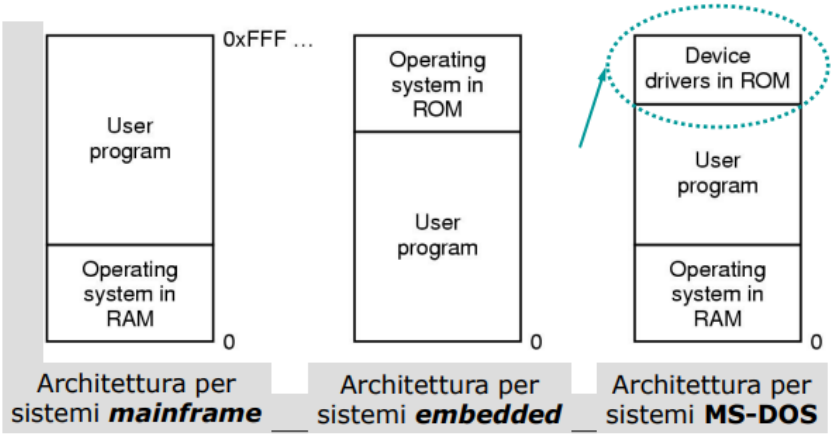
GESTIONE DELLA MEMORIA:

Si vuole una memoria che sia capiente, veloce, permanente ed economica ma tutte queste caratteristiche non possono appartenere ad una sola tipologia di memoria per questo si ricorre alla gerarchia di memorie. Il gestore della memoria è il componente incaricato di soddisfare le esigenze di memoria dei processi.

Esistono due classi di gestione della memoria, quelle per processi allocati a modo fisso e processi soggetti a migrazioni da una memoria all'altra durante l'esecuzione(esp: da memoria principale a disco).La memoria presente è di solito minore di quella necessaria per tutti i processi attivi.

Sistemi monoprogrammati:

esegue un solo processo alla volta e la memoria disponibile è ripartita solo tra lui e il S/O, l'unica scelta progettuale e decidere dove sarà la memoria del S/O. in RAM del S/O sarà presente solo l'ultimo comando invocato dall'utente. *Esempi di monoprogarmmati:*



LA FRAMMENTAZIONE:

La frammentazione indica come vengono suddivisi i dati in più parti e può essere di due tipi, frammentazione interna ed esterna.

- Problemi frammentazione interna: la memoria è divisa in blocchi di grandezza uguale e quindi possono essere riempiti solo in parte se il dato è poco pesante.
- Problemi frammentazione esterna: la memoria viene divisa in blocchi di dimensioni diverse e quindi rimangono aree che non vengono usate non copribili da blocchi.

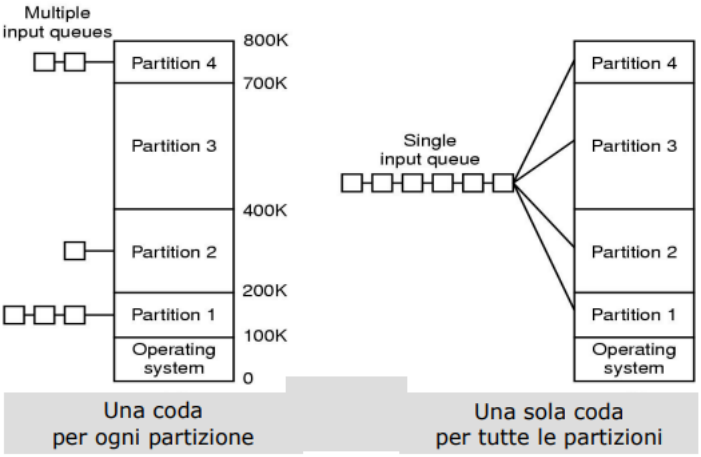
Entrambe causano spreco di memoria in qualche modo.

Sistemi multiprogrammati:

La memoria nei sistemi multiprogrammati viene gestita assegnando una partizione ad ogni processo all'avvio del sistema e possono avere dimensione diversa. Il problema arriva quando si cerca di assegnare dinamicamente i processi alle partizioni, perché si creano problemi di frammentazione, ci sono due tecniche usate per l'assegnazione della memoria:

- A ogni processo viene assegnata la partizione di dimensione più appropriata, i processi vengono messi in coda per la partizione più efficace per loro questo però implica un inefficacia nell'uso della memoria.
- Assegnazione opportunistica, viene usata una sola coda per mettere in processi in attesa della partizione di memoria. L'assegnazione viene fatta dando la partizione libera al processo successivo nella coda o a quello più adatto esaminando tutta la coda. I processi

"piccoli" vengono spesso discriminato quando in realtà dovrebbero essere privilegiati in quanto più interattivi.



Vantaggi e svantaggi multiprogrammazione:

la CPU è più veloce di tutti quindi quando lavora con gerarchie di memoria deve andare al loro ritmo e questo è uno spreco.

Per questo si usa una valutazione probabilistica per determinare il numero di processi che si devono eseguire in parallelo per massimizzare l'uso della CPU.

Ipotizzando che ogni processo utilizzo P% del tempo in attività di I/O e N sia il numero di processi presenti in memoria simultaneamente. L'utilizzo della CPU stimato è **$1-P^N$**

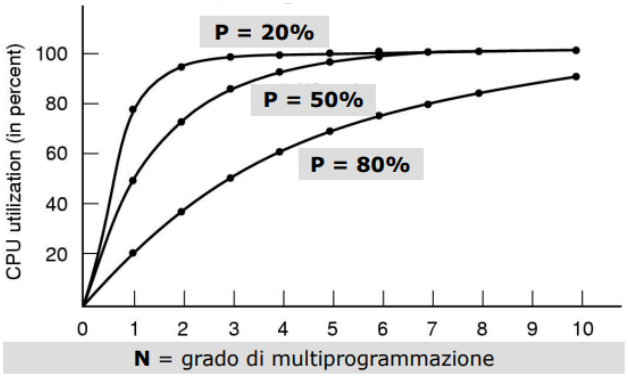


grafico con diverse percentuali di attesa.

Esempio Progettazione Memoria

- Si consideri un computer con 32 MB di memoria e 80% di attesa I/O media per ogni processo
  - 16 MB riservati per il sistema operativo
  - 4 MB riservati per ciascun processo
  - In totale si hanno quindi 4 processi simultaneamente in memoria
  - Con P = 0,8 si ha una utilizzazione della CPU di $1 - 0{,}8^4$ = **60%**
- Aggiungendo altri 16 MB
  - Si possono avere 8 programmi simultaneamente in memoria
  - Con P = 0,8 si ha una utilizzazione della CPU di $1 - 0{,}8^8$ = **83%**
- Aggiungendo altri 16 MB
  - Si possono avere 12 programmi simultaneamente in memoria
  - Con P = 0,8 si ha una utilizzazione della CPU di $1 - 0{,}8^{12}$ = **93%**

RILOCAZIONE E PROTEZIONE:

Rilocazione: interpretazione degli indirizzi emessi da un processo in relazione alla sua posizione in memoria, si dividono in riferimenti assoluti permissibili al programma e da rilocare.

Protezione: assicurarsi che un processo operi soltanto nello spazio di memoria a esso concesso.

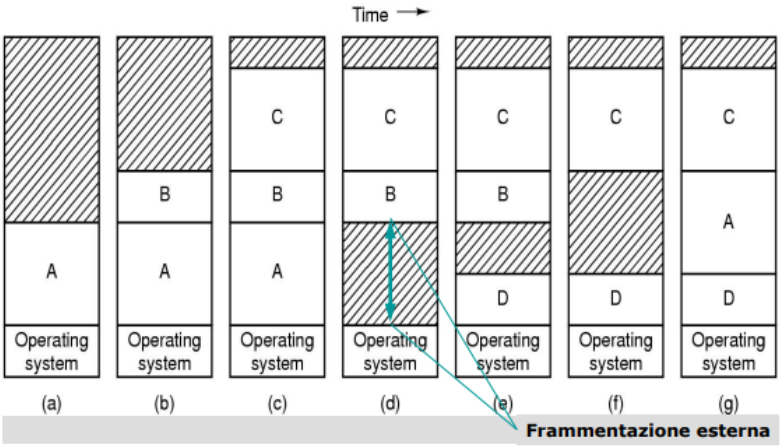
- Soluzione storica adottata da IBM
  - Memoria divisa in blocchi (2 kB) con codice di protezione per blocco (4 *bit*)
  - La PSW di ogni processo indica il suo codice di protezione
  - Il S/O blocca ogni tentativo di accedere a blocchi con codice di protezione diverso da quello della PSW corrente
- Soluzione combinata (rilocazione + protezione)
  - Un processo può accedere memoria solo tra la **base** e il **limite** della partizione a esso assegnata
  - Valore **base** aggiunto al valore di ogni indirizzo riferito (operazione costosa)
  - Il risultato confrontato con il valore **limite** (operazione veloce)

SWAPPING:

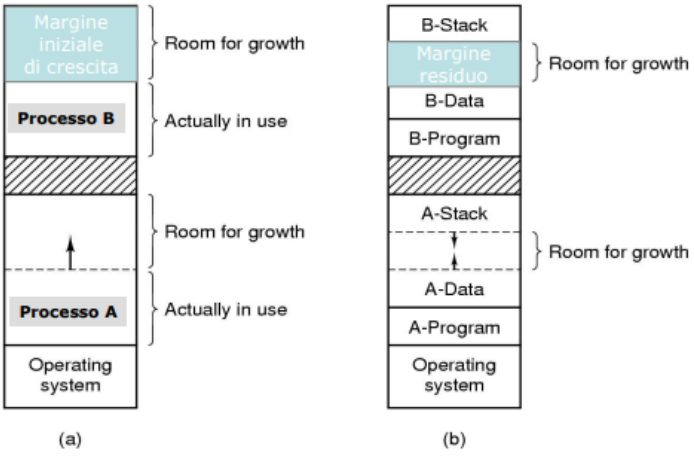
Tecnica usata per alternare processi in memoria principale senza garantire allocazione, trasferisce processi interi tra partizioni diverse nel tempo e il processo rimosso(solo la parte modificata) viene salvato nella memoria secondaria.

Processi diversi richiedono partizioni diverse quindi c'è rischio di frammentazione esterna; occorre ricompattare periodicamente la memoria principale che però ha un costo esagerato, per spostare 4 B in 40 ns servono 5.37 s per una RAM di 512 MB.

La dimensione di memoria di un processo varia nel tempo quindi occorre assegnare l'ampiezza della partizione con margine.



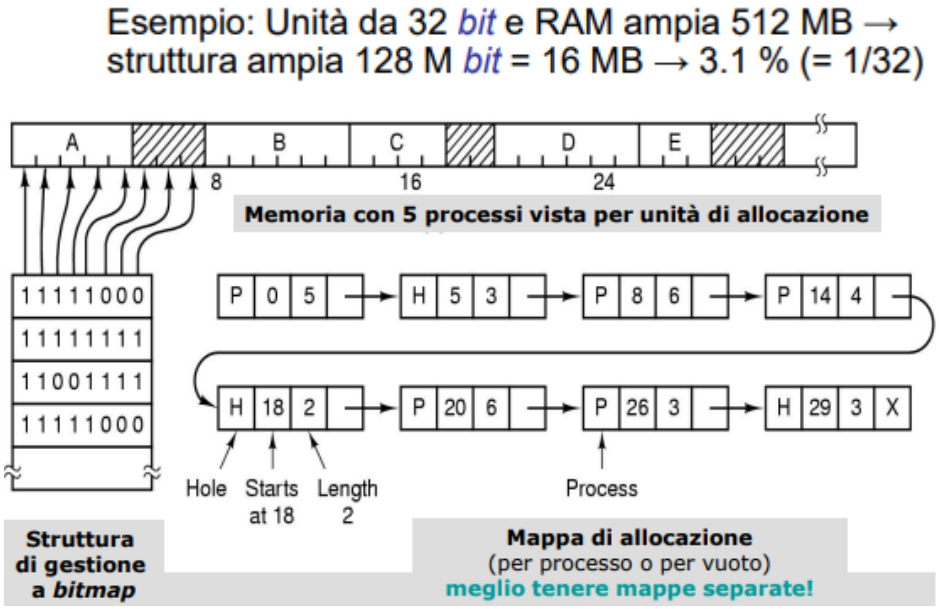
senza margine di allocazione.



con margine.

