



# ARGOMENTI

Adesso stileremo una lista di tutti gli argomenti del corso, basandoci sul libro *Sistemi Operativi, Concetti ed esempi, X Edizione, Silberschatz-Galvin-Gagne*.

## CAPITOLO 3 - PROCESSI

### ▼ 3.1 - Concetto di Processo 75%

- 3.1.1 - Il Processo 
- 3.1.2 - Stato del Processo 
- 3.1.3 - Blocco di Controllo del Processo 
- 3.1.4 - Thread 

### ▼ 3.2 - Scheduling dei Processi 100%

- 3.2.1 - Code di Scheduling 
- 3.2.2 - Scheduling della CPU 
- 3.2.3 - Cambio di Contesto 

### ▼ 3.3 - Operazioni Sui Processi 100%

- 3.3.1 - Creazione di un Processo 
- 3.3.2 - Terminazione di un Processo 

### ▼ 3.4 - Comunicazione tra Processi 100%

### ▼ 3.5 - IPC in Sistemi a Memoria Condivisa 100%

### ▼ 3.6 - IPC in Sistemi a Scambio di Messaggi 100%

- 3.6.1 - Naming 

- 3.6.2 - Sincronizzazione✓
- 3.6.3 - Code di messaggi✓

#### ▼ 3.7 - Esempi di Sistemi IPC 0%

- 3.7.1 - Memoria Condivisa in POSIX-
- 3.7.2 - Mach-
- 3.7.3 - Windows-
- 3.7.4 - Pipe-

#### ▼ 3.8 - Comunicazione nei Sistemi Client-Server 100%

- 3.8.1 - Socket✓
- 3.8.2 - Chiamate di Procedure Remote✓

## CAPITOLO 4 - THREAD E CONCORRENZA

#### ▼ 4.1 - Introduzione 100%

- Motivazioni✓
- Vantaggi✓

#### ▼ 4.2 - Programmazione Multicore 50%

- Le Sfide della Programmazione-
- Tipi di Parallelismo✓

#### ▼ 4.3 - Modelli di Supporto al Multithreading 100%

- Modello da Molti a Uno✓
- Modello da Uno a Uno✓
- Modello da Molti a Molti✓

#### ▼ 4.4 - Librerie dei Thread 0%

- Pthreads-
- Thread in Windows-
- Thread in Java-

## ▼ 4.5 - Threading Implicito 0%

- Gruppi di Thread
- Fork Join
- OpenMP
- Grand Central Dispatch
- Intel Thread Building Blocks

## ▼ 4.6 - Problematiche di Programmazione Multithread 0%

- Chiamate di sistema `fork()` ed `exec()`
- Gestione dei Segnali
- Cancellazione dei Thread
- Dati Locali dei Thread
- Attivazione dello Scheduler

## ▼ 4.7 - Esempi di Sistemi Operativi 0%

- Thread di Windows
- Thread di Linux

# CAPITOLO 5 - SCHEDULING DELLA CPU

## ▼ 5.1 - Concetti Fondamentali 100%

- Ciclicità delle Fasi di Elaborazione e di I/O
- Scheduler della CPU
- Scheduling con e senza Prelazione
- Dispatcher

## ▼ 5.2 - Criteri di Scheduling 100%

## ▼ 5.3 - Algoritmi di Scheduling 100%

- Scheduling in Ordine d'Arrivo

- Scheduling Shortes-Job-First ✓
- Scheduling Circolare ✓
- Scheduling con Priorità ✓
- Scheduling a Code Multilivello ✓
- Scheduling a Code Multilivello con Retroazione ✓

#### ▼ 5.4 - Scheduling dei Thread 0%

- Ambito della Contesa -
- Scheduling di PThread -

#### ▼ 5.5 - Scheduling per Sistemi Multiprocessore 100%

- Approcci allo Scheduling per Multiprocessori ✓
- Processori Multicore ✓
- Bilanciamento del Carico ✓
- Predilezione per il Processore ✓
- Multiprocessing Eterogeneo ✓

#### ▼ 5.6 - Scheduling Real-Time della CPU 16.6%

- Minimizzazione della Latenza ✓
- Scheduling Basato sulla Priorità -
- Scheduling con Priorità Proporzionale alla Frequenza -
- Scheduling EDF -
- Scheduling a Quota Proporzionali -
- Scheduling Real-Time di POSIX -

#### ▼ 5.7 - Esempi di Sistemi operativi 33.3%

- Un Esempio: Scheduling di Linux ✓
- Un Esempio: Scheduling di Windows -
- Un Esempio: Scheduling di Solaris -

#### ▼ 5.8 - Valutazione degli Algoritmi 100%

- Modellazione Deterministica ✓
- Reti di Code ✓
- Simulazioni ✓
- Realizzazione ✓

## CAPITOLO 6 - STRUMENTI DI SINCRONIZZAZIONE

### ▼ 6.1 - Introduzione 100%

### ▼ 6.2 - Problema della Sezione Critica 100%

### ▼ 6.3 - Soluzione di Peterson 100%

### ▼ 6.4 - Supporto Hardware per la Sincronizzazione 100%

- Barriere di Memoria ✓
- Istruzioni Hardware ✓
- Variabili Atomiche ✓

### ▼ 6.5 - Lock mutex 100%

### ▼ 6.6 - Semafori 100%

- Uso dei Semafori ✓
- Implementazione dei Semafori ✓

### ▼ 6.7 - Monitor 33.3%

- Uso del Costrutto Monitor ✓
- Realizzazione di un Monitor per Mezzo di Semafori -
- Ripresa dei Processi all'Interno di un Monitor -

### ▼ 6.8 - Liveness 100%

- Stallo ✓
- Inversione di Priorità ✓

▼ 6.9 - Valutazione delle Soluzioni 100%

## CAPITOLO 7 - ESEMPI DI SINCRONIZZAZIONE

▼ 7.1 - Classici Problemi di Sincronizzazione 100%

- Produttore/Consumatore con Memoria Limitata ✓
- Problema dei Lettori-Scrittori ✓
- Problema dei Cinque Filosofi (dining philosophers) ✓

▼ 7.2 - Sincronizzazione all'Interno del Kernel 0%

- Sincronizzazione in Windows └
- Sincronizzazione dei Processi in Linux └

▼ 7.3 - Sincronizzazione POSIX 0%

- Lock Mutex POSIX └
- Semafori POSIX └
- Variabili Condizionali in POSIX └

▼ 7.4 - Sincronizzazione in Java 0%

- Monitor Java └
- Lock Rientranti └
- Semafori └
- Variabili Condizionali └

▼ 7.5 - Approcci Alternativi 0%

- Memoria Transazionale └
- OpenMP └
- Linguaggi di Programmazione Funzionali └

# CAPITOLO 8 - STALLO DEI PROCESSI

## ▼ 8.1 - Modello di Sistema 100%

## ▼ 8.2 - Situazioni di Stallo in Applicazioni Multithread 100%

- Situazioni di stallo attivo (livelock) ✓

## ▼ 8.3 - Caratterizzazione delle Situazioni di Stallo 100%

- Condizioni necessarie ✓
- Grafo di assegnazione delle risorse ✓

## ▼ 8.4 - Metodi per la Gestione delle Situazioni di Stallo 100%

## ▼ 8.5 - Prevenzione delle Situazioni di Stallo 100%

- Mutua Esclusione ✓
- Possesso e attesa ✓
- Assenza di prelazione ✓
- Attesa circolare ✓

## ▼ 8.6 - Evitare le Situazioni di Stallo 100%

- Stato sicuro ✓
- Algoritmo con grafo di assegnazione delle risorse ✓
- Algoritmo del banchiere ✓

## ▼ 8.7 - Rilevamento delle Situazioni di Stallo 100%

- Istanza singola di ciascun tipo di risorse ✓
- Più istanze di ciascun tipo di risorsa ✓
- Uso dell'algoritmo di rilevamento ✓

## ▼ 8.8 - Ripristino da Situazione di Stallo 100%

- Terminazione dei processi e thread ✓

- Prelazione delle risorse ✓

## CAPITOLO 9 - MEMORIA CENTRALE

### ▼ 9.1 - Introduzione 100%

- Hardware di base ✓
- Associazione degli indirizzi ✓
- Spazi di indirizzi logici e fisici ✓
- Caricamento dinamico ✓
- Linking dinamico e librerie condivise ✓

### ▼ 9.2 - Allocazione Contigua della Memoria 100%

- Protezione della memoria ✓
- Allocazione della memoria ✓
- Frammentazione ✓

### ▼ 9.3 - Paginazione 100%

- Metodo di base ✓
- Supporto hardware alla paginazione ✓
- Protezione ✓
- Pagine condivise ✓

### ▼ 9.4 - Struttura della Tabella delle Pagine 75%

- Paginazione gerarchica ✓
- Tabella delle pagine di tipo hash ✓
- Tabella delle pagine invertita ✓
- Oracle SPARC Solaris -

### ▼ 9.5 - Avvicendamento dei Processi (Swapping) 100%

- Avvicendamento standard ✓
- Avvicendamento con paginazione ✓

- Avvicendamento di processi nei sistemi mobili✓

▼ 9.6 - Esempio: le Architetture Intel a 32 e 64 bit 0%

- Architettura IA-32-
- Architettura x86-64-

▼ 9.7 - Esempio: Architettura ARMv8 0%

## CAPITOLO 10 - MEMORIA VIRTUALE

▼ 10.1 - Introduzione 100%

▼ 10.2 - Paginazione su Richiesta 100%

- Concetti fondamentali✓
- Lista dei frame liberi✓
- Prestazioni della paginazione su richiesta✓

▼ 10.3 - Copiatura su Scrittura 100%

▼ 10.4 - Sostituzione delle Pagine 87.5%

- Sostituzione di pagina✓
- Sostituzione delle pagine secondo l'ordine di arrivo (FIFO)✓
- Sostituzione ottimale delle pagine✓
- Sostituzione delle pagine usate meno recentemente (LRU)✓
- Sostituzione delle pagine per approssimazione a LRU✓
- Sostituzione delle pagine basata su conteggio✓
- Algoritmi con buffering delle pagine✓
- Applicazioni e sostituzione della pagina-

▼ 10.5 - Allocazione dei Frame 75%

- Numero minimo di frame ✓
- Algoritmi di allocazione ✓
- Allocazione globale e allocazione locale ✓
- Accesso non uniforme alla memoria -

#### ▼ 10.6 - Thrashing 100%

- Cause del thrashing ✓
- Modello del working-set ✓
- Frequenza dei page-fault ✓
- Regole correntemente adottate ✓

#### ▼ 10.7 - Compressione della Memoria 0%

#### ▼ 10.8 - Allocazione di Memoria del Kernel 0%

- Sistema buddy -
- Allocazione a lastre -

#### ▼ 10.9 - Altre Considerazioni 0%

- Prepaginazione -
- Dimensione delle pagine -
- Portata del TLB -
- Tabella delle pagine invertita -
- Struttura dei programmi -
- Vincolo di I/O e vincolo delle pagine -

#### ▼ 10.10 - Esempi di Sistemi Operativi 0%

- Linux -
- Windows -
- Solaris -

# CAPITOLO 11 - MEMORIA DI MASSA



## ▼ 11.1 - Struttura dei Dispositivi di Memorizzazione 100%

- Dischi rigidi ✓
- Dispositivi NVM ✓
- Memoria volatile ✓
- Metodi di connessione alla memoria secondaria ✓
- Mappatura degli indirizzi ✓

## ▼ 11.2 - Scheduling dei Dischi Rigidi 100%

- Scheduling in ordine d'arrivo - FCFS ✓
- Scheduling - SCAN ✓
- Scheduling - C-SCAN ✓
- Scelta di un algoritmo di scheduling ✓

## ▼ 11.3 - Scheduling dei Dispositivi NVM 0%

## ▼ 11.4 - Rilevamento e Correzione di Errori 0%

## ▼ 11.5 - Gestione delle Unità di Memoria Secondaria 100%

- Formattazione del disco, partizioni e volumi ✓
- Blocco d'avviamento ✓
- Blocchi difettosi ✓

## ▼ 11.6 - Gestione dell'Area d'Avvicendamento 66 . 6%

- Uso dell'area di avvicendamento ✓
- Collocazione dell'area d'avvicendamento ✓
- Gestione dell'area d'avvicendamento: un esempio ➔

## ▼ 11.7 - Connessione dei Dispositivi di Memorizzazione 0%

- Memoria secondaria connessa alla macchina
- Memoria secondaria connessa alla rete
- Memoria secondaria su cloud
- Storage-area network e storage array

#### ▼ 11.8 - Strutture RAID 85.7%

- Miglioramento dell'affidabilità tramite la ridondanza
- Miglioramento delle prestazioni tramite il parallelismo
- Livelli RAID
- Scelta di un livello RAID
- Estensioni
- Problemi connessi a RAID
- Object Storage

## CAPITOLO 12 - SISTEMI DI I/O

#### ▼ 12.1 - Introduzione 100%

#### ▼ 12.2 - Hardware di I/O 100%

- I/O memory mapped
- Polling
- Interruzioni
- Accesso diretto alla memoria (DMA)
- Concetti principali dell'hardware di I/O

#### ▼ 12.3 - Interfaccia di I/O delle Applicazioni 80%

- Dispositivi con trasferimento a blocchi e a caratteri
- Dispositivi di rete
- Orologi e timer

- I/O non bloccante e asincrono ✓
- I/O vettorizzato -

#### ▼ 12.4 - Sottosistema di I/O del Kernel 66.6%

- Scheduling dell'I/O ✓
- Gestione dei buffer ✓
- Cache ✓
- Code di spooling e riservazione dei dispositivi ✓
- Gestione degli errori ✓
- Protezione dell'I/O -
- Strutture dati del kernel ✓
- Gestione energetica -
- Concetti principali del sottosistema di I/O del kernel -

#### ▼ 12.5 - Trasformazione delle Richieste di I/O in Operazioni Hardware 100%

#### ▼ 12.6 - STREAMS 0%

#### ▼ 12.7 - Prestazioni 100%

## CAPITOLO 13 - INTERFACCIA DEL FILE SYSTEM

#### ▼ 13.1 - Concetto di File 60%

- Attributi dei file ✓
- Operazioni sui file ✓
- Tipi di file ✓
- Struttura dei file -
- Struttura interna dei file -

### ▼ 13.2 - Metodi di Accesso 100%

- Accesso sequenziale ✓
- Accesso diretto ✓
- Altri metodi di accesso ✓

### ▼ 13.3 - Struttura delle Directory 100%

- Directory ad un livello ✓
- Directory a due livelli ✓
- Directory con struttura ad albero ✓
- Directory con struttura a grafo aciclico ✓
- Directory con struttura a grafo generale ✓

### ▼ 13.4 - Protezione 100%

- Tipi di accesso ✓
- Controllo degli accessi ✓
- Altri metodi di protezione ✓

### ▼ 13.5 - File Mappati in Memoria 0%

- Meccanismo di base ❌
- Memoria condivisa nella API di Windows ❌

## CAPITOLO 14 - REALIZZAZIONE DEL FILE SYSTEM

### ▼ 14.1 - Struttura del File System 100%

- Panoramica ✓
- Utilizzo ✓

### ▼ 14.2 - Operazioni del File System 100%

- Lista lineare ✓

### ▼ 14.3 - Realizzazione delle Directory 100%

- Tabella hash✓

▼ 14.4 - Metodi di Allocazione 75%

- Allocazione contigua✓
- Allocazione concatenata✓
- Allocazione indicizzata✓
- Prestazioni-

▼ 14.5 - Gestione dello Spazio Libero 66.6%

- Vettore di bit✓
- Lista concatenata✓
- Raggruppamento✓
- Conteggio✓
- Mappe di spazio-
- TRIM dei blocchi non utilizzati-

▼ 14.6 - Efficienza e Prestazioni 100%

- Efficienza✓
- Prestazioni✓

▼ 14.7 - Ripristino 75%

- Verifica della coerenza✓
- File system con log delle modifiche-
- Altre soluzioni-
- Copie di riserva e recupero dei dati✓

▼ 14.8 - Esempio: il File System WAFL 0%



# INTRODUZIONE AI SISTEMI OPERATIVI

Partendo dall'inizio, un sistema operativo è un insieme di programmi, **software**, che gestisce gli elementi fisici di un calcolatore, **hardware**. Fornisce una piattaforma ai programmi applicativi e agisce da intermediario tra l'utente e la struttura fisica del calcolatore. I SO sono veramente ovunque, al giorno d'oggi e, per capirli a pieno, è estremamente importante conoscere l'organizzazione e l'architettura dell'hardware del computer, quindi la CPU, i dispositivi di I/O e i dispositivi di memorizzazione. In generale, un SO può essere visto sotto due punti di vista, quello dell'**utente** e quello del **sistema**. Dal punto di vista dell'utente, il sistema può essere progettato per:

- **PC**, cioè per facilitare l'uso del computer.
- **Mainframe o minicomputer**, per massimizzare l'uso delle risorse.
- **Workstation**, compromesso tra le risorse individuali e quelle condivise.
- **Palmari e simili**, che prestano attenzione alla batteria.
- **Sistemi Embedded**, cioè che devono funzionare con poca o senza visibilità degli utenti

Dal punto di vista del sistema, invece, il SO è più strettamente connesso all'hardware ed è:

- **allocatore di risorse**: di fronte a conflitti, decide come assegnare equamente o efficientemente le risorse ai programmi.
- **programma di controllo**: garantisce l'esecuzione dei programmi senza errori.
- **esecutore di funzioni comuni**: esegue funzioni di utilità generale comuni a diversi programmi.
- **nucleo (*kernel*)**: l'unico programma sempre in esecuzione.

Tutto quello che descriveremo adesso è soltanto una breve panoramica, dato che tutto verrà approfondito a dovere durante la durata del corso. Uno strumento molto

importante, utilizzato dall'hardware per interagire col SO, consiste nelle **interruzioni**. Un dispositivo attiva un'interruzione, inviando un segnale alla CPU per ravisare che alcuni eventi richiedono la sua attenzione. Un'interruzione è gestita dal **gestore delle interruzioni**. Per essere eseguiti, i programmi dovranno risiedere nella **memoria centrale** del calcolatore, cioè l'unica area di memoria abbastanza grande da essere disponibile e accessibile dalla CPU. La memoria centrale, però, è **volatile**, cioè perde il proprio contenuto quando manca l'alimentazione elettrica. Per questo, utilizziamo la **memoria secondaria**, un'estensione di quella principale, in grado di memorizzare in maniera permanente grandi quantità di dati, solitamente su **dischi** o **nastri**. I sistemi di memorizzazione si possono organizzare in **maniera gerarchica** seconda la velocità e il costo. I livelli più alti rappresentano i dispositivi più veloci, ma anche i più costosi. Scendendo, il costo decresce, ma aumentano i tempi di accesso.

Le moderne architetture degli elaboratori sono sistemi **multiprocessore**, in cui ogni CPU contiene diverse unità di calcolo, detti **core**. Per massimizzare l'utilizzo della CPU, i moderni sistemi utilizzano la **multiprogrammazione**, in cui diversi processi occupano contemporaneamente la memoria e la CPU non è mai inattiva. Con i sistemi **multitasking**, la multiprogrammazione è stata estesa per mezzo di **algoritmi di scheduling**, che commutano tra un processo e l'altro e favoriscono tempi rapidi di risposta. Il **processo** è l'unità fondamentale di lavoro di un SO. La gestione dei processi comprende la loro creazione e cancellazione, nonché la loro comunicazione e sincronizzazione.

Un SO gestisce la memoria, mantenendo traccia di quali parti di essa vengono usate e da chi, oltre che l'**archiviazione dei dati**. Quest'ultima comprende la realizzazione del **file system** e le **directory**. I SO forniscono anche meccanismi di **protezione** per la sicurezza del sistema e degli utenti, controllando e centellinando l'accesso alle risorse che il sistema mette a disposizione. Un altro concetto fondamentale è la **virtualizzazione**, cioè l'astrazione dell'hardware di un computer in molteplici ambienti di destinazione. I SO forniscono diverse strutture dati, tra cui **liste**, **pile**, **code**, **alberi** e **bitmap**. Esistono vari tipi di ambienti elaborativi, che come il **mobile-computing**, i **sistemi client-server**, **peer-to-peer**, **cloud computing** e **embedded-real-time**.

