Capitolo 2: Servizi del sistema operativo

Table of contents

- Servizi di Sistema
  - Registry
  - Supporto per linguaggi di programmazione
  - Servizi di background
  - Linkers e Loaders
- Perché le applicazioni sono tipiche del sistema operativo?
  - Come risolvere il problema
- Come si progetta un Sistema Operativo?
  - Obbiettivi quando si sviluppa un SO
- Principio più importante
- Implementazione (dei sistemi operativi)
  - Mix di Linguaggi di programmazione
- Come è strutturato un Sistema Operativo
  - Struttura Monolitica - UNIX originale
  - Approccio a "strati"
  - Microkernel
  - Quale OS usa il microkernel?

Interfaccia

Alcuni dei servizi del SO sono di diretto interesse per l'utente, in particolare tutti i SO devono fornire un'interfaccia utente.

Un' interfaccia storica era quella a **linea di comando**, dove era presente un cursore con cui si potevano scrivere dei comandi. Questa interfaccia è nota come **CLI**, inoltre essa è **sempre** accessibile anche nei sistemi ad interfaccia grafica, proprio perchè permette di lavorare con una velocità superiore (a chi la sa usare).

Sui dispositivi **mobili** solitamente è presente un'interfaccia di tipo **touch screen**, dove è possibile toccare su ciò che si vuole fare.

Esecuzione di un programma

Ovviamente, lo scopo principale del sistema operativo è quello di **eseguire i programmi**. Il sistema deve quindi essere in grado di caricare un programma in memoria ed eseguirlo, sia in modo normale che anomalo, indicando l'errore.

Operazioni I/O

Le operazioni di I/O sono sotto il controllo del sistema operativo, sottoforma di interfacce **astratte** come il **file system**.

Manipolazione del file system

Il File System è di particolare interesse, perchè i programmi hanno bisogno di leggere e scrivere files e directories, creare o eliminarle, ricercare degli elementi ecc.

Comunicazioni

Questo ambito è uno di quelli nati più tardi nell'ambito dei sistemi operativi. Windows 3.1, che è stato in commercio fino all'avvento di windows 95, non aveva un supporto per le comunicazioni. Questo vuol dire che se avessimo voluto connetterci ad internet, avremmo dovuto utilizzare programmi di **terze parti**. Questo ci fa capire che inizialmente i computers diffusi non si parlavano tra loro. Ai giorni nostri ha una rilevanza fondamentale.

Rilevazione di errori

I SO devono sempre essere a conoscenza di errori possibili.

Questi errori possono verificarsi nell'hardware della CPU o memoria, nei dispositivi di I/O, oppure in un programma utente.

Il sistema deve quindi essere in grado di **gestire** un errore, rilevandolo, e poi comunicandolo all'utente.

Allocazione delle risorse

Questo tipo di servizio non impatta in modo diretto l'utente, proprio perchè egli non è a conoscenza di queste operazioni. Quando ci sono diversi utenti o **jobs** in esecuzioni in modo **concorrente**, le risorse devono essere allocate per ognuno di essi.

Logging

Questa caratteristica del SO ci permette di tenere traccia delle operazioni che sono state eseguite, degli errori che si sono verificati, gli accessi da parte degli utenti. Questa è una caratteristica molto importante.

Protezione e sicurezza

E' una caratteristica che, pur non essendo direttamente connessa all'esecuzione dei programmi, è fondamentale per tutti gli utenti ed il corretto funzionamento del SO.

CLI - interfaccia a linea di comando

La CLI, è anche detta interprete di comandi. Questo perché quando scriviamo un comando sulla linea di comando, c'è un **programma** del SO che prende la nostra stringa, la suddivide nei suoi singoli pezzi, comprende i singoli pezzi ed interpreta il comando eseguendo qualcosa.

Questo interprete è una caratteristica implementata all'interno del **kernel**, altre volte viene implementata mediante un programma di sistema, principalmente nei sistemi **UNIX**. Infatti, in ambiente UNIX, non solo possiamo lanciare l'ambiente a linea di comando, ma possiamo anche scegliere tra diversi interpreti per avere delle **caratteristiche** lievemente diverse.

In windows l'interprete di comandi è il CMD o il PowerShell.

Che cosa fanno questi tipi di programmi?

Questi programmi si limitano a prelevare i comandi digitati, decomporli nei vari pezzi e mandarli in esecuzione. A volte i comandi sono integrati all'interno del sistema, mentre altre volte vengono utilizzati dei programmi esterni.

