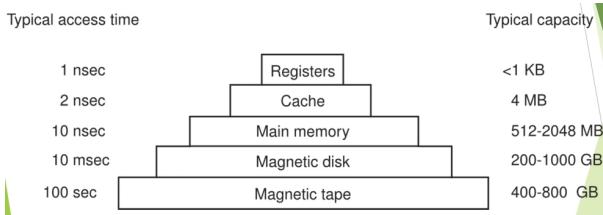


11-05-2023

Gestione della memoria

La gestione della memoria è un aspetto molto delicato visto che il calcolatore lavora con la CPU e la memoria.

La memoria si suddivide in varie parti secondo una determinata **gerarchia**.



Più è grande la memoria, più è lenta e viceversa

Si sfrutta ogni singola memoria in modo gerarchico.

Quando si parlava di processi, in un sistema multiprocessore, tentano di far eseguire i successivi processi usando la stessa area di memoria.

Le **memorie più piccole** contengono i dati che risultano più **RECENTI** e quindi sono **usati più spesso** in modo da avere risposte più veloci e questa caratteristica deve essere *sfruttata al massimo*.

La **RAM** (che è **FINITA**) contiene *tutto quello che deve essere mandato in esecuzione*.

Il **gestore della memoria** è quella parte del SO che gestisce tutto ciò che avviene all'interno della memoria considerando varie sfaccettature:

- *Allocazione* della memoria, tenere traccia di cosa (*blocchi di memoria*) è/non è occupato
- *Deallocazione* della memoria
- *Dove posizionare* gli elementi
- *Prelevare i dati* dal disco e caricarli (e *in che modo caricarli*)

Per la gestione di quanto sopra, serve un'**astrazione della memoria a favore dei processi**.

Quando si alloca uno spazio di memoria, raramente tutta la memoria destinata ad un programma viene occupata e molto spesso si occupa uno spazio di memoria più piccolo dello spazio di memoria effettivamente allocato per tale programma.

Quindi internamente c'è uno spazio di memoria sprecato.

"Allora un blocco di memoria come deve essere rappresentato?"

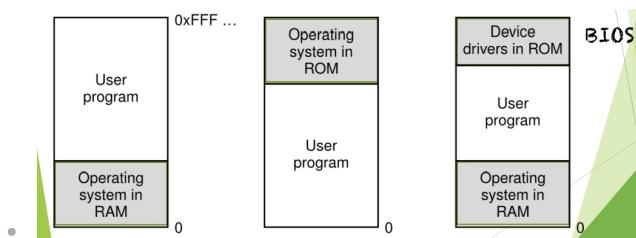
Si fanno quindi determinate scelte per evitare tali problematiche e ridurre al minimo lo spreco

Può capitare che fra 2 blocchi di memoria ci sia un piccolo spazio di memoria non allocato e non allocabile da nessun programma perché troppo piccolo. Anche questo può rappresentare uno spreco di memoria.

I primi PC non avevano un'astrazione della memoria e quindi i programmi operavano direttamente sugli indirizzi fisici, quindi **sulla memoria direttamente** (se si hanno 2 programmi in memoria, possono usare indirizzi che vanno in conflitto fra loro), che partivano da 0 fino alla dimensione massima. C'era il limite di **non poter contenere più di un programma in memoria allo stesso tempo** perché in questo caso nasceva il problema del conflitto di indirizzo.

- I programmi potevano prendere indirizzi che danneggiavano anche il SO

Vi erano diversi modelli (da sx a dx -> A B C):



- (A) Venivano allocati i **programmi e il SO all'inizio della RAM (indirizzo 0, figura a sx)**
- (B) Si memorizzava **tutto il SO all'interno della RAM** (sistemi *embedded*) e tutto il resto era libero ai programmi dell'utente (*figura centrale*)
- (C) I **driver più importanti nella ROM** mentre il **SO all'inizio della RAM** (MS-DOS, *figura a dx*)
 - Il **BIOS** è quella parte dei drivers presente nella ROM

Il modello A e C hanno un problema: il SO è nella RAM e c'è il rischio di conflitti con i programmi utente possono interferire con il SO (potrebbero anche cancellare file importanti). Nel modello B c'è più protezione sotto questo punto di vista.

Per questo motivo **si può eseguire solo un processo alla volta**.

Problemi dei vecchi sistemi:

- Per avere una **gestione più efficiente** i SO si sono evoluti per cercare di **eseguire più programmi contemporaneamente (o quasi)**
- Serviva fare uno swap fra memoria->disco e viceversa
- Caricano un programma alla volta in RAM

Mantenere i dati e più programmi sulla RAM velocizza le operazioni e quindi si diminuiscono gli accessi all'hard disk ma ci sono dei problemi:

- PROTEZIONE DELLA MEMORIA:** presenza di più processi nella RAM e nascono conflitti visto che lavorano su indirizzi fisici (*un processo accede agli indirizzi fisici di un altro processo*)
 - SOLUZIONE:** lock & key -> prende la memoria e la divide in blocchi. Ogni blocco contiene, in una parte, **i dati** e, dall'altra parte, **un'identificativo (un numero)** per quel blocco di memoria dei dati. Si ha anche la **PSW** di ogni processo in esecuzione (**CHIAVE DI PROTEZIONE** di ogni processo).
 - Quando il SO deve operare, vede se la chiave memorizzata nel PSW del processo corrisponde al blocco sul quale vuole operare e in tal caso fa accedere alla memoria ed esegue le operazioni*
 - in questo modo si evita che un processo interferisce su memorie di altri processi
 - Questa *non è la soluzione migliore* perché, usando anche dello spazio di memoria per la PSW, **si perde memoria disponibile** e i programmi lavorano direttamente su **indirizzi fisici**
- RILOCAZIONE DELLA MEMORIA:**
 - STATICI:** si allocano/riallocano i programmi in memoria. Si riallocano, in fase di loading del processo, e si assegnano gli indirizzi. Ma in questo caso, per distinguere gli indirizzi fra i programmi, si aggiunge al caricamento del programma del processo un'**operazione in più** in fase di caricamento (controllare i valori identificativi numerici) e ciò comporta un **RALLENTAMENTO DEL LOADER** dei processi
 - A COMPILE-TIME:** si fa in fase di compilazione in modo da fare un **unico inserimento**. Questo approccio si usa maggiormente nei sistemi *embedded*.

Approccio con lo Spazio degli indirizzi

- Un'altra soluzione per mantenere più processi in memoria allo stesso tempo senza interferenze, si può generare lo **SPAZIO DEGLI INDIRIZZI** ed è un'astrazione della memoria
- Rappresenta l'insieme degli indirizzi che un processo può usare per indirizzare la memoria di suo interesse

Come si assegna lo spazio degli indirizzi ad ogni processo?

Tramite una **rilocazione dinamica** che fa uso di:

- **registro base**: impostato con l'indirizzo fisico a partire dal quale è memorizzato il programma (indirizzo di partenza)
- **registro limite**: contiene la lunghezza del programma
La CPU controlla gli accessi alla memoria in base a questi registri.

Nei sistemi moderni non si usano questi registri ma si usa la **MMU** (Memory Management Unit).

Lo **SVANTAGGIO** di questo approccio è il seguente: ogni volta che si deve allocare, bisogna fare una somma del registro base fino al registro limite e quindi si fa anche un confronto periodicamente.

Anche se il confronto è veloce, la somma non lo è affatto e ciò appesantisce la CPU.

L'Intel 8088, a differenza di prima (CDC 6600), usa **più registri base** ma comunque presenta dei problemi e quindi non è *una soluzione ottimale*.

Sovraccarico della memoria

Nei sistemi moderni il limite è che la **RAM** è comunque **limitata** e i programmi diventano sempre più grandi.

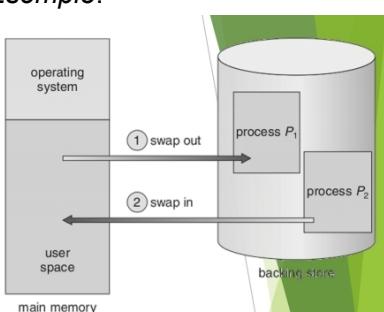
Si deve gestire una situazione dove la **RAM NECESSARIA** per soddisfare tutti i programmi è nettamente superiore rispetto alla **CAPACITA'** della RAM stessa

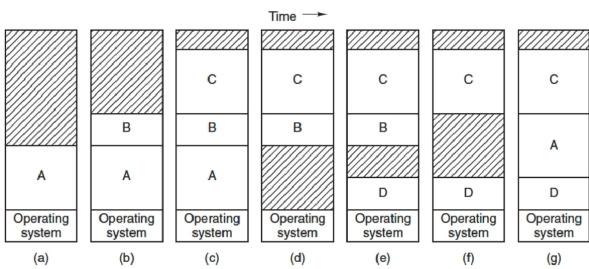
La gestione del sovraccarico si può gestire con 2 approcci: **Swapping** e **Memoria virtuale**

Approccio con Swapping

- Consiste nel prendere ciascun **programma nella sua TOTALITA'** ed eseguirlo per un certo *periodo di tempo* e dopodiché si **swappa** (`pop()`) e si rimette nel disco.
- I programmi non attivi sono mantenuti all'interno del disco
- Lo scambio dei programmi dalla **RAM** al **disco** viene effettuato dallo **scheduler** chiamato **SWAPPER**

Esempio:





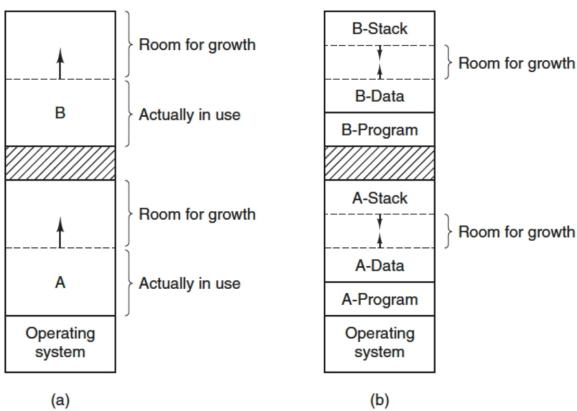
c) A viene swappato nel disco -> g) ritorna A e si trova in una nuova posizione rispetto all'inizio (a) e quindi i nuovi indirizzi si devono rilocare

Se, dopo g) dovesse tornare B, allora qualche programma deve essere sacrificato in modo creare la memoria libera per B. Il programma da sacrificare, però, non deve avere **richieste I/O pendenti**

- Si nota che nei vari **passi intermedi** (d) , e) , f) vi sono degli **spazio vuoti fra blocchi di memoria** e che ovviamente sono **SPRECATI**.
- Gli spazi vuoti all'interno della memoria sono chiamati **FRAMMENTAZIONE ESTERNA** (avviene all'esterno rispetto ai programmi allocati)
- Gli spazi vuoti si devono **compattare fra loro** e si parla di **MEMORY COMPACTATION** ma questa è un'operazione lenta e costosa (per questo motivo viene usata raramente)
- La **FRAMMENTAZIONE INTERNA**: alloco uno spazio di programma più grande rispetto a quanto ne richiede effettivamente e quindi **all'interno dello stesso programma allocato**

Quanta memoria si deve allocare?

Se il blocco è da 64 e il programma da 50, i 14 sono sprecati (frammentazione interna). Se il programma richiede 68, allora devo allocare 2 blocchi da 64 e quindi viene occupato 1 blocco + 4 e il resto è sprecato



Vi sono diverse strategie e difficilmente si riesce a beccare il numero esatto di memoria necessaria:

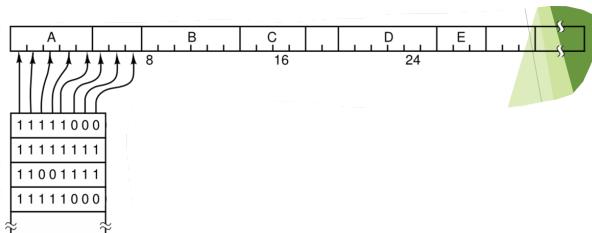
- Alloco blocchi di memoria a **DIMENSIONE FISSA** ma ciò rappresenta un problema con i programmi visto le loro *dimensioni* non sono fisse ma *variano spessissimo*.
- Alloco blocchi dimemoria a **DIMENSIONE DINAMICA** (figura a) dove si alloca una porzione di memoria più una quantità extra all'interno del quale i processi possono **crescere**.
 - Un'altra strategia migliore di questa (figura b), basata sulla dimensione dinamica, è quella di distinguere le parti **rekative** ai dati locali.
 - Si alloca un programma e, nella parte **superiore** extra, si aggiunge nella parte più bassa i dati del processo (che potrebbero crescere dal basso verso l'alto) e gli stack usati dal processo nella parte alte (che cresce dall'alto verso il basso).
 - Si ha quindi **dati** (basso->alto) e **stack** (alto->basso) e quindi le crescite si muovono in direzioni opposte

- Comunque sia la dimensione è limitata e lo spazio si può **esaurire**

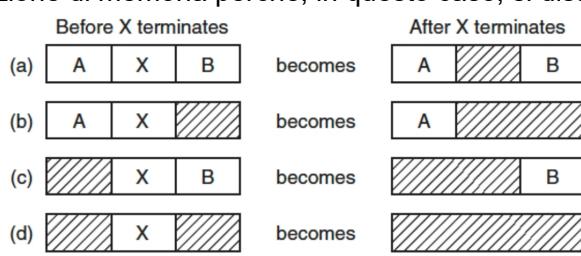
Ma come si tiene traccia dei blocchi liberi o dei blocchi occupati e come si identificano?

