

## 7. GESTIONE DELLA MEMORIA

### 1. Introduzione

#### **Indirizzi, dimensioni, spazio di indirizzamento**

Il modello usuale di una memoria è lineare; in tale modello la memoria è costituita da una sequenza di parole o celle numerate da 0 fino al valore massimo. Il numero che identifica ogni cella è detto **indirizzo**.

La dimensione di ogni cella indirizzabile dipende dal tipo di calcolatore; nel seguito supporremo che ogni cella sia costituita da 8 bit, cioè da un **byte**.

E' opportuno distinguere la **dimensione** (effettiva) di una memoria dal suo **spazio di indirizzamento**: lo spazio di indirizzamento è il numero massimo di indirizzi possibili della memoria e dipende dalla lunghezza dell'indirizzo, cioè dal numero di bit che costituiscono l'indirizzo; se  $N$  è il numero di bit che costituiscono l'indirizzo di una memoria allora il suo spazio di indirizzamento è  $2^N$ .

La dimensione della memoria è il numero di byte che la costituiscono effettivamente; ovviamente, dato che tutti i byte devono essere indirizzabili *la dimensione della memoria è sempre minore o uguale al suo spazio di indirizzamento*. Ad esempio, quando si installa nuova memoria su un calcolatore si aumentano le dimensioni della memoria, restando entro i limiti dello spazio di indirizzamento.

Le dimensioni della memoria sono generalmente espresse in Kbyte (Kilobyte), Mbyte (Megabyte), Gbyte (Gigabyte); queste unità corrispondono rispettivamente a  $2^{10}$ ,  $2^{20}$ ,  $2^{30}$  byte. Dato che quasi sempre la misura è espressa come numero di byte, la parola byte viene spesso omessa; noi scriveremo ad esempio 10M per indicare 10 Megabyte.

Esempio: nei Personal Computer Intel l'indirizzo è di 32 bit e quindi lo spazio di indirizzamento è di  $2^{32}$  byte (4 Giga byte); dato che questa dimensione è molto grande, molti Personal Computer possiedono una memoria di dimensioni più ridotte, ad esempio alcune centinaia di Megabyte.

Nella rappresentazione degli indirizzi si usa la notazione esadecimale invece di quella decimale, perché, come vedremo, questa è più comoda per il tipo di

manipolazioni che si devono fare sugli indirizzi; ogni cifra esadecimale corrisponde a una combinazione di 4 bit, secondo la regola di Tabella 1.

Esad.	Binario	Esad.	Binario	Esad.	Binario	Esad.	Binario
0	0000	4	0100	8	1000	C	1100
1	0001	5	0101	9	1001	D	1101
2	0010	6	0110	A	1010	E	1110
3	0011	7	0111	B	1011	F	1111

**Tabella 1 – Corrispondenza tra cifre esadecimali e numeri binari**

---

I numeri esadecimale vengono rappresentati in questo testo secondo la convenzione del linguaggio C, cioè preceduti dal simbolo 0x ; ad esempio, le seguenti equivalenze tra numeri esadecimali di 8 cifre e numeri binari da 32 bit sono corrette:

0x 0000FFFF = 0000 0000 0000 0000 1111 1111 1111 1111

0x ABCDEF00 = 1010 1011 1100 1101 1110 1111 0000 0000

Se un indirizzo è costituito da un numero di bit che non è un multiplo di 4 la sua rappresentazione esadecimale viene fatta utilizzando un numero di cifre capace di rappresentare il multiplo di 4 immediatamente superiore, ma i valori rappresentati dalla prima cifra esadecimale sono limitati; ad esempio, per rappresentare indirizzi da 29 bit useremo 8 cifre esadecimali, che rappresentano 32 bit, con il vincolo che la prima cifra esadecimale sia tale da iniziare con tre bit a zero, quindi sia inferiore a 2; analogamente per indirizzi da 30 bit richiederemo che la prima cifra sia inferiore a 4 e per indirizzi da 31 bit che sia inferiore a 8.

Attenzione: talvolta, per semplicità di rappresentazione degli esempi, quando non vi è dubbio che un numero deve rappresentare un indirizzo o una sua parte, si userà la notazione esadecimale senza precederla con 0x ; quindi si scriverà “indirizzo 3472” invece di “indirizzo 0x3472”.

### **Memoria fisica e memoria virtuale**

Un programma eseguibile è costituito dalle istruzioni che devono essere caricate in memoria per essere eseguite dal processore. Quando il processore esegue il programma, esso legge continuamente gli indirizzi degli operandi e delle istruzioni che

sono incorporati nel programma, quindi esso utilizza come indirizzi gli indirizzi contenuti nel programma eseguibile. Tali indirizzi sono detti **indirizzi virtuali** e il modello della memoria incorporato nel programma eseguibile è detto **memoria virtuale**. Ad esempio, in figura 1 è mostrato il processore che legge l'istruzione contenuta all'indirizzo virtuale 2 e, eseguendola, va ad incrementare il contenuto della cella di indirizzo virtuale 7<sup>1</sup>.

Anche per la memoria virtuale di un programma possiamo parlare di spazio di indirizzamento e di dimensione; lo spazio di indirizzamento è determinato dalla lunghezza degli indirizzi virtuali, cioè dalla lunghezza degli indirizzi presenti nel programma, mentre la **dimensione iniziale** (virtuale) è determinata dal collegatore durante la costruzione del programma eseguibile e scritta nel file che contiene l'eseguibile. La dimensione virtuale del programma è soggetta a variazione durante l'esecuzione del programma stesso, in base alle esigenze di allocazione di memoria per nuovi dati.

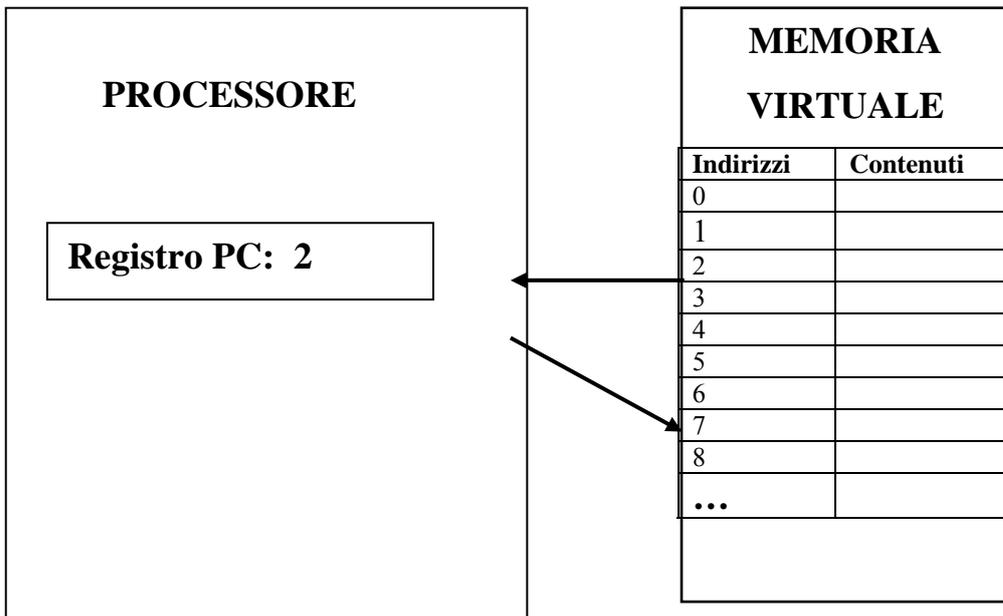
La memoria effettivamente presente sul calcolatore è chiamata **memoria fisica**, e i suoi indirizzi sono detti **indirizzi fisici**. E' evidente che la corretta esecuzione del programma di figura 1 richiederebbe di caricarlo nella memoria fisica a partire dall'indirizzo 0, in modo da far coincidere la memoria virtuale e la memoria fisica durante l'esecuzione.

Tuttavia, come abbiamo visto nel capitolo precedente, ciò non è normalmente possibile e quindi *la memoria virtuale e la memoria fisica non coincidono*. Il meccanismo più elementare di trasformazione tra indirizzi virtuali e fisici che risolve questo problema è la rilocazione dinamica, vista al capitolo precedente. In figura 2 la rilocazione dinamica è richiamata facendo riferimento ai concetti di memoria virtuale e memoria fisica, e quindi come trasformazione di indirizzi virtuali in indirizzi fisici.

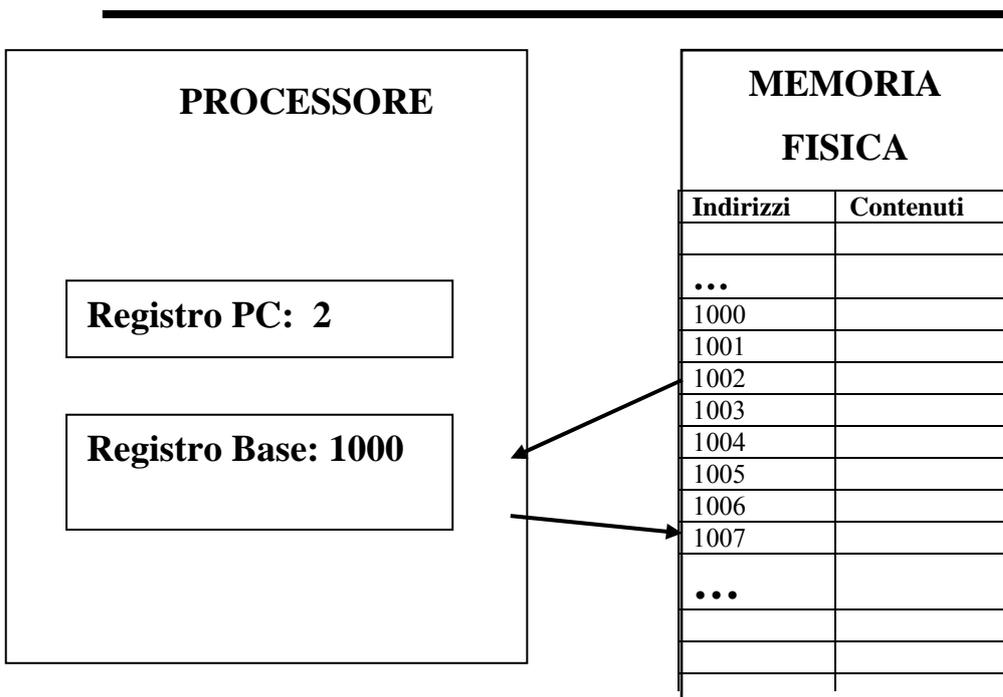
La rilocazione dinamica con un'unica base è uno dei più semplici meccanismi di trasformazione virtuale/fisico, ma, come vedremo nel prossimo paragrafo, i meccanismi adottati nei calcolatori moderni sono più complicati per permettere di ottenere una gestione migliore della memoria.

---

<sup>1</sup> Nei diversi tipi di processori esistono diverse modalità per rappresentare gli indirizzi virtuali all'interno delle istruzioni, dette "Modi di indirizzamento". Tuttavia, il processore interpreta i modi di indirizzamento arrivando a produrre un indirizzo virtuale nel senso adottato in questo capitolo, quindi l'esistenza di diversi modi di indirizzamento nei diversi processori non altera la validità del nostro modello.



