# APPUNTI SISTEMI OPERATIVI

Corso tenuto da Emanuele Lattanzi


creato da

Stefano Zizzi

*Università degli Studi di Urbino*
*Informatica Applicata*



MATRICOLA: 312793
GIUGNO 2023

# Contents

# Chapter 1

# Introduzione

Un sistema operativo (SO) è un programma che funge da intermediario tra l'utente/altri programmi e l'hardware di un computer. Si può definire universalmente come il kernel, ovvero il programma sempre attivo in un computer.

**Evoluzione 0:** Nei primi sistemi operativi, era possibile eseguire solo un'attività alla volta. L'obiettivo del SO era fornire una libreria di driver per l'hardware e dare il pieno controllo all'applicazione. Poiché c'era un solo processo, poteva accedere a tutte le risorse e interagire direttamente con l'hardware senza controlli.

**Svantaggi** :
**Mancanza di controllo:** i processi potevano interagire direttamente con l'hardware, causando danni o monopolizzando il sistema. Se si verificava un errore, il sistema si bloccava.
**Efficienza ridotta:** non consentiva di sfruttare appieno la CPU. Un'applicazione richiede sia la CPU che operazioni di input/output (I/O), quindi la CPU rimaneva spesso inattiva senza motivo.

**Evoluzione 1:** È stato introdotto il concetto di multitasking, ovvero la capacità di eseguire contemporaneamente più attività. I processi vengono controllati dal SO.

**Vantaggi** :
Se un processo si blocca, gli altri possono continuare a eseguire.
È possibile sfruttare al massimo la CPU: mentre un processo esegue operazioni di I/O, un altro può utilizzare la CPU.

**Problemi:** Un processo poteva monopolizzare il sistema. Per risolvere questo problema, è stata introdotta la prelazione.
**Possibilità di danni:** un processo poteva danneggiare un altro processo o il sistema operativo stesso. Per prevenire questo, sono stati introdotti controlli sulle istruzioni dei processi e si è stabilito uno spazio di memoria entro cui un processo può operare.

**Evoluzione 2:** È stata introdotta la multi-utenza, consentendo di utilizzare la macchina al massimo delle sue capacità fornendo la CPU a più utenti per l'esecuzione dei loro processi.

**Problemi:**

- Protezione: è necessario proteggere un utente dagli altri, garantendo la sicurezza dei processi e dei dati (introduzione dei permessi).
- Gestione: gestire un elevato numero di utenti, risolto tramite il controllo degli accessi e l'allocazione di una quantità di risorse specifica per ogni utente.

## 1.1 Strutture dei sistemi operativi

### 1.1.1 MS-DOS

MS-DOS era un sistema operativo molto semplice, ma poco utilizzabile dall'utente. Era basato su una struttura non ben definita, e una volta sviluppato divenne pesante e difficile da debuggare.
In particolare, MS-DOS era strutturato su quattro livelli che non erano completamente separati:

- Driver del BIOS (interfacciamento diretto con l'hardware);
- Driver del dispositivo MS-DOS (interfacciamento al BIOS, che in teoria doveva essere attraversato da tutti gli altri processi);
- Programmi di sistema (interfacciamento con MS-DOS e possibilità di dialogo con il BIOS per accedere all'hardware);
- Applicazioni (possibilità di interagire direttamente con l'hardware attraverso il BIOS)

Questa possibilità per i vari programmi di dialogare direttamente con l'hardware rendeva il sistema vulnerabile.

### 1.1.2 Approccio a livelli

La struttura di questo sistema operativo era basata su un modello a livelli, con il livello 0 rappresentante l'hardware e gli altri livelli collegati in cascata, partendo dal sistema operativo al livello 1. Questo approccio richiedeva che ogni processo passasse attraverso il sistema operativo prima di raggiungere l'hardware, garantendo sicurezza a scapito delle prestazioni. Altri vantaggi erano la facilità di sviluppo e debug, in quanto ogni livello poteva essere sviluppato da un team diverso, che si occupava solo dell'interfacciamento con il livello precedente. Inoltre, gli errori erano facili da individuare ed eliminare. Modificando solo il livello 1, era possibile rendere l'intero sistema portabile su altre macchine, poiché gli altri livelli interagivano solo con quello immediatamente precedente.

### 1.1.3 UNIX

Gli sviluppatori di UNIX avevano l'obiettivo di creare un sistema operativo ridotto che si occupasse solo delle funzionalità essenziali (kernel). Tuttavia, il kernel di UNIX è cresciuto notevolmente, e nei moderni sistemi UNIX-like come Linux e macOS è stata adottata l'idea di un microkernel, eliminando ciò che non era essenziale e assegnandolo ad altri processi. Ciò rende il sistema più stabile, sicuro, facilmente portabile e debuggabile.

### 1.1.4 Moduli

I sistemi operativi moderni adottano un approccio a moduli, in cui ogni modulo è responsabile di specifiche funzionalità e tutti sono collegati centralmente al kernel, che è l'unico a poter comunicare direttamente con l'hardware. Ciò consente di mantenere i vantaggi dell'approccio a livelli senza dover attraversarli tutti ad ogni operazione. Inoltre, tutti i moduli comunicano attraverso un'interfaccia comune, il kernel, che facilita lo sviluppo e la portabilità del sistema operativo.

## 1.2 Principali funzioni di un sistema operativo

**Interfaccia utente** deve fornire un'interfaccia per l'utente per interagire con il sistema operativo.

**Esecuzione e controllo di altri programmi** deve essere in grado di avviare, eseguire e gestire l'esecuzione di altri programmi o processi.

**Protezione delle risorse** deve proteggere le risorse di sistema, i processi, se stesso e i dati degli utenti.

**Gestione delle operazioni di I/O** deve supervisionare e gestire tutte le operazioni di input/output.

**Gestione del filesystem** deve consentire la manipolazione del filesystem e garantire la protezione o condivisione dei file. Comunicazione tra processi: deve consentire la comunicazione e lo scambio di informazioni tra processi differenti.

**Allocazione delle risorse** deve gestire correttamente l'allocazione delle risorse ai processi in modo efficiente e equo.

### 1.2.1 Prelazione (preemption)

La prelazione è la capacità di terminare un processo contro la sua volontà. Per ottenere la prelazione, vengono utilizzati gli interrupt, che sono segnali inviati da un componente hardware per motivi di velocità. Ad esempio, l'interrupt del timer (RTC) viene generato a intervalli regolari (solitamente ogni 50 ms) per consentire anche ad altri processi di eseguire sulla CPU (time-sharing). Se un processo presenta un comportamento anomalo, l'RTC genera un interrupt che viene gestito dalla CPU. La CPU eseguirà le istruzioni specificate nella tabella delle istruzioni per gli interrupt, che sono caricate dal sistema operativo durante l'avvio e di solito sono memorizzate nella memoria di sistema. Inoltre, l'interrupt generato dal timer garantisce che i processi vengano eseguiti in modo equo e che nessun processo monopolizzi la CPU per troppo tempo.

### 1.2.2 Modalità privilegiata

Per proteggere il sistema, sono state definite istruzioni privilegiate che possono essere eseguite solo quando la CPU è in modalità privilegiata. Queste istruzioni includono operazioni di controllo, I/O e gestione della memoria. All'avvio del sistema, la CPU viene impostata in modalità privilegiata e il sistema operativo viene eseguito in questa modalità. Il sistema operativo scrive le istruzioni privilegiate nella memoria di sistema e, prima di passare il controllo ad altri processi, disabilita la modalità privilegiata. In questo modo, solo il sistema operativo può eseguire istruzioni privilegiate. Se un processo dannoso riesce ad eseguire prima del sistema operativo, otterrà il controllo del sistema in modalità privilegiata. Per eseguire istruzioni privilegiate, i processi devono fare delle chiamate di sistema (system call) per interpellare il sistema operativo. In questo modo, passano il controllo al sistema operativo (che opera in modalità privilegiata), che esamina l'istruzione richiesta e la esegue prima di riportare la CPU in modalità non privilegiata e restituire il controllo al processo.

### 1.2.3 Protezione della memoria

Per consentire il corretto funzionamento di un processo, è necessario che venga assegnato uno spazio di memoria specifico, ma allo stesso tempo deve essere garantito che il processo non scriva nella memoria di un altro processo. L'area di memoria dedicata a un processo è chiamata spazio degli indirizzi, e è necessario controllare che un processo non esca dal proprio spazio. Per migliorare l'efficienza di questo controllo, viene utilizzato un componente hardware chiamato Memory Management Unit (MMU), che verifica se un processo sta scrivendo al di fuori del proprio spazio di indirizzamento. Se ciò accade, viene generato un interrupt e il sistema operativo viene coinvolto nella gestione dell'errore.

# Chapter 2

# Processi

## 2.1 System Calls

Le system call (chiamate di sistema) sono un meccanismo attraverso il quale un processo può richiedere servizi al sistema operativo. Per utilizzare le system call, i diversi sistemi operativi forniscono API (Application Program Interface) specifiche, che variano per ogni sistema operativo. Per effettuare una system call, un processo genera una trap, ossia un'eccezione che viene rilevata dalla CPU. La CPU legge quindi le istruzioni da eseguire in caso di trap da una tabella delle trap e richiama il sistema operativo. La trap include l'ID specifico della system call necessaria per indicare al sistema operativo quale operazione eseguire, utilizzando una tabella degli ID delle system call. Ci sono diversi tipi di system call che comprendono funzioni di controllo dei processi, gestione dei file, operazioni di input/output e comunicazione tra processi.

**Passaggio dei parametri:** Quando un processo effettua una system call, spesso è necessario passare dei parametri su cui la system call dovrà operare (ad esempio, copia_file(fileOrigine, fileDestinazione)). Per passare questi parametri, ci sono tre possibilità:

- **Attraverso i registri:** i parametri di piccole dimensioni vengono direttamente caricati nei registri della CPU. Questo approccio è semplice ma limitato dal numero e dalla capacità dei registri.
- **Attraverso tabelle:** viene memorizzato nei registri della CPU solo un puntatore all'area di memoria del processo che contiene i parametri. Questa area di memoria può essere organizzata a blocchi o tabelle.
- **Attraverso lo stack:** alcuni sistemi operativi mettono a disposizione una memoria di scambio, organizzata come uno stack, a cui sia il processo (che scrive i dati in cima allo stack) che il sistema operativo (che legge i dati appena inseriti) possono accedere. Tuttavia, questa soluzione introduce un overhead per la gestione dei dati.

Inoltre, le system call spesso restituiscono dei valori. Tuttavia, questi valori non possono essere letti direttamente dal processo nella memoria del sistema operativo, poiché il processo non può accedervi. Pertanto, si genera comunque un overhead per la gestione dei dati restituiti dalla system call. I valori possono essere:

1. Memorizzati nei registri della CPU.
2. Scritti in un'area di memoria del processo (nella tabella delle system call).
3. Scritti nello stack e successivamente letti dal processo.

## 2.2 Processi

### 2.2.1 Gestione dei processi

I processi costituiscono l'elemento fondamentale di qualsiasi attività svolta dall'utente. La suddivisione dei task in processi distinti consente una gestione più semplice e, soprattutto, mantiene sempre la CPU in uso, poiché un processo può eseguire durante l'attesa di altri processi. Un processo è definito come un programma attualmente in esecuzione, ovvero caricato in RAM. Questo processo deve essere distinto dal programma, che è un oggetto passivo che non viene eseguito e si trova sul disco.
Un processo è composto da diverse aree di memoria:

- **Codice:** contiene le istruzioni macchina da eseguire, prese dal file eseguibile compilato. Queste istruzioni saranno lette dalla CPU durante l'esecuzione del processo.
- **Data:** è l'area di memoria dedicata alle variabili globali del processo.
- **Heap:** è la memoria allocata dinamicamente dal processo durante l'esecuzione.
- **Stack:** è l'area di memoria dedicata alle variabili locali del processo.

Heap e Stack sono due aree di memoria dinamiche che possono cambiare dimensione nel corso del tempo.

Uno stato di un processo può essere uno dei seguenti:

- **New:** il processo viene creato.
- **Running:** il processo è in esecuzione.
- **Waiting:** il processo attende un evento, come l'input da parte dell'utente o il completamento di un'operazione di I/O.
- **Ready:** il processo è pronto per essere eseguito, ma deve ancora ottenere la CPU.
- **Terminated:** il processo è terminato, può essere perché ha completato l'esecuzione, ha riscontrato un errore o è stato terminato dall'OS o da un altro processo a nome dell'utente.

Dopo la creazione, un processo passa dallo stato New allo stato Ready. Quando arriva il suo turno, il processo viene eseguito (stato Running). Se il processo deve attendere un evento, passa allo stato Waiting. Una volta ricevuto l'evento, torna allo stato Ready per essere eseguito di nuovo. Alla fine dell'esecuzione o in caso di errore, il processo passa allo stato Terminated.

Ogni processo ha un blocco di controllo del processo (PCB) associato, che viene creato dal sistema operativo nella sua area di memoria. Il PCB contiene informazioni utili per la gestione dei processi, come:

- PID (Process ID) del processo.
- Stato corrente del processo.
- Copia del program counter (PC), che indica il punto di ripartenza del processo dopo un'interruzione o quando viene interrotto per entrare nello stato di attesa.
- Informazioni sull'area di memoria del processo.
- Informazioni statistiche utilizzate per prevedere il comportamento del processo.
- Informazioni di I/O, ad esempio i file aperti dal processo.

Creazione dei processi: Per avviare un nuovo programma, il sistema operativo permette a ogni processo di creare processi figli tramite una system call.

- Ogni processo deve essere creato da un altro processo che ne è il padre.

- Ogni processo figlio creato è una copia del processo padre, ma successivamente viene sovrascritto con il nuovo eseguibile.
- I due processi possono comunicare tra loro o ignorarsi. Il processo padre può aspettare la terminazione del processo figlio, ma può anche continuare l'esecuzione contemporaneamente.
- Il primo processo ad essere avviato è il sistema operativo stesso, che genera più processi di sistema, tra cui INIT, il processo padre di tutti i processi utente.

Nel caso dei sistemi Linux, l'istruzione utilizzata per creare processi figli è la "FORK". La "FORK" crea un processo identico al processo padre e restituisce al processo padre il PID del processo figlio ("-1" in caso di errore). Al processo figlio viene invece restituito il valore 0 (poiché non può essere confuso con nessun altro processo), consentendo al programmatore di distinguere tra processo padre e processo figlio. Successivamente, viene eseguito un nuovo programma nel processo figlio tramite l'istruzione "EXEC", che sovrascrive completamente il processo (se l'istruzione "EXEC" venisse eseguita nel processo padre, il processo padre verrebbe sovrascritto). Di solito, il processo padre rimane in attesa della terminazione del processo figlio o di un valore restituito da esso tramite l'istruzione "WAIT", cioè dal processo con PID uguale a quello restituito dalla "FORK".

### 2.2.2 Context Switch

Il cambio di contesto, noto anche come context switch, è la necessità di passare dall'esecuzione di un processo a un altro processo (ad esempio, quando il primo processo deve eseguire operazioni di I/O). Per gestire questa funzionalità, il sistema operativo utilizza code i cui elementi sono i blocchi di controllo dei processi (PCB). Oltre alla coda dei processi pronti, esistono code specifiche per i vari dispositivi di I/O.

Quando il sistema operativo decide di spostare un processo che sta eseguendo per passare a un altro processo, il "Short-term Scheduler" seleziona un nuovo processo dalla coda dei processi pronti e lo passa all'esecuzione sulla CPU. Il punto in cui viene effettuato questo passaggio è chiamato punto di scheduling e può verificarsi per diversi motivi:

- Un processo viene sospeso per eseguire operazioni di I/O, attendere un valore, creare un processo figlio o a causa di un errore.
- Scade il tempo di time-sharing assegnato al processo corrente.
- Un altro processo ha una priorità più alta e deve essere eseguito immediatamente.

