

# Sistemi operativi: struttura e servizi

Pietro Braione

Reti e Sistemi Operativi – Anno accademico 2022-2023

# Argomenti

- Cosa sono i sistemi operativi?
- Requisiti per i sistemi operativi
- Struttura e servizi dei sistemi operativi
- Chiamate di sistema ed API
- Implementazione di chiamate di sistema ed API
- I programmi di sistema
- Struttura del kernel

Cosa sono i sistemi operativi?

# Un sistema operativo...

È il primo programma che viene eseguito quando viene acceso il computer

Ci permette di eseguire tanti programmi contemporaneamente, e di far scambiare informazioni tra di loro

Ci permette di gestire il computer, ed in particolare di installare e mandare in esecuzione i programmi veramente utili (applicazioni)

Fornisce un ambiente omogeneo, ed un insieme di «regole» (ad esempio, regole grafiche) alle quali chi sviluppa applicazioni deve attenersi perché le applicazioni si integrino in questo ambiente

Mantiene ed organizza i nostri dati sotto forma di file e cartelle

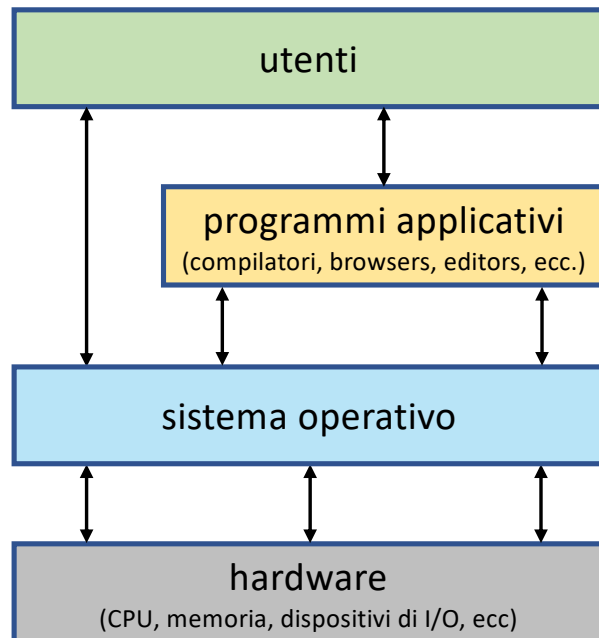
# Un sistema operativo...

- È un insieme di programmi (software) che gestisce gli elementi fisici di un computer (hardware)
- Fornisce una piattaforma di sviluppo per i programmi applicativi che permette loro di **condividere** ed **astrarre** le risorse hardware
- Agisce da intermediario tra utenti e computer, permettendo agli utenti di **controllare** l'esecuzione dei programmi applicativi e l'assegnazione delle risorse hardware ad essi
- **Protegge** le risorse degli utenti (e dei loro programmi) dagli altri utenti (e dai loro programmi) e da eventuali attori esterni

# Di cosa si occupa un sistema operativo?

- Di due aspetti molto diversi, apparentemente scorrelati:
- Da un lato, **astrae** le risorse hardware del computer, presentando agli sviluppatori dei programmi applicativi una macchina più facile da programmare
- Dall'altro, **gestisce, protegge e multiplexa** le risorse hardware del computer, assegnandole (sotto la direzione degli utenti) ai programmi in maniera equa ed efficiente e controllando che questi le usino correttamente

# Componenti di un sistema di elaborazione



- Utenti: persone, macchine, altri computer...
- Programmi applicativi: risolvono i problemi di calcolo degli utenti
- Sistema operativo: coordina e controlla l'uso delle risorse hardware
- Hardware: risorse di calcolo (CPU, periferiche, memorie di massa...)

Requisiti per i sistemi operativi



# Cosa si richiede ad un sistema operativo? (1)

- Oggigiorno i computer sono pervasivi: vi sono molteplici tipologie di computer utilizzati in scenari applicativi diversi
- Ormai in tutte queste tipologie di computer si tende ad installare un sistema operativo allo scopo di gestire l'hardware e semplificare la programmazione
- Ma ogni tipo di computer richiede che il sistema operativo che vi viene installato sopra abbia caratteristiche ben determinate per supportare i suoi scenari applicativi
- Che cosa si richiede ad un sistema operativo per un certo tipo di computer?

## Cosa si richiede ad un sistema operativo? (2)

- **Server, mainframe:** massimizzare la performance, rendere equa la condivisione delle risorse tra molti utenti
- **Laptop, PC, tablet:** massimizzare la facilità d'uso e la produttività della singola persona che lo usa
- **Dispositivi mobili:** ottimizzare i consumi energetici e la connettività
- **Sistemi embedded:** funzionare senza, o con minimo, intervento umano e reagire in tempo reale agli stimoli esterni (interrupt)