Quando la memoria principale è assegnata dinamicamente occorre tenere traccia dello stato di uso e per queto si usano due strategie principali:

- **Mappe di bit:**
  - la memoria viene vista come un insieme di unità di allocazione(1 bit per unità), la tabella di gestione però occupa memoria RAM.



- **Liste collegate:**
  - La memoria è vista come segmenti, ogni segmento equivale a processo e spazio libero tra processi e ogni elemento di lista rappresenta un segmento e ne specifica il punto di inizio, ampiezza e successore e le liste sono ordinate per indirizzo di base.
  - Varie strategie di allocazione:
    - First fit: il primo segmento libero ampio abbastanza riparte sempre da capo.
    - Next fit: il primo segmento libero che è presente dopo quello usato prima.
    - Best fit: segmento libero più adatto, analizza tutta la coda.
    - Worst fit: il segmento più ampio presente.
    - Quick fit: liste diverse di ricerca per ampiezze diverse.

MEMORIE VIRTUALI:

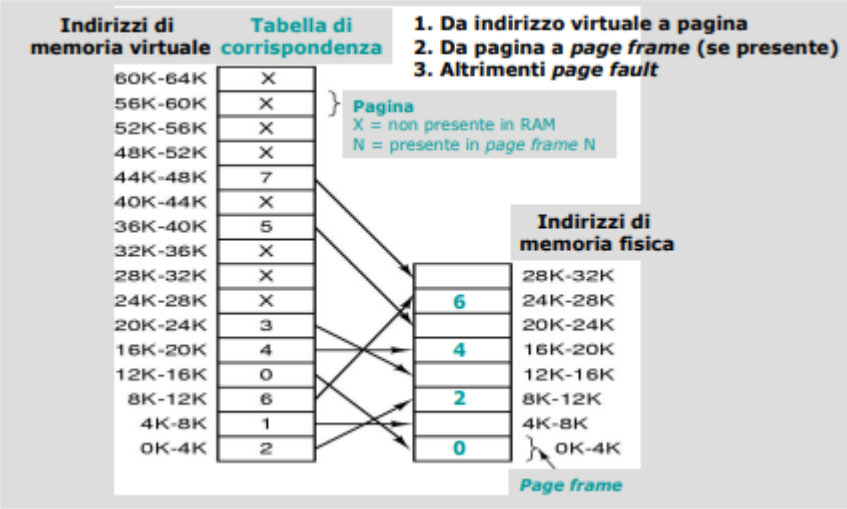
Una soluzione all'insufficienza di memoria per un processo fu la sua suddivisione in parti, *overlay,* veniva caricata una parte alla volta nella RAM e quando finiva una veniva seguita dalla successiva, la suddivisione era decisa dal programmatore.

Un'altra soluzione fu la **memoria virtuale ,** il processo viene caricato solo in parte in RAM e la parte non necessaria viene lasciata nella memoria secondaria creando così l'illusione di avere una memoria principale in grado di ospitare tutto il programma , la suddivisione è senza l'intervento del programmatore e ogni processo ha una propria memoria virtuale; ci sono due tecniche dei gestione della memoria virtuale, paginazione e segmentazione. Siccome ogni processo ha una propria copia virtuale di memoria gli indirizzi generati da esso non equivalgono a quelli fisici della RAM; vengono tradotti dall'**MMU** che prima di essere emessi sul bus vengono mappati verso gli indirizzi fisici reali.

Paginazione:

la memoria virtuale è suddivisa in pagine di dimensione fissa, la RAM è suddivisa in *cornici* ampi come le pagine(page frame). Ogni trasferimento da e verso disco avviene in pagine di

questa pagina occorre sapere se è presente in RAM oppure no, con i Bit di presenza, se una pagina non è presente si genera un *page fault* che viene gestito da S/O con *trap(?).*

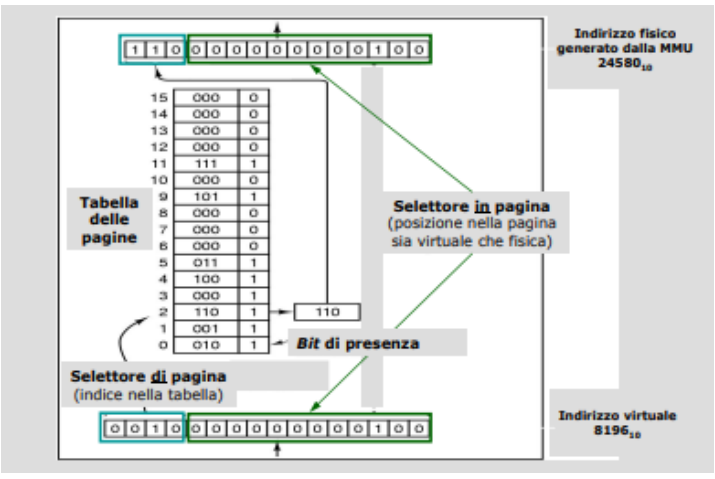


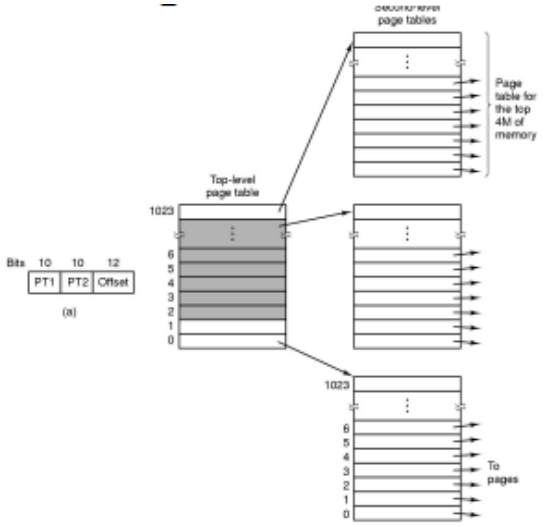
La traduzione dell'indirizzo tra virtuale a fisico avviene mediante una tabella delle pagine, ($indirizzo_{fisico}$= φ $indirizzo_{virtuale}$) la tabella delle pagine ha un grandezza notevole, esempio: Indirizzi virtuali da 32 *bit* e pagine da 4 KB → memoria virtuale da 4 GB = 1 M pagine!

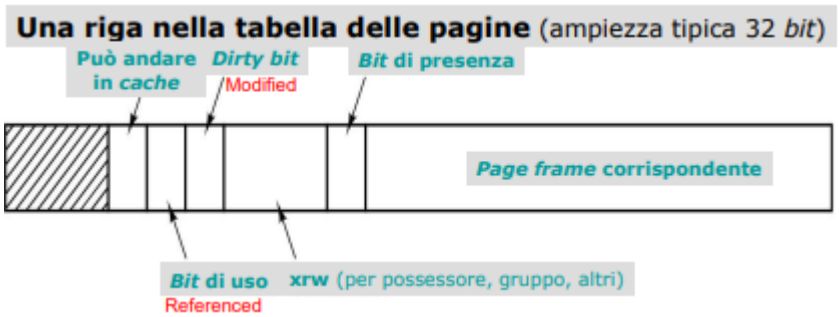
E ciascun processo ha una propria tabella degli indirizzi siccome ha una propria memoria virtuale.

La traduzione deve essere veloce per non causare rallentamenti notevoli durante l'esecuzione, ogni istruzione potrebbe fare riferimento più volte alla tabella delle pagine(possibilità di bottleneck in caso il riferimento alla pagina sia troppo lungo) e ogni indirizzo emesso dal processo deve essere tradotto in modo:

- Semplicemente (e concettualmente) potrebbe utilizzare un vettore di registri (uno per ogni pagina virtuale) caricato a ogni cambio di contesto (vedi figura nella slide seguente)
  - Lineare e non si rischia di dover accedere a memoria per scoprire il riferimento, ma costoso cambiare tutti i registri ad ogni context switch
- Oppure come una struttura sempre residente in RAM
  - Un singolo registro punta all'inizio della page table
  - Difficile che sia usato come soluzione in modo puro







L'indirizzo della pagina quando non si trova in RAM non è nella tabella, la tabella serve alla MMU e il caricamento della pagina viene fatto dal SO quando c'è un *page fault.*

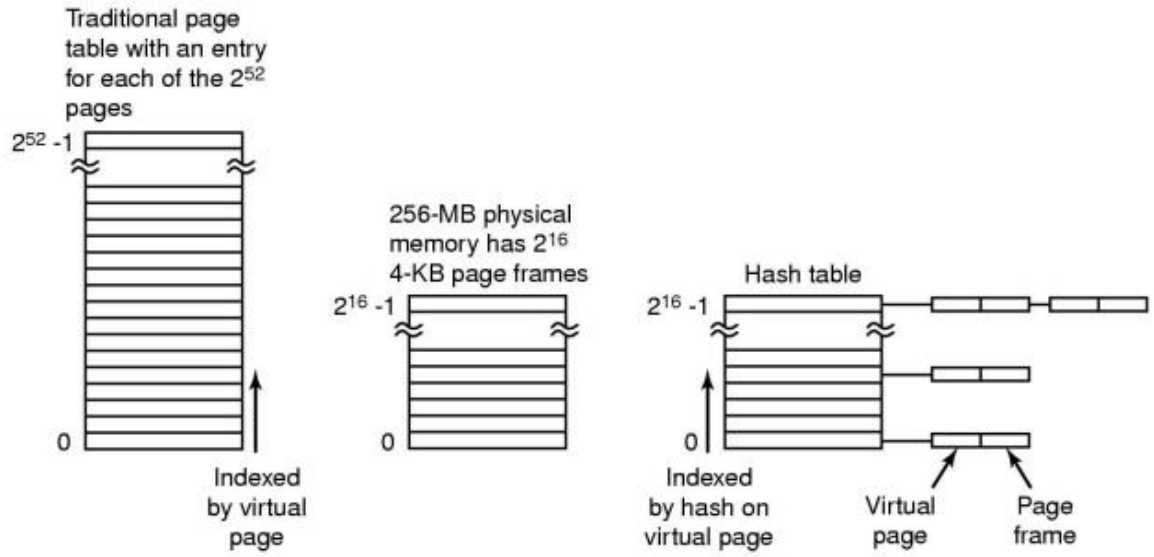
La tabella delle pagine è troppo grande per stare sui registri quindi sta in RAM, ma riferirla per ogni indirizzo è un lavoro che impatta molto sulle prestazioni. Serve quindi utilizzare una soluzione *hardware* supplementare che sia come una cache, la **TLB** è una piccola memoria tipo buffer che risiede dentro alla MMU e consenta una ricerca parallela delle righe. Inoltre basata sul fatto che un processi utilizzi poche pagine e quindi con l'uso di località spaziali e temporali di riferimento.

Ogni indirizzo viene prima trattato con la TLB e poi viene emesso verso la MMU, se la pagina è presente e l'accesso è permesso viene fatta la traduzione con la TLB; se non è presente si ha un *cache miss* e l'informazione richiesta viene caricata in TLB dalla page table rimpiazzando una cella in TLB e riferendone il valore nella page table, solo se cambiato.

