

# L'indirizzamento della memoria

Studio dettagliato del meccanismo di traduzione degli indirizzi virtuali  
in fisici in Linux

Gabriele Vanoni

Politecnico di Milano, Corso di Laurea in Ingegneria Informatica, Architettura dei calcolatori e  
dei sistemi operativi

Anno Accademico 2014/2015

# Indice

## 1 Il modello

## 2 La memoria

- La memoria fisica
- La memoria virtuale
  - Introduzione
  - La gestione hardware
  - La gestione software
- L'organizzazione della memoria

## 3 La traduzione degli indirizzi

- La tabella delle pagine
- Il processo di traduzione HW
- Il processo di traduzione SW
- Le prestazioni

# Il modello

It has been said on the Linux Memory Management mailing list that:

*the VM is poorly documented and difficult to pick up as  
“the implementation is a nightmare to follow”*

- Kernel Linux versione 3.09
- Architettura del processore x86-64 (o Intel 64 o AMD64)
- I puntatori o *unsigned long int* occupano 8 byte (64 bit)

# La memoria fisica

## Fact

- *Il processore legge dati e istruzioni dalla memoria fisica.*
- *Il quantitativo di memoria fisica è dato dalla quantità effettiva di memoria (DRAM) installata.*
- *L'architettura x86-64 può gestire al massimo indirizzi fisici da 52 bit, ovvero uno spazio di indirizzamento fisico al massimo da 4096TiB.*

# La memoria virtuale - Introduzione

## Fact

*È impossibile per il programmatore sapere in quale punto della memoria fisica verrà caricato il programma, dipenderà infatti sia dal quantitativo di RAM installata, sia dagli altri processi in esecuzione.*

## Corollary

*Occorre dunque creare un'astrazione che permetta di scrivere un programma come se fosse l'unico presente in memoria e di conoscerne gli indirizzi indipendentemente dalla macchina su cui viene eseguito.*

## Definition

Diciamo allora che ogni processo avrà un suo spazio virtuale all'interno del quale verranno caricati codice, pila, etc.

# La memoria virtuale - La gestione hardware

## Fact

- *L'architettura x86-64 può gestire al massimo indirizzi fisici da 48 bit, ovvero uno spazio di indirizzamento virtuale al massimo da 256TiB.*
- *Il processore lavora sempre con indirizzi virtuali, che dovrà quindi tradurre in indirizzi fisici per operazioni di lettura/scrittura dalla memoria centrale.*
- *La traduzione degli indirizzi è prerogativa dell'hardware, ma può avvenire solo se il software ha già predisposto le opportune strutture dati.*

# La memoria virtuale - La gestione software

## Corollary

*Il software deve quindi occuparsi di creare le condizioni per cui un indirizzo virtuale di un processo possa essere tradotto dall'hardware in un indirizzo fisico.*

# L'organizzazione della memoria

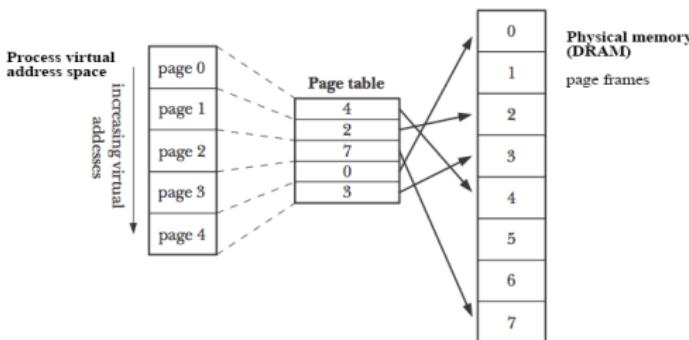
- Una serie di fattori, in particolare la frammentazione (interna ed esterna), hanno fatto sì che i progettisti di hardware e software abbiano optato per la divisione dello spazio di indirizzamento fisico e virtuale in piccole porzioni "atomiche", chiamate rispettivamente *page frames* e *pages*.
- In questo modo ogni pagina virtuale può essere allocata ovunque nella memoria fisica, le varie parti del programma potranno risiedere in porzioni diverse della RAM non fissate a priori ma a seconda del contesto attuale. Si evita così che ci siano "buchi" nella memoria centrale (frammentazione esterna).
- L'architettura x86-64 prevede pagine di 4KiB, 2MiB o in alcuni casi 1GiB. Nei computer domestici vengono normalmente utilizzate le pagine da 4K (verrà dunque seguito questo modello).
- La ridotta dimensione delle pagine fa sì che non si sprechi quasi spazio non riempiendo completamente una pagina (frammentazione interna).



