1 Lezione del 13-11-25

Cominciamo a vedere le tecniche di gestione della memoria viste nella scorsa lezione, in ordine.

1.1 Partizioni fisse

Questo è un approccio a rilocazione *statica*, allocazione *contigua*, spazio virtuale *unico* e caricamento *completo*.

Ciò che facciamo è suddividere lo spazio di indirizzamento in **partizioni fisse**, dedicate ciascuna ad un processo (una specifica partizione andrà dedicata al S/O). Diciamo, quindi, di allocare D_0 Mbyte al sistema operativo, e poi secondo la dimensione della memoria $D_1 \dots D_n$ partizioni di memoria ai processi $1 \dots n$.

Il problema in cui incorreremo sarà la **frammentazione interna**: all'interno della partizione di ogni processo non è detto che il processo utilizzerà sempre la totalità della partizione. Diciamo infatti che il processo i usi N_i Mbyte all'interno della sua partizione: avremo che $D_i - N_i$ sarà spazio sprecato (frammentazione interna) all'interno della partizione.

Tenendo conto di tutti i processi, si avrà che:

$$F_i = \sum_{1}^{n} (D_i - N_i)$$

con F_i appunto la frammentazione interna, quindi la memoria sprecata, complessivamente nel sistema.

Dal punto di vista pratico, il modo più ragionevole per implementare tale soluzione è sfruttare una struttura dati a lista di descrittori di partizione.

- A questo punto si possono effettuare ricerche su tale liste secondo un approccio **first-fit**, cioè caricare il processo nella prima partizione capace di contenere la sua immagine (assumiamo infatti che le partizioni abbiano dimensione variabile). Nel caso in cui tale partizione non esista, chiaramente, siamo in overflow di memoria e il processo non può essere caricato.

  Per questo approccio, va bene che la partizioni in lista siano ordinate per indirizzo (cioè appaiano nella lista nell'ordine in cui vengono disposte in memoria);

- Un approccio più conservativo è quello del **best-fit**: in questo caso si cerca di usare la partizione più piccola possibile che può contenere il processo.

  In questo caso usare l'ordinamento per indirizzo può risultare inefficiente: bisogna controllare per ogni processo ogni partizione. Può essere quindi più ragionevole ordinare le partizioni per dimensione, a partire dalla più piccola: in questo caso il semplice inserimento first-fit sarà automaticamente di tipo best-fit (la prima partizione che contiene è automaticamente la più piccola che contiene).

Avevamo introdotto in 6.1.1 lo scheduling di **medio termine**. Nell'approccio a partizione fisse questo consiste ad adottare più code di processi, una per ogni partizione, e decidere a quale processo dedicare una determinata partizione.

1.2 Partizioni variabili

Evolviamo l'approccio a partizioni fisse. Nelle **partizioni variabili**, infatti, vogliamo permettere alle partizioni non solo di avere memoria diversa, ma di poter variare la loro dimensione nel tempo. Questo singifica che l'approccio è comunque a rilocazione *statica*, allocazione *contigua*, spazio virtuale *unico* e caricamento *completo*.

Diciamo quindi che inizalmente il sistema contiene due partizioni:

- D_0, dedicata come prima al sistema operativo;
- D_1, dedicata a qualsiasi processo.

Quando un processo, diciamo P_1, entra in esecuzione, dividiamo D_1 in due partizioni: una resta D_1, e viene ridimensionata esattamente alla memoria N_1 richiesta dal processo, mentre l'altra viene denominata D_2 e lasciata libera. Questo processo chiaramente si può ripetere per ogni nuovo processo in arrivo, eliminando sostanzialmente il problema della frammentazione interna.

Quello che chiaramente andiamo ad introdurre è pero la **frammentazione esterna**: quando un processo i termina, e libera la sua partizione D_i, ciò che accade è che nello spazio di indirizzamento rimane un *"buco"* di dimensione N_i. Dopo un certo tempo, all'interno del sistema si vano quindi a formare buchi, che impediscono a successivi processi di essere caricati in maniera contigua in memoria.

Potrebbero infatti verificarsi dove la memoria ha complessivamente abbastanza spazio per contenere un nuovo processo, ma la dimensione di nessuno dei singoli buchi e tale da consentirlo nella pratica.

Anche nelle partizioni variabili è utile discutere sull'approccio alla scelta della partizione dove allocare nuovi processi.

