

Capitolo 1 - Introduzione ai Sistemi Operativi

Federico Segala

Definizione di Sistema Operativo

Un **sistema operativo** è un *insieme di programmi* che fa da **intermediario tra hardware e utente** al fine di:

- facilitare l'utilizzo del computer
- rendere efficiente l'utilizzo dell'hardware
- evitare conflitti nell'allocazione delle risorse

Un sistema operativo ha principalmente due funzionalità di base: quella di **gestore di risorse** e quella di **programma di controllo**

Gli obiettivi di un sistema operativo sono principalmente *astrazione* ed *efficienza*, ovviamente che sono tipicamente in contrasto tra di loro un classico esempio di questa conflittualità è visibile nella disputa Windows vs. Linux, il primo molto astratto ma meno efficiente, il secondo invece, segue una filosofia contraria.

I sistemi operativi hanno vissuto diversi step evolutivi per arrivare allo stato odierno della tecnologia, questi step sono categorizzabili in **4 generazioni** legate alla tipologia di calcolatori presenti sul mercato, ognuna nata con uno scopo comune, quello di **massimizzare l'utilizzo della CPU**

1° Generazione

Questa generazione comprende i primissimi sistemi di calcolo, fatti di enormi valvole, in questi **non** era presente alcun **sistema operativo**. L'operatore era lo stesso programmatore e l'accesso alla macchina avveniva tramite prenotazione. Per eseguire un programma, esso veniva caricato in memoria un'istruzione per volta agendo su degli interruttori.

A seguito di questo primo e rudimentale approccio, si registra una evoluzione con la diffusione delle prime **periferiche** quali ad esempio *lettori e perforatori di schede, nastri, stampanti* per gestire i quali si sono resi necessari i **device driver**. Assieme alle periferiche vengono sviluppate le prime **librerie di funzioni**, e i primi **compilatori, linker, loader**.

Il tutto era caratterizzato da **scarsa efficienza**, la programmazione è stata facilitata, ma le operazioni da svolgere sono rimaste complesse, con tempi di setup molto elevanti e un conseguente basso utilizzo delle risorse a disposizione.

2° Generazione

Vengono introdotti nei calcolatori i primi **transistor**. Si tratta del primo caso nel quale gli operatori non sono i programmatori, dato che non serve più grande specializzazione e conoscenza dell'architettura sottostante per poter programmare. Ciò comporta l'eliminazione dello schema a prenotazione con gli operatori che eliminano gran parte dei tempi morti.

Batching

Un batch corrisponde ad un raggruppamento di job simili nella loro esecuzione, in questa maniera vengono raggruppati i programmi simili tra di loro, permettendo di eliminare lavoro superfluo.

Ad esempio *senza batching*:

- *sequenza*: fortran, cobol, fortran, cobol → comp. fortran, link fortran, comp. cobol, link cobol, comp. fortran, ilnk fortran, ... Mentre *con batching*:
- *sequenza*: fortran, fortran, cobol, cobol → comp. fortran, link fortran, comp. cobol, link cobol

Un'evoluzione rispetto a ciò si ha con l'**automatic job sequencing** grazie al quale il sistema o_perat_ivo si occupa di passare da un job ad un altro facendo il lavoro svolto altrimenti da un operatore e abbassando notevolmente la quantità di tempi morti.

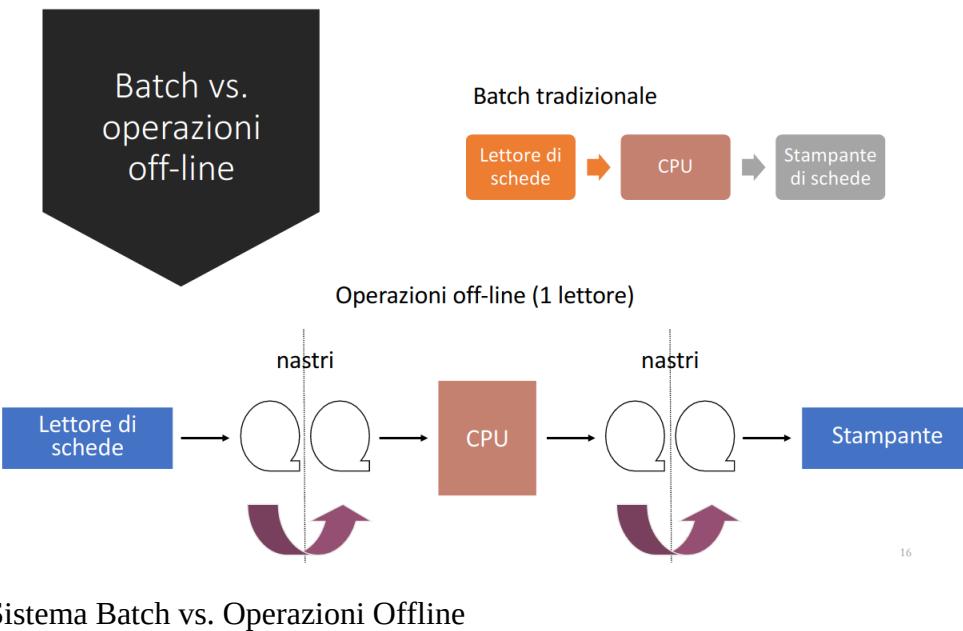
Si tratta del primo vero esempio di **sistema operativo**, ovvero un **monitor** che fa da gestore residente perennemente in memoria. Le componenti di questo primo sistema operativo sono:

- *driver* per dispositivi di I/O
- *sequenzializzatore* dei job
- *interprete* di schede di controllo

La **sequenzializzazione** dei job viene realizzata tramite il cosiddetto **job control language** che utilizza *schede e record di controllo*

Questa seconda generazione comporta alcune limitazioni, in particolare l'utilizzo del sistema è ancora molto basso dato che la velocità dell'input/output è ancora molto minore rispetto a quella della CPU. In particolare fino a questo momento la cpu resta inutilizzata durante le operazioni di input/output.

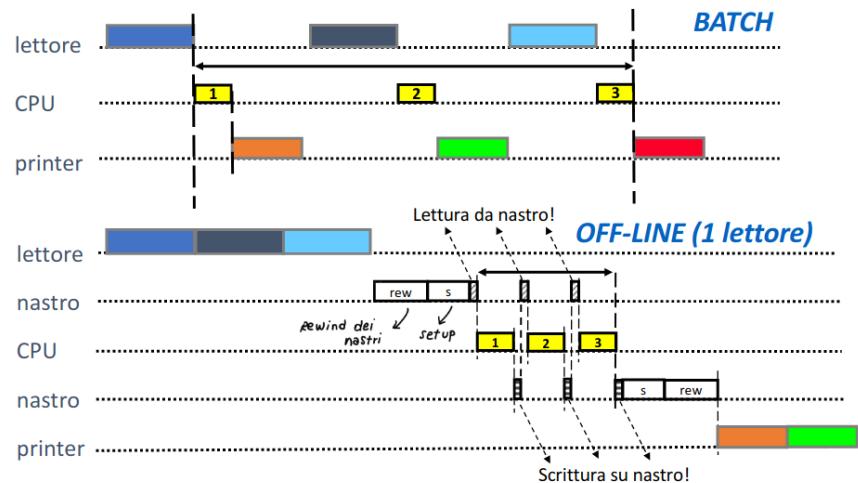
Una soluzione a ciò si ottiene **sovrapponendo** operazioni di I/O e di elaborazione off-line. Questa elaborazione off-line avviene grazie alla diffusione di nastri magnetici, più capienti e più veloci. In particolare operazioni di input/output e cpu avvengono su macchine indipendenti: da scheda a nastro su una macchia, e da nastro a CPU su un'altra macchina. La vera limitazione ora è data dalla **velocità dei nastri**



16

Sistema Batch vs. Operazioni Offline

Batch vs. operazioni offline



17

È possibile implementare questa modalità offline utilizzando **più lettori** ed è qui che diventa molto più conveniente utilizzare questo tipo di approccio. La cosa ottimale sarebbe arrivare a **sovraporre** operazioni di I/O e di CPU su una **stessa macchina**, ciò è possibile, ma si rende necessario del *supporto architettonico*

Per fare ciò abbiamo a disposizione due strategie:

- **polling**: un dispositivo viene interrogato continuamente tramite esplicite istruzioni bloccanti

- **meccanismo asincrono:** in questo approccio si fa in modo che le richieste di I/O non siano bloccanti, vengono utilizzati degli *interrupt* e la tecnica del *direct memory access*

Interrupt & Input/Output

Lo schema di base del meccanismo di interrupt è il seguente:

- la CPU programma cosa il dispositivo di I/O deve fare
- A questo punto il controllore di tale dispositivo viene messo in esecuzione e la CPU prosegue se possibile la propria esecuzione
- Al termine delle operazioni di I/O il controllore avvisa la CPU del termine delle sue attività, la CPU riceve un **interrupt**

Al *ricevimento* di tale interrupt la CPU deve interrumpere l'istruzione correntemente in esecuzione salvando lo stato di quanto in esecuzione, dopo aver fatto ciò salta ad una locazione predefinita X in base all'interrupt che arriva, tale locazione informa la CPU sulla posizione della **routine di servizio** che deve svolgere per servire l'interruzione. Al termine dell'esecuzione della routine, viene ripresa l'istruzione interrotta in precedenza.

DMA & Input/Output

Nel caso di dispositivi di input/output molto veloci il rischio è quello che si arrivi ad una situazione in cui gli interrupt sono molto frequenti, portando a scarsissima efficienza. Una soluzione a ciò viene proposta dal meccanismo del **direct memory access**. Tale meccanismo, grazie ad uno specifico controllore, permette di trasferire blocchi di dati tra I/O e memoria senza dover necessariamente passare per la CPU. In questo modo si riduce notevolmente il numero di interrupt, venendone generato uno per ogni blocco di dati.

Buffering & Spooling

Altre due strategie che corrono in supporto alle necessità poco prima evidenziate sono quelle del **buffering** e dello **spooling**.

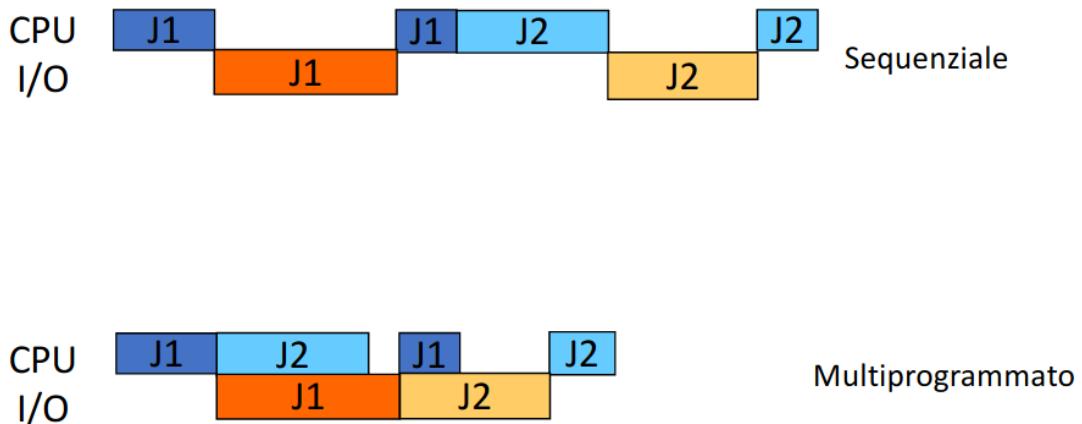
Nel **buffering** si ottiene la sovrapposizione di CPU e I/O di uno **stesso job**. Questa tecnica prevede che il dispositivo di I/O legga e scriva più dati di quanti siano quelli effettivamente richiesti. Si tratta di una tecnica molto utile quando la velocità di I/O sia molto simile a quella di una CPU. Questo però non è così e dunque porta ad un miglioramento soltanto marginale. Per questa limitazione nella velocità si è passati allo **spooling**.

Spooling sta per “**simultaneous peripheral operations on-line**” grazie al quale vengono sovrapposte operazioni di CPU e I/O da parte di **job diversi**. Questo nasce dal fatto che sono stati introdotti i *dischi magnetici* che a differenza dei nastri non garantiscono

accesso casuale. Tale tecnica prevede infatti di utilizzare il disco come un grande buffer unico per tutti i job: nasce quindi il concetto di **pool di job** ovvero i programmi sono su disco e devono essere caricati in memoria, si renderà quindi necessario anche il *job scheduling* (chi può essere caricato su disco)

3° Generazione

Si notò con il passare del tempo che un singolo job non poteva tenere occupata in maniera sufficiente la CPU, per fare ciò diventa necessaria la **competizione di più job**, i quali saranno tutti caricati in memoria. Le fasi di attesa (I/O) di un singolo job vengono sfruttate al fine di poter mettere in esecuzione altri job. Si introduce il concetto di **multiprogrammazione**



Nei sistemi tradizionali c’è alta tendenza alla non interattività, dato che è **importante il tempo di completamento** di un singolo job. In un sistema multiprogrammato la necessità diventa quella di soddisfare **molti utenti** che **operano in modo interattivo**, diventa quindi di fondamentale importanza il **tempo di risposta**

Nasce il concetto di **time sharing**, che è una estensione logica del concetto di multiprogrammazione nella quale l’utente ha l’impressione di avere la macchina soltanto per sé. Ciò va a migliorare ulteriormente la interattività. Nascono i sistemi moderni.

Protezione

Nei sistemi precedenti il problema della condivisione non si poneva, ma da questo punto in poi è necessario tenere in considerazione che l’esecuzione di un programma può influenzare l’esecuzione degli altri. Serve quindi tenere in considerazione tre tipologie di protezione:

- **Input/Output:** programmi diversi non devono usare lo stesso dispositivo di I/O in modo contemporaneo

- **Memoria:** un programma non può leggere/scrivere in una zona di memoria che non gli appartiene
- **CPU:** prima o poi il controllo della CPU deve ritornare al sistema operativo

Per quanto riguarda la protezione dell'I/O questa viene realizzata con il meccanismo del **modo duale** di esecuzione, il quale può essere **modalità USER** e **modalità KERNEL**. La differenza tra le due modalità consiste nel fatto che in modalità user i job non possono accedere in maniera diretta alle risorse di I/O. Qualsiasi operazione di input o output viene categorizzata come *operazione privilegiata*: per accedere ad una risorsa di input/output è necessario il meccanismo delle **system call**:

- le istruzioni per accesso ad I/O invocano delle system call
- le system call (interrupt software) cambiano la modalità da USER a KERNEL e svolgono le operazioni I/O richieste
- al termine il sistema operativo ripristina la modalità USER

Per quanto riguarda la protezione della memoria, come detto, è necessario proteggere lo spazio dei vari processi, compreso dunque anche quello del sistema operativo (monitor). Ciò si può realizzare associando ad ogni processo un **registro limite**, ognuno di questi può essere modificato dal sistema operativo soltanto per mezzo di istruzioni privilegiate.

Passando alla protezione della CPU, serve appunto la garanzia che il sistema operativo mantenga il controllo del sistema, per fare ciò si utilizza un **timer** associato ad ogni job. Alla scadenza di tale timer, il controllo del sistema operativo ritorna al monitor.

4° Generazione

Abbiamo svariate tipologie di sistemi operativi a seconda dell'utilizzo che si va a fare del calcolatore in questione:

- S.O. per PC e workstation: utilizzo personale del calcolatori
- S.O. di rete: caratterizzati da una separazione logica delle risorse remote che non corrispondono alle risorse locali
- S.O. distribuiti: le risorse remote non sono separate logicamente e vengono viste come risorse locali
- S.O. real time: presentano vincoli molto stretti sui tempi di risposta del sistema
- S.O. embedded: per realizzare applicazioni specifiche

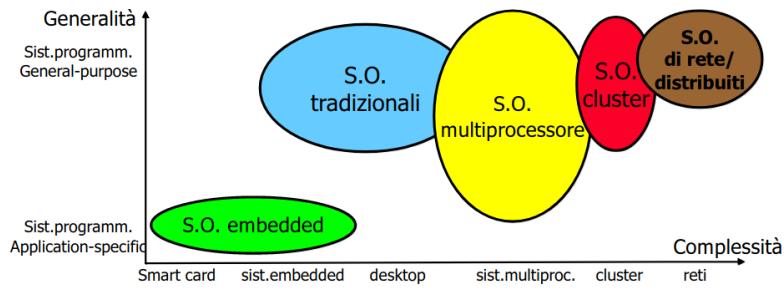
Riassumendo...

- **1° Generazione**
 - Device driver
 - Librerie

- **2° Generazione**
 - Batching
 - Automatic Job Sequencing
 - Off-line processing
 - Sovrapposizione CPU e I/O
- **3° Generazione**
 - Multiprogrammazione
 - Time Sharing

Lo spazio dei S.O.

Real-time = dimensione ulteriore!



Spazio dei Sistemi Operativi

Capitolo 2 - Componenti di un Sistema Operativo

Federico Segala

Componenti di un Sistema Operativo

Un sistema operativo, come immaginabile di può scomporre in più elementi:

- gestione dei processi
- gestione della memoria primaria
- gestione della memoria secondaria
- gestione dell'input e dell'output
- gestione dei file
- protezione
- rete
- interprete dei comandi

Gestione dei Processi

Con processo intendiamo un **istanza di programma** che si trova in stato di **esecuzione**.

Un processo necessita di risorse e viene eseguito in modo sequenziale, dunque un'istruzione alla volta. Il sistema operativo, che è esso stesso un processo, ha diverse mansioni da questo punto di vista:

- creare e distruggere i processi
- sospendere e riesumare i processi
- sincronizzare e far comunicare tra di loro i vari processi in esecuzione

Gestione della Memoria Primaria

Questa memoria va a conservare i dati condivisi tra CPU e dispositivi di I/O. Ogni programma per poter essere eseguito deve essere caricato in memoria principale. In questo ambito il sistema operativo è responsabile di:

- gestire lo spazio di memoria (quali parti e da chi sono utilizzate)

- decidere quale processo caricare in memoria quando esiste spazio disponibile
- gestire l'allocazione e il rilascio dello spazio in memoria ai vari processi

Gestione della Memoria Secondaria

Siccome la memoria principale è di dimensioni ridotte e soprattutto **volatile**, si rende indispensabile un dispositivo di archiviazione che permetta di mantenere in maniera permanente grandi quantità di dati. In questo ambito il sistema operativo si occupa di:

- gestire lo spazio libero sulla memoria di massa
- allocare lo spazio su memoria di massa
- ordinare gli accessi ai dispositivi

Gestione dell'Input/Output

È necessario che in un sistema operativo di semplice utilizzo il sistema operativo nasconde all'utente le specifiche caratteristiche dei diversi dispositivi di input e output. Il sistema di I/O consiste dunque di:

- sistema per accumulare gli accessi ai dispositivi (*buffering*)
- generica interfaccia verso i device driver
- device driver specifici per alcuni dispositivi (vedi schede grafiche)

Gestione dei File

Le informazioni spesso e volentieri sono memorizzate su supporti fisici diversi controllati da driver con caratteristiche differenti. Un **file** è un'astrazione logica per rendere conveniente l'utilizzo della memoria non volatile, e corrisponde ad una **raccolta di informazioni correlate**. Il sistema operativo in questo ambito è responsabile di:

- creare e cancellare file e directory
- fornire primitive per la gestione di questi
- gestire corrispondenza tra file e spazio fisico su memoria di massa
- salvare informazioni a scopo di backup

Protezione

Come accennato nel Capitolo 1 si rende necessario un meccanismo per controllare l'accesso alle risorse da parte di utenti e processi. Il sistema operativo è dunque responsabile di:

- definire accessi autorizzati e non

- definire controlli per accessi
- fornire strumenti per verificare la politica di accesso

Rete

Un **sistema distribuito** è una collezione di elementi di calcolo che non condividono né la memoria né un clock nel quale le **risorse di calcolo** sono connesse tramite una **rete**. Il sistema operativo sarà responsabile di gestire le seguenti componenti:

- processi distribuiti
- memoria distribuita
- file system distribuito
- ...

Interprete dei comandi

La maggior parte dei comandi viene fornita al sistema operativo per mezzo di operazioni di controllo che permettono varie azioni:

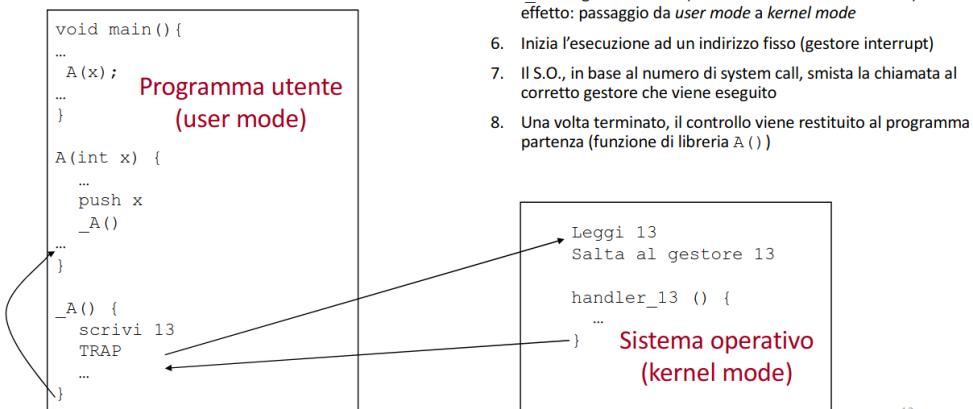
- creare e gestire processi
- gestire l'input/output
- gestire il disco, la memoria, il file system
- gestire le protezioni
- gestire la rete

Definiamo **shell** un programma che legge e interpreta comandi la cui funzione è quella di leggere ed eseguire la successiva istruzione di controllo fornитagli.

Se l'utente utilizza una shell, i processi invece utilizzano il meccanismo già citato delle **system call** che fanno da interfaccia per i processi verso il sistema operativo. Per comunicare tra processi e sistema operativo ci sono a disposizione varie opzioni:

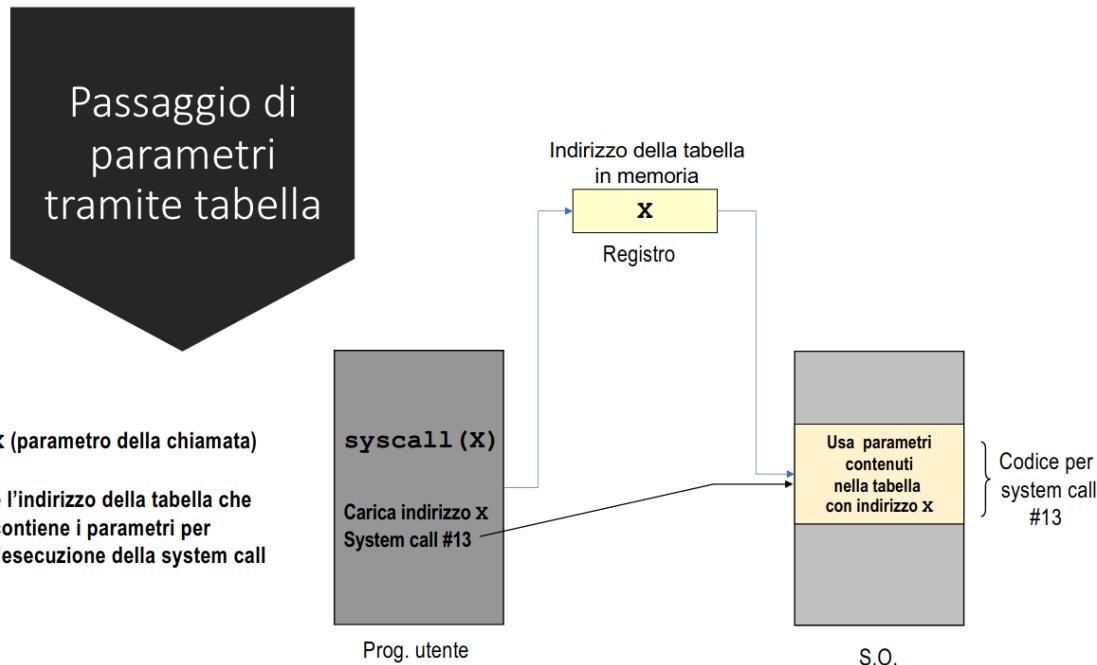
- passare i parametri della system call tramite registri
- passare i parametri tramite lo stack del programma
- memorizzare i parametri in una tabella in memoria, l'indirizzo di tale tabella viene passato in un registro o nello stack

Passaggio di parametri nello stack



12

Parametri passati tramite stack



In sintesi se le system call forniscono una vista lato programma delle operazioni di un sistema, i programmi di sistema forniscono una vista lato utente delle stesse:

- gestione e manipolazione dei file
- informazioni sullo stato del sistema
- strumenti di supporto alla programmazione
- formattazione documenti
- mail
- programmi di gestione della rete
- interprete dei comandi

Capitolo 3 - Architettura di un Sistema Operativo

Federico Segala

Struttura di un Sistema Operativo

I sistemi operativi si possono dividere in diverse categorie a seconda delle caratteristiche dello stesso. Vediamo i vari tipi di sistemi che andremo a trattare in questo capitolo:

- sistemi **monolitici**
- sistemi “a struttura semplice”
- sistemi **a livelli**
- **virtual machine**
- sistemi **kernel based**
- sistemi **client-server**

Sistemi Monolitici

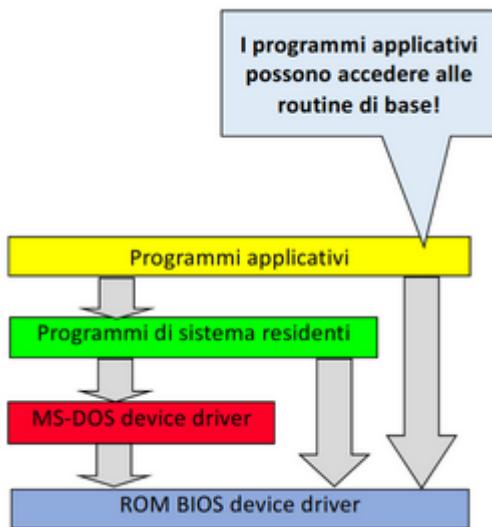
In questa tipologia di sistemi operativi la caratteristica principale consiste nell’assenza di gerarchia, esiste un **unico strato SW** tra utente e hardware. Questo fa in modo che tutte le componenti del sistema operativo si trovino allo stesso livello. Abbiamo in pratica a disposizione un insieme di procedure che possono chiamarsi vicendevolmente.

Gli svantaggi che comporta l’utilizzo di questa architettura derivano dal fatto che tutto il codice è dipendente dall’architettura hardware e distribuito all’interno di tutte le funzionalità del sistema operativo. Ciò rende molto difficili test e debugging.

Sistemi “a struttura semplice”

Questi sistemi operativi sono caratterizzati da una **minima organizzazione gerarchica**, che definisce dei livelli dalla gerarchia molto flessibile. La strutturazione mira a ridurre costi di sviluppo e manutenzione

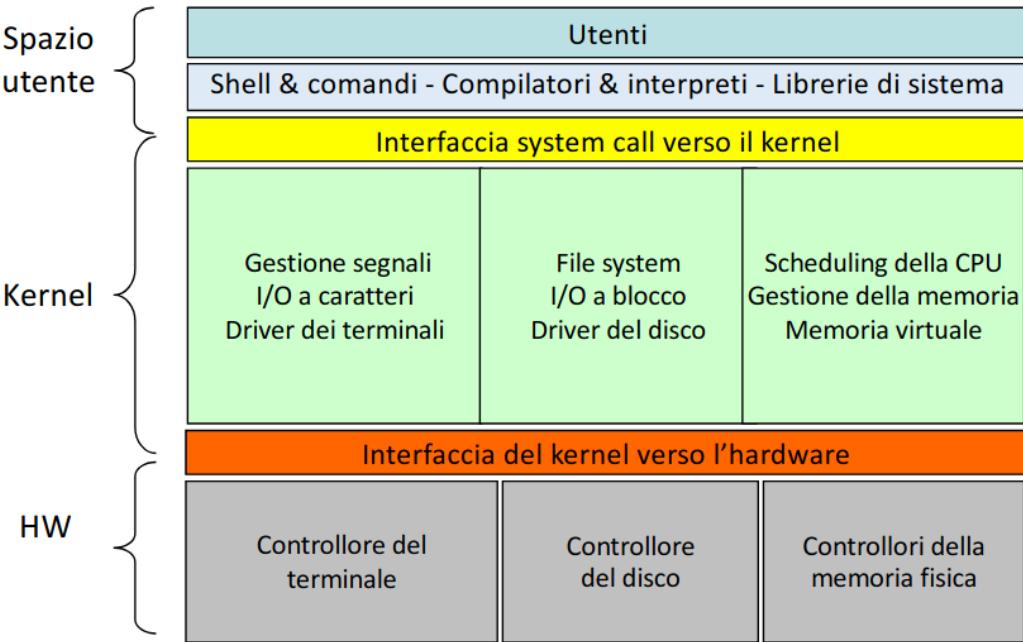
Un esempio tipico di questi tipi di sistemi è visibile in **MS-DOS** che è stato pensato per fornire il maggior numero di funzionalità nel minimo spazio possibile. Questo non è un sistema diviso in moduli, è organizzato secondo una struttura minima ma le interfacce e i livelli di funzionalità non sono appunto ben definiti. Non è prevista modalità duale



Un altro esempio si può trovare nella versione base di **UNIX**, nel quale il **kernel** è tutto ciò che sta tra il livello dell’interfaccia delle system call e l’HW e fornisce:

- file system
- scheduling della CPU
- gestione della memoria
- altre funzionalità

La struttura base di UNIX è molto limitata a causa delle limitate funzionalità hardware:



Sistemi a Livelli In questa architettura i servizi sono organizzati per livelli gerarchici, gerarchia nella quale l'interfaccia utente è al livello più alto, mentre l'hardware sta al livello più basso. Ogni livello può utilizzare soltanto funzionalità fornite dai livelli inferiori e definisce precisamente il tipo di servizio offerto e l'interfaccia verso il livello superiore nascondendone l'implementazione

Questa architettura comporta l'importante vantaggio della **modularità** che facilita la messa a punto e la manutenibilità del sistema. Alcuni **svantaggi** sono invece dati dalla difficoltà nel definire in modo appropriato i vari strati, una minore **efficienza**, dato che ogni strato aggiunge overhead alle system call, e una minor **portabilità**, dato che le funzionalità dipendenti dall'architettura sono sparse sui vari livelli

Un esempio di sistema a livelli si può trovare in **THE** inventato da *Dijkstra* che è un insieme di processi cooperanti sincronizzati tramite semafori:

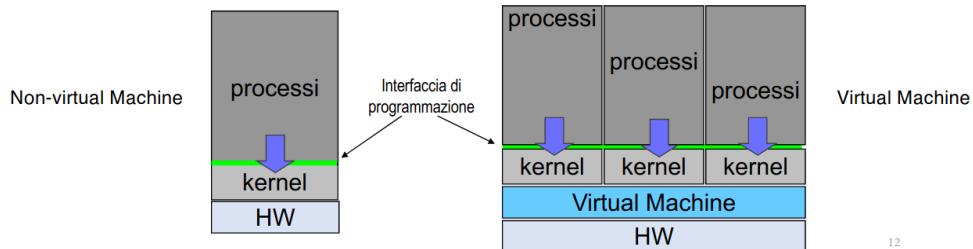
- Livello 5: programmi utente
- Livello 4: gestione I/O
- Livello 3: device driver della console
- Livello 2: gestione della memoria
- Livello 1: scheduling della CPU
- Livello 0: hardware

Virtual Machine

Questa architettura porta ad un'estremizzazione dell'approccio a livelli, ed è stata pensata per offrire un sistema timesharing multiplo. Hardware e sistema operativo sono trattati come hardware:

- una virtual machine fornisce un'interfaccia identica all'hardware sottostante
- il sistema operativo esegue sopra tale virtual machine
- la virtual machine dà l'illusione di processi multipli, ciascuno in esecuzione sul proprio hardware

Questa architettura dà la possibilità di avere più sistemi operativi in esecuzione nello stesso momento



Non-virtual Machine vs. Virtual Machine

Un approccio di questo tipo comporta vari vantaggi:

- **protezione completa del sistema** nel senso che ogni VM è isolata dalle altre
- più di un sistema operativo eseguito sulla stessa macchina host
- **ottimizzazione delle risorse**: la stessa macchina può ospitare quello che senza una VM doveva essere eseguito su macchine separate
- ottimo strumento per lo sviluppo di sistemi operativi
- buona **portabilità**

Alcuni vantaggi sono:

- **problemi di prestazione**
- necessità di gestire modalità duale virtuale: il sistema di gestione delle VM esegue in modalità kernel, ma la VM esegue in modalità user
- nessuna possibilità di condivisione dato che ogni VM è isolata dalle altre, per ovviare a ciò si può condividere un volume nel file system, oppure definire una rete virtuale tra VM via software

Sistemi Kernel Based

In questo tipologia di architettura sono presenti soltanto **due livelli**: servizi kernel e servizi non kernel. I vantaggi che questa architettura comporta sono di fatto gli stessi dei sistemi a livelli ma senza avere troppi livelli (di conseguenza meno overhead). Alcuni svantaggi sono:

- perdita di generalità rispetto ad un sistema a livelli
- nessuna regola organizzativa per parti del S.O. fuori dal kernel
- kernel complesso tende a diventare monolitico

Questo tipo di architettura si trova in moltissime implementazioni moderne di UNIX

Sistemi Client-Server

In questo architettura tutti i servizi del sistema operativo sono realizzati come precessi utente. Il client chiama un processo servitore per usufruire di un servizio. Il server, dopo l'esecuzione, restituisce il risultato al client. Il kernel si occupa soltanto della gestione della comunicazione tra client e eserver.