Esistono 2 modi con cui il SO tiene traccia dei blocchi liberi/occupati:

- **BITMAP** (*matrice di bit*): una volta stabilito che la memoria è divisa in blocchi, per identificare i blocchi liberi/occupati, a ogni blocco si associa un bit che identifica 0 se la locazione è **LIBERA** e 1 se la locazione è **OCCUPATA**:



- - Questo singolo bit quanto deve essere grande?
 - Se è da 64 posso identificare blocchi più piccoli di 64
 - La dimensione del bit incide su vari aspetti: la dimensione della bitmap -> **Più è piccola la dimensione di ogni bit, più è grande la bitmap.**
 - **Più è grande la bitmap, minore** è la probabilità di generare **frammentazione interna**.
 - **Più è piccola la bitmap**, più è grande la dimensione del singolo bit e quindi **maggiorre** è la probabilità di generare **frammentazione interna**
 - Nell'**ultimo blocco**, se la dimensione non corrisponde, c'è **per forza frammentazione interna** e quindi l'obiettivo è **MINIMIZZARLA** al fine di non sprecare memoria
 - Lo **svantaggio** è la **RICERCA DELLO SPAZIO DA ALLOCARE** in caso di allocazione:
 - Se devo allocare un blocco di k unità, devo cercare nella bitmap una sequenza di k **bit consecutivi a 0** e questa è un'**operazione lenta** (in particolar modo quando la memoria è a regime)
 - L'alternativa è usare una lista concatenata
- **LISTA CONCATENATA**: ogni blocco dice se il blocco è **occupato** o **no**. Ogni blocco contiene **4 informazioni**:
 - - Se è il blocco occupato/libero (P Processo presente, H Hole -> Spazio vuoto)
 - Da dove parte il processo/segmento
 - Quanto è lungo il processo
 - Puntatore al nodo successivo
 - Se la **lista è ordinata per indirizzo**, la ricerca è semplice ma in particolar modo quando si libera una locazione di memoria perchè, in questo caso, si distingono varie casistiche:



- La gestione è molto più semplice nelle 4 casistiche sopra indicate: si analizza il precedente/successivo del programma X e spesso si usa una **LISTA DOPPIAMENTE CONCATENATA** e si mantiene sempre **ordinata per indirizzo**

Vi sono 4 **ALGORITMI DI RICERCA** per allocare blocchi di memoria:

- **FIRST FIT**: parto dalla testa, appena trovo **il primo blocco libero per soddisfare la richiesta**, lo prendo e lo occupo (molto veloce e molto usato). Il problema: se prendo il primo nodo utile è probabile che **creo frammentazione interna**
- **NEXT FIT**: la stessa del precedente ma questa volta parte da dove si era fermato al passo precedente (*quindi non parte più dalla testa*). Le prestazioni sono leggermente inferiori al **FIRST FIT** (*anche questo fa frammentazione interna*)
- **BEST FIT**: cerca di trovare la **migliore locazione vuota** la cui dimensione è quanto più **vicina a quella richiesta**. Si ha spazio vuoto interno ma minore rispetto al **FIRST FIT**. E' un algoritmo meno efficiente rispetto al **FIRST FIT** perchè deve **scansionare tutta la lista**
- **WORST FIT**: Cerca e alloca **lo spazio di memoria più grande possibile**. Crea grande frammentazione interna. Non è usato proprio per questo motivo perchè crea grande frammentazione interna

Si può **ancora ottimizzare** questo approccio separando le liste: una lista doppiamente concatenata per **blocchi vuoti** e un'altra per **blocchi occupati**.

In questo caso, se da un lato ottimizzo la gestione degli algoritmi in caso di allocazione, dall'altro lato sto peggiorando il caso di deallocazione: **le due liste devono INTERAGIRE FRA LORO** in caso di deallocazione e quindi si perde più tempo ma si guadagna nei tempi di ricerca.

Si possono separare le liste e **ordinarle per dimensione** piuttosto che per indirizzo. In questo caso il **BEST FIT diventa l'algoritmo migliore** perchè trovo subito la locazione utile in testa.

*Se lo ordino per indirizzo **FIRST FIT** è il più veloce (frammentazione alta). Il **BEST FIT** scorre tutta la lista ma la frammentazione è piccola

*Se ordino la lista per dimensione **BEST FIT** è il miglior algoritmo da usare e **FIRST FIT** fa, di conseguenza, la stessa cosa.*

Approccio con Memoria virtuale

Consiste nel consentire a più programmi di essere caricati anche **PARZIALMENTE** nella memoria.

I programmi richiedono una dimensione superiore a quella che si ha realmente (fisicamente)

I piccoli pezzi parziali di programmi sono detti **OVERLAY**. All'inizio parte il gestore di overlay che carica e avvia l'overlay 0, poi l'overlay 1 e così via..

Problema: l'idea è quella di mantenere pezzi di programma in memoria ma gli overlay si trovano nel disco (operazioni `disco->memoria` **frequenti**) ma il limite principale è che gli overlay venivano **gestiti dal programmatore** e per questo motivo gli **overlay richiedono più tempo** e quindi si è più soggetti a **errori**.

L'idea è di affidare la gestione degli overlay **AL SISTEMA OPERATIVO**. Si assegna uno spazio degli indirizzi personale a ogni programma. Ogni pezzo di programma è detto **PAGINA**

La memoria virtuale prende l'idea precedente e la affida al SO

Pagina

La pagina rappresenta **insieme di indirizzi CONTIGUI mappati sulla memoria fisica**. Più pezzi dello stesso programma possono essere mappati su memoria fisica **ANCHE NON CONTIGUA** quindi i vari pezzi del programma si possono trovare **sparsi per la memoria fisica**.

- Lo spazio di indirizzamento virtuale viene assegnato a ogni programma ed è diviso in pagine. Ogni pagina è associata ad un indirizzo fisico
- Il SO (in particolare la component MMU) farà la **traduzione da indirizzo logico(virtuale)->indirizzo fisico** e funziona nei sistemi moderni dove si usa un *sistema multiprogrammato* (*più pezzi di programmi in memoria allo stesso momento*)
- Lo spazio degli indirizzi virtuale è diviso in blocchi di **DIMENSIONI FISSE**. Ogni blocco è detta **PAGINA**.
- Le unità (blocco di memoria fisica) corrispondenti della memoria fisica vengono chiamate **FRAME**.
- Le **pagine** e i **frame** hanno la **STESSA DIMENSIONE**, in genere, MA il *numero di pagine* è **SUPERIORE** al numero di frame

Quindi gli indirizzi virtuali devono essere di più rispetto al numero di indirizzi fisici altrimenti questo discorso non avrebbe senso

- Si ha una **protezione implicita fra processi**: nessun processo ha modo di **accedere** a parti di memoria che **NON gli sono state assegnate**, o che non possiede.

Si ha una netta separazione fra la **memoria virtuale dell'utente** e fra **quella fisica effettiva**. L'utente vede un unico spazio di memoria che contiene il programma considerato ma in realtà si riesce a gestire l'esecuzione del programma dell'utente dividendo il programma in pezzi sparsi per la memoria.

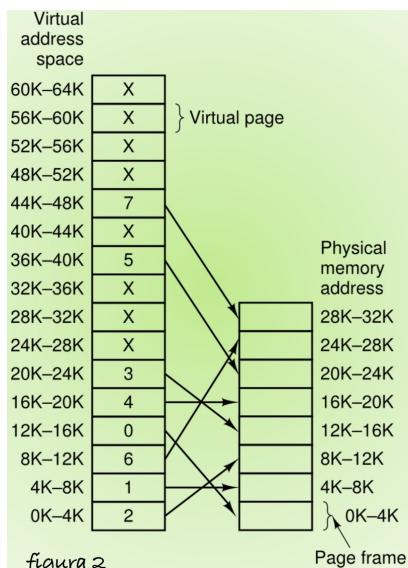
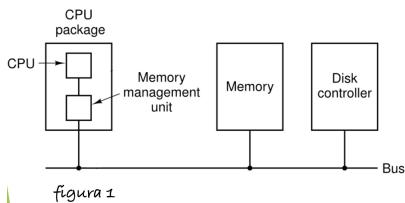
Lo **spazio virtuale degli indirizzi** è l'**insieme** degli indirizzi virtuali generati *dal programma*.

Paginazione

La paginazione è il meccanismo sopra descritto.

Quando si parlava `enter_region` e `Leave_region` nei processi, c'era l'istruzione `MV R2, #1000 che sono istruzioni macchina che copiano il contenuto di 1000 nel registro R2.`

Il programma genera da sè gli indirizzi virtuali e forma lo spazio degli indirizzi virtuali. Gli indirizzi virtuali vengono inviati al MMU che li traduce in indirizzo fisico e lo invia all'unità di elaborazione.



Nell'esempio sopra in figura:

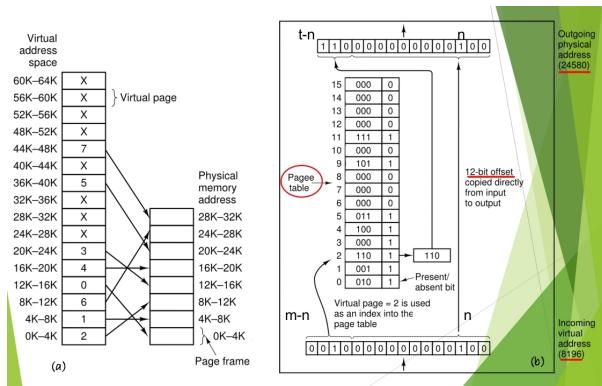
- Si hanno 16 pagine virtuali e 8 pagine fisiche, e ogni cella è grande 4K
- Se ho `MOV REG #0`, si accede all'indirizzo virtuale 0 e viene spedito a MMU e si nota che corrisponde il **frame 2** nella memoria fisica e corrisponde, quindi, **all'inizio** del frame 2 (perchè non ho nessuno sfasamento, o **offset**)
 - quindi all'indirizzo virtuale 0 corrisponde l'indirizzo fisico 8K (**inizio della frame**)
 - Inizio della pagina = Inizio del frame (se non c'è offset)
- Può capitare che una **pagina virtuale non sia assegnata ad alcun frame fisico** (si vede la X al posto di un numero), quindi si ha necessità di **trovare un modo per tenere traccia di quello che è associato a un frame** per distinguerlo da quando non lo è.
 - In questo caso si usa un bit (**BIT DI PRESENZA/ASSENZA**) che, settandolo a 0 o a 1, si usa per capire se una pagina è associata a un frame oppure no.
- Ad ogni processo si associa una **TABELLA DELLE PAGINE** e contiene: *il bit di presenza/assenza, frame associato (se associato a un frame)*
- Un frame allocato ad una pagina di **un processo**, ovviamente, **non può essere allocata** a una pagina di **un altro processo**.

Esempio (vedi figura2)

Data l'istruzione `MOV REG #32780`:

- L'indirizzo deve essere prima tradotto
- 32780 appartiene al blocco 32K-36K
- MMU riceve l'indirizzo, accede alla tabella degli indirizzi associata al processo.

- Si accede all'ottava pagina corrispondente e accede al bit presenza/assenza e ne controlla il valore. In questo caso è 0 (X).
- In questo caso MMU genera una **TRAP** per chiamare il Sistema Operativo e questo evento viene detto **PAGE FAULT (Errore di Pagina)**
- Il SO associa un frame alla pagina che non la ha. **Associa** la pagina a qualcosa di **libero** oppure effettua una **sostituzione di pagina**.
 - Se c'è una **pagina di lettura** (es: *ascolto musica*) allora si può rimuovere.
 - Se c'è una **pagina di lettura/scrittura**, allora i dati devono **PRIMA essere scritti in memoria** e poi può essere rimossa.



- Ogni pagina è grande 4K.
- Per identificare univocamente le pagine virtuali servono **16 bit** mentre per quelli fisici servono **15 bit**
- MMU riceve indirizzo virtuale (16 bit). I tot bit vengono suddivisi in **2 parti** non uguali:
 - **I più significativi** (4) indicano il **numero di pagina** che accede alla pagina stessa
 - **i meno significativi** (12) vengono detti **OFFSET** e sono messi da parte
- Verifica il bit di presenza/assenza
- Per esempio ho `0010` che corrisponde alla pagina di indice 2. Controllo bit di preseza e trovo 1, prendo i bit `110` e **li aggancio all'offset** e trovo l'indirizzo fisico (vedi disegno figura b)

In generale:

► spazio indirizzi virtuali: 2^m

► dimensione pagina: 2^n

► numero di pagina: $(m-n)$

► bit più significativi dell'indirizzo virtuale

► 2^{m-n} pagine

► offset: n bit meno significativi

► spazio indirizzi fisici: 2^t

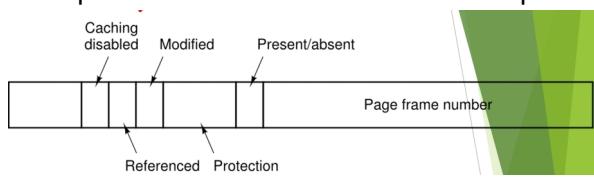
→ 2^{t-n} frame

→ $m > t$

La suddivisione può essere differente in base alle dimensioni delle pagine. Lo **SCOOPO** della tabella delle pagine è mappare le pagine virtuali in frame delle pagine.