Il "Long-term Scheduler" interviene quando la memoria RAM è quasi piena. In questo caso, il sistema operativo utilizza il disco per allocare spazio per i processi. Il "Long-term Scheduler" si occupa di spostare i processi tra la coda dei processi pronti in memoria e la memoria di massa. Poiché il "Short-term Scheduler" deve agire rapidamente, il "Long-term Scheduler" può dedicare più tempo per valutare le decisioni.

Durante l'esecuzione di un processo, il sistema operativo effettua valutazioni e classifica i processi in modo da assegnare loro la priorità appropriata.

Le politiche di scheduling possono includere:

- Gestione dei processi in base alla loro priorità individuale.
- Fornire a tutti i processi opportunità di esecuzione senza privilegiarne troppo uno rispetto agli altri.
- Vincoli di tempo reale, che richiedono l'esecuzione di un processo entro determinati limiti di tempo (ad esempio, per la riproduzione video che richiede un frame rate costante).
- Ottimizzazione, che può favorire i processi intensivi in operazioni di I/O per garantire una maggiore reattività del sistema, o i processi intensivi in operazioni di calcolo per completare le attività più velocemente a discapito della reattività del sistema.

Overhead del Context Switch:

- Overhead diretto: durante un context switch, è necessario salvare i dati del processo in esecuzione e caricare i dati del processo successivo.
- Overhead indiretto: una volta che il nuovo processo viene eseguito, potrebbe trovarsi in una cache che contiene dati relativi al processo precedente, causando diverse cache miss. Lo stesso problema può verificarsi con il Translation Lookaside Buffer (TLB), che gestisce la conversione degli indirizzi logici in indirizzi fisici. Per mitigare questo problema, si cerca di eseguire un processo sulla stessa CPU su cui ha precedentemente eseguito, in modo che possa beneficiare di una cache "calda" (cioè una cache popolata con i dati del processo).

**Kernel Control Path (KCP):** Ogni volta che viene eseguita una system call, il sistema operativo opera su un percorso chiamato Kernel Control Path (KCP), che è l'equivalente dei processi all'interno del sistema operativo. Inizialmente, si pensava di utilizzare un unico KCP, ma questa scelta impediva la preemption all'interno del sistema operativo in caso di errori.

**Kernel Rientranti:** È possibile percorrere più KCP contemporaneamente, consentendo l'esecuzione simultanea di più system call e la possibilità di interrompere anche il sistema operativo in caso di problemi, allo stesso modo dei processi (kernel preemptive).

Il context switch è una parte fondamentale della gestione dei processi all'interno di un sistema operativo. Consente di passare tra i processi in modo efficiente e di bilanciare l'utilizzo della CPU, garantendo una corretta esecuzione dei processi e una buona reattività del sistema.

## 2.3 Thread vs. Processi

I thread sono processi più leggeri che consentono la divisione interna di un processo in percorsi di esecuzione distinti chiamati thread. Presentano diversi vantaggi:

- **Condivisione dell'area di dati**: I thread appartengono allo stesso processo e condividono l'area di dati del processo senza la necessità di memorie condivise aggiuntive.
- **Accesso alla memoria condivisa**: La lettura e la scrittura sulla memoria condivisa avvengono senza il coinvolgimento del sistema operativo, poiché avvengono all'interno del processo.
- **Eliminazione della copia del processo**: Non è necessario duplicare completamente il processo per creare un thread.
- **Eliminazione delle system call**: I thread possono eseguire senza la necessità di system call per la comunicazione tra di loro.
- **Parallelismo su sistemi multi-core**: Su sistemi multi-core, più thread possono essere eseguiti contemporaneamente, fornendo migliori prestazioni rispetto ai processi.

Tuttavia, i thread presentano anche alcuni svantaggi, come il rischio di danneggiare i dati condivisi o leggere dati non validi da parte di un altro thread.

### 2.3.1 Modelli di mappature

Per gestire la divisione dei flussi di esecuzione, sono stati introdotti diversi modelli di mappatura dei thread a livello del kernel, all'interno del processo:

- **Many-to-One**: Più thread vengono mappati in un unico kernel thread eseguito dal sistema operativo. Tuttavia, questa modalità presenta il problema dell'esecuzione sequenziale dei thread, poiché solo un thread può essere eseguito alla volta (Process Contention Scope).
- **One-to-One**: Ogni thread viene mappato su un diverso kernel thread. Il sistema operativo gestisce questi kernel thread come processi separati e li schedula come gli altri processi (System Contention Scope). Tuttavia, questa modalità può generare uno spreco di risorse se un processo genera un numero elevato di thread.

- **Many-to-Many**: Consente di mappare N thread in M kernel thread, in cui il sistema operativo decide il numero massimo di kernel thread disponibili per ogni processo. Questo approccio permette di ridurre o aumentare il numero massimo di kernel thread a disposizione di ogni processo in base alla disponibilità di memoria, limitando eventuali danni causati da processi errati.

È importante notare che la scelta del modello di mappatura dei thread dipende dalle caratteristiche del sistema operativo e dalle esigenze delle applicazioni.

### 2.3.2 Problemi dei Thread

L'utilizzo dei thread può presentare alcune problematiche che devono essere gestite adeguatamente:

- **Cancellazione**: A differenza dei processi, i thread non possono essere terminati immediatamente in modo asincrono. Prima di terminare, un thread deve gestire le attività che ha lasciato in sospeso, specialmente se sta lavorando in modo concorrente con altri thread.
- **Segnali**: Poiché non vi è una corrispondenza diretta tra thread e kernel thread, i segnali devono essere pianificati quando arrivano e indirizzati ai thread corretti. Questo compito è affidato all'handler dei segnali, che raccoglie i segnali e li inoltra al thread appropriato.
- **Dati privati**: In alcuni casi può essere necessario avere aree di memoria accessibili solo da un singolo thread.
- **Comunicazione**: Per comunicare tra thread, il sistema operativo utilizza le upcall, che consentono al sistema operativo di interpellare un thread specifico, a differenza delle system call che vengono effettuate dal thread.

## 2.4 Comunicazione fra processi

A differenza dei thread, i processi non possono comunicare direttamente tra loro a causa dell'isolamento delle loro aree di memoria. Tuttavia, esistono diverse modalità di comunicazione tra processi.

### 2.4.1 Memoria condivisa

La memoria condivisa prevede l'allocazione di un'area di memoria separata ai processi e condivisa tra di loro nello spazio di indirizzamento. In questo modo, il sistema operativo non si preoccupa di ciò che i processi fanno in quell'area, che viene considerata parte del loro spazio di indirizzamento. I processi possono cooperare senza l'uso di system call, ma questa modalità può comportare il rischio di danneggiare i dati condivisi a causa della mancanza di controllo.

Il seguente é il principale paradigma per la comunicazione tramite memoria condivisa:

- **Paradigma Produttore-Consumatore**: In questo paradigma, un processo produttore scrive dati in un'area di memoria condivisa chiamata buffer, mentre un processo consumatore legge i dati dallo stesso buffer. Può essere implementato con un buffer limitato, in cui il produttore si ferma quando il buffer è pieno e il consumatore si ferma quando il buffer è vuoto, oppure con un buffer illimitato in cui i processi scrivono/leggono senza preoccuparsi dello stato del buffer.

### 2.4.2 Message passing

Il message passing prevede che un processo A invii messaggi al sistema operativo per essere recapitati al processo B, che si mette in ascolto per riceverli. Il sistema operativo si occupa di schedulare i messaggi quando rileva che il processo B è in attesa. Questo metodo è più lento in quanto richiede l'utilizzo di system call per ogni comunicazione e comporta un overhead dei dati (memoria dell'OS e memoria del processo ricevente). Tuttavia, è più sicuro poiché consente al sistema operativo di controllare la comunicazione.

Il message passing può avvenire in due modalità:

- **Bloccante**: Il mittente invia il messaggio e attende che qualcuno lo riceva, bloccandosi finché non riceve una risposta. Il destinatario si mette in attesa e attende il messaggio.
- **Non bloccante**: Il mittente invia il messaggio e continua le sue operazioni senza attendere una risposta immediata. Il destinatario si mette in ascolto, ma se può, continua a eseguire altre operazioni.

È importante notare che se la coda dei messaggi è piena, il mittente può scegliere di attendere o sovrascrivere i messaggi presenti.

### 2.4.3 Comunicazione Client-Server

I processi su macchine distinte possono comunicare tra loro utilizzando il concetto di client-server. In questo modello, i processi possono comunicare attraverso socket, che combinano l'indirizzo IP della macchina su cui è in esecuzione il processo da contattare con l'ID o la porta dello specifico processo.

Un'altra modalità di comunicazione client-server è la Remote Procedure Call (RPC), che consente di invocare una funzione di processo situata su un'altra macchina. In questo caso, è necessario passare attraverso un software Stub che si occupa della traduzione tra i diversi sistemi.

# Chapter 3

# Sincronizzazione

## 3.1 Il problema della sezione critica

La sezione critica è una porzione di codice in cui un processo utilizza una risorsa o una memoria condivisa. Quando due o più processi lavorano sulla stessa memoria condivisa, è possibile che si verifichino problemi come l'inconsistenza dei dati o la corruzione delle informazioni. Inoltre, a causa del time-sharing, un processo può essere sospeso durante l'esecuzione della sezione critica, causando possibili errori per il processo successivo che accede alla memoria condivisa.

Per risolvere il problema della sezione critica, devono essere soddisfatte tre condizioni:

1. **Mutua esclusione**: Se un processo sta eseguendo nella sua sezione critica, nessun altro processo può eseguire nella propria sezione critica contemporaneamente.
2. **Progresso**: Se nessun processo sta eseguendo nella sezione critica e ci sono processi che desiderano eseguire, un processo deve essere scelto in tempo finito per poter eseguire la sua sezione critica.
3. **Attesa limitata**: Un processo che desidera entrare nella sezione critica non può essere costretto ad attendere indefinitamente. Deve essere garantito a tutti i processi l'accesso alla sezione critica.

Una soluzione classica per il problema della sezione critica è la soluzione di Peterson, valida per due processi. Tuttavia, soluzioni più moderne utilizzano strumenti di sincronizzazione hardware, come le istruzioni atomiche.

## 3.2 Soluzione di Peterson

La soluzione di Peterson è una soluzione per il problema della sezione critica che coinvolge due processi. La soluzione prevede l'utilizzo di due variabili booleane condivise e una variabile intera condivisa.

Prima di entrare nella sezione critica, ogni processo alza il proprio flag di "pronto" e imposta il turno all'altro processo, quindi attende finché l'altro processo non completa la sua sezione critica. In questo modo, si evitano conflitti tra i due processi, poiché uno dei due processi passerà il turno all'altro e solo uno dei processi sarà in esecuzione. Al termine della propria sezione critica, il processo corrente passa il turno all'altro processo e reimposta a falso il proprio flag di "pronto".

Tuttavia, la soluzione di Peterson non è più ampiamente utilizzata in quanto si preferiscono soluzioni che sfruttano strumenti di sincronizzazione hardware più efficienti.

## 3.3 Strumenti di sincronizzazione hardware

Al giorno d'oggi, per affrontare il problema della sezione critica, vengono utilizzati strumenti di sincronizzazione hardware che offrono istruzioni atomiche non interrompibili e veloci. Due esempi comuni di istruzioni atomiche sono TestAndSet e Swap.

- **TestAndSet**: Questa istruzione legge un bit in memoria e restituisce il suo valore, contemporaneamente sovrascrive il bit a "vero" a tempo di clock. Se due processi eseguono una TestAndSet su una variabile booleana condivisa prima di entrare nella propria sezione critica, solo il primo processo che accede restituirà il valore "falso" e quindi avrà il permesso di eseguire la sezione critica. Gli altri processi, trovando il bit già impostato a "vero", dovranno attendere.

- **Swap**: L'istruzione Swap scambia il valore di due variabili a tempo di clock. Utilizzando questa istruzione, è possibile implementare una soluzione per la sezione critica in cui ogni processo utilizza una variabile privata e una variabile condivisa. Prima di entrare nella sezione critica, ogni processo esegue una serie di swap sulla variabile condivisa e la propria variabile privata. Solo se la variabile privata passa a "falso", il processo può eseguire la sezione critica. Il primo processo che accede alla sezione critica scambia la variabile condivisa con la propria variabile privata, rendendo così la variabile condivisa "falsa" per gli altri processi. Al termine della sezione critica, il processo reimposta la variabile condivisa a "falso", consentendo ad altri processi di competere nuovamente.

Questi strumenti di sincronizzazione hardware permettono di implementare soluzioni efficienti per il problema della sezione critica senza la necessità di utilizzare soluzioni software complesse come la soluzione di Peterson.

## 3.4 Strumenti di sincronizzazione e il loro utilizzo

### 3.4.1 Semafori

I semafori sono strumenti di sincronizzazione che utilizzano una variabile condivisa chiamata "S" per indicare il numero di permessi disponibili. Essi supportano due operazioni binarie:

- **Wait()**: utilizzata per ottenere un permesso da S. Se S è uguale a 0 (nessun permesso disponibile), la Wait() mette in attesa il processo che la invoca.

- **Signal()**: utilizzata per restituire un permesso a S (incrementa S di 1) una volta che il processo ha finito di utilizzare la risorsa.

Esistono due tipi comuni di semafori:

- **Semaforo binario (lock o mutex)**: S può avere al massimo 1 permesso.

- **Semaforo contatore**: S può avere un numero n di permessi, limitato dalle risorse disponibili.

**Attesa**:

- **Spin Lock (busy-waiting)**: il processo continua a tentare l'acquisizione del semaforo (while(¡condizione¿)). Questo significa che il processo continua a eseguire operazioni inutili in attesa.

- **Context Switch**: quando un processo sa di dover attendere, può richiedere direttamente un context switch. In questo modo, il processo rilascia immediatamente la CPU a favore di altri processi senza occupare inutilmente la CPU stessa. Tuttavia, vi è un costo associato ai context switch. Per implementare questa soluzione, vengono create code di processi in attesa di un semaforo specifico.

Sezione critica breve: può essere utilizzato uno spin lock. Altrimenti, è preferibile utilizzare il context switch. Nei sistemi single-core, il context switch viene utilizzato in quanto non ci sono altri processi che potrebbero interrompere l'attesa prima del context switch.

**Deadlock**: si verifica quando i processi sono in attesa di un evento che solo loro potrebbero generare.

**Starvation**: si verifica quando i processi sono in attesa di essere eseguiti per un tempo potenzialmente infinito (ad esempio, a causa di una priorità troppo bassa senza aging).

### 3.4.2 Problemi classici di sincronizzazione

#### 3.4.2.1 Problema del produttore-consumatore

Questo problema coinvolge un buffer condiviso tra un produttore e un consumatore. Per risolvere questo problema, sono necessari i seguenti strumenti di sincronizzazione:

- Un semaforo binario (mutex = 1) per garantire la mutua esclusione nell'accesso al buffer condiviso.
- Un semaforo contatore (full = 0) che indica il numero di posizioni piene nel buffer.
- Un semaforo contatore (empty = n) che indica il numero di posizioni vuote

Il produttore esegue le seguenti operazioni:

- Effettua la produzione dell'elemento da inserire nel buffer.
- Esegue la wait(empty) per verificare se ci sono posizioni vuote nel buffer. Se il buffer è pieno, il produttore si ferma in attesa.
- Esegue la wait(mutex) per garantire la mutua esclusione sull'accesso al buffer.
- Aggiunge l'elemento al buffer.
- Esegue la signal(mutex) per rilasciare la mutua esclusione sull'accesso al buffer.
- Esegue la signal(full) per incrementare il numero di posizioni piene nel buffer.