# La maledizione della generalità

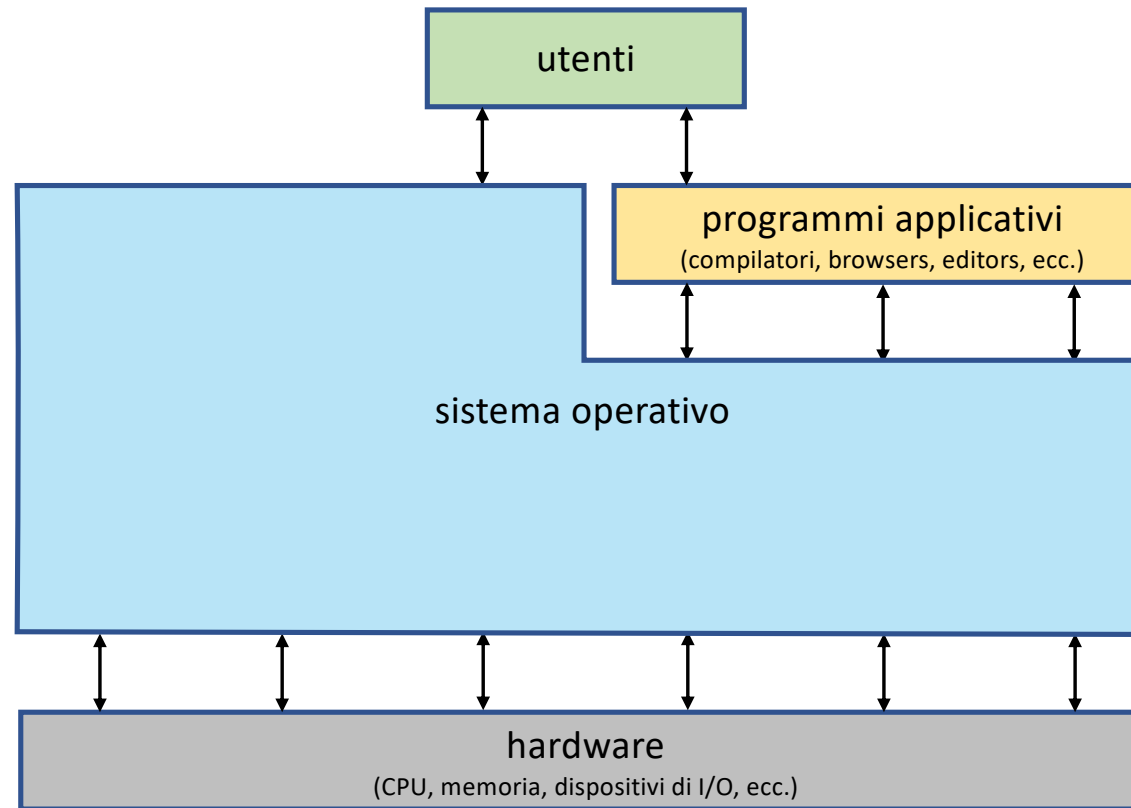
- Nella storia (ed anche oggi) alcuni sistemi operativi sono stati utilizzati per scenari applicativi diversi
- Ad esempio, oggi Linux è usato nei server, nei computer desktop e nei dispositivi mobili (come parte di Android)
- La **maledizione della generalità** afferma che, se un sistema operativo deve supportare un insieme di scenari applicativi troppo ampio, non sarà in grado di supportare nessuno di tali scenari particolarmente bene
- Questo si è visto con OS/360, che per riuscire funzionare su tutte le macchine della linea 360, ha finito per avere prestazioni scarse su ciascuna di esse

# Struttura e servizi dei sistemi operativi

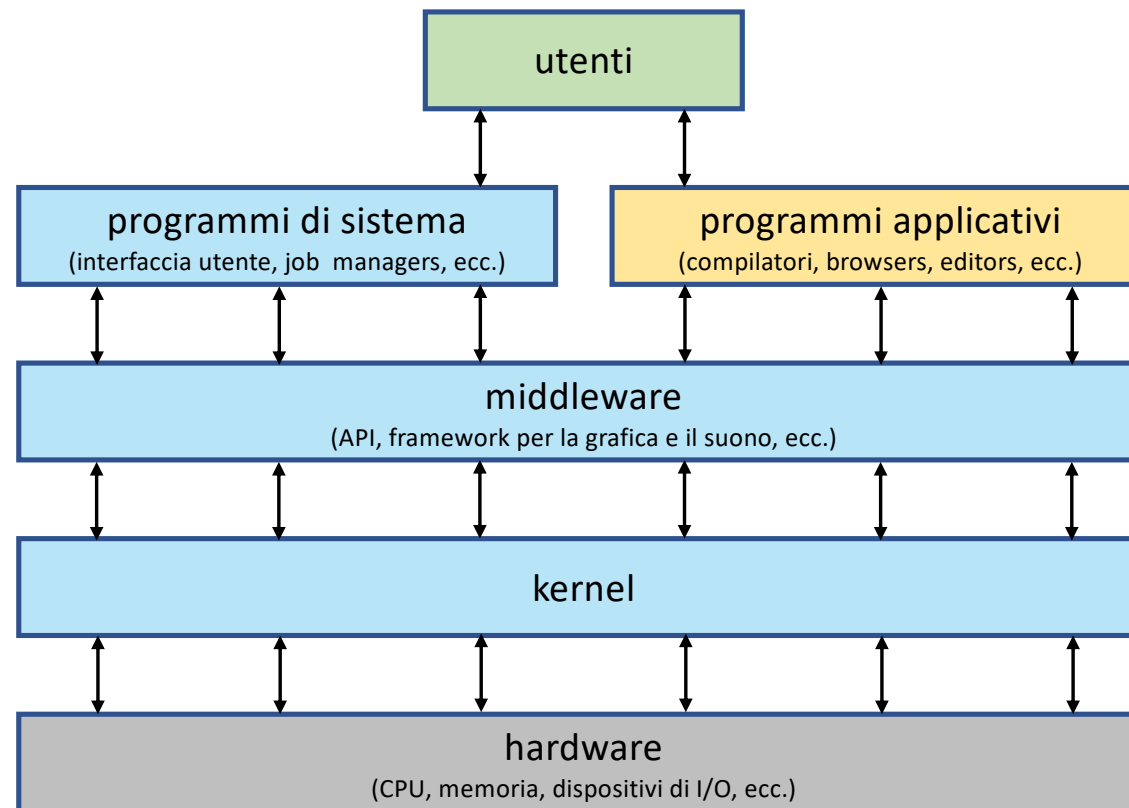
# Struttura dei sistemi operativi

- Non c'è una definizione universalmente accettata di quali programmi fanno parte di un sistema operativo
- In generale però un sistema operativo comprende almeno:
  - **Kernel**: il «programma sempre presente», che «si impadronisce» dell'hardware, lo gestisce, ed offre ai programmi i servizi per poterlo usare in maniera condivisa ed astratta
  - **Middleware**: servizi di alto livello che semplificano la programmazione di applicazioni (API, framework per grafica e per suono...)
  - **Programmi di sistema**: non sempre in esecuzione, offrono ulteriori funzionalità di supporto (gestione di jobs e processi, interfaccia utente...)
- Alcuni sistemi operativi forniscono anche dei programmi applicativi (editor, word processor, fogli di calcolo...), ma non li considereremo parti del sistema operativo stesso

# Componenti di un sistema di elaborazione, rivisitati



# Componenti di un sistema di elaborazione, rivisitati



# Servizi offerti da un sistema operativo

- Un sistema operativo offre un certo numero di servizi:
  - Alcuni per gli **utenti**: per gestire l'esecuzione dei programmi e stabilire a quali risorse hardware i programmi (e gli altri utenti) hanno diritto
  - Altri per i **programmi applicativi**: perché possano eseguire sul sistema di elaborazione usando le risorse astratte esposte dal sistema operativo
  - Infine vi sono dei servizi che garantiscono che il sistema di elaborazione funzioni in maniera efficiente
- Dal momento però che gli utenti interagiscono con il sistema operativo attraverso i programmi di sistema, alla fine il sistema operativo espone i suoi servizi in maniera che siano i programmi (applicativi o di sistema) ad usarli