Dettaglio su una voce della tabella delle pagine

Le informazioni su una pagina servono per gestire nel migliore dei modi la rimozione di una pagina per dare spazio ad un'altra ed eventualmente per altre operazioni.



- Numero del frame : indica il numero del frame associato a quella pagina (la mappatura dalla quale si ricava l'indirizzo fisico)
- bit presente/assente
- bit modificato (**DIRTY BIT**): indica che è a 1 quando il contenuto di una pagina è modificato rispetto alle modifiche presenti in memoria in quell'istante
- bit referenziato : indica quando una **pagina è stata referenziata/utilizzata**. Vale 1 quando si fa riferimento ad un'altra pagina. E' sempre 1 se la pagina è modificata e viceversa.
 - i bit referenziato e modificato servono per capire quando una pagina è *usata e modificata*
 - Una pagina chiamata in causa per operare comporta il bit referenziato a 1. Se poi deve essere modificata, ciò comporta il bit modificato a 1.
- bit di protezione : identifica la **tipologia di accesso** che si può fare alla pagina: *lettura/scrittura/esecuzione*
 - Recentemente questo campo è formato da 3 bit e quindi permette le combinazioni lettura/scrittura, lettura/esecuzione ecc ecc..
- bit per disabilitare la cache per la specifica pagina e si usa quando si lavora sui **registri dei dispositivi**. In particolar modo quando le pagine vengono mappate **direttamente** sui registri dei dispositivi I/O e serve per **evitare accessi inutili alla cache** proprio perchè non viene *mai chiamata in causa*
- bit di validità (o di allocazione): indica se la pagina è in uno spazio di indirizzi valido oppure no. Serve anche per **impedire/permettere l'accesso alla pagina di riferimento**.
 - Se è 0 (non valido) allora la pagina non è associata nello spazio degli indirizzi logici di quel determinato processo. Conseguo che le attività vengono bloccate perché non si ha accesso a quella pagina (perchè *non valida*).
- Le pagine che sono state **usate nel breve** intervallo di tempo, con molta probabilità verranno usate di nuovo **nel breve futuro** e, seguendo questa logica, si decide quale pagina rimuovere dalla tabella in caso di sostituzione.
- Si usano i bit modificato e referenziato per capire se una pagina è stata usata di recente o no.

Con questo modo di agire non si annulla il page fault ma si riduce la probabilità che esso accada

23-05-2023

Tabella dei frame

Traccia lo stato di occupazione di ogni frame. Ogni frame ha le seguenti informazioni:

- **occupato/libero**: se è assegnato ad una pagina o no.
- se è assegnata ad una pagina, allora vuole sapere **l'informazione che contiene** e da **quale processo** è occupata

La tabella dei frame *viene consultata*:

- ogni volta che si **crea un nuovo processo** per creare la relativa tabella delle pagine di quel processo
 - L'associazione pagina->frame avviene solo quando una pagina viene chiamata in causa per la prima volta e quindi si ha page fault quando **avvio per la prima volta un programma** perchè il numero di quel programma si trova in memoria (*perchè aperto in quel preciso momento*).
 - *Per questo motivo i page fault non si possono annullare*
- ogni volta che un processo chiede di **allocare nuove pagine** che non sono caricati e non sono associati dei frame alle pagine richieste

Se ho un sistema di più processi e più utenti e creo page fault (pagina non associata ad alcun frame) allora bisogna scegliere una pagina da togliere per associrane una nuova: qual è la pagina che tolgo ad un processo?

Bisogna fare delle valutazioni e rimuovere una determinata pagina e non una a caso.

Progettazione di una tabella delle pagine

La tabella delle pagine è singola per ogni processo. Per tale motivo le dimensioni occupate in memoria aumentano e diminuisce velocità:

- Il mappaggio deve avvenire il più **velocemente possibile**
- lo **spazio degli indirizzi virtuali** (tabella delle pagine) è **grande** se il mappaggio è grande

Si ha accesso costante alla memoria sia nella traduzione per accedere all'istruzione. Se l'accesso alla memoria diventa lento, questo potrebbe rallentare le prestazioni

Più è grande la tabella delle pagine, più diventa delicata e pesante la gestione delle tabelle

Si ha **una sola tabella delle pagine formata da un array di registri**. I registri sono parti di memoria hardware e quindi sono veloci.

- Si ha un array di registri hardware con una sola voce per ogni pagina virtuale
- All'avvio del processo, il SO carica i registri con la tabella delle pagine del processo interessato e ,durante l'esecuzione, non è necessario accedere alla memoria della tabella ma vengono usati i registri
 - +E' semplice e si usa un array di registri (*evita l'accesso alla memoria*)
 - +Si guadagna in termini di velocità
 - -Il cambio di contesto (se la tabella è **MOLTO ESTESA**) diventa lento

Un'alternativa a ciò è: si tiene una **tabella interamente caricata in memoria** e si ha bisogno di un **registro che punta all'inizio della tabella delle pagine**, detto **PTBR (Page Table Base Register)**.

Differenze

- Con **array di registri**: veloce e si evitano accessi alla memoria ma risulta dispendiosa (cambi di contesto) se la tabella è estesa
- con la tabella **interamente caricata in memoria**: usa un solo registro (più veloce in caso di context switch) ma si devono fare gli accessi alla memoria (*almeno 2 volte -> uno per consultare la tabella delle pagine e uno per recuperare le informazioni necessarie*)

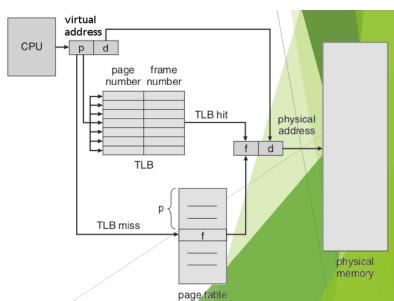
Ogni accesso alla memoria aumenta il rallentamento del sistema a livello complessivo

Alternativa basata su un'osservazione: *i programmi in esecuzione fanno molti riferimenti a un **numero piccolo di pagine associate***, quindi vuol dire che accedono a un numero ristretto di voci nella tabella delle pagine e quindi vengono usate solo determinate voci nella tabella.

Ogni programma fa uso di un gran numero di riferimenti a piccolo numero di pagine, quindi non a tutte le sue pagine associate. Vuol dire che delle voci vengono frequentemente chiamate in causa mentre le altre no, o comunque poco usate.

Le pagine più usate vengono **messe in evidenza**:

- Si usa un apposito dispositivo (nella MMU) che mantiene *le pagine più frequentemente chiamate in causa*
- Il dispositivo è detto **MEMORIA ASSOCiativa (Translation Lookaside Buffer TLB)** ed è qui che avverranno le ricerche delle pagine, piuttosto che cercare nella tabella delle pagine visto che è **MOLTO PROBABILE** che la pagina si trovi proprio qui. (il funzionamento è identico alla **CACHE**)



Le **voci** nella TLB sono:

- il **numero di pagina virtuale**
- bit di **validità**, se il processo ha la validità per usare quella pagina oppure no
- **numero di frame associato**
- il **dirty bit** (bit di modifica)
- **codice di protezione** (come si può accedere a quella pagina)

La traduzione avviene accedendo alla TLB.

- Se **presente** nella TLB (**TLB HIT**) allora determina il frame fisico e fa l'operazione.
- Se **non presente** (**TLB MISS**) allora si accede e si va a ricercare all'interno della memoria. MMU non ha trovato la pagina nel TLB. In questo caso si cerca nella memoria nella tabella delle pagine.
 - La pagina ricercata è stata richiesta recentemente quindi **con molta probabilità verrà richiesta** in futuro. Per questo motivo deve essere **registrata nella TLB DOPO** averla trovata nella tabella

delle pagine e **PRIMA** di restituire l'*indirizzo fisico*

- Se la TLB non ha spazio, allora devo sacrificare una pagina da sostituire e in questo caso verranno usati algoritmi che identificano quelle usate **meno recentemente**
- Alcune voci della TLB possono essere **VINCOLATE**: cioè *non possono essere scartate* in caso di *TLB-miss*
- Alcune TLB memorizzano address-space identifiers(ASID) e identificano in maniera univoca un processo. Se la TLB non li supporta, allora quando viene selezionata una nuova tabella delle pagine, la TLB viene **invalidata** cioè si fa il **FLUSH DELLA TLB**, ovvero il proprio contenuto non è più valido e viene presa una considerata una nuova tabella di riferimento

La TLB deve mantenere le pagine frequentemente utilizzate in modo che l'MMU cerca qui dentro visto che si tratta di record molto minori rispetto alla tabella delle pagine. In questo caso si accelerano le operazioni.

Vantaggi TLB

Si valuta il **grado di successo** della TLB (*TLB HIT*). Se è alto allora il ricavo è sostanziale.

Esempio:

- tempo di accesso alla memoria = 100 nsec;
- tempo di accesso alla TLB = 20 nsec

Il **tempo effettivo di accesso** è:

- 120nsec per *TLB HIT*, non devo accedere alla memoria per ricercare la pagina
- 220nsec per *TLB MISS*, si accede alla memoria per cercare nella tabella delle pagine

Se per ipotesi si ha una TLB ratio (percentuale di successi) dell'80%:

- tempo(medio) effettivo di accesso: $0.8 \times 120 + 0.2 \times 220 = 140\text{nsec}$, ho 40nsec di ritardo (*rispetto al tempo di accesso alla memoria*)

In generale: (**ESAME**)

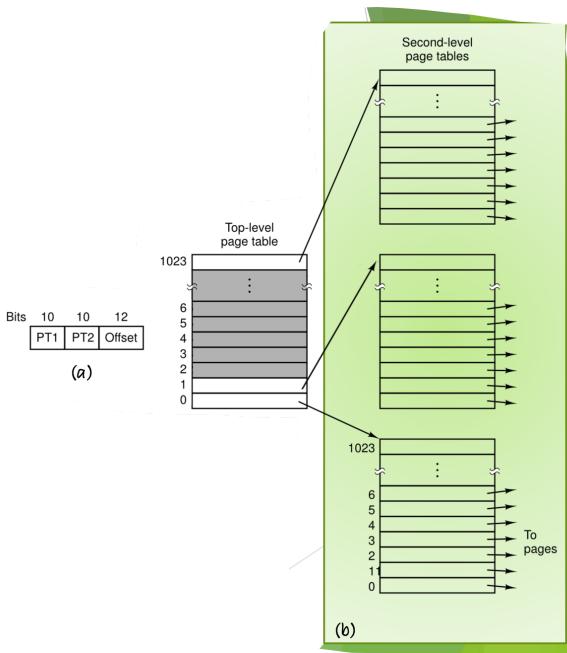
- tempo di accesso alla memoria: α
- tempo di accesso alla TLB: β
- TLB ratio: ε
 - $EAT = \varepsilon(\alpha + \beta) + (1 - \varepsilon)(2\alpha + \beta)$ -> **Tempo di accesso effettivo medio**

Dimensioni

Come si gestisce lo spazio di indirizzi se questo è molto grande?

Si usa tabella delle pagine MULTILIVELLO: L'indirizzo virtuale (32 bit) è spezzato in **3 parti**:

1. PT1(10bit), PT2(10bit), Offset(12bit)
2. Ogni pagina è quindi da 4K
3. Vantaggi: si evita di mantenere tutte le tabelle delle pagine in memoria per tutto il tempo e **si mantiene solo quello che è necessario in quel momento**.



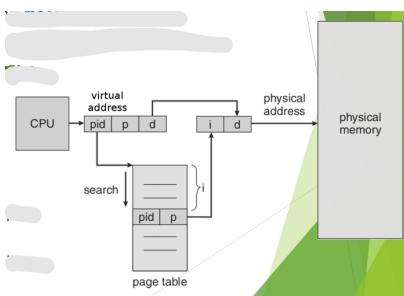
- La voce 0 punta a una tabella delle pagine per il *testo del programma*(?)
- La vove 1 punta ai **dati del processo**
- L'ultima voce punta alla tabella delle pagine per lo **stack del processo**
- Tutte le altre voci (in grigio) sono **inutilizzate**

PT1 = indice della tabella delle pagine di primo livello

PT2 = indice della tabella delle pagine di secondo livello, è usato per cercare il numero del frame della pagina stessa

Se si usano molti più livelli allora potrebbero verificarsi dei rallentamenti e per tale motivo si non si deve esagerare.

Tabella delle pagine invertite



In questo apprccio: si mantiene una sola voce per frame fisico in memoria piuttosto che mannere una voce per pagina virtuale.

Ogni voce ha: indirizzo virtuale con le eventuali informazioni di quella pagina.

Ogni voce viene rappresentata da una coppia (`idprocesso, pagina_virtuale`) dove `pid` = identificativo dello spazio degli indirizzi.

Quando si verifica un riferimento alla memoria, sarà rappresentato da (`pid,npagina`). Viene verificata e se c'è una corrispondenza, viene restituita la voce i -esima associata all'offset (`d`).

