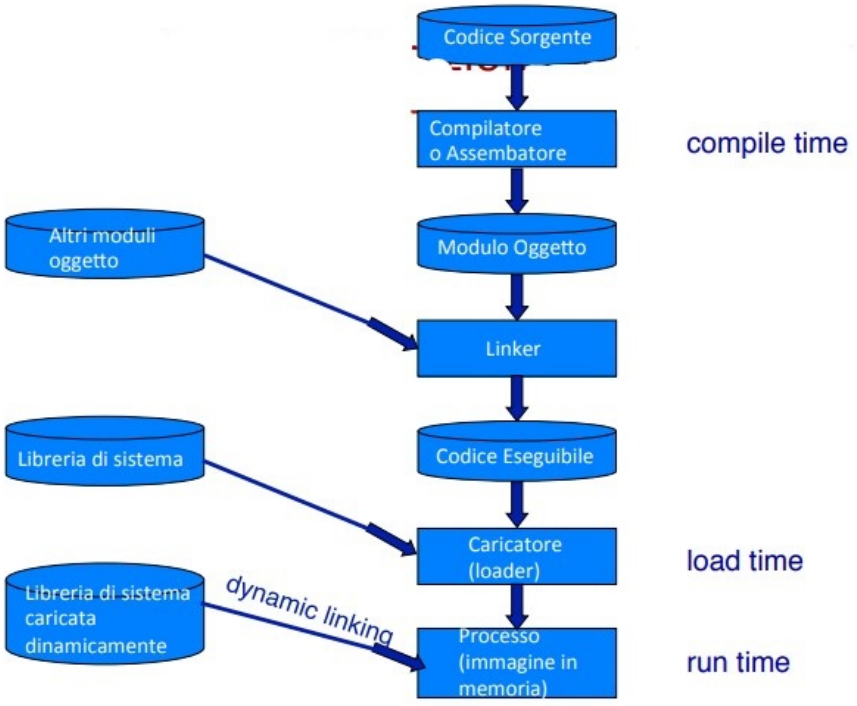
04 - Gestione Memoria Centrale

Introduzione

Un programma risiede in un disco sotto forma di file binario eseguibile. Il programma per essere eseguito deve essere caricato in memoria ed inserito all'interno di un processo.

Avremo quindi una **coda di ingresso** (input queue), cioè un insieme di processi che devono essere caricati in memoria per essere eseguiti (se è presente lo scheduler a lungo termine, la coda viene gestita dallo scheduler a lungo termine).

La procedura normale consiste nello scegliere uno dei processi appartenenti alla coda d'ingresso e caricarlo in memoria. Il processo durante l'esecuzione può accedere alle istruzioni e ai dati in memoria. Quando il processo termina, si dichiara disponibile il suo spazio di memoria. Nella maggior parte dei casi un programma utente, prima di essere eseguito, deve passare attraverso vari stadi, alcuni dei quali possono essere facoltativi, in cui gli indirizzi sono rappresentabili in modi diversi.



L'istruzione *load* richiedeva come parametro l'indirizzo della cella di memoria da cui andava letto il valore che poi veniva caricato nel registro accumulatore.

Come siamo arrivati ad avere l'indirizzo di quella cella di memoria?

In fase di compilazione e di collegamento sappiamo già dove verrà allocato in memoria centrale il nostro programma (quali indirizzi di memoria occuperà), e quindi in fase di traduzione dal programma scritto in un linguaggio di alto livello ad un programma scritto in linguaggio macchina possiamo già utilizzare gli indirizzi definitivi, reali.

Binding di istruzioni e dati a indirizzi di memoria

Quando vengono decisi gli indirizzi dove verranno allocate le variabili?

Generalmente, l'associazione di istruzioni e dati a indirizzi di memoria si può compiere in qualsiasi fase del seguente percorso.

- **Compilazione e Collegamento**. Se è noto a priori dove il processo risiederà in memoria, può essere generato un *codice assoluto*, che contiene direttamente **indirizzi assoluti** (indirizzi *fisici*, reali); se la locazione di inizio cambia, bisogna ricompilare il programma. Un esempio sono i .com di MS-DOS (sistema mono programmato).

- **Caricamento**. Se nella fase di compilazione NON è possibile sapere in che punto della memoria risiederà il processo, il compilatore genera *codice rilocabile*, che contiene **indirizzi rilocabili** (*logici*). In questo caso si ritarda l'associazione finale degli indirizzi (*binding*) alla fase del caricamento. Se l'indirizzo iniziale cambia, è sufficiente ricaricare il codice utente per incorporare il valore modificato. È compito del caricatore fare il binding tra indirizzi rilocabili e gli indirizzi assoluti (*rilocazione statica*→ tradurre gli indirizzi logici in indirizzi fisici (assoluti)).
- **Esecuzione**. Se durante l'esecuzione il processo può essere spostato da un segmento di memoria a un altro, il binding deve essere ritardato fino alla fase d'esecuzione (*rilocazione dinamica* → il codice del programma viene caricato continuando ad avere indirizzi logici, e questi vengono tradotti in indirizzi fisici solo quando viene chiesto di accedere a quegli indirizzi).
  È necessario dell'hardware di supporto per gestire questa situazione.
  La maggior parte dei sistemi operativi d'uso generale impiega questo metodo.

  Spazio logico degli indirizzi : suddivisione della memoria pensata come se si avesse tutta la memoria a disposizione.

Spazio di indirizzi logici e fisici

Per una corretta gestione della memoria abbiamo bisogno di due tipi di spazio di indirizzi separati tra loro, anche se esiste un legame che li unisce. Un indirizzo generato dalla CPU di solito si indica come **indirizzo logico**, mentre un indirizzo visto dall'unità di memoria, cioè caricato nel registro dell'indirizzo di memoria (*memory address register*, MAR) di solito si indica come **indirizzo fisico**.

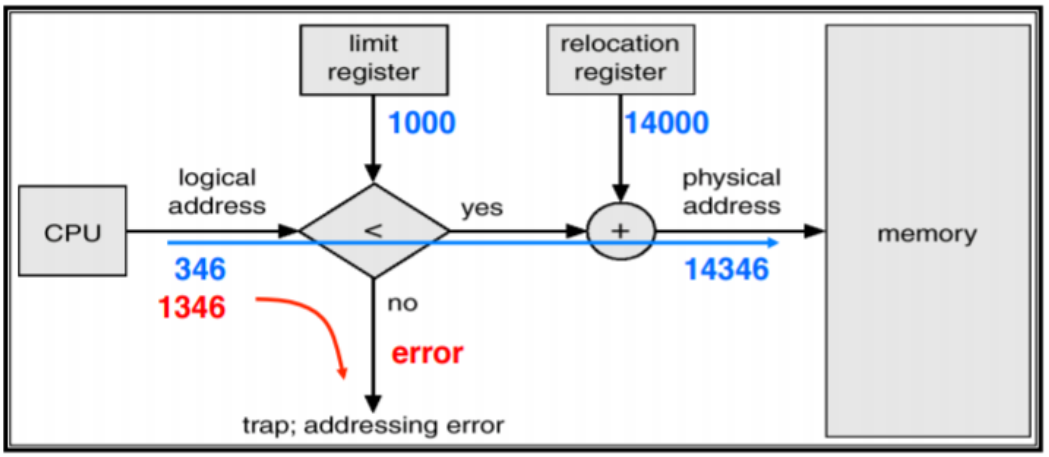
I metodi di associazione degli indirizzi (binding) nelle fasi di compilazione e di caricamento producono indirizzi logici e fisici identici. Con i metodi di associazione nella fase d'esecuzione, invece, gli indirizzi logici non coincidono con gli indirizzi fisici. In questo caso ci si riferisce, di solito, agli indirizzi logici col termine *indirizzi virtuali.*

L'insieme di tutti gli indirizzi logici generati da un programma è lo **spazio degli indirizzi logici**; l'insieme degli indirizzi fisici corrispondenti a tali indirizzi logici è lo **spazio degli indirizzi fisici**.

Il programma utente ha a che fare con gli indirizzi logici e non vede mai gli indirizzi fisici.

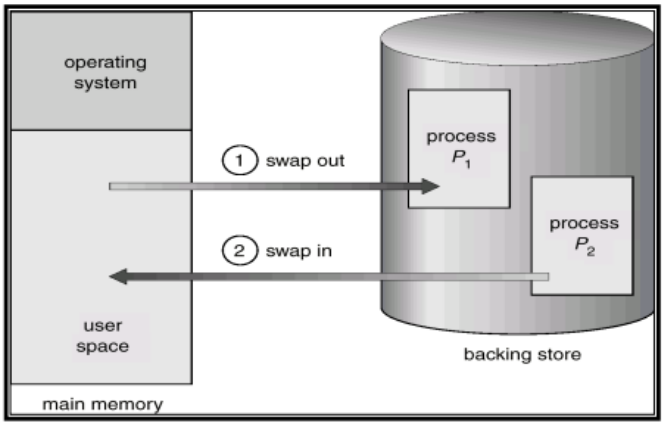
Memory-Management Unit (MMU)

L'associazione nella fase d'esecuzione dagli indirizzi virtuali agli indirizzi fisici è svolta da un dispositivo detto **unità di gestione della memoria** (*Memory-Management Unit*, MMU).
Il registro di base è ora denominato
registro di rilocazione: quando un processo utente genera un indirizzo (*logico*) , prima dell'invio all'unità di memoria, viene sommato a tale indirizzo il valore contenuto nel registro di rilocazione (ottenendo un indirizzo *fisico)*.
Ad esempio, se il registro di rilocazione contiene il valore 14000, un tentativo da parte dell'utente di accedere alla locazione 0 è dinamicamente rilocato alla locazione 14000; un accesso alla locazione 346 corrisponde alla locazione 14346.
Il programma utente non considera mai gli indirizzi fisici reali. Il programma crea un puntatore alla locazione 346, lo memorizza, lo modifica, lo confronta con altri indirizzi, tutto ciò semplicemente come un numero. Solo quando assume il ruolo di un indirizzo di memoria (magari in una load o una store indiretta), si riloca il numero sulla base del contenuto del registro di rilocazione.
Il programma utente tratta indirizzi logici, l'architettura del sistema converte gli indirizzi logici in indirizzi fisici.



Avvicendamento (Swapping)

Per essere eseguito, un processo deve trovarsi in memoria centrale; per far posto in memoria centrale ad un altro processo, si puo trasferire temporaneamente il primo processo in **memoria secondaria** (*backing store*). Successivamente, quando deve riprenderne l'esecuzione, viene riportato in memoria centrale.



Si consideri, per esempio, un ambiente di multiprogrammazione con un algoritmo circolare (*RR*) per lo scheduling della CPU.

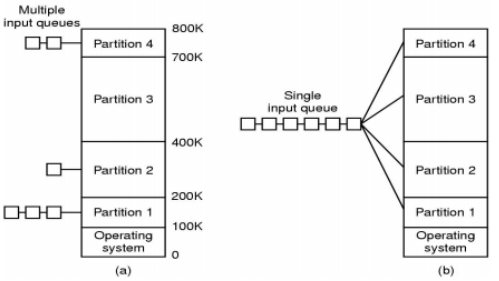
Trascorso un quanto di tempo, il gestore di memoria scarica dalla memoria il processo appena terminato e carica un altro processo nello spazio di memoria appena liberato; questo procedimento si chiama **avvicendamento dei processi in memoria** - o, più brevemente, *avvicendamento* o *scambio* (*swapping*). Nel frattempo lo scheduler della CPU assegna un quanto di tempo a un altro processo presente in memoria. Quando esaurisce il suo quanto di tempo, ciascun processo viene scambiato con un altro processo. In teoria il gestore della memoria può avvicendare i processi in modo sufficientemente rapido da far sì che alcuni siano presenti in memoria, pronti per essere eseguiti, quando lo scheduler della CPU voglia riassegnare la CPU stessa. Anche il quanto di tempo deve essere sufficientemente lungo da permettere che un processo, prima d'essere sostituito, esegua quantità ragionevole di calcolo.
Il tempo di swapping è un fattore critico.

Gestione della memoria

Allocazione contigua - Partizioni Fisse e Variabili

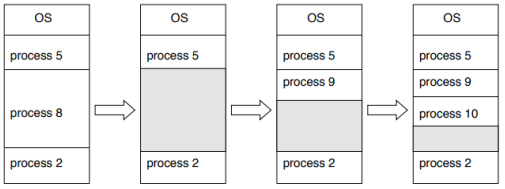
- Uno dei metodi più semplici per l'allocazione della memoria consiste nel suddividere la stessa in **partizioni** di dimensione **fissa**.
  Ogni partizione deve contenere esattamente un processo
  , quindi il grado di multiprogrammazione è limitato dal numero di partizioni.
  Quando un processo termina l'esecuzione, libera una partizione che può essere occupata da un altro processo in attesa nella coda di ingresso.

  La tecnica delle partizioni fisse presenta il problema della **frammentazione interna** : esiste un frammento della partizione interno alla stessa che non riesco ad utilizzare.



