Appunti di Sistemi Operativi

di:

Facchini Luca

Corso tenuto dal prof. Sistemi Operativi

Università degli Studi di Trento

A.A. 2024/2025

Autore:
Facchini Luca
Mat. 245965
Email:luca.facchini-1@studenti.unitn.it
luca@fc-software.it

Corso:
Sistemi Operativi [146065]
CdL: Laurea Triennale in Informatica
Prof. Crispo Bruno
Email: bruno.crispo@unitn.it

Sommario

Appunti del corso di Sistemi Operativi, tenuto dal prof. Crispo Bruno presso l'Università degli Studi di Trento. Corso seguito nell'anno accademico 2024/2025.
Dove non specificato diversamente, le immagini e i contenuti sono tratti dalle slide del corso del prof. Crispo Bruno (bruno.crispo@unitn.it)

Indice

Capitolo 1

Definizioni e Storia

Capitolo 2

Componenti di un sistema operativo

Dopo aver definito cosa sia un sistema operativo, vediamo ora quali siano le sue componenti principali, a partire dalla gestione dei processi e della memoria (primaria e secondaria), per poi passare alla gestione dei dell `I/O` e dei file fino ad arrivare alla protezione, la gestione della rete e l'interprete dei comandi.

2.1 Le Componenti in generale

Gestione dei Processi

Definizione 2.1 (Processo)**.** *Un **processo** è un programma in esecuzione che necessita di **risorse** per poter funzionare. Questo inoltre è eseguito in modo **sequenziale** ed **una istruzione alla volta**, infine è possibile che un processo sia del `SO` o dell'utente.*

In materia di gestione dei processi il sistema operativo è responsabile nella loro creazione e distruzione, nella loro sospensione e ripresa e deve fornire dei meccanismi per la sincronizzazione e la comunicazione tra i processi stessi.

Gestione della memoria primaria

Definizione 2.2 (Memoria primaria)**.** *La **memoria primaria** è la memoria principale del computer che conserva dati condivisi dalla `CPU` e dai dispositivi `I/O` questa è direttamente accessibile dalla `CPU`, per essere eseguito un programma deve essere caricato in memoria.*

La gestione della memoria primaria richiede la gestione dello spazi di memoria oltre alla decisione su quale processo debba essere caricato in memoria e quale debba essere rimosso. Inoltre il sistema operativo deve fornire dei meccanismi allocare e de-allocare la memoria.

Gestione della memoria secondaria

Definizione 2.3 (Memoria secondaria)**.** *La **memoria secondaria** è una memoria **non volatile** ed **grande** rispetto alla memoria primaria, questa è utilizzata per memorizzare i dati e i programmi in modo **permanente**.*

Questa memoria consiste di uno o più dischi (magnetici) ed il sistema operativo deve fornire dei meccanismi per la gestione dello spazio libero, l'allocazione dello spazio ed lo *scheduling* degli accessi ai dischi.

Gestione dell'`I/O`

Il `SO` nasconde la complessità dell'`I/O` ai programmi utente, fornendo un'astrazione dell'`I/O` e fornendo dei meccanismi per: accumulare gli accessi ai dispositivi (*buffering*), fornire una interfaccia generica per i dispositivi e fornire dei *driver* specifici (scritti in `C`, `C++` o *assembly*).

Gestione dei file

Definizione 2.4 (File)**.** *Un **file** è una sequenza di byte memorizzata in un qualsiasi supporto fisico controllato da driver del sistema operativo.*

Un file è dunque un'astrazione logica per rendere più semplice la memorizzazione e l'uso della memoria **non volatile**. Il sistema operativo deve fornire dei meccanismi per la creazione, la cancellazione, la lettura e la scrittura di file e *directory* oltre a fornire delle primitive (copia, sposta, rinomina) per la gestione dei file.

Protezione

Il sistema operativo deve fornire dei meccanismi per controllare l'accesso a tutte le risorse da parte di processi e utenti, inoltre l'`SO` è responsabile della definizione di accessi autorizzati e non autorizzati, oltre a definire i controlli necessari ed a fornire dei meccanismi per verificare le politiche di accesso definite.

2.2 Come usare i servizi dei sistemi operativi

Il sistema operativo metta a disposizione le sue interface tramite delle *system call* che sono delle chiamate a funzione che permettono di accedere ai servizi del sistema operativo precedentemente descritti. Queste chiamate a funzione sono utilizzate per eseguire operazioni che richiedono privilegi di sistema, come ad esempio la gestione dei processi, della memoria, dell'`I/O` e dei file.