Se non c'è corrispondenza, viene generato il *page fault*

Le tabelle invertite rappresentano dei limiti:

- Bisogna andare a ricercare si deve **scorrere l'intera tabella**, cioè svolgere un'**operazione di ricerca lenta**
- Si fa uso allora di una tabella HASH indicizzata sugli indirizzi virtuali.
 - consente di poter velocizzare la ricerca dei dati e si tampona sullo svantaggio della ricerca lenta
- Se si usa anche una TLB, si migliorano ancora di più le prestazioni

Aspetto fondamentale: cosa fare in caso di page fault?

Page fault

Vuol dire che la pagina non è all'interno della TLB. In questo caso si deve sostituire un frame associato a una pagina per associare la nuova pagina se la TLB è piena.

*Quale pagina si rimuove? Si sceglie una **VITTIMA** da rimuovere per dare spazio alla nuova pagina*

L'**obiettivo**, comunque, è quello di minimizzare il numero di page fault in futuro

*Basta capire e individuare quali sono le **pagine più utilizzate***

Allora la soluzione ottimale e teorica è l'uso di un **algoritmo ottimale (OPT)**:

- si deve scegliere la pagina da rimuovere che verrà referenziata in un futuro più lontano
- è **ottimale** ma **difficile da realizzare**
- dà comunque l'idea ottimale da seguire e un termine di paragone con le altre tipologie di algoritmi

Le due **NOZIONI CHIAVE** per gli algoritmi di sostituzione (*che hanno lo stesso obiettivo: scegliere la pagina che, con molta probabilità, evita i page fault nel breve futuro*) sono:

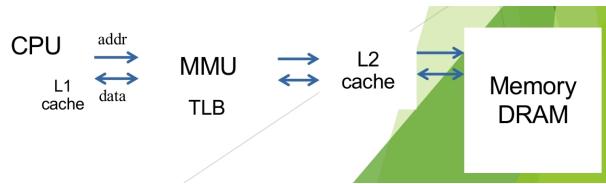
- la pagina da rimuovere è quella che **non è stata usata recentemente**
- **fra 2 pagine** non usate recentemente, sostituire quella **più anziana** fra le due (confrontando il "quando è stata usata l'ultima volta", quindi referenziata)

Come si identifica se è stata referenziata recentemente o no? Si usano i 2 bit: referenziamento e di modifica

Cache della memoria vs Memoria virtuale

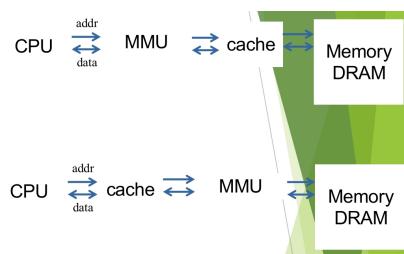
Cache basata su:

- **INDIRIZZI FISICI:**



- **vantaggio:** Non è necessario invalidare la cache ad ogni cambio di contesto (ad ogni nuova tabella delle pagine) perchè, visto che si lavora su indirizzi fisici, non importa quale sia il processo che sta usando quell'indirizzo
- **svantaggio:** si ha che **MMU si trova nel mezzo** fra *CPU* e *CACHE* ed è uno svantaggio perchè l'idea di usare la cache serviva per velocizzare. Se c'è MMU nel mezzo, le operazioni sono più lente
- L1 è interna alla CPU e L2 è quella vicina e in mezzo alle due si trova MMU e L1 non contiene MAI indirizzi fisici mentre quelle di livello superiore sì

- **INDIRIZZI VIRTUALI:**



- **svantaggio:** E' necessario invalidare la cache per ogni *context switch* (operazione lenta) perchè l'indirizzo virtuale potrebbe indicare **indirizzi di memoria fisici appartenenti a DIVERSI PROCESSI** e ciò potrebbe creare dei problemi per quanto riguarda l'esecuzione.
- Il SO coopera con la TLB affinchè ogni singolo processo trovi la sua corretta memoria assegnata
- MMU si trova dopo la cache e quindi prima si va a ricercare nella cache ecc...che velocizza gli algoritmi
- Generalmente L1 è basata su indirizzi virtuali mentre L2 e successive cache sono basate su indirizzi fisici

Page fault

Vuol dire che la pagina non è all'interno della TLB. In questo caso si deve sostituire un frame associato a una pagina per associare la nuova pagina se la TLB è piena

Quale pagina si rimuove? SI sceglie una **VITTIMA** da rimuovere per dare spazio alla nuova pagina

La tabella delle pagine ESTERNA è realizzata per ogni processo e contiene tutte le informazioni sulla posizione di ciascuna pagina virtuale. Viene realizzata solo in caso di **PAGE FAULT**.

L'obiettivo, comunque, è quello di minimizzare il numero di page fault in futuro

*Basta capire e individuare quali sono le **pagine più utilizzate***

- *Può capitare anche che i frame disponibili sono FULL e per tale motivo, allora, si deve disaccoppiare l'indirizzo del frame con il relativo indirizzo virtuale*
- Se il **bit di modifica della pagina** da rimuovere è **1**, allora **NON** si può rimuovere.
- In base a quale pagina si sta togliendo, si possono generare successivi page fault concatenato
- **IN OGNI CASO**, una pagina **deve essere assegnata** a un processo

Algoritmo della soluzione ottimale

Allora la **SOLUZIONE OTTIMALE (IDEALE)** e teorica è l'uso di un **algoritmo ottimale (OPT)**:

- si deve scegliere la pagina da rimuovere che verrà referenziata in un furto più lontano
- **è ottimale ma difficile da realizzare**
- dà comunque l'idea ottimale da seguire e un *criterio di paragone* con le altre tipologie di algoritmi
- L'**idea** è: Scegliere la pagina che verrà referenziata nel futuro più lontano possibile analizzando il numero di istruzioni che contiene (quindi più istruzioni ha) più tempo ci metterà per chiamare in causa tale pagina. Quindi creerà un **page fault nel lontano futuro**

Le **DUE NOZIONI CHIAVE** per gli algoritmi di sostituzione (*che hanno lo stesso obiettivo: scegliere la pagina che, con MOLTA probabilità, **evita i page fault nel breve futuro***, quindi lo ritarda) sono:

- la pagina da rimuovere è quella che **non è stata usata recentemente**
- **fra 2 pagine** non usate recentemente, sostituire quella **più anziana** fra le due (confrontando il "quando è stata usata l'ultima volta", quindi referenziata) controllando, questa volta, il **bit di modifica**

Come si identifica se è stata referenziata recentemente o no? Si usano i 2 bit:

referenziamento e di modifica.

- Ogni volta che una pagina è **chiamata in causa** (*lettura*), allora il **bit di referenziamento** viene posto a **1**

- E' *impossibile* che una pagina venga modificata senza che sia mai stata referenziata, pertanto il bit di modifica viene messo a 1 se la pagina è modificata

*I due bit di stato sopra descritti servono per capire se una pagina è stata **UTILIZZATA** o meno.*

Algoritmo Not Recently Used (NRU)

Ad ogni clock, tutto ciò che non è stato chiamato in causa avrà bit di referenziamento a 0. in questo caso si può differenziare fra una pagina referenziata da poco o da molto:

- Se **referenziata nell'ultimo clock** allora il suo bit di referenziamento sarà a 1
- Se **NON referenziata nell'ultimo clock** allora il suo bit di referenziamento sarà a 0

Le possibili casistiche di una pagina per quanto riguarda la referenziazione sono:

- referenziata un **"sacco"** di tempo fa*: bit referenziamento = 0
- referenziata in **quell'istante**: bit referenziamento = 1

Vengono distinte **4 CLASSI**, in base ai 2 bit citati sopra:

- **classe 0**: non referenziato, non modificato;
- **classe 1**: non referenziato, modificato; (*modificato ma referenziato molto tempo fa*)
- **classe 2**: referenziato, non modificato;
- **classe 3**: referenziato, modificato
- La pagina che l'algoritmo **sostituisce**, di base, è quella di **classe 0**. Se non è presente la classe 0, allora viene presa in considerazione la classe 1 e così via.
- Le classi con "indice più alto(3)" devono essere salvaguardate rispetto a quelle con "indice più basso (0)".
- In caso di **PARITA' di classe fra 2 pagine**, viene scelta **UNA PAGINA CASUALE** da scartare visto che, per l'algoritmo, le pagine sono esattamente **EQUIVALENTI**
- Se la pagina è modificata, allora, **prima di essere rimossa**, deve essere copiata in memoria. Di conseguenza si ha un *accesso in memoria* e quindi una **"perdita di tempo"**

Formalmente: Viene scelta una pagina dalla classe non vuota di numero più basso
Questo algoritmo è semplice da implementare e dà dei risultati abbastanza accettabili

Algoritmo FIFO

Si usa una normalissima **CODA**. Le nuove pagine vengono messe in coda mentre quelle da rimuovere sono le prime arrivate e meno usate, quindi in testa.

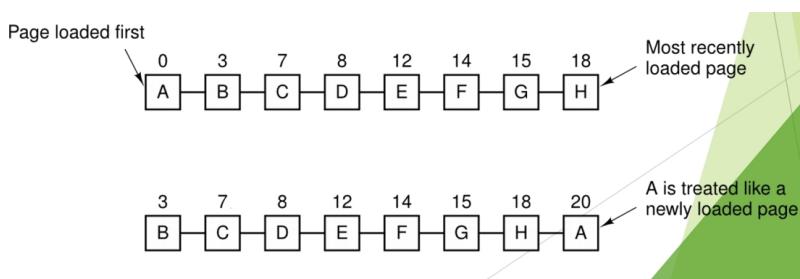
- Si va a rimuovere la pagina chiamata molto tempo fa senza tenere conto se è stata richiamata recentemente oppure no

Questo approccio non fa distinzione: la testa indica la pagina chiamata per prima rispetto alle altre ma non dice quando è stata richiamata l'ultima volta.

- FIFO rimuove la pagina più vecchia e **non tiene conto della referenziazione nell'ultimo intervallo di tempo** (quindi magari **MOLTO USATE**) e per questo motivo l'algoritmo, nonostante sia semplice, non è da implementare

Seconda Chance

- Si prende l'idea della FIFO e si **prende in considerazione il bit di referenziazione**.
- Si prende la **pagina in testa** e si **controlla** tale bit.
 - Se è 0 si può rimuovere
 - altrimenti non si può rimuovere e quindi si mette la pagina in coda (e resettato il bit di referenziazione) e si va avanti finché non si trova il bit di referenziamento a 0
- Viene scartata la **pagina più vecchia** fra quelle non referenziate di recente



Si devono gestire tutti i possibili casi, ma in questo algoritmo ci possono essere delle condizioni che riportano alla FIFO:

- Se **tutte le pagine sono referenziate**, si torna alla FIFO dove ogni pagina ha il bit R (referenziazione) a 0
- In questo caso si ha lo stesso problema di prima
- Ciò potrebbe essere possibile se i processi in esecuzione sono molto veloci

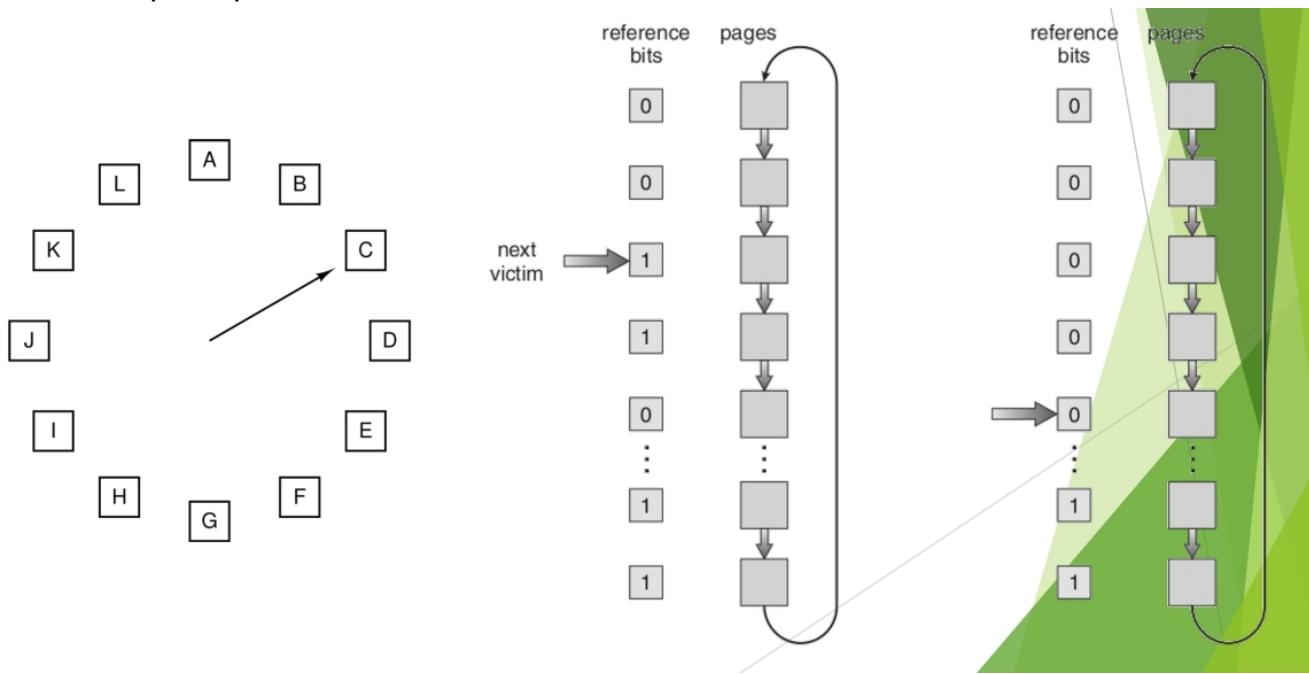
In alcuni casi, questo algoritmo degenera al limite di FIFO (quindi l'algoritmo diventa proprio FIFO):

- Si usa invece l'algoritmo **SECONDA CHANCHE** una **CODA CIRCOLARE**, chiamato **ALGORITMO CLOCK**
- In questo caso non si risolve il problema FIFO ma lo attenua

Algoritmo Clock

Non risolve il problema che può emergere dall'algoritmo Seconda Chance ma lo migliora a **livello di tempistica e di algoritmo in sé**.