- Nello schema a **partizioni variabili** all'inizio ci sono 2 partizioni: una allocata al SO e l'altra (il resto della memoria) è libera.
  Il SO conserva in una tabella le partizioni libere (*buchi*) e le partizioni allocate, ognuna con dimensioni diverse dalle altre.
  Quando arriva un processo, viene allocato in un buco abbastanza grande da contenerlo. La parte di buco non utilizzata viene lasciata come partizione libera per un altro processo.

  La tecnica delle partizioni variabili presenta il problema della **frammentazione esterna** : complessivamente abbiamo lo spazio di memoria sufficiente per far andare in esecuzione un nuovo processo, però siccome questo spazio di memoria non è contiguo (ma è stato frammentato in tante partizioni libere distinte), il nostro processo non riesce ad andare in esecuzione.



In generale, è sempre presente un insieme di buchi di diverse dimensioni sparsi per la memoria. Quando si presenta un processo che necessita di memoria, il sistema cerca nel gruppo un buco di dimensioni sufficienti per contenerlo. Se è troppo grande, il buco viene diviso in due parti: si assegna una parte al processo in arrivo e si riporta l'altra nell'insieme dei buchi. Quando termina, un processo rilascia il blocco di memoria, che si reinserisce nell'insieme dei buchi; se si trova accanto ad altri buchi, si uniscono tutti i buchi adiacenti per formarne uno più grande. A questo punto il sistema deve controllare se vi siano processi nell'attesa di spazio di memoria, e se la memoria appena liberata e ricombinata possa soddisfare le richieste di qualcuno fra tali processi.
Questa procedura è una particolare istanza del più generale problema di **allocazione dinamica della memoria**, che consiste nel soddisfare una richiesta di dimensione n data una lista di buchi liberi. Le soluzioni sono numerose.

I criteri più usati per scegliere un buco libero tra quelli disponibili nell'insieme sono i seguenti.

- **First-fit** : alloca il primo buco di dimensioni sufficienti. La ricerca può cominciare sia dall'inizio dell'insieme di buchi sia dal punto in cui era terminata la ricerca precedente. Si può fermare la ricerca non appena s'individua un buco libero di dimensioni sufficientemente grandi.

- **Best-fit** : alloca il buco più piccolo in grado di soddisfare la richiesta. Si deve compiere la ricerca in tutta la lista, sempre che questa non sia ordinata per dimensione. Tale criterio produce le parti di buco inutilizzate più piccole.
- **Worst-fit** : alloca il buco più grande. Anche in questo caso si deve esaminare tutta la lista, sempre che non sia ordinata per dimensione. Tale criterio produce le parti di buco inutilizzate più grandi, che possono essere più utili delle parti più piccole ottenute col criterio precedente.

Con l'uso di simulazioni si è dimostrato che sia first-fit sia best-fit sono migliori rispetto a worst-fit in termini di risparmio di tempo e di utilizzo di memoria.

Frammentazione

- Entrambi i criteri first-fit e best-fit di allocazione della memoria soffrono di **frammentazione esterna**: quando si caricano e si rimuovono i processi dalla memoria, si frammenta lo spazio libero della memoria in tante piccole parti. Si ha la frammentazione esterna se lo spazio di memoria totale è sufficiente per soddisfare una richiesta, ma non è contiguo; la memoria e frammentata in tanti piccoli buchi. È tipica delle *partizioni variabili*.

  Una soluzione al problema della frammentazione esterna è data dalla **compattazione**. Lo scopo è quello di riordinare il contenuto della memoria per riunire la memoria libera in un unico grosso blocco.

  La compattazione tuttavia non è sempre possibile: non si può realizzare se la rilocazione è statica ed è fatta nella fase di assemblaggio o di caricamento; è possibile solo se la rilocazione è dinamica e si compie nella fase d'esecuzione.
  Quando è possibile eseguire la compattazione, è necessario determinarne il costo.
  Il più semplice algoritmo di compattazione consiste nello spostare tutti i processi verso un'estremità della memoria, mentre tutti i buchi vengono spostati nell'altra direzione formando un grosso buco di memoria.
  Questo metodo può essere assai oneroso.
- Tipica invece delle *partizioni fisse* è la **frammentazione interna**: la memoria allocata può essere più grande di quella che effettivamente serve; la memoria in più è interna alla partizione, ma non può essere usata (perchè è assegnata ad un processo).

Paginazione

La **paginazione** è una soluzione ai problemi della frammentazione : si deve abbandonare l'ipotesi di caricare un processo in un'unica area di memoria (rinuncia alla contiguità).

La paginazione è un metodo di gestione della memoria che permette che lo spazio degli indirizzi logici di un processo non sia contiguo.

Il metodo di base per implementare la paginazione consiste nel suddividere la memoria fisica in blocchi di dimensioni uguali e fisse, detti anche **pagine fisiche** o **frame** (le dimensioni sono potenze di 2, tra $2^9 = 512$ e $2^{13} = 8192$ byte), e nel suddividere la memoria logica in blocchi di dimensione pari a quelle dei frame, detti **pagine logiche** o **pagine**.
Bisogna tenere traccia di tutti i frame liberi.

Per eseguire un programma di dimensioni pari ad n pagine, bisogna trovare n frame liberi e caricare le pagine in quei frame.

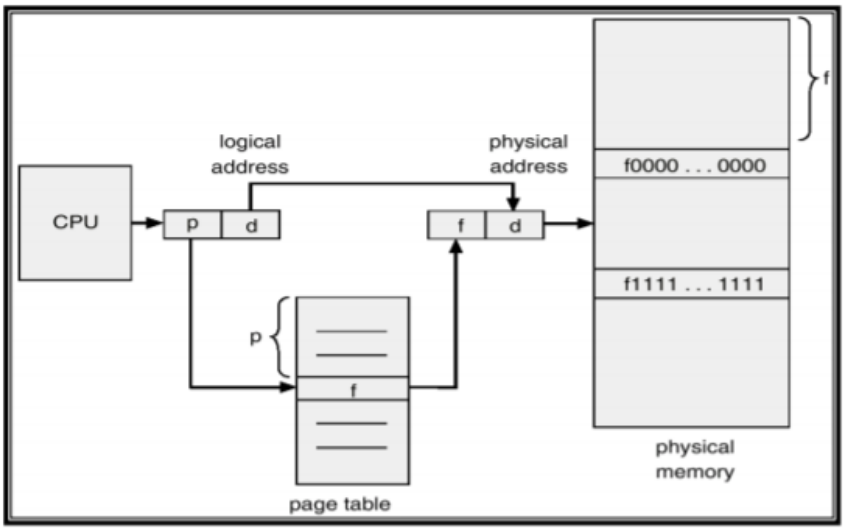
Viene usata una **tabella delle pagine** (di solito una per processo) per tradurre gli indirizzi logici in indirizzi fisici. Contiene tanti elementi quanti sono le pagine logiche del mio processo; ogni elemento della tabella contiene il numero di frame, mentre l'indice è rappesentato dal numero di pagina.

Questo metodo presenta il problema della frammentazione interna.

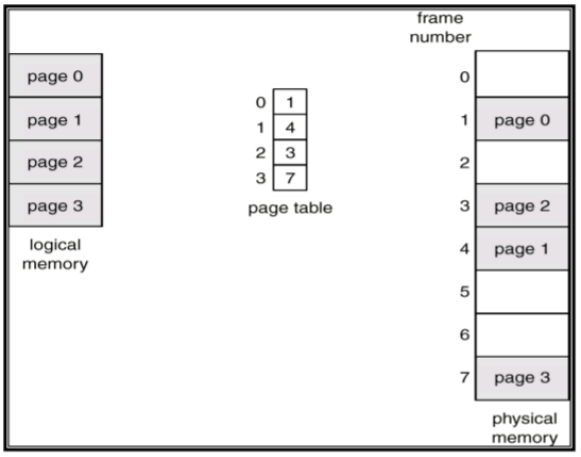
Traduzione degli indirizzi

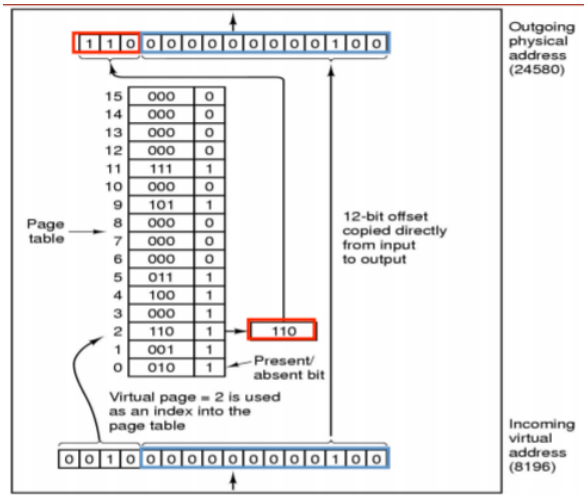
Ogni indirizzo generato dalla CPU (*indirizzo logico*) è diviso in 2 parti: **numero di pagina** (*p*) e **offset** (scostamento, *d*).

Il numero di pagina viene usato come indice della **tabella delle pagine**, contenente l'indirizzo base di ogni pagina nella memoria fisica. Questo indirizzo di base si combina con lo scostamento di pagina per definire l'indirizzo della memoria fisica, che s'invia all'unità di memoria. L'offset è lo scostamento interno alla pagina.



Esempio :





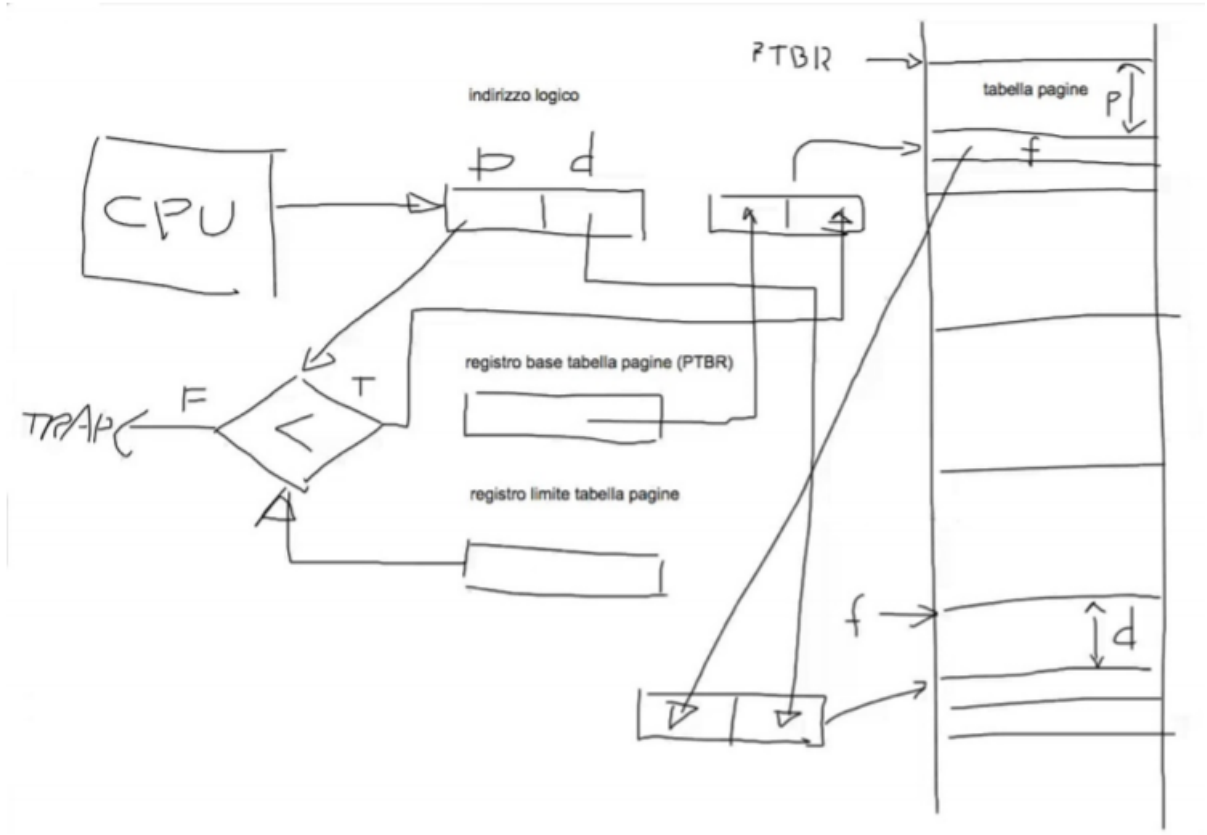
Usiamo come dimensione di pagina le potenze di 2. Se ad esempio utilizzo come dimensione di pagina 2^{12}, le 12 cifre meno significative rappresentano l'offset; le cifre rimanenti, le più signiifcative, le utilizzo come numero di pagina logico-fisico.

Realizzazione della Tabella delle Pagine

Devo accedere in memoria per andare a leggere all'interno della Tabella delle Pagine qual è il frame che mi interessa, dopodiché recuperato il frame ottengo l'indirizzo fisico e a quel punto accedo in memoria una seconda volta per andare a trovare la locazione di memoria.
Raddoppio i tempi di accesso alla memoria (rispetto all'allocazione contigua).

Il **Page-table base register** (*PTBR*) indica l'inizio della Tabella delle Pagine.
Il
Page-table length register (*PTLR*) indica le dimensioni della Tabella delle Pagine.

