Memoria

Gabr1313

December 22, 2023

Contents

Chapter 1

Organizzazione dello spazio virtuale dei processi

1.1 VMA

La memoria virtuale di un processo LINUX è suddivisa in un certo numero di **aree di memoria virtuale** (**VMA**). Ogni VMA è costituita da un numero intero di pagine virtuali consecutive.
Tipologie di VMA:

- **Codice** (**C**): eseguibile e costanti.
- **Costanti per rilocazione dinamica** (**K**): parametri per il collegamento con librerie dinamiche.
- **Dati statici** (**S**): dati inizializzati allocati per tutta la durata del programma.
- **Dati dinamici** (**D**): heap (contiene anche i BSS). Il limite corrente è contenuto nella variabile del descrittore del processo `brk` (Program break).
- **Pile dei Thread** (**T**): contiene gli stack dei threads
- **Memory-Mapped files** (**M**): mappa di un file in aree di memoria virtuale. Ad esempio le **Librerie dinamiche** e la **Memoria condivisa**.
- **Memoria condivisa** (**P**): Upila che contiene le varibaili locali e i parametri delle funzioni.

Per C, K, S, D e P esiste un'area virtuale, mentre per M e per T possono esistere zero o più aree virtuali. Le VMA C, K e S sono l'immagine dei corrispondenti segmenti del file eseguibile (ELF). I **Block Started by Symbol** (**BSS**) sono dati non inizializzati, che nell'eseguibile sono rappresentati sinteticamente da un nome simbolico e lo spazio di memoria occupato. Avendo un comportamento simile a quello di memoria allocata dinamicamente, durante l'esecuzione sono allocati nello heap.

1.1.1 Struttura dati per la VMA

Ogni VMA è definita in una variabile di tipo `vm_area_struct`.

```
struct vm_area_struct {
    struct mm_struct *vm_mm; // puntatore al processo
    unsigned long vm_start;
    unsigned long vm_end;
```

```
    struct vm_area_struct *vm_prev; // linked list
    struct vm_area_struct *vm_next;
    unsigned long vm_flags;
    ...
    unsigned long vm_pgoff;
    struct file *vm_file;
    ...
    struct file * vm_file; // possibile puntatore al file
    unsigned long vm_pgoff; // possibile offset nel file
};
```

Ogni bit di `vm_flags` indica una caratteristica della VMA: `VM_READ`, `VM_WRITE`, `VM_EXEC`, `VM_SHARED`, `VM_GROWSDOWN`, `VM_DENYWRITE`...

Una VMA può essere **anonima** o **mappata su un file** detto **Backing Store**.

Il comando `cat /proc/<pid>/maps` mostra le VMA di un processo. I permessi sono `r` (`VM_READ`), `w` (`VM_WRITE`), `x` (`VM_EXEC`), `p` (private: `VM_SHARED`).

1.1.2 Page fault

Ogni volta che la MMU genera un interrupt di Page Fault su un inidirizzo virtuale NPV, il kernel attiva la routine **Page Fault Handler** (**PFH**).

```
if (NPV notin VMA || NPV hasn't permission) Segmentation Fault;
else if (NPV notin mem) load NPV in mem;
```

1.1.3 Regole generali e implementazione specifica

- **Regole generali**: garantite ufficialmente
- **Implementazione specifica**: comportamento non specificato

1.1.4 Address Space Layout Randomization

La **ASLR** è una tecnica di sicurezza che assegna casualmente l'indirizzo delle funzioni di libreria e delle più importanti aree di memoria. Per semplicità questa non viene considerata.

1.2 Tabella delle Pagine e Aree Virtuali

Il sistema operativo garantisce le seguenti regole:

- In ogni istante esistono esclusivamente le pagine virtuali NPV appartenenti alle VMA del processo.
- La TP è sufficientemente grande
- NPV può essere mappata su una NPF (indicato da una flag sulla TP)

La tecnica **demand paging** consiste nel caricare in memoria le pagine in base ai Page Fault generati.