Un SO offre un ambiente in cui eseguire i programmi e fornirgli servizi. Adesso, elencheremo alcuni servizi comuni alla maggior parte dei sistemi operativi:

- **interfaccia utente (UI)**: ne abbiamo di diversi tipi. Abbiamo l'**interfaccia a riga di comando (CLI)**, basata su stringhe che codificano i comandi, **interfaccia grafica**

**con l'utente (GUI)**, sistema grafico a finestre e dotato di un dispositivo puntatore(mouse).

- **esecuzione di un programma**: il sistema deve poter caricare un programma in memoria ed eseguirlo.
- **operazioni di I/O**: i programmi utenti non possono eseguire direttamente le operazioni di I/O.
- **gestione del file system**: esecuzione di operazioni di lettura, scrittura, creazione e cancellazione dei file.
- **comunicazioni**: scambi di informazioni tra processi, tramite memoria condivisa o scambio di messaggi.
- **rilevamento di errori**: assicurare la correttezza della computazione rilevando eventuali errori di CPU, memoria ecc.
- **allocazione delle risorse**: si assegnano risorse a più utenti o processi che sono concorrentemente in esecuzione.
- **contabilizzazione dell'uso delle risorse**: registrare quali utenti usano il calcolatore, segnalando quali risorse impiegano.
- **protezione e sicurezza**: assicurare il controllo degli accessi a tutte le risorse condivise.

Le **chiamate a sistema (system call)** costituiscono l'interfaccia tra il processo e il SO. Sono generalmente disponibili in forma di **istruzioni** in linguaggio assembly o in linguaggio di alto livello (C, C++). Tipicamente ad ogni SC è associato un numero, registrato in una tabella.

I programmi applicativi accedono ai servizi del sistema tramite le **API**, acronimo di **Application Programming Interface**, piuttosto che con le SC. Questo si fa perché, generalmente vengono mascherati i dettagli delle SC e sono portabili. L'API invoca l'opportuna SC e restituisce lo stato e il valore di ritorno. Le SC sono molteplici, ma generalmente si dividono in alcune macro-categorie:

- **controllo dei processi**: terminazione, caricamento, attesa, ecc.
- **gestione dei file**: creazione, cancellazione, lettura, scrittura, ecc.
- **gestione dei dispositivi**: lettura, scrittura, posizionamento, ecc.

- **gestione delle informazioni**: dati del sistema, attributi, ecc.
- **comunicazione**: creazione e chiusura di una connessione, ecc.

I servizi di sistema offrono un ambiente per lo sviluppo e l'esecuzione dei programmi.

Anche essi si dividono in tante categorie, simili a quelle descritte prima. Anche la progettazione di un SO si divide in alcuni scopi comuni:

- **scopi degli utenti**: il SO deve essere conveniente da usare, semplice da imparare, affidabile, sicuro e veloce.
- **scopi del sistema**: il SO deve essere semplice da progettare, implementare e mantenere, oltre che flessibile e affidabile.

La progettazione viene eseguita attraverso alcuni criteri e meccanismi, racchiusi nella **policy**. Una volta progettato, il SO va realizzato. Oggi, la realizzazione di un SO può essere effettuata anche tramite linguaggi di alto livello, che gli permette di scrivere più velocemente il codice e di renderlo più compatto e semplice. Affinché possa funzionare correttamente ed essere facilmente modificabile, un SO deve avere una struttura ben precisa. Possiamo avere:

- **struttura monolitica**: indica, generalmente, l'assenza di struttura, dove tutte le funzionalità del kernel vengono scritte in un unico file binario statico.
- **struttura stratificata**: il sistema viene suddiviso a strati, dove ne abbiamo uno basso, che è quello fisico, e uno alto, che è l'interfaccia utente.

Come già specificato, il **kernel** è il nucleo del SO. Un sistema può anche essere basato sul **microkernel**, cioè si spostano i servizi del kernel a livello dei programmi utenti. Altri SO, invece, implementano un **kernel modulare**, cioè un approccio object-oriented, in cui ogni componente è separata. Quest'ultime dialogano tra di loro con l'uso di interfacce.

Una **macchina virtuale** tratta hardware e SO come fossero entrambi hardware. Esse creano l'illusione di processi multipli, ciascuno in esecuzione sul suo processore e con la sua memoria, entrambi virtuali. Questo concetto provvede una completa protezione delle risorse del sistema, senza però permettere la condivisione diretta delle risorse. Un **boot loader** carica un SO in memoria, esegue l'inizializzazione e avvia l'esecuzione del sistema. Le prestazioni di un SO possono essere monitorate utilizzando **contatori**, una raccolta di statistiche a livello di sistema, o il **tracing**, che segue l'esecuzione di un programma.





# PROCESSI

In principio, all'interno dei sistemi di elaborazione, veniva eseguito soltanto un programma alla volta, che aveva accesso a tutti le risorse. Ora, però, in memoria vengono caricati più programmi, che sono eseguiti in modo concorrente. In generale, un SO esegue diverse varietà di programmi. Un **sistema batch** esegue **job**, mentre un **sistema time-sharing** esegue **task**. Queste attività sono molto simili tra di loro e vengono indicate come **processi**, cioè le principali unità di lavoro all'interno di un SO. Un processo, in memoria, è composto da diverse sezioni, che sono:

- **contatore di programma (PC)**: il registro della CPU che tiene conto di tutte le attività svolte e da svolgere.
- **sezione di testo**: che contiene il codice eseguibile.
- **sezione dati**: che contiene le variabili globali.
- **heap**: memoria che viene allocata dinamicamente durante l'esecuzione.
- **stack**: memoria allocata temporaneamente e utilizzata durante le chiamate a funzioni.

Prima di continuare è necessario fare una distinzione. Un programma non è un processo. Esso è un'**entità passiva**, cioè un file sul disco che contiene una lista di istruzioni. Il processo, invece, è un'**entità attiva**, che specifica qual è l'istruzione successiva da eseguire e un insieme di risorse.



Mentre un processo è in esecuzione, è soggetto a **cambiamenti di stato**. Gli stati, come indicati nel diagramma sopra, possono essere:

- **New**: si crea il processo.
- **Running**: le istruzioni vengono eseguite.
- **Waiting**: il processo attende che si verifichi qualche evento.
- **Ready**: il processo è pronto e attende di essere assegnato ad un'unità di elaborazione.
- **Terminate**: il processo ha terminato l'esecuzione.

All'interno del SO, il processo è rappresentato da un **blocco di controllo**, detto anche **PCB**, che contiene molte informazioni, tra cui:

- **stato del processo**: in che stato si trova il processo in questo istante, tra quelli che abbiamo descritto sopra.
- **PC**: il contatore di programma contiene l'indirizzo della successiva istruzione da eseguire.
- **registri della CPU**: che possono essere accumulatori, registri indice, stack pointer e condition codes.
- **informazioni sullo scheduling della CPU**: la priorità del processo, i puntatori alle code di scheduling ed altri parametri che approfondiremo.

- **informazioni sulla gestione della memoria:** registri di base e di limite, tabelle delle pagine di memoria o dei segmenti e altri parametri che verranno in seguito approfonditi.
- **informazioni di accounting:** tempo di CPU, tempo reale di CPU, numero dei processi, ecc.
- **informazioni sullo stato dell'I/O:** la lista dei dispositivi di I/O assegnati ad un determinato processo.

Poco fa abbiamo introdotto il concetto di scheduling del processi. Andremo a fare una piccola introduzione, prima di approfondirlo in un capitolo successivo. L'obiettivo della multiprogrammazione, come abbiamo già detto, è quello di massimizzare l'uso della CPU e, quindi è necessario che una qualsiasi entità selezioni il processo da eseguire dall'insieme di quelli disponibili. Questa entità è detta **scheduler dei processi**. Nel sistema, ogni processo è inserito all'interno di una coda di processi pronti, o **ready queue**, che è generalmente vista come una lista concatenata. Nel corso del suo ciclo di vita, un processo si sposta continuamente dalla ready queue alle altre code di attesa. Possiamo individuare diversi tipi di scheduler:

- **scheduler a lungo termine**, che seleziona i processi da inserire nella ready queue.
- **scheduler a breve termine**, che seleziona un processo dalla ready queue e gli assegna la CPU.
- **swapping**, che è una specie di scheduler intermedio che rimuove i processi dalla memoria. Il processo potrà comunque essere reintrodotto e riprendere la propria esecuzione, riducendo, tuttavia, il grado di multiprogrammazione.

In presenza di un'interruzione, il sistema deve salvare il contesto del processo attualmente in esecuzione. Il contesto viene preso, di solito, dal PCB. Viene eseguito un **salvataggio di stato** e, successivamente, un **ripristino di stato**. Questa procedura è detta **cambio di contesto**. Il tempo necessario al cambio di contesto è detto *overhead*, cioè non compie un lavoro utile alla computazione. La sua durata dipende molto dall'architettura che si sta utilizzando.

Dato che i processi possono essere eseguiti in modo concorrente, c'è bisogno di poterli creare e cancellare dinamicamente. Durante la sua esecuzione, un processo può creare altri processi. Il processo creante è chiamato **padre**, mentre il processo creato è chiamato **figlio**. Si tratta di una struttura ricorsiva, dato che il processo figlio può diventare padre di altri processi. Così facendo, si crea una **struttura ad albero**. La

maggior parte dei SO identifica il processo tramite un numero univoco, detto **identificatore del processo**, o ***pid***. Durante la creazione, ci sono due possibilità. Il processo padre continua l'esecuzione in modo concorrente con i suoi processi figli, oppure il processo genitore attende che alcuni o tutti i suoi processi figli terminino. Un processo, invece, termina quando finisce l'esecuzione della sua ultima istruzione. Con un apposita system call, il processo chiede al sistema di essere cancellato. La maggior parte dei sistemi non permette ai processi figli di esistere senza il padre. Così facendo, si avrà una **terminazione a cascata**.

I processi eseguiti concorrentemente possono essere di due tipi:

- **indipendenti**, se non possono influire su altri processi o subirne l'influsso.
- **cooperanti**, se influenzano e possono essere influenzati da altri processi in esecuzione all'interno del sistema.

Consentire la condivisione tra i processi può essere davvero molto utile. Si potrebbero condividere le informazioni, si velocizzerebbe il calcolo e si garantirebbe un sistema modulare. Per la condivisione, i processi necessitano di un **meccanismo di comunicazione**, detto **IPC**, acronimo di *Interprocess Communication*. Ci possono essere due tipologie di comunicazione:

- **memoria condivisa**: i processi cooperanti allocano una zona di memoria condivisa e, altri processi che vorranno utilizzarla, dovranno annetterla al loro spazio di indirizzi. Di solito, si utilizza la dicitura produttore/consumatore. Un **processo produttore** produce informazioni, che sono consumate da un **processo consumatore**. L'esecuzione concorrente richiede la presenza di un **buffer**, cioè una parte di memoria che viene riempita dal produttore e svuotata dal consumatore. Un buffer può essere **illimitato**, cioè senza limiti di dimensioni, o **limitato**, cioè con una dimensione fissa.
- **scambio di messaggi**: due o più processi cooperanti possono comunicare e sincronizzarsi senza bisogno di condividere uno spazio. Si tratta di un meccanismo molto utile nelle architetture distribuite, dove diverse macchine sono connesse ad una rete. Bisogna prevedere almeno due operazioni, cioè `send(message)` e `receive(message)`. I messaggi possono avere una lunghezza fissa o variabile. Deve, ovviamente, esistere un **canale di comunicazione**, che garantiscono una **comunicazione diretta**, in cui si specifica esplicitamente il ricevente e il trasmittente, oppure una **comunicazione indiretta**, dove i messaggi si inviano a delle **mailbox**, portali in cui i processi possono prelevare i messaggi.

Indipendentemente dal tipo di comunicazione, possiamo avere uno scambio di messaggi **sincrono**, in cui inviante e ricevente si bloccano in attesa dei rispettivi eventi, oppure uno scambio di messaggi **asincrono**, in cui inviante e ricevente procedono la loro esecuzione. I messaggi scambiati risiedono in **code temporanee**. Possiamo avere code a **capacità zero**, che non si riempiono mai e non hanno messaggi in attesa al loro interno. Oppure, potremo avere code a **capacità limitata**, con una lunghezza finita  $n$ , in cui il messaggio ultimo arrivato, si posiziona in fondo alla coda. Infine, possiamo avere code a **capacità illimitata**, che contengono un numero indefinito di messaggi.

Le tecniche appena descritte sono utilizzabili anche per la comunicazione **client/server**. Per quest'ultima, però, possiamo anche indicare altre due strategie:

- **socket**: un socket è l'estremità di un canale di comunicazione. I processi comunicano utilizzando un socket ciascuno. Ogni socket è identificato da un **indirizzo IP concatenato** e un numero di **porta**. Il server attende le richieste del client, stando in ascolto ad una determinata porta; quando il server riceve la richiesta, accetta la connessione proveniente dalla socket del client e, così facendo, si stabilisce la comunicazione.
- **chiamate di procedure remote (RPC)**: permettono ad un client di richiamare una procedura presente in un sistema remoto nello stesso modo in cui invocherebbe una procedura locale. I dettagli della comunicazione vengono nascosti e viene fornito uno **stub** per il lato client, una porzione di codice che simula il comportamento reale. Quando il client invoca una procedura remota, chiama l'appropriato stub, passando i suoi parametri. Lo stub individua la porta del server e struttura i parametri, in un processo detto **marshalling**. Il server riceve il messaggio tramite il proprio stub, invoca la procedura e riporta i risultati al client.



# THREAD E CONCORRENZA

Alcune volte, una singola applicazione deve poter gestire più compiti diversi, che verrebbero eseguiti concorrentemente. L'idea principale sarebbe quella di creare più processi, ma la creazione e il cambio di contesto sono effettivamente operazioni molto complicate, che richiedono un tempo di risposta molto elevato. La soluzione risulta essere nei thread. Un thread viene erroneamente detto **processo leggero**, *lightweight process*. Esso comprende un ID, un PC, un insieme di registri e una pila. Un thread può condividere il suo codice, i suoi dati e le risorse di sistema. La maggior parte delle moderne applicazioni è **multithread**, cioè un'applicazione codificata come un processo a sé stante, con più thread di controllo. La programmazione multithread ha molti vantaggi:

- **tempo di risposta**: l'esecuzione continua anche se una parte del SO è bloccata.
- **condivisione delle risorse**: i thread condividono per default la loro memoria, senza bisogno di ricorrere a spazi condivisi o scambio di messaggi.
- **economia**: i thread condividono le risorse per default e, quindi, risultano essere anche più economici in questi termini.
- **scalabilità**: i thread possono essere eseguiti in parallelo anche su core differenti.

Prima di andare avanti, distinguiamo la differenza tra **concorrenza** e **parallelismo**. Un sistema concorrente supporta più task, permettendo a ciascuno di progredire la propria esecuzione. Un sistema parallelo permette a più task di essere eseguiti simultaneamente. Esistono due tipi di parallelismo. Il **parallelismo dei dati** permette la distribuzione di sottoinsiemi dei dati su più core di elaborazione, mentre il **parallelismo delle attività** prevede la distribuzione dei thread su più core.

In generale possiamo avere **thread a livello utente** e **thread a livello kernel**. In base alla relazione che esiste tra loro, identifichiamo tre strategie possibili:

- **modello da molti a uno**: molti thread a livello utente con un singolo thread a livello kernel. Si tratta di una gestione efficiente, ma bloccante, dato che, se si bloccasse un thread a livello utente, verrebbe bloccato tutto il sistema.

- **modello da uno a uno**: ogni thread a livello utente ha un thread a livello kernel corrispondente. Questo modello supera i limiti del precedente, ma risulta essere molto costoso in termini di risorse e sensibile ad eventuali sovraccarichi.
- **modello da molti a molti**: molti thread a livello utente con un numero minore o uguale di thread a livello kernel. Si preoccupa di superare i problemi di concorrenza, cioè permette di creare liberamente thread a livello utente, a cui non corrisponderà per forza un thread a livello kernel, eseguibili parallelamente nelle architetture multiprocessore.



# SCHEDULING DELLA CPU

In un sistema multicore, è necessario eseguire più processi contemporaneamente, per massimizzare la produttività. Vengono sottoposte allo scheduling quasi tutte le risorse, tra cui la CPU. Il successo dello scheduling dipende da alcuni criteri. L'esecuzione di un processo comincia con un lavoro di elaborazione, detto **CPU burst**, e termina con una sequenza di operazioni di I/O, dette **I/O burst**. All'interno della CPU, lo **scheduler** seleziona tra i processi pronti e fa partire l'esecuzione, cioè quelli a cui assegnare la CPU. Esistono, ovviamente, diversi **algoritmi di scheduling**, che vedremo più nel dettaglio. Di solito si prendono decisioni in base:

- un processo passa dallo stato di esecuzione allo stato di attesa.
- un processo passa dallo stato di esecuzione allo stato di pronto.
- un processo passa dallo stato di attesa allo stato pronto.
- un processo termina.

Se lo scheduler interviene sulle condizioni 1 e 4, allora si parla di uno schema **senza prelazione**, *nonpreemptive*, cioè il processo rimane in possesso della CPU fino al momento del suo rilascio. Altrimenti, è uno schema **con prelazione**, *preemptive*, dove possiamo incorrere in una *race condition*, cioè la condivisione dei dati, che approfondiremo successivamente, poiché si dovrà ricorrere a meccanismi di sincronizzazione.

Un altro elemento coinvolto nelle funzioni di scheduling, è il **dispatcher**, cioè un modulo che passa il controllo della CPU al processo scelto dallo scheduler. Il tempo richiesto dal dispatcher per fermare un processo e avviare l'esecuzione di un altro è detto **latenza di dispatch**. Prima di scegliere l'algoritmo che andrà ad effettuare lo scheduling tra i processi, bisogna tener conto di alcuni criteri:

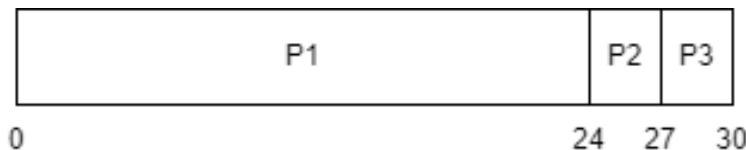
- **utilizzo della CPU**: essa deve essere quanto più attiva possibile.
- **throughput**: la CPU è attiva quando svolge lavoro. La misura del lavoro è detta throughput, cioè produttività.

- **tempo di completamento:** l'intervallo che intercorre tra la sottomissione del processo e il completamento dell'esecuzione.
- **tempo di attesa:** la somma di tutti gli intervalli di attesa.
- **tempo di risposta:** l'intervallo che intercorre tra la sottomissione di una richiesta e la prima risposta prodotta.

I primi due criteri devono essere sfruttati al massimo, mentre gli ultimi tre al minimo. Come abbiamo già anticipato, esistono diversi algoritmi per poter effettuare lo scheduling. Ora ne discuteremo alcuni, mostrando il loro funzionamento attraverso il **diagramma di Gantt**:

- **scheduling in ordine di arrivo** (*first-come, first-served - FCFS*): si assegna la CPU al processo che la richiede per primo. Quando la CPU è libera, si sceglie il processo in testa alla ready queue. Di solito, i tempi di attesa sono abbastanza lunghi.

Processo	Burst Time
P1	24
P2	3
P3	3



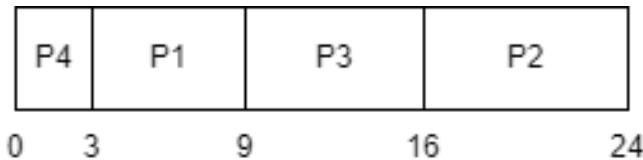
- Tempo di Attesa P1 = 0
- Tempo di Attesa P2 = 24
- Tempo di attesa P3 = 27

$$\text{Tempo di Attesa Medio} = (0 + 24 + 27)/3 = 17$$

Finché non termina il processo in esecuzione, tutti gli altri attendono nella ready queue. Questo algoritmo non è conveniente, dato che ci potrebbe essere un processo con un tempo di completamento piuttosto lungo e, all'interno della coda, si creerebbe un **effetto convoglio**, con tutti i processi che attendono ad oltranza all'interno della ready queue.

- **scheduling per brevità** (*shortest-job-first* - **SJF**): si assegna la CPU al processo con la durata della sequenza più bassa. Se due processi hanno la stessa durata, allora si applica il FCFS tra di loro. Si tratta di un algoritmo ottimale, poiché supera i limiti del precedente e permette di ridurre drasticamente il tempo di attesa medio. Però, uno scheduling SJF è impossibile da realizzare in maniera ottimale, perché lo scheduler non conosce le durate successive. Per questo, si tende ad ovviare a questa cosa, predicendo la durata, calcolando la **media esponenziale** delle sequenze precedenti. Questo algoritmo può essere sia *non-preemptive* che *preemptive*, cioè può sia sostituire un processo in esecuzione, se quello appena arrivato ha una durata minore, e sia aspettare che esso termini.