Con questo schema, per ogni accesso a dati/istruzioni ci sono due accessi in memoria: uno alla Tabella delle Pagine ed uno ai dati/istruzioni veri e propri.



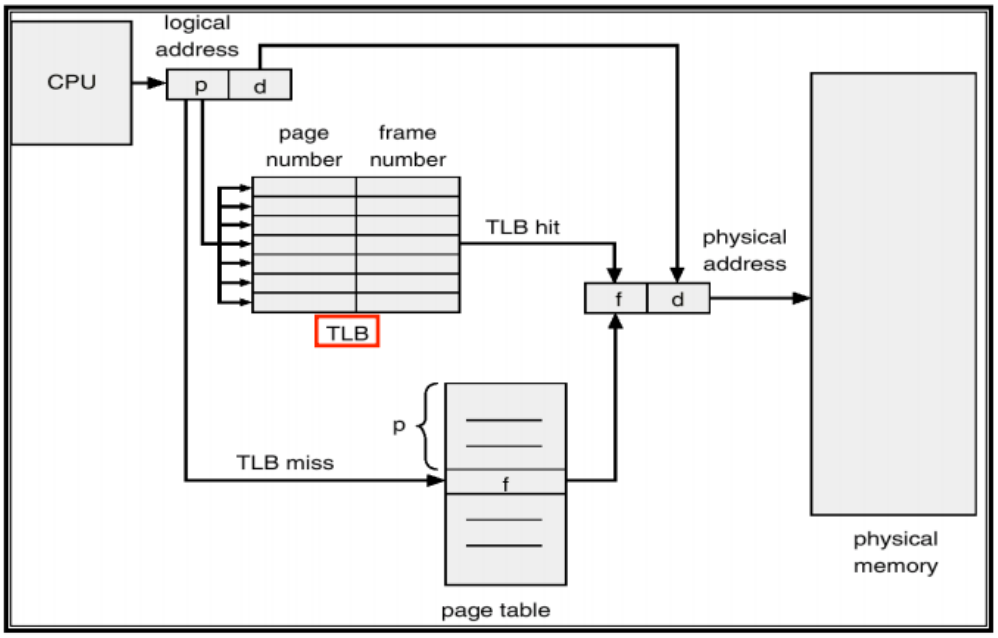
Il problema del duplice accesso in memoria può essere risolto con una speciale memoria associativa usata come cache. Questa memoria viene chiamata **translation look-aside buffer** (**TLB**).

La memoria associativa permette di cercare in parallelo su tutti gli elementi quale sia uguale ad un valore dato: riesce con un unico accesso alla TLB a darci il numero di frame corrispondente alla pagina che ci interessa (se è presente nella TLB; in tal caso si parla di *TLB-hit*); se invece non riesce a trovarla genera un'eccezione *TLB-miss* (l'elemento corrispondente al numero di pagina che sto cercando non è presente dentro la TLB), e devo andare a trovare il numero di frame associato al numero della pagina in questione all'interno della Tabella delle Pagine.

- Se ho un TLB-hit ho un accesso alla TLB e un accesso alla memoria centrale (la situazione migliora rispetto a prima perché l'accesso alla TLB avviene con un tempo inferiore rispetto al tempo di accesso alla memoria centrale, ho un solo accesso alla memoria centrale)
- Se ho un TLB-miss la situazione peggiora, perché ho comunque due accessi in memoria centrale più ho avuto un accesso alla TLB che non mi è servito a nulla.

In entrambi gli scenari le cose vanno peggio rispetto all'allocazione contigua, dove ho un solo accesso in memoria centrale; però con la TLB riesco ad avere tempi di accesso che sono leggermente peggiori

dell'allocazione contigua ma mediamente migliori rispetto al caso in cui non venga usata la TLB.



Tempo effettivo di accesso con TLB

- Tempo di accesso alla TLB → ε (unità di tempo).
- Tempo di accesso in memoria centrale → 1 (unità di tempo).
- *Hit ratio* : probabilità di trovare il numero di frame che mi interessa all'interno della TLB → α

→ **Tempo Effettivo di Accesso** (**EAT**)

$$EAT = (1+\varepsilon)\alpha \ + \ (2+\varepsilon)(1-\alpha) = 2 \ + \ \varepsilon \ - \ \alpha$$

$(1 \ + \ \varepsilon)$: tempo quando si ha successo (TLB-hit).

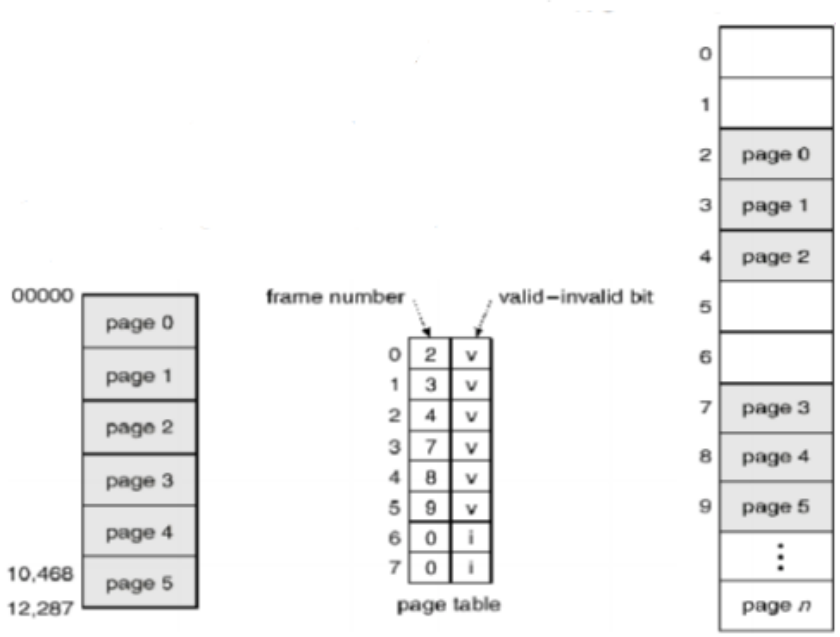
$(2 \ + \ \varepsilon)$: tempo quando si ha fallimento (TLB-miss).

Vogliamo che mediamente il tempo di accesso sia inferiore a 2, perché se fosse maggiore o uguale a 2 vorrebbe dire che aver introdotto la TLB non ha migliorato le cose : le ha lasciate così come sono (= 2) o addirittura le ha peggiorate (> 2). Vogliamo che $2 \ + \ \varepsilon \ - \ \alpha$ sia < 2, cioè ε deve essere minore di α (tanto più $\varepsilon < \alpha$, tanto più sarà il vantaggio).

Protezione della memoria

La protezione della memoria è assicurata dal **bit di protezione** associato ogni frame.
Il bit determina se un processo può leggere e scrivere (
non protetto) oppure soltanto leggere (*protetto*).

Di solito si associa a ciascun elemento della Tabella delle Pagine un ulteriore bit, detto **bit di validità**. Tale bit, impostato a *valido*, indica che la pagina corrispondente è nello spazio degli indirizzi logici del processo, quindi è una pagina valida; impostato a *non valido*, indica che la pagina non è nello spazio d'indirizzi logici del processo.
Il bit di validità consente quindi di riconoscere gli indirizzi illegali e di notificarne la presenza attraverso un segnale di eccezione. Il SO concede o revoca la possibilità d'accesso a una pagina impostando in modo appropriato tale bit.

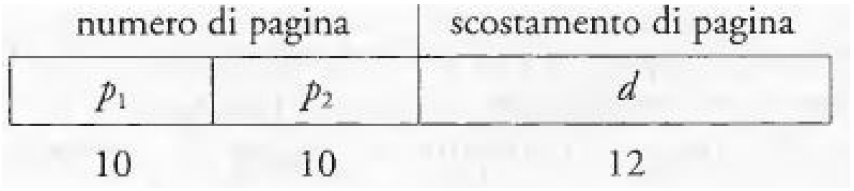


In alternativa si può usare il *PTLR*.

Ogni violazione produce una *trap*.

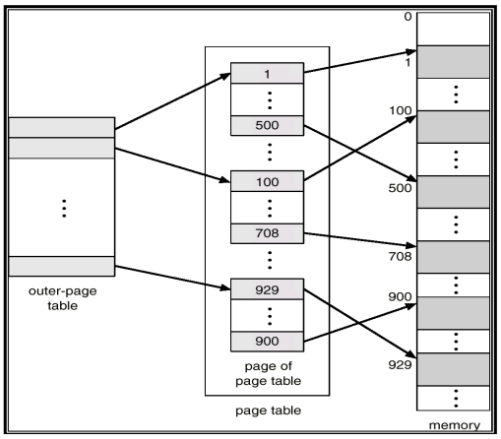
Paginazione a Due Livelli

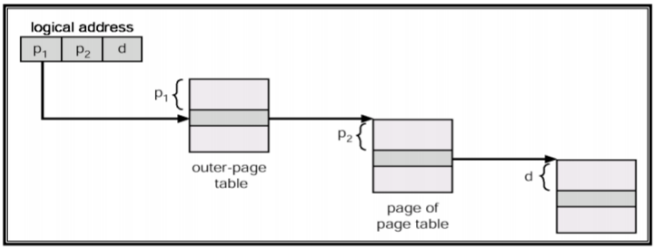
Consiste in una tecnica di paginazione per ridurre le dimensioni delle pagine. È comune in macchine con indirizzi a 32 bit o meno. Consiste nel paginare la stessa tabella delle pagine.
Si consideri un esempio di macchina a 32 bit con dimensione delle pagine di 4 KB (2^{12}). Si suddivide ciascun indirizzo logico in un numero di pagina di 20 bit e in uno scostamento di pagina di 12 bit. Paginando la tabella delle pagine, anche il numero di pagina è a sua volta suddiviso in un numero di pagina di 10 bit e uno scostamento di pagina di 10 bit. Quindi, l'indirizzo logico é:



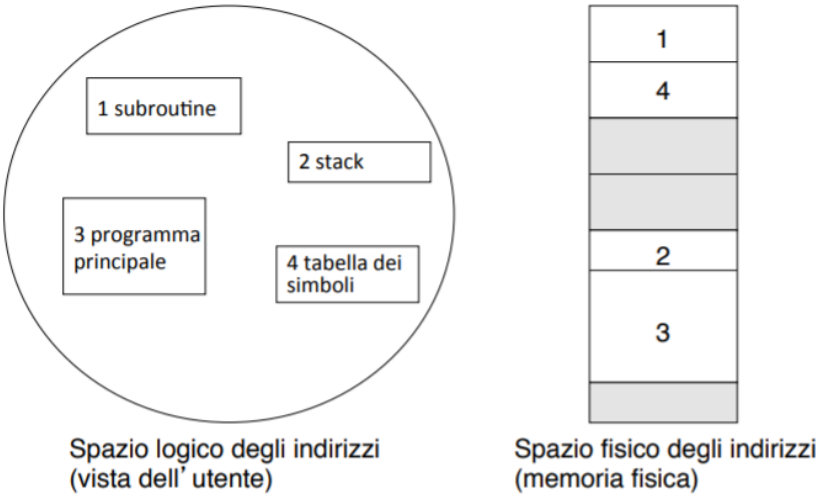
dove p_1 è il *n° di pagina esterno* : accedo alla tabella delle pagine esterne, ovvero la tabella unica che dice dove sono distribuiti tutti i pezzi della tabella delle pagine interne (quindi è un indice della tabella delle pagine di primo livello); p_2 è il *n° di pagina interno*: trovo il numero di frame (quindi è lo scostamento all'interno della pagina indicata dalla tabella esterna delle pagine).

La Tabella delle Pagine, che mi permette di ritrovare il frame dove è allocato il mio processo (che adesso chiameremo tabella delle pagine interna), è stata suddivisa in tanti pezzetti, ognuno dei quali è stato caricato in un frame diverso; per avere la mappa di dove i vari pezzetti della mia tabella delle mie pagine interne è stata caricata, uso un'altra tabella delle pagine, detta tabella delle pagine esterna (unica, sta all'interno di un unico frame).
Uso il numero di pagine esterno per accedere a questa tabella che, grazie al registro base, mi permette di individuare il frame che contiene il pezzo di tabella che mi interessa; una volta che ho individuato questo pezzo di tabella qui, uso il numero di pagine interno per andare a vedere qui dentro dove è il frame che contiene la locazione di memoria che mi interessa.
A questo punto con lo scostamento ottengo la locazione di memoria che mi interessa.





Segmentazione



La **segmentazione** è uno schema di gestione della memoria che consente di gestire questa rappresentazione della memoria dal punto di vista dell'utente.

Un programma è una collezione di segmenti. Ogni segmento è un'unità logica.

Gli indirizzi logici sono costituiti da una coppia <*numero-di-segmento*, *offset*>

Sebbene l'utente possa far riferimento ai dati del programma per mezzo di un indirizzo bidimensionale, la memoria fisica è in ogni caso una sequenza di byte unidimensionale. Per questo motivo occorre tradurre gli indirizzi bidimensionali definiti dall'utente negli indirizzi fisici unidimensionali.

