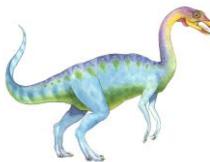




UNIX

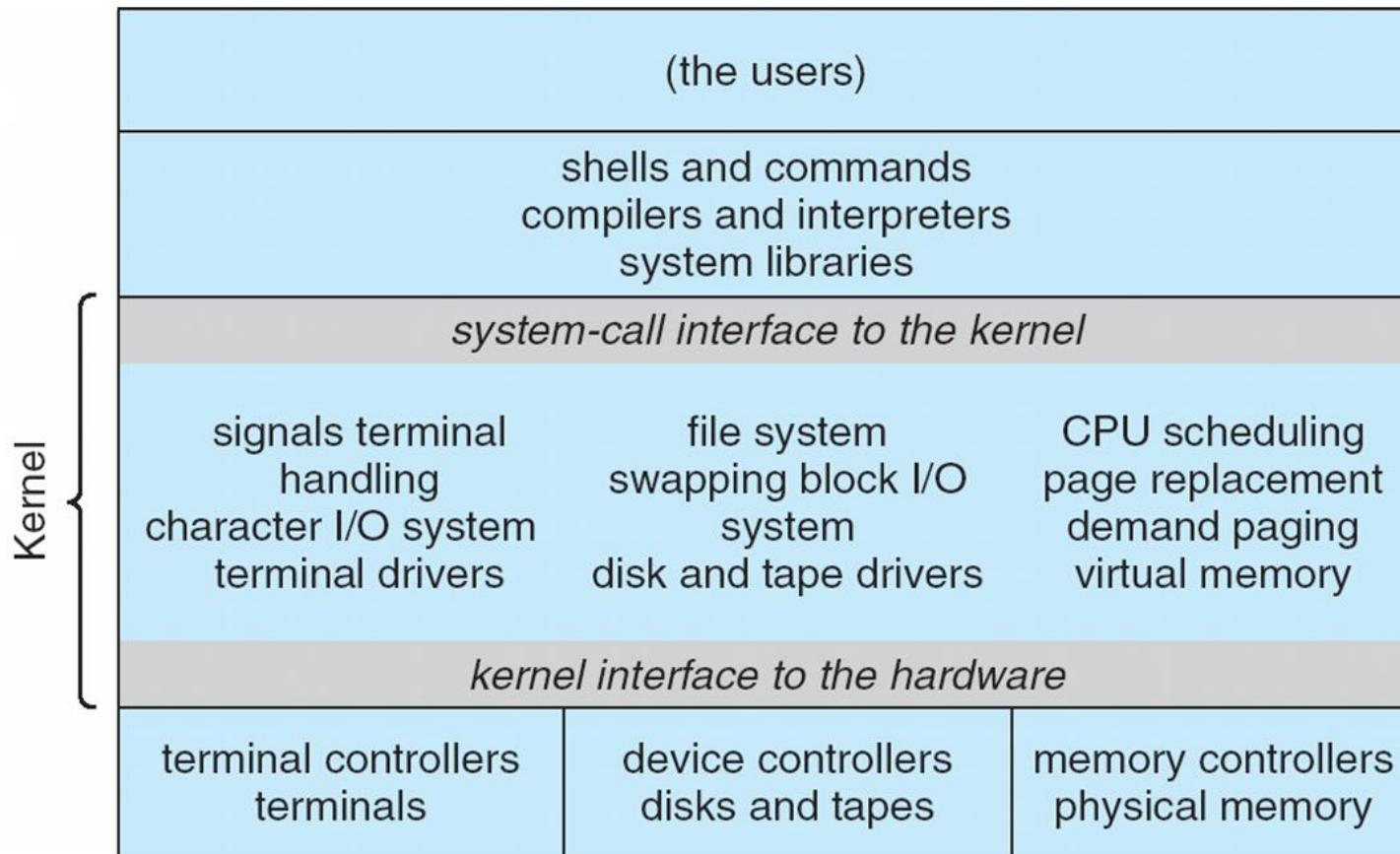
- ❖ **UNIX** — a causa delle limitate funzionalità hardware disponibili all'epoca della sua realizzazione, il sistema originale aveva una struttura scarsamente stratificata
- ❖ UNIX è costituito di due parti separate:
 - I programmi di sistema
 - Il **kernel**
 - ▶ È formato da tutto ciò che si trova sotto l'interfaccia delle chiamate di sistema e sopra l'hardware
 - ▶ Fornisce il file system, lo scheduling della CPU, la gestione della memoria ⇒ un gran numero di funzioni per un solo livello!





Struttura a strati di UNIX

Beyond simple but not fully layered



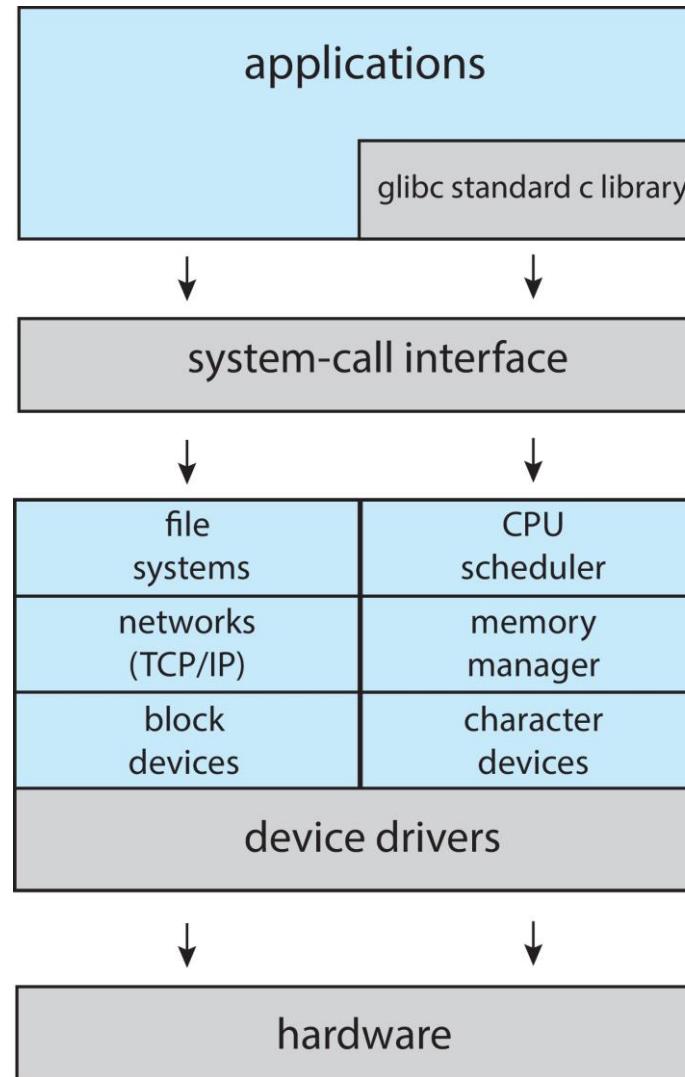


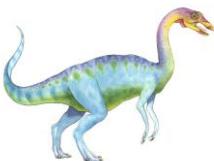
Struttura a strati di LINUX

glibc costituisce la API

Il kernel di Linux è monolitico, in quanto tutte le funzioni del SO vengono eseguite in modalità kernel in un unico spazio di indirizzi, ma è dotato anche di struttura modulare, dato che nuovi moduli possono essere caricati a runtime

