Memoria Centrale

La **memoria centrale** è un componente essenziale di un moderno elaboratore, e consiste in un vettore di **parole** di memoria (o *byte*), ognuna delle quali ha un proprio indirizzo. Il processore caricherà nei propri registri istruzioni e dati presi direttamente da essa, in particolare da quella posizione indicata dal *program counter*. Il numero di accessi alla memoria durante la normale esecuzione di un processo può essere elevatissimo, dunque la sua gestione deve essere efficiente e rigorosa.
La memoria centrale vede solo flussi di indirizzi, ciò significa che ignora chi e come li generi e con quale scopo.

Collegamento degli indirizzi

I programmi risiedono generalmente su memorie secondarie sottoforma di file binari eseguibili, pronti ad essere attivati come processi. L'insieme dei processi in attesa di essere caricati in memoria centrale forma la cosiddetta **coda di entrata**, dalla quale ne verrà selezionato uno (o più) da caricare e mandare in esecuzione; a computazione terminata, le risorse allocate (quindi anche i bytes) torneranno ad essere disponibili per gli altri.
I processi non possono essere caricati a partire da un indirizzo qualunque, ma bisogna effettuare diversi passaggi (alcuni facoltativi) durante i quali dati/istruzioni e rispettivi indirizzi saranno collegati in modo diverso.

1. **Collegamento in fase di compilazione**, quando si conosce esattamente dove verrà caricato il processo già al momento della compilazione. Viene generato un **codice assoluto**, che sarà *statico*, *fisso* e *immutabile* (se la locazione di partenza cambia bisogna ricompilare il codice)
2. **Collegamento in fase di caricamento**. Se al momento della compilazione non si sa ancora dove risiederà il processo, allora il compilatore dovrà generare un **codice rilocabile**. In pratica il sistema operativo decide a partire da quale indirizzo caricare il processo in memoria, e quella locazione sarà l'indirizzo 0 relativo in base al quale verranno allocati tutti gli altri dati/istruzioni del processo. Il collegamento all'indirizzo assoluto in memoria centrale avviene in *fase di caricamento*
3. **Collegamento in fase di esecuzione**, caratteristico di quei processi che possono essere spostati da una zona all'altra anche durante l'esecuzione. Necessita di un hardware preposto a tale compito, ovvero un' **unità di gestione della memoria centrale** (*MMU*) configurata opportunamente, ed è il tipo di collegamento maggiormente usato nei vari sistemi operativi

Spazio di indirizzamento logico e fisico

L' **indirizzo fisico** è l'indirizzo nello spazio di memoria centrale che individua in modo univoco una parola in esso contenuta. Lo **spazio di indirizzamento fisico** è un vettore lineare che va da 0 all'ultimo spazio indirizzabile del chip di memoria.
L' **indirizzo logico** è invece un indirizzo astratto, consistente solo all'interno dello spazio di indirizzamento del processo, rappresentandone lo spiazzamento rispetto alla prima parola. Lo **spazio di indirizzamento logico** è dunque lo spazio indirizzabile dal nostro processo, e la sua prima parola corrisponde ad un certo indirizzo fisico detto*indirizzo di base*. Al processo non interessa se al di là dei confini del proprio spazio di indirizzamento siano caricate parole associate ad altri processi, tanto lui non potrà accedervi come gli altri non possono accedere al suo.
I metodi di collegamento degli indirizzi in fase di compilazione e caricamento generano indirizzi logici e fisici identici, cosa che invece non accade in fase di esecuzione. In quest'ultimo caso la responsabilità di trasformarli è demandata a un dispositivo ausiliario hardware che abbiamo già nominato, ovvero la **MMU**. Il sistema di trasformazione semplice consiste nell'utilizzo del **registro di rilocazione**, il cui valore viene aggiunto ad ogni indirizzo generato dal processo nel momento in cui viene inviato in memoria. La formula generale è:indirizzo fisico = indirizzo logico + offset, dove l'offset è lo spiazzamento tra lo 0 logico del processo e la sua mappatura effettiva in memoria centrale. Grazie a questo mappaggio il processo non vede mai gli indirizzi fisici reali, ma tratta solo quelli logici che sono molti meno del totale quindi più semplici e veloci da manipolare.
Ricapitolando: gli indirizzi logici vanno da 0 a un valore massimo, mentre i corrispondenti indirizzi fisici vanno da R+0a R+valore massimo (dove R è il valore del registro di rilocazione). Nella fase di **linking** dei programmi, viene eseguito il calcolo degli indirizzi logici a partire da quelli simbolici generati dalle chiamate di sistema invocate. Nella fase di **binding** si passa invece dall'indirizzo logico a quello fisico; può avvenire durante la compilazione, o nel momento del loading del programma, o anche durante l'esecuzione (soprattutto se ho librerie dinamiche).

Caricamento dinamico

Dato che per essere eseguito un programma deve risiedere in memoria centrale, come fare quando le sue dimensioni superano quelle della memoria stessa? La soluzione è il **caricamento dinamico**, che prevede inizialmente il caricamento di un insieme di procedure fondamentali e il mantenimento delle altre su disco in formato di caricamento

rilocabile; nel caso in cui queste ultime si rivelino necessarie, verrà richiesto a un *loader* di caricarle e di aggiornare la tabella degli indirizzi del programma. Il vantaggio di tale sistema è che non si spreca inutilmente spazio di memoria per procedure che vengono utilizzate raramente (come quelle di gestione degli errori). Il sistema operativo non gestisce direttamente il caricamento dinamico, la sua reale implementazione è lasciata ai programmatori.

Collegamento dinamico e librerie condivise

Utilizzare collegamenti dinamici piuttosto che statici alle librerie di sistema consente di risparmiare spazio sia su memorie di massa che in quella centrale. Il motivo è semplice: un collegamento statico considererebbe le librerie come un qualsiasi altro modulo da caricare, quindi ogni programma dovrebbe incorporarne una copia nel file eseguibile e nel suo spazio di indirizzamento. Con i **collegamenti dinamici** (attuati in fase di esecuzione) viene invece inclusa nell'eseguibile un' *immagine* (*stub*) che indica come individuare la procedura di libreria desiderata già residente in memoria, o come reperirla dal disco se non è presente. In questo modo tutti i processi che usano una stessa libreria eseguiranno una sola copia del codice.

Il sistema a **librerie condivise** estende i vantaggi del collegamento dinamico, consentendo inoltre di aggiornare le versioni delle librerie e di fare in modo che ogni programma utilizzi quella a sé compatibile.

Al contrario del caricamento dinamico, il collegamento dinamico può richiedere l'intervento del sistema operativo, ad esempio in quei casi in cui un processo avrebbe bisogno di una procedura contenuta nello spazio di indirizzamento di un altro.

Overlay

Se il caricamento dinamico offriva una soluzione all'esecuzione dei processi con dimensioni maggiori della memoria centrale, la tecnica dell' **overlay** consente a un processo di essere più grande della quantità di memoria a esso allocata, ovvero al suo spazio di indirizzamento. Il principio è simile: mantenere in memoria solo le istruzioni e i dati che sono necessari in un certo momento, rimpiazzando man mano quelle più vecchie. Si può dire che un overlay è una sorta di partizione del programma di partenza, indipendente dagli altri e quindi in grado di compiere autonomamente tutta una serie di operazioni che lo caratterizzano. Quando la sua esecuzione parziale del programma si esaurisce, viene deallocato e al suo posto ne viene caricato un altro.

Gli overlay non richiedono alcun supporto speciale da parte del sistema operativo, è compito del programmatore definirne il numero e le suddivisioni. Questo si rivela però un compito improbo, dato che i programmi in questione sono generalmente molto grandi (per quelli piccoli questa tecnica non è evidentemente necessaria) e suddividerli in parti implica una conoscenza puntigliosa della loro struttura. Inoltre i moduli possono avere dimensioni diverse, con conseguenti sprechi di spazio quando se ne caricheranno alcuni di grandezza più ridotta. Troppe responsabilità al programmatore e sfruttamento della memoria centrale poco efficiente fanno così dell'overlay una tecnica utilizzata solo per sistemi con poca memoria fisica e supporto hardware poco avanzato.

Swapping

In un sistema operativo multitasking è naturale che tra i processi caricati in memoria centrale alcuni non stiano facendo niente, o perché sono in attesa o perché hanno bassa priorità. Ciò però comporta uno spreco di spazio, cui si può far fronte adottando la tecnica dello **swapping**. In breve essa consiste nello scambiare temporaneamente un processo in attesa con un altro in stato di pronto o running che si trova in una *memoria temporanea* (*backing storage*), tipicamente un disco veloce. Nello specifico avremo quattro fasi:

- *identificare* i processi in stato di attesa
- *salvare sulla memoria temporanea* i loro dati sensibili (dati globali, heap, stack)
- rimuovere dalla memoria centrale tali processi (*scaricamento*)
- caricare nello spazio appena liberato quei processi in stato di pronto o running che si trovavano nella memoria temporanea (*caricamento*).

Una variante di questa tecnica è il **roll out/roll in**, che individua i processi da caricare e scaricare in base alla loro priorità.

Lo swapping si applica sugli interi processi ed impone almeno due vincoli. Il primo è che si possono spostare solo quei processi in stato di riposo; il secondo è che non si possono spostare quei processi che hanno invocato una chiamata di funzione e sono ancora in attesa di una risposta, o il processo che gli subentrerebbe la riceverebbe al posto suo. Possibili soluzioni sono impedire lo swap di processi in attesa di I/O oppure permetterlo solo a quelli che usano i buffer condivisi del sistema operativo.

Lo swapping standard ha come vantaggio l'aumento del grado di multiprogrammazione del sistema, ma essendo gestito automaticamente dal sistema operativo ed essendo applicato agli interi processi comporta un netto

rallentamento delle prestazioni. Una versione modificata viene invece utilizzata in molti sistemi operativi UNIX, in cui ad esempio viene attivato solo quando esaurisce lo spazio in memoria centrale e solo fintanto non se ne libera un po'.

Allocazione contigua di memoria

L' *allocazione contigua di memoria* è un metodo per allocare nel modo più efficiente possibile sia il sistema operativo che i processi degli utenti in memoria centrale. Il primo viene generalmente memorizzato nella parte bassa della memoria centrale, separato dalle sezioni riservate ai processi da un *vettore di interrupt*.

Protezione della memoria centrale

Proteggere la memoria centrale significa garantire che non avvengano al suo interno *accessi illegali* da parte dei processi, dove per accessi illegali si intendono quelli al di fuori del proprio spazio di indirizzamento (che sconfinano quindi in quello del sistema operativo o di altri processi). La protezione avviene attraverso l'utilizzo congiunto di due registri che fanno parte del contesto di ogni processo:

- il **registro di rilocazione**, che contiene il valore del più piccolo indirizzo fisico
- il **registro limite** che contiene l'intervallo degli indirizzi logici, quindi la dimensione in byte dello spazio di indirizzamento

Quando lo schedulatore seleziona un processo da mandare in esecuzione, la *Memory Management Unit* si occuperà di verificare che ogni indirizzo logico sia inferiore del registro limite associato, dopodiché lo mapperà dinamicamente aggiungendogli il valore del registro di rilocazione. L'indirizzo così ottenuto è inviato in memoria centrale.
Il registro di rilocazione fa inoltre in modo che le dimensioni del sistema operativo cambino dinamicamente, rendendo possibile l'utilizzo o meno di parte del codice (detto *transiente*).

Allocazione della memoria centrale

Uno dei metodi più semplici per allocare memoria centrale consiste nel suddividerla in *partizioni*, ciascuna delle quali può contenere al più un processo. Dal loro numero dipende il grado di multiprogrammazione del sistema. In particolare, con il *metodo delle partizioni multiple* quando un processo termina la sua computazione viene sostituito con uno selezionato dalla coda dei processi pronti.
Abbiamo due schemi di partizionamento:

- **schema a partizione fissa**, in cui si hanno partizioni di dimensione *statica* (definite al bootstrap) e una tabella aggiornata che indica quali parti della memoria centrale sono disponibili e quali occupati (inizialmente sono tutti liberi). Man mano che i processi si attivano, vengono messi in una coda di entrata gestita con un qualsiasi algoritmo di schedulazione; il sistema operativo valuterà le loro richieste di memoria e cercherà di allocarli in blocchi abbastanza grandi da ospitarli. Se tale ricerca fallisce, il processo entra in stato di attesa e vi rimarrà finché un altro non avrà terminato la sua computazione e quindi reso disponibile la partizione che occupava
- **schema a partizione variabile**, che a differenza dello schema precedente consente di modificare l'indirizzo di base delle varie partizioni rendendo di fatto possibile variarne le dimensioni, ad esempio unendo blocchi contigui precedentemente distinti o suddividendone uno particolarmente sovradimensionato.

Nello schema a partizione variabile la configurazione dello spazio disponibile continua a variare, dunque il sistema dovrà controllare spesso se la situazione è diventata favorevole per il caricamento di uno dei processi in attesa nella coda di entrata. Questo viene anche chiamato *problema dell'allocazione dinamica della memoria centrale*, per il quale sono percorribili tre strategie:

- ***first-fit***, che assegna il primo blocco libero abbastanza grande per contenere lo spazio richiesto
- ***best-fit***, che assegna il più piccolo blocco libero che lo può contenere
- ***worst-fit***, che assegna il più grande blocco libero che lo può contenere

Le simulazioni hanno dimostrato che le strategie migliori sono le prime due, in particolar modo la *first-fit* che è la più veloce.

Frammentazione

Il problema del *first-fit* e del *best-fit* è che soffrono di **frammentazione esterna**, ovvero lasciano dei blocchi liberi fra quelli occupati, blocchi che se fossero uniti e contigui potrebbero ospitare un altro processo. Questo accumularsi di spazio non sfruttato alla lunga comporta un considerevole abbassamento delle prestazioni, cosa particolarmente grave se si pensa che nel caso peggiore potrebbe soffrire di frammentazione esterna ogni coppia di blocchi. Il problema non è evitabile: la *regola del 50 per cento* dice che su N blocchi allocati con la first fit ne andranno persi N/2 a causa della frammentazione. Bisogna dunque fare in modo di ottimizzare la situazione a posteriori.

Una prima soluzione è la *compattazione*, ovvero la fusione di tutti i blocchi liberi in uno solo. Tuttavia questa tecnica può essere applicata solo se la rilocazione è dinamica ed è fatta al momento dell'esecuzione, dato che in tal caso basterebbe spostare programma e dati. Si tratta inoltre di un procedimento piuttosto costoso.
Altra contromisura è permettere allo spazio di indirizzo logico di un processo di essere non contiguo, un'idea che verrà usata con profitto dalle tecniche di *paginazione* e *segmentazione* che vedremo poi.
Se abbiamo la *frammentazione esterna* non possiamo certo farci mancare anche la **frammentazione interna**, che si presenta quando carichiamo all'interno di un blocco un processo più piccolo della sua dimensione. Lo scarto di spazio tra la dimensioni del processo e del blocco rappresenta la frammentazione.

Paginazione

La **paginazione** è uno schema di gestione della memoria centrale che, nel tentativo di superare gran parte dei problemi di gestione visti finora, si prefigge i seguenti obiettivi:

- caricare e scaricare solo piccole porzioni di memoria, evitando così l'overhead dello swapping
- mantenere in memoria solo le parti che servono
- non sprecare spazio, per quanto possibile
- poter utilizzare porzioni di memoria non contigue per lo stesso programma
- essere indipendentemente dal programmatore, ma gestito dall'hardware e in modo integrato col sistema operativo

E' usata nelle sue varie forme nella maggior parte dei sistemi operativi, ed è quasi esclusivamente gestita dall'hardware anche se negli ultimi processori a 64 bit è sempre più integrata con il sistema operativo stesso.

Metodo base

Il metodo base per implementare la paginazione è suddividere la memoria fisica (centrale e ausiliaria) in blocchi di dimensione fissa chiamati **frame** (o pagine fisiche), e la memoria logica in blocchi di uguale dimensione detti **pagine** (o pagine logiche).
Gli indirizzi generati dalla CPU sono divisi in due parti, ovvero il **numero di pagina** (*p*) e lo **spiazzamento nella pagina** o offset (*d*). Il numero di pagina viene usato come indice nella **tabella delle pagine**, che contiene gli indirizzi iniziali di tutti i frame presenti nella memoria fisica; la combinazione di tali indirizzi base con lo spiazzamento permette di ottenere l'indirizzo fisico associato. Dato poi che è il sistema operativo a gestire la memoria centrale, dovrà sapere esattamente quali blocchi sono liberi e quali in uso; è stata a tal fine introdotta la **tabella dei frame**, che per ogni frame sa dire se è libero o occupato, e in quest'ultimo caso anche da chi. Riassumendo:

- **indirizzo logico** = (**p**, **d**) dove *p* è il numero di pagina logica e *d* è lo spiazzamento
- **indirizzo fisico** = (**f**, **d**) dove *f* è il numero di pagina fisica
- **tabellaPagine[*PaginaLogica*]** = **PaginaFisica** se è caricata, niente altrimenti

La dimensione dei blocchi è decisa dall'hardware ed è generalmente una potenza di 2 per semplicità rappresentative.
Con la paginazione non si ha frammentazione esterna perché le dimensioni di pagine e frame coincidono e possono essere assegnate anche in modo non contiguo. Perdura invece il problema della frammentazione interna, dato che non è sempre vero (anzi, lo è di rado) che un processo riempia interamente tutti i suoi frame. Riducendo le dimensioni dei frame l'entità della frammentazione interna diminuisce, ma allo stesso tempo si riducono le prestazioni e l'efficacia di operazioni importanti quali i trasferimenti.
Notare la netta separazione tra la visione che ha l'utente della memoria centrale con la sua realtà implementativa: l'utente vede la memoria come un unico singolo spazio contiguo in cui è caricato tutto e solo il suo programma che invece si trova sparpagliato nella memoria fisica in mezzo ad altri. Il sistema operativo si prende l'onere di rimappare gli indirizzi fisici in modo da garantire al tempo stesso il corretto funzionamento del processo e l'impedimento di accessi non autorizzati.
Come supporto per la gestione della memoria fisica il sistema operativo mantiene per ogni processo una copia della tabella delle pagine, che indica per ogni pagina se questa è allocata e in quest'ultimo caso in quale frame.
Ultima nota: il mantenimento per ogni processo di una copia della tabella delle pagine inciderà sui tempi di cambiamento di contesto.

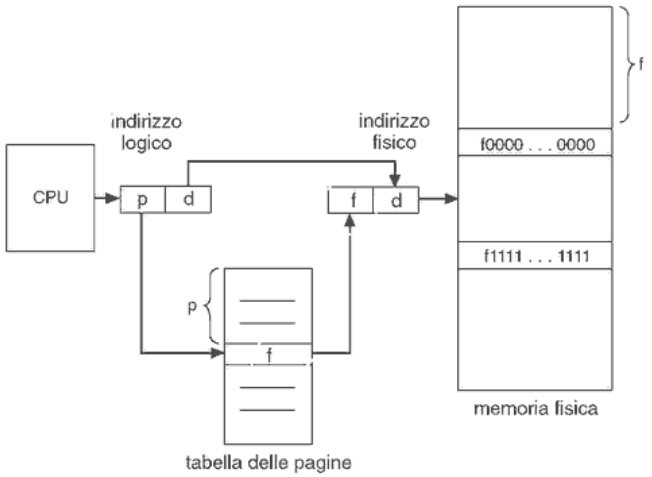
Supporto hardware

Ogni sistema operativo ha il suo sistema di memorizzazione della tabella delle pagine, la maggior parte dei quali ne alloca una per ogni processo. Nel caso più semplice, ovvero quando sono pochi i processi da gestire, vengono utilizzati con profitto una serie di *registri* ad alta velocità dedicati.
Nei computer odierni però la tabella delle pagine è estremamente grande (fino a un milione di elementi), quindi ai

registri si preferisce un'altra soluzione: il mantenimento della tabella in memoria centrale con l'utilizzo di un *registro base* che la referenzi, così che sia sufficiente modificare quest'ultimo per puntare a un'altra. L'aspetto negativo di questo sistema è che per ogni accesso a un byte se ne deve prima effettuare un altro in tabella, dunque avrei un rallentamento di fattore 2. Per questo motivo viene utilizzata una piccola cache hardware per l'indicizzazione veloce, la **memoria associativa** (*Translation Look-Aside Buffer*, **TLB**). Ogni elemento contenuto in essa è costituito di due parti: una *chiave* (tag) e un valore. Il meccanismo è il seguente:

1. la CPU genera un indirizzo logico il cui numero di pagina viene presentato alla TLB
2. se il numero di pagina viene trovato significa che il corrispondente numero di frame è immediatamente disponibile e viene rilasciato per l'accesso alla memoria centrale
3. se il numero di pagina non è presente (*TLB-miss*) la TLB deve procurarselo insieme al numero di frame facendo riferimento alla memoria centrale. Il nuovo elemento della TLB può essere aggiunto alla tabella e se non c'è abbastanza spazio si possono rimpiazzare gli elementi esistenti con logiche *LRU* ("quello usato meno di recente viene rimpiazzato") o più semplicemente a caso

Vediamo uno schema:



La frequenza con cui un particolare numero di pagina viene richiesto nella TLB è il **tasso di accesso con successo**(*hit-ratio*).

Alcune TLB memorizzano anche gli **identificatori dello spazio degli indirizzi** (*ASID*) che identificano appunto in modo univoco a quale processo è associata una determinata coppia chiave-valore; in questo modo la TLB può contenere contemporaneamente gli indirizzi di diversi processi. Se invece gli ASID non sono supportati, la TLB andrebbe flushata ad ogni cambio di contesto o si rischierebbe di ricevere come risposta dei numeri di frame che appartengono allo spazio di indirizzamento di altri processi.

Protezione

La protezione della memoria centrale viene garantita esplicitamente con **bit di protezione** associati a ogni frame. Questi possono indicare se l'accesso al frame è consentito in sola lettura (es. costanti), o in lettura-scrittura (es. dati), o in sola esecuzione (es. codice) o ancora se è dentro o fuori lo spazio di indirizzamento del processo. Quest'ultimo bit è detto di *validità/non validità* e protegge dunque la memoria centrale da accessi illegali.

Alcuni sistemi hanno invece un hardware detto **registro della lunghezza della pagina** preposto alla protezione degli spazi di indirizzamento, che indica la dimensione della tabella delle pagine.

Tutti i tentativi illegali di accesso sollevano in genere una *trap* del sistema operativo, che la gestirà in modo opportuno.

Struttura della tabella delle pagine

Paginazione gerarchica

Dato che i sistemi operativi moderni hanno un vasto spazio di indirizzamento logico, la tabella delle pagine diventa eccessivamente grande, dunque è preferibile suddividerla in parti più piccole. Questo sistema è detto delle **paginazioni gerarchiche**.

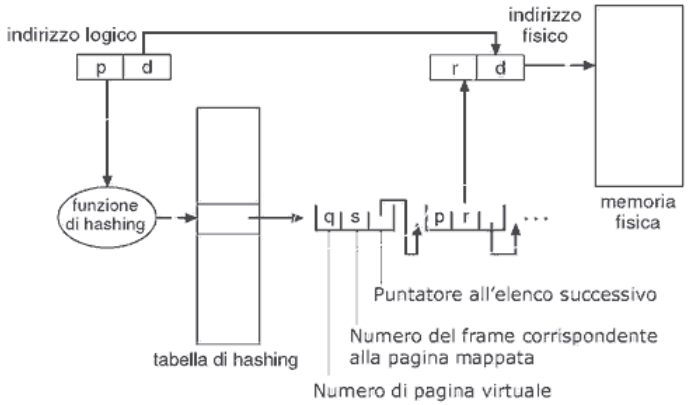
Ci sono diversi modi per realizzarla, il primo dei quali potrebbe essere l'utilizzo di una *paginazione a due livelli*, in cui la tabella stessa è paginata. Avremo dunque un indirizzo logico con tre valori: p_1 (indice nella tabella esterna), p_2(spostamento nella pagina delle tabelle a partire da p_1), d (spiazzamento della pagina). Questo schema è detto anche della **tabella delle pagine mappate in avanti**.