Tale modello si presta molto bene per sistemi operativi distribuiti. Molti sistemi operativi moderni realizzano alcuni servizi in questa modalità.

Se un tempo i sistemi operativi erano scritti in linguaggio assembler, quelli moderni vengono scritti in linguaggi ad alto livello, comportando numerosi vantaggi, tra cui implementazione più rapida, maggiore compattezza, facilità di comprensione e manutenzione, portabilità

Capitolo 4 - Processi e Thread

Federico Segala

Concetto di Processo

È innanzitutto necessario definire la differenza tra *programma* e *processo*. Per **processo** intendiamo un'**istanza di programma in esecuzione**. Dunque se il concetto di programma è statico, quello di processo è un concetto dinamico.

Ogni processo è eseguito in maniera sequenziale, quindi una istruzione alla volta, ma in un sistema multiprogrammato i processi evolvono in maniera concorrente. Facendo questa considerazione è necessario ricordare che le risorse, sia logiche che fisiche, sono limitate, e che il sistema operativo stesso è composto da un'insieme di più processi in esecuzione.

Un processo consiste di:

- **istruzioni**: parte statica del codice
- **dati**: sezione della memoria contenente le variabili globali del codice
- **stack**: contenente chiamate a procedura ed eventuali parametri, e variabili locali ad ogni funzione
- **heap**: memoria allocata in modo dinamico
- **attributi**: informazioni riguardanti il processo, come ad esempio *id*, *stato*, *controllo*,
...

Process Control Block

All'interno del sistema operativo ogni processo è rappresentato dal **process control block** che contiene varie informazioni riguardanti il processo stesso:

- stato del processo
- program counter
- valori dei registri
- informazioni sulla memoria (registri limite, tabella pagine)
- informazioni sullo stato dell'I/O (richieste pendenti, file)
- informazioni sull'utilizzo del sistema (CPU)
- informazioni di scheduling (priorità)

Stati di un Processo

Durante la sua esecuzione, un processo evolve attraverso diversi stati, rappresentabili in modo generale attraverso un **diagramma degli stati**.

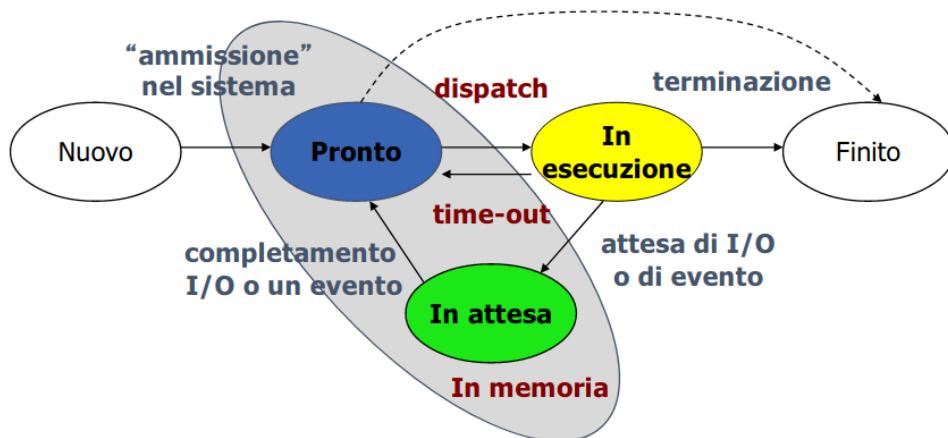


Diagramma degli Stati di un Processo

Scheduling

Il concetto di **scheduling** consiste nella *selezione* del processo da eseguire nella CPU al fine di garantire **multiprogrammazione** con la quale massimizzare l'utilizzo della CPU e **time sharing** con il quale commutare frequentemente la CPU tra processi in modo che ognuno creda di avere la CPU tutta per sé.

Ogni processo viene inserito in una serie di **code di scheduling**:

- **ready queue:** coda dei processi pronti all'esecuzione
- **coda di un dispositivo:** coda dei processi in attesa che un particolare dispositivo si liberi

Inizialmente ogni processo si trova nella *ready queue* in attesa che venga selezionato per essere eseguito (**dispatching**), durante l'esecuzione di tale processo può succedere che:

- il processo necessiti di I/O e viene inserito nella coda di un dispositivo
- il processo termini il **quanto di tempo** e venga rimosso in modo forzato dalla CPU venendo inserito di nuovo nella ready queue
- il processo crei un **figlio** e ne attenda la terminazione
- il processo si metta in attesa di un **evento**

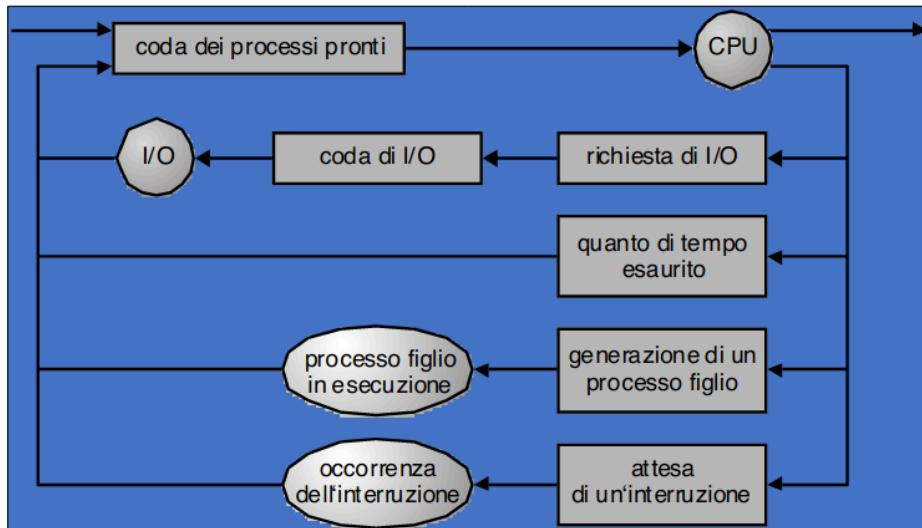
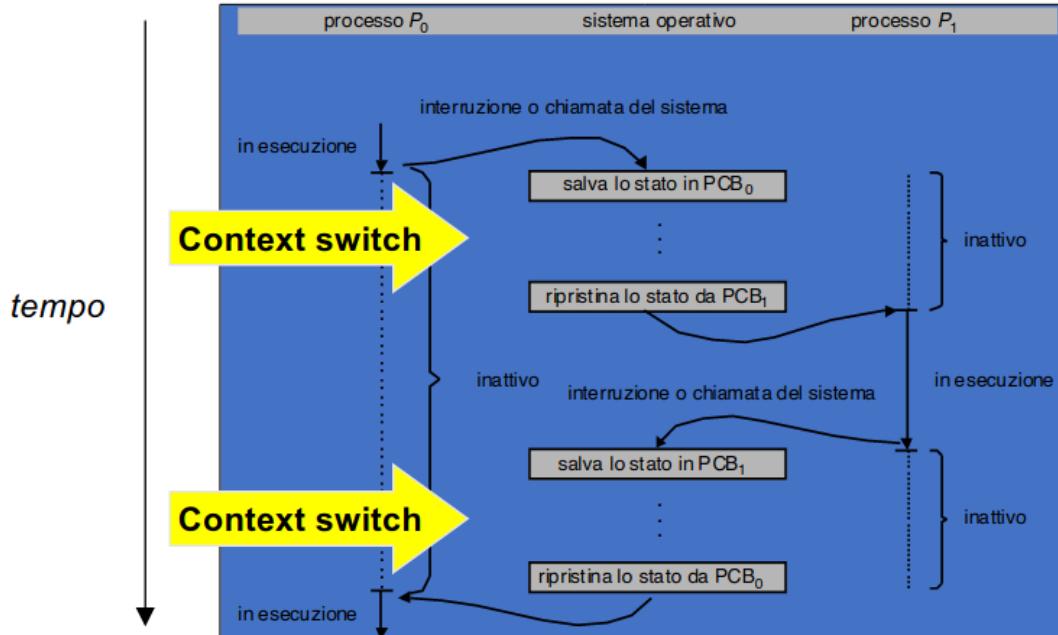


Diagramma di Accodamento

Dispatching e Context Switch

Una volta che un processo viene “dispatchato” la CPU deve effettuare un **cambio di contesto**: il PCB del processo uscente viene salvato e viene caricato il PCB del processo entrante, in seguito si passa alla modalità utente. Questo perché all’inizio della fase di dispatch il sistema si deve trovare in modalità kernel, è infatti necessario saltare all’istruzione da eseguire del processo appena entrato nella CPU (zone di memoria diverse).

Con un context switch la CPU viene “ceduta” ad un altro processo: viene registrato lo stato del vecchio processo e viene caricato lo stato precedentemente registrato del nuovo processo. Il tempo necessario al context switch è **puro sovraccarico** dato che il questo tempo il sistema non compie alcun lavoro utile durante la commutazione, la durata di tale tempo dipende strettamente dall’architettura del sistema preso in considerazione.



Operazioni e Relazioni tra Processi

Sui processi è possibile svolgere alcune operazioni, inoltre è importante ricordare che i processi sono spesso in stretta relazione tra di loro. Andando a considerare le operazioni eseguibili con i processi abbiamo a disposizione le seguenti possibilità:

Creazione di un Processo

Un processo può **creare un figlio**: in questo caso il figlio può ottenere risorse dal sistema operativo oppure dal padre per *spartizione* o *condivisione*. Ci sono a disposizione due tipi di esecuzione:

- **sincrona**: il padre attende la terminazione dei figli
- **asincrona**: il padre evolve in modo parallelo ai figli

In **UNIX** abbiamo a disposizione tre primitive per la creazione e la gestione della modalità di esecuzione:

- **fork**: crea un figlio che è un duplicato esatto del padre
- **exec**: carica sul figlio un programma diverso da quello del padre
- **wait**: permette esecuzione sincrona tra padre e figlio

```
#include <stdio.h>
void main (int argc, char *argv[])
{
    int pid;
    pid = fork();      // genera un nuovo processo
    if (pid < 0) {    // errore
        // ...
    }
}
```

```

        fprintf(stderr, "Errore di creazione");
        exit(-1);
    } else if (pid == 0) {      // codice del figlio
        execlp("/bin/ls", "ls", NULL);
    } else {                  // codice del padre
        wait(NULL); // padre attende il figlio
        printf("Figlio ha terminato.");
        exit(0)
    }
}

```

Terminazione di un Processo

Ciò avviene quando un processo termina la sua esecuzione, ciò può avvenire con la normale terminazione dell'esecuzione oppure in altre due circostanze:

- **terminazione forzata dal padre**: può avvenire nel caso in cui ci sia un eccesso nell'utilizzo delle risorse, oppure quando il compito richiesto al figlio non è più necessario, oppure ancora quando il padre termina e il sistema operativo non permette ai figli di sopravvivere al padre
- **terminazione forzata dal sistema operativo**: avviene quando l'utente chiude l'applicazione, oppure nel caso in cui si verifichino dei particolari errori

Relazioni tra Processi

Andando invece a parlare di relazioni che possono essere presenti tra i processi abbiamo due possibilità, la prima è quella di **processi indipendenti**, in tal caso si ha esecuzione deterministica dipendente unicamente dall'input e riproducibile, il processo non viene influenzato, né influenza altri processi, non c'è ovviamente condivisione dei dati con altri processi.

L'altra alternativa è quella di **processi cooperanti** nei quali un processo influenza e può essere influenzato da altri processi. L'esecuzione è inoltre non deterministica e non riproducibile. La soluzione dei processi cooperanti porta con sé molti vantaggi tra cui:

- **condivisione**
- **accelerazione del calcolo** grazie all'esecuzione parallela di subtask
- **modularità**
- **convenienza**

Concetto di Thread

Un processo unifica al suo interno due concetti:

- possesso delle risorse come spazio di memoria, file, I/O, ...
- utilizzo della CPU, caratterizzato da priorità, stato, registri, ...

Queste due caratteristiche sono tra di loro indipendenti e possono essere considerate separatamente:

- **thread** = unità minima di utilizzo della CPU
- **processo** = unità minima di possesso delle risorse

Un processo si vede assegnato lo spazio di indirizzamento e le risorse del sistema, mentre ad una singola thread sono associati:

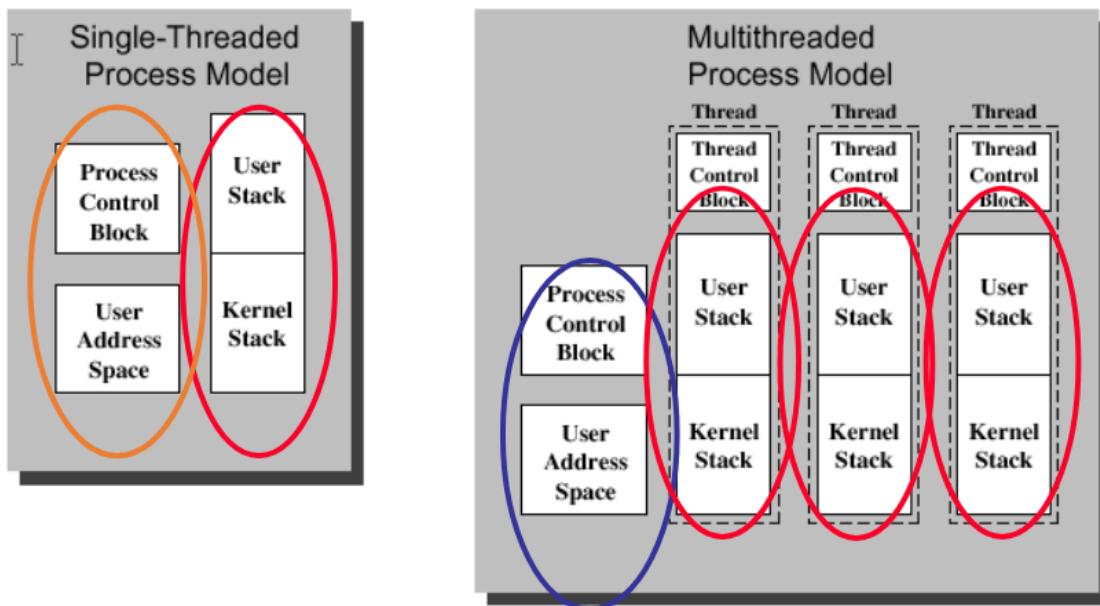
- **stato di esecuzione**
- **program counter**
- **insieme di registri**
- **stack**

Le thread condividono tra di loro spazio di indirizzamento, risorse e stato del processo

Questa breve introduzione, ci permette di definire il seguente concetto

Multithreading

Se in un sistema operativo classico 1 processo = 1 thread, con il **multithreading** si ha la possibilità di supportare più thread per un singolo processo. Di conseguenza si arriva ad una distinzione tra flusso di esecuzione e spazio di indirizzamento: un processo con thread singola vede associare un flusso esecutivo ad uno spazio di indirizzamento, mentre un processo multithreaded associa più flussi ad un singolo spazio di indirizzamento



L'introduzione delle thread comporta diversi vantaggi, il primo è la **riduzione del tempo di risposta**, dato che mentre una thread è bloccata (I/O o elaborazione lunga) un'altra thread può continuare a interagire con l'utente.

Il secondo vantaggio consiste nella **condivisione delle risorse**, dato che le thread di uno stesso processo condividono la memoria senza dover introdurre tenciche esplicite di condivisione come avviene per i processi, ciò comporta notevoli agevolazioni in termini di sincronizzazione e comunicazione.

Altro punto a favore è senza dubbio l'**economia**, infatti creazione/terminazione e context switch tra thread risultano molto più veloci che tra processi. L'ultimo importante vantaggio sta nella **scalabilità** dato che il multithreading aumenta la possibilità di parallelismo se l'esecuzione avviene su multiprocessore.

Come per i processi anche le thread hanno degli **stati**:

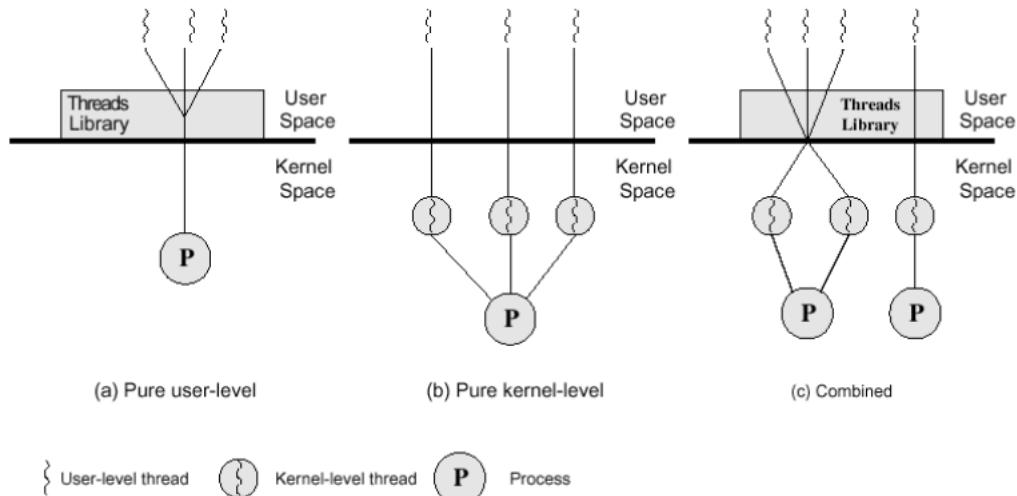
- pronta
- in esecuzione
- in attesa

In generale lo stato di un processo può non coincidere con lo stato della thread. Si verifica un problema dunque: una thread in attesa deve bloccare l'intero processo? Ciò dipende dall'**implementazione**

Implementazione delle Thread

Abbiamo diverse soluzioni per risolvere il problema dell'implementazione delle thread:

- **user level thread**: la gestione viene affidata alle applicazioni, il kernel ignora l'esistenza delle thread. Le funzionalità sono rese disponibili grazie ad una libreria di programmazione
- **kernel level thread**: la gestione è affidata in questo caso al kernel, le applicazioni utilizzano le thread tramite system call
- **approcci combinati**



Vantaggi User-Level Threads:

- Non è necessario passare alla kernel mode per utilizzare una thread: *efficienza*
- Scheduling variabile da applicazione ad applicazione
- Portabilità, dato che girano su qualunque S.O. senza dover modificare il kernel

Svantaggi User-Level Threads:

- Blocco di una thread blocca l'intero processo, superabile tramite accorgimenti specifici
- Non è possibile sfruttare multiprocessore, dato che lo scheduling di una thread avviene sempre sullo stesso processore

Vantaggi Kernel-Level Threads:

- Scheduling a livello di thread, dunque il blocco di una thread non è bloccante
- Più thread dello stesso processo possono essere eseguite in contemporanea su CPU differenti
- Le funzioni del sistema operativo stesso possono essere gestite in modalità multithreaded

Svantaggi Kernel-Level Threads:

- Scarsa efficienza, dato che il passaggio tra thread implica un passaggio attraverso il kernel

Gestione dei Processi del S.O.

Il sistema operativo è un programma a tutti gli effetti. Diventa abbastanza naturale chiedersi se tale S.O. in esecuzione può essere considerato un processo. Ci sono diverse opzioni:

- kernel eseguito separatamente
- kernel eseguito all'interno di un processo utente
- kernel eseguito come processo

Kernel Separato

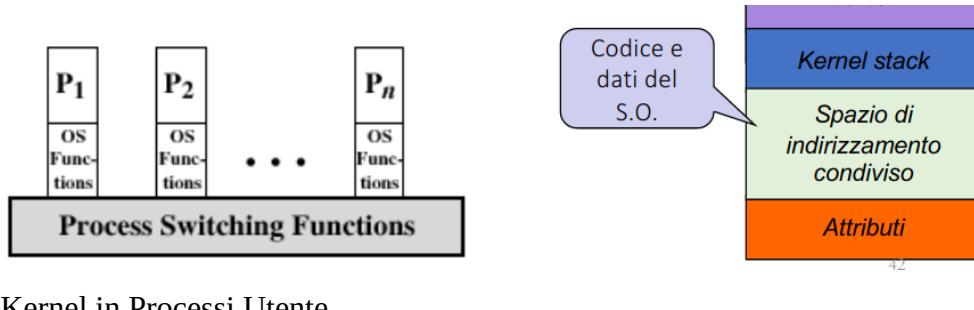
Il kernel esegue al di fuori di ogni processo, il sistema operativo possiede infatti uno spazio riservato in memoria. Il sistema operativo prende il controllo del sistema e viene sempre eseguito in maniera privilegiata. Il concetto di processo può essere applicato soltanto ai processi utente. Questo approccio è tipico dei primi sistemi operativi.

Kernel in Processi Utente

I servizi del sistema operativo sono procedure chiamabili da programmi utente accessibili in modalità kernel. L'immagine dei processi prevede:

- **kernel stack** per gestire il funzionamento di un processo in modalità protetta
- **codice/dati del sistema operativo** condiviso tra processi utente

Questo approccio porta con sé alcuni vantaggi, ad esempio in occasione di interrupt o trap durante esecuzione di un processo utente serve soltanto un mode switch. Dopo il completamento del suo lavoro il sistema operativo può decidere di riattivare lo stesso processo utente (mode switch) oppure un altro (context switch)

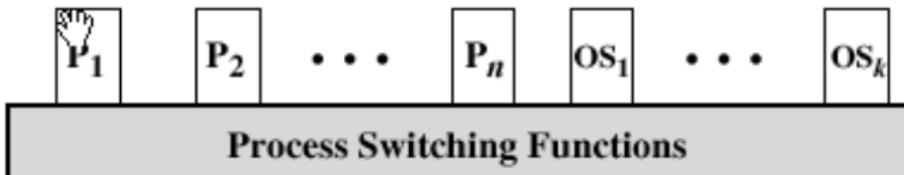


Kernel in Processi Utente

Kernel come Processo

I servizi del sistema operativo diventano processi individuali eseguiti in modalità protetta. In questo caso una minima parte del sistema operativo deve comunque eseguire al di fuori di tutti i processi, lo scheduler ad esempio.

Si tratta di una soluzione vantaggiosa per sistemi multiprocessore dove processi del sistema operativo possono essere eseguiti su processore ad hoc



Kernel come Processo

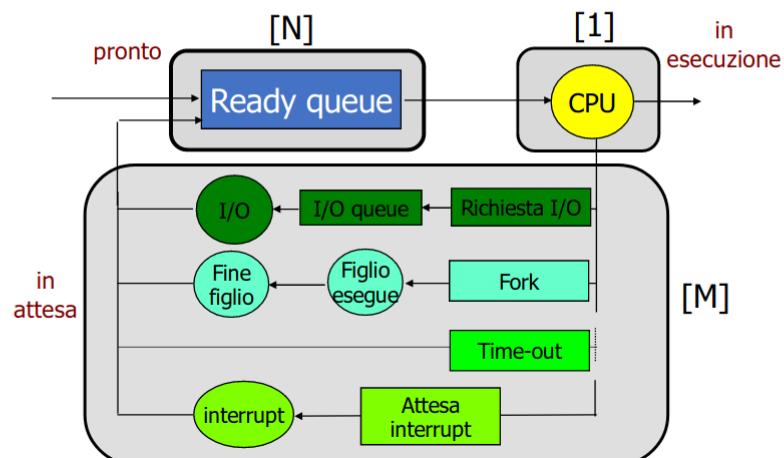
Capitolo 5 - Scheduling

Federico Segala

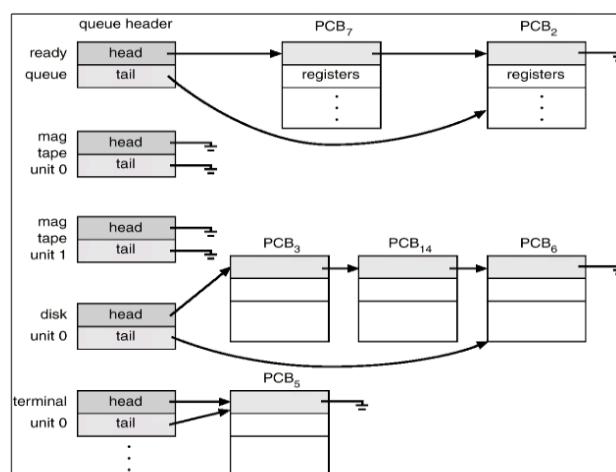
Concetto di Scheduling

Lo **scheduling*** è il processo di assegnazione di attività nel tempo. L'utilizzo della multiprogrammazione impone necessariamente l'esistenza di una strategia per regolamentare l'*ammissione* dei processi nel sistema (*memoria*) e l'ammissione dei processi all'esecuzione (*CPU*)

Diagramma di accodamento



Implementazione delle code



Tipi di Scheduling

Ci sono diversi tipi di scheduler, ognuno con le proprie caratteristiche. Il primo tipo di scheduler introdotto è lo scheduler a **lungo termine**, anche detto *job scheduler*, che è colui che si occupa di selezionare quali processi devono essere portati dalla memoria alla ready queue.

Altro tipo di scheduler è quello a **breve termine** che seleziona quale processo deve essere eseguito. Questo tipo di scheduler viene invocato molto spesso, deve quindi essere molto veloce.

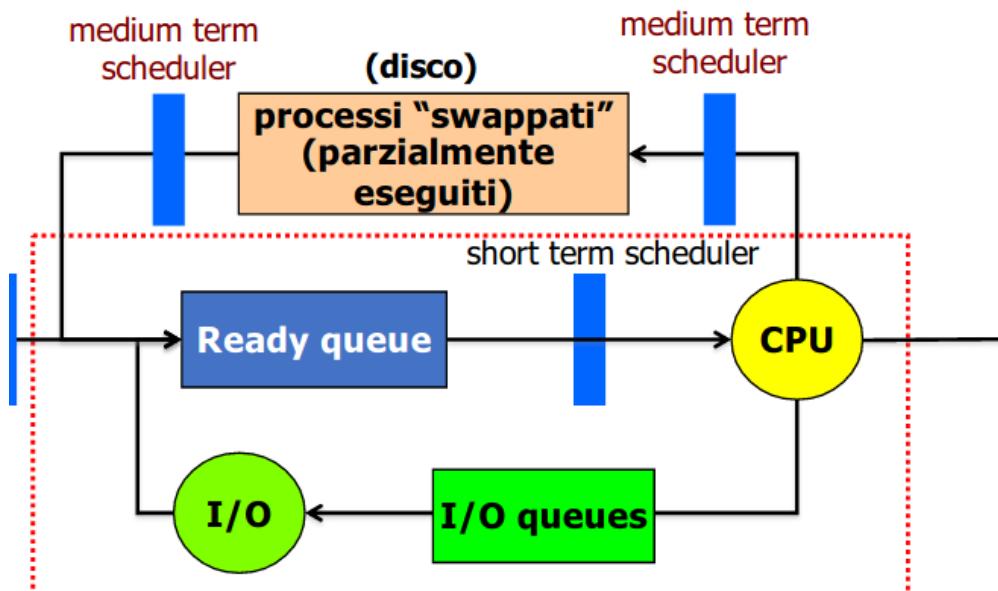
A differenza di quello a breve termine, lo scheduler a lungo termine non ha particolari vincoli sulle prestazioni, di fatto è colui che si occupa di controllare il grado di multiprogrammazione e il mix di processi attivi all'interno del sistema. Tali processi, in base alle loro caratteristiche possono essere:

- **I/O bound**: processi in cui è molto frequente la necessità di input/output, molti burst di CPU brevi
- **CPU bound**: molti calcoli, pochi burst di CPU molto lunghi

Lo scheduler a lungo termine può anche essere assente, è tipicamente utilizzato in sistemi a basso quantitativo di risorse

Scheduling a Medio Termine

Nei sistemi operativi con **memoria virtuale** è previsto un livello intermedio di scheduling detto a **medio termine** che serve alla momentanea rimozione forzata di unprocesso dalla CPU. Questa pratica viene effettuata per ridurre in via temporanea il grado di multiprogrammazione.



Scheduler a medio termine

Scheduling della CPU

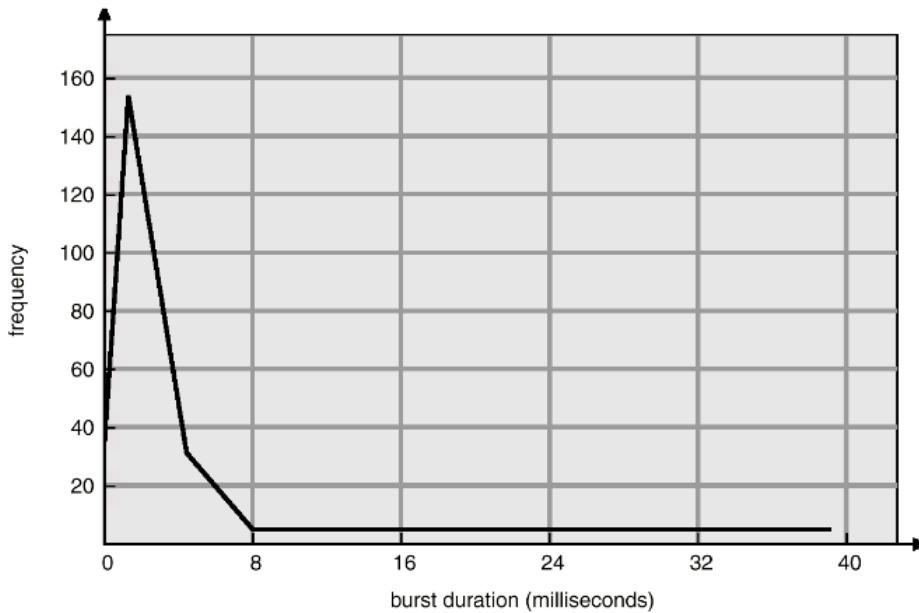
Il **CPU scheduler** è un modulo del sistema operativo che seleziona un processo tra quelli in memoria pronti per l'esecuzione e gli alloca la CPU. Data la frequenza di invocazione è una **componente critica** del sistema: sono necessari algoritmi di scheduling.

Il **dispatcher** è un modulo del sistema operativo che passa il controllo dalla CPU al processo scelto dallo scheduler, quando entra in azione, vengono svolte le azioni seguenti:

- context switch
- passaggio in modalità user
- salto alla opportuna locazione nel programma per farlo ripartire

Definiamo **latenza di dispatch** il tempo necessario al dispatcher per fermare un processo e farne partire un altro, questo tempo deve essere il più basso possibile.

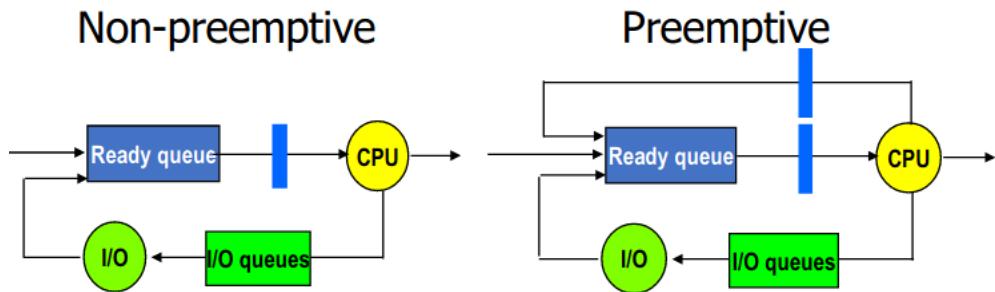
L'esecuzione di un processo consiste astrattamente nell'alternanza ciclica di un burst di CPU e di un burst di I/O. La distribuzione dei CPU burst è una distribuzione **esponenziale** nel senso che più sono numerosi, più i burst tendono ad essere brevi e viceversa



Distribuzione Esponenziale CPU Burst

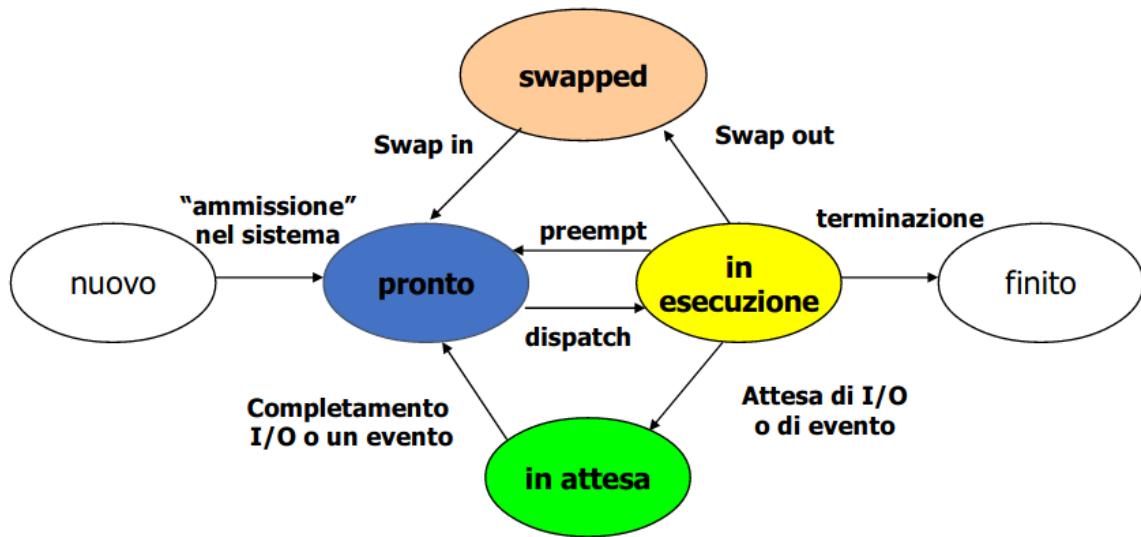
Prelazione

Con **prelazione** definiamo il rilascio forzato della CPU. Lo scheduling può essere fatto **con o senza prelazione**. Nel primo caso il processo che detiene la CPU non la rilascia fino al termine del burst, nel secondo caso un processo può essere forzato a rilasciare la CPU prima del termine del burst



Non-Preemptive vs. Preemptive

Vediamo a questo punto uno schema complessivo del **diagramma degli stati di un processo**:



Metriche di Scheduling

Come già detto l’obiettivo dello scheduling è quello di massimizzare l’utilizzo della CPU. Definiamo **throughput** il numero di processi completati per unità di tempo.