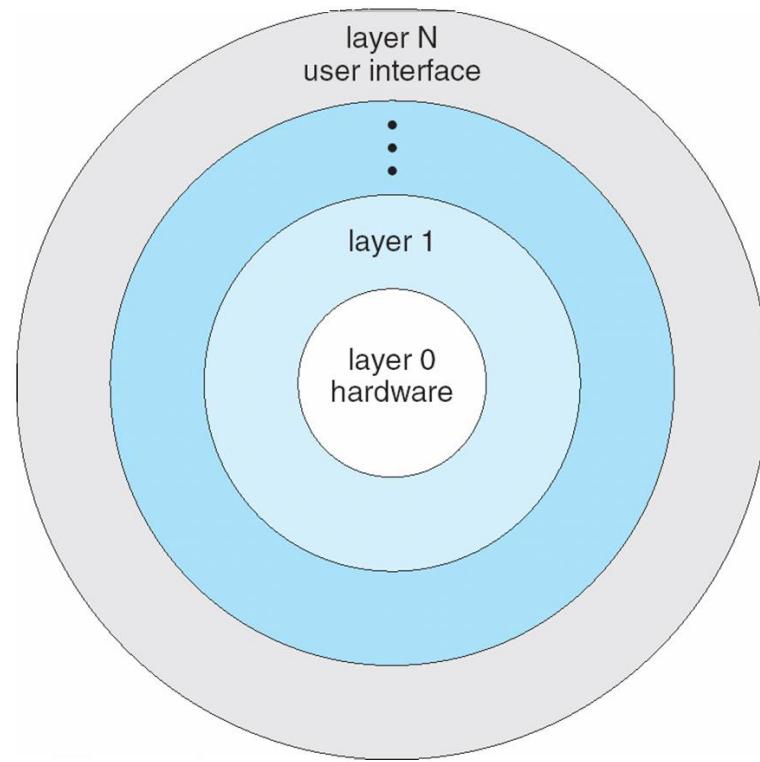
Monolithic plus modular design

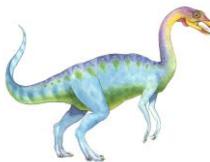




Approccio stratificato – 1

- ❖ In presenza di hardware appropriato, i SO possono assumere architettura stratificata, per meglio garantire il controllo sulle applicazioni
- ❖ Il SO è suddiviso in un certo numero di strati (livelli), ciascuno costruito sopra gli strati inferiori
 - Il livello più basso (strato 0) è l'hardware, il più alto (strato N) è l'interfaccia utente
- ❖ L'architettura degli strati è tale che ciascuno strato impiega esclusivamente funzioni (operazioni) e servizi di strati di livello inferiore (usati come black–box)
 - Ogni strato è un “oggetto astratto”, che incapsula i dati e le operazioni che trattano tali dati





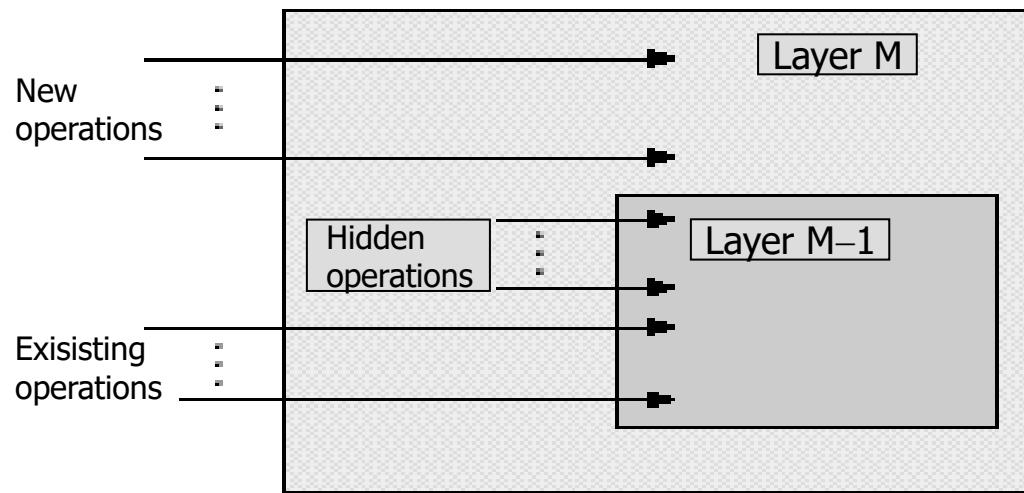
Approccio stratificato – 2

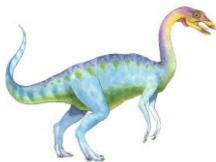
❖ Vantaggi

- Semplicità di realizzazione e messa a punto (che viene attuata strato per strato)

❖ Svantaggi

- Difficoltà nella definizione appropriata dei diversi strati, poiché ogni strato può sfruttare esclusivamente le funzionalità degli strati su cui poggia
- Tempi lunghi di attraversamento degli strati (passaggio di dati) per portare a termine l'esecuzione di una syscall





Approccio stratificato – 3

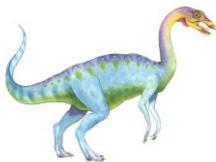
❖ Esempio 1 – Difficoltà di definizione degli strati

- Il driver della memoria virtuale (*backing store*) dovrebbe trovarsi sopra lo scheduler della CPU, perché può accadere che il driver debba attendere un'istruzione di I/O e, in questo periodo, la CPU viene sottoposta a scheduling
- Lo scheduler della CPU deve mantenere più informazioni sui processi attivi di quante ne possono essere contenute in memoria: deve fare uso del driver della memoria ausiliaria

❖ Esempio 2 – Scarsa efficienza del SO

- Per eseguire un'operazione di I/O, un programma utente invoca una system call che è intercettata dallo strato di I/O...
- ...che esegue una chiamata allo strato di gestione della memoria...
- ...che richiama lo strato di scheduling della CPU...
- ...che la passa all'opportuno dispositivo di I/O

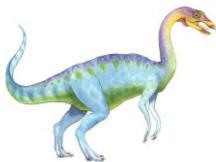




Approccio stratificato – 4

- ❖ I sistemi stratificati sono stati utilizzati con successo nelle reti (TCP/IP) e nelle applicazioni Web
- ❖ Pochi SO utilizzano un approccio a strati puro
- ❖ Tuttavia, una parziale stratificazione della struttura è comune nei SO contemporanei (pochi strati multi-funzionali, internamente modulari)
- ❖ Linux è “stratificato” poiché si effettuano diversi livelli (6–10) di chiamate di sistema prima di svolgere concretamente un compito





Struttura dei sistemi microkernel – 1

- ❖ Quasi tutte le funzionalità del kernel sono spostate nello spazio utente
- ❖ Un **microkernel** offre i servizi minimi di gestione dei processi, gestione della memoria e di comunicazione
 - Scopo principale: fornire funzioni di comunicazione fra programmi client e servizi (implementati esternamente)
 - Le comunicazioni hanno luogo tra moduli utente mediante scambio di messaggi (mediati dal kernel)
 - **Esempi:** prime versioni di Windows NT, Mach, GNU Hurd; Mac OS, con kernel Darwin, parzialmente basato su Mach