2.2.1 Interprete dei comandi

Un esempio di utilizzo delle *system call* è l'interazione con l'interprete dei comandi, che permette di eseguire comandi e programmi tramite una interfaccia testuale. Questo interprete tramuta i comandi in *system call* che vengono poi eseguite dal sistema operativo. Questo permette di creare e gestire processi, gestire `I/O`, disco, memoria e file oltre alla gestione delle protezione e della rete.
Nel `SO` esistono dei comandi predefiniti che possono essere chiamati direttamente per il loro nome, questi sono implementati con una semantica specifica e possono essere utilizzati per eseguire operazioni di base, nel caso di comandi non predefiniti è possibile scrivere dei programmi che vengono eseguiti dall'interprete dei comandi.

2.2.2 L'interfaccia grafica

Un'altra interfaccia che permette di interagire con il sistema operativo è l'interfaccia grafica, che permette di interagire con il sistema operativo tramite il *mouse* e la tastiera. Questa interfaccia più intuitiva e facile da usare rispetto all'interprete dei comandi, permette di interagire con il `SO` tramite icone e finestre. Questa interfaccia, anche se più semplice, non è per forza più veloce dell'interprete dei comandi, in quanto l'interfaccia grafica è più lenta e richiede più risorse rispetto all'interprete dei comandi.

2.2.3 *System calls*

I processi non usano le *shell* per eseguire le *system call*, ma usano delle `API` (*Application Programming Interface*) che permettono di accedere ai servizi del sistema operativo. Queste `API` sono delle librerie di funzioni ad alto livello che permettono di accedere ai servizi del sistema operativo. Queste librerie sono scritte in `C` o `C++` e permettono di accedere ai servizi del sistema operativo in modo più semplice e più sicuro rispetto all'uso diretto delle *system call*.

Esempio di `API` Un esempio di `API` è la `Win32`, prendiamo in esame la funzione `ReadFile` che permette di leggere un file:

```
BOOL ReadFile (
    HANDLE file,
    LPVOID buffer,
    DWORD bytes to read,
    LPDWORD bytes read,
    LPOVERLAPPED ovl
);
```

Questa funzione ritorna un valore booleano che indica se la funzione è andata a buon fine o meno, inoltre questa funzione prende in input il file da leggere, il buffer in cui scrivere i dati letti, il numero di byte da leggere, il numero di byte letti e un puntatore a una struttura `OVERLAPPED` che permette di specificare un offset per la lettura.

Le `API` nei diversi `SO`

Le 2 `API` più comuni per Windows sono: `Win32` e `Win64` mentre per Linux sono: `POSIX` (*Portable Operating-System Interface*) che includono le *system call* per tutte le versioni di `UNIX`, *Linux* e *Mac OS X*, o tutte le distribuzioni `POSIX`-*compliant*.

Windows* su *Linux Per eseguire programmi *Windows* su *Linux* è possibile usare `Wine` che è un *emulatore* il quale traduce le chiamate `API` di *Windows* in chiamate `API` di *Linux on-the-fly*, ovvero durante l'esecuzione del programma. Questo permette di eseguire programmi *Windows* su *Linux* senza dover riscrivere il codice del programma.

Implementazione delle *System Call*

Ad ogni *system call* è associato un numero univoco, che permette al sistema operativo di identificare la *system call* richiesta. È compito dell'interfaccia tenere traccia dei numeri associati alle *system call* e di passare i parametri alla *system call* richiesta. Questa interfaccia invoca la *system call* nel *kernel* del sistema operativo, che esegue la *system call* e ritorna il risultato al chiamante. Questo meccanismo permette al chiamante di non dover conoscere i dettagli di implementazione della *system call* ma solo la sua interfaccia.

Esecuzione delle *system calls* Per eseguire una *system call* dopo che il processo ha eseguito la chiamata all'interfaccia del `SO` il quale conoscendo il numero della *system call* controlla dove questa è implementata tramite la *system call table* (una tabella che contiene i puntatori alla implementazione delle *system call*). Una volta trovata la *system call* il `SO` esegue la *system call* e ritorna il risultato al chiamante.