Processo	Burst Time
P1	6
P2	8
P3	7
P4	3



- Tempo di Attesa P1 = 3
- Tempo di Attesa P2 = 16
- Tempo di Attesa P3 = 9
- Tempo di Attesa P4 = 0

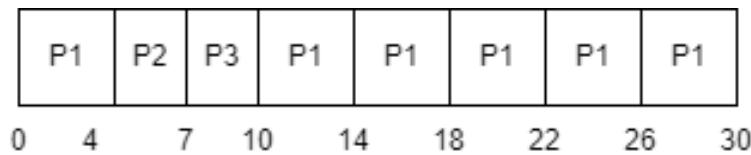
$$\text{Tempo di Attesa Medio} = (3 + 16 + 9 + 0)/4 = 7$$

- **scheduling circolare** (*round-robin* - **RR**): si tratta di un algoritmo di scheduling molto simile al FCFS, soltanto che si aggiunge la capacità di prelazione, in modo che il sistema possa scegliere in maniera ottimale. Ad ogni processo viene assegnato una piccola quantità fissata del tempo della CPU, detto **quanto di tempo**, che è di solito tra 10 e 100 millisecondi. Allo scadere di questo tempo, il processo viene stoppato e aggiunto alla fine della coda. Di solito, il tempo di attesa

medio è abbastanza lungo e dipende molto dalle dimensioni del quanto di tempo.

### QUANTO DI TEMPO = 4 millisecondi

Processo	Burst Time
P1	24
P2	3
P3	3

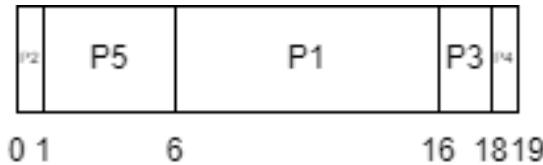


- Tempo di Attesa P1 = 6
- Tempo di Attesa P2 = 4
- Tempo di Attesa P3 = 7

$$\text{Tempo di Attesa Medio} = (6 + 4 + 7)/3 = 5,66$$

- **scheduling con priorità:** si tratta di un algoritmo che associa una priorità ad ogni processo e si assegna la CPU al processo con la priorità più alta. I processi con priorità uguale si assegnano in base al FCFS. Le priorità sono indicate con un intervallo fisso di numeri, che può essere da 0 a 7. Liberamente, si possono scegliere numeri piccoli per indicare priorità alte o viceversa. Ora faremo un esempio con la prima opzione, cioè più piccolo è il numero più alta sarà la priorità.

Processo	Burst Time	Priorità
P1	10	3
P2	1	1
P3	2	4
P4	1	5
P5	5	2



- Tempo di Attesa P1 = 6
- Tempo di Attesa P2 = 0
- Tempo di Attesa P3 = 16
- Tempo di Attesa P4 = 18
- Tempo di Attesa P5 = 1

$$\text{Tempo di Attesa Medio} = (6 + 0 + 16 + 18 + 1) / 5 = 8,2$$

Questo algoritmo presenta un problema molto grave, cioè l'**attesa indefinita**, più comunemente nota come *starvation*. Un processo può essere pronto per l'esecuzione, ma avere una priorità bassa e, quindi, restare in attesa ad oltranza, finendo poi per crashare. Una soluzione all'attesa indefinita è l'**invecchiamento**, detto *aging*. Con questo meccanismo si tende ad aumentare la priorità del processo nel tempo.

- **scheduling a code multilivello:** la ready queue è spesso divisa in più code separate, in cui i processi possono spostarsi da essa. I processi vengono di solito divisi in processi in primo piano, interattivi, o processi in background, batch. Avendo requisiti diversi, hanno anche esigenze di scheduling differenti. Ciascuna coda ha il proprio algoritmo di scheduling, ma è necessario che anche tra di loro ci sia qualche sorta di meccanismo di partizione, di solito uno scheduling a priorità fissa.
- **scheduling a code multilivello con retroazione:** i processi, normalmente, non possono spostarsi dalle code, data la loro natura diversa. Con questo algoritmo si separano i processi che hanno caratteristiche diverse in termini di CPU burst. Se un processo usa troppa CPU, viene spostato in una coda con priorità più bassa. Quindi, parliamo di una sorta di invecchiamento. Si tratta di un algoritmo molto efficiente, ma anche il più complesso da realizzare.

Questi algoritmi che abbiamo descritto riguardano lo scheduling della CPU all'interno di un sistema dotato di un singolo core di elaborazione. In caso di più unità di elaborazione potremo passare alla **distribuzione del carico** ma, se parliamo di

scheduling, diventa tutto molto più complesso. Andiamo, ora, a descrivere alcune strategie di scheduling per multiprocessore:

- **multielaborazione simmestrica**: tutte le decisioni, le operazioni di I/O ecc, sono designate ad un singolo processore, chiamato **master-server**. Si riduce la necessità di condividere, ma è possibile incappare nel cosiddetto collo di bottiglia, cioè un sovraccarico che ridurrà le prestazioni del sistema.
- **multielaborazione simmestrica (SMP)**: lo scheduler di ogni processore esamina la ready queue e seleziona i processi da eseguire, sia che sia una coda universale per tutto il sistema e sia che sia a disposizione soltanto di quel processore. Tutti i SO moderni supportano SMP. Questa tecnica, col tempo, ha subito una variante, dando origine ad un **processore multicore**, in cui ogni core mantiene il proprio stato e risulta essere un processore fisico separato. Nei sistemi SMP è necessario che il lavoro sia distribuito equamente su tutte le unità di elaborazione. Per questo, è necessario un **bilanciamento di carico**. Può avvenire una **migrazione push**, dove un processo apposito controlla periodicamente il carico di ogni processore, o una **migrazione pull**, quando un processore inattivo dà una mano ad un altro in sovraccarico. I due approcci sono mutuamente esclusivi, cioè vengono usati anche in contemporanea. Sempre all'interno dei sistemi SMP, possiamo parlare del concetto di **predilezione per il processore**. Quando un processo si sposta da un processore all'altro, ci sono elevati costi di svuotamento e riempimento della cache, quindi si cerca sempre di evitarlo. Per questo motivo, possiamo parlare di un processo che ha la predilezione per il processore su cui è esecuzione, sia essa **debole**, quando il sistema si propone di mantenerlo, senza garantirlo, oppure **forte**, dove si forza il mantenimento del processo.
- **multielaborazione eterogenea (HMP)**: alcuni sistemi che progettano i core per eseguire lo stesso numero di istruzioni, con velocità di clock diverse, includendo la possibilità di regolare il consumo energetico di un core.

All'inizio del corso abbiamo parlato dei **sistemi real-time**, cioè dei calcolatori la cui correttezza computazionale non dipende soltanto dalla logica, ma anche dal tempo massimo di risposta. Possiamo individuare:

- **sistemi real-time soft**: non offrono garanzie sul momento in cui un processo critico sarà eseguito, ma assicurano che gli sarà data precedenza su quelli meno critici.

- **sistemi real-time hard**: i processi devono essere eseguiti entro una scadenza prefissata ed eseguirli oltre, risulterebbe inutile.

Prima di proseguire, faremo un breve excursus sullo scheduling dei processi in **Linux**, anche se sarebbe più corretto chiamarli **task**. Esso si basa su delle **classi di scheduling**. Ad ogni classe è associata una **priorità**. Invece di assegnare delle rigide regole per i quanti di tempo, lo scheduler assegna ad ogni task un tempo di elaborazione della CPU, calcolato sulla base delle **nice-value**, dove un valore numero basso indica una priorità superiore. I quanti di tempo non hanno valori fissi, ma si basano sulla **latenza obiettivo**, cioè un lasso di tempo in cui essi dovrebbero essere eseguiti almeno una volta. Inoltre, lo scheduler tiene conto anche del **tempo di esecuzione virtuale**, cioè per quanto tempo un task è stato eseguito, salvando tutto nella variabile `vruntime`.

Qualcuno potrebbe chiedersi come scegliere un algoritmo di scheduling della CPU. All'inizio di questo capitolo abbiamo definito alcuni criteri, ma spesso occorre stabilire l'importanza di essi. Definiamo, ora, alcuni metodi di **valutazione** degli algoritmi:

- **modellazione deterministica**: viene considerato un carico di lavoro predeterminato e definisce le prestazioni di ciascun algoritmo per quel carico di lavoro. Si tratta di un metodo semplice e rapido, anche se viene preso in considerazione soltanto per la descrizione degli algoritmi di scheduling.
- **reti di code**: il sistema di calcolo viene descritto come una rete di server, ciascuno con la propria ready queue. Conoscendo l'andamento degli arrivi e dei servizi, è possibile calcolare l'utilizzo, la lunghezza delle code e il tempo medio di attesa. Anche questo è un meccanismo molto utile per il confronto, ma presenta alcuni limiti, poiché è difficile prevedere distribuzioni di arrivi e di servizi in maniera realistica.
- **simulazioni**: viene programmato un modello di calcolo, dove le strutture dati rappresentano le unità principali del sistema, mentre una variabile rappresenta un clock. In genere, un simulatore definisce un generatore di numeri casuali, programmato per definire processi, CPU burst, arrivi, servizi, ecc. Le simulazioni richiedono diverse ore di tempo di elaborazione e possono essere molto onerose, anche perché i risultati migliori si hanno con simulazioni più dettagliate.
- **realizzazione**: l'algoritmo di scheduling viene codificato, inserito nel sistema e ne viene osservato il comportamento in tempo reale. Si tratta di un metodo totalmente

efficace, ma che richiede anch'esso molto tempo di elaborazione e, inoltre, il codice può variare in base all'ambiente in cui viene eseguito



# STRUMENTI DI SINCRONIZZAZIONE

Abbiamo già parlato dei **processi cooperanti**, cioè quei processi che possono influenzare l'esecuzione di un altro processo o subirla. Spesso, però, per farlo, i processi devono condividere il loro spazio di indirizzi e questo può causare incoerenze tra i dati. C'è bisogno, quindi, di meccanismi che garantiscono un'ordinata esecuzione. Un problema comune risulta essere la **race condition**, cioè quando più processi accedono e modificano gli stessi dati concorrentemente e i risultati dipendono dall'ordine degli accessi. Per ovviare alla RC, è necessario che i processi siano **sincronizzati**. Per cominciare, definiamo correttamente il concetto di **sezione critica**. Prendiamo come esempio un sistema composto da  $n$  processi. La sua sezione critica è un segmento di codice in cui il processo può modificare variabili comuni, aggiornare tabelle, scrivere file, ecc. Quando un processo è nella sua sezione critica, nessun altro processo è autorizzato ad entrare. Un processo deve chiedere il permesso di entrare all'interno della propria sezione critica, tramite un segmento di codice detto **sezione d'ingresso**. Successivamente, la sezione critica viene eseguita da un altro spezzone, detto **sezione di uscita**. Il resto del codice è detto **sezione non critica**. Per trovare una soluzione alla sezione critica e, quindi, a permettere la cooperazione tra processi, bisogna soddisfare tre requisiti:

- **mutua esclusione**: se un processo è in esecuzione nella propria sezione critica, nessun altro processo può essere in esecuzione nella propria.
- **progresso**: se un processo chiede il permesso di entrare all'interno della propria sezione critica, solo i processi che ne sono fuori possono partecipare alla decisione riguardante la scelta.
- **attesa limitata**: se un processo ha richiesto l'ingresso nella propria sezione critica, esiste un limite al numero di volte che si consente ad altri processi di entrare nelle loro, prima che si accordi la richiesta.

Una classica soluzione software al problema della sezione critica, è noto come **Soluzione di Peterson**. Nonostante non funzioni correttamente sugli attuali sistemi, rappresenta comunque una soluzione ottimale. Essa è limitata da **due processi**, P1 e P2, ognuno dei quali esegue alternativamente propria sezione critica e non critica. Questo algoritmo prevede che i processi condividano:

- `int turn`: segnala di chi sia il turno di accesso alla sezione critica.
- `boolean flag[2]`: indica se un processo è pronto ad entrare nella sezione critica.

Ne descriviamo brevemente il funzionamento. Per entrare nella sezione critica, il processo  $P_i$  assegna a `flag[i]` il valore `true`, dove  $P_i$  avverte di voler entrare in sezione critica. Poi,  $P_i$  suppone che la sezione critica sia già occupata dall'altro processo, quindi setta `turn` uguale a `j`. Fino a quando `turn` sarà uguale a `j` e anche `flag[j]` sarà uguale a `true`, allora  $P_i$  si mette in attesa. Quando  $P_i$  uscirà dalla sua sezione critica, setterà `flag[i]` a `false`. Per dimostrarne la correttezza, dobbiamo dimostrare che sia la mutua esclusione, che il progresso e l'attesa limitata siano preservati. Prima di procedere le verifiche, ecco l'algoritmo scritto in pseudocodice e basato sul lavoro di  $P_i$ :

```

while(true){
    flag[i] = true;
    turn = j;
    while(flag[j] && turn == j);
    /* sezione critica */
    flag[i] = false;
    /* sezione non critica */
}

```

1. Dimostriamo che la mutua esclusione è preservata.  $P_i$  accede alla propria sezione critica soltanto quando `flag[j] == false` oppure `turn == i`. Notiamo anche che, nel caso  $P_i$  e  $P_j$  decidano di accedere concorrentemente, si verifica `flag[i] == flag[j] == true`. Quindi, automaticamente, non è possibile eseguire concorrentemente  $P_i$  e  $P_j$ , dato che `turn` può valere `i` o `j`.
2. Possiamo impedire ad un ipotetico processo  $P_i$  di entrare nella propria sezione critica solo se questo è bloccato nel ciclo `while` della condizione `flag[j] == true` e `turn == j`. Se  $P_j$  non è pronto ad entrare, allora `flag[j]==false` e così  $P_i$  può entrare nella propria sezione critica. Invece, se  $P_j$  è pronto per entrare, allora `flag[j] == true`, se `turn == i` entra in  $P_i$ , mentre se `turn == j`, allora entra  $P_j$ .

Se entra  $P_j$ , all'uscita imposta `flag[j] == false` e  $P_i$  è libero di entrare.  $P_i$ , quindi, entra nella sezione critica (progresso soddisfatto) al massimo dopo un ingresso da parte di  $P_j$  (attesa limitata rispettata).

Ovviamente, come abbiamo specificato, quella di Peterson risulta essere una soluzione poco ottimale sulle macchine moderne. Le architetture moderne offrono particolari istruzioni per controllare o modificare la memoria, in modo **atomico**, cioè non interrompibile. Queste istruzioni permettono di risolvere il problema della sezione critica. Ne distinguiamo due tipi di categorie:

- `test_and_set()` : modifica il contenuto di una variabile in maniera atomica. Quindi, se si eseguono contemporaneamente due istruzioni, ciascuna su un'unità di elaborazione diversa, queste vengono eseguite in maniera sequenziale in un ordine arbitrario. Con questa istruzione, la mutua esclusione è garantita dichiarando la variabile globale `lock` e inizializzandola a false.
- `compare_and_swap()` : fa più o meno la stessa cosa, soltanto che utilizza un meccanismo che si basa sullo scambio di contenuto tra parole. Anche qui si realizza facilmente la mutua esclusione, inizializzando a 0 una variabile globale `lock`. Il primo processo che la richiama, la settnerà ad uno, entrando poi nella sezione critica.

Queste soluzioni appena descritte, di solito, sono molto complicate e inaccessibili ai programmatore, per questo si tende ad utilizzare strumenti software. Uno di questi può essere il **lock mutex**. Con questa soluzione, un processo deve acquisire il lock prima di entrare nella sezione critica e rilasciarlo quando esce. Per questo, vengono fornite le funzioni `acquire()` e `release()`. Inoltre è prevista anche una variabile booleana `available`, il cui valore indica se il lock è disponibile o meno. Dato che le due funzioni devono essere eseguite atomicamente, vengono utilizzate le soluzioni hardware descritte prima. L'implementazione richiede, però, l'**attesa attiva**, o *busy-waiting*. Mentre un processo si trova nella sezione critica, un altro che cerca di entrare deve continuare a ciclare la funzione `acquire()`, sprecando cicli di CPU che potrebbero essere usati in maniera più produttiva.

I lock mutex rappresentano una soluzione semplice, ma è ora di descrivere in maniera dettagliata una soluzione più robusta e che permetta di fornire metodi più complessi per la sincronizzazione dei processi. Introduciamo, quindi, i **semafori**. Un semaforo è una variabile intera a cui si può accedere utilizzando due operazioni atomiche predefinite, cioè `wait()` e `signal()`. In generale distinguiamo due tipi di semafori:

- **semafori contatore**: il loro valore è un numero intero, che può variare in un dominio limitato.
- **semafori binari**: il loro valore è un numero intero, che può essere o 0 o 1.

I semafori vengono impostati in base al numero di risorse disponibili. Quando un processo desidera utilizzare una risorsa, invoca `wait()` sul semaforo, che decrementa il suo valore. I processi che restituiscono una risorsa, invece, invocano `signal()`, che incrementa il suo valore. Se il semaforo vale 0, tutte le risorse sono occupate e i processi sono costretti a bloccarsi.

```

wait(S){
    while(S<=0)
        ; //attesa attiva
    S--;
}

signal(S){
    S++;
}

```

Come per i lock mutex, anche con i semafori si presentano la stessa problematica dell'attesa attiva. Nel caso un processo trovasse il valore del semaforo uguale a zero, dovrà aspettare ma, anziché restare in attesa attiva, può bloccare sé stesso, mettendosi in una coda di attesa associata al semaforo, permettendo allo scheduler di scegliere un altro processo pronto per l'esecuzione. Il processo bloccato sarà riavviato in seguito all'esecuzione di una `signal()`. Per realizzare questa cosa è necessario creare il semaforo con un valore intero e una lista di processi. La `signal()` preleverà un processo dalla lista e lo attiverà, invocando l'operazione `wakeup()`, mentre la `wait()` lo aggiungerà alla lista in caso di valore uguale a zero, invocando l'operazione `sleep()`. Con questa nuova definizione, il valore del semaforo può scendere sotto lo zero. Il totale sotto lo zero sarà il totale dei processi bloccati e in attesa. Inoltre, con questa definizione, non si elimina totalmente l'attesa attiva, ma la si rimuove, quantomeno, dalla sezione di ingresso.

Per quanto i semafori possano risultare efficaci, il loro uso scorretto genera errori di difficile individuazione. Per esempio, nel caso `wait()` e `signal()` non venissero invocate in sequenza, può capitare che due processi occupino simultaneamente le rispettive sezioni critiche. Per questo si è passato alla realizzazione di costrutti di sincronizzazione scritti in linguaggio di alto livello, detti monitor. Partiamo col dire che un

tipo di dato astratto, ADT, incapsula i dati, mettendo a disposizione una serie di funzioni per operare su di essi. Il tipo monitor contiene un insieme di operazioni in mutua esclusione, la dichiarazione delle variabili condivise e il corpo delle funzioni e delle procedure.

```
monitor Monitor
{
    /* dichiarazione di variabili condivise */
    functionP1(){
        // corpo della funzione
    }
    functionP2(){
        // corpo della funzione
    }
    initialization_code(){
        // corpo della funzione
    }
}
```

Il costrutto assicura che all'interno di un monitor possa essere attivo un unico processo alla volta. Ovviamente, per una corretta sincronizzazione, bisogna definire il costrutto `condition`, che definisce una o più variabili, le cui operazioni consentite sono solo `wait()` e `signal()`. Ci possiamo trovare di fronte ad una problematica. Pensiamo di avere un processo  $P$  che invoca `x.signal()` ed un processo  $Q$  associato alla variabile `x` di tipo `condition`. Se  $Q$  riprende l'esecuzione,  $P$  è costretto ad attendere, altrimenti  $P$  e  $Q$  sarebbero entrambi attivi all'interno del monitor. I processi, però, possono continuare la loro esecuzione. Entrambe possono attendere che l'altra lascia il monitor o la variabile.

Il termine ***liveness*** fa riferimento ad alcune proprietà che un sistema deve soddisfare per garantire che i processi facciano progressi durante la loro esecuzione. Alcune situazioni che portano ad una mancanza di liveness sono:

- **stallo (deadlock)**: due processi attendono indefinitamente che uno di loro faccia qualcosa. Così, vanno in stallo.
- **inversione di priorità**: quando processi a priorità alta hanno bisogno di cooperare con processi a priorità bassa. Dato che i dati vengono protetti dal lock, il processo a priorità alta dovrà comunque attendere che quello a priorità bassa finisca il suo compito.

Terminando, è necessario capire quale strumento di sincronizzazione utilizzare, tra le soluzioni hardware e quelle tradizionali (lock mutex, semafori e monitor). Si fa una distinzione in base al **livello di contesa** delle risorse.:

- **nessuna contesa**: in questo caso, entrambe le soluzioni vanno bene, anche se quelle hardware sarebbero nettamente più rapide.
- **contesa moderata**: le soluzioni hardware sarebbero ancora molto più rapide.
- **alta contesa**: la sincronizzazione tradizionale sarà molto più veloce.



