Giuseppe Pelagatti

Programmazione e Struttura del sistema operativo Linux

Appunti del corso di
Architettura dei Calcolatori e Sistemi Operativi (AXO)

versione 2016

Parte M: La gestione della Memoria

cap. M2 – Organizzazione dello spazio virtuale dei processi

M.2 Organizzazione dello spazio virtuale dei processi

1. Aree virtuali dei processi

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

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 alle ***caratteristiche di accesso alla memoria***, ad esempio in base ai permessi di accesso (eseguibile, solo lettura, lettura e scrittura); in questo modo è possibile evitare che il contenuto di un'area di memoria venga modificato 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 di base della paginazione 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 a**ree di memoria virtuale** o **VMA** (virtual memory area).

Ogni VMA è costituita da un ***numero intero*** di pagine virtuali ***consecutive*** e dotate di caratteristiche di accesso alla memoria ***omogenee***; pertanto un'area virtuale può essere caratterizzata da una coppia di indirizzi virtuali che ne definiscono l'inizio e la fine: $NPV_{iniziale}$, NPV_{finale}.

1.1 Tipi di aree virtuali

I tipi di VMA di un processo sono i seguenti (tra parentesi è indicata una lettera che verrà utilizzata come abbreviazione):

- **Codice (C)**: è l'area che contiene le istruzioni che costituiscono il programma da eseguire; quest'area contiene anche le costanti definite all'interno del codice (ad esempio, le stringhe da passare alle `printf`)
- **Costanti per rilocazione dinamica (K):** è un'area destinata a contenere dei parametri determinati in fase di link per il collegamento con le librerie dinamiche
- **Dati statici (S)**: è l'area destinata a contenere i dati inizializzati allocati per tutta la durata di un programma
- **Dati dinamici (D)**: 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`** (**Program break**) contenuta nel descrittore del processo – questa VMA contiene anche gli eventuali dati non inizializzati definiti nell'eseguibile
- **Pile dei Thread (T)**: sono le aree utilizzate per le pile dei thread; verranno trattate più avanti
- **Aree per Memory-Mapped files (M):** queste aree permettono di mappare un file su una porzione di memoria virtuale di un processo in modo che il file possa essere letto o scritto come se fosse un array di byte in memoria. Le aree di questo tipo sono utilizzate per diversi scopi, tra i quali in particolare:
  - *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. Le librerie dinamiche sono caricate mappando il loro file eseguibile su una o più aree virtuali di tipo M**.** Una caratteristica fondamentale delle librerie dinamiche è di poter essere condivise tra diversi programmi, che mappano lo stesso file della libreria su loro aree virtuali
  - *Memoria condivisa*: tramite la mappatura di un file su aree virtuali di diversi processi si ottiene la realizzazione di un meccanismo di condivisione della memoria tra tutti i processi che mappano tale file; in questo caso le aree potranno essere in lettura o lettura e scrittura.
- **Pila (P)**: è l'area di pila di modo U del processo, che contiene tutte le variabili locali delle funzioni di un programma C (o di un linguaggio equivalente)

Per ogni tipo C, K, S , D e P esiste una unica area virtuale, invece per i tipi M e T possono esistere zero o più aree. 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.

VMA ed Eseguibile

Esiste una ovvia corrispondenza tra le VMA di un programma in esecuzione e le componenti (dette ***segments***) di un file Eseguibile in formato ELF (il formato utilizzato in LINUX): le VMA C, K e S sono l'immagine dei corrispondenti segmenti del file, mentre le VMA D, T, M e P non esistono nell'eseguibile. L'unico aspetto che richiede un precisazione riguarda i dati non inizializzati, generalmente chiamati **BSS (Block Started by Symbol)**:

- **nell'eseguibile** tali dati sono logicamente dei dati statici che, non avendo un valore iniziale, possono essere rappresentati sinteticamente, associando al nome simbolico (etichetta) la quantità di spazio di memoria da utilizzare, invece
- **nello spazio virtuale del processo** sono considerati come una porzione allocata inizialmente dei dati dinamici e quindi appartengono alla VMA D, non a quella S, perché il loro comportamento dal punto di vista della gestione della memoria è simile a quello dei dati dinamici

In figura 1.1 è mostrata la tipica struttura di un processo LINUX durante l'esecuzione. L'ultima pagina è riservata per scopi particolari.

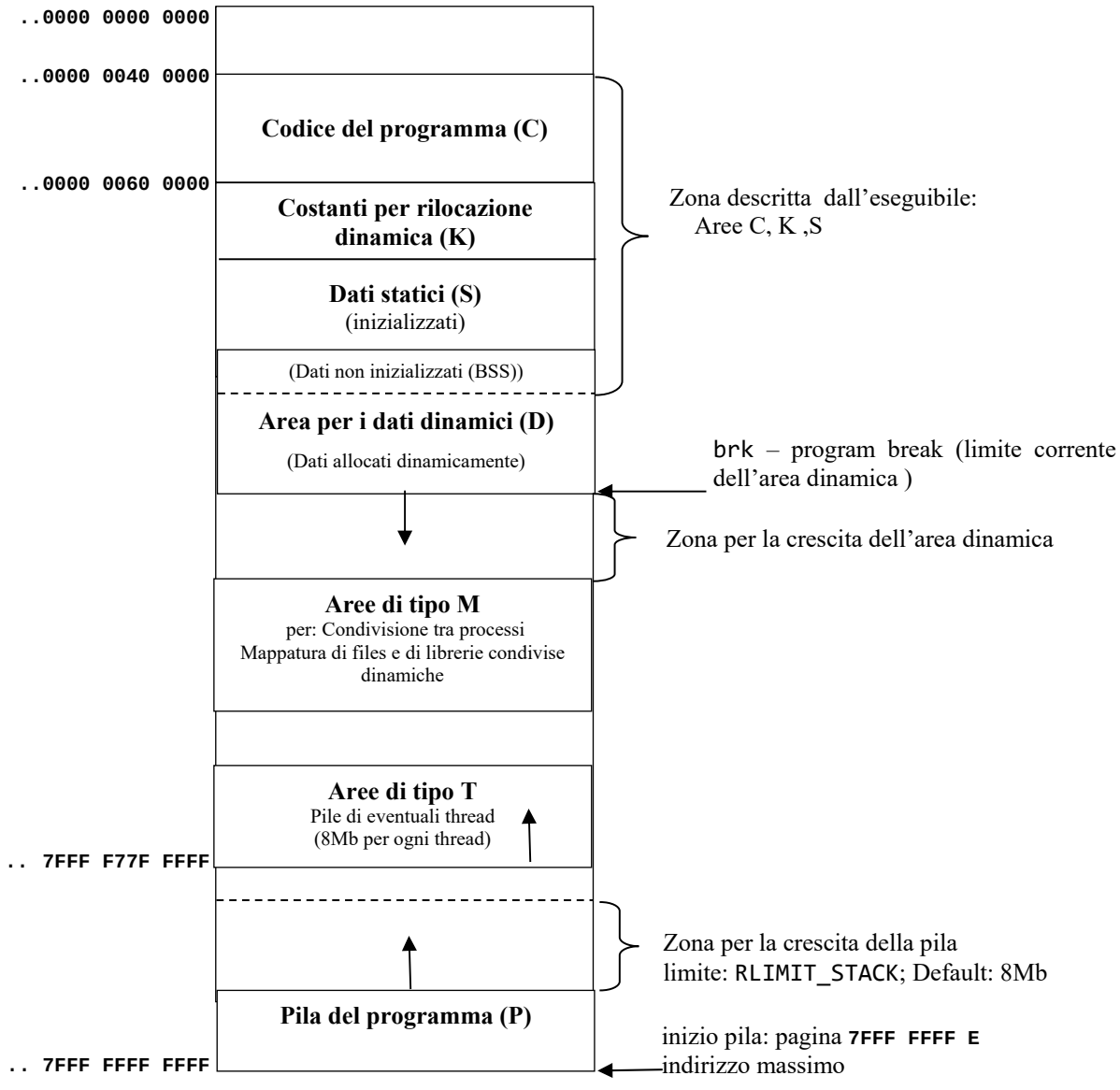


Figura 1.1 – Struttura di un programma in esecuzione –sono indicati 48 bit degli indirizzi

ATTENZIONE: a differenza del testo H-P, in Figura 1.1 e in alcune altre figure di questi appunti la memoria è rappresentata graficamente con l'indirizzo 0 in alto e l'indirizzo massimo in basso, cioè con indirizzi crescenti verso il basso. Tuttavia la terminologia di riferimento agli indirizzi, come indirizzi *alti* o *bassi*, aree di memoria *sopra* o *sotto* un'altra area di memoria, ecc… non cambiano riferimento: un indirizzo è detto più *alto* di un altro se il suo valore numerico è maggiore del valore dell'altro, un'area di memoria è *sopra* un'altra se i suoi indirizzi sono più grandi (cioè più alti), ecc…

1.2 Struttura dati per la rappresentazione delle Aree Virtuali

Ogni VMA in LINUX è definita da una variabile strutturata di tipo `vm_area_struct`.. In figura 1.2 sono rappresentati i campi principali di tale struttura.
I primi 3 campi hanno un significato ovvio:

- `vm_start` è l'indirizzo più basso dell'area
- `vm_end` è l'indirizzo del *primo byte successivo all'area* , quindi `vm_end` non appartiene all'area

Le strutture delle singole aree sono collegate tra loro in una lista (`*vm_next, *vm_prev`) ordinata per indirizzi crescenti.
Il campo `vm_flags` è utilizzato per caratterizzare le proprietà dell'area. Ogni bit di tale campo, detto flag, possiede un preciso significato; nella stessa figura 1.2 sono indicate le definizioni di alcune costanti numeriche che definiscono la posizione dei principali flag:

- VM_READ (R) – è permessa la lettura
- VM_WRITE (W) – è permessa la scrittura
- VM_EXEC (X) – l'area contiene codice eseguibile
- VM_SHARED (S) - le pagine dell'area sono condivise tra diversi processi
- VM_GROWSDOWN (G) - l'area può crescere automaticamente verso il basso (tipicamente usato per aree di pila)
- VM_DENYWRITE (D) - l'area si mappa su un file che non può essere scritto

Linux/include/linux/mm_types.h

```
211 struct vm_area_struct {
212         struct mm_struct * vm_mm;       /* La mm_struct del processo alla quale
                                                quest'area appartiene */
213         unsigned long vm_start;         /* indirizzo iniziale dell'area */
214         unsigned long vm_end;           /* indirizzo finale dell'area */
216
217         /* linked list of VM areas per task, sorted by address */
218         struct vm_area_struct *vm_next, *vm_prev;
219
220         unsigned long vm_flags;          /* flags – vedi sotto */
224           ...
253
254         /* Information about our backing store: */
255         unsigned long vm_pgoff;         /* Offset (within vm_file) in PAGE_SIZE
256                                            units*/
257         struct file * vm_file;          /* File we map to (can be NULL). */
              ...
266 };
```

Principali VMA Flags

```
       #define VM_READ         0x00000001
       #define VM_WRITE        0x00000002
       #define VM_EXEC         0x00000004
       #define VM_SHARED       0x00000008
       #define VM_GROWSDOWN    0x00000100
       #define VM_DENYWRITE    0x00000800
```

Figura 1.2

Aree mappate su file

Una VMA può essere mappata su un file detto "***backing store***"; in caso contrario l'area è detta anonima (ANONYMOUS). A questo scopo la struttura `vm_area_struct` contiene anche l'informazione (`struct file * vm_file;`) necessaria a individuare il file utilizzato come backing store, e la posizione (offset) all'interno del file stesso (`unsigned long vm_pgoff;`).

Come semplice esempio intuitivo della nozione di backing store si considerino le VMA C, K e S: a tutte queste aree viene associato come backing store il file eseguibile, con offset che indicano il punto in cui nell'eseguibile inizia il corrispondente segmento. La funzione del backing store verrà approfondita più avanti.

Esempio: mappa di un processo in esecuzione
In tabella 1.1 è riportata la **mappa di memoria** di un processo che esegue un semplicissimo programma costituito praticamente dal solo main() che non fa nulla. Tale mappa è stata ottenuta con il comando **cat /proc/NN/maps** (dove NN rappresenta il pid del processo) e poi inserita in una tabella per comodità di lettura. Ogni riga si riferisce a una diversa VMA.

Il significato delle colonne è il seguente:

- start-end: indirizzi virtuali di inizio e fine della VMA
- perm: permessi di accesso all'area – r w x hanno il significato già visto, p indica che l'area non è condivisa (SHARED), ma privata (PRIVATE); il significato di questa distinzione verrà spiegato più avanti
- offset, dev, i-node: definiscono la posizione fisica dell'area su file; come vedremo nel capitolo sul file system, un i-node individua un file fisico all'interno di un dispositivo (device) caratterizzato da una coppia di numeri detti major e minor; offset è la distanza dall'inizio del file
- file name è il path-name del file

start-end	perm	offset	device	i-node	file name
00400000-00401000	r-xp	000000	08:01	394275	…/user.exe
00600000-00601000	r--p	000000	08:01	394275	…/user.exe
00601000-00602000	rw-p	001000	08:01	394275	…/user.exe
7ffff7a1c000-7ffff7bd0000	r-xp	000000	08:01	271666	/lib/x86_64-linux-gnu/libc-2.15.so
7ffff7bd0000-7ffff7dcf000	---p	1b4000	08:01	271666	/lib/x86_64-linux-gnu/libc-2.15.so
7ffff7dcf000-7ffff7dd3000	r--p	1b3000	08:01	271666	/lib/x86_64-linux-gnu/libc-2.15.so
7ffff7dd3000-7ffff7dd5000	rw-p	1b7000	08:01	271666	/lib/x86_64-linux-gnu/libc-2.15.so
...					
(altre aree M)					
...					
7fffffde000-7fffffff000	rw-p				[stack]

Tabella 1.1

Possiamo osservare che

- le prime 3 righe si riferiscono alle aree C, K e S dell'eseguibile e si mappano su un file …/user.exe
- le successive 4 righe si riferiscono a 4 aree di tipo M con diversi permessi di accesso mappate sulla libreria libc
- altre aree di tipo M sono state omesse
- infine l'area di pila è stata allocata con una dimensione iniziale di 34 pagine (144 Kbyte) – l'area di pila è anonima, quindi non ha un file associato; l'indicazione [stack] è solo un commento aggiunto dal comando maps

Tutti i file coinvolti risiedono sullo stesso dispositivo 08:01, ma l'eseguibile del programma e quello della libreria libc sono diversi ed hanno quindi diverso i-node.

Al momento dell'`exec` di un programma eseguibile LINUX costruisce la struttura delle aree virtuali del processo in base alla struttura definita dall'eseguibile. In generale le aree create sono quelle di tipo C, K, S, P e talvolta D (in presenza di dati statici non inizializzati) – inoltre sono presenti molte aree di tipo M per l'accesso alle librerie dinamiche.

1.3 Algoritmo base di gestione dei Page Fault

Ogni volta che la MMU genera un interrupt di Page Fault su un indirizzo virtuale appartenente ad una pagina virtuale NPV viene attivata una routine detta **Page Fault Handler (PFH)**.

Il PFH esegue concettualmente il seguente algoritmo:

if (NPV non appartiene alla memoria virtuale del processo)
    il processo viene abortito e viene segnalato un "**Segmentation Fault**"
else if (NPV appartiene alla memoria virtuale del processo ma l'accesso non è legittimo
    perché viola le protezioni)
    il processo viene abortito e viene segnalato un "**Segmentation Fault**"
else if (l'accesso è legittimo, ma NPV non è residente in memoria)
    invoca la routine che deve caricare in memoria la pagina virtuale NPV dal file di backing store

Queste azioni corrispondono al normale meccanismo di sostituzione di pagine nel funzionamento di un programma; come vedremo, l'algoritmo dovrà essere arricchito con alcune condizioni ulteriori per gestire alcune situazioni particolari.

1.4 Regole generali di LINUX e implementazione specifica

Nell'analisi del funzionamento della memoria di LINUX è necessario talvolta distinguere tra:

- **regole (o modello) generale** del sistema: è il comportamento che LINUX definisce e garantisce ufficialmente
- **implementazione specifica**: dato che le regole generali non specificano necessariamente tutti gli aspetti del comportamento del sistema, in alcuni casi faremo riferimento alla specifica implementazione utilizzata per gli esempi di questo testo; ad esempio, da un punto di vista generale non è possibile sapere quale processo prosegue dopo una `fork`, ma nella implementazione specifica abbiamo constatato che prosegue il processo padre

1.5 Address Space Layout Randomization

Prima di analizzare ulteriormente questi aspetti è importante precisare che l'analisi degli indirizzi virtuali appena svolta è stata ottenuta **disabilitando la ASLR (Address Space Layout Randomization)**.
La ASLR è una tecnica di modificazione casuale degli indirizzi applicata da LINUX che consiste nel rendere casuale l'indirizzo delle funzioni di libreria e delle più importanti aree di memoria. In questo modo un attacco informatico che cerca di eseguire codice malevolo su un computer è costretto a cercare gli indirizzi del codice e dei dati che gli servono prima di poterli usare, provocando una serie di crash del programma vettore (infettato o apertamente malevolo).
La ASLR rende la interpretazione degli indirizzi molto più complessa, perciò *tutti gli esempi ed esercizi mostrati nel seguito sono eseguiti con ASLR disabilitata***.** Si tenga però presente che normalmente in una configurazione standard di LINUX la ASLR è invece abilitata.

2. Tabella delle Pagine (TP) e Aree Virtuali

A causa della dimensione potenzialmente molto grande della TP, la sua gestione è complessa e basata sull'esistenza di opportuni meccanismi Hardware che verranno analizzati in un successivo capitolo. Per ora ci limitiamo ad enunciare le seguenti regole:

- in ogni istante esistono esclusivamente le pagine virtuali NPV appartenenti alle VMA del processo
- la TP contiene sempre le righe sufficienti a rappresentare tali NPV
- una NPV può essere residente in memoria fisica, cioè essere mappata su un NPF, oppure non essere residente; nella riga corrispondente della TP l'esistenza dell'allocazione fisica è rappresentata dal flag P (presente)

Il Sistema Operativo gestisce l'evoluzione delle VMA e della TP di un processo in base agli eventi che si verificano, garantendo che le regole appena indicate siano rispettate.
Nel seguito analizziamo come tale evoluzione è realizzata per le 5 aree fondamentali C, K, S, D e P.

