

Servizi e struttura dei sistemi operativi

Pietro Braione

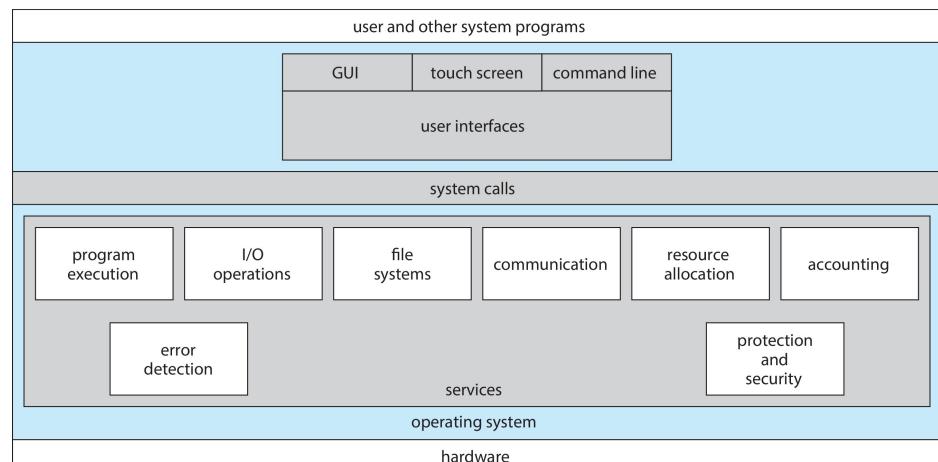
Reti e Sistemi Operativi – Anno accademico 2021-2022

Obiettivi

- Identificare i servizi offerti da un OS
- Illustrare le interfacce del sistema operativo verso l'utente e verso i programmi (chiamate di sistema) per utilizzare i servizi dell'OS
- Illustrare la differenza tra politiche e meccanismi negli OS

Servizi offerti da un sistema operativo

- Un sistema operativo offre un ambiente che offre un certo numero di servizi
- Alcuni servizi servono a facilitare i programmatore di applicazioni nella scrittura dei programmi
- Altri servizi servono agli utenti per gestire l'hardware, ed in particolare l'esecuzione dei programmi su di esso
- Infine vi sono dei servizi che garantiscono che il sistema di elaborazione funzioni in maniera efficiente



Servizi per l'utente e per i programmi

- **Interfaccia utente (UI)**: può essere grafica (GUI) o a riga di comando (CLI); i sistemi mobili hanno un'interfaccia touch
- **Esecuzione di un programma**: caricamento in memoria di un programma, sua esecuzione, registrazione della condizione di terminazione (normale o erronea)
- **Operazioni di I/O**: i processi possono richiedere di effettuare operazioni di input/output, ad esempio leggere da/scrivere su una console
- **Gestione del file system**: i processi possono richiedere di leggere, scrivere, manipolare files e directory
- **Comunicazione tra processi**: processi diversi possono dover scambiare informazioni; le tecniche usate sono la memoria condivisa e lo scambio di messaggi
- **Rilevamento errori**: possono avvenire nell'hardware o nel software (es. divisione per zero), il sistema operativo deve intraprendere opportune azioni (recupero, terminazione del processo, segnalazione al processo)

Servizi che assicurano il funzionamento efficiente del sistema

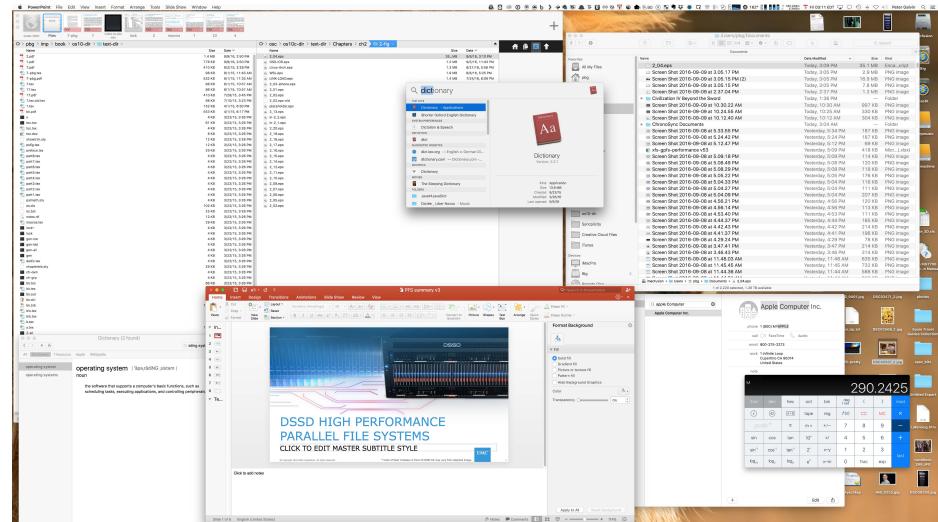
- **Allocazione delle risorse:** il sistema operativo alloca diverse risorse (CPU, memoria, dispositivi di I/O) ai programmi in esecuzione
- **Logging:** mantiene traccia di quali programmi usano quali risorse, allo scopo di contabilizzarle
- **Protezione e sicurezza:** permette ai proprietari delle informazioni in un sistema multiutente o in rete di controllare l'uso delle informazioni:
 - **Protezione:** assicura che l'accesso alle risorse sia controllato
 - **Sicurezza:** identificazione degli utenti, difesa da accessi non autorizzati

