09 Maggio 2023

ARCHITETTURE DI CALCOLO LEZIONE 15

System call, sistema operativo monolitico e a strati

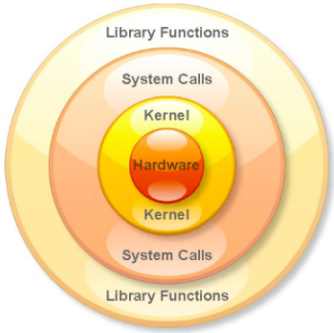
L’interfaccia utente esiste in due forme: la shell e l’interfaccia grafica.
Mentre la shell è sempre esistita (sin dagli anni ‘60), l'interfaccia grafica nasce negli anni ‘80. Una pietra miliare in tal senso è l'interfaccia GUI di Apple Macintosh. Essa si basa sulle icone.
Nel mondo dei software liberi (New Linux) ci sono due principali ambienti:
- KDE
- GNOM

I sistemi operativi in cui l'interfaccia utente è indispensabile è indubbiamente lo smartphone per le seguenti ragioni:
- 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.

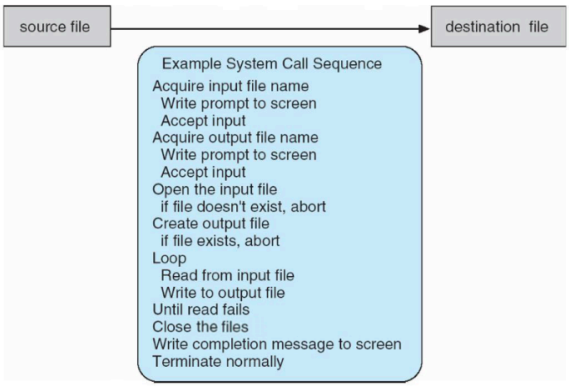
SYSTEM CALL

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++). Come tutte le funzioni devono avere un nome, un tipo di ritorno e dei parametri ricevuti. Normalmente, vengono richiamati degli 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).
Le API generalmente sono librerie di più alto livello rispetto alle system call.
Uno standard mondiale è POSIX: Portable Operating System Interface per UNIX. Tale formato permette di definire in modo uniforme dei comandi al di là dell'interfaccia usata (API). Oggi il programmatore presenta due alternative nella costruzione di un sistema:
- POSIX, standardizzato;
- Non POSIX, che non soddisfa gli standard mondiali.

Le system call di Windows, Mac e Linux sono di tipo POSIX.
Nota: C e C++ sono linguaggi di alto livello perché lontani dal linguaggio assembly ma allo stesso tempo è meno alto, ad esempio, di Java o Python; ciò permette di creare sistemi più dettagliati nelle implementazioni di basso livello (scheduler, gestione dei file, ecc.).

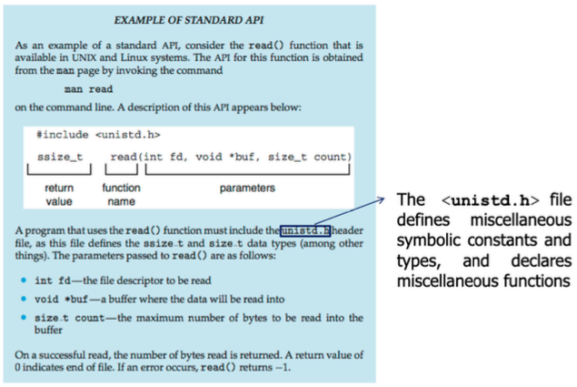


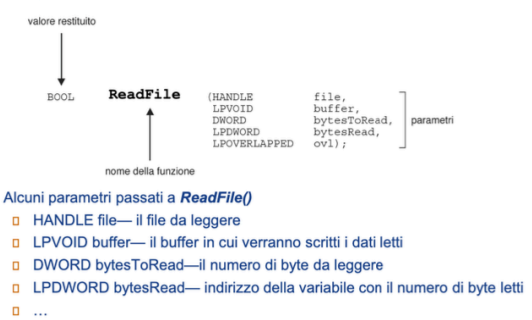
Le system call offerte da un sistema operativo riguardano le diverse funzionalità del sistema, dividendosi in 5 categorie:
•Controllo dei processi;
•Gestione dei file;
•Gestione dei dispositivi I/O;
•Informazioni sul sistema;
•Comunicazioni.