Questo è quello più **robusto** ed **efficiente**



Se si hanno più pagine con bit $R=0$ appartenenti a una specifica classe:

- Alcune pagine sono state referenziate in un tempo recente e alcune in un tempo più lontano
- Fra 2 pagine referenziate nel lontano passato, si deve scegliere quella **PIU' ANZIANA** fra entrambe
- *Non si hanno informazioni sull'esatto momento di chiamata in causa*
- Con l'algoritmo che si basa solo su bit R non permette di distinguere fra una pagina referenziata 2 clock fa e una 4 clock fa (*per esempio*)

Si usa Least Recently Used

Algoritmo Least Recently Used (LRU)

Fra 2 pagine non usate di recente, si cerca di eliminare la pagina usata nel tempo più lontano fra le due.

Si basa sull'idea: le pagine più recentemente usate hanno maggior probabilità di essere chiamate in causa, e quindi referenziate.

Un **PRIMO METODO** è quello di usare un contatore:

- Si basa su un **CONTATORE** (*nella CPU*) usualmente di una lunghezza di 64 bit: ogni volta che si esegue l'istruzione, **questo contatore viene incrementato**. Quando una pagina viene referenziata, il contatore viene copiato in un apposito campo della pagina.
- Ogni volta che referenzio una pagina, si copia il contatore di che è stato calcolato in quell'istante su quella pagina
- In caso di PAGE FAULT:

- Si vanno a confrontare i contatori alle pagine e il **CONTATORE PIU' BASSO** numericamente rappresenta la pagina più vecchia **DA RIMUOVERE** perchè vuol dire che quella *pagina è quella che è stata chiamata in causa di meno rispetto a tutte le altre (parlando di referenziamento)*

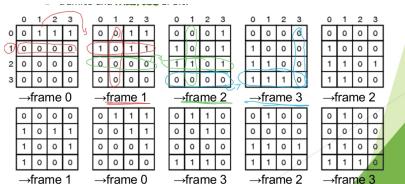
Un **SECONDO METODO** è quello di usare una **MATRICE DI BIT**:

- E' grande NxN dove N = *numero di frame*
- L'idea è: inizialmente le matrici sono tutte inizializzate a 0
 - ogni volta che una pagina è referenziata (corrispondente a un frame k), il frame corrispondente viene così settato:
 - la k -esima **riga** di quel frame viene settata a **1**
 - la k -esima **colonna** viene settata a **0**
 - In caso di page-fault, la riga più bassa sarà **quella meno utilizzata**, cioè **quella riga dove saranno presenti più zeri**

Se più zeri sono presenti nella riga 2, allora la pagina di indice 2 è quella da rimuovere

Per gestire i contatori serve un **supporto hardware opportuno** e si tratta di operazioni **dispendiose** (soprattutto per le **matrici utilizzate**, che risultano piuttosto **enormi**)

Esempio



Algoritmo Not Frequently Used (NFU)

Si usa l'idea di LRU(approssimazione di LRU) usando una **VERSIONE PIU' SOFTWARE** LRU: è gestito di più dal punto di vista software e usa proprio un contatore **SOFTWARE**

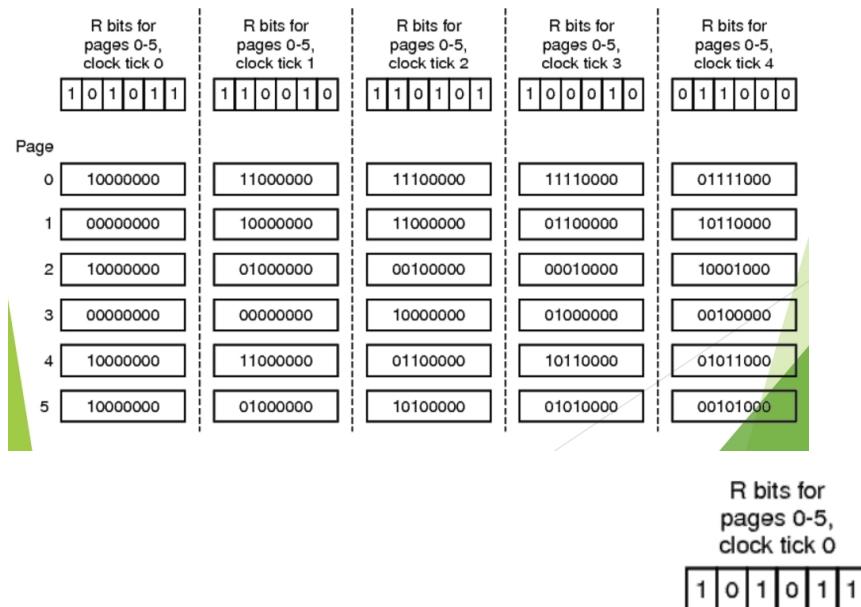
- A ogni pagina viene **sommato al contatore** il valore del bit di referenziamento
- In questo modo, periodicamente, si ha il bit R il quale viene sommato al contatore
- Quando si ha un page-fault, la pagina da sostituire è quella che avrà il **CONTATORE PIU' BASSO** numericamente

Problema di questo algoritmo: esso dipende dal tempo di esecuzione dei processi ed inoltre può privilegiare (erroneamente) pagine molto usate nel passato rispetto a quelle usate di recente e questo problema si può risolvere nel seguente modo:

- Si usa l'algoritmo di AGING usando un meccanismo di contatori.
- Fa solo 2 modifiche:

- Si ha un contatore (*sequenza di bit*) e si **SHIFTANO** i suoi bit **verso destra** di una posizione
- Dopo lo shift **si libera il bit più significativo** e proprio in quella posizione **si copia il bit R di referenziamento**
- In questo modo si distinguono i tempi esatti di referenziamento di ogni singola pagina

Algoritmo di Aging

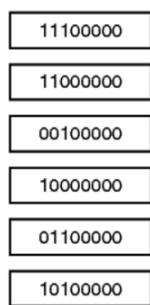


Guardando la sequenza di bit in alto a sinistra:

- nella posizione i-esima si ha la pagina i-esima che è stata referenziata (bit=1) oppure no (bit=0)

Leggendo *dall'alto verso il basso* l'immagine sopra:

- La sequenza **più in alto** viene **shiftata a destra** di una posizione
- nella posizione 0 si mette il **bit R** (*presente nella posizione i-esima della pagina i*)



- In caso di *page-fault* in tick 2 (*clock 2*), allora **la pagina da rimuovere**, guardando quelle disponibili in colonna, **è quella che ha una sequenza di bit che compone un numero decimale più basso**.

In caso di una pagina con sequenza di bit 0010000 vuol dire che è stata chiamata in causa 3 *clock fa* e si capisce **dal numero di 0 più significativi** (*quindi a sinistra della sequenza*)

*Fra 2 pagine non recentemente usate, si identifica quella ancora più vecchia fra entrambe, cioè quella che ha più zeri a sinistra perchè, in questo caso, **più il contatore è basso** (confrontando i **NUMERI BINARI**)*

In particolar modo..

Clock 0:

- Le pagine referenziate sono quelle di indice 0,2,4,5. (*si legge dall'array più in alto di tutti*)
- Si prende l'array che è pieno di zeri (per ogni pagina)
- Considerando un singolo array nella pagina 0:
 - Si shifta a destra di una posizione
 - Si copia il valore i-esimo e, visto che è stata usata, allora si avrà 10000000

Clock 1:

- Le pagine referenziate sono quelle di indice 0,1,4;
- Si prende l'array di prima corrispondente per una singola pagina, nell'esempio è la pagina 0:
 - Si shifta a destra e si ottiene 01000000
 - Nella posizione 0 dell'array sopra, si copia il bit di referenziamento per quella specifica pagina (che è 1)
 - Si ottiene 11000000

Clock 2: ecc....

Clock 3 -> Page fault:

- Si ha la seguente situazione:

11110000
01100000
00010000
01000000
10110000
01010000

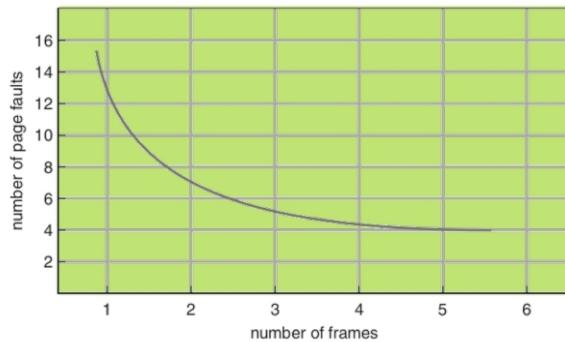
- In questo caso è da rimuovere quella con "valore più basso", quindi quella più vecchia
- Verrà rimossa 00010000, quindi la pagina di *indice 2*

Solo l'AGING tiene in considerazione l'ATTIVITA' PASSATA di ogni singola pagina

Considerazioni dell'algoritmo di Aging

Come si confrontano gli algoritmi fra loro? Si prende in considerazione la seguente metrica: il **NUMERO DI PAGE FAULT** che la politica dell'algoritmo genera, **rispetto all'algoritmo ottimale oppure rispetto a diversi algoritmi**

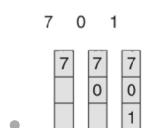
- Si suppone di avere una memoria suddivisa in **3 frame**. Questo numero determina l'efficienza di un sistema. Teoricamente, più frame ci sono, meno è la probabilità di avere page fault



- Si suppone che si ha la sequenza di referenziazione: 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1
 - Cioè si referenzia la pagina 7, poi 0, poi 1 ecc..

Si referenzia la pagina 7:

- Inizialmente si ha un page fault perché niente è caricato in memoria.
- Quindi idem per la pagina 0 e per la 1
- Si generano 3 page fault obbligatori



Si referenzia la pagina 2:

- Quella chiamata in causa più lontana è la 7 e viene sostituita con 2

L'algoritmo ottimale deve rimuovere quella che, vedendo la sequenza, viene chiamata più lontano fra quelle disponibili:



Quindi l'algoritmo genera 9 fault di pagina complessivi

Come detto prima, l'algoritmo ottimale non è realizzabile. Di conseguenza risulta che più ci si avvicina a 9, migliore è l'algoritmo..

Valutazione Algoritmo FIFO

► algoritmo FIFO:

7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1	7	0	1
7	7	7	2	2	2	4	4	4	0	0	0	0	1	1	1	1	7	7	7
0	0	0	3	3	3	2	2	2	3	3	2	2	3	2	2	2	1	0	0
1	1	1	1	0	0	0	3	0	3	3	3	2	2	2	2	2	1	2	1

► 15 fault di pagina

- Ogni volta che avviene un page fault, viene fatta una `dequeue()` quindi viene rimossa la testa della coda e viene sostituita.
- Vengono fatte le seguenti operazioni: `dequeue()` seguita da una `enqueue()`

NOTA: Quando si fa una `dequeue()` la testa avanza di 1, quindi la coda si modifica ad ogni page fault

L'algoritmo FIFO ha generato 15 page fault.

Valutazione algoritmo LRU

► algoritmo LRU:

7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1	7	0	1
7	7	7	2	2	4	4	4	0	0	0	3	3	1	1	1	1	0	0	7
0	0	0	3	3	3	2	2	2	2	2	2	2	3	2	2	2	2	2	2
1	1	1	1	0	0	0	3	0	3	3	3	2	2	2	2	2	2	2	7

► 12 fault di pagina

- Sostituisce la pagina meno usata di frequente fra le presenti.
- In coda si avranno quelle pagine, ordinate, referenziate dalla più vecchia -> alla più recente
 - Quindi in caso si abbia 7,0,1 vuol dire che la più vecchia è 7 che va rimossa.
- Data la sequenza sopra (7,0,1) e viene referenziata la pagina 0, la sequenza in coda diventa 7,1,0 perchè viene aumentato il contatore per quella pagina
- La LRU genera 12 page fault.
- FIFO generava 15 page fault.
- **LRU risulta migliore** rispetto all'algoritmo FIFO perchè si avvicina di più alla soluzione ottimale OPT

Anomalia FIFO di Belady

Nell'algoritmo FIFO si erano generati 15 fault. Si diceva che più sono i frame, meno sono i fault.

