Il Sistema Operativo: Fondamenti e Architettura

Introduzione

Quando accendiamo un computer, uno smartphone o qualsiasi dispositivo computazionale moderno, diamo per scontato che tutto "funzioni semplicemente". Clicchiamo su un'icona e un'applicazione si apre. Salviamo un file e questo viene memorizzato in modo persistente. Connettiamo una stampante e il sistema la riconosce automaticamente. Dietro questa apparente semplicità si nasconde uno dei componenti software più complessi e sofisticati mai creati dall'ingegneria informatica: il sistema operativo.

Il sistema operativo rappresenta il cuore pulsante di ogni sistema computazionale, un layer software fondamentale che si interpone tra l'hardware fisico e le applicazioni utente, orchestrando e coordinando tutte le risorse della macchina. Senza un sistema operativo, un computer sarebbe poco più di un insieme di componenti elettronici incapaci di eseguire qualsiasi compito utile in modo pratico ed efficiente.

In questa lezione esploreremo in profondità cosa sia un sistema operativo, perché sia necessario, come funzioni e quali siano le sue componenti fondamentali. Partiremo dalle motivazioni storiche che hanno portato alla sua nascita, per poi analizzare le sue funzioni principali e i meccanismi attraverso cui gestisce le risorse hardware e coordina l'esecuzione dei processi.

Parte I: Le Origini e la Necessità del Sistema Operativo

1.1 I Computer Primitivi: Un Mondo Senza Sistema Operativo

Per comprendere appieno l'importanza e la funzione di un sistema operativo, è utile fare un passo indietro nella storia dell'informatica e immaginare come funzionassero i primi computer elettronici, quelli degli anni '40 e '50.

I primi calcolatori elettronici, come l'ENIAC (Electronic Numerical Integrator and Computer) costruito nel 1945, erano macchine enormi che occupavano intere stanze e consumavano quantità impressionanti di energia elettrica. La loro programmazione era un processo estremamente laborioso: non esistevano linguaggi di programmazione di alto livello, non c'erano compilatori, e certamente non esisteva alcun sistema operativo.

Per eseguire un programma su queste macchine, un operatore doveva:

1. **Configurare fisicamente la macchina**: Spesso questo significava collegare cavi, impostare interruttori e configurare pannelli di controllo. La programmazione era letteralmente "hard-wired" nella macchina.
2. **Caricare manualmente dati e istruzioni**: I dati e le istruzioni del programma venivano inseriti attraverso schede perforate o nastri magnetici. L'operatore doveva caricare fisicamente questi supporti nella macchina.
3. **Avviare l'esecuzione**: Una volta configurato tutto, l'operatore avviava manualmente l'esecuzione del programma.

4. **Attendere il completamento**: Il computer eseguiva il programma dall'inizio alla fine senza interruzioni. Se si verificava un errore, l'intera procedura doveva essere ripetuta.

5. **Raccogliere i risultati**: I risultati venivano stampati o perforati su schede, e l'operatore doveva raccoglierli manualmente.

Questo approccio aveva numerosi problemi evidenti:

- **Tempo sprecato**: Gran parte del tempo della costosa macchina veniva speso in operazioni manuali di caricamento e configurazione, piuttosto che in calcoli effettivi.
- **Sottoutilizzo delle risorse**: Mentre l'operatore configurava il prossimo job, la CPU rimaneva inattiva.
- **Complessità per il programmatore**: Ogni programmatore doveva conoscere intimamente l'hardware della macchina e gestire direttamente tutti i dispositivi di I/O.
- **Mancanza di isolamento**: Un errore in un programma poteva corrompere l'intera macchina, richiedendo un riavvio completo.

1.2 L'Evoluzione: Dai Monitor Residenti ai Sistemi Operativi Moderni

Con l'aumento della potenza computazionale e del costo delle macchine, divenne sempre più importante massimizzare l'utilizzo dell'hardware. Questo portò allo sviluppo dei primi rudimentali sistemi operativi.

I Sistemi Batch

La prima evoluzione furono i **sistemi batch** (a lotti) degli anni '50. L'idea era semplice ma rivoluzionaria: invece di eseguire un programma alla volta con lunghe pause tra l'uno e l'altro, si raggruppavano più programmi (job) in un "lotto" che veniva elaborato sequenzialmente senza intervento umano.

Un operatore caricava un nastro o un set di schede perforate contenente diversi programmi. Un programma semplice, chiamato **monitor residente**, rimaneva sempre in memoria e aveva il compito di:

- Caricare automaticamente il prossimo programma dal lotto
- Avviarne l'esecuzione
- Gestire il passaggio tra un programma e il successivo
- Raccogliere output e segnalare errori

Questo monitor residente può essere considerato l'antenato dei moderni sistemi operativi. Tuttavia, aveva ancora limitazioni significative:

- I programmi venivano eseguiti rigorosamente in sequenza
- Non c'era multiprogrammazione (esecuzione concorrente)
- L'interazione utente era inesistente durante l'esecuzione

La Multiprogrammazione

Negli anni '60, con l'introduzione di CPU più veloci e memorie più capienti, emerse un nuovo problema: la CPU rimaneva spesso inattiva mentre attendeva il completamento di operazioni di I/O, che erano (e sono tuttora) ordini di grandezza più lente delle operazioni di calcolo.

La soluzione fu la **multiprogrammazione**: mantenere in memoria più programmi contemporaneamente. Quando un programma si metteva in attesa per un'operazione di I/O (ad esempio, leggere da disco), il

sistema operativo poteva sospenderlo temporaneamente e passare l'esecuzione a un altro programma pronto. In questo modo, la CPU rimaneva costantemente occupata.

Questo approccio richiedeva un sistema operativo molto più sofisticato, capace di:

- Gestire la memoria per ospitare più programmi contemporaneamente
- Decidere quale programma eseguire in ogni momento (scheduling)
- Proteggere i programmi l'uno dall'altro
- Coordinare l'accesso ai dispositivi di I/O condivisi

I Sistemi Time-Sharing

Negli anni '70 si diffusero i **sistemi time-sharing** (condivisione del tempo), che estendevano il concetto di multiprogrammazione per fornire l'illusione a più utenti di avere ciascuno un computer dedicato.

Il sistema operativo assegnava a ciascun processo una piccola frazione di tempo CPU (time slice o quantum), tipicamente nell'ordine di millisecondi. I processi venivano eseguiti a turno così rapidamente che ogni utente aveva l'impressione di avere l'intero sistema a disposizione.

Sistemi come UNIX (sviluppato nei Bell Labs da Ken Thompson e Dennis Ritchie) e MULTICS rappresentarono pietre miliari in questa evoluzione, introducendo concetti che sono ancora alla base dei sistemi operativi moderni.

1.3 Perché Serve un Sistema Operativo: Le Motivazioni Fondamentali

Dopo questo excursus storico, possiamo identificare le ragioni fondamentali per cui un sistema operativo è necessario:

1. Astrazione dell'Hardware

L'hardware di un computer è complesso e varia enormemente tra diversi modelli e produttori. Ogni componente (CPU, memoria, dischi, schede di rete, GPU) ha le sue peculiarità, i suoi protocolli di comunicazione, i suoi registri di controllo.

Il sistema operativo fornisce un'**astrazione** di questo hardware, presentando ai programmatori e alle applicazioni un'interfaccia unificata, coerente e molto più semplice. Invece di dover conoscere i dettagli specifici di ogni disco rigido, un programmatore può semplicemente usare funzioni come `open()`, `read()`, `write()` e `close()`, lasciando al sistema operativo il compito di tradurre queste operazioni ad alto livello nei comandi specifici per l'hardware sottostante.