# ESEMPI DI SINCRONIZZAZIONE

Dopo averli descritti, è necessario anche andare ad applicare i vari meccanismi di sincronizzazione, vedendo pratici esempi. Questi esempi serviranno a risolvere diversi problemi di concorrenza. Vediamone alcuni classici:

- **produttore/consumatore con memoria limitata.** Nel nostro caso, produttore e consumatore condividono un numero intero `n`, e tre variabili semaforo, `mutex`, `empty` e `full`. Supponendo di avere un buffer a memoria limitata di  $n$  dimensioni. Il semaforo `mutex` è inizializzato ad 1 e garantisce la mutua esclusione agli degli accessi al buffer. I semafory `empty` e `full` rappresentano rispettivamente le posizioni vuote e piene del buffer.
- **problema dei lettori-scrittori.** Supponendo di avere un database a cui accedono molti processi concorrenti, che si dividono in lettori e scrittori. Due lettori possono accedere insieme ai dati, senza alcun problema, ma nel caso uno scrittore e un lettore si trovassero nella medesima situazione, allora insorgerebbero problemi. Per questo, è necessario che gli scrittori abbiano un accesso esclusivo al database condiviso. Esistono due soluzioni principali. Nessun lettore deve attendere, a meno che uno scrittore non abbia già ottenuto il permesso di usare il database. Oppure, uno scrittore deve adempiere ad i suoi compiti al più presto. In generale, entrambe le soluzioni possono presentare problemi e portare alla *starvation*.  
Una prima soluzione prevede che i lettori condividano alcune strutture dati. I semafori `rw_mutex` e `mutex` e l'intero `read_count`, che indica i processi che, in quell'istante, stanno leggendo i dati. Il semaforo `rw_mutex` è comune sia a scrittori che lettori, mentre `mutex` si assicura la mutua esclusione all'aggiornamento di `read_count`. In questo modo, si creano e si forniscono dei **lock di lettura-scrittura**.
- **problema dei cinque filosofi.** Consideriamo di avere cinque filosofi che trascorrono la loro esistenza pensando e mangiando. Sono seduti ad una tavola rotonda e al centro si trova una ciotola piena di riso, che loro dovranno raccogliere con cinque bacchette. Quando un filosofo pensa, non interagisce con gli altri. Quando vuole mangiare, deve prendere le bacchette più vicine a lui, alla sua destra

o alla sua sinistra. Può prendere una bacchetta alla volta, a meno che non si trovi in mano ad un altro filosofo. Quando un filosofo ha in mano due bacchette contemporaneamente, mangia senza lasciare le bacchette. Finito il pasto, posa le bacchette e torna a pensare. Può sembrare un esempio futile, ma è estremamente pratico e può essere risolto sia con i **semafori** che con i monitor. Per la prima opzione, si tende a rappresentare ogni bacchetta come un semaforo. Un filosofo tenta di afferrare una bacchetta tramite l'operazione `wait()` e la posa tramite una `signal()`. Ogni bacchetta è inizializzata ad uno. Questa soluzione garantisce che due filosofi mangino contemporaneamente, ma non esclude situazioni di stallo, dato che se tutti e cinque i filosofi avessero fame contemporaneamente, allora le bacchette sarebbero uguali a zero e ogni filosofo andrebbe in stallo. Per evitare lo stallo, allora si ricorre all'uso dei **monitor**. Questa soluzione impone che un filosofo possa prendere le bacchette solo quando sono entrambe disponibili. Si istituisce una struttura dati, che rappresenta gli stati in cui versa un filosofo: `enum {THINKING, HUNGRY, EATING} state[]`. Il filosofo *i* può impostare la variabile `state[i] = HUNGRY` solo se i suoi due vicini non stanno mangiando, quindi:

```
((state[(i+4)%5] != EATING) && (state[(i+1)%5] != EATING)).
```

Inoltre, è necessario dichiarare `condition self[5]`, che permette di ritardare il filosofo *i* quando ha fame, ma non riesce a prendere le bacchette. Fatte queste premesse, possiamo scrivere tutto nel monitor `DiningPhilosophers`. Ogni filosofo, prima di mangiare, invoca l'operazione `pickup()`, che può terminare con successo e sospensione. Finito di mangiare, il filosofo invoca `putdown()` e comincia a pensare. All'interno del codice di inizializzazione, si settano tutti i filosofi come pensanti.



# STALLO DEI PROCESSI

Spesso, in un ambiente di multiprogrammazione, possiamo trovarci di fronte ad una situazione in cui due o più processi si bloccano a vicenda, aspettando che uno esegua un'azione che serve all'altro o viceversa. Situazioni come queste sono dette di stallo, o **deadlock**. Un sistema, generalmente, è composto da un numero finito di risorse, come cicli di CPU, file, dispositivi di I/O, ecc. In una normale condizione di funzionamento, un processo, per utilizzare una risorsa, deve rispettare il seguente ordine:

1. **Richiesta**: il processo richiede la risorsa. Se la richiesta non può essere soddisfatta immediatamente, il processo deve attendere affinché non può acquisire tale risorsa.
2. **Uso**: il processo può operare sulla risorsa.
3. **Rilascio**: il processo rilascia la risorsa.

Sia richiesta che rilascio avvengono tramite chiamate a sistema. Una situazione particolare di stallo, è lo **stallo attivo**, detto **livelock**. Un deadlock si verifica quando ogni processo di un insieme viene bloccato in attesa di un evento che può essere causato solo da un processo dell'insieme. Il livelock si verifica, invece, quando un processo non si blocca, ma continua un'azione che non ha successo. Prendiamo l'esempio di due persone che stanno camminando in un corridoio stretto, nella direzione opposta. Nel deadlock, le due persone si bloccano, si impuntano e rimangono ferme. Nel livelock, invece, le persone si bloccano lo stesso, soltanto che si spostano a sinistra, a destra e viceversa, senza però riuscire comunque a sbloccare la situazione. In un sistema si può avere una situazione di stallo *solo* se si verificano contemporaneamente le seguenti condizioni:

- **mutua esclusione**: almeno una risorsa deve essere non condivisibile, cioè utilizzabile solo da un processo alla volta.
- **possesso e attesa**: un processo deve essere in possesso di almeno una risorsa e attendere di acquisire risorse ulteriori, già occupate.
- **assenza di prelazione**: le risorse non possono essere prelazionate, cioè possono rilasciare una risorsa solo volontariamente.

- **attesa circolare**: deve esistere un insieme di processi, tale che P1 attende una risorsa posseduta da P2, P2 attende una risorsa posseduta da P3 e così via, fino a tornare a P1.

Le situazioni di stallo si possono descrivere facilmente con il **grafo di assegnazione delle risorse**. Si tratta di un insieme di **vertici**  $V$  e di **archi**  $E$ . L'insieme  $V$  contiene due sottoinsiemi:  $T$  che rappresenta tutti i processi del sistema ed  $R$ , che rappresenta tutte le risorse del sistema. Un arco che va da un qualsiasi  $T$  ad un qualsiasi  $R$  indica la richiesta di una risorsa, ed è chiamato **arco di richiesta**. Un arco che va da  $R$  a  $T$  significa che la risorsa è stata assegnata al processo e si chiama **arco di assegnazione**. Graficamente, un cerchio rappresenta un processo e un rettangolo una risorsa. Se all'interno di questo grafico non ci sono **cicli**, allora non si verificherà nessuna situazione di stallo.

Quando si presenta una situazione di stallo, generalmente si possono fare tre cose:

- si ignora il problema, fingendo che lo stallo non possa mai verificarsi all'interno del sistema.
- si utilizza un protocollo per prevenire le situazioni, assicurandosi che non entri mai in stallo.
- si permette al sistema di entrare in stallo, per poi ripristinarlo quando la situazione è stata individuata.

Non sto qui a dire che la prima soluzione non è assolutamente praticabile. Per la seconda, invece, si può discutere. Come abbiamo già specificato, uno stallo si verifica quando accadono contemporaneamente quattro condizioni. Quindi, automaticamente, si può prevenire lo stallo, impedendo che si verifichi una di loro:

- possiamo evitare che le risorse siano mutuamente esclusive, così facendo, le renderemo condivisibili e non si verificherà nessuno stallo. Per esempio, potremo scegliere dei file aperti per sola lettura. In generale, alcune risorse sono di default non condivisibile e, a volte, potremo non poter evitare la mutua esclusione.
- per evitare il possesso e attesa, dobbiamo garantire che il processo non richieda mai una risorsa se ne possiede altre. Si può forzare il processo a rilasciare tutte le sue risorse prima di richiederle, con la consapevolezza che risulti tutto poco efficiente. Il processo potrebbe richiedere delle risorse molto utilizzate, che però resterebbero per molto tempo fuori dalla sua disponibilità.

- per evitare l'assenza di prelazione, si può utilizzare un protocollo molto semplice. Se un processo possiede una risorsa e ne richiede un'altra che non gli può essere assegnata immediatamente, allora si forza la prelazione su tutte le risorse in suo possesso, che vengono rilasciate e si aggiungono alla lista di quelle in attesa. Il processo viene rilanciato di nuovo soltanto dopo che potrà ottenere sia le risorse in attesa che quelle nuove.
- per impedire l'attesa circolare, si impone un ordinamento totale su tutti i tipi di risorse e ciascun processo può richiederle solo in ordine crescente di numerazione. In generale, si può procedere in questo modo. Tramite una funzione iniettiva  $f$ , l'insieme delle risorse viene numerato e si associa ad ogni risorsa un numero intero distinto. Se un processo richiede un qualsiasi numero di istanze di un tipo di risorsa, per esempio  $R_i$ , il processo può richiedere istante di tipo  $R_j$  soltanto se  $f(R_j) > f(R_i)$ .

I meccanismi di prevenzione che abbiamo appena descritto si basano sull'intercettazione delle richieste. Questo metodo, però, può causare uno scarso utilizzo dei dispositivi, riducendo quindi il throughput. Il modo più semplice sarebbe quello di richiedere che ogni processo dichiari il **numero massimo** di risorse di cui avrà bisogno. Un ipotetico algoritmo, quindi, esaminerà dinamicamente lo stato di assegnazione delle risorse per garantire che non si verifichi un caso di attesa circolare. Lo **stato di allocazione delle risorse** sarà definito dal numero di risorse disponibili allocate, e al numero massimo di risorse richieste da ciascun processo. Uno stato può essere **sicuro** se il sistema è in grado di assegnare risorse a ciascun processo in un certo ordine e, quindi, di impedire lo stallo. Il sistema si trova in uno stato sicuro solo se esiste una **sequenza sicura** tra i processi.

Quando il sistema per l'assegnazione delle risorse è tale che ogni tipo di risorsa ha una sola istanza, per evitare situazioni di stallo si può far uso di una variante del grafo di assegnazione delle risorse. Si introduce un nuovo tipo di arco, detto **arco di rivendicazione**, indicato con una linea tratteggiata, che indica che un processo T può richiedere la risorsa R in un qualsiasi momento futuro. Le risorse sono rivendicate a priori dal sistema. Questo **algoritmo col grafo di assegnazione** non è utilizzabile in presenza di sistemi che hanno più istanze per ogni tipo di risorsa. Così, ricorriamo all'**algoritmo del banchiere**. Ogni processo deve dichiarare a priori il massimo impiego di risorse. Quando un processo richiede una risorsa, non può essere servito immediatamente. Una volta allocate le risorse, deve restituirle entro un tempo finito. In questo ambito, possono essere realizzati anche altri due algoritmi. L'**algoritmo di**

**verifica della sicurezza**, che scopre se il sistema è in uno stato sicuro o no, e **l'algoritmo di richiesta delle risorse**, che descrive se le risorse possono essere soddisfatte mantenendo la condizione di sicurezza.

Finora abbiamo descritto meccanismi per prevenire o evitare lo stallo. Spesso, però, lo stallo si può anche verificare e, di fronte a questa evenienza, abbiamo due opzioni:

- un algoritmo che esamini lo stato del sistema per stabilire se si è verificato uno stallo.
- un algoritmo che ripristini il sistema dalla condizione di stallo.

Andiamo, adesso, ad esaminare alcuni meccanismi che potrebbero rilevare le situazioni di stallo. Se tutte le risorse hanno una singola istanza, si fa uso di una variante del grafo di assegnazione delle risorse, detta **grafo di attesa**, dove si rimuovono i nodi dei tipi di risorse e si compongono archi tra i processi. Per individuare lo stallo, bisogna individuare un ciclo, quindi si deve aggiornare lo stato di attesa e invocare periodicamente un algoritmo che individui un ciclo. In caso di più istanze di una singola risorsa, allora useremo una variante dell'algoritmo del banchiere. Quando, e quanto spesso, richiamare l'algoritmo di rilevamento dipende dalla frequenza con la quale si verificano i deadlock e dal numero di processi che sono influenzati dal deadlock. L'algoritmo può essere chiamato ogni volta che una richiesta di assegnazione non può essere soddisfatta immediatamente.

Rilevato lo stallo, esso si può eliminare in diversi modi. Si può notificare a terzi l'avvenuta dello stallo o, semplicemente, possiamo fare in modo che il sistema ripristini automaticamente la situazione prima dello stallo. Il sistema, a sua volta, può farlo in due modi, che vedremo ora nel dettaglio:

- **terminazione del processo.** Si possono sia terminare tutti i processi in stallo, oppure si può terminare un processo alla volta fino all'eliminazione del ciclo. Terminare un processo non è sempre la soluzione ottimale. Il processo potrebbe essere in procinto di effettuare aggiornamenti sulle risorse e, quindi, ci sarebbero poi incoerenze nei dati.
- **prelazione delle risorse.** Le risorse si sottraggono in successione ad alcuni processi e si assegnano ad altri finché viene interrotto il ciclo di stallo. Per prima cosa si seleziona una vittima, cioè i processi che devono essere sottoposti a prelazione. Poi, si cerca di ristabilire il processo ad un precedente stato sicuro, in una procedura detta **rollback**. Durante questo procedimento è necessario che non

si verifichi la starvation, cioè le risorse non devono essere sottratte allo stesso processo.



# MEMORIA CENTRALE

La memoria non è altro che un grande **vettore di byte**, dove il processore preleva e deposita i dati in attesa di essere elaborati. La memoria centrale e i conseguenti **registri**, sono le uniche aree di memoria a cui la CPU può accedere direttamente. Infatti, non esistono delle istruzioni macchina in grado di accettare gli indirizzi dei dischi come parametri. I registri della CPU, invece, sono accessibili più volte all'interno dello stesso ciclo di clock. Questa cosa risulta impossibile da fare con la memoria centrale, dato che è estremamente più lenta. Questo enorme divario di velocità può portare a situazioni di **stallo**, in cui si verificano inconsistenze dei dati. Per risolvere questo problema è stata introdotta la memoria **cache**, una memoria molto veloce che si trova tra i registri e la memoria centrale. Bisogna suddividere la memoria in modo che ogni processo abbia un proprio spazio di indirizzamento. Per farlo, utilizziamo due registri:

- **registro base**: indirizzo di memoria più basso.
- **registro limite**: indirizzo di memoria più alto.

In generale, quando un programma non è in esecuzione, risiede nel disco sotto forma di un file binario. Una volta lanciato, il programma viene caricato in memoria e inserito nel contesto di un processo. La maggior parte dei sistemi permette al programma di risiedere in qualsiasi parte della memoria e, per essere eseguito, deve passare attraverso una serie di passi. Inoltre, è importante notare che gli indirizzi del programma sono di solito **simbolici** ed è il compilatore ad **associare (bind)**, gli indirizzi simbolici a quelli fisici. Questo tipo di associazione può essere fatta sia in fase di **compilazione**, dove si genera del codice assoluto, di **caricamento**, dove si genera codice allocabile o in fase di **esecuzione**, dove si ritarda l'associazione. Un indirizzo generato dalla CPU è normalmente un **indirizzo logico**, mentre un indirizzo visto dall'unità di memoria è un **indirizzo fisico**. Durante la fase di associazione, indirizzi logici e fisici non coincidono e, quindi, ci si riferisce ad essi come **indirizzi virtuali**. L'insieme di tutti gli indirizzi generati da un programma viene definito **spazio degli indirizzi logici/fisici**.

L'associazione di indirizzi virtuali a quelli fisici, è svolta dall'**MMU**, acronimo di *Memory Management Unit*. Per migliorare l'utilizzo della memoria, può essere utile passare ad un **caricamento dinamico**. Un processo viene caricato in memoria solo quando viene

richiamato. In questo caso, una procedura viene chiamata solo quando serve ed è molto utile nella gestione degli errori e, inoltre, non richiede molto supporto da parte del SO, dato che sta al programmatore gestire le procedure che permettono il caricamento dinamico.

Le **librerie collegate dinamicamente (DLL)** sono librerie di sistema che vengono collegati ai programmi quando vengono eseguiti. Possiamo trovarci anche di fronte ad un **collegamento statico**, in cui le librerie vengono trattate come un qualsiasi modulo o oggetto. Molto simile al caricamento dinamico, è il **collegamento dinamico**, che permette di caricare solo le librerie che servono durante l'esecuzione. Le DDL sono **librerie condivise**, cioè possono essere condivise tra più processi. Un altro concetto molto importante è l'**overlay**. Con questo termine si intende la possibilità di mantenere in memoria solo le istruzioni e i dati che si usano con maggior frequenza. Quando c'è bisogno, queste istruzioni vengono caricate in memoria, rimpiazzando quelle non più in uso. Viene usato soprattutto dai sistemi che hanno una memoria limitata e devono contenere processi di grandi dimensioni. Come abbiamo già detto, un processo, per essere eseguito, deve poter essere caricato in memoria. A volte, però, si potrebbe avere l'esigenza di aumentare il grado di multiprogrammazione e, quindi, è necessario spostarlo dalla memoria centrale ad una memoria ausiliaria. Questo procedimento è detto **avvicendamento dei processi (swapping)**. Possiamo avere due tipi di swapping:

- **swapping standard.** Il classico spostamento dalla memoria centrale a quella ausiliaria. Per fare ciò viene mantenuta una ready queue, con tutti i processi pronti e, quando lo scheduler decide di avviare un determinato processo, il controllo passa al dispatcher, che verifica la posizione del processo ed effettua le operazioni di upload o download. La latenza di dispatch è generalmente molto alta.
- **swapping nei sistemi mobili.** I dispositivi mobili utilizzano, di solito, memorie flash, visto il numero limitato di spazio. In questo caso, è impossibile effettuare lo swapping e, quindi, si procede con la terminazione dei processi per liberare la memoria.

La memoria centrale deve contenere sia il SO che i vari processi utenti. Per questo motivo, è necessario assegnare le diverse parti della memoria nella maniera più efficiente. Uno dei primi metodi che analizzeremo è l'**allocazione contigua**. Prima di scendere nel dettaglio, affrontiamo il concetto di **protezione della memoria**. Possiamo evitare che un processo acceda ad un'area di memoria che non gli compete, utilizzando

contemporaneamente un registro di rilocazione e un registro limite. Tali registri contengono un pool di indirizzi fisici della memoria, in cui il processo può operare. Il SO cambia dinamicamente le proprie dimensioni, eliminando i dati non utilizzati. Uno dei metodi più semplici per allocare memoria, è dividerla in **partizioni** di dimensione variabile, dove ogni partizione contiene un processo. Così, il SO avrà uno schema in cui può visualizzare le partizioni occupate e quelle disponibili. Inizialmente, tutta la memoria è vista come un grande blocco, detto **bucco**, che verrà occupato dal primo processo che ha dimensioni minori o uguali a quelli del buco. Vediamo tre metodi per scegliere un buco da occupare:

- **First-fit.** Si assegna il primo buco abbastanza grande.
- **Best-fit.** Si assegna il più piccolo buco in grado di contenere il processo.
- **Worst-fit.** Si assegna il buco più grande.

È abbastanza chiaro che i primi due sono meglio del terzo ma, nel caso dovessimo paragonare first-fit e best-fit, avremo una velocità maggiore con il primo.

Entrambi questi criteri, però, soffrono di **frammentazione**. Caricando e rimuovendo i processi dalla memoria, si frammenta lo spazio libero della memoria. Abbiamo due tipi di frammentazione:

- **frammentazione esterna:** lo spazio di memoria disponibile è necessario per riempire il buco, ma non è contiguo. La frammentazione esterna è un problema molto grave e, se non corretta, si potrebbe bloccare la memoria.
- **frammentazione interna:** la memoria allocata è leggermente più grande di quella richiesta e, questa memoria non viene utilizzata.

Per questo, in presenza di frammentazione esterna, è necessario risolvere il problema. Lo si può fare attraverso la **compattazione**. Si riunisce la memoria frammentata in un unico blocco. La compattazione, però, non è sempre la soluzione migliore, perché molto costosa. Per questo, un metodo davvero efficace risulta essere la **non-contiguità** dello spazio degli indirizzi. Si tratta della tecnica su cui si basa maggiormente la **paginazione**.