Questa operazione si compie tramite una **Tabella dei segmenti** : permette la traduzione dallo spazio bidimensionale degli indirizzi logici allo spazio unidimensionale degli indirizzi fisici. Ogni suo elemento è una coppia ordinata: la *base* del segmento e il *limite* del segmento. La base contiene l'indirizzo fisico iniziale dell'area di memoria dove risiede il segmento, mentre il limite indica la lunghezza del segmento.

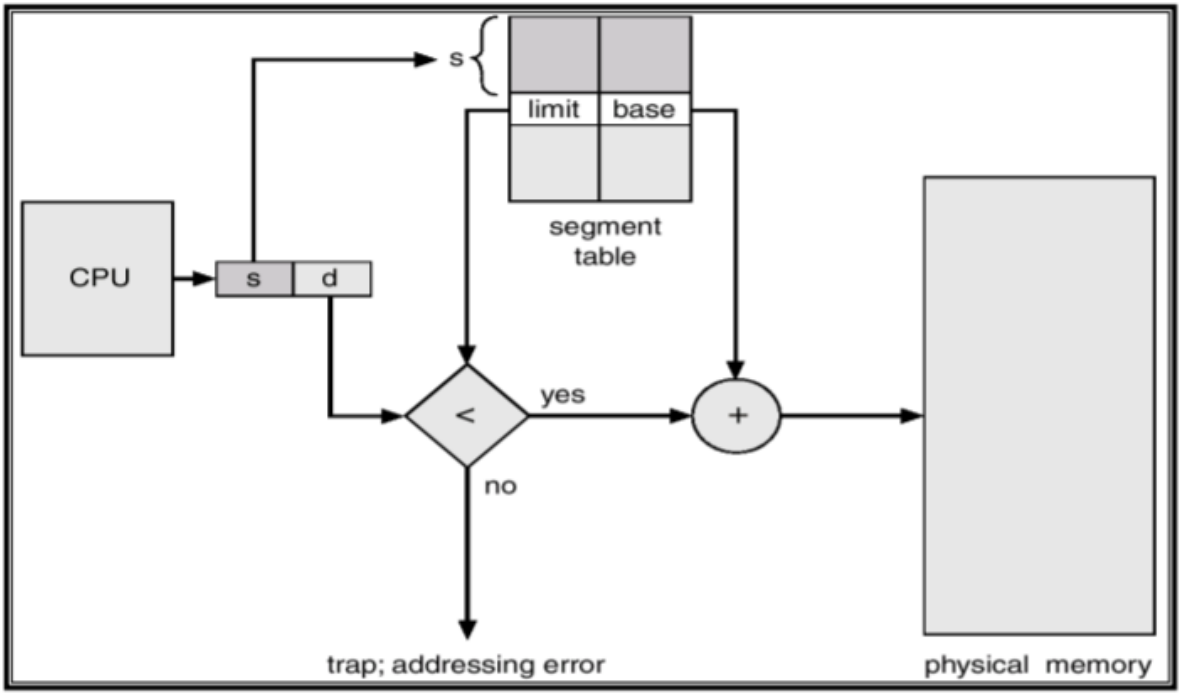
Un indirizzo logico è formato da due parti: un numero di segmento *s* e uno scostamento in tale segmento *d*. Il numero di segmento si usa come indice per la tabella dei segmenti; lo scostamento dell'indirizzo logico deve essere compreso tra 0 e il limite del segmento, altrimenti s'invia un segnale di eccezione al sistema operativo.

Il **Segment-table base register** (*STBR*) punta alla locazione di memoria dove risiede la tabella dei segmenti.
Il
Segment-table length register (*STLR*) indica il numero di segmenti.

Il numero-di-segmento *s* è legale se *s* <*STLR*.

Ad ogni elemento della tabella vengono associati:
-
Bit di validità (quando = 0 → segmento illegale)
- Privilegi di lettura/scrittura/esecuzione
Siccome la protezione avviene a livello di ogni singolo segmento, i segmenti possono essere condivisi tra processi diversi.
Siccome i segmenti sono di lunghezza variabile, si presentano gli stessi problemi affrontati per l'allocazione contigua a partizioni variabili (frammentazione esterna, first/best/worst fit)

Vado a suddividere l'indirizzo logico in due parti: numero di segmento e scostamento. Ogni segmento è identificato da un numero e devo poter identificare dove voglio andare ad operare all'interno del segmento (grazie allo scostamento).
La mia tabella, detta tabella dei segmenti, ha un certo insieme di elementi, ognuno dei quali fa riferimento ad un determinato segmento; questa volta quello che vado a leggere non è il numero di frame (perché non ho più i frame), ma devo sapere a partire da quale indirizzo il mio segmento è stato memorizzato, quindi ho l'indirizzo base del segmento.
All'indirizzo base del segmento devo andare a sommare lo scostamento e successivamente trovo la cella di memoria che mi interessa; ovviamente essendo ogni segmento di dimensione diversa dagli altri mi devo chiedere se lo scostamento è comunque contenuto all'interno del segmento oppure no: per ogni segmento c'è anche il limite (la sua dimensione) e io innanzitutto vado a controllare se lo scostamento è inferiore al limite (quindi è un indirizzo valido) oppure no (indirizzo non valido e quindi ho una trap, segmentation fault).
Avrò un registro base ed un registro limite per ogni processo che mi indicano rispettivamente qual è l'indirizzo dove si trova la tabella dei segmenti e qual è la dimensione della tabella dei segmenti.



I tempi di accesso vengono raddoppiati rispetto all'allocazione contigua perché dobbiamo accedere alla tabella dei segmenti e poi da lì accedere al segmento che ci interessa; usando una TLB possiamo mitigare

questo problema.

Se il numero di segmenti è elevato potrei trovarmi nella necessità di avere tabelle dei segmenti molto grandi; comunque gli stessi segmenti possono essere più piccoli rispetto ad un processo intero, ma allo stesso tempo essere di grandi dimensioni: dunque potrei trovarmi di fronte ad un problema di frammentazione esterna, che sono sì riuscito a ridurre rispetto all'allocazione contigua, ma continua ad essere ancora abbastanza importante. Applichiamo ad ogni singolo segmento i criteri che abbiamo già applicato prima: se invece di trovare spazio per un intero segmento il mio segmento lo spezzetto, potrei migliorare le cose. Spezzettare un segmento in sotto-segmenti non ha molto senso; quello che invece ha senso è spezzettare un segmento in pagine: anche qui si può parlare di paginazione, ma questa volta è applicata ai singoli segmenti, quindi abbiamo una segmentazione con paginazione.

Memoria Virtuale

Introduzione memoria virtuale

Ad ogni processo è associato uno spazio degli indirizzi (tutta la memoria logica di quel processo); la memoria fisica che viene assegnata a quel processo è inferiore allo spazio degli indirizzi (ha una dimensione inferiore alla dimensione della memoria logica); teniamo in memoria centrale solo una parte dello spazio degli indirizzi (memoria logica) del nostro processo.

Quale parte teniamo in memoria centrale ?

Teniamo in memoria centrale la parte di memoria logica con la quale il processo sta lavorando in quel momento; quello che in quel momento non gli serve lo può tenere in memoria di massa.

Avvicendamento: le parti di memoria di cui non ho bisogno le vado a memorizzare nella memoria di massa, le parti di memoria di cui ho bisogno le vado a prendere dalla memoria di massa e le carico in memoria centrale. Si può realizzare tramite la paginazione *su richiesta* (usiamo tecniche di memoria virtuale) *pura* (fino a quando una pagina o un segmento non viene richiesto io non lo carico).
Questa tecnica va sotto il nome di
memoria virtuale, perché virtualmente un processo continua a vedere e a usare l'intero spazio degli indirizzi, però poi di fatto fisicamente solo una parte è realmente tenuta in memoria centrale; le tecniche di allocazione della memoria che sono compatibili con l'idea di memoria virtuale sono sia la paginazione che la segmentazione (in realtà anche l'allocazione contigua).

Vogliamo applicare una tecnica di caching: la memoria centrale fa da cache (tiene una parte dell'informazione), mentre la memoria di massa tiene tutta l'informazione.

La creazione dei processi risulta essere più efficiente: prima, quando chiedevo la creazione di un processo (mi trovavo nello stato di *new*), il processo passava allo stato di *ready* quando avevo caricato in memoria centrale lo spazio degli indirizzi (memoria logica) del processo; adesso abbiamo rinunciato a caricare tutto quanto, carichiamo solo una parte.
Addirittura si può inizialmente non caricare nulla in memoria centrale, facendo passare immediatamente il processo da
new a *ready* senza aver caricato fisicamente niente; nel momento in cui diventa *run* inizierà a caricare le parti che gli servono.

La
memoria virtuale è una tecnica che permette di eseguire processi che possono anche non essere completamente contenuti in memoria. Il vantaggio principale offerto da questa tecnica è quello di permettere che i programmi siano più grandi della memoria fisica; inoltre la memoria virtuale astrae la memoria centrale in un vettore di memorizzazione molto grande e uniforme, separando la memoria logica, com'e vista dall'utente, da quella fisica.

La memoria virtuale si fonda sulla separazione della memoria logica percepita dall'utente dalla memoria fisica.

Questa separazione permette di offrire ai programmatori una memoria virtuale molto ampia, anche se la memoria fisica disponibile è più piccola.
L'espressione
spazio degli indirizzi virtuali si riferisce alla collocazione dei processi in memoria dal punto di vista logico (o virtuale). Da tale punto di vista, un processo inizia in corrispondenza di un certo indirizzo logico e si estende sulla memoria contigua.

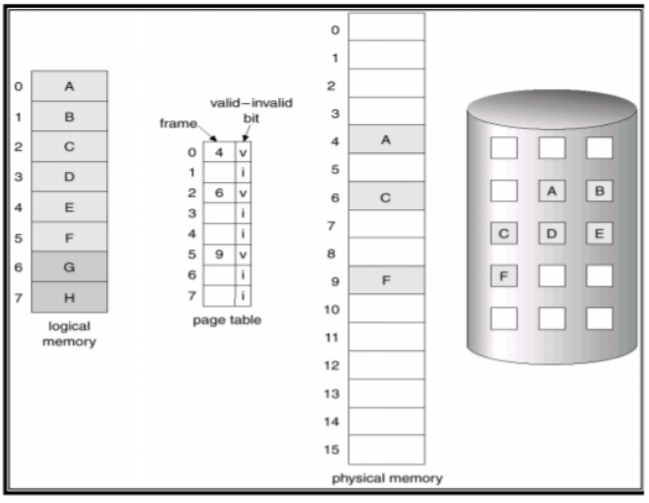
Paginazione su richiesta

Si consideri il caricamento in memoria di un eseguibile residente su disco. Una possibilità è quella di caricare l'intero programma nella memoria fisica al momento dell'esecuzione. Il problema, però, è che all'inizio non è detto che serva avere tutto il programma in memoria: se il programma, per esempio, fornisce all'avvio una lista di opzioni all'utente, è inutile caricare il codice per l'esecuzione di *tutte* le opzioni previste, senza tener conto di quella effettivamente scelta dall'utente. Una strategia alternativa consiste nel caricare le pagine nel momento in cui servono realmente; si tratta di una tecnica detta **paginazione su richiesta**, comunemente adottata dai sistemi con memoria virtuale. Secondo questo schema, le pagine sono caricate in memoria solo quando richieste durante l'esecuzione del programma: ne consegue che le pagine cui non si accede mai non sono mai caricate nella memoria fisica (centrale).
Nell'ambito della paginazione su richiesta, il modulo del sistema operativo che si occupa della sostituzione delle pagine si chiama paginatore (
pager)

Il bit di validità è inizializzato a 0. In fase di traduzione, se bit = 0 → **page fault** : riferimento non valido oppure pagina non in memoria centrale.

Se il bit di validità è 1, quello che è scritto nella prima colonna corrisponde effettivamente al numero del frame che in questo momento sta contenendo la nostra pagina.

Se il bit di validità è 0, questa pagina non è presente in memoria centrale e quello che è scritto nella prima colonna non è un numero di frame.



Page fault: tecnicamente è un'interruzione software, perché durante l'esecuzione di un'istruzione si è verificato un errore e quindi il processo che stava eseguendo quella istruzione viene interrotto.

Quando arriva un page fault viene mandata in esecuzione una routine di gestione di quell'interruzione che va a cercare nella memoria di massa la pagina corrispondente dove è memorizzata la pagina che ci interessa e la carica in memoria centrale, ovvero va a cercare in memoria centrale un frame libero dove andare a caricare quella pagina che ci serve in quel momento.

A questo punto si possono presentare due possibili scenari:

- un frame libero effettivamente esiste, prendo quella pagina nella memoria di massa, la carico in quel frame libero, vado ad aggiornare la mia tabella delle pagine e l'elemento corrispondente, il bit di validità

viene messo a 1, nel campo relativo al numero di frame viene messo il frame dove ho caricato quella pagina e riparto; riassegno immediatamente il controllo al mio processo e non necessariamente viene invocato lo scheduler (faccio ripartire il processo esattamente da dove si era interrotto), prova a cercare la tabella delle pagine, questa volta trova che il bit di validità è a 1, trova il numero di frame, completa la traduzione dell'indirizzo logico in indirizzo fisico e accede alla locazione di memoria di cui aveva richiesto l'accesso.