*Figura 1 – Modello di esecuzione del programma: il Processore legge un'istruzione all'indirizzo virtuale 2 e modifica un operando all'indirizzo virtuale 7*



*Figura 2 – Trasformazione della memoria virtuale in memoria fisica tramite rilocazione dinamica*

---

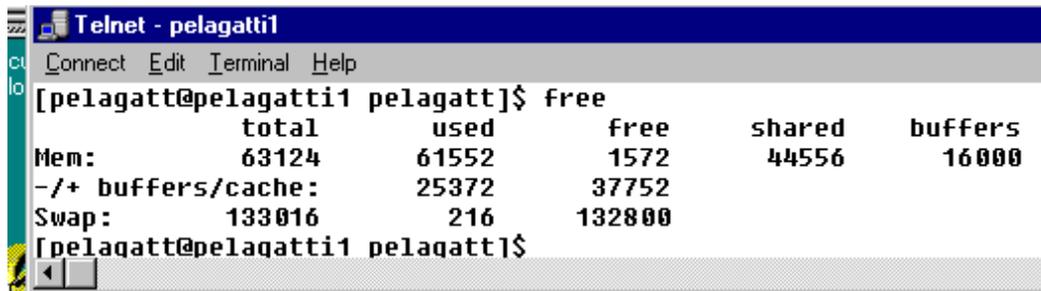
### **Strategie di LINUX per superare la limitazione della memoria fisica disponibile**

Un calcolatore possiede una certa quantità di memoria fisica installata. Questa memoria deve contenere in ogni istante durante il funzionamento il Sistema Operativo e tutti i processi che sono stati creati sulla macchina stessa. Inoltre, una certa quantità di memoria viene allocata per mantenere i **buffer** relativi ai dischi magnetici: i buffer sono zone di memoria fisica utilizzate come aree di transito nelle quali vengono letti i dati provenienti dai dischi tramite adattatori in DMA. Dato che la lettura su disco è molto lenta relativamente alla lettura in memoria, LINUX mantiene in memoria il più a lungo possibile tutti i dati letti dal disco, in modo da poterli eventualmente riutilizzare senza dover rileggere il disco. Tuttavia, mano a mano che i processi utilizzano memoria lo spazio riservato ai buffer viene ridotto.

LINUX permette di gestire più processi di quelli che possono risiedere contemporaneamente in memoria. In generale, la memoria fisica è inizialmente vuota e tutto ciò che vi viene scritto proviene da file su disco oppure dall'esecuzione di programmi. LINUX tenta di mantenere in memoria i dati letti dal disco o scritti dai programmi il più a lungo possibile, tuttavia quando il numero di processi attivi e le loro dimensioni superano un certo livello LINUX è obbligato a liberare delle porzioni di memoria; se libera una porzione di memoria che è stata letta dal disco e non è mai stata modificata (ad esempio una parte di programma) potrà senza difficoltà rileggerla dal disco stesso quando sarà necessario; in caso contrario, cioè se la zona di memoria da liberare è stata scritta o modificata da un programma, dovrà salvarla su disco per poterla recuperare più tardi. La zona del disco in cui vengono scritte le porzioni di memoria salvate è detta **Swap file**.

Una ulteriore tecnica usata da LINUX per risparmiare memoria fisica è la **condivisione** di memoria tra processi (**sharing**): se due processi eseguono lo stesso programma o le stesse librerie di funzioni, allora LINUX tiene in memoria una unica copia di tale codice condiviso permettendo ai diversi processi di leggere dalla stessa memoria fisica. La condivisione di memoria è anche utilizzata talvolta esplicitamente dai programmi per permettere a due processi di scambiarsi informazione (comunicazione tra processi tramite memoria condivisa).

Per permettere agli utilizzatori di conoscere lo stato della memoria fisica in un certo istante LINUX fornisce il comando “free” che produce una risposta come quella esemplificata in tabella 2; in tale tabella tutti i numeri sono espressi in blocchi da 1Kbyte.



```
Telnet - pelagatti1
Connect Edit Terminal Help
[pelagatt@pelagatti1 pelagatt]$ free
              total        used         free       shared    buffers
Mem:           63124         61552          1572        44556     16000
-/+ buffers/cache:    25372         37752
Swap:        133016           216       132800
[pelagatt@pelagatti1 pelagatt]$
```

*Tabella 2 – il comando free*

Il significato delle diverse colonne della prima riga è il seguente:

1. total: è la quantità di memoria disponibile a LINUX (è praticamente tutta la memoria fisica installata)
2. used: è la quantità di memoria utilizzata
3. free: è la quantità di memoria ancora utilizzabile (ovviamente,  $used+free=total$ )
4. shared: è la quantità di memoria condivisa tra processi
5. buffers: è la quantità di memoria utilizzata per i “buffer”,

La seconda riga contiene i valori della prima riga depurati della parte relativa ai buffer; dato che lo spazio allocato ai buffer può essere ridotto a favore dei processi, questo dato indica quanto spazio può essere ulteriormente richiesto per i processi.

Infine, la terza riga indica lo spazio totale, utilizzato e libero sul disco per la funzione di swap.

I meccanismi utilizzati per realizzare questa complessa gestione della memoria sono descritti nei prossimi paragrafi.

## 2 Meccanismi di corrispondenza tra memoria virtuale e memoria fisica: paginazione

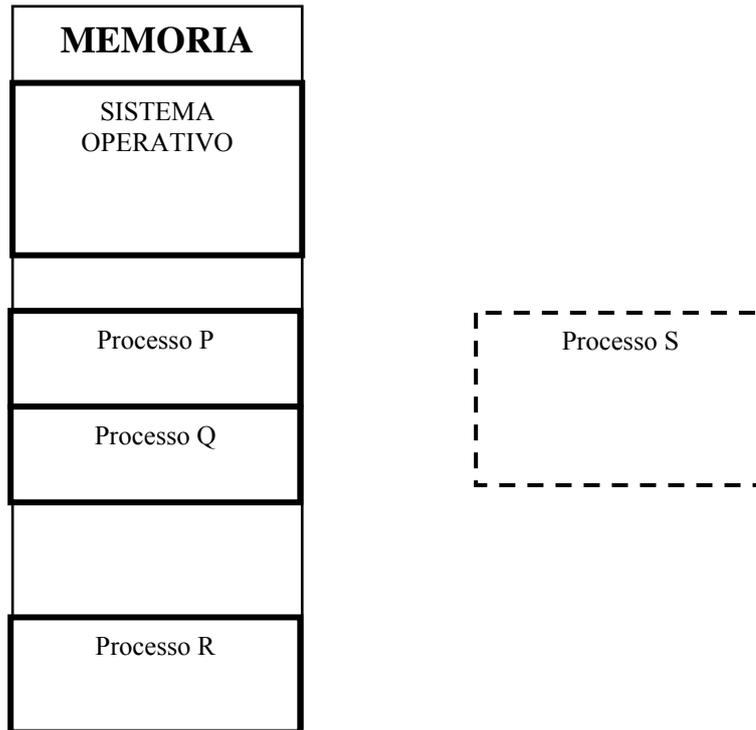
### 2.1 Motivazioni

In generale non è possibile caricare un programma in modo che la memoria virtuale e quella fisica coincidano perfettamente, a causa dei seguenti motivi:

1. Nella memoria fisica devono risiedere simultaneamente sia il sistema operativo che diversi processi.
2. E' conveniente mantenere nella memoria fisica una sola copia di parti di programmi che sono identici in diversi processi (condivisione della memoria). I casi di maggiore utilità della condivisione della memoria verranno analizzati nel prossimo paragrafo.
3. L'allocazione di memoria fisica a diversi processi, la variazione delle dimensioni dei processi, ecc ... tendono a creare buchi nella memoria fisica; questo fenomeno, illustrato in figura 3 è detto **frammentazione della memoria**. Per ridurre la frammentazione è utile allocare i programmi suddividendoli in "pezzi" più piccoli;
4. Le dimensioni della memoria fisica possono essere insufficienti a contenere la memoria virtuale di tutti i processi esistenti, per cui è necessario eseguire i programmi anche se non sono completamente contenuti nella memoria fisica, quindi anche se la memoria fisica disponibile è inferiore alla somma delle dimensioni virtuali dei processi creati.

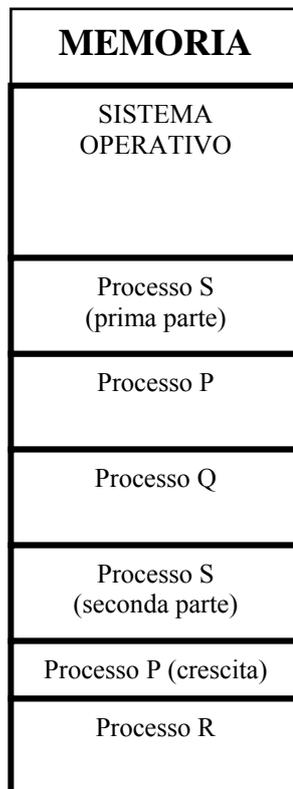
A causa di questi motivi la corrispondenza ottimale tra la memoria fisica e la memoria virtuale dei processi che il SO deve realizzare è simile a quella di figura 4, dove si è ipotizzato che nella memoria fisica sia stato caricato prima il Sistema Operativo, poi il processo P nella sua configurazione iniziale, poi il processo Q nella sua configurazione iniziale, poi è stato allocato nuovo spazio per la crescita del processo P, riempiendo tutta la memoria fisica, e infine il processo Q è cresciuto ma la memoria virtuale aggiuntiva non ha potuto essere caricata nella memoria fisica.

Naturalmente, dato che ogni programma è costruito per funzionare come se il modello fosse quello ideale, con indirizzi fisici identici agli indirizzi virtuale, devono esistere dei meccanismi che permettono ai programmi rappresentati in figura 4 di funzionare come se il modello di corrispondenza fosse quello ideale.



***Figura 3 – Frammentazione della memoria: lo spazio tra il Sistema Operativo e il processo P oppure tra i processi Q ed R sono troppo piccoli per allocarvi un nuovo processo S; inoltre il processo P non può crescere.***

---



***Figura 4 – La suddivisione dei processi P ed S in porzioni non contigue risolve i problemi di frammentazione di figura 3***

---

Tali meccanismi sono ovviamente “trasparenti” al programmatore e al compilatore e collegatore, perché il programma è costruito secondo il modello ideale. In sostanza lo scopo di tali meccanismi è proprio quello di permettere di creare i programmi secondo il comodo modello della memoria virtuale, anche se la situazione della memoria fisica del calcolatore durante l’esecuzione è molto diversa.