Una variante è il **VAX**, che divide lo spazio di indirizzamento logico in quattro sezioni uguali, individuate dai 2 bit più significativi.
La tabella esterna delle pagine può essere a sua volta paginata, arrivando a 3 livelli (accade ad esempio nello SPARC); addirittura alcuni Motorola arrivano a 4.
Per le architetture a 64bit le tabelle di paginazione gerarchiche sono invece inadeguate perché richiederebbero un numero proibitivo di accessi ai vari livelli di paginazione per tradurre con profitto l'indirizzo logico.

Tabella delle pagine con hashing

Nella tecnica della **tabella delle pagine con hashing** ogni elemento della tabella è composto da tre campi: *q* (numero della pagina virtuale), *s* (numero del frame corrispondente), puntatore all'elemento successivo.
La pagina virtuale (*p*) dell'indirizzo logico generato dalla CPU è usata come argomento di una funzione di hash, che permette di ottenere una *tabella di hashing* più piccola di quella di partenza. L'algoritmo funziona così:

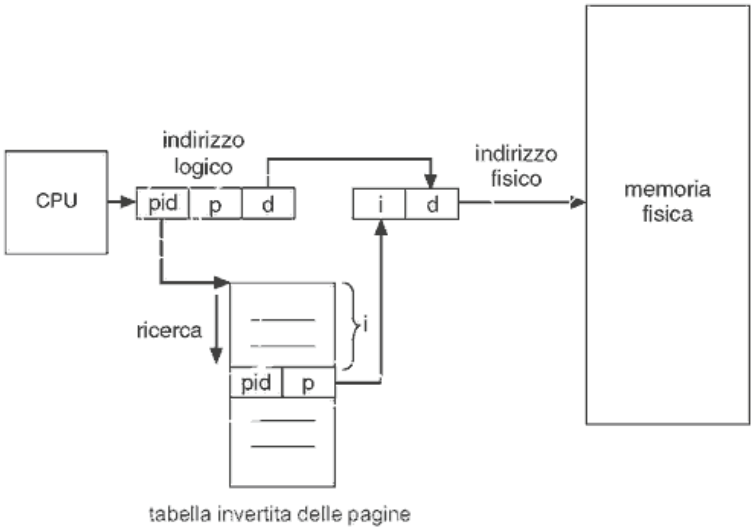
1. si applica la funzione di hash al numero della pagina contenuto nell'indirizzo virtuale, identificando un elemento nella tabella di hashing
2. si confronta il numero di pagina virtuale con l'elemento contenuto nella *q* del primo elemento della lista concatenata corrispondente
    - se i valori coincidono, si usa il campo *s* per generare l'indirizzo fisico
    - altrimenti si esamina l'elemento successivo della lista



Una variante è la **tabella delle pagine a gruppi**, particolarmente adatta nei sistemi a 64bit, in cui ogni elemento della tabella di hashing si riferisce a più pagine (di solito 16).

Tabella delle pagine invertita

Finora abbiamo visto che le tabelle delle pagine associate a ciascun processo restituivano il numero di frame corrispondente al numero di pagina indicato nell'indirizzo logico. Dato però che ognuna di esse può contenere milioni di elementi e che ci possono essere molti processi attivi simultaneamente si avrà un'enorme occupazione di spazio.
Una soluzione che ribalta la prospettiva appena illustrata è quella della **tabella delle pagine invertita**, in cui si mantiene un'unica tabella che avrà un elemento per ogni pagina fisica della memoria centrale. Ognuno di questi elementi è composto da un *ASID* (necessario perché la tabella contiene parecchi spazi di indirizzamento appartenenti a processi differenti), il numero di pagina logica memorizzata in quel frame e lo spiazzamento.
Questa tecnica permette di diminuire la quantità di memoria centrale necessaria per memorizzare ogni tabella, garantendo sempre la legalità degli accessi. Per contro ha però l'aumento del tempo necessario per cercare nella tabella quando si hanno riferimenti alla pagina, dal momento che nel caso peggiore bisogna scandirla tutta. A tal proposito si può utilizzare una **tabella invertita con hashing**, che velocizza la ricerca ma richiede due accessi.
Insomma, le soluzioni ci sono, va solo cercato il giusto equilibrio.



Pagine condivise

La paginazione consente con relativa facilità la condivisione di alcune pagine fisiche tra diversi processi, in modo da risparmiare una considerevole quantità di spazio. Ad esempio imporre una protezione in sola esecuzione su alcune pagine realizza di fatto la condivisione di tali porzioni di memoria seppur virtualmente disgiunte.

I sistemi che usano tabelle delle pagine invertite hanno però maggiori difficoltà implementative: se infatti in quelle standard è possibile fare in modo che a diversi numeri di pagina siano associati gli stessi frame, quelle invertite assegnano per definizione un'unica pagina a un dato frame. Vanno trovate soluzioni alternative.

Segmentazione

Metodo base

Un limite della paginazione è che non consente di tipizzare le varie porzioni di spazio di indirizzamento logico, distinguendo ad esempio l'area del codice da quella dei dati, il che diventa un problema quando si vogliono fare controlli mirati su parti specifiche di programmi di grosse dimensioni. Per lo stesso motivo non si possono nemmeno effettuare condivisioni semplici ed efficienti di porzioni di memoria omogenei per scopo e tipo, ad esempio tutta la parte di codice.

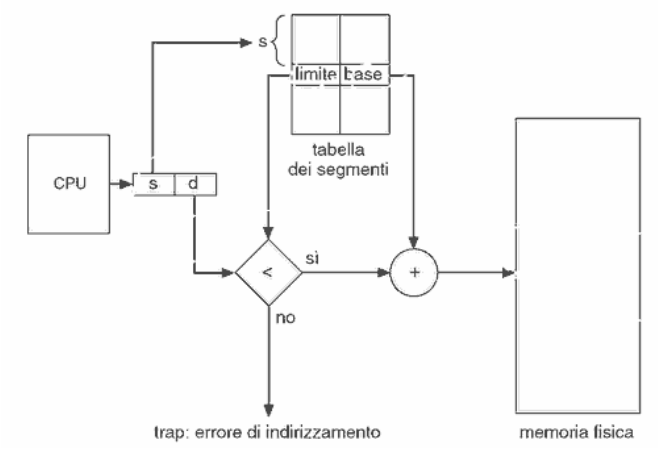
La **segmentazione** è uno schema di gestione della memoria centrale che mantiene gran parte degli obiettivi della paginazione, compresa la separazione tra memoria logica e fisica, ma suddivide quest'ultima in segmenti di dimensione variabile ognuno con un proprio nome che lo individua (spesso un numero per semplicità implementativa). Il suo nuovo obiettivo sarà dunque dare una consistenza logica alle porzioni di spazio di indirizzamento dei processi, supportandone la tipizzazione e una più efficace condivisione.

Hardware

Nella segmentazione la memoria centrale fisica è divisa in **segmenti fisici** (*frame*) di dimensioni diverse, mentre lo spazio di indirizzamento del processo è diviso in **segmenti logici** (*segmenti*). Notare che un segmento è tipizzato ed è caricato in un frame di medesima dimensione.

In modo analogo alla paginazione, la traduzione degli indirizzi logici avviene attraverso una **tabella dei segmenti** in cui ogni elemento ha una *base di segmento* (indirizzo fisico di partenza) e un *limite del segmento* (la sua lunghezza). I segmenti di un processo possono essere caricati in frame non contigui in memoria centrale fisica, mentre quelli non caricati sono conservati nell'area di swap.

L'indirizzo logico è costituito dal numero di segmento (*s*) e dallo spiazzamento (*d*). Ogni indirizzo prodotto dalla CPU viene controllato utilizzando la tabella dei segmenti, accertandosi che rientri nei valori legali ad esso consentiti. Ad esempio se lo spiazzamento è maggiore del valore del limite, allora verrà impedito l'accesso.



L'hardware dedicato per il supporto alla segmentazione è anche in questo caso la **MMU**, stavolta orientata alla gestione dei segmenti.

Protezione e condivisione

Il fatto stesso di suddividere la memoria centrale (e dunque di riflesso anche i processi che la occupano) in blocchi semanticamente omogenei, i segmenti appunto, facilita e velocizza le operazioni di protezione e condivisione dato che è plausibile che elementi facenti parte dello stesso segmento vadano trattati nello stesso modo. Le tecniche sono quelle già viste per la paginazione.

Frammentazione

La frammentazione esterna è un problema per la segmentazione dal momento che i segmenti sono di dimensione variabile e quindi non sempre coincidono con le pagine di dimensione fissa della memoria fisica. Dato però che per sua stessa natura la segmentazione avviene con una rilocazione dinamica, è possibile adottare sistemi di compattazione che riducono il problema. Se in aggiunta vengono effettuate valutazioni ben ponderate da parte dello schedulatore a lungo termine, si può mantenere la frammentazione esterna sotto controllo.

Segmentazione con paginazione

Abbiamo visto che paginazione e segmentazione presentano vantaggi e svantaggi, ma con la loro combinazione è possibile migliorarle a vicenda prendendo il meglio di ognuna:

- dalla paginazione si prende l'identificazione dei frame liberi, la scelta del frame libero in cui caricare una pagina, nessuna frammentazione
- dalla segmentazione si prende la verifica degli accessi e delle operazioni, la condivisione di porzioni di memoria

La memoria centrale fisica è divisa in **pagine fisiche** (*frame*) di dimensione fissa, mentre lo spazio di indirizzamento del processo è suddiviso in **segmenti logici** (*segmenti*) di dimensioni diverse ciascuno suddiviso in *pagine logiche*. I segmenti contengono informazioni di tipo diverso (sono quindi tipizzati), mentre le pagine di cui sono costituiti sono porzioni indifferenziate del loro spazio di indirizzamento. Notare che i frame hanno la stessa dimensione delle pagine logiche, che infatti vengono caricate in essi.

L'indirizzo logico è dunque costituito dalla tripla: *s* (numero di segmento), *p* (numero di pagina), *d* (spiazzamento della pagina).

L'indirizzo fisico invece è rappresentato come: *f* (numero di frame), *d* (spiazzamento nel frame).

E' il sistema operativo che gestisce tutto automaticamente, avvalendosi della **MMU** opportunamente configurata come supporto hardware dedicato per la traduzione dell'indirizzo logico in quello fisico.

La Memoria Virtuale

La **memoria virtuale** è una tecnica che consente l'esecuzione dei processi che non sono completamente in memoria, astraendo la memoria centrale in un vettore di memorizzazione molto più grande e uniforme. Essa consente inoltre di condividere facilmente porzioni di memoria e semplifica la creazione dei processi. Per contro è piuttosto difficile da implementare e può rallentare il sistema se progettata e adoperata con poca attenzione.

Ambiente

Le istruzioni dei processi per poter essere eseguite devono risiedere in memoria centrale, con le limitazioni sulle dimensioni che ne conseguono. Spesso è però inutile caricare tutto il codice, o perché contiene procedure che si

usano raramente (ad esempio gestione degli errori), o perché si alloca più memoria del necessario per array o liste, o ancora perché è difficile che diverse porzioni del programma servano contemporaneamente. L'esecuzione di un programma caricato solo parzialmente in memoria ha invece diversi vantaggi, dato che non è più vincolato alla dimensione della memoria centrale e permette il caricamento di più processi aumentando così il livello di multiprogrammazione (dunque lo sfruttamento della CPU).
La memoria virtuale comporta la netta separazione tra memoria logica e fisica. Il programmatore non dovrà più preoccuparsi della quantità di memoria fisica disponibile o di quale codice dovrà collegare nell'overlay, ma potrà concentrarsi sul programma con soddisfazione di tutti. Lo *spazio di indirizzamento virtuale* è in sostanza quello logico di cui si parlava nel capitolo precedente, dunque deve essere mappato sulla memoria fisica dalla **MMU**. La mappatura può essere contigua o più spesso sparsa, quest'ultima con importanti benefici come una maggiore semplicità di gestione di heap e stack, e la condivisione di librerie o altri oggetti.

Richiesta di paginazione

La memoria virtuale viene implementata in genere con la **richiesta di paginazione** (per la cronaca, esiste ovviamente anche una *richiesta di segmentazione*, ma è più complessa a causa delle dimensioni variabili dei segmenti). Tale sistema è simile alla paginazione con swap, ma gli scambi di memoria dei processi non avvengono sull'intero codice e vengono attuati da uno *scambiatore pigro* che non swappa mai una pagina in memoria a meno che non sia assolutamente necessaria. Definirlo quindi **paginatore** è più appropriato, dato che tratta le singole pagine.

Concetti fondamentali

Il paginatore porta in memoria solo quelle pagine che ipotizza saranno necessarie. Per distinguere le pagine caricate da quelle che non lo sono si utilizza il bit di *validità/non validità*, che indica rispettivamente se la pagina è legale e caricata in memoria centrale o viceversa; quest'informazione è ovviamente riportata nella *tabella delle pagine*. Se con un'attenta paginazione si riesce a indovinare e caricare tutte e sole quelle pagine che saranno richieste, è come se fosse stato caricato l'intero processo dato che una pagina non richiesta anche se non valida non provocherà alcun effetto sull'esecuzione.
E se invece il processo cerca di accedere proprio a una pagina non valida? In questo caso scatta una *trap di mancanza di pagina* (**page fault**) che mette in moto la seguente routine:

1. si ricontrolla la tabella delle pagine del processo per verificare che la pagina richiesta non sia effettivamente valida
2. se la pagina non è legale si termina il processo, altrimenti bisogna paginarla
3. si cerca un frame libero
4. si schedula un'operazione su disco per leggere la pagina voluta nel frame appena allocato
5. a operazione terminata si aggiorna la tabella delle pagine
6. si fa ripartire l'istruzione che era stata interrotta dalla trap

Il caso limite è far partire un processo senza aver caricato nessuna parte del suo codice in memoria: si avranno una novena di page fault finché non riuscire a funzionare normalmente. Questo sistema si chiama **richiesta pura di paginazione** e garantisce che venga caricato il minimo numero indispensabile di pagine. Ovviamente troppi page fault appesantiscono il sistema, ma vedremo come limitarne l'occorrenza con alcuni accorgimenti e osservazioni, come quelle sulla *località dei riferimenti*.

Prestazioni della richiesta di paginazione

Il tempo di accesso effettivo per una memoria a una richiesta di paginazione coincide pressapoco al tempo di accesso (da 10 a 200 ns) in mancanza di page fault, altrimenti aumenta proporzionalmente al loro numero. La gestione di un page fault abbiamo infatti visto che è piuttosto onerosa (si arriva anche a centinaia di istruzioni).
Se chiamiamo *p* la probabilità di riscontrare un page fault, avremo che il
tempo di accesso effettivo = (1-p) x ma + p x tpf
, dove ma è il tempo di accesso alla memoria e tpf il tempo di servizio di un page fault.

Copia durante la scrittura

Un nuovo processo viene generato con la chiamata di sistema fork(), che nella sua implementazione originale provoca una duplicazione dello spazio di indirizzamento del padre che sarà assegnata al figlio. Dato però che molti processi eseguono una exec() subito dopo la creazione (modificando così il proprio codice di partenza), la copia completa degli spazi di indirizzamento è spesso uno spreco di tempo e spazio.
Una tecnica alternativa è la **copy-on-write**, che permette ai processi padri e figli di condividere inizialmente le stesse pagine (opportunamente contrassegnate), e se poi uno dei due ne vuole modificare una ne viene generata una copia

che sarà mappata nello spazio logico di quel processo. Tutte le pagine non modificate o marcate come non modificabili (ad esempio quelle contenenti il codice) sono condivise con considerevole risparmio. Le copie generate durante una copy on write vengono allocate in pagine libere facenti parte di un pool appositamente mantenuto, su cui vengono applicati dei meccanismi *zero-fill-on-demand*, ovvero vengono azzerate le pagine (cancellando tutto il loro contenuto) prima di essere allocate.

Associazione dei file alla memoria

Sostituzione della pagina

Abbiamo visto come il non allocare tutte le pagine di un processo possa aumentare il livello di multiprogrammazione, ma si potrebbe incorrere in un nuovo problema: la *sovra-allocazione*. Se infatti il processo (che ricordiamo condivide la memoria centrale anche con i buffer di I/O) improvvisamente ha bisogno di allocare tutte le sue pagine e non ci sono più frame a disposizione che si fa? Terminarlo non è saggio, lo swap lo vedremo poi, vada per la **sostituzione delle pagine**!

Sostituzione di base della pagina

Di seguito illustreremo i passi della sostituzione di base della pagina:

1. trovare la posizione della pagina desiderata su disco
2. trovare un frame libero
    - se c'è, usarlo
    - se non c'è, selezionare un frame vittima
    - scrivere quest'ultimo su disco e aggiornare la tabella delle pagine e quella dei frame
3. leggere la pagina desiderata nel frame libero e aggiornare le tabelle
4. riprendere il processo

Notare che in mancanza di frame liberi si rende necessario il trasferimento di due pagine, una verso il disco e l'altra verso la memoria centrale. Con un semplice accorgimento hardware, il *bit di modifica*, si riesce però a fare in modo di evitare che siano trasferite su disco le pagine non modificate o non modificabili.

La sostituzione di base della pagina garantisce la separazione tra memoria fisica e virtuale, ed il suo algoritmo implementativo assieme a quello per l'allocazione dei frame deve essere oggetto di scrupolosa attenzione dato che piccoli miglioramenti su di essi comportano notevoli guadagni sull'intero sistema. Questi algoritmi vengono valutati facendoli operare su **stringhe di riferimento**, ovvero stringhe degli identificatori delle pagine richieste, generate casualmente o prese direttamente dall'esecuzione di un sistema reale. Vediamo ora alcuni algoritmi di sostituzione.

Sostituzione FIFO della pagina

Quello **FIFO** è l'algoritmo di sostituzione più facile da capire e programmare, e consiste nella sostituzione della pagina più vecchia (calcolata con un timestamp o grazie a una coda FIFO).

Le sue prestazioni non sono notevoli dato che l'idea di fondo "le pagine richieste tempo fa non saranno più necessarie" non regge con quelle usate molto frequentemente. Inoltre essendo possibile la sostituzione di una pagina attiva, il numero di page fault può aumentare sensibilmente.

Inoltre l'algoritmo FIFO soffre dell' **anomalia di Belady**, ovvero il suo tasso di mancanza di pagina all'aumentare del numero di frame può aumentare invece che diminuire come ci si aspetterebbe.

Sostituzione ottimale della pagina

L'algoritmo **ottimale di sostituzione** della pagina è quello col più basso tasso di page fault e che non soffre dell'anomalia di Belady. E' anche chiamato *OPT* o *MIN* ed è definito così: viene sostituita la pagina che non sarà usata per il più lungo periodo di tempo.

Non è implementabile dato che implica la conoscenza aprioristica della stringa di riferimento, il che è praticamente impossibile. Viene tuttavia utilizzato come termine di paragone.

Sostituzione LRU della pagina

La **sostituzione LRU** (*Least-Recently-Used*) ha come politica quella di sostituire la pagina che non è stata usata per il periodo di tempo più lungo. Se la OPT rappresenta l'algoritmo ottimale tra quelli che guardano in avanti nella stringa di riferimento, la LRU è quella ottimale tra quelli che guardano indietro. E' considerata in genere molto buona, seppur richieda un notevole supporto hardware e non tutti i sistemi operativi potrebbero tollerare un simile appesantimento della gestione della memoria

Per determinare qual è la pagina che non è stata usata per il periodo di tempo più lungo sono percorribili due strade:

- *contatori*, che si aggiungono come nuovo campo per ogni elemento della tabella delle pagine e che vengono incrementati per ogni nuovo riferimento ad esse

- *stack*, ogni volta che si fa riferimento a una pagina questa viene rimossa dallo stack e messa in cima. In questo modo tutte le pagine usate più di recente sono nella parte alta

E' dimostrabile che gli algoritmi LRU non soffrono dell'anomalia di Belady.

Sostituzione della pagina con approssimazione dell'algoritmo LRU

Per quei sistemi che non sono in grado di utilizzare l'algoritmo di sostituzione LRU sono possibili un certo numero di approssimazioni, spesso realizzate utilizzando un bit di riferimento associato a ogni pagina (e riportato nella solita tabella delle pagine) che dice se questa è stata referenziata o meno.
Vediamoli.

Algoritmo dei bit di riferimento addizionali

Ad ogni pagina è associato un byte di 8 bit. Ad intervalli regolari il sistema operativo provvede a inserire il bit di riferimento della pagina nella cifra più significativa del byte, shiftando tutte le altre cifre a destra. In questo modo si può stabilire che quella usata meno di recente è la pagina col byte più basso. Se le pagine vittima individuate sono poi più di una, si possono sostituire entrambe o applicare una FIFO.

Algoritmo della seconda possibilità

Si basa sul FIFO, con la differenza che quando una pagina viene selezionata si controlla il suo bit di riferimento: se è 0 si sostituisce, altrimenti lo si azzera e gli si dà una seconda possibilità passando all'elemento successivo. E' anche detto *algoritmo dell'orologio* e può essere implementato con una coda circolare delle pagine. Notare che se una pagina viene utilizzata abbastanza spesso può non essere mai sostituita, e che l'algoritmo degenera in FIFO se tutti gli elementi sono settati a 1.

Algoritmo della seconda possibilità migliorato

In questo algoritmo si utilizzano come chiavi sia il bit di riferimento che quello di modifica. Possiamo dunque individuare quattro classi di pagine col seguente ordinamento:

1. (0,0): pagina non usata di recente e non modificata, la migliore da sostituire
2. (0,1): pagina non usata di recente recente ma modificata, meno preferibile dato che dovrà essere scritta completamente prima della sostituzione
3. (1,0): pagina usata di recente non modificata
4. (1,1): pagina usata di recente e modificata

Si sostituisce quindi la prima pagina incontrata con l'ordine più basso, così da ridurre il numero di chiamate I/O richieste.

Sostituzione della pagina basata su conteggio

Si tiene traccia del numero di riferimenti di ogni pagina e si può scegliere tra due politiche da applicare:

- l'algoritmo di *sostituzione delle pagine usate meno di frequente*, che sostituisce le pagine col conteggio più basso basandosi sull'idea che le pagine attive avranno valori alti. Ma che fare se ad esempio una pagina viene usata pesantemente in un limitato intervallo di tempo e poi non è mai più richiesta? In questo caso una possibile soluzione è far decadere il valore del contatore a intervalli regolari
- l'algoritmo di *sostituzione delle pagine più frequentemente usate*, la cui idea di fondo è che probabilmente se una pagina ha conteggio basso è perché deve ancora essere utilizzata

Algoritmo per l'uso del buffer delle pagine

Agli algoritmi di sostituzione si affiancano altre procedure che ottimizzano ulteriormente le prestazioni. Ad esempio il sistema operativo mantiene un certo numero di frame liberi nei quali verrà scritta la pagina richiesta prima che la vittima venga salvata fuori dalla memoria centrale. In questo modo il processo potrà ricominciare il prima possibile dopo un page fault senza dover aspettare che la vittima venga salvata.
Un'altra procedura utile è fare in modo che quando il paginatore è a riposo provveda a scrivere su disco le pagine con bit di modifica settata a 1, così che non dovranno essere riscritte al momento della sostituzione.

Le applicazioni e la sostituzione delle pagine

Ci sono alcune applicazioni che hanno prestazioni peggiori con la memoria virtuale a causa della gestione particolare che essi fanno della memoria stessa, profondamente differente da quella convenzionale del file system. Stiamo parlando ad esempio dei database. Per queste applicazioni possono essere previste aree di memoria grezze, da utilizzare come se fossero una grande successione sequenziale di blocchi logici, in cui è assente una qualsiasi struttura dati.