- un frame libero in questo momento non c'è, cerco una pagina che in questo momento non mi serve, ne faccio lo swap-out, quindi vado a salvare il suo contenuto nella memoria di massa in modo che le eventuali modifiche vengano rese permanenti, dopodiché il frame occupato da questa pagina che ho scelto come pagina vittima lo utilizzo per caricare la pagina che mi interessa (swap-in); poi si riparte come detto prima: andiamo ad aggiornare la tabella delle pagine (prendo la pagina che ho scelto come vittima e metto a 0 il suo bit di validità, mentre al contrario alla pagina a cui ho fatto lo swap-in metto il bit di validità a 1 e come numero di frame metto il frame dove l'ho caricato).

Prestazioni della paginazione su richiesta & Tempo di accesso effettivo (EAT)

La probabilità che si verifichi un Page Fault è un valore p con 0 ≤ p ≤ 1.0 deve se:
– p = 0 allora il page fault non si verifica mai
– p = 1 allora il page fault si verifica sempre

Possiamo definire il **Tempo di Accesso Effettivo (EAT)** come:

$$EAT = (1 - p) * accesso\_in\_memoria \ + \ p * (page\_fault \ + \ swap\_out \ + \ swap\_in \ + \ restart)$$

$accesso\_in\_memoria$: tempo di accesso nel caso in cui non ho page fault.

$(page\_fault \ + \ swap\_out \ + \ swap\_in \ + \ restart)$: tempo di accesso nel caso in cui ho page fault.

Cos'è che rende la probabilità di page fault molto piccola?

Dobbiamo cercare di avere in memoria centrale le pagine che utilizziamo in questo momento. Se in memoria centrale abbiamo tutte le pagine che ci servono in questo momento non avremo page fault; inizieremo ad avere page fault quando le pagine che abbiamo in questo momento non ci servono più e dobbiamo caricare in memoria le pagine che ci servono.

Come si può fare per avere un tempo di avvicendamento contenuto?

Posso agire in 2 modi:
- posso utilizzare una memoria di massa con tempi di accesso particolarmente veloci
-
evitare di fare swap-out (prendo la pagina vittima e la salvo nella memoria di massa) quando non mi serve. **Ha senso fare swap-out solo quando la copia che ho in memoria centrale è stata modificata rispetto all'originale che ho nella memoria di massa**; se la copia che ho in memoria centrale è rimasta identica a quella in memoria di massa, è inutile fare lo swap-out.

Realizzazione della paginazione su richiesta

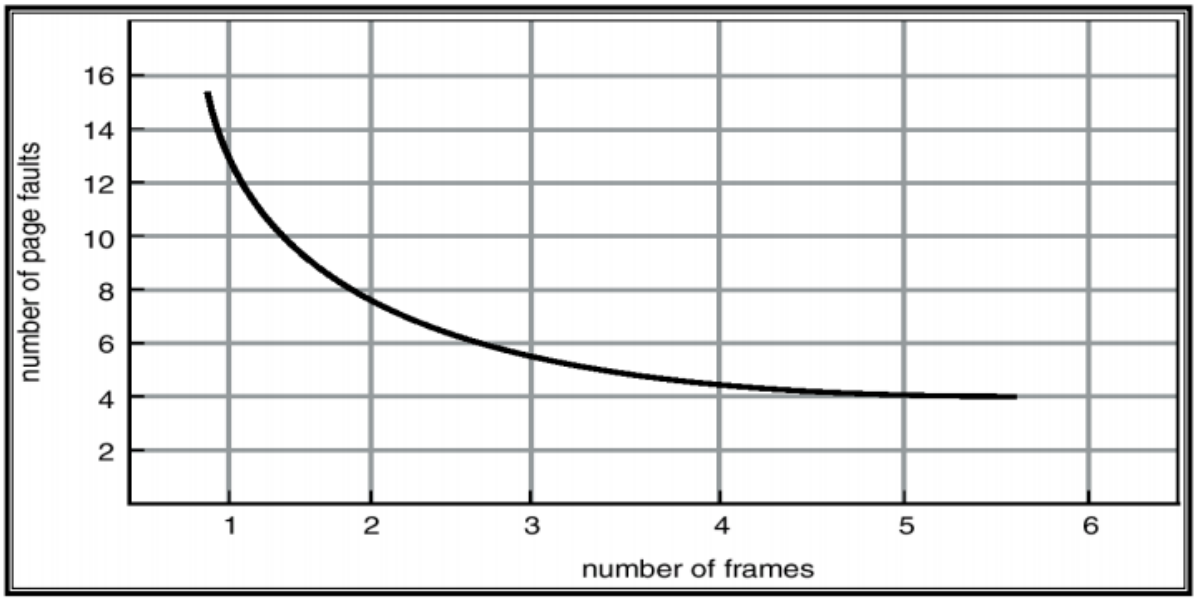
Per ridurre il numero di swap-out richiesti, si può utilizzare un **bit di modifica** (*dirty bit*) andando ad agire sul tempo di avvicendamento:

- Il bit viene messo ad 1 ogni volta che la pagina viene modificata.
- Durante lo swap-out la pagina viene effettivamente copiata su disco solo se il bit di modifica è 1.

Servono inoltre un *algoritmo di assegnazione dei blocchi di memoria* e un *algoritmo di sostituzione delle pagine*.

L'algoritmo di sostituzione delle pagine deve stabilire (quando non ci sono frame liberi) quale tra le pagine attualmente presenti in memoria centrale deve lasciare il frame alla pagina che mi serve.

L'algoritmo deve minimizzare la probabilità di page-fault. Gli algoritmi vengono valutati considerando il loro comportamento (il numero di page fault prodotti) con determinate **successioni dei riferimenti**.

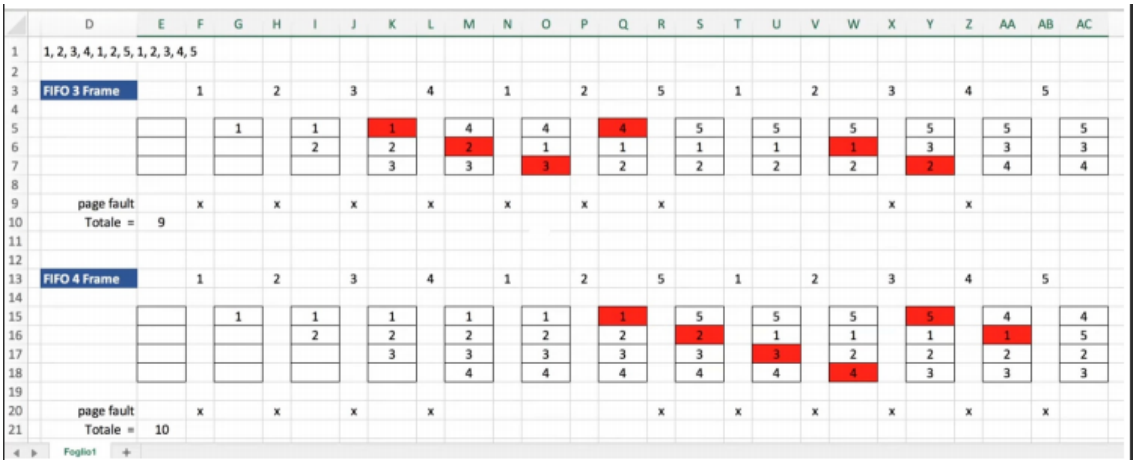


First-In-First-Out (FIFO)

SOSTITUIRE LA PRIMA PAGINA CARICATA IN MEMORIA

Si associa a ogni pagina l'istante di tempo in cui essa è stata portata in memoria, quindi se si deve sostituire una pagina si seleziona tra quella presente in memoria da più tempo. Senza registrare l'istante di tempo, basta strutturare tutte le pagine in memoria in una coda FIFO. In questo caso si sostituisce la pagina che si trova nel primo elemento della coda. Quando si carica una pagina in memoria, la si inserisce nell'ultimo elemento della coda.

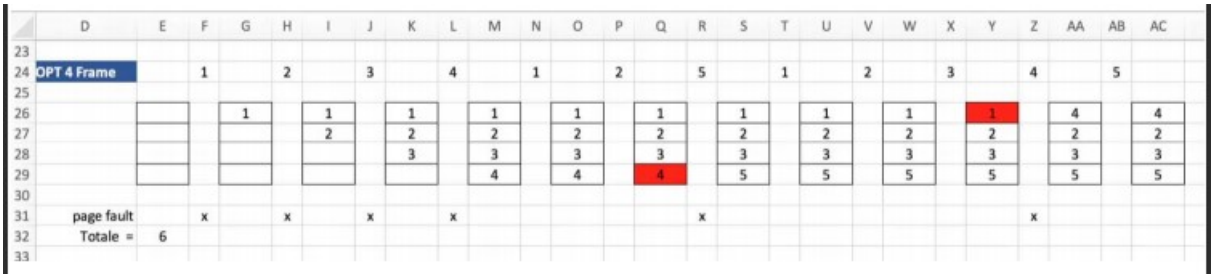
Anomalia di Belady *: non sempre però l'aumento dei frame riduce il numero di page fault.*



Algoritmo Ottimo

SOSTITUIRE LA PAGINA CHE SI UTILIZZERÀ IL PIÙ TARDI POSSIBILE

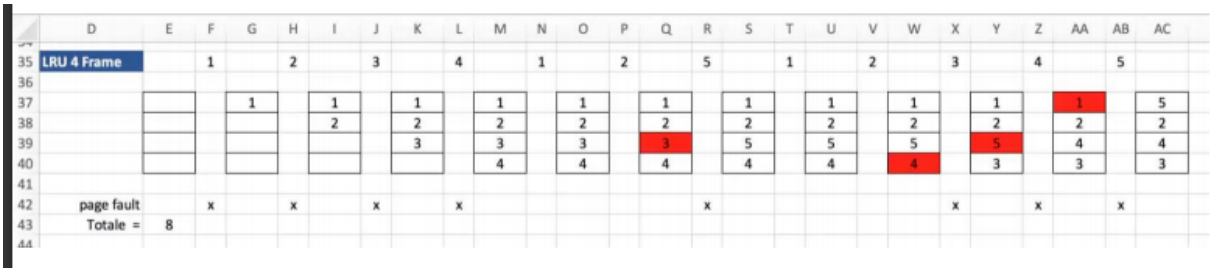
NB. Lazy : Una pagina viene caricata in memoria solo quando ne ho bisogno. L'algoritmo è quello che fra tutti gli algoritmi presenta la minima frequenza di page fault e non presenta mai l'anomalia di Belady. Questo algoritmo esiste ed è stato chiamato "OPT" o "MIN".
Semplicemente si sostituisce la pagina che non si userà per il periodo di tempo più lungo. L'uso di quest'algoritmo di sostituzione delle pagine assicura la frequenza di page fault più bassa possibile per un numero fisso di frame. Sfortunatamente
l'algoritmo ottimale di sostituzione delle pagine è impossibile da realizzare, perché richiede la conoscenza futura della successione dei riferimenti.



Least Recently Used (LRU)

SOSTITUIRE LA PAGINA CHE È STATA USATA MENO DI RECENTE

Si sostituisce la pagina che non è stata usata per il periodo più lungo. Il metodo appena descritto è noto come algoritmo LRU. La sostituzione LRU associa a ogni pagina l'istante in cui è stata usata per l'ultima volta. Quando occorre sostituire una pagina, l'algoritmo LRU sceglie quella che non è stata usata per il periodo più lungo. Questa strategia costituisce l'algoritmo ottimale di sostituzione delle pagine con ricerca all'indietro nel tempo, anziché in avanti.



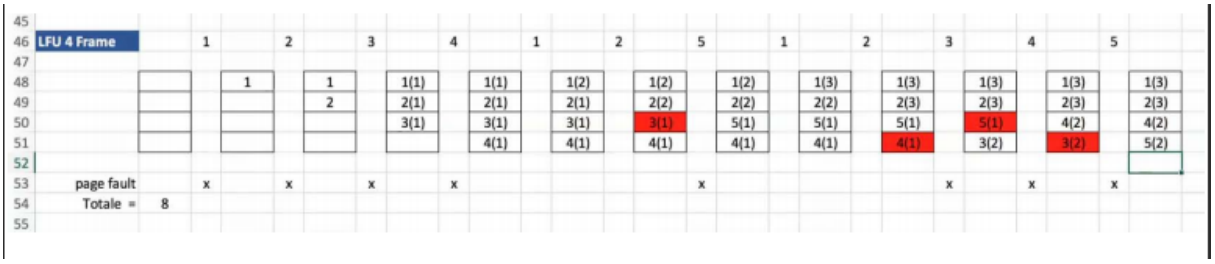
Least Frequently Used (LFU)

SOSTITUIRE LA PAGINA CHE È STATA USATA MENO DI FREQUENTE

Mi baso su quali pagine uso con più frequenza, se c'è una pagina che uso più frequentemente è ipotizzabile che ne abbia bisogno anche in futuro, mentre la pagina usate meno frequentemente è probabile che non ne ho più "bisogno" quindi sarà scelta quest'ultima come pagina vittima.

Frequenza : apice sopra la pagina in questione.
In condizione di parità scelgo la pagina che è entrata prima in memoria centrale.