### **La concezione di base della paginazione**

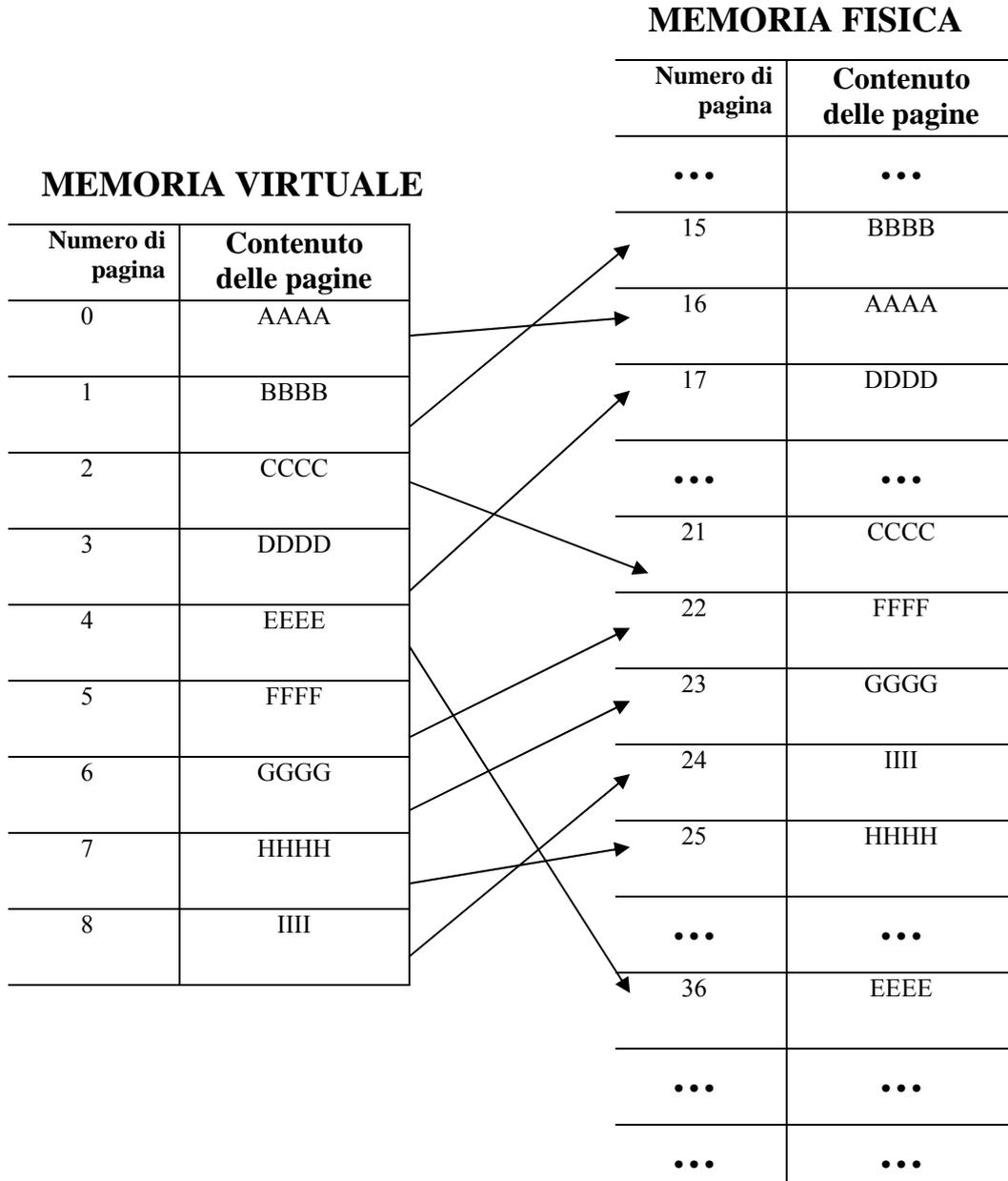
Il meccanismo di paginazione permette di ottenere una gestione migliore della memoria eliminando l'ipotesi che un programma sia memorizzato, durante l'esecuzione, in una zona contigua della memoria fisica. Questo meccanismo permette di eliminare i problemi di frammentazione della memoria, permette di estendere la dimensione della memoria di un processo, anche se non c'è spazio libero immediatamente dopo, purché ci siano altre aree libere sufficienti anche se non contigue (ad esempio, permette di estendere il processo P e di allocare il processo S nel modo illustrato in figura 4).

L'idea base della paginazione è molto semplice e consiste nelle seguenti regole (figura 5):

- La memoria virtuale del programma viene suddivisa in porzioni di lunghezza fissa dette pagine virtuali aventi una lunghezza che è una potenza di 2 (in figura le pagine sono lunghe 4K).
- La memoria fisica viene anch'essa suddivisa in pagine fisiche della stessa dimensione delle pagine virtuali.
- Le pagine virtuali di un programma da eseguire vengono caricate in altrettante pagine fisiche, prese arbitrariamente e non necessariamente contigue.
- Un indirizzo virtuale del programma viene considerato costituito da un **numero di pagina virtuale NPV** e da uno **spiazzamento (offset)** all'interno della pagina; l'indirizzo virtuale viene trasformato nel corrispondente indirizzo fisico, che a sua volta può essere considerato costituito da un **numero di pagina fisica NPF** e da uno spiazzamento, sostituendo al valore di NPV il valore della corrispondente pagina fisica NPF e lasciando inalterato lo spiazzamento<sup>2</sup>.

---

<sup>2</sup> Nel caso in cui la porzione spiazzamento dell'indirizzo sia un multiplo di 4 bit (come è il caso con pagine di 4K), l'indirizzo esadecimale si suddivide in maniera particolarmente facile nelle due porzioni di numero pagina e spiazzamento. Questa facilità riguarda solo noi che utilizziamo la notazione



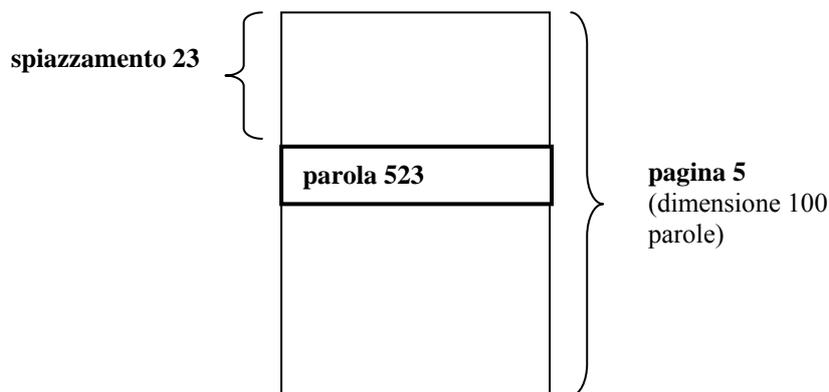
*Figura 5 – Paginazione: corrispondenza tra pagine virtuali e pagine fisiche*

esadecimale; per la macchina la condizione importante è che le pagine abbiano una dimensione che è una potenza di 2.

E' fondamentale, per capire il meccanismo di paginazione, tenere presente che *il fatto di avere stabilito che le pagine abbiano una lunghezza che e' potenza di 2 permette di considerare un indirizzo come composto dal concatenamento di un numero di pagina e di uno spiazzamento.*

Questo aspetto è facilmente comprensibile facendo riferimento al sistema decimale invece di quello binario; consideriamo quindi il seguente esempio basato su indirizzi decimali:

supponiamo di avere una memoria di 1000 indirizzi decimali da 0 a 999 e di suddividerla in 10 pagine di lunghezza 100 – allora ogni indirizzo può essere considerato costituito da un numero di pagina di una cifra (da 0 a 9) e da uno spiazzamento di due cifre (da 00 a 99); ad esempio, l'indirizzo 523 può essere considerato come il concatenamento del numero di pagina 5 e dello spiazzamento 23 (figura 6).



**Figura 6 – La scomposizione di un indirizzo decimale in numero di pagina e spiazzamento**

Per realizzare la paginazione e' necessario che il sistema esegua la trasformazione degli indirizzi virtuali in indirizzi fisici, cioè la sostituzione dei numeri di pagina virtuali con i numeri di pagina fisici, dato che lo spiazzamento rimane inalterato.

**MEMORIA  
VIRTUALE**

Numero di pagina	Contenuto delle pagine
0	AAAA
1	BBBB
2	CCCC
3	DDDD
4	EEEE
5	FFFF
6	GGGG
7	HHHH
8	IIII

**TABELLA  
DELLE  
PAGINE**

NPV	NPF
0	16
1	15
2	21
3	17
4	36
5	22
6	23
7	25
8	24

**MEMORIA FISICA**

Numero di pagina	Contenuto delle pagine
...	...
15	BBBB
16	AAAA
17	DDDD
...	...
21	CCCC
22	FFFF
23	GGGG
24	IIII
25	HHHH
...	...
36	EEEE
...	...
...	...

*Figura 7 – La Tabella delle pagine per rappresentare la corrispondenza tra pagine virtuali e pagine fisiche (con riferimento all’esempio di figura 5)*

---

Ciò è ottenuto creando una **Tabella delle Pagine** che contiene tante righe quante sono le pagine virtuali di un programma, ponendo in tali righe i numeri delle

pagine fisiche corrispondenti e sostituendo, prima di accedere alla memoria, il numero di pagina virtuale con il corrispondente numero di pagina fisico.

In figura 7 e' mostrato questo meccanismo con riferimento all'esempio di figura 5.

### Esempio – processore PROC1

Consideriamo un processore PROC1 la cui gestione della memoria sia caratterizzata dai seguenti dati:

- -spazio di indirizzamento virtuale: 4G (32 bit di indirizzo virtuale)
- -spazio di indirizzamento fisico: 1G (30 bit di indirizzo fisico)
- -dimensione pagine: 4K (12 bit di spiazzamento)
- -numero massimo di pagine virtuali:  $2^{20}$
- -numero massimo di pagine fisiche:  $2^{18}$
- -struttura di un indirizzo virtuale:
  - 20 bit per NPV
  - 12 bit per spiazzamento
- -struttura di un indirizzo fisico:
  - 18 bit per NPF
  - 12 bit per lo spiazzamento

Inoltre, quando il processore PROC1 funziona in modalità privilegiata gli indirizzi non vengono modificati, quindi il meccanismo di trasformazione da NPV a NPF è applicato solamente ai programmi funzionanti in modalità non- privilegiata e il sistema operativo non viene rilocato.