# Principali servizi (1)

- **Controllo processi:** questi servizi permettono di caricare in memoria un programma, eseguirlo, identificare la sua terminazione e registrarne la condizione di terminazione (normale o errorea)
- **Gestione file:** questi servizi permettono di leggere, scrivere, e manipolare files e directory
- **Gestione dispositivi:** questi servizi permettono ai programmi di effettuare operazioni di input/output, ad esempio leggere da/scrivere su un terminale
- **Comunicazione tra processi:** i programmi in esecuzione possono collaborare tra di loro scambiandosi informazioni: questi servizi permettono ai programmi in esecuzione di comunicare
- **Protezione e sicurezza:** permette ai proprietari delle informazioni in un sistema multiutente o in rete di controllarne l'uso da parte di altri utenti e di difendere il sistema dagli accessi illegali

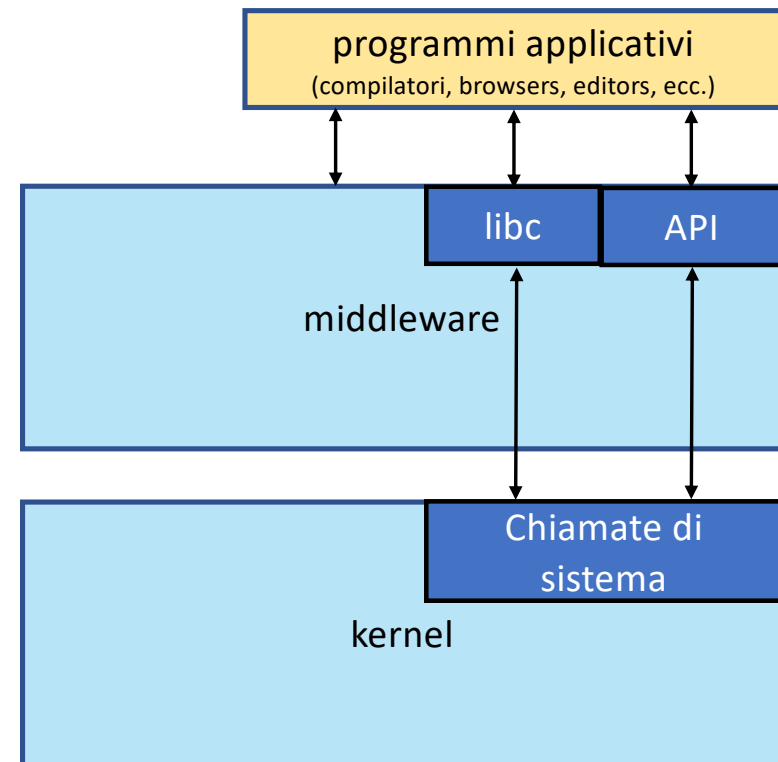
## Principali servizi (2)

- **Allocazione delle risorse:** alloca le risorse hardware (CPU, memoria, dispositivi di I/O) ai programmi in esecuzione in maniera equa ed efficiente
- **Rilevamento errori:** gli errori possono avvenire nell'hardware o nel software (es. divisione per zero); quando avvengono il sistema operativo deve intraprendere opportune azioni (recupero, terminazione del programma o segnalazione della condizione di errore al programma)
- **Logging:** mantiene traccia di quali programmi usano quali risorse, allo scopo di contabilizzarle

Chiamate di sistema ed API

# Chiamate di sistema ed API

- Il kernel offre i propri servizi ai programmi nella forma di **chiamate di sistema**, ossia di funzioni invocabili in un determinato linguaggio di programmazione (C, C++...)
- I programmi però non utilizzano direttamente le chiamate di sistema, ma delle librerie di middleware dette **Application Program Interface (API)** implementate invocando le chiamate di sistema
- Spesso le API sono fortemente legate con le librerie standard del linguaggio di implementazione (es. libc se le API sono implementate in C), al punto che anche queste diventano una parte implicita dell'API
- Perché questa distinzione?
  - Le API sono standardizzate, le chiamate di sistema no
  - Le API sono stabili, le chiamate di sistema possono variare al variare della versione del sistema operativo
  - Le API offrono funzionalità più ad alto livello e più semplici da usare, le chiamate di sistema offrono funzionalità più elementari e più complesse da usare



# Esempio confronto API Win32 e POSIX

Esempi di API:

- Win32 (sistemi Windows-like)
- POSIX (sistemi Unix-like, inclusi Linux e macOS)

	Win32	POSIX
<b>Controllo processi</b>	CreateProcess() ExitProcess() WaitForSingleObject()	fork() exit() wait()
<b>Gestione file</b>	CreateFile() ReadFile() WriteFile() CloseHandle()	open() read() write() close()
<b>Gestione dispositivi</b>	SetConsoleMode() ReadConsole() WriteConsole()	ioctl() read() write()
<b>Comunicazione tra processi</b>	CreatePipe() CreateFileMapping() MapViewOfFile()	pipe() shm_open() mmap()
<b>Protezione e sicurezza</b>	SetFileSecurity() InitializeSecurityDescriptor() SetSecurityDescriptorGroup()	chmod() umask() chown()

# Esempio di API POSIX

## 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:

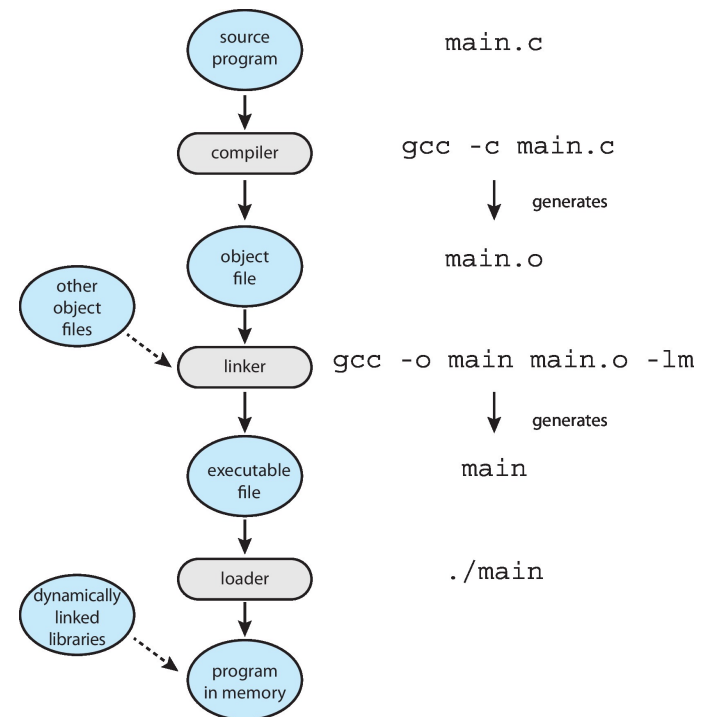
- `int fd`—the file descriptor to be read
- `void *buf`—a buffer into which the data will be read
- `size_t count`—the maximum number of bytes to be read into the buffer

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`.