GUI - Interfaccia Grafica

L'interfaccia grafica, nella maggioranza dei casi, usa la **metafora** del **desktop**, o scrivania. Ovvero abbiamo uno spazio virtuale in cui sono organizzati tutti i file e programmi. Questo tipo di visualzzazione è un concetto relativamente recente, infatti nel mondo delle GUI l'interfaccia a desktop è stata portata per la prima volta da parte di windows.

Questo tipo di visualizzazione è nata nei laboratori Xerox PARC. La Xerox aveva dei laboratri dove è stata fatta molta attività di ricerca, come alcuni schermi ed addirittura l'attuale **mouse**.

System Calls

Le Syscalls forniscono l'interfaccia da programma per i servizi forniti dal sistema operativo.

Per accedere alle Syscalls ci sono delle funzioni scritte in linguaggi ad alto livello, solitamente C o C++, che permettono direttamente di invocare queste funzioni senza scrivere **codice macchina**. Un esempio di Syscall potrebbe essere la funzione **open** che si trova all'interno del linguaccio C.

Le Syscall forniscono le API per l'accesso ai servizi del sistema operativo. I servizi del SO sono accessibili attraverso delle API che sono diverse a seconda del SO stesso. Nei sistemi windows l'API si chiama **Win32**, nei sistemi POSIX, e tutti i sistemi basati su di esso (linux, mac os) abbiamo **POSIX API**.

Quante Syscall vengono eseguite?

Se prendiamo un programma che permettono di visualizzare le Syscalls nel momento in cui lanciamo in esecuzione un programma, scopriamo che ogni programma, durante la sua esecuzione, esegue migliaia di chiamate prima di terminare l'esecuzione.

In realtà i programmi non possono effettuare operazioni I/O, quindi ogni volta che è necessaria un'interazione con il file system, bisogna richiedere dei servizi al sistema operativo. Questo vale per ogni operazione che richiede I/O, come leggere da tastiera, scrivere un file o stampare a video.

API

Per un esempio di una API standard, consideriamo la funzione **read()** disponibile nei sistemi UNIX. L'API per questa funzione è ottenuta dalla pagina **man** invocando il comando:

`man read`

Il comando **man** è un comando dei sistemi UNIX che permette di accedere al manuale sia delle syscall che dei programmi di sistema. Quindi sia i programmi di sistema, che tutto cio che riguarda il programmatore è disponibile nel manuale.

Nei sistemi UNIX le API e manuali sono disponibili online, questo che vuol dire che qualsiasi manuale è accessibile facilmente.

Esempio

Se ad esempio digitiamo `man open` in un CLI UNIX otteniamo:

```
OPEN(1)          BSD General Commands Manual          OPEN(1)

NAME
```

open -- open files and directories

SYNOPSIS

open [-e] [-t] [-f] [-F] [-W] [-R] [-n] [-g] [-j] [-h] [-s sdk]
[-b bundle_identifier] [-a application] file ... [--args arg1 ...]

DESCRIPTION

The open command opens a file (or a directory or URL), just as if you had double-clicked the file's icon. If no application name is specified, the default application as determined via LaunchServices is used to open the specified files.

If the file is in the form of a URL, the file will be opened as a URL.

You can specify one or more file names (or pathnames), which are interpreted relative to the shell or Terminal window's current working directory. For example, the following command would open all Word files in the current working directory:

open *.doc

Opened applications inherit environment variables just as if you had launched the application directly through its full path. This behavior was also present in Tiger.

The options are as follows:

-a application
Specifies the application to use for opening the file

-b bundle_indentifier
Specifies the bundle identifier for the application to use when opening the file

-e Causes the file to be opened with /Applications/TextEdit

-t Causes the file to be opened with the default text editor, as deter-

```
     mined via LaunchServices

-f   Reads input from standard input and opens the results in the default
     text editor.  End input by sending EOF character (type Control-D).
     Also useful for piping output to open and having it open in the
     default text editor.
```