Interfaccia utente: L'interprete dei comandi

- L'interprete dei comandi permette agli utenti di immettere direttamente (in maniera testuale) le istruzioni che il sistema operativo deve eseguire
- Nella maggior parte dei sistemi operativi è un programma speciale
- In molti sistemi operativi (es. Unix o Linux) 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)

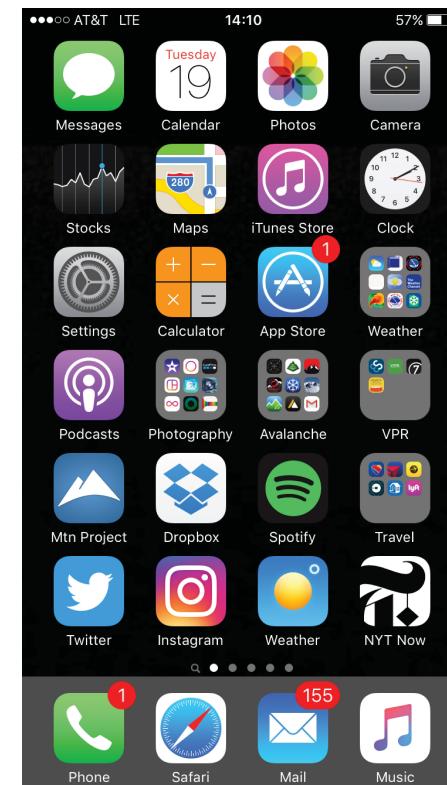
Interfaccia utente: Le interfacce grafiche

- 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 UNIX/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

Chiamate di sistema

- Le chiamate di sistema costituiscono l'interfaccia che il sistema operativo offre ai programmi per accedere ai servizi che offre
- Tipicamente funzioni scritte in C o C++, in alcuni casi in assembly
- I programmi di regola non utilizzano direttamente le chiamate di sistema, ma piuttosto delle librerie di livello più alto, di solito standardizzate, dette **Application Program Interface (API)**
- Esempi di API:
 - Win32 (sistemi Windows-like), POSIX (sistemi UNIX-like, inclusi Linux e macOS), Java API (per la Java Virtual Machine)
 - Spesso incapsulate nelle librerie standard dei linguaggi di programmazione (es. libc)
- È preferibile usare le API piuttosto che direttamente le chiamate di sistema per diversi motivi:
 - Le API sono standard, le chiamate di sistema sono OS-specific
 - Le API sono stabili, le chiamate di sistema potrebbero variare nel tempo
 - Le API sono più semplici da usare (di regola) delle chiamate di sistema