Questa astrazione ha benefici enormi:

- **Portabilità**: Le applicazioni possono girare su hardware differente senza modifiche
- **Semplicità**: I programmatori possono concentrarsi sulla logica applicativa anziché sui dettagli hardware
- **Evoluzione**: L'hardware può essere aggiornato senza dover riscrivere tutte le applicazioni

2. Gestione delle Risorse

Un computer moderno ha risorse limitate che devono essere condivise tra molteplici programmi in esecuzione simultanea:

- **CPU**: Uno o più core di elaborazione
- **Memoria RAM**: Spazio limitato per memorizzare dati e istruzioni
- **Dispositivi di I/O**: Dischi, stampanti, schede di rete, ecc.
- **Risorse software**: File, socket di rete, oggetti di sincronizzazione

Il sistema operativo agisce come un **gestore delle risorse**, decidendo:

- Quale processo può usare la CPU e per quanto tempo
- Quanta memoria allocare a ciascun processo
- Come schedulare le richieste di I/O
- Come risolvere conflitti quando più processi richiedono la stessa risorsa

Senza questa gestione centralizzata, il caos regnerebbe: i programmi si contenderebbero le risorse in modo anarchico, portando a inconsistenze, corruzione dei dati e prestazioni imprevedibili.

3. Isolamento e Protezione

Quando più programmi sono in esecuzione contemporaneamente, è fondamentale che siano isolati l'uno dall'altro. Un programma malfunzionante o malevolo non dovrebbe poter:

- Leggere o modificare la memoria di altri processi
- Interferire con l'esecuzione di altri programmi
- Accedere a file per cui non ha permessi
- Monopolizzare le risorse a scapito degli altri

Il sistema operativo implementa meccanismi di **protezione** a più livelli:

- Isolamento della memoria attraverso la memoria virtuale
- Controllo degli accessi tramite permessi su file e risorse
- Separazione tra modalità utente e modalità kernel
- Limiti su risorse utilizzabili (CPU time, memoria, file aperti)

4. Servizi Comuni

Molte operazioni sono comuni a quasi tutte le applicazioni: leggere e scrivere file, comunicare in rete, gestire l'input da tastiera e mouse, visualizzare informazioni sullo schermo.

Invece di far implementare a ogni applicazione queste funzionalità da zero, il sistema operativo fornisce **servizi comuni** attraverso system call (chiamate di sistema). Questo non solo semplifica lo sviluppo software, ma garantisce anche coerenza e affidabilità.

Parte II: Cos'è un Sistema Operativo - Definizione e Componenti

2.1 Definizione Formale

Possiamo ora fornire una definizione più rigorosa di sistema operativo:

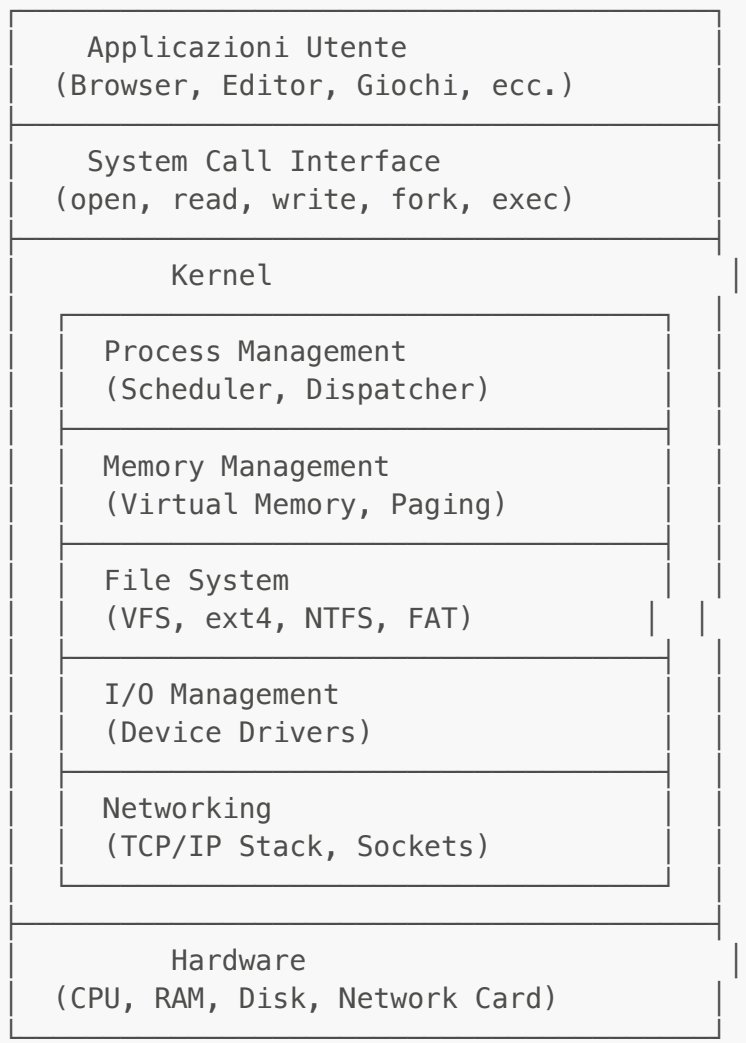
> **Un sistema operativo è un insieme di programmi che gestiscono le risorse hardware di un computer, forniscono servizi comuni alle applicazioni utente, e creano un ambiente di esecuzione controllato e isolato per i processi.**

Più precisamente, un sistema operativo svolge due ruoli principali:

1. **Gestore delle risorse**: Alloca e coordina l'uso di CPU, memoria, dispositivi di I/O e altre risorse tra i processi in competizione.
2. **Macchina estesa**: Fornisce un'astrazione dell'hardware sottostante, presentando un'interfaccia più semplice e uniforme per le applicazioni.

2.2 Architettura di un Sistema Operativo

Un sistema operativo moderno ha un'architettura complessa e stratificata. Possiamo visualizzarlo come una serie di layer concentrici:



Il Kernel

Il **kernel** (nucleo) è la parte centrale e più privilegiata del sistema operativo. È l'unico componente software che ha accesso diretto e completo all'hardware. Il kernel esegue in una modalità speciale della CPU

chiamata **kernel mode** (o supervisor mode), che permette l'esecuzione di istruzioni privilegiate e l'accesso a tutta la memoria.

Il kernel è responsabile delle funzioni più critiche:

- Scheduling dei processi
- Gestione della memoria virtuale
- Gestione dell'I/O
- Gestione del file system
- Comunicazione tra processi (IPC)
- Networking di basso livello

Le System Call

Le **system call** (chiamate di sistema) costituiscono l'interfaccia tra il kernel e le applicazioni utente. Sono funzioni speciali che permettono a un programma in modalità utente di richiedere servizi al kernel.

Quando un programma effettua una system call, avviene una **trap** (interruzione software): il processore passa dalla modalità utente alla modalità kernel, il controllo viene trasferito al kernel, che esegue l'operazione richiesta e poi restituisce il risultato all'applicazione, tornando in modalità utente.

Esempi di system call comuni:

- `fork()`: Crea un nuovo processo
- `exec()`: Sostituisce il programma corrente con uno nuovo
- `open()`, `read()`, `write()`, `close()`: Operazioni su file
- `malloc()`, `free()`: Gestione della memoria (spesso implementate in user space sopra `brk()` o `mmap()`)
- `socket()`, `connect()`, `send()`, `recv()`: Comunicazione di rete

I Driver dei Dispositivi

I **device driver** sono moduli software che permettono al sistema operativo di comunicare con specifici dispositivi hardware. Ogni tipo di dispositivo (schede video, dischi, stampanti, schede audio) ha il suo driver.

I driver traducono le richieste generiche del kernel in comandi specifici per il dispositivo. Ad esempio, quando il file system richiede di leggere un blocco di dati dal disco, il driver del disco traduce questa richiesta in comandi specifici per quel particolare modello di disco (IDE, SATA, NVMe, ecc.).

I driver possono essere:

- **Staticamente linkati** nel kernel (compiled-in)
- **Caricabili dinamicamente** come moduli kernel (più flessibile)

2.3 Modalità di Esecuzione: User Mode vs Kernel Mode

La separazione tra modalità utente e modalità kernel è un concetto fondamentale per la sicurezza e la stabilità del sistema.