1.2.1 Riassunto

- viene creata la VMA di pila con n NPV, di cui ne vengono fisicamente allocate 1 o 2.
- l'accesso a NPV non mappate ne cause l'allocazione (Page Fault)
- tentativo di accesso a pagine subito sotto l'area esistente (area di growsdown) produce un tentativo di crescita della VMA
- tentativo di accesso a pagine non esistenti o diverse dall'area di growsdown produce un Segmentation Fault

```
if (NPV notin VMA || NPV hasn't permission) Segmentation Fault;
else if (NPV notin mem) {
    if (NPV is start_address of VMA with Growsdown == 1) add page NPV-1;
    load NPV in mem;
}
```

1.2.2 Servizio brk

Con **brk** si intende **break**, ovvero il limite superiore di una VMA.
La `malloc()` svolge 2 operazioni distinte:

- crescita della VMA D (`brk()` o `sbrk()`)
- allocazione fisica delle pagine (se necessario riempirle coi dati)

`int brk(void * end_data_segment)` assegna il valore `end_data_segment` alla VMA D. Restituisce 0 in caso di successo, -1 in caso di errore. `void * sbrk(intptr_t increment)` incrementa la VMA D di `increment` e restituisce un puntatore alla posizione iniziale della nuova area (program break position).

1.3 Gestione della memoria virtuale nella creazione di processi

1.3.1 Fork

Dato che il processo è una copia del padre, non viene allocata nuova memoria, ma vengono solo aggiornate le informazione della tabella dei processi. Una pagina viene duplicata solo nel momento di scrittura sulla pagina stessa: **Copy on Write** (**COW**).

- durante la `fork()`, i permessi di tutte le pagine vengono cambiati in sola lettura (R)
- quando si verifica un page fault, il PFH verifica se la pagina protetta R appartenga a una VMA scrivibile. In tal caso, se `ref_count() > 1`, duplica le pagine. Successivamente cambia la protezione in scrittura (W).

La variabile `ref_count` nel `page_descriptor` indica il numero di utilizzatori della pagina fisica.

```
if (NPV notin VMA) Segmentation Fault;
else if (NPV hasn't permission) {
    if (NPV is in VMA with COW) {
        if (ref_count() > 1) duplicate page;
        else change protection in W;
```

```
    }
    else Segmentation Fault;
}
else if (NPV notin mem) {
    if (NPV is start_address of VMA with Growsdown == 1) add page NPV-1;
    load NPV in mem;
}
```

1.3.2 Context Switch

Un context switch causa un svuotamento (**flush**) della TLB (Table Lookaside Buffer). Nel caso il descrittore della pagine fisica abbia `dirty_bit = 1` le pagine vengono copiate su disco.

1.3.3 Exit

- eliminazione della struct della VMA
- eliminazione della TP del processo
- deallocazione delle NPV o decremento del `ref_count`
- context switch (con flush del TLB)

1.4 Creazione dei Thread e aree di tipo T

I thread condividono la stessa struttura di aree virtuali e TP del processo padre. L'unica area di memoria non condivisa è lo stack, la cui dimensione massima è limitata a 8Mb (il flag `growsdown = 0`).

1.5 Creazione di VMA e mmap

La realizzazione di una VMA può essere invocata direttamente dal SO, o tramite la syscall `mmap()`. Le VMA possono essere **mapped** su un file (backing store) o **anonymous**, **shared** o **private** (la combinazione anonymous-shared non è trattata).

1.5.1 Mapped

Nella struct `vm_area_struct` sono contenuti i campi `vm_file` e `vm_pgoff`.

La funzione `mmap()` ha la seguente sintassi:

```
void * mmap(void * addr, size_t len, int prot, int flags, int fd, off_t off);
```

- `addr`: opzionale indirizzo di virtuale iniziale (Linux sceglie l'indirizzo disponibile più vicino)
- `len`: dimensione della VMA
- `prot`: permessi di accesso
- `flags`: `MAP_SHARED`, `MAP_PRIVATE` o `MAP_ANONYMOUS`
- `fd`: file descriptor

- `off`: offset nel file (multiplo della dimensione della pagina)

In caso di successo `mapp()` restituisce un puntatore all'area allocata.

La funzione `munmap()` elimina la mappatura dell'intervallo virtuale specificato:

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

1.5.2 Page Cache

è un meccanismo che serve ad evitare la rilettura da disco di pagine già caricate in memoria. è costituita da un insieme di pagine fisiche e strutture dati ausiliarie. Le strutture dati ausiliarie contengono l'insieme dei descrittori delle pagine fisiche presenti nella cache oltre che un meccanismo `Page_Cache_Index` per la riesca di una pagina identificata da `<indentificatore file, offset>`. Le pagine caricate nella Paga Cache non vengono liberate nessun processo le sta utilizzando, ma solo quando la Page Cache è piena.