# I problemi della paginazione lineare

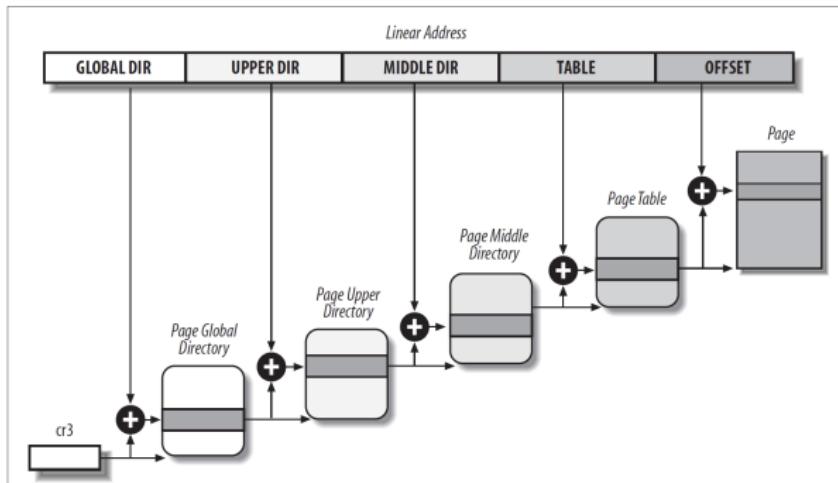
## Fact

- Se si utilizzasse una tabella lineare per la traduzione degli indirizzi si avrebbe una struttura da 512 GiB per ogni processo! (Infatti 48bit di indirizzo virtuale meno i 12 per lo spiazzamento all'interno della pagina = 36. Quindi si hanno  $2^{36}$  elementi della tabella, che occupano 8 byte ciascuno)
- È tuttavia inutile mappare l'intero spazio di ogni processo (256TiB), quando questo rimarrà per la maggior parte inutilizzato.



# La paginazione multi-livello

- Ecco allora che sorge spontanea l'idea di creare tabelle di tabelle di pagine.
- Nell'architettura x86-64 vengono gestiti 4 livelli di pagine. L'indirizzo virtuale viene dunque spezzato in modo tale da poter accedere alla gerarchia delle tabelle.



# Il processo di traduzione HW - Le ipotesi

- Al processore vengono inviati sempre indirizzi virtuali. Perciò è presente un componente, l'MMU (Memory management unit) che si occupa della traduzione.
- Nell'architettura x86-64 è presente un registro, chiamato CR3 che contiene l'indirizzo fisico del primo elemento del primo livello della tabella delle pagine del processo in esecuzione.
- Tutti gli indirizzi di tutti i livelli della tabella delle pagine sono fisici.
- La tabella delle pagine del processo in esecuzione è sempre residente.
- Ogni sottotabella occupa esattamente una pagina, contiene dunque 512 elementi ( $512 \times 8B = 4KiB$ ), servono dunque 9 bit per indirizzarla.