Struttura dei sistemi microkernel – 2

❖ Vantaggi

- Funzionalità del sistema più semplici da estendere: i nuovi servizi sono programmi di sistema che si eseguono nello spazio utente e non comportano modifiche al kernel
- Facilità di modifica del kernel
- Sistema più facile da portare su nuove architetture
- Più sicuro e affidabile (meno codice viene eseguito in modo kernel)

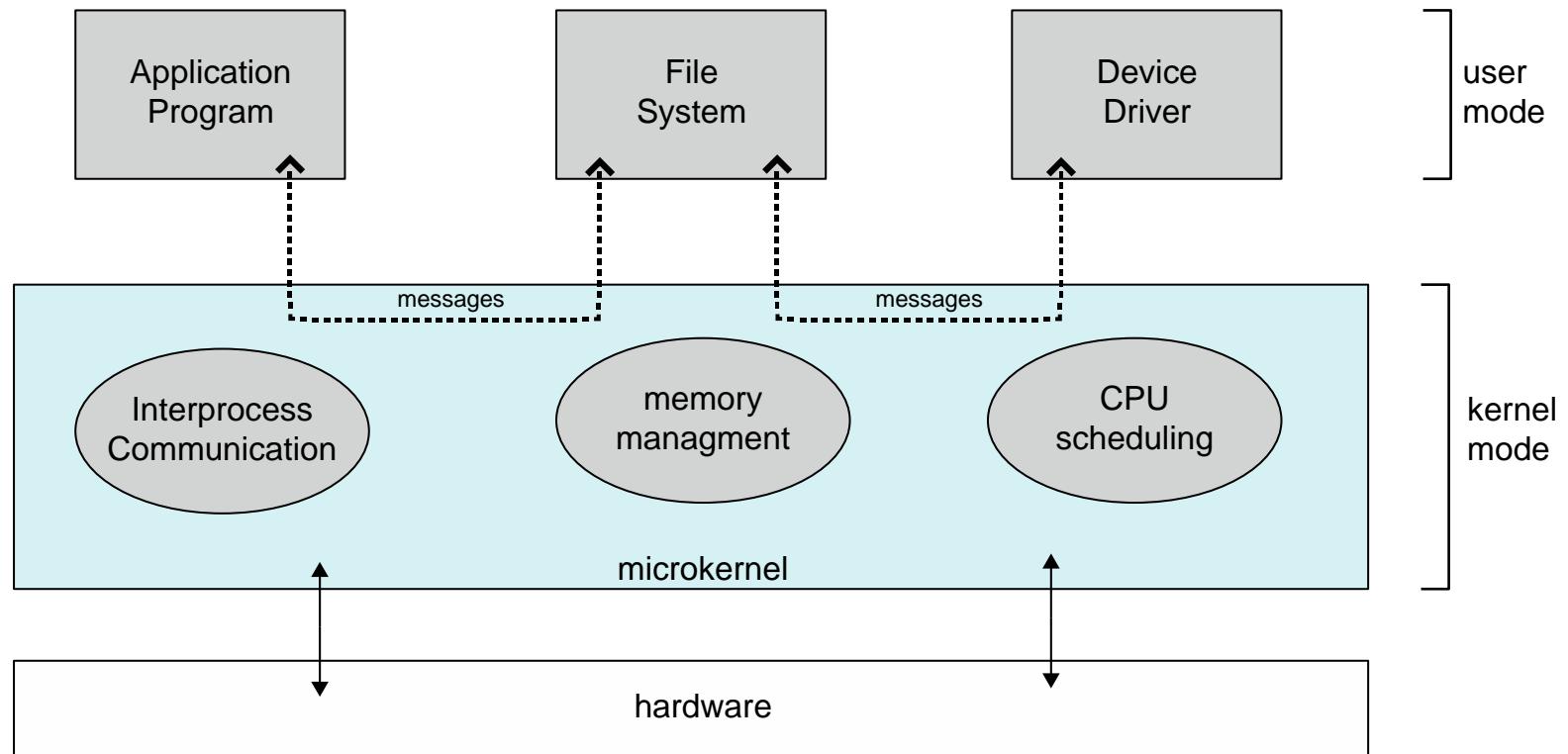
❖ Svantaggi

- Possibile decadimento delle prestazioni a causa dell'overhead di comunicazione fra spazio utente e spazio kernel