Definiamo invece **tempo di attesa** t_w la quantità totale di tempo spesa da un processo nella coda di attesa ed è influenzato dall’algoritmo di scheduling.

Definiamo il **tempo di completamento** t_t (tempo di turnaround) il tempo necessario ad eseguire un particolare processo dal momento della sottomissione al momento del completamento.

Definiamo **tempo di risposta** t_r il tempo trascorso da quando una richiesta è stata sottosposta al sistema fino alla prima risposta del sistema stesso.

Nell’ottimizzazione dei nostri algoritmi di scheduling vorremo andare a **massimizzare**

- utilizzo della cpu
- throughput

e andare invece a **minimizzare**

- tempo di turnaround
- tempo di attesa
- tempo di risposta

Algoritmi di Scheduling

First-Come, First-Served

Il concetto alla base di questo algoritmo è il seguente: la coda dei processi è una coda FIFO, il primo processo arrivato è il primo ad essere servito. Il grosso vantaggio di questo tipo di implementazione è chiaramente la **semplicità** di implementazione. Il grosso svantaggio dato da questo algoritmo è l'**effetto convoglio**: processi brevi si accodano ai processi lunghi precedentemente arrivati

Shortest-Job-First

Questo algoritmo associa ad ogni processo la lunghezza del prossimo burst di CPU, il processo con il burst di CPU più breve viene poi selezionato per l'esecuzione. Abbiamo a disposizione due schemi:

- **non preemptive**
- **preemptive**: se arriva un nuovo processo con un burst di CPU più breve del tempo che rimane da eseguire, quest'ultimo viene rimosso dalla CPU per fare spazio a quello appena arrivato → **shortest-remaining-time-first** (SRTF)

SJF è un algoritmo ottimo dato che porta al tempo medio di attesa minimo.

Il calcolo del prossimo burst di CPU è però un calcolo di fatto impossibile da fare, è possibile farne soltanto una **stima** utilizzando le lunghezze dei burst precedenti come proiezione di quelli futuri, utilizzando la media esponenziale:

- t_n = lunghezza reale dell'n-esimo burst
- τ_{n+1} = valore stimato per il prossimo burst
- α = coefficiente ($0 < \alpha < 1$)
- $\tau_{n+1} = \alpha * t_n + (1 - \alpha) * \tau_n$

Scheduling a Priorità

Viene associata una priorità a ogni processo. La CPU viene allocata al processo con priorità più alta. Anche in questo caso abbiamo la possibilità di algoritmi preemptive e non-preemptive. Ad esempio SJF è un algoritmo a priorità dove tale priorità è data da $1/\tau_{n+1}$

L'assegnamento delle priorità può avvenire in base a diverse politiche che possono essere interne (limiti di tempo, requisiti di memoria, ...) o esterne (importanza del processo, ...) al sistema operativo

Questo tipo di scheduling può portare al problema della **starvation**: processi a bassa priorità non possono mai essere eseguiti. Una soluzione a ciò si ottiene implementando l'**aging**, ovvero l'aumento della priorità di un processo con il passare del tempo.

Highest-Response-Ratio-Next

Si tratta di un algoritmo a priorità non-preemptive in cui la priorità $R = 1 + t_{attesa} / t_{burst}$. Tale R è dinamica in quanto dipende anche dal tempo di attesa di un processo e può essere ricalcolata al termine di un processo solo se nel frattempo ne sono arrivati altri oppure, al termine di qualsiasi processo.

Tramite questa politica sono favoriti i processi che completano in poco tempo oppure che hanno atteso per molto tempo, superando il favoritismo di SJF nei confronti dei job corti

Round-Robin

Si tratta di uno scheduling basato su **time-out**: a ogni processo viene assegnata una piccolaparte del tempo di CPU. Al termine di questo quanto, il processo è prelazionato e messo nella ready queue (coda circolare).

Nel caso ad esempio in cui ci siano n processi nella coda e un quanto di tempo q ogni processo ottiene $1/n$ del tempo di CPU in blocchi di q unità di tempo alla volta. Nessun processo attende più di $(n - 1)q$ unità di tempo.

Si tratta di un algoritmo intrinsecamente preemptive, in pratica implementa un FCFS con prelazione. Diventa importantissima la **scelta del quanto**:

- q grande → tendenza a FCFS
- q piccolo → context switch troppo frequente

Un valore ragionevole di q è da scegliere facendo in modo che l'80% dei burst di CPU siano $< q$

A livello di prestazioni, il tempo di turnaround è sempre maggiore o uguale a quello in SJF, mentre il tempo di risposta è minore o uguale a quello di SJF.

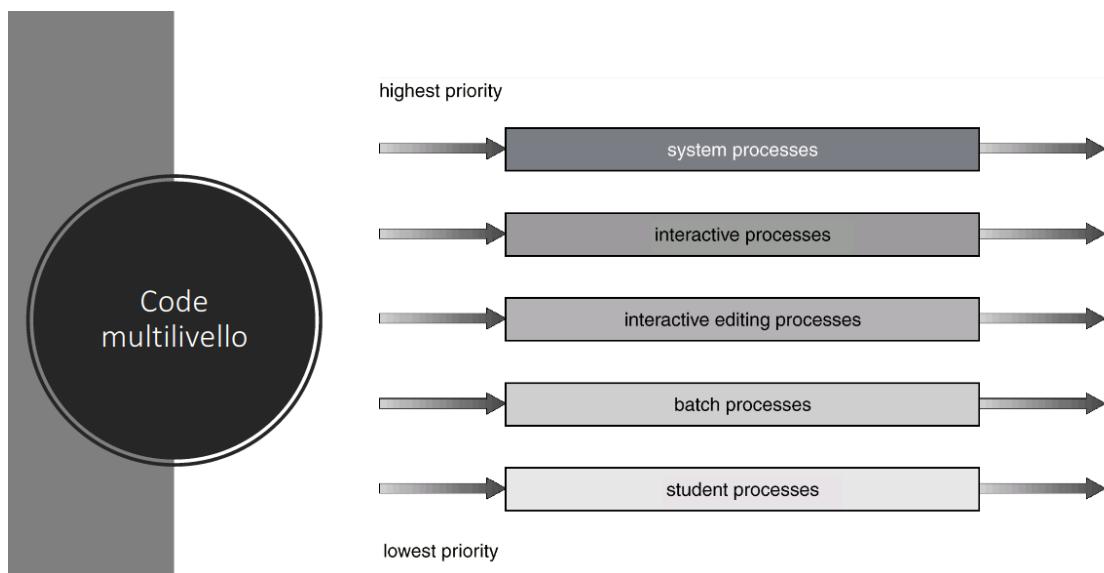
Mentre il context switch aumenta sempre al diminuire del quanto, il tempo di turnaround non diminuisce sempre all'aumentare del quanto.

Code Multilivello

Esistono classi di algoritmi in cui la **ready queue è partizionata in più code** ad esempio una coda per job in foreground (interattivi) e una coda invece per i job in background (batch). Ogni coda ha il suo algoritmo di scheduling. Ad esempio per i job in foreground è possibile implementare round robin, mentre in background utilizzare FCFS potrebbe essere una buona idea.

Si tratta di un meccanismo più generale ma di gestione più complessa, infatti si rende necessario anche uno **scheduling tra le code**:

- *priorità fissa*: ad esempio si può pensare di servire prima tutti i job di sistema, poi quelli in foreground, poi quelli in background. In questo caso si avrebbe possibilità di starvation per code a priorità bassa
- *time slice based*: ogni coda ottiene un quanto del tempo di CPU che può usare per schedulare i suoi processi



Code Multilivello con Feedback

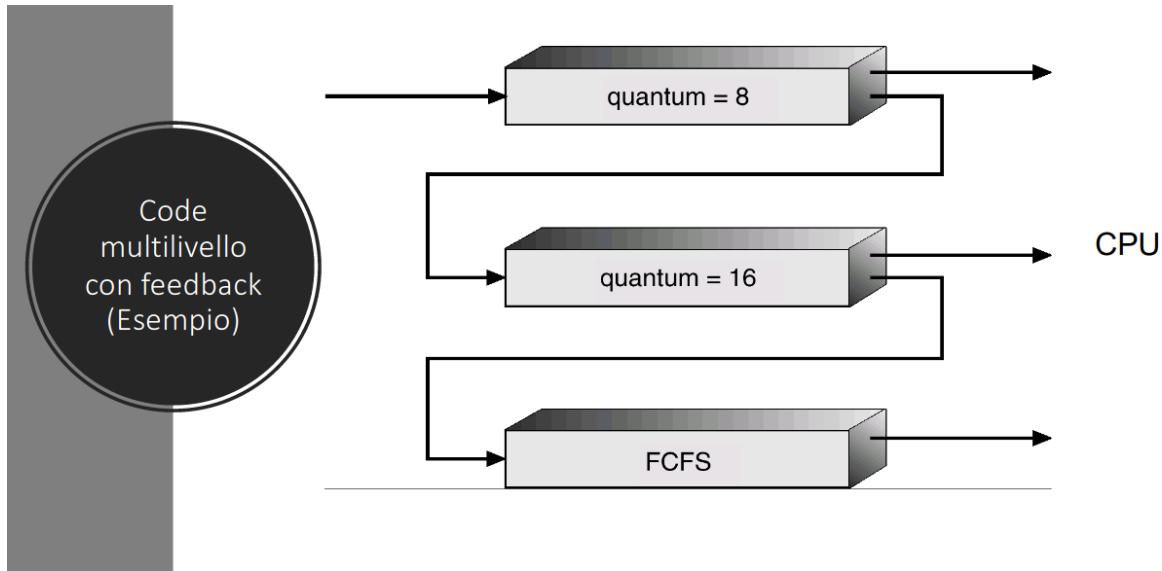
Nelle code multilivello classiche un processo viene allocato definitivamente ad una coda, in questo tipo di approccio un processo può spostarsi da una coda all'altra a seconda delle sue caratteristiche. Questo approccio viene utilizzato anche per implementare l'**aging**. Lo scheduler utilizza come parametri il numero di code, gli algoritmi utilizzati per ogni coda, i criteri per la promozione o la degradazione di un processo e i criteri per definire la coda di ingresso di un processo.

Ad esempio ipotizziamo di avere 3 code:

- Q_0 : RR con quanto da 8ms
- Q_1 : RR con quanto da 16 ms
- Q_2 : FCFS

La CPU serve nell'ordine 0,1,2, i processi in Q_j vengono serviti se e solo se Q_i è vuota $\forall i < j$. Funzionamento:

- un nuovo job entra in Q_0 . Quando ottiene la CPU riceve 8ms di quanto. Se non finisce entro in quanto, viene prelazionato e degradato alla coda Q_1
- se Q_0 è vuota, si seleziona un job di Q_1 che riceve 16ms di quanto. Se non finisce viene prelazionato e messo in Q_2
- se Q_0 e Q_1 sono vuote, viene selezionato un job in Q_2 con FCFS



Scheduling Fair Share

Le politiche di scheduling precedenti sono orientate al processo, ma non tengono conto del fatto che un'applicazione possa essere composta di più processi. **Fair Share** cerca di fornire **equità** alle applicazioni e non ai singoli processi, suddividendo le risorse tra **gruppi di processi**

In un contesto reale, l'obiettivo è sempre quello di minimizzare la complessità degli algoritmi, dunque gli algoritmi reali in genere utilizzano prelazione e sono basati su round robin.

Valutazione degli Algoritmi

Esistono varie tecniche che verranno illustrate in seguito per andare a valutare i vari algoritmi di scheduling:

- ## Modello deterministico Si basa sull'algoritmo e su un preciso carico di lavoro. Va a definire le prestazioni di ogni algoritmo per uno specifico carico di lavoro, dunque le risposte sono applicabili unicamente al caso considerato. Di solito viene utilizzato per illustrare gli algoritmi. Richiede però conoscenze troppo specifiche sulla natura dei processi

Modello a reti di code

Non esiste un preciso gruppo di processi sempre uguali per utilizzare il modello deterministico, è però possibile determinare le distribuzioni di CPU e I/O burst. Il sistema di calcolo è descritto come una rete di server ognuno con la propria coda

Simulazione

Si rende in questo caso necessario programmare un modello del sistema, utilizzando dati statistici o reali. Si tratta di una tecnica molto costosa ma abbastanza precisa

Capitolo 6 - Sincronizzazione

Federico Segala

Introduzione

Supponiamo di avere il seguente problema di sincronizzazione tra processi, ovvero il problema del **produttore-consumatore**:

- il produttore produce un messaggio
- il consumatore consuma tale messaggio

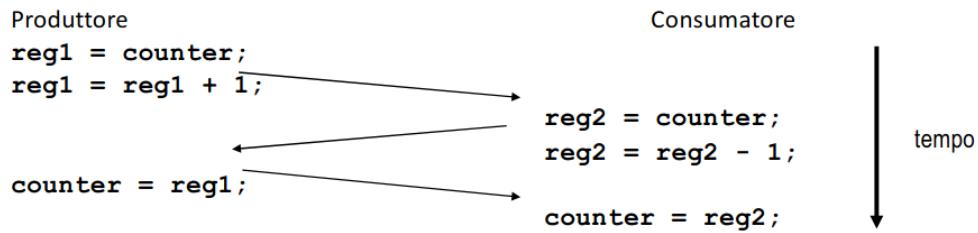
L'esecuzione di produttore e consumatore è concorrente: il produttore aggiunge al buffer e il consumatore toglie da tale buffer. Se tale buffer è pieno non è possibile aggiungere elementi, al contrario se tale buffer è vuoto non è possibile consumare elementi.

Consideriamo ad esempio un buffer circolare a N posizioni, in cui `in` identifica la prima posizione libera, mentre `out` identifica la prima posizione occupata. Nel momento in cui il buffer è vuoto avremo `in = out`. Se il buffer è pieno abbiamo invece `out = (in + 1) % n`

Per semplicità utilizzeremo una variabile `counter` per indicare il numero di elementi nel buffer.

Il problema consiste nel fatto che `counter += 1` e `counter -= 1` sono implementate in istruzioni assembly non atomiche, dunque lo scheduler potrebbe portare a una situazione del tipo seguente:

- Possibile un'esecuzione alternata come segue:



- Inconsistenza!
- Es.: supponiamo counter = 5
 - P produce un item → counter++ (idealmente counter diventa 6)
 - C consuma un item → counter-- (idealmente counter ritorna 5)
 - Quanto vale counter? 4 invece di 5!

Il problema consiste nel fatto che **P e C possono modificare counter in modo contemporaneo**. L'obiettivo di questo capitolo sarà studiare delle tecniche che permettano di proteggere l'accesso alla **sezione critica**

Sezione Critica

Definiamo **sezione critica** una porzione di codice in cui si accede ad una **risorsa condivisa**. La soluzione al problema della sezione critica deve necessariamente rispettare tre criteri:

- mutua esclusione**: un processo alla volta può accedere alla sezione critica
- progresso**: solo i processi che stanno per entrare in sezione critica possono decidere chi entra, tale decisione non può essere rimandata all'infinito
- attesa limitata**: deve esistere un massimo numero di volte per cui un processo può entrare consecutivamente in tale sezione

Per risolvere questo problema abbiamo a disposizione:

- soluzioni software**: aggiunta di codice alle applicazioni, senza alcun supporto hardware o del sistema operativo
- soluzioni hardware**: viene aggiunto codice alle applicazioni, coadiuvato da supporto hardware

Soluzioni Software alla Sezione Critica

Algoritmo 1

Processi i;

```
int turn; // se turn = i allora entra il processo i
```

```

while (1) {
    while (turn != i);      // sezione di entrata
    /* sezione critica */
    turn = j;      // sezione di uscita
    /* sezione non critica */
}

```

Il **problema** di questo algoritmo consiste nel fatto che è richiesta stretta alternanza tra processi: quando i o j non sono interessati ad entrare in SC, anche l'altro processo non può entrare in SC. Non è rispettato il criterio del **progresso**

Algoritmo 2

```

Process i;

boolean flag[2];      /* inizializzato a FALSE */

while (1) {
    flag[i] = true;      // il processo i vuole entrare in SC
    while (flag[j] == true);      // sezione di entrata
    /* sezione critica */
    flag[i] = false;      // sezione di uscita
    /* sezione non critica */
}

```

Tale algoritmo risolve il problema dell'algoritmo 1 ma l'esecuzione in sequenza di dell'istruzione `flag[] = true` da parte dei due processi porta a **deadlock**:

- t₀: P0 esegue `flag[0] = True
- t₁: P1 esegue `flag[1] = True
- t₂: P0 esegue `while (flag[1] == True);`
- t₃: P1 esegue `while (flag[0] == True);`
- t₄: **DEADLOCK**

Possiamo provare a scrivere una variante di tale algoritmo invertendo le istruzioni della sezione di entrata:

```

Process i;

boolean flag[2];      /* inizializzato a FALSE */

while (1) {
    while (flag[j] == true);      // sezione di entrata
    flag[i] = true;      // il processo i vuole entrare in SC
    /* sezione critica */
}

```

```

    flag[i] = false; // sezione di uscita
    /* sezione non critica */
}

```

Il **problema** a questo punto diventa che invertendo le istruzioni della sezione di entrata viene violata la **mutua esclusione**: entrambi i processi possono trovarsi in SC se eseguono in sequenza il while prima di impostare il flag a True.

Algoritmo 3

```

Process i;

int turn;
boolean flag[2]; /* inizializzato a FALSE */
while (1) {
    flag[i] = True; // processo i vuole entrare
    turn = j;         // tocca a te se vuoi
    while (flag[j] == True && turn == j);
    /* sezione critica */
    flag[i] = False;
    /* sezione non critica */
}

```

Si può notare come questa sia la **soluzione corretta**, infatti entra il primo processo che esegue `turn = j` oppure `turn = i`. La **mutua esclusione** è infatti garantita nel seguente modo:

- P_i entra nella SC se e solo se `flag[i] = false` o `turn = i`
- Se P_i e P_j sono entrambi in SC allora `flag[i] = flag[j] = True`
- Ma P_i e P_j non possono aver superato entrambi il while dato che `turn` vale i oppure j
- Soltanto uno dei due processi è quindi entrato.

Il **progresso** e l'**attesa limitata** sono garantiti, infatti:

- Se P_j non è pronto ad entrare nella SC allora `flag[j] = false` e P_i può entrare (esce dal ciclo while)
- Se P_j ha impostato `flag[j] = true` e si trova nel while allora `turn = i` oppure `turn = j` (vedi prima)
- Se `turn = i` entra P_i , viceversa se `turn = j` entra P_j
- In ogni caso quando P_j esce dalla SC imposta `flag[j] = false` e quindi P_i può entrare
- Abbiamo quindi che P_i entra nella SC al massimo dopo un'entrata di P_j

Algoritmo del Fornaio

Gli algoritmi proposti in precedenza risolvono il problema della sezione critica nel momento in cui siano presenti soltanto **due processi** nel sistema preso in considerazione, tramite l'**algoritmo del fornaio** si vuole risolvere tale problema con N processi.

L'idea alla base dell'algoritmo è che ogni processo sceglie un numero. Il numero più basso verrà servito per primo. Per situazioni di numero identico, che può accadere, si utilizza un confronto a due livelli (*numero pescato, id processo*)

È dimostrabile che questo è l'**algoritmo corretto**

Process i;

```
boolean choosing[N];      // P_i sceglie un numero      /* inizializzato a
                           FALSE */
int number[N];           // ultimo numero scelto      /* inizializzato a
                           0 */

while (1){
    // processo i prende un numero
    choosing[i] = True;
    number[i] = max(number[0], ..., number[N-1]) + 1;
    choosing[i] = False;

    for (j = 0; j < N; j++) {
        while (choosing[j] == True);      // il processo j sta
                                           scegliendo
        while (number[j] != 0 && number[j] < number[i]);      // j è in
                                                       SC e ha un numero inferiore
    }
    /* sezione critica */
    number[i] = 0
    /* sezione critica */
}
```

Soluzioni Hardware alla Sezione Critica

Un modo hardware per risolvere il problema della sezione critica consiste nel disabilitare gli interrupt mentre una variabile condivisa viene modificata. Il problema è che se il test per l'accesso è troppo lungo, gli interrupt devono essere disabilitati per troppo tempo.

L'alternativa a ciò è che l'operazione per l'accesso ad una risorsa deve occupare un unico ciclo di istruzione non interrompibile. Ciò si ottiene tramite delle **istruzioni atomiche**:

- **test & set**

- swap

Test & Set

```
bool TestAndSet (boolean * var)
{
    boolean temp;
    temp = *var;
    *var = True;
    return temp;
}
```

Il valore di ritorno di tale funzione è il **vecchio valore di var**, e tale funzione **assegna True a var**. Vediamo un esempio di utilizzo di questa istruzione atomica:

```
boolean lock; /* globale inizializzata a FALSE */

while (1) {
    while (TestAndSet(&lock));
    /* sezione critica */ // passa solato il primo processo che
                           // arriva e trova lock = FALSE
    lock = False;
    /* sezione no critica */
}
```

Swap

```
void Swap (boolean * a, boolean * b)
{
    boolean temp;
    temp = *a;
    *a = *b;
    *b = temp;
}
```

Questa istruzione si basa sullo scambio dei valori di a e b. Vediamo un esempio di utilizzo di tale istruzione atomica.

```
boolean lock; /* globale inizializzata a FALSE */

while (1) {
    dummy = True;
    do {
        Swap(&dummy, &lock); // quando dummy = flase, P_i accede a
                           // SC
```

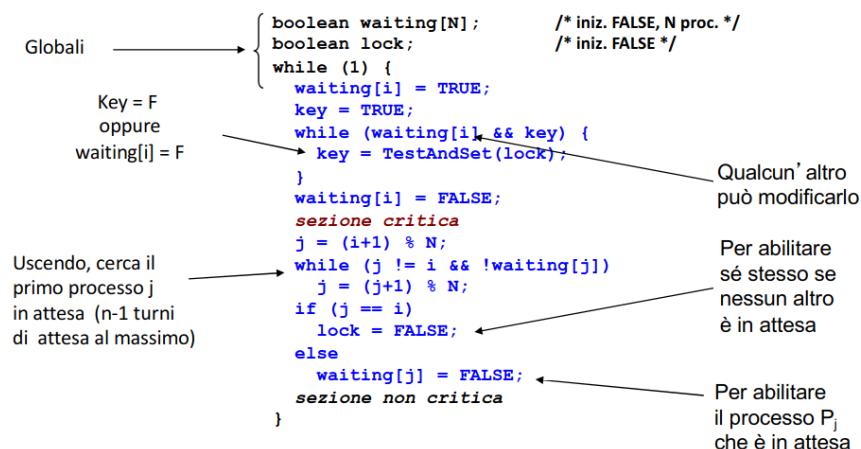
```

} while (dummy == True);
/* sezione critica */
lock = False;           // gli altri processi continuano a
                        // scambiare true con true, finché
/* sezione non critica */ // P_i non pone lock = false
}

```

Il problema consiste nel fatto che **TestAndSet** e **Swap** non rispettano attesa limitata. Si può infatti notare come manchi una variabile equivalente a **turn**. Sono quindi necessarie variabili addizionali

Test and Set con attesa limitata



Test and Set Attesa Limitata

I vantaggi di queste soluzioni, consistono innanzitutto in una grande **scalabilità** dato che sono indipendenti dal numero di processi coinvolti. Gli svantaggi introdotti sono però una **maggior complessità** rispetto alle soluzioni SW. Per imporre l'attesa limitata alle soluzioni è necessario **busy waiting** che spreca prezioso tempo di CPU

Semafori

Le soluzioni introdotte in precedenza sono abbastanza complesse da aggiungere ai programmi e sono basate, come già detto, su busy waiting. I **semafori** sono una alternativa generica che funziona sempre.

Un **semaphore** è una **variabile intera S** a cui si accede per mezzo di due **primitive atomiche**:

- **Signal (V(s))** → incrementa il valore di S di 1
- **Wait (P(s))** → tenta di incrementare il valore di S di 1, tenta, per il fatto che se $S = 0$, allora non è possibile decrementare il semaforo, è necessario **attendere**

I semafori possono essere **binari** oppure **generici**, la distinzione è chiaramente data dal dominio di valori che assume la variabile S.

Semafori - Implementazione

Di seguito una implementazione **concettuale** dei semafori appena descritti.

Andiamo a vedere di seguito, una **implementazione reale** di questi semafori:

Semafori binari – implementazione

- Con busy waiting

```
/* s inizializzato a TRUE */
P(bool &s)
{
    key = FALSE;
    do {
        Swap(s, key);
    } while (key == FALSE);
}

V(bool &s)
{
    s = TRUE;
}
```

Implementazione Semafori Binari con Busy Waiting

Semafori interi – implementazione

- Con busy-waiting

```
bool mutex; /* Sem. binario iniz. TRUE */
bool delay; /* Sem. intero iniz. FALSE */

P(int &s)
{
    P(mutex);
    s = s - 1;
    if (s < 0) {
        V(mutex);
        P(delay);
    } else V(mutex);
}

V(int &s)
{
    P(mutex);
    s = s + 1;
    if (s <= 0) {
        V(delay);
    }
    V(mutex);
}
```

P, V = semaforo intero P, V = semaforo binario

implementazione Semafori Interi con Busy Waiting

Semafori interi – implementazione

- Senza busy-waiting

```
bool mutex /*sem bin. inizializzato a true*/
```

```
typedef struct {
    int value;
    PCB *List;
} Sem;
```

Diagramma di struttura del semaforo:

value
list

list → PCB → PCB

Etichetta: Sem

```
P (Sem &s) {
    P(mutex);
    s.value = s.value - 1;
    if (s.value < 0) {
        V(mutex);
        append (process i, s.List);
        sleep();
    } else
        V(mutex);
}
```

Diagramma di flusso per la operazione P:

- Senza busy waiting?
- Mette il processo nello stato waiting

P semaforo intero - no busy waiting

Semafori interi – implementazione

- Senza busy-waiting

```
bool mutex /*sem bin. inizializzato a true*/
```

```
typedef struct {
    int value;
    PCB *List;
} Sem;
```

Diagramma di struttura del semaforo:

value
list

list → PCB → PCB

Etichetta: Sem

```
V (Sem &s) {
    P(mutex);
    s.value = s.value + 1;
    if (s.value <= 0) {
        V(mutex);
        PCB *p = remove(s.List);
        wakeup(p);
    } else
        V(mutex);
}
```

Diagramma di flusso per la operazione V:

- Senza busy waiting?
- Mette il processo nello stato ready. In che ordine?

V semaforo intero - no busy waiting

Possiamo notare come il **busy waiting** sia stato **eliminato dalla entry section** dato che questa può essere lunga. Il busy waiting rimane invece nelle P e nella V di mutex, dato che questa modifica è tendenzialmente molto veloce, comporta poco spreco. In alternativa è possibile disabilitare gli interrupt durante P e V, ma in questo modo non è possibile interfogliare istruzioni di processi differenti.

È inoltre importante notare come l'implementazione reale sia diversa da quella concettuale: il valore della variabile s di un semaforo può diventare < 0 , quando raggiunge valori negativi, conta quanti processi sono in attesa di accedere ad una particolare sezione critica. La lista dei PCB può essere FIFO, che garantisce attesa limitata.

Semafori - Utilizzi Principali

Un semaforo binario con valore iniziale = 1 è utilizzato come **mutex**, serve cioè a proteggere la sezione critica per n processi. Un semaforo binario con valore iniziale = 0 è invece utilizzato per la **sincronizzazione** tra i processi.

Un classico esempio di gestione della **mutua esclusione** sulla sezione critica è il seguente:

- Mutex = semaforo binario di mutua esclusione
- N processi condividono la variabile S

```
/* valore iniziale di s = 1 (mutex) */
while (1) {
    P(s);
    sezione critica
    V(s);
    sezione non critica
}
```

Semaforo Binario - Mutua Esclusione

Un altro tipico esempio è quello di semafori utilizzato per l'**attesa di un evento**, come nel caso seguente:

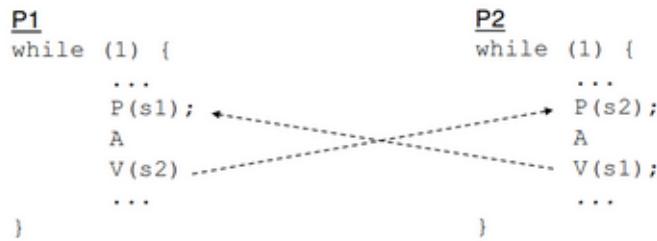
- Sincronizzazione generica
 - Processi P1 e P2 devono sincronizzarsi rispetto all'esecuzione di due operazioni A e B
 - P2 può eseguire B soltanto dopo che P1 ha eseguito A
 - Soluzione: uso di un semaforo binario s inizializzato a 0



Semaforo Binario - Attesa di un Evento

Un altro esempio di **attesa di un evento** è il seguente caso:

- Sincronizzazione generica
 - P1 e P2 devono sincronizzarsi rispetto all'esecuzione di un'operazione A
 - Utilizzo di A: P1 \Rightarrow P2 \Rightarrow P1 \Rightarrow P2 \Rightarrow ...
- Soluzione: Uso di due semafori binari: S1 inizializzato a 1 e s2 inizializzato a 0



Semafori - Problemi

Uno dei principali problemi dovuti all'utilizzo dei semafori consiste nella possibilità di **deadlock**: un processo viene bloccato in attesa di un evento che soltanto lui, o un altro processo in attesa può generare.

Un altro possibile problema è quello della **starvation**, ovvero l'attesa indefinita all'interno di un semaforo.

Assieme ai loro vantaggi, i semafori portano anche delle forti limitazioni, la prima e forse più importante consiste nella **difficoltà di scrittura dei programmi**.

Monitor

Si tratta di costrutti per la condivisione sicura ed efficiente di dati tra i vari processi, si tratta di un concetto molto simile a quello di **classe**. La struttura di un monitor è la seguente:

```

Monitor xyz {
    // dichiarazione di variabili (stato del monitor)

    entry P1 (...) {
        ...
    }

    entry P2 (...) {
        ...
    }

    {
        // codice di inizializzazione
    }
}
  
```

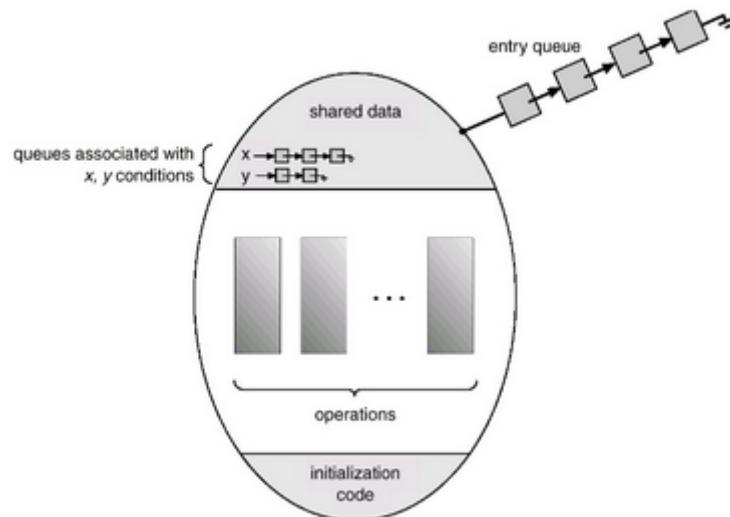
Le variabili di un monitor sono visibili unicamente all'interno del monitor stesso. Le procedure del monitor accedono soltanto alle variabili definite nel monitor. In ogni istante è possibile che all'interno di un monitor sia attivo un solo processo: il programmatore **non deve codificare esplicitamente la mutua esclusione***.