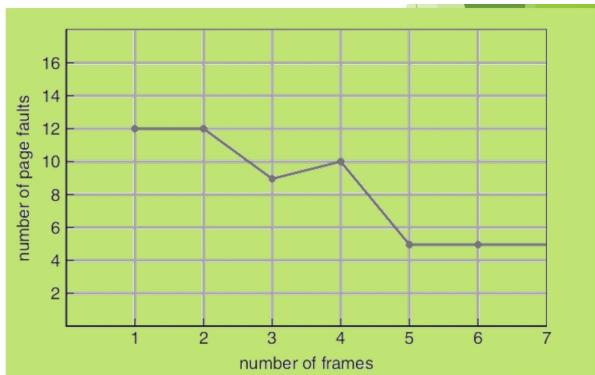
Data la sequenza 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5 e si va a valutare il numero di fault **usando 3 frame e usando 4 frame** (ci si aspetta che il numero di fault con 3 frame sia maggiore del numero di fault con 4 frame):

- Con **3 frame** si generano **9 page fault**
- Con **4 frame** si generano, invece, **10 fault**

Si presenta l'**ANOMALIA DI BELADY** cioè **aumentando** i frame disponibili, **aumentano** i page di fault

L'algoritmo FIFO (e i suoi derivati) presenta questa anomalia

Si ha, appunto, il seguente grafico:



L'algoritmo LRU NON soffre dell'anomalia di Belady e gode della **PROPRIETA' DI INCLUSIONE**: cioè l'insieme delle pagine caricate avendo n frame è incluso in quello che si avrebbe avendo n+1 frame. Cioè più aumento i frame e meno page fault si avranno

In generale...

- **NFU e Aging** non soffrono dell'anomalia di Belady ma godono della proprietà di inclusione
- **Seconda Chance e Clock** (derivati da FIFO) soffrono dell'anomalia di Belady (più frame, più fault) (*si riduce a FIFO*)
- **NRU** soffre dell'anomalia di Belady e si riduce a FIFO

Riepilogo Algoritmi di Sostituzione

OPT: non implementabile, ma utile come termine di paragone;	NRU*: approssimazione rossa dell'LRU;	FIFO*: può portare all'eliminazione di pagine importanti;
Seconda chance*: un netto miglioramento rispetto a FIFO;	Clock*: come S.C. ma più efficiente;	LRU: eccellente idea (vicina a quella ottima) ma difficilmente realizzabile se non in hardware;
NFU: approssimazione software abbastanza rossa dell'LRU;	Aging: buona approssimazione di LRU con implementazione software efficiente.	* soffre dell'anomalia di Belady

Solo l'Aging tiene in considerazione l'ATTIVITA' PASSATA di ogni singola pagina

Allocazione dei Frame

Appena ci si riferisce ad una **pagina per la prima volta**, all'inizio del programma, si avrà un **page fault**. Ogni pagina avrà delle variabili locali e riferimenti ad altre pagine, quindi aumentano i page fault collegati alla pagina iniziale. Il numero di page fault, quindi, inizialmente sarà alto (*perchè inizialmente non si hanno tutte le informazioni necessarie*) e poi andranno a ridursi.

- La strategia di **ASSEGNARE PAGINE SU RICHIESTA** prende il nome di **DEMAND PAGING**.
- Il numero di frame assegnati è determinante per l'esecuzione del processo e per le prestazioni del sistema (*meno frame si hanno, più fault si generano e viceversa -> tranne FIFO*)

Quanta memoria bisogna allocare per i processi?

- Una piccola porzione serve per il sistema operativo
- Il resto serve per l'esecuzione delle varie attività
- Con il *Demand Paging*, il primo frame genera fault. Si avranno fault **fino a quanto tutto sarà caricato in memoria**
 - Per questo motivo si avranno **SICURAMENTE** n fault, con n = dimensione della frame.

IDEALMENTE l'idea è quella di avere dei frame da gestire i frame:

- **IN PIENO**, quindi fault fino al riempimento della frame
- **RISERVARE DEI FRAME LIBERI**. Quando si ha page-fault, si avrà già un frame libero da assegnare. Vi sarà, quindi, un **processo demone** che terrà questi frame liberi)

Questo ragionamento sottintende che c'è **UN SOLO PROCESSO**

Strategie di allocazione

Vi sono 2 strategie e, qualsiasi essa si utilizzi, devono sottostare ad alcuni limiti e condizioni:

- Ad ogni processo **si deve assegnare** un **NUMERO MINIMO DI FRAME**
- Ad ogni processo si deve assegnare un **NUMERO MASSIMO DI FRAME** al quale esso può accedere
Come si assegnano questi frame? si tengono in conto i seguenti **VINCOLI**:
- Non si può allocare *un numero superiore* al numero totale di frame
- ad ogni processo spetta *un numero minimo di frame* che servono per lo svolgimento delle operazioni

Strategia: **allocazione equa**

Il numero di frame assegnati dipende dal numero di fault che ne vengono fuori.

- Assegna **un numero di frame UGUALE** per ciascun processo
- Tutti i processi non hanno le stesse dimensioni fra loro.
- Quindi **questa strategia non è del tutto corretta** perché potrebbe capitare che un processo **A** sia piccolo (quindi può avere frame maggiori del necessario) e un processo **B** che hanno un numero di frame assegnati minori rispetto a quelli necessari ad esso
- L'idea di base di questa strategia è corretta solo **IDEALMENTE**

Per questo motivo si allocano i frame in modo **PROPORZIONALE**

Strategia: **allocazione proporzionale**

Vengono allocati i frame in modo da RIDURRE IL NUMERO DI FAULT complessivamente (anche aumentando di poco il numero di fault di un singolo processo)

Si tiene in considerazione la seguente procedura:

- $S = \text{somma di tutte le dimensioni di tutti i processi}$
- stima del numero di frame da assegnare al processi a_i
- $a_i = (s_i/S) \times m$

Si può osservare che l'allocazione proporzionale è più idonea e l'allocazione (in generale) tiene in considerazione i processi come se fossero **TUTTI UGUALI**. I processi non sono tutti uguali perchè possono esistere processi con priorità diversa rispetto ad altri.

Allora, in questo caso, si usa un'allocazione proporzionale **RISPETTO ALLE PRIORITA'** oppure una combinazione di allocazione (*giustamente pesata*) fra **DIMENSIONE E PRIORITA'**

- Dal *numero di frame assegnati* ad un processo **dipende l'efficienza dell'intero sistema**. Il numero di page fault che un processo produce determina l'efficienza del sistema. (ogni page fault causa un rallentamento delle operazioni)

Quando si fa una sostituzione di una pagina, la pagina sostituita come si sceglie?

Ogni processo ha delle pagine assegnate su cui lavora

Gli algoritmi di sostituzione di pagine possono agire secondo:

- **allocazione locale**: la pagina da rimuovere deve essere locale a quel determinato processo
- **allocazione globale**: la pagina da rimuovere deve essere fra tutte le possibili pagine

Esempio:

	Age
A0	10
A1	7
A2	5
A3	4
A4	6
A5	3
B0	9
B1	4
B2	6
B3	2
B4	5
B5	6
B6	12
C1	3
C2	5
C3	6

(a)

Si debba sostituire una pagina, considerando Age = ultimo referenziamento:

- Quale pagina si sostituisce? Le pagine fra il processo A? (processo con Age più basso)
- Se tolgo A5 allora si parla di **sostituzione locale** (rimozione della pagina fra le pagine assegnate a quel processo)
- Se tolgo B3 allora si parla di **sostituzione globale** (rimozione della pagina fra tutte le pagine disponibili -> **anche di altri processi**)

In caso di allocazione globale, il numero di frame allocati non è più fisso ma diventa **DINAMICO** (in caso di page fault -> capiterà che **un processo avrà un frame IN PIU' e un processo avrà un frame IN MENO**)

In caso di allocazione locale, i frame allocati a quel processo sono **STATICI** (quindi non cambiano mai)

In caso di allocazione globale, il **numero di fault** che ne vengono fuori non dipende più dal singolo processo ma **da fattori non controllabili da tale algoritmo** (perchè aumenta la possibilità di fault per i processi che perdono frame). Diversamente dall'allocazione locale, il numero di frame dipende esclusivamente dal **processo stesso**, quindi **controllabili da lui**

Complessivamente l'allocazione globale **sembra essere la scelta più efficiente** a livello di produttività del sistema. Per questo motivo è il metodo più comune utilizzato

L'obiettivo, per limitare i page fault, è quello di cercare costantemente di **assegnare il numero minimale necessario di frame** per avere una maggiore produttività.

Ogni singolo processo non deve scendere sotto una **QUOTA MINIMA DI FRAME ASSEGNOTI**. Se ciò accade, il processo viene **SOSPESO** e attende l'assegnazione dei frame minimi

- Se si scende sotto il numero minimo di frame necessari per un processo, **vi sono frequenti e consecutivi page-fault**. Si parla allora di **TRASHING**.

Quando il trashing **riguarda tutti i processi**, si parla di **SISTEMA IN SOVRACCARICO**

Se ho solo una pagina e quindi un frame, allora i page-fault sono per ogni richiesta di pagina perchè la stessa locazione deve essere sostituita

- Se i fault iniziano a diventare **troppi** (trashing) vuol dire che **si devono allocare più frame** perchè sono meno di quelli minimali per quel processo
- Se i fault iniziano a diventare **tropo pochi**, allora vuol dire che quel processso ha **troppi frame assegnati**
- Si deve mantenere un **numero di fault** in un **determinato RANGE** che consentono prestazioni migliori per l'intero sistema

Vantaggi e strategie per migliorare le prestazioni

Il trashing si può ridurre ed eliminare. Invece non si può eliminare il page fault.

Come si elimina il trashing? Come si fa a sapere di quanti frame ha bisogno un processo?

Bisognerebbe assegnare un numero di frame commisurato alle "necessità del processo":

L'allocazione dei frame aiuta ad allocare i frame ma non dà un numero esplicito minimo di frame necessari per un processo.

Concetto di località

Mentre un processo viene eseguito, esso si sposta da una parte all'altra. Una **LOCALITA'** è un **insieme di pagine** che vengono **usate attivamente tutte insieme**. Ne segue che un programma è composto da **differenti località** che pian piano andranno a sovrapporsi.

Programma, struttura delle istruzioni ecc...

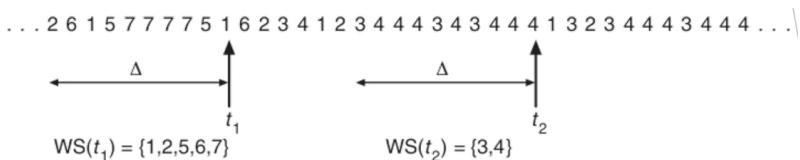
Quando si avvia un processo, lavora su una località.

Definizione: Tutti i programmi mostreranno la struttura di riferimento della memoria di base

- Quando si prende per la prima volta una determinata località, allora in quel momento si va a prendere qualcosa di nuovo che non è ricato in memoria. Quindi, ne segue, che si hanno i page fault
- Alla fine della località avrà anche dei page fault perché si sta uscendo per avviare qualcosa (una sequenza di istruzioni) di nuovo
- All'interno della località non si dovrebbero avere frequenti page fault

Working Set (WS)

L'idea è quella di osservare con attenzione il numero di **frame e pagine utilizzate dai processi** nell'ultimo Δ intervallo di tempo.



- Si identifica, durante l'esecuzione del processo, se vengono referenziate un numero di pagine sufficienti o meno.

Il Working Set risiede in memoria e tutte le pagine al suo interno sono pagine che non generano page fault perché, appunto, sono quelle utilizzate in quell'istante.

Il WS si basa su Delta: Δ = **dimensione della finestra temporale**, quindi contiene il numero di pagine utilizzate in quell'istante

- Ne segue che, se una pagina non è attiva, allora essa non si troverà nel WS
- Δ rappresenta un'approssimazione del concetto di località del processo
- Ci interessa il **numero di frame utilizzati** nel Δ intervallo di tempo

L'efficienza del **WS** è determinato dalla **dimensione del Δ** : più è grande Δ , più si ha precisione nell'osservazione del numero di frame usati dal processo. Quindi Δ evidenzia la **precisione**. Per quanto riguarda Δ :

- Se è **troppo piccolo**, non si riesce a capire il numero esatto di frame usati, e si ha una **PICCOLA LOCALITA'**
- Se è **troppo grande**, si ha una cronologia maggiore, ma si avrà comprendere nel WS informazioni dove le **LOCALITA' GRANDI CHE SI SOVRAPPONGONO**

Se $D = \sum WSS_i$ è la dimensione del WS ($WWS_i = \text{Working Set Space}$), si può calcolare la **richiesta totale** di frame che il processo ne ha di bisogno. D deve essere minore della memoria disponibile

Vantaggio nell'uso del WS: il *Sistema Operativo* controlla *WS* di ogni processo per capire quanti frame assegnare a ogni singolo processo

Prepaginazione

Se si tiene conto del WS, **dopo** che il processo **si sospende** (*si memorizza l'ultimo WS per quel processo*) e poi deve essere **RIESEGUITO**, si va a **controllare il suo WS** e quindi il numero di pagine usate nell'ultimo Δ e quindi si sa **QUANTE** e **QUALI** pagine ha frequentemente utilizzato

L'operazione che salva l'ultimo WS di un processo dopo essere stato sospeso, permette di **CARICARE IN ANTICIPO** l'ultimo WS per quel processo e in questo caso si parla di **PREPAGINAZIONE**. Decisamente si risparmiano i passati iniziali page fault *obbligatori* per ogni processo (*tranne quelli che sono avviati per la prima volta, ovviamente*).

