INTERFACCIA E STRUTTURA DEL KERNEL

Duplice modalità di funzionamento

Permette al sistema operativo di proteggere sé stesso dai programmi in esecuzione. La CPU può funzionare in:

- **Modalità utente** (user mode);
- **Modalità di sistema** (kernel mode).

Per passare da una modalità ad un'altra viene impostato un opportuno bit di modalità. Alcune istruzioni del processore sono privilegiate, ossia eseguibili solo in modalità di sistema. Inoltre, in user mode la CPU non può accedere alla memoria del kernel.

Esempio: istruzione macchina per comunicare con la tastiera e schermo e non possono essere usate quando siamo in modalità utente. In questo modo permettiamo solo al kernel di gestire queste operazioni quando è in modalità kernel.

Implementazione delle chiamate di sistema

Quando parliamo di chiamata di sistema, è importante proteggere il codice e memoria del kernel dai programmi eseguiti dagli utenti poiché non dovrebbero essere in grado di sovrascrivere il codice del kernel o utilizzare la memoria del kernel. Questo viene realizzato in HW in modo tale che possa essere usato in due modalità.

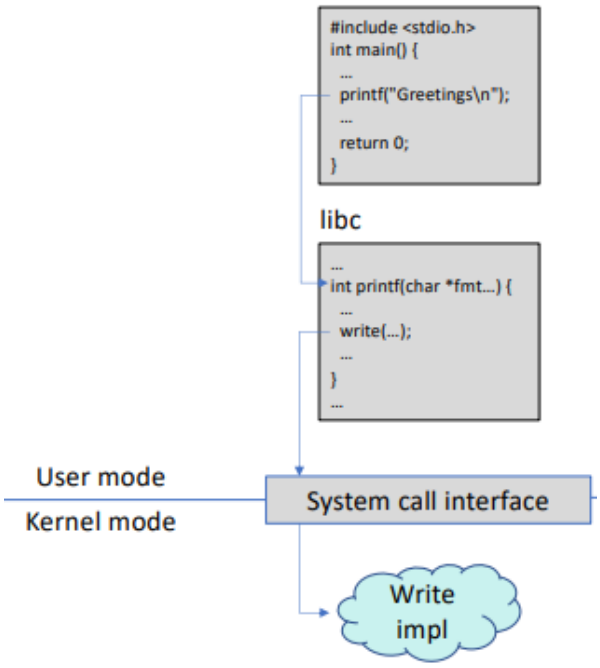
Una chiamata di sistema non è semplice da implementare come una normale chiamata di funzione, perché occorre effettuare una transizione da modalità utente a modalità di sistema. Viene impostato un bit di modalità presente nei registri del processore.
Prima vengono preparati i parametri necessari:

- Un numero che identifica quale chiamata di sistema va effettuata;
- Tutti i parametri necessari alla specifica chiamata di sistema.

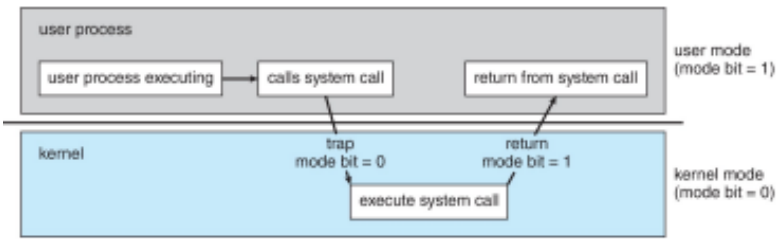
Quindi viene invocata un'opportuna istruzione macchina che genera un'eccezione software; essa fa passare la CPU dalla modalità utente alla modalità di sistema e questo viene fatto utilizzando un approccio molto simile a quello degli interrupt (eccezioni software). Trasferisce il controllo ad una subroutine (punto del kernel) ad un determinato indirizzo di memoria. All'avvio del computer, il processore è in modalità di sistema poiché il kernel sta caricando e quindi deve essere in grado di configurare la sua macchina a suo piacimento. C'è un punto del kernel che è come se fosse un punto di ingresso per tutte le chiamate di sistema. Questo punto viene configurato come punto in cui saltare quando