Per permettere ad un processo di attendere all'interno di un monitor viene fatto utilizzo delle **variabili condition**, che sono dichiarate all'interno del monitor tramite la sintassi `condition x, y;`. Queste variabili sono accessibili tramite delle *primitive* come nei semafori:

- **wait**
- **signal**

Il processo che invoca `x.wait()` viene bloccato fino all'invocazione della corrispondente `x.signal()` da parte di un altro processo.

Un monitor si può vedere un po' come se fosse un *uovo*, nella parte superiore vediamo le code di processi in attesa legate alle condizioni `x, y, ...`, nella parte centrale abbiamo le operazioni che è possibile effettuare sulle variabili condition, in basso è rappresentato il codice di inizializzazione. La *entry queue* corrisponde alla lista dei processi che vogliono interagire con le variabili del monitor.



L'invocazione di una `signal` sveglia esattamente un processo, se più processi erano in attesa, lo scheduler decide quale processo può entrare. Nel caso in cui non ci fosse alcun processo in attesa, non succede niente. Successivamente ad una sua invocazione abbiamo a disposizione diverse scelte:

- il processo chiamante si blocca e l'esecuzione passa all'eventuale processo sbloccato
- il processo chiamante esce dal monitor, per ottenere ciò `signal` deve essere l'ultima istruzione di una procedura

Vediamo alcuni esempi nell'utilizzo dei monitor:

Monitor - Esempio 1

```
monitor BinSem
{
    boolean busy;      /* inizializzato a false */
    condition idle;

    entry void P ()
    {
        if (busy) idle.wait();
        busy = true;
    }

    entry void V ()
    {
        busy = false;
        idle.signal();
    }

    busy = false;      /* inizializzazione */
}
```

Monitor - Esempio 2 - Buffer Producer/Consumer

```
Producer ()
{
    while (true) {
        make_item();      // crea un nuovo item
        ProducerConsumer.enter();    // chiamata alla funzione enter
    }
}

Consumer ()
{
    while (true) {
        ProducerConsumer.remove();    // chiamata alla funzione remove
        consume_item();
    }
}

monitor ProducerConsumer
{
    condition full, empty;
```

```

int count;

entry enter ()
{
    if (count == N)
        full.wait();      // se il buffer è pieno, blocca

    put_item();          // mette item nel buffer
    count = count + 1; // incrementa count

    if (count == 1)
        empty.signal(); // se il buffer era vuoto, il
consumatore essere in attesa
}

empty remove ()
{
    if (count == 0)
        empty.wait();

    remove_item();
    count = count - 1;

    if (count == N - 1)
        full.signal(); // se il buffer era pieno, il
produttore viene svegliato
}

count = 0;      // inizializzazione
empty = true;
full = false;
}

```

Monitor - Problemi

I monitor, come si può notare introducono meno errori rispetto alla programmazione con semafori, il problema è che pochi linguaggi forniscono tali monitor, e questa implementazione necessita l'utilizzo della memoria condivisa.

Sincronizzazione in Java

In java è possibile gestire la sezione critica di un programma tramite la parola chiave **synchronized**. I metodi synchronized possono essere eseguiti da una sola thread alla volta e sono realizzati mantenendo un singolo lock (detto monitor) per ogni oggetto.

I metodi **synchronized static** prevedono la presenza di un lock per classe. È possibile avere dei **blocchi synchronized**, in pratica si può impostare un lock su un qualsiasi oggetto per definire una sezione critica. Altre sincronizzazioni sono possibili tramite i metodi `wait()`, `notify()`, `notifyAll()`

Sincronizzazione in Ambiente non Globale

Gli schemi visti in precedenza prevedono che a supporto delle tecniche utilizzate ci sia il meccanismo della memoria condivisa che mette a disposizione di più processi la visibilità sulle variabili. Esistono però dei casi in cui questo non è possibile, si adotta quindi uno schema basato sulla **comunicazione** tra processi tramite uno **scambio di messaggi**. Le funzioni di base saranno:

- **send**
- **receive**

Per affrontare questo tema è necessario affrontare il concetto di **canale di comunicazione**, in particolare come questo venga stabilito, se sia uni oppure bidirezionale, la sua capacità e le dimensioni dei messaggi che vengono trasmessi.

Un'altra tematica è il come ci si possa **riferire ad un processo**. Ci sono due possibilità, una comunicazione **diretta** o **indiretta**.

Comunicazione Diretta

In questo scenario è necessario che i processi si nominino in modo esplicito. La comunicazione diretta può avvenire in modo **simmetrico**:

- `send(P1, message)` : invia il messaggio a P₁
- `receive(P2, message)` : riceve all'interno del campo message un messaggio da P₂

L'alternativa a ciò è la comunicazione **asimmetrica**:

- `send(P1, message)` : invia il messaggio a P₁
- `receive(id, message)` : riceve tutti i messaggi e in id si troverà il nome del processo che ha eseguito la send

Uno svantaggio di questo tipo di comunicazione è che se un processo cambia nome, tutti gli altri processi andranno ricodificati. ## Comunicazione Indiretta Nella comunicazione **indiretta** si ricorre al concetto di **mailbox** (o port): due processi comunicano solo se hanno mailbox in comune:

- `send(A, message)` : invia il messaggio al mailbox A
- `receive(A, message)` : riceve messaggio dal mailbox A

È importante considerare che tra due processi esiste un canale se hanno un mailbox in comune. Un canale può essere associato a più di due processi. Tra due processi possono esistere più canali associati a mailbox differenti. Un canale può sempre essere bidirezionale o unidirezionale.

Un problema legato a questa strategia è il seguente: se più processi eseguono una `receive` su una mailbox, chi riceve il messaggio? In genere la risposta è uno solo di questi processi, e la decisione spetta al sistema operativo.

Buffering

È chiaro che la capacità di un canale vada a limitare la quantità di messaggi scambiabili, ad esempio se la capacità è zero, il sender deve attendere la ricezione di un messaggio prima di inviarne un'altro. Le questa capacità invece diventa un numero finito, il mittente si metterà in attesa in caso di buffer pieno. Se la capacità di questo canale è infinita non ci sarà chiaramente mai attesa.

Ci sarà possibile conoscere quando un messaggio è arrivato a destinazione soltanto nel primo caso.

Capitolo 7 - Deadlock

Federico Segala

Introduzione

Un insieme di processi si dice in stato di **deadlock** nel momento in cui ogni processo è in attesa di un evento che può essere causato da un processo dello stesso insieme.

Questa situazione si verifica dato che nel momento in cui un processo fa richiesta di una risorsa che non può essere immediatamente soddisfatta, il processo si mette in attesa fino a quando tale richiesta non è soddisfacibile, una volta terminato l'utilizzo della risorsa richiesta, questa viene rilasciata.

Un semplice esempio di deadlock si ottiene in questo caso:

Non è sempre detto che si verifichi deadlock, questo dipende infatti dalla **dinamica dell'esecuzione**. Un deadlock si verifica quando si verificano **contemporaneamente** le seguenti 4 condizioni:

- **mutua esclusione**: almeno una risorsa deve essere non condivisibile
- **hold and wait**: deve esistere un processo che detiene una risorsa e che attende di acquisirne un'altra, detenuta da un altro processo
- **no preemption**: le risorse non possono essere rilasciate se non volontariamente dal processo che le sta usando
- **attesa circolare**: deve esistere un insieme di processi che attendono ciclicamente il liberarsi di una risorsa

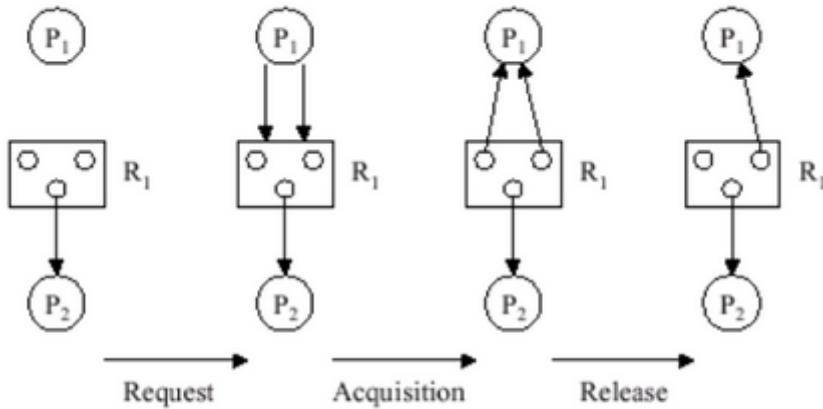
Se anche una sola delle seguenti condizioni non si verifica, non si ha deadlock.

Resource Allocation Graph

Definiamo ora un **modello astratto** per la rappresentazione dello stato delle allocazioni e delle richieste di risorse all'interno del nostro sistema. Si tratta di un **grafo** $G(V, E)$:

- V = insieme dei nodi, che possono essere:
 - cerchi: processi (CPU, I/O, memoria)
 - rettangoli: rappresentano le risorse, in ogni rettangoli sono presenti dei pallini, ognuno dei quali rappresenta un'istanza della risorsa in questione
- E = insieme degli archi

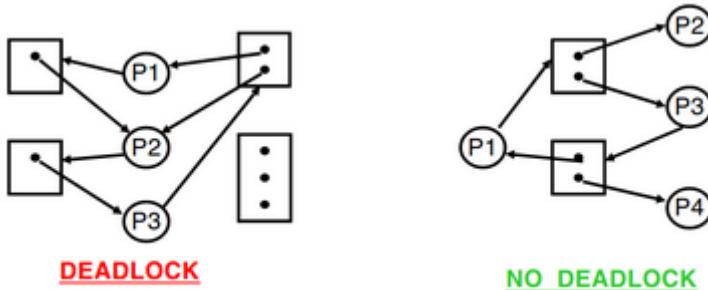
- processi → risorse: processo **richiede** risorsa
- processi ← risorse: processo **dichiara** di detenere risorsa



- $V = \{\{P1, P2\}, \{R1\}\}$
- $E_{iniziale} = \{(R1, P2)\}$ $E_{finale} = \{(R1, P1), (R1, P2)\}$

Nel momento in cui il RAG **non contiene cicli**, siamo sicuri dell'**assenza di deadlock**, nel caso in cui ci siano cicli:

- se sia ha una sola istanza per risorsa, si ha **deadlock**
- se ci sono più istanze, dipende dallo **schema di allocazione** di queste



Deadlock vs. No Deadlock

Gestione dei Deadlock

Per gestire questo fenomeno abbiamo a disposizione diverse alternative:

- prevenzione **statica**: evita che una delle 4 condizioni di sopra si verifichi
- prevenzione **dinamica** basata su allocazione delle risorse: mai utilizzata perché richiede conoscenza troppo approfondita delle richieste di risorse
- rilevazione e ripristino**: permette che si verifichino i deadlock, prevede l'utilizzo di alcuni metodi per riportare il sistema al normale funzionamento

- **algoritmo dello struzzo:** non fa nulla, i deadlock sono rari e gestirli costa troppo

Prevenzione Statica

Iniziamo a vedere quali delle quattro condizioni per il verificarsi di un deadlock sono sacrificabili all'interno del nostro sistema:

la **mutua esclusione** è di fatto irrinunciabile per certi tipi di risorsa

per andare a togliere **hold and wait** abbiamo a disposizione diverse soluzioni:

- un processo alloca all'inizio tutte le risorse che deve utilizzare
- un processo può ottenere una risorsa solo se non ne ha altre, di conseguenza per richiedere nuove risorse, il processo deve prima rilasciare tutte quelle di cui è in possesso

queste tecniche portano con sé alcuni problemi, tra cui un basso utilizzo delle risorse, possibilità di starvation per risorse molto popolari, e la possibilità del numero di risorse richieste da un processo

per andare a evitare la **no preemption** abbiamo a disposizione alcune soluzioni:

- un processo che richiede una risorsa non disponibile deve cedere tutte le altre risorse che detiene
- in alternativa, può cedere risorse che detiene su richiesta di un altro processo

questa tecnica è applicabile soltanto per risorse il cui stato può facilmente essere ristabilito

per andare ad evitare l'**attesa circolare** la soluzione è la seguente:

- assegnamo una priorità ad ogni risorsa, definendo una funzione F che va dalle risorse ai naturali: $F : R \rightarrow N$ tale per cui abbiamo che $F(R_0) < F(R_1) < \dots < F(R_n)$
- un processo può richiedere risorse solo in ordine crescente di priorità
- l'attesa circolare diventa quindi impossibile poiché se $P_0 \rightarrow R_0 \rightarrow P_1 \rightarrow \dots \rightarrow R_{n-1} \rightarrow P_n \rightarrow R_n \rightarrow P_0$ allora avremmo $F(R_0) < F(R_1) < \dots < F(R_n) < F(R_0)$ il che risulta *impossibile*

Prevenzione Dinamica

Le tecniche di prevenzione statica possono portare a un basso utilizzo delle risorse perché mettono vincoli sul modo in cui i processi possono accedere alle risorse.

L'**obiettivo** della prevenzione dinamica è proprio quello di andare a fare prevenzione in

base alle richieste, facendo un'*analisi dinamica del grafo delle risorse* per evitare situazioni cicliche.

Il requisito alla base di questo tipo di prevenzione è la **conoscenza del caso peggiore**, ovvero, serve conoscere il numero massimo di istanze di una risorsa che ogni processo può arrivare a richiedere.

Stato Safe

Lo stato di una risorsa viene calcolato in termine di **numero di istanze allocate** e di **numero di istanze disponibili**. Il sistema si trova in uno stato **safe** nel momento in cui esiste una *sequenza safe*, ovvero, se usando le risorse disponibili, il sistema può allocare risorse ad ogni processo, in qualche ordine, in modo che ciascuno di essi possa terminare la propria esecuzione.

Una **sequenza** di processi (P_1, \dots, P_n) è detta **safe** se $\forall P_i$ le risorse che tale P_i può richiedere possono essere esauridite utilizzando

- le risorse disponibili
- quelle detenute da $P_j, j < i$ attenendendo che P_j termini

Se tale sequenza non esiste, siamo in stato **unsafe**. È importante però ricordare che non tutti gli stati *unsafe* portano ad uno stato di deadlock, ma da uno stato unsafe è possibile arrivare a deadlock.

L'idea della prevenzione dinamica è quella di utilizzare algoritmi che lasciano sempre il sistema in uno stato safe, all'inizio il sistema si trova in tale stato, ogni volta che un processo P richiede una risorsa R , questa viene assegnata al processo soltanto se si rimane in uno stato safe.

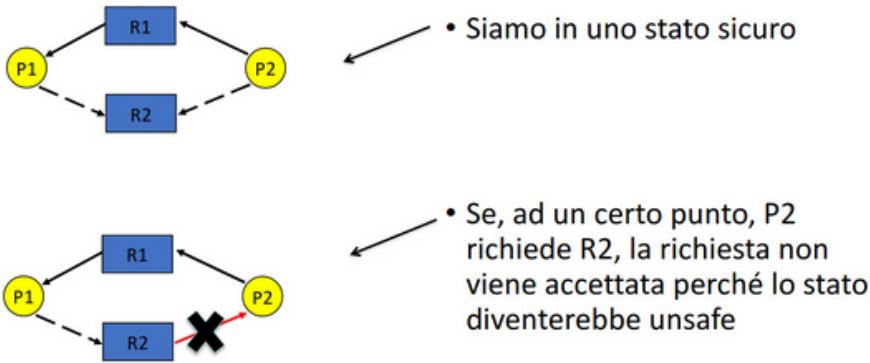
È naturale capire come l'utilizzo delle risorse risulti in questo caso sempre minore rispetto al caso in cui non vengono utilizzate tecniche di prevenzione dinamica. Abbiamo due alternative:

- **Algoritmo con RAG**: funziona soltanto se c'è una sola istanza per risorsa
- **Algoritmo del Banchiere**: funziona qualunque sia il numero di istanze

Algoritmo con RAG

In questo algoritmo andiamo ad **aumentare** il RAG con **archi di reclamo**: $P_i \rightarrow R_j$ nel momento in cui il processo i possa richiedere la risorsa j nel futuro. Sono indicati con una *freccia tratteggiata*.

All'inizio ogni processo deve dire quali risorse intende utilizzare durante la propria esecuzione. Una richiesta viene soddisfatta se e solo se l'allocazione della risorsa non va a creare un ciclo nel RAG. Serve quindi un **algoritmo** per la **rilevazione dei cicli**: $O(n^2)$



Algoritmo con RAG - Esempio

Algoritmo del Banchiere

È un algoritmo meno efficiente dell'algoritmo con RAG ma funziona con più istanze delle risorse, l'idea è quella che un banchiere non debba mai distribuire tutto il denaro che ha in cassa perché altrimenti non potrebbe più soddisfare successivi clienti.

Tale algoritmo si compone di due parti:

- algoritmo di **allocazione**
- algoritmo di **verifica dello stato**

Ogni processo dichiara la sua massima richiesta, ogni volta che un processo richiede una risorsa, si determina se soddisfarla ci lascia in uno stato safe.

```
int available[m]; /* # istanze di R_i disponibili */
int max [n][m]; /* matrice delle richieste di risorse */
int alloc [n][m]; /* matrice di allocazione corrente */
int need [n][m]; /* matrice bisogno rimanente: need[i][j] = max[i]
[j] - alloc[i][j]
```

Vediamo ora l'algoritmo di **allocazione**:

```
void request (int req_vec[]) // richieste del process P_i
{
    if (req_vec[] > need[i])
        error(); // superato il massimo preventivo
    if (req_vec[] > available[])
        wait(); // attendo che si liberino risorse

    /* simulo allocazione delle risorse */
    available[] = available[] - req_vec[];
    alloc[i][] = alloc[i][] + req_vec[];
    need[i][] = need[i][] - req_vec[];
```

```

if (!state_safe()) {      // se lo stato dopo la simulazione non è
    safe, ripristino
    available[] = available[] + req_vec[];
    alloc[i][] = alloc[i][] - req_vec[];
    need[i][] = need[i][] + req_vec[];
}
}

```

L'algoritmo di **verifica dello stato** è invece il seguente:

```

boolean state_safe()
{
    int work[m] = available[];
    boolean finish[n] = (false,...,false);
    int i;

    while (finish != (true,...,true)) {      // finché non trovo
        sequenza safe
        /* cerca P_i che NON abbia terminato e che possa completare
        con le risorse disponibili nel vettore work */
        for (i = 0; i < n && (finish[i] || need[i][] > work[])); i++);
        if (i == n) // nessuno degli elementi riesce a terminare
            return false; // non abbiamo stato safe: UNSAFE
        else { // trovo un P_i che non ha finito e la cui necessità
            sta nei limiti di work
            work = work[] + alloc[i][]; // proseguo la simulazione
            finish[i] = true;
        }
    }
}

```

Rilevazione Deadlock & Ripristino

Prevenzione statica e dinamica sono due algoritmi troppo conservativi e tendono a ridurre in modo eccessivo l'utilizzo delle risorse. Abbiamo a disposizione due approcci alternativi:

- rilevazione e ripristino tramite **RAG**
- algoritmo di **rilevazione**

Rilevazione e Ripristino con RAG

Tale algoritmo, come nel caso della prevenzione dinamica, funziona con una sola risorsa per tipo. La tecnica consiste nell'analizzare periodicamente il RAG e verificare se esistono deadlock (*detection*) e, in caso affermativo, iniziare il ripristino (*recovery*). Trai vantaggi che sono introdotti di questa tecnica troviamo:

- **conoscenza anticipata** delle risorse richieste non necessaria
- utilizzo migliore

Lo svantaggio principale è sicuramente dato dal costo di effettuare una **recovery**

Algoritmo di Rilevazione

L'algoritmo di **rilevazione** è basato sull'esplorazione di ogni possibile sequenza di allocazione per i processi che non hanno ancora terminato. Se la sequenza va a buon fine (ho almeno una sequenza *safe*), non c'è deadlock. Le strutture dati utilizzate da questo algoritmo sono molto simili a quelle utilizzate nell'algoritmo del banchiere

```

int work[m] = available[m];
boolean finish[] = (false, ..., false);
boolean found = true;

while (found) {
    int i;
    found = false;
    for (i = 0; i < n && !found; i++) {
        // cerca P_i la cui richiesta sia soddisfacibile
        if (!finish && req_ve[i][] <= work[]) {
            /*
                assume OTTIMISTICAMENTE che P_i esegua fino al termine e
                che quindi restituisca le risorse, se non è così il
                possibile
                deadlock sarà trovato alla prossima esecuzione
            */
            work[] = work[] + alloc[i];
            finish[i] = true;
            found = true;
        }
    }
} /* se finish[i] = false per qualsiasi i, P_i è in deadlock */

```

È facile vedere come l'algoritmo appena presentato sia $O(mn^2)$

Una domanda che è possibile farsi è la seguente: **quanto spesso chiamare l'algoritmo di detection?** Ci sono disponibili diverse strade:

- dopo ogni richiesta
- ogni N secondi
- quando l'utilizzo della CPU scende sotto una **soglia T**

In caso di deadlock abbiamo a disposizione diverse per il **ripristino**:

- uccisione dei processi coinvolti
- prelazione delle risorse dai processi bloccati nel deadlock

Ripristino - Uccisione dei processi coinvolti

La prima alternativa è l'uccisione di **tutti i processi** ma si tratta di una strategia **costosa** in quanto tutti i processi devono ripartire e perdono il lavoro svolto fino a quel momento. Una possibile alternativa è quella dell'**uccisione selettiva** fino alla scomparsa dei deadlock, ma anche questa strada è abbastanza costosa dato che serve invocare la detection dopo ogni uccisione.

Ripristino - Prelazione delle Risorse

Il problema che si pone diventa a questo punto **a chi togliere risorse e in quale ordine**: il problema è che il processo che subisce la prelazione non può continuare la sua normale esecuzione. La soluzione adottata è quello di un **rollback** in uno stato safe. Eventualmente è possibile anche un *total rollback*.

È possibile starvation, dato che è possibile togliere risorse sempre agli stessi processi, la soluzione a ciò è considerare il numero di rollback nei fattori di costo.

Conclusione

Ognuno degli approcci visti ha i suoi vantaggi ed i suoi svantaggi, nessuno dei tre algoritmi è superiore agli altri. Spesso si fa uso di una **soluzione combinata** nella quale le risorse sono partizionate in classi. Viene utilizzata una strategia di ordinamento tra classi di risorse. All'interno di una classe si potrà utilizzare l'algoritmo che si ritiene più appropriato.

Un esempio di suddivisione in classi può essere ad esempio:

- **risorse di sistema** (PCB, I/O, ...)
- **memoria**
- **risorse di processo** (File)
- **spazio di swap** (blocchi su disco)

Alcuni algoritmi specifici possono essere ad esempio:

- prevenzione tramite ordinamento delle risorse

- prevenzione tramite prelazione: quando un job può essere swappato
- prevenzione dinamica: quando la richiesta massima di risorse è tipicamente nota a priori
- prevenzione tramite preallocazione: richiesta massima di memoria tipicamente nota a priori

Capitolo 8 - Memoria Principale

Federico Segala

Introduzione

La condivisione della memoria da parte di più processi è un aspetto fondamentale per l'efficienza del sistema. Le problematiche che si pongono sono le seguenti:

- allocazione della memoria ai singoli job
- protezione dello spazio di indirizzamento
- condivisione dello spazio di indirizzamento
- gestione dello swap

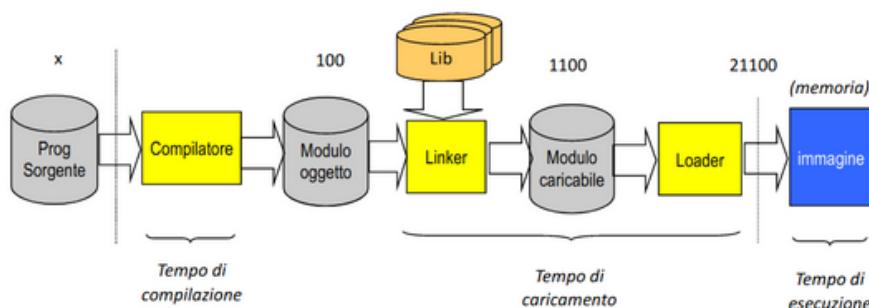
Nei sistemi moderni la gestione della memoria è inseparabile dal concetto di memoria virtuale. Abbiamo da considerare un importante **vincolo**: ogni programma deve essere portato in memoria e trasformato in processo per essere eseguito:

- CPU **preleva istruzioni** dalla memoria in base al program counter
- L'istruzione prelevata può prevedere il **prelievo di operandi** dalla memoria
- Al termine dell'istruzione, il risultato può essere **scritto in memoria**
- Quando il processo termina, la **memoria** viene **rilasciata**

Per passare da programma a processo ci sono diverse fasi per cui passare, in ognuna di queste fasi la **semantica** degli **indirizzi** è differente. È necessario distinguere il concetto di **spazio logico** e di **spazio fisico**: se il programma sorgente è composto di *indirizzi simbolici*, la memoria si compone di *indirizzi fisici*.

Questa trasformazione avviene per mezzo di due fasi:

- **compilatore**: associa gli indirizzi simbolici a **indirizzi rilocabili**
- **linker e/o loader**: associano indirizzi rilocabili ad **indirizzi assoluti**



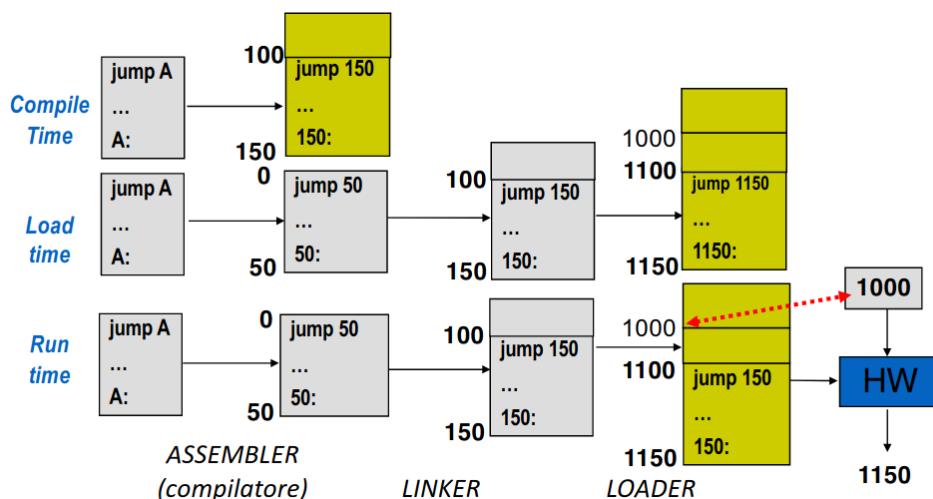
Fasi della Trasformazione

Gli indirizzi hanno diverse rappresentazioni nelle varie fasi di costruzione di un programma. Il collegamento tra indirizzi simbolici e indirizzi fisici è detto **binding**.

Binding dati/istruzioni - indirizzi di memoria

Questa pratica può avvenire in tre momenti distinti:

- **tempo di compilazione**: si utilizza per generare *codice assoluto* se noto a priori in quale parte della memoria risiederà il processo. Se la locazione di partenza cambia, si rende necessaria una ricompilazione
- **tempo di caricamento**: si utilizza nel caso sia necessario generare del *codice rilocabile*, vengono utilizzati *indirizzi relativi*. Nel caso in cui cambi l'indirizzo di riferimento di rende necessario un ricaricamento
- **tempo di esecuzione**: il binding viene posticipato nel caso in cui il processo possa essere spostato durante l'esecuzione in posizioni diverse della memoria. Per fare ciò è necessario del *supporto hardware*.



Binding - Esempio

Diciamo che il binding effettuato a tempo di compilazione e caricamento si dice **statico**, quello effettuato a tempo di esecuzione, è invece detto **dinamico**

Collegamento - Linking

Il linking, come il binding può essere sia statico che dinamico:

- **statico**: si tratta della tecnica tradizionale in cui tutti i riferimenti sono definiti prima dell'esecuzione. L'immagine del processo contiene una copia delle librerie utilizzate
- **dinamico**: il link viene posticipato a tempo di esecuzione, dunque il codice del programma non contiene il codice delle librerie ma soltanto un *riferimento* per poterle recuperare. Un esempio di questo approccio è Windows DLL

Collegamento - Loading

Anche in questo caso abbiamo due opzioni:

- **statico**: si tratta dell'approccio tradizionale in cui tutto il codice è caricato in memoria al tempo dell'esecuzione
- **dinamico**: il caricamento dei moduli viene posticipato in corrispondenza del primo utilizzo di questi. In questo modo, il codice non utilizzato non viene caricato. È un approccio molto efficace nel caso in cui il codice contenga molti casi speciali (moduli)

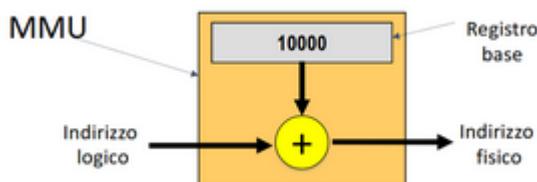
Spazi di Indirizzamento

Ovviamente lo spazio di indirizzamento logico è strettamente collegato allo spazio di indirizzamento fisico. Un indirizzo logico viene generato dalla CPU ed è detto anche **indirizzo virtuale**, gli indirizzi fisici sono invece visti unicamente dalla memoria.

Nel caso di binding statico, questi due indirizzi coincidono, nel caso di binding dinamico, tali indirizzi sono generalmente diversi.

Memory Management Unit

Si tratta di un dispositivo hardware che **mappa** indirizzi virtuali in indirizzi fisici. Lo schema di base è il seguente: abbiamo un **registro di rilocazione** il cui valore è aggiunto ad ogni indirizzo generato da un processo e viene inviato alla memoria.



MMU - Schema di Base

In un sistema multiprogrammato in cui non è noto in anticipo dove un processo si troverà in memoria **non è possibile binding a tempo di compilazione**. Inoltre l'esigenza di avere area di swap impedisce di utilizzare indirizzi rilocati in modo statico: **il binding a tempo di caricamento non è possibile**.

Possiamo quindi trarre le seguenti conclusioni sulle varie tecniche di rilocazione:

- *dinamica*: utilizzata nei sistemi complessi, la gestione della memoria è affidata al sistema operativo
- *statica*: possibile in sistemi application-specific, la componente di gestione della memoria affidata al sistema operativo è abbastanza limitata

Allocazione Contigua

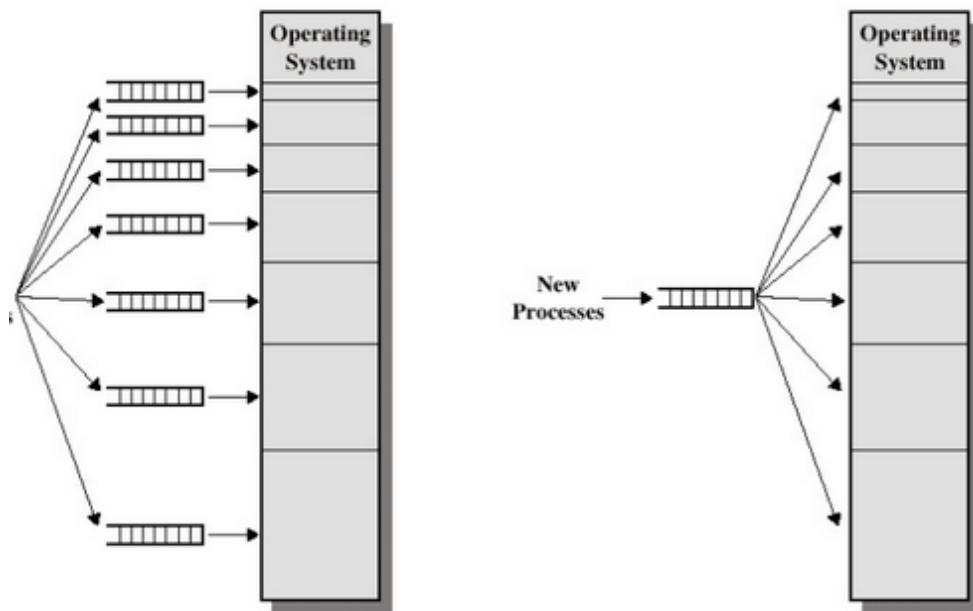
In questo schema di allocazione i processi vengono allocati in memoria in posizioni contigue all'interno di una partizione. La memoria è divisa in **partizioni** che possono essere:

- **fisse**
- **variabili**

Tecnica delle Partizioni Fisse

In questo approccio la memoria è vista come un **insieme di partizioni di dimensioni predefinite** che tipicamente sono diverse. Dobbiamo affrontare due questioni. La prima è quella dell'**assegnazione della memoria ai job**.