> Per uscire dalla visualizzazione premiamo il tasto **q**

Passaggio dei parametri nelle Syscalls

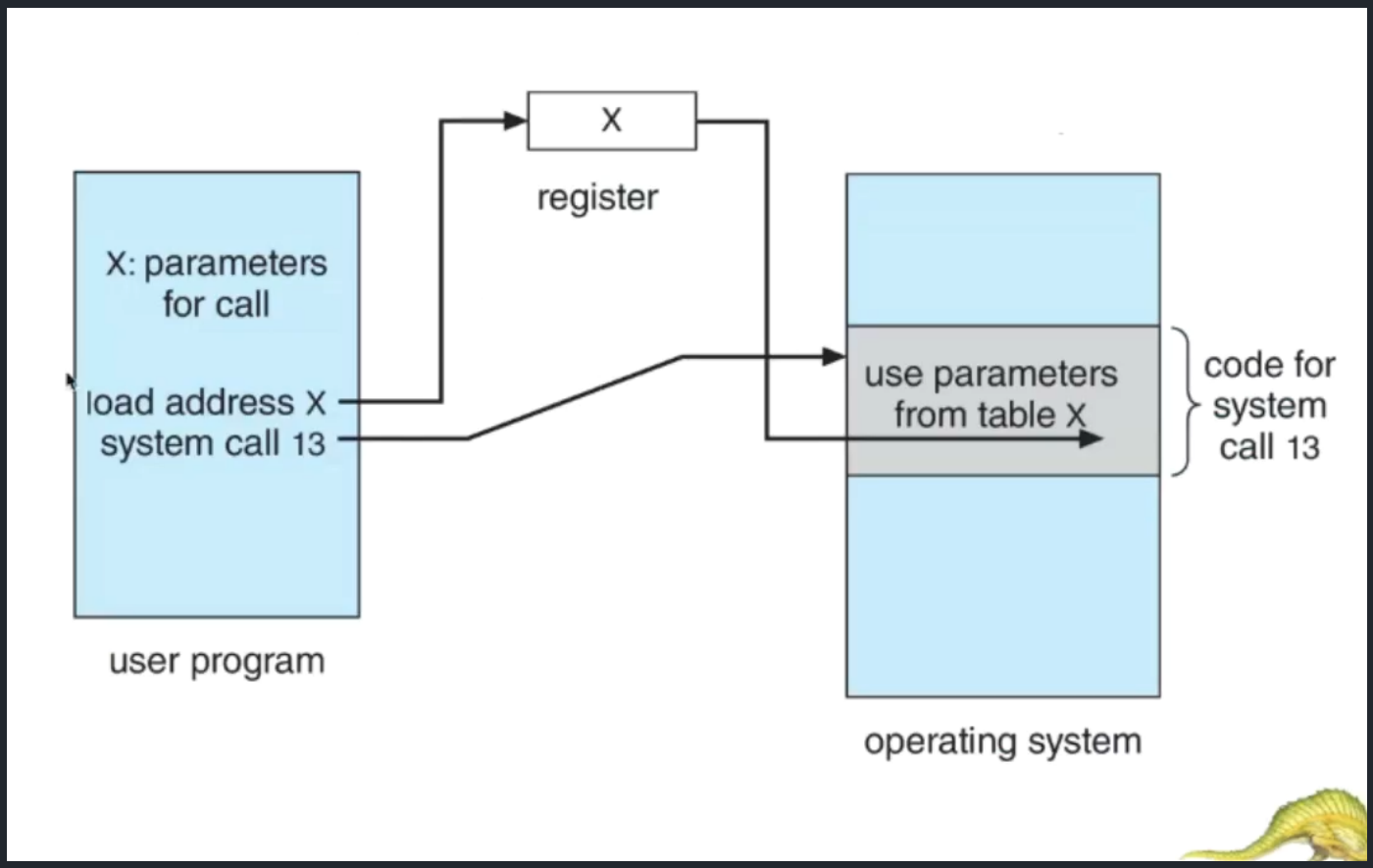
Il metodo più semplice è quello di passare i parametri all'interno dei registri del processore: basta caricare all'interno di un registro il **numero della syscall** ed eventuali parametri, ad esempio l'indirizzo dov'è salvata una stringa, e con questo sistema si riesce agevolmente ad utilizzare le syscalls quando abbiamo a disposizione **molti registri**, e non ci sono molti parametri.

Un altro modo è quello di usare una **tabella in memoria**, e quindi fornire l'indirizzo di memoria dove sono posizionati tutti i parametri, ed è un metodo abbondantemente usato nei sistemi UNIX.

Un altro modo ancora è quello di salvare i parametri sullo **stack**, effettuiamo quindi un **push sullo stack** dei parametri ed il sistema operativo andrà a prelevarli.