Struttura dei sistemi microkernel – 3

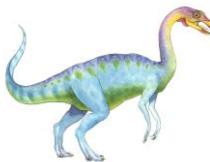




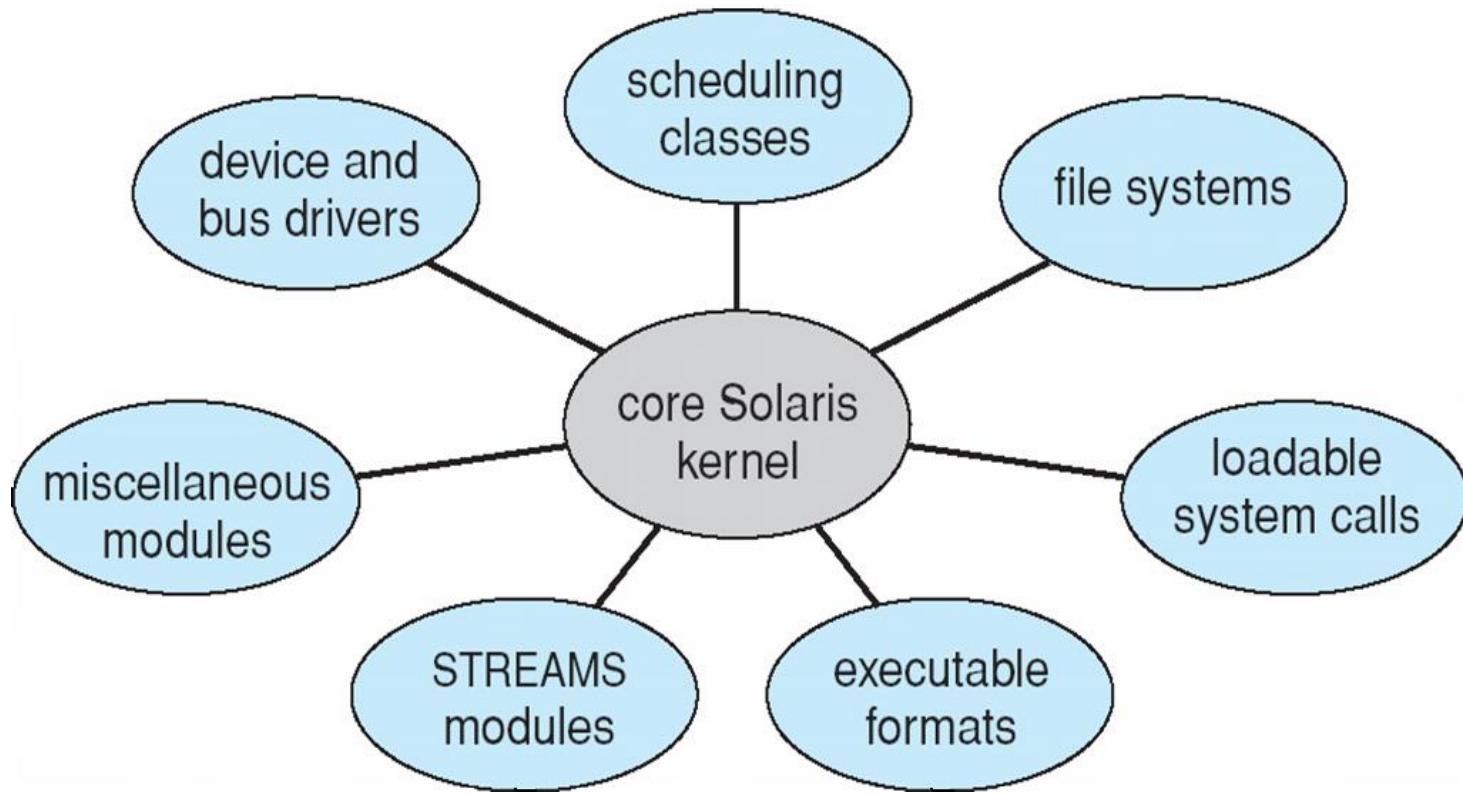
Kernel modulari

- ❖ Molti degli attuali SO implementano **kernel modulari** (**Loadable Kernel Modules, LKMs**)
 - Si utilizza un approccio alla programmazione object-oriented
 - Ciascun modulo implementa una componente base del kernel, con interfacce e funzioni definite con precisione
 - Ciascun modulo colloquia con gli altri mediante l'interfaccia comune
 - Ciascun modulo può essere o meno caricato in memoria come parte del kernel, secondo le esigenze (caricamento dinamico dei moduli, all'avvio o a run-time)
- ❖ L'architettura a moduli è simile all'architettura a strati, ma garantisce SO più flessibili (ogni modulo può invocare funzionalità da qualsiasi altro modulo): più facili da manutenere ed evolvere
 - Linux (periferiche, file system), Solaris





Approccio modulare di Solaris



- ❖ L'organizzazione modulare lascia la possibilità al kernel di fornire i servizi essenziali, ma permette anche di implementare dinamicamente servizi aggiuntivi, specifici per il particolare sistema di calcolo

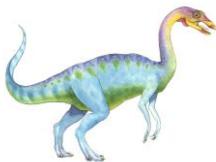




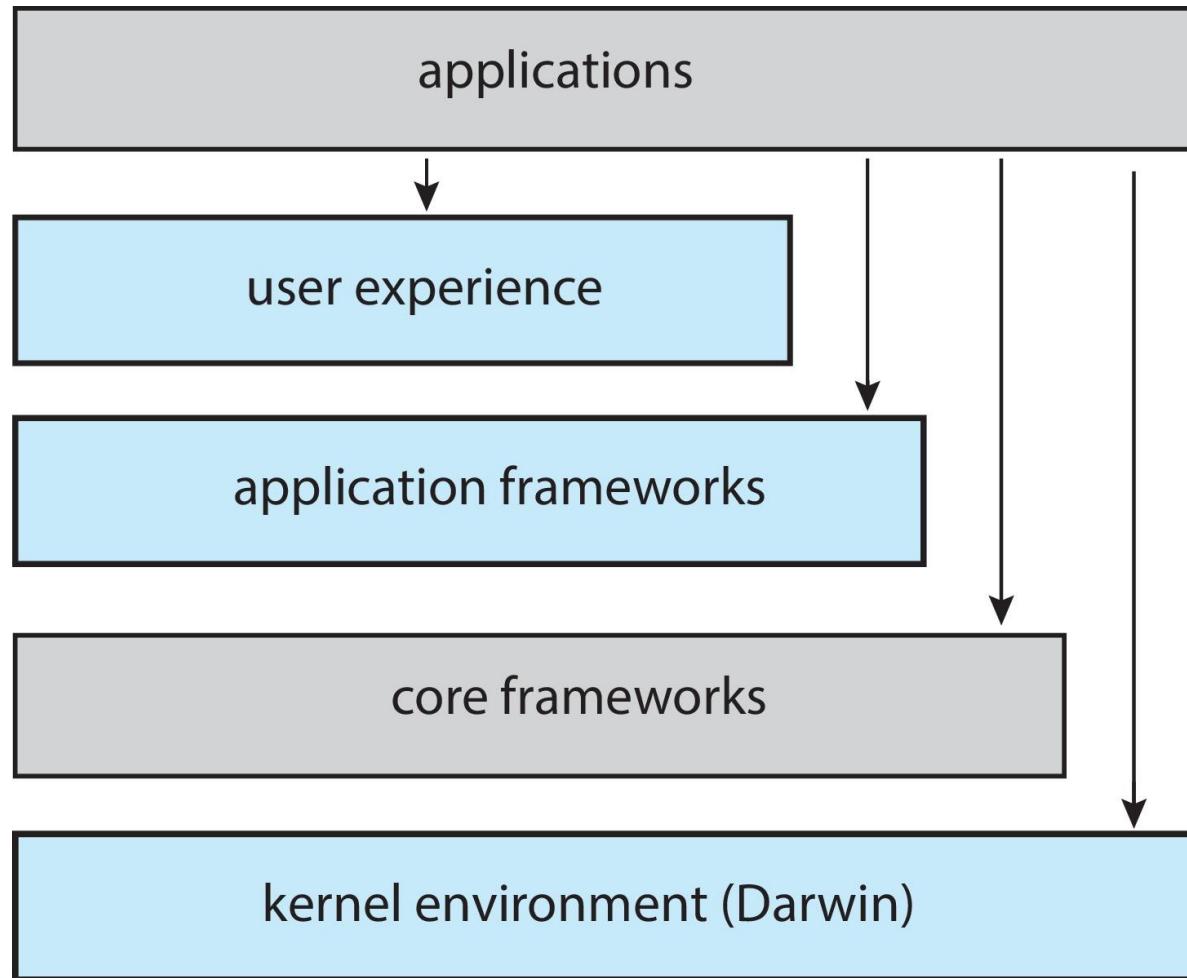
Sistemi ibridi

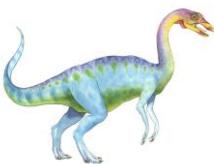
- ❖ La maggior parte dei SO attuali non adotta un modello “puro”
 - I modelli ibridi combinano diversi approcci implementativi allo scopo di migliorare le performance, la sicurezza e l’usabilità
 - I kernel di Linux e Solaris sono fondamentalmente monolitici, perché mantenere il SO in un unico spazio di indirizzamento garantisce prestazioni migliori; sono però anche modulari, per cui le nuove funzionalità possono essere aggiunte dinamicamente al kernel
 - Windows è perlopiù monolitico, ma conserva alcuni comportamenti tipici dei sistemi microkernel, tra cui il supporto per sottosistemi separati (detti **personalità**) che vengono eseguiti come processi in modalità utente