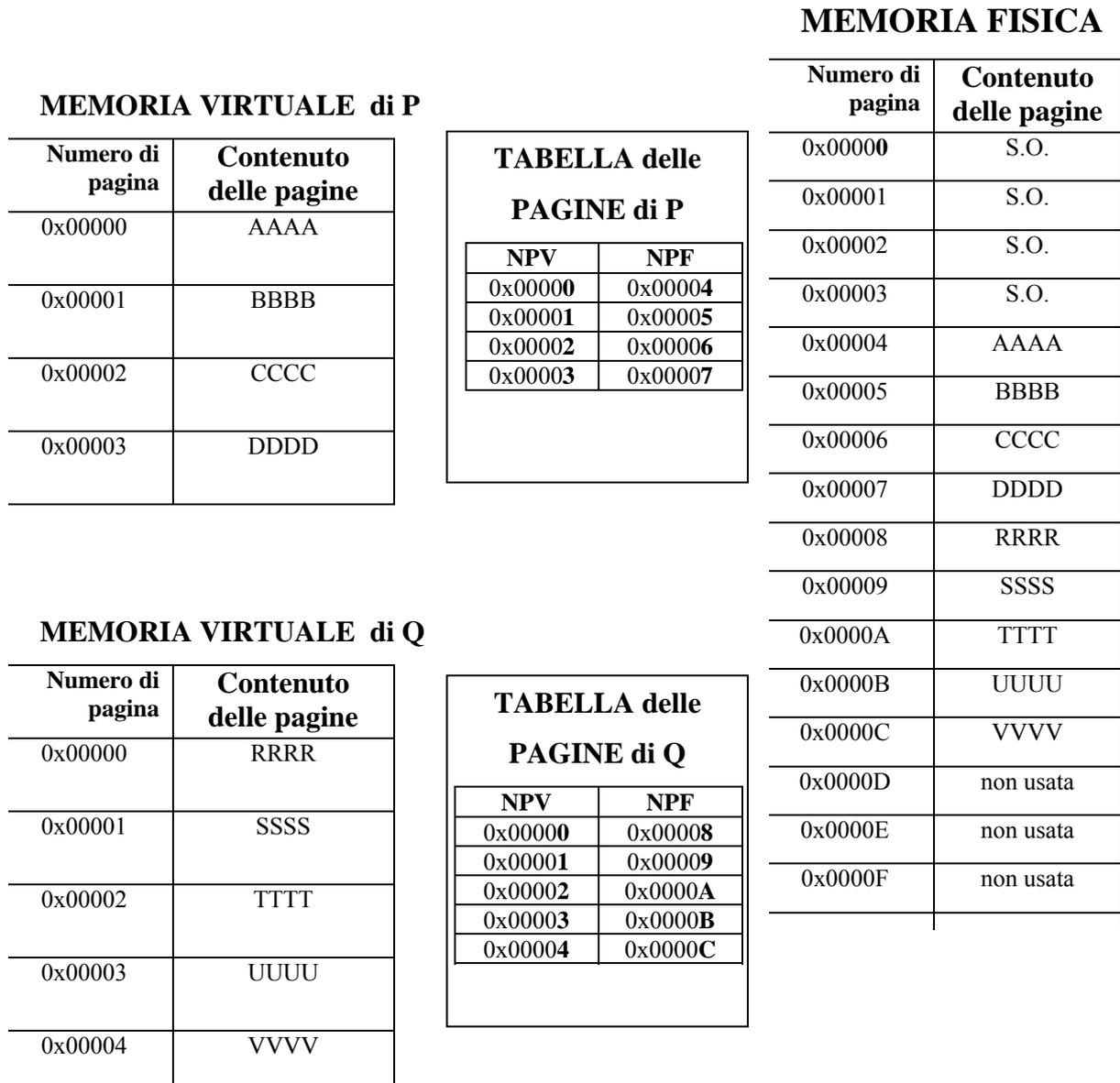
Ipotizziamo ora una specifica configurazione di memoria con riferimento al calcolatore PROC1 (si tenga presente che tutte le dimensioni che verranno utilizzate in questo e in alcuni esempi successivi sono irrealisticamente piccole, per evitare che gli esempi diventino troppo estesi).

- dimensione della memoria fisica: 16 pagine
- dimensione del Sistema Operativo: 4 pagine

Dopo il caricamento del SO sono stati creati 2 processi P e Q, con le seguenti dimensioni

- dimensioni del processo P: 4 pagine
- dimensioni del processo Q: 5 pagine

In figura 8 è rappresentato il contenuto della memoria fisica e delle tabelle delle pagine relative ai due processi P e Q ipotizzando che la memoria fisica sia stata allocata scegliendo ogni volta le prime pagine libere disponibili. ■



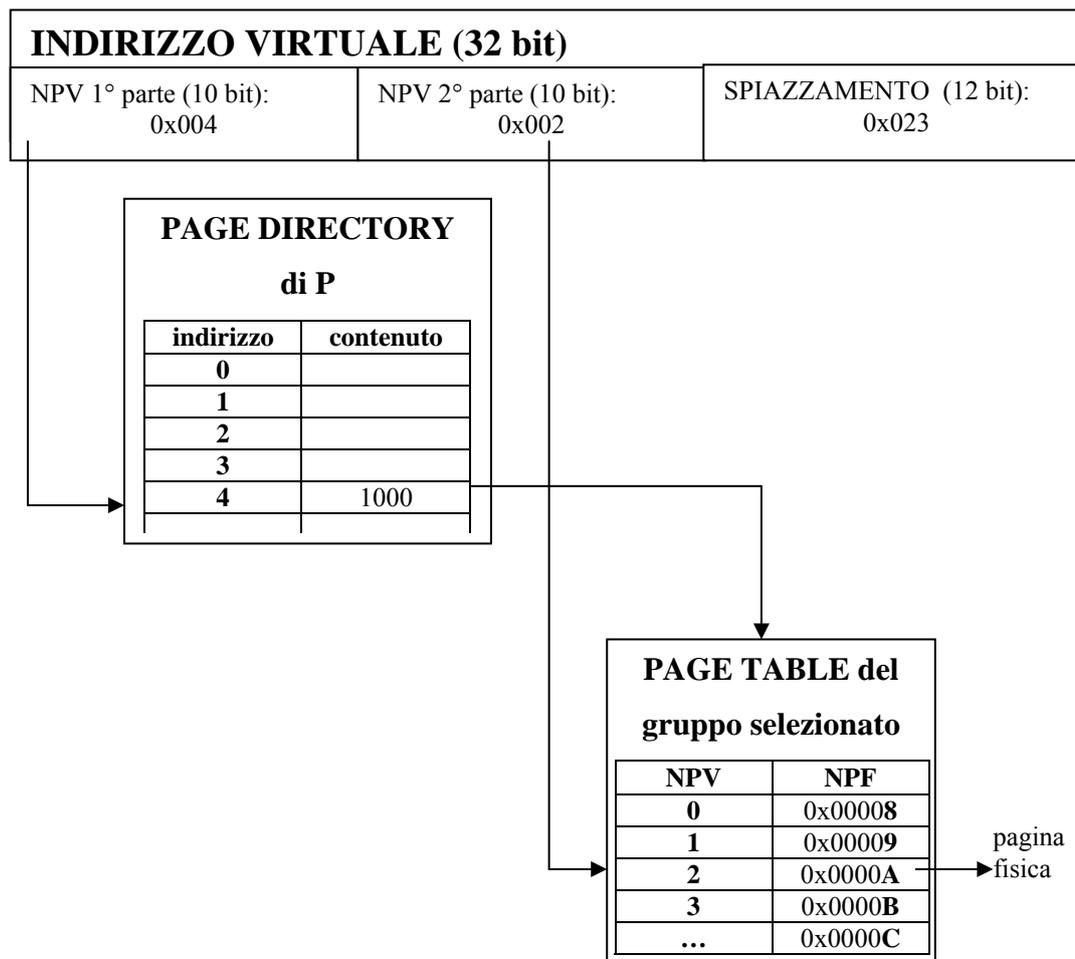
*Figura 8 – Due processi e le relative tabelle delle pagine*

---

### Gestione della Tabella delle Pagine

La tabella delle pagine è una struttura dati che può assumere dimensioni notevoli. Ad esempio, considerando il processore PROC1 dell'esempio precedente, con

indirizzo virtuale di 32 bit e pagine di 4K, il numero di righe della tabella risulta essere di  $2^{20}$  (una per ogni pagina virtuale). Ipotizzando che ogni riga di tale tabella sia costituita da 32 bit (18 bit per il NPF corrispondente, più alcuni bit di controllo di protezione – descritti più avanti), una pagina di memoria da 4Kbyte potrebbe contenere  $2^{10}$  righe e quindi l'intera tabella richiederebbe  $2^{10}$  pagine consecutive della memoria del sistema operativo (per ogni processo). Per rendere più flessibile la gestione della tabella, LINUX prevede di utilizzare una struttura a due livelli (figura 9): le  $2^{20}$  pagine virtuali sono suddivise in  $2^{10}$  gruppi da  $2^{10}$  pagine e i 20 bit del NPV sono suddivisi in 2 parti da 10 bit; la prima parte è utilizzata per trovare, all'interno di un **page directory**, il gruppo al quale appartiene la pagina virtuale e la seconda parte trova la pagina virtuale vera e propria nella **page table** relativa al gruppo selezionato.



*Figura 9 – Gestione a due livelli della tabella delle pagine*

### **Meccanismi Hardware di accelerazione - MMU**

Il meccanismo illustrato sopra è molto elegante, ma non può funzionare senza particolari supporti circuitali, perchè ogni volta che un processo deve accedere alla memoria (cioè almeno una volta per ogni istruzione!) si dovrebbero leggere il page directory e la page table, cioè per ogni accesso necessario si avrebbero 2 accessi supplementari. Per evitare questo onere inaccettabile è necessario che un sistema paginato sia supportato da un dispositivo specializzato, detto **Memory Management Unit (MMU)**, capace di convertire rapidissimamente (operando alla stessa velocità del processore) un NPV in un NPF. Esistono molti tipi di MMU diverse progettate per supportare i diversi tipi di processori, ma la maggior parte si basa sulle idee presentate di seguito.

In generale una MMU è dotata di una memoria molto veloce dedicata a contenere la tabella delle pagine. Dato che tali memorie molto veloci non possono essere così grandi da contenere un'intera tabella delle pagine di un processo, tanto meno molte tabelle di diversi processi, vengono utilizzate mantenendo al loro interno solo quelle porzioni della tabella delle pagine che sono maggiormente utilizzate a un certo istante. In questo modo però non è possibile utilizzare il numero di pagina virtuale come indirizzo, e il numero di pagina virtuale deve essere memorizzato nella tabella stessa, come mostrato dalla figura 10 su un esempio molto piccolo. La memoria di figura 10 deve essere una **memoria associativa**, cioè una memoria nella quale la selezione di una riga non si basa sull'indirizzo, ma su una parte del contenuto, detta **descrittore**. Nella lettura di una memoria associativa si fornisce quindi il valore cercato del descrittore e si ottiene la restante parte del contenuto (cioè il contenuto associato al descrittore). Ad esempio, in figura 10 si mostra l'accesso alla memoria associativa fornendo 23897 come descrittore e ottenendo il valore 9 come risultato. Se il valore del descrittore non è contenuto nella memoria, questa genera un segnale di errore.

Basandosi sulla nozione di memoria associativa è possibile costruire diversi tipi di MMU. Il modo più semplice è quello esemplificato sopra, cioè una tabella associativa contenente solo gli NPV del processo in esecuzione e i corrispondenti NPF. Dato che gli NPV contenuti nella MMU non sono tutti quelli del processo, è possibile che ogni tanto venga richiesto l'accesso a una NPV non presente nella MMU. Questo evento è chiamato spesso un **Table Miss**. Per ridurre la probabilità di non trovare un NPV nella MMU è necessario che la memoria associativa abbia una dimensione che si avvicini al numero R di pagine **residenti** del processo, cioè di pagine presenti in

memoria; in quest'ultimo caso la mancanza della riga della tabella delle pagine nella MMU coinciderebbe con la mancanza della pagina virtuale in memoria.