Opzioni per il passaggio dei parametri I parametri di una *system call* possono essere passati in diversi modi. I più comuni sono: passaggio tramite registri, passaggio tramite lo `stack` e passaggio tramite puntatori. Il passaggio tramite registri è il più veloce ma permette di passare pochi parametri e di piccola dimensione, il passaggio tramite lo `stack` permette di passare più parametri e di dimensioni maggiori, infine il passaggio tramite puntatori permette di passare parametri di dimensioni maggiori e di passare parametri complessi, ma và passata una tabella di parametri che deve essete passata tramite `stack` o registri.

Parametri tramite `stack`Il passaggio dei parametri tramite `stack` avviene in questo modo:

1-3 Salvataggio parametri sullo `stack`

4 Chiamata della funzione di libreria

5 Caricamento del numero della *system call* su un registro `Rx`

6 Esecuzione `TRAP` (Passaggio in *kernel mode*)

7-8 Esecuzione della *system call*

9 Ritorno al chiamante

10-11 Ritorno al codice utente ed incremento dello `stack pointer`

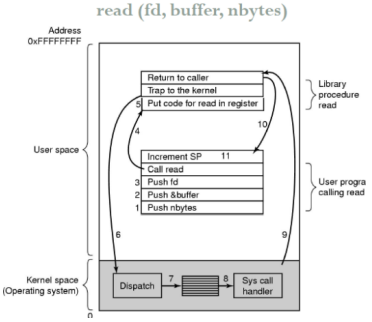

Figura 2.1: Passaggio dei parametri tramite `stack`

Passaggio di parametri tramite tabella Come anticipato il passaggio di parametri tramite tabella viene utilizzato per passare parametri complessi o di dimensioni maggiori andando a passare un puntatore alla tabella che contiene i parametri. Questo metodo permette di passare un numero maggiore di parametri e di dimensioni maggiori in quanto i parametri sono passati per riferimento alla memoria primaria.

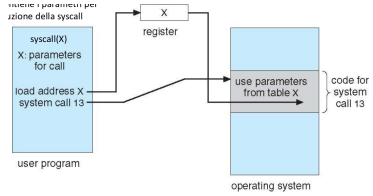

Figura 2.2: Passaggio dei parametri tramite tabella

Capitolo 3

Architettura di un Sistema Operativo

In un sistema operativo è molto importante separare le *policy* dai *meccanismi*. I meccanismi sono le funzionalità che il sistema operativo mette a disposizione, mentre le *policy* sono le regole che il sistema operativo segue per decidere come utilizzare i meccanismi.

Principi di progettazione Il principio di progettazione di un sistema operativo è quello di `KISS` (*Keep It Small and Simple*) usato per ottimizzare al meglio le *performance* implementando solo lo stretto necessario. Altro principio è il `POLP` (*Principle of Least Privilege*), ovvero dare il minimo dei privilegi necessari ad ogni componente per svolgere il proprio compito. Quest'ultimo principio è molto importante per garantire affidabilità e sicurezza.

3.1 Tipi di architetture

Sistemi monoblocco

Nei sistemi monoblocco non è presente una gerarchia tra i vari livelli del sistema operativo. Questo tipo di architettura è molto semplice e consiste in un unico strato *software* tra l'utente ed l'*hardware* del sistema. Le componenti sono dunque tutte allo stesso livello permettendo una comunicazione diretta tra l'utente e l'*hardware*. Questo tipo di architettura è molto semplice e veloce, ma il codice risulta interamente dipendente dall'architettura ed è distribuito su tutto il sistema operativo. Inoltre per testare ed eseguire il *debugging* di un singolo componente è necessario analizzare l'intero sistema operativo.

Sistemi a struttura semplice

Nei sistemi a struttura semplice è presente una piccola gerarchia, molto flessibile, tra i vari livelli del sistema operativo. Questo tipo di architettura mira ad una riduzione dei costi di sviluppo ed di manutenzione del sistema operativo. Non avendo una struttura ben definita, i componenti possono comunicare tra loro in modo diretto. Questo tipo di architettura è molto flessibile e permette di avere un sistema operativo molto piccolo e veloce come `MS-DOS` o `UNIX` originale.

`MS-DOS` Il sistema operativo `MS-DOS` è un sistema operativo a struttura semplice, molto piccolo e veloce. Questo sistema operativo è pensato per fornire il maggior numero di funzionalità in uno spazio ridotto. Infatti non sussistono suddivisioni in moduli, ed le interfacce e livelli non sono ben definiti. È infatti possibile accedere direttamente alle *routine* del sistema operativo ed non è prevista la *dual mode*.