Come detto, la paginazione permette di evitare la frammentazione esterna e la sua conseguente compattazione. Grazie ai molteplici vantaggi offerti, è usata nella maggior parte dei SO. Il metodo di base per implementare la paginazione consiste nel suddividere la memoria fisica in blocchi di dimensione fissa, detti **frame**, e la memoria logica in blocchi di pari dimensioni, dette **pagine**. Ogni indirizzo generato dalla CPU

contiene un **numero di pagina** e un **offset di pagina**. Il numero indica la posizione all'interno della tabella delle pagine, mentre l'offset indica il frame a cui fa riferimento. Le dimensioni delle pagine e dei frame sono generalmente definiti dall'hardware. Come abbiamo già anticipato, la paginazione evita la frammentazione esterna, dato che qualsiasi frame libero si può assegnare ad un processo che ne ha bisogno. Tuttavia, potremo trovarci di fronte alla frammentazione interna, poiché lo spazio richiesto dal processo non è multiplo delle dimensioni delle pagine e, così, l'ultimo frame potrebbe non essere completamente pieno. Il consiglio per evitare questo problema è quello di usare pagine di piccole dimensioni ma, col tempo, questa cosa è diventata impossibile, dato che i processi e i flussi di dati sono andati sempre ad aumentare. Durante l'esecuzione, il sistema esamina la dimensione del processo in pagine e, se sono richieste  $n$  pagine, devono esserci almeno  $n$  frame. Il SO gestisce la memoria fisica, ma ha bisogno di essere informato sui dettagli dell'allocazione, prendendo tutte le informazioni che servono dalla **tabella dei frame**, verificando il numero di frame liberi e, poi, caricando il processo in memoria.

All'interno dei moderni sistemi, la tabella delle pagine è molto grande e deve essere salvata per forza nella memoria centrale, dove è puntata da un **registro di base della tabella delle pagine (PTBR)**, in modo che, modificando solo questo registro, si riducono notevolmente i tempi durante il cambio di contesto. Questo metodo ha un problema, infatti richiede due accessi in memoria (uno per la tabella e uno per la locazione fisica). Per risolvere questo problema ricorriamo al **TLB**, acronimo di *Translation Look-aside Buffer*, che è una piccola cache hardware. Si tratta di una memoria molto veloce, che contiene una **chiave** e un **valore**. Quando la CPU genera un nuovo indirizzo logico, si presenta il suo numero di pagina al TLB. Se il suo indirizzo, e la relativa chiave, sono presenti, si ottiene direttamente l'indirizzo fisico. Nel caso in cui l'indirizzo appena generato dalla CPU non sia presente all'interno del TLB, siamo costretti ad accedere alla memoria, avendo così, un **insuccesso del TLB**. Quando andiamo ad aggiornare il TLB, è possibile trovarlo pieno e, quindi, bisognerà scegliere un elemento per sostituirlo. È possibile usare l'elemento usato meno recentemente, o compiere una scelta completamente casuale. Per evitare che vengano rimossi elementi vitali, uno di essi può essere **vincolato**, cioè non rimovibile. Alcuni TLB utilizzano gli **identificatori dello spazio di indirizzi (ASID)**, che identificano in modo univoco ogni processo in corrispondenza del suo spazio di indirizzo. La percentuale di volte che un numero di pagina si trova all'interno del TLB è detto **tasso di successo**.

In un ambiente paginato, la protezione della memoria è affidata ai **bit di protezione**, che sono associati ad ogni frame. Un bit può determinare se una pagina si può leggere

o scrivere. Di solito, però, si tende ad associare un ulteriore bit, detto **bit di validità**. Può essere impostato a *valido*, se la pagina corrispondente è nello spazio di indirizzi logici del processo, o a *non valido*, cioè il contrario. In caso di indirizzi illegali, si genera un'eccezione. Spesso, però, i processi utilizzano solo una piccola parte del loro spazio di indirizzi e sarebbe uno spreco creare una tabella delle pagine che resterebbe per gran parte inutilizzata. Per questo, si ricorre ad un **registro di lunghezza della tabella delle pagine (PTRL)**, per indicare la lunghezza effettiva delle tabelle.

Un vantaggio della paginazione può essere anche la possibilità di realizzare **pagine condivise**, cioè condividere codice comune. Un codice può essere condiviso se è **rientrante**, cioè se non è auto-modificante, e non cambia mai durante l'esecuzione. In questo caso, due o più processi, hanno la possibilità di eseguire lo stesso codice nello stesso momento. Ciascun processo ha una propria copia dei registri e una sua memoria, dove conserva i propri dati durante l'esecuzione.

Esistono diversi metodi per realizzare la tabella delle pagine e, ora, ne vedremo alcuni:

- **paginazione gerarchica**: alcuni sistemi dispongono di uno spazio di indirizzi molto grande, quindi, di conseguenza, anche una tabella delle pagine molto grande. Per questo motivo, potremo suddividerla in parti più piccole, magari adottando un tipo di paginazione a due livelli (**tabella paginata**). Si tratta di un metodo non adatto per le architetture a 64 bit, dato che si dovrebbe andare a paginare la tabella in quattro o più livelli e risulterebbero troppi accessi in memoria.
- **tabella delle pagine di tipo hash**: si associa ad ogni indirizzo una funzione hash, che è il numero della pagina virtuale. Utilizzando funzioni hash, però, possiamo spesso incorrere nelle cosiddette **collisioni**, cioè quando una funzione hash, affidata a due chiavi diverse, genera un medesimo indirizzo. Per evitare questo problema, si usa una **lista concatenata** per disambiguare. Ogni elemento della tabella è composto dal **numero di pagina virtuale**, **indirizzo del frame** e il **puntatore al successivo elemento della lista**. L'algoritmo applica la funzione hash al numero di pagina virtuale, identificando un elemento della tabella. Poi, confronta l'indirizzo logico con i vari elementi della lista concatenata.
- **tabella delle pagine invertita**: ha un elemento per ogni frame. Ciascun elemento ha un indirizzo virtuale della pagina memorizzata in quella reale locazione di memoria, con informazioni sul processo che risiede in quella pagina. Quindi, ogni indirizzo virtuale avrà un **id del processo**, un **numero di pagina** e un **offset di pagina**. L'id del processo è l'identificatore dello spazio di indirizzo. Quando si fa

riferimento alla memoria si genera una parte dell'indirizzo virtuale, formato dall'id e dal numero di pagina. Se si trova corrispondenza, si genera un indirizzo fisico dell'elemento insieme all'offset.



# MEMORIA VIRTUALE

Finora abbiamo visto come le tecniche di allocazione permettano di tenere in memoria più processi per favorire la multiprogrammazione. Tuttavia, serve che l'intero processo risieda in memoria. Per far fronte a questo problema, ci affidiamo alla memoria virtuale. Essa si basa sulla separazione tra memoria logica, vista dall'utente e memoria fisica. In parole poche, un programmatore non deve più preoccuparsi della quantità di memoria fisica disponibile, perché il sistema offrirà una memoria virtuale molto ampia. Prima di proseguire, ci teniamo a dire che lo spazio degli indirizzi virtuali si riferisce alla collocazione dei processi in memoria dal punto di vista logico. Esso è collocato tra l'heap e lo stack ed è un grande buco vuoto. Questi buchi, poi, andranno riempiti grazie all'espansione dell'heap o dello stack, permettendo il processo di memorizzazione virtuale.

Una delle principali tecniche di memorizzazione virtuale, risulta essere la **paginazione su richiesta**. Le pagine vengono caricate in memoria solo quando richieste durante l'esecuzione del programma. Le pagine a cui non si accede, quindi, non vengono mai caricate in memoria. Per favorire questo processo, cioè per distinguere le pagine che sono in memoria centrale e altre in una memoria secondaria, utilizzeremo sempre la tecnica del **bit di validità**, che sarà *valido* se la pagina corrisponde ed è presente in memoria, oppure *non-valido*, se la pagina non corrisponde o si trova nella memoria secondaria. Se un processo tenta di accedere ad una pagina non caricata in memoria, allora, si ha un **page-fault**, un'eccezione. Per ovviare a questo problema, si controlla la tabella interna di questo processo, per verificarne la validità. Se non è valido, allora termina, se è valido, allora si carica la pagina in memoria, individuando un frame libero. Si trasferisce la pagina nel frame appena trovato. Poi, si modifica la tabella interna, per avvisare che si trova effettivamente in memoria. Si riavvia dopo l'interruzione. Di solito, si mantiene sempre una **lista di frame liberi**, che serve soltanto per soddisfare queste richieste.

Il supporto hardware alla paginazione su richiesta è lo stesso per la paginazione, cioè una **tabella delle pagine** e una **memoria secondaria**, che conserva le pagine non presenti in memoria centrale. Un requisito fondamentale per la paginazione su richiesta è la possibilità di riprendere l'esecuzione di un'istruzione dopo il page-fault. Avendo

salvato lo stato del processo, esso verrà riavviato allo stesso punto, con lo stesso stato, ad eccezione per la presenza della suddetta pagina mancante in memoria. Si tratta di un requisito facile da rispettare nella maggior parte dei casi.

La paginazione su richiesta può avere effetti anche sulle **prestazioni** di un calcolatore. Per visualizzarlo, possiamo calcolare il **tempo d'accesso effettivo**. Per calcolarlo dobbiamo definire il **tempo d'accesso alla memoria**, detto *ma*, che di solito è 10 a 200 nanosecondi. Indicheremo, poi, con *p*, la probabilità che si verifichi un page-fault, di solito molto vicino allo zero. Infine, ci serve il **tempo di gestione del page-fault**, che indichiamo con *tgpf*. Quest'ultimo è, di solito, una sequenza, che effettua tutte le verifiche per il ripristino che abbiamo visto prima (validità, frame liberi, riavvio). Fatte queste premesse, per calcolare il *TAE*, il tempo d'accesso effettivo, scriveremo la seguente espressione:

$$\bullet \quad TAE = (1 - p) * ma + p * tgpf$$

Solitamente, quando viene chiamata una `fork()`, si crea un processo figlio, dal padre, che è anche un suo duplicato. Quest'operazione, però, non è efficiente. Per questo, ricorreremo alla tecnica di **copiatura su scrittura**. Essa si basa sulla condivisione iniziale delle pagine tra processi genitori e processi figli. Così facendo, le pagine condivise, vengono copiate su scrittura, quindi, se un processo scrive su una pagina condivisa, il sistema ne crea una copia.

Descrivendo le prestazioni della memoria virtuale, abbiamo supposto un numero basso di page-fault. Tuttavia, si tratta di una rappresentazione poco corretta. In questo caso, potrebbero insorgere problemi di **sovrallocazione**. Si verifica un page-fault e non c'è nessun frame libero, cioè tutta la memoria è in uso. Il sistema potrebbe terminare il processo, oppure forzare la liberazione dei frame e diminuire la multiprogrammazione. Nessuna delle due è la soluzione migliore. Per questo, si ricorre alla **sostituzione delle pagine**. Se nessun frame è libero, allora ne viene liberato uno inutilizzato, scrivendo il suo contenuto nell'area di swap. Generalmente, si usa un **algoritmo di sostituzione** per scegliere il **frame vittima**, scrivendo poi la pagina sul disco. Questo metodo richiede due trasferimenti di pagine, uno fuori e uno dentro. Per non diminuire le prestazioni, si adopera un **bit di modifica** che, se settato a 1, significa che la pagina ha scritto un byte. Quando si va a scegliere la vittima, l'algoritmo di sostituzione controlla questo bit che, se settato a 0, indica che la pagina è già stata scritta in memoria. La sostituzione delle pagine è fondamentale per completare la paginazione su richiesta e separare definitivamente la memoria logica da quella fisica. Andiamo ad analizzare, adesso, alcuni algoritmi di sostituzione delle pagine:

- **sostituzione delle pagine secondo l'ordine di arrivo (FIFO)**: è obiettivamente l'algoritmo più semplice, cioè quello che associa ad ogni pagina l'istante di tempo in cui quella pagina è stata portata in memoria. Se si deve sostituire, si sostituisce quella presente in memoria da più tempo. Come già detto, è un algoritmo semplice, ma non efficiente, dato che può sfociare nella cosiddetta **anomalia di Belady**, in cui i page-fault aumentano con il numero di frame assegnati.
- **sostituzione ottimale delle pagine**: per evitare l'anomalia di Belady, si è ricorso a questo algoritmo. Si sostituisce la pagina che non verrà usata per il periodo di tempo più lungo. Si tratta di un algoritmo difficile da realizzare, perché è difficile prevedere quali pagine non verranno usate. Per questo motivo, è usato soltanto in fase di analisi.
- **sostituzione delle pagine usate meno recentemente (LRU)**: dato che i due algoritmo descritti hanno dei difetti, con questo si è cercato di trovare un compromesso tra i due. Si sostituisce la pagina che non è stata usata per il periodo di tempo più lungo. Questo algoritmo è molto utilizzato, ma presenta tanti problemi sulla sua implementazione. In generale, si hanno problemi a definire i frame dal momento del loro ultimo uso. Per ovviare a ciò, usiamo o dei **contatori**, che si incrementano ad ogni riferimento alla memoria, o uno **stack**, dove si immagazzinano i numeri di pagina. Così facendo, la pagina in cima allo stack sarà la pagina meno utilizzata.
- **sostituzione delle pagine per approssimazione a LRU**: non tutti i sistemi hanno un supporto hardware adatto per LRU. Per questo, è possibile utilizzare un **bit di riferimento**, che viene assegnato automaticamente ogni volta che si fa riferimento a quella pagina. Così facendo, almeno, sapremo se le pagine sono state usate o no, anche se non possiamo conoscerne l'ordine. Per ovviare a questo problema, si potrebbero memorizzare i bit di riferimento ad **intervalli regolari**, anche se l'unicità non è garantita. Oppure, si potrebbe usare un **algoritmo di seconda chance**. Se il valore de bit di riferimento è 0, si sostituisce la pagina, se è 1, le si dà una seconda chance, facendo uso di una coda circolare, in cui il puntatore punta alla prima pagina da sostituire. Questo algoritmo si può migliorare aggiungendo un **bit di modifica** che, in caso di (0, 0) significa che la pagina non è stata usata recentemente e né modificata, quindi può essere sostituita.
- **sostituzione basata sul conteggio**: si utilizza un contatore dei riferimenti. Questo algoritmo, a sua volta, può essere di sostituzione delle pagine meno

frequentemente usate (**LFU**), cioè sostituisce le pagine con il conteggio più basso, o di sostituzione delle pagine più frequentemente usate (**MFU**), cioè sostituisce sempre la pagina col conteggio più basso, ma basandosi sul fatto che essa è stata appena inserita e mai usata. Sono due algoritmi poco usati.

- **sostituzione con buffering delle pagine:** i sistemi hanno, di solito, un **gruppo di frame liberi**. Quando si verifica un page-fault e si è scelta la pagina vittima, allora essa si trasferisce in questo gruppo, in modo che il processo possa riprendere subito l'esecuzione senza dover aspettare che la pagina venga scritta in memoria secondaria.

Dopo aver descritto i vari metodi di sostituzione delle pagine, soffermiamoci adesso sull'**allocazione dei frame**, cioè come il sistema assegna i frame liberi ai diversi processi. Le varie strategie sono soggette a diversi vincoli, tra cui un **numero minimo** di frame. Al decrescere del numero di frame allocati, aumenta il tasso di page-fault, rallentando l'esecuzione. Di solito, il numero minimo dei frame, viene definito dall'architettura del calcolatore.

Il metodo più semplice per allocare  $m$  frame a  $n$  processi, è quello di effettuare un'**allocazione uniforme**, del tipo  $m/n$ . Però, diversi processi, potrebbero aver bisogno di quantità di memoria differenti. Per questo, è possibile ricorrere ad un'**allocazione proporzionale** secondo le necessità dei processi. In questo frangente, anche gli algoritmi di sostituzione si possono catalogare in due gruppi:

- **sostituzione globale:** si sceglie il frame vittima dall'insieme di tutti i frame. Il processo non può controllare il proprio tasso di page-fault, dato che non può basarsi solo sui suoi dati.
- **sostituzione locale:** si sceglie il frame vittima solo dal proprio insieme di frame. Un processo può venir penalizzato perché non gli vengono rese disponibili altre pagine in memoria.

Consideriamo un processo che non disponga di un numero di frame sufficiente e, quindi, incorre in un page-fault. Qui, allora, bisognerà sostituire la pagina ma, dato che tutte le sue pagine sono attive, ne viene sostituita una che sarà subito necessaria e che, quindi, dovrà essere subito riportata in memoria. Questa procedura è detta **thrashing**. Un processo in thrashing spende più tempo per la paginazione che per l'esecuzione. Il thrashing si verifica quando il sistema rileva un basso utilizzo di CPU. Per aumentare la produttività, aumenta il grado di multiprogrammazione, introducendo nuovi processi in memoria che, di conseguenza, allocano nuovi frame a discapito dei processi già in

memoria e, quindi, vanno anch'essi in page-fault. Questo non fa altro che far svuotare la coda dei processi in esecuzione e, automaticamente, diminuisce l'utilizzo della CPU. E, così, torniamo alla situazione iniziale, quando il SO rileva una bassa produttività e continua ad aumentare il grado di multiprogrammazione in un loop infinito. Per questo motivo, il thrashing causa gravi problemi di prestazioni. È anche possibile limitare questo fenomeno, adoperando **algoritmi di sostituzione locale**, ma non lo risolveremo del tutto. Per questo motivo, è necessario che un processo abbia a disposizione tutti i frame di cui necessita. Uno degli approcci più utilizzati è il **working-set**, che definisce il **modello di località** d'esecuzione del processo. Una località è un insieme di pagine attive e, questo sistema, permette al processo di spostarsi da località in località. Ma vediamolo ora nel dettaglio.

Il working-set utilizza un parametro, che indicheremo con  $\Delta$ , detto **finestra del working-set**. Si esaminano i più recenti  $\Delta$  riferimenti alle pagine. L'insieme di pagine nei più recenti  $\Delta$  riferimenti è il working-set. Se una pagina è attiva, allora è nel working-set. Altrimenti, ne esce  $\Delta$  unità di tempo dopo il suo ultimo riferimento. Questo modello risulta avere successo, ma non è sempre efficace per risolvere il thrashing. Per questo motivo, si ricorre alla **frequenza dei page-fault**. Si tratta di una strategia più diretta, che serve a prevenire il thrashing. Se la frequenza è eccessiva, significa che c'è bisogno di allocare più frame. Altrimenti, se è bassa, significa che un processo dispone di troppi frame. In genere, si possono fissare dei **limiti** entro cui occorre allocare o revocare i frame.

In generale, molti SO moderni, tendono ad evitare il thrashing semplicemente costruendo abbastanza memoria fisica.



# MEMORIA DI MASSA

Il principale sistema di memorizzazione di massa nei computer è la memoria secondaria. I dischi magnetici e i dispositivi NVM (*non-volatile memory*) costituiscono il principale dispositivo di memorizzazione secondaria per i moderni computer.

Un **disco rigido** è generalmente composto da un **piatto**, come quello di un CD, il cui diametro varia da 1.8 a 3.5 pollici, con le due superfici ricoperte di **materiale magnetico**. Una **testina** di lettura e scrittura è sospesa su ogni lato del piatto. Le testine sono attaccate al **braccio** del disco che le muove in blocco. La superficie del disco è divisa in **tracce circolari**, che sono al loro volta suddivise in **settori**, a dimensione fissa. L'insieme delle tracce è detto **cilindro**. Quando il disco è in funzione, un motore lo fa girare ad alta velocità, espressa in **RPM**, cioè giri a minuto. La **velocità di rotazione** è in relazione alla **velocità di trasferimento**, cioè il tempo con cui i dati fluiscono. Un altro concetto importante è il **tempo di posizionamento**, che mette in relazione il **tempo di ricerca**, quello impiegato per spostare il braccio al cilindro desiderato, e la **latenza di rotazione**, impiegato dalla testina a spostarsi sul settore. Con le testine possono verificarsi i cosiddetti incidenti da **urto della testina**, cioè quando la punta supera il cuscinetto d'aria e sbatte sul disco, graffiando la superficie magnetica.

Al contrario, i **dispositivi NVM** stanno acquisendo sempre più popolarità. Possono essere comunemente chiamati **SSD** (*solid-state drive*), cioè **dischi allo stato solido**. Essi possono assumere la forma di un'**unità USB**, ma anche di un **modulo DRAM** (*Dynamic Random Access Memory*). Gli SSD possono essere più affidabili dei dischi rigidi, dato che non hanno componenti meccaniche e sono più veloci, dato che manca il tempo di posizionamento e la latenza di rotazione. Hanno, però, una capacità inferiore. Un dispositivo di memoria secondaria è collegato al sistema tramite un **bus**, cioè un canale di comunicazione tra dispositivi e periferiche. I trasferimenti dati su un bus vengono eseguiti da speciali processori, detti **adattatori bus-host (HBA)**, che hanno una cache integrata per favorire il trasferimento. I dispositivi di memoria secondaria sono visti come grandi vettori di blocchi logici e, per fare riferimento ad essi, viene utilizzato un indirizzo di blocco logico.