# Il processo di traduzione HW - Il page-walk

- ➊ Il processore riceve l'indirizzo virtuale e lo invia all'MMU
- ➋ L'MMU, trattando il CR3 come la testa di un vettore trova in memoria fisica l'indirizzo fisico del PUD ( $PUD = CR3[index]$   
*//index sono i primi 9 bit dell'indirizzo virtuale*)
- ➌ L'MMU, trattando il PUD come la testa di un vettore trova in memoria fisica l'indirizzo fisico del PMD ( $PMD = PUD[index]$   
*//index sono i secondi 9 bit dell'indirizzo virtuale*)
- ➍ L'MMU, trattando il PMD come la testa di un vettore trova in memoria fisica l'indirizzo fisico del PTE ( $PTE = PMD[index]$   
*//index sono i terzi 9 bit dell'indirizzo virtuale*)
- ➎ L'MMU, trattando il PTE come la testa di un vettore trova in memoria fisica l'indirizzo fisico della pagina ( $PAGE = PTE[index]$   
*//index sono i quarti 9 bit dell'indirizzo virtuale*)

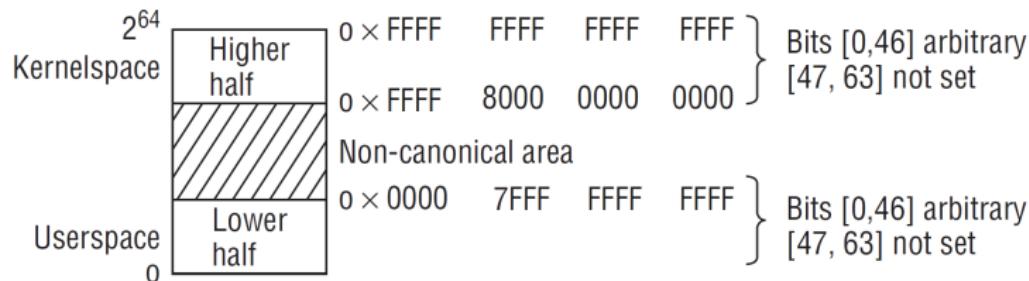
# Che cosa c'entra il software?

- Il software ha la necessità di accedere, leggere e modificare le tabelle delle pagine dei diversi processi. Un processo potrebbe ad esempio necessitare l'allocazione di una nuova pagina fisica di pila o di heap. Il SO dovrà allora integrare la tabella delle pagine, nei diversi livelli.
- Tutti gli indirizzi contenuti nella tabella delle pagine però sono fisici, e il processore accetta solo indirizzi virtuali.
- Qual è l'indirizzo virtuale della tabella delle pagine?
- Dove sono collocate? In quale spazio di indirizzamento?

# Le ipotesi - Lo spazio dei processi

## Fact

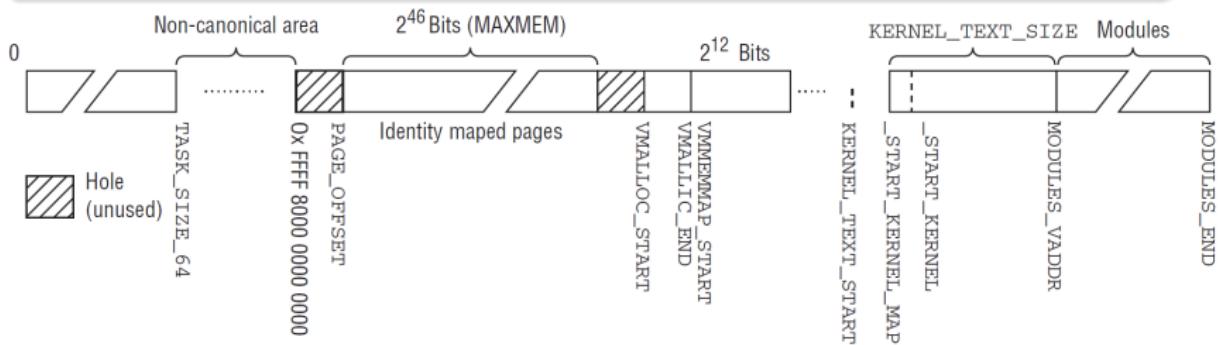
- Linux divide in due parti lo spazio di indirizzamento di ogni processo, la prima metà (bit più significativo 0) è dedicata al processo, mentre la seconda metà (bit più significativo 1) è accessibile solo al kernel e rimane invariata quando avviene una commutazione.
- Dato che i puntatori sono da 64 bit e non da 48 (dimensione massima dell'indirizzo virtuale per x86-64) viene utilizzato il meccanismo dell'estensione del segno, i 17 bit più significativi saranno tutti 0 per gli indirizzi user e tutti 1 per gli indirizzi kernel.