In ogni caso il metodo dello **stack o della tabella** non impone limiti sul numero di parametri da usare.

Passare i parametri con una tabella



Nel registro viene caricato solo l'indirizzo della tabella, dopodichè la syscall farà sì che il SO prenda l'indirizzo della tabella dal registro, ed accederà alla tabella per prelevare i parametri.

🏁 lez 2 00:51

Servizi di Sistema

Esiste una grande varietà di programmi di sistema che permettono di gestire i file, come copia, Rename, stampa ecc.
Per quanto riguarda le informazioni di **stato**, esistono dei servizi che permettono di accedervi, e che quindi permetto di accedere ad informazioni come data, ora ecc.

Questi programmi di sistema **stampano** l'output del sistema sul terminale o altri Devices, per permettere all'utente di consultarli.

Registry

Alcuni sistemi implementano un **registry**, tipico dei sistemi Windows odierni.

Sistemi Windows

L'idea era quella di usare un deposito **unico** in cui conservare tutte le informazioni di configurazione di sistema, in modo da avere sempre pronte queste informazioni.
Il problema fuoriesce nel momento in cui vogliamo disinstallare un programma che utilizza il registry, perché ci

Sistemi Unix

Nei sistemi Unix, invece di avere un unico file dove vengono conservate tutte le informazioni, esiste una directory apposita, chiamata **etc**, in cui, racchiuse all'interno di diverse directory, sono presenti tutte le informazioni di sistema.
Tutti questi file sono sotto forma di **file di testo**, quindi completamente analizzabili dall'utente.

Supporto per linguaggi di programmazione

Alcuni sistemi operativi forniscono degli strumenti per sviluppatori, come **compilatori, assemblato, sistemi di debuggano ecc.**

Servizi di background

Oltre ai programmi che vengono lanciati dall'utente, qualsiasi SO lancia dei **servizi in background**, ovvero dei programmi che restano in attesa di essere utilizzati.
Nel mondo **UNIX** questo tipo di programma è chiamato **Deamon**. Nel mondo **Windows** invece sono chiamati **Services**.

Linkers e Loaders

- Un **Linker** è un software che **unisce** tutti i file oggetto a disposizione, in maniera da ottenere un **eseguibile binario completo** .
- Un **Loader** è in grado di prelevare un eseguibile dal disco e gli assegna un indirizzo di memoria per essere eseguito.

Bonus

Il comando **MAN** su UNIX è suddiviso in **sezioni**. La sezione 1 è quella destinata ai **programmi di sistema**, e non alle syscalls. Per cui se si volesse accedere alla syscall dovrebbe digitare: `man -S 2 open`.

Inoltre, per "uccidere" un processo su unix basta digitare `ps`, controllare il il **PID** del processo ed usarlo per chiudere il processo con `kill -9 2793`.

Perché le applicazioni sono tipiche del sistema operativo?

Posso prendere un determinato programma scritto per un determinato processore, su un determinato sistema operativo, ed eseguirlo sulla stessa macchina ma con SO diverso?
Ovviamente la risposta è NO.

Posso trasferire una **libreria**, in formato già compilato, ed utilizzarla sulla stessa macchina con SO diverso? Anche in questo caso la risposta è NO.

- **Compatibilità del formato dell'eseguibile** : il formato con cui viene creato il binario, è diverso da SO a SO.
- **System calls diverse** : anche le System calls sono diverse.
- **Parametri delle Syscalls diversi** : anche i parametri, da passare con le chiamate di sistema, variano da SO a SO.

Non è possibile, quindi, far eseguire lo stesso programma, anche sulla stessa macchina, con due Sistemi operativi diversi.

Come risolvere il problema

Emulazione

É possibile eseguire, ad esempio, un programma windows su sistema Linux. Si può fare ciò attraverso l'uso degli **emulatori**.
Un **Emulatore** (ai tempi veniva usato Wine-Windows emulator) che al momento dell'esecuzione del programma, intercettava tutte le syscalls del programma, trasformandole in syscalls di linux.
Il problema è che microsoft (che aveva scritto il programma da emulare) aveva usato delle syscalls **non note**, e di conseguenza non era possibile effettuare la conversione delle syscalls.

Virtualizzazione

Il problema viene in realtà risolto con l'emulazione, ovvero eseguendo, all'interno di Linux, una sessione di Windows, dove verrà eseguito il programma nitidamente.

Come si progetta un Sistema Operativo?