User Mode (Modalità Utente)

Quando un'applicazione normale è in esecuzione, la CPU opera in **user mode**. In questa modalità:

- Non è possibile eseguire istruzioni privilegiate (es. manipolare direttamente l'hardware)
- Non è possibile accedere alla memoria del kernel o di altri processi
- Non è possibile disabilitare gli interrupt
- Non è possibile modificare i registri di controllo della MMU (Memory Management Unit)

Queste restrizioni sono imposte dall'hardware stesso: se un programma in user mode tenta di eseguire un'istruzione privilegiata, la CPU genera un'eccezione che causa la terminazione del processo.

Kernel Mode (Modalità Kernel)

Quando la CPU esegue codice del kernel, opera in **kernel mode**. In questa modalità:

- Tutte le istruzioni sono permesse
- L'intero spazio di indirizzamento è accessibile
- I dispositivi hardware possono essere controllati direttamente
- Gli interrupt possono essere gestiti e disabilitati

Il passaggio da user mode a kernel mode avviene solo in situazioni controllate:

1. **System call**: Il programma richiede esplicitamente un servizio al kernel
2. **Interrupt hardware**: Un dispositivo richiede attenzione (es. un pacchetto di rete è arrivato)
3. **Eccezione**: Si verifica un errore (es. divisione per zero, page fault)

Dopo aver gestito la situazione, il kernel restituisce il controllo al programma utente, tornando in user mode.

Questa architettura a due livelli protegge il sistema da programmi malfunzionanti o malevoli, garantendo che solo il kernel possa eseguire operazioni critiche.

Parte III: Gestione dei Processi

3.1 Cos'è un Processo

Un **processo** è un programma in esecuzione. Mentre un programma è un'entità statica (un file eseguibile sul disco), un processo è un'entità dinamica che evolve nel tempo.

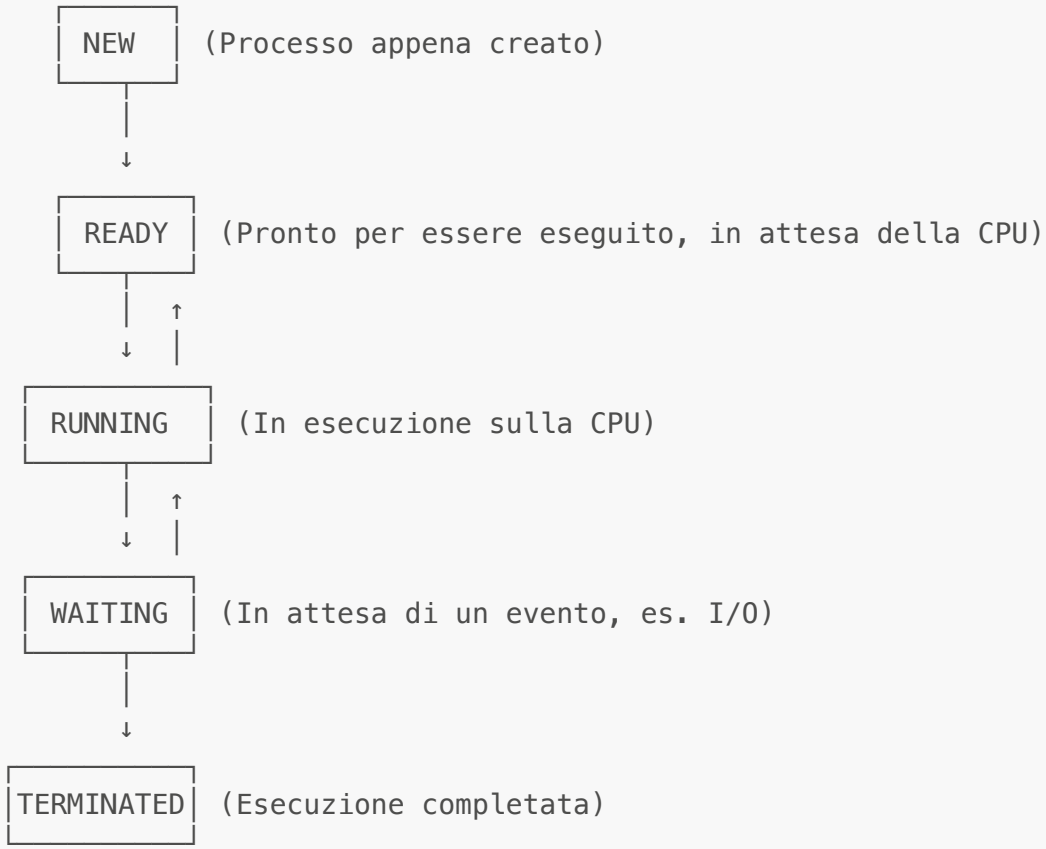
Più precisamente, un processo comprende:

1. **Codice del programma** (text section): Le istruzioni macchina da eseguire
2. **Program Counter**: L'indirizzo dell'istruzione corrente
3. **Registri della CPU**: Lo stato corrente dei registri generali
4. **Stack**: Memoria per variabili locali, parametri di funzione, indirizzi di ritorno
5. **Heap**: Memoria allocata dinamicamente durante l'esecuzione
6. **Dati**: Variabili globali e statiche
7. **File aperti**: Descrittori dei file attualmente in uso dal processo
8. **Informazioni di accounting**: CPU time usato, priorità, PID (Process ID)

Il sistema operativo mantiene per ogni processo una struttura dati chiamata **PCB (Process Control Block)** che contiene tutte queste informazioni. Il PCB è fondamentale per il context switching.

3.2 Stati di un Processo

Durante la sua vita, un processo passa attraverso diversi stati. Il modello semplificato a cinque stati è il seguente:



Transizioni tra stati:

1. **NEW → READY**: Il processo è stato creato e il suo PCB inizializzato. È ora pronto per essere schedulato.
2. **READY → RUNNING**: Lo scheduler seleziona questo processo e gli assegna la CPU.
3. **RUNNING → READY**: Il processo ha esaurito il suo time quantum (preemption) o è stato sospeso per dare priorità a un altro processo.
4. **RUNNING → WAITING**: Il processo ha richiesto un'operazione di I/O o sta attendendo un evento (es. un segnale, il completamento di un altro processo).
5. **WAITING → READY**: L'evento atteso si è verificato (es. l'I/O è completato). Il processo torna nella coda dei processi pronti.
6. **RUNNING → TERMINATED**: Il processo ha completato la sua esecuzione o è stato terminato (volontariamente o a causa di un errore).

3.3 Scheduling della CPU

Uno dei compiti più importanti del sistema operativo è decidere quale processo deve ottenere la CPU in ogni momento. Questo compito è svolto dallo **scheduler della CPU**.

Obiettivi dello Scheduling

Gli obiettivi dello scheduling sono spesso in conflitto tra loro:

- **Throughput**: Massimizzare il numero di processi completati per unità di tempo
- **Tempo di turnaround**: Minimizzare il tempo totale per completare un processo (dal submission al completion)
- **Tempo di risposta**: Minimizzare il tempo che un processo interattivo deve attendere prima di ricevere una risposta
- **Utilizzo della CPU**: Mantenere la CPU occupata il più possibile
- **Fairness**: Assicurare che ogni processo ottenga una quota equa di CPU
- **Predicibilità**: I processi dovrebbero essere eseguiti con latenze prevedibili

Algoritmi di Scheduling

Esistono numerosi algoritmi di scheduling, ciascuno con vantaggi e svantaggi:

1. First-Come, First-Served (FCFS)

Il processo che arriva per primo viene eseguito per primo. Semplice ma inefficiente: se un processo lungo arriva per primo, blocca tutti gli altri (**convoy effect**).

2. Shortest Job First (SJF)

Viene eseguito per primo il processo con il tempo di esecuzione più breve. Ottimale per minimizzare il tempo medio di attesa, ma richiede conoscere in anticipo i tempi di esecuzione (impossibile in pratica) e può causare **starvation** dei processi lunghi.