Le TLB oggigiorno sono soluzioni software invece che hardware che sono accettabili in termini di prestazione e garantiscono semplicità e riduzione dello spazio utilizzato nella MMU.
Con le architetture a 64 le tabelle delle pagine sono diventate di dimensioni proibitive (64 bit -> memoria virtuale da 16 EB (1 E= 1 M * 1 G), quindi serve un'altra soluzione.

La soluzione impiega una tabella invertita ovvero non più una riga per pagina ma un *page frame* in RAM con un considerevole risparmio di memoria. Ma la traduzione da virtuale a fisico diventa però molto più complessa poiché la pagina potrebbe risiedere in un qualsiasi page frame e quindi bisognerebbe perlustrare l'intera tabella per essere sicuri e ciò causerebbe un grande dispendio di tempo. La soluzione a questo problema è l'utilizzo di TLB per una ricerca più veloce e realizzando la tabella come una *hash table* : f_{hash} (indirizzo$_{virtuale}$).

I dati relativi alle pagine i cui indirizzi indicizzano una stessa riga di tabella vengono collegati in lista.



RIMPIAZZO:

Quando si produce un *page fault* il SO deve rimpiazzare una pagina e salva su disco quella rimossa se è stata modificata durante l'utilizzo. Se una pagina è di uso frequente non viene rimpiazzata per non avere un continuo cambio tra RAM e disco della stessa pagina.

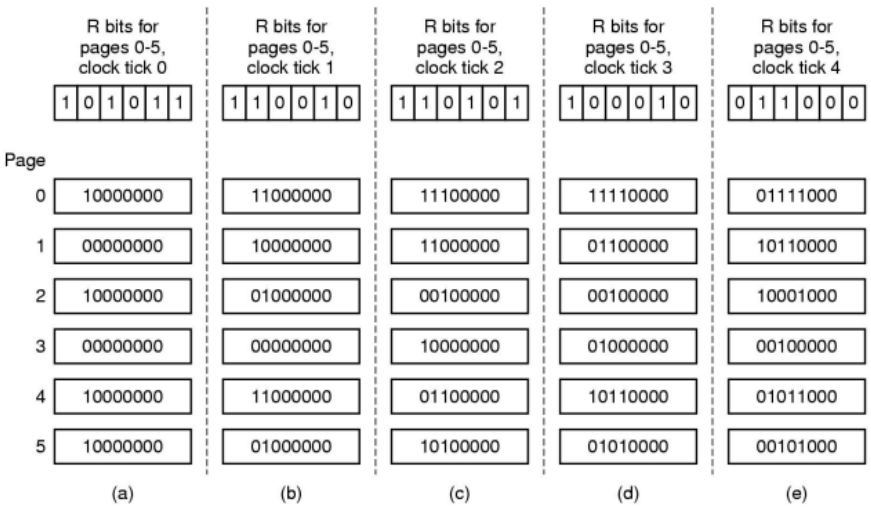
Rimpiazzo ottimale : rimpiazza la pagina che non verrà usata per maggior tempo, la scelta perfetta non è realizzabile perché il SO non sa a quali pagine accederà il processo in futuro; quindi le scelte realizzabili sono solo approssimazioni sotto-ottimali basate sull'osservazione dell'uso delle pagine che sono in RAM attualmente.

POLITICHE DI RIMPIAZZO:

- **NRU** : ci sono 2 bit che caratterizzano ogni page frame, Bit M(modificato) e Bit R(riferita) che viene periodicamente messo a 0. Con questi due bit i page frame vengono suddivisi in classi:
  - Classe 0: non riferita e non modificata.
  - Classe 1: non riferita e modificata.
  - Classe 2: riferita e non modificata.
  - Classe 3: riferita e modificata.

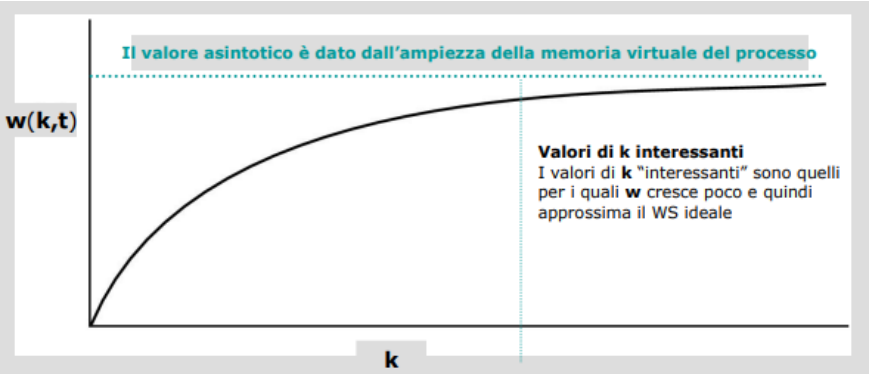
  Viene scelta una pagina a caso della classe più bassa possibile. (NB. Se è stata riferita e non modificata ha classe maggiore perché ha più probabilità di essere richiamata dopo di una non riferita).

- **FIFO :** li funzionamento è semplice, prevede il rimpiazzo con la pagina presente in RAM da più tempo. Il problema è che se ho page frame importanti come quelle relative a dispositivi di IO verranno rimpiazzati per poi essere di nuovo riallocati. Soluzione: **second chance,** con l'uso di un bit R che viene messo a 1 nei page frame importanti quando c'è da fare il rimpiazzo il page frame viene messo all'inizio della coda con bit R=0 invece che andare sul disco.
- **Orologio :** simile al second chance ma i page frame sono mantenuti in una lista circolare.
- **LUR :** il meno utilizzato recentemente, richiede una lista aggiornata ad ogni riferimento in memoria e necessita di un hardware dedicato.
- **NFU:** si appoggia su di un software e utilizza un contatore per ogni page frame che salva le volte in cui viene riferito soprattutto se è stato riferito(R=1), il problema è che non viene mai azzerato e quindi i page frame nuovi sono disavvantaggiati.
- **AGING:** simile al not frequently used , aggiorna periodicamente il contatore che cresce di più se R=1, non aumenta C con R ma lo inserisce a sinistra utilizzando N bit per C così da perdere la memoria dopo N aggiornamenti.



- **WORKING SET:** è dimostrato che i processi emettano la maggior parte dei loro riferimenti in un ristretto spazio locale, *località di riferimento.* Il **working set** è il numero di pagine che un processo ha in uso in un determinato istante, se la RAM non basta per tutto il working set si ha un fenomeno di **thrashing** ovvero una serie di page fault continui(quando il disco gratta). Se invece il working set viene caricato prima dell'esecuzione si ha il **prepaging.**

  W(k, t) è l'insieme delle pagine che soddisfano i **k** riferimenti al tempo **t.**

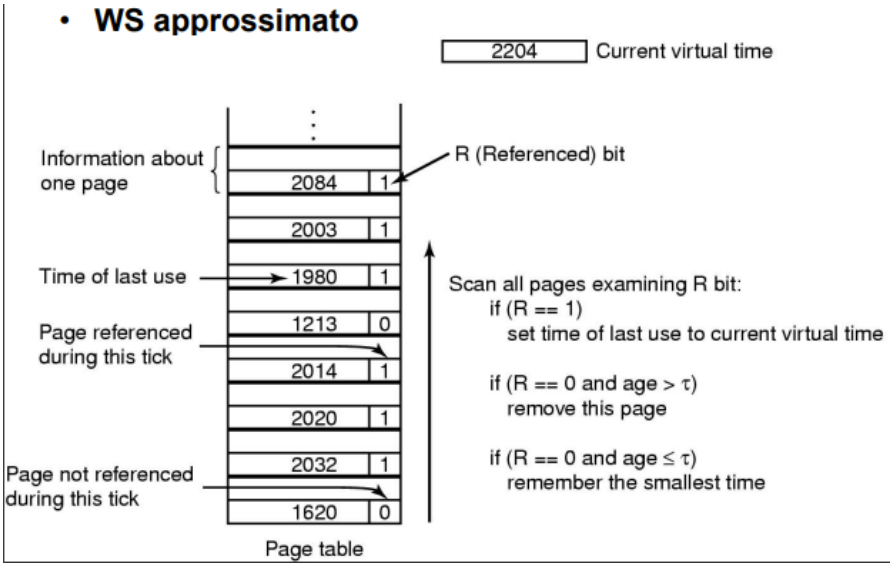


Conoscendo il WS di un processo le pagine da rimpiazzare sarebbero quelle non comprese ma è troppo costoso. È più facile fissare t come (t, t + Δt) e considerando t come valore del tempo di esecuzione del processo allora WS è fatto dalle pagine riferite dal processo nell'ultimo Δt.

WS approssimato

- Simile all'*Aging*
  - Ogni *page frame* in RAM ha un attributo temporale che viene utilizzato insieme all'attributo riferito (R = 1)
    - Tale attributo prende il valore **t** del **tempo virtuale corrente** all'arrivo di un *page fault*
    - R e M sono posti a 1 dall'*hardware*
    - R è posto a 0 (se non in uso) da un controllo periodico e al *page fault*
  - Al *page fault* sono rimpiazzabili le pagine con R = 0 e valore di attributo **antecedente** all'intervallo (**t** − Δt, **t**)
    - Se non ci sono, si prende la più vecchia con R = 0
    - Se all'istante **t** tutti i *page frame* avessero R = 1 verrebbe rimpiazzata una pagina scelta a caso, con M = 0
  - Nel caso peggiore bisogna scandire l'**intera** RAM!

- **WS approssimato**



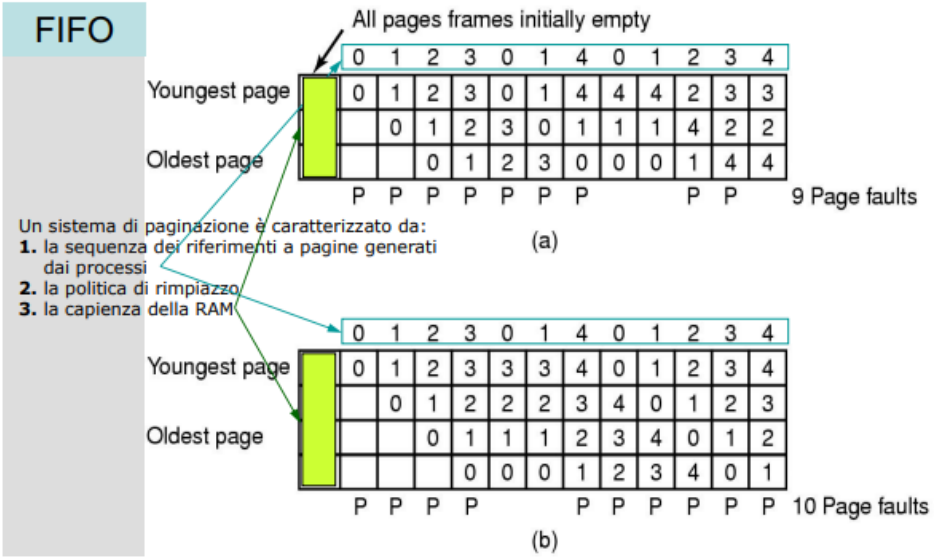
WS approssimato con orologio

- *Page frame* organizzati in lista circolare
  - Come per l'orologio semplice
  - Ma con le informazioni del WS approssimato
- Una "lancetta" indica il *page frame* corrente
  - Al *page fault* se R = 1 la lancetta avanza e R = 0
  - Se R = 0 si valuta l'attributo temporale
    - Se fuori da **w(k,t)** e con M = 0 allora rimpiazzo
    - Altrimenti il *page frame* va in una coda di trasferimento su disco e la lancetta avanza
      - » Alla ricerca di un *page frame* rimpiazzabile direttamente
      - » Quando **N** pagine in coda si trasferisce su disco
  - Se nessun *page frame* è rimpiazzabile allora si sceglie una pagina con M = 0 altrimenti quella cui punta la lancetta

Algorithm	Comment
Optimal	Not implementable, but useful as a benchmark
NRU (Not Recently Used)	Very crude
FIFO (First-In, First-Out)	Might throw out important pages
Second chance	Big improvement over FIFO
Clock	Realistic
LRU (Least Recently Used)	Excellent, but difficult to implement exactly
NFU (Not Frequently Used)	Fairly crude approximation to LRU
Aging	Efficient algorithm that approximates LRU well
Working set	Somewhat expensive to implement
WSClock	Good efficient algorithm

ANOMALIA DI BELADY:

normalmente aggiungendo memoria RAM diminuisce il page fault ed è cosi per la maggior parte dei casi, ma Belady con un contro esempio dimostro che con alcune politiche avveniva il contrario ovvero con l'aumento di page frame aumentavano i page fault.