Calcolo WS

- Attraverso **interrupt periodici**: valuta quali pagine sono state chiamate in causa
- **BIT** di referenziamento R
- un **LOG** che conserva la **cronologia** dei referenziamenti legati al Δ :

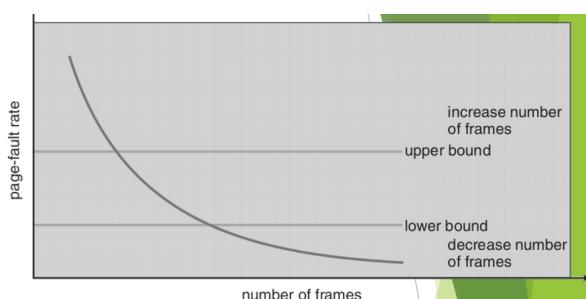
Ci si deve fare un'idea dei frame da assegnare.

Page fault frequency

L'obiettivo principale del sistema operativo è quello di **ridurre** il numero di page fault e, in particolare, **ridurre a 0** il trashing.

Il trashing si ha quando è dovuto al numero (inferiore del minimo) di frame assegnati a un processo.*
Aumento Memoria -> Diminuisco Page Fault -> Diminuisco Trashing

QUOTIDIANAMENTE si controlla il numero di page fault generati da un processo. Si parla allora di **Page Fault Frequency**.



Si definiscono **Upper Bound (UB)** e **Lower Bound (LB)** i limiti superiori/inferiori del numero di page fault:

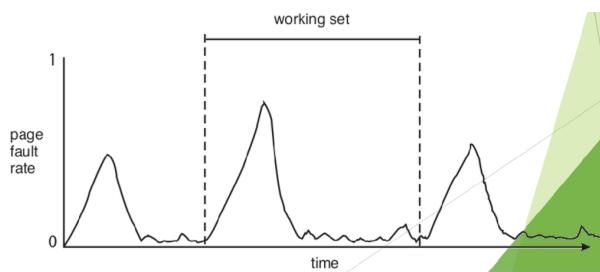
- Se il numero di page fault ricade nell'intervallo (UB, LB) vuol dire che si ricade in un **numero normale di page fault**
- Se si **scende sotto la soglia del LB (inferiore)**, allora vuol dire che il numero di page fault è **BASSO**. Ne segue che si hanno **TROPPI FRAME** assegnati per quel processo, quindi *PIU' del dovuto*. Quindi alcuni di loro *possono essere assegnati ad altri processi*.
- Se si **sale sopra la soglia UB (superiore)** vuol dire che il numero di page fault è **ALTO**. Ne segue che si hanno **POCHI FRAME** assegnati per quel processo, quindi *MENO del dovuto*. Quindi si devono **aggiungere nuovi frame da altri processi**

REMINDER

Quando si entra in una località, si accede a qualcosa di nuovo, quindi si genereranno page fault obbligatori.

Quando si è dentro la località, non dovrebbero esserci page fault perchè tutto è caricato in memoria

Quando si esce dalla località, si generano nuovi page fault perchè si devono caricare nuovi dati dalla memoria



- All'ingresso della località si ha il numero massimo di page fault (PICCO MASSIMO nel grafico)
- Il picco indica quando ci si trova all'interno di una nuova località: vuol dire che tutto quello che serve è caricato in memoria.
- Si hanno **diversi picchi** perchè, essendo che non vengono caricati PEZZI DI PROGRAMMA, allora "le istruzioni successive" dello stesso programma devono essere caricate. Quindi avviene il **PASSAGGIO FRA LOCALITA'**

Il **trashing** si riduce (*quasi a 0*) monitorando il numero di page fault che ogni processo esegue e mantenendosi all'interno dell'intervallo (LB, UB)

Politica di Pulitura

Per rendere i sistemi *più efficienti* bisogna garantire che ci sia un numero opportuno di **frame liberi immediatamente disponibili** in caso di page fault.

Vi è un **meccanismo** che spesso **svuota questi frame liberi** (*non è necessario farlo nell'immediato*, tipo quando si deve attendere una risposta e si è in attesa), se occupati: il meccanismo è eseguito da un **Processo Demone**, chiamato **PAGING DAEMON**.

Questo processo potrebbe, eventualmente, anche **ripescare tali frame in caso di richiesta** perchè, in teoria, quando si "svuota la memoria" si imposta un **FLAG libero** ma effettivamente non viene proprio svuotata. In questo caso è possibile recuperarne il contenuto cambiando il flag precedentemente impostato

Dimensione della pagina

- Solitamente questo parametro è scelto dal SO. In base alle dimensioni delle pagine ci sono vantaggi e svantaggi ma, comunque, sono sempre *potenze di 2*.
- La **dimensione della pagina** dipende dalla **tabella delle pagine**
- Si hanno bisogno di *più pagine* per memorizzare *più informazioni*
- La dimensione di una pagina è uguale alla dimensione di un frame, quindi più sono grandi i frame, più si riducono i page fault

Vantaggi di una pagina grande

- Si **minimizzano i page fault** visto che la dimensione della pagina è uguale alla dimensione del frame
- Richiede una **piccola tabella delle pagine**, cioè si devono registrare meno pagine
- Migliore **efficienza nel trasferimento I/O**.
- Maggiore è la **frammentazione interna**
- Si impiega più tempo per la fase di scrittura/lettura sulla pagina

Vantaggi di una pagina piccola

- Si ha una **minore frammentazione interna**
- Migliore risoluzione nel definire il working set in memoria perchè si ha *più dettaglio*
- Richiede la **tabella pagine grande**
- Si impiega meno tempo per la fase di scrittura/lettura sulla pagina

In generale...

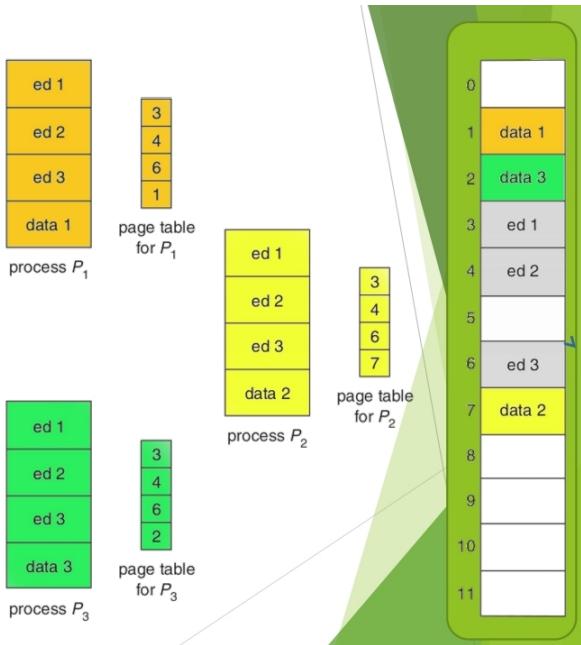
Osservazione: in media **metà dell'ultima pagina viene sprecata**. Seguendo questa osservazione **conviene usare una pagina di piccole dimensioni**.

Esempio:

- Latenza=8ms
- Tempo di ricerca=20ms
- Tempo per trasferire 512b = 0,2ms
 - Se la pagina è 512, il trasferimento avviene in 28.2 ms
 - Se la pagina è 1024, il trasferimento avviene in 28.4

Pagine condivise

E' possibile condividere del codice che deve essere **rientrante**, cioè non che non cambia durante l'esecuzione. Se non cambia mai allora si può condividere in **modalità LETTURA** con i processi che lo richiedono.



`ed1` = editor 1, `data1` = dati 1

Si condivide tutto ciò che è condivisibile mentre i dati per ogni singolo utente saranno indipendenti gli uni dagli altri.

Una sola copia di un editor (ed) sarà in memoria.

Gestione della cache

Se lavora su indirizzi logici, è direttamente lei che comunica con la CPU. Altrimenti ci deve essere MMU che traduce gli indirizzi.

Quando si usa una cache con indirizzi virtuali, allora rende la ricerca al suo interno molto più veloce visto che non serve passare dalla MMU per determinare l'indirizzo fisico corrispondente a uno specifico indirizzo virtuale.

Problema degli ASID

Nella parte della convivzione, diversi indirizzi virtuali, fanno riferimento ad un unico indirizzo fisico. In memoria si ha un unico blocco al quale fanno riferimento diversi processi.

Quindi diversi indirizzi virtuali puntano allo stesso indirizzo fisico

Potrebbe sorgere un **problema di coerenza** dovuto proprio a questo aspetto appena descritto. Il problema di incoerenza è detto **ALIASING**

- Per risolvere questo problema si può usare una cache *Virtually Indexed Physically Tagged (VIPT)*, cache che **fa uso di indirizzi virtuali e tag fisici**. Indirizzo virtuale come indice di ricerca e indirizzo

fisico come tag.

- **VANTAGGI:** Rispetto a una cache con indirizzi fisici, la latenza è inferiore perché **si può eseguire in parallelo con la ricerca della TLB**
- **Il tag non può essere confrontato finché non sarà trovato un indirizzo virtuale corrispondente**

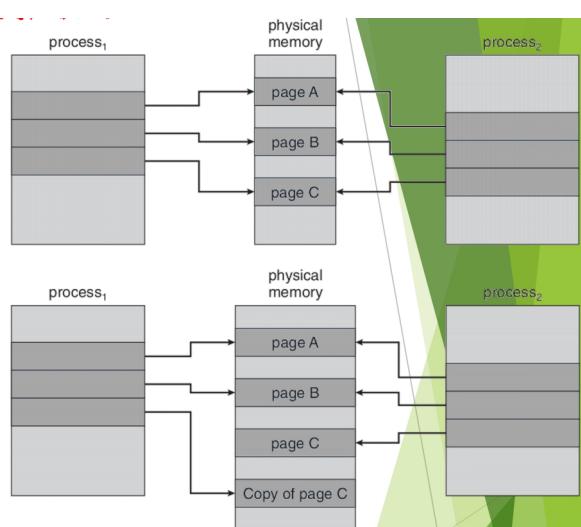
Nel caso di cache con indirizzi fisici, per capire se si tratta di un duplicato si deve aspettare che la TLB dia in output l'indirizzo fisico

Nel caso di cache con indirizzi virtuali, si riescono a individuare i duplicati che puntano allo stesso frame in maniera molto più velocemente

Se uso la **tavella delle pagine invertite**, allora la memoria condivisa è implementata in maniera tale che non viene considerata l'ipotesi "Più indirizzi virtuali corrispondono a un unico frame" ma viene assunto che la **connessione virtuale-frame è 1 a 1 e basta**.

- Questo approccio funziona in maniera semplice quando si ha un **SINGOLO CORE**
- Diventa più complessi quando si ha un sistema **MULTICORE (MULTIPROCESSORE)**

Visto che **l'associazione indirizzo virtuale-frame è 1 a 1**, quando gli altri indirizzi virtuali fanno riferimento allo stesso frame e si tenterà di accedervi, **avrà un page fault**, ma in questo caso, il SO capisce che l'indirizzo è semplicemente **mappato da un altro indirizzo virtuale**



Se accadono delle modifiche alla pagina condivisa durante l'esecuzione dei processi (anche se un processo figlio modifica un file condiviso `fork()`), visto che le pagine vengono etichettate di *sola lettura*, si riceverà un errore e si ha una TRAP che passa al SO:

- Dopo la TRAP, si devono creare necessariamente 2 copie del file condiviso per i processi che lo devono modificare. In questo caso il processo padre/figlio avranno copie di file diverse, ma non conterranno l'intero blocco di memoria (perchè è stata modificata solo una parte del file condiviso e quindi non ha senso copiare di nuovo la parte non modificata) ma si usa la tecnica **COPY-ON-WRITE**: le porzioni inalterate continueranno a essere condivise e verranno copiate fra i 2 processi solo le parti del file che sono state modificate. (Si **condivide finché è possibile**)

Da qui nasce un altro problema: la *parte da copiare* (che dovrà essere modificata) dovrà copiare questi *dati in un nuovo blocco di memoria* del nuovo processo, che dovrà essere cercato. **Trovare blocchi liberi** è proprio il problema che segue.

Molti SO, per migliorare le prestazioni complessive, tengono un **POOL DI FRAME LIBERI** in modo tale da usarli proprio in casi come questi e avere già a disposizione i frame disponibili per l'occupazione.

I "blocchi liberi" non sono effettivamente liberi perchè i dati sono comunque lì ma non sono raggiungibili. Per svuotare un blocco di memoria si usa la tecnica **ZERO-FILL-ON-DEMAND** che azzerà i blocchi prima che vengano assegnati nuovamente. Vengono cancellati, attraverso una sequenza di zeri, i contenuti precedenti del frame stesso.