NB: Le scelte prese portano a dire che è simile al LRU, ma è dato dal caso
NB: LFU, non soffre dell'anomalia di Belady



LFU 4 Frame	1	2	3	4	1	2	5	1	2	3	4	5
	1	1	1(1)	1(1)	1(2)	1(2)	1(2)	1(3)	1(3)	1(3)	1(3)	1(3)
		2	2(1)	2(1)	2(1)	2(2)	2(2)	2(2)	2(3)	2(3)	2(3)	2(3)
			3(1)	3(1)	3(1)	3(1)	5(1)	5(1)	5(1)	5(1)	4(2)	4(2)
				4(1)	4(1)	4(1)	4(1)	4(1)	4(1)	3(2)	3(2)	5(2)
page fault	x	x	x	x			x			x	x	x
Totale = 8												

Realizzazione LRU e LFU :

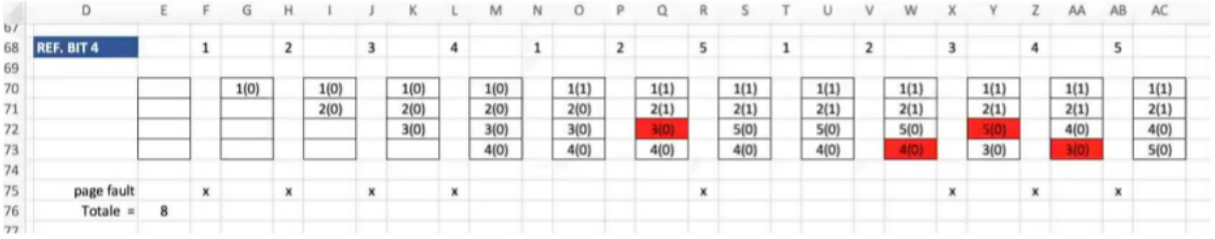
La tabella delle pagine adesso occupa più memoria: prima per ogni elemento doveva riportare un numero di frame e alcuni bit aggiuntivi (bit di validità, bit di modifica, bit di protezione); adesso deve riportare anche un time-stamp (LRU) o un contatore (LFU).

Con i **Contatori** :
- ogni elemento delle pagine ha un contatore;
- ogni volta che c'è un riferimento alla pagina, il valore del clock viene copiato nel contatore (LRU);
- ogni volta che c'è un riferimento alla pagina, il valore del contatore viene incrementato (LFU);
- quando bisogna selezionare una pagina, si sceglie quella che ha il valore più piccolo del contatore.

Con lo **Stack** :
- i numeri delle pagine sono organizzati in una lista doppiamente concatenata;
- ogni volta che c'è un riferimento alla pagina, il suo numero viene inserito/spostato in cima allo stack;
- quando bisogna selezionare una pagina, si sceglie quella presente in fondo allo stack → la sostituzione non richiede nessuna ricerca nella lista.

Bit di riferimento

- Ad ogni pagina è associato un *bit di riferimento*, inizialmente = 0
- ogni volta che c'è un accesso alla pagina il bit viene posto a 1
- la lista delle pagine è gestita FIFO, viene sostituita la pagina che ha il bit di riferimento a 0 (se esiste)
- il bit dice se la pagina è stata usata recentemente, ma non mi dice l'ordine d'uso.

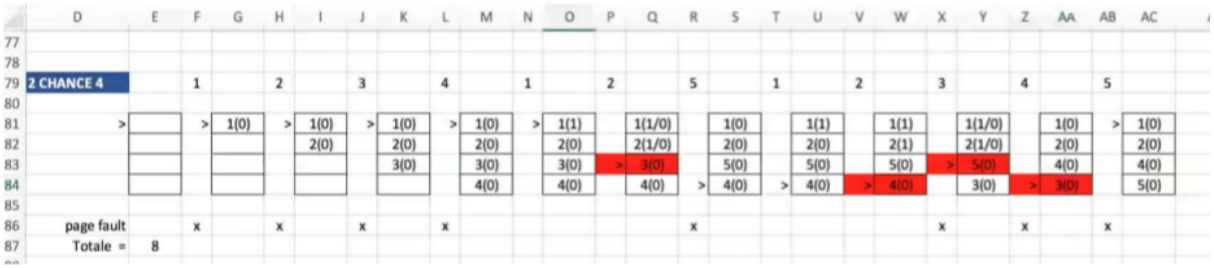


REF. BIT 4	1	2	3	4	1	2	5	1	2	3	4	5
	1(0)	1(0)	1(0)	1(0)	1(1)	1(1)	1(1)	1(1)	1(1)	1(1)	1(1)	1(1)
		2(0)	2(0)	2(0)	2(0)	2(1)	2(1)	2(1)	2(1)	2(1)	2(1)	2(1)
			3(0)	3(0)	3(0)	3(0)	5(0)	5(0)	5(0)	5(0)	4(0)	4(0)
				4(0)	4(0)	4(0)	4(0)	4(0)	4(0)	3(0)	3(0)	5(0)
page fault	x	x	x	x			x			x	x	x
Totale = 8												

Seconda chance (orologio)

- Utilizza il bit di riferimento
- la lista delle pagine è una lista circolare (per questo viene anche detto *algoritmo dell'orologio*; utilizza un puntatore) gestita FIFO, viene selezionata una pagina:
  - se il bit di riferimento = 0 → viene sostituita
  - se il bit di riferimento = 1 → il bit viene messo a 0; la pagina è lasciata in memoria; viene selezionata la pagina successiva a cui vengono applicate le stesse regole.

Se il bit di riferimento è a 0 vuol dire che dall'ultimo avvicendamento quella pagina non è stata più utilizzata.
Se il bit di riferimento è 1 vuol dire che dall'ultimo avvicendamento quella pagina è stata utilizzata almeno una volta.



	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	AA	AB	AC
77																										
78																										
79	2 CHANCE 4		1		2		3		4		1		2		5		1		2		3		4		5	
80																										
81		>		>	1(0)	>	1(0)	>	1(0)	>	1(0)	>	1(1)		1(1/0)		1(0)		1(1)		1(1)		1(1/0)		1(0)	>	1(0)
82							2(0)		2(0)		2(0)		2(0)		2(1/0)		2(0)		2(0)		2(1)		2(1/0)		2(0)		2(0)
83									3(0)		3(0)		3(0)	>	3(0)		5(0)		5(0)		5(0)	>	5(0)		4(0)		4(0)
84											4(0)		4(0)		4(0)	>	4(0)	>	4(0)	>	4(0)		3(0)	>	3(0)		5(0)
85																										
86	page fault		x		x		x		x						x						x		x		x	
87	Totale =	8																								

Assegnazione dei blocchi di memoria

Nella paginazione su richiesta pura, inizialmente i frame sono posti nella lista dei frame liberi; quando il primo processo deve essere caricato in memoria, il processo genera una sequenza di page fault.
Un metodo più efficiente consiste nell'assegnare al processo direttamente in fase di caricamento un certo numero di frame liberi.
Ogni processo ha bisogno di un numero minimo di pagine che devono essere assegnate ad ogni processo. Tale numero dipende dal linguaggio macchina.

Assegnazione uniforme

I frame vengono assegnati in base al numero di processi: sia m il numero di frame liberi ed n il numero di processi → ogni processo riceve $\frac{m}{n}$ frame.

Assegnazione proporzionale

Con l'assegnazione uniforme ci sono processi di piccole dimensioni che ricevono molti frame e processi di grandi dimensioni che ricevono pochi frame.
Con l'assegnazione proporzionale
i frame vengono assegnati in base alle dimensioni del processo:

- s_i : dimensione del processo p_i
- $S = \sum s_i$
- m : numero totale di frame
- $a_i = m \cdot \frac{s_i}{S}$: numero di frame assegnati al processo

Assegnazione per priorità

Sia con l'assegnazione uniforme sia con l'assegnazione proporzionale, un processo con priorità elevata viene trattato nello stesso modo di un processo a bassa priorità. L'assegnazione per priorità si basa su una assegnazione proporzionale alla priorità dei processi invece che alla loro dimensione.
Se un processo genera un page fault:

- si sostituisce uno dei suoi frame (**assegnazione locale**)
- si sostituisce un frame di un processo con priorità minore (**assegnazione globale**)

Questo permette ad un processo con una priorità elevata di aumentare il numero di frame che gli sono stati assegnati.

Applicazione:
Calcolare i reciproci delle priorità di ogni processo e poi li sommo. Il numero di frame per ogni processo è dato da:

$$n^\circ frame_i = \frac{\frac{1}{priorità_i} \cdot n^\circ frame_{memoria}}{somma_{reciproci\ prior.}}$$

Assegnazione globale e locale

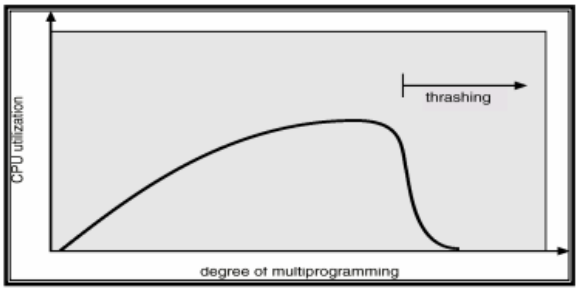
Assegnazione Globale: per la sostituzione viene selezionato un frame tra tutti quelli esistenti, anche se appartiene ad un processo diverso da quello che ha generato il page fault.
In questo modo ci sono processi che vedono aumentare il numero di frame a loro disposizione e processi che vedono tale numero diminuire.

Assegnazione Locale: per la sostituzione viene selezionato un frame del processo che ha generato il page fault.

Paginazione degenere (Thrashing)

Se un processo non ha un numero “sufficiente” di pagine, il tasso di page fault diventa troppo alto e a causa dei continui swap-in e swap-out si verifica un rallentamento generale dell'intero SO che si deve far carico di queste onerose procedure (**thrashing**).
Questo comportamento si verifica in presenza di:

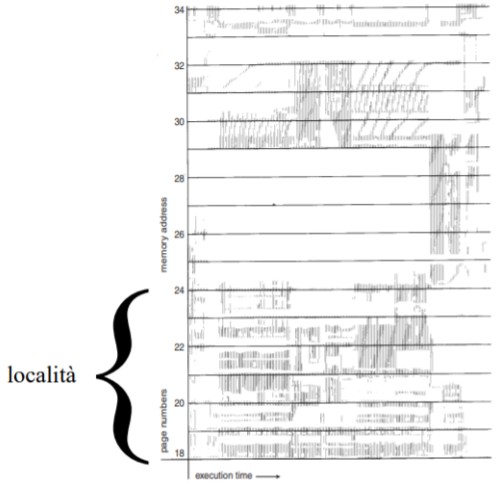
1. Un basso utilizzo della CPU
2. Un aumento di grado di multi-programmazione
3. Si aggiunge un processo in memoria



In particolare ciò accade quando si utilizza un’assegnazione globale e non locale, ovvero quando nella selezione di un frame, si sceglie di tenere conto di chi ha generato un page fault, ma lo si fa in maniera del tutto arbitraria.

Principio di località

Spesso le pagine di un processo durante l’esistenza di quest’ultimo formano una località in cui si trova il processo in quel momento. Il processo potrebbe migrare verso altre località oppure potrebbe scegliere una località ove si sovrappone ad una già esistente.
Il
trashing (o *paginazione degenere*) si verifica se $\sum dim(località) > dim(memoria)$, ovvero se la dimensione di tutte le località è maggiore della memoria stessa.



Si attua così un’assegnazione basata appunto sulla località (**WORKING SET**) : quando un processo inizia ad utilizzare un certo insieme di pagine, per una certa quantità di tempo non trascurabile avrà la tendenza a continuare ad utilizzare sempre quelle pagine; poi ad un certo punto e in maniera abbastanza repentina cambierà l'insieme di

pagine di cui ha bisogno.
Un processo, per un certo intervallo di tempo, si trova in una certa località; poi in maniera improvvisa in poco tempo cambia località.

Assegnazione Basata sul Working-Set

Periodicamente (all'interno di una finestra temporale) si registra quali pagine vengono utilizzate dal processo e si costruisce il working-set; a quel punto si va a vedere qual è la dimensione del working-set e quanti frame sono stati assegnati al processo, sulla base di questo si cerca di capire se sono stati assegnati più o meno frame di quelli che servono al processo in quel momento.

Working-set di un processo : insieme delle pagine referenziate negli ultimi Δ istanti (dove Δ ≡ finestra del working-set).

- WSS_i = dimensione del working-set del processo P_i (varia nel tempo)
  - se Δ troppo piccolo non include l'intera località.
  - se Δ troppo grande può sovrapporre più località.
  - se Δ = ∞ ⇒ include l'intero processo.
- D = Σ WSS_i ≡ totale della domanda di frame (se D > memoria ⇒ Thrashing)
  - Schema basato sul working-set
    - assegnazione pari a dimensione del working set (varia nel tempo)
    - se D < m, allora viene attivato un nuovo processo;
    - se D > m, allora un processo viene sospeso.