L'LRU invece è immune all'anomalia di Belady come anche il rimpiazzo ottimale, questi sono detti **stack algorithms** e soddisfano la proprietà: $M(m, r) \subseteq M(m+1, r)$ dove m=numero dei page frame e r=riferimenti, quindi dati gli stessi riferimenti le pagine caricate con m page frame sono sottoinsieme di quelle con m+1 page frame, ovvero con più RAM.

CRITERI DI PROGETTO:

quando si fa il rimpiazzo occorre scegliere tra politiche locali o politiche globali, quelle **locali** effettuano il rimpiazzo nel working set del processo che ha causato il page fault e in questo modo ogni processo a una dimensione fissa di RAM. Con le politiche **globali** invece si può rimpiazzare con il page frame di un qualsiasi processo e quindi la dimensione della RAM disponibile varia nel tempo.

Le politiche **globali** sono più efficaci soprattutto se l'ampiezza del working set varia durante l'esecuzione, perché consentirebbe di avere un page frame di dimensione ottimale anche più grande di quello prefissato se necessario ed eviterebbe il *thrashing,* ma questo significa dover decidere ogni volta quanti page frame dare ad ogni processo.

Le politiche **locali** invece hanno prestazioni inferiori perché se il ws cresce l'allocazione è fissa e questo causa *Thrashing.*

Non tutte le politiche si adattano all'uso in entrambe le varianti.

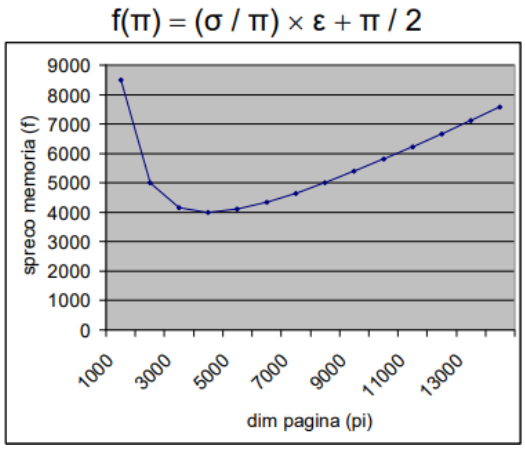
CONTROLLO DEL CARICO: a volte anche le migliori politiche subiscono thrashing, se i working set eccedono la capacità della RAM. Il **PFF***, page fault frequency,* indica che tutti hanno bisogno di più memoria ma nessuno ha bisogno di meno. Si interviene con lo SWAP, rimuovendo alcuni processi finché non si ferma il thrashing.

DIMENSIONE OTTIMALE PER LA PAGINA: usando pagine di dimensione grande si rischia frammentazione interna, siccome ogni processo lascia in media metà del page frame inutilizzato, mentre con pagine piccole si avrà una page table più grande.

Il valore ottima viene definito con la formula:

- σ B dimensione media di un processo
- π B dimensione media di una pagina
- ε B per riga in tabella delle pagine
- Spreco per processo come $f(\pi) = (\sigma / \pi) \times \varepsilon + \pi / 2$
  - Parte di tabella delle pagine + frammentazione interna
  - Derivata prima è $- \sigma \varepsilon / \pi^2 + 1/2$
  - Ponendo uguale a zero si ha che il minimo di $f(\pi)$ si ha per $\pi = \sqrt{(2 \sigma \varepsilon)}$

Esempio per $\sigma = 1$ MB e $\varepsilon = 8$ B:



si ha $\pi = 4$ KB

Per RAM di ampiezza crescente può convenire un valore di π maggiore ma di certo non linearmente, in generale la memoria virtuale non è distinta per dati e istruzioni.

Il S/O compie azioni chiave per la paginazione:

- Ad ogni creazione di processo: per determinare l'ampiezza della sua allocazione e creare la tabella delle pagine corrispondente .
- A ogni cambio di contesto: per caricare MMU e pulire la TLB.
- A ogni page fault: per analizzare il problema e operare il rimpiazzo.
- A ogni terminazione di processo: per rilasciarne i page frame e rimuovere le tabelle delle pagine.

Per trattare un *page fault* bisogna capire quale riferimento sia fallito, il program counter dice l'indirizzo dove si è verificato il problema ma non distingue tra istruzione e operando, a capirlo ci pensa il S/O che annulla lo stato erroneo e ripete da capo l'istruzione fallita:

1.L'hadware fa un trap al kernel e salva il PC sullo stack;

2. Un programma assembler salva i dati nei registri e poi chiama il sitema operativo;

3. Il SO scopre il page fault e cerca di capire quale pagine lo abbia causato;

4. Ottenuto l'indirizzo virtuale causa del page fault, il SO verifica che si tratti di indirizzo valido e cerca page frame rimpiazzabile;

5. Se il page frame è sporco, si imposta il suo spostamento su disco;

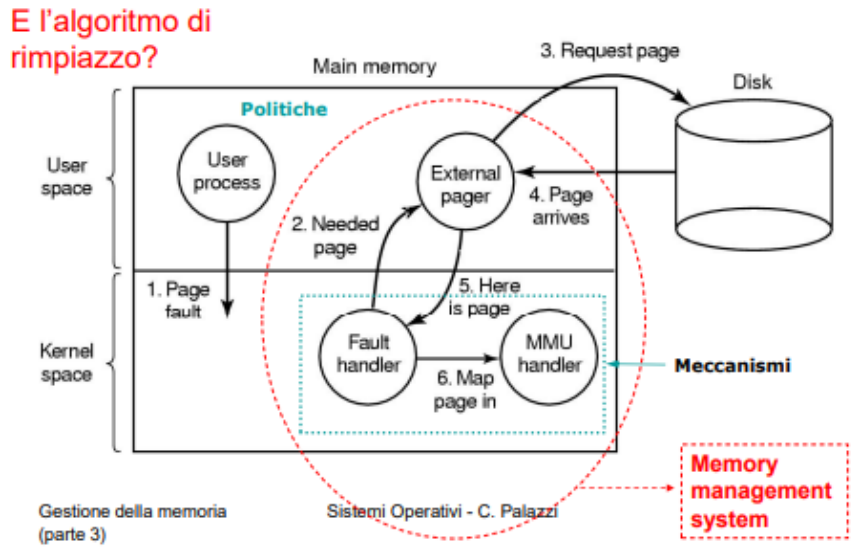
6. Quando il page frame è libero, vi copia la pagina richiesta;

7. All'arrivo dell'interrupt del disco, la page table è aggiornata e il frame è indicato come normale;

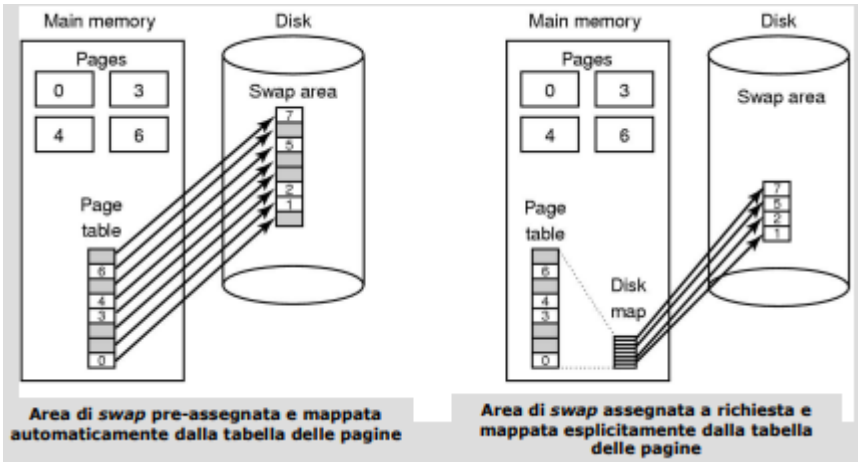
8. Il PC viene reimpostato per puntare all'istruzione causa page fault;

9. Il processo causa del page fault è pronto per l'esecuzione e il SO ritorna al programma assembler che lo aveva chiamato;

10. Il programma assembler ricarica i registri e altre info, poi torna in user space per continauare l'esecuzione.



Area di **swap:** area del disco che è riservata per ospitare le pagine temporaneamente rimpiazzate, viene data ad ogni processo una frazione che rilascia quando finisce e i puntatori *base e ampiezza* relativi a questa zona vengono salvati nella tabella delle pagine del processo. Di solito l'intera immagine del processo va nell'area di *swap* alla creazione del processo, oppure andare in RAM e poi finirci quando necessario. Inoltre l'area di swap è frazionata per codice e dati visto che i processi non hanno dimensione costante, se non venisse riservata l'area di swap bisognerebbe ricordare in RAM l'indirizzo su disco di ogni pagina rimpiazzata(informazione associata alle page table).



LINUX:

- Partizione dedicata allo swap con file system dedicato -> consumo di una possibile partizione del disco.
- Dimensione impostabile dall'utente durante l'installazione. Deve essere grande almeno quanto l'intera RAM per poter gestire l'ibernazione(ovvero la copi di tutto il contenuto della RAM nell'area di swap per poi ricaricarlo nella fase di riattivazione).

WINDOWS:

- Uso di file di swap, hiberfil.sys (usato per copiare la RAM in caso di ibernazione del sistema) e pagefile.sys (usato quando la memoria RAM non è sufficiente). Se questi file vengono frammentati le prestazioni calano.

Per separare le politiche dai meccanismi occorre svolgere nel nucleo del SO solo le azioni più delicate, come:

- Gestione della MMU, che è specifica dell'architettura hardware.
- Trattamento immediato dei page fault, indipendente dall'hardware.

E dando il resto della gestione a un processo esterno al nucleo, scelta delle pagine e del loro trasferimento.

SEGMENTAZIONE:

Caratterizzata da spazi di indirizzamento completamente indipendenti tra di loro sia per posizione che dimensione in RAM.

Entità logica nota al programmatore e destinata a contenere informazioni condivise, codice di una procedura, dati di inizializzazione di un processo, stack di un processo.

Si presta a schemi di protezione specifici specifici perché il tipo del suo contenuto può essere stabilito a priori diversamente dalla programmazione. E causa frammentazione esterna.