Per analizzare oltre alle aree virtuali anche le relative porzioni di TP utilizziamo un Modulo opportuno inserito nel SO. L'invocazione del modulo è realizzata dalla macro **VIRTUAL_AREAS**. L'esecuzione di tale macro, diversamente dal comando di Shell visto in precedenza, produce gli NPV e NPF depurati dalla porzione di offset. e le porzioni di TP relative alle VMA presenti.

2.1 Le aree definite dall'eseguibile del programma: C, K, S

In figura 3.1 (Esempio 1) è mostrata la struttura delle VMA fondamentali e le porzioni di TP relative a tali VMA di un processo che esegue un programma con le seguenti caratteristiche:

- il codice occupa una sola pagina
- esiste una pagina di costanti di rilocazione (K)
- esiste una pagina di dati statici
- il programma non ha allocato memoria dinamica
- il programma ha utilizzato solo 2 pagine di pila

I flags presenti nelle righe di TP non sono identici a quelli presenti nella vm_area_struct:

- P – indica che la pagina è presente, quindi una riga con P=0 è una riga di TP relativa a una NPV non allocata fisicamente
- X, R, W indicano i permessi di lettura, scrittura ed esecuzione già visti

Come si vede, LINUX ha creato 34 pagine virtuali di pila, anche se solo 2 sono effettivamente utilizzate. La pagina iniziale della VMA di pila (NPV `7FFFFFF`**`DD`** - pagina di growsdown) non è mai allocata fisicamente e viene utilizzata al momento della crescita virtuale della pila, come vedremo più avanti. Il comando `mmap` utilizzato per generare la Tabella 1.1 ha escluso questa pagina nell'indicare l'indirizzo iniziale dell'area e per questo motivo lo start address è indicato come 7ffffff**de**000.

Analizziamo ora come LINUX si comporta nel caricamento di un programma con più pagine di codice e dati. La tecnica applicata è il ***demand paging***, cioè il caricamento delle pagine in base ai page fault che vengono generati.

```
[11299.304331]          MAPPA DELLE AREE VIRTUALI:
[11299.304331]
[11299.304334] START: 000000400   END: 000000401  SIZE:   1  FLAGS: X,  , D
[11299.304335] START: 000000600   END: 000000601  SIZE:   1  FLAGS: R,  , D
[11299.304336] START: 000000601   END: 000000602  SIZE:   1  FLAGS: W,  , D
...
[11299.304351] START: 7FFFFFFDD   END: 7FFFFFFFF  SIZE:  34  FLAGS: W, G,
[11299.304351]
[11299.304351]         PAGE TABLE DELLE AREE VIRTUALI:
[11299.304352]         (  NPV  ::     NPF      ::   FLAGS  )
[11299.304354] VMA start address  000000400000 ======== size: 1
[11299.304356]       000000400 :: 00008875C    ::    P,X
[11299.304358] VMA start address  000000600000 ======== size: 1
[11299.304359]       000000600 :: 0000841EF    ::    P,R
[11299.304360] VMA start address  000000601000 ======== size: 1
[11299.304361]       000000601 :: 000087753    ::    P,W
[11299.304363] VMA start address  7FFFFFFDD000 ======== size: 34
[11299.304365]       7FFFFFFDD :: 000000000    ::     ,
...
[11299.304393]       7FFFFFFFD :: 00009D04B    ::    P,W
[11299.304394]       7FFFFFFFE :: 0000989A0    ::    P,W
```

Figura 3.1 (Esempio 1)

a) Allocazione delle pagine di codice (Esempio 2)
Vediamo come questo funziona in concreto. A questo scopo eseguiamo il programma di figura 3.2

Si noti l'inclusione del file "long_code.c", che contiene la funzione long_code() , che occupa circa 3 pagine. Nel codice del main c'è l'invocazione di tale funzione, ma per il momento è commentata, in modo che non venga eseguita.
L'esecuzione del programma produce le seguenti stampe da printf:

NPV of main 000000000404
NPV of printf 000000000400

che ci dicono che l'istruzione iniziale del programma, o ***entry point***, (contenuto in main) è nella NPV 404 mentre la funzione di sistema printf è nel NPV 400 (le funzioni di sistema sono evidentemente collegate prima del programma utente)
La porzione di TP relativa al codice, stampata dalla macro VIRTUAL_AREAS, mostrata in figura, ci indica il seguente comportamento di LINUX:

1. ha definito una VMA di 5 pagine per il codice, a partire da 400
2. poi ha allocato fisicamente la NPV 404 perché contiene l'entry point
3. poi ha allocato 400, perché contiene il codice delle funzioni di libreria (printf e quelle necessarie alla macro)

```
#include <stdio.h>
#include "long_code.c"  //questa funzione occupa circa 3 pagine di codice
#define PAGE_SIZE 4096
long unsigned pointer;
int main() {
        pointer = &main;
        printf("NPV of main  %12.12lx \n", pointer/ PAGE_SIZE);
        pointer = &printf;
        printf("NPV of printf  %12.12lx \n", pointer/ PAGE_SIZE);
        //long_code(); commentato nella prima prova
        VIRTUAL_AREAS;
        return;
}
```

a) codice del programma

```
[12051.688104] VMA start address  000000400000 ======== size: 5
[12051.688105]       000000400 :: 00007C9E9    ::    P,X
[12051.688106]       000000401 :: 000000000    ::     ,
[12051.688107]       000000402 :: 000000000    ::     ,
[12051.688108]       000000403 :: 000000000    ::     ,
[12051.688109]       000000404 :: 0000875A1    ::    P,X
```

b) TP del codice

```
[11895.930276] VMA start address  000000400000 ======== size: 5
[11895.930277]       000000400 :: 00008D3DE    ::    P,X
[11895.930278]       000000401 :: 00008353B    ::    P,X
[11895.930279]       000000402 :: 000087C22    ::    P,X
[11895.930280]       000000403 :: 000087C23    ::    P,X
[11895.930281]       000000404 :: 000087746    ::    P,X
```

c) TP del codice prodotta eseguendo la funzione long_code

Figura 3.2 (Esempio 2)

.

Si osservi che le NPV relative al codice presente in "long_code.c" sono presenti in VMA ma non allocate fisicamente.
Se ora si riesegue il programma attivando la funzione long_code (cioè decommentandola), si ottiene la TP mostrata nella sezione (c) della figura: tutte le NPV sono fisicamente allocate.

b)Aree dati (Esempio 3)
Anche le aree dati sono gestite in demand paging,
Consideriamo il seguente programma (figura 3.3), che definisce 3 stringhe di circa 16K ciascuna, poi accede solo alle stringhe 1 e 3.

```
#include "long_string.h" //contiene LONG_STRING, una stringa lunga circa 4 pagine
static char stringa1[] = LONG_STRING;
static char stringa2[] = LONG_STRING;
static char stringa3[] = LONG_STRING;
long unsigned pointer;
void access_strings( ){
        int i;
        for (i=0; i<sizeof(LONG_STRING); i++){
                stringa3[i] = stringa1[i];
                stringa1[i] = stringa3[i];
        }
}
int main(long argc, char * argv[], char * envp[]) {
        // … stampa gli NPV delle 3 stringhe …
        access_strings( );
        VIRTUAL_AREAS;
        return;
}
```

```
[11578.788335] VMA start address  000000601000 ======== size: 13
[11578.788336]       000000601 :: 0000B0C37    ::    P,W
[11578.788337]       000000602 :: 00009E5EC    ::    P,W
[11578.788338]       000000603 :: 0000AD4AA    ::    P,W
[11578.788339]       000000604 :: 0000828C2    ::    P,W
[11578.788340]       000000605 :: 00008C273    ::    P,W
[11578.788341]       000000606 :: 000000000    ::     ,
[11578.788342]       000000607 :: 000000000    ::     ,
[11578.788343]       000000608 :: 000000000    ::     ,
[11578.788344]       000000609 :: 00007EBAD    ::    P,W
[11578.788345]       00000060A :: 00008D2C3    ::    P,W
[11578.788346]       00000060B :: 0000888FF    ::    P,W
[11578.788347]       00000060C :: 00007DDA7    ::    P,W
[11578.788348]       00000060D :: 000080199    ::    P,W
```

Figura 3.3 – Esempio 3 – programma

Gli NPV delle 3 stringhe risultano i seguenti (la stringa è leggermante più lunga di 4 pagine):
NPV of stringa1 da 000000000601 a 000000000605
NPV of stringa2 da 000000000605 a 000000000609
NPV of stringa3 da 000000000609 a 00000000060D
e la TP relativa alla VMA dei dati statici mostra che stata dimensionata una VMA di 13 pagine, in grado di contenere tutte le stringhe, ma le pagine relative a stringa2 non sono fisicamente allocate.

3.2 Area di pila (Esempio 4)

L'area di pila è gestita in maniera leggermente diversa dalle precedenti, perché si tratta di un'area le cui dimensioni virtuali non sono staticamente prefissate, ma possono crescere in base alle esigenze fino a un certo limite massimo (`RLIMIT_STACK`).

Abbiamo già visto in Esempio 1 (figura 3.1) che LINUX crea una VMA di un certo numero di pagine (34 nella versione di LINUX utilizzata per questi esempi) già al momento della exec e poi alloca fisicamente le pagine di pila quando vengono richieste.

Per osservare la crescita della VMA di pila eseguiamo il seguente programma (Figura 3.4 – Esempio 4), che invoca una funzione ricorsiva con una variabile locale che occupa una pagina intera. Il numero di invocazioni ricorsive è determinato dalla costante STACK_ITERATION ed è posto al valore 35. Quindi ci aspettiamo che questo programma richieda 35 pagine di pila in più rispetto all'esempio 1.

```
#define PAGE_SIZE (1024*4)
#define STACK_ITERATION 35
int alloc_stack(int iterations){
        char local[PAGE_SIZE];
        if (iterations > 0){
                return alloc_stack(iterations-1);
        }
        else return 0;
}
int main(long argc, char * argv[], char * envp[]) {
        alloc_stack(STACK_ITERATION);
        VIRTUAL_AREAS;
        return;
}
```

```
[12558.634402] VMA start address  7FFFFFFDA000 ======== size: 37
[12558.634403]       7FFFFFFDA :: 000000000    ::     ,          36
[12558.634404]       7FFFFFFDB :: 00008B9C6    ::    P,W         35
[12558.634405]       7FFFFFFDC :: 00008C09B    ::    P,W         34
[12558.634406]       7FFFFFFDD :: 00008C09A    ::    P,W         33
[12558.634406]       7FFFFFFDE :: 000088661    ::    P,W         32
[12558.634407]       7FFFFFFDF :: 000083B6D    ::    P,W         31
[12558.634408]       7FFFFFFE0 :: 00007DBBE    ::    P,W         30
[12558.634409]       7FFFFFFE1 :: 00007EBD1    ::    P,W         29
[12558.634410]       7FFFFFFE2 :: 00008CEB3    ::    P,W         28
...
[12558.634430]       7FFFFFFFC :: 00007B9BC    ::    P,W         2
[12558.634431]       7FFFFFFFD :: 00009BDE4    ::    P,W         1
[12558.634432]       7FFFFFFFE :: 00009F363    ::    P,W         0
```

Figura 3.4a – crescita ordinata della pila

In effetti, la porzione di TP relativa alla pila prodotta dalla macro VIRTUAL_AREAS, mostrata nella stessa figura, permette di osservare che la VMA è cresciuta automaticamente in base alle esigenze ed è dimensionata a 37 pagine, di cui 36 sono affettivamente allocate fisicamente e l'ultima è non allocata (pagina di growsdown). Si osservi inoltre che la VMA di pila e le relative pagine non vengono deallocate anche quando la pila decresce; infatti nel programma la macro è posizionata dopo che la funzione che consuma molta pila è terminata. In sostanza, esiste un meccanismo di crescita automatica della allocazione fisica della pila, ma non di decrescita – quando la pila decresce le pagine rimangono allocate (ma verranno sovrascritte da una successiva ricrescita della pila).

Il comportamento osservato non sorprende ed è coerente con l'usuale modello di memoria dei programmi in linguaggio C. Tuttavia è importante tenere presente che in realtà la Gestione della Memoria a livello del SO non implementa la nozione di pila, ma si limita alla gestione di un'Area Virtuale con accesso RW e crescita automatica (indicata dal flag Growsdown); i meccanismi legati all'esistenza di uno Stack Pointer sono realizzati dal Compilatore e dal sistema Run Time del linguaggio di programmazione utilizzato.
Per capire il senso di quest'ultima affermazione consideriamo alcuni esempi che utilizzano i puntatori C in modo bizzarro e non conforme alle regole di buona programmazione.

Esempio 4 – variazione 1
Il programma di figura 3.4b accede la pila in maniera casuale invece che con una crescita ordinata. Il programma ha utilizzato 2 pagine di pila iniziali. Si noti che la variabile address è un puntatore a long, quindi il successivo incremento di 512*32 è di 32 pagine. Il primo accesso a pila è quindi alla 33-esime pagina della pila (NPV = **7FFFFFDE**) e il secondo accesso è alla quinta pagina di pila (NPV = 7FFFFFFA), come confermato dalla porzione di PT; questo conferma che l'area di pila è accessibile in maniera casuale.

```
int main(long argc, char * argv[], char * envp[]) {
      unsigned long int * address;
      address = address - 512 * 32; //primo accesso a pila
      *address = 1;
      address = address + 512 * 28; //secondo accesso a pila
      *address = 1;
      VIRTUAL_AREAS;
      return;
}

[13181.957140] VMA start address  7FFFFFFDD000 ======== size: 34
[13181.957142]       7FFFFFFDD :: 000000000    ::     ,
[13181.957143]       7FFFFFFDE :: 00008CF58    ::    P,W
[13181.957143]       7FFFFFFDF :: 000000000    ::     ,
...
[13181.957164]       7FFFFFFF9 :: 000000000    ::     ,
[13181.957165]       7FFFFFFFA :: 00008FD1E    ::    P,W
[13181.957165]       7FFFFFFFB :: 000000000    ::     ,
[13181.957166]       7FFFFFFFC :: 000000000    ::     ,
[13181.957167]       7FFFFFFFD :: 0000833BF    ::    P,W
[13181.957168]       7FFFFFFFE :: 00008705B    ::    P,W
```

Figura 3.4b - accesso casuale all'area di pila

Esempio 4 – variazione 2
Il programma di figura 3.4c esegue un accesso a pagina come il precedente, poi due ulteriori accessi a pagine singole. come possiamo vedere dalla porzione di PT, la VMA di pila è cresciuta passando da 34 a 36 pagine in modo da permettere l'allocazione fisica delle 2 pagine richieste.

```
int main(long argc, char * argv[], char * envp[]) {
      unsigned long int * address;
      address = address - 512 * 32;
      *address = 1;
      address = address - 512;
      *address = 1;
      address = address - 512;
      *address = 1;
      VIRTUAL_AREAS
      return;
}
[13539.427898] VMA start address  7FFFFFFDB000 ======== size: 36
[13539.427899]       7FFFFFFDB :: 000000000    ::     ,
[13539.427900]       7FFFFFFDC :: 000087026    ::    P,W
[13539.427901]       7FFFFFFDD :: 00009F369    ::    P,W
[13539.427902]       7FFFFFFDE :: 0000873E7    ::    P,W

[13539.427925]       7FFFFFFFC :: 000000000    ::     ,
[13539.427926]       7FFFFFFFD :: 0000871BA    ::    P,W
[13539.427927]       7FFFFFFFE :: 000084249    ::    P,W
```

Figura 3.4c

Esempio 4 – variazione 3
Il programma di figura 3.4d tenta di accedere a una pagina di pila che richiederebbe la crescita della VMA di 2 unità. Il risultato dell'esecuzione è che il processo viene abortito con segnalazione di "**`Segmentation fault`**"; questa è la segnalazione standard fornita da LINUX quando un programma viene abortito per una violazione nell'accesso a memoria.

```
int main(long argc, char * argv[], char * envp[]) {
      unsigned long int * address;
      address = address - 512 * 32;
      *address = 1;
      address = address - 512 * 2;
      *address = 1;
      VIRTUAL_AREAS
      return;
}

>Segmentation fault
```

Figura 3.4d

Riassunto del modello di gestione della VMA di pila
In base agli esperimenti svolti possiamo riassumere il comportamento della gestione della VMA di pila con le seguenti regole:

- al momento dell'exec la VMA di pila viene creata con un certo numero di NPV iniziali non allocate fisicamente eccetto quelle utilizzate immediatamente (tipicamente la prima o le prime due)
- tale area iniziale può essere acceduta in qualsiasi posizione

- ogni accesso a NPV non allocate fisicamente ne causa l'allocazione
- il tentativo di accedere una NPV posta subito sotto l'area esistente (pagina di growsdown) produce una crescita automatica dell'area
- il tentativo di accedere pagine diverse da quelle esistenti o quella di growsdown produce un Segmentation fault

L'algoritmo di gestione dei Page Fault di una pagina virtuale NPV visto prima deve essere quindi arricchito con la gestione della crescita automatica della pila nel modo seguente (modifiche in corsivo):

if (NPV non appartiene alla memoria virtuale del processo)
    il processo viene abortito e viene segnalato un "**Segmentation Fault**";
else if (NPV appartiene alla memoria virtuale del processo ma l'accesso non è legittimo perché viola le protezioni)
    il processo viene abortito e viene segnalato un "**Segmentation Fault**";
else if (l'accesso è legittimo, ma NPV non è allocata in memoria) {
    if (la NPV è la start address di una VMA che possiede il flag Growsdown) {
        aggiungi una nuova pagina virtuale NPV-1 all'area virtuale,
        che diventa la nuova start address della VMA;
    }
    invoca la routine che deve caricare in memoria la pagina virtuale NPV;
}

2.3 Crescita delle aree dinamiche – servizio brk.

Prima di poter allocare fisicamente una NPV di area dinamica è necessario richiedere la crescita dell'area stessa.