Implementazione di chiamate di  
sistema ed API

# Loader e linker

- Un programma sorgente è compilato in un file oggetto che deve poter essere caricato a partire da qualsiasi locazione di memoria fisica (**file oggetto rilocabile**)
- I **linker**, o linkage editor, combinano più file oggetto (diversi file sorgente + librerie) per formare un file eseguibile, anch'esso rilocabile
- I **loader** si occupano di caricare in memoria i file eseguibili nel momento in cui devono essere eseguiti
- I loader effettuano la **rilocalizzazione**, ossia assegnano al programma gli indirizzi definitivi in memoria corrispondenti alle locazioni fisiche dove le diverse componenti del programma sono caricate
- Inoltre i loader effettuano il linking delle **librerie dinamiche**

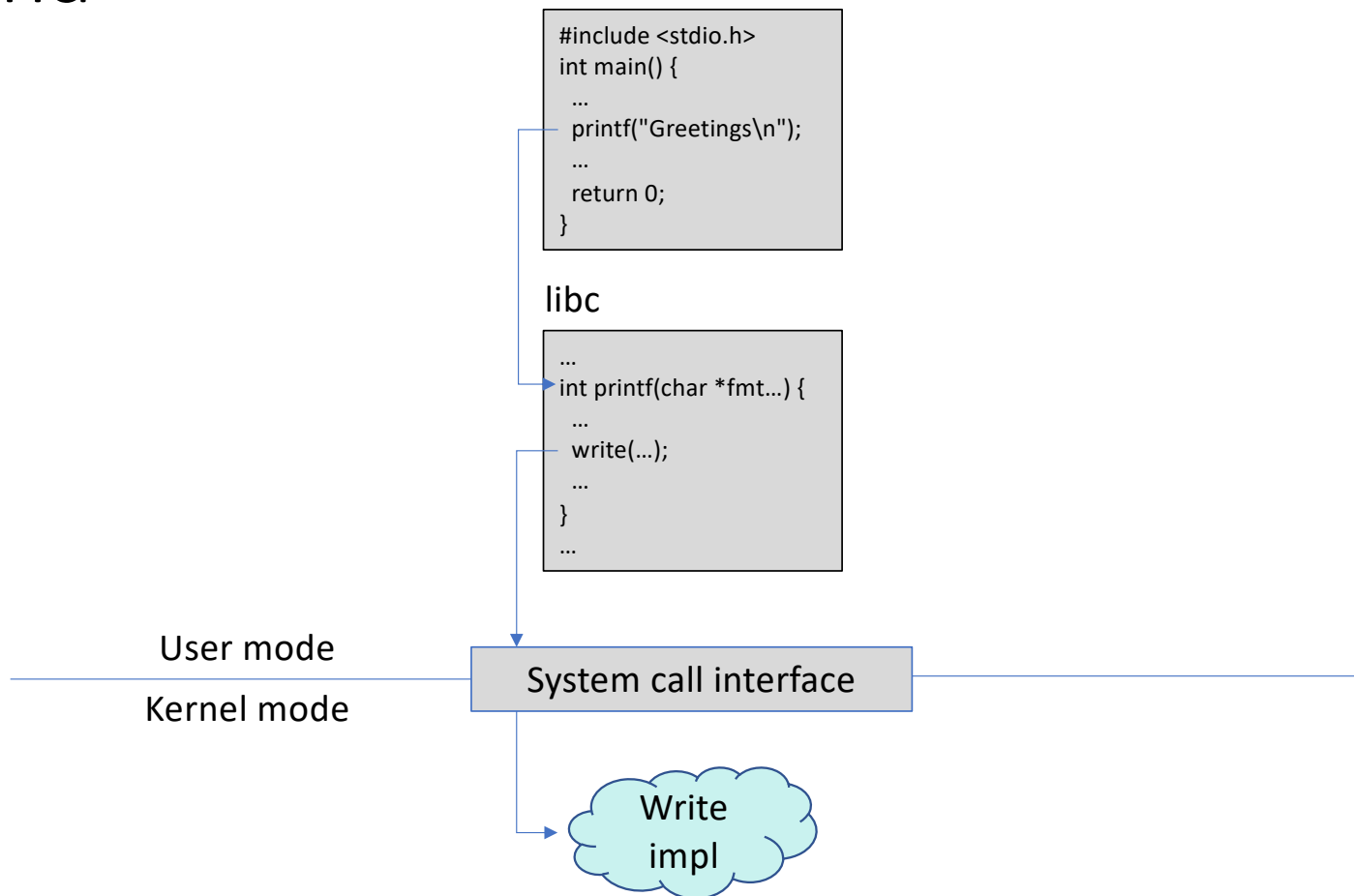




# Librerie dinamiche

- Nei sistemi operativi odierni non tutto il linking viene fatto a compile time: le librerie dinamiche vengono collegate quando il programma è caricato
- Il vantaggio delle librerie dinamiche è che queste possono essere condivise tra diversi programmi, riducendo le dimensioni dei programmi stessi e risparmiando memoria
- Un altro vantaggio si ha realizzando le API e le librerie standard del linguaggio come librerie dinamiche: se queste devono essere modificate, non occorre ricompilare tutti gli eseguibili per aggiornarli alla nuova versione delle librerie (purché non cambi l'ABI)