Allocazione dei frame

La paginazione pura che abbiamo visto qualche capitolo fa è una possibile tecnica per l' **allocazione dei frame**. Inizialmente tutti i frame disponibili vengono messi nella lista dei frame liberi, e saranno occupati man mano che l'esecuzione di un processo genererà una sequenza di page fault. Quando il processo termina i frame tornano ad essere disponibili.
Sono possibili dei miglioramenti, ad esempio far rientrare nelle pagine libere anche alcune di quelle riservate al sistema operativo, o sfruttare lo swap, o riservare alcuni frame da mantenere sempre liberi per la paginazione.

Numero minimo di frame

Qual è il numero di frame da associare a un processo? Esiste una regola generale?
In realtà ogni sistema ha una propria politica, ma tutti devono rispettare un limite superiore e uno inferiore. Quello superiore è ovviamente la quantità di memoria fisica disponibile, mentre quello inferiore dipende dall'architettura del computer. Più il numero minimo di frame è piccolo maggiore è la probabilità che si abbiano page fault, quindi va stabilito con attenzione. Altro fattore critico è la possibilità di avere sistemi che consentono *livelli multipli di indirezione*, ovvero istruzioni che fanno riferimento ad indirizzi indiretti. In questi scenari la memoria fisica richiesta per una singola istruzione potrebbe coinvolgere un gran numero di frame, dunque è opportuno fissare un limite al numero di livelli.

Algoritmi di allocazione

L' **allocazione omogenea** è l'algoritmo più semplice da applicare per ripartire i frame tra i vari processi, dandone a ognuno la stessa quantità; quelli che avanzano vengono tenuti da parte come frame liberi. Si dimostra poco soddisfacente nel caso in cui un processo debba accedere spesso a grosse strutture dati o a numerose porzioni di codice.
Un'alternativa è l' **allocazione proporzionale**, che assegna la memoria in base alle dimensioni del processo. Definendo$\mathbf{S = \sum s_i}$, con s_i memoria virtuale del processo p_i, allora se il numero totale di frame disponibili è m vengono allocati $a_i$frame al processo, dove a_i è definito come: $\mathbf{a_i = (s_i / S) \times m}$. Nella maggior parte dei casi si dimostra la soluzione migliore. Una variante prevede l'utilizzo della priorità come fattore di proporzionalità.
Che sia proporzionale o omogenea, l'allocazione dei singoli processi cambia a seconda del livello di multiprogrammazione perché ci saranno più o meno processi a contendersi i frame.

Confronto tra le allocazioni con sostituzione locale e globale

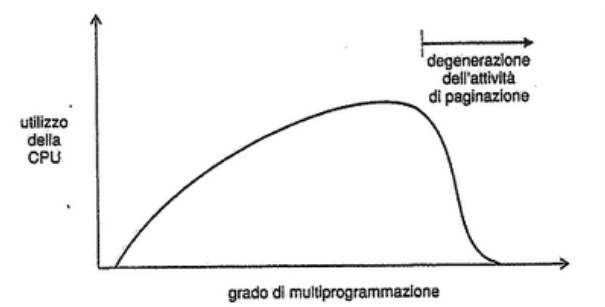
Una classificazione degli algoritmi di sostituzione delle pagine prevede la distinzione tra **sostituzioni locali** e **globali**. Queste ultime consentono a un processo di selezionare la vittima considerando tutti i frame, compresi quelli allocati ad altri processi. La sostituzione locale richiede invece che ogni processo attui la scelta solo tra i frame ad esso allocati. Il vantaggio della tecnica globale è che ogni processo può aumentare il numero di frame assegnati, ma ha per contro l'impossibilità di controllare il proprio tasso di page fault dal momento che dipende da altri processi. E' comunque l'algoritmo che dà risultati migliori ed è il più usato.

Thrashing

Se i frame assegnati a un processo scendono al di sotto del numero minimo definito dall'architettura (ad esempio nello scenario di un'allocazione proporzionale basata sulla priorità), il processo deve essere sospeso e la memoria da lui allocata va liberata. Se fosse andato avanti con un numero insufficiente di frame sarebbe presto andato incontro a un page fault, che avrebbe richiesto una sostituzione di pagine tra quelle ancora attive, il che avrebbe causato un altro page fault, e così via. Quest'alta attività di paginazione prende il nome di **thrashing**, ed il processo che ne soffre spenderà più tempo a paginare che a proseguire nell'esecuzione.

Cause del thrashing

Consideriamo questa situazione: il sistema operativo controlla l'utilizzo della CPU e se questo è troppo basso aumenta il livello di multiprogrammazione introducendo un nuovo processo. Con un algoritmo di sostituzione globale ci saranno più processi a concorrere per gli stessi frame, e rubandoseli a vicenda faranno più richieste al paginatore. L'attesa dei processi in questa coda farà diminuire ulteriormente l'utilizzo del processore, che reagirà al solito attivando un nuovo processo alimentando la spirale. Si arriverà a un punto in cui l'attività produttiva del sistema crolla mentre aumenta a dismisura il tasso dei page fault.



Questo è il classico esempio di *thrashing*, da cui si può uscire soltanto diminuendo il livello di multiprogrammazione. Si può limitarne gli effetti col rimpiazzamento locale (o a priorità) imponendo che un processo in thrashing non possa richiedere frame da altri ma se la cavi con i suoi. Questa non è una vera soluzione dal momento che il processo occuperà comunque per moltissimo tempo la coda di paginazione.
Per risolvere completamente il problema bisognerebbe invece sapere a priori di quanti frame avrà bisogno un processo, un numero che può essere approssimato applicando il **modello di località** di esecuzione. Questo modello afferma che durante la sua esecuzione il processo si sposta da una località di memoria all'altra, ovvero in gruppi di pagine (che possono anche sovrapporsi) che vengono usate insieme. La loro esistenza è diretta conseguenza della programmazione strutturata di programmi e basi di dati.

Il modello working set

Il modello working set è un'approssimazione di quello delle località e fa uso del parametro **Δ** per definire la *finestra* del working set, ovvero l'insieme dei riferimenti più recenti esaminati. Se una pagina è attiva sarà sicuramente nel working set, mentre se non è più in uso ne uscirà dopo Δ unità di tempo dal suo ultimo utilizzo. Si intuisce dunque che la precisione del modello dipende da Δ: se è troppo piccolo non rappresenterà la località in modo adeguato, se è troppo grande si sovrapporrà ad altre località (troppe).
Indicando con **wss** la dimensione del working set calcolata per ogni processo p_i del sistema, diremo che **D = ∑ wss**, dove **D** è la richiesta totale di frame. Se D è maggiore del numero di frame disponibili si avrà il thrashing.
Il sistema operativo controlla il working set di ogni processo assegnandogli frame sufficienti: se ce ne sono abbastanza fa partire un nuovo processo, ma se D supera il numero di frame disponibili allora farà in modo di sostituire le pagine inattive. In questo modo si previene il thrashing mantenendo il livello di multiprogrammazione il più alto possibile.
La difficoltà del modello è tener traccia del working set, per questo viene generalmente approssimato utilizzando un*interrupt di un temporizzatore a intervalli fissi di tempo* e un *bit di riferimento*.

Frequenza delle mancanze di pagina

Il modello working set è efficace per controllare il working set (ovviamente) e si rivela utile anche per la *prepaginazione*, ma non affronta il problema del thrashing in modo diretto quanto la strategia di **controllo della frequenza delle mancanze di pagina** (*Page-Fault Frequency*, **PFF**). Il principio è questo: si sa che il thrashing ha un alto tasso di page fault, dunque è possibile fissare un limite superiore a tale tasso superato il quale si assegna al processo un nuovo frame. Si fissa inoltre un limite inferiore per evitare che un processo abbia inutilmente più frame del necessario, sceso al di sotto del quale gliene vengono tolti uno o più. Ciò che avviene è un bilanciamento dinamico del numero dei frame assegnato ai processi in base alla situazione attuale, senza sapere a priori quanti gliene servono davvero.
Come per il working set, anche con questa tecnica può essere necessario sospendere qualche processo finché non ci sono abbastanza frame disponibili; in questo caso i frame liberati vengono distribuiti tra i processi che hanno frequenza di mancanze di pagina maggiore.

File mappati in memoria

Normalmente ogni accesso al file richiede una chiamata di sistema e un accesso al disco. Si può in alternativa usare le tecniche di accesso alla memoria virtuale per trattare le chiamate I/O al file come se fossero accessi alle procedure di gestione della memoria. Questo metodo è noto come **mappatura in memoria di un file**, ed associa logicamente un file a una parte dello spazio degli indirizzi virtuali.

Altre considerazioni

Prepaginazione

La più evidente caratteristica della pura richiesta di paginazione è il grande numero di page fault riscontrati all'avvio del processo o alla sua ripresa dopo la sospensione. La **prepaginazione** è un sistema che ha come obiettivo quello di ridurre il numero elevato di richieste di paginazione, caricando contemporaneamente in memoria tutte quelle pagine

che crede saranno necessarie nell'immediato futuro. Ad esempio nel modello working set al momento di sospendere un processo potrà tenere traccia delle pagine che fanno parte della sua finestra, così che al momento della riattivazione potrà caricarle contemporaneamente prevenendo la richiesta.
Le intenzioni della prepaginazione sono buone, bisogna però fare in modo che le pagine prepaginate non siano tra quelle che non verranno mai usate dal processo, o i vantaggi svanirebbero.

La dimensione della pagina

Come si sceglie la dimensione della pagina? Vanno considerati diversi fattori, ad esempio con pagine molto piccole si utilizzerà meglio la memoria minimizzando le frammentazioni, ma si avranno tabelle delle pagine molto più grandi. Per quanto riguarda le operazioni di I/O sulle pagine va considerato che spesso i tempi di latenza e di ricerca superano quelli effettivi di trasferimento, il che sembrerebbe avvantaggiare le pagine più grandi. Allo stesso tempo però pagine più piccole accentuerebbero le località, migliorando così la risoluzione e dunque permettendo di isolare esclusivamente le aree di memoria realmente necessarie. Inoltre per pagine grandi una minuscola modifica comporterebbe comunque il trasferimento dell'intero blocco.
Il problema non ha dunque una risposta migliore in assoluto, anche se storicamente si sta tendendo a preferire pagine con dimensioni più grandi.

Estensione della TLB

L' **estensione della TLB** è una misurazione data dalla dimensione della pagina moltiplicata per il numero di elementi nella tabella. Maggiore è il suo valore e minore è il tasso di page fault ed il tempo di accesso medio alla memoria. Si può dunque tentare di intervenire sull'estensione della TLB aumentando il numero di elementi o la dimensione delle pagine, magari usandone di dimensioni diverse.

Tabella delle pagine invertita

La **tabella delle pagine invertita** riduce la quantità di memoria fisica necessaria per la traduzione da indirizzo logico a fisico, ma non ha alcune informazioni necessarie alla paginazione. Per questo motivo vengono mantenute tabelle esterne delle pagine referenziate quando c'è un page fault.

Struttura del programma

Abbiamo visto che i page fault possono essere ridotti se il programma ha una forte località, che può essere ottenuta perseguendo una buona strutturazione del programma, quindi applicando le modularità, studiando attentamente le figure strutturali ed analizzando quali strutture dati sono più adeguate per il problema. In questo modo compilatori e linker produrranno codici decisamente più efficienti sotto il profilo della paginazione.

Blocco dei dispositivi di I/O

Un'operazione di I/O prevede l'utilizzo di un certo numero di frame, e bisogna fare in modo che non avvenga alcuna sostituzione globale su di essi prima che l'I/O non sia concluso, o la risposta sarebbe ricevuta dal processo sbagliato. Una soluzione è fare in modo che i buffer I/O siano nello spazio di indirizzamento del sistema operativo (il risultato andrà poi però copiato nella memoria del processo, con un conseguente appesantimento del sistema), un'altra è bloccare tali pagine adoperando un *bit di blocco*.

Elaborazione in tempo reale

La memoria virtuale può introdurre dei ritardi inaspettati nella computazione a causa dei tempi di gestione della paginazione, una caratteristica incompatibile con i processi in tempo reale, per i quali il fattore critico è proprio il tempo.
Una soluzione potrebbe essere fare in modo che i processi in real time possano dire al sistema operativo quali sono per loro le pagine critiche, così che possano essere precaricate ed eventualmente anche bloccate. In caso di abuso tutti gli altri processi potrebbero essere sospesi: priorità massima al processo in tempo reale.

File System

Il **file system** fornisce il supporto per la memorizzazione e l'accesso ai dati e ai programmi, ed è l'aspetto più visibile agli utenti del sistema operativo.
E' composto dai *file* (che contengono dati o programmi), dalla *struttura della directory* (che li organizza e ne fornisce le informazioni) e in alcuni sistemi dalle *partizioni* (che separano logicamente o fisicamente grandi insiemi di directory).

Il concetto di file

Il concetto di **file** è qualcosa di estremamente generale, cerchiamo di darne alcune definizioni. In primo luogo esso rappresenta un insieme di informazioni, è identificato in modo univoco da un nome ed è memorizzato in un

dispositivo di memoria secondaria così che sia sempre disponibile anche in seguito a un riavvio del sistema. Può essere anche considerato come una sequenza di bit (o di byte, o di righe, o di record) il cui significato è definito dal creatore del file e dall'utente.
In un file possono essere memorizzati molti tipi diversi di informazioni: programmi sorgente o eseguibili, testi, immagini, ... Ognuno di essi ha una specifica struttura strettamente legata al tipo, così che possa essere trattato opportunamente dal sistema operativo.

Attributi del file

Gli *attributi* del file possono variare col sistema operativo, ma in genere sono:

- **nome**, ovvero il nome simbolico del file composto generalmente da una stringa di caratteri. Viene mantenuto in forma leggibile dagli utenti ed è indipendente dal processo, dall'utente e dal sistema operativo che lo hanno creato
- **identificatore**, un'etichetta solitamente numerica che identifica in modo univoco il file all'interno del file system
- **tipo**
- **locazione**, il puntatore al dispositivo e alla locazione fisica del file
- **dimensione corrente**, espressa in byte, parole o blocchi
- **protezione**, ovvero informazioni per il controllo dell'accesso
- **tempo**, **data** e **identificativo dell'utente**, tutte informazioni utili per la sicurezza e il controllo

I valori degli attributi dei file vengono mantenuti nel **descrittore del file** contenuto nella directory, anch'essa residente in un'unità di memoria secondaria.

Operazioni sui file

Per poter meglio definire un file è opportuno descrivere il tipo di operazioni che lo riguardano:

- **creazione**, che consta di due passaggi: ricerca dello spazio nel file system e creazione di un descrittore nella directory in cui registrare il nome, la locazione ed altre informazioni
- **scrittura**, che avviene attraverso una chiamata di sistema in cui si specifica sia il nome del file che le informazioni da scriverci dentro. Dato il nome, il sistema operativo cerca nella directory la posizione fisica del file, quindi mantiene un puntatore di scrittura alla locazione in cui avverrà la scrittura successiva
- **lettura**, che avviene per mezzo di una chiamata di sistema in cui si indica il nome e dove dovrebbe essere allocato in memoria il record successivo. Sia per la lettura che per la scrittura viene mantenuto un puntatore alla posizione corrente del file per risparmiare spazio e ridurre la complessita delle operazioni
- **riposizionamento** all'interno di un file, in cui si setta il puntatore alla posizione corrente con un determinato valore. E' anche detto *ricerca nel file* e non implica un accesso alle informazioni
- **cancellazione**, che dopo aver cercato nella directory il file col nome indicato prevede il rilascio di tutto lo spazio a lui assegnato (disponibile ora per gli altri) e l'eliminazione del suo descrittore
- **troncamento**, che cancella il contenuto del file ma non i suoi attributi (salvo naturalmente la dimensione, che viene azzerata)

A queste sei operazioni di base possono esserne aggiunte altre (come l' *accodamento* o la *rinomina*) e possono essere combinate tra loro (ad esempio la *copia* o lo *spostamento*).
Dato che la maggior parte di queste operazioni prevede la ricerca nelle directory del descrittore del file (operazione lenta se effettuata su memorie secondarie), molti sistemi operativi prevedono l'*apertura* di un file con la chiamata di sistema open(). Questa operazione ne comprende molte altre, ovvero:

- la verifica delle autorizzazioni all'accesso
- l'identificazione del descrittore del file nel file system
- l'identificazione della locazione nei dispositivi fisici
- la verifica e gestione dello stato di uso condiviso
- l'inizializzazione delle informazioni per una gestione efficiente

Viene inoltre mantenuta una **tabella dei file aperti** che contiene tutte le informazioni sui file attualmente utilizzati, e dalla quale vengono tolti solo in seguito a una chiamata di sistema close() o nel momento in cui i processi che li hanno aperti terminano.

In sistemi più complessi multiutente come UNIX si preferisce usare due tabelle, una per il processo e una per il sistema, così da permettere a due processi diversi di accedere allo stesso file. A tal fine vengono mantenute nelle tabelle le seguenti informazioni:

- **puntatore al file**, che indica l'ultima posizione di lettura o scrittura ed è unico per ogni processo operante sul file
- **contatore delle aperture di un file**. Solo quando il suo valore arriva a 0 il sistema operativo può rimuovere il descrittore del file dalla tabella dei file aperti
- **posizione del file su disco**
- **diritti d'accesso**, che specificano le modalità d'accesso al file e sono riportate nella tabella dei file aperti relativa al processo

Infine, alcuni sistemi operativi permettono di bloccare un file aperto (o parti di esso) per impedire ad altri processi di accedervi. Ne esistono di tre tipi: **blocco esaustivo** o **condiviso** (in lettura, su richiesta), **blocco esclusivo** (in scrittura, su richiesta) e **blocco consigliato** (in scrittura, obbligatorio, imposto).

Tipi di file

Una tecnica comune per indicare il tipo di file è includerlo nel nome sottoforma di **estensione**, ovvero un codice (generalmente di tre caratteri) che lo identifica. Le estensioni non sono richieste, quindi se un'applicazione apre un file che non ne ha cercherà di aprirlo nel formato che si aspetta.

Struttura del file

Abbiamo detto che dal tipo del file dipende anche la sua struttura interna. Il sistema operativo deve supportare ognuna di queste strutture, quindi deve avere un codice di gestione specifico per ognuna di esse, il che può diventare molto ingombrante.

UNIX ha un numero minimo di strutture di file supportate e considera ogni file come una sequenza di byte (di 8 bit) che il sistema operativo non interpreta in alcun modo. Se questo da un lato garantisce la flessibilità, dall'altro dà poco supporto demandando alle applicazioni il compito di includere il proprio codice interpretativo.

In generale un sistema operativo non deve avere troppi tipi supportati o sarebbe troppo voluminoso, ma non deve averne nemmeno troppo pochi o renderebbe la programmazione difficoltosa.

Struttura interna del file

Esistono tre tipi di strutturazione di un file:

- **nessuna**, ovvero una sequenza di byte o parole una dietro l'altra
- **struttura semplice**, suddivisa per linee a lunghezza fissa o variabile
- **struttura complessa**, tipica dei documenti formattati o dei file caricabili rilocabili

Metodi di accesso

Accesso sequenziale

L' **accesso sequenziale** è il metodo più semplice, in cui le informazioni nel file vengono elaborate in ordine un record dopo l'altro. E' l'accesso tipico degli editor e dei compilatori.

Accesso diretto

L' **accesso diretto** o *relativo* è basato su un modello di file composti da record logici di lunghezza fissa che permettono ai programmi di leggere e scrivere velocemente senza un ordine particolare. Sono molto utili per l'accesso immediato a grandi quantità di informazioni, per questo motivo le basi di dati li adottano frequentemente.

Le operazioni sui file devono essere modificate per includere come parametro il numero del blocco fisico relativo, che rappresenta un indice riguardante l'inizio del file e che fa in modo che l'utente non possa accedere a blocchi che non fanno parte del file stesso.

Alcuni sistemi operativi supportano sia l'accesso diretto che quello sequenziale, altri solo uno dei due. E' possibile inoltre simulare il sequenziale a partire dal diretto, ma con risultati poco soddisfacenti.

Altri metodi di accesso

Un possibile metodo di accesso basato su quello diretto è l' **accesso indicizzato** che prevede la costruzione di un' *indice*, ovvero una serie ordinata di puntatori a vari blocchi fisici. Per trovare un record logico nel file bisogna prima cercarlo nell'indice, quindi usare il puntatore per accedere direttamente al file e al record voluto. Questa struttura permette di eseguire una ricerca in un file di grosse dimensioni con poche chiamate di I/O.

Se l'indice diventa troppo grande è a sua volta indicizzabile; si avrà dunque un *indice primario* che punterà a quello *secondario* che referenzia i dati effettivi.

Struttura della directory

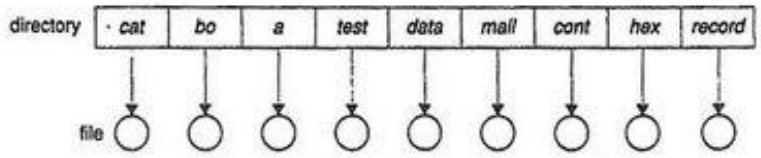
I file system possono essere molto vasti dunque è prioritario organizzarli al meglio. Anzitutto ogni disco è diviso in una o più **partizioni** (o *volumi*), cioè delle strutture a basso livello in cui sono memorizzati file e cartelle. Notare che un sistema può avere più partizioni o averne una in comune con altri sistemi. Esse consentono all'utente di preoccuparsi esclusivamente della directory logica e dei file, trascurando tutti i problemi di allocazione che ci sono dietro. Ogni partizione mantiene le informazioni sui file in essa contenuti nella *directory del dispositivo* (o *indice del volume*).
La **directory** è una struttura informativa che può essere vista come una tabella simbolica il cui compito è tradurre i nomi dei file nei corrispondenti descrittori. E' in grado di supportare il raggruppamento di file in base a criteri logici, la gestione efficiente dell'accesso ad essi, la loro condivisione e protezione.
Le operazioni che si possono compiere sulle directory sono:

- **ricerca** di un file
- **creazione** e **cancellazione**
- **elenco** di una directory
- **rinomina**, che può far cambiare la posizione dei file nella struttura generale
- **attraversamento del file system**, che permette di accedere a ogni directory e ad ogni file all'interno della struttura

Per strutturazione del file system si intende l'organizzazione globale dei file del sistema nelle varie directory.
Vediamone alcuni tipi.

Directory a singolo livello

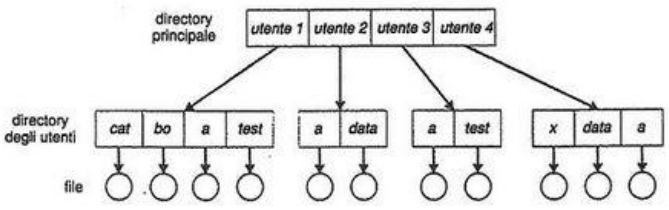
La **directory a singolo livello** è la struttura più semplice, in cui tutti i file si trovano in un'unica directory.



Ha un grosso limite: dato che i nomi dei file in una stessa cartella devono essere univoci, quando il loro numero aumenta (soprattutto se si pensasse di rendere tale sistema multiutente) la loro gestione e raggiungibilità diventerebbe decisamente ostica. Inoltre non essendo permesso alcun raggruppamento risulta impossibile dare una qualsiasi struttura logica a file legati a diverse attività.
Questa strutturazione era pensata per sistemi monoutente con una piccola memoria di massa, in cui non esisteva ancora il concetto di omogeneizzazione tra risorse fisiche e informative.

Directory a due livelli

Nella struttura della **directory a due livelli** ogni utente ha una propria *home directory* (**UFD**) che contiene solo e soltanto i suoi file. Quando ha inizio un processo dell'utente o un utente si connette, viene effettuata una ricerca nella *directory principale* (**MFD**) indicizzata per nome utente o numero dell'account, in cui ogni descrittore punta al rispettivo UFD. In questo modo utenti diversi possono avere file con lo stesso nome dato che ognuno opera esclusivamente sui file nella propria home directory.