Una cosa che spesso sfugge quando si pensa al concetto di system call è la quantità di chiamate che vengono eseguite durante un'operazione. Ad esempio, vediamo a sinistra la sequenza di chiamate di sistema necessarie per realizzare la copia di un file in un altro; le system call sono migliaia al secondo.
La system call risulta, così, essere l'operazione più ripetuta dal SO e, poiché ogni system call genera un interrupt, si dice che tale SO e interrupt driven.

Librerie API

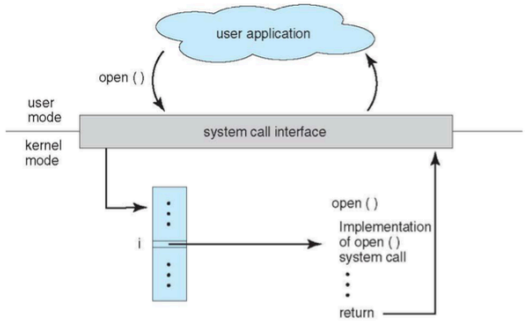




In figura un esempio di standard del C come API.
Immaginiamo esista una libreria che si trova in unistd.h.
Nel seguente codice:
•ssize_t è il valore da restituire;
•read è la funzione di lettura;
•int "fd" chiede il nome del file;
•void * buf è un buffer dentro cui vengono letti i dati;
•size_t "count" richiede il range di lettura.
Ad esempio, read(int 123,void [buffer]; size_t 100).
Generalmente questa operazione va in loop (ad esempio, fin tanto size_t count >0).
Diverso è il formato di API di Windows, illustrato nella slide a destra. Al di là del formato, entrambe le tipologie di API presentano delle funzioni di lettura di un file.

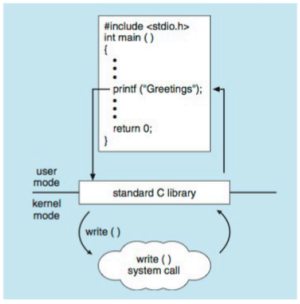
Run-time support system

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



Nella maggior parte dei linguaggi di programmazione il sistema di supporto all'esecuzione (run-time support system), un insieme di funzioni strutturate in librerie incluse nel compilatore, fornisce un'interfaccia alle chiamate di sistema che collega il linguaggio utente alle system call rese disponibili dal sistema operativo. L'interfaccia intercetta le chiamate a funzioni nella API, e invoca le relative system call. Di solito, ogni chiamata di sistema è codificata da un numero.

- 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

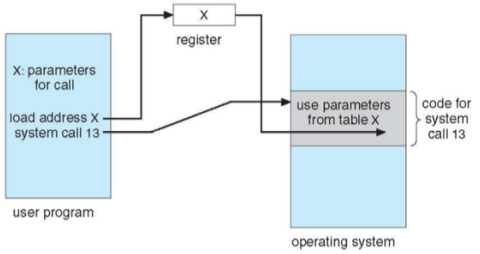
L'interfaccia alle chiamate di sistema mantiene una tabella delle chiamate e invoca di volta in volta la chiamata richiesta, che risiede nel kernel del sistema, restituendo al chiamante lo stato della chiamata e i valori di ritorno.
Ad esempio, una volta avviata la open, si passa da modalità utente a kernel mode; parte la system call di apertura di un file; viene ritornato il file e si ritorna alla modalità utente.