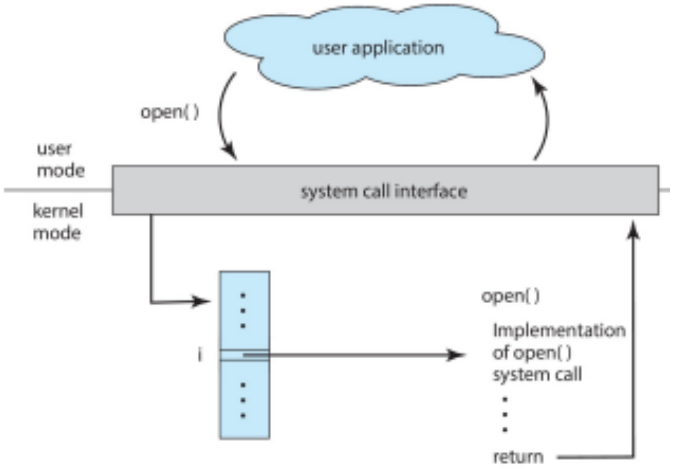
viene chiamata questa eccezione software che viene messa all'interno del codice delle librerie quando viene chiamata la System Call Interface.
La subroutine (system call interface) legge il numero identificativo della chiamata di sistema, effettua un lookup da una tabella interna dell'indirizzo della 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.

Esempio: Quando la libc chiama write(…) che è una chiamata di sistema, all'interno del codice assembler della write viene chiamata un'istruzione di eccezione software che fa passare in modalità di sistema e che fa saltare all'interfaccia della system call.



Il salto poi va in una zona del SO del kernel in cui c'è una tabella con tanti indirizzi di tutte le varie chiamate di sistema e in funzione del numero si salta alla chiamata di sistema giusta. Dopodichè, saltando alle varie syscall, è poi l'implementazione della syscall che recupera gli altri parametri e dopo implementa la syscall. Alla fine, c'è una return che oltre a ritornare al chiamante, deve ritornare dalla modalità kernel alla modalità utente. Dopo le API, si torna al codice dell'utente.





Passaggio dei parametri alle chiamate di sistema

Prima del passaggio dei parametri, si devono effettuare molte operazioni come passare dalla modalità utente alla modalità kernel e non è un processo velocissimo. Si cercano tutti i modi per il miglior compromesso tra velocità di esecuzione della chiamata di sistema e flessibilità dell'implementazione. Si va in uno spettro di soluzioni tra basso livello ma veloce a alto livello ma più lento.
Dal momento che l'invocazione delle chiamate di sistema passa per un'eccezione software, il passaggio di parametri è più complesso rispetto a quello di una normale chiamata di procedura:

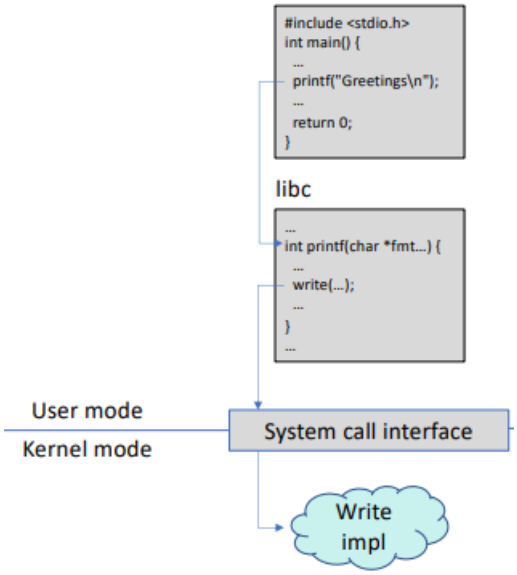
1. Metodo più semplice: passare i parametri nei registri del processore
   Vantaggio: rapido (non bisogna accedere alla memoria ma ai registri)
   Svantaggio: utile solo per pochi parametri i cui tipi di dati hanno dimensione limitata e fisse (perché i registri sono pochi e dimensioni piccoli e fissa).

2. Altro metodo: passo in uno dei registri un indirizzo di memoria ad un blocco della memoria centrale nel quale sono memorizzati i parametri.
   Usato da Linux in combinazione con il primo metodo: passa i parametri nei registri del processore, ma se ci sono parametri di struttura complicata che non riescono ad essere messi nei registri, vengono messi in memoria. In registro ci sarà il puntatore all'indirizzo di memoria dove è messo il parametro.