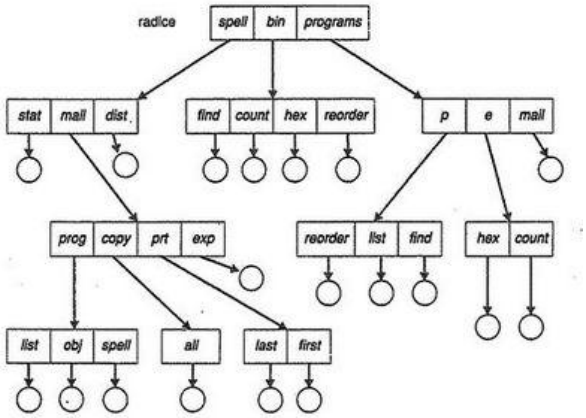
Tale struttura viene mantenuta aggiornata da un programma di sistema che si preoccupa di creare o cancellare le UFD e i rispettivi descrittori nelle MFD. Può essere anche vista come un albero a due livelli, dove la directory principale è la radice, le home sono i figli e i file sono le foglie. Con questa rappresentazione è possibile definire un percorso univoco per ciascun file lungo l'albero.
Il limite delle **directory a due livelli** è che risolvendo il problema del conflitto dei nomi isolando un utente dall'altro non ha tenuto conto dell'eventualità che in alcuni casi gli utenti vogliano cooperare e accedere a file condivisi.
Per quanto riguarda la condivisione esiste però un'eccezione, ovvero una speciale directory utente che contiene tutti i file di sistema configurata in modo che se un utente non trova un file nella propria UFD potrà provare a cercarlo lì.

Visione logica del file system

Come avere una vera astrazione dell'informazione? Come organizzarla in modo logico prescindendo da dettagli come proprietà, condivisioni, ... ? Il modo più naturale per trovare informazioni è raggrupparle per aspetti comuni in sottoinsiemi disgiunti via via più specifici. Si può intuitivamente rappresentare questa struttura con un albero, dal quale è facilmente deducibile che nessun file potrà appartenere a direttori diversi dato che può avere un unico percorso radice-file.
Con questa struttura non sappiamo più dove si trovano fisicamente i file nel disco, ma lo sappiamo a livello logico ed è tutto di guadagnato. La posizione logica è il percorso del file, ovvero la sequenza di cartelle a partire dalla radice che attraverso per arrivare al file stesso.

Directory strutturata ad albero

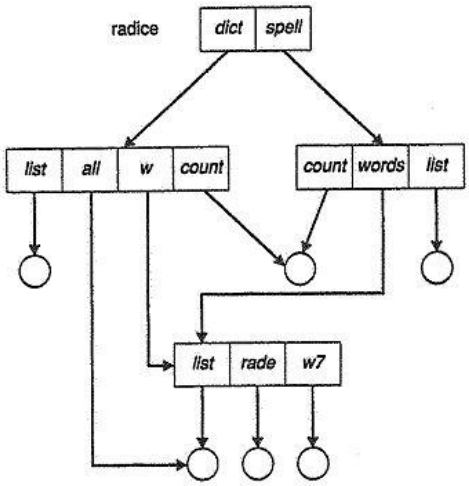
La **directory strutturata ad albero** estende di fatto la struttura della directory a un albero di altezza arbitraria. Gli utenti possono così creare delle proprie sottodirectory e organizzare in esse i file, applicando una visione logica al file system. L'albero ha una *directory radice* (*root*) e ogni file ha un percorso assoluto (che inizia dalla radice e attraversa tutte le sottodirectory fino al file specificato) e uno relativo (che parte dalla directory corrente).



La *cartella corrente* è quella attualmente richiesta dall'utente e dovrebbe essere quella che contiene i file di attuale interesse; proprio per questo motivo si può cambiare in qualsiasi momento.
Questo sistema consente all'utente di realizzare una struttura arbitraria dell'insieme dei file, e gli permette di accedere oltre che ai propri anche a quelli altrui.
Un'interessante decisione politica in una struttura ad albero riguarda la cancellazione di una directory: si possono eliminare cartelle non vuote oppure sì? Da un punto di vista implementativo entrambi i meccanismi sono piuttosto facili da realizzare, e in particolare il secondo è più conveniente ma più pericoloso.

Directory a grafo aciclico

La **struttura a grafo aciclico** è un'estensione della struttura ad albero e consente a due o più utenti di condividere uno stesso file o una stessa cartella facendola apparire in due diverse sottodirectory. Notare due cose: 1. non si tratta di copie, gli interventi avvengono sullo stesso elemento 2. non ci sono cicli, come immaginabile.



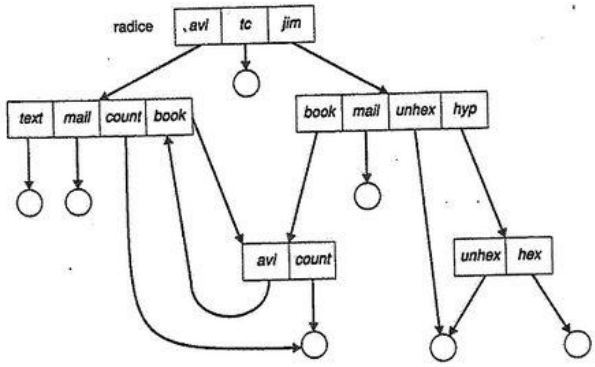
Come si implementa? O utilizzando i *link* (quindi puntatori ad altri file o cartelle) o *duplicando le informazioni contenute nei loro descrittori* (uno per directory). In quest'ultimo caso abbiamo però alcuni fattori critici:

- bisogna sempre garantire la coerenza delle copie in seguito a scritture

- un file può avere diversi percorsi assoluti
- bisogna fare attenzione al momento della cancellazione di file condivisi perché potrei rimanere con puntatori pendenti. Una soluzione potrebbe essere non cancellare un file finché non sono stati cancellati tutti i riferimenti ad esso, il che implica il mantenimento di una lista o di un contatore

Directory a grafo generale

La **struttura a grafo generale** è la generalizzazione (appunto) di quella precedente e permette l'esistenza di cicli al suo interno. Se in questo modo viene risparmiato il dispendioso onere di assicurare che non ci siano cicli, dall'altro si perdono la semplicità degli algoritmi di attraversamento del grafo e la possibilità di determinare quando non ci sono più riferimenti ad un file. Se infatti un file si auto-referenziasse, anche se non venisse mai più usato non si potrebbe mai cancellarlo dato che il suo contatore sarebbe diverso da 0.



Il sistema adottato per individuare i file e le cartelle da cancellare è chiamato **garbage collection** e consiste nel raggiungimento e marcatura di tutti gli elementi del file system in cui si riesce ad arrivare. Tutto ciò che non è stato marcato viene inserito in una lista dello spazio libero e dunque considerato disponibile. Questo sistema è tuttavia estremamente dispendioso sui dischi e per questo motivo viene usato raramente.

Montaggio del file system

Il **montaggio di un file system** è quell'operazione che lo rende disponibile ai processi del sistema. La procedura è diretta: al sistema operativo viene fornito il nome del dispositivo e il *punto di montaggio*, ovvero la locazione in cui il file system da montare si aggancerà a quello esistente. A questo punto viene verificato che il dispositivo abbia un file system valido (del formato previsto) e quindi annota nella propria struttura della directory che tutto è andato a buon fine.

Questa procedura è configurabile, ad esempio è possibile specificare se il punto di montaggio deve essere una cartella vuota oppure no, se il file system agganciato può vedere quello esistente oppure no, se un file system può essere montato in diversi punti oppure no, ecc.

Condivisione dei file

Il caso di più utenti

Per implementare la **condivisione** e la **protezione** in sistemi operativi che supportano la multiutenza si deve conservare per ogni file o cartella più attributi di quanti ne sarebbero necessari in un sistema monoutente.

La strategia più diffusa introduce il concetto di **proprietario** e di **gruppo**. Il primo è l'utente che può modificare gli attributi e garantire l'accesso, il secondo invece definisce un sottoinsieme di utenti che possono voler condividere l'accesso al file. Nella maggior parte dei sistemi operativi l'attributo del proprietario si realizza con l'identificatore di utente (*ID*) e quello di gruppo con l'ID di gruppo. Notare che ogni utente può appartenere a più gruppi.

File system remoti

Con l'evoluzione della tecnologia delle reti è oggi possibile avere numerose strade per la condivisione dei file: trasferimento manuale tra macchine mediante programmi come *ftp*, utilizzo di file system distribuiti (*DFS*), world wide web, ecc.

Il modello Client-Server

I file system remoti permettono a un computer di montare uno o più file system da una o più macchine remote. La macchina che contiene i file è il *server*, che li specifica a livello di partizione o sottodirectory. La macchina che chiede accesso a tali file è il *client*, che può essere identificato con un nome di rete o un indirizzo IP. La comunicazione client-server deve essere quanto più controllata e sicura possibile data la sensibilità dell'operazione.

Una volta montato il file system remoto i client effettuano le richieste d'accesso ai file condivisi attraverso il protocollo DFS. Il server controlla se su quella determinata macchina l'utente con quell'ID che ha richiesto un accesso è autorizzato; se sì gli viene ritornato un descrittore (*file handle*) e l'applicazione può compiere l'operazione desiderata.

Sistemi informativi distribuiti

I **sistemi informativi distribuiti** forniscono un accesso unificato alle informazioni necessarie per il calcolo remoto. Ad esempio il *server dei nomi del dominio* (*DNS*) effettua la traduzione del nome dell'host in un indirizzo di rete valido per tutta Internet. Altri sistemi informativi distribuiti utilizzano una combinazione di username/password/user ID/group ID per identificare i file ed effettuare operazioni in ambiente distribuito.

Per quanto riguarda i meccanismi di assegnazione dei nomi si sta consolidando il *protocollo leggero di accesso alle directory* (**LDAP**), che garantisce con un'autenticazione singola sicura l'accesso a tutti i computer del sistema e alle informazioni in essi contenuti.

Modalità di guasto

Nei file system remoti le possibilità di guasto sono molto maggiori rispetto a quelli locali a causa della maggiore complessità dell'architettura, che si estende lungo una rete. Oltre ai guasti locali si sommano dunque gli altri legati alla comunicazione tra host, danni che rendono impraticabili gli accessi ai file system remoti.

Va detto però che i rischi ci sono, ma non così alti. E' tuttavia preferibile in caso di guasto evitare di terminare tutte le operazioni in atto e di attendere qualche tempo confidando che il malfunzionamento venga riparato o bypassato.

Semantica della coerenza

La **semantica della coerenza** si applica a quelle operazioni in cui più utenti accedono contemporaneamente a un file condiviso e ne specifica le modalità d'accesso per garantire la consistenza delle informazioni. In altre parole definisce le modalità di aggiornamento dei file condivisi: modifiche immediatamente visibili, modifiche visibili solo dopo la chiusura dei file, visibili solo nelle sessioni successive alla chiusura del file, file condivisi immutabili, ...

Per garantirla è necessario introdurre il concetto di **sessione**, ovvero quella serie di accessi (in lettura o scrittura) che si effettuano sul file tra una open() e una close(), quindi per tutta la sua durata di apertura.

Protezione

Quando le informazioni sono memorizzate in un computer si desidera mantenerle al sicuro dai danni fisici (*affidabilità*) e da accessi impropri (*protezione*).

Tipi di accesso

La risposta più adeguata alla necessità di proteggere i file è l' **accesso controllato**, che tende a limitare i tipi di accesso accordando o meno permessi per operazioni di lettura, scrittura, esecuzione, accodamento, cancellazione o elenco. Esistono altre tipologie di accesso che possono essere controllate (ad esempio la rinomina) ma sono a livello più alto.

Controllo dell'accesso

Il sistema più conveniente per controllare e proteggere gli accessi è considerarli dipendenti dall'identità degli utenti. Si può dunque associare ad ogni file una **lista di controllo degli accessi** (**ACL**) che il sistema operativo consulta per verificare che l'utente è autorizzato o meno all'accesso richiesto. La versione più usata della ACL è quella ridotta, in cui sono definiti tre soli tipi di utente: *proprietario* (chi ha creato il file), *gruppo* (utenti che condividono il file e hanno tipi di accessi simili), *universo* (tutti gli altri). In questo modo sono necessari solo tre campi per definire la protezione, in UNIX di 3 bit ciascuno: rwx-rwx-rwx.

Altri metodi di protezione

Un altro sistema di protezione potrebbe essere associare ad ogni file una password, ma poi si richiederebbe all'utente di ricordarsene troppe e sicuramente userà sempre la stessa vanificando l'utilità della tecnica. Sarebbe meglio applicare una password solo alle directory, ma basterebbe scoprirne una per accedere a molte risorse informative.

Implementazione File System

Struttura del file system

Il file system risiede permanentemente in un' *unità di memorizzazione secondaria*, ovvero su un disco in grado di contenere una grande quantità di dati anche in assenza di alimentazione. Su tali supporti è possibile riscrivere i file sempre nella stessa locazione e si può accedere a qualsiasi blocco di informazioni in modo sia sequenziale che diretto. I dischi si dimostrano dunque una scelta particolarmente efficace per la memorizzazione. Per migliorare ulteriormente l'efficienza dell'I/O i trasferimenti tra memoria centrale e secondaria vengono effettuati in *blocchi* costituiti da più *settori*.

Quando si progetta un file system bisogna valutare attentamente come dovrà apparire all'utente e bisogna studiare quali algoritmi e strutture dati si dimostrano più idonei per mappare la struttura logica nei dispositivi fisici.
Il file system è composto da vari livelli, ognuno dei quali sfrutta le funzionalità di quelli sottostanti. Al livello più basso c'è il **controllo I/O**, composto dai *driver del dispositivo* e dai *gestori delle interruzioni*. I primi si occupano delle traduzioni delle istruzioni ad alto livello in un linguaggio comprensibile alla macchina, i secondi vengono invece utilizzati per trasferire le informazioni tra memoria centrale e i dispositivi.
Abbiamo poi il **file system di base**, che invia comandi generici al dispositivo interessato per leggere e scrivere nei blocchi fisici (identificati con l'indirizzo numerico del disco).
Il livello superiore è il **modulo di organizzazione dei file**, che oltre a gestire l'organizzazione dei blocchi fisici in modo da ricostruire ogni file traducendo gli indirizzi si coordina col gestore dello spazio libero.
Infine c'è il **file system logico** che si occupa dei *metadati*, ovvero tutte le strutture del file system eccetto il contenuto dei file. Gestisce la struttura della directory e la struttura di ogni file tramite il **blocco di controllo di file**, ed è responsabile anche della loro protezione e sicurezza.

Realizzazione del file system

Panoramica sulla realizzazione del file system

Per realizzare un file system vengono utilizzate diverse strutture su disco e in memoria, che possono variare sensibilmente a seconda del sistema operativo. Su disco abbiamo:

- **blocco di controllo del boot**, tipicamente il primo blocco della partizione, che contiene le informazioni necessarie per avviare il sistema operativo. E' vuoto se sul disco non è presente alcun sistema operativo
- **blocco di controllo della partizione**, che contiene dettagli sulla partizione: dimensione e numero dei blocchi, contatore e puntatori di quelli liberi, ecc
- **directory** per raggruppare i file in sottoinsiemi
- **blocchi di controllo di file** (**FCB**), che contengono informazioni sui file quali proprietà e attributi (proprietario, permessi, numero e dimensione dei blocchi, ...)

Le informazioni in memoria centrale sono usate sia per la gestione del file system che per il miglioramento delle prestazioni tramite la cache. Abbiamo le seguenti strutture:

- la **tabella delle partizioni**, che contiene informazioni su ciascuna delle partizioni montate
- **descrittori delle directory**
- la **tabella dei file aperti** di tutto il sistema, che tiene traccia anche del numero di processi che hanno aperto un certo file
- la **tabella dei file aperti** per ciascun processo
- la **tabella di montaggio del file system**, che serve a costruire un'unica visione del file system anche se la sua struttura è spezzata in volumi logici diversi

Per creare un nuovo file il programma fa una chiamata di sistema al file system logico che, una volta assegnatogli un nuovo descrittore, carica nella memoria centrale la directory appropriata e la aggiorna con il nome e il descrittore del nuovo file. A questo punto riscrive la directory anche in memoria secondaria.
Per poter essere utilizzato in operazioni di I/O il file deve essere *aperto* con la open(), che subito verifica se si trova nella tabella dei file aperti di sistema. Se c'è, si crea un puntatore nella tabella del processo a quella generale; altrimenti il file system cerca il file e ne copia il descrittore nella tabella di sistema, che referenzierà con un puntatore nella tabella del processo. Il puntatore con cui vengono effettuate tutte queste operazioni si chiama **descrittore del file** o **file handle**.
Quando un processo chiude il file si rimuove l'elemento dalla tabella del processo e viene decrementato di uno il contatore associato al file nella tabella di sistema dei file aperti; in particolare se il valore di tale contatore arriva a 0, si rimuove l'elemento anche dalla tabella generale.

Partizioni e montaggio

Ogni partizione può essere *formattata* o *grezza* a seconda che contenga o meno un file system. Il disco grezzo si usa quando non c'è un file system adatto all'uso che si vuol fare della partizione, ad esempio un'area di swap, o un'area di memoria per un database, o ancora informazioni per dischi RAID, ecc.
Le informazioni sull'avvio del sistema possono essere mantenute in una partizione separata dato che in quella fase non sono ancora caricati i driver del file system. In questa partizione vengono generalmente mantenute una sequenza di istruzioni sul come caricare il sistema operativo, o più di uno se c'è un loader che permette il dual-boot.
La *partizione radice* (root partition) contiene il kernel del sistema operativo più altri file di sistema e viene montata

anch'essa all'avvio; tutte le altre partizioni possono essere montate automaticamente di seguito a queste o manualmente più tardi.
Durante il montaggio viene verificato che il file system sia valido e compatibile, e se non lo è bisogna controllare la coerenza della partizione ed eventualmente correggerla. Se tutto va buon fine non rimane che annotare nella tabella di montaggio in memoria quanti e quali tipi di file system sono presenti. *Windows* li indica con lettere seguite dal due punti.

File system virtuali

Come integrare nello stesso sistema file system di tipo diverso in modo trasparente all'utente? Semplice: isolando le funzionalità delle chiamate di sistema dai dettagli di realizzazione, implementando il file system in tre strati principali:*interfaccia* (con le chiamate open, read, write e close più i descrittori), *file system virtuale* e quelli *locali* o *remoti*.
Il **file system virtuale** (**VFS**) ha due funzioni:

- separare le operazioni generiche del file system dalla loro realizzazione utilizzando un'interfaccia chiara e semplice. In uno stesso sistema possono coesistere più interfacce
- rappresentare i file col meccanismo del *vnode*, che contiene un identificatore numerico univoco del file in tutta la rete. Il kernel mantiene una struttura vnode per ogni nodo attivo, file o cartella che sia. Permette di vedere il file system come se fosse interamente in locale anche se alcune parti sono remote

Realizzazione della directory

Lista

Quello della **lista** è il metodo più semplice per realizzare una directory: una lista dei nomi dei file con puntatori ai blocchi dei dati. Lo svantaggio è però la necessità di effettuare una ricerca lineare nella lista quando si vuole trovare un file, o aprirlo o cancellarlo. Questa ricerca implica un costo computazionale non indifferente, riducibile con una cache o ordinando la lista (anche se in questo caso ciò che risparmio nella ricerca lo impegno per l'ordinamento) ma senza risultati entusiasmanti.
Per gestire gli elementi cancellati dalla lista si può scegliere di marcarli per individuare più facilmente gli spazi disponibili, oppure si può spostare l'ultimo elemento negli spazi liberati così da ottenere le minori dimensioni possibili.

Tabella di hash

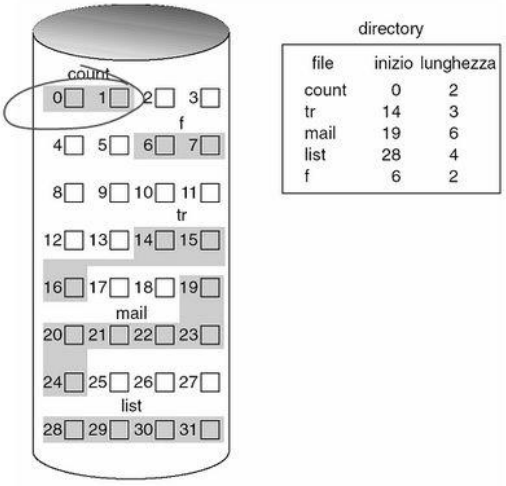
Alla lista degli elementi di una directory si può affiancare una **tabella di hash**, che permette di ridurre notevolmente i tempi di ricerca prendendo il nome del file e - applicando una funzione - restituendo un puntatore nella lista lineare. Avendo ridotto il numero di elementi da scandire questa tecnica appare più veloce, ma nasce il rischio di *collisioni*, cioè due file identificati dalla stessa posizione nella tabella. Per limitarne il numero è possibile aumentare le dimensioni della tabella o estenderla in una *tabella di hash a overflow concatenato*, in cui le collisioni vengono comunque rappresentate mettendo una lista nella cella anziché un valore unico.

Metodi di allocazione

Come allocare i file su disco nel modo più efficiente possibile? Abbiamo tre strategie: *allocazione contigua*, *collegata* o *indicizzata*. Qualche sistema le supporta tutte, ad esempio *RDOS*. Vediamole.

Allocazione contigua

L' **allocazione contigua** prevede che ogni file occupi un certo numero di blocchi contigui su disco, quindi i loro indirizzi avranno un ordinamento lineare. E' definita dall'indirizzo di inizio (*b*) sul disco e dalla lunghezza in numero di blocchi.

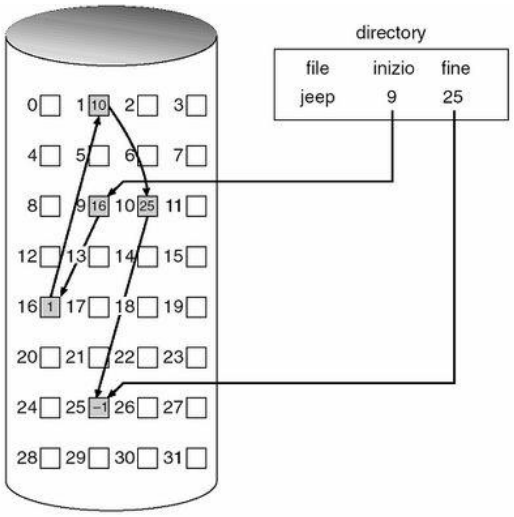


http://www.swappa.it

Offre buone prestazioni e consente sia l'accesso sequenziale che quello diretto, dato che per accedere al blocco *i*basta puntare al blocco *b+i*. Uno dei problemi di questo sistema è la difficoltà nel trovare uno spazio disponibile contiguo abbastanza grande da contenere il file, l'altro è che soffre di frammentazione esterna. Va inoltre considerato che non è sempre dato sapere quanto sarà grande un file, quindi può essergli assegnato un numero di blocchi insufficiente (e se ne chiederà altri bisognerà terminare il programma o copiarlo in uno spazio più grande) o sovrastimato (con gli sprechi che comporta).
Esiste una variante che cerca di superare l'ultimo problema affrontato ed è l' **allocazione contigua modificata**, in cui quando un pezzo di memoria non si dimostra abbastanza grande per contenere il file si aggiunge un'estensione, ovvero un altro spazio di memoria contiguo che andrà collegato al blocco iniziale. Migliora le prestazioni ma vanno tenute sotto controllo le frammentazioni.

Allocazione collegata

L' **allocazione collegata** supera tutti i problemi di quella contigua eliminando il problema della frammentazione e della crescita del file, facendo in modo che possa essere utilizzato qualsiasi blocco libero della lista dello spazio libero per le allocazioni. Ogni file è infatti una lista collegata di blocchi del disco, ognuno dei quali contiene il puntatore a quello successivo. La directory contiene un puntatore al primo e all'ultimo blocco del file e quindi si dimostra particolarmente adatta all'accesso sequenziale ma inefficiente per quello diretto dato che per trovare l'elemento i-esimo bisognerà scandirli tutti fino ad esso.