Passaggio di parametri alle system call

Spesso l'informazione necessaria alla chiamata di sistema 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:

- Passaggio di parametri nei registri (il più semplice). Questo pone dei limiti perché in alcuni casi il numero di parametri da passare è maggiore del numero dei registri disponibili.
- Memorizzazione dei parametri in un blocco in memoria e passaggio dell'indirizzo del blocco come parametro in registro. Approccio seguito da Linux e Solaris.
- Push dei parametri nello stack da parte del programma; il SO recupera i parametri con un pop.

Gli ultimi due metodi, meno intuitivi del primo, hanno il vantaggio di non porre limiti al numero ed alla lunghezza dei parametri passati.
Importante: le system call hanno dei codici che rappresentano le varie funzioni (che vengono mantenute in assembly e nel linguaggio macchina sottostante).



In figura osserviamo il percorso dei parametri dall'input utente al kernel tramite system call.

	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() InitlializeSecurityDescriptor() SetSecurityDescriptorGroup()	chmod() umask() chown()

In tabella, invece, vengono elencate le funzioni di system call, solo a titolo illustrativo (non li chiederà all'esame). Tra Unix e windows la nomenclatura è diversa ma la funzione eseguita è la stessa; per esempio, fork() è la funzione che crea i processi in unix, in windows viene chiamata CreateProcess().

PROGRAMMA DI SISTEMA

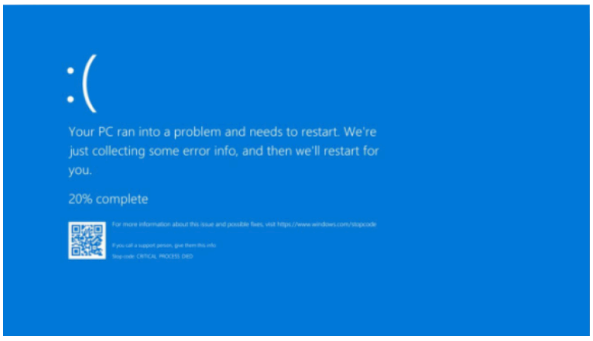
I programmi di sistema sono programmi non kernel (non sono sempre attivi in modalità kernel) che possono privilegiare della possibilità di avere accesso allo sviluppo ed esecuzione delle applicazioni. Questo è stato fatto innanzitutto per ridurre le probabilità di creare errori nel sistema. Ovviamente tali programmi (come la shell) fanno uso delle system call.
Essi si occupano di:
- Gestione die file;
- Informazione di stato;
- Supporto ai linguaggi di programmazione;
- Caricamento ed esecuzione dei programmi;
- Comunicazioni.

I primi programmi di sistema prevedevano la programmazione in linguaggio assembly per ragioni di efficienza e di accesso alle risorse hardware. Tale linguaggio è:
- Maggiormente efficiente;
- Più verboso e complesso;
- Specifico per tipologia di macchina (non traducibile, va riscritto).

Oggi, invece, vengono implementati tramite linguaggi di alto livello. Il codice scritto in questi linguaggi è:
- Più veloce da sviluppare;
- Più compatto;
- È facile da comprendere e da correggere.

DEBUGGING DEI SISTEMI OPERATIVI

A volte può capitare di imbattersi in questa schermata:

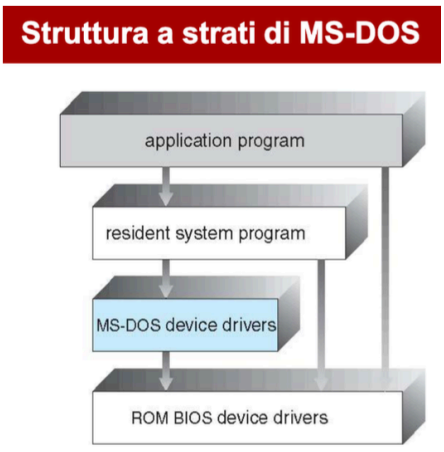


Ciò indica che per qualche ragione (memoria piena, ecc.) il SO va in errore, non previsto dal programmatore. L'attività atta alla ricerca e risoluzione degli errori o bug prende il nome di debugging. Quando si genera un errore, i SO generano dei file di log che contengono informazioni che vengono raccolte in delle banche dati della casa produttrice in modo che il programmatore possa essere a conoscenza di tale bug.