Il consumatore esegue le seguenti operazioni:

- Esegue la wait(full) per verificare se ci sono posizioni piene nel buffer. Se il buffer è vuoto, il consumatore si ferma in attesa.
- Esegue la wait(mutex) per garantire la mutua esclusione sull'accesso al buffer.
- Legge l'elemento dal buffer.
- Esegue la signal(mutex) per rilasciare la mutua esclusione sull'accesso al buffer.
- Esegue la signal(empty) per incrementare il numero di posizioni vuote nel buffer.

#### 3.4.2.2 Problema dei lettori-scrittori

Questo problema coinvolge più processi lettori e scrittori che operano su una memoria condivisa. Le regole per la sincronizzazione sono le seguenti:

- Solo uno scrittore alla volta può accedere alla memoria condivisa (senza lettori).
- I lettori possono accedere alla memoria condivisa contemporaneamente (senza scrittori).

Per risolvere questo problema, sono necessari i seguenti strumenti di sincronizzazione:

- Un mutex (mutex = 1) per garantire la mutua esclusione nell'accesso alla variabile "lettori_attivi".
- Un mutex (wrt = 1) per garantire il permesso di scrittura.
- Un intero (lettori_attivi = 0) che indica il numero di lettori attivi.

Lo scrittore esegue le seguenti operazioni:

- Esegue la wait(wrt) per acquisire il permesso di scrittura.
- Esegue l'operazione di scrittura sulla memoria condivisa.
- Esegue la signal(wrt) per rilasciare il permesso di scrittura.

Il lettore esegue le seguenti operazioni:

- Esegue la wait(mutex) per garantire la mutua esclusione nell'accesso alla variabile "lettori_attivi".
- Incrementa lettori_attivi di 1.
- Se lettori_attivi è uguale a 1, esegue la wait(wrt) per ottenere il permesso di scrittura e impedire agli scrittori di accedere.
- Esegue la signal(mutex) per rilasciare la mutua esclusione sull'accesso alla variabile "lettori_attivi".
- Esegue l'operazione di lettura sulla memoria condivisa.
- Esegue la wait(mutex) per garantire la mutua esclusione nell'accesso alla variabile "lettori_attivi".
- Decrementa lettori_attivi di 1.
- Se lettori_attivi è uguale a 0, esegue la signal(wrt) per rilasciare il permesso di scrittura.
- Esegue la signal(mutex) per rilasciare la mutua esclusione sull'accesso alla variabile "lettori_attivi".

Questo metodo potrebbe causare starvation degli scrittori se continuano ad arrivare lettori. Per risolvere questo problema, si può utilizzare un semaforo contatore sui lettori.

#### 3.4.2.3 Problema della cena dei 5 filosofi

Questo problema coinvolge 5 processi (i filosofi) che devono accedere a una risorsa condivisa (il riso) utilizzando due bacchette ciascuno. Per risolvere questo problema, sono necessari i seguenti strumenti di sincronizzazione:

- 5 mutex (bacchette), una per ogni filosofo.
- Un semaforo contatore (tavolo = 4) che regola l'accesso al tavolo dei filosofi.

Ogni filosofo esegue le seguenti operazioni:

- Esegue la wait(bacchetta[i]) per richiedere la bacchetta alla sua destra.
- Esegue la wait(bacchetta[(i+1)
- Mangia (accede alla risorsa condivisa).
- Esegue la signal(bacchetta[i]) per rilasciare la bacchetta alla sua destra.
- Esegue la signal(bacchetta[(i+1)
- Pensa o esegue altre operazioni.

Se tutti i filosofi cercano di acquisire la bacchetta alla loro destra contemporaneamente, nessuno sarà in grado di ottenere la bacchetta alla loro sinistra, causando un deadlock. Per evitare questa situazione, si può utilizzare un semaforo contatore sul tavolo dei filosofi, inizializzato a 4, in modo che almeno un filosofo possa mangiare.

### 3.4.3 Monitor

Il monitor è uno strumento di sincronizzazione che fornisce la mutua esclusione nativa. Solo un processo alla volta può accedere al monitor e utilizzare le sue funzionalità, mentre gli altri processi vengono messi in attesa in una coda. Utilizzando l'area di memoria del monitor come memoria condivisa, è possibile coordinare i processi che operano all'interno del monitor.

È possibile che più processi si trovino contemporaneamente all'interno del monitor, ma solo uno di essi può essere attivo alla volta, mentre gli altri rimangono in attesa. Questo assicura che le operazioni all'interno del monitor siano eseguite in modo sequenziale.

Ad esempio, nel caso del problema produttore-consumatore, il produttore accede al monitor per scrivere sul buffer condiviso, mentre il consumatore accede al monitor per leggere dal buffer. In questo modo, si evita la concorrenza e si garantisce la correttezza delle operazioni.

Tuttavia, l'implementazione dei monitor può variare a seconda del sistema operativo e del linguaggio di programmazione utilizzato. Alcuni linguaggi di programmazione forniscono costrutti specifici per la gestione dei monitor, come ad esempio Java con il suo meccanismo di synchronized.

# Chapter 4

# Scheduling

## 4.1 Concetti fondamentali

La multiprogrammazione consente di massimizzare l'utilizzo della CPU consentendo a più processi di essere caricati in memoria contemporaneamente. Ogni processo esegue una serie di burst, che rappresentano i periodi di utilizzo della CPU e delle operazioni di I/O.

Lo scheduler è responsabile di selezionare quale processo deve essere eseguito dalla CPU tra quelli presenti nella ready queue. Una volta presa la decisione, il dispatcher si occupa di effettuare il cambio di contesto, caricando i registri del processo selezionato e concedendo il controllo della CPU al processo stesso.

## 4.2 Criteri di scheduling

Esistono diversi criteri di scheduling utilizzati per selezionare i processi da eseguire:

- **Massimizzare l'utilizzo della CPU**: l'obiettivo è mantenere la CPU occupata il più possibile, evitando tempi di inattività. Questo criterio consente di elaborare più processi in modo più rapido e di mettere la CPU a riposo prima.
- **Massimizzare la frequenza**: l'obiettivo è eseguire il maggior numero possibile di processi in un'unità di tempo. Questo criterio è utile in ambienti in cui vi è una grande quantità di processi e una breve durata di ogni burst di CPU.
- **Minimizzare il tempo di turnaround**: l'obiettivo è ridurre il tempo trascorso dal momento in cui un processo entra nella ready queue e quando termina la sua esecuzione. Questo tempo comprende sia il tempo di attesa che il tempo di esecuzione effettivo.
- **Minimizzare il tempo di attesa**: l'obiettivo è ridurre il tempo medio di attesa dei processi nella ready queue. Questo criterio mira a garantire una giusta distribuzione delle risorse e a evitare situazioni in cui alcuni processi rimangono in attesa per un tempo molto lungo.
- **Minimizzare il tempo di risposta**: l'obiettivo è ridurre il tempo che intercorre tra una richiesta di un processo e la produzione della risposta. Questo criterio è importante in applicazioni interattive, in cui l'utente richiede una risposta immediata.

È importante notare che questi criteri di scheduling non sono compatibili tra loro, il che significa che favorire un criterio può portare a uno svantaggio in un altro criterio. Pertanto, i sistemi operativi adottano diverse politiche di scheduling per bilanciare questi obiettivi e adattarsi alle esigenze specifiche dell'ambiente e dei processi in esecuzione.

### 4.2.1 Algoritmi di scheduling

Esistono diversi algoritmi di scheduling utilizzati nei sistemi operativi. Alcuni dei principali sono:

#### 4.2.1.1 First-Come First-Served (FCFS)

Nell'algoritmo FCFS, i processi vengono schedulati nell'ordine in cui entrano nella ready queue. Ogni processo esegue fino al termine senza essere interrotto, e gli unici punti di scheduling si verificano alla fine dei processi. Tuttavia, questa scelta è poco performante in quanto dipende strettamente dall'ordine di arrivo dei processi.

#### 4.2.1.2 Shortest Job First (SJF)

L'algoritmo SJF tiene conto del burst time di ogni processo e schedula sempre il processo con il burst time più breve.

- **SJF non preemptive:** una volta selezionato il processo con il burst time più breve, questo esegue fino al termine senza essere interrotto.
- **SJF preemptive (Shortest Remaining Time First):** ogni volta che arriva un nuovo processo nella ready queue, si verifica quale processo ha il burst time rimanente più breve. Se il nuovo processo ha un burst time più breve del processo in esecuzione, viene effettuato uno scheduling preemptive e il nuovo processo prende il controllo della CPU.

Tuttavia, l'algoritmo SJF presenta alcuni problemi, come la difficoltà nell'ottenere il burst time dei processi in modo accurato e il rischio di starvation per i processi con burst time elevato. Per mitigare questo problema, vengono utilizzate politiche di aging, che assegnano bonus di priorità ai processi in attesa da molto tempo.

#### 4.2.1.3 Round-Robin (RR)

L'algoritmo Round-Robin assegna a ogni processo un *quanto di tempo* (quantum). Un processo inizia ad eseguire e continua fino a quando non esaurisce il suo quanto di tempo, dopodiché viene interrotto e viene selezionato il prossimo processo nella ready queue. Questo algoritmo impedisce la starvation, in quanto ogni processo deve attendere al massimo $(n-1) \times q$ tempo, dove $n$ è il numero totale di processi. Tuttavia, l'algoritmo RR può essere poco stabile con un gran numero di processi, poiché comporta numerosi context switch. Di solito viene utilizzato solo per processi in tempo reale, in cui è necessario garantire l'esecuzione a intervalli regolari di tempo. È importante impostare un valore adeguato per il quanto di tempo, in quanto un quanto troppo grande riduce l'efficienza, mentre un quanto troppo piccolo comporta un elevato numero di context switch.

#### 4.2.1.4 Code multi-livello

L'algoritmo delle code multi-livello prevede l'utilizzo di diverse code di processi con priorità decrescente. I processi nelle code con priorità più bassa vengono eseguiti solo quando le code precedenti hanno terminato l'esecuzione dei processi. Le code con priorità più alta utilizzano spesso l'algoritmo RR, e in queste code vengono inseriti i processi in tempo reale o con privilegi elevati. Tuttavia, per gli altri processi, non è possibile determinare la priorità in anticipo. Attraverso politiche di aging, è possibile spostare i processi tra le code in base al loro comportamento. Ad esempio, i nuovi processi entrano nella coda con priorità più alta e, se un processo non ha esaurito il suo quanto di tempo dopo un certo intervallo, viene spostato nelle code a priorità più bassa. In questo modo, i processi intensivi in I/O possono essere favoriti nelle code appropriate.

#### 4.2.1.5 Sistemi multi-core

Nei sistemi multi-core, lo scheduling è più complesso. Esistono due tipi principali di sistemi multi-core:

- **Asimmetrici:** in questi sistemi, c'è un processore principale (master) che controlla gli altri processori (slave). È il processore principale a gestire lo scheduling sulle altre CPU e a coordinare le attività dei processi.
- **Simmetrici (SMP - Symmetric Multiprocessing):** in questi sistemi, tutti i processori sono identici e sono in grado di effettuare lo scheduling autonomamente. Esistono due varianti di sistemi SMP:

  - **SMP a coda condivisa:** in questo approccio, c'è una singola coda di processi pronti condivisa tra tutti i processori. Tuttavia, per evitare problemi di race condition, questa coda è regolata da meccanismi di sincronizzazione. Anche se è meno efficiente, non richiede un bilanciamento dei carichi tra i processori.
  - **SMP a code private:** in questo approccio, ogni processore ha la propria coda di processi pronti. Questo può portare a uno squilibrio nel sistema, con una CPU completamente occupata e un'altra sottoutilizzata. Pertanto, è necessario effettuare un bilanciamento dei carichi per ridistribuire i processi tra i processori e garantire un utilizzo equo delle risorse.

Inoltre, nei sistemi multi-core possono essere utilizzate diverse strategie di affinità per indirizzare i processi ai core appropriati:

- **Affinità soft:** un processo sviluppa una certa affinità per il processore su cui viene eseguito, popolando la cache di quel processore. Pertanto, si cerca di mantenere il processo nello stesso core, a meno che non vi sia uno squilibrio del carico di lavoro.
- **Affinità hard:** alcuni processi richiedono di non essere mai spostati dal core su cui stanno eseguendo. Questa affinità rigida può essere necessaria per garantire la corretta esecuzione di certe applicazioni o servizi critici.

È importante adottare strategie di scheduling appropriate per massimizzare l'efficienza e l'utilizzo delle risorse in un ambiente multi-core.

# Chapter 5

# Scheduling in Linux

## 5.1 Linux Process Descriptor

In Linux, i PCB (Process Control Block) sono chiamati Linux Process Descriptor. Tra i numerosi campi presenti, i più importanti sono:

- **Priority:** indica la priorità iniziale del processo.
- **Counter:** indica il quanto di tempo rimanente e funge da indicatore di priorità durante l'esecuzione.
- **Policy:** indica la politica di scheduling da adottare.
- **Area di memoria del processo.**
- **Puntatori ai files aperti.**
- **Puntatori ai processi che precedono/seguono nelle code di scheduling.**

Stati di un processo:

- **Running:** processi in esecuzione o pronti ad eseguire (ready-queue).
- **Interruptible:** processi in attesa il cui stato di attesa può essere interrotto dall'arrivo dei dati attesi o dal sistema.
- **Uninterruptible:** processi in attesa che terminano l'attesa solo quando arrivano i dati attesi (solitamente utilizzato quando è necessario attendere il completamento di un'operazione per evitare problemi).
- **Stopped:** processo terminato (in modo normale o a causa di errori o terminazione forzata), i suoi dati vengono liberati.
- **Zombie:** processi che hanno terminato l'esecuzione ma non hanno più un padre che ne prenda il valore di ritorno. Vengono periodicamente puliti dal processo Init.

## 5.2 Caratteristiche principali

- **Time Sharing:** ogni processo riceve un quantum di tempo durante il quale può essere eseguito. Una volta esaurito il suo quantum, il processo viene sospeso fino alla fine dell'epoca.
- **Priorità dinamiche:** lo scheduler prevede un sistema di priorità che può essere modificato durante l'esecuzione del processo.
- **Real Time:** i processi vengono suddivisi in real-time e non real-time.

- **Processi IO-Intensive:** i processi che effettuano molte operazioni di input/output vengono favoriti. Ciò consente di avere un sistema apparentemente più reattivo.
- **Modifica delle priorità:** è possibile modificare o impostare manualmente le priorità dei processi utilizzando le opportune system call.

## 5.3 Scheduling

La priorità di un processo in Linux è basata sul suo quanto di tempo rimanente (memorizzato nel campo "counter" del processo). Quando un processo termina il suo quanto di tempo, non può più essere eseguito. Quando tutti i processi in esecuzione hanno counter = 0, si verifica la fine di un'epoca. Nell'epoca successiva, viene assegnato un nuovo quanto di tempo a tutti i processi. In questo modo, i processi che erano in attesa durante l'epoca precedente (quindi con counter > 0) vengono favoriti nella successiva epoca. I processi che eseguono operazioni di input/output intensive non sono necessariamente considerati interattivi con l'utente.

I processi figli ricevono la metà della priorità del processo padre.

Le priorità possono essere di due tipi:

- **Statiche:** sono fisse e vengono assegnate ai processi in tempo reale (real-time). Gli altri processi non possono avere una priorità superiore a quella dei processi real-time.
- **Dinamiche:** sono assegnate agli altri tipi di processi e possono variare nel tempo.