Strati in Mac OS e iOS





Mac OS e iOS – 1

- ❖ Anche se progettati per hardware diversi, Mac OS e iOS hanno architettura simile
 - Strato dell'interfaccia utente (*user experience*): **Aqua** per Mac OS, progettata per interazione con mouse e trackpad, **Springboard** per iOS, per touch screen
 - Strato degli ambienti applicativi: **Cocoa**, per Mac OS, **Cocoa Touch**, per iOS, forniscono un'API per i linguaggi di programmazione Objective-C e Swift
 - Ambienti di base (core): supportano grafica e contenuti multimediali, inclusi Quicktime e OpenGL
 - Ambiente kernel: **Darwin**, include il microkernel Mach e il kernel BSD UNIX
- ❖ Le applicazioni possono essere progettate per sfruttare le funzionalità di *user experience* o dell'*application framework* o per aggirarle completamente





Mac OS e iOS – 2

❖ Differenze significative:

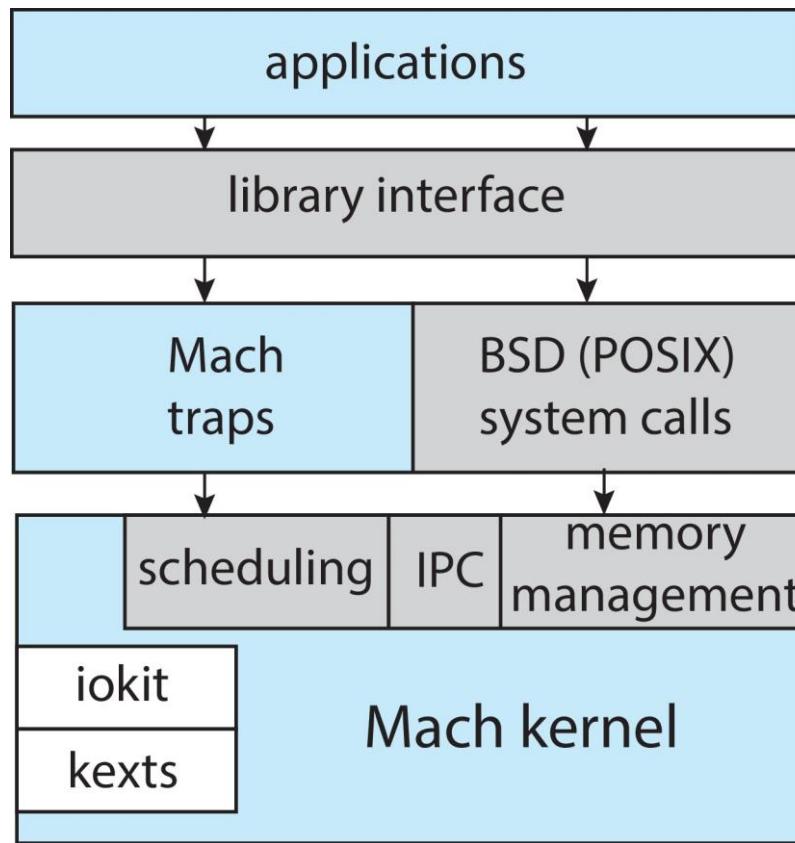
- Poiché Mac OS è destinato a laptop e desktop, è compilato per architetture M1/M2 (o Intel); iOS invece è compilato su ARM
 - Il kernel di iOS garantisce risparmio energetico e gestione ottimizzata della memoria
 - Impostazioni di sicurezza più severe
- Il sistema iOS è molto più restrittivo nei confronti degli sviluppatori; limita l'accesso alle API POSIX e BSD, che sono liberamente disponibili agli sviluppatori di Mac OS





Il kernel Darwin

- ❖ Il microkernel Mach gestisce la memoria, la comunicazione fra processi (IPC), le chiamate di procedura remota (RPC) e lo scheduling dei thread
- ❖ Il kernel BSD mette a disposizione una CLI, i servizi legati al file system e alla rete e la API POSIX





Struttura di Mac OS

- ❖ Il SO Apple Mac OS adotta una struttura ibrida (stratificata)
 - Gli strati superiori comprendono l'interfaccia utente **Aqua** e una collezione di ambienti di sviluppo e servizi per le applicazioni; l'ambiente **Cocoa** viene utilizzato per la scrittura di applicazioni native in Objective-C
 - Il kernel si trova in uno strato sottostante e, oltre a al microkernel Mach e al kernel BSD, include un kit di strumenti per lo sviluppo di driver di I/O e moduli caricabili dinamicamente, detti **estensioni del kernel (kext)**





Struttura di iOS

- ❖ SO progettato da Apple per i dispositivi mobili, *iPhone* e *iPad*

- Strutturato sul Mac OS, con l'aggiunta di funzionalità specifiche per il mobile
- Non esegue direttamente le applicazioni native di Mac OS perché “gira” su processori diversi (ARM vs. M1/M2/Intel)
- **Cocoa Touch** è un Objective-C API per lo sviluppo di app
- I **media service** costituiscono un layer per le applicazioni multimediali (grafica, audio, video)
- I **core service** forniscono supporto al cloud computing e ai database
- Il **core** è basato sul kernel di Mac OS (Darwin)

Cocoa Touch

Media Services

Core Services

Core OS





Struttura di Android – 1

- ❖ Sviluppato dalla Open Handset Alliance – guidata da Google e costituita da oltre 30 compagnie, tra cui ASUS, HTC, Intel, Motorola, Qualcomm, T-Mobile, Samsung e NVIDIA – gestisce una grande varietà di piattaforme mobile ed è open-source
- ❖ Costituito da una “pila” di strati software (come iOS)
- ❖ Basato su un kernel Linux modificato (al di fuori delle distribuzioni standard)
 - Gestione dei processi, della memoria, delle periferiche
 - Ampliato per includere l’ottimizzazione dei consumi energetici

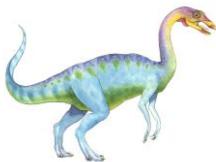




Struttura di Android – 2

- ❖ L'ambiente di run-time include un insieme di librerie di base e la macchina virtuale **ART** (**Android Run Time**)
 - App sviluppate in Java con il supporto dell'Android API
 - ▶ Class file di Java compilati in bytecode e quindi tradotti in eseguibili per la ART virtual machine
 - Compilazione anticipata, per migliorare l'efficienza delle applicazioni
- ❖ Disponibile anche l'interfaccia Java nativa, **JNI**, per bypassare ART, e scrivere programmi Java con accesso più diretto all'hardware