Uno dei compiti del sistema operativo è quello di garantire un uso efficiente

dell'hardware. Nel caso dei dischi rigidi, c'è bisogno di tempi di accesso contenuti e ampiezza di banda elevata. Gestendo l'ordine delle richieste di I/O relative al disco, si può ovviare a questi due aspetti. Queste richieste, essendo in una coda, devono essere gestite da uno scheduler, che adopera opportuni **algoritmi di scheduling**:

- **scheduling in ordine di arrivo (FCFS)**: lo abbiamo già descritto approfonditamente. La prima richiesta che arriva, essa viene eseguita. Algoritmo semplice ma, come sempre, non molto veloce e poco efficiente.
- **scheduling per brevità (SSTF)**: seleziona la richiesta con il tempo di ricerca minimo, con la testina che si sposta sulla traccia più vicina. Anch'esso può portare a starvation e non risulta ottimale.
- **scheduling SCAN**: il braccio parte da un estremo del disco e si sposta verso l'altro estremo, servendo man mano le richieste mentre attraversa i cilindri. Una volta giunto a destinazione, fa retromarcia e riprende.
- **scheduling C-SCAN**: detto anche circolare, anch'esso sposta la testina da un estremo all'altro del disco. Una volta giunta a destinazione, però, non fa retromarcia, ma torna direttamente all'inizio, senza servire le richieste lungo il percorso.
- **scheduling LOOK e C-LOOK**: superano alcune limitazioni di SCAN e C-SCAN. Quest'ultime effettuano sempre scansioni complete del disco, mentre questi proposti valutano ad ogni passo, se ci sono scansioni da eseguire in quella direzione, altrimenti si inverte il verso della testina.

Per scegliere l'opportuno algoritmo di scheduling, è necessario valutare la mole di richieste. Esse possono essere influenzate dal metodo di allocazione dei file, ma anche la posizione della directory può influire è importante.

Il SO non gestisce soltanto lo scheduling. Uno dei suoi compiti può essere la **formattazione di basso livello**. Un dispositivo di memorizzazione nuovo è una *tabula rasa*, composto solo dai suoi elementi fisici. Effettuare una formattazione di basso livello significa riempire il dispositivo con una speciale struttura dati, composta da **un'intestazione**, **un'area per i dati** e **una coda**, per ogni locazione di memoria. L'intestazione e la coda contengono informazioni per il controllore e un **codice di correzione degli errori (ECC)**. Una volta fatto questo, il SO deve registrare le proprie strutture dati nel disco. Per farlo, suddivide il dispositivo in **partizioni**, per trattarle come unità a sé stanti. Il secondo passo è la **gestione del volume**, di solito un passaggio

implicito. Il terzo passo è la formattazione logica, cioè la **creazione di un file system**. Per una maggiore efficienza, il file system accoppia i blocchi in unità a gruppi, detti **cluster**.

Affinché un calcolatore possa entrare in funzione, è necessario un **programma di avviamento iniziale**, detto **bootstrap**. Si occupa di inizializzare tutti gli aspetti del sistema, i registri e i controllori. Le unità di memorizzazione di massa sono portate ai malfunzionamenti, che si tramutano in blocchi difettosi. Essi possono essere gestiti manualmente, cioè si effettua una scansione e si individuano i blocchi difettosi, dicendo poi al file system di non utilizzarli. Ovviamente, esistono anche strategie più complesse, tipo quelle che prevedono un controllore che mantenga una lista di tutti i blocchi malfunzionanti e che usi dei settori di riserva per rimpiazzarli, in una tecnica chiamata **accantonamento dei settori**. Esistono, però, **errori irreversibili**, che non si possono riparare e che causano una perdita di dati.

Quando parliamo di avvicendamento in termini di memoria di massa, ci riferiamo a quella situazione che vede l'ammontare di quest'ultima alzarsi fino ad una soglia critica, fino a costringere il SO a spostare i processi verso l'**area di avvicendamento**.

Possiamo collocare l'area di avvicendamento:

- nel file system: qui si utilizzano le varie tecniche messe a disposizione per crearla, ma è molto facile incorrere in errori di frammentazione e, inoltre, i tempi sono molto lunghi, dato che i processi sono costretti ad attraversare tutte le strutture e le directory.
- in una partizione del disco a sé stante: qui bisogna usare un gestore dell'area di avviamento. I processi rimangono qui per poco tempo e, quindi, si evitano problemi di frammentazione.

Il progresso tecnologico ha reso le unità di memorizzazione sempre più piccole e meno costose. Per questo, i computer sono equipaggiati con molti dischi e, spesso, per migliorare l'affidabilità, vengono immagazzinate le informazioni in modo ridondante, in modo che un guasto non comporti la perdita di dati. Di solito, vengono realizzate **batterie ridondanti di dischi**, o **RAID**.

Come abbiamo già detto, l'affidabilità si migliora con la ridondanza. Vengono memorizzati dati che, normalmente, non servirebbero, ma che, in caso di guasto, possono andare a ricostruire le informazioni perse. Il metodo più semplice di ridondanza è il **mirroring**: ogni disco logico ha due dischi fisici e ogni scrittura si effettua su entrambi. L'alimentazione elettrica, però, è un problema. In caso di interruzione

improvvisa, ci sarebbero dati incoerenti su entrambi i dischi. Si può impostare la scrittura prima su un disco e poi su un'altra. Oppure, si potrebbe aggiungere una **memoria non volatile allo stato solido**, detta **NVRAM**, dove la scrittura si completa anche in caso di interruzione dell'alimentazione elettrica.

Il mirroring è, però, molto costoso. Per questo si tende a prediligere schemi per ridondanza a basso costo. Questi sono detti livelli RAID e ora li vediamo:

- **RAID 0**: distribuzione a livello di blocchi, ma senza ridondanza.
- **RAID 1**: si riferisce alla tecnica di mirroring, cioè una copiatura speculare.

Prima di proseguire, chiariamo il concetto di **bit di parità**. Ad ogni byte viene associato un bit che indica se gli 1 presenti nel byte sono in numero parità o dispari. Servono a identificare tutti gli errori su un singolo bit. Usando questi bit supplementari si riescono a individuare e correggere un maggior numero di bit.

- **RAID 2**: il sezionamento viene fatto a livello di byte, utilizzando i codici per la correzione degli errori (ECC). Il primo bit di ogni byte viene memorizzato nel disco 1, il secondo nel disco 2 e così via. I bit di parità vengono memorizzati singolarmente in dischi separati e differenti da quelli usati per i dati. Se un disco si guasta, i bit rimanenti nel byte e i bit di parità vengono usati per riparare il danno. Per quattro dischi dati, sono richiesti soltanto tre dischi RAID.
- **RAID 3**: il sezionamento avviene sempre a livello di byte, soltanto che c'è un disco in più dedicato ai bit di parità. Si tratta di un livello efficiente, ma piuttosto lento, dato che ogni disco è coinvolto da tutte le richieste.
- **RAID 4**: il sezionamento avviene a livello di blocchi, con un disco dedicato alla parità. Ha una grande tolleranza ai guasti, ma nell'unico disco di parità può verificarsi il classico collo di bottiglia, cioè un sovraccarico.
- **RAID 5**: il sezionamento avviene sempre a livello di blocchi, solo che ogni disco mantiene il bit di parità. Quindi, è simile al RAID 4, soltanto che, essendo i bit di parità distribuiti, non si verifica alcun collo di bottiglia, però la scrittura resta comunque lenta.
- **RAID 6**: il sezionamento avviene a blocchi, ma vengono memorizzate più informazioni ridondanti. Oltre a contenere i bit di parità, ogni disco contiene codici di correzione degli errori basati sulla teoria di Galois. Quindi, può supportare due

guasti e aumentare al massimo la ridondanza. Ovviamente, la scrittura risulta comunque lenta e sono molto costosi.

- **RAID 0+1:** è una combinazione di 0 e 1, atta a fornire sia la prestazioni del primo, che l'affidabilità del secondo. Risulta più affidabile dei RAID a blocchi, però richiede un numero superiore di dischi di memorizzazione e non supporta il guasto di due dischi.
- **RAID 1+0:** è simile al precedente, soltanto che effettua prima il mirroring e poi la distribuzione, in modo da sostenere il guasto di due dischi.

Un sistema RAID può essere implementato sia a livello software che al livello hardware. Con la prima, è il SO a gestire l'insieme dei dischi attraverso un controllore. Si tratta di un'opzione più lenta, rispetto ad una gestione hardware, anche se non richiede l'acquisto di componenti extra. Inoltre, i dischi RAID possono anche implementare funzionalità aggiuntive, tra cui l'**istantanea**, cioè un'immagine del file system com'era all'ultimo aggiornamento, la **replica**, che prevede la duplicazioni su locazioni diverse, e il **disco di scorta**, che sostituisce il disco danneggiato.

Il criterio principale per scegliere un livello RAID è, sicuramente, il tempo di ricostruzione, cioè quanto tempo si impiega a riparare un danno. Il RAID 0 si utilizza per ottenere grandi prestazioni in sistemi in cui la perdita di dati non risulterebbe critica. Il RAID 1 si usa per ottenere un rapido ripristino. I RAID 0+1 e 1+0 si usano per sistemi che non richiedono alte prestazioni e affidabilità. I RAID 5 si usano quando c'è una grande mole di dati.

I RAID, però, non assicurano sempre la disponibilità dei dati. Per ovviare a questo problema, si applica una **checksum** interna ad ogni blocco, che verificherà l'integrità dei dati.



# SISTEMI DI I/O

Sappiamo con certezza che i due principali compiti del calcolatore sono l'I/O e l'elaborazione. Il calcolatore deve occuparsi di gestire e controllare le operazioni e i dispositivi di I/O. Questi ultimi sono diversi per funzioni e velocità, quindi esistono diversi metodi di controllo. L'insieme di tutti i dispositivi di I/O costituisce il **sottosistema di I/O** del kernel. I **driver dei dispositivi** offrono al sottosistema di I/O un'interfaccia uniforme per l'accesso ai dispositivi.

Un dispositivo comunica con il SO inviando segnali attraverso un cavo o attraverso l'etere, invece comunica con il calcolatore attraverso un punto di connessione, detto **porta**. Se più dispositivi condividono una serie di fili, la connessione è detta **bus**, cioè un insieme di fili e un protocollo comune per lo scambio di messaggi. Generalmente una tipica struttura bus prevede un **bus PCIe**, che connette il sottosistema CPU-memoria ai dispositivi veloci e un **bus di espansione**, che connette i dispositivi lenti come mouse e tastiera. Un sistema è anche dotato di un **controllore**, cioè un insieme di componenti elettroniche che può far funzionare una porta, un bus o un dispositivo. Esistono controllori semplici, come quelli delle porte, e altri controllori che possono addirittura contenere un proprio microprocessore. Il processore fornisce comandi e dati al controllore per portare al termine trasferimenti di I/O tramite uno o più registri per dati e segnali di controllo. La comunicazione tra di loro può avvenire tramite lettura e scrittura di configurazioni di bit in questi registri. In alternativa, è possibile utilizzare la tecnica dell'**I/O memory mapped**, in cui i registri vengono mappati in un sottoinsieme dello spazio di indirizzi della CPU. Il controllo di un dispositivo di I/O comprende quattro registri:

- **data-in**, in cui la CPU legge per ricevere i dati.
- **data-out**, in cui la CPU scrive per emettere dati.
- **status**, che contiene alcuni bit che possono essere letti dalla CPU e indicano lo stato della porta.
- **control**, che può essere scritto per attivare un comando e per cambiare il modo di funzionamento del dispositivo.

Questo protocollo di comunicazione può sembrare intricato, ma la **negoziazione**, detta **handshaking**, è relativamente semplice. Assumiamo l'uso di due bit per coordinare la relazione tra CPU e controllore. Il controllore specifica il suo stato tramite il bit **busy**, che vale **1** quando è occupato. La CPU, invece, comunica le sue richieste tramite il bit **command-ready**, che vale **1** quando il controllore deve eseguire un comando. Vediamo i passi:

1. La CPU legge **busy** fino a quando non vale **0**.
2. La CPU scrive un byte nel registro **data-out**.
3. La CPU pone a **1 command-ready**.
4. Il controllore legge **command-ready** e pone ad **1 busy**.
5. Il controllore legge **data-out**, trova il byte da scrivere e compie le sue operazioni.
6. Il controllore pone a **0 command-ready** e il bit **error** di **status**, per notificare che tutto sia andato a buon fine.

Durante il passo 1, la CPU è in **interrogazione ciclica**, detta **polling**. Il polling è efficiente, però diminuisce la sua efficacia se la CPU itera senza trovare un dispositivo pronto. In tal caso, potrebbe essere necessario che sia direttamente il controllore a notificare alla CPU la prontezza del dispositivo. Per questa comunicazione, di solito, si utilizzano le **interruzioni**.

L'hardware della CPU ha un input, detto **linea di richiesta dell'interruzione**. Quando viene rilevato un segnale, la CPU salva lo stato corrente e salta alla **routine di gestione dell'interruzione**. Le interruzioni sono estremamente importanti, perché spesso i sistemi ne gestiscono centinaia al secondo. In parole poche il controllore genera un'interruzione sulla linea di richiesta dell'interruzione, che la CPU rileva e recapita al gestore delle interruzioni, che la gestisce e serve il dispositivo. In genere, esistono due tipi di linee di richiesta dell'interruzione:

- **interruzioni non mascherabili**, come errori di memoria irrecuperabili.
- **interruzioni mascherabili**, che possono essere disattivate dalla CPU prima dell'esecuzione di una sequenza critica che non deve essere interrotta.

Il meccanismo delle istruzioni accetta un **indirizzo**, che fa riferimento ad un **vettore di interruzioni**, che però non è sempre utilizzabile, dato che i dispositivi utilizzano più gestori di interruzioni. Per questo motivo, si tende spesso ad accoppiare il vettore con una **lista concatenata**, in cui ogni elemento punta ad una lista di gestori. Vengono

anche identificati dei livelli di priorità delle interruzioni, che permettono alla CPU di differire la gestione delle interruzioni, tra quelle mascherabili e non mascherabili. Le interruzioni si possono utilizzare anche per gestire le eccezioni. Spesso, però, le interruzioni risultano difficili da implementare e, per questo motivo, si adoperano particolari **system-call**, che permettono di generare le classiche **trap**, o interruzioni software.

Quando un dispositivo compie il trasferimento di un'ingente quantità di dati, risulta uno spreco effettuare la negoziazione un byte alla volta. Per questo motivo, questo compito si delega ad un controllore, detto **accesso diretto alla memoria (DMA)**. La CPU scrive in memoria il comando per il DMA, che contiene un puntatore per la locazione dei dati e il numero di byte da trasferire. Il DMA agisce direttamente sul bus ed esegue il trasferimento senza la CPU. La negoziazione tra DMA e dispositivo avviene attraverso una coppia di fili. Il controllore del dispositivo invia un segnale ad un filo, detto **DMA-request**, dicendo che è disponibile per il trasferimento. Il controllore del DMA accede al bus di memoria, invia un segnale all'altro filo, detto **DMA-acknowledge**. Il dispositivo riceve il segnale, effettua il trasferimento e rimuove il segnale. Dato che questo meccanismo impedisce alla CPU di accedere momentaneamente alla memoria centrale, spesso si utilizza l'**accesso diretto alla memoria virtuale (DVMA)**, che utilizza indirizzi virtuali che si tramutano poi, in indirizzi fisici.

Dopo aver descritto i concetti principali dei dispositivi di I/O, adesso dobbiamo soffermarci sulle **interfacce** di un SO che permettono un trattamento uniforme dei dispositivi di I/O. Ogni SO ha le proprie convenzioni sulle interfacce dei driver dei dispositivi. Possiamo avere diversi tipi di differenze:

- **trasferimento a flusso di caratteri o a blocchi**: il primo trasferisce un byte alla volta, mentre il secondo un blocco di byte alla volta.
- **accesso sequenziale o diretto**: il primo trasferisce i dati secondo un ordine fisso, mentre il secondo può richiedere l'accesso ad una qualunque delle possibili locazioni di memoria.
- **sincroni o asincroni**: i primi trasferiscono i dati con tempi di risposta prevedibili, mentre i secondi hanno tempo di risposta irregolari che subiscono le influenze dalle altre regioni del computer.
- **condivisibili o dedicati**: i primi possono essere usati in maniera concorrente, i secondi no.

- **velocità di funzionamento**: che varia da pochi byte al secondo, fino a qualche gigabyte al secondo.
- **lettura e scrittura, solo lettura o solo scrittura**: alcuni dispositivi possono emettere e ricevere dati, altri solo emettere e altri ancora solo ricevere.

Un'altra interfaccia molto importante, può essere quella per i **dispositivi a blocchi e a caratteri**. Essa sintetizza tutti gli aspetti necessari per accedere alle unità a disco e ad altri dispositivi basati sul trasferimento di blocchi di dati. Deve supportare le operazioni di `read()`, `write()` e `seek()`. Di solito le applicazioni comunicano con questi dispositivi tramite un'interfaccia del file system, dove i dispositivi vengono trattati come una sequenza lineare di blocchi, detto **I/O di basso livello**. I file possono essere associati alla memoria, facendo coincidere una parte dello spazio di indirizzi virtuale di un processo, con il contenuto di un file. I dispositivi a caratteri, invece, generano o accettano stream di dati, quindi possono essere tastiere, mouse, porte e supportano i comandi `get()` e `put()`. Le modalità di indirizzamento e le prestazioni di I/O sono differenti per i **dispositivi di rete**. La maggior parte dei SO fornisce un'**interfaccia di rete socket**. Le system call associate a questo meccanismo permettono di creare un socket, collegarlo in un altro punto della rete, controllare che il collegamento sia andato a buon fine e, infine, inviare o ricevere pacchetti lungo la connessione.

Molti moderni calcolatori dispongono di **timer e orologi**, che permettono di visualizzare l'ora corrente, segnalare il tempo trascorso e regolare un timer per avviare un'operazione ad un determinato tempo. Il dispositivo che misura la durata di un lasso di tempo si chiama **timer programmabile**. Si può regolare per attendere un certo tempo e poi inviare un'interruzione.

Un altro aspetto molto importante delle system call è la scelta tra **I/O bloccante e non bloccante**. Una system call bloccante sospende l'esecuzione dell'applicazione che l'ha invocata. Essa passa dalla coda dei processi pronti a quella di attesa. Alcuni processi, però, richiedono che l'I/O sia non bloccante. Una system call di questo tipo restituisce il controllo all'applicazione che l'ha invocata, fornendo un parametro che indica quanti byte sono stati trasferiti. Un'alternativa alle system call non bloccante sono quelle **asincrone**, in cui l'applicazione continua ad essere eseguita, completando le sue operazioni di I/O.

Il sottosistema I/O del kernel fornisce diversi servizi realizzati a partire dai dispositivi e dai relativi driver. Ne vediamo alcuni:

- **scheduling**: significa stabilire un ordine di esecuzione efficace per le richieste di I/O. Di solito, esso viene realizzato mantenendo una **coda di richieste**, anche se, alcuni sistemi che richiedono di avere più richieste contemporaneamente, possono anche implementare una **tavella dello stato dei dispositivi**, in cui viene indicato il tipo, l'indirizzo e lo stato del dispositivo.
- **gestione dei buffer**: un buffer è un'area di memoria che contiene dati mentre essi sono trasferiti tra due dispositivi. I buffer sono molto utili e vengono utilizzati per molteplici motivi. Per esempio, si potrebbe avere la necessità di sincronizzare dispositivi con diverse velocità di trasferimento. Oppure, sarebbe necessario gestire i dispositivi che trasferiscono dei blocchi di dati in blocchi di dimensione diversa, come uno scambio di messaggi. E, infine, si potrebbe realizzare la cosiddetta **semantica delle copie**, che garantisce che la versione dei dati scritta sul disco sia conforme a quella al momento della system call.
- **cache**: la cache è una memoria ad alta velocità utilizzata dai dispositivi di I/O per rendere più efficace il trasferimento dei dati. Sembra essere simile al buffering, ma i due meccanismi differiscono del fatto che un buffer non contiene dati di cui ne esiste un'altra copia, mentre la cache contiene copie già memorizzate.
- **code di spooling e riservazione dei dispositivi**: lo spooling, o accodamento, si riferisce allo spostamento dei dati in un buffer in attesa di essere smistati verso il dispositivo o l'applicazione che li deve elaborare. Il buffer rappresenta una specie di stazione di attesa, dove i dati rimangono in attesa che il dispositivo più lento porti a termine il proprio compito. Quindi, lo spooling viene usato per ottenere un uso esclusivo dei dispositivi. In alcuni SO, lo spooling è gestito da un processo di sistema specializzato, detto **demone**.
- **gestione degli errori**: il SO deve poter proteggersi dal malfunzionamento dei dispositivi. Possiamo avere **errori transitori**, tipo una rete sovraccarica, o **errori permanenti**, come un disco rotto. Con la prima situazione, il SO può tentare di recuperare la situazione, mentre con la seconda, tutto diventa più difficile e, quasi sempre, irrecuperabile. Alcuni tipi di hardware forniscono dei protocolli per il **rilevamento dell'errore**.
- **strutture dati del kernel**: il SO utilizza particolari strutture dati interne al kernel per mantenere informazioni riguardo lo stato delle componenti coinvolte nelle operazioni di I/O. Spesso, i moderni SO, migliorano le prestazioni di queste strutture

dati ricorrendo alla **programmazione ad oggetti**, cioè aggregando sia metodi che dati.