Le politiche di scheduling possono essere diverse a seconda del tipo di processo:

- **Processi real-time:**
  - *FIFO (First-In-First-Out):* i processi vengono eseguiti in base all'ordine di arrivo.
  - *Round Robin:* ogni processo riceve un certo quantum di tempo di esecuzione. Una volta esaurito il quantum, il processo viene sospeso e un altro processo viene eseguito. Al termine di tutti i processi, il quantum di tempo viene ridistribuito.
- **Altri processi:**
  - *Other:* è la politica di scheduling standard in Linux, basata sul valore del campo "counter" e sulle epoche.
  - *Yield:* un processo può richiedere di non essere eseguito per un determinato periodo di tempo.

## 5.4 Scheduler fino alla versione 2.4.xx

Lo scheduler operava attraverso due funzioni:

- *Goodness():* valutava la "bontà" di ogni processo esaminando il Linux Process Descriptor del processo in esame e del processo che ha appena terminato l'esecuzione.
  - Per i processi real-time, la funzione restituiva il valore massimo.
  - Per i processi con counter > 0, la funzione restituiva un valore minore rispetto ai processi real-time ma maggiore rispetto agli altri.
  - Per i processi con counter = 0, la funzione restituiva zero.
  - Se il processo in esame condivideva memoria con il processo che aveva appena eseguito, gli veniva assegnato un bonus per evitare la sostituzione e il costo di caricamento della memoria.
- *Schedule():* chiamata nei punti di scheduling per selezionare il processo da eseguire confrontando i valori restituiti dalla funzione Goodness().
  - Se il processo precedente è entrato in attesa, viene inserito nella coda corrispondente.

  - Vengono valutati i valori restituiti dalla Goodness() per ogni processo e viene scelto il processo migliore.
  - Se tutti i valori sono zero, viene conclusa un'epoca e tutti i processi vengono aggiornati con un nuovo quanto di tempo calcolato come quanto_rimanente / 2 + priorità_originale. In questo modo, i processi con counter > 0 vengono leggermente favoriti nella prossima epoca, senza mai superare i processi real-time.

È importante notare che è possibile modificare manualmente la priorità di un processo rispetto alla priorità standard utilizzando le funzioni *nice()* e *set_priority()*. È inoltre possibile impostare una politica di scheduling specifica utilizzando *sched_set_scheduling()*.
Tuttavia, questo approccio presentava alcuni problemi:

- Non era scalabile all'aumentare del numero di processi, poiché era necessario valutare tutti i processi e aggiornarli al termine di ogni epoca.
- Non distingueva tra processi interattivi IO-intensive e processi non interattivi IO-intensive.

## 5.5 Scheduler dalla versione 2.6.xx

Nelle versioni più recenti del kernel Linux, è stato introdotto un nuovo approccio per migliorare le prestazioni dello scheduler.

**Cambio di epoca:** Per migliorare le prestazioni del cambio di epoca, sono state introdotte due code di processi pronti: "active" per i processi con counter > 0 e "expired" per i processi con counter = 0. Quando un processo esaurisce il suo quanto di tempo, viene spostato dalla coda "active" alla coda "expired". Alla fine dell'epoca, le due code vengono scambiate, modificando il puntatore che punta alla coda dei processi pronti.

**Scheduling:** Per migliorare le prestazioni durante lo scheduling, le due liste sono state implementate come array di 256 liste, ognuna con una priorità decrescente. In questo modo, i processi vengono mappati su una tabella di bit in cui ogni bit corrisponde a un processo pronto con counter > 0. La scelta del processo da eseguire diventa molto semplice, poiché viene selezionato il primo bit uguale a 1 nella tabella di bit, corrispondente al primo processo nella lista con la priorità più alta tra quelli pronti ad eseguire. Questo permette di effettuare la scelta in un tempo indipendente dal numero di processi, ma dipendente solo dalla dimensione fissa della tabella.

## 5.6 Scheduler in sistemi SMP

Nei sistemi multi-core (SMP), lo scheduler Linux tiene conto dell'affinità dei processi con le CPU. L'affinità di un processo si sviluppa quando viene eseguito sulla cache di una CPU. Maggiore è la dimensione della cache, maggiore sarà l'affinità. Linux evita di spostare i processi da una CPU all'altra se il tempo richiesto per ricaricare la cache è maggiore del tempo di esecuzione del processo stesso. Altrimenti, si avrebbe uno spreco di tempo di esecuzione solo per ricaricare la cache.

Il bilanciamento delle code dei processi viene effettuato periodicamente in modo forzato se una coda rimane vuota. Durante il bilanciamento, viene selezionato il gruppo di processori ritenuto più occupato e all'interno di esso viene scelta la coda più carica. I processi vengono quindi migrati dalla coda vuota o dalle code meno cariche verso la coda più carica.

In conclusione, lo scheduler in Linux gestisce il scheduling dei processi attraverso politiche di priorità dinamiche, time-sharing e real-time. Il suo funzionamento è stato migliorato nel corso delle diverse versioni del kernel, introducendo nuovi approcci come l'utilizzo di code separate per i processi con counter > 0 e counter = 0 e l'implementazione di tabelle di bit per accelerare la scelta del processo da eseguire. Inoltre, nei sistemi multi-core, viene considerata l'affinità dei processi con le CPU e vengono effettuati bilanciamenti periodici delle code per garantire una distribuzione equa del carico di lavoro.

## 5.7 Il "Completely Fair Scheduler" (CFS)

A partire dalla versione 2.6.23, il kernel Linux ha incluso un nuovo scheduler, sostituendo il "O(1) Scheduler" precedentemente utilizzato. Il nuovo scheduler, chiamato "Completely Fair Scheduler" (CFS), rappresenta una significativa deviazione dal modello precedente.

Il CFS elimina molte delle caratteristiche tracciate dalle versioni precedenti: non ci sono più timeslice, non viene monitorato il tempo di sleep e non viene identificato il tipo di processo. Invece, il CFS cerca di modellare una "CPU multitasking ideale e precisa", in grado di eseguire contemporaneamente più processi, assegnando a ciascuno la stessa potenza di elaborazione. Ovviamente, questo è puramente teorico, quindi come possiamo modellarlo?

### 5.7.1 Implementazione del CFS

Si misura quanto tempo di esecuzione ha avuto ogni task e si cerca di garantire che ognuno riceva la sua giusta quota di tempo. Questo viene memorizzato nella variabile *vruntime* per ogni task, registrata al livello dei nanosecondi. Un valore *vruntime* più basso indica che il task ha avuto meno tempo per l'elaborazione e quindi ha più bisogno del processore. Inoltre, anziché utilizzare una coda, il CFS utilizza un Red-Black Tree per memorizzare, ordinare e schedulare i task.

### 5.7.2 Red-Black Trees

Un Red-Black Tree è un albero di ricerca binario in cui il sottoalbero sinistro contiene solo chiavi inferiori alla chiave del nodo e il sottoalbero destro contiene chiavi maggiori o uguali ad essa. Un Red-Black Tree ha ulteriori restrizioni: il percorso radice-foglia più lungo è al massimo il doppio del percorso radice-foglia più corto. Questo limite sull'altezza rende gli alberi Red-Black più efficienti dei normali alberi binari di ricerca. Le operazioni hanno complessità O(log n).

### 5.7.3 L'Albero CFS

La chiave per ogni nodo è il *vruntime* del task corrispondente. Per selezionare il prossimo task da eseguire, basta prendere il nodo più a sinistra dell'albero.

Il CFS ha tre strutture dati principali:

- **task_struct**: è l'entità di livello superiore, contiene informazioni come le priorità dei task, la classe di scheduling e la struttura *sched_entity*.
- **sched_entity**: include un nodo per l'albero Red-Black e la statistica *vruntime*, tra le altre informazioni.
- **cfs_rq**: contiene il nodo radice e il gruppo di task.

**Priorità e altro** Sebbene il CFS non utilizzi direttamente le priorità o le code prioritarie, le utilizza per modulare l'accumulo di *vruntime*. Un task con priorità più alta accumulerà *vruntime* più lentamente, poiché richiede più tempo di CPU. Allo stesso modo, un task con priorità bassa vedrà incrementare il proprio *vruntime* più velocemente, causando la sua prelazione anticipata.

## 5.8 Politiche di scheduling

- **SCHED_NORMAL** (tradizionalmente chiamato SCHED_OTHER): la politica di scheduling utilizzata per i task regolari.
- **SCHED_BATCH**: non preclude con la stessa frequenza dei task regolari, consentendo ai task di eseguire per un periodo più lungo e sfruttare meglio le cache, ma a costo della reattività.
- **SCHED_IDLE**: ancora più debole di "nice 19".
- **SCHED_FIFO** e **SCHED_RR** come nella precedente implementazione.

### 5.8.1 Preemption

Il tempo di prelazione è variabile, a seconda delle priorità e del tempo di esecuzione effettivo. Un task viene eseguito per un po' di tempo e l'utilizzo della CPU del task viene aggiunto a $p- > se.vruntime$. Una volta che $p- > se.vruntime$ diventa sufficientemente alto da far sì che un altro task diventi il "task più a sinistra" dell'albero ordinato per tempo, viene selezionato il nuovo task più a sinistra e viene preluso il task corrente.

### 5.8.2 Parametri dello scheduler

- **sched_child_runs_first**: quando si imposta questo parametro a 1, un nuovo task figlio viene eseguito prima che il genitore continui l'esecuzione.
- **sched_migration_cost**: quantità di tempo trascorsa dall'ultima esecuzione affinché un task venga considerato "cache hot" nelle decisioni di migrazione. Il valore predefinito è 500.000 ns.
- **sched_latency_ns**: latenza di prelazione mirata per i task CPU-bound. Aumentare questa variabile aumenta il timeslice di un task CPU-bound. Il valore predefinito è 20.000.000 ns.
- **sched_min_granularity_ns**: granularità di prelazione minima per i task CPU-bound. Il valore predefinito è 4.000.000 ns.
- **sched_wakeup_granularity_ns**: granularità di prelazione al risveglio. Aumentare questa variabile riduce la prelazione al risveglio, riducendo il disturbo dei task bound al calcolo. Il valore predefinito è 5.000.000 ns.
- **sched_wakeup_granularity_ns**: granularità di prelazione al risveglio. Aumentare questa variabile riduce la prelazione al risveglio, riducendo il disturbo dei task bound al calcolo. Il valore predefinito è 5.000.000 ns.

# Chapter 6

# Memoria principale

## 6.1 Organizzazione della memoria principale

Ogni processo in esecuzione ha il suo spazio in memoria, chiamato spazio di indirizzamento. Allo stesso modo, l'OS si alloca direttamente nei primi indirizzi della RAM, noti come indirizzi bassi.

**Binding:** Il binding è la relazione tra il codice e gli indirizzi di memoria su cui il processo agirà.

- **Binding a tempo di compilazione:** Genera codice assoluto che specifica gli indirizzi fisici esatti su cui operare. Il processo deve essere allocato in un'area specifica della memoria fisica, e se quegli indirizzi sono occupati, il processo non può essere allocato.
- **Binding a tempo di caricamento:** Genera codice rilocabile che utilizza indirizzi relativi. Quando il processo viene caricato, gli viene assegnato un indirizzo base. Il codice specifica tutti gli indirizzi come indirizzo_base + i. In questo modo, il processo può essere allocato ovunque, ma richiede memoria contigua.
- **Binding a tempo di esecuzione:** Utilizza indirizzi logici che vengono tradotti a tempo di esecuzione, consentendo la divisione dell'area di memoria di un processo.

**Indirizzi:**

- **Fisici:** Indirizzano la RAM fisicamente presente.
- **Logici:** Indirizzi generati dalla CPU (ad es. 32 bit, 64 bit).

La conversione degli indirizzi logici in indirizzi fisici viene effettuata a tempo di esecuzione dalla MMU. L'uso degli indirizzi logici consente la divisione dei processi in RAM e l'allocazione di un processo ovunque, poiché gli indirizzi vengono tradotti a tempo di esecuzione.

**Allocazione contigua:**

**Svantaggi:**

- Non consente l'allocazione dinamica (la dimensione non può cambiare).
- Alla fine dell'esecuzione, il processo lascia un buco in RAM che potrebbe non essere utilizzabile.

Gestione dei buchi in RAM all'arrivo di nuovi processi:

- **First-Fit:** Il processo viene allocato nel primo buco abbastanza grande.
- **Best-Fit:** Il processo viene allocato nel buco più piccolo in grado di ospitarlo (richiede di scorrere tutta la RAM, ma permette di risparmiare buchi grandi).
- **Worst-Fit:** Il processo viene allocato nel buco più grande (richiede di scorrere tutta la RAM, ma utilizza lo spazio rimanente, anche se può compromettere buchi che potrebbero essere utili).

**Frammentazione:**

- **Frammentazione esterna:** Si verifica quando ci sono buchi tra i processi nella memoria principale che non possono essere utilizzati per allocare altri processi.
- **Frammentazione interna:** Si verifica quando i blocchi di memoria sono troppo grandi rispetto alle dimensioni effettive dei processi che li occupano, generando spazio non utilizzato all'interno dei blocchi.

Nota: L'OS potrebbe eseguire periodicamente una deframmentazione, ma questo processo è lento e non consente l'utilizzo della macchina durante l'operazione.

## 6.2 Paginazione

La memoria fisica viene divisa in aree di dimensione costante chiamate **frame**. Allo stesso modo, lo spazio di indirizzamento logico viene diviso in **pagine**, della stessa dimensione dei frame. L'OS può allocare solo multipli di frame.

Le pagine e i frame sono indicizzati (ad esempio frame-1, frame-2, frame-n), e la corrispondenza tra una pagina e il suo frame fisico è garantita tramite la **tabella delle pagine**.

**Traduzione degli indirizzi:**

- La CPU genera un indirizzo logico costituito da: **PAGE_NUMBER + OFFSET**.
- Si cerca il frame corrispondente nella tabella delle pagine, indicizzata per PAGE_NUMBER.
- All'interno di ogni casella della tabella delle pagine, è presente il corrispondente FRAME_NUMBER.
- Si genera l'indirizzo fisico costituito da: **FRAME_NUMBER + OFFSET**.
- Si cerca in memoria il frame corrispondente e si seleziona il byte identificato dall'OFFSET.

Vantaggi della paginazione:

- Elimina la frammentazione esterna, poiché i blocchi di memoria sono di dimensione costante e occupano l'intera memoria fisica.
- Consente una traduzione rapida degli indirizzi logici in indirizzi fisici (O(1)), in quanto è sufficiente controllare il valore di una voce nella tabella delle pagine.

Svantaggi della paginazione:

- La tabella delle pagine è ospitata in RAM, quindi sono necessari due accessi in memoria per accedere ai dati (uno per consultare la tabella e uno per acquisire il dato).

**Translation Lookaside Buffer (TLB):** È una cache degli indirizzi che memorizza le traduzioni degli indirizzi più recentemente richiesti. Quando un processo richiede una pagina, viene prima cercata la traduzione nel TLB. Se presente, si accede direttamente alla memoria fisica all'indirizzo fisico corrispondente. In caso contrario, viene eseguito un accesso alla tabella delle pagine in memoria principale per ottenere la traduzione, e successivamente la traduzione viene memorizzata nel TLB per le successive richieste.

Il TLB consente di incrementare le prestazioni, aumentando la probabilità di trovare l'indirizzo tradotto nel TLB. È possibile migliorare le prestazioni del TLB in diversi modi:

- Aumentare la dimensione del TLB per memorizzare più traduzioni.
- Utilizzare pagine più grandi, in modo da ridurre il numero di traduzioni necessarie.