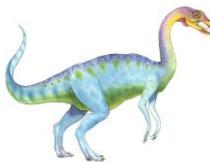




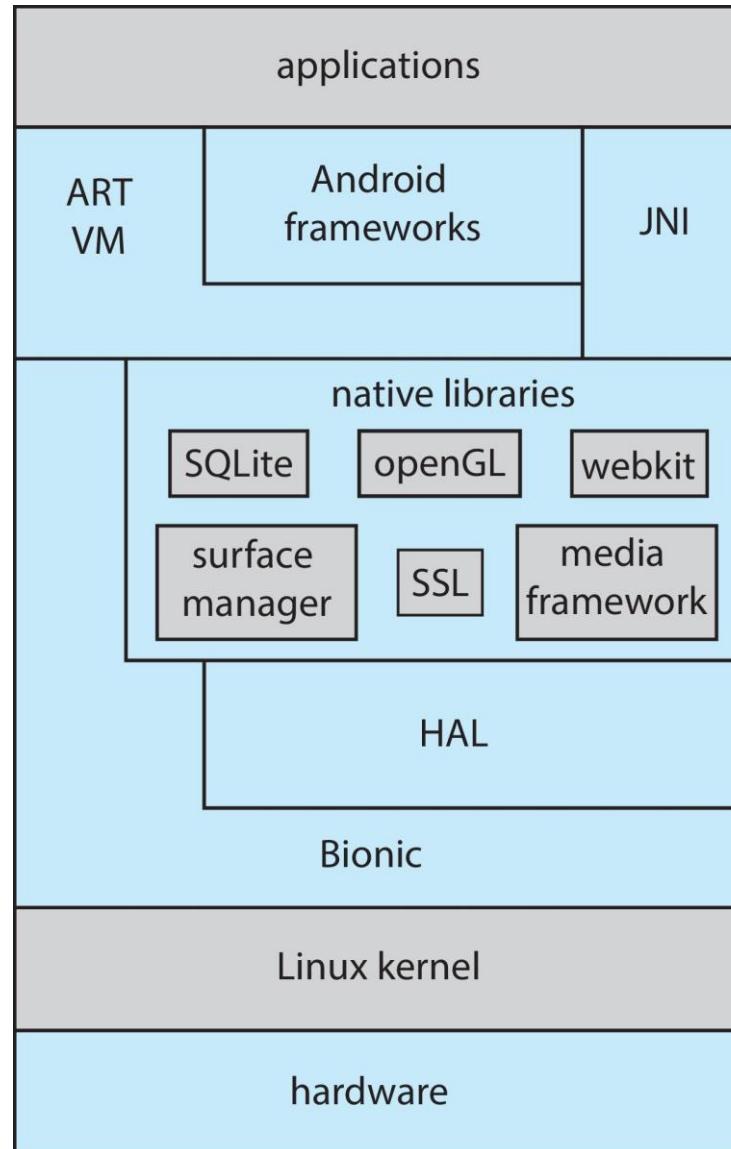
Struttura di Android – 3

- ❖ Le librerie includono ambienti per lo sviluppo di browser (**webkit**), di supporto ai database (**SQLite**) e all'accesso alla rete (**SSL**), ambienti multimediali e una libreria C standard minimale (**Bionic**, libera da GPL)
- ❖ Poiché Android può essere eseguito su un numero quasi illimitato di dispositivi, include uno strato di astrazione dell'hardware (**HAL** – Hardware Abstraction Layer)
- ❖ Kernel Linux modificato: gestione e allocazione di memoria, IPC (fornita da **Binder**)





Struttura di Android – 4





Generazione e avvio del SO – 1

- ❖ I sistemi operativi sono generalmente progettati per funzionare su una classe di architetture, che possono disporre di una grande varietà di periferiche
- ❖ Comunemente, il sistema operativo è già installato quando si acquista un computer
 - Nulla vieta tuttavia che se ne possano (*realizzare e*) installare altri
 - Se si genera un sistema operativo *from scratch*, occorre (*scrivere il codice sorgente del sistema operativo*)
 - ▶ Configurare il sistema operativo per il sistema di calcolo su cui verrà eseguito
 - ▶ Compilare il sistema operativo
 - ▶ Installare il sistema operativo
 - ▶ Avviare il computer con il nuovo SO





Generazione e avvio del SO – 2

- ❖ In particolare, per generare un SO è necessario...
 - leggere da un file le informazioni riguardanti la configurazione specifica del sistema o, alternativamente...
 - esplorare il sistema di calcolo per determinarne i componenti
- ❖ Informazioni necessarie:
 - Tipo di CPU impiegate e opzioni installate
 - Tipo di formattazione del dispositivo di memoria di avvio (es. numero di partizioni)
 - Quantità di memoria disponibile
 - Dispositivi disponibili
 - Scelta delle politiche (algoritmi di scheduling, numero massimo di processi sostenibili, etc.)
- ❖ Generare il sistema serve per:
 - ottenere un kernel system-specific
 - garantire un sistema più efficiente perché customizzato





Generazione e avvio di Linux

- ❖ Accedere a <http://www.kernel.org> per scaricare il codice sorgente di Linux
- ❖ Configurare il kernel tramite **make menuconfig**; si genera il file di configurazione **.config**
- ❖ Compilare il kernel usando **make** (che utilizza i parametri di configurazione presenti in **.config**)
 - Si produce **vmlinuz**, l'immagine del kernel
- ❖ Compilare i moduli del kernel tramite **make modules** (che utilizza i parametri di configurazione presenti in **.config**)
- ❖ Installare i moduli del kernel in **vmlinuz** tramite **make modules_install**
- ❖ Installare il nuovo kernel sul sistema tramite **make install**

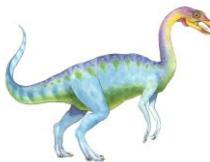




Avvio del sistema operativo – 1

- ❖ Quando si accende un computer, l'esecuzione inizia da una posizione di memoria prefissata
- ❖ Il sistema operativo deve essere reso disponibile all'hardware affinché l'hardware possa avviarlo
 - Piccolo pezzo di codice – **bootstrap loader** memorizzato in ROM o nella EEPROM, individua il kernel, lo carica in memoria e lo avvia
 - A volte il bootstrap avviene in due passi: si carica il **bootloader**, contenuto nel **blocco di avvio**, che provvede a caricare il kernel
- ❖ I sistemi moderni sostituiscono il BIOS con l'interfaccia **UEFI (Unified Extensible Firmware Interface)**
- ❖ I comuni bootstrap, come **GRUB** di Linux, permettono la selezione del kernel in varie versioni, con differenti opzioni e da memorie diverse





Avvio del sistema operativo – 2

❖ In Linux:

- Per risparmiare spazio e ridurre il tempo di avvio, l'immagine del kernel è un file compresso, che viene estratto in RAM
- Il bootloader crea un file system temporaneo in RAM, chiamato **initramfs**: contiene driver e moduli del kernel
- Dopo l'avvio del kernel e l'installazione dei driver necessari, viene montato il file system principale su memoria di massa (nella posizione appropriata)
- Si crea il processo iniziale di sistema **systemd**
- Si presenta il prompt di login

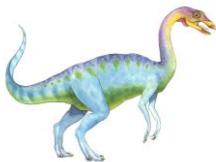




Debugging del sistema operativo – 1

- ❖ Il **debugging** è l'attività di individuazione e risoluzione di errori nel sistema, i cosiddetti **bachi** (*bug*); comprende anche il **performance tuning**
- ❖ Il SO può generare **file di log** che danno informazioni sugli errori rilevati durante l'esecuzione di un processo
- ❖ Il SO può anche acquisire e memorizzare in un file un'immagine del contenuto della memoria utilizzata dal processo, chiamata **core dump**
- ❖ Il core dump, un'istantanea post-mortem dello stato del processo al momento della terminazione anomala, può essere passato a un programma di supporto, un debugger, in grado di rilevare l'errore





Debugging del sistema operativo – 2

❖ Un guasto nel kernel viene chiamato **crash**

- Come avviene per i processi utente, l'informazione riguardante l'errore viene salvata in un file di log, mentre lo stato della memoria viene salvato in un'immagine su memoria di massa (**crash dump**)
- Tecniche più complesse per la natura delle attività svolte dal kernel
 - ▶ Il salvataggio del crash dump su file potrebbe essere rischioso se il kernel è in stato inconsistente (es.: malfunzionamento del codice kernel relativo allo stesso file system)
 - ▶ Il dump viene salvato in un'area di memoria dedicata e da lì recuperato per non rischiare di compromettere il file system





Debugging del sistema operativo – 3

❖ Legge di Kernigham

“Il debugging è due volte più difficile rispetto alla stesura del codice. Di conseguenza, chi scrive il codice nella maniera più intelligente possibile non è, per definizione, abbastanza intelligente per eseguire il debugging.”