Questa viene effettuata dallo sceduler a lungo termine. Abbiamo due opzioni anche qui, una coda per ogni partizione, oppure una singola coda



Coda singola vs. Coda multipla

Nel caso di **una coda per partizione**, il processo viene assegnato alla partizione più piccola in grado di contenerlo, questo approccio però è molto poco flessibile, in quanto possono esserci partizioni vuote e job nelle altre code.

Una **singola coda** si può invece gestire con diverse politiche:

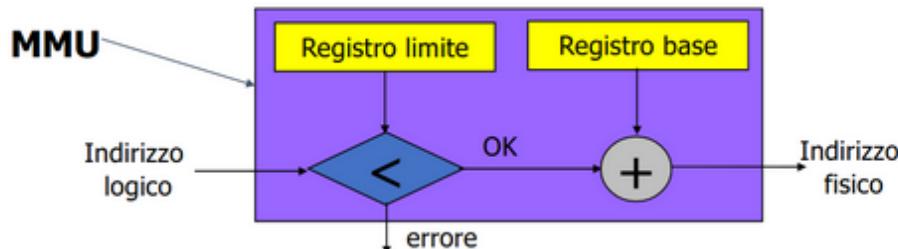
- FCFS: facile ma vi è un basso utilizzo della memoria
- Scansione della coda: si può implementare con best-fit-only, best-available-fit

La seconda questione da affrontare è quella del **supporto per la rilocazione dinamica**.

La MMU consiste in un insieme di **registri di rilocazione** utilizzati per proteggere lo spazio dei vari processi (in modo attivo e passivo). Questi registri contengono:

- *valore dell'indirizzo più basso*, anche detto registro base o di rilocazione
- *limite superiore dello spazio logico*

Ogni indirizzo logico deve risalire < del limite stabilito dal registro:



Partizioni Fisse - MMU

Il vantaggio di questa tecnica è dato sicuramente dalla relativa **semplicità di implementazione**. Ci sono però diversi svantaggi, in primis il grado di multiprogrammazione viene limitato dal numero di partizioni. In secondo luogo si dà vita a **frammentazione**:

- *interna*: avviene se la dimensione della partizione è più grande di quella del job
- *esterna*: avviene se vi sono partizioni non utilizzate che non soddisfano le esigenze dei processi in attesa

Tecnica delle Partizioni Variabili

Lo spazio utente in questo caso viene diviso in **partizioni a dimensione variabile** che sono di dimensioni identiche alla dimensione del processo. Il motivo dell'invenzione di questa tecnica è quello dell'*eliminazione della frammentazione interna*.

Andiamo a vedere come si risolve in questo caso la questione dell'**assegnazione di memoria** ai job:

La memoria è vista come un **insieme di buche** dove una buca è un blocco di memoria disponibile. Il sistema operativo mantiene informazioni sulle partizioni allocate e sulle buche. Quando arriva un processo gli viene allocata memoria utilizzando una buca che sia in grado di contenerlo.

Per soddisfare la richiesta di n celle di memoria data una lista di buche libere abbiamo diverse strategie:

- *first-fit*: alloca la prima buca grande a sufficienza

- best-fit: alloca la più piccola buca grande a sufficienza, è richiesta la scansione di tutta la lista, ma comporta il minimo spreco.
- worst-fit: alloca la più grande buca, richiede anche in questo caso la scansione della lista, lascia la buca di dimensioni più grandi.

Tipicamente la *migliore strategia* adottabile è sempre **first-fit**

Il **supporto per la rilocazione** è gestito allo stesso modo che nella tecnica delle partizioni fisse.

Il vantaggio comportato da questa tecnica è l'eliminazione di frammentazione interna. Lo svantaggio principale è la **frammentazione esterna**, dato che esiste spazio disponibile in memoria ma questo non è contiguo. Con first-fit, dati N blocchi allocati, $0.5N$ blocchi vanno persi.

Per limitare il problema, potremmo proporre intuitivamente di **compattare** il contenuto della memoria in modo da rendere contigui i blocchi di un processo. Ciò è possibile soltanto nel caso in cui la rilocazione è dinamica, e ciò richiederebbe la modifica del registro base. Si tratta inoltre di una tecnica costosa, dato che viene spostato un grande quantitativo di memoria.

Buddy System

Una soluzione meno intuitiva al problema della frammentazione è la **tecnica del buddy system** che corrisponde ad un compromesso tra partizioni fisse e variabili.

La memoria è vista come liste di blocchi di dimensione 2^k con $L < k < U$ dove 2^L è il più piccolo blocco allocato, mentre 2^k corrisponde al più grande blocco allocato (tutta la memoria). La memoria si rende disponibile sotto forma di questi blocchi.

Inizialmente tutta la memoria è disponibile e la lista di blocchi di dimensione 2^U contiene un solo blocco che rappresenta tutta la memoria, mentre tutte le altre liste sono vuote.

Allocazione

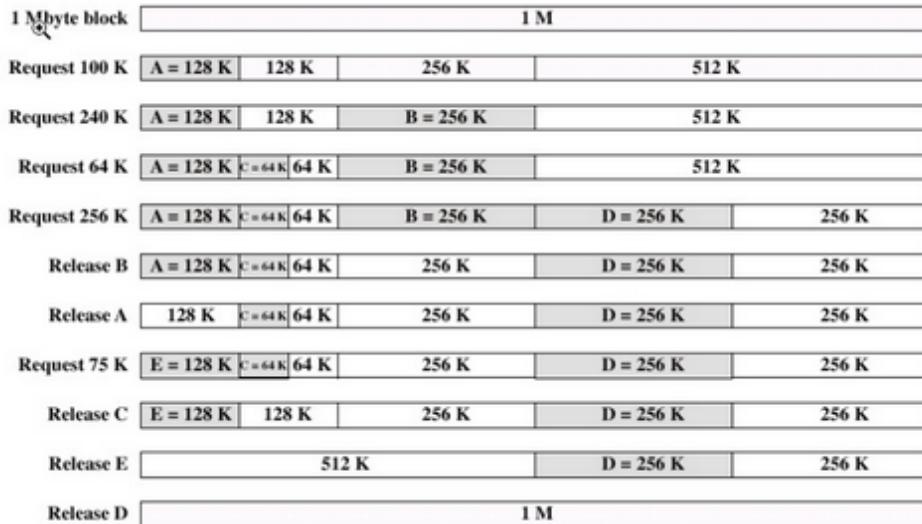
Quando arriva una richiesta di una data dimensione s si cerca un blocco libero con dimensione adatta purché sia pari ad una potenza di 2:

- se $2^{U-1} < s \leq 2^U$ l'intero blocco di dimensione U viene allocato
- altrimenti il blocco 2^U viene diviso in due blocchi di dimensione 2^{U-1}
- se $2^{U-2} < s \leq 2^{U-1}$ l'intero blocco di dimensione 2^{U-1} viene allocato
- altrimenti il blocco viene diviso in due blocchi grandi la metà e così via fino ad arrivare al limite con due blocchi di dimensione 2^L

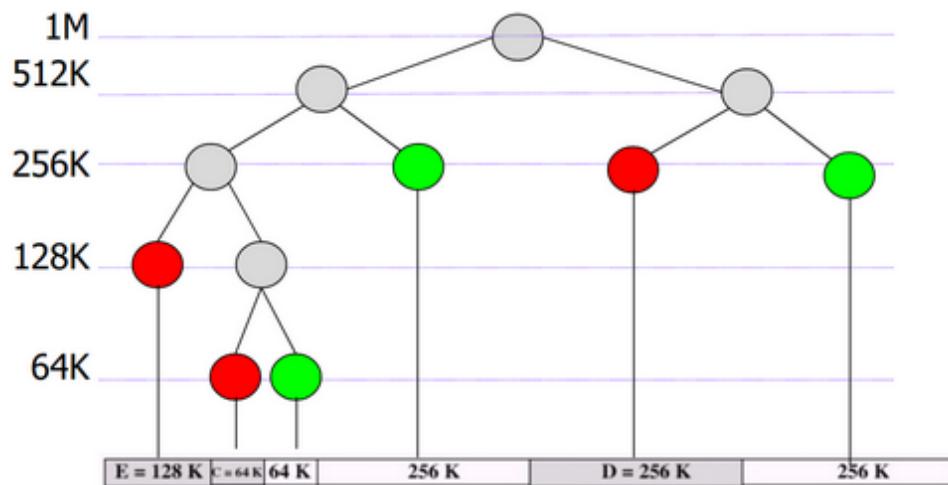
Rilascio

Quando un processo rilascia la memoria, il suo blocco torna a far parte della lista dei blocchi di dimensione corrispondente. Se nel fare ciò si formano due blocchi adiacenti di dimensione 2^k è possibile compattarli ottenendo un unico blocco di dimensione 2^{k+1}

Il vantaggio introdotto da quest'tecnicha consta nel fatto che la compattazione richiede solo di scorrere la lista dei blocchi di dimensione 2^k quindi è una procedura veloce. Lo svantaggio è che si presenta frammentazione interna anche se dovuta soltanto ai blocchi di dimensione 2^L



Buddy System - Esempio



Buddy System - Esempio

Si può notare come i blocchi si possano dividere in un albero binario:

Paginazione

Si tratta di una tecnica per eliminare la frammentazione esterna. L'idea è quella di permettere che lo spazio di indirizzamento fisico di un processo sia **non-contiguo**. La memoria fisica viene allocata dove disponibile.

La memoria **fisica** viene divisa in blocchi detti **frame**, le cui dimensioni tipiche sono 512 byte ... 8 kbyte. La memoria **logica** viene divisa in blocchi della stessa dimensione detti **pagine**.

Per eseguire un programma avente dimensione n pagine serve trovare n frame liberi. Si utilizza una **tavella delle pagine** per mantenere traccia di quale frame corrisponde a quale pagina. Utilizzeremo una tabella delle pagine per ogni processo, essa viene utilizzata per tradurre un indirizzo logico in un indirizzo fisico. Ad esempio:

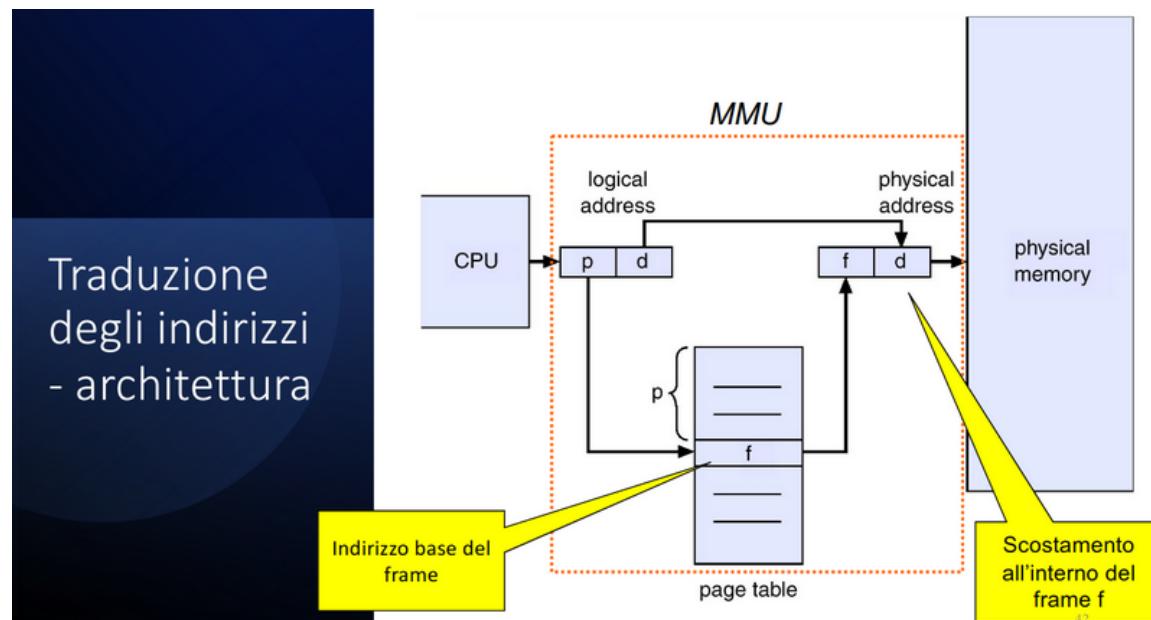
- sia 1KB la dimensione della pagina
- sia 2.3KB la dimensione del programma
- saranno quindi necessarie 3 pagine, dell'ultima verranno utilizzati soltanto 0.3KB
- è ancora possibile avere frammentazione interna, ma soltanto nell'ultima pagina

Traduzione degli Indirizzi

L'indirizzo generato dalla CPU viene diviso in due parti:

- **numero di pagina p**: utilizzato come indice nella tabella delle pagine che contiene l'indirizzo di base di ogni frame
- **offset d**: che viene combinato con l'indirizzo di base per definire l'indirizzo fisica che viene inviato alla memoria

Se la dimensione della memoria è 2^m e quella di una pagina è 2^n , l'indirizzo sarà suddiviso in modo che per il *numero di pagina* saranno utilizzati $m - n$ bit, mentre per l'*offset* saranno utilizzati n bit



Implementazione della Page Table

L'efficienza in questo contesto è fondamentale. Alcune soluzioni sono:

- implementazione tramite registri
- implementazione in memoria
 - tabella delle pagine multilivello
 - tabella delle pagine invertita

Implementazione tramite Registri

Le entry della tabella delle pagine sono mantenute nei registri. Si tratta di una soluzione efficiente che risulta fattibile soltanto se il numero di entry è limitato, questo dato che l'architettura di un calcolatore mette a disposizione un basso numero di registri.

È da considerare con il salvataggio dei registri allunga i tempi di context switch.

Implementazione in Memoria

La tabella delle pagine risiede in memoria, vengono comunque utilizzati due registri:

- **page table base register** che punta alla tabella delle pagine
- **page table length register** che contiene la dimensione della tabella delle pagine

Il context switch in questo caso è più breve in quanto richiede la modifica soltanto di questi due registri. Il problema consiste nel fatto che ogni accesso a dati/istruzioni richiede due accessi in memoria, ovvero l'accesso per la tabella delle pagine e l'accesso per il dato effettivo

Questo problema del doppio accesso si può risolvere tramite una veloce cache detta **transaction lookaside buffer (TLB)** il quale confronta l'elemento fornito con il campo chiave di tutte le entry (in modo contemporaneo) e che associa un numero di pagina ad un numero di frame (come del resto la tabella delle pagine)

Il TLB è un componente molto costoso, per questo motivo vi viene memorizzato solo un piccolo sottoinsieme delle entry della tabella delle pagine. Ad ogni context switch il TLB viene ripulito per evitare il mapping di indirizzi errati.

Durante un accesso in memoria, se la pagina è nel TLB questo restituisce il numero di frame con un singolo accesso, con un tempo richiesto <10% del tempo richiesto in assenza di TLB. Se la pagina non è nel TLB è necessario accedere alla tabella delle pagine in memoria. Definiamo **hit ratio** α la % delle volte in cui una pagina si trova nel TLB.

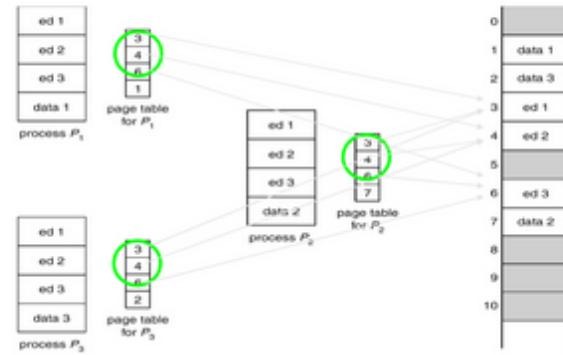
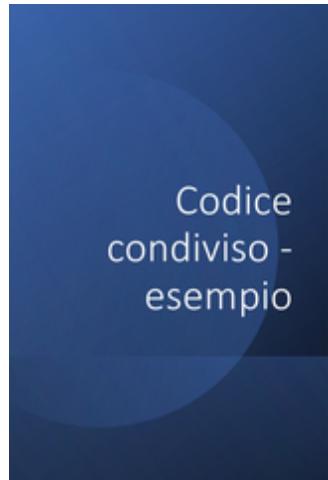
Definiamo **tempo di accesso effettivo** il valore

$$EAT = (T_{mem} + T_{tlb}\alpha + (2T_{mem} + T_{tlb})(1 - \alpha))$$

Protezione

Questa viene realizzata associando *ad ogni coppia frame/pagina* dei **bit di protezione**, ad esempio il bit di validità, il bit di accesso, e simili...

Per quando riguarda il **codice condiviso** abbiamo un'unica copia fisica, ma più copie logiche (una per processo). Il codice read-only può essere condiviso tra i processi. I dati in generale saranno diversi da processo a processo

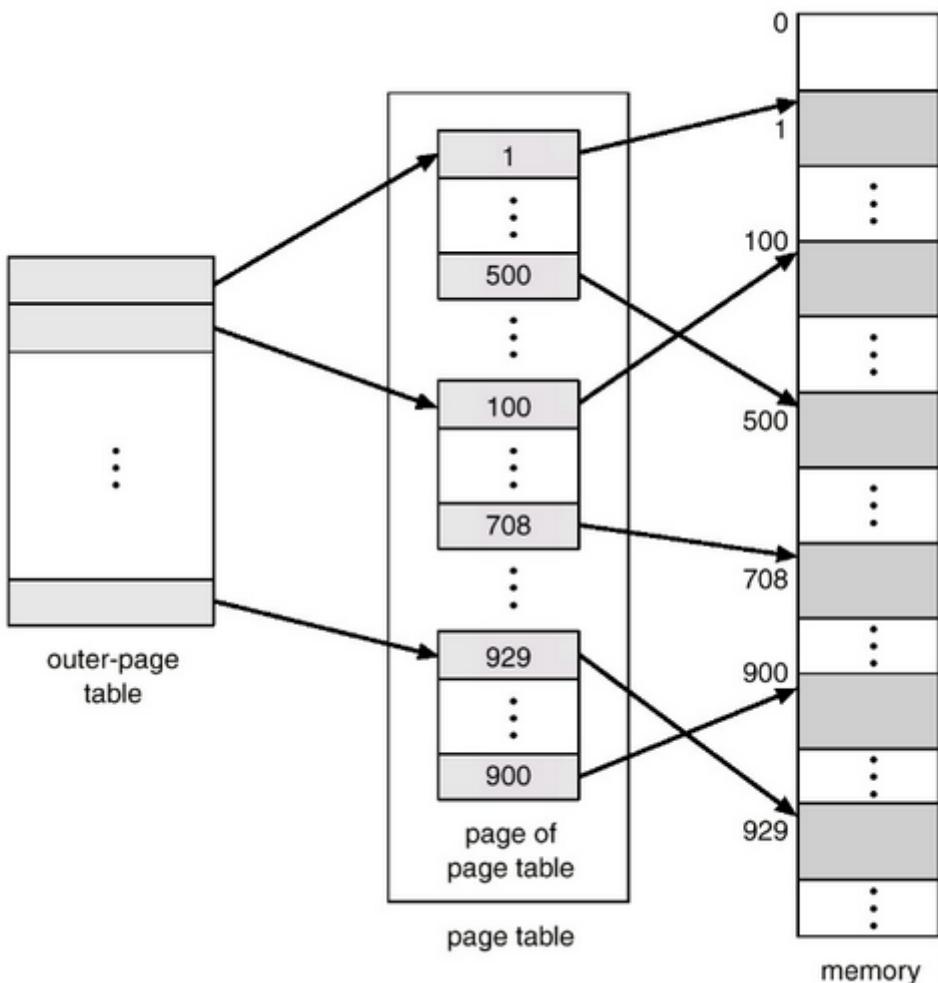


Codice Condiviso - Esempio

Dato uno spazio di indirizzamento virtuale pari a 2^{64} molto maggiore dello spazio fisico e un numero di frame = $4K = 2^{12}$, avremo $2^{64}/2^{12}$ entry nella tabella delle pagine per ogni processo. È chiaro che siano necessari dei meccanismi per gestire la dimensione di questa tabella

Tabella delle Pagine Multilivello

Questa strategia consiste praticamente nel *paginare* la tabella delle pagine. Soltanto alcune parti della tabella delle pagine sono memorizzate esplicitamente in memoria, le altre sono su disco. Abbiamo a disposizione versioni a 2, 3 o 4 livelli.



Ogni livello è memorizzato come una tabella separata in memoria, la conversione dell'indirizzo logico in quello fisico può richiedere, nel caso di paginazione a 3 livelli, può richiedere 4 accessi in memoria. Il TLB mantiene comunque ragionevoli le prestazioni. Ad esempio dato un tempo di accesso alla memoria di 90ns, e un tempo di TLB pari a 10ns, con un $\alpha = 90/100$ otteniamo

$$EAT = (10 + 90)0.9 + (10 + 360)0.1 = 1.4T_{mem}$$

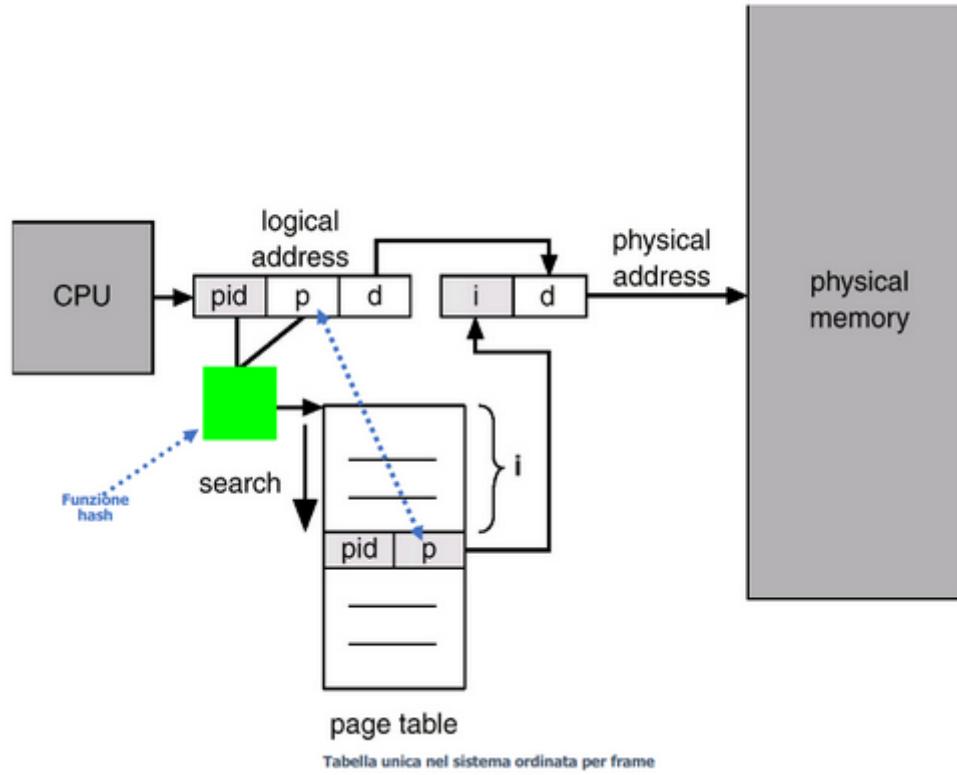
Tabella delle Pagine Invertita

Questa tecnica consiste nel tenere una **unica** tabella avente una entry per ogni frame contenente:

- indirizzo virtuale della pagina che occupa quel frame
- informazioni sul processo che usa quella pagina

Il problema a questo punto consiste nel fatto che per un dato indirizzo fisico possono essere mappati più indirizzi virtuali. Di conseguenza sarà necessario ricercare il valore desiderato, con conseguente aumento del tempo necessario a ricercare un riferimento ad una pagina.

Per questioni di efficienza, la ricerca nella tabella non può essere sequenziale: utilizziamo l'equivalente di una **tabella hash** che ci permette di ridurre il tempo di ricerca a $\theta(1)$. In pratica è necessario un meccanismo per gestire le collisioni quando diversi indirizzi virtuali corrispondono allo stesso frame.



Segmentazione

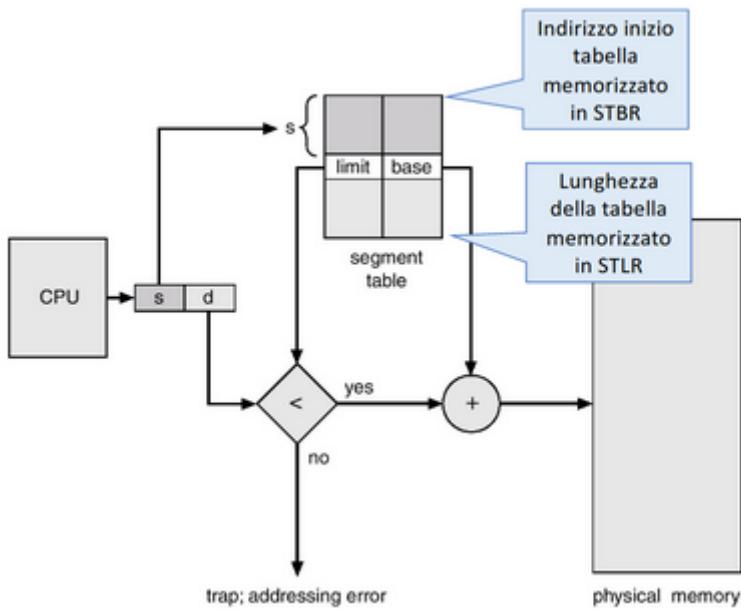
Quello della segmentazione è uno schema nato per gestire la memoria che supporta la vista che l'utente ha della memoria, infatti il programma è una collezione di segmenti, ovvero **unità logiche**.

Questi segmenti non hanno tutti la stessa dimensione, sono indicati da nome e lunghezza.

Un indirizzo **logico** consiste di $<$ numero_segmento, offset $>$. La **tabella dei segmenti** mappa indirizzi logici bidimensionali in indirizzi fisici unidimensionali. Ogni entry contiene una **base** ovvero l'indirizzo fisico di partenza del segmento in memoria, e un **limite**, ovvero la lunghezza del segmento. Questa tabella ha una struttura abbastanza simile a quella della page table: in memoria sono infatti memorizzati

- **segment table base register**
- **page table base register**, il quale indica il numero di segmenti utilizzati da un programma. Un indirizzo logico $< s, d >$ è valido se $s < \text{STLR}$. $\text{STBR} + s =$ indirizzo dell'elemento della tabella dei segmenti da recuperare.

Il TLB, come nella paginazione, viene utilizzato per memorizzare le entry maggiormente utilizzate

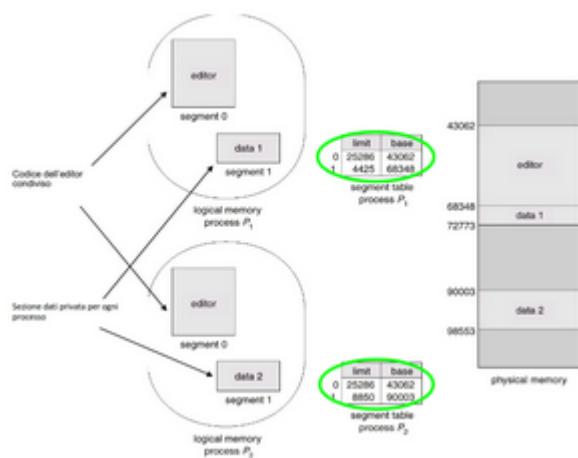


Protezione

La segmentazione supporta naturalmente la protezione e la condivisione di porzioni di codice. Un segmento è infatti un’entità semantica ben definita. Per la protezione associamo ad ogni segmento:

- **bit di modalità**
- **bit di validità**

Condivisione



Vediamo ora di fare un confronto tra segmentazione e frammentazione.

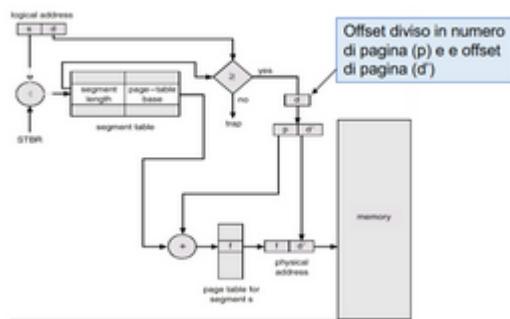
Il sistema operativo deve allocare spazio in memoria per tutti i segmenti di un programma, segmenti che hanno lunghezza variabile. L’allocazione dei segmenti è un problema di allocazione dinamica risolto con first-fit o best-fit. C’è possibilità di frammentazione esterna, specialmente per segmenti di dimensione significativa.

Dunque se la paginazione permette assenza di frammentazione esterna con minima frammentazione interna e l'allocazione dei frame non richiede specifici algoritmi al netto di una separazione tra vista utente e vista fisica della memoria; la segmentazione fa l'esatto contrario, ovvero garantisce consistenza tra vista utente e vista fisica della memoria, associando protezione e condivisione ai segmenti, al netto di allocazione dinamica dei segmenti e di potenziale frammentazione esterna

Segmentazione Paginata

Ci permette di combinare i due schemi appena visti per migliorarli entrambi. La soluzione adottata consiste nel **paginare i segmenti**, ovvero, ogni segmento è suddiviso in pagine e possiede la sua tabella delle pagine. La tabella dei segmenti non contiene l'indirizzo base di ogni segmento, ma l'indirizzo base delle tabelle delle pagine per ogni segmento.

In questo modo eliminiamo il problema dell'allocazione dei segmenti e della frammentazione esterna.



Capitolo 9 - Memoria Virtuale

Federico Segala

Motivazioni

La caratteristica degli schemi precedenti per la gestione della memoria prevede che l'intero programma sia caricato in memoria prima di essere eseguito. In generale questa cosa non è strettamente necessario, infatti spesso basta che soltanto parte del programma sia presente in memoria. Di conseguenza possiamo ottenere uno spazio di indirizzi logici ancora più grande dello spazio degli indirizzi fisici. E quindi possiamo mantenere in memoria ancora più processi.

Il concetto chiave della **memoria virtuale** è la possibilità di **swappare** pagine da e verso la memoria e non l'intero processo. La memoria virtuale permette di separare la memoria logica (utente) dalla memoria fisica.