Esempio di API standard

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)
```

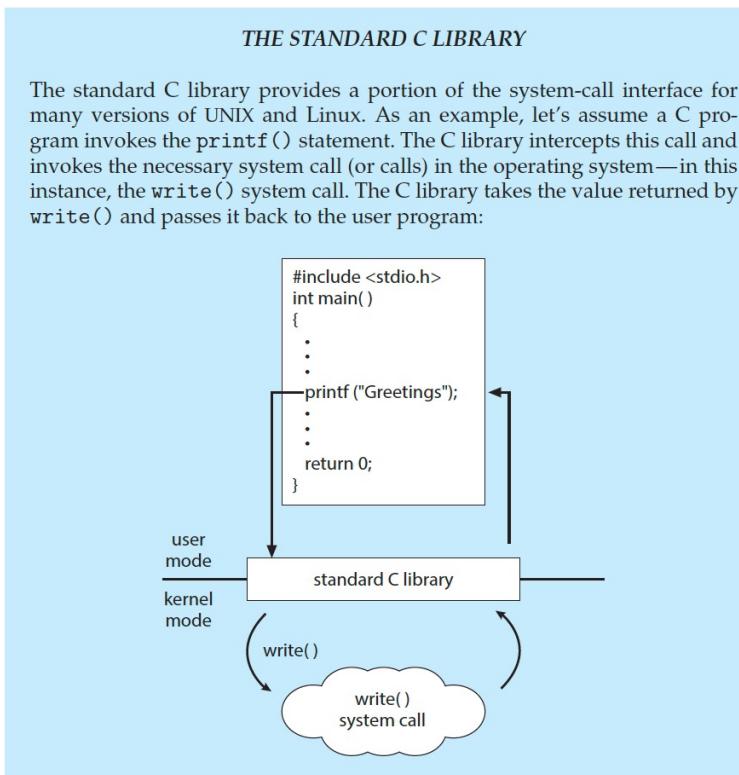
return function parameters
value name

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.

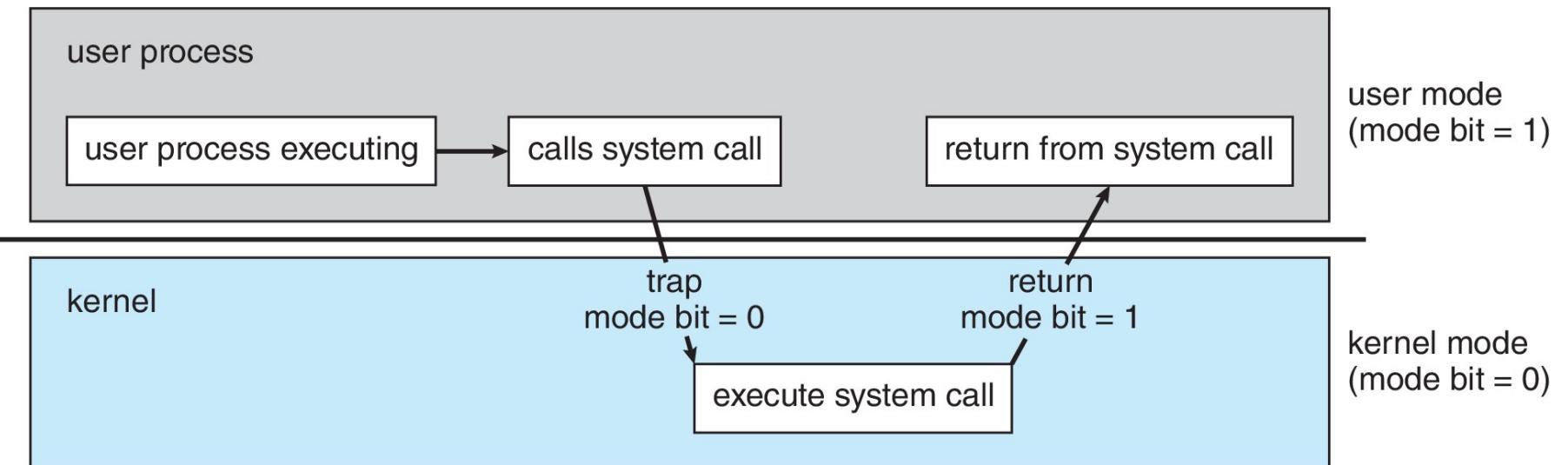
Implementazione API standard attraverso chiamata di sistema



Duplice modalità di funzionamento

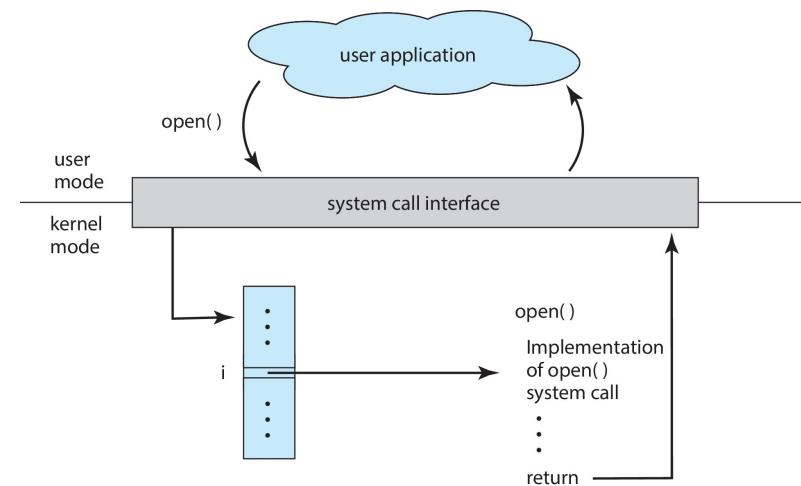
- La duplice modalità (dual mode) permette al sistema operativo di proteggere se stesso dai programmi in esecuzione, e i programmi in esecuzione l'uno dall'altro
- La CPU deve offrire due modalità di funzionamento, una modalità utente (**user mode**) e una modalità di sistema (**kernel mode**), di solito attivata attraverso un opportuno **bit di modalità** fornito dall'hardware:
 - Permette di distinguere l'esecuzione di codice utente e di codice del kernel
 - Alcune istruzioni del processore sono **privilegiate**, ossia eseguibili solo in kernel mode
- Le chiamate di sistema e gli interrupt portano la CPU in modalità di sistema, al ritorno viene reimpostata la modalità utente
- Diversi processori hanno più di due modalità, sfruttate ad esempio per la virtualizzazione (modalità virtual machine monitor, o VMM)

Transizioni tra modalità



Implementazione delle chiamate di sistema

- Tipicamente viene invocata un'istruzione che genera un'eccezione
- L'applicazione passa al kernel un numero che identifica quale chiamata di sistema vuole effettuare
 - Vedremo dopo come vengono passati i parametri
- La system call interface intercetta l'eccezione, legge il numero identificativo ed effettua un lookup dell'indirizzo della routine da eseguire in una tabella interna



Passaggio dei parametri alle chiamate di sistema

- Dal momento che l'invocazione delle chiamate di sistema passa per un'istruzione di eccezione software, il passaggio di parametri è un po' complicato
- Metodo più semplice: nei registri del processore
 - Vantaggio: rapido
 - Svantaggio: utile solo per pochi parametri di tipi limitati
- 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 il codice del kernel (system call interface) esamina lo stack del processo e recupera i parametri
 - Più flessibile
 - Ma anche più lento e macchinoso

Perché le applicazioni dipendono dall'OS? (1)

- Le applicazioni compilate per un determinato OS di regola non sono eseguibili su un OS diverso, eppure abbiamo applicazioni portabili (esempio, Chrome su Windows, macOS e Linux)
- Tre possibili approcci alla portabilità:
 - Scrivere l'applicazione in un linguaggio interpretato con un interprete portabile, ad esempio Python
 - Scrivere l'applicazione in un linguaggio che utilizza una macchina virtuale portabile, ad esempio Java