# Implementazione API attraverso chiamata di sistema



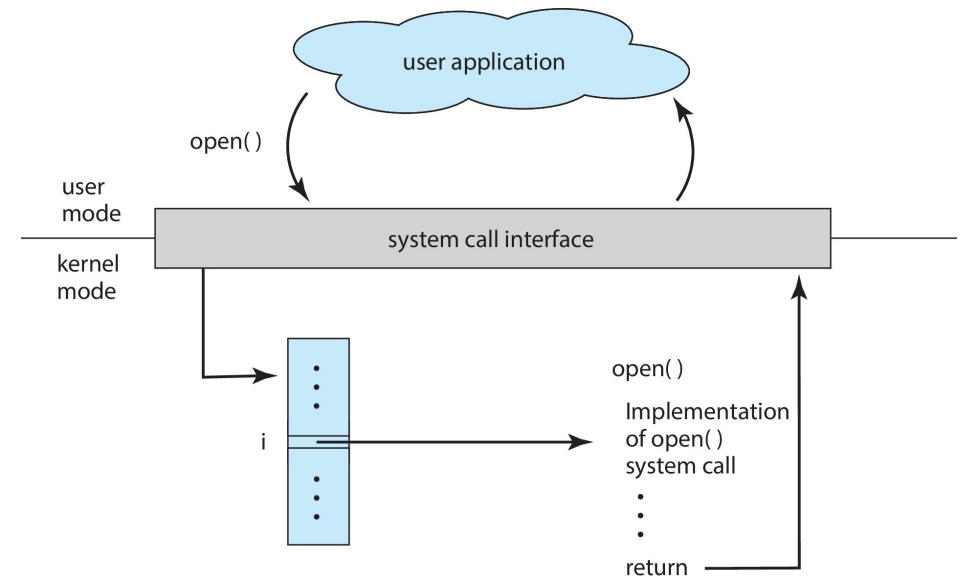
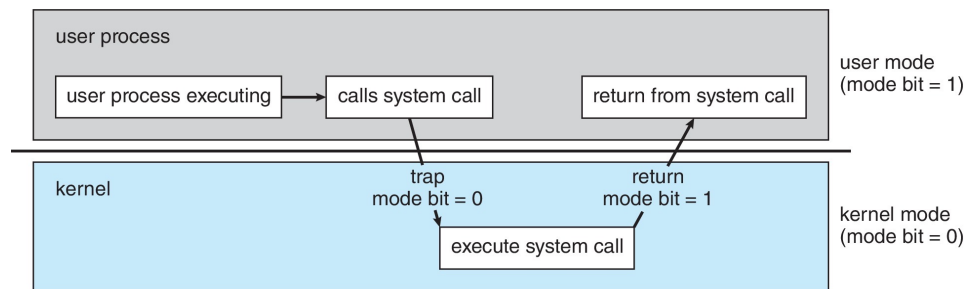
# Duplici modalità di funzionamento

- Permette al sistema operativo di proteggere se stesso dai programmi in esecuzione
- La CPU può funzionare in **modalità utente (user mode)** o in **modalità di sistema (kernel mode)** impostando un opportuno **bit di modalità**
- Alcune istruzioni del processore sono **privilegiate**, ossia eseguibili solo in modalità di sistema: in particolare, in user mode la CPU non può accedere alla memoria del kernel

# Implementazione delle chiamate di sistema (1)

- Una chiamata di sistema non è semplice da implementare come una normale chiamata di funzione, perché occorre effettuare una transizione da modalità utente a modalità di sistema
- Prima vengono preparati i parametri necessari:
  - Un numero che identifica quale chiamata di sistema va effettuata...
  - ...più tutti i parametri necessari alla specifica chiamata di sistema
  - (Vedremo nella slide successiva come vengono effettivamente passati questi dati)
- Quindi viene invocata un'opportuna istruzione macchina che genera un'eccezione software; essa fa passare la CPU in modalità di sistema e trasferisce il controllo ad una subroutine ad un determinato indirizzo di memoria
- La subroutine (**system call interface**) legge il numero identificativo della chiamata di sistema, effettua un lookup da una tabella interna dell'indirizzo della routine che effettivamente implementa la chiamata di sistema, e salta a tale indirizzo
- La routine invocata legge i parametri ed esegue la funzionalità richiesta
- Al ritorno il processore passa di nuovo in modalità utente

# Implementazione delle chiamate di sistema (2)



# Passaggio dei parametri alle chiamate di sistema

- Dal momento che l'invocazione delle chiamate di sistema passa per un'eccezione software, il passaggio di parametri è un po' complesso
- Metodo più semplice: nei registri del processore
  - Vantaggio: rapido
  - Svantaggio: utile solo per pochi parametri di tipi di dimensione limitata
- Altro metodo: passo in uno dei registri un indirizzo di memoria ad un blocco nel quale sono memorizzati i parametri
  - Usato da Linux in combinazione con il primo metodo
- Altro metodo: faccio push dei parametri sullo stack, dopo di che la system call interface esamina lo stack e recupera i parametri
  - Più flessibile
  - Ma anche più lento e macchinoso
  - (notare che, per ragioni di sicurezza, ogni programma ha uno stack per la modalità di sistema distinto dallo stack usato in modalità utente)

# Application binary interface (ABI)

- Si vuole far sì che, anche se il sistema operativo viene aggiornato, non vi sia bisogno di ricompilare/linkare le applicazioni se l'interfaccia delle API restano le stesse (ossia se cambiano le chiamate di sistema, o l'implementazione delle API)
- Un primo modo è implementare le API come librerie dinamiche: in tal modo non devo effettuare di nuovo il linking (statico) dell'applicazione al cambiare della implementazione delle API
- Occorre però un'ulteriore accortezza: non deve cambiare l'**application binary interface (ABI)**, ossia le convenzioni attraverso le quali il codice binario dell'applicazione si interfaccia con il codice binario della libreria dinamica delle API:
  - Come si chiama la funzione da invocare?
  - Quanti parametri ha? Di che tipo? In che ordine?
  - I parametri sono passati attraverso i registri, nei blocchi di memoria o sullo stack?
  - Come sono strutturati i tipi di dati?
- Se cambia anche l'interfaccia delle API (numero o tipo dei parametri, o i tipi di dati) le cose si complicano ulteriormente

# La scarsa portabilità degli eseguibili binari (1)

- Come facciamo ad avere applicazioni portabili su diversi sistemi di elaborazione?
- Tre possibili approcci:
  - Scrivere l'applicazione in un linguaggio con un interprete portabile (es. Python, Ruby): l'eseguibile in tal caso è il sorgente
  - Scrivere l'applicazione in un linguaggio con un ambiente runtime portabile (es. Java, .NET): l'eseguibile in tal caso è il bytecode
  - Scrivere l'applicazione utilizzando un linguaggio con un compilatore portabile ed API standardizzate: l'eseguibile è il file binario compilato e linkato
- Nei primi due casi l'eseguibile è normalmente uno solo per tutte le architetture
- Nel terzo caso invece occorre, di norma, generare un eseguibile distinto a variazioni anche minime del sistema di elaborazione (spesso anche solo al variare della versione del sistema operativo)
- Come mai?