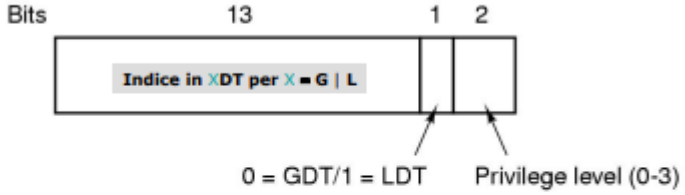
Differenza tra paginazione e segmentazione:

	Paginazione	Segmentazione
Il programmatore ne deve essere consapevole	No	Si
Consente N spazi di indirizzamento lineari	N = 1	N ≥ 1
La sua ampiezza può eccedere la capacità della RAM	Si	Si
Consente di separare e distinguere tra codice e dati	No	Si
Consente di gestire contenuti a dimensione variabile nel tempo	No	Si
Consente di condividere parti di programmi tra processi	No	Si
A quale obiettivo risponde	Consentire spazi di indirizzamento più grandi della RAM	Consentire la separazione logica tra aree dei processi e la loro protezione specifica

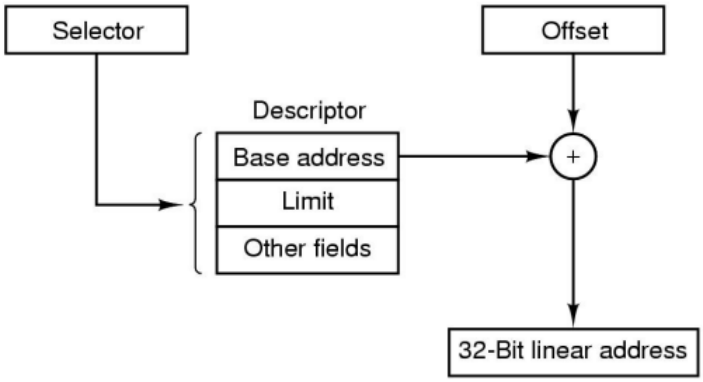
Vista la grande ampiezza potenziale i segmenti sono spesso paginato, come nel caso del Pentium di Intel:

- Fino a 16 K segmenti indipendenti
  - Di ampiezza massima 4 GB (32 *bit*)
- Una LDT per processo
  - *Local Descriptor Table*
    - Descrive i segmenti del processo
- Una singola GDT per l'intero sistema
  - *Global Descriptor Table*
    - Descrive i segmenti del S/O

Per accedere a un segmento, un programma Pentium prima carica selettore di quel segmento in uno dei sei registri di segmento

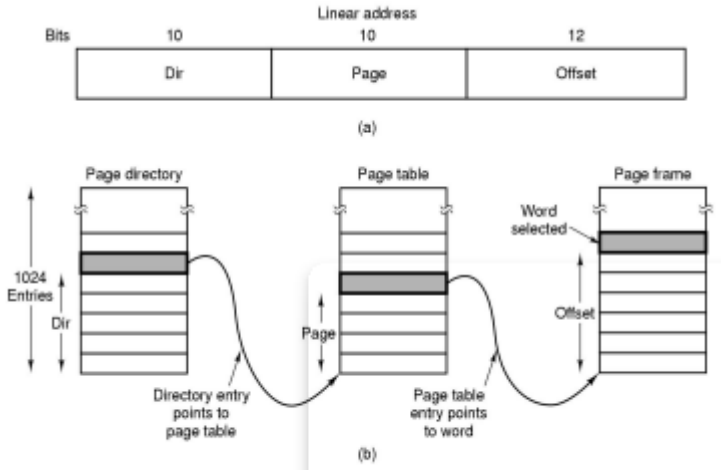


- 6 registri di segmento
  - Di cui 1 denota il segmento corrente
- LDT e GDT contengono $2^{13} = 8$ K descrittori di segmento
  - I descrittori di segmento sono espressi su 8 B
    - La **base** del segmento in RAM è espressa su 32 *bit*
    - Il **limite** su 20 *bit* per verificare la legalità dell'*offset* fornito dal processo
      - Consente ampiezza massima a 1 MB (per **granularità** a B)
      - Oppure 1 M pagine da 4 KB ovvero 4 GB (per granularità a pagine)



Conversion di una coppia (selettore, offset) in un indirizzo lineare

L'indirizzo lineare è ottenuto da *base di segmento + offset* e può essere interpretato come: -Indirizzo Fisico se il segmento considerato non è paginato; -Indirizzo Logico se il segmento è paginato. In questo caso il segmento viene visto come una memoria virtuale paginata e l'indirizzo come virtuale in essa.



- L'indirizzo **lineare** mappato sullo spazio virtuale

FILE SYSTEM:

La maggior parte delle info applicative ha hanno durata, ambito e dimensione più ampie delle applicazioni che la usano.

Il ***file system*** è un servizio del S/O che soddisfa i bisogni relativi ai dati:

- Persistenza dei dati.
- La possibilità di condividere dati tra applicazioni diverse.
- Nessun limite do dimensione fissato.

Un **file** è un insieme di dati correlati tra di loro che risiedono in memoria secondaria e vendono trattati insieme.

Il file system si occupa dei file in termini di organizzazione, gestione, realizzazione e accesso. La progettazione dei file system affronta due problemi, **cosa** e **come**.

- **Cosa** offrire all'utente e in quale modo: modalità di accesso ai file, la loro struttura logica e fisica e le operazioni che l'utente può farci.
- **Come** offrire queste cose in modo che sia indipendente dall'architettura fisica, pratico ed economico.

Il **file:** è un concetto logico realizzato tramite meccanismi di astrazione per salvare informazioni e poterle ritrovare in seguito senza conoscere la struttura fisica e logica, né il funzionamento.

All'utente interesse solo poter salvare e trovare i suoi file attraverso dei nomi logici(nome del file) unici. Ogni file ha delle **caratteristiche** distintive: attributi, struttura interna e operazioni ammesse su di esso.

Il nome del file è composto da 8 a 255 caratteri ed un estensione di >= 1 caratteri che designa il tipo d file, ovvero come viene visto dall'utente.

MS-DOS Nomi da 1 – 8 caratteri e case insensitive. | UNIX Nomi fino a 14 e case sensitive.

Altre proprietà di un file sono:

- Dimensione corrente: quanto pesa.
- Data di creazione: data del primo salvataggio e può non essere mostrata.
- Data dell'ultima modifica.
- Creatore e possessore: possono essere diverse, esp: il compilatore che crea un file che appartiene all'utente.
- Permessi di accesso: lettura scrittura ed esecuzione.

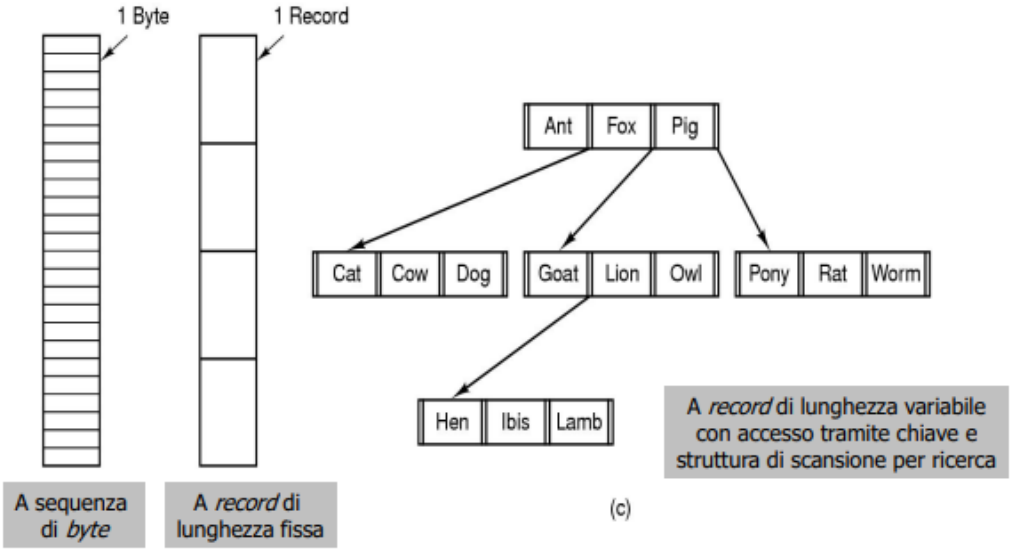
Protezione	Permesso di accesso al *file*	*Flag*
Password	Chiave di accesso al *file*	
Creatore	Identità del processo che ha creato il *file*	
Proprietario	Identità del processo utilizzatore del *file*	
Uso	0 – lettura/scrittura	1 – sola lettura (*read-only*)
Visibilità	0 – normale	1 – *file* non visibile (*hidden*)
Livello	0 – normale	1 – *file* di sistema
Archiviazione	0 – salvato (*backed up*)	1 – non salvato
Tipo di contenuto	0 – ASCII	1 – binario
Tipo di accesso	0 – sequenziale	1 – casuale (*random*)
Permanenza	0 – normale	1 – da eliminare dopo l'uso (*temporary*)
Accesso esclusivo	0 – libero	≠ 0 – bloccato (*locked*)

Ci sono 3 modi per vedere la struttura dei dati all'interno di un file:

- Livello **utente:** il programma applicativo associa automaticamente il significato al contenuto grezzo del file.
- Livello di **struttura logica:** a discapito dell'interpretazione dell'utente il S/O organizza i dati grezzi in strutture logiche facili da trattare.
- Livello di struttura **fisica**: il sistema operativo mappa le strutture logiche sulle strutture fisiche presenti(settori o blocchi del disco).

Le diverse strutture **logiche** possibili:

- Sequenza di byte (usata su linux e windows).
- A record di lunghezza e struttura interna fissa.
- A record di lunghezza e struttura interna variabile.



Sequenza di byte: struttura più rudimentale e flessibile infatti è la scelta di UNIX e MS windows, il programma applicativo sa come dare significato al contenuto del file con il minimo sforzo. L'accesso ai dati utilizza un puntatore che punta all'inizio del file e la lettura e la scrittura avvengono a blocchi di byte.

Record di lunghezza variabile: il SO deve conoscere la struttura interna del file per muoversi al sui interno, c'è un accesso sequenziale ai dati con un puntatore al record corrente. Lettura e scrittura avvengono su record singoli e gli spazi vuoti sono occupati da caratteri speciali di tipo(*null o space).*

Record di lunghezza variabile: la struttura interna di ogni record viene descritta e identificata da una **key** in una posizione fissa e nota entro il record che viene usata per accedere ai dati. Le chiavi sono raccolte in una tabella con assieme i puntatori di ciascun record. Diffuso in sistemi *mainframe.*

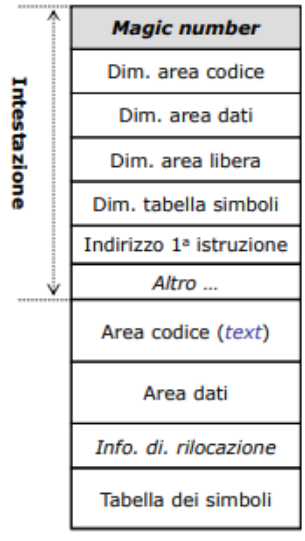
MODALITA' DI ACCESSO:

- **Accesso sequenziale**: vien trattato un gruppo di byte o record alla volta, un puntatore riferisce il record corrente e avanza ad ogni operazione. La lettura può essere fatta in qualsiasi punto mentre la scrittura solo in coda(*append)* perché sul file si può operare solo sequenzialmente, ad ogni nuova operazione il puntatore riparte dall'inizio.
- **Accesso diretto**: opera su record di dati posti in posizione arbitraria nel file rispetto alla base offset=0.
- **Accesso indicizzato**: per ogni file c'è una tabella di chiavi con gli offset dei record nel file quindi le informazioni di navigazione non sono più nei record ma in una struttura ad accesso veloce(*hash, principio delle basi di dati).* Si fa una ricerca binaria della chiave e poi un accesso diretto. **ISAM**(index sequential access method) consente sia indicizzato che sequenziale.