- L'approccio **first-fit** resta il più veloce, in quanto non abbiamo bisogno di riordinare la lista quando si alloca memoria (e quindi si dividono le partizioni).
- Nel caso si scelga di usare un approccio **best-fit**, si riduce chiaramente la frammentazione, a costo di dover riordinare la lista in fase di divisione di partizioni.

  L'approccio alternativo è quello di scansionare l'intera lista di partizioni per ogni nuova inserzione, che però avevamo detto parlando delle partizioni fisse è meno efficiente.

Notiamo inoltre che un vantaggio non indifferente che possiamo avere è la possibilità di fare **fusione** delle partizioni al momento di liberazione di spazio dedicato a un processo. Infatti, se la partizione liberata è adiacente ad una partizione libera, si possono combinare le due partizioni in un'unica partizione di dimensione maggiore.

1.2.1 Riassunto sui criteri di allocazione

Possiamo quindi riassumere velocemente le tecniche di allocazione viste nel partizionamento fisso e variabile:

- **First-fit**: è il più veloce, non ha pretese particolari sulle modalità di scansione della lista di partizioni o sul suo ordinamento;
- **Best-fit**: permette di ridurre la frammentazione (*interna* per partizioni fisse ed *esterna* per partizioni multiple), ma è più costoso in fase di creazione di nuovi processi in quanto ha delle pretese sulle modalità di inserzione.

1.3 Segmentazione

L'approccio a **segmentazione** è a rilocazione *dinamica*, allocazione *contigua*, spazio virtuale *segmentato* e caricamento *completo*.

Ciò che vogliamo fare è dividere lo spazio virtuale stesso in più *segmenti*. Ciò significherà che lo spazio di indirizzamento, visto dai processi, diverrà *bidimensionale*: gli indirizzi saranno infatti formati da coppie:

$$x =< \text{segmento}, \text{offset} >$$

dove si specifica il segmento di riferimento e l'*offset* all'interno di tale segmento.

Dal punto di vista implementativo, avremmo bisogno di una **tabella dei segmenti**. Ogni segmento sarà composto da 2 valori:

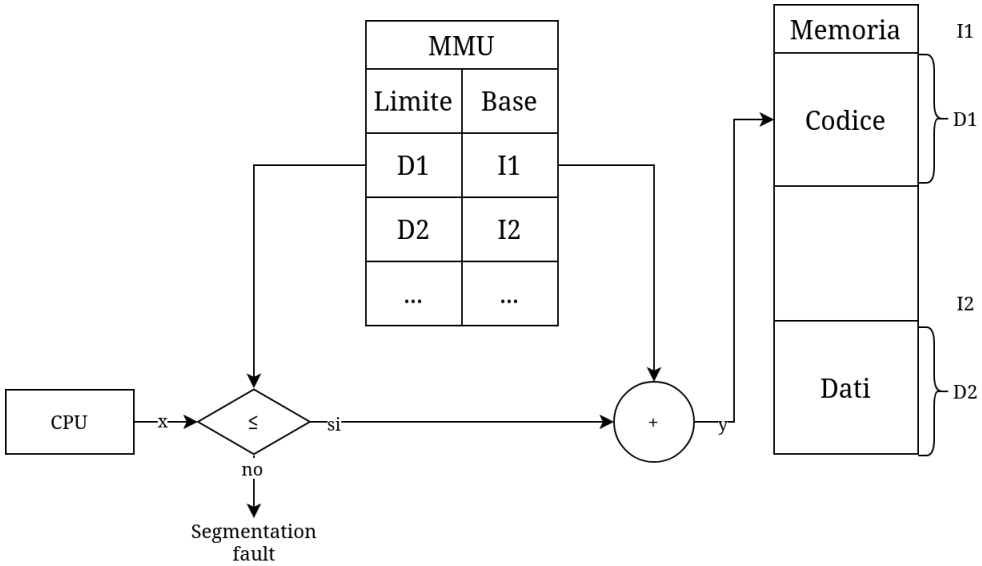
- L'indirizzo al partire dal quale il segmento è allocato in memoria (detto **base**);
- La dimensione del segmento, detta **limite**.

Ogni segmento sarà quindi individuato nella regione di memoria $[\text{base}, \text{base} + \text{limite}]$.

Dotare la macchina di una tale tabella richiede l'introduzione di 2 nuovi registri di controllo, il registro **STBR** (*Segment Table Base Register*), ed il registro **STLR** (*Segment Table Length Register*), contenenti rispettivamente l'indirizzo a partire dal cui si memorizza la tabella dei segmenti e la sua dimensione.