Richiesta di accesso a una pagina virtuale mappata su file:

- determinazione file e offset: il file è indicato nella VMA, e l'offset è la somma tra l'offset della VMA rispetto all'inizio del file e l'offset della pagina virtuale rispetto all'inizio della VMA (evidentemente nella VMA può essere caricata solo una porzione di un file).
- Se la pagina è nella Paga Cache, la pagina virtual e viene semplicemente mappata, altrimenti viene allocata una nuova pagina fisica nella paga cache.

1.5.3 Scrittura sulle pagine

Scrittura su pagine condivise

La pagina fisica viene scritta e il bit di dirty viene settato. I cambiamenti sono immediatamente visibili da ogni processo.

Scrittura su pagine private

Il meccanismo di funzionamento è il COW. Le pagine modificate non mai ricopiate su disco, in quanto appartengono al singolo processo.

Scrittura su pagine anonime

Lo stack e l'heap sono pagine anonime. Tutte le pagine anonime sono mappate sulla **ZeroPage**, così che VMA trovi una pagina inizializzata a zero senza richiedere una nuova allocazione. Il meccanismo di funzionamento è il COW.

1.5.4 Mmap e fork

Le aree di memoria create tramite mmap si trasmettono ai figli durante una fork grazie al meccanismo di COW.

1.5.5 Condivisione degli eseguibili

Gli eseguibili sono VMA senza diritti di scrittura. Sono definite come aree private (seppur ciò sia irrilevante). Anche il linker dinamico utilizza le VMA private per condividere le pagine fisiche delle librerie.

Chapter 2

Gestione della memoria fisica

Le allocazioni sono di 2 tipi:

- **a grana grossa**: intere porzioni di memoria (`brk()`: unità di allocazione = pagina/gruppo di pagine)
- **a grana fine**: piccole strutture dati in porzioni gestite a grana grossa (`malloc()`

2.1 Allocazione e deallocazione

La RAM ha principalmente 3 usi:

- sistema operativo
- processi
- pagine lette da disco (buffer/disk cache)

2.1.1 Page reclaming

Prima che avvenga una deallocazione, vengono salvate su disco le pagine con dirty bit settato. La deallocazione ha una gerarchia di priorità:

- le pagine non più utilizzate dai processi vengono scaricate
- alcune pagine utilizzate dai processi vengono scaricate
- un processo viene ucciso: **OOMK** (Out Of Memory Killer)

Il comando `free` mostra lo stato della memoria. Ovviamente avendo la memoria swap a disposizione, l'OOMK avviene più tardivamente.

2.2 Allocazione della memoria a grana grossa

Linux cerca di allocare blocchi di pagine contigue e di frammentare la memoria il meno possibile: il DMA è così più efficiente e la memoria risulta più compatta.

- La dimensione di un blocco di pagine contigue è una potenza di 2, detta **ordine** (esiste una costante `MAX_ORDER`).
- Per ogni ordine esiste una lista che collega tutti i blocchi.

- Le richieste di allocazione indicano l'ordine del blocco desiderato.
- se il blocco richiesto non è disponibile, viene diviso in 2 blocchi (**buddies**). Se necessario il processo viene reiterato.
- quando un blocco viene liberato, se il suo buddy è libero, i 2 blocchi vengono riuniti.

2.3 Deallocazione della memoria a grana grossa

La memoria di un processo viene rilasciata appena il processo termina, mentre la disk cache cresce indefinitivamente.

Quando la memoria scende sotto un livello critico **minFree**, interviene la procedura **PFRA** (Page Frame Reclaiming Algorithm). PFRA necessita esso stesso di memoria, per questo l'esistenza del minFree. Per ridurre il numero di attivazioni di PFRA, si cercano di scaricare **maxFree** pagine.

La scelta delle pagine da liberare si basa sui principi di LRU.

2.4 Determinazione delle pagine da liberare

2.4.1 Quante

- **freePages**: numero di pagine libere.
- **requiredPages**: numero di pagine richieste da un processo o dal SO.
- **maxFree**: numero di pagine che il PFRA tenta di liberare.
- **minFree**: livello critico di memoria libera.

Il PFRA viene invocato se:

- `freePages - requiredPages < minFree`
- `kswapd` (kernel swap daemon): funzione invocata periodicamente se `freePages < maxFree`

Pfra cerca quindi di liberare `toFree = maxFree - freePages + requiredPages` pagine.

In caso di carico leggero `kswapd` causa rare attivazione dirette di PFRA, viceversa con uso intensivo della memoria.

Nel caso in cui il PFRA non riesce a liberare pagine e la memoria residua è insufficiente, viene invocato l'OMMK. L'OMMK chiama la funzione `select_bad_process()` che considera i seguenti criteri: il processo abbia molte pagine occupate, abbia svolto poco lavoro, abbia bassa priorità, non abbia privilegi di root, non gestisca componenti HW (rimangono altrimenti in stato incosciente) e non sia un kernel thread.

