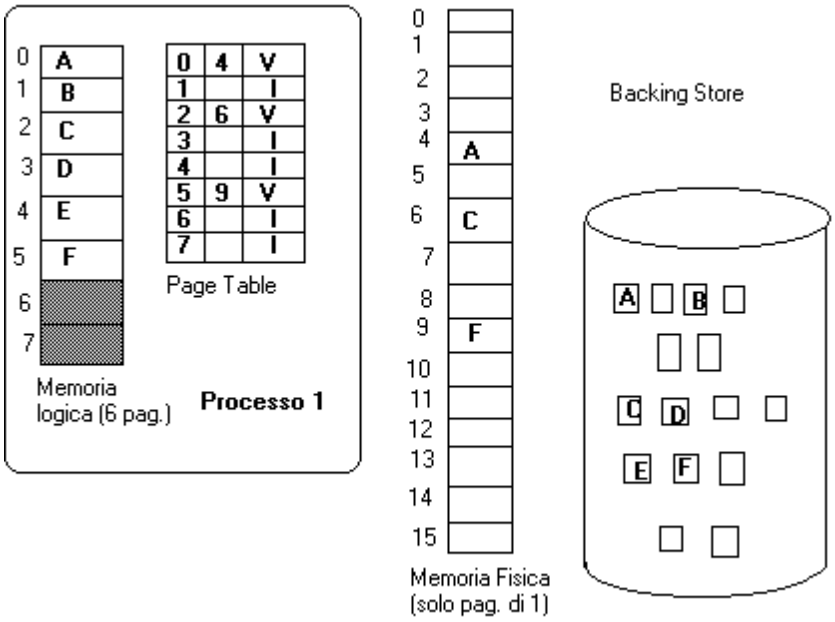
Memoria virtuale

Tecniche per eseguire processi che non sono interamente contenuti in memoria.
Metodo fondamentale e' la **<u>Paginazione a richiesta</u>** (*demand paging*)

Supponiamo di avere un backing store che implementa lo swapping, ma dotato di *<u>swapper pigro</u>*: non si caricano le pagine in memoria fintantoche` non e` richiesto il loro contenuto

Al primo tentativo di usare una pagina non presente in memoria la page table non riesce a tradurre l'indirizzo logico in indirizzo fisico **➔ TRAP pagina non valida.**



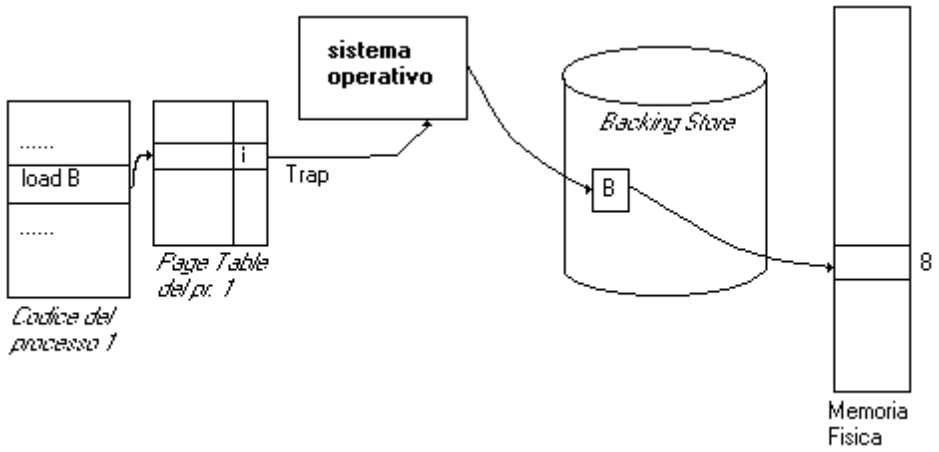
Nella figura vediamo solo le pagine del processo 1 caricate nella memoria fisica. Altri frames conterranno altri processi, il sistema operativo, oppure potrebbero essere vuote.

Supponiamo che per hardware servano 3 bits per indirizzare la pagina (valori 0-7).

