Interfaccia e struttura del kernel

Pietro Braione

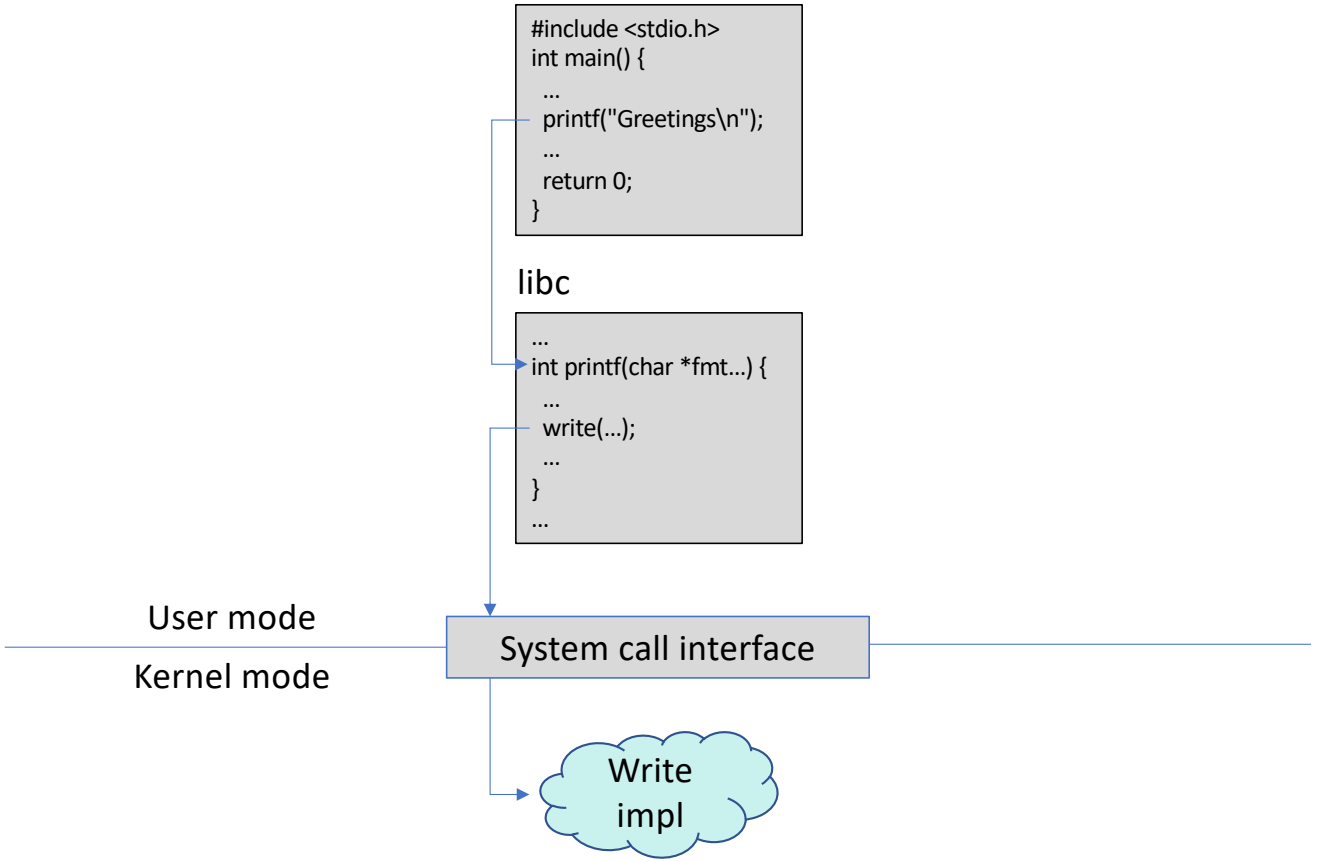
Reti e Sistemi Operativi – Anno accademico 2025-2026

