Giuseppe Pelagatti

Programmazione e Struttura del sistema operativo Linux

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

Parte M: La gestione della Memoria

cap. M2 – Organizzazione dello spazio virtuale

M.2 Organizzazione dello spazio virtuale

La memoria virtuale del Sistema Operativo e dei processi in LINUX è strutturata in base a diverse esigenze. Nel seguito analizziamo prima la struttura del Sistema operativo, poi quella dei processi.

1. Struttura della memoria virtuale del Sistema Operativo

Gli indirizzi virtuali del Kernel si estendono come abbiamo visto

da `FFFF 8000 0000 0000`

a `FFFF FFFF FFFF FFFF`

rendendo disponibile uno spazio di indirizzamento virtuale di 2^{47} byte (128 Terabyte).

La strutturazione interna di questo spazio è rappresentata in Tabella 1. Osserviamo che:

- il codice e i dati del sistema operativo occupano solamente 0,5 Gb (512 Mb), cioè una piccolissima percentuale dell'intero spazio virtuale disponibile
- per i moduli a caricamento dinamico sono riservati 1,5 Gb (3 volte il codice statico; il sistema cresce infatti principalmente in forma di nuovi moduli)
- un'area enorme di 2^{46} byte = 64 Terabyte, cioè metà dell'intero spazio virtuale disponibile, è riservata alla mappatura della memoria fisica, cioè a permettere al codice del Kernel di accedere direttamente a indirizzi fisici (meccanismo spiegato più avanti)
- un'area grande viene riservata per le strutture dinamiche del Kernel
- l'area indicata come mappatura della memoria virtuale è utilizzata per ottimizzare particolari configurazioni discontinue della memoria, e non verrà discussa qui
- esistono varie altre porzioni non utilizzate e lasciate per usi futuri o utilizzate per scopi che non vengono trattati qui

Area	Sotto aree	Costanti Simboliche che definiscono gli indirizzi iniziali	Indirizzo Iniziale (solo inizio)	Indirizzo Finale (solo inizio)	Dimensione
	spazio inutilizzato		`ffff8000..`		
Mappatura Mem. Fisica		PAGE_OFFSET	`ffff8800..`	`ffffC7ff..`	64 Tb
	spazio inutilizzato				1 Tb
Memoria dinamica Kernel		VMALLOC_START	`ffffc900…`	`ffffe8ff…`	32 Tb
	spazio inutilizzato				
Mappatura memoria virtuale		VMEMMAP_START	`ffffea00..`		1 Tb
	spazio inutilizzato				
Codice e dati		_START_KERNEL_MAP	`ffffffff 80..`		0,5 Gb
	codice	_text			
	dati inizializzti	_etext			
	dati non inizializzati	_edata			
Area per caricare i moduli		MODULES_VADDR	`ffffffff a0…`		1,5 Gb

Tabella 1

Nella interpretazione degli indirizzi di memoria è sufficiente tenere presente che fondamentalmente lo spazio di indirizzamento del Kernel è suddiviso in 5 grandi aree, come mostrato in Tabella 2, che riassume Tabella 1.

Area	**Costanti Simboliche per indirizzi iniziali**	**Indirizzo Iniziale**	**Dim**
Mappatura Mem. Fisica	PAGE_OFFSET	`ffff 8800..`	64 Tb
Memoria dinamica Kernel	VMALLOC_START	`ffff c900..`	32 Tb
Mappatura memoria virtuale	VMEMMAP_START	`ffff ea00..`	1 Tb
Codice e dati	_START_KERNEL_MAP	`ffff ffff 80..`	0,5 Gb
Area per caricare i moduli	MODULES_VADDR	`ffff ffff a0..`	1,5 Gb
Tabella 2			

Buona parte di questi indirizzi e dimensioni sono definiti nel file `Linux/arch/x86/include/asm/page_64_types.h` .

PAGE_OFFSET - Accesso agli indirizzi fisici da parte del SO

Nella gestione della memoria il SO deve essere in grado di utilizzare gli indirizzi fisici, anche se, come tutto il Software, esso opera su indirizzi virtuali. Un esempio significativo di questa esigenza è il seguente: nella Tabella delle Pagine gli indirizzi sono fisici, perché vengono utilizzati direttamente dall'Hardware nell'accesso alla memoria. Per operare sulla TP il SO deve quindi essere in grado di accedere alla memoria anche tramite indirizzi fisici.

Questo problema è risolto dedicando una parte dello spazio virtuale del SO alla mappatura 1:1 della memoria fisica (Figura 1). L'indirizzo iniziale di tale area virtuale è definito dalla costante PAGE_OFFSET, il cui valore varia nelle diverse architetture.

In pratica questo significa che l'indirizzo virtuale PAGE_OFFSET corrisponde all'indirizzo fisico 0 e la conversione tra i 2 tipi di indirizzi è quindi

indirizzo fisico = indirizzo virtuale - PAGE_OFFSET
indirizzo virtuale = indirizzo fisico + PAGE_OFFSET

Nel codice del SO esistono funzioni che eseguono la conversione tra indirizzi fisici e virtuali dell'area di rimappatura con gli opportuni controlli. Ad esempio, la funzione

```
unsigned long __phys_addr(unsigned long x)
```

esegue una serie di controlli e se tutto va bene restituisce x – PAGE_OFFSET.

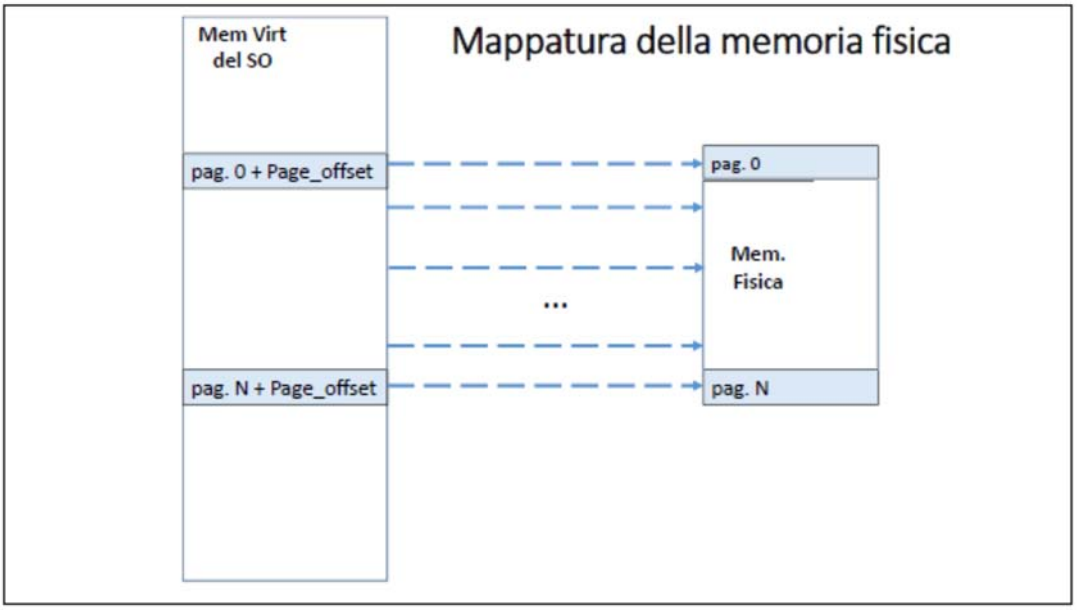

Figura 1

In un capitolo precedente abbiamo stampato gli indirizzi limite della sPila di un processo, che erano

- inizio `0xFFFF 8800 5C64 4000`
- fine `0xFFFF 8800 5C64 6000`

Confrontandoli con la mappa fornita in Tabella 1 vediamo che tali indirizzi si trovano nell'area di mappatura fisica del nucleo; infatti le aree di sPila dei processi sono allocate dinamicamente dal SO e il loro indirizzo fisico viene ottenuto al momento di tale allocazione - tale indirizzo fisico viene trasformato in un indirizzo virtuale dell'area di mappatura della memoria fisica per essere utilizzato dal sistema.

2. Struttura della memoria virtuale 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 eseguibile.

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

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

Per questi motivi la memoria virtuale di un processo LINUX è suddivisa in un certo numero di a**ree di memoria virtuale** (**virtual memory area**).

Ogni area di memoria virtuale è caratterizzata da una coppia di indirizzi virtuali che ne definiscono l'inizio e la fine. Dato che le aree virtuali devono essere costituite da un numero intero di pagine, in realtà la loro definizione richiede una coppia $NPV_{iniziale}$, NPV_{finale}.
I tipi più importanti di aree di memoria virtuale di un processo sono i seguenti:

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

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

In figura 1 è mostrata la tipica struttura di un processo in LINUX al momento del lancio in esecuzione del programma (tramite il servizio exec).

ATTENZIONE: a differenza del testo H-P, in Figura 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…

Alcune aree, in particolare la pila e l'area dei dati dinamici, iniziano con uno spazio di dimensione casuale, calcolata ogni volta che viene creata la memoria del processo, lasciato libero allo scopo di rendere più difficili gli attacchi informatici (tecnica chiamata *randomizzazione degli indirizzi*). Purtroppo questo fenomeno rende anche più difficile l'analisi delle mappe di memoria.

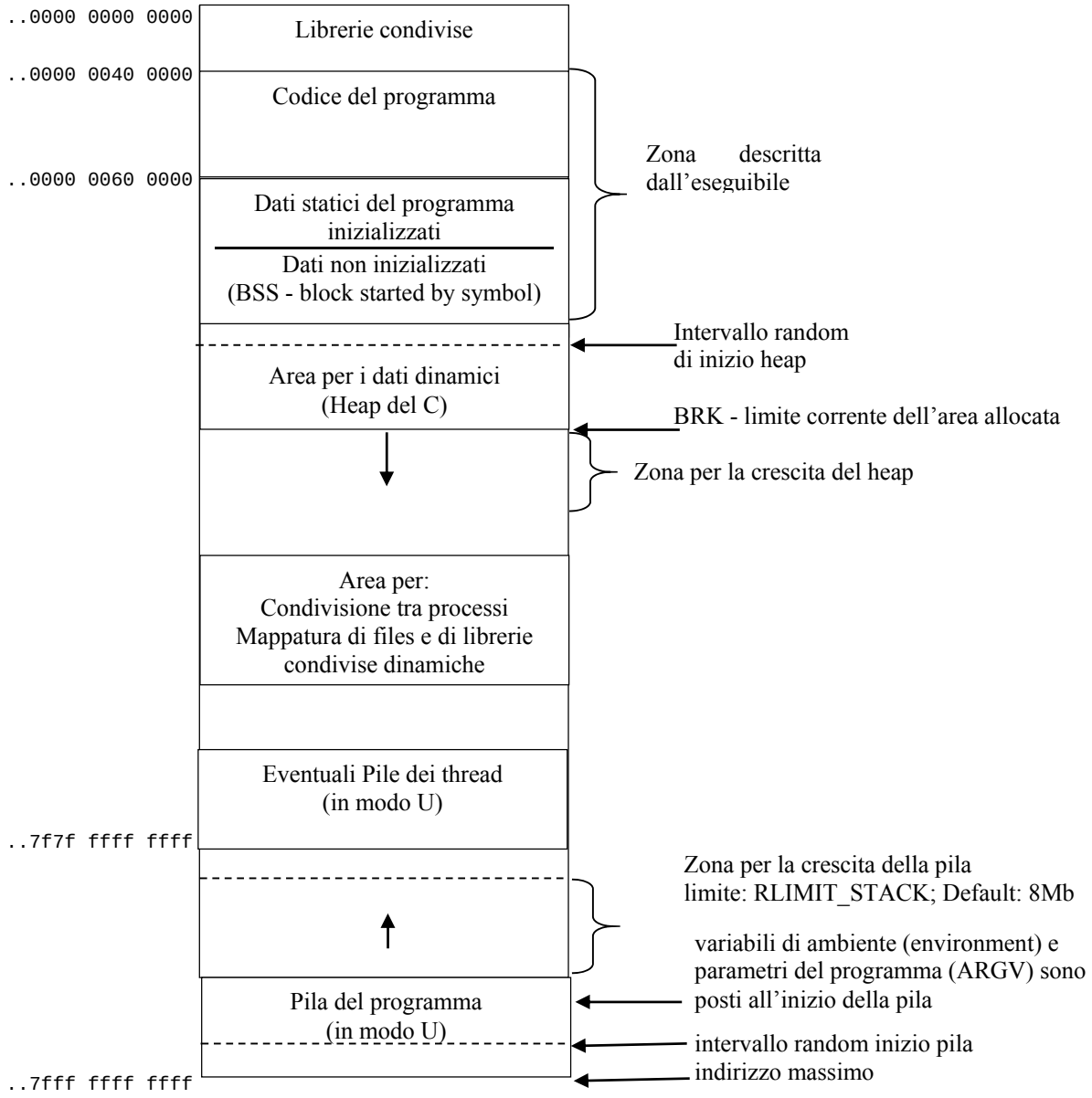

Figura 1 – Struttura di un programma eseguibile –sono indicati 48 bit degli indirizzi

Per quanto riguarda la struttura dell'eseguibile, abbiamo:

- il codice inizia all'indirizzo 0x0000 0000 0040 0000
- i dati iniziano all'indirizzo 0x0000 0000 0060 0000

Area per le pile dei thread

Le pile per i thread vengono create in un'area che inizia all'indirizzo 7f7f ffff ffff e cresce verso indirizzi bassi. La realizzazione della funzione `thread_create` fornita nel capitolo N4, che allocava l'area di pila tramite `malloc()` era indicativa di quanto deve accadere, ma l'area di pila non è in realtà allocata nello heap.

Crescita delle aree dinamiche – servizio brk

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

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

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

**void sbrk(int incremento)*

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

```
#include <stdio.h>
int Global[10];
int main(argc, argv)
{       void f( );
        int Local[10];
        void *Brk;
        extern char **environ;
        Brk=sbrk(0);
        printf( "indirizzo esadecimale environ: 0x%16.16lx\n",environ);
        printf( "indirizzo esadecimale Local:   0x%16.16lx\n",Local);
        printf( "indirizzo esadecimale Global:  0x%16.16lx\n",Global);
        printf( "indirizzo esadecimale f:       0x%16.16lx\n",f);
        printf( "indirizzo esadecimale main:    0x%16.16lx\n",main);
        printf( "indirizzo esadecimale printf:  0x%16.16lx\n",printf);
        printf( "indirizzo esadecimale Brk:     0x%16.16lx\n",Brk);
        Brk=sbrk(4096);
        Brk=sbrk(0);
        printf( "Brk dopo l'incremento:         0x%16.16lx\n",Brk);
        }
void f( ){          }
```

a) Un programma che stampa indirizzi di memoria

```
indirizzo esadecimale environ: 0x00007fff5 eabb 6c8
indirizzo esadecimale Local:   0x00007fff5 eabb 5a0
indirizzo esadecimale Global:  0x000000000 0601 060
indirizzo esadecimale f:       0x000000000 0400 6d6
indirizzo esadecimale main:    0x000000000 0400 5c4
indirizzo esadecimale printf:  0x000000000 0400 4b0
indirizzo esadecimale Brk:     0x000000000 06ef 000
Brk dopo l'incremento:         0x000000000 06f0 000
```

b) Risultati dell'esecuzione del programma

Figura 2 – Esplorazione della struttura virtuale di un processo LINUX tramite un programma C

In figura 2a è riportato un semplice programma C che illustra, stampando in esadecimale i valori di alcuni puntatori, la struttura di un programma eseguibile in LINUX e la sua modifica tramite sbrk.

I risultati dell'esecuzione su un calcolatore x64, mostrati in figura 2b, forniscono una conferma della struttura generale di figura 1; in particolare:

- La pila è posta nella zona alta dello spazio di indirizzamento, cresce verso indirizzi più bassi;
- La pagina contenente le variabili globali è sopra la pagina del codice
- Nella pagina del codice le funzioni dell'utente sono caricate sopra il main; le funzioni di libreria sotto il main
- l'area dati dinamica è sopra l'area dati statica; la funzione `brk` ha incrementato tale area esattamente di una pagina (da ef a f0)

Gestione della memoria

Ogni area virtuale in LINUX è definita da una variabile strutturata di tipo `vm_area_struct`. (in figura 3 sono rappresentati alcuni campi selezionati di tale struttura).

I primi 3 campi hanno un significato ovvio.

Le strutture delle singole aree sono collegate tra loro in una lista (`*vm_next, *vm_prev`) ordinata per indirizzi crescenti.

I permessi di accesso dell'area sono descritti in `vm_page_prot.` Tali permessi sono read only (RO), read/write (RW) ed execute (X).

La struttura `vm_area_struct` contiene anche l'informazione (`struct file * vm_file;`) necessaria a individuare il file utilizzato come "backing store", cioè per salvare il contenuto della memoria quando è necessario, e la posizione (offset) all'interno del file stesso (`unsigned long vm_pgoff;`).

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         pgprot_t vm_page_prot;          /* Access permissions of this VMA. */
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 };
```

Figura 3

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

`fork` (Clone)

Un momento fondamentale è l'esecuzione di una `fork`, perchè la creazione di un nuovo processo implicherebbe la creazione di tutta la struttura di memoria di un nuovo processo. Dato che il nuovo processo è l'immagine del padre, LINUX in realtà si limita ad aggiornare le informazioni generali della tabella dei processi, ma non alloca nuova memoria.

L'unica pagina che viene duplicata fisicamente è quella contenente la variabile restituita dalla `fork` (il pid); nell'attuale implementazione la pagina fisica originale contenente il pid è attribuita al processo figlio, mentre per il processo padre viene allocata una pagina fisica nuova (Figura 4).

In realtà, 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 principio "**copy on write**" (Figura 5)

Per permettere di scoprire l'operazione di scrittura, a tutte le pagine, anche quelle appartenenti ad aree virtuali scrivibili, viene associata una abilitazione in sola lettura, causando quindi un errore di accesso alla prima scrittura che viene gestito opportunamente (vedi sotto, gestione degli errori).

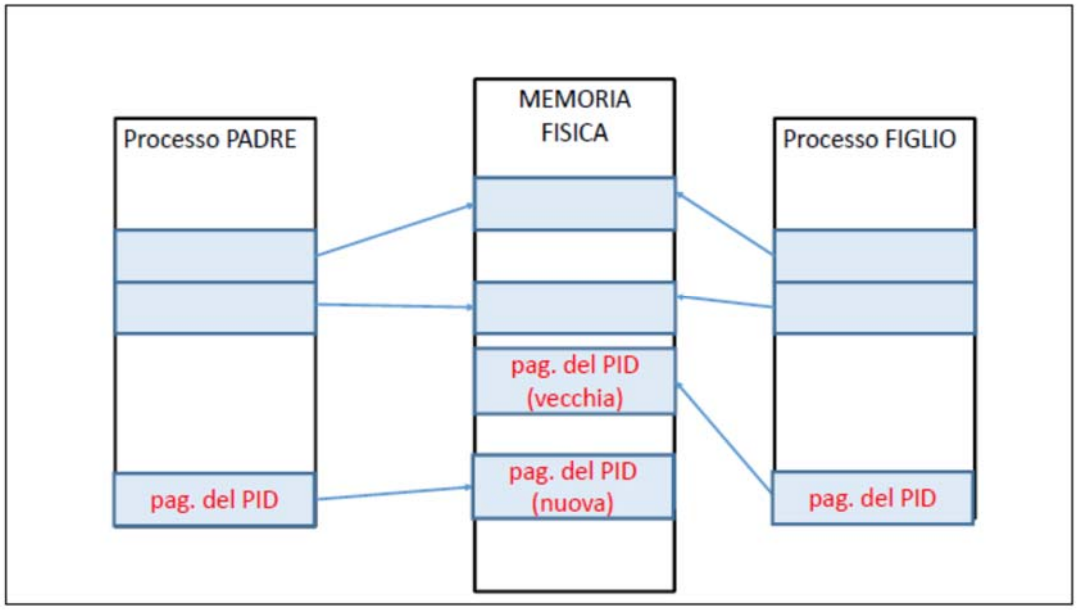

Figura 4 – padre e figlio condividono tutte le pagine tranne quella del pid

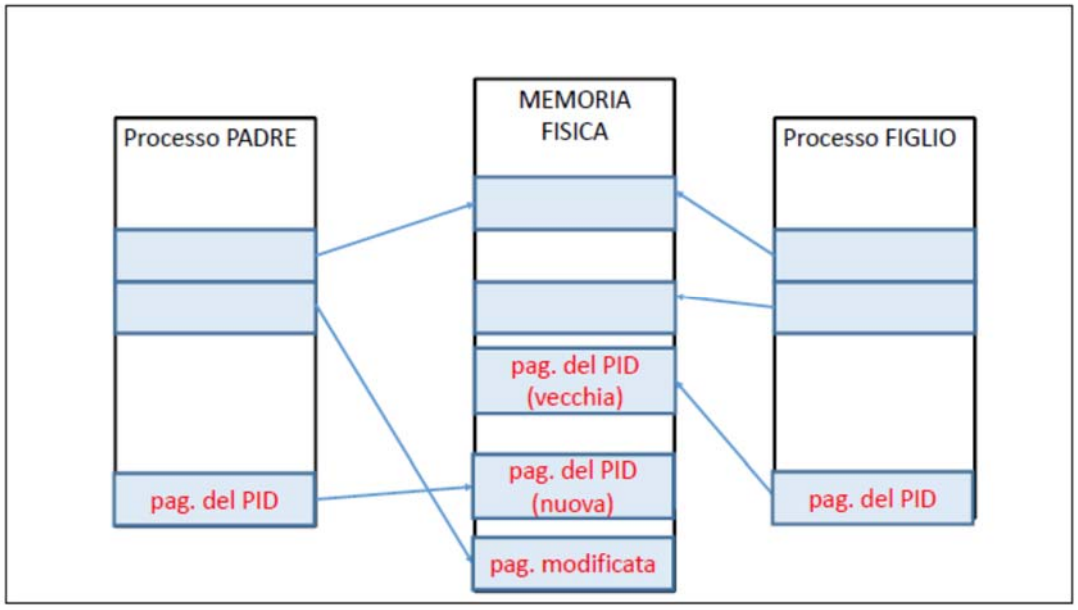

Figura 5 – il processo padre ha modificato una pagina, che viene duplicata

Exec

Ovviamente, molti programmi dopo una `fork` eseguono una `exec`. Al momento dell'`exec` LINUX invalida tutta la Tabella delle Pagine del processo e dovrebbe caricare un certo numero di pagine dal nuovo file eseguibile nella memoria. Dato che non è possibile sapere quali pagine caricare, LINUX non ne carica nessuna, applicando la tecnica del demand paging, cioè il caricamento delle pagine in base ai page fault che vengono generati.

Page Fault e altri errori di accesso alla memoria

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

1. Se V non appartiene alla memoria virtuale del processo, allora
- **1.1** se si tratta di una crescita di pila (per determinare questo fatto LINUX controlla se la pagina V+1 contiene l'indicatore growsdown) allora LINUX alloca la nuova pagina virtuale V al processo;
- **1.2** altrimenti il processo viene abortito;

2. Se V appartiene alla memoria virtuale del processo, LINUX controlla se l'accesso richiesto era legittimo rispetto ai codici di protezione:
- **2.1** se l'accesso non è legittimo,sono possibili due casi:
  - **2.1.1** se la violazione era dovuta a una pagina posta come "sola lettura" in seguito a una `fork`, ma appartenente ad un'area virtuale sulla quale la scrittura è ammessa, viene creata una nuova copia della pagina con l'indicazione di scrittura abilitata;
  - **2.1.2** altrimenti viene abortito il processo;
- **2.2** altrimenti viene invocata la routine che deve caricare in memoria la pagina virtuale V.

Le azioni "normali" sono quindi 1.1 e 2.2, che corrispondono al normale meccanismo di sostituzione di pagine nel funzionamento di un programma; le azioni 1.2 e 2.1.2 sono situazioni di errore per indirizzo errato o per violazione di protezione di accesso; l'azione 2.1.1 corrisponde a una situazione corretta gestita tramite la generazione di un errore fittizio.

Le azioni 1.1, 2.1.1 e 2.2 richiedono di allocare una nuova pagina al processo; se l'allocazione di una nuova pagina in memoria riduce il numero di pagine libere sotto una soglia minima, LINUX invoca la funzione `try_to_free_pages` del PFRA (`Page Frame Reclaiming Algorithm`) per scaricare alcune pagine su disco (argomento trattato in un capitolo successivo)