# Le ipotesi - Lo spazio del kernel

## Fact

*Lo spazio del kernel a sua volta è diviso in diverse zone. In particolare uno spazio da 46 bit (64TiB) che parte all'indirizzo della macro PAGE\_OFFSET è dedicato alla mappatura 1:1 della memoria fisica. Vediamo allora che la memoria fisica massima gestibile dal nostro modello HW-SW è 64 TiB. Non ci occupiamo delle altre zone che contengono ad esempio l'immagine del kernel.*



# Le ipotesi - Le strutture del kernel

## Fact

- *La tabella delle pagine è sempre residente in memoria fisica, mappa tutto lo spazio di indirizzamento del processo, user e kernel, e non fa parte della parte user. È quindi mappata nella parte kernel al proprio indirizzo fisico + PAGE\_OFFSET.*
- *Il kernel per ogni processo in esecuzione ha in memoria un'istanza di task\_struct, una struttura che contiene tutte le informazioni necessarie sul processo in esecuzione (nome, pid, file aperti, etc.). Contiene anche il puntatore a un'istanza di mm\_struct, la struttura contenente tutte le informazioni sulla memoria. Tra queste c'è un campo chiamato pgd, contenente l'indirizzo virtuale del primo elemento del primo livello della tabella delle pagine. Questo sarà uguale, per quanto abbiamo detto prima al suo indirizzo fisico + PAGE\_OFFSET. Capiamo allora da dove viene preso il valore da caricare nel registro CR3 quando avviene una commutazione di contesto.*



## Il processo di traduzione SW - Il page-walk

- L'algoritmo del page-walk software è lo stesso del page-walk hardware, semplicemente ogni volta che si ha un indirizzo fisico gli si aggiunge PAGE\_OFFSET per trasformarlo in indirizzo virtuale.
- Attenzione però perché l'algoritmo non è completamente gestito dal software, infatti gli indirizzi virtuali dei vari PGD, PUD, PMD, PTE vengono tradotti in indirizzi fisici tramite il meccanismo hardware, le tabelle delle pagine infatti mappano anche loro stesse.

# Il problema delle prestazioni

## Fact

- *Il meccanismo di traduzione hardware è lento, dovendo fare svariati accessi a memoria per ogni parola (almeno 5)*
- *Il meccanismo di traduzione software è ancora più lento basandosi su quello hardware, in particolare per ogni passaggio SW viene utilizzato l'intero meccanismo HW, quindi per ogni parola servono almeno 25 accessi a memoria (tuttavia questo sistema è utilizzato solo per aggiornare le tabelle delle pagine quindi molto raramente).*

## Corollary

*Quindi in tutte le macchine moderne viene implementata una cache chiamata Translation lookaside buffer (TLB) che permette una significativa accelerazione del processo di traduzione. Il TLB infatti contiene un certo numero di traduzioni numero di pagina virtuale - numero di pagina fisico.*

# Il guadagno in prestazioni

## Example

Se consideriamo delle ipotesi ragionevoli come:

- dimensione del TLB: 1024 voci
- hit time: 1 ciclo di clock
- miss penalty: 30 cicli di clock
- miss rate: 1%
- la coppia NPV-NPF viene sempre letta dal TLB,

la media del tempo di accesso a memoria è  
 $di: 1 \times 0.99 + (1 + 30) \times 0.01 = 1.3$  cicli di clock

# Il TLB nell'architettura x86-64

## Fact

I moderni PC Intel (Core 4<sup>a</sup> generazione) integrano un TLB con queste caratteristiche, in particolare le due cache di primo livello sono set-associative a 4 vie, mentre quella di secondo livello è a 8 vie.