- Se un processo necessita di meno pagine, in corrispondenza nella page table porremo il bit “invalido” (e’ il caso delle pagine 6 e 7, che non esistono). Un tentativo di accedere a queste pagine provoca una trap (“indirizzo inesistente”).

- Se invece si tenta di accedere alla pagina “**B**” nessun frame e’ associato ad essa perche’ e’ ancora nel backing store ➔ ancora presenza di bit “invalido” e di conseguenza trap (“page fault”).

NOTA: per “riferimento” alla pagina **B** intendiamo un qualsiasi indirizzo in cui i bits di pagina individuano **B** (quindi sono 001) e l’offset e’ qualsiasi, sia per istruzioni di lettura sia di scrittura.



Vediamo ad esempio:

1. una “load” provoca una decodifica di indirizzo di **B** tramite la page table
2. il bit “invalido” causa la trap
3. il sistema ‘ricupera” la pagina dal backing store
4. la pagina e’ ricopiata in un frame libero

Al termine del caricamento, la entry relativa a “**B**” nella page table e’ aggiornata con il frame occupato e il bit “valido”.

Gestione della trap di page fault:

- Controllo di validita' della pagina richiesta. Se non e' valida per il processo➔ processo ucciso, errore
- Si deve caricare una pagina ➔ cerco nella lista dei frames liberi se ce n'e uno disponibile
- Si invia al backing store la richiesta di lettura della pagina nel frame libero
- Quando il trasferimento e' completo, si aggiorna la page table del processo
- Il processo puo' essere rischedulato e ricominciare l'istruzione che ha causato la trap

Attenzione: ci sono operazioni molto costose da compiere tra queste. Non e' detto che tutto questo si debba eseguire "nella trap routine", a meno di non pensare che sia a sua volta interrompibile.
La trap routine potrebbe limitarsi a "marcare" il processo e inviare un messaggio al Memory Manager, che si incarichera' delle operazioni successive.

Possiamo anche pensare che inizialmente il processo non abbia NESSUNA pagina in memoria, e che tutte le pagine che gli servono in esecuzione siano caricate mediante page faults. In generale, non tutte le pagine potrebbero essere usate a run-time (es. quelle che implementano funzioni particolari che non sono usate nell'esecuzione corrente).

Hardware necessario

- Page table con invalid bit
- Backing Store
- Possibilita' di far ripartire una istruzione parzialmente eseguita

Tempo richiesto per servire un page fault del processo P

1. Trap, salvataggio stato del processo, determinazione del tipo di trap, controllo validita' pagina e sua posizione nel backing store.
2. Richiesta al backing store di lettura in un frame libero: (possibile rischedulazione)
   attesa in coda per il periferico,
   attesa tempo di latenza,
   attesa tempo di trasferimento dati
3. Interrupt dal backing store per trasferimento completato: salvataggio del processo interrotto, determinazione del tipo di trap, aggiornamento page table, cambio di stato di P che ritorna *Ready.*
4. Riattivazione di P, caricamento del suo stato e prosecuzione (dopo rischedulazione)

Le operazioni 1) 3) e 4) sono abbastanza veloci, tipiche del tempo di servizio di un interrupt. Possiamo stimarle ad es. in 100 μsec.
L'operazione 2) e' invece il tempo necessario allo swap di una pagina. Possiamo stimare intorno ai 10 msec il tempo totale per servire il page fault.
La rischedulazione garantisce che la CPU stia comunque svolgendo lavoro utile, sia pure per altri processi.
In definitiva, un accesso a memoria potrebbe richiedere

- 500 nsec per dati gia' in memoria (m_a)
- 10 msec per dati non in memoria, che provocano un page fault (m_f).

Qual'e' allora il tempo medio di accesso alla memoria?