`UNIX` **(Originale)** Struttura semplice limitata dalle poche funzionalità disponibili all'epoca in materia di *hardware*, con un *kernel* molto piccolo e veloce il quale scopo è risiedere tra l'interfaccia delle *system call* e l'*hardware*. Questo sistema operativo è stato progettato per essere molto flessibile e fornisce: *File System*, *Scheduling* della `CPU`, gestione della memoria e molto altro.

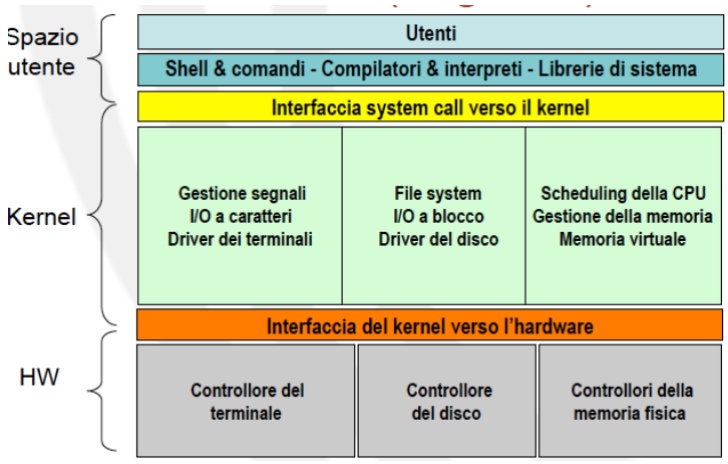

Figura 3.1: Struttura di `UNIX` originale

Sistema a livelli

Nei sistemi operativi organizzati a livelli gerarchici l'interfaccia utente risiede al livello più altro mentre l'*hardware* dal lato opposto. Ogni livello intermedio può solo usare funzioni fornite dal livello inferiore ed offrire funzionalità al livello superiore. Principale vantaggio di questa architettura è la modularità, infatti ogni livello può essere sviluppato e testato indipendentemente dagli altri. Questo tipo di architettura, d'altronde non è priva di svantaggi, infatti diventa difficile definire in modo approssimato gli strati, l'efficienza decresce in quanto ogni singolo strato aggiunge un costo di *overhead* ed le funzionalità dipendenti dal'*hardware* sono sparse su più livelli.

`THE` Il sistema operativo `THE` è un sistema d'uso accademico ed è il primo sistema operativo a struttura a livelli. Questo `SO` consiste in un insieme di processo che cooperano tra di loro usando la tecnica dei "semafori" per la sincronizzazione.

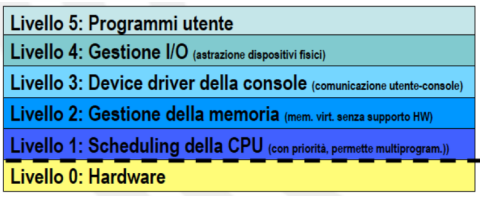

Figura 3.2: Struttura di `THE`

Sistemi basati su *Kernel*

I sistemi di questo genere hanno due soli livelli: i servizi *kernel* e quelli non-*kernel* (o *utente*). Il *file system* è un esempio di servizio non-*kernel*. Questo tipo di architettura è molto diffuso in quanto il ridotto e ben definito numero di livelli ne permette una facile implementazione e manutenzione, spesso però questo sistema può risultare troppo rigido e non adatto a tutti i tipi di applicazioni, oltre alla totale assenza di regole organizzative per le parti del `SO` al di fuori del *kernel*.

Micro-kenrel

Questo tipo di *kernel* è molto piccolo e fornisce solo i servizi essenziali per il funzionamento del sistema operativo. Tutte le altre funzionalità sono implementate come processi utente. Un esempio di ciò è `seL4` un *kernel open source* che implementa un *micro-kernel* e fornisce un'interfaccia per la gestione della memoria, dei processi e della comunicazione tra processi. `SeL4` è matematicamente verificato e privo di bug rispetto alle sue specifiche di forte sicurezza

Virtual Machine

L'architettura a VM è una estremizzazione dell'approccio a più livelli di IBM (1972), questo è pensato per offrire un sistema di *timesharing* "multiplo" dove il sistema operativo viene eseguito su una VM ed questa dà illusione di processi multipli, ma nella realtà ognuno di questi è in esecuzione sul proprio HW. In questo paradigma sono possibili più SO in una unica macchina.