3. Priority Scheduling

A ogni processo è assegnata una priorità, e la CPU è allocata al processo con priorità più alta. Problema: i processi a bassa priorità possono soffrire di starvation. Soluzione: **aging** (aumentare gradualmente la priorità dei processi che attendono da molto tempo).

4. Round Robin (RR)

Ogni processo riceve un time quantum (tipicamente 10-100 ms). Quando il quantum scade, il processo viene sospeso e rimesso in coda. Eccellente per sistemi time-sharing e processi interattivi, ma overhead per i context switch.

5. Multilevel Queue Scheduling

I processi sono divisi in categorie (es. system processes, interactive processes, batch processes), ognuna con la sua coda e il suo algoritmo di scheduling. Tra le code, si può usare priority scheduling o time slicing.

6. Multilevel Feedback Queue

Simile al multilevel queue, ma i processi possono muoversi tra code in base al loro comportamento. Ad esempio, un processo che usa molto la CPU viene spostato in code a priorità più bassa, mentre processi I/O-bound rimangono in code ad alta priorità. Molto flessibile e usato in sistemi come UNIX.

7. Completely Fair Scheduler (CFS)

Usato in Linux dal 2007, il CFS cerca di dare a ogni processo una quota "giusta" di CPU. Mantiene un albero rosso-nero ordinato per "virtual runtime" (tempo di esecuzione pesato per priorità). Il processo con minor virtual runtime viene eseguito. Molto efficiente e scalabile.

Context Switch

Il **context switch** (cambio di contesto) è l'operazione con cui il sistema operativo sospende un processo in esecuzione e ne avvia (o riprende) un altro.

Durante un context switch:

1. Lo stato del processo corrente (registri, program counter, ecc.) viene salvato nel suo PCB
2. Il PCB del processo da eseguire viene caricato
3. I suoi registri vengono ripristinati
4. Il program counter viene impostato all'istruzione corretta
5. L'esecuzione riprende

Il context switch è **costoso** in termini di performance:

- Richiede la salva/ripristino di tutti i registri
- Può invalidare le cache della CPU
- Richiede l'aggiornamento della MMU per la memoria virtuale
- Tipicamente richiede da 1 a 10 microsecondi

Per questo motivo, un sistema operativo deve bilanciare tra responsiveness (context switch frequenti) ed efficienza (context switch rari).

3.4 Creazione e Terminazione di Processi

Creazione di Processi

In sistemi UNIX-like, i processi sono creati tramite la system call `fork()`. Questa chiamata:

1. Crea un nuovo processo (child) che è una copia quasi esatta del processo chiamante (parent)
2. Il child riceve una copia dello spazio di indirizzamento del parent
3. Entrambi i processi continuano l'esecuzione dall'istruzione successiva alla `fork()`
4. `fork()` restituisce 0 nel child, e il PID del child nel parent

Tipicamente, dopo `fork()` si chiama `exec()` nel processo child per caricare un nuovo programma:

```
pid_t pid = fork();
if (pid == 0) {
    // Codice del processo child
    exec("/bin/ls", "-l", NULL);
```

```
    // Se exec ha successo, questo codice non viene mai raggiunto
} else {
    // Codice del processo parent
    wait(NULL); // Attende il completamento del child
}
```

In Windows, la creazione di processi è differente: `CreateProcess()` crea direttamente un nuovo processo eseguendo un programma specificato, senza la fase di `fork()`.

Terminazione di Processi

Un processo può terminare in diversi modi:

1. **Terminazione normale**: Il processo esegue un'istruzione `exit()` o ritorna da `main()`
2. **Terminazione per errore**: Il processo chiama `exit()` con un codice di errore
3. **Terminazione forzata**: Il processo riceve un segnale che causa la sua terminazione (es. SIGKILL)
4. **Terminazione da parte del parent**: In alcuni casi, un parent può terminare i suoi child

Quando un processo termina, il sistema operativo:

- Rilascia tutte le risorse allocate (memoria, file aperti, ecc.)
- Mantiene alcune informazioni minimali (exit status, statistiche) finché il parent non fa `wait()`
- Se il parent non fa `wait()`, il processo diventa uno **zombie** (terminato ma non ancora rimosso dalla tabella dei processi)
- Se il parent termina prima del child, il child diventa **orfano** e viene adottato dal processo `init` (PID 1)

3.5 Comunicazione tra Processi (IPC)

I processi spesso devono comunicare e sincronizzarsi. Il sistema operativo fornisce diversi meccanismi di **IPC (Inter-Process Communication)**:

1. Pipe

Le **pipe** sono canali di comunicazione unidirezionali tra processi, tipicamente tra parent e child:

```
int fd[2];
pipe(fd); // fd[0] è per leggere, fd[1] per scrivere

if (fork() == 0) {
    // Child: legge dalla pipe
    close(fd[1]);
    read(fd[0], buffer, size);
} else {
    // Parent: scrive nella pipe
    close(fd[0]);
    write(fd[1], data, size);
}
```

Le **named pipe** (FIFO) permettono comunicazione tra processi non correlati.

2. Message Queue

Le **code di messaggi** permettono a processi di scambiarsi messaggi strutturati in modo asincrono. I messaggi sono ordinati e possono avere priorità.

3. Shared Memory

La **memoria condivisa** permette a più processi di accedere alla stessa regione di memoria fisica. È il meccanismo IPC più veloce (nessuna copia di dati), ma richiede sincronizzazione esplicita per evitare race condition.

4. Semafori

I **semafori** sono meccanismi di sincronizzazione che permettono di controllare l'accesso a risorse condivise. Un semaforo è essenzialmente un contatore con due operazioni atomiche:

- **P (wait)**: Decrementa il semaforo. Se diventa negativo, il processo si blocca.
- **V (signal)**: Incrementa il semaforo. Se c'erano processi bloccati, uno viene svegliato.

5. Socket

I **socket** permettono comunicazione tra processi su macchine diverse attraverso la rete, ma possono essere usati anche localmente (UNIX domain sockets).

Parte IV: Gestione della Memoria

4.1 Gerarchia della Memoria

I sistemi di memoria moderni sono organizzati in una gerarchia, con livelli più veloci ma più piccoli vicino alla CPU, e livelli più lenti ma più capienti verso l'esterno:

```
Velocità ↑     Costo/Byte ↑     Capacità ↓

┌──────────────────┐
│  Registri CPU    │  < 1 ns, ~1 KB
├──────────────────┤
│   Cache L1       │  ~1 ns, 32–64 KB per core
├──────────────────┤
│   Cache L2       │  ~5 ns, 256 KB – 1 MB per core
├──────────────────┤
│   Cache L3       │  ~20 ns, 8–64 MB condivisa
├──────────────────┤
│   Memoria RAM    │  ~100 ns, 4–64 GB
├──────────────────┤
│   SSD            │  ~100 µs, 256 GB – 2 TB
├──────────────────┤
│   Hard Disk      │  ~10 ms, 1–10 TB
└──────────────────┘

Velocità ↓     Costo/Byte ↓     Capacità ↑
```

Il sistema operativo deve gestire principalmente la RAM e il disco, ma deve anche essere consapevole delle cache per ottimizzare le performance.

4.2 Allocazione della Memoria

Nei primi sistemi, ogni processo aveva accesso diretto alla memoria fisica. Questo creava problemi:

- Un processo poteva corrompere la memoria di altri processi o del sistema operativo
- La memoria doveva essere allocata in modo contiguo, causando frammentazione
- Era difficile eseguire più programmi contemporaneamente

Partizioni Fisse

Una soluzione primitiva era dividere la memoria in partizioni di dimensione fissa. Ogni processo veniva caricato in una partizione. Problemi:

- Frammentazione interna (spazio sprecato in partizioni troppo grandi)
- Limiti al numero di processi (numero fisso di partizioni)

Partizioni Variabili

Le partizioni potevano avere dimensioni variabili, allocate dinamicamente. Problemi:

- Frammentazione esterna (buchi di memoria tra partizioni allocate)
- Necessità di compattazione periodica (molto costosa)

4.3 Memoria Virtuale

La soluzione moderna ai problemi di gestione della memoria è la **memoria virtuale**, uno dei concetti più brillanti dell'informatica.