In conclusione, arriviamo a chiederci una cosa. Come si può giungere dal nome di un file ad un controllore di un dispositivo. Ne abbiamo un esempio lampante in **MS-DOS**, dove la prima parte del nome di un file identifica in modo univoco la periferica (ad es. —> **C:** è la parte iniziale dei file all'interno dell'unità disco principale, a cui è associato un indirizzo di porta per mezzo della tabella dei dispositivi). In **UNIX**, però, non c'è una vera e propria separazione tra il dispositivo e il nome del file. Così, si impiega una **tabella di montaggio** per associare le due cose. Il nome del dispositivo è un oggetto del file system ed è visto come una coppia di numeri del tipo **<principale, secondario>**, dove il primo è il driver del dispositivo, mentre il secondo individua l'indirizzo della porta. L'I/O è un fattore predominante nelle **prestazioni** del sistema. Richiede un notevole consumo di CPU, continui cambi di contesto, spazio di memoria e banda di rete. Per evitare questi problemi, è possibile adottare diverse strategie:

- si riduce il numero di cambi di contesto.
- si riduce il numero di copiature dei dati durante i trasferimenti.
- si riduce la frequenza delle interruzioni, preferendo trasferimenti di grandi quantità di dati.
- si aumenta il tasso di concorrenza usando controllori DMA o bus dedicati.
- si implementano primitive in hardware.
- si equilibrano le prestazioni di CPU



# INTERFACCIA DEL FILE SYSTEM

I calcolatori possono memorizzare le informazioni in maniera diversa ma, per offrire una visione logica e uniforme delle informazioni memorizzate, esso fornisce un'astrazione delle caratteristiche fisiche dei propri dispositivi di memoria, attraverso l'unità principale di memorizzazione logica, il **file**. Un file è un insieme di informazioni correlate, registrate in memoria secondaria, a cui è stato assegnato un nome. I file possono essere numerici, alfabetici, alfanumerici o binari. Possono essere in un formato specifico oppure rigidamente formattati. Un file ha una **struttura** definita sul tipo: per esempio, un file di testo è formato da una sequenza di caratteri organizzati in righe o, eventualmente, pagine; un file sorgente è una sequenza di funzioni, mentre un file eseguibile contiene una serie di istruzioni che possono essere caricate in memoria ed eseguite.

Un file ha degli attributi:

- **nome**: l'unica informazione umanamente leggibile ed è prettamente simbolico
- **identificatore**: un'etichetta univoca, di solito un numero, che identifica il file all'interno del sistema.
- **tipo**: necessaria ai sistemi che gestiscono diversi tipi di file.
- **locazione**: un puntatore al dispositivo e alla locazione dei file in tale dispositivo.
- **dimensione**: la dimensione corrente del file e, eventualmente, la dimensione massima consentita.
- **protezione**: informazioni sul controllo degli accessi in lettura e scrittura.
- **ora, data, identificazione dell'utente**: informazioni relative la creazione, la modifica e l'ultimo uso

Alcuni SO supportano anche **attributi estesi**, come funzioni di sicurezza o di codifica.

Un file è un ADT, cioè un tipo di dato astratto. Per definirlo, quindi, è necessario citare le **operazioni** che possono essere effettuate su di esso. Ne vediamo alcune:

- **creazione di un file**: in primis, bisogna trovare lo spazio nel file system e poi si deve creare un nuovo elemento nella directory.
- **scrittura di un file**: viene effettuata una system call, che specifica il nome del file e le informazioni di scrittura. Inoltre, il file system deve sempre mantenere un puntatore di scrittura, che punta alla locazione in cui avverrà la scrittura successiva.
- **lettura di un file**: anche qui viene effettuata una system call, che specifica il nome e la posizione del file. Il sistema cerca la sua posizione nella directory, mantenendo il solito puntatore di lettura per la lettura successiva.
- **riposizionamento di un file**: si cerca l'elemento appropriato nella directory e si assegna un nuovo valore al puntatore alla posizione corrente del file.
- **cancellazione di un file**: si cerca l'elemento nella directory, si rilascia lo spazio occupato dal file e si rimuove l'elemento dalla directory.
- **troncamento di un file**: si potrebbe voler cancellare il contenuto di un file, mantenendo, tuttavia, i suoi attributi. Questa operazione rilascia lo spazio occupato, senza però cancellare gli attributi.

Esistono, ovviamente altre operazioni (aggiunta, ridenominazione) e, inoltre, queste operazioni possono essere anche combinate per crearne di nuove più complesse (una copia è la creazione di un nuovo file, la lettura del file vecchio e la scrittura nel file nuovo).

Il SO mantiene una **tabella dei file aperti**. Quando si richiede un'operazione su quel file, questo viene individuato tramite un indice e, quando il file non è più attivo, esso viene chiuso. Il tutto viene gestito da apposite system call, di solito `open()` e `close()`. A ciascun file aperto sono associate diverse informazioni:

- **puntatore al file**: ultima posizione di lettura o scrittura sotto forma di un puntatore alla locazione del file.
- **contatore dei file aperti**: numero di `open()` e `close()`. Raggiunto il valore 0, il file viene rimosso dalla tabella.
- **posizione nel disco del file**: informazioni per localizzare il file nel disco.
- **diritti di accesso**: ogni processo apre un file secondo una modalità di accesso e questa informazione è contenuta all'interno della tabella. Spesso, in questo caso, i processi utilizzano dei lock specifici, sia per la lettura che la scrittura, che però richiedono anche l'implementazione di meccanismi di sincronizzazione.

È necessario che il SO sappia riconoscere il tipo di file per poterlo trattare in maniera coerente. Di solito, un file contiene sempre la doppia nome ed estensione, entrambe separate da un punto. Non c'è bisogno di ricordarle tutte, ma ne vediamo adesso qualcuna.

TIPO DI FILE	ESTENSIONE	FUNZIONE
Eseguibile	<code>exe</code> , <code>com</code> , <code>bin</code>	Programma eseguibile in linguaggio macchina
Oggetto	<code>obj</code> , <code>o</code>	Compilato in linguaggio macchina, ma non linkato
Codice sorgente	<code>c</code> , <code>cc</code> , <code>java</code> , <code>perl</code> , <code>asm</code>	File compilabili e scritti nei vari linguaggi di programmazione
Batch	<code>bat</code> , <code>sh</code>	Comandi per l'interprete dei comandi
Markup	<code>xml</code> , <code>html</code> , <code>tex</code>	Dati testuali o documenti
Word Processor	<code>xml</code> , <code>rtf</code> , <code>docx</code>	Vari formati per processare file di testo
Libreria	<code>lib</code> , <code>a</code> , <code>so</code> , <code>dll</code>	Librerie di procedure per la programmazione
Stampa o visualizzazione	<code>gif</code> , <code>pdf</code> , <code>jpg</code>	File ASCII per la stampa o la visualizzazione
Archivio	<code>rar</code> , <code>zip</code> , <code>tar</code>	File contenenti più file tra loro collegati, talvolta compressi per favorirne l'archiviazione
Multimediali	<code>mpeg</code> , <code>mov</code> , <code>mp3</code> , <code>mp4</code> , <code>avi</code>	File binari contenenti informazioni audiovisive

Abbiamo compreso che i file, per natura, devono memorizzare delle informazioni. Per usarle, è necessario accedere ad esse e trasferirle in memoria. Esistono due tipologie di accesso:

- **accesso sequenziale**: si tratta del metodo più comune, dove le informazioni vengono elaborate in maniera ordinata, un record dopo l'altro. In genere, però, vengono effettuate soltanto operazioni di lettura e scrittura.
- **accesso diretto**: il file viene scomposto in elementi logici, detti **record**, di lunghezza fissa. Il file è visto come una sequenza numerata di record, con questi ultimi che si possono leggere e scrivere in maniera arbitraria senza un ordine preciso e in maniera veloce. I numeri dei record sono **relativi**, cioè permettono al SO di decidere dove posizionare il file, aiutando l'utente a non accedere a regioni del file system che non gli competono.

Esistono altre metodologie di accesso e, molte di queste, indicano la costruzione di un **indice** per il file. L'indice contiene puntatori ai vari record. Per trovare un elemento del file bisogna prima cercare nell'indice e, quindi, usare lui stesso per accedere direttamente al file e trovare l'elemento. Si tratta di un metodo che permette di velocizzare le ricerche in file molto grandi

Soltamente, i file vengono raggruppati in base alle loro caratteristiche o secondo dei criteri di comodità che vengono stabiliti dagli utenti. Le **directory** consentono di ordinare il contenuto del disco secondo le necessità. Le directory permettono di:

- **ricerca di un file**: si può scorrere una directory per individuare l'elemento associato ad un particolare file.
- **creazione di un file**: i nuovi file creati devono poter essere aggiunti alla directory.
- **cancellazione di un file**: i file eliminati devono poter essere eliminati dalla directory.
- **elenco di una directory**: deve esserci la possibilità di elencare tutti gli elementi di una directory, ognuno associato al file corrispettivo.
- **ridenominazione di un file**: il nome del file, che rappresenta il suo contenuto, deve poter essere modificato se quest'ultimo cambia.
- **attraversamento del file system**: si deve poter creare delle copie di backup che sarebbero utili in caso di guasto del sistema.

Descriviamo, adesso, i principali metodi di organizzazione delle directory:

- **directory ad un livello**: tutti i file sono gestiti all'interno della stessa directory, che è facilmente gestibile e comprensibile. Si tratta di un metodo che però risulta inefficiente se ci sono molti file al suo interno oppure se più utenti accedono ad essa.
- **directory a due livelli**: ogni utente dispone della propria directory, detta **UFD**, e tutte hanno una struttura simile. Quando c'è bisogno di effettuare una delle operazioni descritte prima, per esempio, l'UFD si rifà alla directory principale del sistema, detta **MFD**, che contiene un puntatore alla rispettiva UFD. In questo modo, più utenti possono interagire con il sistema. Per quanto possa superare gli svantaggi della struttura precedentemente descritta, questo tipo di directory non permette la cooperazione tra gli utenti.

- **directory con struttura ad albero**: esiste una **directory radice** e ogni file ha un unico nome di percorso. Una **sottodirectory** può contenere un file o altre sottodirectory. Ogni utente dispone di una **directory corrente**, che contiene la maggior parte dei file di interesse al processo corrente. Un nome di percorso può essere **assoluto**, che comincia dalla radice, o **relativo**, che comincia dalla posizione corrente. In questo modo, si risolve il problema della cooperazione e più utenti possono accedere ai file di altri utenti.
- **directory con struttura a grafo aciclico**: uno dei problemi della struttura ad albero, è che non permette la condivisione di directory e file. Un grafo aciclico, cioè senza cicli, permette che lo stesso file o la stessa sottodirectory possano essere in due directory diverse. Questo significa che esistono più copie dello stesso file. La condivisione viene effettuata tramite un collegamento, detto **link**, che è un puntatore ad un altro file o ad un'altra sottodirectory. Si tratta di una struttura estremamente complessa da realizzare, oltre che difficile da rendere efficiente, poiché molte operazioni, come la cancellazione, risulterebbe lunghe dato che bisogna cancellare anche tutte le copie e i puntatori.
- **directory con struttura a grafo generale**: il problema principale della struttura aciclica è proprio il fatto che devono esserci cicli. Ma, progettare cattivi algoritmi di ricerca, potrebbe far incappare l'utente in dei cicli infiniti, che non devono esserci. La presenza di un contatore dei riferimenti potrebbe risolvere in parte questo problema, dato che, se è zero, allora il file viene cancellato. Ma, con un loop infinito, il contatore potrebbe non valere mai zero. Per questo, si tende ad utilizzare un **garbage collector**, cioè un meccanismo di ripulitura per stabilire quando l'ultimo riferimento è stato cancellato.

Le informazioni di un calcolatore devono essere protette dai danni fisici, quindi devono essere affidabili, e dagli accessi improprio, quindi devono essere protette. L'affidabilità si garantisce con la copia dei file, mentre per quanto concerne la **protezione**, possiamo ottenerla in molti modi. Ciò che serve è un **accesso controllato**, cioè si limitano i possibili tentativi di accesso che potrebbero andare a leggere, scrivere, eseguire o cancellare un determinato file. Lo schema più comune risulta essere una **lista di controllo degli accessi (ACL)** che viene assegnata ad ogni file o directory. Qui sono specificati i vari utenti e i tipi di accesso consentiti. Si tratta di un meccanismo, tuttavia, poco efficiente, perché liste troppo lunghe andrebbero a rallentare le operazioni di

accesso. Per risolvere questo problema, si tende a raggruppare gli utenti in determinate **classi di accesso**:

- **proprietario**: l'utente che ha creato il file.
- **gruppo**: un insieme di utenti che condividono il file e hanno bisogno di simili tipologie di accesso.
- **universo**: tutti gli altri utenti del sistema.

In generale, per un sistema più efficiente, si può utilizzare anche la combinazione dei due metodi, cioè la suddivisione in classi di accesso ma, in caso si desiderasse maggiore selettività, è possibile anche implementare una ACL.

In alternativa, possiamo anche utilizzare una **password** di protezione a ciascun file. Per quanto questo sistema risulti sicuro, ha numerosi svantaggi. Nel caso si scegliesse di cambiare regolarmente le password, si arriverebbe ad un numero di parole d'ordine troppo elevato e difficile da ricordare per gli utenti. Se, invece, si scegliesse una sola password, la sua scoperta renderebbe il file accessibile a tutti, per sempre.



# REALIZZAZIONE DEL FILE SYSTEM

I dischi costituiscono la maggior parte dei dispositivi di memorizzazione per il file system. In generale essi vengono usati perché si possono riscrivere localmente ed è possibile accedere direttamente ad ogni blocco di informazioni. Per fornire un efficiente e conveniente accesso al disco, il sistema operativo può usare uno o più file system. Il file system ha una **struttura stratificata**, in cui possiamo distinguere più livelli:

1. Il livello più basso è il **controllo dell'I/O**, costituito dai driver dei dispositivi, i gestori delle interruzioni e si occupa del trasferimento tra memoria centrale e memoria secondaria.
2. Il livello successivo è il **file system di base**, che invia generici comandi all'appropriato driver di dispositivo, per leggere e scrivere blocchi fisici sul disco. Ogni blocco è identificato dal suo indirizzo numerico.
3. Il livello successivo è il **modulo di organizzazione dei file**, che è a conoscenza dei file e dei loro blocchi logici e fisici. Conoscendo il tipo di allocazione, il modulo può tradurre gli indirizzi logici negli indirizzi fisici. Inoltre è composto anche dal gestore dello spazio libero, che registra i blocchi non allocati e li mette a disposizione.
4. L'ultimo livello è il **file system logico**, che gestisce i metadati, cioè tutte le altre strutture del file, eccetto i dati. Inoltre mette a disposizione la struttura di una directory per l'organizzazione dei file. Inoltre, è anche responsabile di protezione e sicurezza.

Per realizzare un file system è necessario usare numerose strutture dati. Le distinguiamo in base alla loro locazione. All'interno del disco potremo trovare:

- **blocco di controllo dell'avviamento**: contiene le informazioni necessarie per l'avviamento del SO partendo da quella partizione.
- **blocco di controllo del volume**: contiene dettagli riguardanti quella specifica partizione, come il numero e la dimensione dei blocchi, il contatore dei blocchi liberi.

- **struttura delle directory**: usate per l'organizzazione dei file.
- **blocco di controllo del file**: contiene dettagli relativi al file specifico, come un identificatore, il proprietario, le dimensioni e la locazione.

Invece, all'interno della memoria centrale potremo trovare:

- **tabella di montaggio**: contiene informazioni riguardo ogni partizione montata.
- **cache della struttura della directory**: contiene le informazioni relative a tutte le directory i cui processi hanno avuto accesso di recente.
- **tabella di sistema per i file aperti**: una copia del blocco di controllo di ogni file aperto.
- **tabella dei file aperti per ciascun processo**: contiene un puntatore alla tabella generale.
- **buffer**: che conservano blocchi del file system durante la lettura o la scrittura.

Una volta creato un file, per essere usato, deve essere aperto. Usando la system call `open()`, si passa il nome del file al file system e vengono esaminati tutti i file aperti all'interno della tabella. Se il file non è presente, allora si crea un nuovo elemento nella tabella, con un puntatore alla tabella generale. Il nome dato all'interno della tabella è detto **file descriptor**.

Selezionare algoritmi di gestione delle directory e di allocazione della memoria è un compito a cui bisogna adempiere per realizzare un buon file system. Vediamo alcuni metodi per la gestione delle directory:

- **lista lineare**: questa lista contiene i nomi dei file con i puntatori ai blocchi di dati. Si tratta di un metodo molto semplice, ma la sua esecuzione è molto onerosa in termini di tempo e gli accessi risultano molto lenti. Durante la creazione si crea un file, controllando che non ce ne sia uno uguale nella directory, mentre per cancellarlo, si cerca un file con quel nome nella directory.
- **tabella hash**: si usa sempre una lista lineare, però accoppiata con una struttura dati hash. Vengono ridotti i tempi di esecuzione e di accesso ma bisogna comunque provvedere a gestire le collisioni, cosa di cui abbiamo già parlato. L'unico svantaggio di questo metodo è la dimensione, che di solito è fissa.

Vediamo adesso alcuni metodi di allocazione, che servono a garantire efficienza durante il riempimento del disco:

- **allocazione contigua:** ogni file viene memorizzato come un blocco contiguo sul disco. Per reperire il file occorre sapere solo la locazione iniziale e la lunghezza. L'accesso è relativamente semplice, ma ci sono alcuni svantaggi importanti. Uno di questi può essere l'**allocazione dinamica della memoria**, cioè la scelta ottimale di un buco libero che, come abbiamo visto, qualsiasi metodo non è esattamente il migliore. Inoltre, questo tipo di allocazione soffre di **frammentazione esterna**, cioè quando lo spazio libero si frammenta in vari blocchi liberi e, il più grande di questi, non è in grado di contenere abbastanza dati. Per risolvere il problema di questa frammentazione, come abbiamo visto, si utilizza la **compattazione**, che può essere eseguita **off-line**, cioè quando il file system non è operativo, oppure **on-line**. Generalmente, per diminuire i lunghi tempi della compattazione, è preferibile eseguirla off-line. Un modo per evitare questi problemi, può essere l'aggiunta di un'**estensione**, cioè vengono allocate porzioni di spazio contiguo e, se non bastano, se estende un'altra porzione.
- **allocazione concatenata:** vengono risolti i problemi dell'allocazione contigua. Ogni file è costituito da una lista concatenata di blocchi del disco, i quali possono essere sparsi in qualsiasi punto del disco stesso. La directory contiene un puntatore al primo e all'ultimo blocco del file. Qui non esiste la frammentazione esterna, poiché per soddisfare la richiesta si può usare qualsiasi blocco libero della lista. Questo metodo presenta alcuni svantaggi, per esempio può essere usato soltanto per i file che richiedono un accesso sequenziale. Inoltre, a causa dei puntatori, ogni file richiede un po' più dello spazio effettivo. Per questo, è preferibile usare i cluster, cioè gruppi di blocchi. Una variante di questo metodo risulta essere la **tabella di assegnazione del file**, detta **FAT**. La FAT ha un elemento per ogni blocco del disco ed è indicizzata dal numero di blocco.
- **allocazione indicizzata:** serve per risolvere i problemi relativi all'accesso diretto dell'allocazione concatenata. Tutti i puntatori vengono raggruppati in una singola locazione, detto **blocco indice**. Per accedere ad un file, si carica in memoria il suo blocco indice e si utilizzano i puntatori in esso contenuti. Così, l'intero blocco diventa disponibile per contenere dati. Qui, però, arriva un dilemma. Per far sì che ogni file contenga un blocco indice, esso deve essere quanto più piccolo possibile. Ma, se piccolo, determina anche una dimensione massima relativamente bassa per il file. Anche qui, possiamo adottare delle contromisure. Il primo meccanismo può essere uno **schema concatenato**, cioè si collegano insieme tutti i blocchi. Oppure, potremo adoperare un **indice a più livelli**, cioè un blocco indice di primo livello

punta ad un blocco indice di secondo livello che, a loro volta, puntano ai blocchi dei file. L'ultima strategia può essere uno **schema combinato**. Ad ogni file è associato un inode, che contiene 15 puntatori. I primi 12 sono puntatori a blocchi diretti, gli altri 3 a blocchi indiretti. Il primo puntatore va ad un blocco indiretto singolo, il secondo ad uno doppio. Il terzo ad uno triplo.