Il supporto hardware alla segmentazione sarà quindi quello di una **MMU** (*Memory Management Unit*), introdotto in 16.4, che ha accesso ai registri STBR e STLR e alla memoria.

Il suo funzionamento può grossomodo essere schematizzato come segue:



dove l'eccezione di *Segmentation fault* viene introdotta appositamente per rilevare accessi al di fuori dello spazio dedicato ai segmenti a tempo di esecuzione. Di base, avremo bisogno di alcuni segmenti di default per ogni programma, fra cui individuiamo:

- Segmento **codice**, che contiene il programma stesso (e magari dati in sola lettura);
- Segmento **dati**, che contiene i dati su cui il programma fa elaborazione;
- Segmento **pila**, che ormai sappiamo è fondamentale all'esecuzione di codice scritto in linguaggio di alto livello.

Questi segmenti corrispondono essenzialmente con i segmenti di immagine di processo che abbiamo introdotto 16.3.1.

Notiamo però che l'uso di un approccio a segmentazione richiede che il processore sia al corrente di quali segmenti riferire in diverse fasi di operazione. Ad esempio:

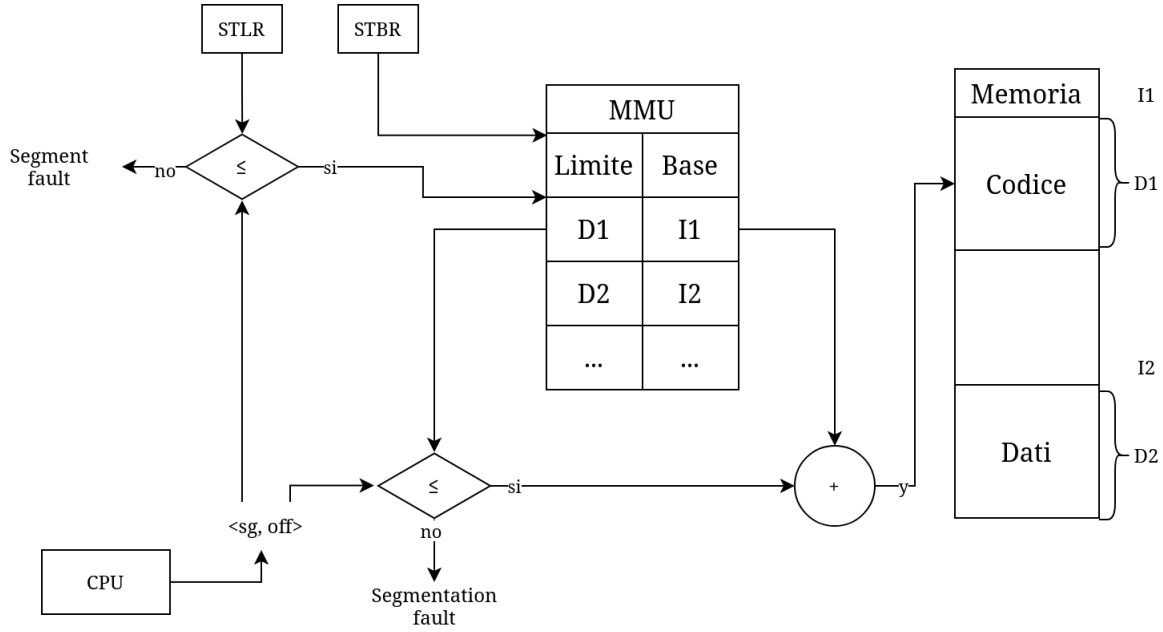
- Nella fase di **fetch**, vogliamo riferirci al segmento *codice* per la lettura di istruzioni;
- Nella fase di **esecuzione**, vogliamo riferirci al segmento *dati* per le operazioni in memoria;
- Nel caso di esecuzione di operazioni in **pila** (ad esempio le `PUSH` e `POP`), vogliamo riferirci al segmento **pila** (per ovvi motivi).

Abbiano quindi che la segmentazione necessita di una modifica del programma: bisogna scrivere programmi che siano a conosecnza di questa funzionalità, e si riferiscano quindi ad indirizzi ben formati (da coppie segmento/offset). L'unica eccezione è chiaramente quella di programmi che richiedono solo un segmento.

Questo perché abbiamo effettivamente fatto una semplificazione non indifferente rispetto ai sistemi reali. Questi infatti:

- Riferiscono la tabella dei segmenti attraverso i registri STLR e STBR: non dobbiamo dimenticarci che la tabella dei segmenti stessa è in memoria fisica;
- Permettono un numero arbitrario di segmenti: si richiede infatti alla CPU di offrire due costanti per indirizzo, cioè come avevamo detto *segmento* e *offset*.