<b>DESCRITTORE (NPV)</b>	<b>CONTENUTO ASSOCIATO (NPF)</b>
00734	7
00042	3
23897	9
03560	2

Leggi(23897) → 9

***Figura 10 – Esempio di memoria associativa***

Questo metodo è semplice ma ha il seguente difetto: *dato che gli NPV si riferiscono ad un unico processo, ogni volta che il sistema esegue una commutazione di processo l'intero contenuto della MMU deve essere salvato in memoria e sostituito da quello relativo al nuovo processo che viene eseguito*

Una MMU più complessa che elimina il difetto indicato sopra può essere realizzata includendo nel descrittore anche il PID del processo; in questo modo invece di cercare semplicemente un NPV viene cercata una coppia <PID,NPV>. Un registro della MMU, che chiameremo PIDREG, è dedicato a contenere il valore del PID del processo in esecuzione; è compito del S.O. aggiornare tale valore ad ogni commutazione di contesto. In figura 11 è esemplificato questo meccanismo con riferimento ai due processi P e Q di figura 8; in figura 11 è riportata la porzione della tabella delle pagine relativa ai due processi, poi è mostrato il contenuto della MMU. L'ordinamento delle righe della MMU in figura è casuale, perchè è dovuto all'ordine in cui si sono svolte le operazioni precedenti di allocazione delle pagine.

**MEMORIA VIRTUALE di P**

Numero di pagina	Contenuto delle pagine
0x00000	AAAA
0x00001	BBBB
0x00002	CCCC
0x00003	DDDD

**TABELLA delle PAGINE di P**

NPV	NPF
0x00000	0x00004
0x00001	0x00005
0x00002	0x00006
0x00003	0x00007

**MEMORIA VIRTUALE di Q**

Numero di pagina	Contenuto delle pagine
0x00000	RRRR
0x00001	SSSS
0x00002	TTTT
0x00003	UUUU
0x00004	VVVV

**TABELLA delle PAGINE di Q**

NPV	NPF
0x00000	0x00008
0x00001	0x00009
0x00002	0x0000A
0x00003	0x0000B
0x00004	0x0000C

**MMU dotata di PID (Q è in esecuzione)**

registro del PID corrente: <b>pidQ</b>	<b>PID</b>	<b>NPV</b>	<b>NPF</b>
	<b>pidP</b>	0x00000	0x00004
	<b>pidQ</b>	0x00002	0x0000A
	<b>pidQ</b>	0x00004	0x0000C
	<b>pidP</b>	0x00003	0x00007
	<b>pidQ</b>	0x00000	0x00008
	<b>pidP</b>	0x00001	0x00005
	<b>pidQ</b>	0x00001	0x00009
	<b>pidP</b>	0x00002	0x00006
	<b>pidQ</b>	0x00003	0x0000B

*Figura 11 – Contenuto di una MMU dotata di PID (Q è in esecuzione)*

### **Condivisione delle pagine**

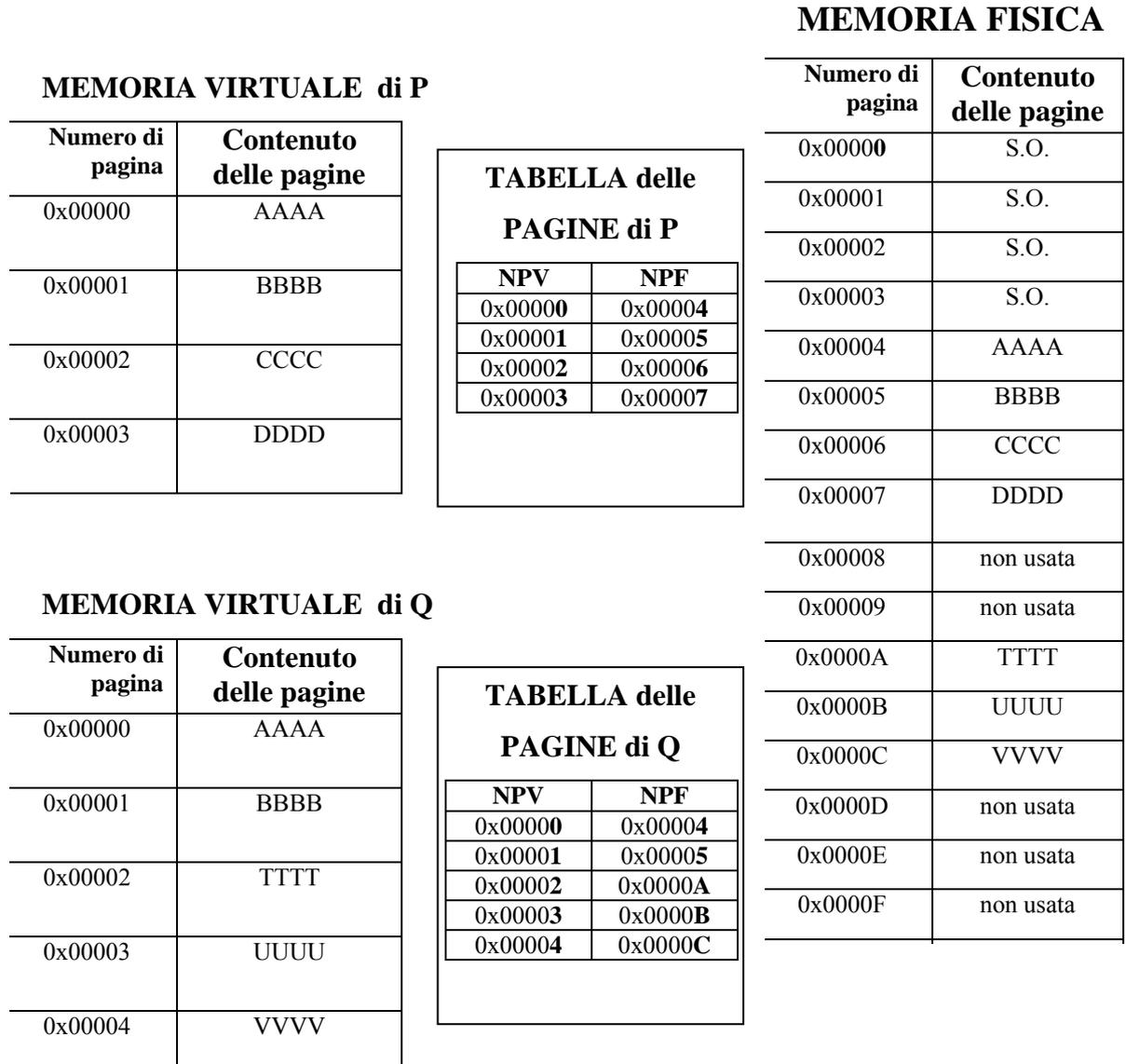
Come enunciato nell'introduzione, talvolta è utile o necessario mantenere in memoria una sola copia di pagine che sono condivise da più processi. Le motivazioni e i meccanismi per la condivisione saranno analizzati nel paragrafo relativo alla struttura interna della memoria virtuale. Per ora, assumiamo come tipico esempio di opportunità di condivisione di memoria tra due processi il caso in cui i due processi eseguono lo stesso programma; è evidente che il codice del programma può essere memorizzato una volta sola, dato che i due processi si limitano a leggerlo senza modificarlo.

Ai fini della paginazione la condivisione della memoria si realizza molto semplicemente: basta imporre che la porzione condivisa della memoria sia costituita da un numero intero di pagine e quindi mappare, nelle tabelle delle pagine dei processi interessati, sulla stessa pagina fisica le pagine virtuali da condividere.

#### **Esempio 1 (continua)**

Supponiamo che i due processi P e Q dell'esempio 1 condividano le loro due pagine virtuali iniziali, che per ipotesi contengono il codice di uno stesso programma. In questo caso la situazione della memoria è quella di figura 12 (con riferimento allo stesso esempio di figura 8), nella quale si è ipotizzato che il processo P sia stato creato per primo e quindi, quando è stato creato il processo Q, invece di allocare nuova memoria per le prime due pagine virtuali, ci si è limitati a indicare, nella sua tabella delle pagine, il riferimento alle due pagine fisiche già allocate per il processo P .

Per mettere più in evidenza l'effetto della condivisione, nella figura 12 le pagine fisiche non più utilizzate sono state lasciate libere, invece di compattare la memoria allocando la porzione restante del processo Q subito dopo P ■



*Figura 12 – I processi P e Q dell'esempio di figura 8 condividono le due pagine iniziali*

### Protezione delle pagine

Dato che molti processi utilizzano contemporaneamente la memoria fisica, è necessario garantire che ogni processo sia limitato a svolgere le operazioni sulla propria memoria. Dato che molti linguaggi, in particolare il linguaggio C, permettono un uso molto libero dei puntatori, un programma potrebbe generare indirizzi virtuali scorretti, che il processo di traduzione e collegamento non sarebbero in grado di evitare. La protezione è resa ancora più necessaria dall'esigenza di condivisione della memoria tra processi; la condivisione impedisce di limitare il controllo alla garanzia che un processo operi solo sulla propria memoria – bisogna invece controllare che un processo operi correttamente su tutta la memoria alla quale può accedere.

Il meccanismo di protezione più semplice consiste nell'associare ad ogni pagina un'informazione che indica se il processo può accederla in lettura (**R**), scrittura (**W**) oppure esecuzione (**X**) (quest'ultimo tipo di accesso significa che la pagina contiene istruzioni che devono essere lette ed eseguite). Il tipo di diritto di accesso a una pagina può essere inserito nella riga corrispondente della tabella delle pagine e deve essere gestito anche dalla MMU, che, in presenza di violazione di un diritto di accesso, genera un interrupt di violazione della memoria.

### Gestione di pagine virtuali non residenti in memoria

I meccanismi visti finora permettono di raggiungere i primi 3 obiettivi citati nelle motivazioni iniziali; l'ultimo obiettivo, cioè l'esecuzione di un numero di programmi di dimensioni virtuali tali da non poter essere contenuti nella memoria fisica deve essere ancora affrontato.

In sostanza, quello che vogliamo ottenere è la possibilità di eseguire un programma anche se non possiamo caricare tutte le sue pagine virtuali in memoria<sup>3</sup>. Il meccanismo di virtualizzazione si basa sulla paginazione e sui seguenti principi:

1. Durante l'esecuzione di un programma solo un certo numero delle sue pagine virtuali è caricato in altrettante pagine fisiche; tali pagine sono dette **residenti**.
2. Ad ogni accesso alla memoria da parte del programma si controlla che l'indirizzo virtuale generato appartenga a una pagina residente, altrimenti si

---

<sup>3</sup>Il meccanismo che risolve questo problema è quello che ha dato il nome alla memoria virtuale di un processo, perché il processo viene eseguito anche se una parte della sua memoria non esiste fisicamente – la memoria esiste solo virtualmente.

produce un interrupt di segnalazione di errore, detto **page-fault**, e il processo viene sospeso in attesa che la pagina contenente l'indirizzo virtuale richiesto venga caricata dal disco<sup>4</sup>.

3. Se necessario, una pagina già residente viene scaricata su disco per liberare una pagina fisica che possa contenere la nuova pagina virtuale.