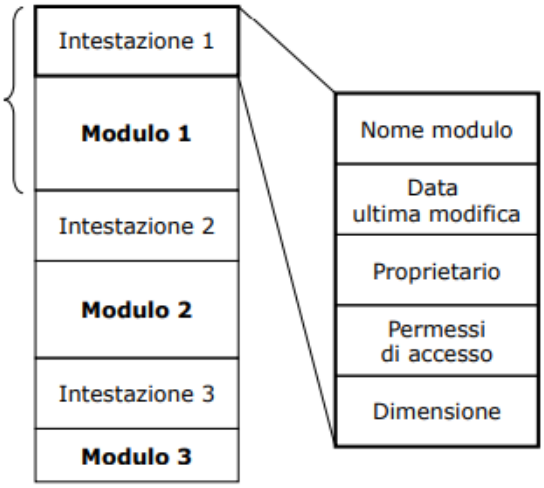
TIPOLOGIA DI FILE TRATTATI DAL FS:

- File **regolari:** sui quali l'utente può operare normalmente. (WIN + UNIX/GNU/Linux)
- File **catalogo(directory):** con cui il file system permette di descrivere l'organizzazione di gruppi di file. (WIN + UNIX/GNU/Linux)
- File speciali: con cui il file system rappresenta logicamente dispositivi orientati a carattere o blocco(terminale o disco). (Linux users only!).

File binari in UNIX e GNU/Linux



Struttura di un *file* eseguibile



Struttura di un *file* archivio
(**tar** : *tape archive*)

Possibili operazioni su file:

- Creazione: file vuoto con attributi inizializzati.
- Apertura: predisposizione delle informazioni necessarie per l'accesso.
- Cerca posizione: solo per accesso casuale.
- Cambia nome: *rename*, puo causare spostamenti nella struttura logica del file system.
- Distruzione: tolto dalla memoria.
- Chiusura: rilascio delle strutture usate per l'accesso al file.
- Lettura e scrittura.
- Trova e modifica attributi (permessi).

Sessione d'uso di un file: accesso al file possibile solo se già aperto, all'apertura di un file il SO predispone un strumento specifico per l'accesso, *handle.* Dopo l'utilizzo il file deve essere chiuso. UNIX/LINUX tiene una tabella dei file aperti con: informazioni sui file comuni a processi diversi e dei dati relativi ad un processo specifico.

FILE MAPPATI IN MEMORIA:

È possibile mappare un file in memoria virtuale ovvero tenere il file in memoria secondaria e far corrispondere ad ogni suo dato un indirizzo in memoria virtuale, composto da base+*offset.*

Con la segmentazione si potrebbe avere il file in un unico segmento così da utilizzare lo stesso offset.

Le operazioni vengono fatte in memoria principale poi alla fine della sessione vengono salvate anche in quella secondaria.

L'uso della memoria virtuale riduce gli accessi al disco, quindi elaborazione più veloce, ma è problematica quando si tratta di file di grosse dimensioni.

STRUTTURA DI DIRECTORY:

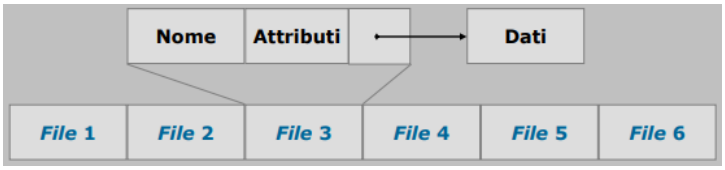
Ogni file usa una *directory* o un *folder* per tenere traccia dei suoi file regolari.

I requisiti dell'utente sono:

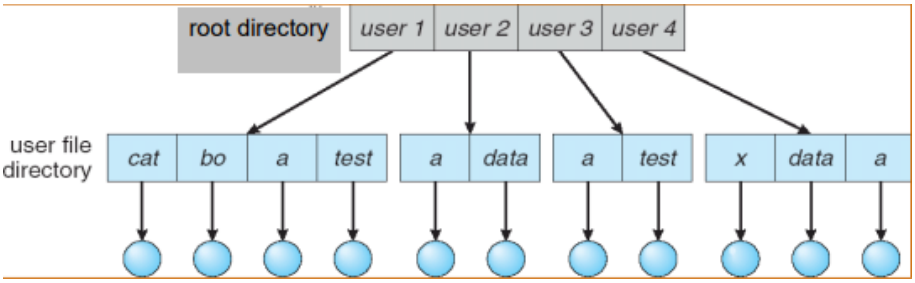
- Efficienza: creare e trovare un file deve essere semplice e veloce.
- Libertà di denominazione: lo stesso nome deve poter essere utilizzato da utenti diversi, oggi in cartelle diverse. E un file può essere visto con un nome diverso da utenti diversi.
- Libertà di raggruppamento: creare connessioni, gruppi logici, tra file in base a delle proprietà specifiche.

Le directory possono essere organizzate rispetto all'organizzazione di file che esse consentono:

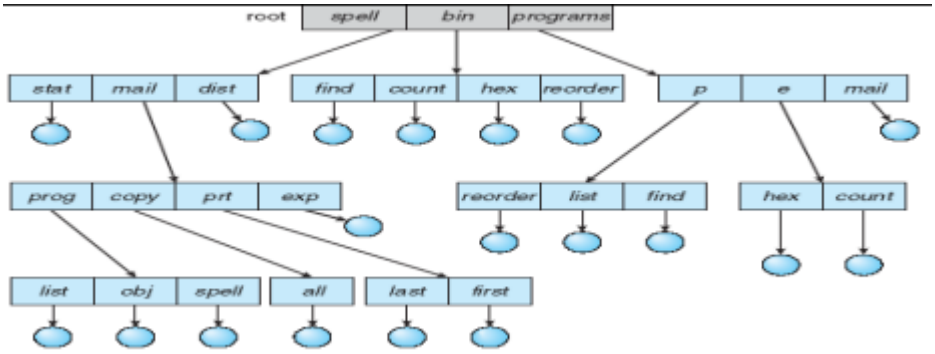
- **A livello singolo:**
  tutti i file sono disposti da elenco in un'unica lista, *root directory,* ciascuno con il proprio nome. Questo permette ricerca facile ed una realizzazione semplice ma una gestione onerosa all'aumento dei file.



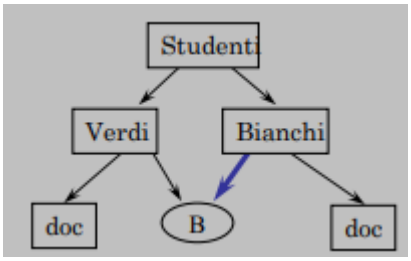
- **A due livelli:**
  una *User File Directory* per ciascun utente di sistema è contenuta in una *Root Directory,* ogni utente può vedere solo la sua UFD e i suoi file sono localizzati attraverso un *path name.* i file di sistema per essere visti da tutti gli utenti vengono messi su una directory condivisa e localizzati attraverso cammini predefiniti oppure copiati su ogni UFD.



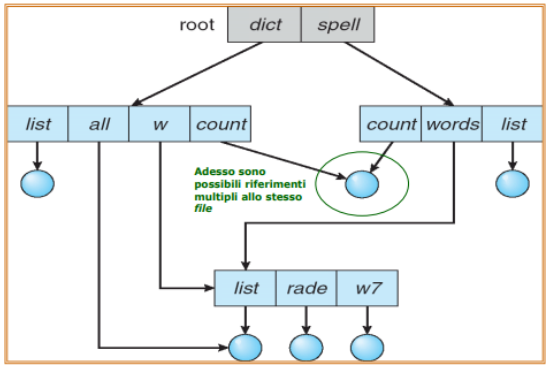
  Le directory a due livelli permettono di avere una ricerca efficace e una libera denominazione ma non un riferimento multiutente allo stesso file o un raggruppamento libero.
- **Ad albero:**
  le directory sono raggruppate a livelli ogni livello contiene file e/o altre directory che definiscono livelli inferiori, il livello più in alto da cui partono tutti rami è detto *root*. La navigazione è semplice, si procede di cartella in cartella se non si specifica il percorso l'operazione viene fatta sulla cartella corrente. Un cammino può essere **assoluto** ovvero che parte dalla radice(*root)* oppure **relativo** ovvero dalla alla posizione corrente.



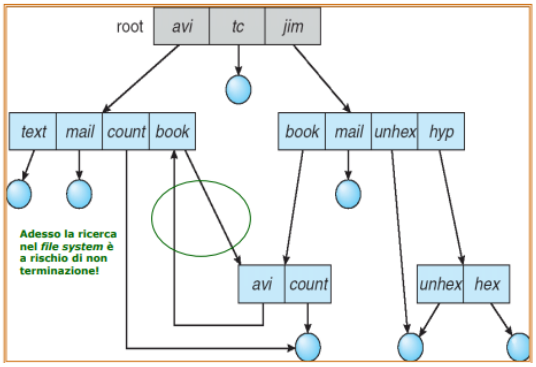
  Questo tipo di raggruppamento permette una ricerca efficiente, libertà di denominazione e di raggruppamneto.
- **A grafo aperto:**
  simile a quello ad albero ma con la presenza di cicli(directory che si riferisce ad un'altra directory e viceversa) e riferimenti doppi, un file può appartenere a più directory in contemporanea. SO cone unix e linux utilizzano collegamenti simbolici(*link*) tra nome del file e la sua presenza virtuale. Il SO potrebbe duplicare gli identificatori di accesso al file rendendo più difficile assicurarsi della sua coerenza.



Struttura a grafo aciclico



Struttura a grafo generalizzato



Hard link:

Un puntatore diretto a un file regolare viene inserito in una directory a esso remota che deve risiedere nello stesso FS del file. Questo crea 2 vie d'accesso distinte dirette a uno stesso file

Simbolic link:

Viene creato un file speciale il cui contenuto è il cammino del file originario, il file originario può avere qualunque tipo e risiedere anche in un FS remoto. Questo riferimento mantiene 1 sola via d'accesso al file originario

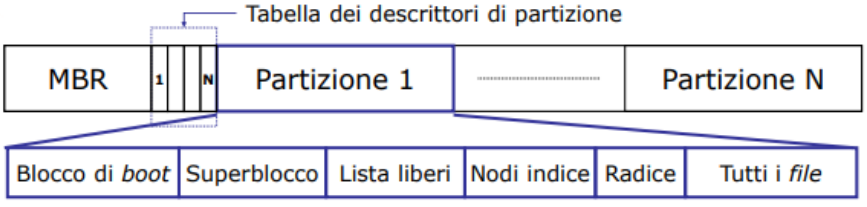
Operazioni su directory GNU/Linux:

Crea *directory*	`mkdir`	Create
Cancella *directory*	`rmdir`	Delete
Cambia nome a *directory*	`mv`	Rename
Apri, chiudi, leggi *directory*		Opendir, Closedir, Readdir
Crea collegamento a *file*	`ln`	Link
Rimuovi collegamento a *file*	`rm`	Unlink

I **file system** sono memorizzati su disco in una partizione, un disco può avere più partizioni quindi contenere più file sytem. Nel settore n.0 si trovano le info per l'inizializzazione del sistema che viene fatta dal BIOS, il **MBR(master boot record**) contiene una descrizione delle attività da svolgere.