Esistono dei criteri di progetto che rendono il processo "semplice"? Risposta veloce: No.

Fondamentalmente la struttura interna di un SO può variare, ed anche di molto, da SO a SO. Anche nello stesso mondo Unix, la struttura interna cambia moltissimo.

Obbiettivi quando si sviluppa un SO

Quando si sviluppa un sistema, ci sono due obbiettivi principali:

- Obbiettivi di Utente : Il sistema dovrebbe essere di facile utilizzo, facile da comprendere, affidabile sicuro e veloce.
- Obbiettivi di Sistema : Il sistema dovrebbe essere semplice da progettare, implementare e sopratutto **efficiente** .

Come si nota, gli obbiettivi non coincidono completamente, infatti l'utente finale è poco interessato a sapere che il SO sia efficiente, ma il suo interesse è quello che esso sia veloce ed affidabile.

Principio più importante

Nel progetto di una qualsiasi cosa, bisognerebbe distinguere "ciò che voglio fare" e "qual è il modo per farlo".
Dovrei ragionare sulla **politica**, ovvero il risultato che voglio ottenere, senza farmi influenzare fin dall'inizio dalla modalità con cui implementerò.

In poche parole, questo concetto ci permette di **progettare** il nostro obbiettivo, senza farci influenzare dalle problematiche del "come realizzarlo". In questo modo, gli obbiettivi, e quindi le funzioni, rimarranno gli stessi nel tempo, anche se l'implementazione potrà cambiare.

1. Decidere cosa voglio che il mio software faccia
2. Impazzire, solo dopo, per implementarlo

🏁 00:34 05-06

Implementazione (dei sistemi operativi)

I primi SO venivano implementati rigorosamente in **assembly**, dopodiché sono stati utilizzati dei linguaggi come l'**Algol**, che permettevano di scrivere SO.

Infine, usato ancora oggi, venne impiegato il **C, C++**. In effetti, il C venne proprio creato per lo sviluppo di sistemi operativi, per via del bisogno di avere un linguaggio che fosse a basso livello, ma che allo stesso momento permettesse di eseguire operazioni complesse.

Mix di Linguaggi di programmazione

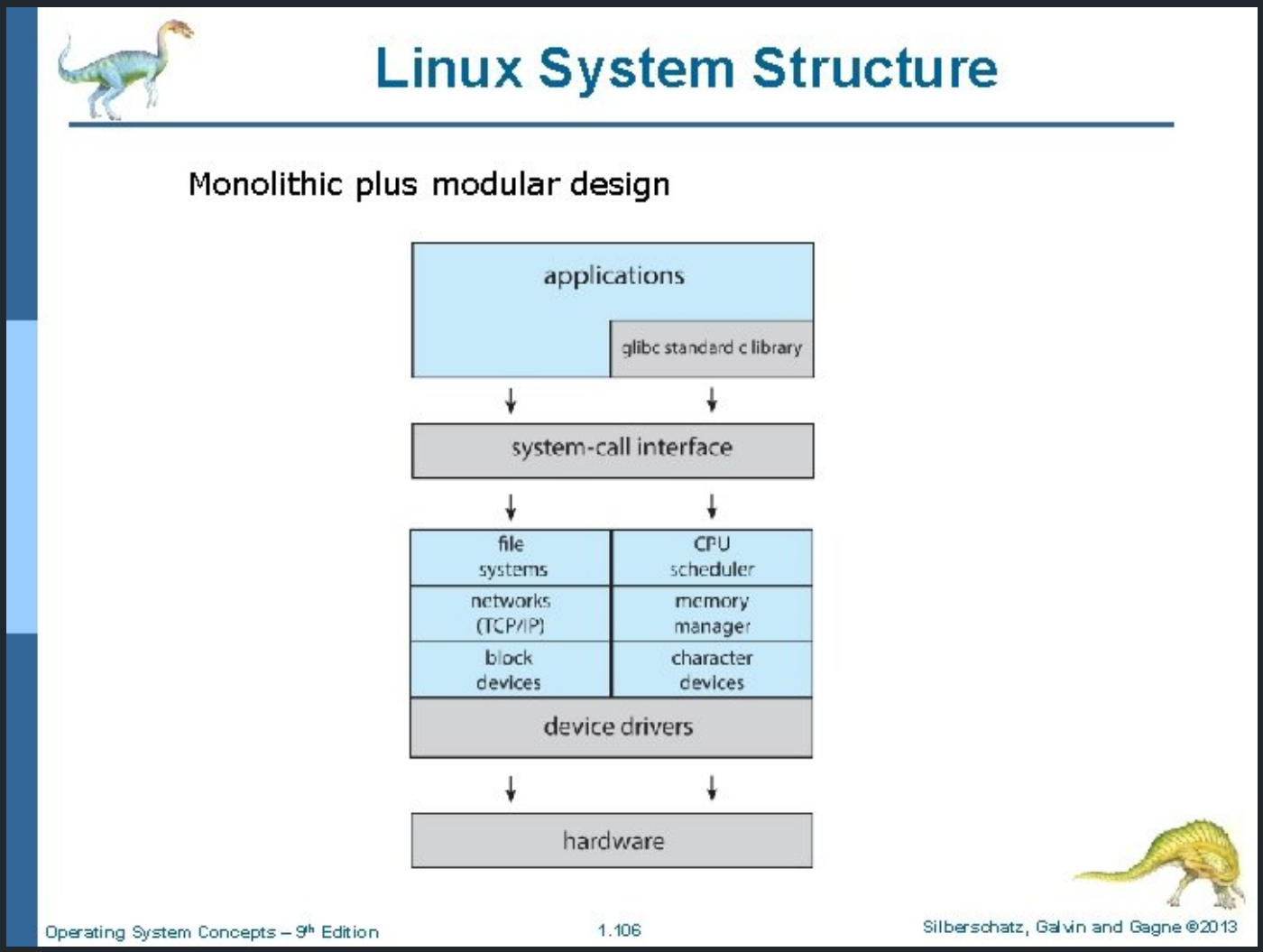
I SO, in verità, sono composti da un mix di linguaggi. Infatti i livelli più bassi del SO vengono scritti in linguaggio macchina, e quindi in **assembly**, per via della manipolazione dei registri, il corpo principale viene scritto in **C**, per le motivazioni già spiegate precedentemente, ed infine i programmi di sistema possono essere scritti, ancora una volta, in vari linguaggi, come il C, C++, linguaggi di scripting come **Python** e **Shell**.