Con questo modo di operare il programma viene eseguito utilizzando solo un numero prestabilito  $R$  di pagine fisiche, indipendentemente dalle sue dimensioni virtuali. *La condizione fondamentale affinché questo meccanismo funzioni è che il numero di richieste di accessi alla memoria che causano un page fault sia basso rispetto al numero complessivo di accessi (ad esempio, <5%).* Perciò, prima di considerare i dettagli della realizzazione del meccanismo è necessario fare alcune considerazioni relativamente al comportamento dei programmi.

### **Caratteristiche dell'accesso dei programmi alla memoria**

E' stato rilevato statisticamente che i programmi tendono ad esibire, nel modo di accedere alla memoria, una caratteristica detta **località**, per cui la distribuzione degli accessi nello spazio e nel tempo non è omogenea. In particolare, parliamo di località temporale e di località spaziale in base alle seguenti definizioni.

**Località temporale:** indica che un programma accederà, nel prossimo futuro, gli indirizzi che ha referenziato nel passato più recente. Questo tipo di località è tipicamente dovuto all'esecuzione di cicli, che riaccedono le stesse istruzioni e gli stessi dati.

**Località spaziale:** indica che un programma accede con maggiore probabilità indirizzi vicini a quello utilizzato più recentemente: Questo tipo di località è dovuto alla sequenzialità delle istruzioni e all'attraversamento di strutture come gli array.

Naturalmente, il grado di località dipende dai programmi e differisce da un programma all'altro.

Questo comportamento può essere caratterizzato tramite la nozione di **Working Set**. Il Working Set di ordine  $k$  di un programma è l'insieme delle pagine referenziate durante gli ultimi  $k$  accessi alla memoria. Per  $k$  sufficientemente grande il Working Set

---

<sup>4</sup> Un processore sul quale si vuole far funzionare il meccanismo di virtualizzazione della memoria deve avere la capacità di interrompere l'esecuzione di un'istruzione a metà e di rieseguire più tardi la stessa istruzione come se non avesse mai iniziato ad eseguirla, perché la generazione di un page fault può causare la sospensione dell'esecuzione di un'istruzione dopo che questa è iniziata, ad esempio durante l'accesso a un operando (restartable instructions).

di un programma varia molto lentamente a causa delle proprietà di località esposte sopra, quindi se si mantengono in memoria le  $k$  pagine accedute più recentemente è molto probabile che il prossimo accesso sia all'interno di tali pagine, cioè che non si verifichi un page fault.

Il sistema operativo gestisce perciò la memoria virtuale sulla base di un parametro  $R$ , che esprime il numero delle pagine di un programma che devono essere residenti durante la sua esecuzione, che viene scelto durante la configurazione del sistema sulla base di una stima del Working Set. La scelta di tale parametro determina la frequenza dei page fault.  $R$  deve esprimere una mediazione tra l'esigenza di avere pochi page fault (che spinge a scegliere  $R$  grande) e l'esigenza di far convivere molti processi in una memoria fisica limitata (che spinge a scegliere  $R$  piccolo).

Si noti che la nozione di Working Set porta a mantenere residenti in memoria le  $R$  pagine di un processo che sono state accedute per ultime; infatti, se un programma che ha già  $R$  pagine residenti genera un page fault, richiedendo il caricamento di una nuova pagina, per non superare il valore di  $R$  deve essere scaricata (cioè resa non residente) una delle  $R$  pagine già residenti – in base al principio di località la pagina da scaricare sarà quella non acceduta da più tempo. L'algoritmo che realizza lo scaricamento delle pagine secondo questo principio è detto **LRU** (least recently used); LRU sceglie quindi come prossima pagina da scaricare quella alla quale non si accede da più tempo.

L'algoritmo LRU è difficile da realizzare, perchè richiede di poter determinare quale sia la pagina non utilizzata da più tempo, e questa informazione non è facile da ottenere senza il supporto della MMU. Un tipico supporto è il seguente: la MMU possiede un bit per ogni riga, detto **bit di accesso**, che viene posto a 1 ogni volta che la pagina viene acceduta, e può essere azzerato esplicitamente dal sistema operativo. Il sistema operativo esegue periodicamente una routine che controlla tali bit e li riazzera; durante tale controllo il sistema incrementa una variabile detta "invecchiamento" per ogni pagina con il bit di accesso = 0. Questa variabile quindi indica, con una certa approssimazione, il grado di non-accesso a una pagina e quindi la scelta della pagina più vecchia può essere basata su di essa.

**Immagine su disco e immagine in memoria – Copy on write**

Un aspetto importante della virtualizzazione consiste nel fatto che la memoria virtuale di un processo non è contenuta completamente nella memoria fisica durante l'esecuzione; *la parte non contenuta in memoria fisica deve esistere su disco*.

In particolare, alcune parti della memoria virtuale del processo possiedono un'immagine su disco che è costituita direttamente dal file contenente l'eseguibile: ad esempio, la parte che costituisce il codice. Dato che il codice non viene modificato, anche se una pagina di codice non è residente in memoria, la sua immagine esiste però nel file eseguibile.

Per le parti del processo che vengono modificate e/o allocate dinamicamente invece è necessario che esista un file particolare, detto **swap file**, sul quale viene salvata l'immagine delle pagine quando queste vengono scaricate. Dato che la suddivisione del processo in "parti" verrà trattata al prossimo paragrafo, qui non analizziamo come essa sia realizzata; ci basta sapere che tutte le pagine di un processo sono in corrispondenza con le corrispondenti pagine su file.

Infine, per le parti del processo che vengono modificate e/o allocate dinamicamente, quando una pagina viene scaricata, il sistema operativo deve decidere se tale pagina deve essere riscritta sul disco perchè è stata modificata oppure no: per permettere questa scelta la MMU possiede in genere un bit per ogni riga, detto **bit di modifica**, azzerato quando la pagina viene caricata in memoria e posto a uno ogni volta che viene scritta una parola di tale pagina. Ovviamente, le pagine che al momento dello scaricamento hanno il bit di modifica uguale a 0 non richiedono di essere ricopiate sul disco.

**Caricamento iniziale del programma – demand paging**

Un programma può essere lanciato in esecuzione anche se nessuna sua pagina è residente in memoria. La sua page table dovrà indicare che nessuna pagina virtuale è residente. Ovviamente, quando il processore tenta di leggere la prima istruzione genera immediatamente un page fault, e la relativa pagina virtuale verrà caricata in memoria. A questo punto inizia l'esecuzione del programma, che genererà progressivamente dei page fault a fronte dei quali verranno caricate nuove pagine senza scaricarne alcuna; dopo R page faults il programma avrà R pagine residenti e i page fault si ridurranno a quelli statisticamente accettabili.

### 3. Strutturazione della memoria virtuale - Segmentazione

La memoria virtuale in LINUX non viene considerata come un'unica memoria lineare, ma viene suddivisa in porzioni che corrispondono alle diverse parti di un programma eseguibile.

Esistono svariate motivazioni per considerare il modello della memoria virtuale non come un modello lineare indifferenziato ma come un insieme di pezzi dotati di diverse caratteristiche; le principali sono le seguenti:

- Distinguere diverse parti del programma in base ai permessi di accesso (solo lettura, lettura e scrittura) alla relativa memoria; in questo modo è possibile evitare che il contenuto di un'area di memoria venga modificata per errore
- Permettere a diverse parti del programma di crescere separatamente (in particolare, come vedremo, le due aree dette pila e dati dinamici rientrano in questa categoria)
- Permettere di individuare aree di memoria che è opportuno condividere tra processi diversi; infatti, i meccanismi visti nel paragrafo precedente permettono la condivisione di una pagina tra due processi, ma non supportano la possibilità di definire quali pagine dei processi devono essere condivise

Per questi motivi la memoria virtuale di un processo LINUX è suddivisa in un certo numero di **aree di memoria virtuale (virtual memory area) o segmenti**<sup>5</sup>. Ogni area di memoria virtuale è caratterizzata da una coppia di indirizzi virtuali che ne definiscono l'inizio e la fine. Dato che le aree virtuali devono essere costituite da un numero intero di pagine, in realtà la loro definizione richiede una coppia  $NPV_{iniziale}$ ,  $NPV_{finale}$ .

I tipi più importanti di aree di memoria virtuale di un processo sono i seguenti:

- **Codice:** è l'area che contiene le istruzioni che costituiscono il programma da eseguire

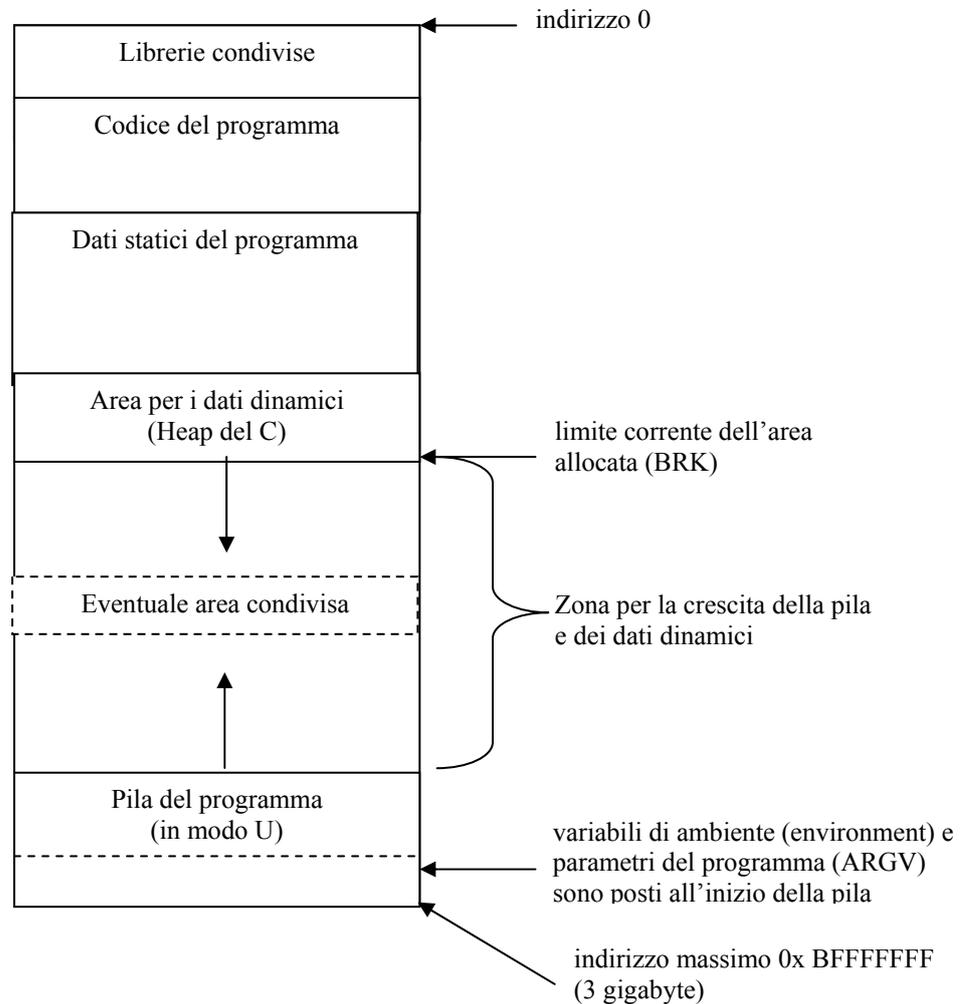
---

<sup>5</sup> Nota terminologica: usualmente ciò che è chiamato virtual memory area in LINUX è chiamato "segmento" nella letteratura. Nella documentazione di LINUX il termine segmento è poco usato in questo contesto, perché nella implementazione su processore x386 il segmento è una nozione Hardware che non coincide con la Virtual memory area. Inoltre, il termine segmento è utilizzato anche per indicare la suddivisione in segmento codice, dati di utente e dati di sistema, che non coincide esattamente con la suddivisione in aree virtuali. Noi useremo invece indifferentemente i termini area di memoria virtuale e segmento.