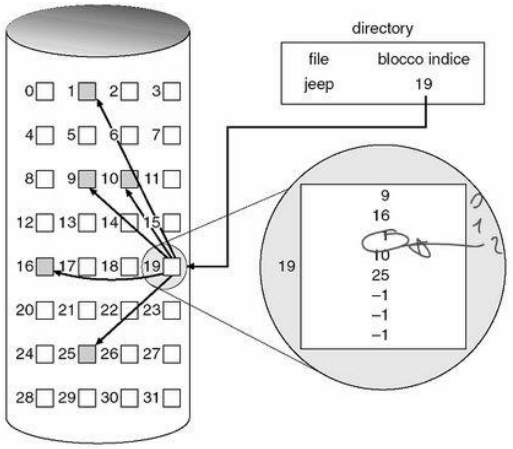


Il puntatore occupa uno spazio di memoria nel blocco che, per quanto piccolo, considerato nell'insieme diventa notevole. Per ridurre tale overhead si possono raggruppare più blocchi in unità dette **cluster**, che mantengono semplice la mappatura dei vari blocchi aumentando il rendimento del disco (meno ricerche) e la percentuale di spazio utilizzato per le informazioni e non per i puntatori. Ha come svantaggio il fatto di aumentare la frammentazione interna, ma questa dei cluster è un'ottima strategia adottata da molti sistemi operativi.
Consideriamo un altro problema, quello dell'affidabilità: cosa succederebbe se un puntatore fosse errato? Possiamo immaginarlo. Possibili soluzioni sono le *liste a doppio collegamento* o la *scrittura del nome del file in ogni blocco* (che però comporta maggior overhead).
La soluzione migliore è però un'importante variante dell'allocazione collegata, ed è la **tabella di allocazione dei file**(FAT) che viene memorizzata all'inizio di ogni partizione o in cache. Ha un elemento per ogni blocco ed è indicizzata in base al loro numero. L'elemento della directory contiene il numero di blocco del primo file; in sua corrispondenza è riportato il numero del blocco successivo e così via via fino all'ultimo elemento che ha un valore speciale di *end of file*. I blocchi liberi hanno invece valore 0.
Grazie alla FAT si ha un miglioramento del tempo di accesso diretto perché la testina del disco trova velocemente la locazione di qualsiasi blocco in essa.

Allocazione indicizzata

L' **allocazione indicizzata** supera il problema di frammentazione esterna e della dichiarazione della dimensione del file introducendo il **blocco indice** (o *i-node*), cioè un array di indirizzi dei blocchi del file su disco. L'i-esimo elemento del blocco indice punta all'i-esimo blocco del file, mentre la directory contiene l'indirizzo del blocco indice stesso. Quando si crea un file tutti i puntatori sono impostati a nil, e questo valore sarà sostituito man mano che si allocheranno

blocchi. A differenza della FAT che era unica per ogni partizione, con l'allocazione indicizzata ogni file ha la propria tabella indice.



L'allocazione indicizzata supporta dunque con semplicità l'accesso diretto senza frammentazione esterna (perché può essere usato qualsiasi blocco) ed è affidabile (perché il malfunzionamento di un blocco non si riflette sulla coda), ma comporta uno spreco di spazio non indifferente dato che l'i-node rappresenta un overhead in generale maggiore di quello dovuto ai puntatori dell'allocazione collegata. Quanto deve essere grande allora? Troppo no o si sprecherebbe spazio, troppo piccolo nemmeno o non si riuscirebbe a contenere abbastanza puntatori per i file grandi. Il problema è risolvibile con tre tecniche diverse:

- *schema collegato*: si collegano più blocchi indice tra loro. L'accesso diretto diventa un po' più lento perché bisogna leggere prima un i-node per sapere dove sono gli altri
- *indice multi-livello*: realizza un indice del blocco indice (a sua volta indicizzabile e così via), ottenendo così una struttura ad albero. I tempi di accesso si riducono in modo logaritmico
- *schema combinato*: adotta entrambe le soluzioni, con alcuni puntatori a blocchi diretti ed altri multilivello

Si può aumentare la prestazione generale dell'allocazione indicizzata spostandone la gestione nelle cache.

Prestazioni

I criteri da tenere in considerazione per decidere quali tecniche di allocazione adottare sono sostanzialmente l'*efficienza di memorizzazione* ed il *tempo di accesso ai blocchi dati*. Va inoltre considerato come avverrà l'accesso, se in modo diretto o sequenziale.

E' possibile combinare diversi tipi di allocazione cercando di sfruttare al meglio i vantaggi di ognuno. Ad esempio si può usare l'allocazione contigua per gli accessi diretti e la collegata per quelli sequenziali; oppure quella contigua per i file piccoli e quella indicizzata per i grandi, una soluzione che si è dimostrata particolarmente vantaggiosa.

Data la disparità crescente tra velocità della CPU e dei dischi si potrebbero sfruttare i tempi di allocazione per far eseguire alla CPU migliaia di istruzioni per ottimizzare i movimenti della testina, accelerando così i tempi.

Gestione dello spazio libero

Lo spazio su disco è limitato, dunque bisogna riutilizzare quello liberato dai file cancellati. E' necessario perciò mantenere una **lista dello spazio libero** in cui tener traccia di tutti i blocchi disponibili del disco, così da poterla consultare ogni qual volta si dovrà creare un nuovo file. Come implementarla?

Vettore di bit

La lista dello spazio libero viene spesso realizzata come **mappa di bit** (*bit map*) o **vettore di bit** (*bit vector*) in cui ogni blocco è rappresentato da un bit che se è libero vale 1, altrimenti 0. Il vantaggio di questo sistema è la sua semplicità ed efficienza nel trovare il primo blocco libero o *n* blocchi liberi consecutivi. Per contro si dimostra inefficiente se non è memorizzato in memoria centrale, il che rappresenta un problema per i grossi dischi dato che questi vettori sono piuttosto grandi.

Lista collegata

Il sistema della **lista collegata** prevede di collegare tutti i blocchi liberi del disco tramite puntatori, e mantenendo un puntatore verso il primo blocco in una locazione speciale del disco o in una cache. Questo schema non è efficiente per l'attraversamento della lista, ma è piuttosto utile per trovare il primo blocco libero disponibile.

Raggruppamento

Il **raggruppamento** fa in modo che vengano memorizzati gli indirizzi di *n* blocchi liberi nel primo blocco libero. L'ultimo degli n blocchi segnalati contiene in realtà gli indirizzi degli n blocchi successivi e così via. Questo sistema permette, a differenza della lista collegata standard, di trovare più rapidamente i blocchi disponibili.

Conteggio

Col **conteggio** viene tenuta in memoria non solo l'indice del primo blocco libero, ma anche un contatore che indica il numero di blocchi liberi contigui. Ogni elemento richiede dunque più informazioni da registrare come overhead, ma la lista sarà sicuramente più corta.

Efficienza e prestazioni

Avere un occhio di riguardo per le prestazioni e l'uso efficiente del disco è **fondamentale** dato che stiamo parlando del componente più lento del computer. Bisogna evitare che esso diventi un collo di bottiglia.

Efficienza

L'uso efficiente dello spazio del disco dipende dagli algoritmi usati per l'allocazione. Ad esempio è conveniente sparpagliare gli *inode* per la partizione, permettendo così che i blocchi dati di un file abbiano più probabilità di essere vicini al proprio inode riducendo così i tempi di ricerca.

La frammentazione interna dei cluster si può ridurre facendo in modo che le loro dimensioni si riducano man mano che il file si ingrandisce. Alcuni sistemi operativi applicano questa tecnica.

Altri fattori critici sono le dimensioni di blocchi, puntatori e metadati, il cui valore deve essere valutato in base alla tecnologia adoperata e alla modalità d'uso.

Prestazioni

Una volte scelte le procedure di base del file system ci sono ancora margini di miglioramento per le prestazioni. Alcune tecniche software sono la progettazione e l'impiego di algoritmi semplici ed efficaci e strutture dati ad accesso veloce. Fare in modo ad esempio che venga messa in memoria centrale la tabella degli inode aumenterà la rapidità di accesso alle sue informazioni.

Esistono poi vari tipi di supporti hardware dedicati all'accesso ai dischi:

- *cache del disco*, dove sono mantenuti i blocchi che si suppone verranno presto riusati. Abbassa il tempo di latenza
- *cache delle pagine*, che usano le tecniche di memoria virtuale per memorizzare i dati dei file come pagine invece che come blocchi per il file system, aumentando così l'efficienza del sistema. Questo sistema prende il nome di *memoria virtuale unificata*
- *buffer cache unificata*, che risolve il seguente problema della *doppia cache*: se mappo un file in memoria ed ho un buffer I/O che deve flushare su di lui, ho la riscrittura degli stessi dati su due cache diverse, quella del buffer e quella della memoria virtuale. Con la buffer cache unificata si usa invece la stessa unica cache di pagina

Che stia mettendo in cache blocchi o pagine, l'algoritmo di sostituzione LRU è il più performante. In particolare è cosa ragionevole assegnare alla paginazione la più alta priorità, o in sistemi con alto tasso di I/O la cache sarebbe oberata di richieste di pagina e la paginazione andrebbe a scapito dei processi. La cache usata per le I/O rende tali operazioni più efficienti, soprattutto quando permette di schedulare le richieste in modo da muovere la testina il meno possibile.

Comporta però particolare attenzione per le scritture asincrone da cache a disco (quelle più frequenti) perché in caso di guasto del sistema potrebbero esserci processi che non trovano alcuni file che invece in memoria centrale sembrano assegnati. Per alcuni dati critici si può dunque imporre la scrittura sincrona.

Per accessi sequenziali l'algoritmo LRU può dimostrarsi molto poco efficiente e gli si preferiscono le tecniche del *free-behind* (che rimuove una pagina dal buffer non appena si richiede la successiva) e la *read-ahead* (che carica in cache la pagina richiesta e parecchie altre tra quelle successive).

Infine, menzioniamo le *RAM-disc* o *dischi virtuali* che vengono gestite come normali file system dall'utente ma risiedono completamente in memoria centrale. Il loro vantaggio è l'indubbia velocità, che si paga con la volatilità dei dati.

Recupero del file system

Controllo della coerenza

Abbiamo visto che parte delle informazioni delle directory sono memorizzate in cache o memoria centrale per accelerarne l'accesso, dunque queste sono in media più aggiornate di quelle su disco. Cosa accadrebbe se un crash del sistema provocasse la perdita della RAM? Si perderebbe la **coerenza** tra le informazioni nei diversi supporti, con risultati spesso gravi.

Il *controllore della coerenza* ha il compito fondamentale di confrontare i dati nelle strutture delle directory e i blocchi di dati su disco: se non corrispondono cerca di ripristinare la situazione in uno stato coerente con delle tecniche che differiscono a seconda del tipo di allocazione adottata dal sistema.

Backup e ripristino

Poiché i dischi sono soggetti a possibili guasti e contengono una notevole quantità di informazioni, si dovrebbero effettuare periodicamente dei salvataggi di sicurezza (**backup**) dei dati su altri dispositivi di memorizzazione, così che in caso di necessità possano essere ripristinati.

Per evitare di ricopiare ogni volta anche quei file o cartelle che nel frattempo non sono stati modificati, è possibile consultare prima i descrittori delle directory quindi salvare solo i file effettivamente cambiati.

I backup possono essere *completi* o *incrementali*. E' cosa saggia effettuare a scadenze regolari dei backup completi e custodirli con cura nel caso in cui un utente abbia bisogno di un file molto tempo dopo che si è guastato.

Sottoinsiemi di I/O

Panoramica

Il corretto controllo dei dispositivi di I/O è uno dei compiti principali di un sistema operativo, e data l'estrema eterogeneità delle periferiche sono necessari diversi sistemi di gestione, il cui insieme va a formare il **sottosistema di I/O** del kernel.

L'hardware di I/O

I computer utilizzano molti tipi diversi di periferiche, la maggior parte delle quali rientra in tre categorie:*memorizzazione*, *trasmissione* e *interfaccia*.

In generale una periferica comunica con un computer inviando segnali via cavo o via etere attraverso un punto di connessione detto **porta**.

Se uno o più dispositivi condividono lo stesso insieme di cavi e di protocolli per la comunicazione, questa connessione prende il nome di **bus**. Sono molto comuni nelle architetture hardware.

I **controller** sono invece dei componenti elettronici che possono operare su una porta, un bus o una periferica ed hanno alcune responsabilità di gestione su esse. A seconda dalla loro complessità possono essere singoli chip o circuiti separati (come per l'hard disk).

In che modo il processore invia comandi e dati al controller per eseguire un trasferimento di I/O? Un primo sistema è fornire ad esso una serie di *registri* dedicati, quali:

- **registro di stato della periferica**, che può essere letto dal computer e che notifica ad esempio un'avvenuta esecuzione, la disponibilità di byte per la lettura, errori, ecc
- **registro di controllo** o **di comando**, che viene scritto dal computer per lanciare un comando o cambiare le modalità di una periferica
- **registro di dati in ingresso** e **registro dei dati in uscita**. A questi dure registri possono essere associati dei chip FIFO integrati per estendere di parecchi byte la capacità del controller

Un altro sistema è il **mappaggio dell'I/O in memoria**, con la CPU che esegue le richieste di I/O utilizzando le istruzioni standard di trasferimento dati in memoria centrale. Offre maggiore facilità di lettura e viene utilizzata comunemente con le schede video.

Attesa attiva

Il protocollo per l'interazione tra computer e controller si basa sulla nozione di *handshaking*. Sostanzialmente ciò che accade è questo:

1. il controller indica il suo stato per mezzo del bit *busy*, che il computer continua a leggere finché non lo trova sul valore 0 (disponibile)
2. il processore specifica il comando nel registro di controllo e setta il bit *command-ready* su 1, così che la periferica se ne accorga e lo esegua
3. terminata l'esecuzione vengono azzerati entrambi i bit.

La lettura del bit busy da parte della CPU prende il nome di **attesa attiva** perché è un'attesa che presuppone comunque un controllo. Se diventa estremamente lunga e dispendiosa le si preferisce il *meccanismo degli interrupt*, in cui è la periferica stessa che notifica la propria disponibilità al computer.

Interrupt

Il meccanismo base degli **interrupt** è il seguente: la CPU ha una connessione chiamata *linea di richiesta di interrupt* che viene verificata dopo l'esecuzione di ogni istruzione. Quando il processore si accorge che un controller le ha mandato un segnale, salva il valore corrente del program counter e passa immediatamente alla procedura di gestione dell'interrupt memorizzata in un preciso indirizzo di memoria; una volta eseguita, la CPU viene ritornata allo stato precedente.
Esistono alcuni raffinamenti a questo meccanismo, che nelle moderne architetture vengono coadiuvati dai *controller degli interrupt*. Vediamone alcune:

- possibilità di differire la gestione delle interruzioni durante le elaborazioni critiche distinguendo tra *interrupt mascherabili* e *non*, i secondi dei quali gestiscono procedure che non si possono interrompere
- interrupt rinviati nel caso di esecuzione di un processo critico
- utilizzo di un sistema efficiente per recapitare un segnale d'interruzione al corretto gestore. A tal fine si mantiene un *vettore degli interrupt* che contiene gli indirizzi di memoria dei gestori appropriati. Inoltre con la tecnica dell'interrupt chaining ("prova una procedura finché non trovi quella giusta") si possono migliorare ulteriormente i tempi di risposta
- introdurre il concetto di livello di priorità tra interrupt

Gli interrupt sono usati anche per realizzare **trap** (interruzioni software) alle procedure del kernel in modalità supervisore, quindi per l'implementazione delle chiamate di sistema.

Accesso diretto alla memoria

I trasferimenti che coinvolgono grosse entità di dati non sono efficienti se effettuati un singolo bit alla volta come abbiamo visto finora, ma è meglio affidarli a processori specifici detti controller di **accesso diretto alla memoria**(**DMA**). Un blocco di comandi DMA contiene un puntatore alla sorgente del trasferimento, uno alla destinazione e un contatore col numero dei byte da trasferire. Il controller DMA opera dunque direttamente sul bus di memoria senza alcun intervento della CPU, salvo poi lanciarle un interrupt a trasferimento compiuto.
Alcune architetture usano indirizzi fisici per il DMA, altre invece accessi diretti in memoria virtuale (DVMA) che permettono ad esempio di trasferire dati tra due periferiche senza l'intervento del processore.

Le interfacce di I/O per le applicazioni

Per far sì che un sistema operativo tratti le periferiche allo stesso modo la risposta è l'astrazione, l'incapsulamento e la stratificazione del software; si cerca cioè di isolare le differenze tra i dispositivi I/O individuandone pochi tipi generali. A ciascuno di questi tipi si accede tramite un insieme standard di funzioni che ne costituiscono l' **interfaccia**, mentre le differenze sono incapsulate in moduli del kernel detti **driver del dispositivo**. Internamente i driver sono specifici per una particolare periferica, mentre all'esterno comunicano con perfetta integrazione all'interfaccia standard.
Lo strato di *device driver* nasconde perciò le differenze tra i controller delle periferiche al sottosistema di I/O del kernel, rendendo più semplice il lavoro sia ai programmatori di sistemi operativi che ai costruttori hardware. Se però questi ultimi da un lato sanno già che tipo di interfacce offrono i sistemi, dall'altro hanno il dramma che ogni sistema operativo ha le proprie.
La migliore strutturazione del software di gestione dell'I/O è la seguente:

- *gestione del canale di comunicazione*, che deve rendere trasparente il modo in cui viene gestito il flusso di informazione tra calcolatore e periferiche
- *device dependent driver*, che rende omogenea la visione dei diversi modelli delle periferiche di uno stesso tipo (ad esempio, tutti i mouse)
- *device independent driver*, che dà invece una visione omogenea di tutti i tipi di dispositivi

Le periferiche possono differire per diversi aspetti:

- trasferimento sincrono o asincrono
- periferica condivisibile o dedicata
- velocità di elaborazione (latenza, tempo di ricerca, velocità di trasferimento, attesa tra le operazioni)
- direzione di I/O (lettura, scrittura, lettura-scrittura)
- metodo d'accesso, se sequenziale o diretto
- modo di trasferimento dei dati, se a caratteri o a blocchi

Molte di queste differenze saranno comunque nascoste dal sistema operativo.
Ultima cosa da ricordare sono le funzioni di *controllo diretto* delle periferiche, dette *escape* o *back door*, che permettono il passaggio trasparente di comandi direttamente dalle applicazioni ai driver dei dispositivi.

Dispositivi con trasferimento dei dati a caratteri e a blocchi

L'interfaccia dei dispositivi con trasferimento dei dati a blocchi si occupa di tutti gli aspetti necessari per accedere ai drive di disco e ad altre periferiche a blocchi. Le periferiche dovrebbero prevedere comandi
come read() write() oseek() per l'accesso diretto.
Gli accessi stessi da parte del sistema operativo e delle applicazioni possono sfruttare le interfacce del file system o considerare le periferiche a blocchi come semplici array lineari di blocchi. In questo secondo caso si parla di *I/O grezzo*.
Se invece l'accesso avviene per mappaggio in memoria, l'interfaccia deve fornire l'accesso alla memoria secondaria tramite un array di byte posto in memoria centrale; è quello che succede anche per l'accesso del kernel nell'area di swap.
Le periferiche con trasferimento dati a caratteri sono ad esempio tastiere mouse e modem, che producono dati in ingresso sottoforma di flussi sequenziali di byte in modo non predicibile dalle applicazioni. Sono tipici anche di schede audio e stampanti.

Le periferiche di rete

Le prestazioni e le caratteristiche di indirizzamento di una rete informatica sono molto diverse da quelle delle operazioni di I/O su disco, dunque la maggior parte dei sistemi operativi fornisce interfacce I/O di rete differenti da quelle viste finora. Tra quelle più diffuse in sistemi come UNIX e Windows c'è quella del *socket di rete*.

Orologi e temporizzatori

Nella maggior parte dei sistemi vengono utilizzati *orologi* (*clock*) e *temporizzatori* (*timer*) per tener traccia dell'ora corrente, del tempo trascorso, o ancora - è il caso del *timer programmabile* - per impostare un certo intervallo prima di scatenare un interrupt. Quest'ultima funzione può essere utilizzata con profitto per implementare ad esempio il round-robin tra processi, o per svuotare periodicamente la cache del buffer, o per interrompere le operazioni di rete che ci stanno mettendo troppo tempo.

I/O bloccante e non bloccante

Le operazioni di **I/O bloccanti** sospendono l'esecuzione dell'applicazione, che viene spostata nella coda di quelle in attesa finché la chiamata di sistema non è stata completamente eseguita; solo a qual punto l'applicazione potrà tornare nella coda dei processi pronti.
Gli **I/O non bloccanti** non prevedono tale sospensione e sono tipici delle interfacce utente d'ingresso (tastiera, mouse).

Il sottosistema di I/O del kernel

Molti servizi di I/O sono messi a disposizione dal sottosistema di I/O del kernel e montati direttamente sull'hardware e sull'infrastruttura dei driver delle periferiche.

Schedulazione dell'I/O

Schedulare una serie di richieste di I/O significa trovare un buon ordine di esecuzione delle richieste stesse, il che migliora certamente le prestazioni globali del sistema dato che l'ordine dettato dalle applicazioni è raramente lungimirante. La schedulazione viene implementata mantenendo una coda di richieste per ogni periferica e facendovi operare uno schedulatore dedicato.

Buffering

Il **buffer** è una zona di memoria in cui vengono memorizzati temporaneamente i dati mentre vengono trasferiti tra due periferiche o tra periferiche e applicazioni. Si usano principalmente per tre motivi:

1. appianano la differenza di velocità tra mittente e destinatario di un flusso di dati
2. fungono da adattatori tra periferiche con blocchi di dimensioni diverse
3. supportano la *semantica della copia*, che garantisce che la versione dei dati scritta su un dispositivo sia la stessa versione presente in memoria al momento della chiamata di sistema dell'applicazione.

Caching

La **cache** è una zona di memoria veloce che conserva copie dei dati così che l'accesso ad essi sia più veloce che su disco. La differenza col buffer è che in questi ultimi è presente la sola copia esistente di un certo dato; ciò nonostante alcune regioni di memoria possono essere utilizzate per entrambi gli scopi.

Spooling e prenotazione dei dispositivi

Lo **spool** è un buffer su memoria di massa che conserva l'output per una periferica che non può accettare flussi di dati da più processi contemporaneamente, come ad esempio le stampanti. Lo *spooler* è l'unico programma che ha accesso diretto alla periferica, e gestisce il buffer offrendo agli utenti alcune operazioni aggiuntive come la visualizzazione della coda di attesa dei processi o la cancellazione/sospensione di uno di essi.

Un'altra tecnica che consente di coordinare quelle periferiche che non possono gestire in modo efficace le richieste di I/O da parte di applicazioni concorrenti è la **prenotazione** (*locking*) delle stesse, così da ottenere l'accesso esclusivo ad esse per poi renderle di nuovo disponibili una volta terminata l'operazione.