Può essere applicata in modi diversi:

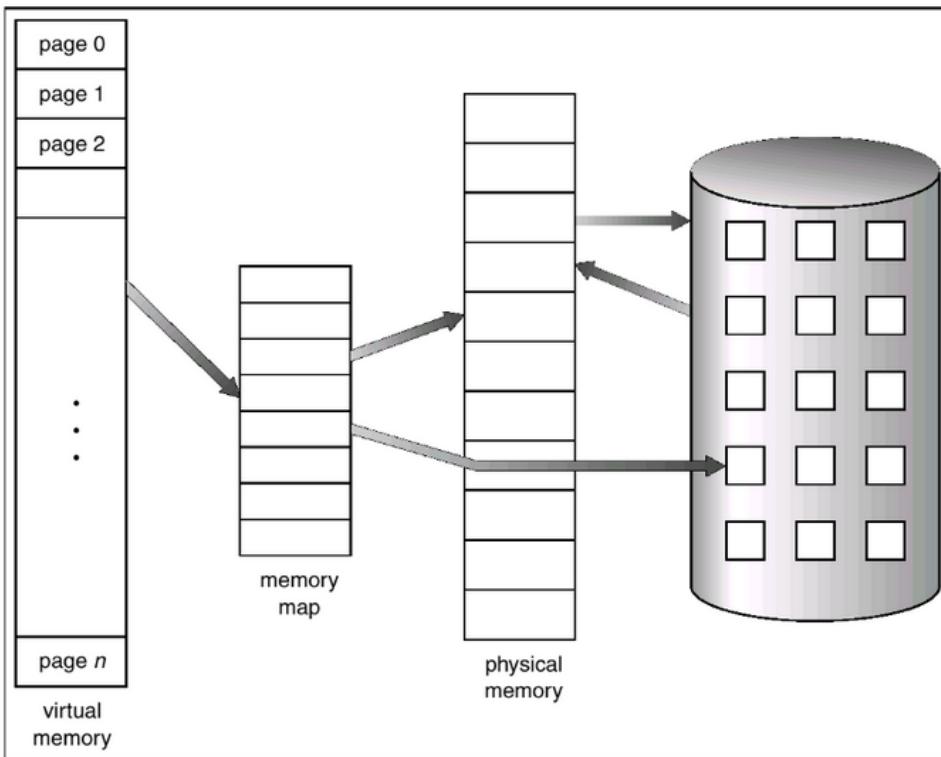
- **paginazione su domanda**
- **segmentazione su domanda**

Paginazione su Domanda

Il concetto è che una pagina viene caricata in memoria soltanto quando è strettamente necessario. I vantaggi di questa tecnica sono i seguenti:

- meno richieste di I/O quando è necessario lo swapping, ciò porta ad ottenere una risposta più rapida
- meno memoria, più processi hanno accesso alla memoria

È fondamentale conoscere lo stato di una pagina, ovvero se si trova in memoria o meno



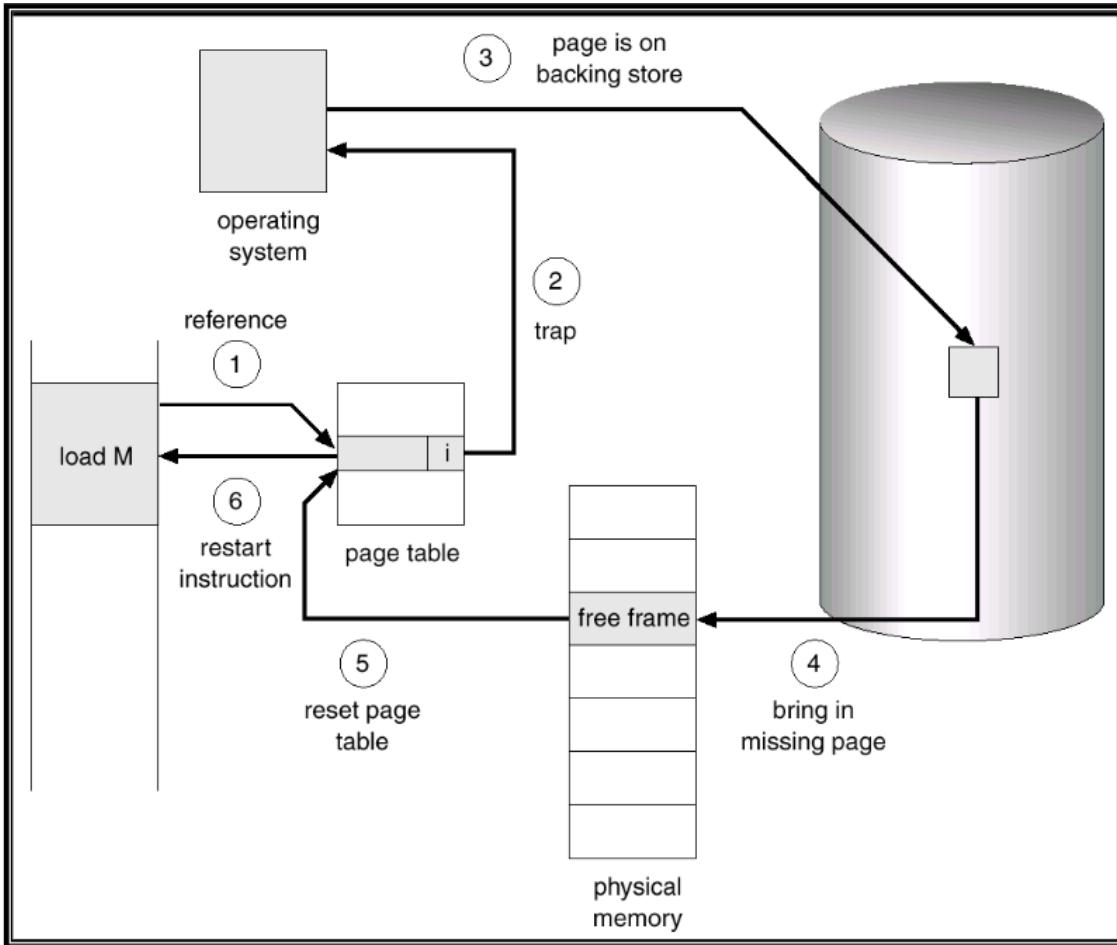
f22ed02b84ebfaca5e9401789b26780d.png

Page Fault

Per conoscere lo stato di una pagina, in ogni entry della page table è presente un bit di validità, se 1 tale pagina non è in memoria, se 0 non è in memoria. Si verifica un **page fault** quando durante la traduzione da indirizzo logico a indirizzo fisico una entry ha bit di validità pari a 0.

Una volta generato, un page fault genera un **interrupt** verso il sistema operativo. A questo punto, il sistema operativo verifica una tabella associata a tale processo che comunica se il riferimento è valido o meno. In caso di validità la pagina richiesta viene caricata.

A questo punto il sistema operativo cercherà un frame vuoto ed effettuerà lo swap della pagina del frame. In seguito modifica le tabelle (bit di validità della page table, tabella interna del processo) e ripristina l'istruzione che ha causato il page fault.



La paginazione su domanda influenza il tempo di accesso effettivo alla memoria: il tasso di page fault è una probabilità, $0 \leq p \leq 1$, più tale probabilità è alta, più alto sarà in media il numero di page fault. E AT = $(1 - p)t_{mem} + p * t_{pagefault}$

Il **costo di un page fault** è dato da tre componenti:

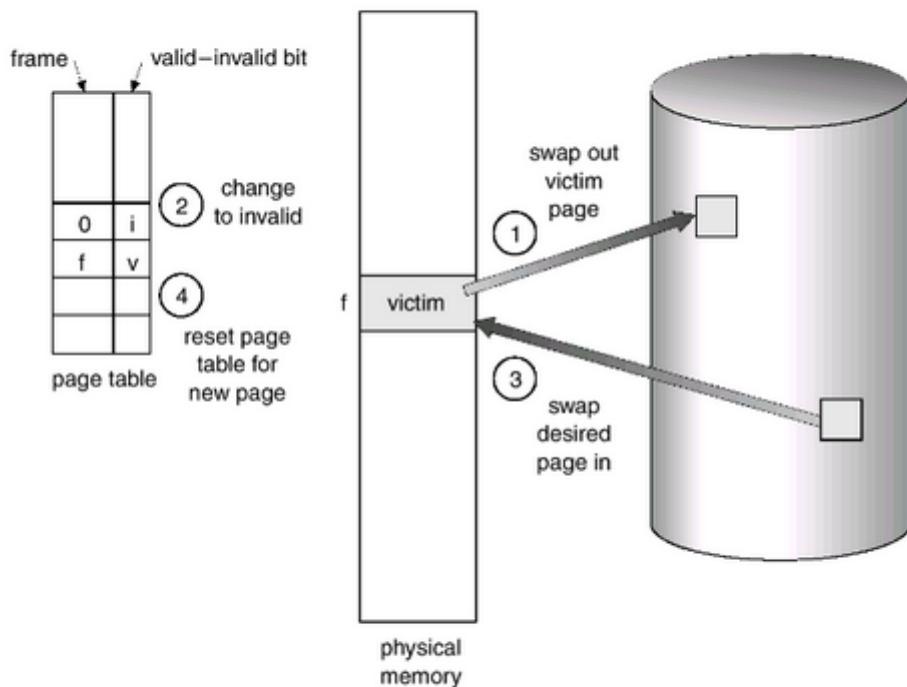
- servizio dell'interrupt
- swap in (lettura della pagina)
- costo del riavvio del processo
- eventuale swap out

Rimpiazzamento delle Pagine

Nel caso in cui non ci siano pagine libere è necessario il **rimpiazzamento delle pagine**: si cercano frame in memoria e si swappano su disco. Per realizzare questa pratica è necessario un opportuno algoritmo. L'obiettivo è ottimizzare le prestazioni, cioè **minimizzare il numero di page fault**

Gestione dei Page Fault

In caso di assenza di frame liberi, il sistema operativo verifica la tabella associata al processo citata in precedenza e, in caso di riferimento valido, cerca un frame vuoto, se questo è presente, si procede come in seguito. Altrimenti serve usare un algoritmo di rimpiazzamento delle pagine per scegliere la vittima, si procede con lo swap out della vittima su disco e si procede in modo normale.



In assenza di frame liberi, sono necessari due accessi alla memoria: uno per lo swap out della vittima, uno per lo swap in del frame da caricare. Il risultato è che il **tempo di page fault raddoppia**.

Per ottimizzare ciò utilizziamo un **dirty bit** nella page table che impostiamo a 1 se la pagina è stata modificata dal momento in cui viene caricata. Scriveremo su disco soltanto le vittime il cui dirty bit = 1. Le altre non sono state modificate, dunque non è necessario il primo accesso.

Il problema a cui risponde il rimpiazzamento delle pagine è **quale pagina rimpiazzare**, l'allocazione dei frame risponde alla domanda **quanti frame assegnare a un processo al momento dell'esecuzione**.

Algoritmi di Rimpiazzamento

Come detto, l'obiettivo di questi algoritmi è quello di minimizzare il tasso di page fault. Per valutare questi algoritmi eseguiamo una particolare stringa di riferimenti a memoria (*reference string*) e calcoliamo il numero di page fault che si verificano su tale stringa. È necessario conoscere il numero di frame disponibili per il processo.

Supponiamo di avere una pagina da 100 byte e la seguente lista di indirizzi:

- 100, 604, 128, 130, 256, 260, 264, 268
- *Reference String*: 1, 6, 1, 1, 2, 2, 2, 2,

In realtà la reference string è soltanto composta da 1, 6, 1, 2. Questo siccome gli accessi consecutivi ad una stessa pagina generano al massimo 1 page fault.

Il tasso di page fault è **inversamente proporzionale** al numero di frame.

Algoritmo FIFO

La prima pagina introdotta è anche la prima ad essere rimossa. Si dice che questo algoritmo sia cieco, in quanto non va a valutare l'importanza della pagina rimossa. L'importanza della pagina che viene rimossa è dato dalla frequenza della pagina nella reference string.

Questo algoritmo tende ad **aumentare il tasso di page fault**. Soffre dell'**anomalia di Belady**: a volte più frames generano un maggior numero di page fault

- Reference string

1	2	3	4	1	2	5	1	2	3	4	5
---	---	---	---	---	---	---	---	---	---	---	---

- Con 3 frame 9 page fault

1	1	1	4	4	4	5	5	5	5	5	5
	2	2	2	1	1	1	1	1	3	3	3
		3	3	3	2	2	2	2	2	4	4

- Con 4 frame 10 page fault

1	1	1	1	1	1	5	5	5	5	4	4
	2	2	2	2	2	2	1	1	1	1	5
		3	3	3	3	3	3	2	2	2	2
			4	4	4	4	4	4	3	3	3

Algoritmo Ideale

Questo algoritmo garantisce il minimo numero di page fault. L'idea è quella di rimpiazzare le pagine che non saranno utilizzate per il periodo di tempo più lungo. Il problema diventa come ricavare questa informazione: è richiesta infatti conoscenza anticipata della stringa dei riferimenti (similmente a SJF). L'implementazione è difficile, richiede supporto HW

È un algoritmo utile come riferimento per altri algoritmi ma sono necessarie approssimazioni.

Algoritmo LRU

Si tratta di un'approssimazione dell'algoritmo ottimo. Utilizza il passato recente come previsione del futuro. Si rimpiazza la pagina che non viene utilizzata da più tempo

1	2	3	4	1	2	5	1	2	3	4	5
1	1*	1*	1*	1	1	1	1	1	1	1*	5
	2	2	2	2*	2	2	2	2	2	2	2
		3	3	3	3*	5	5	5	5*	4	4
			4	4	4	4*	4*	4*	3	3	3

L'implementazione di questo algoritmo non è banale, dato che può risultare complesso ricavare il tempo dell'ultimo utilizzo, e può richiedere molto hardware addizionale.

È possibile implementare questo algoritmo tramite **contatore**: ad ogni pagina viene associato un contatore, ogni volta che la pagina viene referenziata, il clock di sistema è copiato nel contatore. Ogni volta viene rimpiazzata la pagina con il valore del contatore più piccolo.

Un'altra possibile implementazione è quella tramite **stack**: viene mantenuto uno stack di numero di pagina. Per ogni riferimento ad una pagina, questa viene messa in cima allo stack. L'aggiornamento richiede l'estrazione di un elemento interno allo stack. Il fondo dello stack sarà la nostra pagina LRU

È possibile applicare alcune **approssimazioni**, come ad esempio l'utilizzo del **bit di reference** associato ad ogni pagina e inizialmente posto a 0. Quando la pagina viene referenziata, viene messo ad 1 dall'hardware. Nel rimpiazzamento viene scelta una pagina che ha il bit a 0. Non viene verificato in questo caso l'ordine di riferimento delle pagine

Una possibile alternativa a ciò è l'utilizzo di **più bit di reference** per ogni pagina che vengono aggiornati periodicamente. Tali bit vengono utilizzati come valore intero per la scelta di LRU.

Un'altra approssimazione possibile è quella del **rimpiazzamento second chance**, il principio di base è quello di una FIFO circolare, basato su bit di reference. Se tale bit è a 0, il frame viene rimpiazzato, se il bit è a 1, metto a 0 il bit ma la pagina viene lasciata in memoria e analizzo la pagina successiva utilizzando la stessa regola. Una possibile variante è l'utilizzo di più bit di reference

Altre buone tecniche sono quelle **basate su conteggio**:

- **least frequently used**: viene contato il numero di riferimenti fatti a ogni pagina e viene rimpiazzata la pagina con il conteggio più basso. Questo può non corrispondere a pagina LRU dato che se ci sono molti riferimenti iniziali una pagina può avere conteggio alto e non essere eseguita da molto tempo.
- **most frequently used**: è l'esatto opposto di LFU, la pagina con conteggio più basso è probabilmente appena stata cariata e dovrà essere presumibilmente utilizzata ancora.

Allocazione dei Frame

Data una memoria di N frame ed M processi vogliamo capire quanti frame allocare per ogni processo. Abbiamo i seguenti vincoli:

- ogni processo necessita di un minimo numero di pagine per essere eseguito
 - l'istruzione interrotta da un page fault deve essere fatta ripartire

La conseguenza di ciò è che il numero minimo di pagine è uguale al massimo numero di indirizzo specificabili in una istruzione, i valori tipici sono perciò compresi tra i 2 e i 4 frame.

Ad esempio supponiamo di avere una macchina nella quale l'istruzione `move` occupa 6 byte e può coinvolgere 2 pagine, di conseguenza avremo bisogno di 2 pagine per l'istruzione, 2 pagine per gestire la sorgente e 2 ulteriori pagine per gestire la destinazione

L'allocazione dei frame può essere **fissa**, in cui un processo si vede allocato sempre lo stesso numero di frame, oppure **variabile**, in cui il numero di frame allocati ad un processo può variare durante l'esecuzione

Il rimpiazzamento può avvenire in **contesti diversi**:

- **locale**: ogni processo seleziona vittime soltanto tra i suoi frame
- **globale**: ogni processo sceglie frame dall'insieme di tutti i frame, ciò permette di migliorare il throughput

Allocazione Fissa

Questo schema prevede due alternative:

- **allocazione in parti uguali**: dati m frame ed n processi alloca a ogni processo m/n frame
- **allocazione proporzionale**: alloca secondo la dimensione del processo, ma questo può essere un parametro non significativo e potrebbe in generale importare di più la priorità del processo

Allocazione Variabile

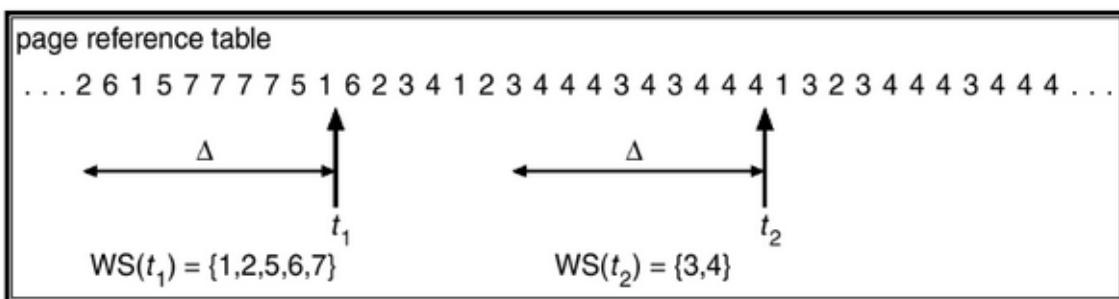
Permette di modificare in modo dinamico le allocazioni ai vari processi sulla base delle effettive necessità. Il problema è scegliere in base a cosa modificare: Si può rispondere a questa domanda tramite due strade:

- calcolo del **working set**
- calcolo del **page fault frequency**

Vediamo come funziona la strada con il **calcolo del working set**:

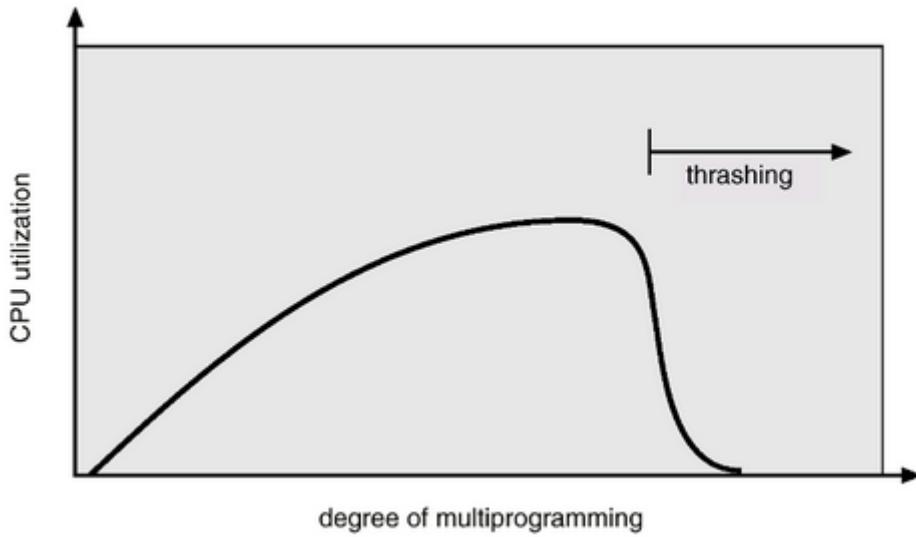
abbiamo in questo modo un criterio per rimodulare l'allocazione dei frame in base alle richieste effettive di ogni processo. Viene sfruttato il modello della **località**: un processo passa da una località di indirizzi all'altra durante la sua esecuzione. Un processo necessita quindi di un numero di frame che sia pari alla sua località.

Utilizzando questo modello abbiamo che P_i riceve frame sufficienti a mantenere in memoria il suo **working set** $WS_i(t, \Delta)$. Definiamo Δ come **finestra temporale** del working set, se tale valore è piccolo è poco significativo, se troppo grande può coprire varie località. Ad ogni istante predefinito andrà a monitorare il numero di pagine a cui ho fatto riferimento, notare che tale dato è presente per ogni pagina.



Se mi rendo conto che la richiesta totale di frame (somma del working set di tutti i processi) è maggiore del numero totale di frame che ho in memoria si verifica **trashing**. Questo vuol dire che un processo spende tempo di CPU continuando a effettuare swap di pagine da e verso la memoria. Questo fenomeno è la conseguenza di un numero di frame troppo basse di frame che porta ad un *circolo vizioso*.

Andando ad approfondire il **trashing**, questo comporta che il livello complessivo di utilizzo della CPU precipita:



Vediamo quindi che è necessaria una strategia abbastanza precisa per il calcolo del working set in modo da non finire nel fenomeno del thrashing. Abbiamo a disposizione un'**approssimazione tramite bit di reference**: un timer interrompe in modo periodico la CPU. All'inizio di ogni periodo, i bit di reference sono tutti quanti posti a 0 e, ad ogni interruzione del timer, vengono controllate le pagine: quelle con bit di reference a 1 rimangono nel working set, quelle con tale bit ancora a 0, vengono scartate. L'accuratezza di questo metodo dipende dal numero di bit di reference utilizzati e dalla frequenza delle interruzioni.

Un'altra strategia possibile è quella dell'utilizzo della **frequenza dei page fault** che è una soluzione più accurata al nostro problema. In pratica stabiliamo un tasso di page-fault che viene ritenuto accettabile. Nel momento in cui il numero di page fault è troppo basso il processo rilascerà dei frame (ne ha troppi e potrebbe verificarsi thrashing), se tale tasso effettivo risultasse troppo alto, il processo andrebbe ad acquisire dei frame.

Capitolo 10 - Memoria Secondaria

Federico Segala

Tipologia del Supporto

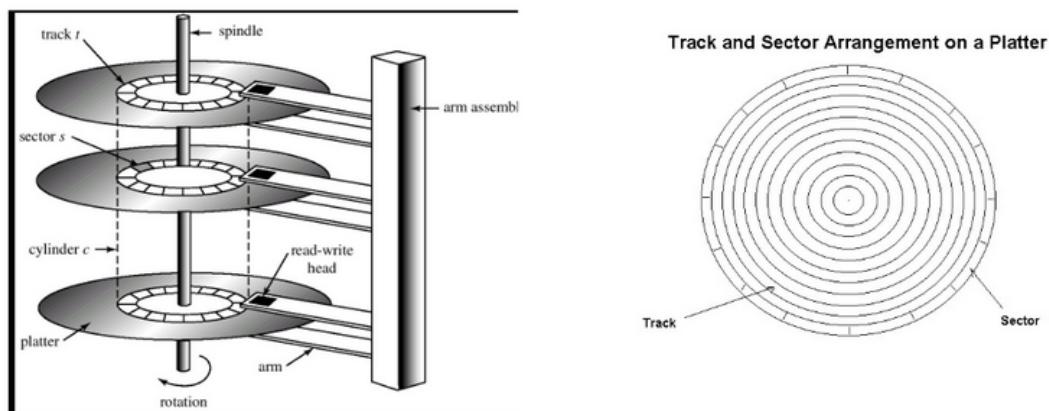
I dispositivi di memoria secondaria servono a tenere traccia permanente di informazione, nel corso del tempo si sono susseguite diverse tecnologie che in questo capitolo andiamo ad approfondire ## Nastri Magnetici Sono strutturalmente composti da una sottile striscia di materiale plastico rivestita da **materiale magnetizzabile**. Al 2011 la capacità massima si attestava sui 5TB. L'accesso è **sequenziale** di conseguenza sono molto più lenti di supporti come la memoria principale o banalmente dei dischi magnetici in termini di tempi di accesso. Questa lentezza nell'accesso è data dal fatto che per accedere ad un dato è necessario il riposizionamento di una **testina** che può richiedere decine di secondi.

Al giorno d'oggi il loro maggiore impiego è nell'ambito del **backup** dato che si tratta di supporti molto economici e che occupano poco spazio.

Dischi Magnetici

Si parla di **piatti di alluminio** ricoperti da materiale ferromagnetico. Al 2017 la capacità si attesta sui 10TB, l'evoluzione tecnologica ha permesso di arrivare ad un form factor sempre più piccolo.

La lettura e la scrittura avvengono per mezzo di una **testina sospesa** sulla superficie magnetica.



Definiamo **settore** la più piccola unità di informazione che può essere letta o scritta su disco. La dimensione di questi settori è variabile [32B, 4K B] e per motivi di efficienza si tende a raggruppare i settori in **cluster**, un file tipicamente occupa sempre almeno un cluster.

Definiamo il **tempo di accesso** come **somma** dei seguenti tempi:

- **seek time**: si tratta del tempo necessario per spostare la testina sulla traccia corretta (10 ms)
- **latency time**: tempo necessario a posizionare il settore desiderato sotto la testina (4 ms)
- **transfer time**: tempo necessario al settore per passare sotto la testina (disk to buffer: 1000 Mbit/s, buffer to computer: 400 MByte/sec)

Il tempo di accesso è fortemente dipendente dal tempo di **seek**, gli altri due tempi non influenzano la velocità di accesso in modo così importante rispetto a quest'ultimo.

Possiamo cercare di minimizzare tale valore tramite degli algoritmi di **scheduling degli accessi al disco** in modo da:

- minimizzare il tempo di accesso totale
- massimizzare la banda che va a misurare la velocità effettiva

Dispositivi a Stato Solido

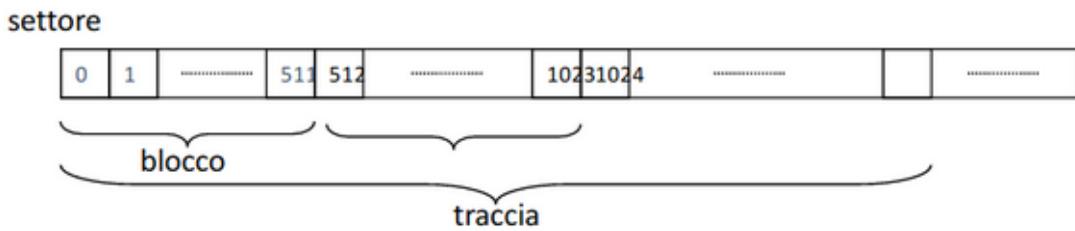
Vengono utilizzati dei **chip** per memorizzare i dati in modo non volatile. Utilizzano la stessa interfaccia dei dischi fissi e possono dunque essere facilmente rimpiazzati. Si tratta di dispositivi molto meno soggetti a danni, con lo svantaggio dato dal fatto che queste memorie hanno un limite sul numero di write.

Un grosso vantaggio introdotto però da questi dispositivi consiste nella velocità di lettura e scrittura e nel fatto che non sia necessaria la deframmentazione periodica del disco.

Scheduling degli Accessi a Disco

Un disco è considerabile a livello logico come un **vettore unidimensionale** di **blocchi** (cluster). Un blocco corrisponde all'unità minima di trasferimento. Tale vettore viene mappato in modo sequenziale nei vari settori del disco:

- settore 0: primo settore della prima traccia del cilindro più esterno
- la numerazione procede in modo gerarchico per settori, tracce, cilindri



Un processo che necessiti di effettuare input/output esegue una system call, a questo punto il sistema operativo utilizza la vista logica del disco: la sequenza di accessi corrisponde alla sequenza di indici del vettore.

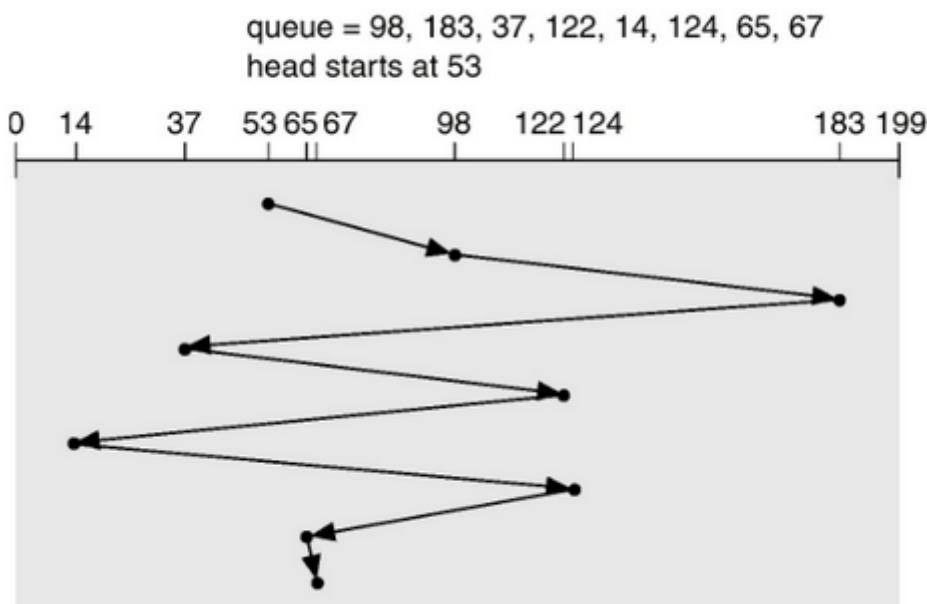
All'interno di una richiesta di I/O sono memorizzate altre informazioni quali ad esempio il tipo dell'accesso che si intende effettuare, l'indirizzo di trasferimento, la quantità di dati che è necessario trasferire, ...

Nello sviluppare algoritmi di scheduling degli accessi al disco vogliamo raggiungere il **compromesso ottimale tra costo ed efficacia**. Tali algoritmi vengono valutati per mezzo di una *sequenza di accessi di esempio*:

- inizialmente la testina si trova sulla traccia 53
- accessi: 98, 183, 37, 122, 14, 124, 65, 67
- intervallo dei valori ammissibili = [0, 199]

FCFS

In questa semplice implementazione le richieste sono processate nell'ordine in cui arrivano.



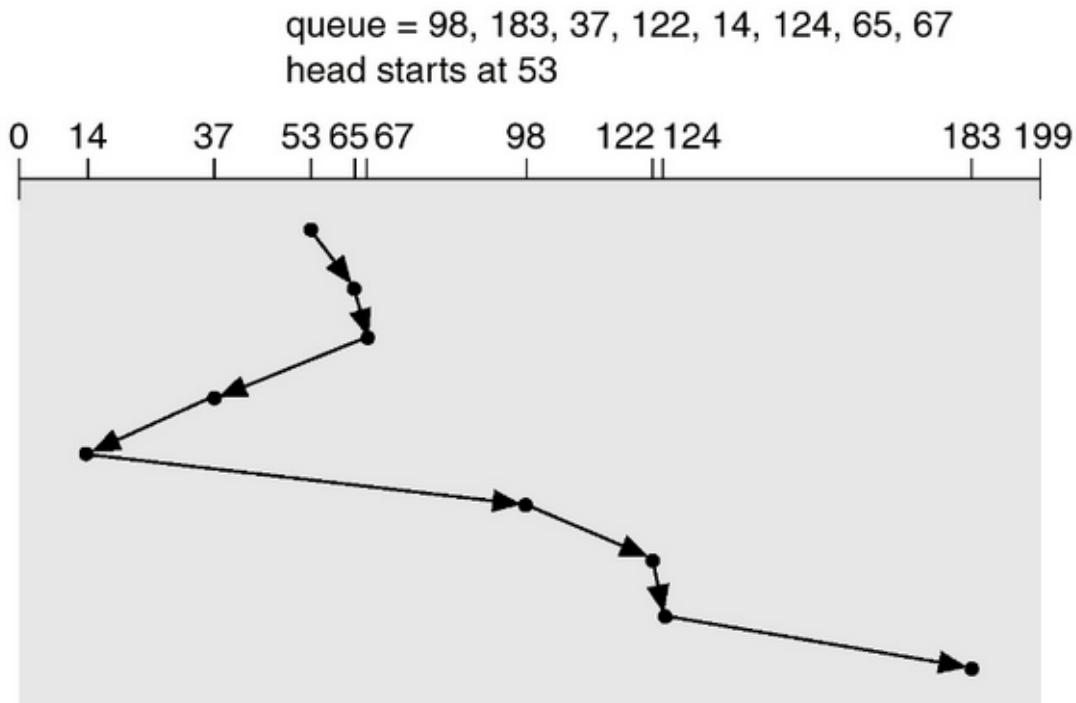
Con questa particolare sequenza di accessi si ottiene uno spostamento totale della testina di 640 tracce

Shortest Seek Time First

Si tratta di una scelta più efficiente, in quanto viene selezionata la richiesta con il minimo spostamento rispetto alla posizione attuale della testina.

Si può notare una cera analogia con l'algoritmo SJF per lo scheduling dei processi, non dovrebbe quindi sorprendere la possibilità di avere **starvation** per alcune richieste.

Questo algoritmo va a migliorare FCFS ma non è ancora ottimo.



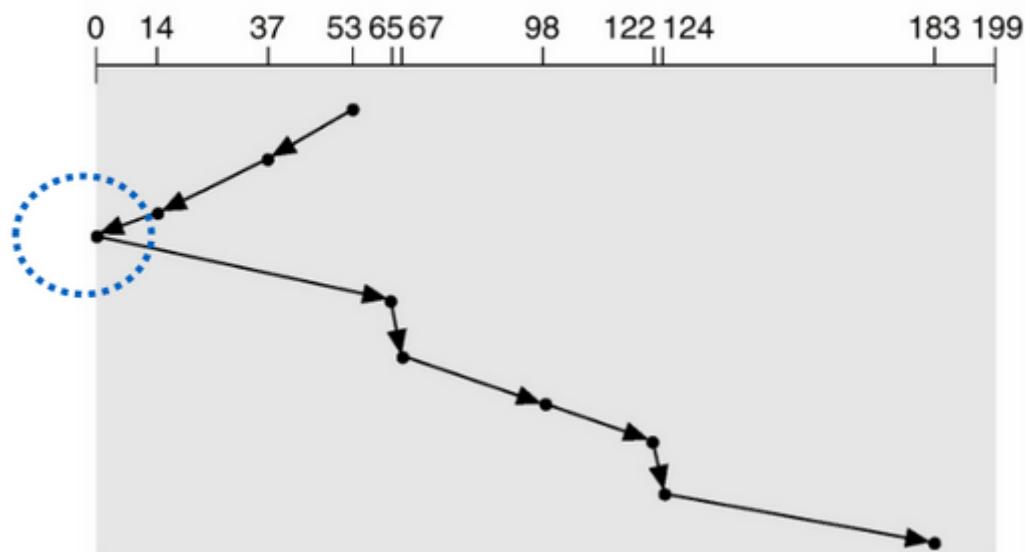
Si ottiene in questo caso uno spostamento totale della testina pari a 236 tracce.