- **Pila:** è l'area di pila di modo U del processo, che contiene tutte le variabili ad allocazione automatica delle funzioni di un programma C (o di un linguaggio equivalente)
- **Dati statici:** è l'area destinata a contenere i dati allocati per tutta la durata di un programma
- **Dati dinamici:** è l'area destinata a contenere i dati allocati dinamicamente su richiesta del programma (nel caso di programmi C, è la memoria allocata tramite la funzione malloc, detta **heap**); il limite corrente di quest'area è indicato dalla variabile BRK contenuta nell'informazione di sistema del processo
- **Memoria condivisa:** è un'area dati di un processo che può essere acceduta anche da altri processi in base a regole definite dai meccanismi di comunicazione tra processi tramite condivisione della memoria. Il meccanismo adottato da LINUX per la comunicazione è, per ragioni storiche di compatibilità, troppo complesso per essere spiegato qui; noi ci limitiamo a considerare che due o più processi possano dichiarare al loro interno un'area virtuale destinata ad essere condivisa; tale area avrà indirizzi virtuali diversi nei diversi processi, ma sarà mappata su un'unica area fisica, tramite pagine fisiche condivise (come abbiamo visto precedentemente per il codice condiviso)
- **Librerie dinamiche (shared libraries o dynamic linked libraries):** sono librerie il cui codice non viene incorporato staticamente nel programma eseguibile dal collegatore ma che vengono caricate in memoria durante l'esecuzione del programma in base alle esigenze del programma stesso. Una caratteristica fondamentale delle librerie dinamiche è quella di poter essere condivise tra diversi programmi.

Per ognuno di questi tipi possono esistere una o più aree di memoria virtuale. Alcune di queste aree, in particolare la pila e lo heap sono di tipo dinamico e quindi generalmente sono piccole quando il programma viene lanciato in esecuzione ma devono poter crescere durante l'esecuzione del programma.

In figura 13 è mostrata la tipica struttura di un programma eseguibile in LINUX, quindi questa è la struttura della memoria virtuale del processo al momento del lancio in esecuzione del programma (tramite il servizio exec).



**Figura 13 – Struttura di un programma eseguibile**

### Crescita delle aree dinamiche – servizio brk

La pila cresce automaticamente, nel senso che il programma non contiene comandi espliciti per la crescita della pila.

La memoria dati dinamica (heap) invece cresce quando il programma lo richiede (in linguaggio C tramite malloc). In realtà la funzione malloc ha bisogno di richiedere al SO di allocare nuovo spazio, e questo avviene tramite l'invocazione del servizio di sistema `brk()` o `sbrk()`.

*Brk( )* e *sbrk( )* sono due funzioni di libreria che invocano in forma diversa lo stesso servizio; analizziamo solamente la forma *sbrk*, che ha la seguente sintassi:

*\*void sbrk(int incremento)*

e incrementa l'area dati dinamici del valore incremento e restituisce un puntatore alla posizione iniziale della nuova area. Il servizio *sbrk* modifica l'indirizzo massimo del segmento dati dinamici, arrotondandolo ad un limite di pagina e controllando che la crescita dei dati non si avvicini troppo alla pila. *Sbrk(0)* restituisce il valore corrente della cima dell'area dinamica.

In figura 14a è riportato un semplice programma C che illustra, stampando in esadecimale i valori di alcuni puntatori, la struttura di un programma eseguibile in LINUX e la sua modifica tramite *sbrk*. I risultati dell'esecuzione su un calcolatore della famiglia x386, mostrati in figura 14b, forniscono una conferma della struttura generale di figura 13; in particolare:

- La pagina più alta della memoria virtuale è 0xBFFFF; questo fatto è dovuto alla scelta di assegnare ai programmi uno spazio di indirizzamento virtuale di 3 Gbyte, lasciando 1 Gbyte per l'indirizzamento del S.O. (l'indirizzo più alto di 3 Gbyte inizia con i bit 1011 che corrispondono alla B esadecimale);
- La pila è posta nella zona alta dello spazio di indirizzamento, cresce verso indirizzi più bassi e contiene inizialmente i parametri del programma (*argv*);
- Le funzioni dell'utente vengono caricate sopra il *main*; le funzioni di libreria prima del *main*;

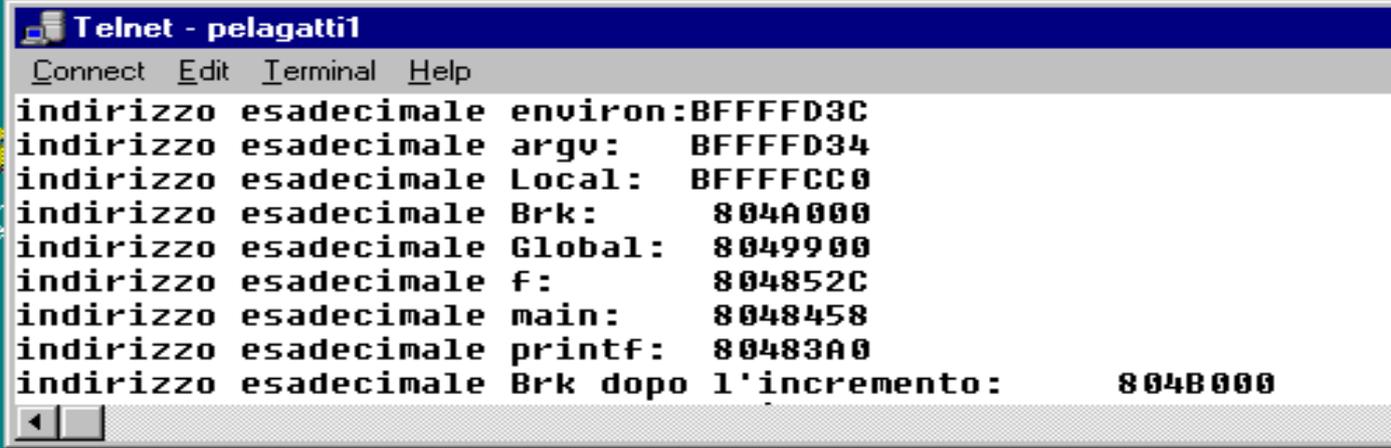
```

#include <stdio.h>
int Global[10];
int main(argc, argv)
{
    void f();
    int Local[10];
    void *Brk;
    extern char **environ;
    Brk=sbrk(0);
    printf( "indirizzo esadecimale environ:%X\n",environ);
    printf( "indirizzo esadecimale argv:  %X\n",argv);
    printf( "indirizzo esadecimale Local:  %X\n",Local);
    printf( "indirizzo esadecimale Brk:   %X\n",Brk);
    printf( "indirizzo esadecimale Global: %X\n",Global);
    printf( "indirizzo esadecimale f:    %X\n",f);
    printf( "indirizzo esadecimale main:  %X\n",main);
    printf( "indirizzo esadecimale printf: %X\n",printf);

    Brk=sbrk(4096);
    Brk=sbrk(0);
    printf( "indirizzo esadecimale Brk dopo l'incremento:  %X\n",Brk);
}
void f(){
}

```

a) *Un programma che stampa indirizzi di memoria*



```

Telnet - pelagattil
Connect Edit Terminal Help
indirizzo esadecimale environ:BFFFFFFD3C
indirizzo esadecimale argv:  BFFFFFFD34
indirizzo esadecimale Local:  BFFFFFFC0
indirizzo esadecimale Brk:   804A000
indirizzo esadecimale Global: 8049900
indirizzo esadecimale f:    804852C
indirizzo esadecimale main:  8048458
indirizzo esadecimale printf: 80483A0
indirizzo esadecimale Brk dopo l'incremento:      804B000

```

b) *Risultati dell'esecuzione del programma*

*Figura 14 – Esplorazione della struttura virtuale di un processo LINUX  
tramite un programma C*

## Gestione della memoria

Ogni area virtuale in LINUX è definita da una variabile strutturata di tipo `vm_area_struct`. Le strutture delle singole aree sono collegate tra loro in vario modo per rendere efficienti gli algoritmi di gestione della memoria.

La struttura `vm_area_struct` contiene tra l'altro il puntatore `vm_inode`, che è un puntatore a un file il cui contenuto è mappato sull'area considerata e un numero `vm_offset`, che indica l'inizio dell'area all'interno del file. In questo modo per ogni area virtuale è definita una zona di un file sul quale l'area viene salvata.

Il sistema di gestione della memoria deve intervenire in molti diversi momenti durante il funzionamento del sistema. Analizziamone alcuni dei più importanti.

### Fork

Un momento fondamentale è l'esecuzione di una Fork, perchè la creazione di un nuovo processo implicherebbe la creazione di tutta la struttura di memoria di un nuovo processo. Dato che il nuovo processo è l'immagine del padre, LINUX in realtà si limita ad aggiornare le informazioni generali della tabella dei processi, ma non alloca nuova memoria. In realtà, LINUX opera come se il nuovo processo condividesse tutta la memoria con il padre. Questo vale finchè uno dei due processi non scrive in memoria; in quel momento la pagina scritta viene duplicata, perché i due processi hanno dati diversi in quella stessa pagina virtuale. Questa tecnica è una realizzazione del principio "copy on write" descritto sopra. Per permettere di scoprire l'operazione di scrittura, a tutte le pagine, anche quelle appartenenti ad aree virtuali scrivibili, viene associata una abilitazione in sola lettura, causando quindi un errore di accesso alla prima scrittura che viene gestito opportunamente (vedi sotto, gestione degli errori).

### Exec

Ovviamente, molti programmi dopo una fork eseguono una exec. Al momento dell'exec LINUX invalida tutta la page table del processo e dovrebbe caricare un certo numero di pagine, determinato dal parametro R, dal nuovo file eseguibile nella memoria. Dato che non è possibile sapere quali pagine caricare, LINUX applica la tecnica del demand paging, cioè carica le pagine in base ai page fault che vengono generati.