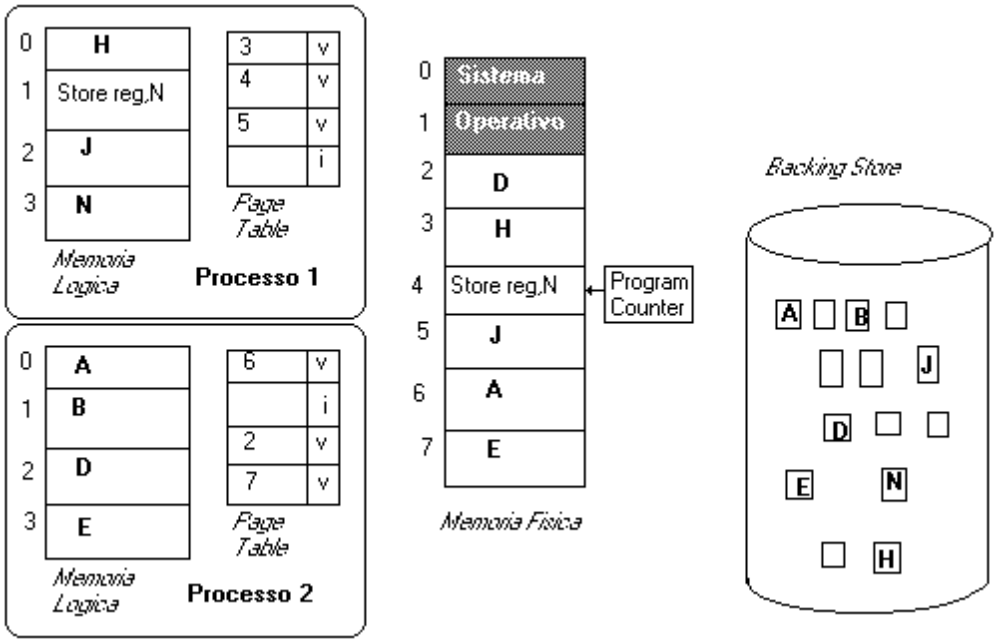
$T_{medio} = (1-p)\, m_a + p\, m_f$.

Dove p e' la *probabilita' di un page fault*.
Supponiamo che $p=10^{-3}$, un page fault ogni 1000 accessi alla memoria.

T_{medio}= 11 μsec circa ovvero l'accesso alla memoria rallenta di un fattore 11!

Se accettiamo un rallentamento intorno al 10% (caso piu' ragionevole) dobbiamo avere $p<10^{-5}$.

Osservazione: al punto 2, potremmo non avere un frame libero, situazione illustrata qui sotto.



In questo caso occorre:

- swap out di un intero processo dalla memoria, diminuendo il livello di multiprogrammazione,

oppure

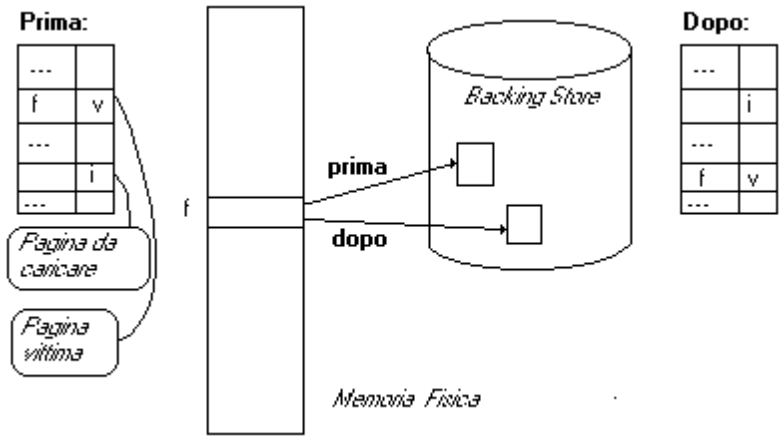
- swap out di una pagina per creare un frame libero.

Questa seconda soluzione e' detta **Page Replacement**.

Prima del punto (2) occorre quindi svolgere le seguenti azioni:

a) Verificare se esiste un frame libero, altrimenti attivare un algoritmo di page replacement per la scelta di una *pagina vittima*

b) Scrivere la pagina vittima sul backing store e aggiornare la page table della pagina vittima.