Problemi dell'algoritmo del working-set:
1)
Efficacia : bisogna calibrare bene i due parametri utilizzati dall'algoritmo: frequenza di invocazione dell'algoritmo e ampiezza della finestra; se questi due paramentri non sono calibrati bene possiamo non accorgerci di essere entrati in una nuova località, possiamo sovrastimare o sottostimare il numero di frame necessari al nostro processo.
2) Per ogni processo serve anche un ulteriore struttura dati per il working-set e ogni volta che chiedo la traduzione di indirizzo logico in indirizzo fisico devo andare ad aggiornare questa struttura dati (oltre alle varie operazioni di traduzione); periodicamente si osservano dei rallentamenti nell'esecuzione del mio processo

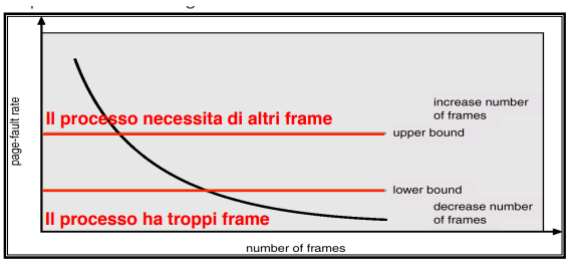
Assegnazione Basata su Page Fault Rate

Schema basato sulla frequenza di page-fault:

- Se la frequenza scende sotto il limite inferiore, al processo viene tolto un frame (perchè ne ha troppi inutilmente).

- Se la frequenza sale sopra il limite superiore al processo viene assegnato un altro frame.

Relazione diretta tra il working set di un processo e il suo page fault rate: il working set cambia nel tempo e anche i picchi e le valli cambiano nel tempo.



Gestione memoria multiprocessori

UMA

Tutti i processori condividono lo stesso spazio di indirizzi con la stessa latenza. Semplice estensione della memoria virtuale per sistemi monoprocessore.

NUMA

Ogni processore ha una sua memoria locale e non accede solo alla sua memoria locale, ma può accedere anche alle altre memorie.

È preferibile accedere alla propria memoria locale, poichè ha un tempo di accesso minore.

Abbiamo visto che nessun algoritmo di quelli visti finora tiene in considerazione questi aspetti.

Se la pagina non si trova nella memoria locale ma nella memoria di un altro processore ?

Tre possibili soluzioni :

- *MIGRAZIONE*: la pagina si trova nella memoria locale di un altro processore, la sposto nella memoria locale del mio processore (se ho dei frame liberi, altrimenti avvicendamento). Si adotta soltanto con l'ipotesi che queste pagine non servano ad ambo i processori (miglioro uno, ma peggioro l'altro)
- *REPLICAZIONE*: la pagina si trova nella memoria locale di un altro processore, creo una copia di quella pagina anche nella mia memoria locale. Si applica nel caso in cui io abbia più thread che lavorano sulle stesse pagine, quindi ognuno lavora più rapidamente. Mi devo però ricordare di mantenere la coerenza di tutte queste copie. Nel momento in cui modifico una di queste copie, devo avvisare gli altri processori di tale modifica.
- Nessuna delle due.

Gestione della memoria Linux

Limitiamoci a Linux a 32 bit di indirizzo. Una macchina con indirizzi a 32 bit mi dà la possibilità di indirizzare 2^{32} locazioni di memoria, 2^{32} byte (sappiamo che 2^{30}= 1 GB, allora 2^{32}= 4 GB).

Ogni task ha a disposizione uno spazio di indirizzamento virtuale di 3 GB, il restante 1 GB è riservato al kernel.

Un task in user mode non vede lo spazio riservato al kernel, riesce a vederlo nel momento in cui passa in kernel mode.

Lo spazio di indirizzamento è costituito da un insieme di regioni omogenee e contigue (ogni segmento viene diviso in pagine; in LInux i segmenti vengono detti *regioni*).
Ogni regione è costituita da un insieme di pagine con stesse proprietà di protezione e di paginazione.

Ogni pagina ha dimensione pari a 4 KB su architetture x86.

Tutta la memoria che serve al kernel la metto dentro lo spazio degli indirizzi di un task; questo vuol dire che dentro la tabella delle pagine di un task di Linux si trovano alcune pagine che in realtà sono pagine del kernel.
Significa che quando il mio task esegue un'interruzione software, deve andare in esecuzione una routine di gestione di quell'interruzione software; io non devo cambiare tabella delle pagine, perché la routine che deve andare in esecuzione si trova all'interno dello spazio degli indirizzi del task che si è interrotto;

dopodiché va in esecuzione lo scheduler, ma anche lo scheduler essendo parte del kernel si trova dentro lo spazio degli indirizzi del mio task.

Spazio di indirizzamento di un task

Linux crea un nuovo spazio di indirizzamento virtuale quando:

- Viene eseguita una **exec** :
  - viene creato uno spazio di indirizzi completamente vuoto.
  - è compito del caricatore popolare lo spazio con regioni di memoria virtuale.

  Non punta a creare una copia esatta del genitore, ma punta a caricare in memoria il codice di un nuovo programma diverso da quello del genitore
- Viene eseguita una **fork** (eseguire una fork → creare un job diverso ⇒ crea una replica del genitore) :

  - viene creata una copia dello spazio di indirizzamento del task genitore.

  - il kernel copia tutti i vm_area_struct del genitore.
  - crea un nuovo insieme di tabelle delle pagine per il figlio.

  - il contenuto delle tabelle delle pagine del genitore è copiato nelle tabelle del figlio.
  - dopo la fork, genitore e figlio condividono le stesse pagine fisiche.
  - la pagina di una regione privata viene duplicata solo in fase di esecuzione, quando uno dei due task chiede di modificarla.
  - così le copie delle pagine si creano solo quando è necessario.

Le pagine del kernel non verranno mai modificate perchè sono protette sia il lettura che in scrittura. (Funzionamento in dual mode)

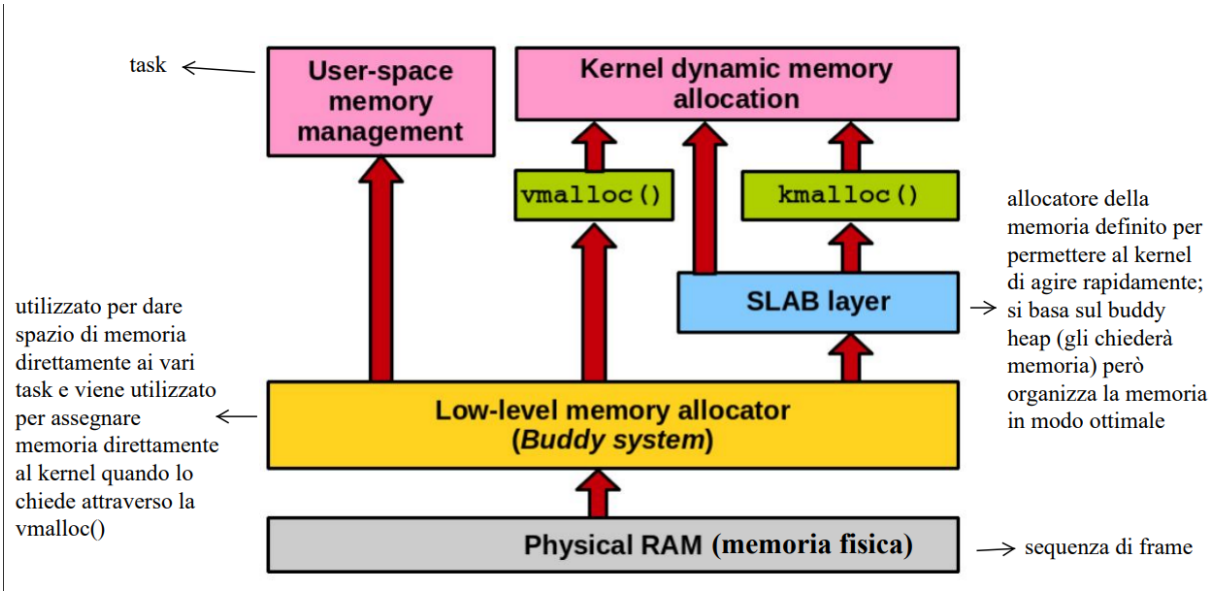
Linux adotta per i programmi eseguibili:
i) il formato a.out
ii) il formato ELF

Inizialmente il programma eseguibile viene mappato nello spazio di indirizzamento virtuale, ma le pagine non vengono caricate (paginazione pura); solo quando il task chiede di accedere ad una data pagina, ci sarà un page fault e di conseguenza la pagina verrà caricata (swapped-in) nella memoria fisica. Il compito del caricatore è quello di instaurare le associazioni iniziali per permettere l'avvvio dell'esecuzione del programma.

Gestore della memoria

Il gestore della memoria di Linux si divide in due tipi :

- **Gestore della memoria fisica** : si occupa dell'assegnazione e del rilascio di pagine fisiche, gruppi di pagine fisiche e piccoli blocchi di memoria (blocchi di memoria di dimensione inferiore alla pagina).
- **Gestore della memoria virtuale** : si occupa della gestione della memoria virtuale, ovvero la memoria indirizzabile dei task in esecuzione.



Anche qui ci servono uno o più algoritmi per allocare la memoria fisica (quindi decidere quali e quanti frame assegnare ad un task) e uno per gestire la memoria virtuale.

Una semplificazione della gestione della memoria in Linux può essere rappresentata in questo modo: la memoria fisica e l'allocatore della memoria fisica al livello più basso (*zoned buddy system*, zoned perché divide la memoria fisica in zone e quando ad un task viene assegnata una certa zona fa in modo che le pagine finiscano sempre in quella zona e questo aiuta la gestione in stile NUMA della memoria). Il buddy system viene utilizzato per allocare la memoria a un task, sopra quest'ultimo è stato costruito un secondo livello di allocatore della memoria fisica, chiamato **SLAB** ed alloca la memoria che serve al kernel.

Uno SLAB è un insieme di pagine fisiche contigue. Una cache è composta da uno o più SLAB. Qui è stato usato il termine cache perché l'obiettivo è quello di avere una raccolta di SLAB all'interno della memoria cache, in modo che il kernel trovi già all'interno della memoria cache una serie di strutture dati che gli serve.

Quindi la cache è un insieme di SLAB. Un oggetto è un'istanza di una struttura dati del kernel. Il kernel utilizza diversi tipi di strutture dati: descrittore di processo, descrittore di task, tabella delle pagine, tabella delle regioni, ecc. Una cache viene riempita con oggetti tutti dello stesso tipo, quindi il kernel avrà diverse cache, una cache diversa per ogni struttura dati in quanto ogni cache raccoglie dati dello stesso tipo. Quando viene creata una cache questa sarà creata da una serie di oggetti liberi che non sono stati ancora istanziati. Saranno slot vuoti che potranno essere utilizzati. Quando il kernel crea una nuova istanza, va in questi slot e occupa uno in base al tipo. A quel punto l'oggetto verrà contrassegnato come usato, come occupato. Ovviamente nel momento in cui poi li libero verranno contrassegnati come tali.
Inizialmente quando creo una cache, ad una cache assegno uno SLAB che conterà un certo numero di oggetti liberi. Nel momento in cui io dovessi riempire tutti gli oggetti, alla mia cache aggiungo un secondo SLAB che sarà costituito da oggetti liberi che posso utilizzare. Notate che la dimensione dello SLAB, cioè il numero di pagine corrispondente, dipende dalla dimensione degli oggetti (il numero di oggetti in uno SLAB dipende dalla dimensione degli oggetti). Quindi gli oggetti all'interno della cache hanno dimensione diversa, in base anche a quale cache si sta utilizzando. Ciò comporta due vantaggi :

- **Minor frammentazione interna**, perché vado a mettere nella stessa cache oggetti dello stesso tipo, tutti con la stessa lunghezza.
- **Velocità** (tranne nelle situazioni in cui ho finito gli oggetti liberi e ho riempito lo SLAB e ne devo allocare uno nuovo). La memoria ce l'ho già allocata e devo solamente riempirla, quindi ho un passaggio in meno e sicuramente ho una maggiore rapidità poiché la velocità di riempimento, di scrittura è veloce.

Memoria Fisica

L'allocazione della memoria può avvenire in due modi:
-
staticamente : i driver riservano aree di memoria contigua durante l'avvio del sistema; la memoria viene allocata e poi non viene più riassegnata.
-
dinamicamente : tramite un allocatore di pagine (quello base oppure per alcuni moduli del kernel dei propri allocatori specializzati).

Allocatore delle pagine di base (**zone buddy allocator**) :
- è responsabile dell'allocazione e del rilascio di pagine fisiche
- è in grado di assegnare su richiesta gruppi di pagine fisicamente contigue (regioni)
- utilizza l'algoritmo
buddy-heap

Allocatore delle pagine del kernel (**slab allocator**) :
- è responsabile dell'allocazione delle pagine del kernel in modo da ottimizzare l'uso della cache
- utilizza l'algoritmo
SLAB

Buddy-heap

La memoria fisica viene vista come un array di frame.

Ogni regione allocabile ha :
- dimensioni pari a
2^nframe contigui (quindi la regione più piccola ha dimensioni pari ad una pagina)

- una compagna adiacente (detta *buddy*) avente pari dimensione

Per gestire le regioni libere, vengono utilizzate liste concatenate distinte, una per ogni dimensione differente.