3. Altro metodo: faccio push dei parametri sullo stack. Tutti i parametri sono in stack ed è simile ad una normale chiamata di procedura.
   Vantaggio: flessibile e utilizzato già nei linguaggi di programmazione.
   Svantaggio: lento e macchinoso. Ogni volta si deve passare per la memoria centrale.

Necessità del doppio stack

L'applicazione quando chiamata una funzione, ha uno stack del processo che pusha i parametri per quella funzione. Quando viene chiamata la syscall e il codice del kernel inizia ad eseguire, si dice che il codice del kernel è eseguito nel contesto del processo che è stato chiamato.



Esempio: Quando il write esegue, qualsiasi cosa che esegue avviene nell'ambiente che è costituito dal processo che ha chiamato. Se il write scrive su un file aperto, lo farà su i file aperti del processo che lo ha chiamato. Una volta che la write viene chiamato da un altro processo, la write andrà a scrivere su i file aperti da quell'altro processo. Supponendo che la write in tutti i due casi gli passi un file identifier n.4, per il processo 1 può corrispondere ad un file diverso dal processo 2 perché siamo in due ambienti diversi. Quindi il file identifier n.4 deve essere interpretato a seconda dell'ambiente in cui siamo.

Notare che ogni thread ha di solito **due stack**:

- Quello che viene utilizzato dal programma in **modalità utente**;
- Uno distinto che viene utilizzato quando il thread passa in **modalità di sistema**.

Per quale motivo?
Per sicurezza: Una chiamata di sistema non si può fidare dello stack del processo della modalità utente del processo che è stato impostato dal processo in modalità utente. Per esempio, non è detto che un certo processo è stato scritto in linguaggio C, ma potrebbe essere scritto in assembler da un attaccante che ha modificato il registro pointer e puntarlo ad una zona di memoria non permessa oppure ostile.
Dal momento che il processo potrebbe modificare a suo piacimento il registro stack pointer (che non è privilegiato) non è possibile fidarsi che questo punti ad uno stack «sano»: la chiamata di sistema, prima di vedere che tipo di chiamata di sistema si tratta, effettua un'operazione che è quella di impostare il registro stack pointer facendono puntare ad un secondo stack. Pertanto in modalità di sistema occorre usare uno stack sicuramente corretto.

Quando un thread viene creato, ha uno stack sia per la modalità utente che per la modalità di sistema. Una chiamata di sistema, come prima cosa, imposta lo stack del thread corrente allo stack di sistema, e al termine della chiamata di sistema ripristina lo stack a quello utente.

Uso delle librerie dinamiche per le API

Si vuole far sì che, anche se il sistema operativo viene aggiornato, non vi sia bisogno di ricompilare/linkare le applicazioni qualora siano cambiate le chiamate di sistema, o l'implementazione delle API, purché l'interfaccia delle API resti la stessa.
Se usiamo le librerie statiche, esse vengono linkate quando il programma viene compilato per realizzare il programma eseguibile. Quindi quando viene generato l'eseguibile, la libreria viene linkato staticamente nella creazione dell'eseguibile.
Un vantaggio si ha realizzando API e le librerie standard del linguaggio come librerie dinamiche. Infatti, se queste sono modificate (nell'implementazione, non nell'interfaccia), non occorre ricompilare tutti gli eseguibili per aggiornarli alla nuova versione delle librerie.

Supponiamo che cambi l'implementazione delle API (quindi cambia le chiamate di sistema). Se la libreria delle API fosse linkata staticamente agli eseguibili, si dovrebbe ricompilare e necessariamente re-linkare tutti gli eseguibili con la nuova libreria delle API. Se fosse linkata dinamicamente, l'eseguibile avrebbe solo un riferimento esterno a questa libreria e quindi viene linkata solo durante l'esecuzione. Quindi questa libreria viene modificata, basterebbe solo mettere la nuova libreria e l'eseguibile avrà un link dinamico e verrà collegato durante l'esecuzione.