Il Concetto

L'idea fondamentale è separare gli **indirizzi virtuali** (usati dai programmi) dagli **indirizzi fisici** (della RAM reale). Ogni processo ha il suo spazio di indirizzamento virtuale, che viene mappato sulla memoria fisica dalla **MMU (Memory Management Unit)**.

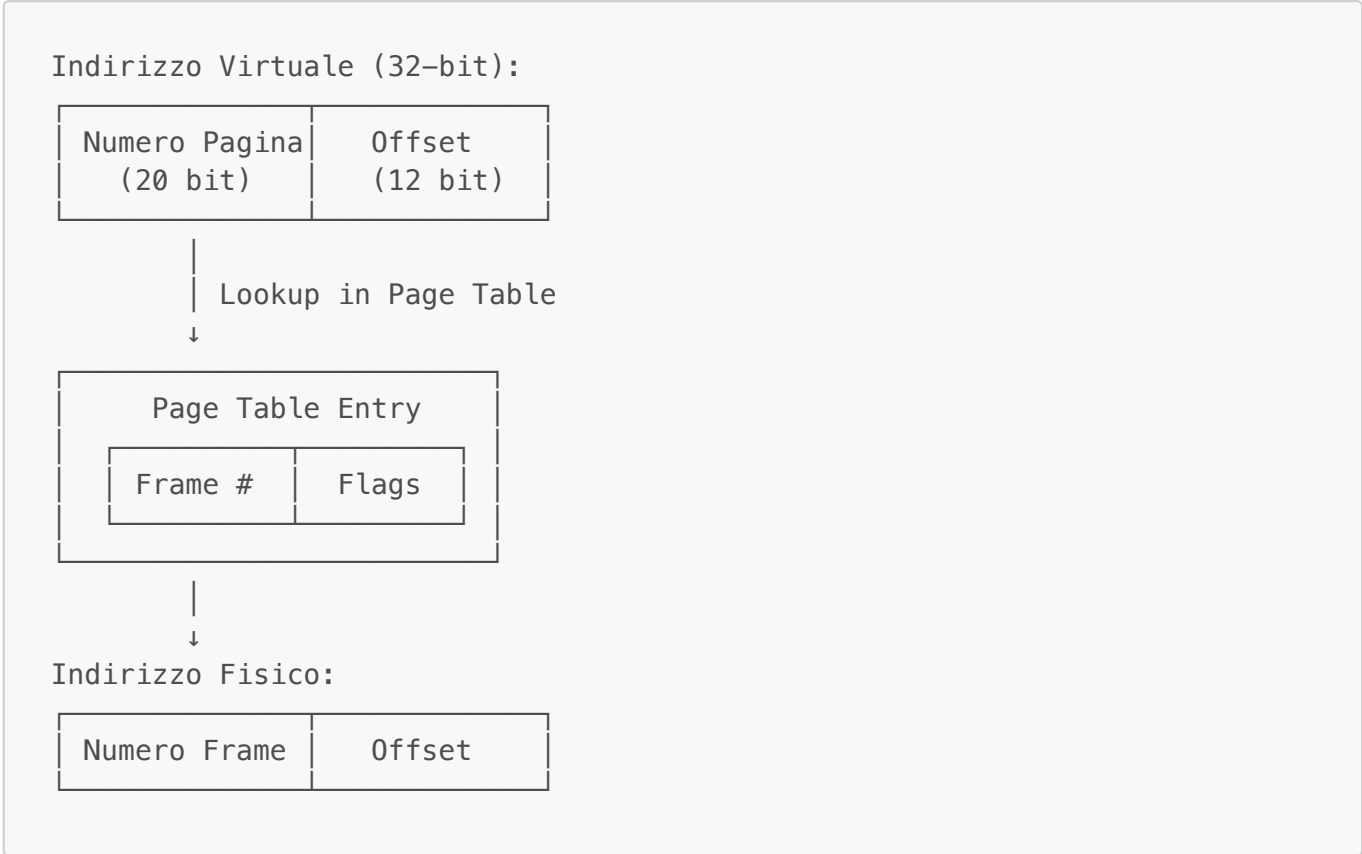
Vantaggi della memoria virtuale:

1. **Isolamento**: Ogni processo ha il suo spazio di indirizzi, non può accedere alla memoria di altri processi
2. **Semplicità**: I programmi possono usare un grande spazio di indirizzi lineare, senza preoccuparsi di dove è fisicamente la memoria
3. **Protezione**: Il sistema operativo può controllare quali pagine sono accessibili a ciascun processo
4. **Condivisione**: Pagine possono essere condivise tra processi (es. librerie condivise)
5. **Memoria apparentemente infinita**: Lo spazio virtuale può essere più grande della RAM fisica, usando il disco come estensione

Paginazione (Paging)

La tecnica più comune per implementare la memoria virtuale è la **paginazione**. Lo spazio di indirizzi virtuale è diviso in **pagine** di dimensione fissa (tipicamente 4 KB), e la memoria fisica in **frame** della stessa dimensione.

La mappatura tra pagine virtuali e frame fisici è mantenuta in una **tabella delle pagine** (page table), una per ogni processo.



Ogni entry della page table contiene:

- **Numero del frame fisico**: Dove si trova la pagina in RAM
- **Present bit**: 1 se la pagina è in RAM, 0 se è su disco
- **Read/Write bit**: Permessi di accesso
- **User/Supervisor bit**: Se accessibile solo in kernel mode
- **Dirty bit**: 1 se la pagina è stata modificata
- **Accessed bit**: 1 se la pagina è stata recentemente usata
- **Cache control bits**: Informazioni per la cache

Page Fault

Quando un processo accede a una pagina virtuale non presente in RAM (present bit = 0), si verifica un **page fault** (eccezione hardware):

1. La MMU genera un'eccezione
2. Il controllo passa al kernel (page fault handler)
3. Il kernel identifica la pagina richiesta
4. Se la pagina è valida ma su disco, il kernel la carica in un frame libero (o ne libera uno)
5. La page table viene aggiornata
6. L'istruzione che ha causato il page fault viene rieseguita

Questo meccanismo permette al sistema operativo di gestire trasparentemente più memoria virtuale di quanta RAM fisica sia disponibile, usando il disco come "memoria secondaria" (swap space).

Translation Lookaside Buffer (TLB)

Consultare la page table per ogni accesso alla memoria sarebbe proibitivamente lento. Per velocizzare la traduzione, le CPU moderne hanno una cache speciale chiamata **TLB (Translation Lookaside Buffer)** che memorizza le traduzioni più recenti.

- **TLB hit**: La traduzione è in cache, velocissima (~1 ciclo)
- **TLB miss**: Bisogna consultare la page table in RAM (~100 cicli)

Il TLB è piccolo (tipicamente 64-512 entry), ma grazie alla località spaziale e temporale degli accessi, ha tipicamente un hit rate > 95%.

4.4 Algoritmi di Sostituzione delle Pagine

Quando si verifica un page fault e tutti i frame sono occupati, il sistema operativo deve scegliere quale pagina rimuovere per fare spazio. La scelta dell'algoritmo di sostituzione è critica per le performance.

1. FIFO (First-In, First-Out)

La pagina più vecchia viene sostituita. Semplice ma inefficiente: una pagina può essere vecchia ma ancora molto usata.

2. Optimal Algorithm

Sostituisce la pagina che non verrà usata per il periodo più lungo nel futuro. Ottimale ma impossibile da implementare (richiede conoscere il futuro). Usato come benchmark teorico.

3. LRU (Least Recently Used)

Sostituisce la pagina che non è stata usata da più tempo. Buone performance, ma costoso da implementare esattamente (richiede timestamp su ogni accesso).