Osservazione:
Lo swap out della pagina vittima potrebbe non essere necessario, se ad es. siamo certi che non e' stata modificata rispetto alla copia presente sul backing store. Percio' la page table sovente memorizza se sono state fatte modifiche alla pagina in memoria in un apposito campo detto **dirty bit**.

Differenza tra paginazione "pura" e paginazione a richiesta:

- la paginazione "pura" e' una corrispondenza biunivoca tra indirizzi logici e indirizzi fisici;
- la paginazione a richiesta elimina la necessita' di una corrispondenza totale ed e' definita "istante per istante", infatti la traduzione di un indirizzo logico puo' essere indefinita (se la pagina non e' in memoria) o definita con valori diversi in diversi momenti dell'esecuzione del processo.

Nel caso della paginazione a richiesta useremo il termine *indirizzo virtuale* (anziché logico) e diremo che un sistema che implementa la paginazione a richiesta supporta la *memoria virtuale*.

Curiosita`: sono stati sviluppati in passato alcuni sistemi che supportano memoria virtuale tramite *segmentazione a richiesta* (anziché paginazione a richiesta).

Paginazione a richiesta

Vi sono due problemi da risolvere per ogni sistema che realizza la paginazione a richiesta:

1. Criteri per stabilire quante pagine assegnare a ogni processo (e di conseguenza quale e` il grado di multiprogrammazione del sistema): *Frame Allocation*
2. Algoritmi per scegliere una pagina vittima, in modo che si abbia una frequenza di page fault piu' bassa possibile: *Page Replacement*.

Per confrontare tra loro vari algoritmi si sceglie di solito una stringa di riferimenti a pagina, costruita da una sequenza di riferimenti a memoria (veri o generati artificialmente) cosi' semplificata:

- si memorizzano solo i riferimenti a pagina (eliminazione dei displacements)
- si eliminano tutti i riferimenti consecutivi a una stessa pagina, tenendo solo il primo.

Negi esempi seguenti useremo la stringa:

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

e supporremo che il processo abbia a disposizione soltanto 3 frames.

Algoritmo FIFO

L' elenco delle pagine e' mantenuto in ordine di caricamento in memoria. Quando occorre sostituire una pagina, la pagina vittima e' quella inserita "per prima" quindi in memoria da piu' tempo.
La struttura che mantiene l'elenco delle pagine e' una coda FIFO. La tabella mostra il contenuto della coda (=colonna, con "head" in alto e "tail" in basso). Un asterisco marca i page faults.

Osserviamo **12** page faults sulla stringa di esempio, a cui occorre aggiungere i tre page faults iniziali per caricare le prime tre pagine.

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

Questo algoritmo potrebbe essere inefficiente rispetto ad altri che vedremo, ma la struttura e' semplice da gestire. Inoltre la gestione della coda avviene "durante" i page faults.
Contrariamente all'intuizione, *non e' detto che allocando piu' pagine a un processo il numero di page faults diminuisca*. Il fenomeno e' detto Anomalia di Belady e si verifica ad esempio per la seguente stringa:

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

Si lascia per esercizio di verificare che:
- con tre frames disponibili si verificano 9 page faults;
- con quattro frames disponibili si verificano 10 page faults.

Algoritmo ottimo (OPT)

L'algoritmo sostituisce la pagina che non verra' richiesta per il tempo piu' lungo ➔ nessun algoritmo puo' fare di meglio!

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

In questo caso vi sono soltanto **6** page faults. Per funzionare pero' ci vuole la 'conoscenza del futuro'! Puo' servire come termine di confronto con gli altri algoritmi.

Algoritmo Least Recently Used (LRU)

Sceglie come pagina vittima *quella che non e' stata usata da piu' lungo tempo*. Si suppone infatti che il "futuro" possa essere simile al "passato", ovvero che una pagina non usata da tempo non verra' piu' usata, e una usata di recente servira' ancora tra breve.
Si puo' anche verificare come LRU generi gli stessi page fault di OPT se si da' in ingresso la stessa stringa di riferimenti, ma "rovesciata".
Se ordino la coda di pagine in funzione dell'utilizzo piu' recente, l'esempio diventa:

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

Vi sono quindi **9** page faults.
Come si puo' mantenere efficacemente la coda ordinata per utilizzo piu' recente?