- ❖ Anche i problemi che condizionano le prestazioni sono considerati bachi, quindi il debugging include anche il *performance tuning*, che ha lo scopo di eliminare i colli di bottiglia del sistema di calcolo





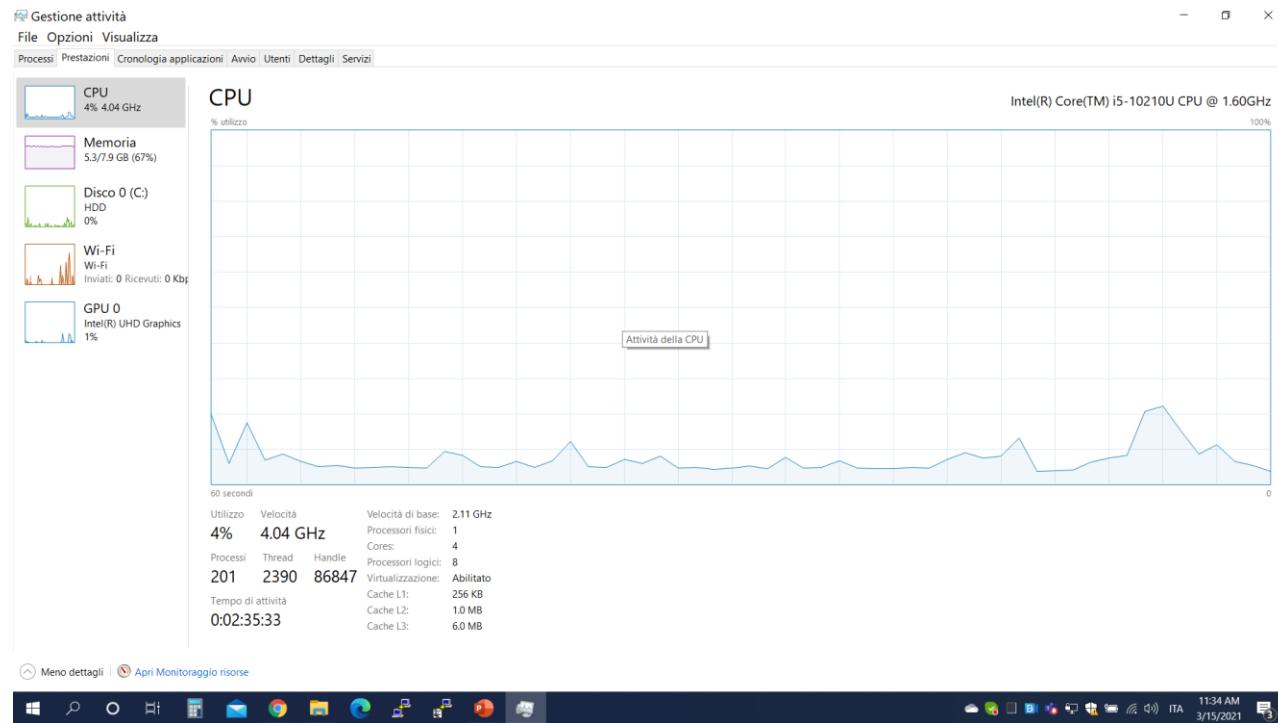
Performance tuning – 1

- ❖ Il **performance tuning** è l'insieme delle tecniche atte ad ottimizzare le prestazioni del sistema, eliminando i colli di bottiglia
 - Esecuzione di codice che effettui misurazioni sul comportamento del sistema e salvi i dati su un file di log
 - ▶ Analisi dei dati salvati nel file di log, che descrive tutti gli eventi di rilievo, per identificare ostacoli ed inefficienze
 - Introduzione, all'interno del SO, di strumenti interattivi che permettano ad amministratore ed utenti di monitorare il sistema (es., istruzione **top** di UNIX: mostra le risorse di sistema impiegate ed un elenco ordinato dei principali processi che le utilizzano)
 - **Profiling**, campionamento periodico dell'IP (Instruction Pointer) per valutare, per esempio, quali chiamate di sistema siano utilizzate maggiormente





Performance tuning – 2



- ❖ In Windows 10, selezionando la voce Gestione attività (dal menù ottenuto posizionandosi sulla barra delle applicazioni e premendo il tasto destro) si può visualizzare, in tempo reale, l'utilizzo delle risorse principali





Linux BCC – 1

- ❖ Il debugging a livello di interazioni tra codice utente e codice kernel è quasi impossibile senza un set di strumenti integrati al SO
- ❖ **BCC** (BPF Compiler Collection) è un ricco toolkit che fornisce funzionalità di tracciamento per Linux
- ❖ Ad esempio, **disksnoop.py** traccia l'attività di I/O su disco

TIME(s)	T	BYTES	LAT(ms)
1946.29186700	R	8	0.27
1946.33965000	R	8	0.26
1948.34585000	W	8192	0.96
1950.43251000	R	4096	0.56
1951.74121000	R	4096	0.35