# La scarsa portabilità degli eseguibili binari (2)

- Una prima banale ragione può essere la differenza nell'architettura hardware: ad esempio, un file binario prodotto per CPU ARM non può essere interpretato da un sistema di elaborazione con CPU x86-64, dal momento che le istruzioni macchina delle due CPU differiscono
- A parità di architettura hardware può esservi differenza nelle API utilizzate: ad esempio, su Windows si utilizzano le API Win32 e Win64, su Linux e macOS le API POSIX
- A parità di architettura e API può esservi differenza nel formato utilizzato per rappresentare i file binari: ad esempio, Linux riconosce il formato ELF, mentre macOS riconosce il formato MachO
- A parità di architettura, formato ed API può esservi differenza nelle chiamate di sistema che le implementano (se la libreria delle API è collegata staticamente)
- A parità di architettura, formato ed API, anche se la libreria delle API è collegata dinamicamente (o le chiamate di sistema sono le stesse), può esservi una differenza nell'ABI
- Solo quando tutti questi fattori sono identici un file binario è portabile da un sistema di elaborazione ad un altro

I programmi di sistema

# Programmi di sistema

- La maggior parte degli utenti utilizza i servizi del sistema operativo attraverso i programmi di sistema
- Questi permettono agli utenti di avere un ambiente più conveniente per l'esecuzione dei programmi, il loro sviluppo, e la gestione delle risorse del sistema

# Tipologie di programmi di sistema (1)

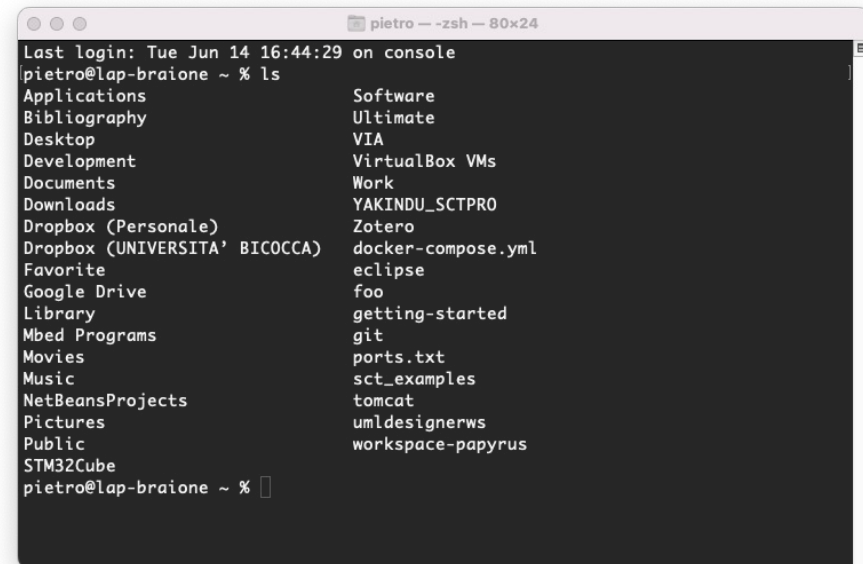
- **Interfaccia utente (UI):** permette agli utenti di interagire con il sistema stesso; può essere grafica (GUI) o a riga di comando (CLI); i sistemi mobili hanno un'interfaccia touch
- **Gestione file:** creazione, modifica, e cancellazione file e directory
- **Modifica dei file:** editor di testo, programmi per la manipolazione del contenuto dei file
- **Visualizzazione e modifica informazioni di stato:** data, ora, memoria disponibile, processi, utenti... fino informazioni complesse su prestazioni, accessi al sistema e debug. Alcuni sistemi implementano un **registry**, ossia un database delle informazioni di configurazione

# Tipologie di programmi di sistema (2)

- **Caricamento ed esecuzione dei programmi:** loader assoluti e rilocabili, linker, debugger
- **Ambienti di supporto alla programmazione:** compilatori, assembleri, debugger, interpreti per diversi linguaggi di programmazione
- **Comunicazione:** forniscono i meccanismi per creare connessioni tra utenti, programmi e sistemi; permettono di inviare messaggi agli schermi di un altro utente, di navigare il web, di inviare messaggi di posta elettronica, di accedere remotamente ad un altro computer, di trasferire file...
- **Servizi in background:** lanciati all'avvio, alcuni terminano, altri continuano l'esecuzione fino allo shutdown. Forniscono servizi quali verifica dello stato dei dischi, scheduling di jobs, logging...

# Interfaccia utente: l'interprete dei comandi

- L'interprete dei comandi permette agli utenti di immettere in maniera testuale le istruzioni che il sistema operativo deve eseguire
- In molti sistemi operativi è possibile configurare quale interprete dei comandi usare, nel qual caso è detto **shell**
- Due modi per implementare un comando:
  - Built-in: l'interprete esegue direttamente il comando (tipico nell'interprete di comandi di Windows)
  - Come programma di sistema: l'interprete manda in esecuzione il programma (tipico delle shell Unix e Unix-like)
- Spesso riconosce un vero e proprio linguaggio di programmazione con variabili, condizionali, cicli...