L'MMU aggiornata si può quindi schematizzare come segue:



dove si introduce una nuova eccezione, la *Segment fault*, che non si riferisce ad accessi invalidi all'interno di segmenti ma accesso a segmenti in primo luogo inesistenti.

1.3.1 Descrittori di segmento

Un vantaggio della segmentazione è che possiamo definire nella tabella dei segmenti, oltre ai semplici valori *base* e *limite* di segmento, anche alte informazioni (dette di **controllo**), all'interno del relativo **descrittore di segmento**.

Potremmo infatti prevedere più informazioni in formato tabulare: Dal punto di vi-

Base	Limite	Controllo
Indirizzo del segmento	Dimensione del segmento	Accesso in lettura? scrittura? ecc...

sta implementativo, queste informazioni verrano rappresentate attraverso apposite maschere di bit.

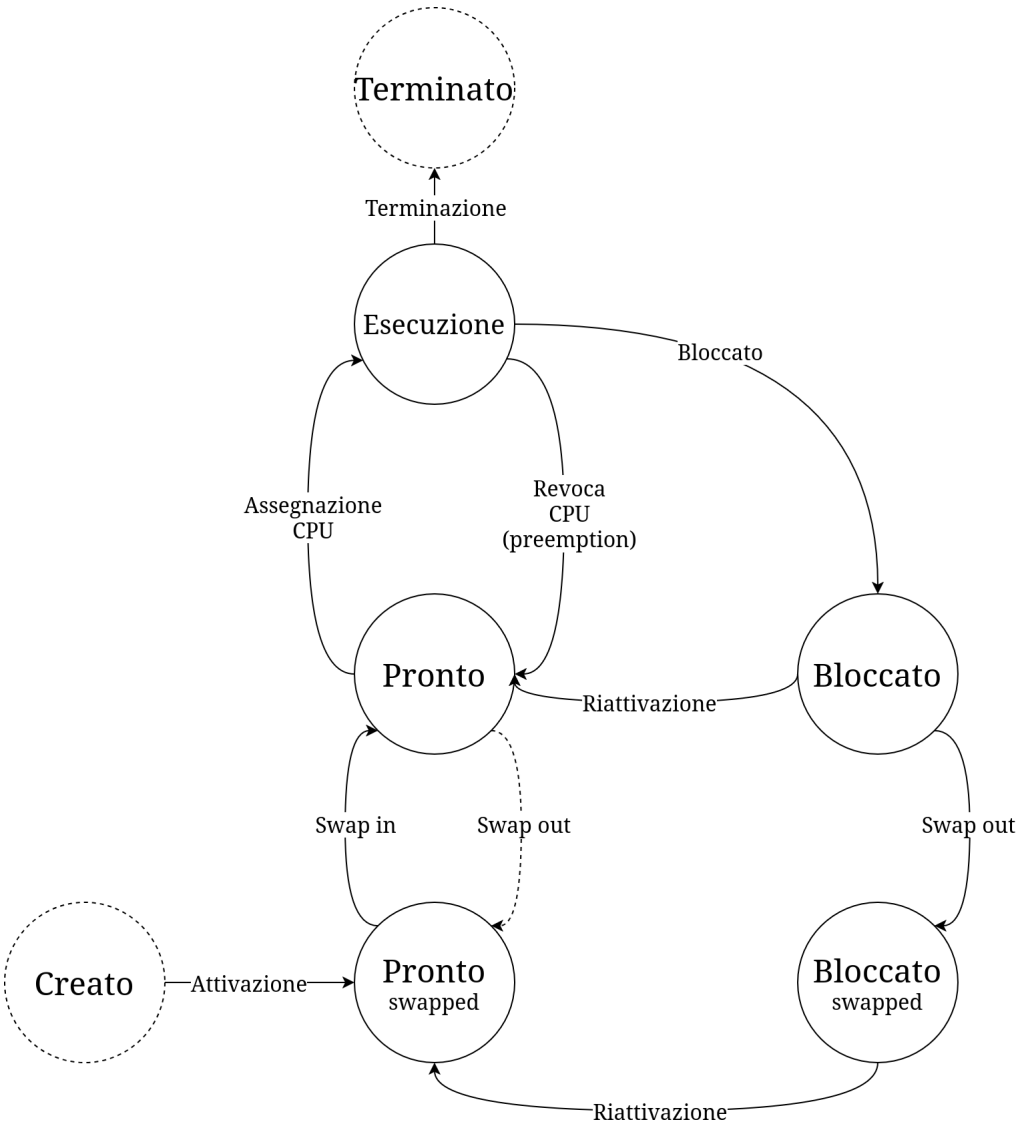
Un esempio tipico è quello di prevedere bit di accesso in lettura (R) o scrittura (W) per segmento. Questo torna utile, prendendo in esame i segmenti visti nella scorsa sezione, come segue:

- Vogliamo che il segmento **codice** sia in sola lettura (quindi R alto e W basso), in quanto come abbiamo introdotto in 5.0.2 il codice può essere condiviso fra processi e non vogliamo che un processo modifichi il codice condiviso con altri processi;
- Vogliamo invece che i segmenti **dati** e **pila** siano in lettura e scrittura (quindi sia R che W alti), per ovvi motivi (il nostro progamma dovrà lavorare su qualcosa, anche scrivendo).

1.3.2 Swap di segmenti

La memoria segmentata supporta un altro meccanismo, che è quello dello **swapping** di segmenti dalla memoria centrale alla memoria secondaria.

Aggiorniamo quindi il grafico introdotto in 5.0.1 relativo allo stato dei processi come segue:



Prevediamo quindi due nuovi stati, *Pronto swapped* e *Bloccato swapped*, su cui si transisce rispettivamente dagli stati pronto e bloccato attraverso le primitive `swap_in()` e `swap_out()`.

Facciamo alcune note su queste transizioni:

- Dallo stato *pronto swapped* vogliamo transire allo stato *pronto*, attraverso la `swap_in()`.

  Potremmo prevedere anche la `swap_out()` da *pronto* a *pronto swapped*, ma questo non è particolarmente utile: quando un processo è pronto, ci aspettiamo di aver fatto del lavoro per renderlo tale. A questo punto, fare swap out invaliderebbe tale lavoro, richiedendo una successiva `swap_in()` prima di poter mettere il processo in esecuzione;

- Dallo stato *bloccato*, invece, è più che ragionevole fare `swap_out()` nello stato *bloccato swapped*. Non si prevede il contrario (prima si diventa pronti e poi si viene swappati in memoria principale).

1.4 Segmentazione su domanda

Veniamo quindi all'evoluzione naturale della segmentazione, la **segmentazione su domanda**. Questa rappresenta un'approccio a rilocazione *dinamica*, allocazione *contigua*, spazio virtuale *segmentato* e caricamento *su domanda*.

Quello che vogliamo effettivamente fare è fornire lo swap out dei segmenti, e fare il successivo swap in solo quando quei segmenti ci vengono effettivamente richiesti.

Per fare ciò ci dotiamo di nuovi bit all'interno del campo di controllo del descrittore di segmento:

- Bit **U**, detto di *uso*, aggiornato quando si usa (legge o scrive) il segmento;
- Bit **M**, detto di *modifica*, aggiornato quando si fa un operazione di scrittura sul segmento;
- Bit **P**, detto di *presenza*, indica se il segmento è effettivamente caricato in memoria o se ne é fatto swap out.

L'operazione che vogliamo effettuare è quindi quello di sfruttare il bit P, in fase di accesso al segmento, per lanciare un eccezione di *segment fault* nel caso tale segmento non sia caricato. In tal caso si cattura l'eccezione e si procede a chiamare la `swap_in()`.

I bit U e M, ricordiamo, forniscono invece delle euristiche utili al S/O per effettuare lo swap out e lo swap in in maniera più efficiente (non copiare segmenti che non sono stati modificati, non deallocare segmenti che vengono usati spesso, ecc...).

1.5 Paginazione

Veniamo quindi alla **paginazione**. Questa rappresenta un'approccio a rilocazione *dinamica*, allocazione *non contigua*, spazio virtuale *unico* e caricamento *completo* (nella versione senza caricamento su domanda).

Abbiamo quindi che lo spazio virtuale viene suddiviso in blocchi di indirizzi virtuali di dimensione fissa (le cosiddette **pagine**). Lo spazio fisico viene suddiviso in blocchi della stessa dimensione delle pagina, detti **frame**.