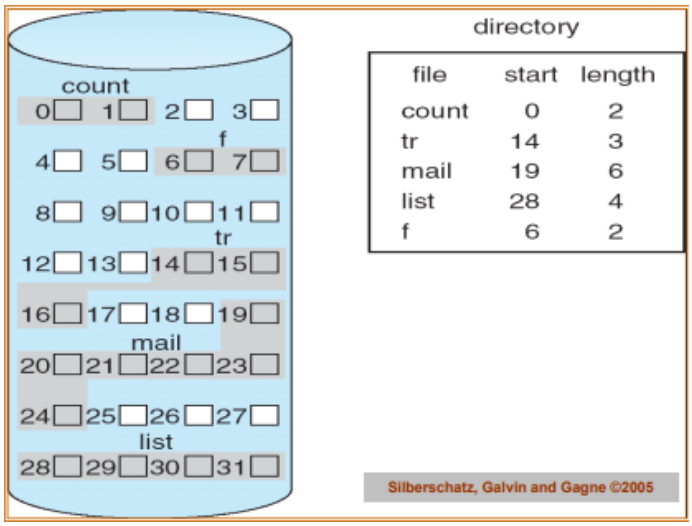
Boot block: il primo blocco di ogni partizione e contiene le informazioni di inizializzazione.

I dischi sono suddivisi in settori però vengono letti e scritti a blocchi, ogni blocco equivale a N settori, c'è quindi un rischio di frammentazione interna. La struttura interna della partizione dipende dal FS.

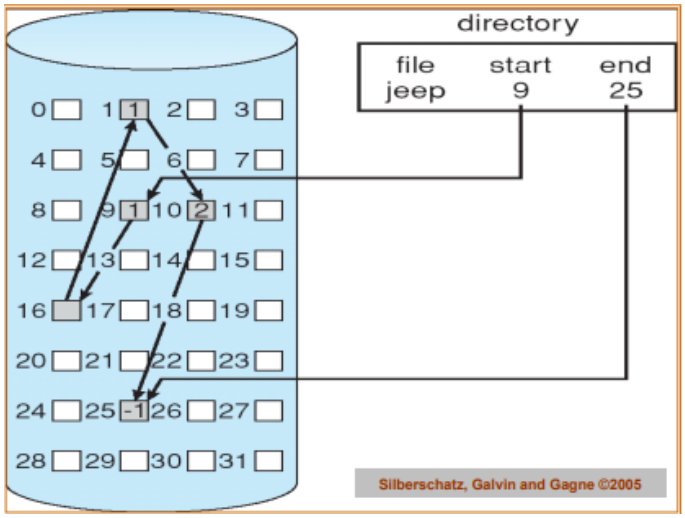


Realizzazione dei file: un file a livello fisico è un insieme di blocchi su disco e ci sono diverse strategie per allocare i blocchi dei file sul disco:

- **Allocazione contigua:** memorizza i file in blocchi consecutivi, ogni file è descritto dal blocco di inizio e dal numero di blocchi che occupa. Questo permette un accesso sia casuale che diretto e quindi un file può essere scritto/letto con un solo accesso al disco.
  Questa tecnica comporta rischio di frammentazione esterna che richiede ricompattazione periodica(molto costosa) oppure mantenendo un lista dei blocchi liberi e della loro dimensione e conoscendo la dimensione dei file futuri(anche questa costosa e rischiosa).

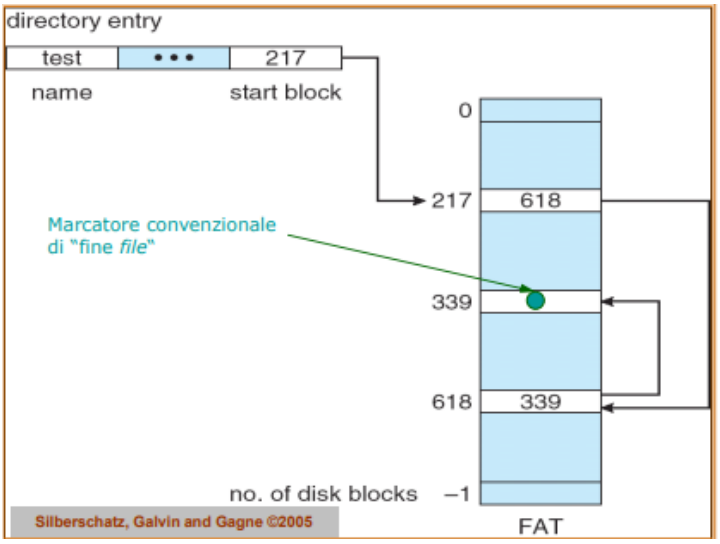


- **Allocazione a lista concatenata:** il file è visto come una lista consecutiva di blocchi collegati tra di loro dove il file è identificato da un puntatore iniziale e ogni blocco contiene il riferimento a quello successivo. Questa tecnica ha molti problemi, l'accesso diretto richiede il passaggio ed il caricamento dei blocchi precedenti, i puntatori occupano spazio e il guasto di un blocco fa perdere tutto quello che c'è dopo del file.

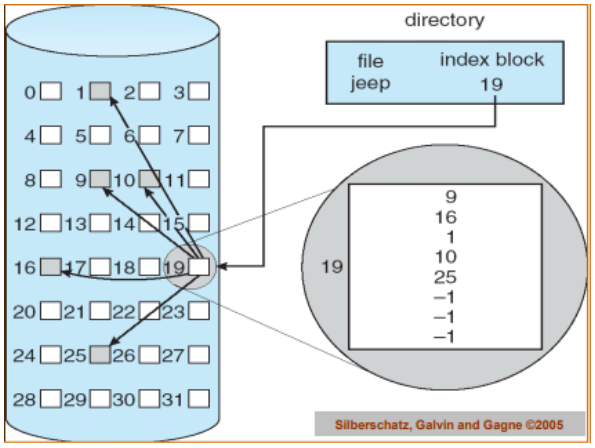


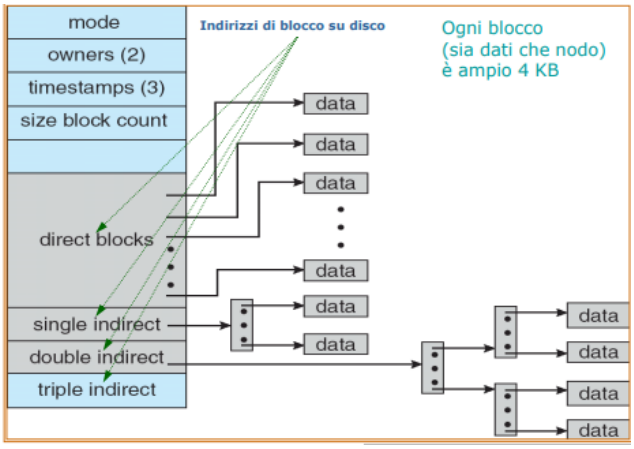
- **Allocazione a lista indicizzata:** i puntatori dei blocchi sono messi in strutture apposite, ogni blocco è composto solo da dati e il file è descritto dall'insieme dei puntatori che lo compongono. Queste tecniche non causano frammentazione, consentono accesso diretto e sequenziale e non richiedono la conoscenza della dimensione dei file in anticipo.
  Ci sono due strategie:
- **FAT:** file allocate table, è la base di MS-DOS, è una tabella ordinata di puntatori a blocchi(*cluster*) che cresce con l'ampiezza della partizione. La FAT relativa al file in uso deve risiedere completamente in RAM, permette l'accesso diretto senza spreco di tempo perché

trova il puntatore del blocco direttamene in RAM e poi accede al disco una sola volta. Il file è una catena di indici.



- **I-NODE:** per unix ->gnu/linux c'è una struttura indice per ogni file con attributi file e puntatori ai blocchi, l'i-node è contenuto un blocco dedicato. In RAM c'è una tabella di i-node dei file in uso, la dimensione della tabella dipende dal massimo numero di file che sono apribili simultaneamente(non la capacità della partizione).
  un i-node può contenere un numero limitato di puntatori a blocchi quindi per usare file con un numero maggiore di blocchi occorre implementare una struttura a nodi con i-node contenti altri i node:



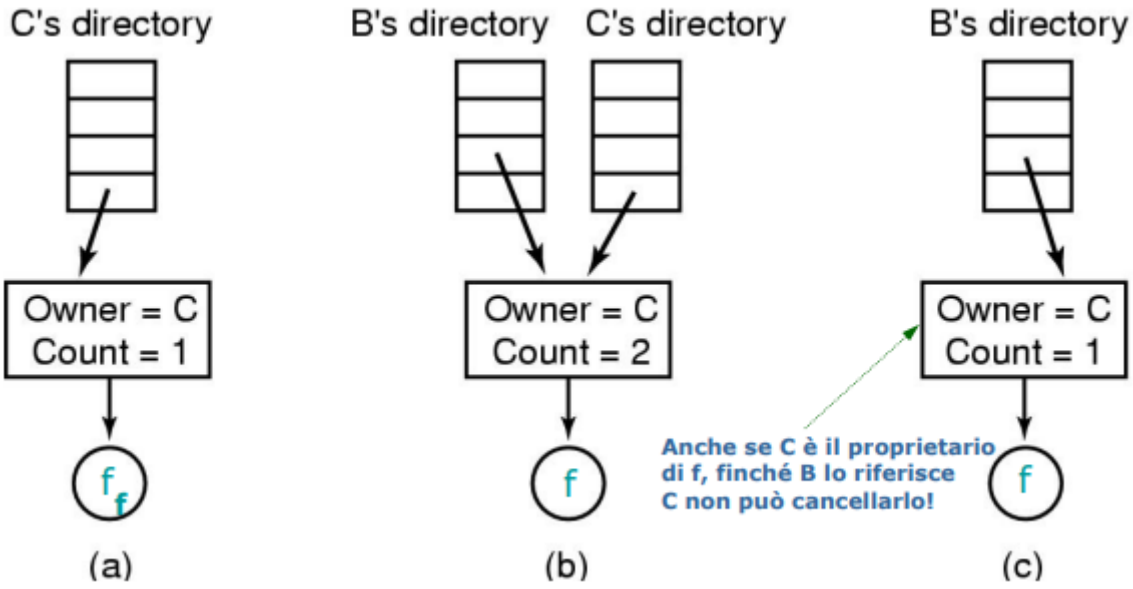


Gestione dei file condivisi:

Per preservare la consistenza dei file senza costi aggiuntivi di calcolo e di memoria senza porre blocchi di dati nella directory di residenza del file ci sono due metodi:

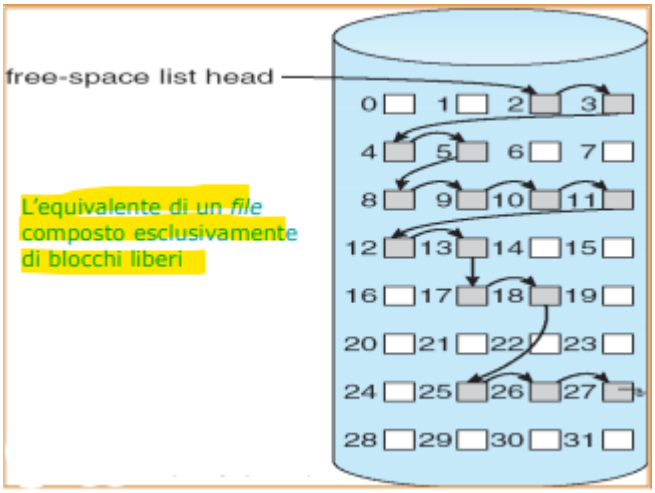
Per ogni file condiviso occorre porre la *directory* remota un *symbolic link* verso il file originale, così esisterà solo un descrittore del file originale e l'accesso condiviso avvera tramite il *path* del file system.