Application binary interface (ABI)

Occorre però un'ulteriore accortezza: oltre all'API non deve cambiare l'application binary interface (ABI).
Definizione: L'**ABI** è l'insieme delle convenzioni attraverso le quali il codice binario dell'applicazione si interfaccia con il codice binario della libreria dinamica delle API.

In particolare, il codice binario dell'eseguibile chiama il codice binario chiama la libreria dinamica delle API. Queste convenzioni devono rimanere stabili altrimenti non avviene il linking dinamico di aggiornamenti di librerie. Le convenzioni ABI riguardano come si chiama internamente alla libreria le funzioni da invocare poiché non è detto che una funzione che si chiama write all'interno del codice sorgente della nostra applicazione si chiami write anche all'interno della libreria. C'è un certo grado di name mangling che gestisce appunto queste situazioni dove abbiamo nomi uguali ma non corrispondono alla stessa funzione. In caso non sia permesso dalla libreria l'esistenza di un'altra funzione con lo stesso nome, bisogna aggiungere dei prefissi per cercare di evitare che i nomi siano uguali.

Definizione di name mangling: Il name mangling è una tecnica utilizzata nei linguaggi di programmazione per modificare il nome di una funzione, di una variabile o di un altro simbolo in modo da incorporare informazioni aggiuntive nel nome stesso. Questo viene

spesso fatto per garantire l'unicità dei nomi dei simboli nel programma, specialmente quando si tratta di linguaggi che supportano la programmazione orientata agli oggetti o altre caratteristiche avanzate. Questo è utile per gestire la possibilità di avere più funzioni con lo stesso nome ma con firme diverse (overloading) o per gestire la compatibilità con il linguaggio C.

Di cosa si occupano le ABI (Come si chiama internamente alla libreria la funzione da invocare (name mangling)? In che ordine i parametri vengono messi sullo stack delle chiamate? Come sono strutturati i tipi di dati? C'è padding? Qual è l'endianess?

La scarsa portabilità degli eseguibili binari

Come facciamo ad avere applicazioni portabili su diversi sistemi di elaborazione?

Tre possibili approcci:

1. Scrivere l'applicazione in un linguaggio con un interprete portabile (es. Python, Ruby): l'eseguibile in tal caso è il sorgente.
2. Scrivere l'applicazione in un linguaggio con un ambiente runtime portabile (es. Java, .NET): l'eseguibile in tal caso è il bytecode che è un binario che non è il codice macchina o il codice del processore. È il codice macchina della JVM.
3. Scrivere l'applicazione utilizzando un linguaggio con un compilatore portabile ed API standardizzate: Difficile perché bisognerebbe scrivere l'applicazione con un linguaggio che abbia un ambiente runtime portabile, API standardizzate e 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? Il problema è che non è sufficiente scrivere un'applicazione su C, mettere i POSIX e caricarlo.

- 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 sistemi diversi possono supportare API diverse: ad esempio, Windows supporta le API Win32 e Win64 e non le API POSIX, supportate da Linux e MacOS;
- A parità di architettura e API sistemi diversi possono supportare diversi formati per 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.

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 (lo vedremo)
- Gestione della memoria
- Gestione dell'I/O (non lo vedremo)
- 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 che sono il vero obiettivo di un SO. Il kernel è solo un mezzo per un fine.
- Il kernel è un punto di dependability molto importante, in 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.

1. **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 etc.

   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

2. **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).

3. **Struttura a microkernel**

   Il principale problema dei kernel monolitici è la loro complessità, e di conseguenza fragilità e inaffidabilità.

   Punto importante: 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.

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

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 etc.
- 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)

4. 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)

5. Struttura ibrida

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 etc.

Esempi:

1. Linux e Solaris sono monolitici per avere prestazioni elevate, ma supportano anche i moduli del kernel per poter caricare e scaricare dinamicamente funzionalità

2. 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.

Politica	Meccanismo
Una politica dice **quando** una certa operazione viene effettuata. 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. Esempio: come fa il kernel a sospendere l'esecuzione di un programma in maniera che successivamente 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, se sono sufficientemente rapidi, non impattano profondamente sulle caratteristiche percepite del sistema di elaborazione.
Le politiche non sono stabili, perché 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.	Sono più stabili delle politiche.