In linguaggio C la funzione malloc( ) svolge ambedue i compiti contemporaneamente, ma a livello del Sistema Operativo le 2 operazioni sono distinte:

- la crescita della VMA di tipo D è ottenuta tramite un servizio di sistema brk( ) o sbrk( )
- l'allocazione fisica delle pagine è svolta dal Sistema Operativo quando una pagina viene effettivamente scritta

brk( ) e *sbrk( )* sono due funzioni di libreria che invocano in forma diversa lo stesso servizio:

int brk(void * end_data_segment) – assegna il valore end_data_segment al campo end della VMA dei dati dinamici; restituisce 0 se ha successo, -1 in caso di errore

void *sbrk(intptr_t increment) - incrementa l'area dati dinamici del valore incremento e restituisce un puntatore alla posizione iniziale della nuova area.

sbrk(0) restituisce il valore corrente della cima dell'area dinamica.

Esempio 5
Consideriamo il seguente programma (figura 3.5a) che incrementa la VMA Dati dinamici di 3 pagine.
La mappa delle aree virtuali e la porzione di TP relativa alla VMA dati dinamici mostrano l'esistenza della nuova area di dimensione 3 e che nessuna pagina è fisicamente allocata.

```
#define MAX_PAGES 3
#define PAGE_SIZE 1024*4
int main(long argc, char * argv[], char * envp[]) {
        sbrk(PAGE_SIZE * MAX_PAGES);
        VIRTUAL_AREAS
        return;
}

[14101.714207]        MAPPA DELLE AREE VIRTUALI:
[14101.714207]
[14101.714209] START: 000000400   END: 000000401  SIZE:   1  FLAGS: X,  , D
[14101.714211] START: 000000600   END: 000000601  SIZE:   1  FLAGS: R,  , D
[14101.714212] START: 000000601   END: 000000602  SIZE:   1  FLAGS: W,  , D
[14101.714213] START: 000000602   END: 000000605  SIZE:   3  FLAGS: W,  ,
...
[14101.714224] START: 7FFFFFFDD   END: 7FFFFFFFF  SIZE:  34  FLAGS: W, G,
[14101.714225]
[14101.714225]       PAGE TABLE DELLE AREE VIRTUALI:
[14101.714226]       ( NPV ::    NPF    ::  FLAGS  )
[14101.714235] VMA start address  000000602000 ======== size: 3
[14101.714236]      000000602 :: 000000000   ::    ,
[14101.714236]      000000603 :: 000000000   ::    ,
[14101.714237]      000000604 :: 000000000   ::    ,
```

Figura 3.5a – Esempio 5a - programma e TP

Se aggiungiamo al main precedente una scrittura nella prima delle 3 pagine (programma di figura 3.5b) otteniamo una TP che mostra che solo la pagina acceduta è stata allocata fisicamente.

```
int main(long argc, char * argv[], char * envp[]) {
        unsigned long * brk;
        brk = sbrk(PAGE_SIZE * MAX_PAGES);
        *brk = 1;
        VIRTUAL_AREAS
        return;
}

[13897.637399] VMA start address  000000602000 ======== size: 3
[13897.637399]      000000602 :: 00009D049   ::   P,W
[13897.637399]      000000603 :: 000000000   ::    ,
[13897.637400]      000000604 :: 000000000   ::    ,
```

Figura 3.5b – Esempio 5b - programma e TP

4. Gestione della memoria virtuale nella creazione dei processi

4.1 Gestione nel caso di fork

La creazione di un nuovo processo implicherebbe la creazione di tutta la struttura di memoria del 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.
LINUX opera come se il nuovo processo condividesse tutta la memoria con il padre. Questo vale finché 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 meccanismo generale detto ***copy on write***.

Copy on write (COW) e conteggio degli utilizzatori di una pagina fisica
Il meccanismo COW è utilizzato nella realizzazione della fork e, come vedremo più avanti, in molte altre situazioni.
Il COW serve a evitare di duplicare pagine fisiche condivise tra processi se non è necessario; la duplicazione diventa necessaria solo quando un processo scrive sulla pagina fisica.
L'idea base del COW è la seguente:

1. al momento della creazione della situazione di condivisione attribuisce una abilitazione in sola lettura (R) alle pagine condivise, anche quelle appartenenti ad aree virtuali scrivibili, di ambedue i processi
2. quando si verifica un page fault per violazione di protezione in scrittura, il Page Fault Handler (PFH) verifica se la pagina protetta R appartiene a una VMA scrivibile, e in tal caso **se necessario** la duplica e le cambia la protezione in W, rendendo la scrittura possibile.

La necessità di duplicare dipende dal fatto che la pagina sia ancora condivisa oppure no. Infatti, supponiamo che una pagina fisica Px sia condivisa da 2 pagine virtuali V1 e V2, ambedue con protezione R in base al punto 1; supponiamo che un primo processo tenti di scrivere V1: Px viene duplicata e la nuova pagina fisica Py viene destinata a contenere V1 con protezione W, ma la pagina V2 rimane allocata in Px con protezione R. Quando il secondo processo tenterà di scrivere in V2 si verificherà di nuovo un page fault, ma questa volta non è più necessario duplicare la pagina fisica, perché non è più condivisa – è sufficiente cambiare la protezione da R a W.
Per permettere al PFH di sapere se una pagina richiede duplicazione oppure no ad ogni pagina fisica è attribuito, in una apposita struttura dati detta **descrittore di pagina** (`page_descriptor`), un contatore `ref_count` che indica il numero di utilizzatori della pagina fisica (in realtà il meccanismo di LINUX è un po' più involuto):

- se la pagina fisica è libera, `ref_count` restituisce 0
- se la pagina fisica ha N utilizzatori, `ref_count` restituisce N

Si tenga presente che, come vedremo più avanti, gli utilizzatori di una pagina fisica non sono esclusivamente i processi; una pagina fisica può mappare una pagina di un file e questo essere considerato un utilizzo da conteggiare.

Utilizzando `ref_count` la funzione del PFH descritta sopra al passo 2 si può dettagliare nel modo seguente, ipotizzando che la pagina fisica che contiene il NPV che ha causato il page fault sia PFx:

```
if  (la violazione è causata da accesso in scrittura a pagina con protezione R di una
                                                                    VMA con protezione W){
            if (ref_count(di PFx) > 1)   {
                                    copia PFx in una pagina fisica libera (PFy);
                                    poni ref_count di PFy a 1;
                                    decrementa ref_count di PFx ;
                                    assegna NPV a PFy e scrivi in PFy
            }
            else abilita NPV in scrittura; //pagina utilizzata solo da questo processo
}
```

L'algoritmo complessivo del PFH deve quindi essere ulteriormente modificato per tener conto del meccanismo di COW, nel modo seguente:

```
if (NPV non appartiene alla memoria virtuale del processo)
        il processo viene abortito e viene segnalato un "Segmentation Fault";
else if (NPV è allocata in pagina PFx, ma l'accesso non è legittimo perché viola le protezioni) {
        if  (la violazione è causata da accesso in scrittura a pagina con protezione R di una
                                                                        VMA con protezione W){
                if (ref_count(di PFx) > 1)   {
                                        copia PFx in una pagina fisica libera (PFy);
                                        poni ref_count di PFy a 1;
                                        decrementa ref_count di PFx ;
                                        assegna NPV a PFy e scrivi in PFy
                }
                else abilita NPV in scrittura;  //pagina utilizzata solo da questo processo
        }
        else  il processo viene abortito e viene segnalato un "Segmentation Fault";
}
else if (l'accesso è legittimo, ma NPV non è allocata in memoria) {
        if  (la NPV è la start address di una VMA che possiede il flag Growsdown) {
                        aggiungi una nuova pagina virtuale NPV-1 all'area virtuale,
                        che diventa la nuova start address della VMA;
        }
        invoca la routine che deve caricare in memoria la pagina virtuale NPV;
}
```

Pertanto, subito dopo una fork le sole pagine che vengono duplicate fisicamente sono quelle scritte durante l'esecuzione del servizio di fork stesso; ad esempio, l'esecuzione di

```
fork( );
```

esegue una scrittura sulla pagina in cima alla pila (dove la fork deposita il proprio risultato).
Tuttavia, nel caso di fork il valore restituito dal servizio viene praticamente sempre assegnato a una variabile, quindi ci troviamo generalmente in presenza di un assegnamento del tipo

```
pid = fork( );
```

e questo implica anche una scrittura nella pagina contenente la variabile pid (che dipende da dove tale variabile è allocata).
Nell'implementazione di LINUX considerata da noi ***la pagina fisica originale è attribuita al processo figlio, mentre per il processo padre viene allocata una pagina fisica nuova***.

Per analizzare il comportamento utilizziamo il programma di figura 4.1 Questo programma utilizza le funzioni già viste in precedenza per allocare 3 pagine dinamiche, scrivendo nelle prime 2, e allocare 3 pagine di pila.

Successivamente il programma:

1. stampa le aree virtuali,
2. poi esegue una fork
3. nel processo figlio scrive la seconda pagina dinamica, poi stampa le aree virtuali
4. nel processo padre scrive la prima pagina dinamica, poi stampa le aree virtuali

Il valore di indirizzo del pid stampato dal programma è **7FFFFFFE104**; quindi il pid si trova nella prima pagina di pila.

```
int main(int argc, char * argv[], char * envp[]) {
      ...
      int pid;
      alloc_memory( ); //funzione che alloca 3 pagine e scrive nelle prime 2
      alloc_stack(3); //funzione che alloca 3 pagine di pila
      printf("indirizzo di pid %12lX \n ", &pid);
      VIRTUAL_AREAS;
      pid = fork( );
      if (pid == 0){
            *brk2 = 1; //scrive nella seconda pagina dinamica
            VIRTUAL_AREAS;
            return;
      }
      else {
            *brk1 = 1; //scrive nella prima pagina dinamica
            VIRTUAL_AREAS;
            wait( );
            return
      }
}
```

Figura 4.1

I risultati delle printf sono illustrati in figura 4.2 . In rosso sono evidenziati i valori R delle pagine che sono provvisoriamente condivise ma dovranno essere duplicate in scrittura; in verde i valori W delle pagine che sono state duplicate in seguito a sdoppiamento dovuto a una scrittura dopo la fork; infine in blu sono evidenziati i numeri delle pagine fisiche di pila duplicate durante la fork stessa.
Il risultato ci mostra il comportamento di dettaglio nel caso di scrittura su una pagina condivisa; analizziamo ad esempio la pagina scritta dal main tramite l'istruzione `*brk1 = 1:`

- nel processo iniziale abbiamo: `000000602 :: 00007BC7A :: P,W,`
- nel processo padre dopo la fork `000000602 :: 000087C69 :: P,W,` cioè la pagina fisica è cambiata e la protezione è W
- nel processo figlio `000000602 :: 00007BC7A :: P,R,` cioè la pagina è rimasta la stessa e la protezione è ancora R

Per quanto riguarda la pila, dato che la variabile pid è allocata nella stessa NPV in cui la fork restituisce il suo valore (si tratta del frame di attivazione del main) solo la pagina alla base della VMA di pila è stata modificata, perché la fork è eseguita nel main, non nella funzione alloc_stack.

Proviamo a variare questo comportamento definendo la variabile pid in alloc_stack e eseguendo la fork e la stampa della TP nel momento di massima allocazione della pila, come nel programma di figura 5.3; in tal caso le pagine duplicate e poste in W sono quelle in cima alla VMA di pila.

```
                PROCESSO INIZIALE - PROCESS: 7384, GROUP: 7384
[14551.161779]          MAPPA DELLE AREE VIRTUALI:
[14551.161783] START: 000000400   END: 000000401  SIZE:   1  FLAGS: X,  , D
[14551.161784] START: 000000600   END: 000000601  SIZE:   1  FLAGS: R,  , D
[14551.161785] START: 000000601   END: 000000602  SIZE:   1  FLAGS: W,  , D
[14551.161786] START: 000000602   END: 000000605  SIZE:   3  FLAGS: W,  ,
...
[14551.161799] START: 7FFFFFDD   END: 7FFFFFFF  SIZE:  34  FLAGS: W, G,
[14551.161799]         PAGE TABLE DELLE AREE VIRTUALI:
[14551.161802] VMA start address  000000400000 ======== size: 1
[14551.161803]       000000400 :: 0000A1BF8    ::    P,X
[14551.161805] VMA start address  000000600000 ======== size: 1
[14551.161806]       000000600 :: 00008875D    ::    P,R
[14551.161807] VMA start address  000000601000 ======== size: 1
[14551.161808]       000000601 :: 00009FBB7    ::    P,W
[14551.161809] VMA start address  000000602000 ======== size: 3
[14551.161810]       000000602 :: 00007BC7A    ::    P,W
[14551.161811]       000000603 :: 000087698    ::    P,W
[14551.161812]       000000604 :: 000000000    ::     ,
[14551.161814] VMA start address  7FFFFFFDD000 ======== size: 34
[14551.161837]       7FFFFFFF8 :: 000000000    ::     ,
[14551.161839]       7FFFFFFFB :: 0000795B6    ::    P,W
[14551.161840]       7FFFFFFFC :: 000083359    ::    P,W
[14551.161841]       7FFFFFFFD :: 000096EE8    ::    P,W
[14551.161842]       7FFFFFFFE :: 00008A03D    ::    P,W
       PROCESSO PADRE DOPO LA FORK:   PAGE TABLE DELLE AREE VIRTUALI
[14551.161939] VMA start address  000000400000 ======== size: 1
[14551.161940]       000000400 :: 0000A1BF8    ::    P,X
[14551.161942] VMA start address  000000600000 ======== size: 1
[14551.161943]       000000600 :: 00008875D    ::    P,R
[14551.161944] VMA start address  000000601000 ======== size: 1
[14551.161945]       000000601 :: 00009FBB7    ::    P,R
[14551.161946] VMA start address  000000602000 ======== size: 3
[14551.161947]       000000602 :: 000087C69    ::    P,W
[14551.161948]       000000603 :: 000087698    ::    P,R
[14551.161949]       000000604 :: 000000000    ::     ,
[14551.161950] VMA start address  7FFFFFFDD000 ======== size: 34
[14551.161952]       7FFFFFFDD :: 000000000    ::     ,
[14551.161976]       7FFFFFFFB :: 0000795B6    ::    P,R
[14551.161977]       7FFFFFFFC :: 000083359    ::    P,R
[14551.161978]       7FFFFFFFD :: 000096EE8    ::    P,R
[14551.161979]       7FFFFFFFE :: 000097F23    ::    P,W
[14551.162002]
       PROCESSO FIGLIO DOPO LA FORK: PAGE TABLE DELLE AREE VIRTUALI
[14551.162022] VMA start address  000000400000 ======== size: 1
[14551.162023]       000000400 :: 0000A1BF8    ::    P,X
[14551.162024] VMA start address  000000600000 ======== size: 1
[14551.162025]       000000600 :: 00008875D    ::    P,R
[14551.162026] VMA start address  000000601000 ======== size: 1
[14551.162027]       000000601 :: 00009FBB7    ::    P,R
[14551.162029] VMA start address  000000602000 ======== size: 3
[14551.162029]       000000602 :: 00007BC7A    ::    P,R
[14551.162030]       000000603 :: 00007B99E    ::    P,W
[14551.162031]       000000604 :: 000000000    ::     ,
[14551.162032] VMA start address  7FFFFFFDD000 ======== size: 34
[14551.162058]       7FFFFFFFB :: 0000795B6    ::    P,R
[14551.162059]       7FFFFFFFC :: 000083359    ::    P,R
[14551.162060]       7FFFFFFFD :: 000096EE8    ::    P,R
[14551.162061]       7FFFFFFFE :: 00008A03D    ::    P,W
```

Figura 4.2

```
int alloc_stack(int fp, unsigned long int * address, int iterations){
      int pid;
      char local[PAGE_SIZE -300];
      if (iterations > 0){
            return alloc_stack(fp, address, iterations-1);
      }
      else {
            pid = fork( );
            if (pid == 0){VIRTUAL_AREAS; return; }
            else { VIRTUAL_AREAS; wait( ); return; }
      }
}

int main(int argc, char * argv[], char * envp[]) {
      alloc_stack(fp, address, 3);
}
PROCESS: 3119 – PROCESSO PADRE
[ 1299.750245] VMA start address  7FFFFFDD000 ======== size: 34
...
[ 1299.750269]       7FFFFFF9 :: 000000000    ::     ,
[ 1299.750269]       7FFFFFFA :: 000073612    ::    P,W
[ 1299.750270]       7FFFFFFB :: 0000798DE    ::    P,R
[ 1299.750271]       7FFFFFFC :: 00008BFBF    ::    P,R
[ 1299.750272]       7FFFFFFD :: 000076558    ::    P,R
[ 1299.750273]       7FFFFFFE :: 00006EFD0    ::    P,R

PROCESS: 3120 – PROCESSO FIGLIO
...
[ 1299.750332] VMA start address  7FFFFFDD000 ======== size: 34
[ 1299.750355]       7FFFFFF9 :: 000000000    ::     ,
[ 1299.750356]       7FFFFFFA :: 000073611    ::    P,W
[ 1299.750357]       7FFFFFFB :: 0000798DE    ::    P,R
[ 1299.750358]       7FFFFFFC :: 00008BFBF    ::    P,R
[ 1299.750359]       7FFFFFFD :: 000076558    ::    P,R
[ 1299.750359]       7FFFFFFE :: 00006EFD0    ::    P,R
```

Figura 4.3

4.2 Context Switch e Exit

L'esecuzione di un Context Switch non ha effetti diretti sulla memoria, ma causa uno svuotamento (**flush**) del TLB, perché rende tutte le pagine del processo corrente non utilizzabili nel prossimo futuro. Dato che alcune di queste pagine potrebbero avere il dirty bit a 1, questa informazione deve essere salvata; a questo scopo il descrittore di pagina fisica contiene anch'esso un dirty bit utilizzato per inserirvi il valore del dirty bit del TLB al momento del flush.