L'osservazione della *natura dinamica delle richieste* porta allo sviluppo del seguente algoritmo

SCAN

La testina parte da un'estremità del disco, si sposta verso l'altra estremità, servendo tutte le richieste correnti, arrivato all'altra estremità, riparte nella direzione opposta servendo le nuove richiesta. Si dice anche **algoritmo dell'ascensore**.

queue = 98, 183, 37, 122, 14, 124, 65, 67
head starts at 53



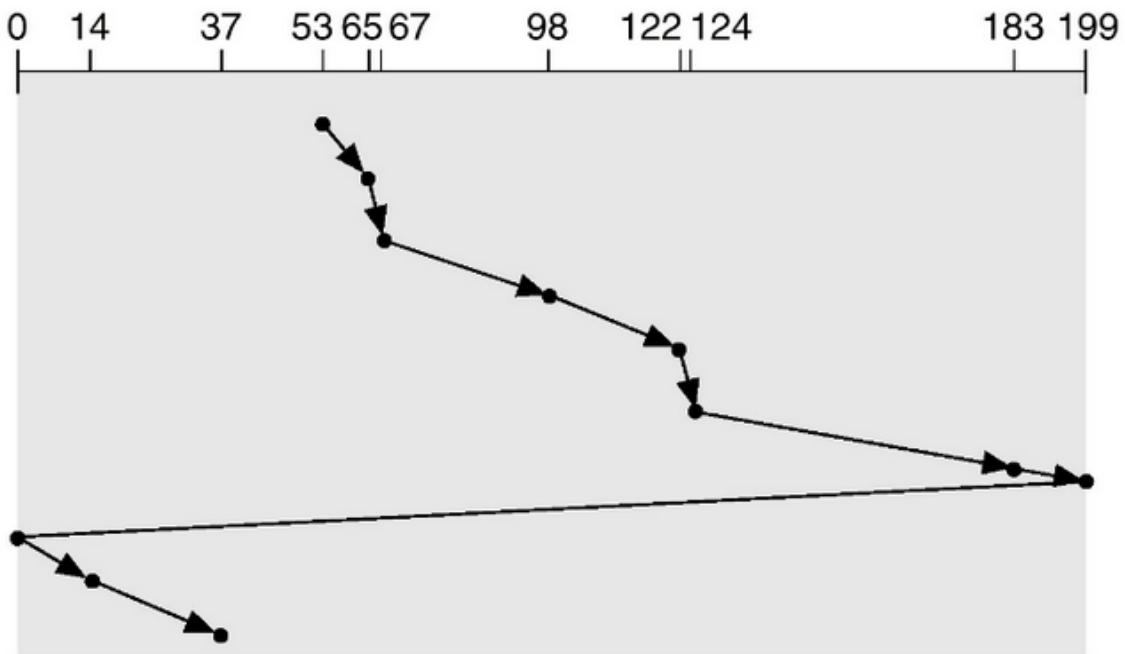
Anche in questo caso si arriva ad avere uno spostamento totale della testina pari a 236 tracce. Abbiamo a disposizione alcune varianti per ottimizzare il funzionamento di SCAN, tra le più importanti ricordiamo quelle che seguono

C - SCAN

Si tratta di uno SCAN implementato in modo **circolare**, funziona ugualmente allo SCAN ma una volta che la testina arriva ad un'estremità, riparte immediatamente da 0 senza servire altre richieste. Il disco viene visto in pratica come una lista circolare. Questa variante porta ad ottenere un tempo di attesa più uniforme rispetto a SCAN.

Il problema di questo algoritmo consiste nel fatto che alla fine di una scansione ci sarà un maggior numero di richiesta all'estremo opposto, un po' come uno spalatore di neve che spala durante una bufera di neve: dove ha spalato ma non sta spalando continua ad accumularsi neve.

queue = 98, 183, 37, 122, 14, 124, 65, 67
head starts at 53

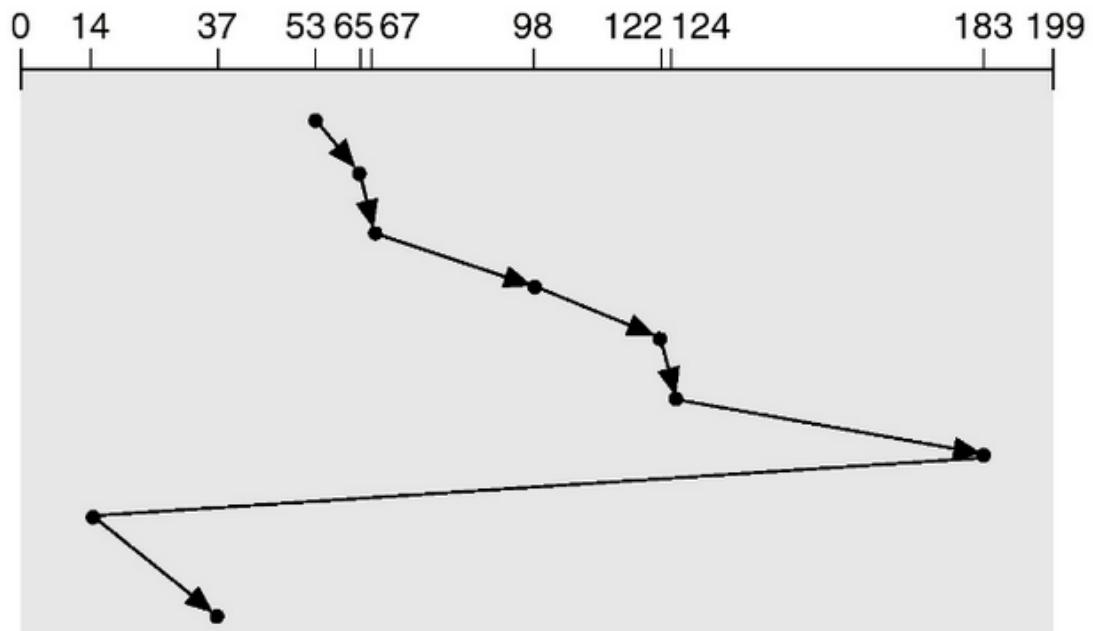


Tale algoritmo comporta uno spostamento pari a 382 tracce

(C-)LOOK

La testina in questo caso non arriva all'estremità del disco, ma cambia direzione (**look**) o riparte dalla prima traccia (**c-look**) non appena si accorge che non ci sono più richiesta in quella particolare direzione

queue = 98, 183, 37, 122, 14, 124, 65, 67
head starts at 53



In questo caso avremo uno spostamento totale pari a 322 tracce

N-step SCAN

Per evitare che la testina rimanga sempre nella stessa zona, la coda delle richieste viene partizionata in **più code** di dimensione massima N. Quando una coda viene processata per il servizio (scan in una direzione), gli accessi in arrivo riempiono altre code. Dopo che una coda è stata riempita non sarà possibile riordinare le richieste.

Le code sature verranno servite nello scan successivo. Bisogna stare attenti alla scelta del numero di code, nel senso che nel caso in cui abbiamo N troppo grande si rischia di degenerare in un normale SCAN altrimenti con N = 1, si ottiene un FCFS

LIFO

Si tratta di una strategia *last in first out*, in certi casi può essere utile schedulare gli accessi in base all'ordine inverso di arrivo. L'idea alla base di ciò è che potrebbe tornare utile nel caso di accessi con elevata località, ad esempio una serie di accessi vicini.

Il problema, anche in questo caso è la possibilità di **starvation**

Analisi Conclusiva

Nessuno degli algoritmi visti in questa sezione è ottimo, l'algoritmo ottimo in realtà esiste ma la sua complessità non va a ripagare del risparmio potenzialmente ottenibile con le tecniche appena introdotte. Abbiamo diversi fattori che vanno ad influenzare l'analisi dei nostri algoritmi di scheduling:

- distribuzione degli accessi che possono essere
 - molti medio-piccoli oppure pochi molto grandi
 - in generale più l'accesso è grande, più il peso relativo del tempo di seek diminuisce
- organizzazione delle informazioni su disco
 - l'accesso alle directory è un fattore essenziale, queste sono tipicamente posizionate lontane dalle estremità del disco.

In generale lo scheduling del disco viene implementato come un **modulo indipendente dal sistema operativo** e i migliori schemi per sistemi caratterizzati da molti accessi al disco sono SCAN e C-SCAN.

Gestione del Disco

La gestione del disco consiste nel dare all'utente la possibilità di eseguire sul disco in questione diversi tipi di operazioni, quali ad esempio:

Formattazione del Disco

Il disco viene diviso in settori che il controllore può leggere o scrivere. Per utilizzare il disco come un contenitore di file, il sistema operativo deve memorizzare le proprie strutture sul disco:

- partizionamento del disco in uno o più gruppi di cilindri
 - formattazione logica per creare un file system
 - programma di boot per inizializzare il sistema
 - programma di bootstrap memorizzato nella ROM carica il bootstrap dai blocchi di boot
 - tale bootstrap si occupa di caricare i driver dei dispositivi e di lanciare l'avvio del sistema operativo
- Vista concettuale:
• Disco "nudo"
-
- Formattazione fisica:
• Suddivisione in settori
• Identificazione dei settori
• Aggiunta di spazio per correzione di errori (ECC)
- | | | | | | | | | | |
|---|--|-----|---|--|-----|---|--|-----|--|
| 1 | | ECC | 2 | | ECC | 3 | | ECC | |
|---|--|-----|---|--|-----|---|--|-----|--|

1	FAT	ECC	2	FL	ECC	3	DIR	ECC	
---	-----	-----	---	----	-----	---	-----	-----	--
- Formattazione logica:
• File system
• Lista spazio occupato (FAT) e libero (Free List – FL)
• Directory vuote
- | | | | | | | | | | |
|---|-----|-----|---|----|-----|---|-----|-----|--|
| 1 | FAT | ECC | 2 | FL | ECC | 3 | DIR | ECC | |
|---|-----|-----|---|----|-----|---|-----|-----|--|

Gestione dei Blocchi Difettosi

Chiaramente in un disco può succedere che alcuni blocchi si corrompano e che si vada a perdere l'informazione in essi contenuti. Abbiamo a disposizione in un sistema operativo un modulo che si occupa di gestire tali blocchi.

Una pratica utilizzata pressoché sempre è quella dell'**error correction code** che serve a capire (in lettura o scrittura) se un settore contiene o meno dati corretti. Lo schema tipico di funzionamento è il seguente:

- il controllore legge un settore, incluso il suo error correction code
- viene calcolato l'error correction code sul blocco appena letto
- se il risultato è diverso: ERROR (**bad block**)

Nel caso di bad block abbiamo due possibilità di risoluzione:

- **offline:** viene effettuata durante la formattazione logica si individuano i bad block e si mettono in una lista che indica che tali blocchi vengono esclusi dal nostro sistema operativo. Le aree danneggiate vengono marcate in modo che si possa evitare di

scrivere in tali settore. Questo lavoro si può effettuare tramite comando utente ad esempio chkdsk

- **online:** il controllore mappa il bad block su dei blocchi buoni che erano stati mantenuti come scorta. Tali blocchi vengono riservati proprio per questa evenienza. Questo non permette di recuperare il dato perso, ma alla prossima scrittura evita di utilizzare tale blocco senza necessariamente escluderlo dai blocchi utilizzabili. Ciò avviene in modo *trasparente* e viene effettuato dal controllore. Un possibile svantaggio di questa tecnica consiste nel fatto che potrebbe inficiare i meccanismi di scheduling (andando a spostare la testina in posizioni che non sono prevedibili). Come soluzione a ciò si può prevedere di allocare degli spare block per ogni cilindro.

Gestione dello Spazio di Swap

Come già visto si tratta di uno spazio usato dalla memoria virtuale come estensione della RAM. Abbiamo a disposizione diverse opzioni.

La prima opzione è quella di ricavare tale porzione a partire dal normale **file system**. In questo modo si può accedere a questa partizione usando comuni primitive per accesso ai file. Si tratta però di una strategia inefficiente, dato che comporta un costo addizionale per attraversare le strutture dati per le directory e i file.

La seconda opzione è quella di mantenere lo spazio di swap in una **partizione separata**, accessibile tramite system call separate. In questo modo tale partizione non contiene informazioni o strutture relative al file system e viene utilizzato uno specifico gestore dell'area di swap (**swap daemon**). Si tratta della soluzione più **tipica**.

Capitolo 11 - File System

Federico Segala

Interfaccia

Per **file system** si intende quella componente del sistema operativo che fornisce il meccanismo per la memorizzazione e l'accesso a dati e programmi. Consiste di due parti fondamentali:

- **collezione di file**
- **struttura di cartelle**

Concetto di File

Il sistema operativo astrae dalle caratteristiche fisiche dei supporti di memorizzazione fornendo una **visualizzazione logica** di questi. Per **file** intendiamo uno **spazio di indirizzamento logico e contiguo**, in parole più semplici, un insieme di informazioni correlate identificate da un nome. Esistono differenti tipologie di file, tra cui dati numerici, caratteri, binari, programmi, ...

Tipicamente i file vengono memorizzati su disco nella *struttura della directory*, nella quale per ogni file che contiene manteniamo:

- **nome**: si tratta dell'unica informazione mantenuta in formato leggibile
- **tipo**
- **posizione**: puntatore allo spazio fisico nel dispositivo
- **dimensione**
- **protezione**: controllo su chi può leggere, scrivere o eseguire un determinato file
- **tempo, data, identificazione utente**

Sui file è possibile svolgere diversi tipi di **operazioni**, tra le più importanti andiamo a descrivere le seguenti:

- **creazione di un file**: questa operazione consiste principalmente nella ricerca di spazio libero su disco e del salvataggio degli attributi del file nella struttura della directory
- **scrittura su file**: esiste una system call che specifica nome e dati da scrivere, per effettuare ciò avremo bisogno di un puntatore alla locazione della prossima scrittura

- **lettura da file:** abbiamo a disposizione una system call che specifica il nome del file e dove andare a salvare i dati che vengono letti dalla memoria, anche in questo caso abbiamo bisogno di un puntatore alla locazione della prossima lettura (lo stesso usato in lettura)
- **riposizionamento** all'interno del file: viene aggiornato il puntatore della posizione corrente, in modo da fare letture e scritture in posizioni diverse
- **cancellazione:** libera lo spazio associato al file e l'elemento corrispondente nella directory
- **troncamento:** mantiene gli attributi della directory inalterati, ma cancella il contenuto del file
- **apertura:** viene ricercato il file nella struttura della directory su disco, in seguito il file viene copiato in memoria e viene inserito un riferimento all'interno della **tabella dei file aperti**. Ci sono due tabelle per gestire i file aperti:
 - una tabella per ogni processo, contenente riferimenti per file aperti relativi al processo
 - tabella per tutti i file aperti da tutti i processi, che contiene dati indipendenti dal processo (posizione su disco, dimensione, accessi, ...)
- **chiusura:** viene effettuata una copia del file presente in memoria alla corrispondente posizione su disco

Metodi di Accesso

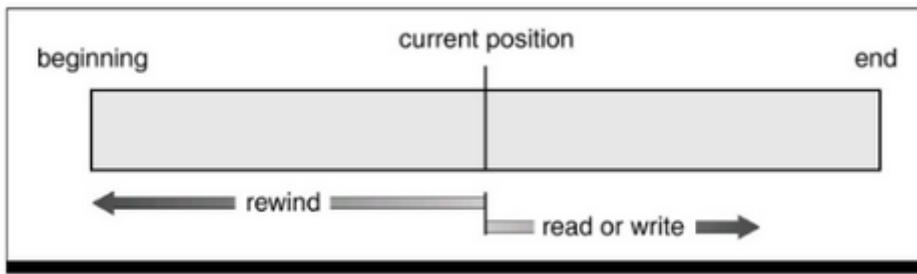
Per accedere ad un file abbiamo a disposizione diverse modalità, per ognuna delle quali andremo a specificare le operazioni permesse.

Accesso Sequenziale

Si tratta del classico approccio adottato da *editor e compilatori*. Le operazioni permesse sono

- **read next**
- **write next**
- **reset (rewind)**

Si noti come utilizzando questo approccio non sia possibile effettuare il rewrite, in quanto si rischia di arrivare ad una inconsistenza scrivendo qualcosa a metà del file andando a cancellare ciò che sta dopo



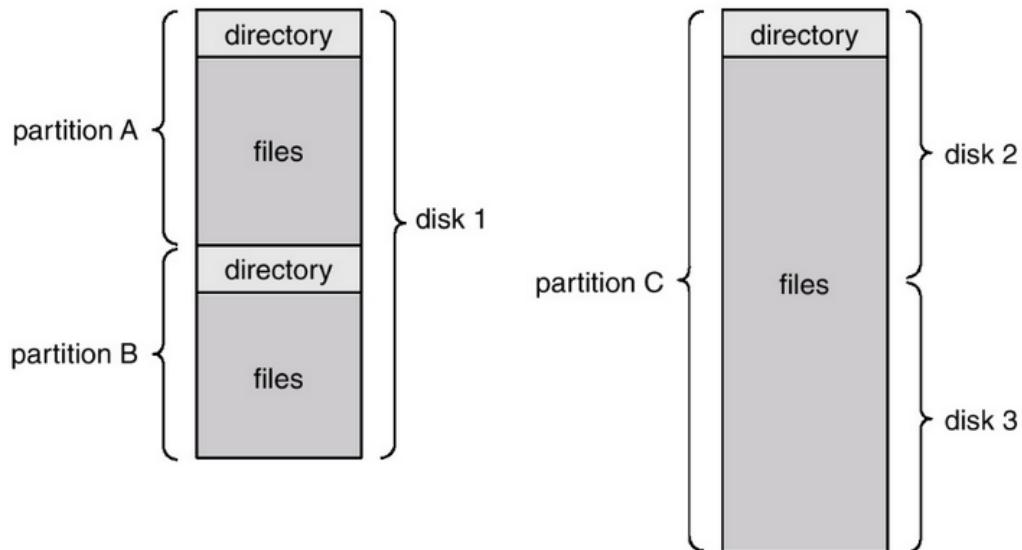
Accesso Diretto

Si tratta del classico approccio utilizzato nei *database*. Le operazioni consentite sono

- **read n**
- **write n**
- **position to n**
- **read next**
- **write next**
- **rewrite n**

Struttura delle Directory

Prima di andare a parlare in modo esplicito della struttura delle directory, è bene mostrare uno schema di base che spieghi come è **organizzato il file system**:



Una **directory** non è altro che una **collezione di nodi** che contengono informazioni sui file che risiedono nel disco. L'insieme delle directory va a determinare la struttura del file system. Per ogni file che contiene una directory andrà a memorizzare le seguenti informazioni:

- nome
- tipo

- indirizzo
- lunghezza attuale
- lunghezza massima
- data di ultimo accesso
- data di ultima modifica
- possessore
- informazioni di protezione

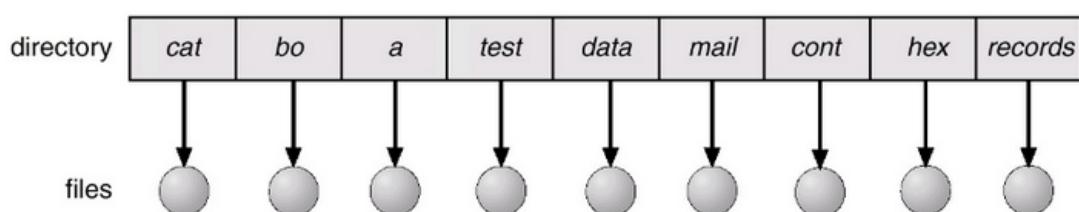
Come per i file, anche sulle directory è possibile svolgere delle **operazioni** che sono le seguenti:

- **aggiunta di un file**
- **cancellazione di un file**
- **visualizzazione del contenuto** di tale directory (`ls`)
- **rinomina di un file**
- **ricerca di un file**
- **attraversamento del file system**

Gli **obiettivi** dell'introduzione delle directory sono diversi, tra cui quello dell'**efficienza**, ovvero di ottenere un accesso ai file che sia il più rapido possibile. Un altro obiettivo importante è quello che il **naming** dei file sia il più conveniente possibile per gli utenti. Altro importante obiettivo è quello del **raggruppamento** dei file in base a degli specifici criteri.

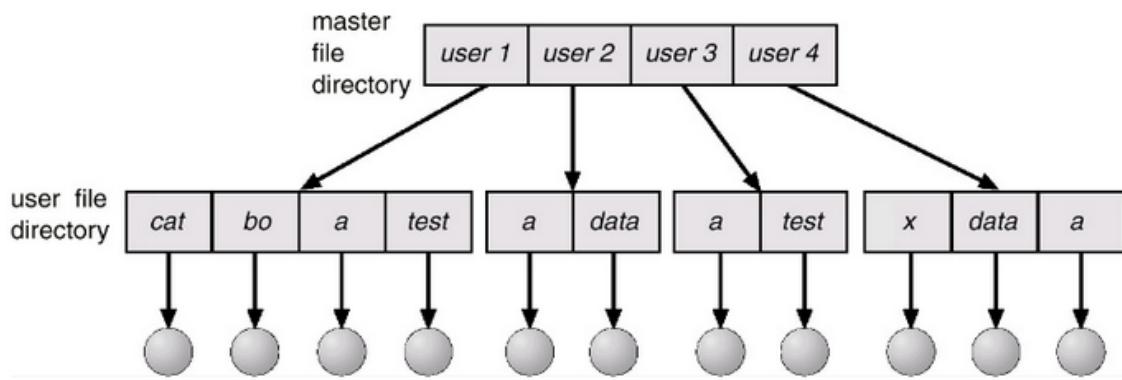
Directory ad un livello

In questa struttura viene utilizzata una sola directory per tutti gli utenti, ovviamente ciò porta a problemi di **nomenclatura e raggruppamento**



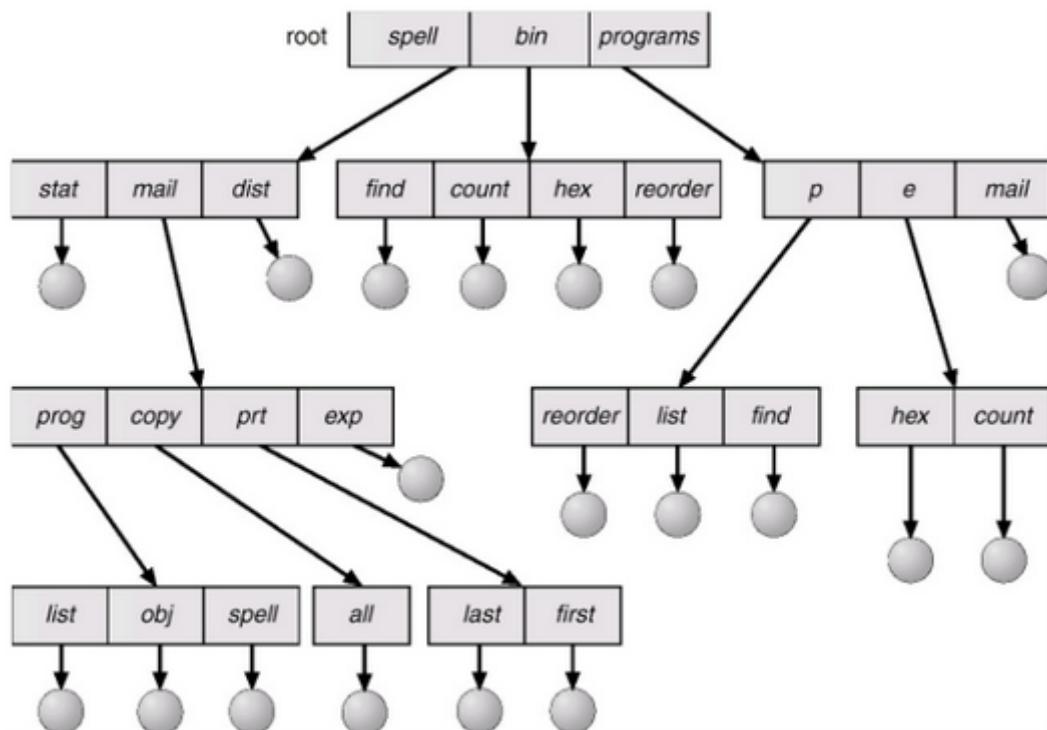
Directory a due livelli

Viene utilizzata una directory separata per ogni utente, viene introdotto per la prima volta il concetto di **percorso**. Si ha in questo caso la possibilità di utilizzare lo stesso nome di file per utenti diversi. Ciò garantisce una ricerca efficiente, ma un basso livello di **raggruppamento**. Il problema che sorge è il dove mettere i programmi di sistema condivisi dagli utenti, in UNIX una soluzione adottata è quella di utilizzare la *variabile di ambiente PATH*



Directory ad Albero

Questa organizzazione delle directory permette ricerca efficiente, e dà possibilità di raggruppamento. Viene introdotto il concetto di **working directory** assieme a **path assoluti e relativi**

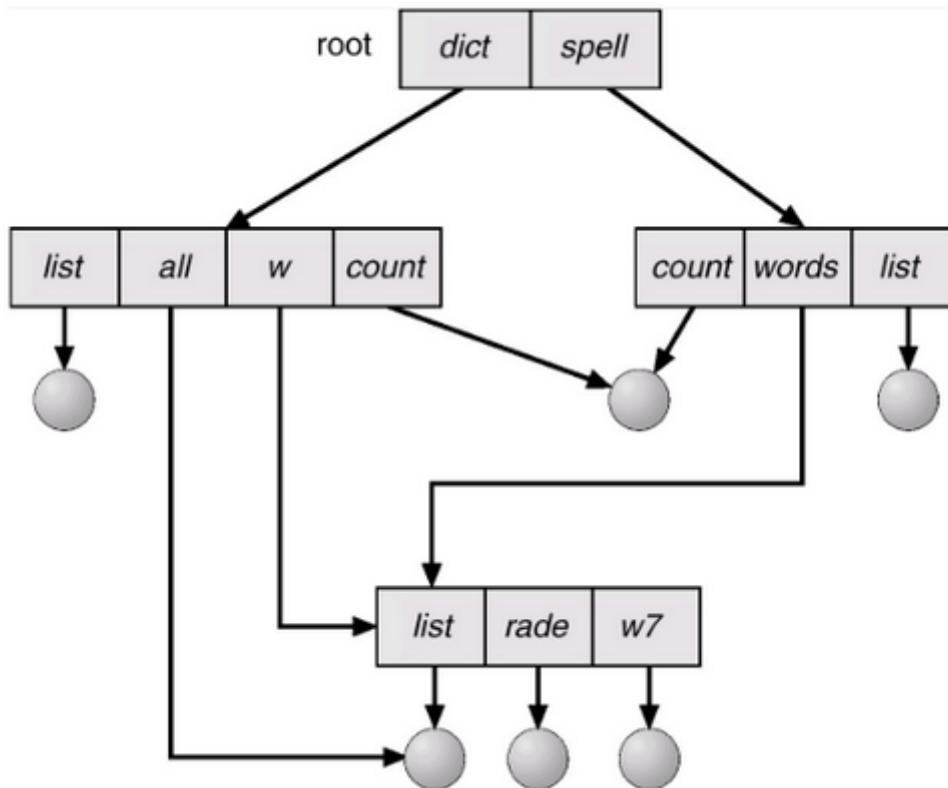


Directory a Grafo Aciclico

Questa struttura viene introdotta in quanto la directory ad albero non permette **condivisione** di file e directory. In questo caso implementiamo la condivisione nel modo seguente:

- **link simbolico**: contiene il pathname del file/directory reale, quando tale file reale viene cancellato il link rimane *pendente*
- **hard link**: viene impostato un contatore che mantiene il numero di riferimenti ad una risorsa. Tale contatore viene decrementato ad ogni riferimento cancellato. Il file viene

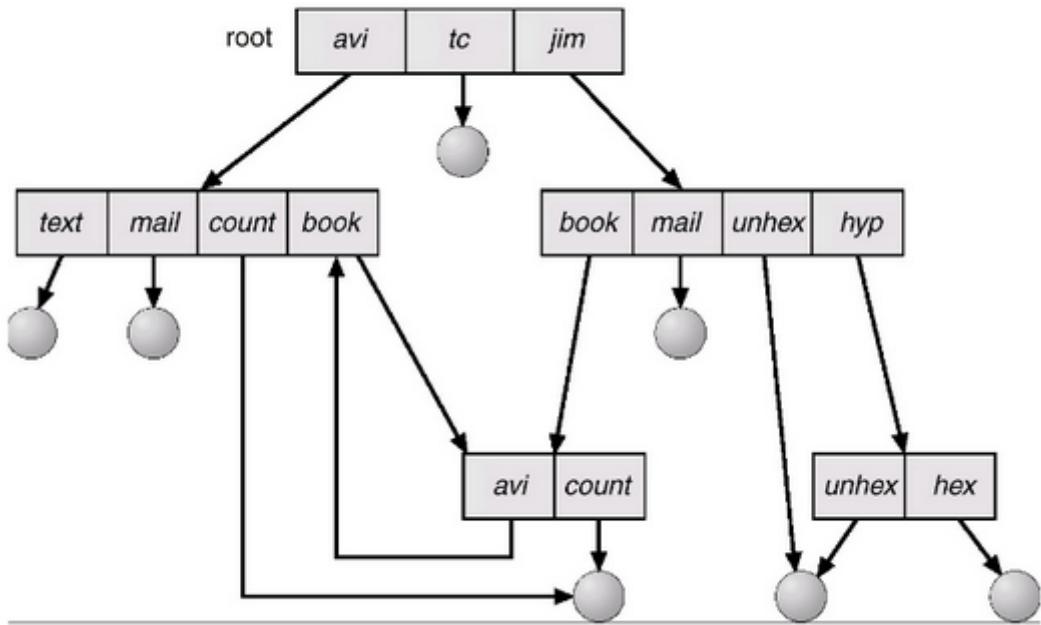
cancellato soltanto quando questo contatore diventa 0



Directory a Grafo Generico

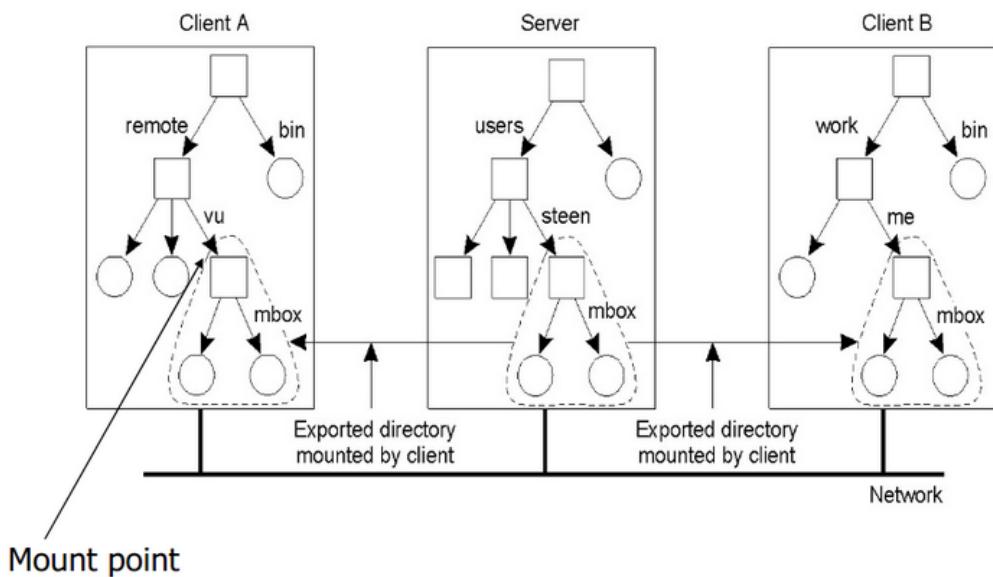
Questo approccio è sicuramente più generale del grafo aciclico, ma è comunque necessario garantire che non si verifichino cicli per evitare loop infiniti nell'attraversare il grafo. Alcune soluzioni sono le seguenti:

- permettere di collegare soltanto file e non directory
- utilizzare garbage collection
- utilizzare un algoritmo di controllo di esistenza di un ciclo ogni volta che si crea un nuovo collegamento



Mount di File System

È possibile realizzare dei **file system modulari** garantendo la possibilità di attaccare e staccare interi file system ad altri file system pre esistenti. Questo avviene tramite `mount` e `umount`. In generale prima di essere utilizzato, un file system deve essere montato. Il punto in cui questo viene montato è detto **mount point**



Condivisione di File

In qualsiasi sistema multiutente che si rispetti è necessario garantire condivisione. Questo è realizzabile tramite uno **schema di protezione**. In sistemi distribuiti i file possono essere condivisi tramite l'utilizzo di una rete.

Protezione

Si tratta di un aspetto fondamentale. È necessario garantire che il possessore di un file possa controllare cosa è possibile fare su un file e chi possa effettuare determinate operazioni su questo. Le operazioni possono essere ad esempio lettura, scrittura, esecuzione, append, ...