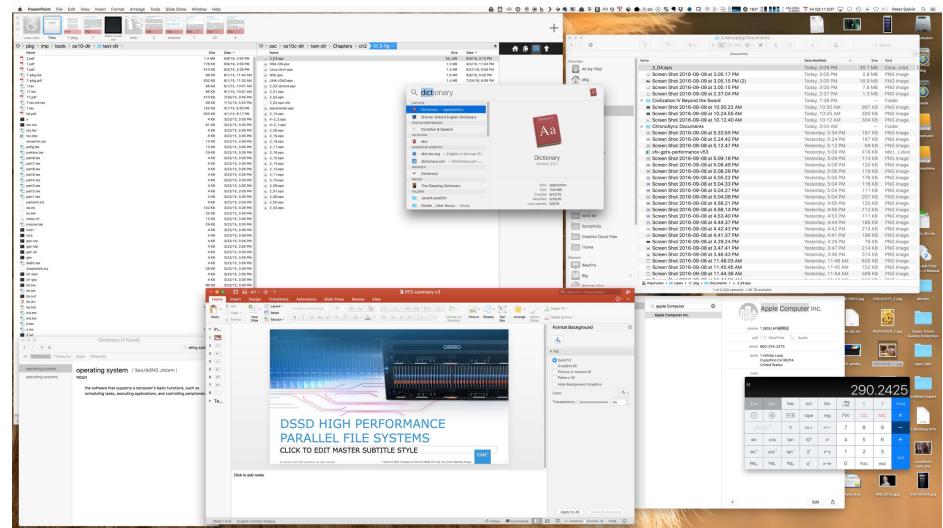


A screenshot of a terminal window titled "pietro -- zsh -- 80x24". The window shows the output of the command "ls" executed by the user "pietro" on the host "lap-braione". The output lists files and directories in two columns. The first column contains: Applications, Bibliography, Desktop, Development, Documents, Downloads, Dropbox (Personale), Dropbox (UNIVERSITA' BICOCCA), Favorite, Google Drive, Library, Mbed Programs, Movies, Music, NetBeansProjects, Pictures, Public, STM32Cube. The second column contains: Software, Ultimate, VIA, VirtualBox VMs, Work, YAKINDU\_SCTPRO, Zotero, docker-compose.yml, eclipse, foo, getting-started, git, ports.txt, sct\_examples, tomcat, umldesignerws, workspace-papyrus. The prompt "pietro@lap-braione ~ %" is visible at the bottom.

```
pietro@lap-braione ~ % ls
Applications      Software
Bibliography      Ultimate
Desktop           VIA
Development       VirtualBox VMs
Documents         Work
Downloads         YAKINDU_SCTPRO
Dropbox (Personale)  Zotero
Dropbox (UNIVERSITA' BICOCCA)  docker-compose.yml
Favorite          eclipse
Google Drive      foo
Library           getting-started
Mbed Programs     git
Movies            ports.txt
Music             sct_examples
NetBeansProjects  tomcat
Pictures          umldesignerws
Public            workspace-papyrus
STM32Cube
pietro@lap-braione ~ %
```

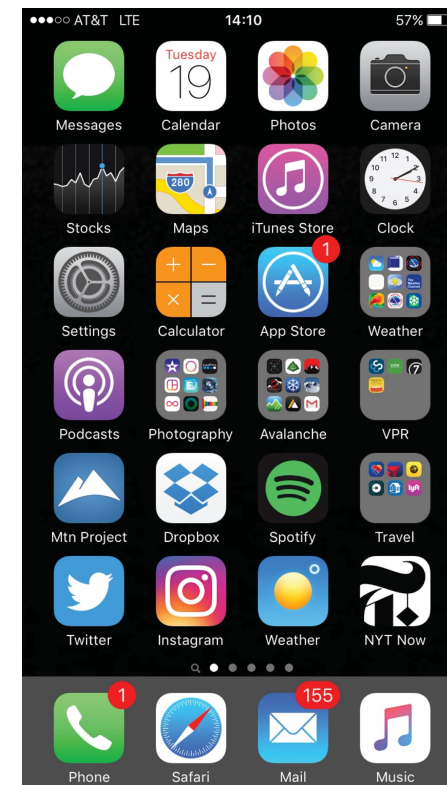
# Interfaccia utente: le interfacce grafiche

- L'interfaccia grafica (GUI) è di solito basata sulla metafora della scrivania, delle icone e delle cartelle (corrispondenti alle directory)
- Nate dalla ricerca presso lo Xerox PARC lab negli anni 70, popolarizzate dal computer Apple Macintosh negli anni 80
- Su Linux le più popolari sono Gnome e KDE



# Interfaccia utente: Le interfacce touch-screen

- I dispositivi mobili richiedono interfacce di nuovo tipo
- Nessun dispositivo di puntamento (mouse)
- Uso dei gesti (gestures)
- Tastiere virtuali
- Comandi vocali





# Scelta dell'interfaccia utente

- Gli utenti esperti tendono ad usare le CLI:
  - Più rapide
  - Programmabili
  - Permettono di accedere a tutte le funzionalità del sistema
- I sistemi Windows e Mac sono stati lungamente vincolati ad interfacce GUI, con limitato supporto alle CLI (la situazione è però cambiata)
- I dispositivi mobili offrono di norma solo l'interfaccia touch-screen
- Nella maggior parte dei sistemi ogni utente può decidere la propria interfaccia utente preferita

# L'implementazione dei programmi di sistema

- I programmi di sistema sono implementati utilizzando le API (né più né meno dei programmi applicativi)
- Consideriamo il comando `cp` delle shell dei sistemi operativi Unix-like:
  - `cp in.txt out.txt`
  - Copia il contenuto del file `in.txt` in un file `out.txt`
  - Se il file `out.txt` esiste, il contenuto precedente viene cancellato, altrimenti `out.txt` viene creato
- È implementato come programma di sistema
- Una possibile struttura del codice è riportata sulla destra (le invocazioni delle API sono riportate in grassetto)

- **Apri** `in.txt` in lettura
- Se non esiste
  - **Scrivi** un messaggio di errore su terminale
  - **Termina** il programma con codice errore
- **Apri** `out.txt` in scrittura
- Se non esiste, **crea** `out.txt`
- Loop
  - **Leggi** da `in.txt`
  - **Scrivi** su `out.txt`
- End loop
- **Chiudi** `in.txt`
- **Chiudi** `out.txt`
- **Termina** normalmente

Struttura del kernel

# Sottosistemi del kernel

- Basati sulle categorie dei servizi offerti dal kernel stesso (e quindi sulle categorie delle chiamate di sistema)
- I principali sono:
  - Gestione dei processi e dei thread
  - Comunicazione tra processi e sincronizzazione
  - Gestione della memoria
  - Gestione dell'I/O
  - File system

# Organizzazione del kernel

- Il kernel di un sistema operativo general-purpose è un programma
  - Di dimensioni elevate e complesso
  - Che deve operare molto rapidamente per non sottrarre tempo di elaborazione ai programmi applicativi
  - Un cui malfunzionamento può provocare il crash dell'intero sistema di elaborazione
- Si pone quindi il problema di come progettarlo in maniera da garantire rapidità e correttezza nonostante dimensioni e complessità
- Alcune possibilità:
  - Struttura monolitica
  - Struttura a strati
  - Struttura a microkernel
  - Struttura a moduli
  - Struttura ibrida

# Struttura monolitica

- Il sistema operativo Unix originale aveva una struttura monolitica, dove il kernel è un singolo file binario statico
- Il kernel forniva un elevato numero di funzionalità:
  - Scheduling CPU
  - File system
  - Gestione della memoria, swapping, memoria virtuale
  - Device drivers
  - ...
- Vantaggi: elevate prestazioni
- Svantaggi:
  - Complessità
  - Fragilità ai bug
  - Necessità di ricompilare il kernel (e riavviare il sistema) se bisogna aggiungere una funzionalità, ad esempio il driver per una nuova periferica