4. Clock (Second Chance)

Approssimazione efficiente di LRU. Le pagine sono in una lista circolare. Quando serve sostituire una pagina:

- Si parte dalla posizione corrente del "puntatore dell'orologio"
- Se il bit "accessed" è 0, la pagina viene sostituita
- Se è 1, viene azzerato e si passa alla prossima pagina
- Si continua finché non si trova una pagina con accessed = 0

5. Working Set

Si basa sul concetto di **working set**: l'insieme di pagine usate da un processo in un recente intervallo di tempo. Si cerca di mantenere in memoria il working set di ciascun processo per minimizzare i page fault.

4.5 Thrashing

Il **thrashing** si verifica quando un sistema passa più tempo a fare paging (spostare pagine tra RAM e disco) che a eseguire codice utile. Accade quando:

- Troppi processi sono in esecuzione contemporaneamente
- Il working set totale eccede la RAM disponibile
- Il sistema entra in un ciclo vizioso: page fault continui → caricamento pagine → page fault di altre pagine

Soluzioni:

- Ridurre il numero di processi attivi (suspend alcuni processi)
- Aumentare la RAM
- Migliorare la località degli accessi
- Usare algoritmi di page replacement migliori

Parte V: File System

5.1 Concetti Fondamentali

Il **file system** è il componente del sistema operativo responsabile dell'organizzazione e gestione dei dati su dispositivi di memorizzazione persistente (dischi, SSD).

Cos'è un File

Un **file** è un'astrazione per rappresentare dati memorizzati. Un file ha:

- **Nome**: Identificativo human-readable
- **Tipo**: Estensione o metadata che indica il tipo di contenuto
- **Contenuto**: I dati effettivi
- **Attributi**: Metadata come dimensione, timestamp, permessi, proprietario

Directory

Le **directory** (cartelle) organizzano i file in una struttura gerarchica. Una directory è essenzialmente un file speciale che contiene una lista di file e altre directory.

La struttura più comune è l'**albero di directory**:

```
/                           (root)
├── bin/                    (programmi eseguibili di sistema)
├── etc/                    (file di configurazione)
├── home/                   (directory home degli utenti)
│   ├── alice/
│   └── bob/
├── usr/                    (programmi e dati utente)
│   ├── bin/
│   ├── lib/
│   └── local/
```

```
└── var/                    (dati variabili, log)
    ├── log/
    └── tmp/
```

Path

Un **path** (percorso) identifica univocamente un file nel file system:

- **Absolute path**: Parte dalla root, es. `/home/alice/documents/report.pdf`
- **Relative path**: Parte dalla directory corrente, es. `../bob/photos/sunset.jpg`

5.2 Operazioni sui File

Il sistema operativo fornisce system call per manipolare file:

- **open()**: Apre un file, restituisce un file descriptor
- **read()**: Legge dati dal file
- **write()**: Scrive dati nel file
- **seek()**: Sposta il puntatore di lettura/scrittura
- **close()**: Chiude il file, rilascia risorse
- **delete()**: Elimina il file
- **truncate()**: Riduce la dimensione del file

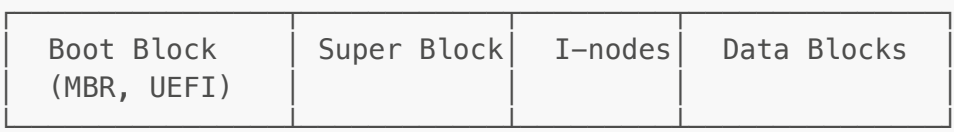
Ogni processo ha una **tabella dei file aperti**, mantenuta dal kernel, che associa file descriptor a file effettivi.

5.3 Implementazione del File System

A livello fisico, un disco è una sequenza di **blocchi** (o settori) di dimensione fissa (tipicamente 512 byte o 4 KB). Il file system deve mappare la struttura logica di file e directory su questi blocchi fisici.

Struttura del Disco

Un disco tipicamente è organizzato così:



- **Boot Block**: Contiene il boot loader per avviare il sistema operativo
- **Super Block**: Contiene metadata del file system (dimensione, numero di blocchi liberi, ecc.)
- **I-node table**: Array di strutture dati, una per ogni file
- **Data Blocks**: I blocchi che contengono effettivamente i dati dei file

I-node

In file system come ext4 (Linux) e UFS (Unix), ogni file è rappresentato da un **i-node** (index node), una struttura dati che contiene:

- **Attributi del file**: dimensione, permessi, timestamps, tipo
- **Puntatori a blocchi dati**: Indirizzi dei blocchi su disco che contengono i dati del file

Per file piccoli, i puntatori diretti nell'i-node sono sufficienti. Per file grandi, si usa una struttura multi-livello:

- **Puntatori diretti**: Puntano direttamente ai blocchi dati (es. 12 puntatori)
- **Puntatore indiretto singolo**: Punta a un blocco che contiene altri puntatori a blocchi dati
- **Puntatore indiretto doppio**: Punta a un blocco di puntatori a blocchi di puntatori
- **Puntatore indiretto triplo**: Un ulteriore livello di indirezione

Questo schema permette di rappresentare file di dimensione arbitraria mantenendo l'i-node di dimensione fissa.

Allocazione dei Blocchi

Il file system deve decidere come allocare blocchi ai file. Le strategie principali sono:

1. Allocazione Contigua

I blocchi di un file sono memorizzati in sequenza. Vantaggi: semplice, accesso veloce. Svantaggi: frammentazione esterna, difficile estendere file.

2. Allocazione Collegata (Linked)

Ogni blocco contiene un puntatore al blocco successivo (come una linked list). Vantaggi: nessuna frammentazione esterna. Svantaggi: accesso sequenziale lento, i puntatori occupano spazio.

3. Allocazione Indicizzata

Tutti i puntatori ai blocchi sono in un'unica struttura (l'i-node). È l'approccio più comune e flessibile.

5.4 Free Space Management

Il file system deve tenere traccia dei blocchi liberi. Tecniche comuni:

1. Bitmap

Un bit per ogni blocco: 0 = libero, 1 = occupato. Semplice e compatto, ma richiede scansione per trovare blocchi liberi.

2. Lista Collegata

I blocchi liberi formano una linked list. Efficiente per allocare un singolo blocco, ma lento per allocazioni grandi.

3. Gruppi di Blocchi Contigui

Si memorizza l'indirizzo e la lunghezza di ogni gruppo di blocchi contigui liberi. Equilibrio tra spazio e velocità.

5.5 Journaling

Un problema critico dei file system è la **consistenza** dopo un crash di sistema. Se il sistema si blocca nel mezzo di un'operazione (es. scrittura di un file), il file system può rimanere in uno stato inconsistente.

I **journaling file system** (come ext4, NTFS) risolvono questo problema con un **journal** (log):

1. Prima di modificare il file system, le operazioni sono scritte nel journal
2. Poi vengono effettivamente applicate al file system
3. Infine, vengono marcate come completate nel journal

In caso di crash, al riavvio il sistema può esaminare il journal:

- Operazioni incomplete vengono scartate (rollback) o completate (rollforward)
- Il file system torna rapidamente a uno stato consistente

Questo approccio sacrifica un po' di performance per guadagnare robustezza e tempi di recovery rapidi.

5.6 Virtual File System (VFS)

I sistemi operativi moderni supportano molteplici file system (ext4, NTFS, FAT32, XFS, Btrfs, ecc.). Per evitare che ogni applicazione debba conoscere i dettagli di ogni file system, Linux (e altri OS) implementano un **Virtual File System (VFS)**.

Il VFS è uno strato di astrazione che:

- Presenta un'interfaccia uniforme alle applicazioni
- Traduce le operazioni generiche in operazioni specifiche del file system sottostante
- Permette di montare diversi file system in un'unica gerarchia di directory

Grazie al VFS, è possibile accedere trasparentemente a file su ext4, su una partizione Windows (NTFS), su una chiavetta USB (FAT32), e persino su file system di rete (NFS, SMB) usando le stesse system call.