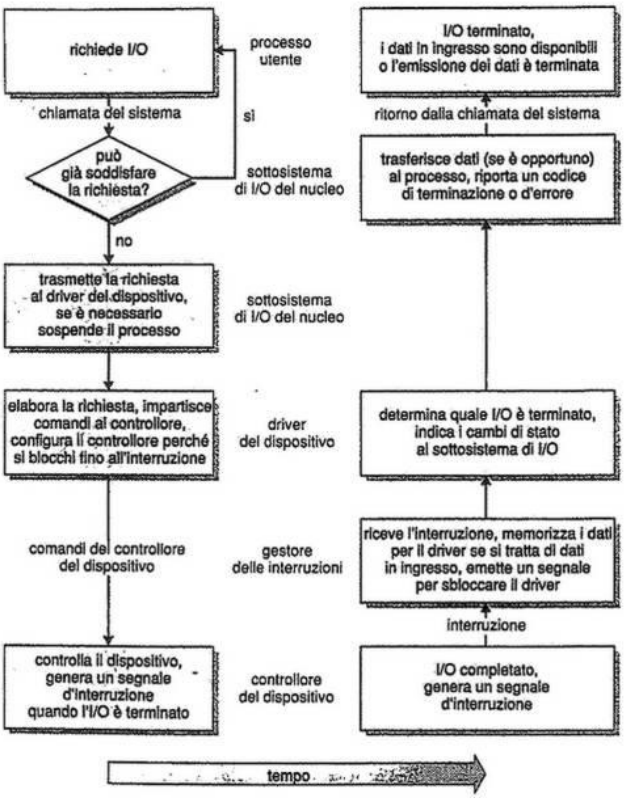
Gestione degli errori

Esistono molti tipi di errori hardware ed i sistemi operativi possono (e devono) far fronte alla maggior parte di essi in modo efficace attraverso opportune procedure di segnalazione dell'errore. Queste fanno uso di bit di informazione o di variabili integrali addizionali, che se ben realizzate potranno dare un grado di dettaglio dell'errore sorprendentemente alto.

Strutture dati del kernel

Il mantenimento continuo di informazioni di stato per l'uso di componenti di I/O da parte del kernel è realizzato attraverso un certo numero di strutture dati interne ad esso, come ad esempio la tabella dei file aperti.

Trasformazione dell'I/O in operazioni hardware



Le prestazioni

La gestione degli I/O è il fattore di maggiore importanza per le prestazioni di un sistema in quanto sottopone la CPU a pesanti requisiti: eseguire il codice dei driver delle periferiche, schedulare i processi, effettuare i cambi di contesto dovuti agli interrupt, copiare i dati tra memorie cache e buffer.

Ci sono diverse strade per migliorare le prestazioni del sistema aumentando l'efficienza dell'I/O:

- ridurre il numero dei cambi di contesto
- ridurre il numero di copie nel passaggio tra dispositivi e applicazioni
- ridurre la frequenza degli interrupt usando grandi trasferimenti e controller capaci
- aumentare la concorrenza con DMA
- spostare le primitive di gestione delle periferiche nell'hardware, a livello più basso
- equilibrare le prestazioni del sistema, dato che un sovraccarico di un componente comporta il rallentamento **di tutta la baracca**.

Struttura Memorie di Massa

Struttura dei dischi

Nei computer attuali i **dischi** rappresentano la principale memoria secondaria. Sono indirizzati come grandi array monodimensionali di *blocchi logici*, ovvero la più piccola unità di dati trasferibile in lettura o scrittura. Questo array

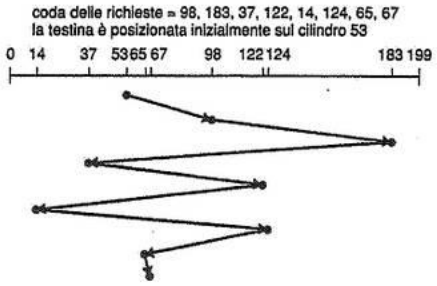
viene mappato nei settori del disco in modo sequenziale, così da poter convertire un indirizzo logico di un blocco nel corrispondente indirizzo fisico.
L'indirizzo fisico è composto da *numero di cilindro*, *numero di traccia* all'interno del cilindro e *numero di settore*all'interno della traccia. In alcuni dispositivi il numero di settori per traccia non è costante perché più la traccia è distante dal centro maggiore è il numero di settori che può contenere. Per mantenere lo stesso tasso di dati scanditi dalla testina si fa in modo di incrementare la velocità di rotazione del disco quando accede alle tracce più interne. Altri dispositivi come i dischi rigidi mantengono invece costante la velocità angolare, ma diminuiscono la densità di bit dall'interno all'esterno.

Schedulazione degli accessi al disco

Per rendere più efficiente l'uso del disco bisogna avere un rapido **tempo di accesso** e una buona **larghezza di banda**. Il tempo di accesso è dato dal *tempo di ricerca* (per muovere la testina fino al cilindro col settore richiesto) e dalla*latenza di rotazione* (che fa in modo che il disco ruoti fino al settore desiderato). La larghezza di banda è invece il numero di byte trasferiti nell'unità di tempo. E' possibile migliorare entrambi questi fattori programmando opportunamente le schedulazioni delle richieste di I/O.
La larghezza di banda è un fattore legato soprattutto all'hardware del sistema, dunque per aumentarne le prestazioni (maggiore ampiezza) bisognerebbe scegliere dischi e interfacce più performanti. I tempi di accesso sono invece migliorabili con la **schedulazione degli accessi**, che si pone come obiettivo quello di fare in modo che il processo ottenga i dati nel minor tempo mediamente possibile.

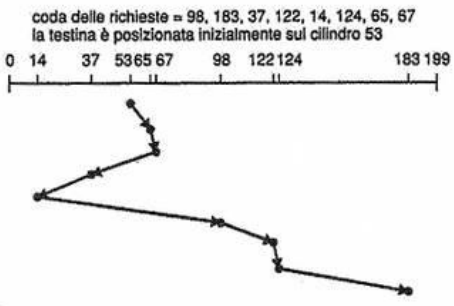
Schedulazione FCFS

La schedulazione **FCFS** (First Come First Served) è la più semplice: "primo arrivato, primo servito". E' indubbiamente l'algoritmo più equo, ma non garantisce affatto le prestazioni migliori.



Schedulazione SSTF

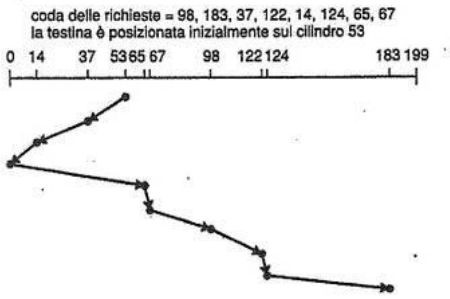
Nella schedulazione **SSTF** (Shortest Seek Time First) si sceglie la richiesta che comporta il minor tempo di ricerca rispetto alla posizione corrente della testina, anche se è stata l'ultima a entrare in coda. Non importa ciò che perde o guadagna un singolo processo, è importante che nel complesso il sistema sia più rapido e lo è di circa tre volte rispetto la FCFS. Poiché la coda viene riempita in continuazione si potrebbero verificare fenomeni di *starvation*.



Abbiamo già detto che la SSTF è migliore del FSFS, ma non è ancora l'algoritmo ottimale.

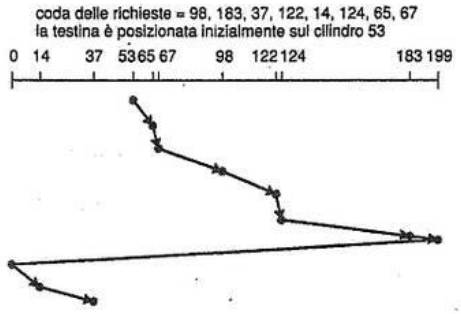
Schedulazione SCAN

Con l'algoritmo di schedulazione **SCAN** viene effettuata una sorta di scansione del disco muovendo il braccio della testina da un'estremità all'altra e servendo tutte le richieste che si incontrano sul percorso. Se dunque durante una scansione entrerà in coda una richiesta ad un cilindro che si trova in quella direzione, verrà presto servita; altrimenti dovrà aspettare il suo ritorno al prossimo giro. La SCAN rende minimo il numero di cambiamenti di direzione della testina, tenendo conto del fatto che essi comportano rallentamenti e usura.
Viene anche chiamato *algoritmo dell'ascensore* perché serve tutte le richieste *in salita* e poi quelle *in discesa*.



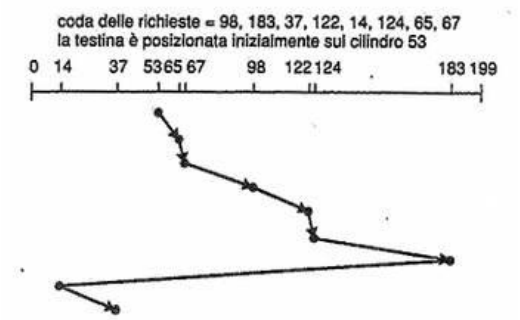
Schedulazione C-SCAN

La schedulazione **C-SCAN** muove la testina da un capo all'altro del disco come una SCAN, ma quando raggiunge un'estremità ritorna a quella opposta senza servire alcuna richiesta durante il rientro. Serve quindi in un'unica direzione, assicurando tempi di attesa più uniformi.



Schedulazione LOOK

La **LOOK** è una versione più efficiente dei due algoritmi precedenti, di cui prende gli stessi principi salvo il fatto che il suo braccio non percorre l'intera larghezza del disco, ma si ferma all'ultima richiesta da servire per ciascuna delle due direzioni. Va da sé che esiste anche una versione **C-LOOK**.



Selezione dell'algoritmo di schedulazione

In generale sia l'SSTF che il LOOK sono una scelta ragionevole come algoritmi predefiniti, perché danno tempi di accesso mediamente bassi. Vanno però considerati altri fattori, tra cui:

- *carico di lavoro*, se è elevato sarebbe preferibile evitare la SSTF che può incorrere facilmente in starvation
- *allocazione dei file*, dato che allocazioni contigue presuppongono movimenti limitati della testina, mentre quelle collegate o indicizzate il contrario
- *posizione della directory e dei blocchi di indici*, sono vicini o no a quelli del file?

Per tutte queste complessità la procedura di schedulazione del disco andrebbe mantenuta separata dal sistema operativo e effettuata direttamente in hardware, dove possono essere più facilmente considerati altri fattori quali il tempo di latenza. Nella pratica però il sistema operativo impone alcuni vincoli anche in questi scenari, intervenendo su queste procedure ad esempio per segnalare delle priorità o per imporre dei vincoli su alcuni ordini di accesso (per prevenire crash del sistema), ecc.

Gli algoritmi visti finora toccano infatti solo l'ordinamento delle richieste e considerano tutti i processi ugualmente importanti, ma per un motivo: politiche di priorità forzerebbero la testina a spostarsi molto di più, il che comporterebbe un aumento dei tempi di accesso globale.

Amministrazione del disco

Configurazione del disco

Per poter memorizzare dati un disco deve essere suddiviso in settori che il controller possa leggere e scrivere. Questo processo prende il nome di **formattazione fisica** o a basso livello, e per ogni settore predispone la seguente struttura dati: *intestazione*, *zona dati* e *terminatore*. Intestazione e terminatore contengono informazioni per il controller, cioè il numero di settore e la parola di verifica. Questa è scritta in un codice di correzione degli errori, quindi non solo è in grado di rilevare i settori corrotti ma anche di ripristinarli.
Per memorizzare i file su disco bisogna a questo punto *partizionarlo* in uno o più insiemi di cilindri detti *partizioni*, ognuno indipendente dall'altro.
Infine si procede con una **formattazione logica** che provveda alla creazione di un file system.
Nel caso in cui non si voglia formattare alcun file system, la partizione viene lasciata come grande array di blocchi sequenziali e assume il nome di *disco grezzo*. Il suo principale utilizzo è per l'area di **swap**.

Il blocco di avvio

Il **bootstrap** è un programma da eseguire a ogni avvio del sistema e ha il compito di inizializzarne tutte le funzioni, quindi i registri della CPU, i controller di periferica ed il contenuto della memoria centrale. In altre parole è preposto all'avvio del sistema operativo.
In molti computer risiede su memorie ROM (non volatili e di sola lettura), che in alcuni casi contengono un *loader* che referenzia una partizione di avvio interamente caricata su disco, chiamata **disco di boot** o *di sistema*. Questa seconda modalità d'avvio è sicuramente più sofisticata di quella interamente memorizzata su ROM.

Blocchi difettosi

I dischi sono dispositivi in continuo movimento e con meccanismi molto delicati, dunque sono particolarmente soggetti a guasti. Su dischi semplici la gestione dei *blocchi difettosi* (*bad block*) prevede il loro isolamento con una marcatura nella tabella FAT, e conseguente perdita dei dati se erano in uso.
Dischi più sofisticati hanno tecniche di gestione più complesse. Anzitutto viene mantenuta una lista dei blocchi guasti costantemente aggiornata direttamente su disco, dopodiché sono previste tutta una serie di contromisure hardware e software. Vengono ad esempio riservati un certo numero di settori aggiuntivi di scorta (*sector sparing*), utilizzabili esclusivamente per sostituire blocchi difettosi e previsti per ogni cilindro così da mantenere i vantaggi della schedulazione. Un altro sistema è quello dello slittamento di settore (*sector slipping*), che prevede il rimappaggio di tutta una serie di settori traslandoli di un'unità.
L'intervento manuale è richiesto comunque poiché si incorre spesso in una perdita di dati.

Gestione dello spazio di swap

La gestione dello spazio di **swap** ha come ovvio obiettivo quello di gestirlo al meglio, compito non banale dato che gli accessi al disco sono di gran lunga le operazioni più lente del sistema.

Uso dello spazio di swap

L'uso dello spazio di swap dipende dal sistema operativo installato e dagli algoritmi di gestione della memoria che implementa. Ad esempio può essere usato per conservare le immagini dei processi (dati e codice compreso), o per immagazzinare le pagine tolte dalla memoria centrale.
Possono essere riservati fino ad alcuni gigabyte di disco per lo swap, ed è sempre meglio sovrastimarlo piuttosto che vedersi abortire alcuni processi perché non hanno abbastanza spazio.

Locazione dello spazio di swap

Lo spazio di swap può risiedere in due posti:

- in un file, quindi gestito attraverso le normali funzioni del file system. Tra i suoi svantaggi ha la frammentazione esterna e un tempo di gestione elevato, colpa della necessità di continuare a scandire la struttura directory e dati
- avere una partizione separata dal disco, senza alcun file system (grezza). Quello che si perde in efficienza di salvataggio e in aumento della frammentazione interna, lo si guadagna enormemente in velocità.

La seconda via è quella più saggia dato che nell'area di swap ci vanno generalmente solo dati con un periodo di vita molto breve e su cui avvengono richieste d'accesso molto frequenti; la velocità in questi casi è il fattore da perseguire. In alcuni sistemi operativi sono implementati entrambi.

La struttura RAID

Con la riduzione delle dimensioni delle unità disco è diventato possibile collegarne più di una insieme, così da sfruttare le maggiori velocità se fatte lavorare in parallelo o per migliorare l'affidabilità memorizzando informazioni ridondanti. Esistono tutta una serie di tecniche di organizzazione dei dischi che prendono il nome di **RAID**, ovvero array ridondante di dischi a basso costo o indipendenti. La struttura RAID può essere definita come gruppo di dischi fisici

gestiti in modo integrato così da apparire un unico disco con caratteristiche elevate di tolleranza ai guasti o di prestazioni.

Il miglioramento dell'affidabilità mediante la ridondanza

Consideriamo l'affidabilità e osserviamo che la possibilità che un certo disco in un insieme di dischi si guasti è molto maggiore rispetto alla probabilità che non ne funzioni più uno indipendente. Tale problema può essere superato con la *ridondanza*, cioè con la memorizzazione di informazioni supplementari normalmente non necessarie ma utili in caso di guasto.

Il metodo più semplice per introdurre la ridondanza è duplicare ogni disco con una tecnica chiamata **mirroring**, creando un disco logico che è in realtà costituito da due dischi fisici identici. Ogni scrittura deve avvenire su entrambi, possibilmente in tempi diversi come precauzione contro l'incoerenza.

Con una ridondanza del 100% dello spazio a disposizione protegge con efficacia dai guasti singoli, ma una volta ripristinato un blocco danneggiato questo rimane "scoperto".

Il tempo medio al guasto di un disco in mirroring dipende sia da quello dei singoli dischi che dai tempi medi di riparazione, ma è comunque molto alto.

Il miglioramento delle prestazioni con il parallelismo

Il mirroring dei dati aumenta l'affidabilità a scapito delle prestazioni dato che le operazioni per unità di tempo sono raddoppiate. Per aumentare la velocità di trasferimento si può invece adottare la tecnica del **data-striping**, che consiste nel dividere i bit di ogni byte su dischi multipli (*bit-level striping*). Se ad esempio abbiamo 8 dischi e distribuiamo gli 8 bit che compongono un byte a ciascuno di essi, avremo un disco logico 8 volte più grande di quello fisico e soprattutto con velocità di accesso 8 volte maggiori!

Questa tecnica - che realizza di fatto il **parallelismo** - può essere generalizzata a un qualunque multiplo o sottomultiplo di 8 dischi, e non si limita esclusivamente allo spezzettamento del byte ma anche a quello dei blocchi (*block-level striping*).

Gli obiettivi del parallelismo sono dunque quelli di aumentare il rendimento di piccoli accessi multipli equilibrando il carico, e ridurre il tempo di risposta per gli accessi di maggiore entità.

I livelli di RAID

Esistono diverse configurazioni di dischi che cercano questo o quel compromesso tra il fattore affidabilità e quello delle prestazioni, e prendono il nome di **livelli di RAID**. Elenchiamoli:

- **RAID livello 0**: array di dischi con spezzettamento dei blocchi ma nessuna ridondanza. Altissime prestazioni ma nessuna protezione contro i guasti
- **RAID livello 1**: mirroring
- **RAID livello 2** o di *organizzazione a codice di correzione di errore (ECC)*:
  si basa sul bit di parità per il rilevamento di errori a singoli bit, mentre utilizza bit supplementari (mantenuti su dischi separati) negli schemi di correzione degli errori per ricostruirli. Per quattro dischi ne sono necessari altri tre per realizzare questo schema. Alte prestazioni ottenute tramite bit-level striping
- **RAID livello 3** o di *organizzazione di parità a bit alternati*:
  si basa sul fatto che i controllori dei dischi possono individuare se un settore è danneggiato, e quindi si può ricostruire il dato danneggiato controllando la parità dei restanti dischi. Migliora il livello 2 dato che è sufficiente un unico disco supplementare a parità di efficacia.
  Rispetto ai livelli senza parità, il livello 3 supporta un numero più basso di I/O al secondo perché ogni disco deve partecipare a ogni richiesta, ed è inoltre più lento perché parte del tempo viene speso per il controllo della parità. Le prestazioni possono essere migliorate introducendo un controller hardware dedicato alla parità e utilizzando una cache per memorizzare i blocchi mentre questa viene calcolata
- **RAID livello 4** o di *organizzazione di parità a blocchi alternati*:
  usa una suddivisione a livello di blocco come nel RAID 0 ma ne mantiene in più uno di parità su un disco separato. Se un disco viene a mancare, il blocco di parità in collaborazione con gli altri blocchi corrispondenti degli altri dischi è in grado di ripristinarlo. La lettura di un blocco accede solo a un disco, mantenendo gli altri liberi per altre richieste; questo rallenta la velocità di trasferimento per ogni accesso ma consente accessi multipli in parallelo. La velocità generale di I/O è quindi elevata, escludendo i casi di accesso a porzioni di dati inferiori ad un blocco
- **RAID livello 5** o di *parità distribuita a blocchi alternati*:
  distribuisce i dati e le parità tra tutti i dischi invece che mantenere queste ultime in un disco separato. Per

ogni blocco c'è quindi un disco che memorizza la parità e gli altri i dati. In questo modo si evita l'uso eccessivo del disco di singola parità in caso di guasto

- **RAID livello 6** o *schema di ridondanza P+Q*:
  è simile al RAID 5 ma memorizza ulteriori informazioni ridondanti così da essere in grado di fronteggiare guasti contemporanei di più dischi. Non usa la parità ma particolari codici correttori degli errori noti come di Reed-Solomon
- **RAID livello 0+1**:
  combina il livello 0 e 1 e i rispettivi benefici. E' composto da due serie di dischi in bit-level striping messi in mirroring tra loro. Esiste anche il livello **1+0** costituito da dischi in mirroring messi in data-striping con altre coppie. Il vantaggio teorico di quest'ultimo è che il guasto su un disco non comporta l'inaccessibilità di tutta la sua suddivisione, ma unicamente la sua coppia in mirroring

Scelta di un livello di RAID

Come scegliere il livello di RAID più adeguato alle proprie necessità? Se ciò che interessano sono le prestazioni e la perdita dei dati non è un fattore critico, allora la scelta è ovvia: RAID 0. Se servono invece alte affidabilità e recuperi veloci, si punti sul RAID 1. Se si vuole un compromesso tra i due fattori si scelga tra RAID 0+1 o 1+0, o in alternativa RAID 5 se si vuole usare un numero minore di dischi; il RAID 6 sarebbe più efficiente, ma non è ancora pienamente supportato.

Ultima nota sul RAID è che è possibile predisporre dischi di ricambio (*hot spare*) non utilizzati per i dati ma configurati in modo da rimpiazzare immediatamente un disco guasto.

Collegamento dei dischi

I computer possono accedere in due modi alla memoria su disco: mediante la *memorizzazione con collegamento all' host* o con quella *a collegamento di rete*.

Memorizzazione con collegamento all'host

A questo tipo di memorizzazione si accede tramite le porte locali di I/O, che possono avere diverse architetture (ad esempio IDE, ATA o SCSI e FC). I comandi I/O che danno inizio ai trasferimento dei dati sono letture e scritture di blocchi logici di dati diretti a ben precise unità di memorizzazione.

Memorizzazione con collegamento di rete

In questi sistemi si accede a distanza su una rete di dati con un'interfaccia di chiamata di procedura remota (RPC), ad esempio NFS in UNIX o CIFS in Windows. Questo collegamento consente ai computer di una LAN di condividere in modo conveniente periferiche di memorizzazione, anche se sicuramente con prestazioni inferiori rispetto ai collegamenti diretti.

Struttura di memoria terziaria

Periferiche di archiviazione terziaria

Il basso costo è la caratteristica principale di una memorizzazione **terziaria**, che viene tipicamente effettuata su supporti rimovibili.

Dischi rimovibili

Ne abbiamo di diversi tipi:

- dischi magnetici (ad esempio i floppy), caratterizzati da alta velocità di accesso ma maggior rischio di danni
- dischi magneto-ottici, che usano tecniche sia magnetiche che ottiche (quindi anche laser). Sono più resistenti alla rottura delle testine
- dischi ottici, tra cui dischi a sola lettura o riscrivibili (anche detti a cambiamento di fase)

I nastri

I nastri contengono più dati di un disco ottico o magnetico e sono più economici a parità di dimensione. Sono particolarmente adatti all'accesso sequenziale e vengono spesso utilizzati per copie di backup e non per accessi diretti.

Le tecnologie future

Vediamone alcune:

- memorizzazione olografica: usa la luce laser per registrare fotografie olografiche da milioni di bit su speciali dispositivi. Ha una velocità di trasferimento altissima e sta diventando sempre più affidabile
- sistemi meccanici microelettronici: potrebbero fornire una tecnologia di memorizzazione non volatile dei dati più veloce del disco magnetico e più economica della DRAM

Compiti del sistema operativo

Due importanti compiti di un sistema operativo sono la gestione dei dispositivi fisici e la presentazione alle applicazioni di un'astrazione di macchine virtuale. Vediamo come vengono gestite sui supporti rimovibili:

Interfaccia dell'applicazione

La maggior parte dei sistemi operativi gestisce i dischi rimovibili come se fossero fissi, dunque possono essere sia formattati con un file system sia gestiti in modo grezzo. Può avvenire un uso condiviso dei propri dati tra più processi attivi, e le modalità d'accesso possono essere sia dirette che sequenziali.

Con le unità nastro l'interfaccia dell'applicazione cambia: vengono utilizzati solo in modo grezzo e il loro usufrutto è esclusivo di un solo processo alla volta. L'accesso è strettamente sequenziale. In linea di principio è possibile implementare un file system su nastro, ma con strutture dati e algoritmi completamente differenti da quelli visti finora.