Ad ogni rilascio di memoria, se la regione rilasciata è adiacente ad un'altra regione libera (della stessa dimensione), le due regioni vengono combinate per formarne una più ampia.

Ad ogni richiesta di memoria (adotta la strategia *best-fit*):
- sia
k il numero di pagine richieste

- si calcola m, ovvero la prima potenza di 2 superiore a k

- se esiste una regione nella lista delle regioni di dimensione m, viene allocata alla richiesta

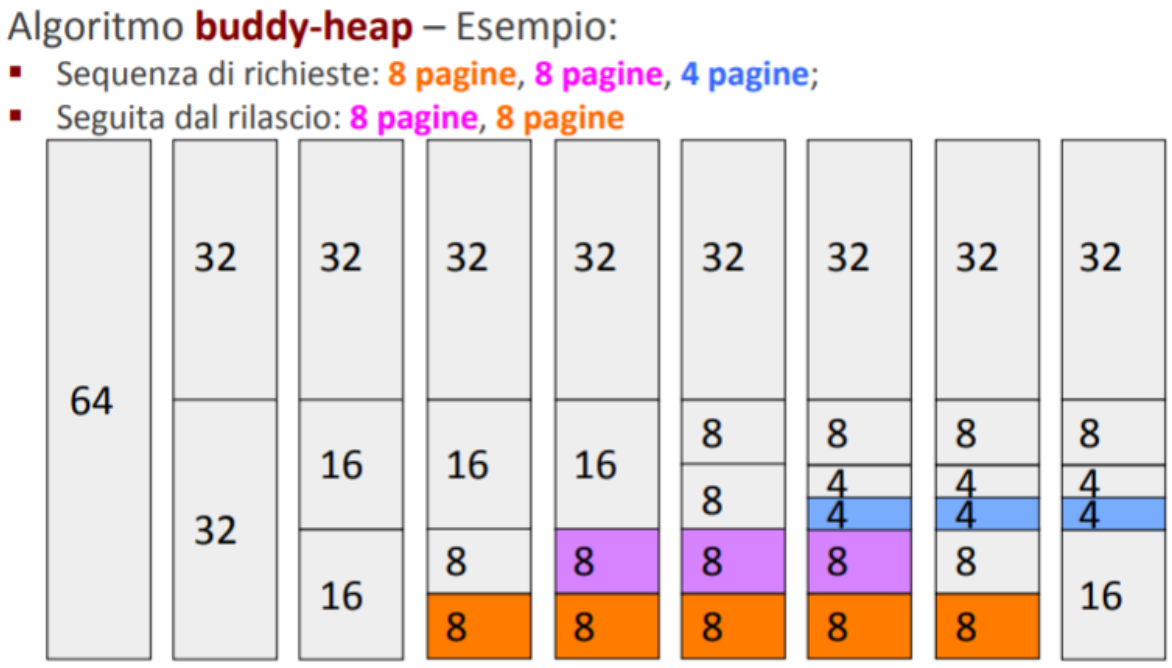
- altrimenti si cerca nelle liste delle regioni di dimensioni immediatamente superiori ($2m, 4m, ...$) fino a trovare una regione libera

- sia $2h * m$ la sua dimensione, essa viene divisa in due buddy per h volte fino ad ottenere una regione della dimensione desiderata. Le metto entrambi nella lista delle dimensioni pari a m; una delle due la assegno al task che aveva fatto richiesta, l'altra rimane come regione libera di dimensione pari a m

Vantaggi : allocazione della memoria molto veloce

Svantaggi :
- frammentazione interna ed esterna

Esistono dei meccanismi per riutilizzare la memoria "sprecata" a causa della frammentazione interna



Slab

Lo slab è organizzato per contenere strutture dati (che conosciamo già a priori) del kernel.

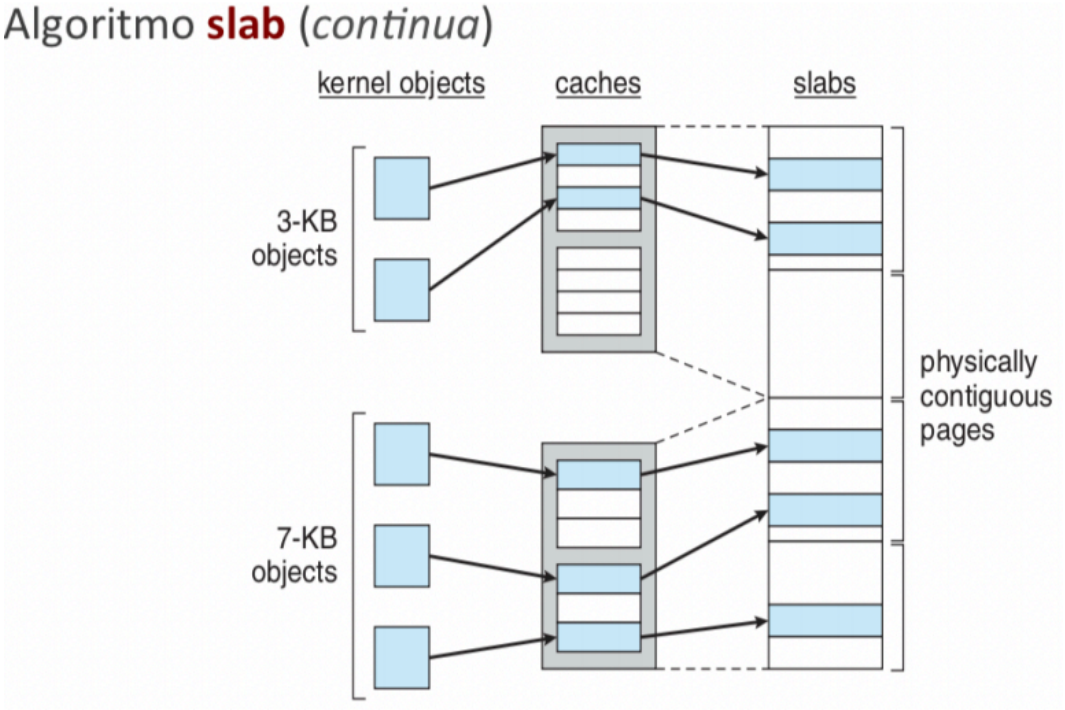
Uno *slab* è formato da uno o più frame contigui.
Ogni struttura del kernel ha una certa dimensione che può non coincidere con le dimensioni dei frame; per questo servono dei frame contigui per facilitare le operazioni.

Una *cache* (residente in memoria centrale) è composta da uno o più slab; ogni cache è dedicata ad un solo tipo di struttura dati del kernel.

Un *oggetto* è una istanza di una struttura dati del kernel.

Ogni cache viene riempita con oggetti (istanze della struttura dati) :
- quando la cache è stata creata, viene riempita di oggetti contrassegnati come liberi
- quando una istanza di struttura viene memorizzata nella cache, l'oggetto che gli è stato assegnato viene contrassegnato come usato
- se uno slab è pieno di oggetti usati, l'oggetto successivo viene assegnato ad un altro slab vuoto
- se non ci sono slab vuoti, viene creato un nuovo slab (viene richiesta l'allocazione di memoria fisica); solo in questo momento viene invocato nuovamente il *buddy system*

I vantaggi includono la rimozione della frammentazione (ogni slab ha una dimensione tale da permettere l'allocazione di un certo numero di istanze di quel tipo di struttura dati, in modo da ridurre al minimo eventuali frammenti) e velocità nell'allocare memoria agli oggetti (quando il kernel chiede la creazione di una nuova istanza, in realtà la trova già lì allocata).



Quindi ogni SLAB può trovarsi in uno di questi 3 stati:

- *Full* quando è stato usato completamente e quindi bisogna passare ad un altro SLAB
- *Empty* quando è tutto libero e posso prendere qualsiasi oggetto lì dentro
- *Partial* quando alcuni oggetti sono liberi e altri no

Lo SLAB allocator segue questi passi:
1. Cerca uno SLAB parzialmente occupato o tutto libero e va a prendere il primo oggetto libero per andare a scriverci l'informazione
2. Se lo SLAB corrente è pieno va a vedere se il successivo è pieno o contiene degli oggetti vuoti
3. Se tutti gli SLAB sono pieni ne crea uno nuovo. Solo in questo caso c'è bisogno di invocare il buddy allocator

.

Linux 2.2 aveva SLAB, ora ha due allocatori: SLOB e SLUB
-SLOB per sistemi con memoria limitata
-SLUB è un SLAB con prestazioni ottimali

Memoria virtuale

Linux usa il concetto di regioni di memoria virtuale, dove ogni regione rappresenta un insieme di pagine logicamente contigue e contenenti informazioni omogenee tra di loro. Ogni regione è descritta da una struttura dati (*vm_area_struct*) che contiene tutte le informazioni tra cui permessi, proprietà, numero della prima e dell'ultima paginazione ecc. Ogni task ha una tabella delle regioni; ogni elemento di questa tabella rimanda alla tabella delle pagine (più altre informazioni che mi servono per gestire quella regione).

Tutte le regioni sono organizzate in un albero binario bilanciato, così da avere una rapida ricerca. Linux adotta un particolare algoritmo di **sostituzione delle pagine** chiamato **KSWAPD** (o lo swapper di Linux).

Algoritmo di sostituzione - KSWAPD

Linux utilizza un approccio diverso. Fino ad ora uno swapper (algoritmo di sostituzione) veniva invocato quando il numero di frame assegnato al nostro processo era esaurito e c'era un page fault, ovvero la necessità di caricare una pagina nuova. In realtà da un certo punto in poi sembrava quasi che il numero di page fault aumentasse perché se l'algoritmo di selezione non sceglieva proprio la pagina giusta, avendo comunque esaurito lo spazio a disposizione, ogni volta che c'era un page fault bisognava fare un avvicendamento. E se la pagina da avvicendare non era scelta in modo giusto poteva capitare più spesso che ci fosse un page fault.

Linux pensò ad un approccio diverso, adotta infatti un approccio tale da avere sempre un certo numero di frame a disposizione in modo che se arriva una nuova richiesta ho comunque un certo numero di frame liberi da poter utilizzare. Quindi anche lui pone una soglia. Quando il numero di frame liberi sull'intera memoria scende al di sotto di un certo livello vuol dire che è arrivato il momento di avvicendare qualche pagina. Questo lo fa nei tempi morti degli altri task, non va ad incidere sui tempi di attesa di un task. Quindi lo swapper viene invocato periodicamente a intervalli di tempo regolari e si accerta della situazione e se il numero di frame è sopra o sotto la soglia. Se il numero di frame è al di sotto della soglia va a vedere quanti frame mancano per raggiungere la soglia e cerca di liberare un numero equivalente di frame.

Linux ha un numero di frame liberi al di sotto del quale non deve mai scendere; per mantenere tale situazione effettua degli swap-out.

Periodicamente controlla il numero di frame liberi a disposizione, e periodicamente decide se c'è bisogno di rimpinguare la scorta di frame liberi (swap-out di qualche pagina in memoria centrale).

Gli swap-out e gli swap-in avvengono in momenti diversi.

Esiste un **demone** (*task sempre in esecuzione*) chiamato ***kswapd*** il cui codice viene eseguito una volta al secondo : esso verifica che ci siano abbastanza pagine libere. In caso contrario, seleziona una pagina occupata e la rende libera.

Le pagine di codice vengono mappate direttamente nei file eseguibili da cui derivano.
Tutte le altre pagine vengono mappate :
- nella partizione di swap (più efficiente, scrittura contigua con accesso diretto al dispositivo)
- in file di swap (meno efficiente, blocchi possono essere fisicamente non contigui con accesso tramite il file system)

L'algoritmo di selezione ha il compito di decidere quale pagina selezionare per effettuare lo swap-out.

Algoritmo di selezione

Una volta appurata la necessità di effettuare degli swap-out di qualche pagina in memoria centrale, bisogna decidere quale pagina va rimossa.

1) Si cerca il processo che ha più pagine in memoria.
2) Si analizzano tutte le sue regioni
(a partire dall'ultima regione analizzata nella ricerca precedente); non vengono considerate le pagine condivise, utilizzate dai canali DMA, locked, assenti dalla memoria.

3) Per selezionare la pagina si adotta l'*algoritmo dell'orologio a doppia scansione*, perché qui è come se avessi due liste : la lista delle pagine attive e la lista delle pagine inattive.

Utilizzo due bit: il bit di *attività* e il bit di *riferimento*.
Se il bit di attività è a 1 la pagina si trova nella lista delle pagine attive; se è a 0 nella lista delle pagine inattive.
Se il bit di riferimento è a 1 la pagina è stata usata di recente; se è a 0 non è stata usata di recente.

Il MMU ogni volta che una pagina viene richiesta, mette il bit di riferimento a 1.

Il kswapd periodicamente controlla :
- se il bit di riferimento = 1, mette il bit di riferimento = 0 e il bit di attività = 0
- se il bit di riferimento = 0 e il bit di attività = 1, mette il bit di attività = 0

Solo le pagine inattive vengono selezionate per essere sostituite.

Lo swap-out viene fatto solo se il *dirty bit* è 1.
Il bit di validità (dirty bit) serve a segnalare se la pagina è stata caricata in memoria centrale (1) oppure no (0).