Ovviamente, la quantità di spazio nei dischi è limitata e per questo è necessario riutilizzare lo spazio lasciato dai file cancellati per scrivere nuovi file. Per far ciò, il sistema si serve di una **lista dello spazio libero**. Essa può essere realizzata in diversi modi e, ora, ne vediamo alcuni:

- **vettore di bit**: ogni blocco è rappresentato da un bit. Se il blocco è libero, allora vale 1, se è assegnato, allora vale 0. Questo metodo ha un vantaggio fondamentale, che è la semplicità e l'efficienza nel trovare il primo blocco libero. Tutta via, questi vettori non risulterebbero altrettanto efficienti se non fossero mantenuti in memoria centrale. E questo è possibile solo se i dischi sono di piccole dimensioni.
- **lista concatenata**: tutti gli spazi liberi vengono collegati, tenendo un puntatore al primo di questi in una speciale locazione del disco e caricandolo in memoria. Si tratta di un metodo non efficiente, perché, per trovare uno spazio libero, bisognerebbe attraversare la lista, leggendo ogni blocco e aumentando i tempi di risposta.
- **raggruppamento**: si memorizzano gli indirizzi di n blocchi liberi nel primo di questi.
- **conteggio**: più blocchi contigui si possono allocare e liberare contemporaneamente, soprattutto quando si utilizzano algoritmi di allocazione contigua o cluster. Quindi, si tende a tenere solo l'indirizzo del primo blocco libero.

Dopo aver descritto le opzioni di allocazione e di gestione delle directory, è necessario soffermarsi per un attimo sull'effetto che questi algoritmi sortiscono sulle **prestazioni** e **l'efficienza** dei dischi che, a loro volta, tendono ad essere il principale collo di bottiglia per le prestazioni dell'intero sistema. L'uso efficiente di un disco dipende dai suoi algoritmi di allocazione e gestione delle directory. Bisogna anche tener conto del tipo di dati che normalmente vengono contenuti al loro interno. Infatti, ogni informazione associata al file finirà per influire sulle prestazioni e sull'efficienza. Anche dopo aver scelto questi algoritmi, le prestazioni possono essere migliorate in maniera differente. Per esempio, molti sistemi utilizzano la cosiddetta **buffer cache**, una sezione separata

di memoria che deve contenere blocchi liberi, in previsione di un loro imminente utilizzo. Altri, invece, utilizzano una **cache delle pagine**, che impiega tecniche di memoria virtuale per gestire i file come pagine anziché come blocchi, aumentando così, l'efficienza. Altri sistemi, invece, utilizzano il **double caching**, cioè una combinazione delle due tecniche di caching. Ovviamente, il doppio passaggio di cache risulta uno spreco di tempo e risorse, oltre che di inconsistenze dei dati. Per questo, i sistemi tendono ad utilizzare una buffer cache unificata, che eviti il double caching e che permetta alla memoria virtuale di gestire i dati del file system. Un altro aspetto che può influenzare le prestazioni può essere l'ottimizzazione degli accessi sequenziali. Lo possiamo fare con:

- **rilascio all'indietro (free-behind)**: rimuove una pagina del buffer non appena si verifica una richiesta alla pagina successiva. Le pagine precedenti, però, non verranno più usate, creando problemi di spazio all'interno del buffer.
- **lettura anticipata (read-ahead)**: si leggono e si mettono nella cache la pagina richiesta ed alcune pagine successive, che probabilmente verranno richieste una volta terminata l'elaborazione della pagina corrente. Questo è il meccanismo più vantaggioso.

File e directory vengono mantenuti sia in memoria centrale che sul disco, quindi è necessario che, in caso di malfunzionamenti, non si verifichino incoerenti tra i dati. Quale che sia l'errore, il file system deve individuarlo e correggerlo. Il **verificatore della coerenza** confronta i dati delle directory con i blocchi dati dei dischi, tentando di correggere l'incoerenza. È necessario prevedere guasti anche ai dischi magnetici, per evitare di perdere definitivamente i dati. A questo scopo, è possibile utilizzare le classiche **copie di riserva (backup)**, posizionate su altri dispositivi di memorizzazione. Il ripristino di una situazione antecedente alla perdita dei dati è detto **recupero (restore)**. La gestione delle copie di riserva dipende dalla criticità dei dati. Potremo avere **copie complete**, che richiedono tempo ma permettono un recupero veloce da perdite totali, e **copie incremental**ali, che durano di meno, ma permettono ripristini solo da situazioni di perdita parziale.



# LABORATORIO

Per utilizzare i servizi del SO, il programmatore ha a disposizione una serie di entry-points per il kernel, chiamate **system call**. Nei sistemi UNIX, che noi approfondiremo, ce ne sono quasi duecento e ogni system call corrisponde ad una **funzione di libreria C**. Il programmatore chiama la funzione utilizzando i solito standard e la funzione invoca l'opportuno servizio del sistema operativo. Dal punto di vista del programmatore, non c'è alcuna differenza tra funzioni di libreria e system call. Dal punto di vista, invece, del realizzatore del SO, la differenza è sostanziale. È possibile sostituire le funzioni di libreria, implementandone altre che facciano la stessa cosa. Al contrario, una system call non può essere sostituita, dato che dipende fortemente dal SO. I concetti base, che vanno seguiti durante la realizzazione dei SO, sono definiti dallo standard **POSIX**, acronimo di *Portable Operating System Interface*.

Alcune delle librerie più usate possono essere:

- `fcntl.h`: per il controllo dei file.
- `unistd.h`: che consente l'accesso alle API di POSIX, fornendo funzioni di libreria per alcune system call basilari.
- `sys/types.h`: include definizioni per tipi di dati primitivi.
- `sys/wait.h`: per il controllo dei processi.

Passiamo, ora, al controllo dei processi. C'è da sapere che ogni processo, all'interno del sistema, ha un **identificatore univoco**, che è un intero positivo, detto **pid** (*process identifier*). Con la SC `getpid(void)`, ci viene restituito il pid del processo chiamante, mentre con `getppid(void)`, il pid del padre del processo chiamante. L'unico modo per creare nuovi processi è una SC chiamata `fork()`, da parte di un processo già esistente. Quando la fork viene chiamata, viene generato un processo figlio. La `fork()` ha due valori di ritorno:

- il valore restituito al processo figlio è `0`.
- il valore restituito al padre è il pid del figlio.
- il valore restituito, in caso di errore, è `-1`.

Un processo può avere più figli e, dato che non esistono SC per recuperare il pid di tutti i suoi figli, occorrerà utilizzare lo stratagemma del salvataggio in una variabile intera. Padre e figlio continueranno ad eseguire **concorrentemente** il programma, a partire dall'istruzione immediatamente successiva alla fork, cioè un assegnamento.

Il figlio sarà una **copia** del padre e condividerà dati, stack e l'heap. Il kernel protegge questi dati settando alcuni permessi. Se un processo figlio modifica una di queste strutture dati, essa viene fisicamente copiata e duplicata (**COW**, *Copy-on-Write*).

Naturalmente, padre e figlio saranno messi nella ready queue e sarà l'**algoritmo di schedulazione** che deciderà se eseguire prima il figlio o prima il padre.

Quando un processo termina, il kernel invia al padre un **segnalet**, detto `SIGCHLD`, che poi approfondiremo. Il padre può ignorare questo segnale, per default, oppure è possibile intercettarlo con un **signal handler**. In ogni caso, il padre può chiedere informazioni su qual è lo stato di terminazione del figlio. Per farlo esistono due SC specifiche:

- `pid_t wait(int *status)`
- `pid_t waitpid(pid_t pid, int *status, int options)`

Entrambe fanno sostanzialmente la stessa cosa, cioè attendono la terminazione di un processo. La funzione `wait()` sospende il processo padre, finché uno dei figli non termina oppure è stato ricevuto un segnale di terminazione. Quando un figlio termina, senza che suo padre lo abbia atteso attraverso la `wait()`, allora il figlio diventa uno **zombie**, come si dice in gergo.

La funzione `waitpid()` sospende il processo padre finché il figlio corrispondente al parametro passato per argomento termina oppure non è stato ricevuto un segnale di terminazione. Il valore del pid può essere uno dei seguenti:

- `pid == -1`: aspetta per un qualsiasi processo figlio, quindi sostanzialmente una `wait()`.
- `pid > 0`: aspetta il figlio che ha il pid uguale al valore passato.
- `pid == 0`: aspetta un qualsiasi figlio che ha il process group id uguale a quello del processo che ha chiamato la `waitpid()`.
- `pid < -1`: aspetta un qualsiasi figlio che ha un process group id uguale al valore assoluto di pid.

Se i **process group id**, che è sostanzialmente una collezione di processi, non esiste, allora la `waitpid()` restituirà un messaggio di errore. Dentro la variabile `status` viene

salvato sempre lo stato di terminazione, come nella `wait()`, mentre `options` serve ad aggiungere informazioni. Il parametro `options` potrà valere `0`, che significa nessuna opzione, oppure sarà uguale a `WNOHANG`, cioè `waitpid()` bloccherà il chiamante se il figlio specificato da pid non è immediatamente disponibile, ma restituirà `0`.

In generale, con la funzione `wait()` il processo padre si blocca in attesa e ritorna immediatamente lo stato del figlio e, nel caso, errore. Invece, `waitpid()`, può scegliere quale figlio aspettare. `wait()` può bloccare il processo chiamante, mentre in `waitpid()` può essere specificata l'opzione di non farlo bloccare e farlo ritornare immediatamente. Quindi `waitpid()` costituisce una versione **non bloccante** di `wait()`.

Sappiamo che la `fork()` è usata per creare un nuovo processo. Il processo figlio esegue lo stesso programma del padre. A volte, però, potrebbe voler eseguire un programma diverso e questo può essere fatto chiamando una SC di `exec`. Dopo la chiamata ad una `exec`, lo spazio indirizzi del figlio sarà rimpiazzato dal nuovo programma, non cambiando però il pid del figlio. La `exec`, in realtà, è una famiglia di SC e ne prevede sei differenti, che sono scritte in maniera sintatticamente diversa, però facendo tutte la stessa cosa, cioè eseguono un nuovo programma. Ogni `exec` restituisce -1 in caso di errore e, se tutto va bene, non restituiscono nulla. Vediamo ora, le varie SC:

- `int execl(const char *pathname, const char *arg, ..., (char *) NULL);`
- `int execle(const char *pathname, const char *arg, ..., (char *) NULL, char *const envp[]);`
- `int execlp(const char *file, const char *arg, ..., (char *) NULL);`
- `int execlepe(const char *path, const char *arg0, ..., const char *const *envp);`
- `int execv(const char *pathname, char *const argv[]);`
- `int execve(const char *pathname, char *const argv[], char *const envp[]);`
- `int execvp(const char *file, char *const argv[]);`
- `int execvpe(const char *file, char *const argv[], char *const envp[]);`

Le lettere finali delle sei funzioni aiutano a ricordare gli argomenti delle stesse. La lettera `l` indica che la funzione avrà una lista di argomenti. Mentre la lettera `v` indica un vettore `argv[]`, cioè un array di puntatori a caratteri agli argomenti sulla linea di comando. La lettera `p` indica che la funzione prende come primo argomento un nome di file e utilizza la variabile di ambiente `PATH` per cercare la posizione nel file system. Una volta trovato il file, lo si esegue. Al contrario, sarà indicato il pathname completo del

file. La lettera `e` indica che la funzione prende un array di variabili ambiente, invece di usare le variabili ambiente vere e proprie.

Un processo può terminare:

- **involontariamente**, cioè quando tenta di effettuare azioni illegali (invadere uno spazio di indirizzi), oppure può essere terminato esternamente tramite un segnale.
- **volontariamente**, cioè con la SC `exit()` oppure con l'esecuzione dell'ultima istruzione.

La `exit()` ha come argomento un intero, detto `status`, mediante il quale il SO comunica al padre del processo che termina alcune informazioni sullo stato di terminazione.

Inoltre, vengono chiusi tutti i descrittori di file del processo, deallocherà l'heap e lo stack e, infine, terminerà l'esecuzione. Si tratta sempre di una chiamata senza ritorno. Ma cosa succede quando un processo termina? Se un figlio termina prima del padre, allora il padre può ottenere lo stato di terminazione tramite una delle SC che abbiamo visto prima. Se un padre termina prima del figlio, allora il processo `init` va ad adottare il figlio che ha perso il padre. Se il figlio termina prima del padre, ma il padre non riesce a recuperare lo stato di terminazione, il processo diventa uno zombie, cioè senza codice o dati allocati, ma esiste ancora e continua ad avere un PCB, in modo che il padre possa recuperare il suo stato di terminazione.

L'ultima SC che vedremo sarà la `system()`. Serve ad eseguire un comando shell all'interno di un programma. Spesso non viene vista come SC, ma può essere implementata tramite `fork()` o `exec()`.

Passiamo, adesso, ad un altro argomento, introducendo i **segnali**. Un segnale è un interrupt software, che permette di gestire **eventi asincroni** (come ad esempio un CTRL^C da tastiera). Un segnale può essere generato da qualsiasi processo, in qualsiasi istante. Ogni segnale ha un nome che comincia con **SIG**, a cui viene associata una costante intera, definita nella libreria `<signal.h>`. I segnali non contengono altre informazioni aggiuntive. Un segnale è un evento asincrono. In parole povere, può arrivare in qualsiasi momento e non richiede necessariamente che sia fatta qualcosa, perché vengono eseguite delle azioni di default:

- **ignorare il segnale**. Tranne `SIGKILL` e `SIGSTOP`, tutti gli altri segnali possono essere ignorati. Quei due perché sono l'unico modo che ha l'utente per interrompere momentaneamente un programma per una qualsiasi ragione.
- **catturare il segnale**. Si associa al segnale l'esecuzione di una funzione utente, che esegue una determinata azione.

- **aspettare l'azione di default.** Di solito è la terminazione del processo stesso.

Vediamo, adesso, i vari tipi di segnali:

- **SIGABRT**: generato da una SC di tipo `abort()`, quando un processo termina anormalmente.
- **SIGCHLD**: ogni volta che un processo termina o viene fermato, questo segnale viene inviato al padre. Il padre lo ignora per default oppure ottiene il pid e lo stato di terminazione tramite le SC opportune.
- **SIGCONT**: inviato ad un processo sospeso per far riprendere l'esecuzione.
- **SIGFPE**: acronimo di *Floating Point Exception* che viene inviato quando si verifica una particolare eccezione, come la divisione per zero.
- **SIGILL**: acronimo di *Illegal Instruction* che viene inviato quando l'hardware individua qualche istruzione illegale che non può essere eseguite.
- **SIGINT**: inviato quando si preme la combinazione di tasti *CTRL^C*.
- **SIGALARM**: generato alla chiamata della SC `alarm()`, che crea una specie di sveglia con un tempo preimpostato.
- **SIGQUIT**: inviato quando si preme la combinazione di tasti *CTRL^/*, generando un core file, cioè un'immagine in memoria del processo.
- **SIGKILL**: termina il processo che lo riceve.
- **SIGSEGV**: acronimo di *Segment Violation*, quando il processo fa riferimento ad un indirizzo non presente all'interno del suo spazio.
- **SIGSTOP**: ferma un processo.
- **SIGSYS**: il processo ha eseguito una SC senza fornire i parametri adeguati.
- **SIGTERM**: inviato per default dalla SC `kill()`.
- **SIGBUS**, **SIGEMT**, **SIGIOT**, **SIGTRAP**: segnali inviati per problemi hardware.
- **SIGUSR1**, **SIGUSR2**: che possono essere definiti dall'utente per la comunicazione tra processi, anche se non è il modo migliore per comunicare.

In presenza di un segnale, abbiamo detto che è possibile catturarlo attraverso la SC `signal()`. Essa restituisce la costante `SIG_ERR` in caso di errore o un puntatore al gestore del segnale se tutto è andato nella norma. Prende due argomenti: il **nome del**

**segnale** ed il **puntatore alla funzione** da eseguire all'arrivo del segnale. Il valore della funzione non deve essere per forza una funzione utente che dovrà essere eseguita in presenza del segnale, ma possiamo anche aggiungere delle costanti come `SIG_IGN`, per ignorare il segnale, o `SIG_DFL` per settare l'azione associata al suo default. Per proseguire, è necessario definire altre due SC. La prima è la `kill()`. Il suo primo argomento è il pid e il secondo è il segnale. La seconda è la `raise()` che prende solo il segnale. Entrambe inviano segnali, con la `kill()` che invia segnali ad uno o più processi. Al contrario, la `raise()` consente ad un processo di inviare un segnale a sé stesso. Oltre ad essere una SC, `kill` è anche un comando shell, che fa più o meno la stessa cosa.

Passiamo, adesso, a descrivere l'ultima parte del laboratorio, ossia i file. Un file, all'interno del SO, è identificato da un numero intero negativo, detto **file descriptor (fd)**. Un fd sarà sempre un intero compreso tra zero e la costante `OPEN_MAX`, all'interno della libreria `<limits.h>`. Esistono tre file standard:

- `STDIN_FILENO`: da dove si prendere, per default, l'input ed è normalmente connesso a mouse e tastiera. `0`
- `STDOUT_FILENO`: dove vanno i messaggi di output, di solito lo schermo. `1`
- `STDERR_FILENO`: dove finiscono i messaggi di errore. `2`

UNIX mette a disposizione alcune SC per la gestione e il controllo dei file:

- `open(const char *pathname, int flag)`: il primo argomento è il nome del file che vogliamo aprire, il secondo parametro indica una serie di opzioni. Di solito è formato dall'OR di una o più costanti simboliche. Possiamo scegliere una sola costante tra `O_RDONLY` (sola lettura), `O_WRONLY` (sola scrittura), `O_RDWR` (per lettura e scrittura). Oltre queste, possiamo aggiungerne altri, optionalmente. `O_APPEND` (tutte le scritture avverranno alla fine del file), `O_CREATE` (se il file non esiste, allora crealo, che però richiederà anche un altro parametro mode), `O_EXCL` (si usa insieme alla precedente, che restituisce errore se il file esiste già), `O_TRUNC` (se il file esiste, la sua lunghezza viene troncata a zero). L'argomento `mode` che abbiamo accennato prima serve a specificare i servizi di accesso.
- `read(int fd, void *buff, int nbytes)`: legge dal file con il file descriptor corrispondente al primo argomento un numero di bytes definiti nel terzo argomento, mettendoli all'interno del buffer corrispondente al secondo argomento. Restituisce il numero di bytes letti, se il file è vuoto restituisce `0` e, se c'è stato un errore, allora

restituisce `-1`. Il fd che prende come parametro è lo stesso che è restituito dalla `open()`. Ogni file aperto ha assegnato un **current offset**, che è un intero positivo, che misura, in numero di byte, la posizione raggiunta nel file. Se nella costante `O_APPEND` non viene specificato, esso è settato all'inizio del file. La lettura parte dal current offset, che alla fine è incrementato col numero di byte letti. Se il current offset si trova alla fine del file o anche dopo, allora la lettura fallisce.

- `write(int fd, void *buff, int nbytes)`: praticamente, l'opposto della `read()`. Scrive un numero di bytes presi dal buffer sul file con il fd corrispondente. Ovviamente, valgono le stesse regole che abbiamo descritto prima.
- `creat(const char *pathname, mode_t mode)`: si tratta di una SC ormai in disuso, dato che con la `open()` possiamo specificare i permessi di accesso. Comunque, crea un file dal nome del pathname con i permessi descritti in mode.
- `lseek(int fd, off_t offset, int whence)`: serve a modificare il valore del current offset. Restituisce il nuovo offset o, in caso di errore, `-1`. Il primo argomento è il fd del file a cui facciamo riferimento. L'argomento whence può assumere diversi valori.  
`SEEK_SET`, che sposta il valore dall'inizio del file, `SEEK_CUR`, che sposta il valore dalla posizione corrente, `SEEK_END`, che sposta il valore a partire dalla fine del file. Da tenere a mente che questa SC non va ad aumentare la dimensione del file, perché se il current offset sfiora dalla dimensione del file, allora restituirà un fallimento.