TIPI DI SO

In generale di distinguono due tipologie di sistemi: monolitico e stratificato.

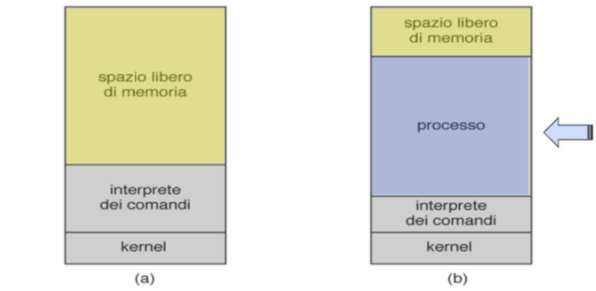
Sistema operativo monolitico



Il sistema monolitico è il tipo storico, il quale prevede che tutte le funzioni di gestione delle risorse siano realizzate nel kernel. Tale scelta ha facilitato la programmazione (unico programma principale) e prevede che l'intero SO tenda a identificarsi con il nucleo stesso. Tuttavia, un bug in una delle varie funzioni può bloccare l'intero sistema. Un esempio è il sistema MS-DOS, ideato per il processore INTEL 8088 che era basato sulla modalità dual bit
mode. Ciò portava ad una vulnerabilità agli errori ed ai virus dei programmi utente che avevano facile così accesso alla modalità Kernel.

Resident system program = SO
I driver della ROM sono delle funzioni per l'accesso alle periferiche di I/O che sono incorporta nel BIOS (Basic Input Output System) della macchina. Le funzioni di accesso ai dispositivi I/O sono quindi integrate nel kernel e il BIOS contiene delle routine di accesso all'I/O.
Questa architettura detta struttura stratificata trasducente (attraversabile) prevedeva limiti sia hardware che implementativi (minore protezione da bug e virus dovuto all'accesso diretto al

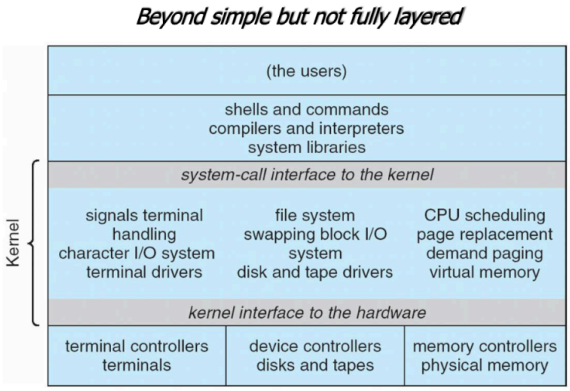
BIOS da qualunque altro “strato”).



(a) All’ avviamento del sistema (b) Durante l’ esecuzione di un programma

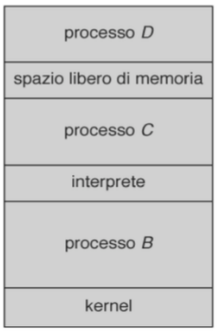
Dal punto di vista della memoria (massimo 640 Kb), si osserva dall'immagine che una volta avviato un processo gran parte della memoria risultava occupato e per questo l'esecuzione contemporanea di più programmi non è possibile.
Lo spazio libero potrebbe essere occupato dallo stesso processo a mano a man che viene utilizzato e, in un eventuale loop, potrebbe portare al blocco del processo stesso.
L'interprete dei comandi, quando si avvia un processo, riduce la memoria che occupa in quanto va in background, al contrasto con la fase in foreground all'avviamento del sistema.
Chi programmava queste macchine, che era a conoscenza di tali limiti, gestiva abilmente la memoria; oggi tale problema non si presenta più al programmatore che ha a disposizione una memoria di grande dimensione.