Nomina dei file

Uno dei maggiori problemi di gestione della memoria terziaria è trovare una denominazione univoca dei file nei supporti rimovibili, che includa un identificativo del dispositivo e che garantisca inoltre la portabilità. Non esiste una soluzione omogenea per tutti i sistemi operativi, salvo per alcuni supporti di memorizzazione standardizzati come i CD e i DVD.

Gestione della memorizzazione gerarchica

La gestione della memorizzazione gerarchica aumenta il grado di astrazione del sistema ed estende la gerarchia di memorizzazione oltre la memoria centrale secondaria per incorporare la terziaria. Fa uso di *jukebox robotizzati* per la sostituzione dei supporti senza intervento umano, e viene utilizzato solitamente nei centri di calcolo con supercomputer e in altre grandi installazioni con enormi quantità di dati.

Struttura Sistemi Distribuiti

Concetti di base

Un **sistema distribuito** è un insieme di sistemi di elaborazione che non condividono memorie o orologi e che sono connessi tra loro mediante una rete di comunicazione. Ognuno dei singoli sistemi può essere chiamato con molti nomi, ad esempio **host** o **macchina**, e mentre le sue risorse interne sono definite *locali*, quelle che può raggiungere attraverso la rete sono dette *remote*.

Obiettivo dei sistemi distribuiti è fornire un ambiente efficace e conveniente per la condivisione di risorse tra macchine diverse.

Vantaggi dei sistemi distribuiti

I principali motivi per costruire sistemi distribuiti sono:

- *condivisione delle risorse*: ogni utente è in grado di usare le risorse (file, periferiche, potenza di calcolo) disponibili su macchine remote
- *velocità di calcolo*: può aumentare se l'elaborazione viene suddivisa ed eseguita in modo concorrente su più host. Si può condividere anche il carico di lavoro, distribuendolo su più macchine così da evitare sovraccarichi
- *affidabilità*: il guasto di un host generalmente non compromette il funzionamento dell'intero sistema. Certo deve essere progettato con intelligenza, con ridondanze adeguate sia hardware che dei dati
- *comunicazione*: molto simile a quella che avviene fra i processi all'interno di un singolo sistema, così che possa essere facilmente estesa all'ambiente distribuito
- *scalabilità*: capacità del sistema di adattarsi al variare del carico di servizio

Tutti questi fattori hanno fatto sì che le industrie si orientassero verso i sistemi distribuiti e allo stesso tempo verso un *downsizing* dei sistemi di calcolo, dal momento che reti di macchine a medio-basse prestazioni hanno nell'insieme ottime funzionalità e costi ridotti.

Tipi di sistemi distribuiti

Esistono due grandi categorie di sistemi orientati alle reti e li vedremo nei prossimi capitoli.

Sistemi operativi di rete

Un **sistema operativo di rete** fornisce un ambiente in cui gli utenti, che sono consapevoli della molteplicità delle macchine, possono accedere a risorse remote.

Tre importanti funzioni di questo sistema sono:

- *sessioni di lavoro remote*, ovvero la possibilità per un utente di collegarsi ad un'altra macchina in remoto (previa login) e interagire con essa come se fosse in locale. E' ciò che fa il programma telnet per Internet

- *trasferimento remoto di file* tra host, con una copia esplicita dal file system remoto a quello locale. In internet ciò è reso possibile dal protocollo ftp
- attivare procedure remote attraverso le *Remote Procedure Calls* (*RPC*)

Notare che ogni computer *conserva il proprio file system locale* e (molto importante) la posizione dei file *non è trasparente* agli utenti, che devono sapere esattamente dove si trovano. Abbiamo già detto infatti che non avviene una vera condivisione di dati: l'utente può solo copiarli da un host all'altro.

Sistemi operativi distribuiti

In un **sistema operativo distribuito** gli utenti accedono alle risorse remote nello stesso modo in cui accedono a quelle locali, e la migrazione dei processi e dei dati da un host all'altro avviene sotto il controllo del sistema operativo distribuito.

La *migrazione dei dati* può avvenire in due modi:

1. quando un utente richiede un file su una macchina remota, questo viene interamente trasferito così da potervi accedere in locale. A elaborazione compiuta, se è stato modificato, viene interamente ricopiato nella macchina di origine
2. simile al precedente, ma a differenza dell'intero file vengono trasferite solo le parti effettivamente necessarie al processo in quel preciso momento

In entrambi i metodi il sistema non si limita a trasferire, ma fa anche le traduzioni opportune se le rappresentazioni dei dati (big endian, little endian) non sono compatibili.

Qual è la tecnica migliore? Dipende dalle porzioni di file che vengono mediamente richieste.

In alcuni casi è preferibile *trasferire la computazione* piuttosto che i dati, scelta particolarmente saggia se il tempo di trasferimento è più lungo del tempo di esecuzione di un comando remoto. Il tutto può essere realizzato con una RPC o con l'attivazione di un processo remoto per mezzo di scambio di messaggi.

Una logica estensione della *migrazione della computazione* è quella dei processi, dove l'esecuzione di questi ultimi non avviene interamente nell'host in cui sono stati attivati. Ci sono due tecniche per spostarli, una che prevede la migrazione in modo totalmente trasparente all'utente (più rapida su sistemi omogenei), l'altra che invece permette o richiede le modalità di migrazione (più portabile e configurabile). Alcuni motivi per la migrazione dei processi sono il bilanciamento del carico, una maggiore velocità di calcolo, maggiore compatibilità dell'hardware o del software, accesso ai dati, ecc.

Topologia

I vari host in un sistema distribuito possono essere connessi fisicamente in molti modi diversi, e le configurazioni ottenute si possono valutare in base al costo dell'installazione e della comunicazione (tempo + soldi) e dalla disponibilità, ovvero dall'ampiezza della zona in cui si può accedere ai dati malgrado il guasto di alcuni collegamenti o macchine.

Le reti completamente connesse sono molto performanti e affidabili, ma hanno costi d'installazione spesso insostenibili; gli si preferisce quelle parzialmente connesse, nelle quali vanno però fornite informazioni d'instradamento a quegli host che non sono direttamente collegati.

La scelta della configurazione migliore dipende dal sistema che si vuole ottenere e dalla situazione ambientale, non esiste una topologia migliore in assoluto.

Comunicazione

Attribuzione e risoluzione dei nomi

Per identificare in modo certo i siti di una rete e i loro processi è necessario dare loro un nome univoco. I processi su sistemi remoti vengono usualmente identificati dalla coppia <nome host, identificatore>, dove nome host è unico nella rete e l'identificatore è il comune *id* del processo.

La risoluzione dei nomi deve tradurre il nome di una macchina in un valore numerico che descriva il sistema di destinazione all'hardware della rete. Ciò avviene o per mezzo di una lista mantenuta in locale da ogni host che riporta tutte le corrispondenze tra nomi e indirizzi (soluzione onerosa), o distribuendo tali informazioni tra i computer della rete ed utilizzando un protocollo per la risoluzione. Questo secondo sistema è quello del *server dei nomi del dominio*(*DNS*), reso nel tempo sempre più efficiente e sicuro grazie a numerosi raffinamenti.

Strategie di instradamento

Se in un sistema distribuito esistono più percorsi per collegare due host, allora ci sono diverse opzioni di instradamento. Ogni sito ha una *tabella di instradamento* mantenuta aggiornata che indica i vari percorsi alternativi per la trasmissione di un messaggio, e che può contenere informazioni aggiuntive come velocità e costi.

Sono tre gli schemi di instradamento più comuni:

- **instradamento fisso**: un percorso tra due macchine viene determinato a priori e non cambia a meno di guasti
- **instradamento virtuale**: viene definito un percorso tra due host valido per tutta la durata di una sessione
- **instradamento dinamico**: il percorso da impiegare per inviare un messaggio viene scelto al momento dell'invio

L'instradamento dinamico è il più complicato da organizzare e gestire, ma è il più efficace in ambienti complessi data l'adattabilità alle variazioni di carico e di gestione dei guasti.
Il **gateway** è un dispositivo hardware dedicato che collega la rete locale ad altre reti, gestendo l'eventuale cambiamento di protocollo di comunicazione. Gli host si collegano al gateway con instradamento statico, mentre quest'ultimo si collega alla rete con schema dinamico. Un altro dispositivo molto importante in una rete è il **router**, responsabile dell'instradamento dei messaggi tra due reti.

Strategie di pacchetto

I messaggi di per sé non hanno una lunghezza standard, ma per semplificare la progettazione del sistema si preferisce realizzare comunicazioni con messaggi a dimensione fissa chiamati **pacchetti** (o *frame* o *datagram*). La comunicazione implementata a pacchetti può avvenire mediante messaggi senza connessione inaffidabili (senza garanzia di ricezione, esempio l'UDP) o affidabili (esempio TCP).

Strategie di connessione

Una volta che i messaggi sono in grado di arrivare a destinazione i processi possono istituire delle **sessioni di comunicazione** per scambiarsi le informazioni. Ne vengono comunemente usate di tre tipi:

- **commutazione di circuito**, con un collegamento fisico permanente che perdura per tutta la comunicazione e che non può essere usato da altri processi. Richiede alti tempi di attivazione e può sprecare molta larghezza di banda
- **commutazione di messaggi**, in cui si stabilisce un collegamento temporaneo tra due processi che termina col completamento del trasferimento. I collegamenti fisici vengono allocati dinamicamente per brevi periodi in base alle necessità
- **commutazione di pacchetto**, dove ciascun pacchetto di un messaggio è inviato separatamente su una connessione attivata dinamicamente. E' la tecnica che fa il miglior uso della larghezza di banda e per questo è tra le più utilizzate. Tuttavia ha un alto overhead per ogni messaggio, dato che deve contenere diverse informazioni per essere suddiviso in pacchetti e correttamente riassemblato

Gestione dei conflitti

Le **collisioni** in un sistema distribuito avvengono quando a un host arrivano contemporaneamente due messaggi, che diventano entrambi indecifrabili. Esistono molte tecniche per evitarlo, tra cui ricordiamo:

- il rilevamento della portante ad accesso multiplo (CSMA) unito al rilevamento delle collisioni (CD). In breve, prima di trasmettere un messaggio l'host verifica che la linea di comunicazione non sia usata da altri; se si rileva una collisione si interrompe la trasmissione e la si ritenta dopo un intervallo di tempo casuale
- passaggio di token per imporre la turnazione, adoperato nei sistemi con struttura ad anello. La macchina può inviare un messaggio solo se è in possesso del token, che viene passato tra i vari host a intervalli regolari

Protocolli di comunicazione

Nella progettazione di una rete di comunicazione i fattori critici da tenere in considerazione sono le *comunicazioni asincrone*, le *interazioni tra ambienti non omogenei* e la *probabilità di avere errori*. L'obiettivo che ci si prefigge è dunque creare un ambiente omogeneo di comunicazione che la astragga e virtualizzi così da semplificare la progettazione e avere una gestione efficiente del sistema.
La soluzione è l'adozione di protocolli di comunicazione da applicare a strati distinti del sistema, utilizzati un po' come se fossero i driver della rete. La comunicazione avviene ovviamente solo tra strati equivalenti.
Il modello teorico standard è l'ISO/OSI, che divide il sistema in:

- **strato fisico**, responsabile della definizione dei dettagli elettromeccanici della trasmissione fisica. Implementato nell'hardware
- **strato di collegamento dei dati**, che gestisce la trasmissione dei pacchetti e rileva e corregge gli errori
- **strato di rete**, che si occupa delle connessioni e dell'instradamento dei pacchetti
- **strato di trasporto**, che partiziona e mantiene ordinati i pacchetti di un messaggio, ne controlla il flusso e gestisce gli errori
- **strato di sessione**, che realizza le sessioni e i protocolli di comunicazione tra processi

- **strato di presentazione**, che risolve le differenze di formato tra le varie macchine
- **strato di applicazione**, responsabile dell'interazione con gli utenti

Questo è un buon modello, ma storicamente si è preferito adottarne uno più semplice ed efficiente: il **TCP/IP**, detto anche *modello reale*. Il protocollo TCP corrisponde allo strato di trasporto, è orientato alle connessioni ed è affidabile (l'opposto dell'UDP). Il protocollo IP è invece responsabile dell'instradamento dei pacchetti attraverso Internet.

Robustezza

In un sistema distribuito sono molti i tipi di guasti in cui si può incorrere, ma per garantire la **robustezza** bisogna comunque individuarli, riconfigurare il sistema per poter continuare la computazione e infine recuperare la configurazione iniziale a guasto riparato.

Individuazione dei guasti

In un ambiente senza memoria condivisa è di solito impossibile distinguere le tipologie di guasto, se hardware o software ad esempio, ma ci si deve accontentare di rilevarli (che è già tanto). Tra le tecniche di rilevamento le più utilizzate sono il *monitoraggio periodico con handshaking*, l'impostazione di un *time-out* (superato il quale viene sollevato un errore) e la *computazione duplicata con confronto dei risultati* (che oltre a rilevare il guasto offre una controprova della correttezza dei risultati).

Riconfigurazione del sistema

Una volta rilevato che un certo host è guasto, bisogna anzitutto aggiornare tutte le tabelle di instradamento della rete affinché la comunicazione non passi da lì. Bisogna poi fare in modo che siano tutti informati del fatto che i servizi erogati da quella macchina non sono temporaneamente disponibili, e anzi bisogna cercare di far migrare tali servizi su altri siti.

Recupero di un guasto

A guasto riparato la macchina deve essere reintegrata nel sistema in modo che gli altri host possano essere informati (tramite handshaking) del suo ripristino. Le tabelle interne della macchina verranno quindi ripristinate e aggiornate, e gli verranno recapitati i messaggi pendenti.

Problemi progettuali

L'obiettivo principale di un progettista di sistemi distribuiti è renderli del tutto trasparenti agli utenti, dando l'illusione che stiano operando su un sistema centralizzato convenzionale.

Altri aspetti che vanno tenuti in considerazione sono:

- la *mobilità dell'utente*, che non deve essere obbligato a collegarsi a un unico host specifico
- la *tolleranza ai guasti*, ovvero il sistema dovrebbe continuare a funzionare anche in caso di guasto - anche se con funzionalità in meno - fino alla riparazione
- la *scalabilità*, ovvero la capacità del sistema di adattarsi all'aumento del carico di servizio

File System Distribuiti

L'ambiente distribuito

Nel capitolo precedente abbiamo definito i **sistemi distribuiti** come un insieme di computer debolmente collegati da una rete di comunicazione. Un **file system distribuito** (**DFS**) è un'implementazione distribuita di quello classico centralizzato, ed è caratterizzato dalle seguenti entità:

- **servizio**, un programma in esecuzione su una o più macchine che fornisce ad altre (non note a priori) le sue funzionalità
- **server**, singola macchina che offre un servizio
- **client**, host che richiede servizi a macchine remote

Un DFS può quindi essere considerato come un *file server*, poiché fornisce attraverso la rete i servizi di accesso e uso dei file ai client. Notare che le macchine possono essere indistintamente sia server che client, l'importante è che il sistema appaia come dotato di un unico file system centralizzato; la dispersione di servizi e risorse deve essere del tutto trasparente agli utenti.

Teoricamente andrebbe esteso il principio di trasparenza anche alla rapidità di accesso ai servizi, ma questo è un obiettivo più difficile da garantire a causa dei tempi e delle esigenze computazionali della comunicazione lungo una rete.

Il DFS controlla un gran numero di dispositivi di memorizzazione, in ognuno dei quali si possono individuare le **unità componenti** del file system distribuito, ovvero il più piccolo gruppo di file che si possa memorizzare su una macchina e che deve risiedere esclusivamente in essa.

Attribuzione dei nomi e trasparenza

L'attribuzione dei nomi su disco locale fornisce agli utenti un'astrazione del file che nasconde i dettagli di come e dove è stato memorizzato. In un DFS trasparente viene aggiunto un nuovo livello di astrazione nascondendo anche l'host su cui è localizzato in rete.

Le strutture di attribuzione dei nomi

In un DFS realmente trasparente dovrebbe essere garantita sia la **trasparenza della locazione** (non si hanno informazioni sull'effettiva locazione fisica del file) che l' **indipendenza della locazione** (il nome di un file non deve cambiare nemmeno se viene fisicamente spostato). Quest'ultimo requisito non è spesso assicurato poiché sono pochi i sistemi che supportano una reale *migrazione dei file*; si parla in questi casi di *file system di rete*.

Su file system realmente distribuiti (come ad esempio l'AFS) potremmo avere client senza dischi che fanno intero affidamento al server sia per accedere ai dati che per avviare il sistema operativo (previa esecuzione di un protocollo speciale memorizzato su ROM). In questo modo si risparmia sì sui dischi, ma a prezzo di una maggiore complessità di avvio e di prestazioni inferiori rispetto ai mappaggi locali.

Gli schemi di attribuzione dei nomi

Per l'attribuzione dei nomi abbiamo tre approcci principali, i primi due per file system di rete e l'ultimo per il DFS:

- identificare il file come nome macchina + nome file in locale. E' il metodo più semplice ma non è né trasparente né indipendente alla locazione
- montaggio del file system remoto sul direttorio locale attraverso un *automount*, così che il nome del file venga calcolato nel file system della macchina. Questa struttura è versatile ma complessa da gestire, e determina percorsi diversi del file a seconda della macchina
- (DFS) esiste una singola struttura globale dei nomi che genera tutti i file del sistema distribuito, che riescono così ad avere un nome univoco. E' uno schema difficile da realizzare a causa dei molti file speciali che devono rimanere specifici per singole macchine

Tecniche di implementazione

L'implementazione dell'attribuzione trasparente dei nomi richiede una mappatura tra il nome del file e la locazione associata; per mantenere più semplice la gestione, i file vengono spesso aggregati in *unità componenti* e trattati in quanto tali.

Per aumentare la disponibilità delle informazioni cruciali per la mappatura vengono adottate tecniche come la *replica* e la *cache locale* (o entrambe), che possono essere migliorate con l'uso di identificatori di file indipendenti dalla locazione per essere costantemente aggiornate.

Accesso remoto ai file

Quando un utente chiede di accedere a un file remoto in un file system di rete, il trasferimento dei dati richiesto avviene con l'uso di un meccanismo di servizio remoto generalmente implementato con una RPC. Tale sistema prevede la copiatura locale in cache e il conseguente aggiornamento, quindi il salvataggio in remoto.

In un DFS queste operazioni vengono trattate mediante meccanismi di servizi locali, eventualmente eseguiti in remoto in modo trasparente.

Schema di base per l'uso della cache

L'uso della cache assicura prestazioni ragionevoli riducendo sia il traffico di rete che gli accessi I/O al disco. Il concetto è che se i dati richiesti non sono già al suo interno, allora bisogna ricopiarli dagli originali contenuti sul server. Si cerca cioè di mantenere in queste memorie veloci le informazioni richieste più spesso, accelerando di conseguenza i tempi di servizio.

E' di fondamentale importanza garantire la coerenza della cache in caso di modifica dei dati, o potrebbero avvenire accessi a dati inconsistenti. Ne vanno perciò studiate la locazione e le politiche di aggiornamento più appropriate, e non ultimo vanno valutate attentamente le sue dimensioni: più la cache è grande più RAM viene richiesta e maggiori sono gli oneri per garantirne la coerenza, ma allo stesso tempo maggiore è la probabilità che contenga il file di interesse.

Locazione della cache

Dove memorizzare la cache? Se viene collocata su disco avremo maggiore affidabilità e dimensione e memorizzazioni non volatili, se invece viene messa su RAM diventa molto più veloce e può essere implementata su client senza memorie secondarie.

Politica di aggiornamento della cache

Le *politiche di aggiornamento della cache* indicano come comportarsi in seguito a modifiche sulle copie, ed hanno un effetto critico sulle prestazioni e affidabilità del sistema.
La politica più semplice è la **scrittura immediata** (write-through), in cui la scrittura avviene contemporaneamente su cache e su disco. E' estremamente affidabile ma ha prestazioni in scrittura molto basse.
Un'alternativa è la **scrittura ritardata**, in cui vengono disaccoppiate le scritture rapide su cache e quelle lente in memoria di massa. Le scritture diventano così molto più veloci a prezzo di una minore affidabilità.
Una variante dell'ultima politica è la **scrittura in chiusura** che ricopia il file dalla cache al disco solo quando viene chiuso. E' una strategia vantaggiosa se i file vengono mantenuti aperti a lungo e aggiornati di frequente.

Coerenza

Abbiamo due metodi per verificare la validità dei dati in cache. Il primo è la **verifica iniziata dal client**, che ad ogni accesso o a intervalli di tempo regolari controlla che i dati locali siano coerenti con quelli principali sul server. Il secondo è la **verifica iniziata dal server**, che registra i file che vengono posti in cache dai vari client, e se le modalità d'accesso richieste potrebbero minare la coerenza (ad esempio la scrittura da parte di almeno uno dei due) vengono fatti partire i controlli.

Confronto tra i servizi con cache e quelli remoti

Vediamo i pro e i contro dell'utilizzo della cache:

- *pro*: supporta una gestione efficace di un numero notevole di accessi remoti, ha maggiore rapidità prestazionale, comporta sovraccarichi minori della rete se gestisce grosse porzioni di dati
- *contro*: garantire la coerenza è un compito piuttosto oneroso soprattutto se vengono effettuate molte scritture, sono necessarie memorie centrali e dischi di grandi dimensioni o l'efficacia andrebbe persa

File server con e senza stato

Lo **stato di un file server** è l'insieme delle informazioni che caratterizzano l'uso di un file aperto. In un file con stato il client per accedere a un file deve effettuare prima una open(), dopodiché riceve dal server un identificatore univoco di connessione che viene mantenuto per tutta la sessione. Ha prestazioni maggiori grazie al mantenimento delle informazioni in memoria centrale, facilmente accessibili grazie agli identificatori.
Un file server senza stato evita di richiedere e mantenere informazioni di stato, quindi soddisfa ogni richiesta in modo indipendente. Ha prestazioni minori ma è più semplice da realizzare e reagisce meglio ai guasti.

Replica dei file

La **replica** dei file su diverse macchine è una ridondanza utile sia per aumentare la disponibilità del servizio che le prestazioni, dato che selezionare una replica in un server vicino per soddisfare una richiesta di accesso comporta un tempo di servizio più breve.
La gestione della replica non è indipendente alla locazione perché la disponibilità di una di esse non può e non deve essere influenzata dalle altre. Inoltre, i suoi dettagli dovrebbero essere mantenuti trasparenti agli utenti di alto livello, o diventerebbe più problematico garantirne la consistenza.
Il problema principale delle repliche è legato al loro aggiornamento, che in teoria dovrebbe riflettersi automaticamente su tutte le istanze per mantenerle coerenti, ma che difficilmente avviene perché comporta una notevole riduzione delle prestazioni.

Coordinamento Distribuito

Ordinamento degli eventi

A differenza di un sistema centralizzato che ha un'unica memoria centrale e un unico orologio di sistema, in un sistema distribuito diventa estremamente complesso definire l'ordine con cui sono avvenuti due eventi. Dato però che la relazione "*accaduto prima*" è cruciale per molte applicazioni (ad esempio per l'allocazione delle risorse), vedremo come estenderla anche a questi sistemi.

La relazione accaduto prima

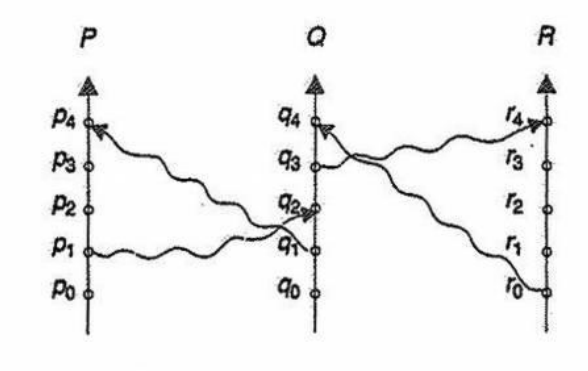
La relazione "*accaduto prima*" può essere definita come segue:

1. se A e B sono eventi dello stesso processo e A è accaduto prima di B, allora **A→B**
2. se A è l'evento di trasmissione di un messaggio in un processo e B è l'evento di ricezione dello stesso messaggio da parte di un altro processo, allora **A→B**
3. se **A→B** e **B→C**, allora **A→C**

Due osservazioni:

- la relazione non è riflessiva
- se due eventi non sono in relazione tra loro, allora sono concorrenti e non si influenzeranno a vicenda. E viceversa

E' possibile rappresentare il tutto con un diagramma spazio-temporale, con gli assi che rappresentano i processi e il tempo, i circoli sono gli eventi e le frecce indicano le relazioni.