Parte VI: Gestione dell'I/O

6.1 Hardware di I/O

I dispositivi di I/O sono estremamente vari: dischi, tastiere, mouse, stampanti, schede di rete, GPU, ecc. Tuttavia, condividono alcuni concetti comuni.

Tipologie di Dispositivi

Dispositivi a Blocchi

Memorizzano informazioni in blocchi di dimensione fissa (es. dischi, SSD). Ogni blocco può essere letto o scritto indipendentemente. Supportano accesso random.

Dispositivi a Caratteri

Forniscono o accettano un flusso di caratteri senza struttura a blocchi (es. tastiere, stampanti, porte seriali). Tipicamente accesso sequenziale.

Interfaccia Hardware

I dispositivi comunicano con la CPU attraverso:

1. I/O Mappato in Memoria (Memory-Mapped I/O)

I registri di controllo del dispositivo sono mappati nello spazio di indirizzi. Leggere/scrivere in questi indirizzi comunica con il dispositivo.

2. I/O Isolato (Port-Mapped I/O)

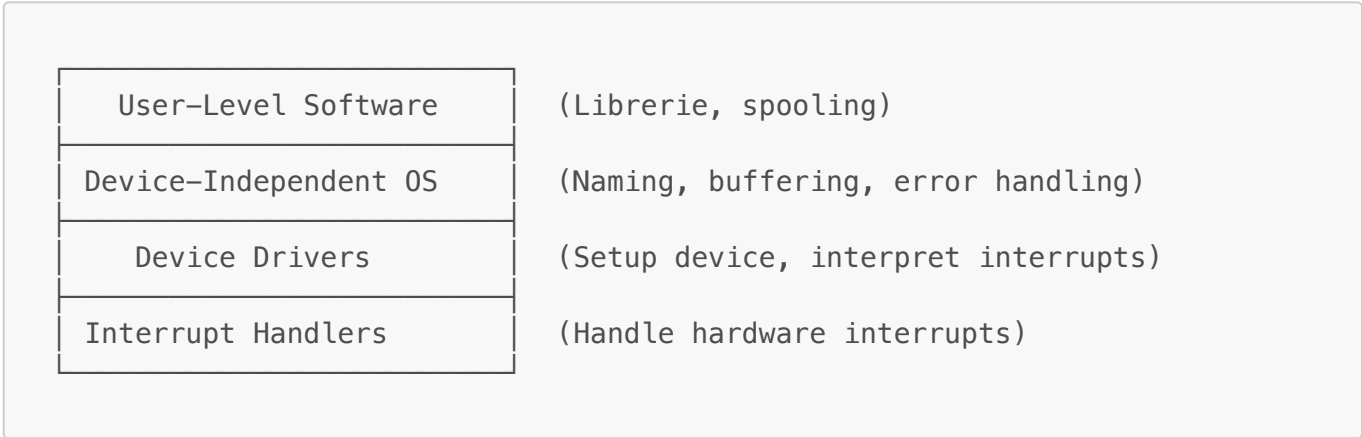
Istruzioni speciali (IN, OUT su x86) accedono a un address space separato per l'I/O.

3. DMA (Direct Memory Access)

Per trasferimenti di dati grandi, il dispositivo accede direttamente alla memoria senza coinvolgere la CPU, che viene notificata solo al completamento via interrupt.

6.2 Software di I/O

Il software di I/O è organizzato in layer:



Interrupt Handling

Quando un dispositivo completa un'operazione, genera un **interrupt** hardware:

1. La CPU salva il contesto corrente
2. Passa il controllo all'**interrupt handler** appropriato (parte del driver)
3. L'handler gestisce l'interrupt (es. legge dati dal dispositivo, aggiorna buffer)
4. La CPU riprende l'esecuzione normale

Gli interrupt permettono al sistema di rispondere rapidamente agli eventi hardware senza polling continuo (che spreca cicli di CPU).

Device Driver

Il **driver** è il componente software che conosce i dettagli specifici di un dispositivo:

- Inizializza il dispositivo
- Traduce richieste generiche in comandi specifici del dispositivo
- Gestisce gli interrupt del dispositivo
- Gestisce errori e recovery

I driver sono spesso il codice più buggy del kernel, perché sono numerosi, complessi, e scritti da vari vendor con qualità variabile.

Buffering

Il **buffering** è essenziale nell'I/O:

- **Unbuffered I/O**: Ogni operazione va direttamente al dispositivo (inefficiente)
- **Buffering singolo**: Un buffer intermedio tra programma e dispositivo
- **Double buffering**: Due buffer, uno si riempie mentre l'altro viene processato
- **Buffering circolare**: Array di buffer gestito come coda circolare

Il buffering mitiga la differenza di velocità tra dispositivi e CPU, e permette operazioni asincrone.

6.3 I/O Sincrono vs Asincrono

I/O Sincrono (Blocking I/O)

Il processo che richiede I/O si blocca finché l'operazione non è completata:

```
read(fd, buffer, size);  // Il processo si blocca qui
// Continua solo quando i dati sono disponibili
process_data(buffer);
```

Semplice da programmare, ma inefficiente: il processo è idle durante l'I/O.

I/O Asincrono (Non-blocking I/O)

Il processo avvia l'operazione e continua immediatamente:

```
aio_read(fd, buffer, size, &callback);
// Il processo continua mentre l'I/O è in corso
do_other_work();
// Verrà chiamato callback quando l'I/O è completo
```

Più complesso ma molto più efficiente, specialmente per server che gestiscono molte connessioni.

I/O Multiplexing

Meccanismi come `select()`, `poll()`, e `epoll()` permettono a un processo di attendere su multipli file descriptor:

```
// Attende finché almeno un fd è pronto per I/O
select(max_fd, &read_fds, &write_fds, &except_fds, &timeout);
// Poi processa solo gli fd pronti
```

Fondamentale per server ad alte prestazioni (es. web server, database).

Parte VII: Concetti Avanzati

7.1 Deadlock

Un **deadlock** si verifica quando un insieme di processi è bloccato, ognuno in attesa di una risorsa detenuta da un altro processo nell'insieme. Nessuno può procedere.

Condizioni Necessarie per il Deadlock

Affinché si verifichi un deadlock, devono essere soddisfatte simultaneamente quattro condizioni (condizioni di Coffman):

1. **Mutua Esclusione**: Almeno una risorsa deve essere non condivisibile (solo un processo alla volta può usarla)
2. **Hold and Wait**: Processi che detengono risorse possono richiederne altre
3. **No Preemption**: Le risorse non possono essere forzatamente tolte a un processo
4. **Attesa Circolare**: Esiste un ciclo di processi, ognuno in attesa di una risorsa detenuta dal successivo

Strategie per Gestire i Deadlock

1. Prevenzione

Garantire che almeno una delle quattro condizioni non possa verificarsi. Ad esempio:

- Negare "Hold and Wait": Richiedere tutte le risorse all'inizio
- Negare "No Preemption": Permettere il rilascio forzato di risorse
- Negare "Attesa Circolare": Imporre un ordinamento totale sulle risorse

2. Evitamento

Il sistema decide dinamicamente se concedere una richiesta di risorsa, basandosi su informazioni sulle risorse disponibili e sulle necessità future. L'algoritmo del **banchiere** (Dijkstra) è un esempio classico.

3. Rilevamento e Recovery

Permettere che i deadlock si verifichino, rilevarli (es. tramite analisi del grafo di allocazione delle risorse), e poi risolverli (terminando processi o preemptando risorse).

4. Ignoranza (Algoritmo dello Struzzo)

Non fare nulla, assumendo che i deadlock siano rari. Riavviare il sistema se si verifica un deadlock. Approccio poco elegante ma pragmatico, usato da molti sistemi reali (es. UNIX).

7.2 Sincronizzazione

Quando più thread o processi accedono a dati condivisi, è fondamentale sincronizzare gli accessi per evitare **race condition** (risultati dipendenti dall'ordine temporale imprevedibile delle operazioni).