Come è strutturato un Sistema Operativo

Fondamentalmente i sistemi più antichi, programmati in modo tale da occupare una piccolissima quantità di memoria, avevano una struttura molto semplice; era il caso di **MS-DOS**.
Altri sistemi sono invece più complessi, come **UNIX**.

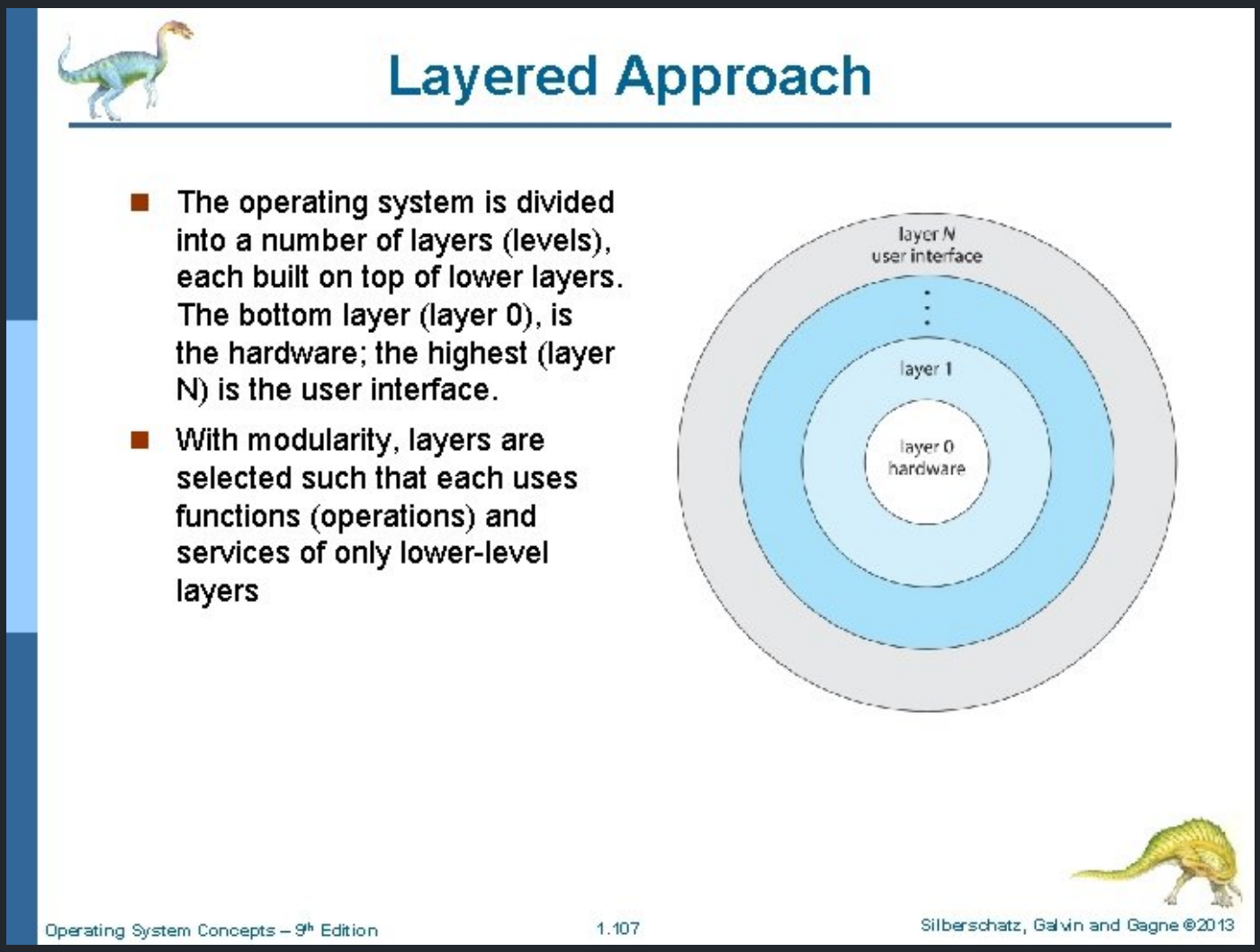
Struttura Monolitica - UNIX originale

In questi sistemi è presente una struttura di vari moduli software, ma il **kernel** è **unico**.
Ciò significa che non solo il kernel è un'unico pezzo, ma che esso viene eseguito in modalità **supervisore**.



Approccio a "strati"

Negli anni è stato suggerito di strutturare il SO a **livelli**:



Questa è una struttura molto "comoda" sulla carta, ma nella pratica non funziona molto bene, perchè tra un'operazione e l'altra c'è il bisogno di effettuare dei "salti" tra uno strato e l'altro.

Per questo motivo non esiste un SO che adotti questa struttura.

Microkernel

Il **microkernel** è "l'avversario" della struttura monolitica. Questo perchè nel kernel resta solo il minimo indispensabile come:

- Scheduling della CPU: perché serve all'esecuzione di processi
- Gestione della memoria
- Comunicazione tra processi

Tutto il resto, vengono eseguiti come **Programmi di sistema**, in **User mode**.
L'idea è la seguente: "cerchiamo di levare quanta più roba possibile dal kernel, facendolo eseguire come programma di sistema".