La **protezione** viene implementata tramite una **lista di accesso** per file e directory, che va a stabilire un elenco di chi può fare che cosa per ogni file o directory, tale lista può essere molto lungo. In UNIX gli utenti sono raggruppati in **tre classi** differenti:

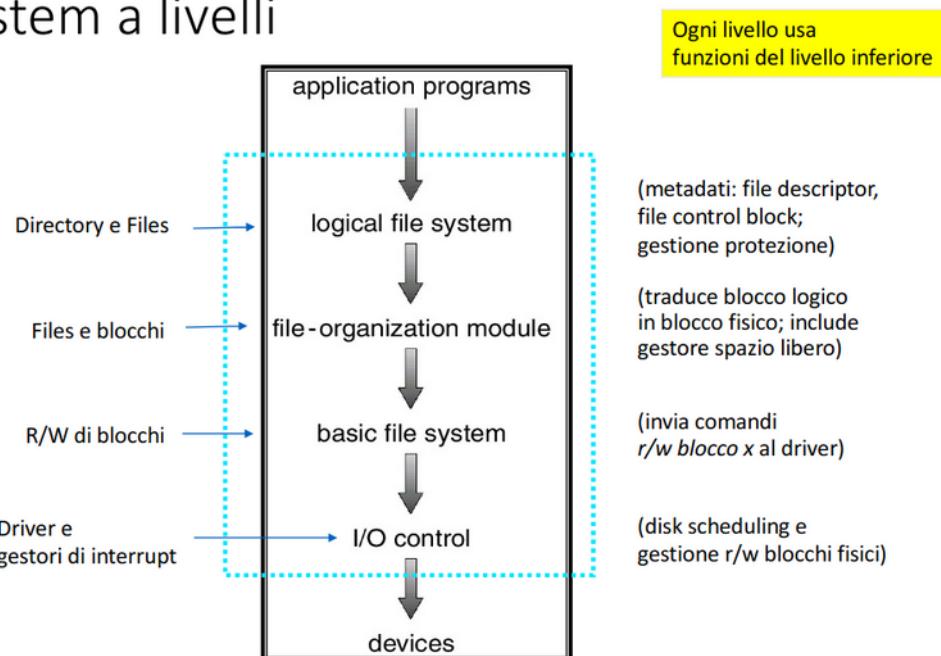
- **owner**
- **gruppo**: ovvero tutti quegli utenti che appartengono allo stesso gruppo del proprietario
- **altri**

Per ognuna di questa classi possono essere associati i permessi **read, write, execute** che possono essere cambiati tramite il comando `chmod`

Implementazione del File System

Quasi tutti i file system moderni sono implementati come un **file system a livelli**, nel quale, analogamente allo stack ISO/OSI, ogni livello utilizza funzioni dei livelli inferiori e ne fornisce ai livelli superiori. Vediamo un semplice schema di cio:

File system a livelli



Implementazione Per la gestione di un file system sono necessarie varie strutture dati, una parte di queste sta su disco, l'altra in memoria. Le caratteristiche di queste strutture dipendono dal sistema operativo e dal tipo di file system, sebbene esistano delle caratteristiche che sono comuni a tutte le tipologie.

Le **strutture** che manteniamo su **disco** per implementare un file system sono le seguenti:

- **blocco di boot**: contiene le informazioni necessarie all'avviamento del sistema operativo
- **blocco di controllo delle partizioni**: contiene vari dettagli riguardanti la partizione, come ad esempio il numero e la dimensione dei blocchi, la lista dei blocchi liberi, la lista dei descrittori liberi, ...
- **strutture di directory**: descrivono l'organizzazione dei file all'interno del sistema operativo
- **descrittori di file (inode)**: contengono vari dettagli sui file e puntatori ai blocchi dati

Vediamo ora le **strutture** mantenute in **memoria**:

- **tabella delle partizioni**: contenente informazioni sulle partizioni attualmente montate
- **strutture di directory**: copia in memoria delle directory a cui si è fatto accesso di recente
- **tabella globale dei file aperti**: copia dei descrittori dei file
- **tabella dei file aperti per ogni processo**: puntatore alla tabella precedente ed informazioni di accesso

Metodi di Allocazione dello Spazio su disco

Andiamo in questa sezione a descrivere come i blocchi su disco sono allocati ai file o alle directory. Gli obiettivi che si pongono questi metodi sono:

- minimizzare tempi di accesso
- massimizzare l'utilizzo dello spazio

Abbiamo diverse alternative che andiamo ora ad esporre

Allocazione Contigua

In questa soluzione, ogni file occupa un insieme di blocchi contigui su disco. Tale tecnica porta i seguenti *vantaggi*:

- entry della directory semplice dato che contiene indirizzo del blocco di partenza e il numero di blocchi
- accesso semplificato dato che accedere al blocco b+1 non richiede lo spostamento della testina rispetto a b, a meno che b non sia l'ultimo blocco di un cilindro e dato che è garantito accesso sia sequenziale che casuale

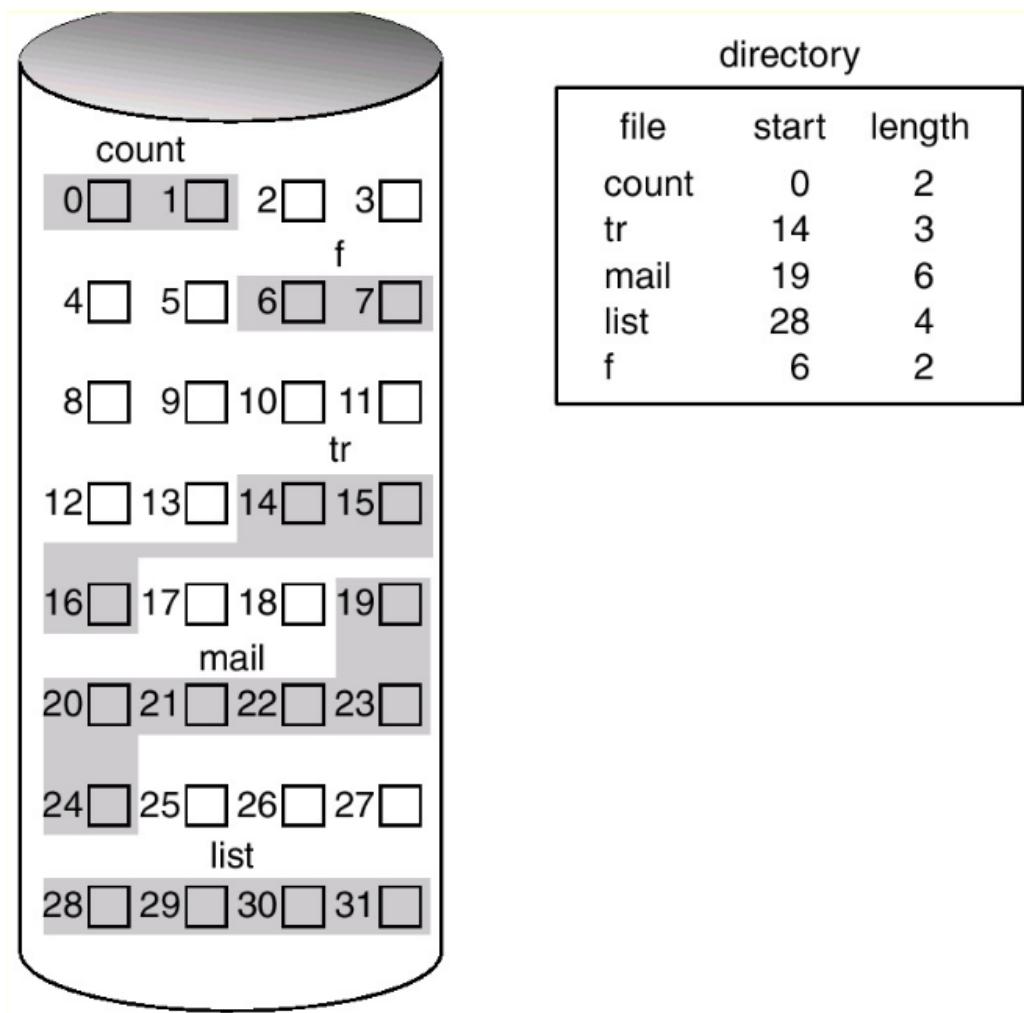
Gli *svantaggi* introdotti da questa tecnica sono simili a quelli dell'allocazione dinamica della memoria:

- scelta di algoritmi best/first/worst fit

- frammentazione esterna
- compattazione periodica dello spazio

Altro problema è dato dal cosa fare nel caso di **crescita di un file**, dato che i file non possono crescere in modo dinamico abbiamo bisogno di capire quanto spazio allocare nel momento in cui viene creato un file. Abbiamo a disposizione diverse soluzioni:

- se il file deve crescere e non c'è spazio generiamo un errore e una terminazione del programma in esecuzione. Per questo motivo si tende a sovrastimare lo spazio necessario: **spreco di spazio**
- trovare un buco più grande e ricopiare tutto quanto in quest'ultimo, questa soluzione è trasparente per l'utente ma rallenta il sistema.



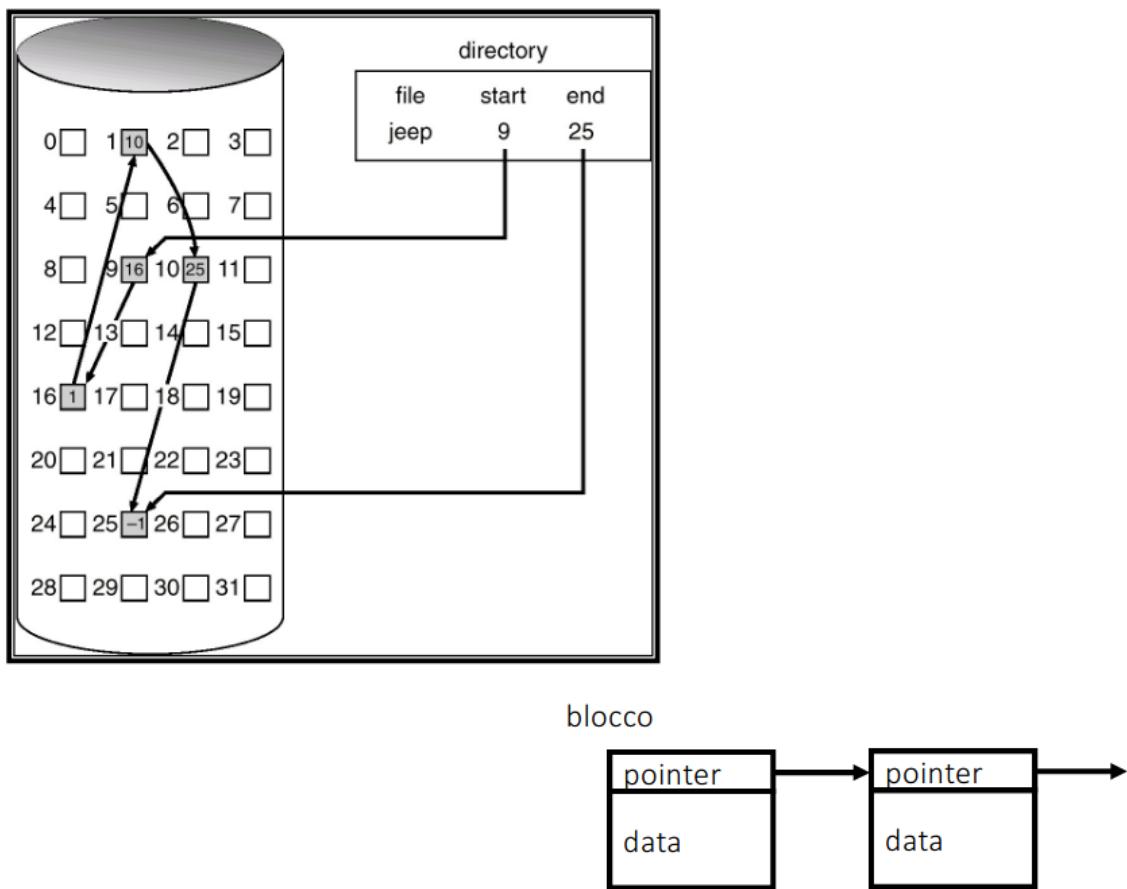
Allocazione a Lista Concatenata

In questo schema ogni file è visto come una lista di blocchi, che possono essere sparsi ovunque all'interno del disco:

- la directory contiene puntatori al primo e all'ultimo blocco
- ogni blocco contiene un puntatore al blocco successivo

A livello di indirizzamento funzionerà in questo modo:

- Sia X un indirizzo logico e N la dimensione di un blocco
- Avremo che $X / (N-1)$ è il numero del blocco nella lista e che $X \% (N-1)$ ci dà l'offset all'interno del blocco



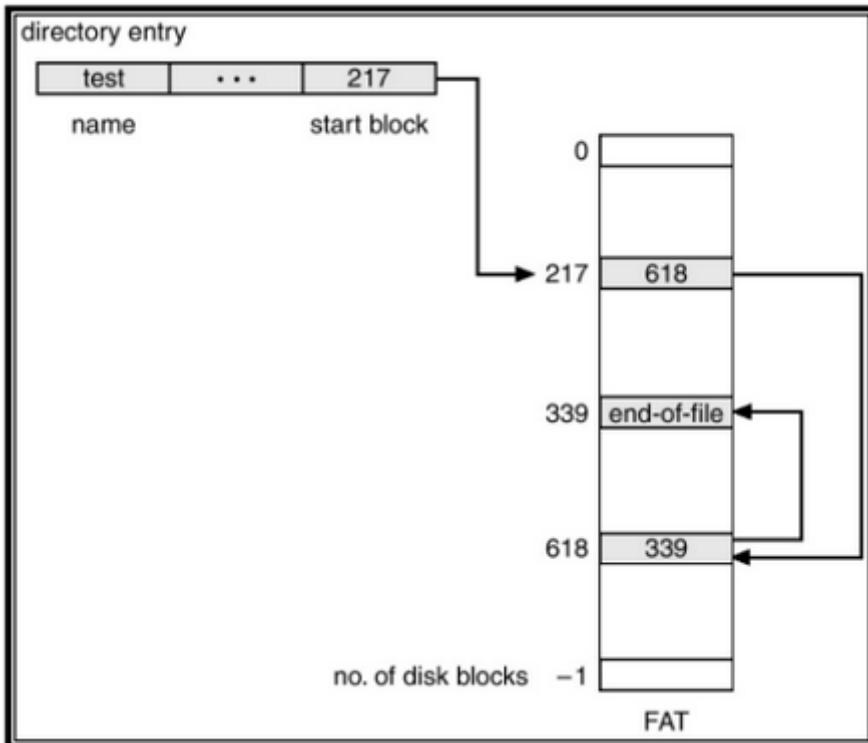
I *vantaggi* di questo approccio sono i seguenti:

- creazione di nuovi file semplice: basta cercare un blocco libero e creare una nuova entry nella directory che punta al blocco
- estensione del file semplice: basta cercare un blocco libero e concatenarlo alla fine del file
- nessuno spreco di spazio eccetto per il puntatore

Tale strategia comporta però alcuni *svantaggi*:

- impossibilità di garantire accesso casuale
- scarsa efficienza a causa di riposizionamenti sparsi
- scarsa affidabilità, dato che nel caso in cui si perda un puntatore si rischia di compromettere l'intero file. Alcune soluzioni che però comportano overhead a ciò sono
 - *liste doppiamente concatenate*
 - memorizzare nome file e n° di blocco in ogni blocco del file

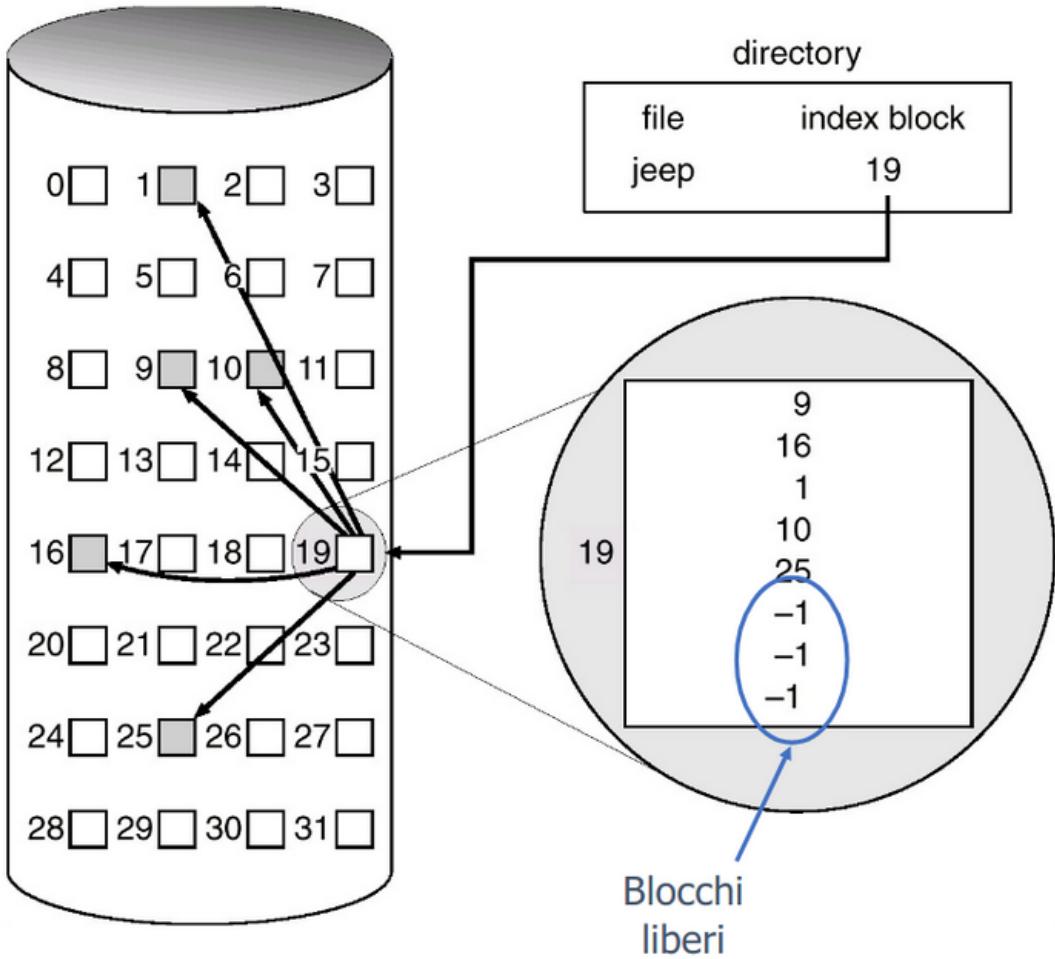
Un'implementazione di questo approccio che migliora le prestazioni della precedente è data dalla **file allocation table** tipicamente utilizzata in sistemi MS-DOS e OS/2



Abbiamo anche a disposizione una variante rispetto all'allocazione contigua, nella quale il concetto fondante è quello di **extent**, ovvero una serie di blocchi contigui su disco (lunghi tanto quanto la quantità di blocchi liberi contigui che si trovano). Il file system alloca degli extent invece una serie di singoli blocchi. Un file viene visto come una serie di extent che possono essere generalmente non contigui. Si tratta di un approccio adatto nell'allocazione a lista.

Allocazione Indicizzata

In questo tipo di schema ogni file ha un **index block** contenente la **index table** (tabella degli indirizzi fisici). La directory contiene l'indirizzo del blocco indice per ogni file



Questo schema permette di ottenere un accesso casuale efficiente. L'accesso è dinamico senza frammentazione esterna, esiste tuttavia dell'overhead dovuto al blocco indice per la index table che è maggiore di quello necessario in allocazione concatenata. Lo schema di indirizzamento in questo caso è il seguente:

- Dato X l'indirizzo logico e data N la dimensione di un blocco
- Avremo che X / N è l'offset nella index table e che $X \% N$ ci da l'offset all'interno del blocco dati

Questo schema comporta delle limitazioni, in primis il fatto che la **dimensione dell'index block va a limitare la dimensione dei file**. Nel caso in cui si vogliano utilizzare file di dimensioni senza limiti è necessario uno schema a *più livelli*:

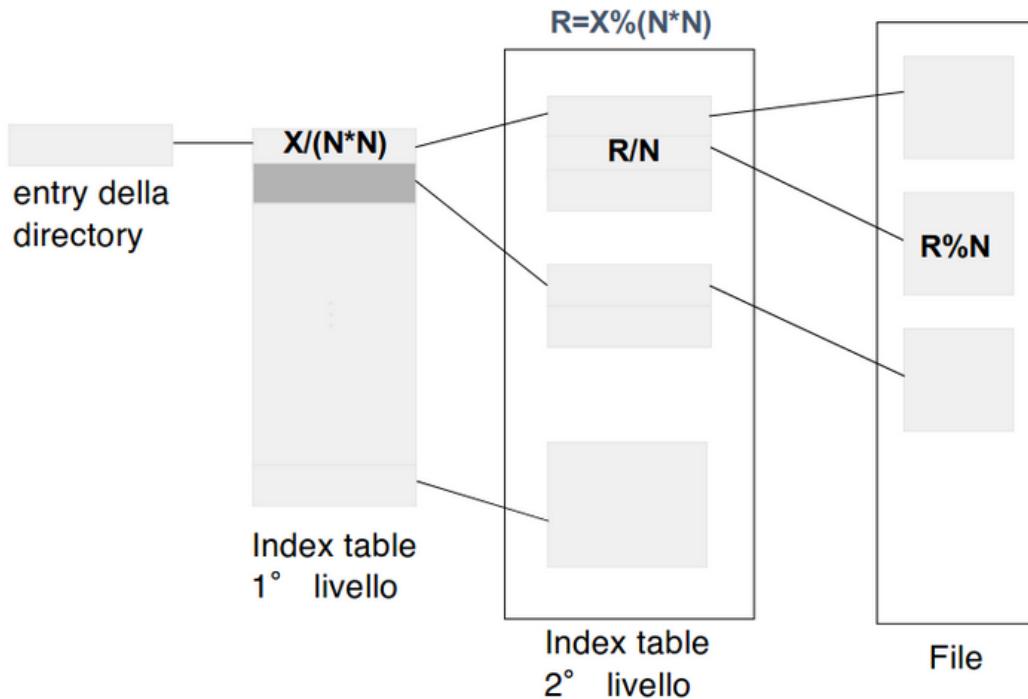
- **indici multilivello**
- **schema concatenato**
- **schema combinato**

Vediamo prima di tutto lo schema degli **indici multilivello** in cui abbiamo una tabella più esterna contenente puntatori alle index table:

- X = indirizzo logico
- N = dimensione del blocco
 - $X / (N * N)$ = blocco della index table di 1° livello

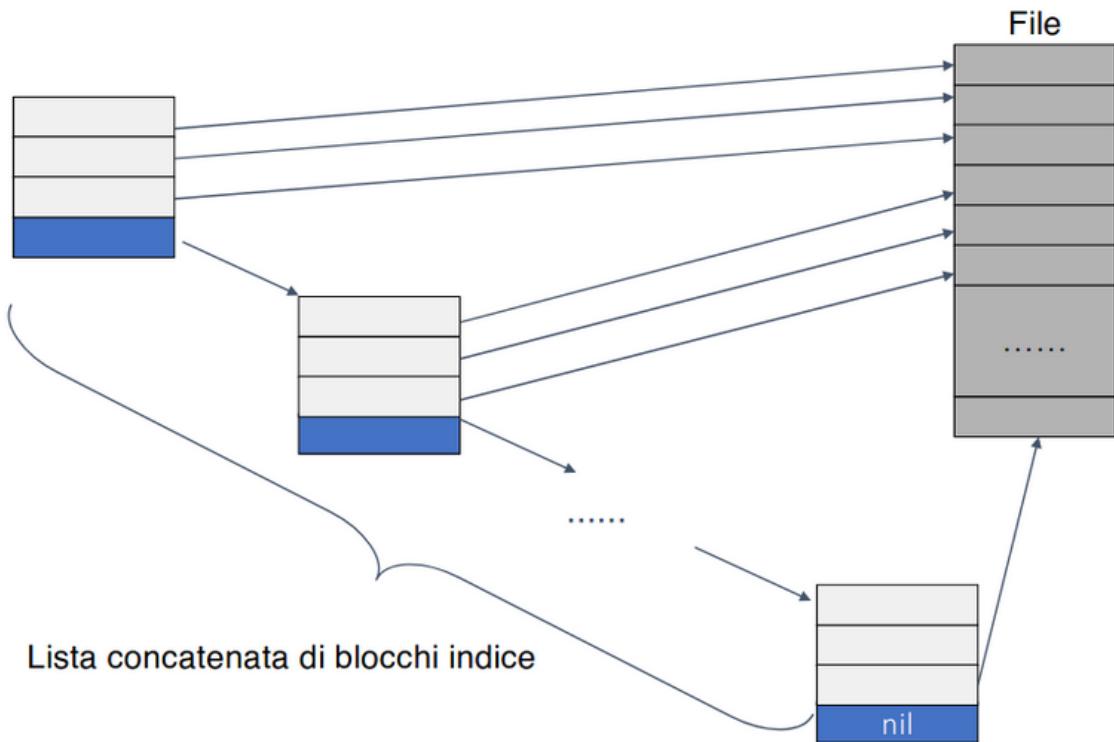
- $X \% (N \times N) = R$ che viene usato nel modo seguente
- R / N = offset nel blocco della index table di 2° livello
- $R \% N$ = offset nel blocco dati

In questo modo ad esempio con blocchi da 4KB abbiamo a disposizioni 1K indici da 4 byte ciascuno, due livello di indici consentono file da 4GB

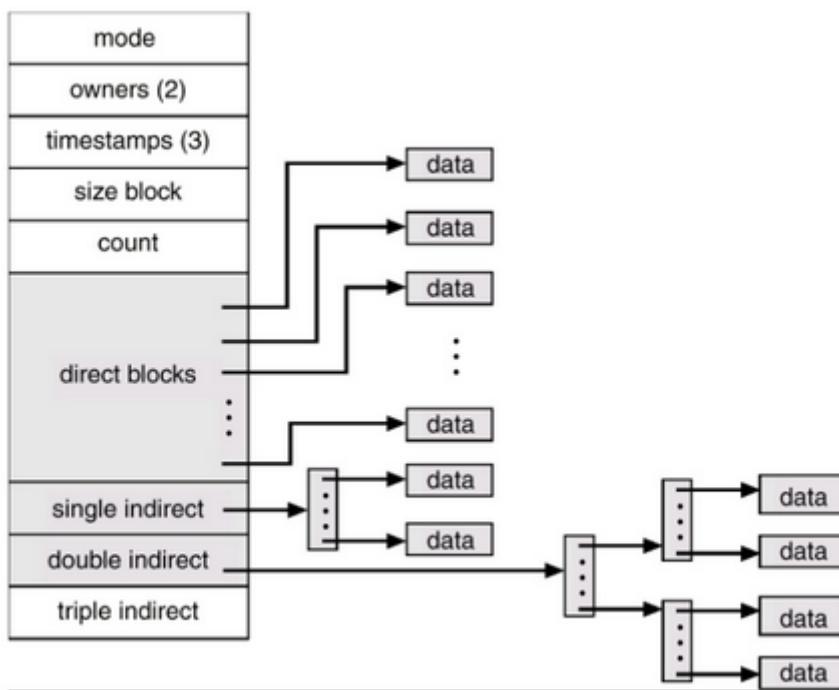


Andiamo ora ad approfondire lo **schema concatenato** in cui abbiamo una lista concatenata di blocchi indice e l'ultimo degli indici di un blocco indice punta a un'altro blocco indice:

- X = indirizzo logico
- N = dimensione del blocco in parole
 - $X / (N(N - 1))$ = numero del blocco indice all'interno della lista dei blocchi indice
 - $X \% (N(N - 1)) = R$ che viene usato come segue
- R / N = offset nel blocco indice
- $R \% N$ = offset nel blocco dati



Vediamo invece ora un esempio di **schema combinato** con index block da 4KB (UNIX):



Implementazione delle Directory

A livello implementativo il meccanismo utilizzato per definire una directory è lo stesso utilizzato nel caso dei file. Queste directory però non contengono dati, tipicamente contengono la lista dei file e delle directory che queste a loro volta contengono. Le problematiche principali sono le seguenti:

- **visualizzazione del contenuto**
- **accesso al contenuto**

A livello implementativo la prima tecnica per implementare una directory che viene in mente è quella di utilizzare una **lista lineare** che contiene i nomi dei file con puntatori ai blocchi di dati. Si tratta di una implementazione sicuramente molto semplice ma che risulta davvero poco efficiente dato che si rende necessario ricercare un file prima di poter eseguire una qualsiasi operazione. Tale ricerca nel caso in cui sia lineare va in $O(n)$, oppure è possibile effettuare una **ricerca binaria** che va in $O(\log(n))$ ma richiede che la lista sia ordinata con conseguenti costi di mantenimento.

Per questi motivi si preferisce una implementazione tramite **tavella hash** che garantisce un tempo di ricerca migliore. L'unico problema associato è il rischio di collisioni ovvero situazioni in cui due nomi di file collidono sulla stessa posizione.

Gestione dello Spazio Libero

Per tenere traccia dello spazio libero su disco si mantiene la lista dei blocchi liberi. A questo punto per creare un file si cercano blocchi liberi all'interno della lista e per rimuovere file si aggiungono i blocchi di tale file a questa lista. Per implementare questa lista abbiamo diverse alternative.

Vettore di Bit

In questa implementazione utilizziamo un bit per ogni blocco, nel caso in cui il `bit[i] = 1` abbiamo un blocco libero, in caso contrario tale blocco è occupato. Per il calcolo del primo blocco libero si ricerca la **prima parola diversa da 0**:

- (# di bit per parola) * (# di parole a 0) + (offset del primo bit a 1)
- Es: 000000000000000000001000010000011110000000000000

La mappa di bit richiede extra spazio, infatti se ad esempio la dimensione di un blocco è 2^{12} byte (4KB) e la dimensione del disco è di 2^{38} byte (256GB) abbiamo bisogno di 2^{26} bit per il solo vettore di bit. Si tratta di un approccio efficiente nel solo in caso in cui il vettore è mantenibili tutto in memoria.

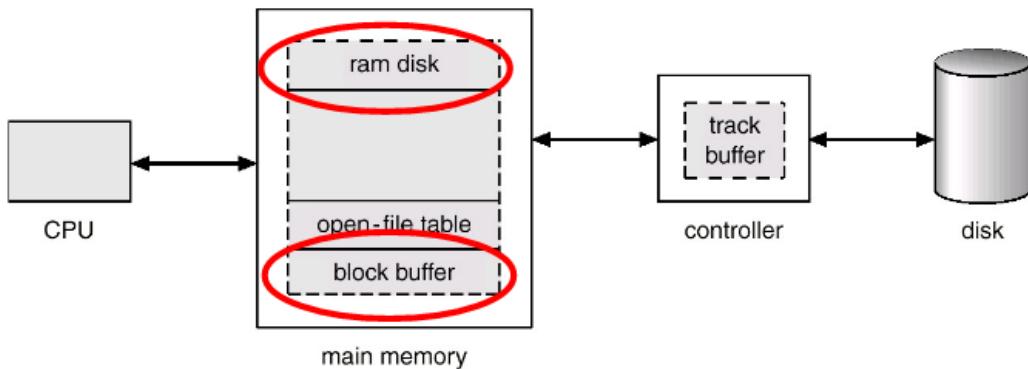
Abbiamo a disposizione alcune alternative a questo approccio quali ad esempio

- **Free List:** si tratta di una **lista concatenata di blocchi liberi**, comporta uno spreco minimo ma lo spazio contiguo non è in questo caso ottenibile
- **Raggruppamento:** modifica della free list in modo che il primo blocco libero sia costituito di indirizzi di n-1 blocchi libero, l'ultima entry del blocco è un puntatore all'indirizzo del primo blocco del gruppo successivo. Questa strategia permette di ottenere rapidamente un grande numero di free blocks
- **Conteggio:** viene mantenuto il conteggio di quanti blocchi liberi seguono in prima in una zona di blocchi liberi contigui. Generalmente la lista risulta più corta

Efficienza e Prestazioni

Il disco è sicuramente il **collo di bottiglia**, l'efficienza dipende principalmente dall'algoritmo di allocazione dello spazio su disco e dal tipo di dati contenuti nella directory.

A livello di **prestazioni** il controller del disco possiede una **cache** che contiene un'intera traccia ma non basta per garantire prestazioni elevate. Per questo motivo utilizziamo **dischi virtuali e cache del disco**



Dischi Virtuali

Si tratta di una parte di memoria che viene gestita come disco. Il **driver RAM disk** accetta operazioni standard dei dischi eseguendole in memoria ed è gestito dall'utente che scrive sul RAM disk invece che sul disco vero e proprio. Questo fornisce un supporto soltanto per i file temporanei. Si tratta di un meccanismo veloce

Cache del Disco

Una porzione di memoria è atta a memorizzare i blocchi usati più frequentemente, in modo analogo a ciò che fa la cache tra memoria e CPU. Si tratta di una componente gestita dal sistema operativo che sfrutta il principio di località spazio temporale. Il trasferimento di dati nella memoria del processo non richiede spostamento di byte. Si possono verificare problemi di consistenza tra disco e cache.

File System Log Structured

Registrano ogni cambiamento del file system come una transazione. Tutte le **transazioni** sono scritte su un **log**. Una transazione è considerata avvenuta quando viene scritta sul log. Queste transazioni sul log vengono scritte in modo asincrono nel file system. Quando il file system è modificato, la transazione viene cancellata dal log.

Se il sistema va in crash, le transazioni non avvenute sono quelle presenti sul log. Questo sistema garantisce un ottimizzazione del numero di seek