Oppure ponendo nella directory remota il puntatore diretto *hard link* al descrittore i-node del file originale, in questo modi ci saranno più possessori del descrittore dello stesso file ma un solo proprietario effettivo del file condiviso. Il file in questione non può essere cancellato nemmeno dal proprietario finché ci saranno dei descrittori attivi.



GESTIONE DEI BLOCCHI LIBERI.

? I blocchi sono associati ad una *bitmap* dove ogni bit indica lo stato di libero o occupato(0 | 1), nella FAT i blocchi sono messi in una lista concatenata collegati da un puntatore al successivo.



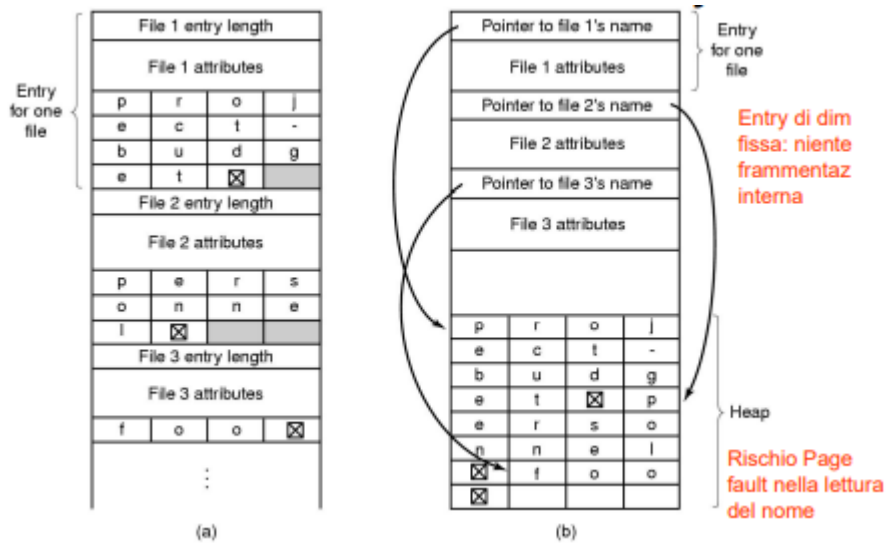
GESTIONE SPAZIO SU DISCO.

Nella RAM c'è un blocco di puntatori a blocchi liberi se cancello un file di tre blocchi e ne riscrivo un altro da 3 devo ricaricare i blocchi di prima. Un'altra strategia consiste nel dividere i puntatori ai blocchi tra RAM e disco una volta finto un insieme si fa lo scambio.

REALIZZAZIONE DELLE DIRECTORY.

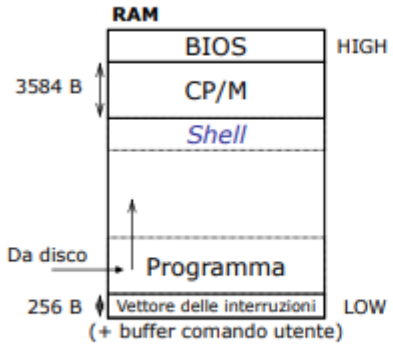
Una directory fornisce le informazioni su nome, collocazione e attributi dei file relativi ad un catalogo, file e directory risiedono in aree logiche distinte. Meglio minimizzare la complessità della directory, conviene una struttura a lunghezza fissa con un contenuto di ampiezza variabile. *[Nome + attributi]* oppure *[Nome + puntatore a nodo indice con attributi].*

La frammentazione interna diventa un problema solo superati gli 8 caratteri per il nome del file.



PROSPETTIVA STORICA.

CP/M:

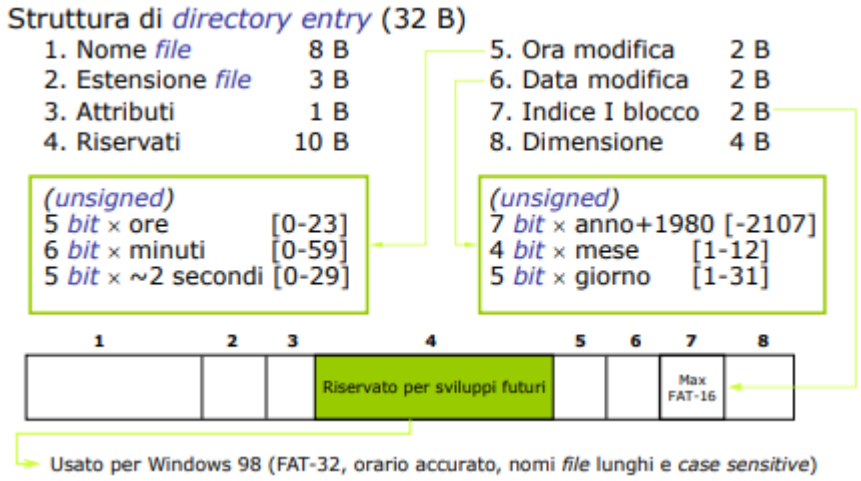


- BIOS minimo
  - 17 I/O calls (massima portabilità)
- Sistema multiprogrammato
  - Ogni utente vede solo i propri *file*
- *Directory* singola con dati a struttura fissa (32 B entry)
  - In RAM solo quando serve
- *Bitmap* in RAM per blocchi di disco liberi
  - Distrutta a fine esecuzione
- Nome *file* limitato a 8 + 3 caratteri
  - Dimensione inizialmente limitata a 16 blocchi da 1 KB
    - Puntati da *directory*

MS-DOS:

- **Non** multiprogrammato
  - Ogni utente vede **tutto** il FS
- FS **gerarchico** senza limite di profondità e **senza condivisione**
  - Fino a 4 partizioni per disco (C: D: E: F:)
- *Directory* a lunghezza variabile con *entry* di 32 B
  - Nomi di *file* a 8+3 caratteri (normalizzati a maiuscolo)
- Allocazione *file* a lista (FAT)
  - **FAT-X** per **X** = numero di *bit* per indirizzo di blocco ($12 \leq$ **X** <32)
  - Blocchi/Cluster di dimensione multipla di 512 B;
    - Max partition size è 2^{12} x 512B = 2MB
    - Estendendo blocchi fino a 4KB si arriva a 16 MB max
  - **FAT-16** *: File* e partizione limitati a 2 GB
    - 2^{16} = 64K (puntatori a) blocchi di 32 KB ciascuno = 2 GB
  - **FAT-32** : blocchi da 4 ÷ 32 KB e indirizzi da 28 *bit* (!)
    - Perché 2 TB è il limite **intrinseco** di capacità per partizione Win95
      - 2^{32} settori (*cluster*) da 512 B = $2^2 \times 2^{30} \times 2^9$ B = 2^{41} B = 2 TB
      - 2^{28} blocchi da **8** KB = $2^8 \times 2^{20} \times 2^3 \times 2^{10}$ B = 2^{41} B = 2 TB

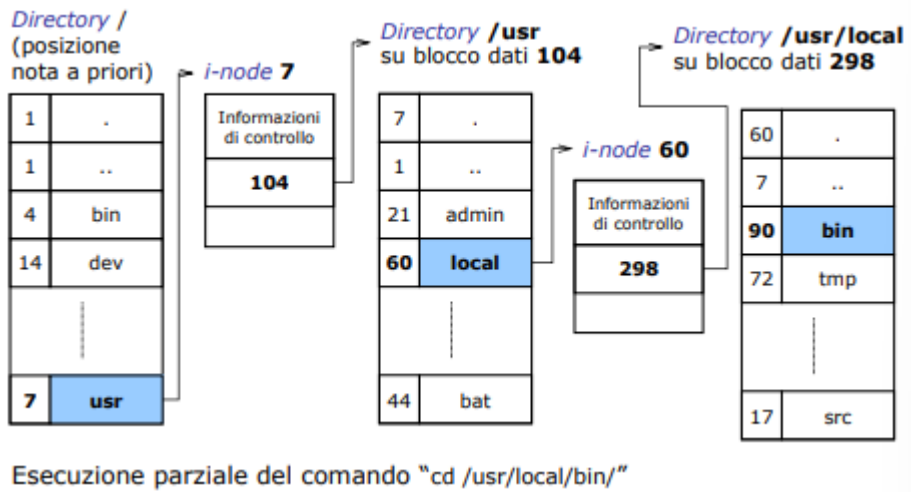
File System in MS-DOS:



UNIX: *da ken thompson e dennis ritchie.*

- Struttura ad albero con radice e condivisione di *file*
  - Grafo **aciclico**
- Nomi di *file* fino a 14 caratteri ASCII (escluso / e NUL)
- *Directory* contiene nome *file* e puntatore (su 2 B) al suo *i-node* descrittore
  - Max 64 K *file* per FS (2^{16} *i-node* distinti)
- L'*i-node* (64 B) contiene gli attributi del *file*
  - Incluso il contatore di *directory* che puntano al *file* tramite un *link* di tipo *hard*
    - Se contatore = 0, il nodo e i blocchi del *file* diventano liberi

File System in UNIX:



INTEGRITA' DEI FILE SYSTEM.

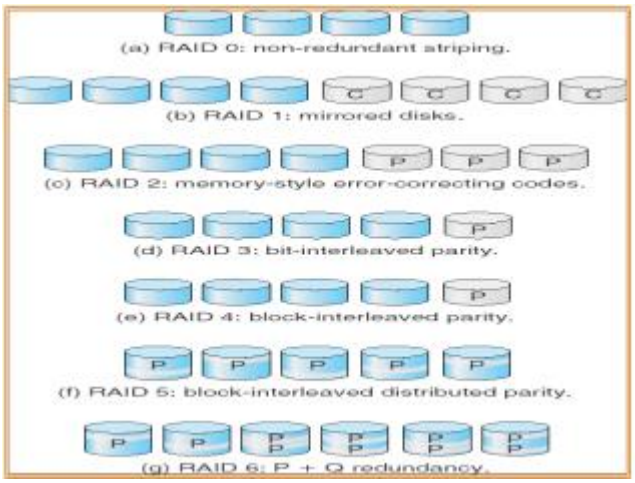
Gestione dei blocchi danneggiati:

- Per via *hardware,* salvando in un settore del disco un elenco dei blocchi danneggiati.
- Per via *software,* occupando con un falso file i blocchi danneggiati

Salvataggio del File System:

- Su nastro, tempi lunghi.
- Su disco con partizione di back-up o con l'uso di architetture RAID(*redundant array of inexpensive/indipendent disks).*

livelli di RAID:



Striping: i dati vengono sezionati (per bit o byte) e ciascuna sezione viene scritta in parallelo su un disco.

Al crescere di livello aumenta la sicurezza. I dischi **C** contengono una copia di altri dischi, i dischi **P** contengono codici di controllo di integrità dei dati di altri dischi.

Consistenza del file system: se il sistema cade mentre il file è tra la fase di modifica e di salvataggio il file diventa inconsistente.

Consistenza dei blocchi: i blocchi hanno i contatori in due liste diverse quella per i blocchi in uso e quella dei blocchi liberi.

Blocco consistente: ovvero che appartiene ad una sola lista.

Perso: se non appartiene a nessuna lista.

Duplicato: il counter è maggiore di 1 in una lista.

Una porzione della memoria principale viene usata come cache di alcuni blocchi per ridurre l'accesso al disco, accesso avviene tramite ricerca *hash* e la sua gestione richiede una specifica politica di rimpiazzo.

Occorre garantire la consistenza dei dati su disco:

- Su **MS-DOS:** i blocchi modificati vengono copiati immediatamente su disco(*write through),* alto costo ma consistenza garantita.
- Su **UNIX/LINUX:** un proesso periodico sync effettua l'aggiornamento dei blocchi su disco. Basso costo e rischio basso se usati dischi affidabili