Cache		Dimensione pagina		
Nome	Livello	4KiB	2MiB	1GiB
DTLB	1°	64	32	4
ITLB	1°	128	8/core logico	0
STLB	2°		1024	0

## Fact

Quando viene caricato un valore nel registro CR3, il contenuto del TLB viene cancellato. Ci sono in realtà delle ottimizzazioni per cui si rimanda al manuale dell'architettura.



# Appendice A - I bit di stato

## Fact

*Le pagine sono sempre allineate, ovvero il loro primo elemento è sempre ad un indirizzo con i 12 bit meno significativi a 0.*

## Corollary

*Questo significa che la traduzione è sempre e solo dei 36 bit più significativi, essendo gli ultimi 12 uguali tra indirizzo fisico e virtuale. Questi 12 bit (in realtà 8) vengono allora utilizzati come bit di stato. Alcuni flag sono user/supervisor, RO/RW, present/swapped. Per una trattazione completa si rimanda al manuale dell'architettura.*

# Lo sanno proprio tutti

(Tratto da “The Social Network”)

Play

# L'algoritmo: page\_walk.h

```
unsigned long shift_dx (int destra, int sinistra, unsigned long addr)
{
    addr = addr>>destra;
    addr = addr<<sinistra;
    return addr;
}

unsigned long shift_sx (int sinistra, int destra, unsigned long addr)
{
    addr = addr<<sinistra;
    addr = addr>>destra;
    return addr;
}
```

# L'algoritmo: page\_walk.h

```
unsigned long remove_flags (unsigned long addr)
{
    addr = shift_dx(12, 12, addr);
    addr = shift_sx(16, 16, addr);
    return addr;
}

unsigned long passo (unsigned long sup, unsigned long index)
{
    unsigned long inf;
    inf = sup[index];
    inf = remove_flags(inf);
    return PAGE_OFFSET + inf;
}
```

# L'algoritmo: page\_walk.c

```
unsigned long page_walk (unsigned long addr, unsigned long pgd)
{
    #include "page_walk.h"                      //e dichiarazione variabili

    pgd_index = shift_sx(16, 55, addr); //calcolo degli indici
    pud_index = shift_sx(25, 55, addr);
    pmd_index = shift_sx(34, 55, addr);
    pte_index = shift_sx(43, 55, addr);
    offset = shift_sx(52, 52, addr);

    pud = passo (pgd, pgd_index);           //page_walk
    pmd = passo (pud, pud_index);
    pte = passo (pmd, pmd_index);
    page = passo (pmd, pmd_index);

    word = page + offset;                  //aggiungo l'offset
    return word;
}
```

## Appendice C - Argomenti non trattati

La trattazione qui presentata copre solo un aspetto limitato e certamente non esaurisce i diversi argomenti relativi alla gestione della memoria in Linux. Per esempio non sono stati trattati i seguenti aspetti:

- L'inizializzazione del sistema di paginazione
- La gestione della memoria fisica (algoritmi e strutture dati)
- La gestione dello spazio di indirizzamento dei processi, in particolare la segmentazione, che nell'architettura x86-64 è solo software e quindi interamente gestita dal sistema operativo
- Il processo di *swapping* e la gestione dei *page fault*
- La mappatura dei file in memoria

# Bibliografia

-  [Linux Kernel code \(with Documentation\)](#)
-  [Mel Gorman, Understanding The Linux Virtual Memory Manager \(Chapter 4\)](#)
-  [Remzi H. e Andrea C. Arpacı-Dusseau, Operating Systems: Three Easy Pieces \(Address Spaces, Address translation, Introduction to Paging, Translation Lookaside Buffer, Advanced Page Tables\)](#)
-  [Ariane Keller, Kernel Space - User Space Interfaces \(Netlink socket\)](#)
-  [Daniel Bovet, Marco Cesati, Understanding the Linux Kernel 3rd Edition \(Chapter 2, Paging in Hardware, Paging in Linux\)](#)
-  [Wolfgang Mauerer, Professional Linux Kernel Architecture \(Sections 3.3, 3.4\)](#)
-  [Intel 64 and IA-32 Architectures Software Developer's Manual \(Volume 3, Chapter 4\)](#)