Sezione Critica

Una **sezione critica** è una porzione di codice che accede a risorse condivise e deve essere eseguita atomicamente (senza interruzioni). La soluzione al problema della sezione critica deve garantire:

1. **Mutua Esclusione**: Un solo processo alla volta nella sezione critica
2. **Progress**: Se nessuno è nella sezione critica e qualcuno vuole entrare, la decisione su chi entra non può essere posticipata indefinitamente
3. **Bounded Waiting**: Esiste un limite al numero di volte che altri processi possono entrare prima che un processo in attesa entri

Meccanismi di Sincronizzazione

1. Mutex (Mutual Exclusion)

Un lock binario: bloccato/sbloccato. Un thread acquisisce il mutex prima di entrare nella sezione critica e lo rilascia all'uscita.

```
pthread_mutex_lock(&mutex);
// Sezione critica
shared_variable++;
pthread_mutex_unlock(&mutex);
```

2. Semafori

Generalizzazione del mutex, un contatore che controlla l'accesso a un pool di risorse.

3. Condition Variables

Permettono a thread di attendere finché una certa condizione non diventa vera, rilasciando atomicamente un mutex durante l'attesa.

4. Monitor

Costrutto di alto livello che incapsula dati condivisi e le procedure per accedervi, garantendo mutua esclusione automaticamente.

5. Read-Write Lock

Permette accesso concorrente per lettori, ma esclusivo per scrittori. Efficiente quando le letture sono molto più comuni delle scritture.

7.3 Thread vs Processi

I **thread** sono "processi leggeri" che condividono lo stesso spazio di indirizzamento e risorse, ma hanno stack e registri separati.

Vantaggi dei Thread

- **Creazione più veloce**: Non serve duplicare lo spazio di indirizzamento
- **Context switch più veloce**: Meno stato da salvare/ripristinare
- **Comunicazione efficiente**: Condivisione diretta della memoria

- **Scalabilità**: Sfruttano CPU multi-core

Svantaggi dei Thread

- **Complessità**: Necessità di sincronizzazione esplicita
- **Bug sottili**: Race condition, deadlock sono difficili da debug
- **Meno isolamento**: Un thread malfunzionante può corrompere l'intero processo

7.4 Virtualizzazione

La **virtualizzazione** permette di eseguire più sistemi operativi contemporaneamente sulla stessa macchina fisica.

Hypervisor

Un **hypervisor** (o Virtual Machine Monitor) è il software che crea e gestisce macchine virtuali:

Type 1 (Bare-metal): Gira direttamente sull'hardware (es. VMware ESXi, Xen, Hyper-V). Più efficiente.

Type 2 (Hosted): Gira sopra un sistema operativo host (es. VirtualBox, VMware Workstation). Più flessibile.

Tecniche di Virtualizzazione

1. Full Virtualization

L'intero hardware è virtualizzato. I guest OS non sono consapevoli di girare su una VM. Richiede supporto hardware (Intel VT-x, AMD-V).

2. Paravirtualization

I guest OS sono modificati per essere consapevoli della virtualizzazione e cooperare con l'hypervisor. Più efficiente ma richiede modifiche al kernel.

3. Container (OS-level Virtualization)

Multipli ambienti isolati condividono lo stesso kernel (es. Docker, LXC). Molto leggeri ed efficienti, ma meno isolamento di una VM completa.

7.5 Sicurezza

Il sistema operativo è la prima linea di difesa per la sicurezza del sistema.

Principi di Sicurezza

1. Principio del Minimo Privilegio

Ogni processo dovrebbe avere solo i permessi minimi necessari per svolgere il suo compito.

2. Defense in Depth

Multipli layer di sicurezza: anche se uno fallisce, altri proteggono il sistema.

3. Fail-Safe Defaults

In caso di dubbio, negare l'accesso (closed by default).

Meccanismi di Protezione

1. Autenticazione

Verifica dell'identità (username/password, biometrica, multi-factor).

2. Controllo degli Accessi

- **DAC (Discretionary Access Control)**: Il proprietario di una risorsa decide chi può accedervi (es. permessi UNIX)
- **MAC (Mandatory Access Control)**: Policy centralizzate, ignorate dai singoli utenti (es. SELinux)
- **RBAC (Role-Based Access Control)**: Permessi assegnati a ruoli, utenti assegnati a ruoli

3. Isolamento

- Address Space Layout Randomization (ASLR)
- Data Execution Prevention (DEP/NX bit)
- Sandboxing
- Secure Boot

Conclusioni

Il sistema operativo è uno dei componenti software più complessi e fondamentali dell'informatica moderna. È il ponte tra l'hardware grezzo e le applicazioni che usiamo quotidianamente, orchestrando risorse, garantendo sicurezza, e fornendo astrazioni che semplificano enormemente lo sviluppo software.

Abbiamo esplorato come un sistema operativo:

- **Astrare l'hardware**, presentando interfacce uniformi e semplificate
- **Gestire i processi**, schedulandoli, proteggendoli, e permettendo loro di comunicare
- **Amministrare la memoria**, usando tecniche sofisticate come la memoria virtuale e la paginazione per creare l'illusione di risorse illimitate
- **Organizzare i dati**, attraverso file system che bilanciano performance, affidabilità e flessibilità
- **Coordinare l'I/O**, gestendo dispositivi eterogenei in modo efficiente
- **Prevenire e gestire problemi** come deadlock e race condition
- **Garantire sicurezza e isolamento**, proteggendo utenti e processi l'uno dall'altro

Comprendere il funzionamento di un sistema operativo è essenziale per ogni informatico, che sia un programmatore di applicazioni (per scrivere codice efficiente e corretto), un amministratore di sistema (per configurare e ottimizzare i sistemi), o un ricercatore (per sviluppare nuove tecnologie).

I concetti che abbiamo esplorato - processi, memoria virtuale, file system, sincronizzazione - sono la base su cui si costruisce tutto il software moderno. Continuano a evolversi (si pensi ai sistemi operativi per il cloud, per dispositivi IoT, o per architetture esotiche come i computer quantistici), ma i principi fondamentali rimangono sorprendentemente stabili e rilevanti.

Lo studio dei sistemi operativi non è solo lo studio di Linux, Windows o macOS, ma lo studio di come organizzare e gestire complessità, come bilanciare competing goals (performance vs sicurezza, semplicità

vs flessibilità), e come creare astrazioni robuste e durature. È, in definitiva, uno studio di architettura software al suo livello più fondamentale.

Bibliografia e Approfondimenti

Per approfondire gli argomenti trattati in questa lezione, si consigliano i seguenti testi:

1. **Silberschatz, A., Galvin, P. B., & Gagne, G.** - "Operating System Concepts" (10th Edition)
   Il testo di riferimento per eccellenza, completo e rigoroso.

2. **Tanenbaum, A. S., & Bos, H.** - "Modern Operating Systems" (4th Edition)
   Eccellente bilanciamento tra teoria e pratica, con molti esempi da sistemi reali.

3. **Love, R.** - "Linux Kernel Development" (3rd Edition)
   Per chi vuole comprendere un sistema operativo reale dall'interno.

4. **Stevens, W. R., & Rago, S. A.** - "Advanced Programming in the UNIX Environment" (3rd Edition)
   La bibbia della programmazione di sistema in ambiente UNIX/Linux.

5. **Arpaci-Dusseau, R. H., & Arpaci-Dusseau, A. C.** - "Operating Systems: Three Easy Pieces"
   Approccio moderno e accessibile, disponibile gratuitamente online.

Per esercitarsi e sperimentare, si consiglia di:

- Analizzare il codice sorgente di sistemi operativi open source (Linux, FreeBSD, xv6)
- Implementare componenti semplificati (scheduler, memory allocator, mini file system)
- Usare tool di system programming (strace, gdb, valgrind, perf)
- Studiare vulnerabilità e bug storici per capire cosa può andare storto

La comprensione profonda di un sistema operativo richiede tempo, pazienza, e molto hands-on experience. Ogni concetto che appare semplice sulla carta rivela complessità inaspettate quando lo si implementa realmente. Questo è parte della bellezza e della sfida dell'informatica di sistema.