Argomenti

- Implementazione di chiamate di sistema ed API
- Struttura del kernel

Implementazione di chiamate di sistema ed API

Implementazione API attraverso chiamata di sistema



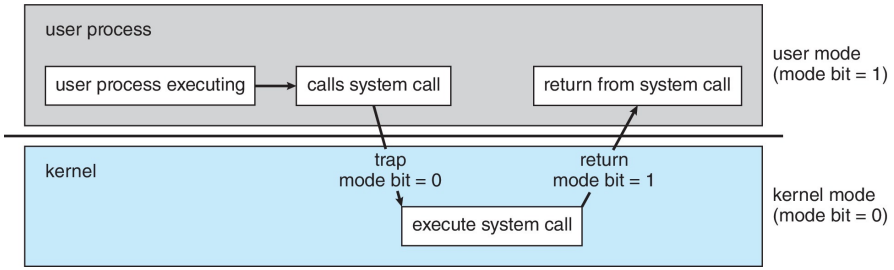
Duplice modalità di funzionamento

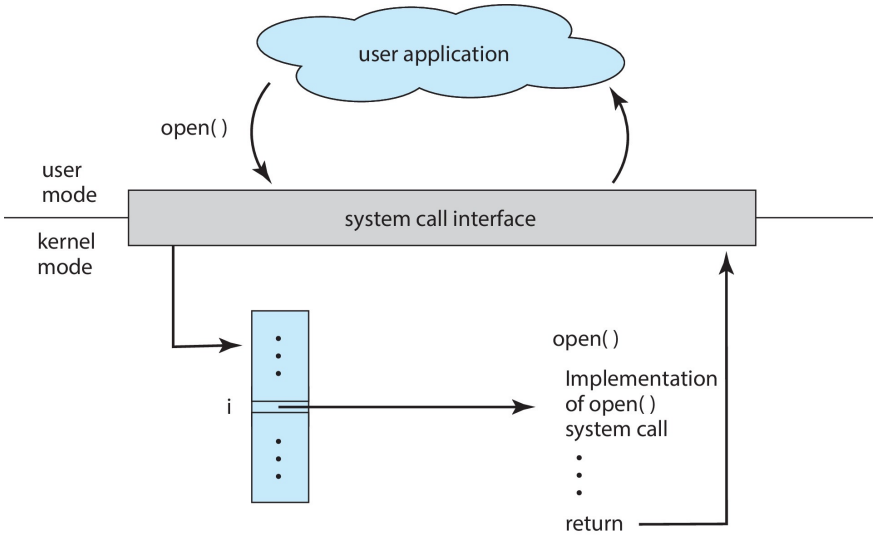
- Permette al sistema operativo di proteggere se stesso dai programmi in esecuzione
- La CPU può funzionare in **modalità utente** (**user mode**) o in **modalità di sistema** (**kernel mode**) impostando un opportuno **bit di modalità**
- Alcune istruzioni del processore sono **privilegiate**, ossia eseguibili solo in modalità di sistema: in particolare, in user mode la CPU non può accedere alla memoria del kernel

Implementazione delle chiamate di sistema (1)

- 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
- Prima vengono preparati i parametri necessari:
  - Un numero che identifica quale chiamata di sistema va effettuata…
  - …più tutti i parametri necessari alla specifica chiamata di sistema
  - (Vedremo nella slide successiva come vengono effettivamente passati questi dati)
- Quindi viene invocata un'opportuna istruzione macchina che genera un'eccezione software; essa fa passare la CPU in modalità di sistema e trasferisce il controllo ad una subroutine ad un determinato indirizzo di memoria
- 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

Implementazione delle chiamate di sistema (2)





Passaggio dei parametri alle chiamate di sistema

- 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
- Metodo più semplice: passare i parametri nei registri del processore
  - Vantaggio: rapido
  - Svantaggio: utile solo per pochi parametri i cui tipi di dati hanno dimensione limitata
- 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
  - Vantaggi: flessibile; simile ad una normale chiamata di procedura
  - Svantaggi: lento e macchinoso

La necessità del doppio stack

- Notare che ogni thread ha di solito *due* stack:
  - quello che viene utilizzato dal programma in modalità utente
  - ed uno distinto che viene utilizzato quando il thread passa in 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
- Per quale motivo? Per sicurezza:
  - Il programma utente potrebbe modificare a suo piacimento il registro stack pointer (che non è privilegiato), quindi non è possibile fidarsi che questo punti ad uno stack «sano»
  - Pertanto in modalità di sistema occorre che lo stack pointer punti ad uno stack sicuramente corretto

Uso delle librerie dinamiche per le API

- Si vuole far si 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
- Un vantaggio si ha realizzando le 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

Application binary interface (ABI)

- Occorre però un'ulteriore accortezza: oltre all'API non deve cambiare l'**application binary interface (ABI)**
- 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:
  - 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 (1)

- Come facciamo ad avere applicazioni portabili su diversi sistemi di elaborazione?
- Tre possibili approcci:
  - Scrivere l'applicazione in un linguaggio con un interprete portabile (es. Python, Ruby): l'eseguibile in tal caso è il sorgente
  - Scrivere l'applicazione in un linguaggio con un ambiente runtime portabile (es. Java, .NET): l'eseguibile in tal caso è il bytecode
  - Scrivere l'applicazione utilizzando un linguaggio con un compilatore portabile ed API standardizzate: 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?