# Struttura a strati

- Negli approcci stratificati il sistema operativo è diviso in un insieme di livelli, o strati
- Lo strato più basso interagisce con l'hardware, lo strato n-esimo interagisce solo con lo strato (n-1)-esimo
- L'approccio offre due vantaggi:
  - Ogni strato può essere progettato e implementato indipendentemente dagli altri
  - È possibile verificare la correttezza di uno strato indipendentemente da quella degli altri
  - Ogni strato nasconde le funzionalità degli strati sottostanti e presenta allo strato soprastante una macchina dalle caratteristiche più astratte
- In realtà pochi sistemi operativi usano questo approccio in maniera pura:
  - È difficile definire esattamente quali funzionalità deve avere uno strato
  - Ogni strato introduce un overhead che peggiora le prestazioni
- È comunque conveniente strutturare alcune parti del sistema operativo a strati (es. file system o stack di rete)

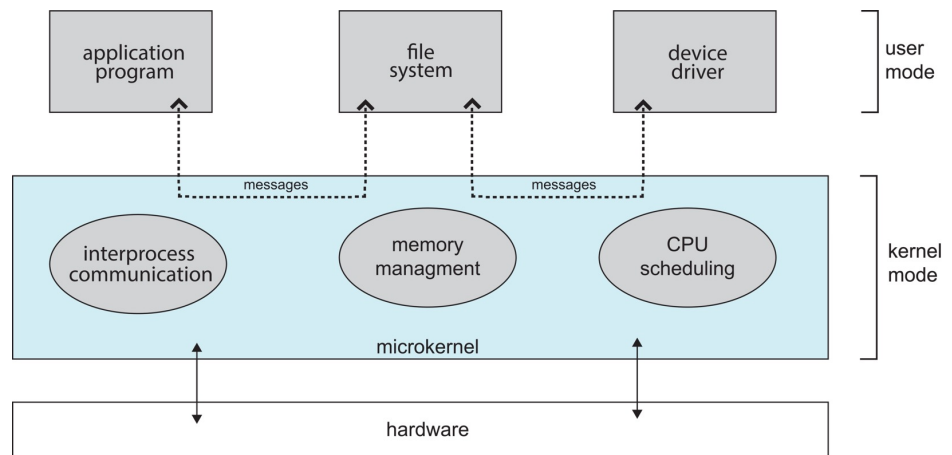
# Struttura a microkernel (1)

- Il principale problema dei kernel monolitici è la loro complessità, e di conseguenza fragilità e inaffidabilità
- La struttura a microkernel sposta quanti più servizi possibile fuori dal kernel in programmi di sistema, mantenendo nel kernel l'insieme minimo di servizi indispensabili per implementare gli altri
- Il kernel è definito microkernel dal momento che ha dimensioni molto ridotte
- L'approccio è stato proposto negli anni 80 con il sistema operativo Mach



## Struttura a microkernel (2)

- Un microkernel offre pochi servizi, di solito lo scheduling dei processi, (una parte della) gestione della memoria e la comunicazione tra processi
- Gli altri servizi (es. filesystem e device drivers) vengono implementati a livello utente
- Per chiedere un servizio, un programma comunica con il programma di sistema che lo implementa attraverso le primitive di comunicazione offerte dal microkernel



# Struttura a microkernel (3)

- Vantaggi:
  - Facilità di estensione del sistema operativo: posso aggiungere un nuovo servizio senza dover modificare il kernel
  - Maggiore affidabilità: se un servizio va in crash non manda in crash il kernel; un kernel piccolo può essere reso più affidabile con meno sforzo
- Svantaggi:
  - Overhead: una tipica richiesta di servizio deve transitare dal processo richiedente al microkernel, al processo di sistema destinatario, e viceversa, con molti passaggi tra user e kernel mode, comunicazioni, cambi di contesto...
- I sistemi a microkernel puri vengono usati nelle applicazioni che richiedono elevata affidabilità (QNX neutrino, L4se)
- Altri sistemi inizialmente a microkernel si sono evoluti in sistemi ibridi (Windows NT, Darwin – kernel di macOS e iOS)

# Struttura a moduli

- Il kernel è strutturato in componenti dinamicamente caricabili (moduli), che parlano tra di loro attraverso interfacce
- Quando il kernel ha bisogno di offrire un certo servizio, carica dinamicamente il modulo che lo implementa; quando il servizio non è più necessario, il kernel può scaricare il modulo
- Questo approccio ha alcune caratteristiche di quelli a strati e a microkernel, ma i moduli eseguono in modalità kernel, e quindi con minore overhead (ma anche con minore isolamento tra di loro)

# Sistemi ibridi

- In pratica pochi sistemi operativi adottano una struttura «pura»: quasi tutti combinano i diversi approcci allo scopo di ottenere sistemi indirizzati alle prestazioni, sicurezza, usabilità...
- Esempio: Linux e Solaris sono monolitici per avere prestazioni elevate, ma supportano anche i moduli del kernel per poter caricare e scaricare dinamicamente funzionalità
- Altro esempio: Windows inizialmente aveva una struttura a microkernel, successivamente diversi servizi sono stati riportati nel kernel per migliorare le prestazioni. Ora è essenzialmente monolitico, pur conservando alcune caratteristiche della precedente architettura a microkernel. Inoltre supporta i moduli del kernel

# Politiche e meccanismi

- Quando discutiamo come è realizzato il kernel è importante distinguere tra **politiche** e **meccanismi**
- Una politica dice *quando* una certa operazione viene effettuata; ad esempio: sotto che condizioni il kernel decide che è il momento di sospendere l'esecuzione di un programma per far riprendere l'esecuzione di un altro?
- Un meccanismo spiega *come* una certa operazione è effettuata; ad esempio: come fa il kernel a sospendere l'esecuzione di un programma in maniera che poi possa riprendere? Come fa a ripristinare l'esecuzione di un programma precedentemente sospeso?
- Le politiche impattano profondamente sulle caratteristiche percepite del sistema di elaborazione, i meccanismi no (se sono sufficientemente rapidi)
- I meccanismi sono più stabili delle politiche, che spesso cambiano in funzione delle caratteristiche percepite che vogliamo che il sistema di elaborazione abbia
- Avere politiche configurabili è utile per combattere la maledizione della generalità