**Protezione di memoria:** Per implementare la protezione di memoria utilizzando la tabella delle pagine, è possibile aggiungere un bit aggiuntivo per ogni pagina che indichi se quell'indirizzo è valido o meno per il processo (bit di validità).

**Condivisione di memoria:** Per implementare la condivisione di memoria utilizzando la tabella delle pagine, è possibile far sì che gli indirizzi logici di due processi diversi puntino allo stesso indirizzo fisico. Solo quando uno dei due processi modifica quella parte di memoria, viene creata una copia separata (copy-on-write).

## 6.3 Struttura della tabella delle pagine

Per ridurre il peso della tabella delle pagine senza aumentarne la dimensione (e quindi senza aumentare la frammentazione interna), è possibile utilizzare diverse strategie:

### 6.3.1 Tabella gerarchica:

Consiste nell'utilizzare più livelli di tabelle. Viene creata una tabella di primo livello che non punta direttamente agli indirizzi fisici, ma ad altre tabelle di secondo livello che contengono gli indirizzi fisici.

**Vantaggi:** Le tabelle di livello interno non vengono allocate se la tabella di livello superiore ha un bit di validità impostato su "non valido".

**Svantaggi:** L'accesso a più livelli di tabelle richiede più accessi in memoria, eccessivi livelli di tabelle possono penalizzare le prestazioni.

### 6.3.2 Tabella hash:

Utilizza una funzione di hash per generare un valore univoco per ogni indirizzo logico. Viene creata una tabella delle pagine unica indicizzata da questi valori hash.

**Vantaggi:** Permette di risparmiare spazio con una sola tabella.

**Svantaggi:** Se lo spazio degli indirizzi è maggiore dello spazio fisico, possono verificarsi collisioni nella funzione di hash. Per risolvere le collisioni, le voci della tabella delle pagine contengono non solo un valore, ma una lista di elementi con coppie di valori: Numero di Pagina + Numero di Frame (per rendere univoci gli elementi). Per trovare un numero di pagina specifico, è necessario scorrere l'intera lista alla ricerca del valore corrispondente, il cui indice è il Numero di Frame.

### 6.3.3 Tabella invertita:

Consiste nell'utilizzare una singola tabella delle pagine indicizzata per il numero di frame.

**Vantaggi:** Riduce significativamente l'utilizzo di memoria, poiché la tabella è unica e dipende solo dalla dimensione della RAM (non dalla capacità dell'architettura).

**Svantaggi:**

- La ricerca di un numero di pagina richiede di scorrere l'intera tabella (O(n)). Si cerca la voce che contiene il numero di pagina desiderato, e l'indice di quella voce corrisponde al Numero di Frame.
- La condivisione di memoria non è più diretta come prima. Per implementarla, è necessario associare a ciascun frame una lista, se del caso, di elementi composti da Numero di Pagina + PID (per renderli univoci).

## 6.4 Segmentazione

I segmenti sono simili alle pagine, ma invece di avere una dimensione fissa, ciascuno ha la propria dimensione. Ciò consente di eliminare la frammentazione interna. Tuttavia, i segmenti devono essere contigui, il che può generare frammentazione esterna. La tabella dei segmenti è indicizzata per il numero di segmento e contiene l'indirizzo base e il limite del frame corrispondente in memoria fisica. L'indirizzo base indica l'inizio della porzione di memoria, mentre l'indirizzo limite indica la sua lunghezza.

Per sfruttare i vantaggi di paginazione e segmentazione, riducendo al minimo gli svantaggi, viene utilizzato un sistema che utilizza la segmentazione sopra la paginazione.

## 6.5 Intel Pentium

A partire dalla generazione Pentium, i processori Intel utilizzano sia la paginazione che la segmentazione. Consentono di utilizzare due tipologie di pagine: da 4KB o 4MB. La CPU genera indirizzi costituiti da un selettore (16 bit) e un OFFSET (32 bit). L'unità di gestione dei segmenti della MMU processa l'indirizzo per selezionare il segmento. I 32 bit dell'OFFSET vengono passati all'unità di gestione delle pagine, e a seconda della dimensione delle pagine, l'indirizzo assume una delle due forme:

**Pagina 4KB:**

- PNum1: 10 bit
- PNum2: 10 bit
- OFFSET: 12 bit (utilizzando la paginazione gerarchica)

**Pagina 4MB:**

- PNum: 10 bit
- OFFSET: 22 bit

L'indirizzo viene quindi individuato nella memoria fisica e viene selezionato il bit richiesto utilizzando l'OFFSET.

# Chapter 7

# Memoria virtuale

Quando un processo richiede una pagina, questa viene caricata dall'HDD in memoria RAM. Quando la RAM è satura, l'HDD viene utilizzato come supporto aggiuntivo. In tali circostanze, le pagine vengono copiate sull'HDD (swap-out) per fare spazio a nuove pagine che devono essere caricate nella RAM (swap-in). Tuttavia, il processo di copia e caricamento delle pagine dall'HDD è lento a causa della natura meccanica dell'HDD, che richiede tempo per spostare le testine di lettura/scrittura. Pertanto, è preferibile caricare in memoria RAM più pagine continue anziché una sola, poiché queste pagine, grazie al principio di località, potrebbero essere richieste dal processo, e la lentezza è principalmente dovuta allo spostamento delle testine.

## 7.1 Paginazione su richiesta

Quando un processo viene avviato, anziché caricare tutte le pagine inizialmente, di solito viene avviato con una memoria vuota. Ogni volta che il processo richiede una pagina che non è ancora presente in memoria, questa viene caricata. In questo modo si riduce significativamente l'occupazione di memoria, poiché solo le pagine effettivamente richieste vengono caricate.

Tuttavia, questo approccio ha uno svantaggio iniziale: la lentezza iniziale, in cui si verificano numerosi page fault. Per evitare questo problema, anziché caricare una singola pagina ogni volta, si effettua il caricamento di un blocco di pagine (cluster) che sono vicine a quella richiesta. In questo modo si riducono le possibilità di futuri page fault, poiché le pagine adiacenti a quella richiesta potrebbero essere richieste in modo sequenziale.

### 7.1.1 Page Fault

Durante l'esecuzione di un processo, può verificarsi un page fault quando il processo richiede l'accesso a una pagina che non è attualmente in memoria. Il page fault può essere causato da due situazioni:

- Se l'indirizzo richiesto è invalido, viene generato un errore.
- Se l'indirizzo corrisponde a una pagina valida ma non presente in memoria, si verifica un page fault.

In caso di page fault, il sistema operativo prende il controllo, cerca un frame libero nella RAM e richiede al dispositivo di archiviazione (HDD) di caricare la pagina richiesta. Se non ci sono frame liberi, è necessario eseguire lo swap-out di un frame occupato per fare spazio alla nuova pagina.

Durante l'attesa dell'HDD, viene generato un punto di scheduling, il sistema operativo può eseguire altri processi in modo da massimizzare l'utilizzo della CPU. Una volta ricevuta la pagina richiesta e caricata in memoria, il sistema operativo dà la precedenza al processo corrispondente, che riprende l'esecuzione dall'istruzione che ha generato il page fault.

Per migliorare le prestazioni e ridurre la possibilità di page fault, possono essere adottate le seguenti strategie:

- Aumentare la quanttà di memoria RAM: aumentare la capacità della RAM può ridurre la probabilità di saturazione e il conseguente swap-out delle pagine.

- Ridurre il numero di processi attivi: mettere in uno stato di sospensione (ibernazione) alcuni processi può liberare spazio per gli altri processi attivi.

- Utilizzare algoritmi di rimpiazzamento efficienti: scegliere algoritmi intelligenti per determinare quali pagine devono essere swap-out in caso di necessità. Questo può aiutare a ridurre la frequenza dei page fault.

## 7.2 Rimpiazzamento delle pagine

Quando viene richiesto di allocare un frame di memoria ma la RAM è completamente occupata, è necessario effettuare lo swap-out di un frame attualmente in uso (frame vittima). Per gestire il rimpiazzamento delle pagine, possono essere utilizzate diverse tecniche e strategie.

- **Dirty-Bit**: Questa tecnica prevede l'utilizzo di un bit aggiuntivo nella tabella delle pagine per distinguere le pagine che sono state modificate (dirty) da quelle che sono rimaste invariate rispetto all'originale. Le pagine invariate non necessitano di essere riscritte sul disco durante lo swap-out, rendendole le scelte migliori per essere swappate.

- **Bit di InterLock**: Questa tecnica prevede l'utilizzo di un bit nella tabella delle pagine che impedisce lo swap-out di una pagina specifica. Questo può essere utile quando si desidera mantenere una pagina specifica in memoria per motivi particolari.

- **Algoritmi di rimpiazzamento**: Per confrontare diversi algoritmi di rimpiazzamento, è possibile farli operare su una sequenza di pagine effettivamente richieste da un sistema e verificare quale algoritmo genera il minor numero di page fault.

Alcuni algoritmi comuni di rimpiazzamento delle pagine includono:

- **Ottimo (Optimal)**: Questo algoritmo ideale prevede di swappare la pagina che verrà utilizzata più lontano nel tempo. Tuttavia, non può essere implementato in pratica poiché richiede di conoscere in anticipo quali pagine verranno richieste.

- **FIFO (First-In-First-Out)**: In questo algoritmo, la pagina caricata per prima viene swappata senza considerare quando è stata utilizzata. Tuttavia, l'algoritmo FIFO è molto semplice e soffre di un'anomalia: all'aumentare della capacità della RAM (entro un certo intervallo limitato), il numero di page fault può aumentare anziché diminuire.

- **LRU (Least Recently Used)**: Questo algoritmo parte dal principio che un processo tende a utilizzare le pagine utilizzate di recente e, quindi, rimuove le pagine utilizzate meno recentemente. L'algoritmo LRU può essere costoso da implementare poiché richiede il monitoraggio dell'ultimo utilizzo di ogni pagina e l'ordinamento delle pagine

- **LRU a Pila**: Questa variante dell'algoritmo LRU utilizza una struttura dati a pila. Le nuove pagine vengono inserite in cima alla pila, mentre una pagina già presente che viene utilizzata di nuovo viene spostata in cima. La rimozione avviene sempre dal fondo della pila, senza bisogno di tenere traccia di informazioni aggiuntive o scorrere l'intera lista. Tuttavia, questo metodo richiede frequenti modifiche ai puntatori della pila, che possono essere dispendiose in termini di tempo.

- **LRU approssimato**: Questa variante semplificata dell'algoritmo LRU utilizza un solo bit per ogni pagina (0: pagina non utilizzata di recente, 1: pagina utilizzata di recente). In questo modo, la prima pagina con bit a 0 viene swappata. Periodicamente, tutti i bit vengono azzerati. Sebbene sia meno accurato dell'LRC completo, è più semplice da implementare e richiede meno memoria per tenere traccia delle informazioni di utilizzo.

- **LRU approssimato con contatore**: In questa variante, invece di utilizzare un solo bit, viene utilizzato un contatore che viene incrementato ogni volta che la pagina viene utilizzata. Viene swappata la pagina con il contatore più basso, e periodicamente tutti i contatori vengono azzerati. Questo metodo offre una maggiore precisione rispetto all'LRC approssimato, ma richiede più memoria per memorizzare i contatori aggiuntivi.

- **LRU con approssimazione a seconda chance**: Questo algoritmo utilizza un singolo bit e un puntatore per scorrere la lista delle pagine. Viene swappata la prima pagina con bit 0 che viene incontrata, mentre le pagine con bit 1 vengono "salvate" e il loro bit viene azzerato. Se una pagina salvata non viene utilizzata durante il secondo passaggio, viene swappata. Questo metodo offre un buon compromesso tra accuratezza e complessità.

## 7.3 Allocazione dei frame

Quando un processo genera un page fault e tutti i frame dedicati ad esso sono occupati, ci sono due modalità di rimpiazzamento tra cui scegliere:

- **Rimpiazzamento locale**: in questa modalità, il processo può rimpiazzare solo i propri frame per evitare di danneggiare o interferire con altri processi.

- **Rimpiazzamento globale**: in questa modalità, il processo può rimpiazzare anche i frame di altri processi se necessario. Questo approccio offre una maggiore flessibilità nella gestione della memoria, ma può richiedere un'adeguata sincronizzazione per evitare conflitti o danni ai processi interessati.

Al momento del lancio di un processo, è necessario decidere quanti frame allocare per esso. Ci sono diverse strategie di allocazione della memoria:

- **Allocazione fissa**: in questo approccio, viene asse gnato uno spazio di memoria fisso per ogni processo, indipendentemente dalle sue reali esigenze. Ci sono due varianti comuni:

- **Allocazione equa**: ogni processo riceve la stessa quantità di memoria.

- **Allocazione proporzionale**: i processi che derivano da eseguibili di dimensioni maggiori ricevono più spazio di memoria rispetto ai processi più piccoli. Questo approccio tiene conto delle differenze di dimensione dei processi e alloca proporzionalmente più memoria ai processi più grandi.

- **Allocazione prioritaria**: in questa strategia, un processo può rimpiazzare i propri frame o i frame di processi con priorità inferiore. Ciò significa che i processi con priorità più alta hanno la priorità nell'allocazione della memoria. Questo approccio mira a garantire che i processi critici o prioritari abbiano sempre abbastanza memoria a loro disposizione.

È importante notare che è impossibile conoscere esattamente quanta memoria un processo richiederà effettivamente, quindi l'allocazione globale viene spesso utilizzata per garantire flessibilità. Tuttavia, è necessario evitare che i processi si danneggino reciprocamente o monopolizzino la memoria. Questo richiede una gestione attenta della memoria virtuale e algoritmi di rimpiazzamento efficienti.

## 7.4 Trashing

Il trashing si verifica quando l'uso insufficiente della CPU porta il Long-Term Scheduler a caricare continuamente nuovi processi in memoria. Se i processi si rubano costantemente le pagine a vicenda, il risultato è un'attività inefficace della CPU e un eccessivo carico di rimpiazzamento delle pagine. Questo crea un circolo vizioso in cui la macchina esegue principalmente operazioni di rimpiazzamento delle pagine anziché eseguire i processi effettivi.

Per affrontare il problema del trashing, possono essere adottate alcune soluzioni:

- **Working set**: questa tecnica prevede di stabilire un intervallo di tempo e individuare il numero di pagine che potrebbero essere richieste durante quell'intervallo di tempo. Queste pagine costituiscono il "working set" e potrebbero essere richieste a causa del principio di località. Se la somma dei working set di tutti i processi supera la memoria disponibile, è necessario effettuare uno swap-out di alcuni processi per liberare spazio.
- **Valutazione del tasso di page fault**: questa tecnica prevede di stabilire delle soglie per il tasso di page fault. Se il tasso di page fault rimane basso, non è necessaria alcuna azione correttiva. Tuttavia, se il tasso di page fault aumenta oltre una determinata soglia, è possibile avviare uno swap-out preventivo di alcuni processi per anticipare il problema e liberare memoria.

## 7.5 File mappati in memoria

I file mappati in memoria consentono di caricare un file aperto direttamente in memoria RAM, dove avvengono le operazioni di lettura e scrittura. Ciò elimina la necessità di operazioni di lettura/scrittura dirette su disco per ogni accesso al file, migliorando le prestazioni complessive.