### Page Fault e altri errori di accesso alla memoria

Quando si genera un page fault relativo a un numero di pagina virtuale V, LINUX deve determinarne la causa e procedere di conseguenza, secondo uno schema di cui la seguente è una versione semplificata:

1. Se V non appartiene alla memoria virtuale del processo, allora
  - 1.1 se si tratta di una crescita di pila (per determinare questo fatto LINUX controlla se la pagina V+1 contiene l'indicatore growsdown) allora LINUX alloca la nuova pagina virtuale V al processo;
  - 1.2 altrimenti il processo viene abortito;
2. Se V appartiene alla memoria virtuale del processo, LINUX controlla se l'accesso richiesto era legittimo rispetto ai codici di protezione:
  - 2.1 se l'accesso non è legittimo, sono possibili due casi:
    - 2.1.1 se la violazione era dovuta a una pagina posta come "sola lettura" in seguito a una fork, ma appartenente ad un'area virtuale sulla quale la scrittura è ammessa, viene creata una nuova copia della pagina con l'indicazione di scrittura abilitata;
    - 2.1.2 altrimenti viene abortito il programma;
  - 2.2 altrimenti viene invocata la routine **swap\_in** che deve caricare in memoria la pagina virtuale V.

La routine swap-in carica una pagina virtuale in una pagina fisica libera, determinata in base a una opportuna struttura dati. Se non la trova, allora deve cercare di ottenere la liberazione di una pagina. L'algoritmo utilizzato per liberare una pagina è piuttosto complesso e non viene descritto qui; si consideri che possono essere inizialmente ridotti i buffer per i dischi fino a un certo livello, poi si deve procedere a scaricare, tramite la routine **swap\_out** le pagine di processi. L'algoritmo utilizzato per scegliere quali pagine di un processo scaricare è sostanzialmente LRU, che, come descritto precedentemente, scarica le pagine più "invecchiate", cioè utilizzate meno recentemente.

## 4. Esercizio conclusivo

### Notazione sintetica per indicare indirizzi di pagina

Negli esercizi utilizziamo la seguente notazione sintetica per indicare le pagine delle aree virtuali dei programmi e dei processi:

La pagina virtuale  $n$  dell'area virtuale  $A$  del programma o processo  $P$  è indicata con la notazione **APn**, dove:

- $A$  indica un tipo di area virtuale secondo la convenzione seguente: **C** (codice), **D** (dati), **P** (pila), **COND** (area dati condivisa)
- $P$  indica il programma o il processo
- $n$  indica il numero di pagina nell'ambito dell'area virtuale

Esempio: DQ3 indica la pagina 3 dell'area dati del processo Q (che non è la pagina con NPV = 3, perchè in generale l'area DP non inizia con NPV = 0).

Inoltre, in questo tipo di esercizi è utile indicare gli NPV omettendo gli zeri iniziali e l'indicazione della notazione esadecimale (ad esempio, N invece di 0xN).

### Esercizio

Un sistema dotato di memoria virtuale con paginazione e segmentazione di tipo UNIX è caratterizzato dai parametri seguenti: la memoria centrale fisica ha capacità di 32 Kbyte, quella logica di 32 Kbyte e la pagina ha dimensione 4 Kbyte. Si chiede di svolgere i punti seguenti:

- a. **Si definisca** la struttura degli indirizzi fisico e logico indicando la lunghezza dei campi NPF, Spiazzamento fisico, NPL, Spiazzamento logico
- b. Nel sistema vengono creati alcuni processi, indicati nel seguito con P, Q, R, S. I programmi eseguiti da tali processi sono due: X e Y. La dimensione iniziale dei segmenti dei programmi è la seguente:

CX: 8 K      DX: 4 K      PX: 4 K

CY: 12 K      DY: 8 K      PY: 4 K

**Si inserisca** in tabella 1 la struttura in pagine della memoria virtuale (mediante la notazione definita sopra: CX0 CX1 DX0 PX0 ... CY0 ...).

indir. virtuale	prog. X	prog. Y
0		
1		
2		
3		
4		
5		
6		
7		

1) memoria logica

indir. fisico	pagine allocate al tempo $t_0$
0	
1	
2	
3	
4	
5	
6	
7	

2) memoria fisica, istante  $t_0$ 

indir. fisico	pagine allocate al tempo $t_1$
0	
1	
2	
3	
4	
5	
6	
7	

3) memoria fisica, istante  $t_1$ 

- c. A un certo istante di tempo  $t_0$  sono terminati, nell'ordine, gli eventi seguenti:
1. creazione del processo P e lancio del programma Y ("fork" di P ed "exec" di Y)
  2. creazione del processo Q e lancio del programma X ("fork" di Q ed "exec" di X)
  3. accesso a 1 pagina dati e creazione di una nuova pagina di pila da parte di P
  4. accesso a 1 pagina dati da parte di Q
  5. creazione del processo R come figlio di P ("fork" eseguita da P)
  6. creazione di 1 pagina di pila da parte di R

Sapendo che:

- il lancio di una programma avviene caricando solamente la pagina di codice con l'istruzione di partenza e una sola pagina di pila
- il caricamento di pagine ulteriori è in Demand Paging (cioè le pagine si caricano su richiesta senza scaricare le precedenti fino al raggiungimento del numero massimo di pagine residenti)
- l'indirizzo (esadecimale) dell'istruzione di partenza di X è 14AF
- l'indirizzo (esadecimale) dell'istruzione di partenza di Y è 0231
- il numero di pagine residenti **R** vale **3** (tre)
- viene utilizzato l'algoritmo LRU (ove richiesto prima si dealloca una pagina di processo e poi si procede alla nuova assegnazione)
- le pagine meno utilizzate in ogni processo sono quelle caricate da più tempo, con la sola eccezione seguente: se è residente una sola pagina di codice, quella è certamente stata utilizzata recentemente

- al momento di una fork viene duplicata solamente la pagina di pila caricata più recentemente
- dopo la fork le pagine di codice possono essere condivise tra i processi padre e figlio, se ambedue i processi usano la stessa pagina virtuale

e ipotizzando che l’allocazione delle pagine virtuali nelle pagine fisiche avvenga **sempre** in sequenza, senza buchi, a partire dalla pagina fisica 0, **si indichi**, completando tabella 2, l’allocazione fisica delle pagine dei tre processi all’istante  $t_0$  (notazione CP0 CP1 DP0 PP0 ... CQ0 ...).

d) A un certo istante di tempo  $t_1 > t_0$  sono terminati gli eventi seguenti:

7. terminazione del processo P (exit)
8. esecuzione della funzione “exec Y” (lancio di Y) nel processo Q e conseguente trasformazione di Q in processo S (si noti che Q si trasforma in S ma il pid resta lo stesso perché non c’è “fork”)
9. accesso a 2 pagine di dati per il processo S

**Si completi** la tabella 3 nelle medesime ipotesi delineate nel precedente punto (c) e supponendo che, dovendo utilizzare una pagina fisica libera, venga **sempre** utilizzata la pagina fisica libera avente indirizzo minore. Si aggiorni anche la tabella 1A (non è indispensabile).

e) **Si indichi** il contenuto della tabella delle pagine della MMU all’istante di tempo  $t_1$  completando la tabella 4. Si ipotizzi che le righe della tabella siano state allocate ordinatamente man mano che venivano allocate le pagine di memoria virtuale e che gli eventi di cui ai punti (c, d), influenzanti la MMU, partano da una situazione di tabella vergine, abbiano utilizzato le righe lasciate libere e che se è richiesta una nuova riga si utilizzi **sempre** la prima riga libera. **Si indichi** anche il valore assunto dal bit di validità di pagina (il valore 1 significa che la pagina è caricata). Il numero di righe nella tabella sotto non è significativo.

<b>PID</b> <i>indicare P Q R o S come pid oppure ns se la riga non è significativa</i>	<b>NPV</b> <i>utilizzare la notazione CP0/0 per indicare “segmento di codice di P numero 0 / pagina virtuale numero 0”, e similmente per le altre</i>	<b>NPF</b>	<b>Bit di Validità</b>

**Tabella 4 (Aggiungere righe quanto necessario)**

**Soluzione**

a) NPF: 3, Spiazzamento fisico: 12, NPL: 3, Spiazzamento logico: 12

b) Il contenuto della tabella 1 è riportato sotto ed è facilmente interpretabile ricordando che l'area di pila è allocata in fondo alla memoria

c) Per facilitare la comprensione del contenuto della tabella 2 (tempo  $t_0$ ), riportato sotto, si forniscono le seguenti spiegazioni, seguendo la numerazione degli eventi:

1. il processo P parte caricando la pagina CP0 (perché il programma Y ha istruzione di partenza in pagina 0) e una pagina di pila PP0
2. successivamente Q carica CQ1 (perché il programma X ha istruzione di partenza in pagina 1) e PQ0
3. P carica DP0, raggiungendo il limite delle pagine residenti (3), e quindi per caricare la pagina PP1 deve eliminare PP0 (vedi regole di utilizzazione indicate nel tema)
4. Q carica DQ0 e raggiunge 3 pagine caricate
5. il nuovo processo R non ha bisogno di caricare CR0, perché esegue lo stesso programma di P e quindi CR0 è uguale a CP0, quindi carica solamente PR1 (che è la copia della pagina PP1, ma conterrà un diverso pid)
6. R carica PR2

indir. virtuale	prog. X	prog. Y
0	CX0	CY0
1	CX1	CY1
2	DX0	CY2
3		DY0
4		DY1
5		
6		
7	PX0	PY0

indir. fisico	pagine allocate al tempo $t_0$
0	CP0 (= CR0)
1	<del>PP0</del> PP1
2	CQ1
3	PQ0
4	DP0
5	DQ0
6	PR1
7	PR2

indir. fisico	pagine allocate al tempo $t_1$
0	CR0 (= CS0)
1	<del>PS0</del> DS1
2	<del>CQ1</del> DS0
3	<del>PQ0</del> ---
4	
5	
6	PR1
7	PR2

d) Il contenuto della tabella 3 si spiega nel modo seguente

7. P termina e libera le pagine fisiche 1 e 4 (non la 0, perché CR0 rimane necessaria)
8. la exec Y da parte di Q non alloca il codice, perché CS0 risulta identica a CR0, ma alloca la pagina PS0 nella pagina fisica 1, già libera; vengono inoltre liberate le pagine fisiche 2 e 3
9. la nuova pagina DS0 viene allocata in pagina fisica 2, ma S raggiunge così il livello massimo di pagine residenti e quindi la successiva (DS1) deve essere allocata al posto di PS0

e) Per permettere di interpretare il risultato riportato nella seguente tabella si indicano, oltre al contenuto della tabella all'istante finale, anche i contenuti precedenti.

<b>PID</b> <i>indicare P Q R o S come pid oppure ns se la riga non è significativa</i>	<b>NPV</b> <i>utilizzare la notazione CP0/0 per indicare "segmento di codice di P numero 0 / pagina virtuale numero 0", e similmente per le altre</i>	<b>NPF</b>	<b>Bit di Validità</b>
<del>P</del> <sub>ns</sub> S	<del>CP0</del> CS0 / 0	0	1
<del>P</del> <sub>ns</sub> S	<del>PP0</del> <del>PP1</del> <del>PS0</del> DS1 / 4	1	1
<del>Q</del> <sub>ns</sub> S	<del>CQ1</del> DS0 / 3	2	1
<del>Q</del> <sub>ns</sub>	<del>PQ0</del>	3	0
<del>P</del> <sub>ns</sub>	<del>DP0</del>	4	0
<del>Q</del> <sub>ns</sub>	<del>DQ0</del>	5	0
R	CR0 / 0	0	1
R	PR1 / 6	6	1
R	PR2 / 5	7	1