Ogni pagina corrisponde ad un frame, e pagine consecutive possono essere allocate in frame non necessariamente consecutivi: questo significa che offriamo ai processi spazi di indirizzamento virtuali effettivamente contigui. Vediamo ad esempio come un sistema può avere una divisone delle pagine apparentemente contigua, quando i numeri dei rispettivi frame sono invece non contigui:

Pagina	Sezione	Frame
0	Null	//
1	Sistema	1
2	Text P_1	2
3	Data P_1	3
...	...	//
7	Stack P_1	4

dove nell'esempio si è iniziato a dare qualche informazione semantica su *cosa* contengono le varie pagine.

L'implementazione di un tale sistema richiede di effetuare la traduzione da indirizzo virtuale a corrispondente indirizzo fisico attraverso **tabelle delle pagine** che rappresentano l'associazione fra pagine e frame.

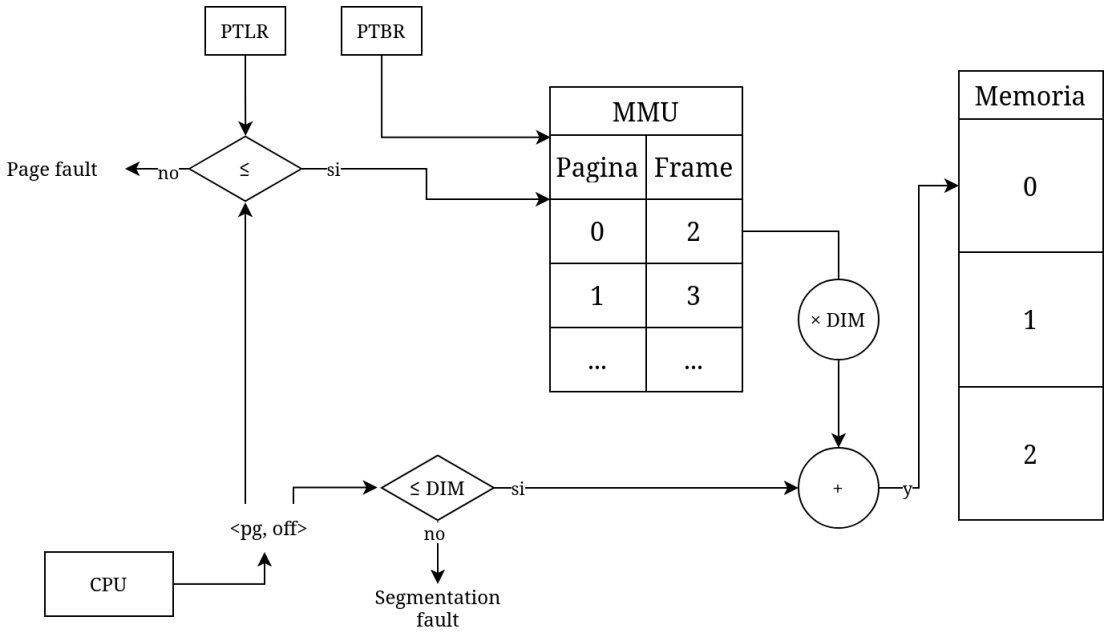
In linea di principio, anche l'MMU in paginazione dovrebbe richiedere dalla CPU una coppia per ogni indirizzo, composta come:

$$x =< \text{pagina}, \text{offset} >$$

dove si specifica la pagina di riferimento e l'*offset* all'interno di tale pagina. La differenza sostanziale dalla segmentazione è però che tutte le pagine sono uguali, per cui le operazioni che dobbiamo fare per passare da indirizzi virtuali a indirizzi fisici sono molto più semplici.

Dettagliamo quindi il funzionamento di tale MMU in paginazione. Come per la segmentazione, necessitiamo dell'introduzione di 2 nuovi registri di controllo, il registro **PTBR** (*Page Table Base Register*), ed il registro **PTLR** (*Page Table Length Register*), contenenti rispettivamente l'indirizzo a partire dal cui si memorizza la tabella delle pagine e la sua dimensione. Viene da sé che la MMU avrà accesso ai registri PTBR e PTLR e alla memoria.

Il suo funzionamento potrà quindi essere schematizzato, senza particolari semplificazioni, come segue:



dove si riportano direttamente 2 eccezioni:

- L'eccezione di *Segmentation fault* viene conservata dalla segmentazione, e usata per rilevare accessi al di fuori dello spazio dedicato alla singola pagina a tempo di esecuzione;
- L'eccezione di *Page fault* viene introdotta per segnalare accessi a pagine inesistenti (non presenti nella tabella delle pagine, vedremo poi con bit P di presenza basso).