 - Scrivere l'applicazione utilizzando un linguaggio con un compilatore portabile, ed usando API standardizzate e portabili
- Mentre nei primi due casi il file eseguibile è di norma uno solo per tutte le architetture, nel terzo caso occorre ricompilare l'applicazione, e quindi generare un nuovo eseguibile, al cambiare del sistema operativo e della architettura hardware
 - Come mai?

Perché le applicazioni dipendono dall'OS? (2)

- Un primo, banale motivo può essere la differenza nel tipo di CPU, se sono basate su set di istruzioni diversi. Ad esempio, un file eseguibile per processore ARM non può essere interpretato da un processore x86-64
- Un secondo motivo è il formato binario utilizzato dal sistema operativo per rappresentare i file eseguibili. Anche in questi casi esistono degli standard (ad esempio, ELF e COFF), ma diversi sistemi operativi potrebbero usare standard diversi
- Un terzo motivo è la dipendenza dalle chiamate di sistema: anche se le API sono uguali su sistemi operativi diversi, le system call che le implementano potrebbero essere diverse, e se il codice macchina che implementa la chiamata della system call è collegato staticamente al binario, questo diventa non portabile
- Un quarto motivo sono le convenzioni utilizzate dal codice binario per interfacciarsi con il sistema operativo (parametri passati attraverso i registri, nei blocchi di memoria o sullo stack? Che tipi di dati vengono usati? ...): anche in questo caso esistono degli standard, detti ABI (Application Binary Interface)
- Solo quando tutti questi fattori sono identici un file binario è portabile da un sistema di elaborazione ad un altro; in pratica, un file binario è di norma fortemente legato alla architettura hardware e al sistema operativo sul quale può eseguire

Meccanismi e politiche

- È importante, quando discutiamo di come è realizzato un sistema operativo, distinguere tra **meccanismi e politiche**:
 - I meccanismi dicono *come* una certa operazione è effettuata; ad esempio: come si fa a portare la memoria di un processo da disco in memoria centrale?
 - Le politiche dicono *quando* una certa operazione viene effettuata; ad esempio: sotto che condizioni il sistema operativo decide che è il momento di portare la memoria di un certo processo da disco in memoria centrale?
- Le politiche impattano profondamente sulle caratteristiche percepite del sistema operativo (in particolare la performance)
- Idealmente, i meccanismi dovrebbero rimanere abbastanza stabili al cambiare delle politiche, le quali sono più «volatili»
- I sistemi a microkernel cercano di offrire meccanismi elementari, in maniera da potere implementare politiche arbitrarie e configurabili, anche dinamicamente, a livello dei servizi utente