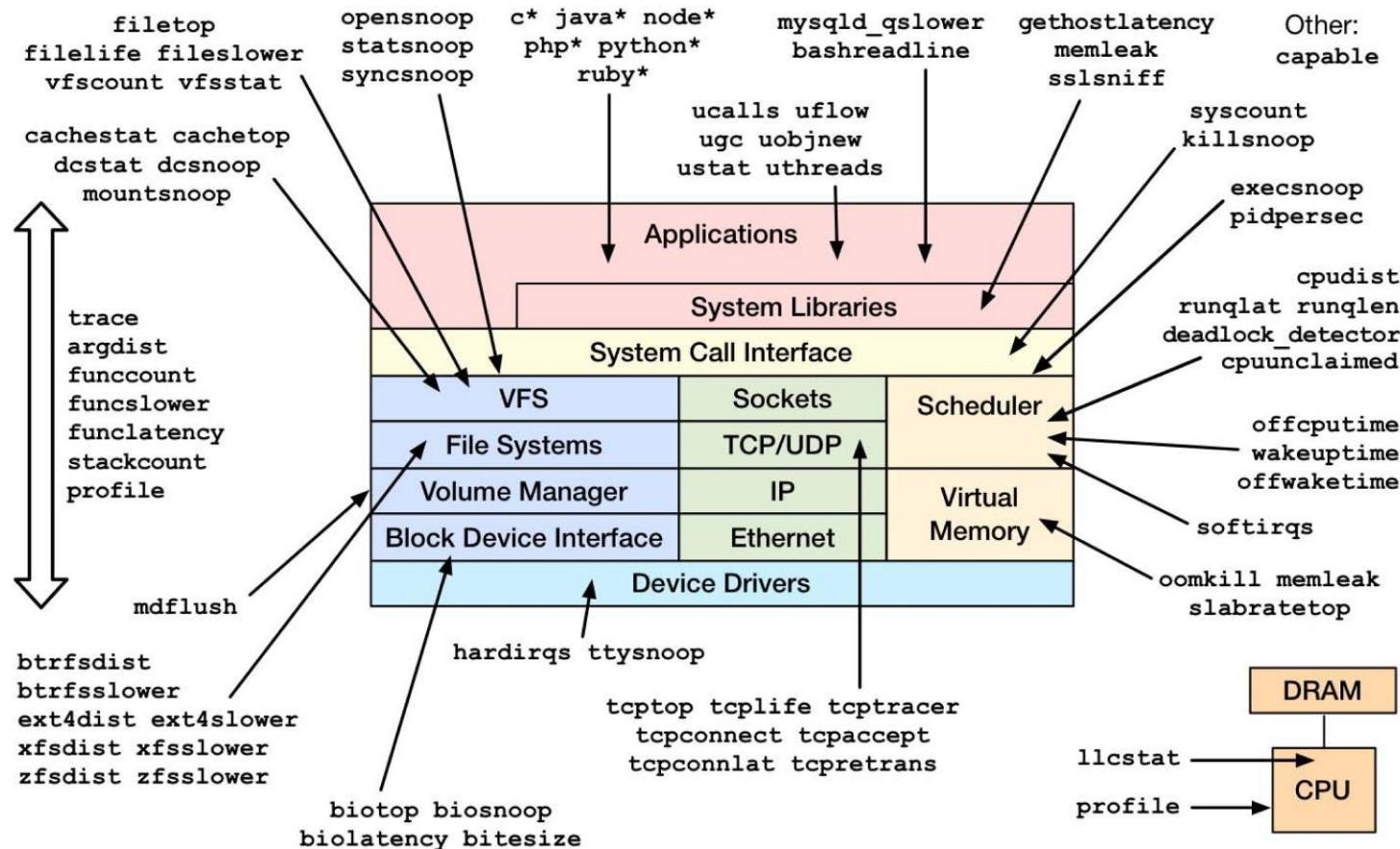
- ❖ BCC include strumenti di monitoraggio di tutti i moduli del sistema





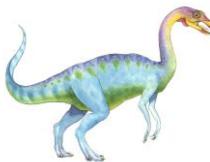
Linux BCC – 2

Linux bcc/BPF Tracing Tools



<https://github.com/iovisor/bcc#tools> 2017

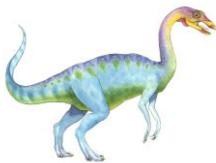




Il sottosistema Windows per Linux – 1

- ❖ Windows possiede un'architettura ibrida che fornisce la possibilità di includere sottosistemi, in particolare capaci di emulare diversi SO
 - I sottosistemi vengono eseguiti in modalità utente e comunicano con il kernel di Windows per fornire i servizi
 - Il sottosistema di Windows per Linux si chiama **WSL** e consente alle applicazioni Linux native (specificate come binari ELF) di essere eseguite su Windows 10
 - Occorre avviare l'applicazione Windows **bash.exe**, che presenta all'utente una shell **bash** che esegue Linux
 - Internamente, WSL crea un'istanza Linux costituita dal processo **init**, che a sua volta crea la shell **bash** che esegue le applicazioni Linux native

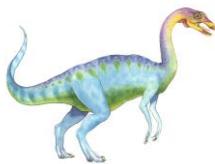




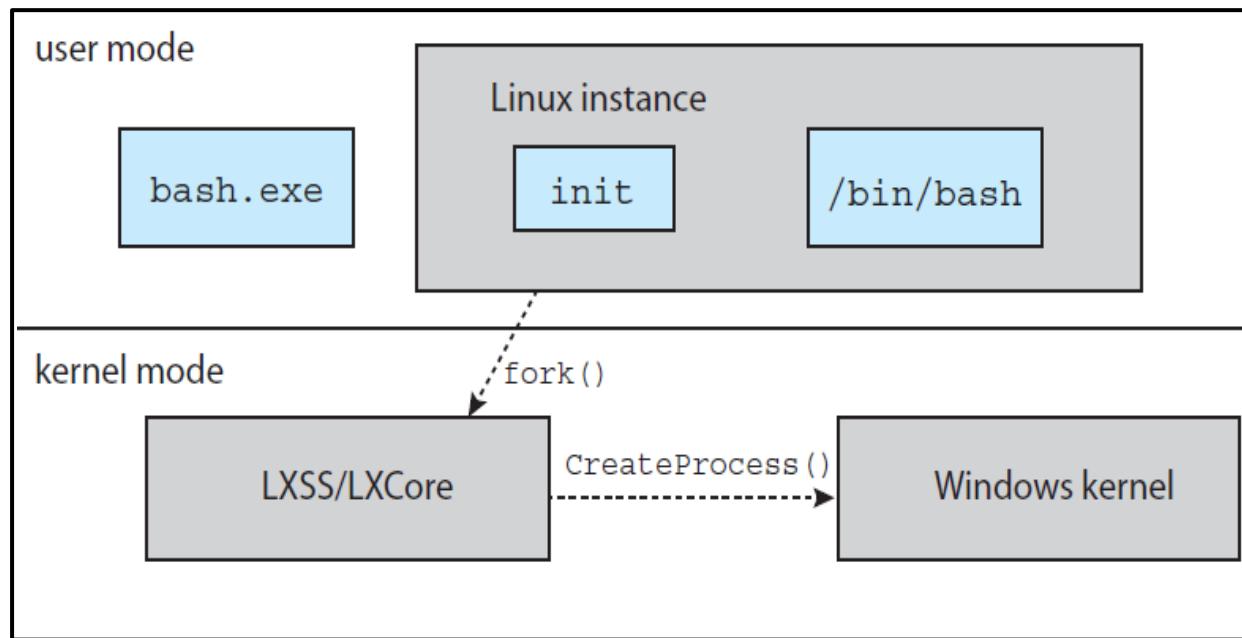
Il sottosistema Windows per Linux – 2

- ❖ Ognuno di questi processi viene eseguito in un processo **Pico** di Windows, che carica il file binario di Linux nel proprio spazio degli indirizzi
 - I processi Pico comunicano con i servizi del kernel **LXCore** e **LXSS**, per tradurre le chiamate di sistema di Linux nelle chiamate di sistema di Windows, quando possibile
 - In mancanza di corrispondenza diretta, LXSS deve fornire una funzionalità equivalente, eventualmente gestendola in proprio
 - **Esempio:** La chiamata di sistema **fork()** di Linux non corrisponde esattamente alla **CreateProcess()** di Windows; LXSS esegue parte del lavoro di **fork()** e, solo successivamente, invoca **CreateProcess()**





Il sottosistema Windows per Linux – 3



Fine del Capitolo 2