Struttura a strati



Il primo SO a strati è quello delle macchine UNIX che prevedevano due parti separate:
•I programmi di sistema;
•Il kernel.
Questo portò all'identificazione del sistema a layer dove troviamo il kernel con le varie funzioni e superiormente la shell con l'interfaccia utente.

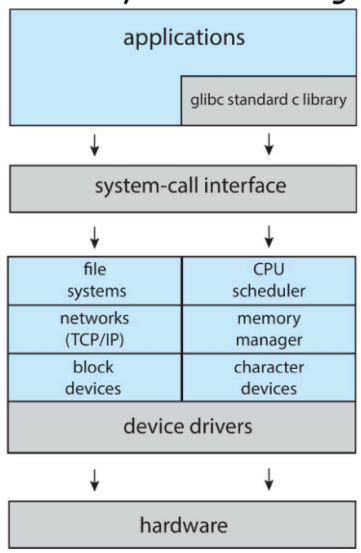
In tali macchine la memoria si presenta come in figura:



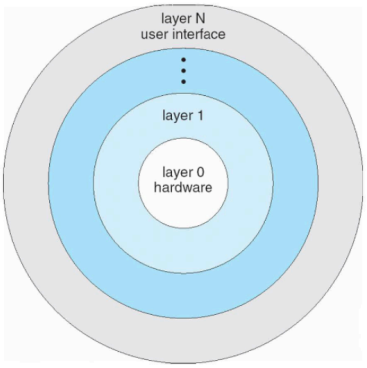
Struttura a strati di LINUX

Il kernel di Linux è un kernel 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 run-time.
Il vantaggio di tale approccio è quello di far girare il SO su macchine meno costose (accessibili anche ad uno studente) e con meno memoria disponibile.

Monolithic plus modular design

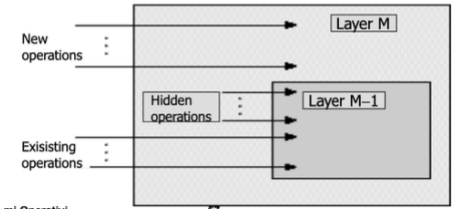


Intorno agli anni 90 c'è chi ha idealizzato un vero e proprio concetto di approccio stratificato a livelli concentrici.
Il numero dei livelli dipende sempre dallo strato precedente: il livello N-esimo è quello dell'interfaccia utente mentre il livello 0 è l'hardware.
L'architettura degli strati è tale 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.



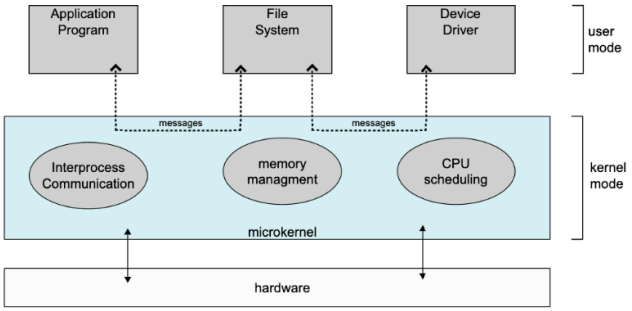
L'approccio stratificato presenta sia vantaggi che svantaggi.
Vantaggi:
•Semplicità di realizzazione e messa a punto (che viene attuata strato per strato).
Svantaggi:
•Difficoltà nella definizione appropriata dei diversi strati, poiché ogni strato può sfruttare esclusivamente le funzionalità degli strati su cui poggia.
•Tempi lunghi di attraversamento degli strati (passaggio di dati) per portare a termine l'esecuzione di una system call.

Attualmente nelle reti si usano i sistemi TCP/IP



Struttura dei sistemi microkernel

Il microkernel è un kernel minimale concettualmente opposto ad un kernel monolitico in quanto contiene la maggior parte delle funzioni nello spazio utente. Esempi sono le prime versioni di windows NT, Mach, GNU Hurd.



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.