Alcuni vantaggi dell'utilizzo di file mappati in memoria sono:

- **Evitare il write-back**: se un file non viene modificato, è possibile evitare l'operazione di scrittura sul disco utilizzando il bit di "dirty" (sporco). Se il bit di "dirty" di una pagina è a 0, significa che la pagina non è stata modificata e quindi non è necessario eseguire il write-back.
- **Caricare solo le parti richieste**: grazie alla tecnica della paginazione su richiesta, è possibile caricare in memoria solo le parti del file richieste dal processo in esecuzione. Ciò riduce l'occupazione della memoria e ottimizza l'utilizzo delle risorse.
- **Condivisione di file**: è possibile aprire lo stesso file in modalità condivisa per più processi. Ciò consente ai processi di accedere contemporaneamente al file senza creare duplicati separati in memoria. È importante notare che alcuni sistemi operativi possono impedire l'apertura di un file in modalità di scrittura se il file è già aperto da un altro processo, mentre altri consentono sovrascritture.

Un esempio di utilizzo dei file mappati in memoria è l'implementazione delle aree di memoria condivisa in Windows.

## 7.6 Gestione della memoria per il sistema operativo

A differenza dei processi, la memoria del sistema operativo deve essere allocata in modo contiguo per garantire prestazioni ottimali e ridurre i tempi di esecuzione delle operazioni del sistema operativo stesso, che spesso richiedono un accesso rapido alla memoria.

Tuttavia, il sistema operativo ha il vantaggio di dover allocare blocchi di memoria di dimensioni note, ad esempio per i blocchi di controllo dei processi (PCB) che hanno dimensioni fisse per tutti i processi.

Esistono diverse tecniche di gestione della memoria per il sistema operativo, tra cui il metodo Buddy e l'allocazione a lastre.

### 7.6.1 Metodo Buddy

Il metodo Buddy prevede l'utilizzo di pagine di dimensione potenza di 2 ottenute da una singola area di memoria contigua. Quando il sistema operativo deve allocare memoria, divide l'area di memoria disponibile in due parti uguali e quindi continua a dividere in modo ricorsivo fino a ottenere la dimensione minima di allocazione.

Quando viene richiesto il rilascio di un'area di memoria, il sistema operativo controlla se l'area "gemella"

(buddy) è libera. In caso affermativo, le due aree vengono unite per formare un blocco di dimensione maggiore.

Questo metodo è efficiente in termini di frammentazione interna, ma può causare spreco di memoria a causa dei requisiti di dimensione potenza di 2.

### 7.6.2 Allocazione a lastre

L'allocazione a lastre prevede la preallocazione di memoria per diverse categorie di risorse necessarie. Vengono allocate lastre di memoria contigue per ciascuna categoria, come ad esempio una lastra per i semafori, una per i PCB, ecc. All'interno di ogni lastra vengono quindi allocati i blocchi di memoria corrispondenti (ad esempio PCB o semafori).

Le lastre di memoria sono di dimensione multipla dei blocchi da allocare, in modo da evitare sprechi di memoria. Se la memoria è completamente occupata, è possibile allocare nuove lastre in qualsiasi posizione della memoria, purché siano contigue all'interno della lastra.

Quando un blocco di memoria può essere liberato, viene semplicemente sovrascritto con nuovi dati. Se una lastra è completamente vuota, può essere disallocata.

Questa tecnica permette di preallocare la memoria per le diverse categorie di risorse, evitando così la necessità di allocare dinamicamente ogni volta che un processo richiede una risorsa. Tuttavia, è importante gestire adeguatamente la dimensione delle lastre e la capacità complessiva della memoria per evitare sprechi.

In conclusione, la gestione della memoria virtuale, il rimpiazzamento delle pagine, l'allocazione dei frame, l'uso dei file mappati in memoria e la gestione della memoria per il sistema operativo sono tutti aspetti cruciali della gestione della memoria in un sistema informatico. Una corretta gestione della memoria può migliorare le prestazioni complessive del sistema, garantendo un utilizzo efficiente e ottimizzato delle risorse disponibili.

# Chapter 8

# File System

## 8.1 Concetto di file system

A differenza della RAM, il File System deve essere interfacciato all'utente per consentirne l'operatività. Ogni file è caratterizzato da attributi che vengono mantenuti nel File Descriptor, simile ai Process Control Blocks per i processi. I principali attributi di un file sono:

- **Nome**: il nome del file, compresa l'estensione.
- **Identificatore**: un identificatore univoco del file utilizzato dal sistema operativo.
- **Tipo**: il tipo di file effettivo, indipendentemente dall'estensione.
- **Posizione**: un puntatore alla posizione del file sul disco.
- **Dimensione**: la dimensione del file.
- **Informazioni di protezione**: informazioni riguardanti i permessi di accesso al file.
- **Informazioni statistiche**: informazioni sul proprietario del file, data di creazione, ecc.

Il file system deve essere progettato in modo da supportare qualsiasi tipo o struttura di file, lasciando al sistema operativo il compito di gestirli.

### 8.1.1 Apertura di un file

Per aprire un file, vengono eseguite le seguenti operazioni:

- Ricerca del File Descriptor corrispondente al file.
- Verifica dei permessi di accesso al file.
- Copia del file nella memoria RAM.
- Gestione del numero di programmi che accedono al file:
  - Impedisce l'eliminazione di un file aperto.
  - In alcuni sistemi operativi, impedisce l'apertura simultanea di un file in scrittura da parte di due programmi.
- Chiusura del file dopo aver effettuato il write-back dei dati modificati.

### 8.1.2 Tipi di accesso

Esistono due tipi di accesso ai file:

- **Accesso sequenziale**: i dati vengono letti o scritti byte per byte, seguendo l'ordine in cui sono memorizzati nel file.
- **Accesso diretto**: permette l'accesso diretto a un blocco specifico del file utilizzando una combinazione di "indirizzo del file" e "indice del blocco". L'accesso diretto è più efficiente, ma richiede l'allocazione contigua dei blocchi.

## 8.2 Directory

Una directory è una struttura che consente di accedere ai File Descriptor. Esistono diversi tipi di directory:

- **Directory a singolo livello**: una lista di File Descriptor che puntano direttamente ai file.
- **Directory a due livelli**: utilizzata per la multi-utenza, consiste in una lista di utenti, ciascuno con una propria lista di file. I file sono protetti e non condivisibili tra gli utenti.
- **Directory ad albero**: utilizza nodi che puntano a blocchi di altri nodi. Viene introdotto un bit speciale per ogni nodo per distinguere tra nodi che puntano direttamente ai file e nodi che puntano a blocchi di altri nodi. Questa struttura ad albero rende le operazioni di ricerca più efficienti, evitando la scansione di tutta la lista. Introduce anche il concetto di percorso e di directory corrente.
- **Directory a grafo aciclico**: permette la condivisione di file, in quanto uno stesso file può essere raggiunto da più percorsi. Tuttavia, il grafo deve essere aciclico per evitare loop durante le operazioni di attraversamento. Per garantire ciò, è impedito di creare collegamenti diretti (link hard) su sottodirectory, consentendo solo link hard su file. I link simbolici (identificati da uno speciale "bit" uguale a 2) vengono ignorati durante l'attraversamento.

I link hard sono puntatori diretti a un file, consentendo la condivisione di file tra più percorsi. L'eliminazione di tutti i link hard che puntano a un file comporta la "eliminazione" del file (in realtà viene solo marcato come "libero").

link simbolici sono puntatori a un file o a una sottodirectory e non hanno lo stesso peso dei link hard. La rimozione di un link simbolico non ha effetti sul file, ma la rimozione di un file (e quindi dei link hard che lo puntano) rende il link simbolico "rotto". I link simbolici non vengono attraversati durante le routine di attraversamento dei file.

## 8.3 Montaggio, condivisione e protezione

Il montaggio è il processo di associazione di un file system alla macchina e di caricamento dell'interfaccia corrispondente. Prima di utilizzare un file system, è necessario montarlo e scegliere un punto di mount, cioè una directory da cui accedere al file system. Una volta montato, il file system sarà associato a quel percorso e l'accesso avverrà tramite quel punto, nascondendo il percorso precedente ospitato nella directory di mount.

La condivisione dei file e dei file system tra utenti è una necessità comune, specialmente quando si lavora in rete. È possibile condividere file system completi in rete e operare su di essi come se fossero locali (ad esempio, file system come NFS o CIFS sono appositamente progettati per essere utilizzati come file system remoti).

La consistenza del file system riguarda la gestione degli accessi multipli a uno stesso file da parte di più utenti e le modifiche effettuate. Esistono diverse semantiche per la gestione di queste situazioni:

- **Semantiche di Andrew FileSystem**: basate su sessioni, consentono a più utenti di operare contemporaneamente su un file o un file system. Tuttavia, ogni utente deve mantenere una copia in cache delle modifiche locali e caricarle tutte insieme alla fine delle modifiche. Ciò può causare problemi di sincronizzazione quando si sovrascrivono parti comuni. Le modifiche apportate alla cache non vengono salvate in tempo reale, il che può portare a problemi di sovrascrittura.

- **Semantiche di UNIX FileSystem**: le modifiche vengono apportate in tempo reale direttamente sul file originale e sono immediatamente visibili a tutti gli utenti. Tuttavia, richiede l'implementazione di un sistema di sincronizzazione per gestire l'accesso concorrente (ad esempio, concedendo l'accesso in scrittura a un utente alla volta). Questo approccio è più efficiente rispetto alle semantiche di Andrew, ma può essere più lento a causa della sincronizzazione e delle prestazioni della rete.

- **Semantiche di immutabilità**: sono molto restrittive e non consentono la modifica di un file condiviso una volta che è stato caricato (ad esempio, nei servizi di file hosting).

La protezione dei file riguarda il controllo degli accessi e delle operazioni che gli utenti possono effettuare su un file. Nei sistemi Windows, questo viene realizzato attraverso liste di controllo degli accessi associate a ogni file o directory. Ogni utente ha una voce nella lista con i relativi permessi. Nei sistemi UNIX, vengono utilizzati 9 bit per ogni file (permessi di lettura, scrittura ed esecuzione per proprietario, gruppo e altri). Questo approccio consente di avere un sistema di file più leggero in termini di memoria (9 bit costanti per ogni file anziché una lista che può crescere notevolmente), e la definizione dei gruppi viene gestita a livello di sistema operativo, non nel file system. Quando è necessario aggiungere un nuovo utente, basta aggiungerlo al gruppo appropriato nel sistema operativo e automaticamente eredita i permessi del gruppo per quel file.

## 8.4 Implementazione del file system

Il sistema operativo utilizza un livello di API di file system virtuale per comunicare con i diversi tipi di file system. Questo livello fornisce un'interfaccia standard per il sistema operativo, che può quindi dialogare con il file system fisico tramite il file system virtuale.

## 8.5 Metodi di allocazione

Ogni Inode (o File Descriptor) contiene un campo "puntatore al file" che punta al file effettivo in memoria. Ogni file system è diviso logicamente in settori o blocchi.

### 8.5.1 Allocazione contigua

Prevede che ogni file venga allocato in blocchi contigui. Il puntatore punta solo al primo blocco del file, e la dimensione del file viene utilizzata per determinare dove termina il file. Questo metodo consente l'accesso diretto al file (se si conosce la posizione relativa del blocco).
Svantaggi dell'allocazione contigua:

- *Frammentazione esterna*: poiché i file devono essere allocati in modo contiguo, possono rimanere spazi inutilizzati nel file system.

- *Crescita del file*: se il blocco successivo è occupato, il file deve essere spostato in una nuova serie di blocchi liberi, il che comporta una lenta operazione di spostamento del file.

- *Ripristino*: eliminare il File Descriptor comporta la perdita del file. Il puntatore nel File Descriptor è l'unico modo per accedere al file, che pur essendo ancora presente e contrassegnato come "libero", diventa irrecuperabile.

### 8.5.2 Allocazione linkata

Ogni blocco contiene dati e un puntatore al blocco successivo.
Vantaggi dell'allocazione linkata:

- Risolve i problemi di crescita e frammentazione.
- È possibile ripristinare un file scansionando tutti i blocchi e ricollegandoli se non sono ancora stati sovrascritti.

Svantaggi dell'allocazione linkata:

- Spreco di memoria per gli indirizzi.
- Non consente l'accesso diretto (è necessario scorrere tutti i blocchi).
- Danneggiare uno dei collegamenti comporta la perdita di tutti i dati.
- È lento perché i blocchi sono dispersi sul disco (la deframmentazione serve a posizionare i blocchi di uno stesso file più vicini o possibilmente sullo stesso cilindro).

Nota: Esiste una variante di allocazione contigua chiamata allocazione contigua patchata, simile all'allocazione linkata. In questo caso, la parte finale dell'ultimo blocco della serie contigua viene utilizzata per collegare la serie successiva di blocchi contigui.

**FAT**: FAT (File Allocation Table) utilizza una tabella di allocazione dei file posizionata nella prima parte del disco. Ogni entry della tabella contiene l'indirizzo del blocco e l'indice dell'entry del prossimo blocco. Il File Descriptor punta all'entry che contiene l'indirizzo del primo blocco del file. Questo consente di seguire tutto il file senza dover leggere i blocchi, fornendo un accesso pseudo-diretto. Tuttavia, poiché la FAT deve essere posizionata in modo contiguo nei primi indirizzi del disco, le testine devono spostarsi continuamente tra la FAT e la parte dati del file system, il che rallenta le prestazioni e può causare danni al disco.

### 8.5.3 Allocazione indicizzata

Prevede l'utilizzo di uno o più blocchi dedicati agli indici. Il File Descriptor punta al blocco di indici, all'interno del quale sono presenti gli indici dei blocchi che ospitano il file. Se un singolo blocco di indici non è sufficiente, l'ultimo indice nel primo blocco funge da indice per il secondo blocco di indici, e così via.
Vantaggi dell'allocazione indicizzata:

- Consente il ripristino del file scansionando tutti i nodi, identificando i nodi indice.
- Consente la creazione di una struttura gerarchica di nodi indice, in modo che i puntatori di un nodo indice di primo livello puntino a nodi indice di secondo livello, e così via. Questo consente di ottenere molto spazio di indirizzamento con meno blocchi indice.

Svantaggi dell'allocazione indicizzata:

- Spreco di memoria per gli indirizzi.
- Non consente l'accesso diretto (è necessario iniziare dall'inizio e scorrere tutti i blocchi).
- Danneggiare un blocco di indici comporta la perdita del file.

L'Inode UNIX utilizza una combinazione di puntatori per gestire l'allocazione dei blocchi dei file. Dispone di 15 puntatori, di cui:

- 12 puntano direttamente ai blocchi dei dati (accesso diretto per i primi 12 blocchi).
- 1 punta a un blocco di indirizzi che punta ai blocchi dei dati.
- 1 punta a un blocco di indirizzi che punta a blocchi di indirizzi di secondo livello.
- 1 punta a un blocco di indirizzi che punta a blocchi di terzo livello, che a loro volta puntano ai blocchi dei dati.

# Chapter 9

# Dischi Magnetici

## 9.1 Struttura del disco

Un hard disk è costituito da più dischi coassiali ricoperti da sostanze ferromagnetiche. La lettura e la scrittura dei dati avvengono tramite bracci mossi da magneti, che valutano o modificano la polarizzazione della sostanza. Durante la lettura, la testina viene posizionata sulla posizione corretta lungo il raggio e attende l'arrivo della sezione richiesta. Il tempo di spostamento è proporzionale alla distanza che la testina deve percorrere ed è il fattore più significativo in termini di attesa. Pertanto, il disco può leggere più blocchi posizionati uno accanto all'altro sulla stessa traccia nello stesso tempo (quasi) impiegato per leggerne uno. Inoltre, una volta posizionata su un blocco, è possibile leggere tutti i blocchi presenti sullo stesso cilindro.