L'esecuzione della `exit` di un processo causa i seguenti effetti sulla gestione della memoria:

1. eliminazione della struttura delle VMA del processo
2. eliminazione della PT del processo
3. deallocazione delle NPV del processo con conseguente liberazione delle pagine fisiche occupate **solamente** da tali NPV, altrimenti la riduzione del loro ref_count
4. esecuzione di un context switch con flush del TLB

Nella terza azione è evidenziato il termine "solamente", perché questo significa che non vengono liberate le pagine fisiche ancora mappate su un file o condivise con un altro processo. Le implicazioni di questo aspetto verranno approfondite più avanti.

5 Creazione dei thread e aree di tipo T

5.1 Analisi sperimentale della creazione dei Thread

La gestione della memoria dei Thread di un processo segue una regola completamente diversa da quella per i figli di un processo: ***i processi leggeri che rappresentano dei Thread condividono la stessa struttura di aree virtuali e TP del processo padre (thread principale).***
Questa regola realizza l'assoluta condivisione della memoria tra il processo padre e i suoi thread; ogni modifica della memoria eseguita da un thread è visibile da tutti gli altri.

In figura 5.1 si mostra l'effetto sulle VMA e sulla TP della creazione di 2 Thread da parte di un processo. La figura mostra la struttura delle aree virtuali nei seguenti casi:

- processo padre, prima della creazione dei thread
- processo padre, dopo la creazione dei thread
- processi figli relativi al primo e secondo thread

In figura sono state omesse le VMA oltre il NPV `7FFFF77FF` , che sono identiche in tutti i processi

Osservando le righe evidenziate in figura

```
START: 7FFFF67FD   END: 7FFFF67FE  SIZE:   1  FLAGS: R,  ,
START: 7FFFF67FE   END: 7FFFF6FFE  SIZE:2048  FLAGS: W,  ,
START: 7FFFF6FFE   END: 7FFFF6FFF  SIZE:   1  FLAGS: R,  ,
START: 7FFFF6FFF   END: 7FFFF77FF  SIZE:2048  FLAGS: W,  ,
```

si constata che LINUX ha allocato un'area di tipo T di 2048 pagine, cioè esattamente 8 Mb, per ogni pila di thread, ed ha inserito delle aree di interposizione in sola lettura della dimensione di una pagina.

Gli inizi effettivi delle due pile sono agli indirizzi:

- `7FFFF77FF FFF` per il primo thread
- `7FFFF6FFE FFF` per il secondo thread

Dato che 8Mb è la dimensione massima prevista per le pile dei thread, non viene settato il flag Growsdown – l'area non può crescere ulteriormente. Le aree di interposizione permettono una (debole) protezione rispetto allo sconfinamento di una pila di thread.

Inoltre si osserva che il NPV `7FFFF77FF` è quello indicato in figura 1.1 come inizio delle aree dei thread, quindi *le aree dei thread sono allocate a partire da questo indirizzo verso indirizzi più bassi*.

Per mostrare la condivisione delle pagine fisiche è mostrata la TP relativa alla VMA S.
La stampa degli indirizzi di una variabile locale delle thread function, che risiedono quindi nelle NPV dei record di attivazione di tali funzioni ha prodotto i seguenti 2 valori: 7FFFF6FFCED8 e 7FFFF77FDED8, che, depurati degli offset, corrispondono ai due NPV seguenti: **7FFFF6FFC** e **7FFFF77FD** – si tratta evidentemente della seconda pagina di pila dei thread.

```
          PROCESS: 6547, GROUP: 6547 – PADRE prima di Creazione Thread
[ 6732.304055]           MAPPA DELLE AREE VIRTUALI:
[ 6732.304057] START: 000000400   END: 000000401  SIZE:   1  FLAGS: X,  , D
[ 6732.304058] START: 000000600   END: 000000601  SIZE:   1  FLAGS: R,  , D
[ 6732.304059] START: 000000601   END: 000000602  SIZE:   1  FLAGS: W,  , D
...
[ 6732.304077]          PAGE TABLE DELLA VMA di tipo S
[ 6732.304085]       000000601 :: 00007F3F4    ::    P,W

           PROCESS: 6547, GROUP: 6547 – PADRE dopo Creazione Thread
[ 6732.304178]           MAPPA DELLE AREE VIRTUALI:
[ 6732.304179] START: 000000400   END: 000000401  SIZE:   1  FLAGS: X,  , D
[ 6732.304180] START: 000000600   END: 000000601  SIZE:   1  FLAGS: R,  , D
[ 6732.304181] START: 000000601   END: 000000602  SIZE:   1  FLAGS: W,  , D
...
[ 6732.304183] START: 7FFFF67FD   END: 7FFFF67FE  SIZE:   1  FLAGS: R,  ,
[ 6732.304184] START: 7FFFF67FE   END: 7FFFF6FFE  SIZE:2048  FLAGS: W,  ,
[ 6732.304185] START: 7FFFF6FFE   END: 7FFFF6FFF  SIZE:   1  FLAGS: R,  ,
[ 6732.304186] START: 7FFFF6FFF   END: 7FFFF77FF  SIZE:2048  FLAGS: W,  ,
...
[ 6732.304203]          PAGE TABLE DELLA VMA di tipo S
[ 6732.304210]       000000601 :: 00007F3F4    ::    P,W

           PROCESS: 6549, GROUP: 6547 – Processo del secondo Thread
[ 6732.304365]           MAPPA DELLE AREE VIRTUALI:
[ 6732.304367] START: 000000400   END: 000000401  SIZE:   1  FLAGS: X,  , D
[ 6732.304368] START: 000000600   END: 000000601  SIZE:   1  FLAGS: R,  , D
[ 6732.304369] START: 000000601   END: 000000602  SIZE:   1  FLAGS: W,  , D
...
[ 6732.304371] START: 7FFFF67FD   END: 7FFFF67FE  SIZE:   1  FLAGS: R,  ,
[ 6732.304372] START: 7FFFF67FE   END: 7FFFF6FFE  SIZE:2048  FLAGS: W,  ,
[ 6732.304373] START: 7FFFF6FFE   END: 7FFFF6FFF  SIZE:   1  FLAGS: R,  ,
[ 6732.304374] START: 7FFFF6FFF   END: 7FFFF77FF  SIZE:2048  FLAGS: W,  ,
...
[ 6732.304391]          PAGE TABLE DELLA VMA di tipo S
[ 6732.304398]       000000601 :: 00007F3F4    ::    P,W

           PROCESS: 6548, GROUP: 6547 – Processo del primo Thread
[ 6732.307634]           MAPPA DELLE AREE VIRTUALI:
[ 6732.307636] START: 000000400   END: 000000401  SIZE:   1  FLAGS: X,  , D
[ 6732.307637] START: 000000600   END: 000000601  SIZE:   1  FLAGS: R,  , D
[ 6732.307638] START: 000000601   END: 000000602  SIZE:   1  FLAGS: W,  , D
...
[ 6732.307640] START: 7FFFF67FD   END: 7FFFF67FE  SIZE:   1  FLAGS: R,  ,
[ 6732.307641] START: 7FFFF67FE   END: 7FFFF6FFE  SIZE:2048  FLAGS: W,  ,
[ 6732.307642] START: 7FFFF6FFE   END: 7FFFF6FFF  SIZE:   1  FLAGS: R,  ,
[ 6732.307643] START: 7FFFF6FFF   END: 7FFFF77FF  SIZE:2048  FLAGS: W,  ,
...
[ 6732.307660]           PAGE TABLE DELLA VMA di tipo S
[ 6732.307668]       000000601 :: 00007F3F4    ::    P,W
```

Figura 5.1

6. Il meccanismo generale per la creazione di VMA e il servizio `mmap`

La realizzazione delle aree virtuali si basa su un meccanismo generale che può essere invocato direttamente dal Sistema Operativo (ad esempio, per creare l'area di codice o l'area di pila di un processo) oppure, tramite la System Call `mmap( )`, da un programma di sistema (ad esempio, il Dynamic Loader per creare le aree relative alle librerie dinamiche) oppure da un normale programma applicativo per vari scopi.
Le VMA sono classificate in base a 2 criteri:

1. Le VMA possono essere mappate su file oppure anonime (ANONYMOUS).
2. Inoltre possono essere di tipo SHARED oppure PRIVATE.

Tutte le combinazioni di queste 2 classificazioni sono supportate in Linux, ma noi non considereremo la combinazione SHARED| ANONYMOUS
Pertanto indicheremo i 3 tipi di aree considerate come

1. SHARED (e mappate su file)
2. PRIVATE (e mappate su file)
3. ANONYMOUS (implicitamente PRIVATE)

6.1 Aree mappate su file – sola lettura

Il concetto di mappatura di una VMA su un file è illustrato in figura 6.1, che mostra una VMA v1 appartenente a un processo P mappata su un file F a partire dalla seconda pagina (pagina 1). Il riferimento al file e lo spiazzamento in pagine rispetto all'inizio del file sono memorizzati nei 2 campi `vm_file` e `vm_pgoff` della struct `vm_area_struct`.
Si osservi che sia le pagine che costituiscono la VMA sia le corrispondenti pagine del file devono essere consecutive.
Un file in Linux è considerato una sequenza di byte (l'argomento dei file verrà trattato nei capitoli sul Filesystem); considerare un file diviso in pagine è solo un modo di trattare le posizioni dei byte. Ad esempio, la pagina 0 del file è la sequenza dei byte da 0 a 4095, la pagina 1 inizia col byte 4096, ecc…

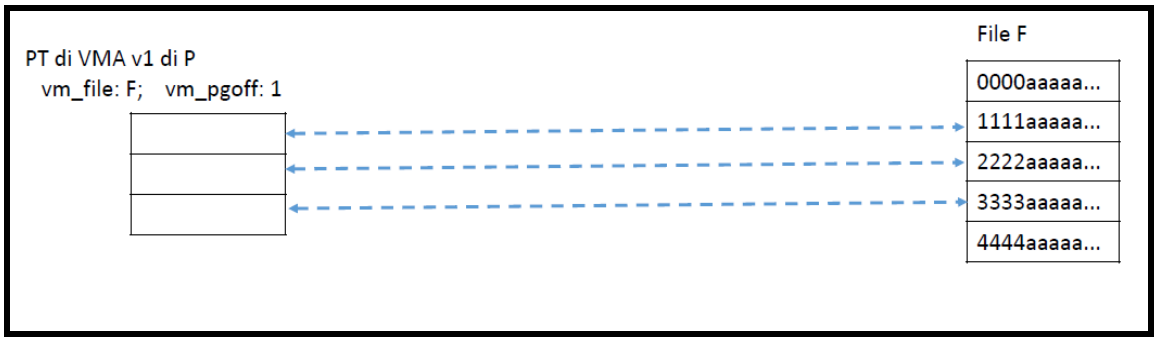


Figura 6.1

Per fare degli esempi di creazione di VMA utilizzando un programma applicativo è necessario utilizzare la funzione `mmap( )` ; conviene quindi analizzare anzitutto la definizione di questa funzione:

```
#include <sys/mmap.h>
void *mmap(void *addr, size_t length, int prot,
           int flags, int fd, off_t offset);
```

I parametri di `mmap` hanno il seguente significato:

- `addr`: permette di suggerire l'indirizzo virtuale iniziale dell'area – se non viene specificato il sistema sceglie un indirizzo autonomamente – il termine "suggerire" significa che LINUX sceglierà l'area il più vicino possibile all'indirizzo suggerito
- `length`: è la dimensione dell'area
- `prot` indica la protezione; può essere: PROT_EXEC, PROT_READ, PROT_WRITE
- `flags` indica numerose opzioni; le sole che noi consideriamo sono `MAP_SHARED` , `MAP_PRIVATE` e `MAP_ANONYMOUS` che corrispondono evidentemente alle 3 tipologie fondamentali di VMA citate sopra
- `fd` indica il descrittore del file da mappare
- `offset` indica la posizione iniziale dell'area rispetto al file (deve essere un multiplo della dimensione di pagina)

In caso di successo, `mmap` restituisce un puntatore all'area allocata.
Ad esempio, i seguenti statement C illustrano come realizzare la VMA v1 di figura 6.1

```
#define PAGESIZE 1024*4
char * base;
fd = open("./F", O_RDWR); //apre il file F
base = mmap(NULL, PAGESIZE*3, PROT_READ, MAP_SHARED, fd, PAGESIZE);
```

Si tenga presente che la differenza di comportamento tra aree di tipo SHARED e di tipo PRIVATE emerge solo nelle operazioni di scrittura; pertanto quanto verrà mostrato nei primi esempi in sola lettura con riferimento ad aree SHARED sarebbe valido anche se le aree fossero state dichiarate PRIVATE.

Esiste anche una System Call per eliminare il mapping di un certo intervallo di pagine virtuali che ha la seguente definizione:

```
int munmap(void *addr, size_t length);
```

dove `addr` deve puntare a un inizio di pagina. Questa funzione elimina la mappatura dell'intero intervallo virtuale che inizia in `addr` e si estende per `length` bytes (arrotondato al multiplo di pagina superiore, se non è un multiplo di pagina)

Una caratteristica di questo meccanismo consiste nel fatto che molte VMA di processi diversi possono mappare le stesse pagine di un file. Ad esempio, in figura 6.2 è mostrata una seconda VMA creata da un diverso processo Q sullo stesso file e con lo stesso spiazzamento.

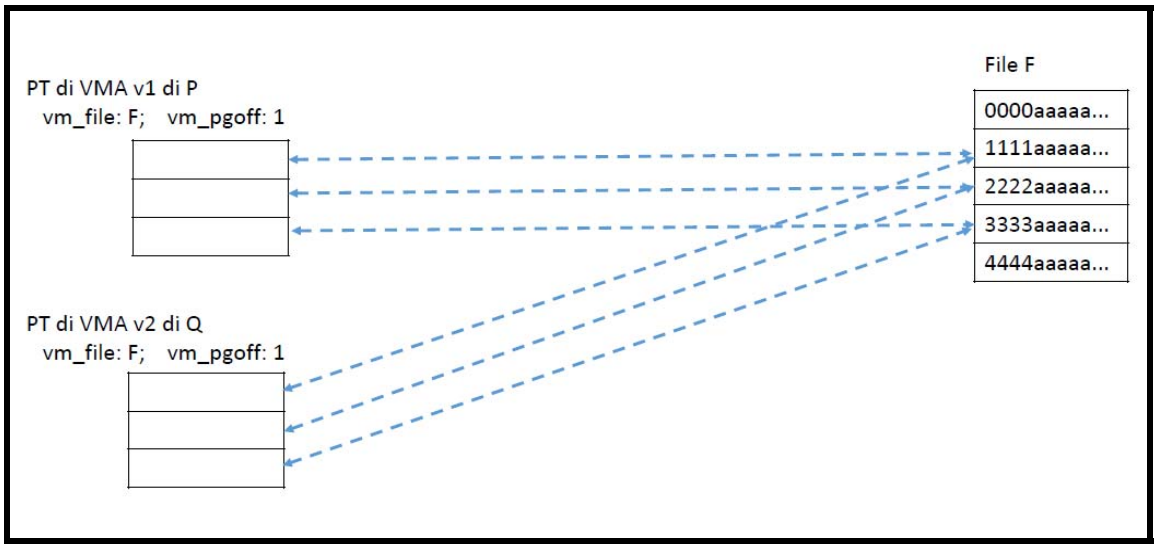


Figura 6.2

Esempio 6.1
In Figura 6.3a è mostrata la parte significativa di un programma che crea 2 processi figli; ognuno dei 2 figli crea una VMA mappata a partire dalla seconda pagina dello stesso file (`F`); nel processo 1 gli statement sono

```
fd = open("./F", O_RDWR);
base = mmap(mapaddress1, PAGESIZE*3, PROT_READ,
                                        MAP_SHARED, fd, PAGESIZE);
```

Nel processo 2 cambia solo l'indirizzo dell'area; le due VMA sono quindi poste a 2 diversi indirizzi virtuali (`mapaddress1 = 0x100000000; mapaddress2 = 0x200000000`)
In questo e tutti i seguenti esempi il file `F` ha il contenuto illustrato in figura 6.2: ogni sua pagina inizia con 4 caratteri che contengono il numero di pagina seguiti da caratteri 'a'.
Il primo programma stampa i primo 10 caratteri dell'area con i 2 statement

```
address = base;
for (i=0; i<10; i++){ printf("%c ", *address); address++; }
```

il secondo stampa i primi 10 caratteri dell'area e poi i primi 10 caratteri della terza pagina dell'area. Per eseguire questa seconda operazione è sufficiente spostare l'indirizzo iniziale della stampa con lo statement

```
address = base + 2*PAGESIZE;
```

Il risultato dell'esecuzione è mostrato in figura 6.3b – i due programmi stampano i caratteri presenti nel file nelle posizioni corrispondenti agli indirizzi nella VMA.

```
#define PAGESIZE 1024*4
int main(long argc, char * argv[], char * envp[]) {
       unsigned long mapaddress1 = 0x100000000;
       unsigned long mapaddress2 = 0x200000000;
       pid = fork();
       //figlio 1
       if (pid == 0){
              fd = open("./F", O_RDWR);
              base = mmap(mapaddress1, PAGESIZE*3, PROT_READ,
                                                MAP_SHARED, fd, PAGESIZE);
              //lettura
              printf("\n processo %d - lettura \n", getpid());
              address = base;
              for (i=0; i<10; i++){printf("%c ", *address);address++; }
              VIRTUAL_AREAS
       }
       ....
       //figlio 2
         if (pid == 0){
              fd = open("./F", O_RDWR);
              base = mmap(mapaddress2, PAGESIZE*3, PROT_READ,
                                                MAP_SHARED, fd, PAGESIZE);
              //lettura 1
              printf("\n processo %d - prima lettura \n", getpid());
              address = base;
              for (i=0; i<10; i++){ printf("%c ", *address); address++; }
              //lettura 2
              printf("\n processo %d - seconda lettura \n", getpid());
              address = base + 2*PAGESIZE;
              for (i=0; i<10; i++){ printf("%c ", *address);  address++; }
              VIRTUAL_AREAS
              }
              ...
}
```

a) codice del programma

```
processo 3358 - lettura
 1 1 1 1 a a a a a a
processo 3361 - prima lettura
1 1 1 1 a a a a a a
processo 3361 - seconda lettura
 3 3 3 3 a a a a a a
```

b) output del programma

Figura 6.3 – 2 processi creano due VMA mappate sullo stesso file

La creazione delle 2 VMA nei 2 processi è riscontrabile stampando le mappe virtuali dei 2 processi, come mostrato in figura 6.4, nella quale sono poste in evidenza le righe relative alle 2 VMA create. Si notino:

- gli indirizzi di base, diversi per le due aree
- la **s** al posto della **p** per indicare che l'area è SHARED,
- gli offset rispetto all'inizio del file (0x1000 = 2^{12}, cioè una pagina)
- il nome del file

00400000-00401000	r-xp	00000000	08:01	396306	..../user.exe
00601000-00602000	r--p	00001000	08:01	396306	.../user.exe
00602000-00603000	rw-p	00002000	08:01	396306	.../user.exe
100000000-100003000	**r--s**	**00001000**	**08:01**	**396260**	**.../F**
...					

a) Processo 1

00400000-00401000	r-xp	00000000	08:01	396306	.../user.exe
00601000-00602000	r—p	00001000	08:01	396306	.../user.exe
00602000-00603000	rw-p	00002000	08:01	396306	.../user.exe
200000000-200003000	**r--s**	**00001000**	**08:01**	**396260**	**.../F**
...					

b) Processo 2

Figura 6.4 – mappe (parziali) delle VMA dei 2 processi

Infine, in figura 6.5 sono mostrate le porzioni di TP relative alle 2 VMA create nei 2 processi. Le pagine hanno protezione R perché l'area alla quale appartengono ha questa protezione. Si noti che la pagina fisica della prima pagina virtuale è condivisa tra i 2 processi. Questo fatto dimostra che *quando una pagina è stata letta in memoria fisica da parte di un processo, l'altro processo non la rilegge dal disco ma accede alla **stessa** pagina fisica*. Questo comportamento richiede un meccanismo che permetta al secondo processo di accorgersi che la pagina cercata è già presente in memoria – questo meccanismo è descritto nella prossima sezione.

```
[ 2792.343262] VMA start address  000100000000 ======== size: 3
[ 2792.343263]      000100000 :: 000071816   ::   P,R
[ 2792.343264]      000100001 :: 000000000   ::    ,
[ 2792.343265]      000100002 :: 000000000   ::    ,
```

a) Processo 1

```
[ 2792.347357] VMA start address  000200000000 ======== size: 3
[ 2792.347358]      000200000 :: 000071816   ::   P,R
[ 2792.347359]      000200001 :: 000000000   ::    ,
[ 2792.347360]      000200002 :: 000071818   ::   P,R
```

b) Processo 2

Figura 6.5 – porzioni delle TP dei 2 processi relativa alle VMA create

6.2 La Page Cache

Il meccanismo che permette di evitare che un processo (ri)legga da disco una pagina già caricata in memoria è basato su una struttura dati fondamentale del sistema: la ***Page Cache***. Con Page Cache si intende *un insieme di pagine fisiche utilizzate per rappresentare i file in memoria e un insieme di strutture dati ausiliarie e di funzioni che ne gestiscono il funzionamento*. In particolare, le strutture ausiliarie contengono l'insieme dei descrittori delle pagine fisiche presenti nella cache; ogni ***descrittore di pagina*** contiene l'identificatore del file e l'offset sul quale è mappata, oltre ad altre informazioni, tra cui il ***ref_count***. Inoltre, le strutture ausiliarie contengono un meccanismo efficiente (**Page_Cache_Index**) per la ricerca di una pagina in base al suo descrittore costituito dalla coppia **<*identificatore file, offset*>.**

Quando un processo richiede di accedere una pagina virtuale mappata su un file il sistema svolge le seguenti operazioni (con riferimento all'esempio considerato in precedenza):

1. determina il file e il page offset richiesto; il file è indicato nella VMA (`F` nell'esempio) e l'offset (in pagine) è la somma dell'offset della VMA rispetto al File (1, nell'esempio) sommato all'offset dell'indirizzo di pagina richiesto rispetto all'inizio della VMA (0 nell'esempio, perché la variabile address punta alla prima pagina della VMA – quindi nell'esempio si trova la coppia <F, 1>
2. verifica se la pagina esiste già nella Page Cache; in caso positivo la pagina virtuale viene semplicemente mappata su tale pagina fisica
3. altrimenti alloca una pagina fisica nella page cache e vi carica la pagina del file cercata

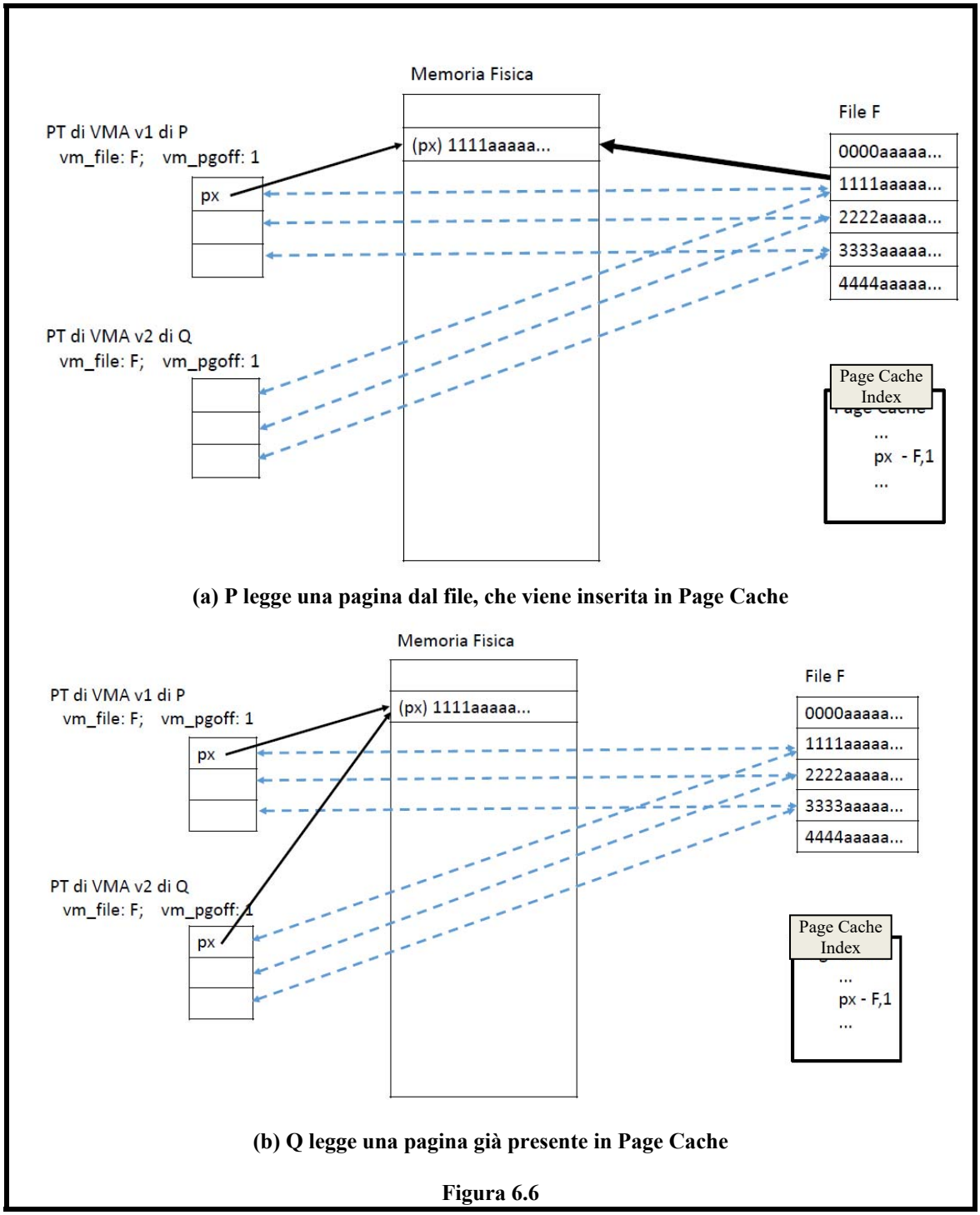
In figura 6.6.a è illustrata la sequenza di operazioni svolte dal primo processo che esegue una lettura (P):

1. la pagina cercata è la prima della VMA v1, quindi è la pagina associata a <F,1>
2. tale pagina non è presente nella page cache,
3. alloca la pagina fisica con NPF=px, la registra nell'elenco delle pagine presenti, carica la pagina leggendo il file F, poi scrive il valore px nella TP di P

Quando il secondo processo, Q, accede alla propria pagina virtuale, gli eventi sono i seguenti (fig. 6.6b):

1. determina il file e il page offset richiesto; di nuovo <F, 1>
2. trova tale valore nell'elenco della page cache, quindi trova il NPF=px e lo scrive nella TP di Q

Il secondo processo non ha quindi richiesto letture dal disco. I due processi condividono ora la pagina fisica px.



(a) P legge una pagina dal file, che viene inserita in Page Cache

(b) Q legge una pagina già presente in Page Cache

Figura 6.6

In figura 6.7 è mostrato l'effetto dell'ulteriore lettura da parte del processo Q relativa alla terza pagina di VMA: una nuova pagina fisica (py) viene allocata in page cache e resa accessibile dalla TP di Q.

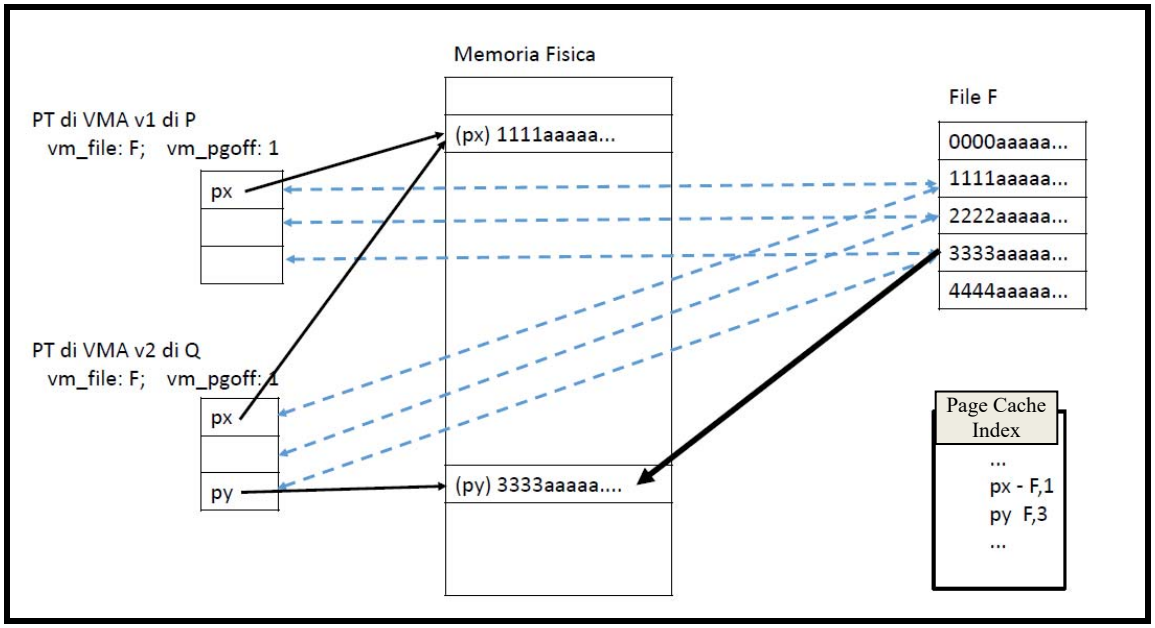


Figura 6.7 – Q legge una seconda pagina, che viene inserita in Page Cache

6.3 Accesso alle pagine - Scrittura

Il comportamento delle operazioni di scrittura su una VMA mappata su file dipende dal tipo di VMA: mappata su file in maniera SHARED oppure mappata su file in maniera PRIVATE.

Scrittura su area SHARED

La scrittura su area SHARED è la più semplice da comprendere: i dati vengono scritti sulla pagina della Page Cache condivisa, quindi:

- la pagina fisica viene modificata e marcata "dirty"
- tutti i processi che mappano tale pagina fisica vedono *immediatamente* le modifiche
- prima o poi la pagina modificata verrà riscritta sul file; non è infatti indispensabile riscrivere subito su disco la pagina, in quanto i processi che la accedono vedono la pagina in Page Cache; in questo modo *la pagina verrà scritta su disco una volta sola anche se venisse aggiornata più volte*

Ovviamente la VMA deve essere abilitata in scrittura; pertanto dobbiamo modificare l'invocazione di mmap nell'esempio precedente, che diviene

```
base = mmap(mapaddress1, PAGESIZE*3, PROT_READ|PROT_WRITE,
                                        MAP_SHARED, fd, PAGESIZE);
```

Semplificando un po' rispetto al meccanismo reale di LINUX possiamo assumere che nella TP le pagine di un'area di questo tipo siano marcate immediatamente con abilitazione in scrittura.

In Figura 6.8 è illustrato questo comportamento sull'esempio corrente modificato nel caso in cui il processo P esegue una scrittura prima che il processo Q legga. Il processo P esegue i seguenti nuovi statement:

```
address = base + 4;
for (i=0; i<10; i++){   *address = 'X';   address++;  }
```

cioè scrive 10 caratteri 'X' a partire dalla posizione 4 della prima pagina della VMA.

L'output prodotto dal secondo processo è il seguente:

```
processo 2 - prima lettura
1 1 1 1 X X X X X X
processo 2 - seconda lettura
3 3 3 3 a a a a a a
```

Come si vede, il secondo processo legge i dati modificati dal primo. questo comportamento è illustrato dalla Figura 6.8.

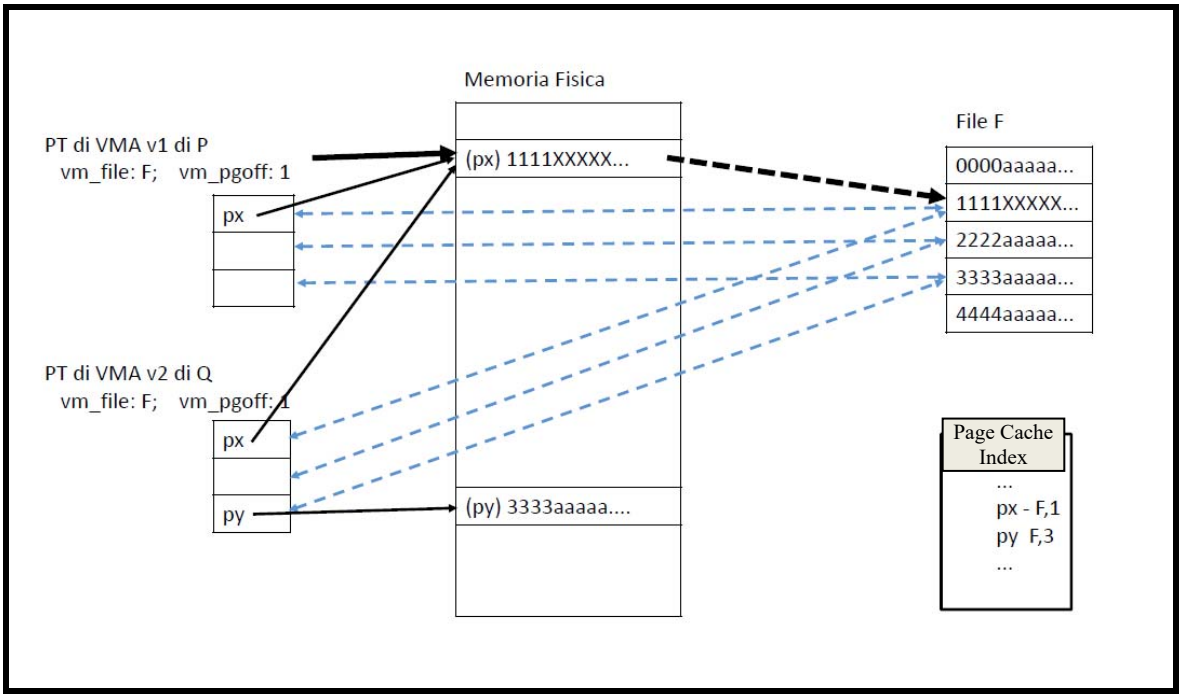


Figura 6.8 – scrittura in VMA di tipo SHARED – P scrive in pagina px
La freccia tratteggiata vuole significare che l'aggiornamento del file avverrà in maniera differita

Scrittura su area PRIVATE
Il comportamento della scrittura su VMA PRIVATE è il ***Copy-on-write*** (***COW***) già visto trattando le pagine condivise dopo una fork. Richiamiamo le regole del meccanismo COW:

- le pagine delle VMA di tipo scrivibile vengono abilitate solo in lettura
- quando un processo P scrive su una di queste pagine, ad esempio NPVx, si verifica un Page Fault per violazione di protezione
- il Page Fault Handler esegue le seguenti operazioni:
  - alloca una nuova pagina in memoria fisica, diciamo NPFy
  - copia in questa nuova pagina il contenuto della pagina che ha generato il Page Fault
  - pone nella PTE relativa a NPV l'indirizzo fisico NPFy e modifica il diritto di accesso a W
- a questo punto il processo P riparte e scrive all'indirizzo NPV, quindi nella pagina fisica NPFy

La pagina NPFy non verrà mai ricopiata sul file di backing store; essa appartiene unicamente al processo P e gli eventuali altri processi che mappano lo stesso file NON vedono le modifiche apportate dalle scritture di P.

Esempio.
Modifichiamo i programmi dell'esempio di scrittura nel modo seguente:

1. la mmap crea una VMA privata:
```
base = mmap(mapaddress1, PAGESIZE *3, PROT_READ|PROT_WRITE,
                                        MAP_PRIVATE, fd, PAGESIZE);
```
2. il processo Q rilegge le pagine 1 e 3 DOPO la scrittura da parte di P e poi stampa i contenuti

Il comportamento del sistema è illustrato in figura 6.9.a e 6.9.b; figura 6.9.a rappresenta lo stato prima dell'operazione di scrittura ed è quasi identica alla figura 6.7, ma mette in evidenza che ***le pagine hanno protezione R***. Figura 6.9.b mette in evidenza l'effetto del COW a seguito dell'operazione di scrittura; la differenza rispetto alla scrittura in area SHARED emerge dal confronto tra figura 6.8 e figura 6.9b.

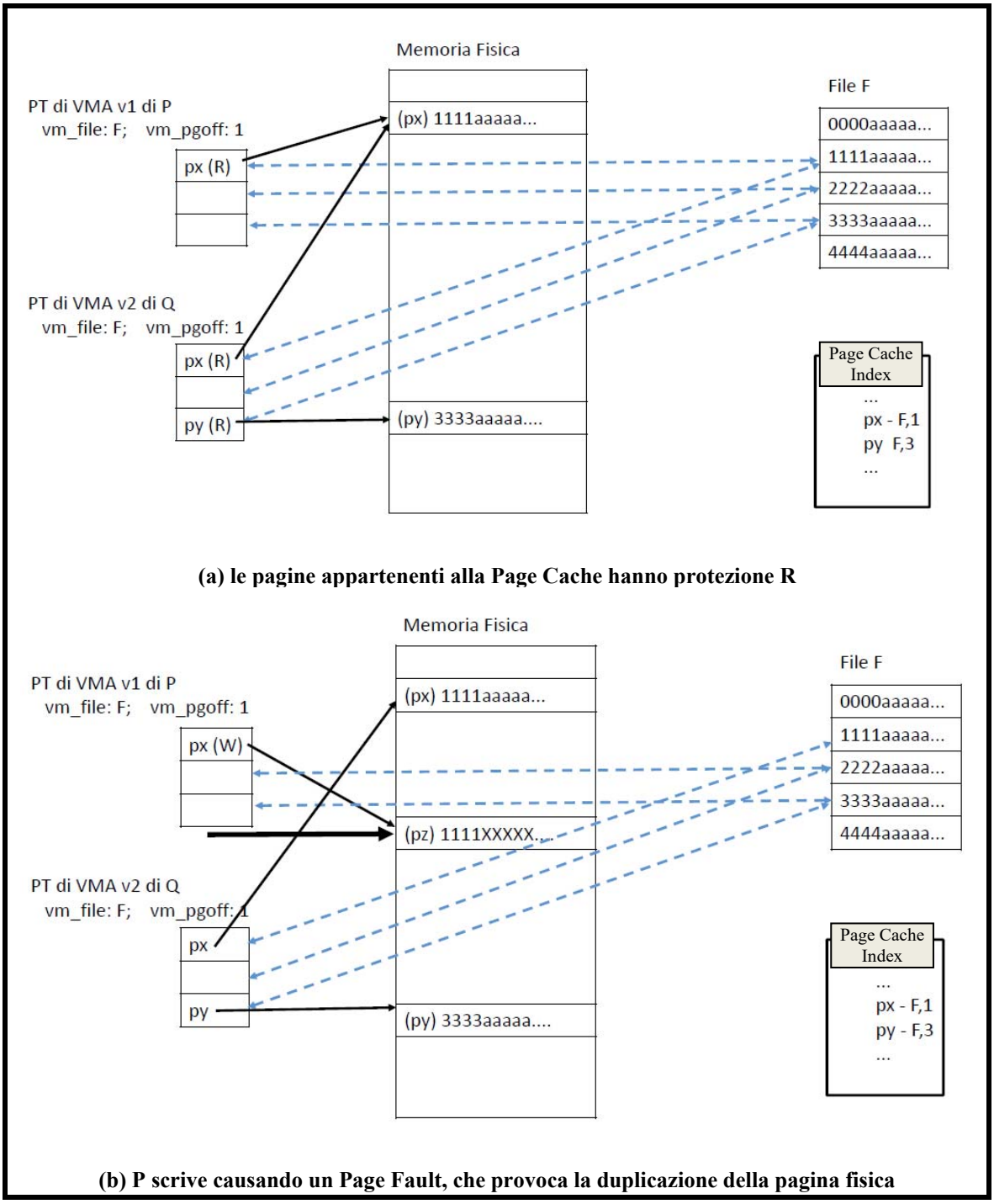


Figura 6.9 – Scrittura in VMA di tipo PRIVATE

In particolare la figura 6.9 mostra che:

- le PTE sono marcate inizialmente come Read Only (R), quindi al momento della scrittura si verifica un Page Fault
- la gestione del page fault ha allocato una nuova pagina in memoria fisica (pz) nella quale è stato copiato il contenuto della pagina px; tale nuova pagina non appartiene alla Page Cache e non è mappata sul file F – è una pagina appartenente esclusivamente al processo P
- il processo P ha modificato la pagina pz
- tali modifiche non sono osservabili dal processo Q e non verranno riportate nel file F

L'output dell'esecuzione del programma è ora (si consideri che le (ri)letture da parte del processo Q avvengono DOPO la scrittura da parte di P)

```
processo 1 - scrittura
processo 2 - prima lettura
1 1 1 1 a a a a a a
processo 2 - seconda lettura
3 3 3 3 a a a a a a
```

A conclusione dell'analisi di questo esempio in figura 6.10 è riportata la TP dalla VMA creata dalla `mmap()` nei seguenti istanti:

1. nel processo P, prima di eseguire la scrittura
2. nel processo P, dopo aver eseguito la scrittura
3. nel processo Q

Si osserva che la prima pagina della VMA è inizialmente condivisa ma viene sostituita dall'operazione di scrittura.

PROCESS: 4510, GROUP: 4510 (processo P, prima della scrittura)
[14451.379018] MAPPA DELLE AREE VIRTUALI:
[14451.379031] START: 7FFFF7FF4 END: 7FFFF7FF8 SIZE: 3 FLAGS: W, ,
[14451.379037] PAGE TABLE DELLE AREE VIRTUALI:
[14451.379038] (NPV :: NPF :: FLAGS)
[14451.379048] 7FFFF7FF4 :: **00006B3DD** :: P,R
[14451.379049] 7FFFF7FF5 :: 000000000 :: ,
[14451.379050] 7FFFF7FF6 :: 000000000 :: ,

PROCESS: 4510, GROUP: 4510 (processo P, dopo la scrittura)
[14451.379058] MAPPA DELLE AREE VIRTUALI:
[14451.379072] START: 7FFFF7FF4 END: 7FFFF7FF8 SIZE: 3 FLAGS: W, ,
[14451.379077] PAGE TABLE DELLE AREE VIRTUALI:
[14451.379088] 7FFFF7FF4 :: **000076DE4** :: P,W
[14451.379089] 7FFFF7FF5 :: 000000000 :: ,
[14451.379090] 7FFFF7FF6 :: 000000000 :: ,

PROCESS: 4513, GROUP: 4513 (processo Q)
[14452.379669] MAPPA DELLE AREE VIRTUALI:
[14452.379696] START: 7FFFF7FF4 END: 7FFFF7FF8 SIZE: 3 FLAGS: W, ,
[14452.379706] PAGE TABLE DELLE AREE VIRTUALI:
[14452.379708] (NPV :: NPF :: FLAGS)
[14452.379729] 7FFFF7FF4 :: **00006B3DD** :: P,R
[14452.379730] 7FFFF7FF5 :: 000000000 :: ,
[14452.379732] 7FFFF7FF6 :: 00006D056 :: P,R

Figura 6.10

Osservazione importante
Le pagine caricate nella Page Cache non vengono liberate neppure quando tutti i processi che le utilizzavano non le utilizzano più, ad esempio perché sono state scritte e quindi duplicate oppure addirittura perché i processi sono terminati (`exit`). Infatti LINUX applica il principio di ***mantenere in memoria le pagine lette da disco il più a lungo possibile***, perché qualche processo potrebbe volerle accedere in futuro trovandole già in memoria e risparmiando così costosi accessi a disco.
L'eventuale liberazione di pagine della Page Cache avviene solo nel contesto della generale politica di gestione della memoria fisica, a fronte di nuove richieste di pagine da parte dei processi su una memoria quasi piena, e verrà trattata nel capitolo relativo alla gestione della memoria fisica.

Scrittura su Aree non mappate su file (ANONYMOUS)
Le aree di tipo anonimo non hanno un file associato. Il sistema utilizza aree anonime per la pila o l'area dati dinamici dei processi.
La definizione di un'area anonima non alloca memoria fisica; le pagine virtuali sono tutte mappate sulla ***ZeroPage*** (una pagina fisica piena di zeri mantenuta dal sistema operativo). In questo modo le operazioni

di lettura di qualsiasi pagina virtuale della VMA trova una pagina inizializzata a zero senza richiedere l'allocazione di alcuna pagina fisica.
La scrittura in una pagina provoca l'esecuzione del meccanismo COW, come per le aree di tipo PRIVATE, e richiede l'allocazione di nuove pagine fisiche.

6.4 Mmap e fork

Le aree create tramite `mmap` si trasmettono ai figli dopo una `fork` senza bisogno di altri accorgimenti; il COW dovuto alla `mmap` e alla `fork` infatti si combinano correttamente grazie al `ref_count`.

Esempio
Nel programma di Figura 6.11 un processo padre P crea 2 processi figli (F1 e F2). Prima della `fork` P crea una VMA anonima. La prima pagina di tale VMA (cioè la pagina con NPV=0x100000) viene letta e scritta da P prima e dopo la fork e viene letta e scritta anche dai figli. Dopo ogni lettura o scrittura viene prodotta (tramite la macro VIRTUAL_AREAS) lo stato della TP del processo.

```
int main(long argc, char * argv[], char * envp[]) {
      unsigned long mapaddr = 0x100000000;
      //Creazione di una VMA anonima
      address=mmap(mapaddr,PAGESIZE*3, PROT_READ|PROT_WRITE,
                                    MAP_PRIVATE|MAP_ANONYMOUS, -1, 0);
      printf("%c ", *address);   //lettura 1
      VIRTUAL_AREAS
      *address = 'x';                 //scrittura 1
      VIRTUAL_AREAS
      pid = fork();
      if (pid == 0){                  //figlio 1
            printf("%c ", *address);  //lettura 2
            VIRTUAL_AREAS
            *address = 'x';           //scrittura 2
            VIRTUAL_AREAS
      }
      else{
            pid = fork();
            //figlio 2
            if (pid == 0){
                  printf("%c ", *address);    //lettura 3
                  VIRTUAL_AREAS
                  *address = 'x';             //scrittura 3
                  VIRTUAL_AREAS
            }
            //padre
            else{
                  printf("%c ", *address);  //lettura 4
                  VIRTUAL_AREAS
                  *address = 'x';           //scrittura 4
                  VIRTUAL_AREAS
                  ...
```

Figura 6.11

In Tabella 6.1 sono riportati gli indirizzi della pagina fisica in cui è allocata la pagina con NPV=0x100000 nei diversi processi e nei diversi momenti. Per individuare tali momenti nel codice le letture e scritture hanno un commento numerato.
La pagina fisica 0x000001E7A è quella riservata dal sistema per contenere solo zeri.
In Tabella è mostrata solo una possibile sequenza di eventi. Altre sequenze di eventi sarebbero compatibili con il risultato, ma sicuramente la scrittura 2 di F1 deve essere stata l'ultima operazione, perché ha trovato il `ref_count` della pagina 000097866 a 1 e non ha eseguito duplicazioni.
Complessivamente le duplicazioni sono state 3 e altre sequenze di eventi avrebbero comunque causato 3 duplicazioni.

Processo	una possibile sequenza di eventi	pagina fisica	prot.	ref_count			
				000001E7A	000097866	000097F4A	00009377D
2722 (P)	mmap + lett. 1	000001E7A	R	0 →1			
2722 (P)	scrittura 1.	000097866	W	1→ 0	0→1		
	fork				1→2		
	fork				2→3		
2722 (P)	lettura 4	000097866	R	0	3		
2722 (P)	scrittura 4 (COW)	000097F4A	W	0	3→2	0 → 1	
2724 (F2)	lettura 3	000097866	R	0	2		
2724 (F2)	scrittura 3 (COW)	00009377D	W	0	2→1		0 → 1
2723 (F1)	lettura 2	000097866	R	0	1		
2723 (F1)	scrittura 2 (NO COW)	000097866	W	0	1		

Tabella 6.1

6.5 Distinzione tra Backing Store e File di Swap

Quando è necessario liberare una pagina fisica che è stata scritta (detta ***dirty page***), il suo contenuto deve essere salvato su un file per poter essere recuperato.
Evidentemente le pagine SHARED sono salvate sul file sul quale sono mappate, cioè sul loro backing store, e non richiedono altri accorgimenti.
Chiamiamo ***pagine anonime*** le pagine che non possono essere salvate su backing store; esse sono di 2 tipi:

- le pagine di VMA di tipo ANONYMOUS
- le pagine di aree PRIVATE duplicate a causa di una scrittura (possiamo infatti dire che le pagine di una VMA di tipo PRIVATE dopo una scrittura diventano pagine anonime, perché non sono più mappate su file, anche se appartengono a un'area mappata su file).

Dato che non hanno un backing store utilizzabile le pagine anonime devono essere salvate su un file dedicato allo scopo, detto SWAP FILE, come vedremo nel capitolo relativo alla gestione della memoria fisica.

6.6 Condivisione degli eseguibili

Linux tratta le VMA di codice dei programmi con il meccanismo delle aree virtuali mappate su file; trattandosi di VMA che non possono essere scritte la distinzione tra SHARED e PRIVATE è irrilevante, ma nelle mappe abbiamo visto che sono definite PRIVATE.
In base a quanto visto, se due processi eseguono contemporaneamente lo stesso programma, le pagine del codice sono condivise, in quanto il secondo processo troverà tali pagine già presenti nella Page Cache.
Grazie al principio di ***mantenere in memoria le pagine lette da disco il più a lungo possibile***, è possibile anche che un processo trovi già nella Page Cache il codice del programma caricato da un precedente processo, ***anche se quest'ultimo è terminato da un pezzo*** – dipende da quanto la memoria fisica è stata occupata durante l'intervallo tra le esecuzioni dei due processi.
Anche il Linker dinamico utilizza le VMA per realizzare la condivisione delle pagine fisiche delle librerie condivise. Il Linker crea le VMA per le varie librerie condivise utilizzando il tipo MAP_PRIVATE; in questo modo:

- il codice, che non viene mai scritto, rimane sempre condiviso
- solo le pagine sulle quali un processo scrive sono riallocate al processo e non più condivise con gli altri processi e con il file

In figura 6.11a è mostrato il codice di un programma che crea due processi figli i quali mandano in esecuzione lo stesso eseguibile ("`./mainforexec.exe`"). Si noti che il padre crea il secondo figlio dopo aver atteso che il primo sia terminato.
L'eseguibile mandato in esecuzione dai figli alloca una stringa di circa 5 pagine, poi scrive nelle pagine 0 e 4 della stringa e ne legge le pagine 1 e 3; la pagina 2 non viene acceduta.
Come si vede nelle due porzioni di TP di figura 6.11b, le pagine del codice e le pagine lette del secondo processo sono allocate in memoria nelle stesse pagine fisiche utilizzate dal primo processo (evidenziate in rosso); questo significa che non sono state eliminate dalla memoria alla terminazione del primo processo.

Le pagine scritte invece sono allocate in pagine fisiche diverse, perché il primo processo le aveva modificate e quindi erano uscite dalla Page Cache.
La pagina non acceduta non è caricata affatto in memoria.

```
//programma del processo padre
int main(...) {
       ...
       pid = fork( );
       if (pid == 0) execl("./mainforexec.exe"...)...
       else { wait();
              pid = fork( );
              if (pid == 0) execl("./mainforexec.exe"...)...
              else ...
}

//programma “mainforexec” mandato in esecuzione dai figli
static char stringa[] = LONG_STRING;
int main(long argc, char * argv[], char * envp[]) {
       ...
       stringa[0] = "a";
       c = stringa[PAGESIZE];
       c = stringa[PAGESIZE*3];
       stringa[PAGESIZE*4] = c;
       VIRTUAL_AREAS; //stampa la TP
       ...
}
```

(a) programma

PROCESS: 2895, GROUP: 2895 (1° figlio)

```
[ 7736.427432]         PAGE TABLE DELLE AREE VIRTUALI:
[ 7736.427434]         (  NPV  ::     NPF     ::   FLAGS  )
[ 7736.427435] TP relativa al Codice (1 pagina)
[ 7736.427437]       000000400 :: 0000A7F3C    ::    P,X
[ 7736.427443] TP relativa ai Dati Statici (5 pagine)
[ 7736.427445]       000000601 :: 0000A0631    ::    P,W
[ 7736.427446]       000000602 :: 0000A20EF    ::    P,R
[ 7736.427447]       000000603 :: 000000000    ::     ,
[ 7736.427448]       000000604 :: 0000A1212    ::    P,R
[ 7736.427449]       000000605 :: 00009C44A    ::    P,W
```

PROCESS: 2894, GROUP: 2894 (2° figlio)

```
[ 7736.428035]         PAGE TABLE DELLE AREE VIRTUALI:
[ 7736.428037]         (  NPV  ::     NPF     ::   FLAGS  )
[ 7736.428038] TP relativa al Codice (1 pagina)
[ 7736.428040]       000000400 :: 0000A7F3C    ::    P,X
[ 7736.428046] TP relativa ai Dati Statici (5 pagine)
[ 7736.428047]       000000601 :: 0000A0B6F    ::    P,W
[ 7736.428048]       000000602 :: 0000A20EF    ::    P,R
[ 7736.428050]       000000603 :: 000000000    ::     ,
[ 7736.428051]       000000604 :: 0000A1212    ::    P,R
[ 7736.428052]       000000605 :: 00009C363    ::    P,W
```

(b) Tabelle delle pagine

Figura 6.11 – condivisione delle pagine fisiche del codice e dai dati tra processi

7. Un modello per la simulazione della memoria

La simulazione del sistema di memoria consiste nel rappresentare il contenuto delle varie strutture dati coinvolte, inclusa la memoria fisica e il TLB, dopo l'esecuzione di una serie di **Eventi**.
Per poter simulare manualmente il comportamento della gestione della memoria utilizziamo un modello basato su alcune convenzioni di rappresentazione degli indirizzi e su alcune (poche) semplificazioni del comportamento del sistema.

7.1 Convenzioni fondamentali

Le strutture dati che vengono rappresentate sono la Mappa di memoria del processo, la Tabella delle Pagine del Processo, il contenuto dei registri PC e SP, la memoria fisica e il TLB, utilizzando le convenzioni descritte nel seguito.
Per esemplificare le convenzioni consideriamo il seguente esempio:

Esempio/Esercizio 1.
Si consideri una memoria fisica di **48 K byte** disponibile per i processi di modo U. In memoria c'è un processo **P** in esecuzione. Rappresentare le strutture dati di memoria dopo che si è verificato il seguente evento:

P esegue un servizio exec(**X**); l'eseguibile X ha le seguenti caratteristiche:
- dimensioni **in pagine** delle aree: C, K, S, D (bss): 2, 1, 1, 2
- indirizzo iniziale del codice: 0x401324

Sinteticamente questo tipo di evento potrà essere indicato come **`exec(2,1,1,2, 0x401324, X)`**

a)Mappa di Memoria del processo P
La mappa di memoria di un processo viene rappresentata sostanzialmente come già visto, con le seguenti semplificazioni:

- viene utilizzata la rappresentazione simbolica già vista per identificare le aree fondamentali (C,K,S,D,P)
- al posto del "end address" viene rappresentata la dimensione in pagine dell'area
- la protezione è indicata solo come R (per read only e per execute, senza distinzione) o W (per writable)
- sono omessi l'inode e il volume
- viene indicato se l'area è mappata (M) oppure anonima (A)
- la mappatura su file è indicata con < nome file, offset in pagine>; <-1,0> se VMA anonima
- la dimensione iniziale della pila viene posta a 3 (invece di 34)

Soluzione Esempio 1 – MAPPA DI MEMORIA

```
VMA      Start Address   dim, R/W   P/S   M/A   mapping
  C   000000400,   2 , R  ,  P  ,  M  , <X,0>
  K   000000600,   1 , R  ,  P  ,  M  , <X,2>
  S   000000601,   1 , W  ,  P  ,  M  , <X,3>
  D   000000602,   2 , W  ,  P  ,  A  , <-1,0>
  P   7FFFFFFC,   3 , W  ,  P  ,  A  , <-1,0>
```

b)Memoria Fisica
La memoria viene rappresentata con una tabella nella quale sono rappresentate le pagine fisiche con il loro NPF e possono essere inseriti gli NPV delle eventuali pagine virtuali contenute. La pagina con NPF=00 è sempre utilizzata come "ZeroPage" dal Sistema Operativo e questo fatto è indicato con <ZP>.
Nel riempimento della memoria fisica utilizzeremo la seguente notazione per rappresentare gli NPV:
La pagina virtuale n appartenente all'area virtuale A viene indicata con la notazione An, dove il simbolo A specifica un tipo di area virtuale secondo la convenzione già vista:

C codice eseguibile, **K** costanti, ***S*** dati statici, ***D*** dati dinamici,
M memoria mappata , ***T*** pila di un *thread,* ***P*** pila di *main*

Importante: consideriamo l'uso di maiuscole e minuscole indifferente, quindi c0 = C0 , ecc…
Dato che un indirizzo virtuale assume un preciso significato solamente nel contesto dello spazio di indirizzamento di un processo, nei punti in cui tale spazio non è univocamente determinato dal contesto aggiungiamo un prefisso di processo alla notazione vista; ad esempio Pc0 indica la NPV c0 del processo P, Qc0 la NPV c0 del processo Q.
Altre convenzioni:

- il meccanismo di allocazione della memoria fisica utilizza sempre la prima pagina libera disponibile

- se una pagina virtuale è mappata su file, viene indicata la pagina di mappatura con le convenzioni di rappresentazione utilizzate per la mappa di memoria e separandola dal NPV tramite una /

Applicando queste convenzioni all'esempio 1 si ottiene la seguente rappresentazione della memoria (si ricorda che al momento di una exec vengono allocate la pagina iniziale del codice e una pagina di pila)

```
Soluzione Esempio 1 - MEMORIA FISICA    (pagine libere: 9)
      00 : <ZP>                ||  01 : Pc1/<X,1>          ||
      02 : Pp0                 ||  03 : ----               ||
      04 : ----                ||  05 : ----               ||
      06 : ----                ||  07 : ----               ||
      08 : ----                ||  09 : ----               ||
      10 : ----                ||  11 : ----               ||
```

La pagina iniziale del codice è c1 perché gli NPV delle pagine di codice sono

```
c0 = 000000400
c1 = 000000401
```

ecc …

quindi l'indirizzo iniziale 0x401324 appartiene a c1.
La pagina iniziale di pila è p0 e il suo NPV è 7FFFFFFE.

c) Tabella delle Pagine del processo P

La TP del processo viene rappresentata come una serie di PTE (PT entries); le PTE sono tutte e solo quelle relative alle pagine delle VMA, cioè le PTE valide.
Ogni PTE è rappresentata nel modo seguente:

< NPV: NPF protezione>, dove:

- NPV è rappresentato secondo le convenzioni simboliche viste sopra
- NPF è il numero di pagina fisica se la pagina è allocata, - se la pagina non è allocata
- la protezione è R oppure W, come per le VMA

Soluzione Esempio 1 – Tabella delle Pagine

```
<c0 :- ->  <c1 :1  R>  <k0 :- ->  <s0 :- ->  <d0 :- ->  <d1 :- ->
<p0 :2  W>  <p1 :- ->  <p2 :- ->
```

d) NPV dei registri PC e SP

Il NPV è attribuito ai registri PC e SP secondo le seguenti ipotesi.

- ogni volta che viene acceduta una pagina di pila (in lettura o scrittura) il suo NPV diventa NPV corrente dello Stack Pointer
- ogni volta che viene acceduta una pagina di codice il suo NPV diventa NPV corrente del Program Counter

```
Soluzione Esempio1 - NPV of PC and SP:   c1, p0
```

e) Stato del TLB

Il TLB è rappresentato come una tabella contenente le righe (entries) di TLB.
Ogni entry di TLB è rappresentata nel seguente formato: **NPV : NPF - D: A:**
Il comportamento del TLB è quello reale:

- nel TLB poniamo A(accessed)=1 ad ogni accesso a una pagina e D (dirty)=1 ad ogni scrittura di pagina
- ad ogni accesso a una pagina non presente nel TLB (TLB miss) carichiamo la entry dalla TP
- supponiamo che non si verifichi mai un problema di saturazione del TLB

Si osservi che la pagina iniziale della pila viene scritta, quindi nel TLB il suo Dirty bit è a 1.

```
Soluzione Esempio1 - STATO del TLB
      Pc1  : 1 -  0: 1:   ||   Pp0  : 2 -  1: 1:   ||
            ----          ||         ----          ||
            ----          ||         ----          ||
```

Raggruppando tutti i passaggi appena descritti la Soluzione dell'Esempio 1 è la seguente (per migliorare la rappresentazione in presenza di diversi processi vengono prima rappresentate la mappa, la PT e i registri relativi ad ogni processo, poi la memoria fisica e il TLB)

Soluzione Esempio 1 - completa
Le pagine della VMA di pila sono 7FFFFFFC, 7FFFFFFD e 7FFFFFFE. Quest'ultima è p0 ed è l'unica allocata.

```
PROCESSO: P  ********************************************************
    VMA : C  000000400,  2 , R  ,  P  ,  M  , <X,0>
          K  000000600,  1 , R  ,  P  ,  M  , <X,2>
          S  000000601,  1 , W  ,  P  ,  M  , <X,3>
          D  000000602,  2 , W  ,  P  ,  A  , <-1,0>
          P  7FFFFFFC,  3 , W  ,  P  ,  A  , <-1,0>
    PT: <c0 :- ->  <c1 :1  R>  <k0 :- ->  <s0 :- ->  <d0 :- ->  <d1 :- ->
             <p0 :2  W>  <p1 :- ->  <p2 :- ->
    NPV of PC and SP:   c1, p0
____MEMORIA FISICA____(pagine libere: 9)_____________________________
      00 : <ZP>                ||  01 : Pc1/<X,1>           ||
      02 : Pp0                 ||  03 : ----                ||
      04 : ----                ||  05 : ----                ||
      06 : ----                ||  07 : ----                ||
      08 : ----                ||  09 : ----                ||
      10 : ----                ||  11 : ----                ||
____STATO del TLB_____________________________________________________
      Pc1  : 1 -  0: 1:   ||   Pp0  : 2 -  1: 1:   ||
             ----         ||          ----         ||
             ----         ||          ----         ||
```

7.2 Lettura e scrittura, crescita della pila

Gli accessi alla memoria sono eventi indicati sinteticamente come

`Read(lista NPV)` e **`Write(lista NPV)`**

Esempio/Esercizio 2
Si consideri lo stato finale di memoria di esempio 1. Descrivere lo stato dopo i 2 seguenti eventi aggiuntivi:
`Read``("pk0", "ps0", "pd0","pd1")),`
`Write``("pp1","pp2","pp3"))`

Il risultato è il seguente (sono evidenziati alcuni punti significativi).

Soluzione

```
PROCESSO: P  ********************************************************
    VMA : C  000000400,  2 , R  ,  P  ,  M  , <X,0>
          K  000000600,  1 , R  ,  P  ,  M  , <X,2>
          S  000000601,  1 , W  ,  P  ,  M  , <X,3>
          D  000000602,  2 , W  ,  P  ,  A  , <-1,0>
          P  7FFFFFFFA,  5 , W  ,  P  ,  A  , <-1,0>
    PT: <c0 :- -> <c1 :1  R> <k0 :3  R> <s0 :4  R> <d0 :0  R> <d1 :0  R>
        <p0 :2  W> <p1 :5  W> <p2 :6  W> <p3 :7  W> <p4 :- ->
NPV of PC and SP:   c1, p3
____MEMORIA FISICA____(pagine libere: 4)_____________________________
      00 : Pd0/Pd1/<ZP>        ||  01 : Pc1/<X,1>           ||
      02 : Pp0                 ||  03 : Pk0/<X,2>           ||
      04 : Ps0/<X,3>           ||  05 : Pp1                 ||
      06 : Pp2                 ||  07 : Pp3                 ||
      08 : ----                ||  09 : ----                ||
      10 : ----                ||  11 : ----                ||
____STATO del TLB_____________________________________________________
      Pc1  : 01 -  0: 1:   ||   Pp0  : 02 -  1: 1:   ||
      Pk0  : 03 -  0: 1:   ||   Ps0  : 04 -  0: 1:   ||
      Pd0  : 00 -  0: 1:   ||   Pd1  : 00 -  0: 1:   ||
      Pp1  : 05 -  1: 1:   ||   Pp2  : 06 -  1: 1:   ||
      Pp3  : 07 -  1: 1:   ||         -----          ||
            -----          ||         -----          ||
```

La scrittura sulla pila ha provocato la crescita automatica della sua VMA a 5 pagine (di cui 4 scritte e l'ultima libera); le pagine dinamiche lette sono mappate sulla ZeroPage; la pagina Ps0 è stata letta in memoria ed ha protezione R, per il COW.

7.3 Scrittura nell'area dinamica e crescita dell'area dinamica

La crescita dell'area dinamica è ottenuta tramite l'evento **sbrk(N)**, dove N è il numero di pagine di incremento.

Esempio/Esercizio 3
Si consideri lo stato finale di memoria di esempio 2. Descrivere lo stato dopo i 2 seguenti eventi aggiuntivi:
`sbrk(2)`
`Write``("pd1","pd2"))`

Soluzione

```
PROCESSO: P  ********************************************************
    VMA : C  000000400,  2 , R  ,  P  ,  M  , <X,0>
          K  000000600,  1 , R  ,  P  ,  M  , <X,2>
          S  000000601,  1 , W  ,  P  ,  M  , <X,3>
          D  000000602,  4 , W  ,  P  ,  A  , <-1,0>
          P  7FFFFFFFA,  5 , W  ,  P  ,  A  , <-1,0>
    PT: <c0 :- -> <c1 :1  R> <k0 :3  R> <s0 :4  R> <d0 :0  R> <d1 :8  W>
        <d2 :9  W> <d3 :- -> <p0 :2  W> <p1 :5  W> <p2 :6  W> <p3 :7  W>
        <p4 :- ->
    NPV of PC and SP:   c1, p3
____MEMORIA FISICA____(pagine libere: 2)_____________________________
      00 : Pd0/<ZP>             ||  01 : Pc1/<X,1>            ||
      02 : Pp0                  ||  03 : Pk0/<X,2>            ||
      04 : Ps0/<X,3>            ||  05 : Pp1                  ||
      06 : Pp2                  ||  07 : Pp3                  ||
      08 : Pd1                  ||  09 : Pd2                  ||
      10 : ----                 ||  11 : ----                 ||
____STATO del TLB_____________________________________________________
      Pc1  : 01 -  0: 1:   ||   Pp0  : 02 -  1: 1:   ||
      Pk0  : 03 -  0: 1:   ||   Ps0  : 04 -  0: 1:   ||
      Pd0  : 00 -  0: 1:   ||   Pd1  : 08 -  1: 1:   ||
      Pp1  : 05 -  1: 1:   ||   Pp2  : 06 -  1: 1:   ||
      Pp3  : 07 -  1: 1:   ||   Pd2  : 09 -  1: 1:   ||
            -----          ||         -----          ||
```

7.4 Creazione di processi

La creazione di processi tramite Fork è indicata come fork("nome del processo figlio"), ad esempio fork(Q).

Questa operazione comporta la creazione di pagine condivise che vengono rappresentate in memoria tramite i loro NPV simbolici separati da una / , ad esempio Pc1/Qc1 indica la condivisione della pagina c1 tra i processi P e Q.

Esempio/Esercizio 4

Si consideri lo stato finale di memoria di esempio 3 (ma con la memoria aumentata di 2 pagine e il TLB aumentato di 2 righe). Descrivere lo stato dopo i 2 seguenti eventi aggiuntivi:

```
fork(Q)
Write("pd0"))
```

Il risultato è ottenuto applicando esattamente le regole fondamentali della operazione fork(), con il COW che determina la duplicazione di 2 pagine: Pp3 durante la fork e Pd0 a causa della scrittura successiva.

E' necessario fare particolare attenzione alla gestione del dirty bit quando si duplica una pagina applicando la seguente regola: ***se la pagina virtuale che viene spostata è marcata dirty nel TLB la pagina originale deve essere posta a D*** (vedi commento alla soluzione dell'esercizio). Inoltre si deve riportare il bit D nella PTE della pagina virtuale rimasta nella pagina fisica originale.

(ad esempio 01 Pp0/Qp0, Pp0 dirty nel TLB, dopo il COW deve essere 01 Qp0 D). Altrimenti Qp0 verrebbe scaricata senza salvarla in swap

ATTENZIONE: la dimensione della memoria è stata aumentata di 2 pagine.

Soluzione

Sono state applicate le regole fondamentali della fork. Per quanto riguarda la pagina di pila (inizialmente 07 : Pp3), questa viene condivisa e poi sdoppiata; queste operazioni si basano sui seguenti passaggi:

condivisione → 07 : Pp3/Qp3

duplicazione (e gestione dirty bit come enunciato sopra:

→ 10: Pp3, 07: Qp3 **D**

Nella PT di Q troviamo anche <p3 :7 **D** W>

```
PROCESSO: P  ********************************************************
    VMA : C  000000400,  2 , R  ,  P  ,  M  , <X,0>
          K  000000600,  1 , R  ,  P  ,  M  , <X,2>
          S  000000601,  1 , W  ,  P  ,  M  , <X,3>
          D  000000602,  4 , W  ,  P  ,  A  , <-1,0>
          P  7FFFFFFFA,  5 , W  ,  P  ,  A  , <-1,0>
    PT: <c0 :- -> <c1 :1  R> <k0 :3  R> <s0 :4  R> <d0 :11  W> <d1 :8  R>
        <d2 :9  R> <d3 :- -> <p0 :2  R> <p1 :5  R> <p2 :6  R> <p3 :10  W>
        <p4 :- ->
    NPV of PC and SP:   c1, p3
PROCESSO: Q  ********************************************************
    VMA : C  000000400,  2 , R  ,  P  ,  M  , <X,0>
          K  000000600,  1 , R  ,  P  ,  M  , <X,2>
          S  000000601,  1 , W  ,  P  ,  M  , <X,3>
          D  000000602,  4 , W  ,  P  ,  A  , <-1,0>
          P  7FFFFFFFA,  5 , W  ,  P  ,  A  , <-1,0>
    PT: <c0 :- -> <c1 :1  R> <k0 :3  R> <s0 :4  R> <d0 :0  R> <d1 :8  R>
        <d2 :9  R> <d3 :- -> <p0 :2  R> <p1 :5  R> <p2 :6  R> <p3 :7  D W>
        <p4 :- ->
    NPV of PC and SP:   c1, p3
____MEMORIA FISICA____(pagine libere: 2)_____________________
      00 : Qd0/<ZP>             ||  01 : Pc1/Qc1/<X,1>       ||
      02 : Pp0/Qp0              ||  03 : Pk0/Qk0/<X,2>       ||
      04 : Ps0/Qs0/<X,3>        ||  05 : Pp1/Qp1             ||
      06 : Pp2/Qp2              ||  07 : Qp3  D               ||
      08 : Pd1/Qd1              ||  09 : Pd2/Qd2             ||
      10 : Pp3                  ||  11 : Pd0                 ||
      12 : ----                 ||  13 : ----                ||
____STATO del TLB_____________________________________________
      Pc1  : 01 -  0: 1:   ||   Pp0  : 02 -  1: 1:   ||
      Pk0  : 03 -  0: 1:   ||   Ps0  : 04 -  0: 1:   ||
      Pd0  : 11 -  1: 1:   ||   Pd1  : 08 -  1: 1:   ||
      Pp1  : 05 -  1: 1:   ||   Pp2  : 06 -  1: 1:   ||
      Pp3  : 10 -  1: 1:   ||   Pd2  : 09 -  1: 1:   ||
            -----          ||         -----          ||
            -----          ||         -----          ||
```

7.5 Creazione di thread

La creazione di thread viene indicata come

`clone(nome del processo nuovo, pagina iniziale della thread function)`

ad esempio clone(R, c2).

L'evento di clone crea un nuovo processo che opera sulle stesse pagine virtuali del processo originale – non si tratta di una condivisione di pagina fisica da parte di diverse pagine virtuali, ma di identità di pagina virtuale nei diversi processi. Anche la PT è condivisa. Pertanto nella gestione della memoria è necessario rappresentare delle pagine virtuali che appartengono a più di un processo – nel modello di simulazione utilizziamo semplicemente il concatenamento dei nomi dei processi, ad esempio PRc1 indica la pagina di codice c1 appartenente ai due processi P ed R.

L'operazione di clone crea anche la pila per il nuovo thread e assegna al suo PC il NPV di inizio della thread_function.

Come dimensione virtuale della pila dei thread assumiamo 2 pagine.

Le eventuali pile di nuovi thread vengono concatenate nella direzione degli indirizzi bassi, come visto nel capitolo 5, con una pagina vuota di interposizione.

La pila del primo thread (VMA T0) inizia logicamente con NPV 0x**7FFFF77FF**, cioè la pagina T00 possiede questo NPV.

La prima pagina di pila di ogni thread viene fisicamente allocata in memoria, perché scritta dalla `clone` (si veda lo pseudocodice di clone nel capitolo N4) con tutte le conseguenze su TP, memoria e TLB.

Esempio/Esercizio 5
Si consideri lo stato finale esempio 4. Descrivere lo stato dopo il seguente evento aggiuntivo:
`clone(R, c0)`
`clone(S, c1)`

Soluzione
L'area T0 è costituita dalle pagine 7FFFF77FE e 7FFFF77FF, la pagina 7FFFF77FD è di interposizione, la pila T1 è costituita dalle pagine 7FFFF77FB e 7FFFF77FC.

```
PROCESSO: P/R/S  ******************************************************
    VMA : C  000000400,  2 , R  ,  P  ,  M  , <X,0>
          K  000000600,  1 , R  ,  P  ,  M  , <X,2>
          S  000000601,  1 , W  ,  P  ,  M  , <X,3>
          D  000000602,  4 , W  ,  P  ,  A  , <-1,0>
          T1 7FFFF77FB,  2 , W  ,  P  ,  A  , <-1,0>
          T0 7FFFF77FE,  2 , W  ,  P  ,  A  , <-1,0>
          P  7FFFFFFFA,  5 , W  ,  P  ,  A  , <-1,0>
    PT: <c0 :- -> <c1 :1  R> <k0 :3  R> <s0 :4  R> <d0 :11  W> <d1 :8  R>
         <d2 :9  R> <d3 :- -> <p0 :2  R> <p1 :5  R> <p2 :6  R> <p3 :10 W>
         <p4 :- -> <t00:12 W> <t01:- -> <t10:13 W> <t11:- ->
    process P - NPV of PC and SP:  c1, p3
    process R - NPV of PC and SP:  c0, t00
    process S - NPV of PC and SP:  c1, t10
PROCESSO: Q  ******************************************************
    VMA : C  000000400,  2 , R  ,  P  ,  M  , <X,0>
          K  000000600,  1 , R  ,  P  ,  M  , <X,2>
          S  000000601,  1 , W  ,  P  ,  M  , <X,3>
          D  000000602,  4 , W  ,  P  ,  A  , <-1,0>
          P  7FFFFFFFA,  5 , W  ,  P  ,  A  , <-1,0>
    PT: <c0 :- -> <c1 :1  R> <k0 :3  R> <s0 :4  R> <d0 :0  R> <d1 :8  R>
        <d2 :9  R> <d3 :- -> <p0 :2  R> <p1 :5  R> <p2 :6  R> <p3 :7 D W>
        <p4 :- ->
    process Q - NPV of PC and SP:  c1, p3
____MEMORIA FISICA____(pagine libere: 0)_____________________
      00 : Qd0/<ZP>            ||  01 : PRSc1/Qc1/<X,1>     ||
      02 : PRSp0/Qp0           ||  03 : PRSk0/Qk0/<X,2>     ||
      04 : PRSs0/Qs0/<X,3>     ||  05 : PRSp1/Qp1           ||
      06 : PRSp2/Qp2           ||  07 : Qp3                 ||
      08 : PRSd1/Qd1           ||  09 : PRSd2/Qd2           ||
      10 : PRSp3               ||  11 : PRSd0               ||
      12 : PRSt00              ||  13 : PRSt10              ||
____STATO del TLB_____________________________________________
      PRSc1 : 01 -  0: 1:   ||   PRSp0 : 02 -  1: 1:   ||
      PRSk0 : 03 -  0: 1:   ||   PRSs0 : 04 -  0: 1:   ||
      PRSd0 : 11 -  1: 1:   ||   PRSd1 : 08 -  1: 1:   ||
      PRSp1 : 05 -  1: 1:   ||   PRSp2 : 06 -  1: 1:   ||
      PRSp3 : 10 -  1: 1:   ||   PRSd2 : 09 -  1: 1:   ||
      PRSt00: 12 -  1: 1:   ||   PRSt10: 13 -  1: 1:   ||
            -----           ||         -----           ||
```

7.6 Commutazione di contesto

L'evento è indicato come `ContextSwitch(nome del processo da mettere in esecuzione).` Questo evento non modifica le mappe di memoria. Il suo effetto principale è sul TLB: infatti il TLB esegue un flush, poi carica le pagine attive di codice e pila (cioè quelle che contengono gli indirizzi di PC e SP del processo che va in esecuzione). Questa operazione può avere un effetto sulla TP.
Una conseguenza importante del flush del TLB è costituita dalla necessità di salvare nei descrittori delle pagine eliminate dal TLB lo stato del dirty bit, che altrimenti andrebbe perso. Nel modello di simulazione marchiamo in memoria le pagine dirty con una D, ad esempio: <02 : PRSp0/Qp0 **D**>

Esempio/Esercizio 6
Si consideri lo stato finale di memoria di esempio 5. Descrivere lo stato dopo il seguente evento aggiuntivo:
`contextSwitch(Q)`

Soluzione
Dato che questa operazione non modifica le mappe di memoria e le TP dei processi riportiamo solamente il nuovo stato di memoria e del TLB. Nella memoria sono marcate con D le 9 pagine che erano dirty nel TLB precedente (più Qp3) e il TLB contiene ora solamente le due pagine iniziali di Q.

```
____MEMORIA FISICA____(pagine libere: 0)_____________________________
      00 : Qd0/<ZP>             ||  01 : PRSc1/Qc1/<X,1>     ||
      02 : PRSp0/Qp0 D          ||  03 : PRSk0/Qk0/<X,2>     ||
      04 : PRSs0/Qs0/<X,3>      ||  05 : PRSp1/Qp1 D         ||
      06 : PRSp2/Qp2 D          ||  07 : Qp3  D              ||
      08 : PRSd1/Qd1 D          ||  09 : PRSd2/Qd2 D         ||
      10 : PRSp3 D              ||  11 : PRSd0 D             ||
      12 : PRSt00 D             ||  13 : PRSt10 D            ||
____STATO del TLB_____________________________________________________
      Qc1 : 01 -  0: 1:    ||   Qp3 : 07 -  1: 1:    ||
            -----          ||         -----          ||
             ...
```

7.7 Exit
L'evento di terminazione di un processo è indicato come

```
exit(nome del processo da mettere in esecuzione)
```

Questo evento elimina tutte le NPV del processo da tutte le strutture dati; tale eliminazione può dar luogo alla creazione di pagine libere in memoria laddove la NPV eliminata era l'unica occupante di una pagina fisica (assenza di condivisione, ref_count = 1).
Alla fine di una exit viene eseguita una operazione equivalente a una commutazione di contesto mettendo in esecuzione il nuovo processo indicato nell'evento.
Nel caso in cui il processo terminato abbia dei thread secondari attivi, anche questi devono essere eliminati
La terminazione di un thread non modifica la memoria in alcun modo; neppure la pila del thread terminato viene eliminata. Nel modello il nome del thread terminato viene eliminato dagli NPV simbolici in cui compariva.

Esempio/Esercizio 7a
Si consideri lo stato finale di memoria di esempio 6 (riportata in figura 7.1a). Descrivere lo stato dopo il seguente evento aggiuntivo:
`exit(S)`

Soluzione (vedi figura 7.1a)
Questo evento elimina il processo corrente Q e mette in esecuzione il processo (thread) S.
La mappa di memoria e la TP di Q sono eliminate. Gli NPV di Q sono eliminati da tutte le pagine in cui comparivano, ma l'unica pagina fisica liberata è quella con NPF=7, perché tutte le altre pagine erano condivise.
Nel TLB ora compaiono solo le pagine attive di S.

Esempio/Esercizio 7b
Descrivere lo stato della sola memoria e del TLB dopo il seguente evento aggiuntivo:
`ContextSwitch(R)`

Soluzione (vedi figura 7.1b)
Questo evento carica in memoria la pagina c0 iniziale del thread R e sostituisce il contenuto del TLB

Esempio/Esercizio 7c
Descrivere lo stato della sola memoria e del TLB dopo il seguente evento aggiuntivo:
`Exit(P)`

Soluzione (vedi figura 7.1c)

```
PROCESSO: P/R/S  ******************************************************
    VMA : C  000000400,  2 , R  ,  P  ,  M  , <X,0>
          K  000000600,  1 , R  ,  P  ,  M  , <X,2>
          S  000000601,  1 , W  ,  P  ,  M  , <X,3>
          D  000000602,  4 , W  ,  P  ,  A  , <-1,0>
          T1 7FFFF77FB,  2 , W  ,  P  ,  A  , <-1,0>
          T0 7FFFF77FE,  2 , W  ,  P  ,  A  , <-1,0>
          P  7FFFFFFFA,  5 , W  ,  P  ,  A  , <-1,0>
    PT: <c0 :- -> <c1 :1  R> <k0 :3  R> <s0 :4  R> <d0 :11  D W> <d1 :8  D R>
      <d2 :9 D R> <d3 :- -> <p0 :2 D R> <p1 :5 D R> <p2 :6 D R> <p3 :10 D W>
      <p4 :- -> <t00:12  D W> <t01:- -> <t10:13  D W> <t11:- ->
    process P - NPV of PC and SP:  c1, p3
    process R - NPV of PC and SP:  c0, t00
    process S - NPV of PC and SP:  c1, t10
____MEMORIA FISICA____(pagine libere: 1)_____________________________
      00 : <ZP>                ||  01 : PRSc1/<X,1>          ||
      02 : PRSp0 D             ||  03 : PRSk0/<X,2>          ||
      04 : PRSs0/<X,3>         ||  05 : PRSp1 D              ||
      06 : PRSp2 D             ||  07 : ----                 ||
      08 : PRSd1 D             ||  09 : PRSd2 D              ||
      10 : PRSp3 D             ||  11 : PRSd0 D              ||
      12 : PRSt00 D            ||  13 : PRSt10 D             ||
____STATO del TLB_____________________________________________________
      PRSc1 : 01 -  0: 1:   ||   PRSt10 : 13 -  1: 1:   ||
            -----           ||         -----            ||
             ...
```

(a)

```
=========== Dopo ContextSwitch R ====
____MEMORIA FISICA____(pagine libere: 0)_____________________________
      00 : <ZP>                ||  01 : PRSc1/<X,1>          ||
      02 : PRSp0 D             ||  03 : PRSk0/<X,2>          ||
      04 : PRSs0/<X,3>         ||  05 : PRSp1 D              ||
      06 : PRSp2 D             ||  07 : PRSc0/<X,0>          ||
      08 : PRSd1 D             ||  09 : PRSd2 D              ||
      10 : PRSp3 D             ||  11 : PRSd0 D              ||
      12 : PRSt00 D            ||  13 : PRSt10 D             ||
____STATO del TLB_____________________________________________________
      PRSc0 : 07 -  0: 1:   ||   PRSt00 : 12 -  1: 1:   ||
            -----           ||         -----           ||
            -----           ||         -----           ||
             ...
```

(b)

```
=========== Dopo exit(P)  ====
____MEMORIA FISICA____(pagine libere: 0)_____________________________
      00 : <ZP>                ||  01 : PSc1/<X,1>           ||
      02 : PSp0 D              ||  03 : PSk0/<X,2>           ||
      04 : PSs0/<X,3>          ||  05 : PSp1 D               ||
      06 : PSp2 D              ||  07 : PSc0/<X,0>           ||
      08 : PSd1 D              ||  09 : PSd2 D               ||
      10 : PSp3 D              ||  11 : PSd0 D               ||
      12 : PSt00 D             ||  13 : PSt10 D              ||
____STATO del TLB_____________________________________________________
      PSc1 : 01 -  0: 1:   ||   PSp3 : 10 -  1: 1:   ||
            -----          ||         -----          ||
             ...
```

(c)

Figura 7.1

7.8 Mmap

L'evento di invocazione di mmap è indicato come

```
mmap(indirizzo, dimensioni area, R/W, S/P, M/A, nome file, fileoffset)
```

Esempio/Esercizio 8

In memoria c'è un processo **P** in esecuzione. Rappresentare le strutture dati di memoria dopo che si sono verificati i seguenti eventi:

```
exec( 2,1,1,2, c1, "XX");
mmap(0x10000000, 2, W, S, M, "G", 2);
mmap(0x20000000, 1, R, S, M, "G", 4);
mmap(0x30000000, 1, W, P, M, "F", 2);
mmap(0x40000000, 1, W, P, A, -1, 0);
```

Soluzione

```
PROCESSO: P  ********************************************************
    VMA : C  000000400,  2 , R  ,  P  ,  M  , <XX,0>
          K  000000600,  1 , R  ,  P  ,  M  , <XX,2>
          S  000000601,  1 , W  ,  P  ,  M  , <XX,3>
          D  000000602,  2 , W  ,  P  ,  A  , <-1,0>
          M0 000010000,  2 , W  ,  S  ,  M  , <G,2>
          M1 000020000,  1 , R  ,  S  ,  M  , <G,4>
          M2 000030000,  1 , W  ,  P  ,  M  , <F,2>
          M3 000040000,  1 , W  ,  P  ,  A  , <-1,0>
          P  7FFFFFFFC,  3 , W  ,  P  ,  A  , <-1,0>
    PT: <c0 :- -> <c1 :1  R> <k0 :- -> <s0 :- -> <d0 :- -> <d1 :- ->
        <p0 :2  W> <p1 :- -> <p2 :- -> <m00:- -> <m01:- -> <m10:- ->
        <m20:- -> <m30:- ->
    process P - NPV of PC and SP:  c1, p0
____MEMORIA FISICA____(pagine libere: 7)_____________________________
      00 : <ZP>                ||  01 : Pc1/<XX,1>          ||
      02 : Pp0                 ||  03 : ----                ||
      04 : ----                ||  05 : ----                ||
      06 : ----                ||  07 : ----                ||
      08 : ----                ||  09 : ----                ||
____STATO del TLB_____________________________________________________
      Pc1 : 01 -  0: 1:    ||   Pp0 : 02 -  1: 1:    ||
            -----          ||         -----          ||
            -----          ||         -----          ||
            -----          ||         -----          ||
```

Esempio/Esercizio 9

Si consideri lo stato finale di memoria di esempio 8. Descrivere lo stato dopo il seguente evento aggiuntivo:

```
Read(Pm10, Pm30);
Write(Pm00, Pm20);
```

```
PROCESSO: P  ********************************************************
    VMA : C  000000400,  2 , R  ,  P  ,  M  , <XX,0>
          K  000000600,  1 , R  ,  P  ,  M  , <XX,2>
          S  000000601,  1 , W  ,  P  ,  M  , <XX,3>
          D  000000602,  2 , W  ,  P  ,  A  , <-1,0>
          M0 000010000,  2 , W  ,  S  ,  M  , <G,2>
          M1 000020000,  1 , R  ,  S  ,  M  , <G,4>
          M2 000030000,  1 , W  ,  P  ,  M  , <F,2>
          M3 000040000,  1 , W  ,  P  ,  A  , <-1,0>
          P  7FFFFFFFC,  3 , W  ,  P  ,  A  , <-1,0>
    PT: <c0 :- -> <c1 :1  R> <k0 :- -> <s0 :- -> <d0 :- -> <d1 :- ->
        <p0 :2  W> <p1 :- -> <p2 :- -> <m00:4  W> <m01:- -> <m10:3  R>
        <m20:6  W> <m30:0  R>
    process P - NPV of PC and SP:  c1, p0
____MEMORIA FISICA____(pagine libere: 3)_____________________________
      00 : Pm30/<ZP>           ||  01 : Pc1/<XX,1>          ||
      02 : Pp0                 ||  03 : Pm10/<G,4>          ||
      04 : Pm00/<G,2>          ||  05 : <F,2>               ||
      06 : Pm20                ||  07 : ----                ||
      08 : ----                ||  09 : ----                ||
____STATO del TLB____________________________________________________
      Pc1 : 01 -  0: 1:    ||   Pp0 : 02 -  1: 1:    ||
      Pm10 : 03 -  0: 1:   ||   Pm30 : 00 -  0: 1:   ||
      Pm00 : 04 -  1: 1:   ||   Pm20 : 06 -  1: 1:   ||
            -----          ||         -----          ||
```