Il vantaggio ottenuto, oltre al fatto di avere un kernel molto compatto, è sopratutto il fatto che molte attività vengono eseguite in modalità **utente**; questo significa che se è presente un guasto o malfunzionamento, non è possibile fare "danni". In questi casi basta semplicemente rilanciare il processo, ed il sistema continua a funzionare senza problemi.

Con questo schema, come già detto, nel kernel è posto solo il minimo indispensabile, il resto invece viene fatto mediante l'invio di "messaggi", ovvero i programmi di sistema comunicano tra loro grazie **all'interprocess communication**, situato nel kernel.

Il primo sistema strutturato su **microkernel** era il sistema **Mach**, che nella sua prima versione era molto lento. Questo perchè scambiare dei messaggi non è veloce come cambiare una funzione all'interno del kernel stesso, e quindi, per i processori di una volta, avere un sistema a Microkernel, voleva dire rallentare l'esecuzione.

Fun fact

Linux è stato strutturato in modo "monolitico", nonostante inizialmente si era insistito per utilizzare un approccio a microkernel, che però non venne ben visto dai primi sviluppatori.
Anche Windows è basato su una struttura Monolitica.

La differenza principale tra i due sistemi è che in linux l'interfaccia grafica gira in modalità **utente**, mentre in windows in modalità **supervisore**.

Quale OS usa il microkernel?

L'unico SO moderno (ampiamente utilizzato) che è basato su una struttura a microkernel, è Mac OS, basato sul sistema Mach.