## 9.2 Performance dei dischi

Le performance di un disco dipendono da diversi fattori:

- **Velocità di rotazione**: più il disco ruota velocemente, meno tempo ci vuole perché il blocco richiesto passi sotto la testina.
- **Tempo di posizionamento delle testine**: il braccio è spostato tramite magneti e il tempo necessario è proporzionale alla distanza da percorrere.

Il sistema operativo può cercare di minimizzare gli spostamenti delle testine, servendo blocchi vicini a quelli attuali. Le richieste ricevute dall'OS devono essere ordinate in modo da migliorare le performance.

## 9.3 Algoritmi di scheduling

Gli algoritmi di scheduling sono utilizzati per minimizzare lo spostamento medio delle testine e, di conseguenza, ridurre i tempi di attesa durante l'accesso ai dischi. Alcuni degli algoritmi più comuni sono:

### 9.3.1 First Come First Served (FCFS)

L'algoritmo FCFS prevede di servire le richieste nell'ordine in cui l'OS le riceve. Tuttavia, questo algoritmo non minimizza i tempi di attesa, poiché le richieste sono completamente casuali. Di conseguenza, risulta poco efficiente in termini di prestazioni.

### 9.3.2 Shortest Seek Time First (SSTF)

L'algoritmo SSTF prevede di ordinare le richieste in base al minor spostamento richiesto dalla testina per raggiungere ciascuna richiesta. Questo algoritmo è più efficiente rispetto a FCFS, in quanto riduce i tempi di attesa. Tuttavia, può generare il fenomeno della starvation, in cui le richieste molto lontane potrebbero

essere continuamente ignorate a causa dell'arrivo di nuove richieste più vicine. Inoltre, l'algoritmo SSTF comporta cambi continui di direzione della testina, che possono causare un'usura eccessiva del disco.

### 9.3.3 Scan (o algoritmi dell'ascensore)

L'algoritmo di Scan, noto anche come algoritmo dell'ascensore, prevede che la testina si muova solo verso la fine del disco senza tornare indietro, servendo tutte le richieste presenti in quella direzione. Una volta raggiunta la fine, la testina si sposta in senso opposto e ripete il processo. Questo algoritmo evita il fenomeno della starvation e riduce i cambi di direzione della testina, migliorando l'usura del disco. Tuttavia, è meno efficiente dell'algoritmo SSTF in termini di tempi di attesa. Inoltre, dopo il passaggio della testina, si accumulano nuove richieste a metà disco e quando la testina torna indietro, deve servire queste richieste prima di quelle più vecchie che si trovano all'inizio del disco.

### 9.3.4 C-SCAN

L'algoritmo C-SCAN è simile all'algoritmo di Scan, ma una volta raggiunta la fine del disco, la testina torna rapidamente all'inizio senza servire le richieste presenti in quella posizione. Questo permette di ridurre notevolmente il tempo necessario per il ritorno, garantendo tempi di servizio più uniformi per le richieste.

### 9.3.5 SCAN-LOOK

Lo SCAN-LOOK è una variante dell'algoritmo di Scan e C-SCAN. Invece di arrivare fino all'inizio o alla fine del disco, la testina si ferma all'ultima o alla prima richiesta presente, evitando spostamenti inutili. Questo contribuisce a migliorare ulteriormente le prestazioni dell'algoritmo.

# Chapter 10

# Sistemi di input/output

## 10.1 Introduzione

Esistono moltissimi tipi di dispositivi di I/O, e per poterli gestire in modo efficiente, gli OS li raggruppano in macro-categorie. I dispositivi devono rispettare certi standard affinché l'OS possa dialogare correttamente con tutti i tipi di dispositivi. Inoltre, gli OS identificano delle porte standard che consentono di individuare immediatamente un determinato dispositivo.

## 10.2 Polling, interrupt e DMA

Per utilizzare un dispositivo, la CPU invia una richiesta al dispositivo stesso, che viene poi messa in coda e successivamente eseguita. Una volta completata l'esecuzione, il dispositivo deve comunicarlo alla CPU. Ci sono diverse metodologie per gestire questa comunicazione:

### 10.2.1 Polling

Con il metodo del polling, la CPU continua a interrogare il dispositivo fino a quando il dispositivo stesso non comunica la sua terminazione. Questo approccio è inefficiente perché richiede una busy-waiting, ovvero la CPU non può fare altro se non continuare a interrogare il dispositivo.

### 10.2.2 Interrupt

Con il metodo degli interrupt, la CPU invia la richiesta al dispositivo e poi ignora il dispositivo continuando a svolgere altre operazioni. Una volta che il dispositivo ha completato la richiesta, alza un segnale di interrupt che fa sì che la CPU interrompa il suo lavoro corrente e utilizzi ciò che il dispositivo ha fornito. Gli interrupt vengono gestiti tramite una tabella degli interrupt dedicata. Gli interrupt possono essere mascherabili o non mascherabili a seconda dell'urgenza che comportano. Questo metodo è più efficiente e consente di ottimizzare l'utilizzo della CPU, ma richiede la gestione dei segnali di interrupt.

### 10.2.3 DMA (Direct Memory Access)

DMA è un componente hardware che, ricevuta una richiesta dalla CPU, è in grado di dialogare direttamente con la memoria e i dispositivi di I/O per effettuare lo spostamento dei dati senza l'intervento della CPU. Quando è in funzione, il DMA occupa il bus dei dati, ma consente alla CPU di occuparsi di altre operazioni. Una volta completato il trasferimento dei dati, il DMA genera un interrupt che viene riconosciuto dalla CPU, che riprende il controllo.

## 10.3 Tipi di dispositivi

I dispositivi di I/O possono essere classificati in tre categorie principali:

### 10.3.1 Dispositivi a caratteri

I dispositivi a caratteri mettono a disposizione operazioni di lettura e scrittura (get e put) e operano su dati di piccole dimensioni. Sono più efficienti per manipolare quantità ridotte di dati. Tuttavia, a causa delle dimensioni limitate dei dati, hanno un tasso di trasferimento basso per dati di grandi dimensioni.

### 10.3.2 Dispositivi a blocchi

I dispositivi a blocchi mettono a disposizione operazioni di lettura, scrittura e spostamento all'interno di un blocco (seek). Consentono di leggere quantità di dati più grandi, che possono anche essere mappati nella memoria RAM per una maggiore efficienza. I dispositivi a blocchi hanno un tasso di trasferimento più elevato rispetto ai dispositivi a caratteri quando si operano su grandi quantità di dati. Tuttavia, sono più lenti nelle operazioni che coinvolgono dati di piccole dimensioni.

### 10.3.3 Dispositivi di rete

I dispositivi di rete sono simili ai dispositivi a blocchi in quanto lavorano su grandi quantità di dati. Tuttavia, hanno la peculiarità di richiedere un'interattività molto elevata per supportare le comunicazioni di rete.

## 10.4 Metodologie di comunicazione

Esistono diverse metodologie per la comunicazione con i dispositivi di I/O. Le principali sono:

### 10.4.1 Comunicazione bloccante

Nella comunicazione bloccante, il processo invocante viene sospeso fino al completamento della richiesta. Questo approccio è semplice da utilizzare e garantisce che le operazioni precedenti siano state completate prima di passare alle successive.

### 10.4.2 Comunicazione non bloccante

Nella comunicazione non bloccante, il processo invocante effettua la richiesta e può quindi svolgere altre attività senza attendere il completamento. Questo è tipico della programmazione multi-thread, in cui un thread viene creato per gestire una richiesta mentre il thread principale continua il proprio lavoro. Questo approccio è più efficiente ma richiede attenzione, poiché non si può fare affidamento immediato sui dati richiesti. Tuttavia, è possibile tentare di accedere ai dati già caricati dal processo invocato.

### 10.4.3 Comunicazione asincrona

Nella comunicazione asincrona, il processo non viene bloccato e i dati vengono passati solo quando l'operazione è stata completata. Questo approccio è particolarmente utile quando si lavora con operazioni lunghe e complesse che richiedono tempo per essere completate.

# Chapter 11

# Crittografia

## 11.1 Introduzione

La crittografia è una disciplina che si occupa delle tecniche per cifrare un messaggio in modo tale che solo il legittimo destinatario sia in grado di leggerlo. Per garantire l'efficacia della crittografia, sono necessari due requisiti fondamentali: la cifratura e decifratura dei messaggi deve essere ragionevolmente efficiente e deve essere "difficile" interpretare un messaggio cifrato da parte di chi non è autorizzato.

## 11.2 Principio di base

Il principio di base della crittografia consiste nell'utilizzare una procedura di cifratura per trasformare un messaggio "in chiaro" $M$ in un messaggio cifrato $M'$ utilizzando una chiave di cifratura $K_{enc}$. In seguito, è possibile ricavare $M$ da $M'$ mediante una procedura di decifratura utilizzando una chiave di decifratura $K_{dec}$.

## 11.3 Notazione

Nella crittografia si utilizzano le seguenti notazioni:

- $E(K_{enc}, M)$ rappresenta la funzione di cifratura che, dato un messaggio $M$ e una chiave di cifratura $K_{enc}$, restituisce il messaggio cifrato $M'$.
- $D(K_{dec}, M')$ rappresenta la funzione di decifratura che, dato un messaggio cifrato $M'$ e una chiave di decifratura $K_{dec}$, restituisce il messaggio "in chiaro" $M$.

È fondamentale che valga sempre la seguente relazione:

$$D(K_{dec}, E(K_{enc}, M)) = M$$

La sicurezza della crittografia risiede esclusivamente nelle chiavi crittografiche. È importante che chiunque non conosca la chiave non sia in grado di interpretare il messaggio cifrato, anche se conosce il metodo con cui il messaggio è stato cifrato. Non si deve fare affidamento sulla segretezza dell'algoritmo crittografico stesso (principio della "security by obscurity").

## 11.4 Sistemi crittografici

Esistono due tipologie principali di sistemi crittografici:

### 11.4.1 Crittografia a chiave segreta (simmetrica)

Nella crittografia a chiave segreta, si utilizza la stessa chiave sia per cifrare che per decifrare il messaggio. In altre parole, $K_{enc} = K_{dec}$. Un esempio di crittografia a chiave segreta è il cifrario di Cesare. Tuttavia, il cifrario di Cesare non è considerato sicuro.

### 11.4.2 Crittografia a chiave pubblica (asimmetrica)

Nella crittografia a chiave pubblica, le chiavi di cifratura e decifratura sono diverse. In altre parole, $K_{enc} \neq K_{dec}$. Un esempio di crittografia a chiave pubblica è il sistema RSA, che utilizza due chiavi distinte: una chiave pubblica per la cifratura e una chiave privata per la decifratura.

## 11.5 Algoritmi crittografici

### 11.5.1 DES (Data Encryption Standard)

Il DES è un algoritmo di crittografia sviluppato da IBM e adottato come standard dal governo degli Stati Uniti nel 1977. Utilizza una chiave di lunghezza di 56 bit e divide il messaggio in blocchi di 64 bit che vengono cifrati singolarmente. Nonostante il grande numero di chiavi possibili (circa $7.2 \times 10^{16}$), il DES è considerato non sicuro in quanto un moderno calcolatore può esaminare tutte le possibili chiavi in poche ore. Una variante del DES chiamata Triplo DES utilizza chiavi più lunghe e fornisce un livello accettabile di sicurezza.

### 11.5.2 AES (Advanced Encryption Standard)

L'AES è un algoritmo di crittografia adottato come standard nel 2001, che ha sostituito il DES. L'AES suddivide il messaggio in blocchi di 128 bit che vengono cifrati individualmente. È possibile utilizzare chiavi di lunghezza di 128, 192 o 256 bit. Con $2^{128}$ (circa $3.4 \times 10^{38}$) chiavi possibili a 128 bit, l'esame di tutte le chiavi risulta al momento impraticabile, garantendo un elevato livello di sicurezza.

## 11.6 Pro e contro della crittografia simmetrica

La crittografia simmetrica presenta vantaggi e svantaggi:
**Vantaggi:**

- Gli algoritmi (come Triplo DES, AES e altri) possono essere implementati in modo efficiente.

**Svantaggi:**

- Le parti che comunicano devono scambiarsi in modo sicuro la chiave prima della comunicazione, il che costituisce un punto critico per il quale non esistono soluzioni generali affidabili al momento.

La crittografia simmetrica è adatta per comunicazioni che coinvolgono entità fidate che possono condividere in modo sicuro la chiave crittografica in anticipo.

## 11.7 Crittografia a chiave pubblica (asimmetrica)

La crittografia a chiave pubblica, introdotta nella seconda metà degli anni '70 da W. Diffie e M. Hellman, si basa su due chiavi, una pubblica e una privata, possedute da ciascun utente. Quando si utilizza una delle due chiavi per cifrare un messaggio, solo l'altra chiave può essere utilizzata per decifrarlo. Le due chiavi sono totalmente indipendenti, il che significa che è impossibile derivare una delle due chiavi anche se si conosce l'altra. Questo assicura che solo il legittimo destinatario possa leggere il messaggio cifrato.

### 11.7.1 Esempio

Per illustrare il funzionamento della crittografia asimmetrica, supponiamo che Alice voglia inviare un messaggio a Bob utilizzando le loro chiavi pubbliche e private. Alice cifra il messaggio M con la sua chiave privata A- e poi con la chiave pubblica di Bob B+. Bob può decifrare il messaggio utilizzando la sua chiave privata B- e poi la chiave pubblica di Alice A+. In questo modo, solo Bob può decifrare il messaggio cifrato da Alice.

## 11.8 Integrità e firma digitale

La crittografia a chiave pubblica può essere combinata con le funzioni hash crittografiche per garantire l'autenticità e l'integrità di un messaggio. Le funzioni hash crittografiche, come SHA-256, producono un digest di un messaggio con alcune proprietà: il digest ha una lunghezza fissa, è difficile costruire un messaggio con un determinato digest, e anche una minima modifica al messaggio produce un digest completamente diverso. Questi digest possono essere utilizzati per autenticare l'origine di un messaggio e garantirne l'integrità.

### 11.8.1 Firma digitale

La firma digitale è un modo per autenticare un messaggio utilizzando la crittografia a chiave pubblica e le funzioni hash crittografiche. Quando Alice vuole inviare un messaggio firmato a Bob, calcola il digest del messaggio utilizzando una funzione hash crittografica e cifra il digest con la sua chiave privata. Questo crea una firma digitale che accompagna il messaggio. Bob può verificare l'autenticità della firma decifrando il digest utilizzando la chiave pubblica di Alice e confrontandolo con il digest calcolato indipendentemente.

## 11.9 Pro e contro della crittografia asimmetrica

La crittografia asimmetrica presenta vantaggi e svantaggi:
**Vantaggi:**

- Non è necessario scambiarsi chiavi segrete in modo sicuro. Le chiavi pubbliche, che per definizione sono pubbliche, sono le uniche chiavi che devono essere comunicate.
- Consente la firma digitale per autenticare i messaggi.

**Svantaggi:**

- Gli algoritmi di crittografia asimmetrica sono più lenti rispetto a quelli di crittografia simmetrica.
- È necessario garantire che le chiavi pubbliche siano autentiche e appartenenti alle persone corrette.

## 11.10 La crittografia nella vita quotidiana: SSL/TLS

SSL (Secure Socket Layer) e TLS (Transport Layer Security) sono protocolli di sicurezza utilizzati per garantire la comunicazione sicura su Internet. TLS viene utilizzato per autenticare l'identità del server a cui ci si connette, rispondendo alla domanda: "Sono veramente connesso al server della mia banca?".