Implementazione

Per stabilire quale tra due eventi è accaduto prima è necessario un orologio comune o più orologi sincronizzati tra loro, dispositivi che in un sistema distribuito sono difficilmente disponibili. La soluzione è attribuire ad ogni evento una **marca di tempo** (*timestamp*) che consenta il seguente ordinamento globale: se A→B allora il timestamp di A è minore di quello di B.

Nel sistema distribuito si definisce inoltre per ogni processo un *orologio logico* che ordina i propri eventi con un incremento monotono, mentre per processi diversi che comunicano tra loro applica un avanzamento forzato quando un processo riceve un messaggio la cui marca di tempo è più grande del valore del proprio orologio logico.

Mutua esclusione

Come implementare la **mutua esclusione** in un sistema distribuito? Di seguito considereremo per semplicità un sistema con *n* processi, ognuno residente su un unico processore.

Metodo centralizzato

Con il **metodo centralizzato** si sceglie un processo come *coordinatore dell'accesso* alla sezione critica. Quando un processo vuole accedervi invia prima una richiesta al coordinatore; questi controlla che la risorsa sia disponibile nel qual caso invia una risposta (che è già implicitamente un permesso), altrimenti lo mette in una coda possibilmente implementata come FCFS (così da evitare starvation). Una volta che il processo termina l'utilizzo della sezione critica invia un messaggio di rilascio al coordinatore, che automaticamente seleziona il primo processo della coda e gli invia una risposta accordandogli l'accesso.

Se il coordinatore crasha, un altro processo verrà eletto al suo posto.

Le prestazioni sono limitate, o per dirla testualmente alla Piuri, "fanno schifo".

Metodo completamente distribuito

Se si desidera distribuire nell'intero sistema la possibilità di assumere decisioni, la soluzione diventa più complicata. Il funzionamento è il seguente:

- quando P vuole entrare in sezione critica genera una marca di tempo ed invia una richiesta di accesso a tutti i processi
- quando un processo Q riceve tale richiesta, può comportarsi in tre modi diversi:
  - ritarda la risposta se è in sezione critica
  - risponde immediatamente se non intende entrare in sezione critica
  - se desidera entrare nella propria sezione critica ma non vi è entrato, confronta la propria marca di tempo con quella di P; se la sua è più grande allora risponde immediatamente, altrimenti ritarda la risposta per entrare prima

Ricordiamo che un processo può entrare nella propria sezione critica quando ha ricevuto un messaggio di risposta da tutti gli altri processi del sistema. In questo modo si ottiene la mutua esclusione garantendo al contempo l'assenza di stalli e starvation. L'aspetto negativo di questa tecnica è che i processi devono conoscere l'identità di tutti gli altri presenti nel sistema, e se uno di questi fallisce l'intera opera di coordinamento fallisce anch'essa.

Questo protocollo è particolarmente adatto a insiemi piccoli e stabili di processi cooperanti.

Il metodo del passaggio di token

I processi vengono organizzati logicamente in una struttura ad anello e viene fatto circolare tra loro un **token**, ovvero un messaggio speciale che autorizza il proprietario temporaneo ad entrare nella sezione critica. Essendo unico il token viene garantita la mutua esclusione.

Per un corretto funzionamento bisogna fare in modo che il token non venga perso e che venga sempre mantenuta la struttura ad anello logico.

Atomicità

Una **transazione atomica** è un insieme di operazioni che devono essere eseguite in modo atomico, cioè o tutte o nessuna. In un sistema distribuito è garantita da un *coordinatore della transazione* che è responsabile dell'invio, della suddivisione in più parti da distribuire ai vari host partecipanti, del coordinamento e infine della sua terminazione.

Il protocollo di commit a due fasi e la gestione dei guasti nel protocollo 2PC

(da Base di Dati Complementi)

Nel **protocollo di commit a due fasi** abbiamo due attori: il coordinatore della transazione (TM, Transaction Manager) e gli host partecipanti (RM, Resource Manager).

Prima fase del protocollo:

- Il TM manda un messaggio prepare a tutti gli RM e imposta un timeout indicando il massimo tempo allocato al completamento della prima fase
- gli RM in stato affidabile scrivono nel loro log il record ready e trasmettono al TM il messaggio ready, che indica la scelta di partecipare al protocollo
- gli RM in stato non affidabile mandano un messaggio no ready
- il TM raccoglie i messaggi di risposta
    - se riceve un messaggio positivo da tutti gli RM, scrive un record global commit nel log e si prosegue con la seconda fase
    - se uno o più dei messaggi ricevuti è negativo o non tutti i messaggi sono ricevuti entro il time out, il TM scrive un record global abort nel log e termina il protocollo

Seconda fase del protocollo:

- il TM trasmette la sua decisione globale a tutti gli RM. Imposta poi un time out.
- gli RM che sono in uno stato ready ricevono la decisione, scrivono il record relativo (commit o abort) nel loro log e mandano un acknowledgement (ack) al TM. Poi eseguono il commit/abort
- il TM raccoglie tutti gli ack dagli RM coinvolti nella seconda fase. Se il time out scade, il TM stabilisce un altro timeout e ripete la trasmissione a tutti gli RM dai quali non ha ricevuto ack
- quando tutti gli ack sono arrivati, il TM scrive il record complete nel suo log

Questo protocollo unito al mantenimento di un file di log assicura una buona tolleranza ai guasti delle macchine e della rete, rendendo vita facile anche ai protocolli di ripristino.

Controllo della concorrenza

Il **controllo della concorrenza** in un ambiente distribuito è garantito da un gestore delle transazioni, che oltre a controllarne l'esecuzione (sia in locale che in globale) mantiene anche un file di log per il recupero del sistema dopo il verificarsi di guasti.

Protocolli bloccanti

I **protocolli bloccanti** a due fasi possono essere usati in un ambiente distribuito facendo particolare attenzione nella scelta implementativa del gestore del blocco.

Schema non replicato

Se nessun dato è replicato nel sistema allora l'implementazione più semplice prevede un unico responsabile locale del lock per macchina, il cui compito è bloccare o sbloccare le risorse in base alle transazioni che le richiedono. Il tutto viene gestito attraverso messaggi: due per la richiesta di blocco ed uno per la risposta.

La gestione dello *stallo* è complicata perché le richieste non avvengono su un unico host; abbiamo però una maggiore tolleranza ai guasti dato che se il coordinatore crasha è impedito l'accesso alle sole risorse di cui era diretto responsabile.

Metodo del coordinatore singolo

A differenza di prima abbiamo un *unico coordinatore centralizzato del lock*, quindi tutte le richieste (realizzate come nello schema non replicato) passano da lui.

L'implementazione è semplice così come la gestione degli stalli, ma si hanno prestazioni limitate (il coordinatore rappresenta un collo di bottiglia) e bassa tolleranza ai guasti (se salta lui addio).

Protocollo di lock a maggioranza

In questo caso è prevista una replica dei dati nel sistema ed un *responsabile dei lock* per ogni sito. Quando una transazione vuole un certo dato invia la richiesta ad almeno la metà più uno degli host che lo detengono; gli viene accordato l'accesso solo se riceve il consenso della maggioranza dei coordinatori (ognuno dei quali risponde in modo indipendente dagli altri).

Il vantaggio di questa tecnica è che tratta i dati in modo decentralizzato, a fronte però di maggiori difficoltà implementative (più messaggi da gestire) e una più complessa e specifica gestione degli stalli.

Protocollo polarizzato

Il **protocollo polarizzato** fa una distinzione tra richieste di blocchi condivisi e esclusivi, favorendo i primi. Un blocco condiviso viene infatti rapidamente gestito in locale, mentre quello esclusivo con il sistema del protocollo a maggioranza.

Comporta un minor sovraccarico per gli accessi in lettura, ma condivide con l'altro i sovraccarichi in scrittura e i problemi di stallo (dato che le macchine non sono organizzate in modo simmetrico).

Copia primaria

Tra tutte le repliche ne viene eletta una come **copia primaria** ed è su essa che avvengono le richieste. E' semplice da implementare e non si ha sovraccarico, ma si ha meno probabilità di trovarla disponibile.

Marca di tempo

Esistono due modi per generare marche di tempo uniche:

- con il **metodo centralizzato** vengono distribuite da un singolo coordinatore, che può utilizzare un contatore logico o il suo orologio locale
- con il **metodo distribuito** ogni sito genera un'unica marca di tempo locale usando un contatore logico o l'orologio locale, mentre quella globale si ottiene concatenando quella locale con l'id del sito (entrambi unici). Poiché un sito potrebbe generare marche di tempo locali più velocemente degli altri, bisogna fare in modo che la loro produzione sia ragionevolmente omogenea. Si può a tal fine utilizzare un orologio logico che viene forzato ad aggiornarsi (con un incremento unitario) ogni volta che riceve un messaggio da qualche altra transazione.

Gestione delle situazioni di stallo

Prevenzione delle situazioni di stallo

Gli algoritmi che vedremo sono un'estensione di quelli già visti per i sistemi centralizzati.

Il primo prevede un ordinamento globale delle risorse del sistema distribuito mediante *id* unici progressivi, impedendo poi a un processo di ottenere una risorsa se è già in possesso di un'altra con identificativo maggiore. E' piuttosto semplice da realizzare e comporta sovraccarichi minimi.

Il secondo è una generalizzazione dell'algoritmo del banchiere, anch'esso semplice ma che può portare ad alti sovraccarichi dato che il banchiere rappresenta un collo di bottiglia. Non è tra le soluzioni preferibili.

Un terzo sistema è quello delle *marche di tempo con rilascio anticipato delle risorse*, che assegna un numero univoco di priorità a ogni processo e funziona come segue: se P possiede la risorsa e Q ha priorità maggiore, allora P (su cui viene effettuato un rollback) rilascia la risorsa che passa all'altro. Questa soluzione impedisce la formazione di stalli anche a livello distribuito, ma può portare a *starvation*, evitabile con due tecniche che utilizzano le marche di tempo:

- schema **wait and die**: quando un processo richiede una risorsa posseduta da un altro, viene messo in attesa solo se ha una marca di tempo minore; altrimenti viene annullato
- schema **wound-wait**: quando un processo richiede una risorsa posseduta da un altro, viene messo in attesa solo se ha marca di tempo maggiore; altrimenti si sottrae la risorsa al processo che la possiede attualmente

Rilevamento delle situazioni di stallo

L'algoritmo di rilevamento degli stalli prevede l'utilizzo dei *grafi di attesa*, che descrivono l'allocazione delle risorse: se c'è un ciclo nel grafo si ha uno stallo tra i processi coinvolti.

In un sistema distribuito ogni sito mantiene un proprio grafo di attesa locale, nel quale però l'assenza di cicli non implica automaticamente l'assenza di deadlock: è infatti nell'unione di tutti i grafi locali che bisogna verificarne l'esistenza.

Ma come possono essere organizzati questi grafi?

Metodo centralizzato

Col **metodo centralizzato** il grafo di attesa globale viene costruito e mantenuto aggiornato da un unico coordinatore centralizzato. A causa dei ritardi dovuti alle transazioni nel sistema distribuito bisogna distinguere tra *grafo reale*(utopico) e *costruito*. Quest'ultimo, pur essendo un'approssimazione di quello reale, deve almeno garantire che se esiste uno stallo allora deve essere segnalato, e viceversa se uno stallo viene segnalato allora deve effettivamente esistere.
L'aggiornamento del grafo può avvenire in seguito all'inserimento o alla rimozione di un arco (la macchina avvisa il coordinatore con un messaggio), oppure dopo un certo numero di cambiamenti o ancora a discrezione del coordinatore.
L' *algoritmo di ricerca* dei cicli consiste nella trasmissione dei grafi locali da parte di tutte le macchine e successiva costruzione del grafo globale da parte del coordinatore, che avvia su esso la ricerca vera e proria. Se riscontra un ciclo seleziona un processo vittima su cui effettua un rollback per uscire dal deadlock.
E' importante segnalare che si potrebbero avere dei rollback inutili a causa di *falsi cicli*, dovuti ai ritardi legati ai tempi di trasmissione, o perché in seguito alla scelta della vittima lo stallo smette di sussistere per cause indipendenti dall'algoritmo di ricerca.

Metodo completamente distribuito

Questo algoritmo fa in modo che la responsabilità del rilevamento degli stalli venga condivisa da tutti i siti, ognuno dei quali ha un controllore che costruisce un proprio grafo di attesa parziale rispetto a quello globale. Ad ognuno di questi grafi viene aggiunto un nodo P_{ex} che indica l'attesa per risorse appartenenti ad altre macchine. Se esiste un ciclo che non coinvolge P_{ex} allora c'è sicuramente uno stallo, se invece lo coinvolge bisogna contattare quella macchina per verificarne l'effettiva esistenza.
Il problema di questo sistema è che il rilevamento contemporaneo di cicli in grafi di attesa locali provoca un sovraccarico di gestione e messaggi ridondanti. Una tecnica per ridurlo è assegnare a ogni processo un *id*, e se una macchina scopre uno stallo che coinvolge P_{ex} si comporta così:

- se il processo prima di P_{ex} nel ciclo ha *id* minore di quello successivo, invia il messaggio di rilevamento alle altre macchine
- altrimenti se ne lava le mani e lascia ad altri il compito di rilevarlo e gestirlo

Algoritmi di elezione del coordinatore

Abbiamo visto come molti algoritmi distribuiti abbiano bisogno di un coordinatore perché siano assicurate un certo numero di funzioni. Se uno di questi viene meno esistono alcuni algoritmi di *elezione del coordinatore*, che presuppongono che ad ogni processo sia associato un numero di priorità: il più alto tra quelli attivi viene eletto.

Algoritmo del bullo

Se un processo invia una richiesta al coordinatore e non riceve risposta, dopo un certo intervallo di tempo suppone che sia guasta e cerca di eleggere sé stesso. La prima cosa che fa è inviare un messaggio di inizio elezione a tutti i processi con priorità più alta della sua, dopodiché si mette in attesa. Se riceve una risposta con l'identificatore del nuovo coordinatore, ne prende atto e registra l'informazione, altrimenti riavvia l'algoritmo. Se invece non riceve alcuna risposta, si auto-elegge e informa tutti i processi del suo nuovo ruolo.

Algoritmo dell'anello

Si basa sul fatto che i collegamenti sono unidirezionali e utilizza una lista attiva per ogni processo. Quando P si accorge che il coordinatore non funziona genera una nuova lista attiva vuota, invia un messaggio di elezione col proprio numero al suo vicino e scrive la sua priorità nella lista. Un processo che riceve questo messaggio:

- se non è contenuto nella lista aggiunge il proprio numero nella sua lista attiva e inoltra il messaggio
- altrimenti significa che la lista contiene già tutti i processi attivi del sistema, dunque è sufficiente controllare qual è quello a priorità maggiore ed eleggerlo coordinatore

Protezione

Obiettivi della protezione

Per **protezione** si intende la messa in sicurezza delle *risorse* (dette anche *oggetti*) da parte di accessi non autorizzati di utenti, programmi o processi. Stabilito quest'obiettivo, bisogna distinguere le *regole* (che tracciano le linee politiche della strategia di protezione) dai *meccanismi* (gli strumenti che le applicano) per garantire maggior flessibilità.

Domini di protezione

Un sistema è normalmente composto sia da risorse fisiche (l'hardware) che informative (i file, le strutture di comunicazione, i programmi), ognuna delle quali è caratterizzata da un nome univoco e da un insieme di operazioni consentite.
I processi dovrebbero poter accedere solo a quelle risorse per cui hanno l'autorizzazione, che andrebbe distribuita secondo il *principio di minima conoscenza*: un processo deve poter accedere solo alle risorse strettamente necessarie per la propria computazione. In questo modo si riesce a limitare la quantità di danni potenziali dovuti ad un processo malfunzionante.

Struttura del dominio di protezione

Per meglio applicare il principio di minima conoscenza ogni processo viene fatto operare in un **dominio di protezione**, che definisce l'insieme di risorse cui può accedere ed il tipo di operazioni lecite che è autorizzato a compiere. Per ognuno di essi vengono perciò definiti una serie di *diritti di accesso*, composti da una coppia<nome oggetto, insieme-diritti>. Notare che i domini non devono essere necessariamente disgiunti, ma possono condividere alcuni dei loro diritti.
L'associazione tra diritto e dominio può essere *statica* o *dinamica*. Nel primo caso il set di risorse disponibili per il processo è fissato all'inizio per tutta la durata dell'esecuzione; è semplice da implementare ma spesso assegna più diritti di quanto effettivamente il processo abbia bisogno in un dato momento. Con l'associazione dinamica invece vengono forniti meccanismi per cambiare dominio di protezione, il che meglio applica il principio di minima conoscenza a fronte però di una maggiore complessità.
Un dominio può coincidere con:

- un utente, quindi il cambio di dominio avviene quando un altro utente effettua l'accesso al sistema
- un processo, quindi si ha un cambio di dominio in concomitanza di un cambio di contesto
- una procedura, in cui si ha un cambio di dominio ad ogni nuova invocazione di procedura

Va però ricordato che nei cambi di dominio (*switch*) la loro struttura rimane inalterata, comprese le operazioni abilitate sulle risorse. Per poter modificare diritti e struttura bisogna averne l'autorità.

Matrice d'accesso

Il modello di protezione può essere rappresentato in modo astratto con la **matrice di accesso**, in cui le righe indicano i domini di protezione e le colonne gli oggetti. In ogni cella vengono invece definiti i diritti d'accesso, ridotti a una lista di operazioni consentite. La decisione del tipo di dominio in cui deve essere incluso un processo spetta generalmente al sistema operativo, mentre è l'utente che decide il contenuto degli elementi nella matrice aggiungendo colonne (quindi creando nuovi oggetti) e impostando alcuni diritti. Notare che la stessa matrice d'accesso può essere considerata come risorsa, dunque si possono impostare diritti d'accesso anche su di essa.

dominio \ oggetto	F_1	F_2	F_3	stampante
D_1	read		read	
D_2				print
D_3		read	execute	
D_4	read write		read write	

La modifica controllata del contenuto degli elementi della matrice d'accesso è resa possibile da tre operazioni:

- *copia* di un diritto d'accesso da un dominio all'altro, ma solo all'interno della stessa colonna. Si attua col comando copy e si rappresenta in matrice con un asterisco in coda al diritto d'accesso. Una variante è il *trasferimento* del diritto (che non è altro che una copia seguita da una cancellazione) e un'altra è la *copia con limitazioni alla diffusione* (il diritto copiato non può ulteriormente propagarsi)
- *proprietà*, che consente a un processo autorizzato di aggiungere o rimuovere diritti
- *controllo*, che può essere applicata solo agli oggetti del dominio e che consente di cambiare gli elementi di una riga della matrice d'accesso

I diritti di copia e proprietà permettono di evitare il propagarsi dei diritti d'accesso, ma non garantiscono il contenimento delle informazioni, che è considerato un problema irrisolvibile.

Implementazione della matrice d'accesso

Tabella globale

La **tabella globale** è la soluzione più semplice e prevede di rappresentare la matrice come una tabella i cui elementi sono terne ordinate <dominio, oggetto, diritti>.
Un primo limite di tale soluzione è che la tabella risulterebbe molto grande (spesso con dimensioni maggiori della memoria centrale) e per giunta la maggior parte delle celle sono vuote dato che non sono definiti diritti per ogni coppia dominio-oggetto. Altro limite è che non sono possibili raggruppamenti di oggetti o domini, cosa che consentirebbe una più facile gestione degli stessi.

Liste di controllo degli accessi

Le **liste di controllo degli accessi** sono una memorizzazione per colonne della matrice degli accessi: per ogni oggetto si crea una lista composta da coppie ordinate <dominio, diritti>. Si può definire anche un insieme di diritti d'accesso predefinito, da controllare prima della lista di controllo degli accessi.

Liste di capacità dei domini

Le **liste di capacità dei domini** sono una memorizzazione per righe della matrice degli accessi, e consistono in una lista per ogni dominio formata da coppie <oggetto, diritti>. Gli oggetti sono rappresentati dal loro nome fisico o dal loro indirizzo, e vengono chiamati *capacità*: il semplice possesso delle capacità autorizza l'accesso.
La lista delle capacità non è direttamente accessibile ma è essa stessa un oggetto protetto. E' gestita dal sistema operativo che concede accessi indiretti ai suoi elementi.

Il meccanismo lock-key

Il **meccanismo lock-key** (serratura-chiave) è un compromesso tra le due liste viste finora: ogni oggetto ha una lista unica di bit detta *lock*, mentre ogni dominio ne ha un'altra detta *key*. Un processo in esecuzione in un dominio può accedere a un oggetto solo se la sua key è in grado di aprire il lock interessato.
Tutto questo meccanismo viene gestito dal sistema operativo.

Confronto

Le liste di controllo degli accessi possono essere specificate dagli utenti e contengono informazioni globali, ma sono inefficienti su grandi sistemi.
Le liste della capacità dei domini sono invece relative agli oggetti dunque hanno informazioni localizzate e sistemi di revoca poco efficienti.

Revoca dei diritti d'accesso

In un sistema di protezione dinamica può essere necessario revocare i diritti di accesso su oggetti condivisi da più utenti. Tali revoche possono essere:

- *immediate* o *ritardate*
- *selettive* (che si riflettono su un numero limitato di utenti) o *generali* (su tutti)
- *parziali* (riguardano un numero limitato di diritti) o *totali* (tutti)
- *temporanee* o *permanenti*

Con lo schema basato sulla lista degli accessi la revoca è un'operazione semplice e immediata: basta cercare il diritto d'interesse e rimuoverlo. Per la lista delle capacità il meccanismo è invece più complesso poiché le informazioni si trovano distribuite su tutto il sistema e bisogna prima recuperarle. Esistono diverse tecniche che si prepongono questo fine:

- *riacquisizione*: le capacità sono periodicamente cancellate. Se un processo tenta di adoperarle e sono state cancellate, tenterà di riacquistarle; se l'accesso è stato revocato non potrà più riuscirci
- *puntatori alle capacità*: si mantiene per ogni oggetto una lista dei puntatori a tutte le capacità ad esso associate; per revocarne una basterà eliminare il puntatore corrispondente. E' una soluzione comune ma piuttosto costosa
- *in direzione*: le capacità puntano indirettamente agli oggetti, o meglio puntano a un unico valore in una tabella globale che a sua volta referenzia l'oggetto. L'operazione di revoca ricerca il valore desiderato in questa tabella e lo cancella. Questo sistema non consente la revoca selettiva
- *chiavi*: le chiavi sono un gruppo di bit associati ad una capacità. Ogni oggetto ha una chiave principale in base alla quale la capacità viene ricercata e verificata. La revoca setta la chiave con un nuovo valore, invalidando quelli precedenti

Sistemi basati sulla capacità

I **sistemi basati sulle capacità** mettono a disposizione un approccio nativo all'uso di risorse basato sulle capacità, offrendo agli utenti meccanismi per definirle e controllarle più efficacemente.

Protezione basata sul linguaggio

La **protezione basata sul linguaggio** evita di scaricare tutte le responsabilità di garantire la sicurezza sul sistema operativo prendendosene parte del carico al momento della compilazione del programma applicativo, incorporandola nel linguaggio di programmazione. Permette di conseguenza un controllo più granulare, guadagnando efficienza e flessibilità rispetto alla protezione basata sul kernel (che rimane comunque la più sicura).