Una idea potrebbe essere quella di memorizzare il *clock* insieme con il numero di pagina e cercare il minimo valore di clock in caso di page fault. Questo pero' presenta degli svantaggi, soprattutto perche' bisognerebbe tenere un valore di molti bits. La memorizzazione dovrebbe essere fatta ad *ogni* accesso a pagina quindi gestita totalmente dall'hardware.

In alternativa si potrebbe tenere una *stack* dei riferimenti a pagina in cui ad ogni accesso si effettua un push, rimuovendo il valore della pagina se gia' presente all'interno della stack. In caso di page fault il *bottom* della stack e' la pagina vittima. Anche questa implementazione deve prevedere un hardware dedicato altrimenti e' troppo costosa.

NOTA: ne' OPT ne' LRU soffrono dell'anomalia di Belady (si puo' dimostrare!)

Approssimazioni di LRU

Si effettuano con uso di un hardware aggiuntivo piu' semplice di quello visto sopra.

Reference bit:

e' associato alla entry nella page table (o frame table). Si esegue un Set ad ogni riferimento alla pagina. Periodicamente si fa un Clear di tutti i reference bits.

In questo modo in caso di page fault non si dispone di un "ordine" totale dei riferimenti a pagina, ma si puo' distinguere tra pagine

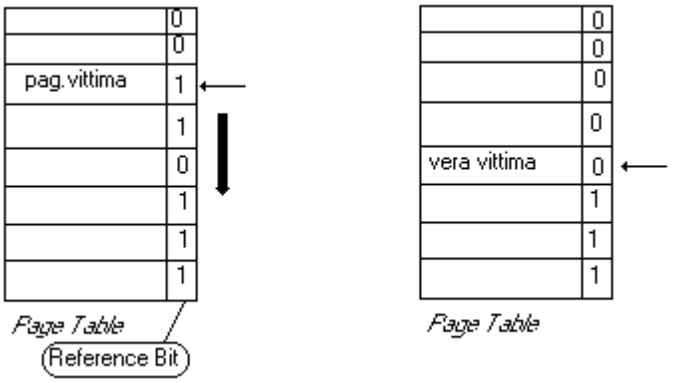
- "usate di recente" (reference bit=1) e
- "non piu' usate di recente" (reference bit=0), cioe' non piu' usate dopo l'ultimo Clear.

Algoritmo Second Chance

- Si sceglie la pagina vittima con l'algoritmo FIFO.
- Se la pagina vittima ha reference bit=0 la sostituisce.
- Se ha reference bit=1 allora:
    - pone reference bit=0
    - cerca nella coda FIFO se c'e' una pagina con reference bit=0, considerando la coda come un buffer circolare, e prende quella come vittima.

Se tutte le pagine hanno reference bit a 1, diventa FIFO. Altrimenti la pagina scelta da FIFO ha una "seconda possibilita'" di non essere vittima, se ce ne sono altre con reference bit=0.

La figura illustra la pagina vittima scelta da FIFO e quella scelta da second chance (procedendo verso il basso si azzerano tutti i reference bit).



Page Classes

Si utilizza il reference bit insieme con il dirty bit, ricavando quattro classi in cui partizionare le pagine.

Ref.bit	Dirty bit	Significato
0	0	pag. non usata rec.te ne' modificata
0	1	pag. non usata rec.te ma modificata
1	0	pag. usata rec.te ma non modificata
1	1	pag. usata rec.te e modificata

Si scelgono le vittime a partire dalle classi con numero piu' piccolo. Se ce ne sono piu' di una la vittima si sceglie FIFO o anche a caso.

Least Frequently Used (LFU)

Presuppone la presenza di "piu' di un" reference bit, ovvero di un *Reference Counter:*

il bit piu' significativo e' Set ad ogni accesso alla pagina. Periodicamente si invia un comando che shifta a destra tutto il contatore.

Si scelgono le pagine vittime nella classe che ha reference counter piu' basso, in modo da approssimare LRU.