organizzazione dello spazio virtuale dei processi

aree virtuali – VMA (*Virtual Memory Areas*)

- la memoria virtuale di un processo LINUX è suddivisa in un certo numero di **aree di memoria virtuale** (***Virtual Memory Area* – VMA**)
- ogni area è costituita da un **numero intero** di pagine **consecutive**, che hanno caratteristiche di accesso alla memoria **omogenee**
- ciascuna area è delimitata da
  - $\text{indirizzo}_{\text{iniziale}}$, $\text{indirizzo}_{\text{finale}}$

  ovvero da
  - $\text{NPV}_{\text{iniziale}}$, $\text{NPV}_{\text{finale}}$
- prima di analizzare ulteriormente questi aspetti è importante precisare che lo studio della memoria sarà condotto **disabilitando la funzione *ASLR* (*Address Space Layout Randomization*)**

aree virtuali di un processo – 1

- **codice (C)**: contiene le istruzioni e anche le costanti definite all'interno del codice (ad esempio le stringhe da passare alla funzione *printf*)
- **costanti per rilocazione dinamica (K):** è un'area destinata a contenere dei parametri determinati 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
  - il limite corrente di quest'area è indicato dalla variabile ***brk*** (***program break***) contenuta nel descrittore del processo
  - contiene anche gli eventuali dati non inizializzati definiti nell'eseguibile (***BSS – Block Started by Symbol***)

aree virtuali di un processo – 2

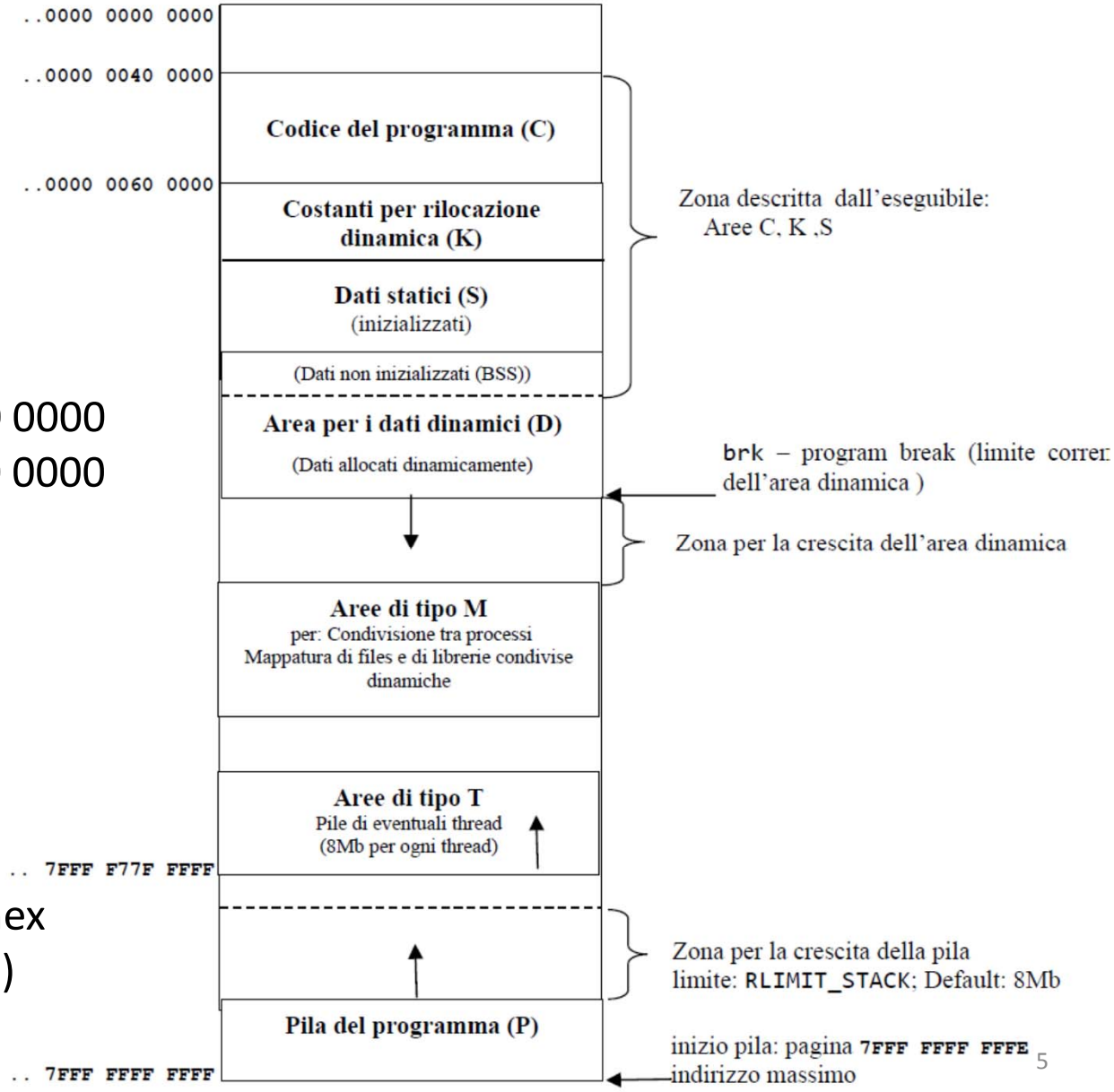
- **aree per Memory-Mapped file (M):** queste aree permettono di mappare un file su una porzione di memoria virtuale di un processo cosicché il file possa essere letto o scritto come se fosse un array di byte in memoria; sono utilizzate per
  - *librerie dinamiche (shared libraries o dynamic linked libraries)*
    - codice che non viene incorporato staticamente nel programma eseguibile
    - vengono caricate in memoria durante l'esecuzione del programma in base alle esigenze del programma stesso
    - sono caricate mappando il loro file eseguibile su una o più aree virtuali di tipo M
    - possono 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 sola lettura o in lettura e scrittura
- **pile dei thread (T)**: sono le aree utilizzate per le pile dei *thread*
- **pila (P)**: è l'area di pila di modo U del processo

aree virtuali del processo

indirizzi dell'eseguibile
codice inizia a indir. 0x 0000 0000 0040 0000
dati iniziano a indir. 0x 0000 0000 0060 0000

indirizzo iniziale delle pile dei thread
0x 0000 7FFF F77F FFFF

ovviamente sono indirizzi da 64 bit di USERSPACE (16 cifre hex con prime 4 cifre hex uguali a 0)
X64 corrente usa solo 48 bit (la 4 cifre hex inziali 0000 sono generalmente omesse)



vm_area_struct

```
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 }
```

vm_area_struct

principali indicatori (flag) di proprietà delle VMA

```
#define VM_READ          0x00000001    // VMA in sola lettura
#define VM_WRITE         0x00000002    // VMA in sola scrittura
#define VM_EXEC          0x00000004    // VMA di codice eseguibile (leggibile)
#define VM_SHARED        0x00000008    // VMA in condivisione
#define VM_GROWSDOWN     0x00000100    // VMA con crescita automatica (pila)
#define VM_DENYWRITE     0x00000800    // VMA mappata su file non scrivibile
```

una VMA può essere **mappata su file** oppure **anonima** (ANONYMOUS)

- il file è detto ***backing store***
- esso è definito in *vm_area_struct* da
  - struct file * **vm_file** → individua il file utilizzato come *backing store*
  - unsigned long **vm_pgoff** → posizione (offset) all'interno del file stesso

esempio: cat /proc/NN/maps (NN è il pid del processo)

start – end (indir. iniz. – fin.)	perm	offset	device	i-node	file name
000000400000 – 000000401000	r-xp	000000	08:01	394275	…/user.exe
000000600000 – 000000601000	r--p	000000	08:01	394275	…/user.exe
000000601000 – 000000602000	rw-p	001000	08:01	394275	…/user.exe
7ffff7a1c000 – 7ffff7bd0000	r-xp	000000	08:01	271666	…/libc-2.15.so
7ffff7bd0000 – 7ffff7dcf000	---p	1b4000	08:01	271666	…/libc-2.15.so
7ffff7dcf000 – 7ffff7dd3000	r--p	1b3000	08:01	271666	…/libc-2.15.so
7ffff7dd3000 – 7ffff7dd5000	rw-p	1b7000	08:01	271666	…/libc-2.15.so
...					
(altre aree M)					
...					
7fffffde000 – 7fffffff000	rw-p				[stack]

commenti alla mappa

al momento della *exec* di un programma eseguibile, **LINUX costruisce la struttura delle aree virtuali del processo in base alla struttura definita dall'eseguibile**

- **l'area di pila** è stata allocata con una dimensione iniziale di **34 pagine** (144 K byte)
- 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 (major:minor, sono i due numeri identificativi del dispositivo di I/O), ma l'eseguibile del programma e quello della libreria *glibc* sono diversi e quindi hanno diverso *i-node*
- l'area D è assente; viene creata solo in presenza di dati statici non inizializzati nell'eseguibile (BSS)

Page Table (visualizzata tramite un modulo apposito)

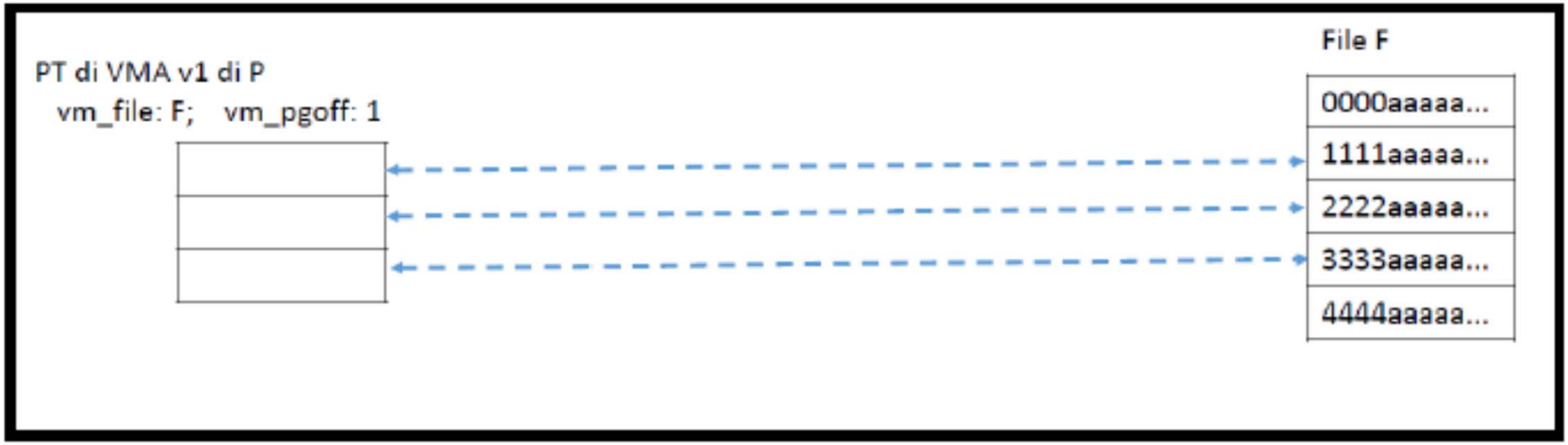
```
[11299.304331]          MAPPA DELLE AREE VIRTUALI:
[11299.304331]
[11299.304334] START: 000000400   END: 000000401  SIZE:   1  FLAGS: X,  , D
[11299.304335] START: 000000600   END: 000000601  SIZE:   1  FLAGS: R,  , D
[11299.304336] START: 000000601   END: 000000602  SIZE:   1  FLAGS: W,  , D
...
[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
```

meccanismo generale delle VMA

- le VMA sono classificate in base a due criteri
  - le VMA possono essere **mappate su file**
  - oppure essere **anonime** (ANONYMOUS)
- inoltre possono essere di tipo **SHARED** oppure **PRIVATE**
- tutte le combinazioni di queste due classificazioni sono supportate in Linux, ma noi non considereremo la combinazione SHARED | ANONYMOUS
- pertanto indicheremo i tre tipi di aree considerate come
  - SHARED (e mappate su file)
  - PRIVATE (e mappate su file)
  - ANONIMOUS (implicitamente PRIVATE)

aree mappate su file – sola lettura

- un file in Linux è considerato una sequenza di byte; 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 con il byte 4096, ecc



mmap – creazione di VMA da programma utente

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

- **`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 o PROT_WRITE
- **`flags`**: indica numerose opzioni; le sole che noi consideriamo sono MAP_SHARED (mappata su file e condivisa), MAP_PRIVATE (mappata su file e privata) e MAP_ANONIMOUS (non mappata su file e privata)
- **`fd`**: indica il descrittore del file in cui mappare l'area
- **`offset`**: indica la posizione iniziale dell'area rispetto al file

creazione di VMA e allocazione fisica delle pagine

- la creazione di una VMA consiste esclusivamente nella definizione dello spazio virtuale associato **senza alcuna allocazione di pagine fisiche**
- la creazione della VMA predispone anche la porzione di Tabella delle Pagine (TP) necessaria a rappresentare le pagine virtuali della VMA – per ogni pagina virtuale la corrispondente PTE (Page Table Entry) indicherà che la pagina non è allocata fisicamente
- l'allocazione delle pagine fisiche avviene solamente quando un processo legge o scrive una pagina virtuale NPV – in tale caso nella PTE associata a tale NPV viene inserito il numero NPF di pagina fisica
- quindi è importante distinguere tra
  - le operazioni che **modificano esclusivamente lo spazio virtuale** del processo:
    - creazione o eliminazione di VMA
    - estensione o riduzione di una VMA esistente
  - le operazioni che **allocano memoria fisica** (lettura e scrittura in memoria)

esempio 1: creazione di VMA condivise (SHARED)

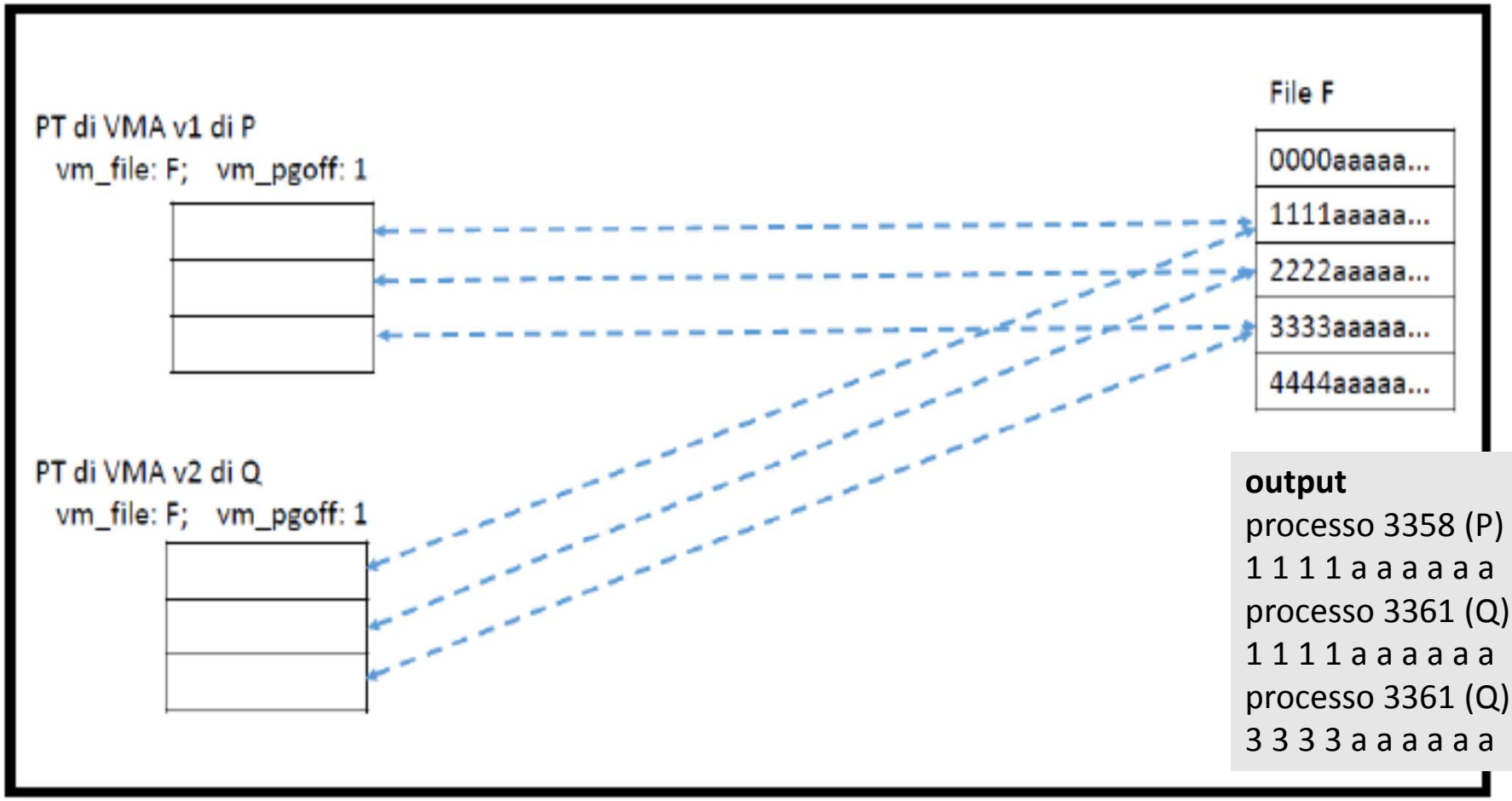
```
#define PAGESIZE 1024 * 4
char * base
unsigned long mapaddress1 = 0x 100000000
unsigned long mapaddress2 = 0x 200000000

/* vengono creati due processi con VMA a indirizzi diversi
 * su file fd con offset = PAGESIZE
 * ambedue i processi leggono la prima pagina
 * il secondo processo legge anche la terza pagina
*/
```

esempio 1 – continua

```
pid = fork ( )
if  (pid == 0)  {      // processo 1 (P)
            fd = open ("./F", O_RDWR)
            base = mmap (mapaddress1, PAGESIZE * 3, PROT_READ, MAP_SHARED, fd, PAGESIZE)
            // lettura  (10 caratteri da indirizzo base)
            }
...
pid = fork ( )
 if  (pid == 0)  {     // processo 2 (Q)
            fd = open ("./F", O_RDWR)
            base = mmap (mapaddress2, PAGESIZE * 3, PROT_READ, MAP_SHARED, fd, PAGESIZE);
            // lettura 1  (10 caratteri da indirizzo base)
            // lettura 2  (10 caratteri da indirizzo base + 2 * PAGESIZE)
}
```

esempio 1



output
processo 3358 (P) – lettura
1 1 1 1 a a a a a a
processo 3361 (Q) – prima lettura
1 1 1 1 a a a a a a
processo 3361 (Q) – seconda lettura
3 3 3 3 a a a a a a

esempio 1 – mappe dei due processi

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	**rw-s**	**00001000**	**08:01**	**396260**	**.../F**

...

a) processo 1 (P)

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	**rw-s**	**00001000**	**08:01**	**396260**	**.../F**

...

b) processo 2 (Q)

esempio 1 – le TP (parziali) dei due processi

[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

- la **prima pagina virtuale** è **condivisa fisicamente**
- la **terza pagina** è stata **caricata in memoria** per il processo 2

→ quindi esiste un meccanismo che ha permesso al processo 2 di accorgersi che la prima pagina era già stata caricata in memoria

Page Cache

tutte le pagine fisiche hanno un ***descrittore di pagina***

la *Page Cache* è

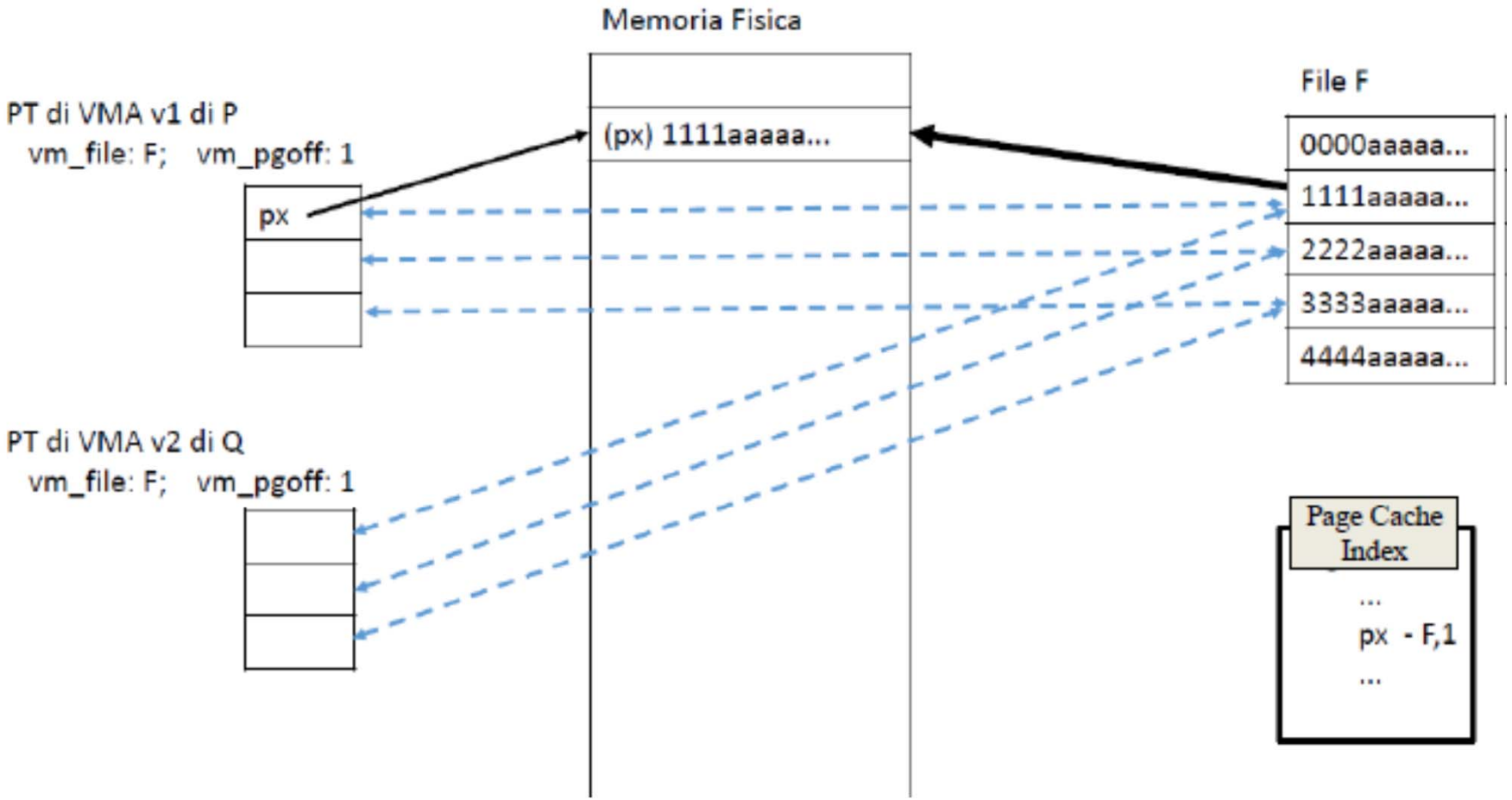
- un insieme di pagine fisiche utilizzate per rappresentare i file in memoria
  - tipicamente sono pagine appartenenti a VMA di tipo MAPPED
- e un insieme di strutture dati ausiliarie e di funzioni che ne gestiscono il funzionamento
- in particolare
  - per ogni pagina fisica in Page Cache, il ***descrittore di pagina*** contiene la coppia ⟨***identificatore_file, offset***⟩, oltre ad altre informazioni tra cui il ***ref_count*** ossia il contatore dei riferimenti alla pagina fisica
  - il ***ref_count*** è uguale al numero di processi che mappano la pagina +1 (vedremo che + 1 rende possibile mantenere allocata la pagina fisica anche se tutti i processi che la utilizzano terminano)
    NB: il +1 non c'è se la pagina non è mappata su file e dunque non sta in Page Cache
  - ***Page_Cache_Index*****:** un meccanismo efficiente per la ricerca di una pagina in base al suo descrittore

accesso alla *Page Cache*

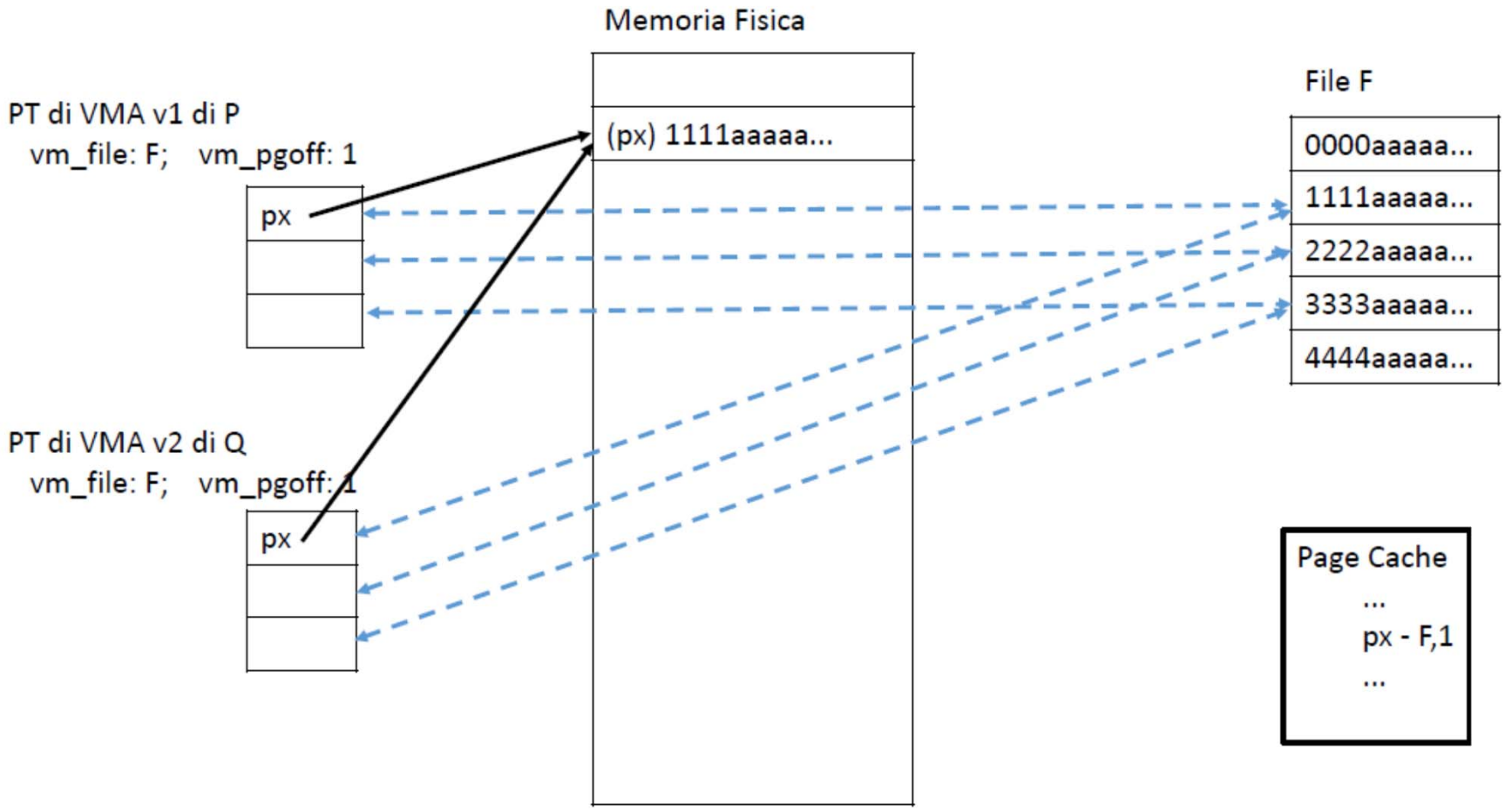
quando un processo richiede di accedere una pagina virtuale mappata su un file, il sistema svolge le operazioni seguenti (con riferimento all'esempio considerato in precedenza):

- determina il file e il page offset richiesto
  - il file è indicato nella VMA («F» nell'esempio)
  - 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 nella prima lettura, perché la variabile base punta alla prima pagina della VMA – quindi nell'esempio si trova la coppia <F, 1>)
- verifica se la pagina esiste già nella *Page Cache*; in caso affermativo la pagina virtuale viene semplicemente mappata su tale pagina fisica
- altrimenti alloca una pagina fisica nella *Page Cache* e vi carica la pagina del file cercata

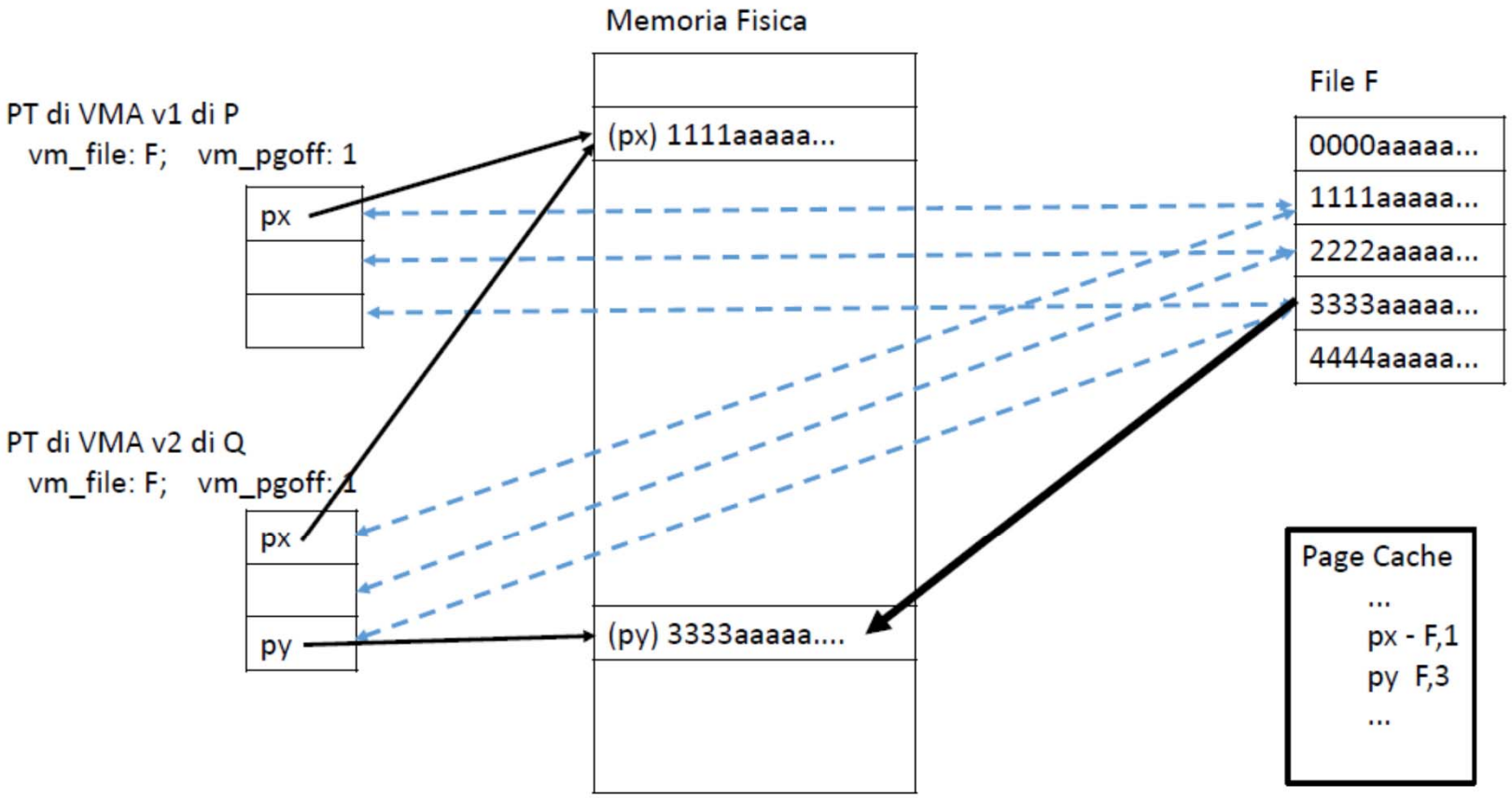
esempio 1 – il processo P esegue la 1a lettura



esempio 1 – il processo Q esegue la 1a lettura



esempio 1 – il processo Q esegue la 2a lettura



scrittura nelle pagine di una VMA SHARED

- 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
    - infatti non è 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)
```

esempio 2

- modifichiamo l'esempio 1 in modo che il processo P, invece di leggere, scriva 10 caratteri X dopo i 4 uni della prima pagina
- a tale scopo aggiungiamo il codice seguente

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

- il secondo processo (Q) produce questo risultato

  processo 2 – prima lettura

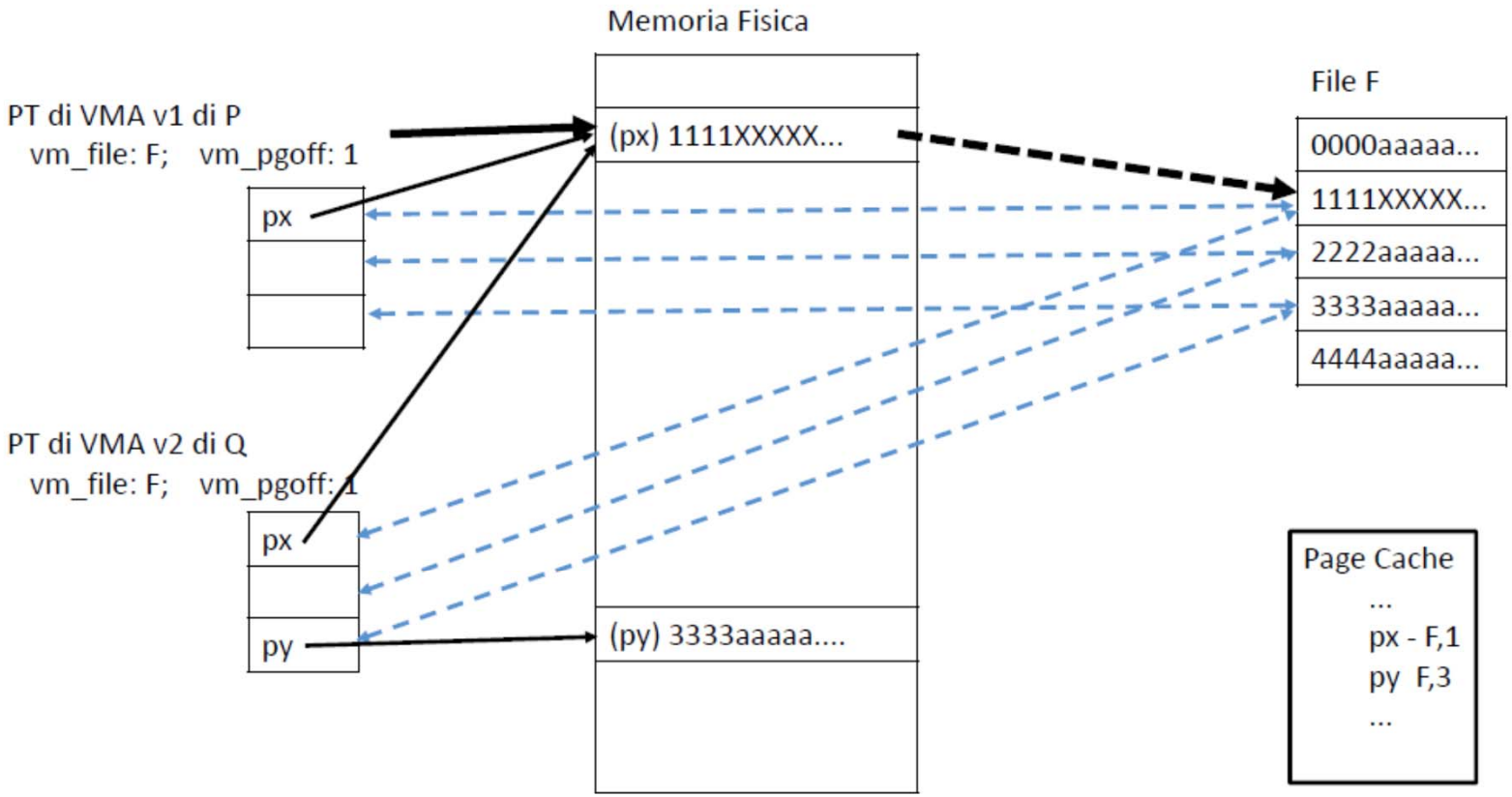
  1 1 1 1 X X X X X X

  processo 2 – seconda lettura

  3 3 3 3 a a a a a a

esempio 2

(la freccia tratteggiata indica che la scrittura sul file è differita)



scrittura in VMA PRIVATE – *Copy-On-Write* (COW)

- la scrittura in una pagina NPV allocata in una pagina fisica PFx di una VMA di tipo PRIVATE si basa sulle seguenti regole
  - la pagina PFx viene duplicata fisicamente allocando una nuova pagina fisica PFy
  - la scrittura viene applicata solamente alla copia PFy
  - la pagina originale e il file rimangono inalterati
  - NPV non risulta più mappata sul file
  - NPV non è più condivisa dagli eventuali processi che la condividevano
- per realizzare questo meccanismo, detto ***Copy-On-Write* (COW)**, *è necessario che il sistema intercetti le scritture su pagine di VMA private;* ciò è ottenuto tramite il seguente accorgimento
  - la protezione delle pagine di una VMA PRIVATE scrivibile viene posta inizialmente a R (non scrivibile)
  - l'eventuale scrittura causa un Page Fault per violazione di protezione
  - il *Page Fault Handler* scopre questa situazione e attua le operazioni necessarie

algoritmo dello *Page Fault Handler* con COW (accesso a NPV)

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)

invoca la routine che deve caricare in memoria la pagina virtuale NPV

esempio 3

- modifichiamo l'esempio 2 creando la VMA del processo P con tipo PRIVATE, come nel codice seguente

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

- il processo Q rilegge le pagine DOPO la scrittura di P
- in questo caso il risultato prodotto dai processi è:

  processo 1 – scrittura

  processo 2 – prima lettura

  1 1 1 1 a a a a a a

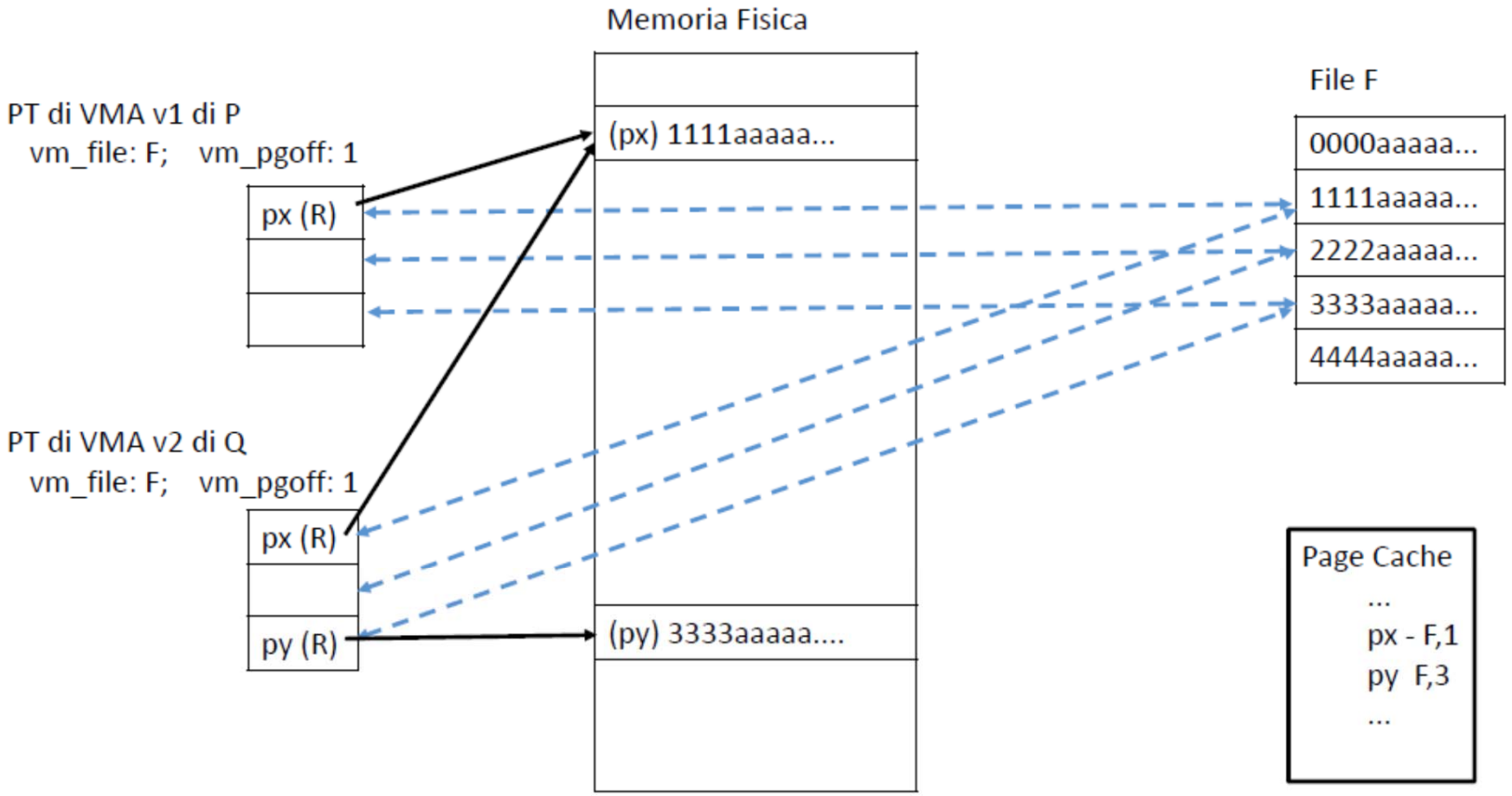
  processo 2 – seconda lettura

  3 3 3 3 a a a a a a

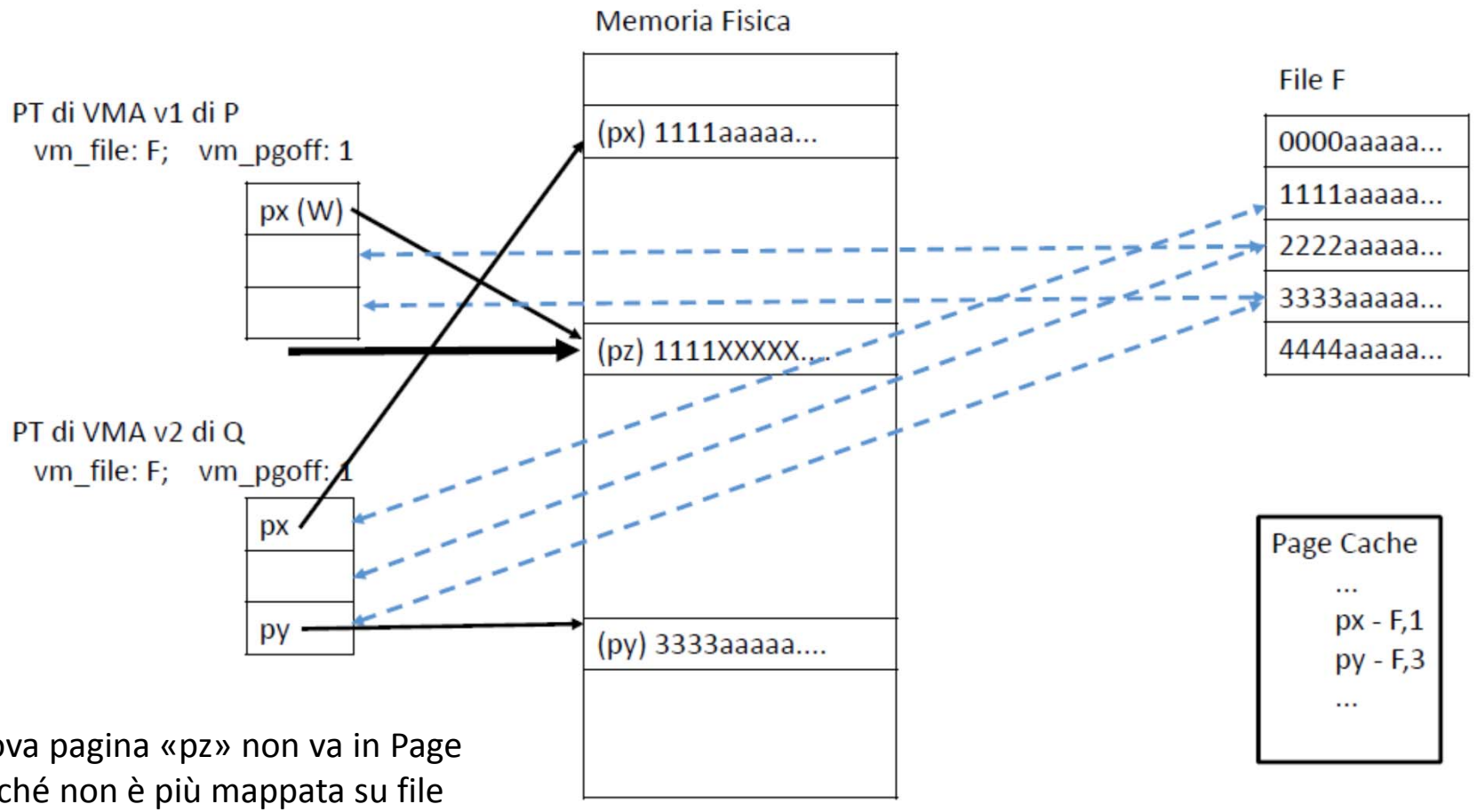
  → il processo 2 non osserva l'effetto della scrittura

esempio 3 – subito dopo la creazione delle VMA e le prime letture da parte dei due processi

(si noti che la protezione è posta a R)



esempio 3 – dopo la scrittura da parte di P (COW effettuato)



NB: la nuova pagina «pz» non va in Page Cache poiché non è più mappata su file

crescita e decrescita della *Page Cache*

- 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
- 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*)
  - si osservi che in questo caso *ref_count* = 1
- 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 fisiche da parte dei processi su una memoria quasi piena, e verrà trattata nel capitolo relativo alla gestione della memoria fisica

VMA di tipo anonimo (implicitamente PRIVATE)

- 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)
  - la 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

applicazione alle aree standard di un processo

- i meccanismi visti non sono applicati solo per realizzare le VMA richieste esplicitamente da un processo tramite *mmap*, ma in generale sono utilizzati dal sistema operativo per gestire le aree virtuali dei processi
- sotto questo punto di vista possiamo suddividere le VMA di un processo nel modo seguente:
  - VMA mappate sull'eseguibile: C, K e S
  - VMA anonime: D, T e P
  - VMA di vario tipo create su richiesta non solo del programma eseguibile, ma anche del SO (ad esempio, librerie dinamiche mappate sui rispettivi eseguibili)
- ricordiamo che il ***demand paging*** (cioè l'allocazione fisica delle pagine solo quando sono accedute) è realizzato dai meccanismi generali delle VMA
- ad esempio, una pagina di codice verrà allocata in memoria fisica solo quando verrà letta (ossia messa in esecuzione) dal processo

VMA mappate sull'eseguibile

- come si vede dall'estratto della mappa riportato, sono tutte di tipo PRIVATE
- per C (codice) e K (costanti) questa scelta è indifferente, poiché non sono scrivibili
- l'area S (dati statici inizializzati) deve essere ovviamente di tipo PRIVATE, poiché la scrittura non deve modificare il file eseguibile e non deve essere osservabile tra processi che eseguono lo stesso programma

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
…					

esempio

```
#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

output

NPV of main 000000000404

NPV of printf 000000000400

```
[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

esempio

```
#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 ( )      // era commentato nella prima prova
        VIRTUAL_AREAS
        return
}
```

a) codice del programma con funzione *long_code* eseguita

output

```
NPV of main   000000000404
NPV of printf 000000000400
```

```
[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
```

condivisione delle aree mappate sull'eseguibile

- se due processi eseguono contemporaneamente lo stesso programma, le pagine mappate sull'eseguibile sono inizialmente 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 processo precedente, *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
- le pagine dei dati statici sono duplicate al momento della scrittura tramite il meccanismo COW

librerie dinamiche

- il linker dinamico utilizza le VMA per realizzare la condivisione delle pagine fisiche delle librerie condivise (ad esempio, vedi sotto *glibc*)
- il linker dinamico 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 sono più condivise con gli altri processi e con il file

start – end	perm	offset	device	i-node	file name
…					
7ffff7a1c000 – 7ffff7bd0000	r-xp	000000	08:01	271666	.../libc-2.15.so
7ffff7bd0000 – 7ffff7dcf000	---p	1b4000	08:01	271666	.../libc-2.15.so
7ffff7dcf000 – 7ffff7dd3000	r--p	1b3000	08:01	271666	.../libc-2.15.so
7ffff7dd3000 – 7ffff7dd5000	rw-p	1b7000	08:01	271666	.../libc-2.15.so
…					

area di pila

- per la pila il sistema crea una VMA con le seguenti caratteristiche
  - anonima
  - con un certo dimensionamento iniziale (34 pagine nel sistema sperimentato)
  - flag GROWSDOWN attivo
- il flag GROWSDOWN indica che la VMA può crescere quando viene acceduta l'ultima pagina disponibile (pagina di growsdown)
- le pagine vengono allocate in base alle richieste di accesso, che possono arrivare in un ordine qualsiasi
- infatti la gestione della memoria non ha la nozione di pila in senso stretto, ma conosce solo le proprietà generali della VMA

esempio: TP della pila

```
#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
}
```

TP relative alla pila

```
[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
```

osservazioni

- la VMA di pila è cresciuta automaticamente (growsdown)
- la prima pagina virtuale della VMA (pagina di growsdown) non è mai allocata, anche quando la pila viene riempita
- la macro VIRTUAL AREAS indica l'operazione di stampa della TP
- la macro è invocata nel *main*, quando la funzione è terminata
- quindi la pila si è logicamente svuotata, ma le pagine sono rimaste allocate

→ esiste un meccanismo di crescita automatica della VMA di pila, ma non di decrescita – quando la pila decresce le pagine rimangono allocate (ma verranno sovrascritte da una successiva ricrescita della pila)

riassunto delle proprietà della VMA di pila

prove ulteriori permettono di completare l'analisi del comportamento della VMA di pila (vedi variazioni 2, 3 e 4 di esempio 4 sulla dispensa M2):

- al momento della *exec* la VMA di pila viene creata con un certo numero di NPV iniziali, non allocate fisicamente tranne quelle utilizzate immediatamente (tipicamente la prima o le prime due)
- tale area iniziale può essere acceduta in qualsiasi posizione, anche secondo schemi non conformi alle pile
- ogni accesso a NPV non allocate fisicamente ne causa l'allocazione
- l'accesso a NPV iniziale (pagina di growsdown) causa una crescita automatica dell'area
- il tentativo di accedere pagine diverse da quelle esistenti causa un Segmentation Fault

algoritmo completo dello *Page Fault Handler*

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 una pagina fisica 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)

... gestisci COW (con ref_count, vedi alg. precedente) ...

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 page di una VMA che possiede il flag GROWSDOWN) {

aggiungi una nuova pagina virtuale NPV – 1 all'area virtuale, che diventa la nuova start address page della VMA

}

invoca la routine che deve caricare in memoria la pagina virtuale NPV

area per i dati dinamici (D)

- per l'area D il sistema crea una VMA con le seguenti caratteristiche:
  - anonima
  - dimensione iniziale uguale a quella dei dati statici non inizializzati (BSS) indicati nell'eseguibile
- l'area può crescere grazie al servizio *brk*
  - la funzione *malloc* ha bisogno di richiedere al SO di allocare nuovo spazio tramite l'invocazione del servizio di sistema ***brk ( )*** o ***sbrk ( )***
  - *brk ( )* e *sbrk ( )* sono due funzioni di libreria che, in forma diversa, invocano lo stesso servizio; analizziamo solamente la forma *sbrk*, che ha la seguente sintassi:

    * **`void sbrk (int incremento)`**

      incrementa l'area dati dinamici del valore incremento, arrotondato a un limite di pagina

      restituisce un puntatore alla posizione iniziale della nuova area – *sbrk* (0) restituisce il valore corrente della cima dell'area dinamica

esempio

```
#define  MAX_PAGES 3
#define  PAGE_SIZE 1024 * 4
int main (long argc,  char * argv [ ],  char * envp [ ])  {
        sbrk (PAGE_SIZE * MAX_PAGES)
        * brk = 1
        VIRTUAL_AREAS
        return
}
```

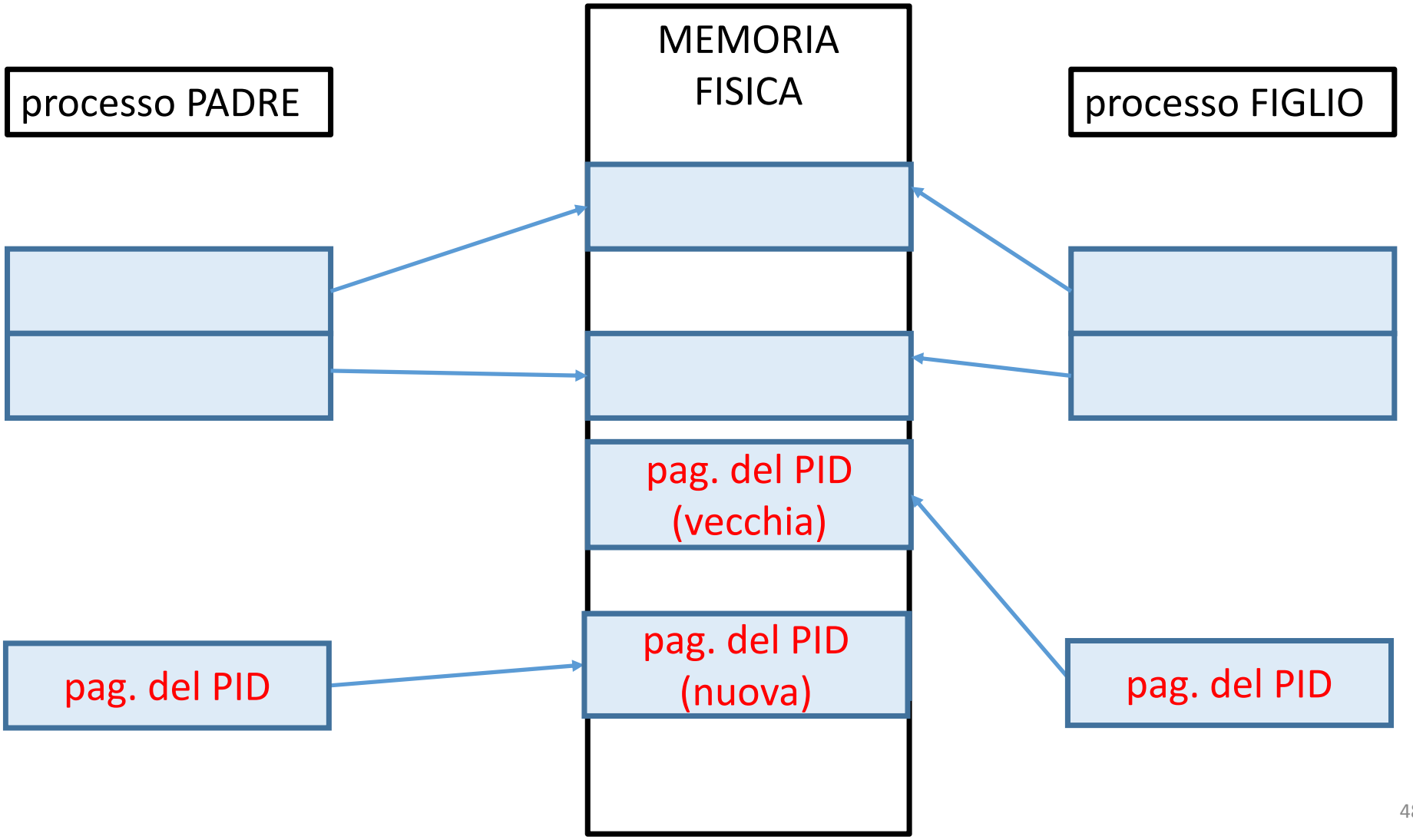
TP dell'area D

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

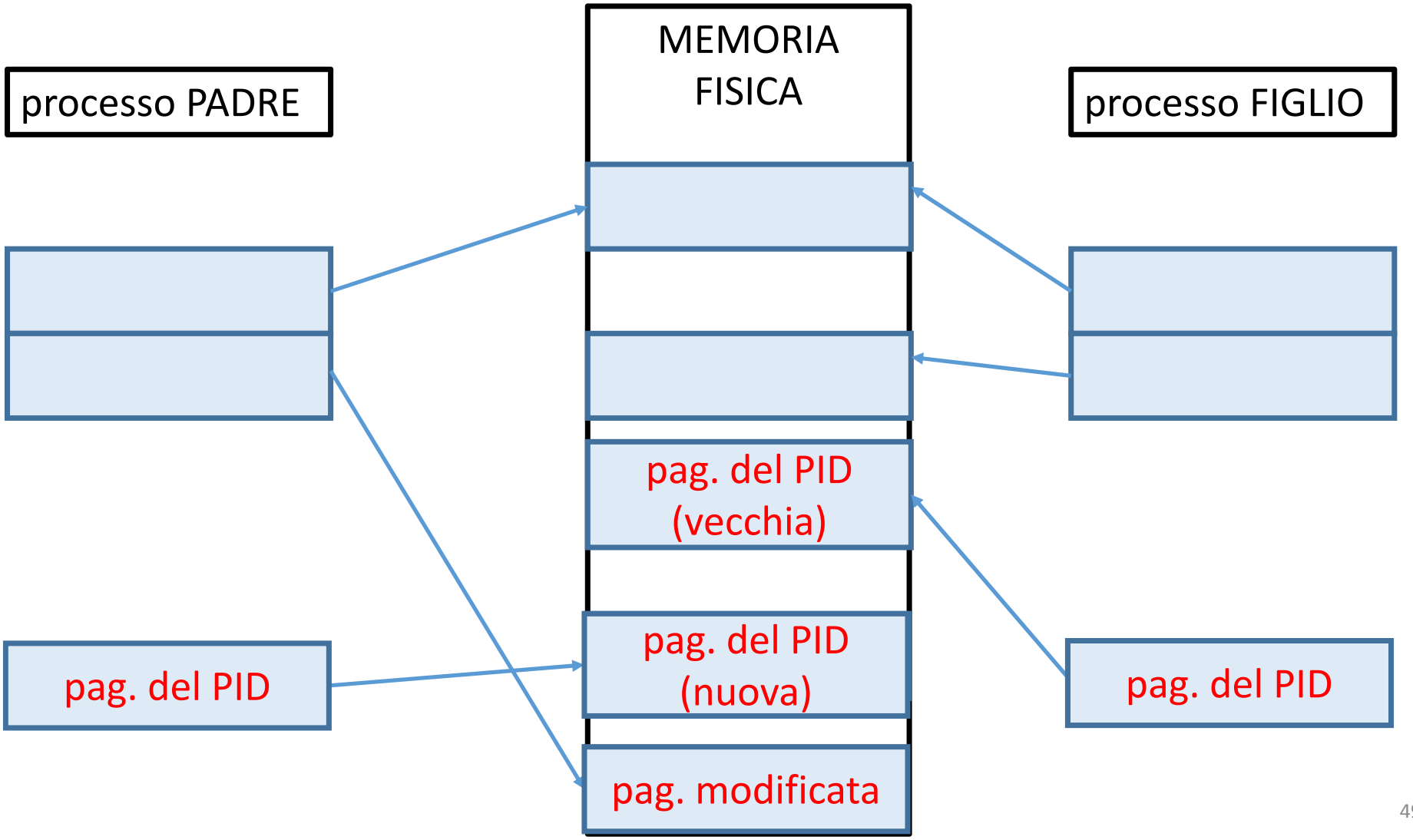
fork e gestione della memoria

- l'esecuzione di una *fork* non crea una copia delle pagine fisiche del processo padre
  - tutte le pagine sono condivise tra processo padre e figlio
  - viene applicato il meccanismo COW per duplicare le pagine che vengono scritte
  - quindi tutte le pagine vengono inizialmente marcate R (sola lettura)
- due pagine vengono poi scritte ed effettivamente duplicate da COW
  - la pagina in cima alla pila, dove *fork* restituisce un valore
  - la pagina che contiene la variabile a cui viene assegnato il valore restituito da *fork* (nei nostri esempi in generale supporremo che si tratti della stessa pagina)
- nel sistema di riferimento, nel quale il processo padre prosegue l'esecuzione, la pagina che viene duplicata è quella del padre

memoria subito dopo *fork*



dopo che il processo padre ha modificata un’altra pagina



mmap e *fork*

le aree create tramite *mmap* si trasmettono ai processi figli dopo una *fork*

i COW dovuti alla *mmap* e alla *fork* si combinano correttamente grazie al *ref_count*

```
int main  (long argc,  char * argv [ ],  char * envp [ ])  {
        unsigned long mapaddr = 0x100000000
        address = mmap (mapaddr, PAGESIZE * 3, PROT_READ | PROT_WRITE,
                        MAP_PRIVATE | MAP_ANONYMOUS, −1, 0)          // mmap eseguito da padre
        printf ("%c ", *address)                                     // lettura 1
        *address = 'x'                                               // scrittura 1
        pid = fork ( )                                               // padre crea figlio 1
        if  (pid == 0)  {                                            // figlio 1
                printf ("%c ", *address)                             // lettura 2
                *address = 'x'                                       // scrittura 2
        }  else  {                                                   // padre
                pid = fork ( )                                       // padre crea figlio 2
                if  (pid == 0)  {                                    // figlio 2
                        printf ("%c ", *address)                     // lettura 3
                        *address = 'x'                               // scrittura 3
                 }  else  {                                          // padre
                        printf ("%c ", *address)                     // lettura 4
                        *address = 'x'                               // scrittura 4
                 }
…                                                                    // parentesi chiuse “} “varie ...
```

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		

zero page del SO

anche qui COW, ma con la *zero page*

NB: sono pagine di una VMA ANONYMOUS (non mappata su file – vedi *mmap* prima), non stanno in *Page Cache* e il *ref_count* non ha il + 1

exec e gestione della memoria

- l'esecuzione della funzione *exec* azzera tutta la situazione della memoria
- la struttura delle VMA viene ricostruita in base al contenuto del file eseguibile e vengono predisposte le PTE necessarie nella TP
- vengono caricate in memoria fisica
  - la pagina di codice contenente la prima istruzione eseguibile
  - la prima pagina di pila nella quale vengono posti i parametri passati al main – (*argc*, *argv* e *env*)
  - il programma inizia l'esecuzione
- tramite la tecnica di ***demand paging*** vengono caricate le pagine di codice e dati del programma lanciato in esecuzione

clone e pila dei *thread*

- i processi leggeri che rappresentano dei *thread* condividono la stessa struttura di aree virtuali e TP del processo padre (*thread* principale)
- per ogni *thread* viene allocata una pila con le seguenti caratteristiche:
  - dimensione: 2048 pagine, cioè esattamente 8 M byte (2048 = 0x 800)
  - più una pagina di interposizione in sola lettura (contro sconfinamento)
- la pila del primo *thread* inizia logicamente al NPV **7FFFF77FF** e si sviluppa verso indirizzi più bassi, quindi la sua VMA è compresa tra
  - start **7FFFF6FFF** (infatti 6FFF + 800 = 77FF)
  - end **7FFFF77FF**
- quindi la pila del secondo *thread* inizia logicamente al NPV **7FFFF6FFE** e si sviluppa verso indirizzi più bassi, quindi la sua VMA è compresa tra
  - start **7FFFF67FE**
  - end **7FFFF6FFE** (= 7FFFF6FFF – 1 per pagina di interposizione)