Il funzionamento di SSL/TLS può essere riassunto in modo approssimativo nel seguente modo: quando un browser si connette a un server, il server invia un certificato digitale che contiene il nome del server e la sua chiave pubblica. Il browser verifica la firma del certificato utilizzando una Certification Authority (CA). Successivamente, il browser e il server generano una chiave casuale, che viene cifrata con la chiave pubblica del server e inviata al server. Entrambi utilizzano questa chiave per cifrare e decifrare i dati durante la comunicazione.

## 11.11 Punti chiave

La crittografia riveste un ruolo fondamentale nella protezione delle comunicazioni e delle informazioni sensibili. Ecco alcuni punti chiave da ricordare:

- La crittografia a chiave pubblica (asimmetrica) utilizza due chiavi, una pubblica e una privata, per cifrare e decifrare i messaggi. È una soluzione efficace per scambiarsi informazioni in modo sicuro senza la necessità di condividere segretamente le chiavi.
- La crittografia simmetrica utilizza la stessa chiave per cifrare e decifrare i messaggi. È efficiente ma richiede la condivisione segreta delle chiavi tra le parti coinvolte nella comunicazione.

- Le funzioni hash crittografiche generano un digest univoco per un messaggio, che può essere utilizzato per verificare l'integrità e l'autenticità dei dati.
- La firma digitale è un meccanismo per autenticare i messaggi utilizzando la crittografia a chiave pubblica e le funzioni hash crittografiche.
- SSL/TLS sono protocolli di sicurezza utilizzati per garantire la comunicazione sicura su Internet, autenticando l'identità del server e proteggendo i dati scambiati durante la comunicazione.
- È importante tenere presente che la sicurezza di un algoritmo crittografico risiede unicamente nelle chiavi utilizzate e non nel mantenere segreto l'algoritmo stesso.

La crittografia svolge un ruolo cruciale nella protezione delle informazioni sensibili e nella sicurezza delle comunicazioni digitali. È fondamentale comprendere i principi di base della crittografia e utilizzare algoritmi e protocolli sicuri per garantire la privacy e l'integrità dei dati.

# Chapter 12

# Macchine virtuali

## 12.1 Introduzione

Le macchine virtuali consentono di simulare macchine fisiche tramite l'aggiunta di uno strato di software tra la macchina fisica e il sistema operativo. Esistono diversi vantaggi nell'utilizzo delle macchine virtuali:

- **Testing**: permette di avere più sistemi operativi diversi in esecuzione su una singola macchina, semplificando le attività di testing e sviluppo.
- **Monitoraggio**: consente di monitorare il comportamento del sistema tramite la macchina virtuale che lo sta eseguendo.
- **Servizi**: un unico hardware può essere utilizzato per fornire più servizi, ognuno all'interno di una macchina virtuale separata, ad esempio per i servizi di rete.
- **Affidabilità**: in caso di problemi hardware, è possibile copiare le macchine virtuali su un nuovo hardware senza interrompere il servizio.
- **Isolamento**: se una delle macchine virtuali dovesse avere un problema, le altre continueranno a funzionare in modo indipendente.

Tuttavia, l'utilizzo delle macchine virtuali comporta anche alcuni svantaggi, tra cui:

- **Ridotte prestazioni**: le macchine virtuali devono simulare in software tutti i dispositivi hardware, incluso quelli ottimizzati per le prestazioni. Inoltre, le risorse hardware sono condivise tra le diverse macchine virtuali e il tempo di esecuzione deve essere suddiviso in modo equo.
- **Traduzione delle istruzioni**: le istruzioni dell'OS guest devono essere tradotte in istruzioni comprensibili per il VMM/OS host, introducendo un certo overhead.
- **Accesso ai dati**: l'accesso ai dati richiede due accessi in RAM, uno dall'OS guest e uno dall'OS host che perfeziona l'operazione. Lo stesso vale per l'accesso al disco, che viene simulato con un file.

Nonostante gli svantaggi legati alle prestazioni, i vantaggi delle macchine virtuali superano di gran lunga le limitazioni, grazie alla riduzione dei costi (condivisione dell'hardware) e alla sicurezza garantita dalle macchine virtuali.

## 12.2 Tipologie di macchine virtuali

Esistono diverse tipologie di macchine virtuali:

### 12.2.1 Tipo 0: Hypervisor

Il tipo 0 prevede l'utilizzo di uno strato di software, chiamato Hypervisor o Virtual Machine Monitor (VMM), che viene eseguito direttamente sull'hardware come un sistema operativo. L'Hypervisor gestisce più macchine virtuali, ognuna delle quali ha il proprio sistema operativo che lavora sui propri processi. Questa tipologia è più performante, poiché vi è solo uno strato di software minimale tra le macchine virtuali e l'hardware. Tuttavia, è più complesso da installare e richiede la gestione dell'hardware da parte dell'Hypervisor.

### 12.2.2 Tipo 2: Processo virtuale

Il tipo 2 prevede che la macchina virtuale sia simulata da un processo che viene eseguito su un normale sistema operativo. Il sistema operativo host può eseguire sia i suoi processi che le macchine virtuali con sistemi operativi guest, che a loro volta lavorano sui propri processi. Questo tipo di macchina virtuale è meno performante, poiché tutto deve passare e essere gestito dall'OS host. Tuttavia, è più facile da installare (è simile a un normale programma) e semplifica la programmazione della macchina virtuale, poiché si interfaccia solo con l'OS host e può utilizzare le API fornite per interagire con l'hardware.

### 12.2.3 Tipo 1: VMM accanto all'OS

Il tipo 1 è un'ibridazione dei due tipi precedenti, in cui il VMM è posizionato sull'hardware accanto all'OS. Questo tipo di macchina virtuale è più performante rispetto al tipo 2, ma richiede una gestione più complessa in quanto deve interagire sia con l'hardware che con l'OS per la gestione delle risorse.

## 12.3 Gestione della memoria

La gestione della memoria è un aspetto importante nelle macchine virtuali, in quanto ogni macchina virtuale deve gestire la memoria per il sistema operativo che esegue al suo interno. Ci sono diverse modalità di allocazione della memoria:

- **Statica**: tutta la memoria necessaria viene allocata inizialmente. Questa modalità può essere limitante in quanto impedisce l'utilizzo di funzionalità come i thread multipli e la ricorsione.
- **Lineare**: la memoria viene allocata e deallocata in ordine LIFO (Last-In, First-Out). Ciò significa che i blocchi di memoria allocati successivamente devono essere deallocati prima di quelli allocati in precedenza. Questa modalità è ancora utilizzata per particolari caratteristiche delle macchine virtuali.
- **Dinamica**: la memoria viene allocata dinamicamente da un blocco di memoria pre-allocato chiamato *heap*. La gestione della memoria in questa modalità può essere automatica, ad esempio tramite un *garbage collector* che libera la memoria non utilizzata. Questo approccio consente una gestione più efficiente e flessibile della memoria.

## 12.4 Interpreti

Ogni macchina virtuale richiede un interprete, che svolge un ruolo simile a quello della CPU in una macchina fisica, con l'aggiunta della traduzione delle istruzioni. Ogni interprete ha il suo *instruction-set*, la sua pila di istruzioni da eseguire e si basa su registri virtuali. Ci sono diverse tipologie di interpreti:

- **Interpretazione a token**: l'interprete interpreta le istruzioni binarie o i *token* e li confronta con quelli dell'*instruction-set* per identificarli.
- **Interpretazione a indirizzi**: l'interprete riceve gli indirizzi delle istruzioni da eseguire.
- **Interpretazione a stringhe**: le istruzioni sono assunte come stringhe. Questo approccio è più semplice, ma richiede una traduzione delle istruzioni e può essere meno efficiente.

In genere, l'interprete è affiancato da un compilatore che compila le parti di codice più frequentemente utilizzate, in modo da renderle disponibili per l'esecuzione senza la necessità di interpretazione.

### 12.4.1 Java Virtual Machine

La Java Virtual Machine (JVM) è una macchina virtuale basata su un sistema di registri a pila, poiché deve gestire ampiamente oggetti. La JVM è composta da diversi componenti, tra cui i più importanti sono:

- **Caricatore di classi**: si occupa di caricare le classi Java durante l'esecuzione del programma.
- **Verificatore di classi**: verifica la correttezza delle classi Java prima che vengano eseguite.
- **Interprete e compilatore**: esegue il codice Java interpretandolo o compilando le parti più utilizzate per migliorare le prestazioni.
- **Interfaccia nativa**: consente di richiamare programmi scritti in altri linguaggi all'interno di una JVM.
- **Strutture Runtime**: forniscono supporto per la gestione delle eccezioni, la gestione della memoria e altre funzionalità necessarie all'esecuzione dei programmi Java.
- **Garbage Collector**: si occupa della gestione dinamica della memoria, liberando automaticamente la memoria non utilizzata per evitare errori comuni.

I programmi Java vengono compilati in file con estensione ".class", che possono essere eseguiti dalla JVM. Questo rende il linguaggio Java portabile, in quanto i file ".class" possono essere eseguiti su diverse piattaforme senza dover riscrivere il codice sorgente. I file ".class" contengono informazioni dettagliate sulla classe, inclusi metodi, attributi, interfacce e altre informazioni necessarie per l'esecuzione del programma.

Prima dell'esecuzione, un'applicazione Java deve passare attraverso una fase di verifica e inizializzazione, in cui vengono verificate e inizializzate le classi Java. Durante l'esecuzione, il garbage collector si occupa della gestione automatica della memoria, evitando la necessità di gestirla manualmente e riducendo gli errori comuni legati alla gestione della memoria.

# Chapter 13

# Il sistema operativo Android

## 13.1 Struttura

Android è costituito da 4+1 livelli:

- **Applicazioni**: includono le app a livello utente e di sistema. Sono applicazioni Java eseguibili dalla macchina virtuale Dalvik.
- **Framework**: fornisce le API utilizzate dalla DalvikVM per interagire con le varie funzionalità dei dispositivi.
- **Librerie**: includono librerie Java standard e di terze parti, escluse quelle non necessarie per il dispositivo specifico, con l'aggiunta di librerie necessarie per il funzionamento su dispositivi mobili.
- **Kernel Linux modificato**: si tratta di un kernel Linux minimale con caratteristiche diverse dal kernel Linux standard, come ad esempio un'unità di power management. Il kernel non può eseguire processi.
- **Dalvik Virtual Machine**: la macchina virtuale su cui devono essere eseguite tutte le app Android.

## 13.2 La Dalvik Virtual Machine

Tutte le app Android devono essere eseguite sulla DalvikVM. La DalvikVM è una macchina virtuale basata su registri, progettata per evitare un ulteriore rallentamento del dispositivo. A differenza della JVM (Java Virtual Machine) che esegue file con estensione ".class", la DalvikVM esegue file con estensione ".dex" (Dalvik Executable), che sono più leggeri dei file ".class". Questa scelta consente a Google di evitare il pagamento dei diritti di licenza per la JVM di Oracle. La DalvikVM è ottimizzata per il multitasking, che era già un requisito essenziale al momento della sua creazione.

Le app Android devono essere scritte in Java, compilate in file con estensione ".class", convertite in file con estensione ".dex" e poi eseguite sulla Dalvik Virtual Machine, che a sua volta viene eseguita sul kernel Linux. Per ogni app viene creata un'istanza di macchina virtuale, il che contribuisce a garantire la sicurezza del sistema a discapito delle prestazioni, a causa dell'occupazione di memoria necessaria per le diverse istanze della VM. Le istanze delle VM non possono comunicare direttamente tra loro come i processi in un normale sistema operativo. Invece, possono comunicare solo tramite il message passing, poiché il metodo della memoria condivisa è stato abolito per garantire maggiore sicurezza. Android non fornisce permessi di root in modo che nessuna app possa eseguire operazioni che richiedono autorizzazioni di alto livello. Inoltre, ogni app deve essere fornita di una documentazione XML che specifichi i permessi richiesti e altre informazioni rilevanti per Android.

## 13.3 La macchina virtuale ART

La macchina virtuale ART (Android Runtime) è un ambiente di esecuzione delle applicazioni. Essa svolge la traduzione del bytecode dell'applicazione in istruzioni native. ART introduce l'utilizzo della compilazione anticipata (AOT) compilando intere applicazioni in codice nativo della macchina al momento dell'installazione.

### 13.3.1 Funzionalità della macchina virtuale ART

La macchina virtuale ART offre diverse funzionalità che migliorano l'efficienza complessiva dell'esecuzione delle applicazioni:

- Riduzione del consumo energetico.
- Esecuzione più rapida delle applicazioni.
- Miglioramento dell'allocazione della memoria e della gestione dei rifiuti (garbage collection).
- Nuove funzionalità di debug delle applicazioni.
- Profilazione più accurata delle applicazioni a un livello più alto.

L'introduzione di ART ha portato significativi miglioramenti nelle prestazioni complessive del sistema Android, consentendo un'esperienza più fluida e riducendo il consumo di risorse.

## 13.4 Applicazioni

Le app Android non hanno un singolo entry point e quindi non seguono un'esecuzione lineare. Invece, hanno diversi metodi che rispondono alle richieste del sistema operativo, come onStop(), onCreate(), onStart(), onDestroy() e onResume().

Esistono diversi tipi di app Android:

- **Activities**: sono schermate che forniscono componenti e sono in grado di rispondere alle richieste tramite un'interfaccia. Ogni schermata di un'app Android deve implementare un'Activity. Le activities vengono memorizzate in uno stack: quando un'activity viene sospesa (resasi non visibile), viene inserita nello stack. Quando si torna indietro, viene ripresa la prima activity nello stack. A partire dalla versione 3.0 di Android, sono stati introdotti anche i frammenti, che consentono di visualizzare contemporaneamente due schermate della stessa activity su dispositivi con schermi di grandi dimensioni, come i tablet, o una sola schermata su dispositivi con schermi più piccoli, come gli smartphone.
- **Services:** sono app che vengono eseguite in background senza un'interfaccia utente visibile, ad esempio un'applicazione di riproduzione musicale. Per interagire con un servizio, è necessario che implementi un'Activity o venga richiamato da altre app.
- **Broadcast Receivers:** servono a reagire a segnali o eventi specifici, ad esempio il premere un certo tasto, una chiamata in arrivo o una batteria scarica. Ogni app può avere più Broadcast Receivers per gestire i segnali rilevanti per l'app stessa.
- **Content Providers:** servono a rendere disponibili determinati contenuti alle app che ne fanno richiesta, ad esempio fornire i contatti dell'utente. I Content Provider devono implementare un insieme di metodi che consentono alle altre app di accedere a tali contenuti.

## 13.5 Message Passing: Intents

Per garantire l'efficienza del message passing, è stato sviluppato un sistema chiamato Intents. Tramite gli Intents, ogni app può fare uso di altre app, sia quelle preinstallate che quelle installate successivamente. Quando un'app ha bisogno di eseguire un'azione che non è in grado di svolgere da sola, ad esempio aprire una pagina web, richiede al sistema operativo di trovare un'app che possa svolgere tale azione. Il sistema operativo genera un Intent per cercare un'app adatta, chiede all'utente di selezionare l'app desiderata e la lancia.

Per essere lanciata, un'app deve dichiarare un Intent Filter, specificando le azioni che è in grado di svolgere, in modo che possa essere resa disponibile per l'arrivo degli Intents dal sistema operativo. Questo meccanismo di Intents permette alle app di interagire in modo flessibile tra loro, aumentando la versatilità e la funzionalità complessiva del sistema Android.

Queste dispense sono state preparate con LaTeX, esse costituiscono soltanto un ausilio per lo studente , quindi non sono in nessun modo sostitutive dei testi consigliati. Il PDF è dotato di hyperlinking quindi è possibile cliccare sui nomi delle sezioni nell'indice per spostarsi tra le pagine, allo stesso modo, per tornare all'indice è sufficiente cliccare sul numero della pagina.