

Struttura monolitica

I primi sistemi operativi avevano una struttura **monolitica**, ovvero tutte le code del software che implementava il SO viene inserite in una memoria e funziona nello spazio del Kernel. (MS-DOS, UNIX)

In MS-DOS non ci è una netta separazione tra i livelli di funzionalità e le applicazioni accedono direttamente all'elenco. Questo grado di libertà rende vulnerabile il sistema.

Anche UNIX era originariamente piuttosto **strutturato**. Esso si componeva di due parti: il Kernel e i programmi di utente. Il Kernel comprendeva il file system, lo scheduling della CPU, gestione dei terminali ecc... si tratta quindi di molte funzionalità gestite da un solo livello.

Non si ha la multiplattformazione, infatti un solo processo può essere in esecuzione in un istante di tempo. Ogni componente può richiamare tutti gli altri.

Metodo stratificato

grazie ad un hardware appropriato i SO possono avere moduli in moduli più piccoli che gestiscono diverse funzionalità.

La struttura del SO viene organizzata a livelli d'strati. Ogni strato offre funzionalità sfruttando i servizi offerti dall'anello inferiore, viene sfruttata molto l'astrazione. In questa organizzazione ogni livello c'è una componente software che implementa una determinata funzionalità.

Alla base dello stack abbiamo l'hardware (lv. 0) mentre il ultimo livello è l'interfaccia utente.

Questo modello implica che i livelli devono essere organizzati in modo tale da sfruttare tutte e solo i servizi offerti dal livello sovrstante, al quale ha accesso ma introducendo un importante overhead.

MicroKernel

Se modellati a statico introduce un overhead, oltre a dare una certa rigidità al codice. Un progetto più moderno è l'idea del **microKernel**, ovvero un SO che contiene un microKernel composto da varie componenti strettamente indipendibili e tutte le altre componenti sono fuori dal Kernel. L'idea è di riducendo il Kernel in una componente più piccola hanno i seguenti vantaggi:

- 1) un codice ridotto ha un numero di bug minore;
- 2) tutte le altre componenti esterne al Kernel (che invece girano in modalità user) vengono collocate in processi esterni e per tanto eventuali bug in questi driver non toccheranno le funzionalità del microKernel e nemmeno quelle di altri processi che risulteranno isolati tra loro.

Tutto questo deve essere contenuto nel microKernel e una componente solida e ben implementata e non deve essere esterna.

=> Comunicazione di messaggi

I vari processi comunicano tra loro e col microkernel attraverso questo funzionalmente a messaggi implementato dal microkernel. Un messaggio parte da una componente e tramite il so, si ha la gestione dei messaggi che fanno per dirsi bus.

La gestione dei dischi e del filesystem è esterna al microkernel e viene isolata in processi esterni. In caso di errore il processo viene arrestato e fatto riflettere, quindi si ha un auto-correttore del sistema.

Struttura a moduli (implementata da Linux)

L'idea è di individuare funzionalità essenziali da inserire in un nucleo incicabile ed estornerne altre non essenziali in dei moduli, ma la differenza col microkernel è che anche i moduli girano in Kernel mode. L'aspetto negativo è che essendo in Kernel mode si

ha la possibilità di avocare problemi al SO (MSDOS).
L'oggetto positivo è che, essendo già all'interno del Kernel non
è necessaria la scambiare messaggi e non si introduce
overhead come nel modello a strati, quindi si ha
un netto miglioramento delle performance

⇒ La modularità è implementata a run-time, a tempo di
esecuzione (memoria del sistema) si decide cosa inserire
all'interno del Kernel e quindi quale modulo (funzionalità) si
aggiunge a funzionare (modulo caricato dinamicamente)

Macchina virtuale (VM)

Si estende il concetto di astrazione, si crea l'astrazione
di una CPU che consente di condurre l'area di memoria
con la macchina in cui viene eseguita la VM.

L'idea è di avere un sistema per utilizzare intere
macchine astratte che ruotano in esecuzione sul SO della
macchina fisica.

Si avitano altri sistemi operativi all'interno di un SO reale con lo stesso livello di efficienza del SO reale ad eccezione delle gestioni operazioni di I/O.

Questo è essenziale per lo sviluppo di software multivettoriale utilizzando diversi browser e diversi SO. Altri vantaggi

sono derivanti dal fatto che nona della sicurezza, in quanto usare servizi in una VM garantisce che potenziale codice malevolo non agisca sulla macchina principale.

Il codice eseguito dalla VM viene in realtà eseguito

dalla CPU fisica, permettendo la stessa efficienza della macchina fisica. Quando una VM esegue una system call

entra in gioco l'hypervisor che si occupa di

simulare il cambiamento della modalità utente (in cui

fissa la VM) alla Kernel quale la cui dovere far fare

con la TRAP) ma di fatto non avviene nessun passaggio

a carico di un rallentamento della CPU: infatti molte

interazioni I/O rallentano il rendimento e le prestazioni.

Affidare hypervisor di due tipi:

Tip. 1 \Rightarrow software eseguito sul hardware che offre il servizio di estensione di multiplica' machine.

Viene eseguito in Kernel mode perché deve gestire il hardware.

Tip. 2 \Rightarrow software che viene eseguito e gestito dal caricatore del SO ospitante. Viene eseguito in User mode. L'hypervisor si occupa di simulare gli effetti del parafisico dalla user mode alla Kernel mode VIRTUALIZZATA. In questo modo la macchina non ha accesso al Kernel ma è l'hypervisor che si occupa di gestire l'accesso alle risorse.

La simulazione si riferisce ad una tecnica molto diversa che ha un obiettivo simile: nella virtualizzazione sono costruiti ad avere un match tra la macchina fisica e quella virtuale dato che il codice della VM viene eseguito dalla reale CPU. Mentre nell'emulazione o simulazione viene

simula l'effetto delle istruzioni sul registro
delle macchine finca, l'interpretazione (non esecuzione)
delle istruzioni edura un importante rallentamento nelle
funzioni: