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. M4 – Sistema Operativo e Tabella delle Pagine

M.4 Sistema Operativo e Tabella delle Pagine

1. Introduzione

Anche la memoria virtuale del Sistema Operativo è gestita tramite paginazione, quindi anche gli indirizzi virtuali del Sistema Operativo vengono trasformati dalla MMU in indirizzi fisici.
A causa di tale rilocazione esistono due problemi fondamentali, collegati tra loro, che richiedono una soluzione complessa che analizzeremo in questo capitolo:

- il Sistema Operativo è anche il gestore della memoria fisica, quindi deve essere in grado di accedere talvolta alla memoria in base agli indirizzi fisici
- le Tabelle delle Pagine sono strutture dati molto grandi, accedute automaticamente dall'Hardware del x64, come vedremo, e il Sistema Operativo da un lato dipende da queste strutture per la propria rilocazione virtuale/fisica, dall'altro deve gestirle opportunamente

La soluzione di questi problemi è fortemente collegata alle funzionalità della MMU Hardware, quindi il contenuto di questo capitolo è dipendente dalla struttura della MMU del x64 alla data – è presumibile che, data la criticità di questi aspetti nuovi modelli di Hardware conterranno nuove funzionalità per semplificare questa gestione.

2. 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 2.1. 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)	Dimen sione
	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 2.1

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 2.2, che riassume Tabella 2.1.

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 2.2

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 1.1). 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.

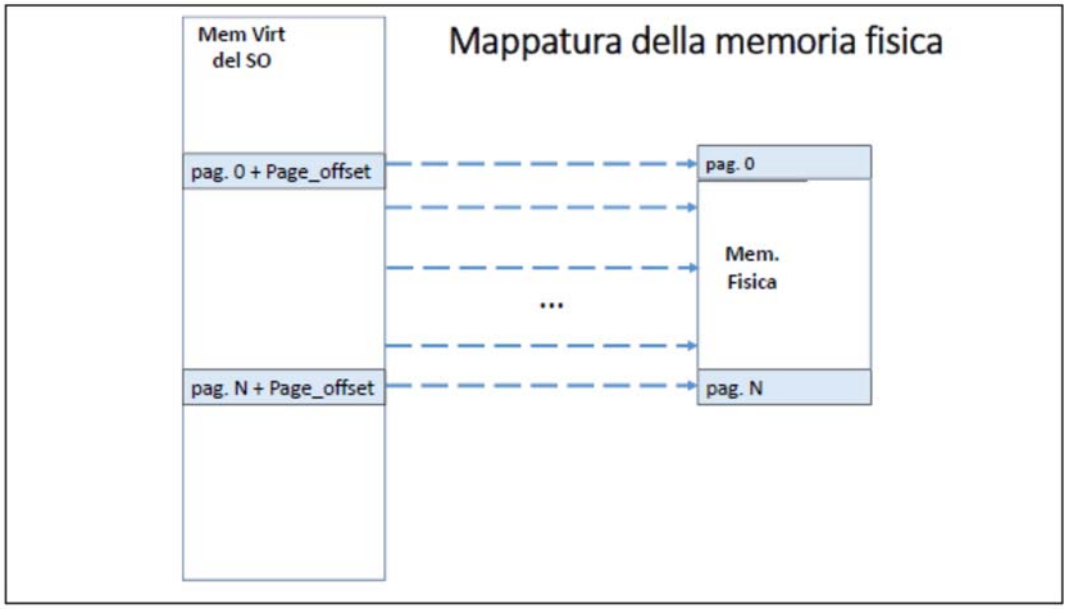

Figura 2.1

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.

3. La paginazione nel x64

Chiamiamo MMU (Memory Management Unit) l'unità che gestisce la memoria a livello Hardware. La MMU può essere un componente della CPU oppure un componente esterno.

L'indirizzo virtuale (effective address nella terminologia del x86) fa riferimento a uno spazio virtuale lineare di 2^{48} byte. La paginazione utilizza pagine di 4Kb. Pertanto la struttura dell'indirizzo si decompone in:

- 12 bit di offset
- 36 bit di NPV (2^{36} pagine virtuali),

Il numero delle pagine virtuali è molto grande, quindi è necessario evitare di allocare una tabella così grande per ogni processo. Per evitarlo la Tabella delle Pagine (TP) è organizzata come un albero su 4 livelli, nel modo seguente (vedi figura 3.1, estratta dal manuale del AMD64):

- i 36 bit del NPV sono suddivisi in 4 gruppi da 9 bit
- ogni gruppo da 9 bit rappresenta lo spiazzamento (offset) all'interno di una tabella (**directory**) contenente 512 righe (chiameremo **PTE** – **Page Table Entry** le righe di queste tabelle); si trascuri l'ulteriore nomenclatura indicata in figura, perché noi utilizzeremo quella del SO, che è diversa
- dato che ogni PTE occupa 64 bit (8 byte), la dimensione di ogni directory è di 4Kb, ovvero ogni directory occupa esattamente una pagina
- l'indirizzo della directory principale è contenuto nel registro CR3 della CPU

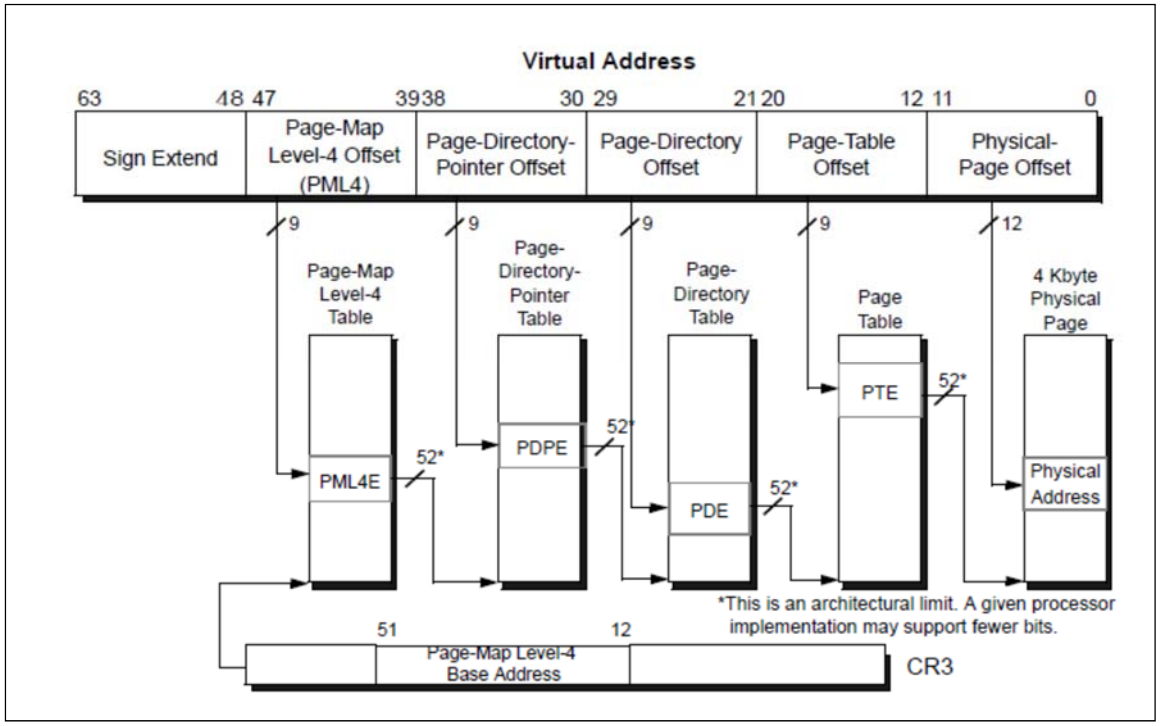

Figura 3.1

Sulla base di questa struttura il meccanismo di conversione di un NPV in un NPF si basa sui seguenti passi:

- accedi alla directory di primo livello tramite CR3
- trova la PTE indicata dall'offset di primo livello di NPV
- leggi a tale offset l'indirizzo di base della directory di livello inferiore
- ripeti la stessa procedura per i livelli inferiori

Questo schema richiedere di eseguire 4 accessi aggiuntivi a memoria per ogni accesso utile.

Si noti che gli indirizzi contenuti nei diversi livelli di directory sono ***indirizzi fisici***, perché la MMU li utilizza direttamente per accedere la memoria fisica.

Ad ogni livello la riga selezionata contiene non solo l'indirizzo del livello inferiore, ma anche un certo numero di ***flags*** che rappresentano delle proprietà della pagina utili nella gestione da parte del SO. Tali flags sono memorizzati nei 12 bit bassi, che non sono utilizzati perché non c'è offset nella Tabella delle Pagine. I principali

tipi di flag usati da Linux per le PTE della PT (cioè della directory di più basso livello) sono riportati in Tabella 3.1.

posizione	sigla	nome	interpretazione valori
0	P	present	la PTE ha un contenuto valido
1	R/W	read/write	la pagina è scrivibile: 1=W, 0=ReadOnly
2	U/S	user/supervisor	la pagina appartiene a spazio User: 1=U, 0=S
5	A	accessed	la pagina è stata acceduta (azzerabile da software)
6	D	dirty	la pagina è stata scritta (azzerabile da software)
8	G	global page	vedi sotto (gestione TLB)
63	NX	no execute	la pagina non contiene codice exeguibile

Tabella 3.1

Il flag in posizione 63 sfrutta il fatto che anche i primi bit di una PTE non servono a rappresentare un NPV e quindi possono essere utilizzati per scopi diversi.

In PTE di livello più alto i bit di controllo hanno un significato parzialmente diverso, per il quale si rimanda al manuale del AMD64.

Translation Lookaside Buffer (TLB)

All'inizio, quando il processore deve accedere un indirizzo fisico in base a un indirizzo virtuale, la MMU deve attraversare tutta la gerarchia della TP per trovare la PTE. Questa operazione, detta ***Page Walk***, richiede 5 accessi a memoria per accedere a una sola cella utile di memoria. Per evitare questo inaccettabile numero di accessi l'architettura x64 possiede un TLB nel quale sono conservate le corrispondenze NPV-NPF più utilizzate recentemente. Il TLB può essere considerato come una memoria cache associativa dedicata alla TP.

Il TLB contiene le PTE relative alle pagine più accedute di recente. In generale, se il SO non modifica la corrispondenza tra NPV e NPF, la MMU gestisce il TLB in maniera trasparente per il Software. Infatti la MMU cerca autonomamente nella TP le PTE di pagine non presenti e le carica (questa operazione richiede i 5 accessi a memoria del Page Walk). La TP può quindi essere considerata una struttura dati ad accesso Hardware (tramite CR3).

Quando viene modificato il contenuto del registro CR3 la MMU invalida tutte le PTE del TLB, escluse quelle marcate come globali (flag G).

Esistono delle istruzioni privilegiate per controllare il TLB, ad esempio per invalidare una singola PTE, che non analizziamo.

4. La Paginazione in Linux

La gestione della paginazione è una funzione che dipende in grande misura dall'architettura Hardware. Linux utilizza un modello parametrizzato per adattarsi alle diverse architetture. Linux caratterizza il comportamento dell'HW tramite una serie di parametri contenuti nei file di architettura (vedi Struttura Software, più avanti), nei quali sono descritti ad esempio:

- La dimensione delle pagine
- La struttura degli indirizzi
- Il numero di livelli della tabella delle pagine e la lunghezza dei diversi offset

Struttura della Tabella delle Pagine in Linux su x64
Nel caso del x64 il modello di Linux risulta essere estremamente aderente a quello del Hardware, come mostrato in figura 4.1.
Noi useremo nel seguito i nomi indicati in figura 4.1 per designare le varie Directory, in forma estesa oppure tramite gli acronimi ***PGD, PUD, PMD*** e ***PT***.
Per evitare un'ambiguità presente talvolta nel codice e nella documentazione di Linux useremo il termine italiano *Tabella delle Pagine (abbreviazione TP)* per indicare la struttura complessiva, e il termine inglese *Page Table (PT)* per indicare la tabella di più basso livello.
La Tabella delle Pagine è sempre residente in memoria fisica e mappa tutto lo spazio di indirizzamento del processo.

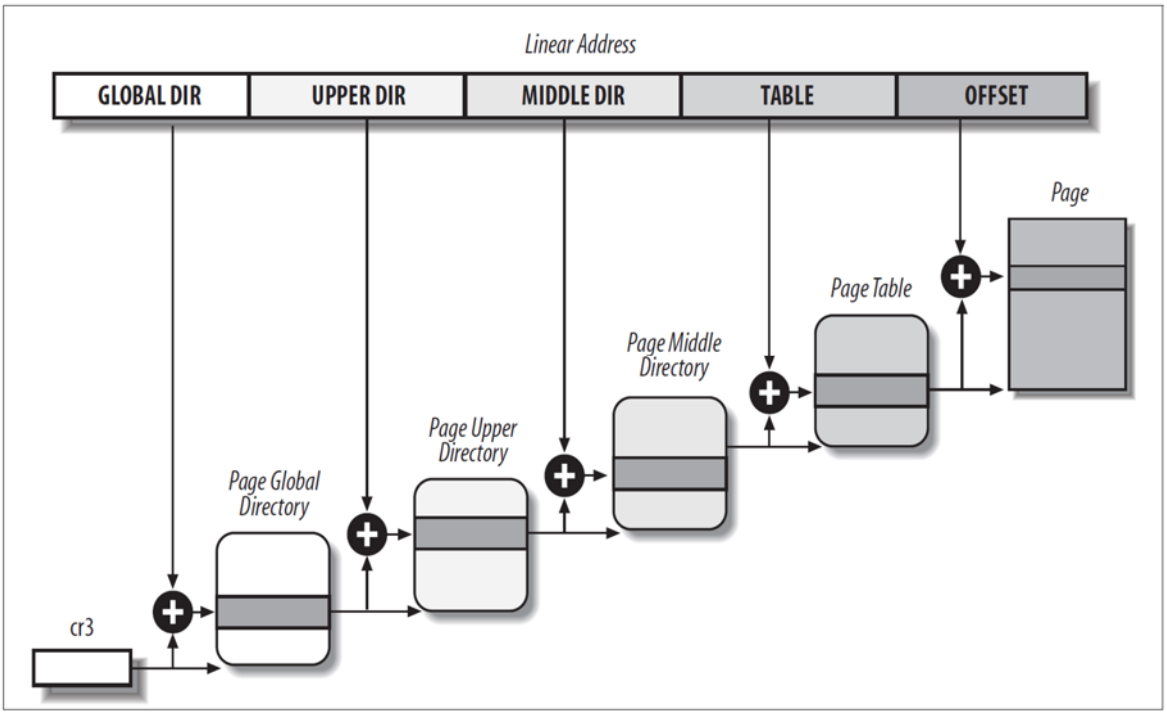

Figura 4.1

Paginazione del Sistema operativo
Dato che nel x64 tutta la memoria è paginata, indipendentemente dal modo di funzionamento della CPU, anche il SO e la stessa Tabella delle Pagine sono paginati. Questo fatto ha una serie di conseguenze:

- all'avviamento del sistema la Tabella delle Pagine non è ancora inizializzata, quindi deve esistere un meccanismo di avviamento che permetta di arrivare a caricare tale tabella per far partire il sistema
- la tabella delle pagine attiva è quella puntata dal registro CR3, che è unico, quindi non può esistere una TP separata per il SO

Avviamento
All'avviamento del sistema x86 la paginazione non è attiva. Le funzioni di caricamento iniziale funzionano quindi accedendo direttamente la memoria fisica, senza rilocazione. Quando è stata caricata una porzione della tabella

delle pagine adeguata a far funzionare almeno una parte del SO, il meccanismo di paginazione viene attivato e il caricamento completo del Kernel viene terminato.

Tabella Pagine del Kernel

Dato che non esiste una TP del SO ma solamente quella dei processi, ***il SO viene mappato dalla TP di ogni processo***. Dato che lo spazio virtuale è suddiviso esattamente a metà tra modo U e modo S, ***la metà superiore della TP di ogni processo è dedicata a mappare il SO***. Questo fatto non genera ridondanza, perché tutte le metà superiori delle TP di ogni processo puntano alla stessa (unica) struttura di sottodirectory relativi al SO, che è quindi fisicamente memorizzata una volta sola (possiamo anche dire che le metà superiori delle TP di tutti i processi sono costituite da pagine fisicamente condivise).

In Figura 4.2 è illustrata la paginazione del SO.

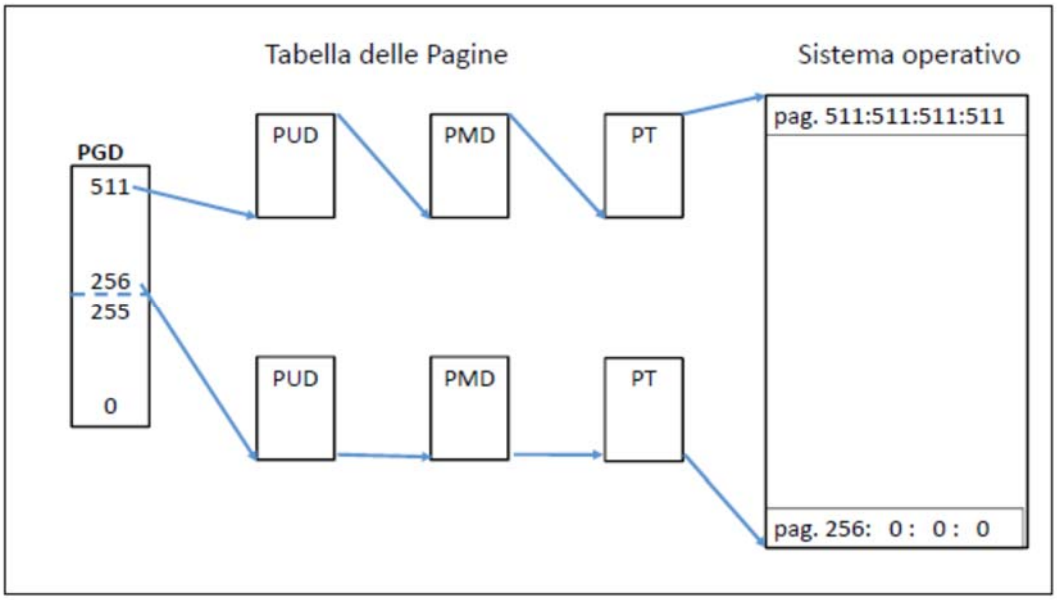

Figura 4.2

5. Dimensione della tabella delle pagine

La dimensione di memoria resa accessibile da una singola PTE ai diversi livelli della gerarchia è determinabile nel modo seguente:

- una PTE di PMD accede una pagina di PT, quindi 512 x 4K = 2M
- una PTE di PUD accede una pagina di PMD, quindi 512 x 2M = 1G
- una PTE di PGD accede una pagina di PUD, quindi 512 x 1G = 512G = 0,5T

Analizziamo alcuni esempi di occupazione di memoria da parte delle TP.

Esempio 1 (Figura 5.1)

Consideriamo un programma molto piccolo, costituito da una sola pagina di codice, una di dati e una pagina di pila, con una dimensione complessiva di 3 pagine. La struttura della corrispondente TP è mostrata in Figura 4 nella quale ogni rettangolo rappresenta una pagina di memoria; osserviamo che:

- Nel PGD, che occupa una sola pagina, sono sufficienti 2 PTE, una posta all'indice 0 per mappare codice e dati e l'altra all'indice 255 per mappare la pila (le PTE da 256 a 511 sono dedicate a mappare il SO)
- Ai livelli inferiori, fino al PMD, sono sufficienti 2 pagine per livello, e 3 pagine per la PT, quindi in totale la TP occupa 8 pagine
- il programma occupa 3 pagine
- Il rapporto tra le dimensioni della TP e quelle del programma risulta 8/3, cioè la TP occupa addirittura molto più spazio del programma

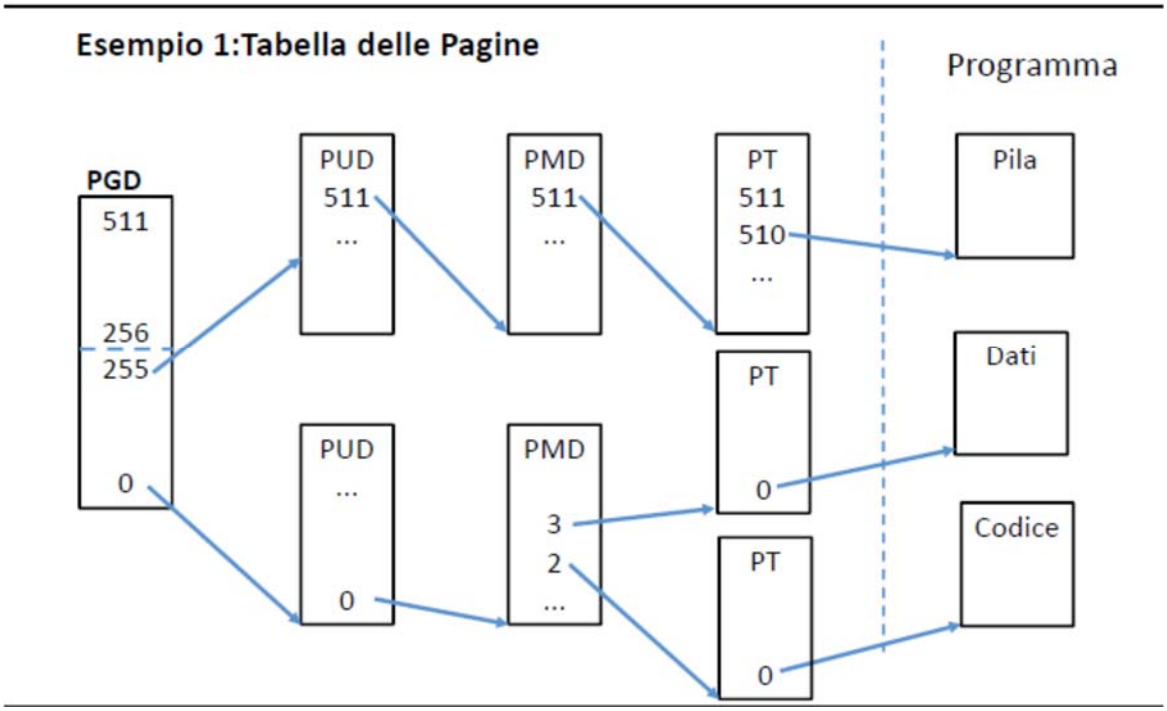

Figura 5.1

Esempio 2

Aumentiamo ora le dimensioni del programma dell'esempio precedente fino al limite massimo possibile senza aumentare le dimensioni della TP, come mostrato in Figura 5.2:

- Le tre aree del programma possono aumentare fino a richiedere l'uso di tutte le PTE (512) di ognuna delle due pagine di PT già allocate
- la dimensione del programma può quindi crescere fino a 3x512 = 1536 pagine senza aumentare il numero di pagine dedicate alla TP
- il rapporto tra le dimensioni della TP e quelle del programma risulta ora 8/1536 = 0,0052, cioè 0,52%.

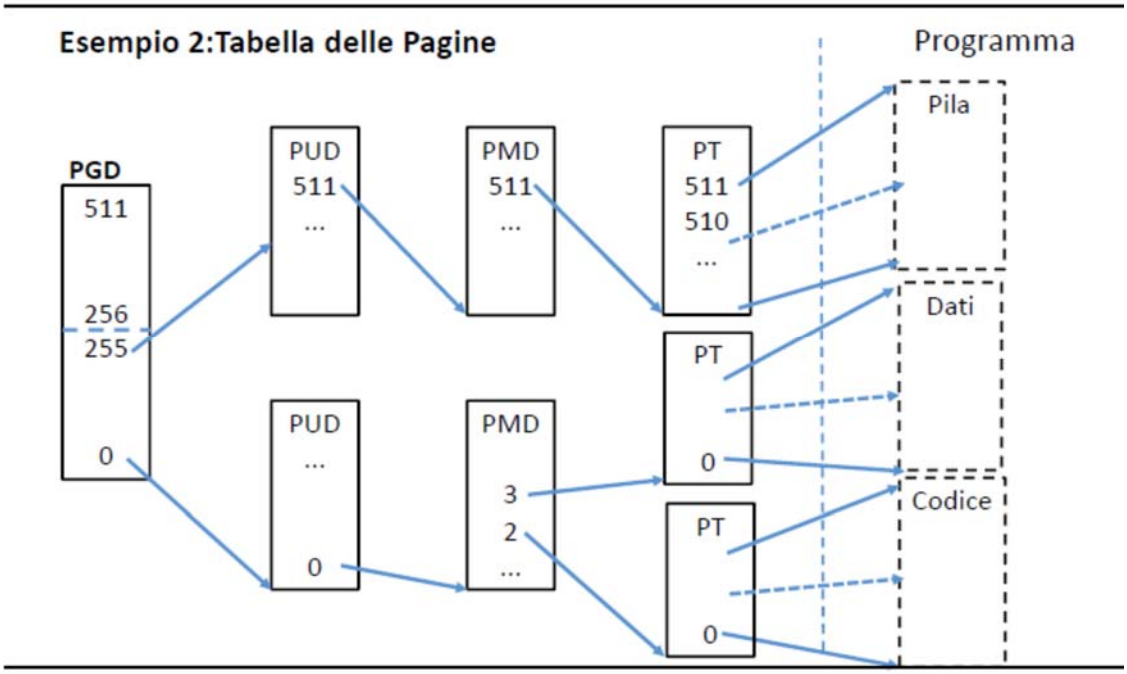

Figura 5.2

I due esempi mostrano che l'occupazione percentuale di memoria dovuta alla TP rispetto al programma diminuisce per programmi grandi fino a scendere nettamente sotto 1% già per un programma che satura solamente l'ultimo livello di directory. Aumentando ulteriormente la dimensione del programma il peso delle pagine che occupano i livelli superiori di directory diventa percentualmente sempre meno rilevante, quindi possiamo dire che il rapporto dimensionale tra TP e programma tende a quello fondamentale tra una pagina di PT e la sua area indirizzabile, cioè 1/512 = 0,00195, cioè circa lo 0,2%.
Si osservi che questo rapporto è lo stesso che esiste tra la dimensione di una pagina e quello di una PTE necessaria ad indirizzarla, cioè 4Kbyte/8byte = 1/512.

Esempio 3

Come ultimo esempio, consideriamo la porzione di TP utilizzata per rimappare la memoria fisica. La dimensione virtuale di quest'area del SO è molto grande (2^{46} byte = 64 Terabyte), ma la dimensione effettivamente mappata è determinata dalla dimensione effettiva della memoria fisica. In questo caso ci interessa il rapporto tra la dimensione della porzione di TP interessata e la dimensione della memoria fisica. Questo rapporto tende, per i motivi analizzati negli esempi precedenti, al valore 1/512. Quindi, ad esempio, su una macchina dotata di 4Gbyte di memoria (1M di pagine), la TP occupa 8Mbyte (2K pagine). In sostanza, la TP occupa uno spazio significativo, ma sempre percentualmente accettabile rispetto allo spazio disponibile.

In Figura 5.3 è rappresentata la porzione di TP utilizzata per mappare una memoria fisica di 2Mb. La figura mostra che le 3 pagine della TP sono contenute anche loro nella memoria fisica, quindi rimappano anche se stesse. La figura richiama anche l'esistenza del TLB durante un accesso a una pagina.

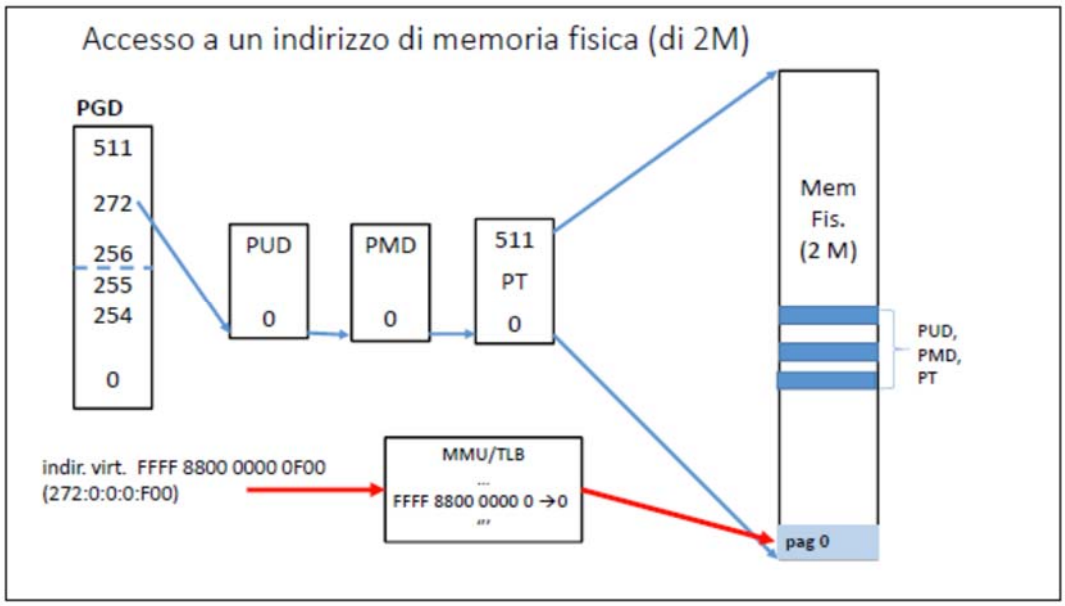

Figura 5.3

6. Gestione della TP da Software – Accesso al PGD del processo

Per iniziare a capire come Linux gestisce la TP di un processo scriviamo una funzione print_pgd che ***stampa il contenuto valido del PGD del processo corrente*** (figura 6.1a).
Ovviamente tale funzione deve essere inserita in un modulo di sistema, perché deve accedere strutture dati interne al Sistema Operativo. Il modulo si chiama axo_kpt ed ha fondamentalmente la stessa struttura del modulo axo_task già visto; tutti i dettagli relativi alla struttura modulare verranno quindi omessi per concentrarci sul modo in cui la funzione opera sulle strutture dati di sistema.
print_pgd stampa il contenuto delle PTE valide (present bit = 1) del PGD del processo corrente ed è invocata da un'altra funzione, task_explore, che ha il compito di recuperare il valore della variabile pgd del processo; tale variabile contiene l'indirizzo del PGD, cioè il valore che viene caricato nel registro CR3 quando il processo viene posto in esecuzione. A questo scopo vengono svolte le seguenti operazioni:

- ts = get_current(); recupera un puntatore al descrittore del processo
- mms = ts->mm; recupera un puntatore alla struttura che decrive la memoria del processo e lo assegna a una variabile mms di tipo struct mm_struct *
- pgd = (unsigned long int *)mms->pgd; estrae dalla struttura il campo che contiene il puntatore alla base del PGD
- print_pgd(pgd); invoca la funzione dedicata a stampare il contenuto del PGD

La funzione print_pgd contiene un ciclo nel quale scandisce, tramite una variabile intera pgd_index che assume i valori da 0 a 511, le PTE del PGD e ne stampa il contenuto solo se rappresentano pagine presenti. In particolare:

- La funzione present(unsigned long int page_entry) restituisce 1 solo se il bit di presenza della riga vale 1
- il bit di presenza è il meno significativo; per selezionarlo è stata definita la costante esadecimale PRESENT, costituita da un 1 preceduto da 63 zeri.
- Il controllo di presenza è quindi definito dall'espressione page_entry & PRESENT.

Il risultato dell'esecuzione di questo programma è mostrato in figura 6.1b.
Tutte le PTE terminano con il valore 067, perché gli ultimi 12 bit contengono i bit di controllo al posto dell'offset; la loro interpretazione non può essere fatta con i dati di Tabella 3.1, perché si tratta di righe del PGD, non della PT.

In base alla suddivisione dello spazio virtuale del x64 possiamo asserire che le prime 2 righe del PGD fanno riferimento allo spazio virtuale di modo U, le successive a quello di modo S. Le 2 righe dello spazio di modo U si posizionano all'inizio e alla fine dello spazio virtuale, coerentemente con quanto mostrato in Figura 3 e servono a rimappare le aree di codice/dati e di pila del processo

Nello spazio del sistema operativo troviamo 4 righe, che mappano le aree virtuali del sistema. Per interpretare tali aree riportiamo in Tabella 6.1 gli indirizzi iniziali delle aree di SO già visti e indichiamo l'interpretazione decimale dei primi 9 bit della porzione virtuale valida dei loro indirizzi iniziali (bit 47 – 39); infine nella colonna pgd index riportiamo da Figura 6 i valori di pgd_index stampati dal programma.

Costanti Simboliche per indirizzi iniziali	**Indirizzo Iniziale**	**Primi 9 bit in HEX**	**valore decimale**	**pgd index**
PAGE_OFFSET	`ffff 8800..`	1000 1000 0	272	272
VMALLOC_START	`ffff c900..`	1100 1001 0	402	402
VMEMMAP_START	`ffff ea00..`	1110 1010 0	468	468
_START_KERNEL_MAP	`ffff ffff 80..`	1111 1111 1	511	511
MODULES_VADDR	`ffff ffff a0..`	idem		
Tabella 6.1				

So constata che il PGD contiene nella parte relativa al Kernel i riferimenti necessari per mappare queste 4 aree (il codice del SO e dei moduli sono sotto la stessa PTE del PGD, perché sono contigui e occupano solo 2G, mentre una PTE di PGD accede 512G.

```
#include <linux/init.h>
#include <linux/module.h>
#include <linux/sched.h>
#include <linux/mm.h>
#include <linux/sched.h>
#include <linux/mm_types.h>
#include <linux/kernel.h>

#define MASK 0x00000000000001ff
#define PRESENT 0x0000000000000001

unsigned long int *pgd;

static int present(unsigned long int page_entry){
        return (page_entry & PRESENT);
}

static void print_pgd(unsigned long int * pgd){
        int pgd_index;
        printk("======= PGD DIRECTORY (solo Entries con Present=1) \n");
        for (pgd_index = 0; pgd_index < 512; pgd_index++){
                if (present(pgd[pgd_index])){
                        printk("pgd_index: %d       PGD entry = 0x%16.16lx\n", pgd_index, (unsigned long
int)pgd[pgd_index]);
                }
        }
}

static void task_explore(void){
        struct task_struct * ts;
        struct mm_struct *mms;
        ts = get_current();
        printk("=========  PID del processo di contesto: %d \n ", ts->pid);
        mms = ts->mm;
        pgd = (unsigned long int *)mms->pgd;
        print_pgd(pgd);
}

static int __init
... (uguale ai moduli già visti)
```

Figura 6.1 (a)

```
[21174.980623] ================================== Inserito modulo Axo_Kpt
[21174.980634] =========  PID del processo di contesto: 9727
[21174.980634]  ======= PGD DIRECTORY (solo Entries con Present=1)
[21174.980640] pgd_index:   0     PGD entry = 0x0000000022143067
[21174.980649] pgd_index: 255     PGD entry = 0x000000005bfa0067

[21174.980652] pgd_index: 272     PGD entry = 0x0000000001fe3067
[21174.980655] pgd_index: 402     PGD entry = 0x000000005cc22067
[21174.980658] pgd_index: 468     PGD entry = 0x000000005f5e8067
[21174.980661] pgd_index: 511     PGD entry = 0x0000000001c10067
[21175.033362] ================================== Rimosso modulo Axo_Kpt
```

Figura 6.1 (b)

```
[21215.652485] ================================  Inserito modulo Axo_Kpt
[21215.652495] =========  PID del processo di contesto: 9937
[21215.652495]  ======= PGD DIRECTORY (solo Entries con Present=1)
[21215.652500] pgd_index:    0      PGD entry = 0x00000000403b0067
[21215.652504] pgd_index: 255      PGD entry = 0x0000000037736067

[21215.652553] pgd_index: 272      PGD entry = 0x0000000001fe3067
[21215.652557] pgd_index: 402      PGD entry = 0x000000005cc22067
[21215.652559] pgd_index: 468      PGD entry = 0x000000005f5e8067
[21215.652562] pgd_index: 511      PGD entry = 0x0000000001c10067
[21215.694885] ================================  Rimosso modulo Axo_Kpt
```

Figura 6.1 (c)

Eseguendo nuovamente questo modulo nel contesto di un diverso processo possiamo verificare quanto asserito sopra relativamente alla definizione della mappatura unica del sistema operativo nelle pagine dei diversi processi. Il risultato di una nuovo esecuzione è mostrato in Figura 6.1c. Confrontando con Figura 6.1b si vede che le PTE relative al processo hanno un valore diverso, perché puntano a PUD diversi, mentre quelle relative al sistema operativo hanno valori identici, quindi puntano agli stessi PUD. Questo significa che l'unica (piccolissima) ridondanza nelle Tabelle delle Pagine dovuta alla mappatura del Kernel in tutti i processi è costituita dalla replicazione delle righe da 256 a 511 dei PGD (che non causa alcuna ridondanza in termini di pagine), mentre tutti gli altri livelli sono condivisi tra le PT dei diversi processi.

7. Gestione della TP da Software – Decomposizione dell'indirizzo virtuale

In Figura 7.1 è riportato il codice della funzione decomponi(unsigned long indir) che, inserita opportunamente nel modulo già analizzato, decompone l'indirizzo virtuale indir nelle sue componenti relative alla Tabella delle Pagine. La funzione riempie i campi della variabile ind, costituita da una struct di 5 interi, e stampa tali componenti.

```
#define PTE_MASK      0x000000000000 01ff    //per selezionare gli ultimi 9 bit
#define OFFSET        0x000000000000 0fff    //per selezionare gli ultimi 12 bit

unsigned long int *pgd;
struct indirizzo{
        int pgd_indice;
        int pud_indice;
        int pmd_indice;
        int pt_indice;
        int offset;
} ind;

void decomponi(unsigned long indir){
        unsigned long temp;
        temp = indir;
        printk(KERN_INFO "\nDECOMPOSIZIONE DELL'INDIRIZZO\n");
        ind.offset = temp & OFFSET;
        printk(KERN_INFO "offset:       %d\n", ind.offset);
        temp = temp >> 12;
        ind.pt_indice = temp & PTE_MASK;
        printk(KERN_INFO "indice in pt:  %d\n", ind.pt_indice);
        temp = temp >> 9;
        ind.pmd_indice = temp & PTE_MASK;
        printk(KERN_INFO "indice in pmd: %d\n", ind.pmd_indice);
        temp = temp >> 9;
        ind.pud_indice = temp & PTE_MASK;
        printk(KERN_INFO "indice in pud: %d\n", ind.pud_indice);
        temp = temp >> 9;
        ind.pgd_indice = temp & PTE_MASK;
        printk(KERN_INFO "indice in pgd: %d\n\n", ind.pgd_indice);
}
```

Figura 7.1

Il funzionamento è semplice e autoesplicativo:

- la funzione isola la porzione offset dell'indirizzo, cioè gli ultimi 12 bit (ind.offset = temp & OFFSET;) e lo stampa
- poi esegue uno scorrimento di 12 bit a destra (temp = temp >> 12;), in modo che nei 9 bit meno significativi ci sia il valore dell'indice di PT,
- isola tale porzione (ind.pt_indice = temp & PTE_MASK;), la stampa, esegue uno scorrimento di 9 bit a destra, e ripete le stesse operazioni per i livelli superiori di directory

Il risultato dell'esecuzione di questo programma con un indirizzo virtuale 0x00007fffb0d42118 è mostrato in Figura 7.2.
Attenzione che i valori sono stampati in decimale, non in esadecimale.

Esempio/Esercizio 1: decomporre l'indirizzo 0x0000 7fff b0d42118 manualmente

- L'offset è 0x118, corrispondente in decimale a 8+16+256=280
- L'indice in PT è 0x142 costituito dall'ultimo bit della cifra d (perché è costituito da 9 bit) seguit da 42, quindi in decimale è 2+64+256=322
- L'indice in PMD è costituito dagli ultimi 2 bit della cifra b, seguita da 0, seguita dai primi 3 bit della cifra d, quindi in decimale è 2+4+128+256= 390
- procedere in maniera simile per i primi 2 livelli

Soluzione

```
[24609.857484] Indirizzo virtuale ricevuto = 0x00007fffb0d42118

[24609.857488] DECOMPOSIZIONE DELL'INDIRIZZO
[24609.857489] offset:          280
[24609.857490] indice in pt:    322
[24609.857491] indice in pmd:   390
[24609.857491] indice in pud:   510
[24609.857492] indice in pgd:   255
```

Figura 7.2

8. Tabella delle pagine e struttura virtuale dei processi

Gli indirizzi indicati nella struttura virtuale dei processi (vedi capitolo M2) sono in buona parte motivati dalla corrispondenza con posizioni significative nella struttura dei processi; in particolare si considerino le decomposizioni dei seguenti indirizzi

- indirizzo iniziale VMA C (..0000 0040 0000) → 0:0:2:0
- indirizzo iniziale VMA K (..0000 0060 0000) → 0:0:3:0

Dato che le pile crescono verso indirizzi bassi e le VMA crescono verso indirizzi alti, per quanto riguarda le VMA di pila è necessario fare attenzione alla terminologia: *la pagina iniziale (logica) di una pila è la pagina finale della sua VMA di pila e, viceversa, la pagina finale (logica) di una pila è la pagina iniziale della sua VMA di pila*.
Applicando alla pila globale nel modello di simulazione (pila di 3 pagine) otteniamo:

- pagina iniziale (logica) della pila: `7FFFFFFE`
- pagina iniziale della VMA di pila: `7FFFFFFC` (cioè `7FFFFFFE - 3`)

Applicando alla pila del primo thread nel modello di simulazione (pila di 2 pagine) otteniamo:

- pagina iniziale della pila del primo thread: `7FFFF77FF` → `255:511:443:511`
- pagina iniziale della corrispondente VMA: `7FFFF77FE` → `255:511:443:510`

La struttura della TP di un processo che ha creato dei thread è illustrata in figura 8.1. Si tenga presente che ***i processi leggeri che costituiscono dei thread condividono la stessa tabella delle pagine del processo padre*** (thread principale).

Le pile dei thread hanno dimensione (2048 + 1) pagina, quindi richiedono 4 PT + 1 pagina per ognuna; corrispondentemente le PTE del PMD tra 2 thread consecutivi sono distanziate di 4 posizioni e le PTE della PT "scorrono" di una pagina, ad esempio (vedi esercizio 2).

inizio pila T1: `255:511:439:509`
inizio pila T0: `255:511:443:510`

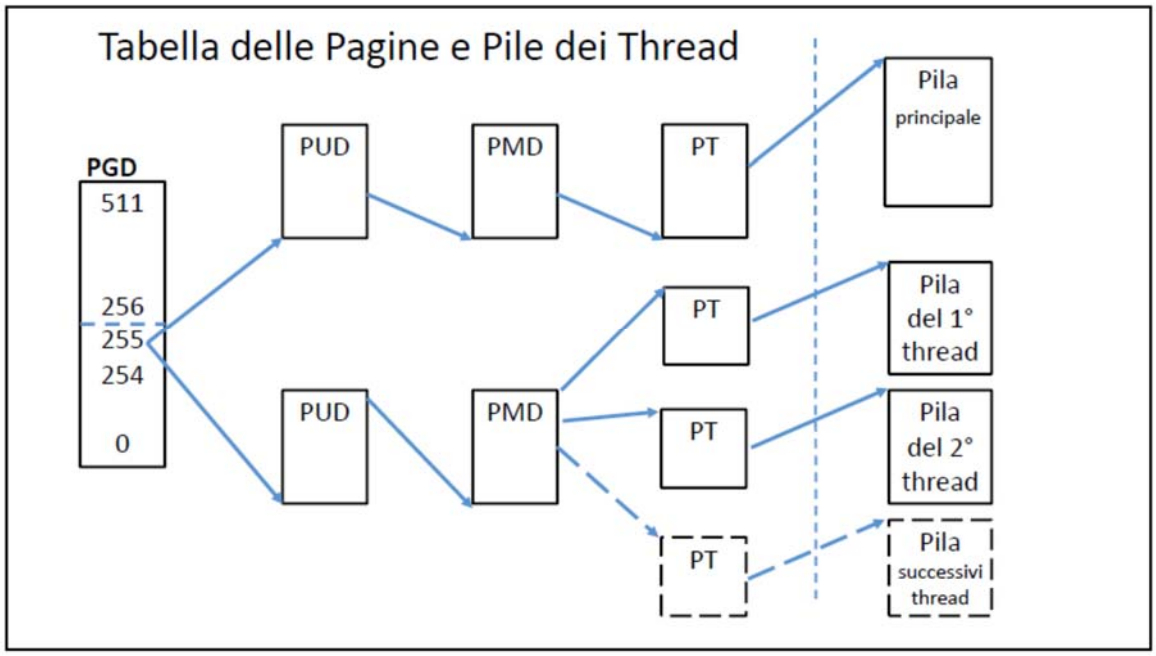

Figura 8.1

Esempio/Esercizio 2
Data la seguente struttura di VMA di un processo P che ha creato 4 aree virtuali di tipo M e 2 thread Q ed R mostrare la struttura della TP a livello delle singole directory (si ricorda che nel modello di simulazione vengono allocate inizialmente solo 2 PTE per ogni pila dei thread)

```
PROCESSO: P/Q/R  ******************************************************
    VMA : C  000000400,  3 , R  ,  P  ,  M  , <X,0>
          K  000000600,  1 , R  ,  P  ,  M  , <X,3>
          S  000000601,  4 , W  ,  P  ,  M  , <X,4>
          D  000000605,  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>
          T1 7FFFF77FB,  2 , W  ,  P  ,  A  , <-1,0>
          T0 7FFFF77FE,  2 , W  ,  P  ,  A  , <-1,0>
          P  7FFFFFFFC,  3 , W  ,  P  ,  A  , <-1,0>
```

Soluzione

```
Struttura della TP ****************
    VMA C   0:  0:  2:  0
            0:  0:  2:  1
            0:  0:  2:  2
    VMA K   0:  0:  3:  0
    VMA S   0:  0:  3:  1
            0:  0:  3:  2
            0:  0:  3:  3
            0:  0:  3:  4
    VMA D   0:  0:  3:  5
            0:  0:  3:  6
    VMA M   0:  0:128:  0
            0:  0:128:  1
    VMA M   0:  0:256:  0
    VMA M   0:  0:384:  0
    VMA M   0:  1:  0:  0
    VMA T 255:511:443:507
          255:511:443:508
    VMA T 255:511:443:510
          255:511:443:511
    VMA P 255:511:511:508
          255:511:511:509
          255:511:511:510
```

9. Gestione della TP da Software – Lettura dei directory

Come già visto il SO è in grado di utilizzare gli indirizzi fisici grazie alla mappatura virtuale/fisica basata sulla costante PAGE_OFFSET.

Per capire come questo avviene aggiungiamo al solito modulo la funzione get_PT_address(void) che, dato un indirizzo virtuale virtual_address già decomposto tramite la funzione decomponi(virtual_address) descritta in precedenza, esegue da Software il Page Walk, cioè attraversa l'intera Tabella delle Pagine e stampa gli indirizzi dei Directory ai diversi livelli e alla fine stampa il contenuto della cella di memoria di indirizzo virtuale virtual_address.

Come verifica di correttezza tale contenuto viene stampato inizialmente, prima di iniziare il Page Walk; i due statement che eseguono questa operazione sono:

```
printk("Indirizzo virtuale ricevuto = 0x%16.16lx\n", (long unsigned int)virtual_address);

printk("Parola letta all'indirizzo virtuale originale: %d\n", *((long int *) virtual_address));
```

Si noti che la stessa parola viene stampata nel modo usuale, cioè stampando il contenuto della cella puntata dalla variabile virtual_address (quindi la conversione virtuale/fisica tramite Page Walk è eseguita dall'Hardware).

Il codice della funzione mem_explore(unsigned long int virtual_address), che viene invocata all'installazione del modulo, è mostrato in Figura 9.1. La funzione svolge le seguenti operazioni:

- inizialmente le operazioni viste in un esempio precedente per accedere l'indirizzo del PGD del processo,
- poi esegue le 2 printk indicate sopra,
- poi invoca la funzione decomponi già vista per decomporre l'indirizzo virtuale inserendolo nella variabili ind
- infine invoca la funzione get_PT_address per eseguire il Page Walk

```
void mem_explore(unsigned long int virtual_address){
        struct task_struct *ts;
        struct mm_struct *mms;
        ts = get_current();
        mms = ts->mm;
        pgd = (unsigned long int *)mms->pgd;
        printk("Indirizzo virtuale ricevuto = 0x%16.16lx\n", (long unsigned int)virtual_address);
       printk("Parola letta all'indirizzo virtuale originale: %d\n", *((long int *) virtual_address));

        printk("indirizzo di PGD = 0x%16.16lx\n", (long unsigned int)pgd);
        decomponi(virtual_address);
        get_PT_address();
}
```

Figura 9.1

La funzione get_PT_(Figura 9.2) esegue (da software) le stesse operazioni che l'Hardware x64 esegue quando accede alla TP, cioè naviga lungo l'albero interpretando le diverse componenti di un indirizzo virtuale. La funzione ripete 3 volte fondamentalmente le stesse operazioni, per attraversare rispettivamente il PUD, il PMD e la PT.

Nel caso del PUD le istruzioni svolgono le seguenti operazioni, semplificate omettendo i recasting, sono:

- pud_phys = pgd[ind.pgd_indice]; accede il puntatore (fisico) al pud all'interno del PGD utilizzando come offset il campo pgd_indice della variabile ind
- pud_phys = pud_phys & NO_OFFSET; azzera nel puntatore gli ultimi 12 bit
- pud = pud_phys + PAGE_OFFSET/8; determina il corrispondente indirizzo virtuale, sommando la costante PAGE_OFFSET divisa per 8 in quanto l'indirizzo virtuale è interpretato in parole da 8 byte
- printk("indirizzo di PUD = 0x%16.16lx\n", pud); stampa l'indirizzo virtuale
- procede al livello inferiore (PMD) utilizzando il puntatore pud appena determinato

```
void get_PT_address( ){
        unsigned long int *pud_phys, *pmd_phys, *pte_phys, *NPF, *pmd, *pte, *pud, *NPV ;
        unsigned long int word_addr;
        int word;
        printk("INIZIO PAGE WALK");
        //accedo pud
        pud_phys = (unsigned long int *)pgd[ind.pgd_indice];
        pud_phys = (unsigned long int)pud_phys & NO_OFFSET;
        pud = pud_phys + PAGE_OFFSET/8;
        printk("indirizzo di PUD = 0x%16.16lx\n", pud);
        //accedo pmd
        pmd_phys = (unsigned long int *)pud[ind.pud_indice];
        pmd_phys = (unsigned long int)pmd_phys & NO_OFFSET;
        pmd = pmd_phys + PAGE_OFFSET/8;
        printk("indirizzo di PMD = 0x%16.16lx\n", pmd);
        //accedo pt
        pte_phys = (unsigned long int *)pmd[ind.pmd_indice];
        pte_phys = (unsigned long int)pte_phys & NO_OFFSET;
        pte = pte_phys + PAGE_OFFSET/8;
        printk("indirizzo di  PT = 0x%16.16lx\n", pte);

        //accedo NPF
        NPF = (unsigned long int *)pte[ind.pt_indice];
        NPF = (unsigned long int)NPF & NO_OFFSET;
        printk("NPF (con NX flag non azzerato) = 0x%16.16lx\n", (long unsigned int)NPF);
        NPF = (unsigned long int)NPF & NO_NX_FLAG;
        printk("NPF (con NX flag azzerato) = 0x%16.16lx\n", (long unsigned int)NPF);

        //determina l'indirizzo completo della parola
        NPV = NPF + PAGE_OFFSET/8;
        printk("NPV = 0x%16.16lx\n", (long unsigned int)NPV);
        word_addr = (unsigned long int)NPV + ind.offset;
        printk("Indirizzo virtuale completo = 0x%16.16lx\n", word_addr);
        word = *((int *)word_addr);
        printk("Parola letta all'indirizzo virtuale derivato da fisico = %d\n", word);
}
```

Figura 9.2

Le costanti NO_OFFSET e NO_NX_FLAG, usate nella funzione, sono definite nel modo seguente

```
#define NO_OFFSET     0xfffffffffffff000
#define NO_NX_FLAG   0x7fffffffffffffff
```

Una volta trovato l'indirizzo della PT la funzione esegue le seguenti operazioni:

- legge il NPF (NPF = pte[ind.pt_indice])
- lo ripulisce dai flag presenti negli ultimi 12 bit (NPF = NPF & NO_OFFSET) e lo stampa
- lo ripulisce anche dal flag in posizione 63 (NPF = NPF & NO_NX_FLAG) e lo stampa
- determina l'indirizzo completo della parola:
    - calcola NPV (NPV = NPF + PAGE_OFFSET/8) corrispondente a NPF – attenzione: questo NPV è l'indirizzo virtuale del SO che si mappa sull'indirizzo fisico NPF
    - gli somma l'offset (word_addr = (unsigned long int)NPV + ind.offset)
- utilizza l'indirizzo word_addr per leggere il contenuto della parola in memoria

Il risultato dell'esecuzione di questo modulo con lo stesso indirizzo virtuale visto nel caso della funzione decomponi è presentato in Figura 9.3. In rosso sono indicate le 2 diverse stampe della parola di indirizzo 0x00007fffb0d42118. La parola è la stessa, ma letta con due modalità diverse.

```
 [24609.851402] START MODULE Axo Page Walk
[24609.857484] Indirizzo virtuale ricevuto = 0x00007fffb0d42118
[24609.857486] Parola letta all'indirizzo virtuale originale: 12345
[24609.857487] indirizzo di PGD = 0xffff88002f6fb000
[24609.857488]
[24609.857488] DECOMPOSIZIONE DELL'INDIRIZZO
[24609.857489] offset:       280
[24609.857490] indice in pt:  322
[24609.857491] indice in pmd: 390
[24609.857491] indice in pud: 510
[24609.857492] indice in pgd: 255
[24609.857492]
[24609.857493] INIZIO PAGE WALK
[24609.857494] indirizzo di PUD = 0xffff88002f694000
[24609.857495] indirizzo di PMD = 0xffff88002f5a3000
[24609.857495] indirizzo di  PT = 0xffff88002f6d7000
[24609.857496] NPF (con NX flag non azzerato) = 0x8000000023af7000
[24609.857497] NPF (con NX flag azzerato) = 0x0000000023af7000
[24609.857498] NPV = 0xffff880023af7000
[24609.857500] Indirizzo virtuale completo = 0xffff880023af7118
[24609.857501] Parola letta all'indirizzo virtuale derivato da fisico = 12345
[24609.867360] EXIT MODULE Axo Page Walk
```

Figura 9.3

La prima modalità di accesso alla parola di indirizzo virtual_address è rappresentata in Figura 9.4. Si tratta della modalità normale: virtual_address è inviato alla MMU, che accede al PGD, poi agli altri directory e infine legge la parola cercata: 12345.
Nella figura gli indirizzi contenuti nella TP sono indicati da frecce rosse tratteggiate, mentre le frecce continue rappresentano un flusso di informazione.

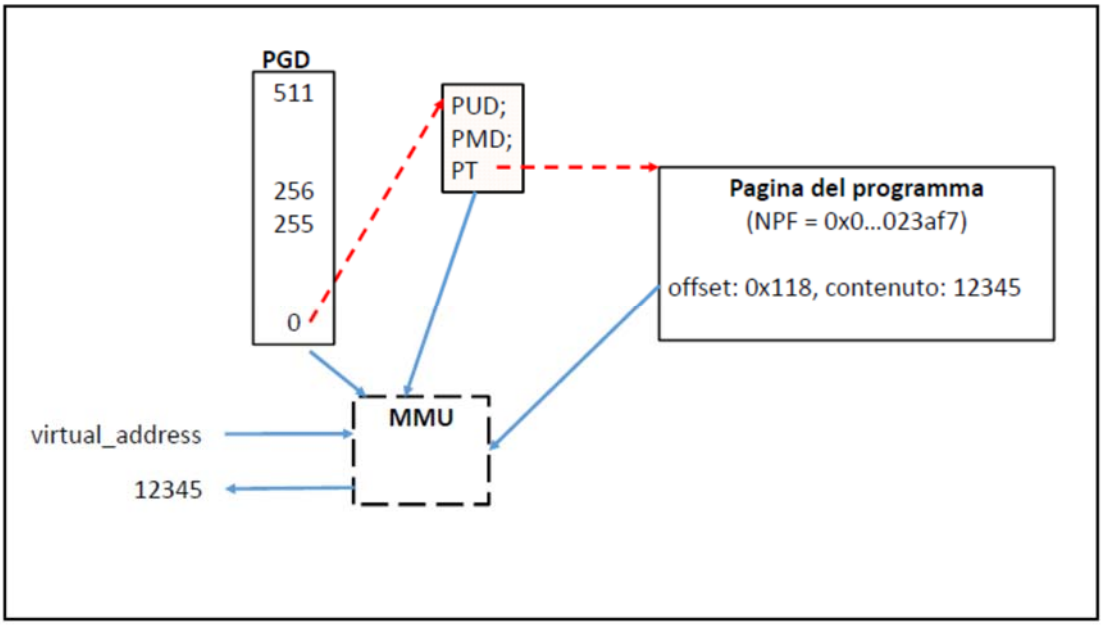

Figura 9.4

Consideriamo ora cosa accade quando viene eseguito il modulo con la funzione get_PT_address. La situazione è rappresentata in Figura 9.5, con le stesse convenzioni utilizzate per Figura 9.4. In Figura 9.5 sono però presenti due gerarchie di Directory sotto lo stesso PGD:

- la gerarchia identica a quella precedente, indicata in rosso

- la gerarchia relativa alla mappatura fisica della memoria, che è indirizzata dalla porzione S del PGD – questa gerarchia e i relativi puntatori sono mostrati i verde

In Figura 9.5 il virtual_address viene passato al modulo get_PD_address, il quale accede la memoria tramite la seconda gerarchia di directory; il modulo accede fisicamente l'altra gerarchia, leggendo i diversi livelli di directory e infine legge la pagina del programma, sempre attraverso la mappatura fisica dell'indirizzo costruito navigando la prima gerarchia. Per interpretare la Figura 9.5 è utile rivedere la figura 6.1: le pagine di PUD, PMD e PT di tale figura sono quelle indicate nel rettangolo orizzontale di figura 9.5.

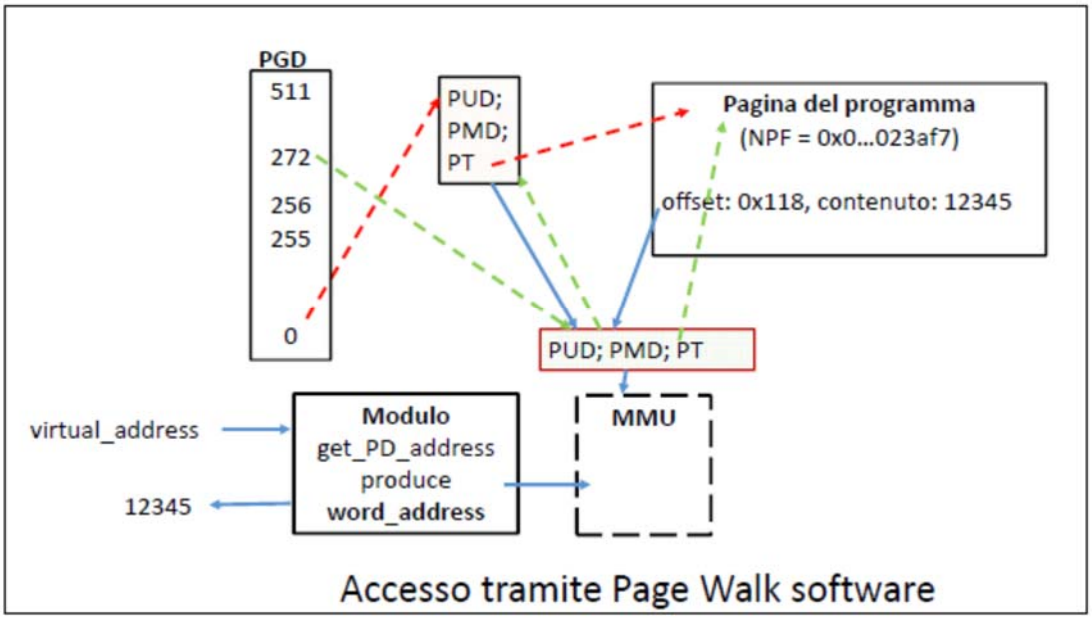

Figura 9.5

10. Gestione della TP da Software – Creazione ed eliminazione di PTE

In maniera simile a quanto visto per la lettura, da parte del SO è possibile la scrittura della TP.

Le principali funzioni del SO per modificare la TP sono:

- funzione per creare una PTE: `mk_pte`; questa macro converte in una PTE in formato corretto un numero di pagina e una struttura di bit di protezione
- funzioni per allocare e inizializzare intere pagine della TP: `pgd_alloc, pud_alloc, pmd_alloc, pte_alloc`;
- funzioni per liberare la memoria occupata dalla TP: `pgd_free, pud_free, pmd_free, pte_free`
- funzioni per assegnare un valore a una PTE: `set_pgd, set_pud, set_pmd, set_pte`

Quando viene richiesto l'accesso a una pagina NPV non ancora mappata dalla TP il SO deve:

- creare e inserire nella PT una nuova PTE, se la PT esiste già
- in caso contrario:
  - deve allocare una nuova PT (`pte_alloc`)
  - creare e inserire nel PMD una nuova PTE che punta alla PT appena creata, se il PMD esiste già
  - in caso contrario, ripetere l'operazione per il livello superiore

11. Approfondimenti

Gestione del TLB
Nel x64 la gestione del TLB è svolta quasi completamente dall'Hardware. L'operazione di TLB flush è molto onerosa ed è eseguita anch'essa dall'Hardware ogni volta che viene modificato il valore di CR3. Linux marca come pagine globali le pagine di SO che sono utilizzate da molti processi, in modo che non vengano invalidate e svolge alcune altre ottimizzazioni fortemente collegate con il funzionamento dell'HW.

Linux e le memoria cache
La gestione delle memoria cache è trasparente al SO e totalmente in carico all'Hardware. Tuttavia, Linux cerca di usare la memoria in modo da ottimizzare l'uso delle cache. Infatti, indipendentemente dall'architettura e dallo schema di indirizzamento specifico, tutte le cache condividono il fatto seguente: *indirizzi vicini tra loro e allineati alla dimensione della cache tendono ad usare diverse righe di cache*. Linux ad esempio definisce i campi più utilizzati delle strutture (struct) all'inizio, per aumentare la probabilità che una sola riga di cache li possa contenere. Altri accorgimenti sono usati per evitare che gli stessi campi finiscano nelle cache di diverse CPU.

Esecuzione del Context Switch relativamente alla memoria
La funzione `schedule( )`, definita nel file `kernel/sched/core.c`, prima di invocare la macro assembler `switch_to(prev, next)`
che esegue la commutazione delle sPile dei 2 processi, invoca la seguente funzione:

```
switch_mm(oldmm, mm, next);
```

Il punto centrale di questa funzione è costituito dall'invocazione della macro assembler **`load_cr3`** che assegna al registro CR3 il valore della variabile `pgd` presa dal descrittore del nuovo (next) task da eseguire.

```
static inline void switch_mm(struct mm_struct *prev,
                        struct mm_struct *next, struct task_struct *tsk)
        {
                ...
                /* Re-load page tables */
               load_cr3(next->pgd);  //assegna nuovo valore a CR3
                ...
        }
```

La macro, riportata sotto nella sua forma originale, è di difficile lettura perchè utilizza l'inline assembler gnu, ma nella sostanza conduce all'esecuzione della seguente istruzione assembler:

```
movq R, cr3
```

che copia il contenuto del registro R nel registro CR3, dove R è un registro scelto dal compilatore nel quale viene caricato il valore di `next->pgd` convertitto in indirizzo fisico dalla funzione `__phys_addr`

La funzione `__phys_addr(x)` esegue una serie di controlli e se tutto va bene restituisce x – PAGE_OFFSET.

```
#define load_cr3(pgdir)                                                 \
         asm volatile("movl %0,%%cr3": :"r" (__pa(pgdir)));

#define __pa(x)         __phys_addr((unsigned long)(x))
```