VANTAGGI:

- questo azzeramento dei dati viene gestita dal kernel. Ne segue che vi è una **maggior efficienza** nell'azzeramento dei blocchi
- incrementa la sicurezza perchè si garantisce che il blocco sia effettivamente libero
- *può avvenire in qualsiasi momento*, cioè quando **il sistema è inattivo**. Quindi non avviene proprio quando si ha necessità

Librerie condivise

Ci sono 2 modi di gestire i collegamenti alle librerie:

- **LINKING STATICO**: Le librerie vengono incluse nel codice **in fase di generazione dell'eseguibile**, quindi nella sua immagine binaria di per sè
- **LINKING DINAMICO**: Il collegamento viene posticipato **in fase di esecuzione**. All'interno dell'eseguibile viene messo uno stack che indica *come individuare la relativa libreria richiesta*, se è già in memoria, oppure indica *come deve essere caricata*
 - In questo caso si *risparmia sullo spazio in RAM e su disco*
 - L'approccio dinamico porta vantaggi anche quando si aggiornano le librerie. Si deve trovare un modo per distinguere le versioni delle librerie. Si usa un contatore: quando gli aggiornamenti sono superiori rispetto a una certa soglia, questo contatore viene incrementato di 1. In tal caso ci si deve riferire alle nuovi versioni di libreria ma comunque i programmi potranno scegliere quale versione di libreria usare. Questo meccanismo prende il nome di **LIBRERIE CONDIVISE**

File mappati in memoria

Le librerie condivise fanno parte di una struttura molto più generica detta "**FILE MAPPATI IN MEMORIA**"

Ogni volta che si fa una chiamata di sistema significa che si deve accedere al disco, quindi il sistema rallenta.

Per migliorare le prestazioni in genrale è quello di **trattare i file I/O in modo alternativo**, cioè come se fossero dei classici accessi di routine alla memoria. Vuol dire **associare una parte dello spazio degli indirizzi virtuali al file**.

- Questo approccio si chiama **MAPPATURA IN MEMORIA DI UN FILE**: si mappa il blocco del disco a una o a più pagine mediante *demand-paging* (**paginazione su richiesta**: cioè che all'inizio del programma si hanno tutti i page fault quindi verranno creati i link fra indirizzi virtuali e frame)
 - | Usando questo approccio si ha **un accesso più veloce al file**
 - Queste operazioni (lettura, e in particolar modo la scrittura) non sono sincrone ma possono avvenire in base a determinate condizioni

Mappatura

Essa avviene una specifica chiamata di sistema (`mmap`).

In generale, diversi SO, a prescindere se è richiesta o meno, per migliorare le prestazioni, operano per default ed effettuano la mappatura del file in memoria senza considerare se è stata eseguita una `mmap` o meno.

E' possibile consentire a più processi di mappare lo stesso file allo stesso momento. In questo caso, un'eventuale modifica è visualizzabile da tutti gli altri processi che mappano la stessa copia del file

La condivisione dei file mappati in memoria è simile e analogo alla memoria condivisa. Molti SO usano diverse chiamate di sistema per diverse operazioni (`mmap()`)

La memoria condivisa si avrà usando le chiamate di sistema `shmget()` (*share memory get) e `shmat()` per agganciare la memoria condivisa.

Allocazione memoria per il kernel

- In generale, la memoria che è unica, serve per le operazioni di sistema gestite dal kernel.
- La suddivisione dei blocchi deve essere in parte per l'utente (paginata ma con frammentazione interna) e in parte per le attività di sistema.
- Un processo eseguito dal kernel è diverso da un processo utente.
- Per questo motivo si ha che le due porzioni sono separate

In generale:

- Tenendo conto che metà di una pagina è sprecata, viene mantenuta una porzione di memoria per le attività del kernel per alcuni motivi:
- Il kernel richiede l'uso di strutture dati di vario tipo che sono spesso molto più piccoli (in dimensioni) di una pagina. Quindi gestirle attraverso una pagina porterebbe ad avere grossa frammentazione interna
- Il kernel cerca di occupare la memoria riducendo il più possibile la frammentazione. Per questo motivo il kernel si **AUTOGESTIONA**
 - Le pagine sono assegnate a frame disposti anche in maniera non contigua
 - In memoria, invece, le informazioni sono disposti in maniera contigua per accelerare le operazioni
 - Visto che si devono cercare parti di memoria non contigui, allora il kernel cerca blocchi contigui per sbrigarsi. Allora, in alcuni casi, **si ha un'impossibilità di paginare l'allocazione** e per questo motivo il kernel gestisce in maniera diversa queste situazioni rispetto all'utente

Per allocare memoria al kernel si usa il metodo **SLAB ALLOCATION**: si alloca in maniera efficiente ed è in grado di ridurre la frammentazione interna

E' formato da 2 parti:

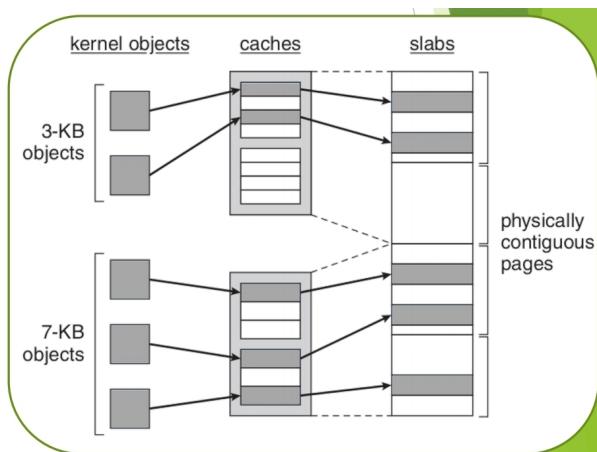
- **slab**: sequenza di una o più pagine **fisicamente contigue**
- **cache**: contiene *una o più slab*
- Ogni cache esiste per ciascuna struttura dati usata dal kernel (*cache per il descrittore dei processi, cache per i semafori ecc...*) ed è popolata da slab che sono istanze delle strutture dati dello **stesso tipo**.

Lo slab allocation usa le cache per memorizzare oggetti del kernel, appunto:

- quando viene creata una cache omogenea, inizialmente molti oggetti allocati vengono contrassegnati come "liberi". Il numero di oggetti nella cache dipende dalla sua dimensione
- Una volta assegnato uno slab, esso diventa "*utilizzato*"

Esempio:

- slab da 12K
- Composto da 3 pagine e ognuna formata da 4K
- Quanti oggetti si possono memorizzare? 6 oggetti da 2K



Gli slab si possono trovare in **3 stati diversi**:

- **PIENO**: tutti gli oggetti dello slab sono contrassegnati come "occupati" (o "usati")
- **VUOTO**: tutti gli oggetti dello slab sono contrassegnati come "liberi"
- **PARZIALE**: alcuni oggetti dello slab sono contrassegnati come "liberi" e altri come "occupati"

In base allo stato dello slab:

- quando si riceve la richiesta di trovare un oggetto vuoto, viene ricercato all'interno dell'insieme degli slab che sono nello stato **PARZIALE**
- se non ci sono slab nello stato parziale si passa allo stato **VUOTO**
- se non ci sono nemmeno questa volta, allora si deve **ALLOCARE UN NUOVO SLAB** e verrà assegnato ad una determinata cache allocando *pagine fisicamente contigue*

Vantaggi dell'algoritmo dello Slab Allocation

- **NESSUNA MEMORIA VIENE SPRECATA** per frammentazione: il kernel richiede ***la quantità esatta di memoria*** per quell'oggetto
- le richieste di memoria possono essere eseguite molto rapidamente (**efficienza**). L'efficienza è più elevata quando si hanno **frequenti richieste di allocazione e deallocazione**

File System

La parte del SO che si occupa dei dati memorizzati viene detto **FILE SYSTEM**. Essa gestisce la creazione, modifica, lettura, modo di accesso ecc..

I file rappresentano sia **programmi** (codice ecc..) e sia **dati**. E' una sequenza di byte o di record il cui significato lo dà il creatore.

Ogni file ha deti dettagli:

- tipo di file
- nomenclatura
- posizione nel file system
- dimensione del file (data in byte o *in blocchi*)
- i tipi di accessi permessi a tale file
- data di creazione/modifica con il relativo proprietario

La **struttura di directory** risiede **nella memoria secondaria** e tiene tutte le informazioni relative ai file.

Dall'**identificatore del file** si riesce a risalire a tutti gli **attributi del file**.

La directory è un file speciale che può contenere altre directory o file

Le operazioni possibili su un file sono: lettura, scrittura, esecuzione, creare, spostamento.

Problema: (da approfondire da altri appunti)

- Tutte le operazioni possono essere fatte **contemporaneamente da diversi processi**
- In questo caso, quando più processi operano all'interno di un file, si usa una **tabella a 2 livelli**:
 - **un livello** viene usata per ogni processo (tiene conto del numero di processi che usano quel file) e punta a una tabella di file a un'altra livello di sistema
 - **nel secondo livello** usa una *tabella a livello di sistema* (contiene le **informazioni indipendenti del processo**).
- Ogni volta che viene eseguita una open() viene incrementato un contatore all'interno del file che tiene conto del numero di aperture. Di conseguenza tiene traccia di quanti processi stanno usando quel file. (in caso di close() il contatore viene decrementato)

Alcuni SO usano dei meccanismi per bloccare un file quando è aperto, quindi evitare il suo uso da altri processi. Si usano i **LOCK**. Possono essere:

- **shared**: possono essere acquisiti allo stesso momento da più processi
- **exclusive**: quando si ottiene lock di scrittura (se il file è libero da qualsiasi altro lock scrittura) allora nessun altro può acquisire un lock di scrittura finché il primo processo non termina
- **mandatory**: il SO impedisce a qualsiasi altro processo di poter accedere al file bloccato mentre un altro ha il lock di scrittura
- **advisory**: è un blocco di avviso. SO non blocca l'acquisizione di lock da parte di un processo

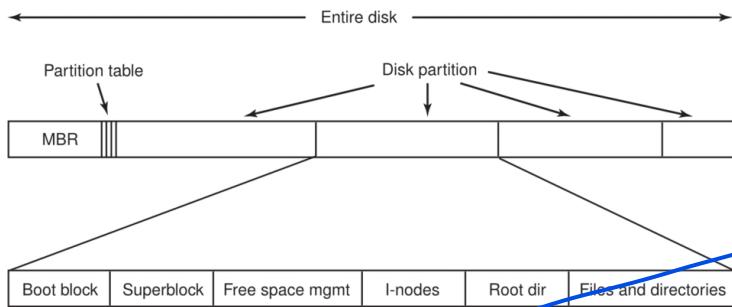
Molti SO usano i lock esclusivi sui file

Possono esserci anche file **REGOLARI** binari o ASCII e **SPECIALE** a carattere o a blocchi

Struttura file system

Un disco è diviso da una o più **partizioni** e in ognuna di essa si installano SO diversi e ne segue che i file system sono diversi.

La parte iniziale è sempre il settore 0, chiamato **MBR** (*Master Boot Record*). Contiene una serie di tabella relative alle partizioni (vuote o contenenti altri file system) e indicano gli indirizzi iniziali/finali delle partizioni. Fra tutte le partizioni, una è segnata come **ATTIVA** ed è quella che verrà presa in considerazione all'avvio della macchina



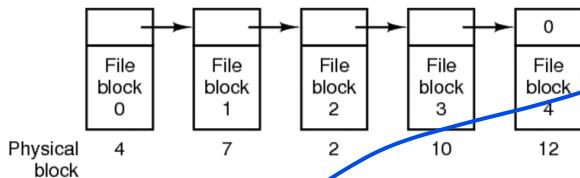
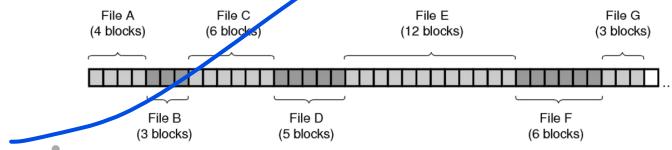
Una partizione del disco contiene:

- **BLOCCO DI AVVIO**
- **~~SUPERBLOCCO~~**: contiene tutti i parametri necessari al file system per l'avvio: il tipo di file system e il numero di blocchi che ha e altre informazioni amministrative
- **BLOCCHI LIBERI DI FILE SYSTEM**: sono blocchi liberi sottoforma di **bitmap** o di **puntatori**
- **I-NODE**: è un array, una per ciascun file, che contiene tutte le info relative al file stesso, tranne il suo contenuto
- **ROOT DIRECTORY**: è la radice dell'albero del file system
- **FILES E DIRECTORY** del file system

Implementazione file

l'implementazione può avvenire mediante allocazione:

- **contigua**:
 - Ha un limite: l'**accesso al blocco è diretto**, quindi **molto veloce**. Siccome si va a blocchi contigui, questo è **soggetto a frammentazione interna/esterna**
- **con le liste concatenate**: contiene un puntatore al successivo nodo e una parte dedicata ai file. In questo caso si ha che:
 - Il problema della frammentazione interna rimane ma si evita la frammentazione esterna. La ricerca **non è diretta** e quest'operazione è molto **lenta**



La tabella di allocazione del file **FAT** viene caricata per andare interamente in memoria. Quindi:

- non richiede accessi al disco visto che già si trova in memoria
- occupa spazio in memoria, chiaramente (ma non è di grandi dimensioni, quindi è **trascurabile**)
- il blocco di dati è sempre disponibile mediante puntatori
- È più usato nei sistemi Windows mentre in Linux si usa I-Node che contiene le info inerenti a un file e ha il seguente vantaggio: tiene traccia mediante un indice dei vari blocchi di un file. Rispetto al FAT risulta **più piccola in dimensioni**