La scarsa portabilità degli eseguibili binari (2)

- 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 di elaborazione ad un altro

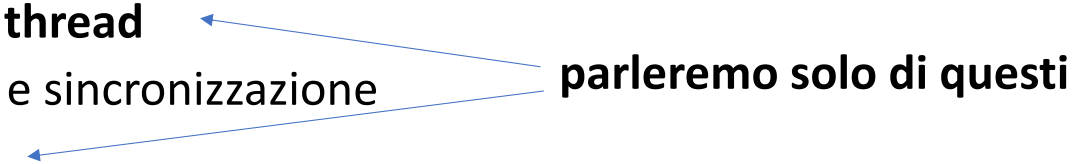
Struttura del kernel

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
  - Gestione della memoria
  - Gestione dell'I/O
  - File system

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
  - **Gestione della memoria**
  - Gestione dell'I/O
  - 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
  - Un 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

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 processi
  - File system
  - Gestione della memoria: swapping, memoria virtuale...
  - Gestione I/O: device drivers
  - IPC
- 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

Struttura a strati

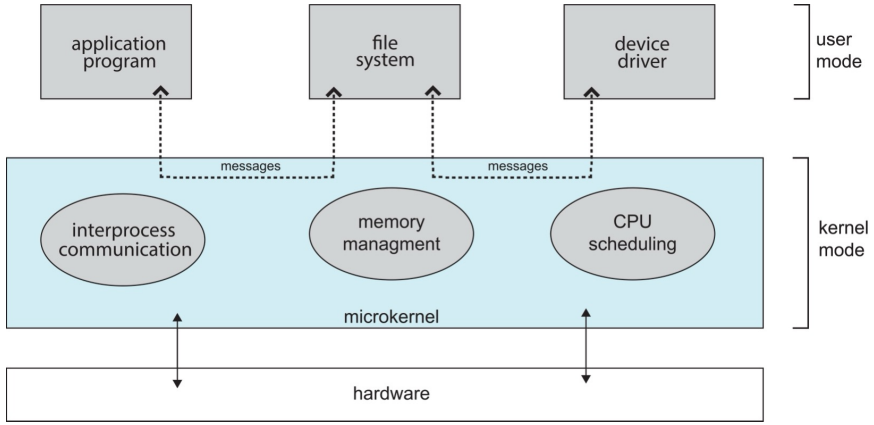
- Negli approcci stratificati il sistema operativo è diviso in un insieme di livelli, o strati
- Lo strato più basso interagisce con l'hardware, lo strato n-esimo interagisce solo con lo strato (n-1)-esimo
- L'approccio offre due vantaggi:
  - Ogni strato può essere progettato e implementato indipendentemente 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 conveniente strutturare a strati alcuni sottosistemi del kernel (es. file system o driver di rete)

Struttura a microkernel (1)

- Il principale problema dei kernel monolitici è la loro complessità, e di conseguenza fragilità e inaffidabilità
- La struttura a microkernel sposta quanti più sottosistemi 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
- L'approccio è stato proposto negli anni 80 con il sistema operativo Mach

Struttura a microkernel (2)

- Un microkernel offre pochi servizi, di solito lo scheduling dei processi, (una parte della) gestione della memoria e la comunicazione tra processi
- Gli altri sottosistemi (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



Struttura a microkernel (3)

- Vantaggi:
  - Facilità di estensione del sistema operativo: posso aggiungere un nuovo servizio senza dover modificare il kernel
  - Maggiore affidabilità: se un sottosistema va in crash non manda in crash l'intero sistema; 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…
- 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, XNU – kernel di macOS e iOS)

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)

Sistemi ibridi

- 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…
- Esempio: Linux e Solaris sono monolitici per avere prestazioni elevate, ma supportano anche i moduli del kernel
- Altro esempio: Windows NT
  - 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