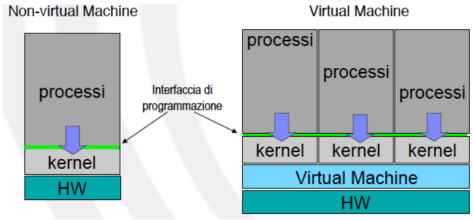

Figura 3.3: Differenze tra una macchina senza e con VM

Come è possibile notare ogni singolo processo è separato ed possiede il proprio *kernel*. Vengono quindi separata la multiprogrammazione ed la presentazione.

Tipo di *Hypervisor*

- **Tipo 1** (*Bare Metal*): Questo tipo di *Hypervisor* è installato direttamente sul'*hardware* e non necessita di un sistema operativo ospite. Questo tipo di *Hypervisor* è molto veloce e sicuro, ma è molto complesso da installare e configurare.
- **Tipo 2** (*Hosted*): Questo tipo di *Hypervisor* è installato sopra un sistema operativo ospite. Questo tipo di *Hypervisor* è molto più semplice da installare e configurare rispetto al tipo 1, ma è più lento e meno sicuro, inoltre è possibile avere problemi di compatibilità tra il sistema operativo ospite e il *Hypervisor*.

***Monolithic* vs *Micro-kernel* VM** Prima di fare una distinzione tra i due tipi di VM è necessario dire che entrambi rientrano nel tipo 1 di *Hypervisor* e dunque tutti i SO sono eseguiti direttamente sul'*hardware* virtualizzato.

- ***Monolithic***: Questo tipo di VM è molto simile ad un sistema operativo tradizionale, infatti ogni *VM* è un processo separato che esegue il proprio *kernel*. Questo tipo di VM è molto veloce, ma è molto complesso da implementare e mantenere.
- ***Micro-kernel***: Questo tipo di VM è molto simile ad un sistema operativo a *micro-kernel*, infatti il *kernel* della VM fornisce solo i servizi essenziali per il funzionamento del sistema operativo. Tutte le altre funzionalità sono implementate come processi utente. Questo tipo di VM è molto più semplice da implementare e mantenere rispetto al tipo 1, ma è più lento e meno sicuro.

Vantaggi - Svantaggi Principale vantaggio di questo tipo di architettura è la completa protezione del sistema, infatti ogni SO è separato e non può accedere alle risorse degli altri SO. Inoltre è possibile avere più SO in una sola macchina andando ad ottimizzare le risorse e ridurre i costi di sviluppo di un sistema operativo, oltre ad aumentare la portabilità delle applicazioni. Principale svantaggio riguardano le prestazioni del sistema, infatti ogni SO è eseguito su una VM e questo può portare ad un aumento dei tempi di esecuzione delle applicazioni. Inoltre è necessario avere gestire una *dual-mode* virtuale e non è possibile avere un sistema operativo in tempo reale, inoltre il fatto che una VM non possa accedere alle altre VM può portare ad un aumento dei costi di sviluppo e manutenzione del sistema.

Sistemi *client-server*

Poco diffusi ai giorni nostri, i sistemi *client-server* sono basati su un'architettura a due livelli: il *client* e il *server*. Questo sistema di basa sull'idea che il codice del sistema operativo vada portato sul livello superiore (il *client*) e il *server* rimanga molto piccolo e veloce andando solo a fornire i servizi essenziali per il funzionamento del sistema operativo ed la comunicazione tra il *client* e l'*hardware*. Questo tipo di architettura si presta bene per sistemi distribuiti.

3.2 Implementazione di un SO

I sistemi operativi sono tradizionalmente scritti in linguaggio *assembler* anche se è possibile scriverli in linguaggi di alto livello, come `C` o `C++`. La scrittura di un sistema operativo in linguaggio di alto livello permette di avere una implementazione molto rapida oltre ad aumentarne la compattezza e la mantienebilità. Inoltre è possibile avere una maggiore portabilità del sistema operativo, in quanto è possibile compilare il codice sorgente su più architetture. Tuttavia la scrittura di un sistema operativo in linguaggio di alto livello può portare ad un aumento dei tempi di esecuzione delle applicazioni e ad un aumento dei costi di sviluppo e manutenzione del sistema operativo.