Potrebbe verificarsi uno o stato di **Thrashing**: i processi continuano a richiedere pagine che vengono deallocate senza fare progredire con l'esecuzione.

2.4.2 Quali

Le pagine si dividono in 4 categorie:

- **Non scaricabili**: pagine di sistema (statiche o dinamiche) e pagine della sPila dei processi.
- **Richiedono una Swap** per essere scaricate: pagine dati, uPila e Heap.
- **Buffer**: pagine mappate su file
- **Mappate su file eseguibile**: non necessitano di riscrittura.

Esistono 2 liste globali, le **LRU list**, che contengono tutte le pagine appartenenti ai processi:

- **Active list**: non possono essere scaricate
- **Inactive list**

L'ordine delle liste è un'approssimazione di LRU: l'x64 non tiene traccia del numero di accessi alla memoria, ma utilizza solamente un bit (**A**) di accesso presente nel TLB (translation lookaside buffer) gestito dall'HW. Questo bit viene settato ogni volta che la memoria viene acceduta, e riportato a 0 periodicamente dal kernel.

Oltre a ciò, ad ogni pagina viene associata una flag **ref** (referenced). Periodicamente `kswapd` esegue una funzione di controllo delle liste:

- Scansione della coda della active list:

```
if (A) {
    A = 0;
    if (ref) move Page to the head of active list;
    else ref = 1;
} else {
    if (ref) ref = 0;
    else move Page to the head of inactive list;
}
```

- Scansione della testa della inactive list:

```
if (A) {
    A = 0;
    if (ref) {
        ref = 0;
        move Page to the head of active list;
    } else ref = 1;
} else {
    if (ref) ref = 0;
    else move Page to the tail of inactive list;
}
```

- Caricamento di nuove pagine

  - se richieste da un processo vengono poste in testa alla active con `ref = 1`
  - le pagine di un nuovo processo possiedono lo stesso `ref` del padre e poste in testa alla stessa lista del padre.

- Eliminazione di pagine: swapped out o exit dal processo.

2.4.3 Scaricamento delle pagine

- Vengono prima scaricate le pagine della Page Cache non più utilizzate in ordine di NPF.
- Vengono poi scaricate le pagine fisiche della inactive list (che contiene NPV). Ovviamente una NPF viene scaricata solo se si incontrano tutte le NPV che a lei fanno riferimento.

2.5 Swapping

Il meccanismo di swapping necessita almeno una **Swap Area** (risiedente su file o partizione). Questa contiene dei **Page slot** che hanno dimensione di 1 pagina, identificati da <SwapArea, PageSlot> (**SWPN**). A ogni Swap Area è associato un **descrittore**, contente tra le altre cose un `swap_map_counter` per ogni slot, che tiene traccia del numero di PTE (page table entries) riferiti all'area swappata.

2.5.1 Swap Out

Quando una pagina viene liberata, se il bit di dirty è 0, viene eliminata, altrimenti viene salvata. Se la pagina è mappata su file, viene salvata in tale file; se invece è anonima, viene salvata nella Swap Area.

- Allocazione di un SWPN
- Copia di NPF in SWPN
- aggiramento di `swap_map_counter`
- ogni PTE che condivideva la pagina fisica viene aggiornata: NPF viene sostituito da SWPN, e il bit di presenza viene resettato.

Per determinare se una pagina fisica condivisa sia dirty, oltre a controllare il bit di dirty della pagina stessa, bisogna controllare nel TLB se le pagine virtuali a lei associata siano dirty.

2.5.2 Accesso alla swap

- page fault
- gestore del page fault chiama `swap_in`
- allocazione pagina fisica in memoria
- Copia di SWPN (indicato in PTE) in NPF
- ogni PTE che condivideva la pagina fisica viene aggiornata: SWPN viene sostituito da NPF, e il bit di presenza viene settato.

2.5.3 Swap In

Se avviene la Swap In di una pagina, e successivamente una Swap Out senza che ci siano state modifiche al contenuto della pagina stessa, Linux cerca di riutilizzare lo stesso SWPN. Utilizza quindi una **Swap Cache**, una struttura dati simile alla Page Cache. Una pagina appartiene alla Swap Cache viene registrata nello **Swap Cache Index** se è presente sia in Memoria, che nella Swap Area. Le pagine presenti in Swap sono solo anonime (e quindi private nella nostra semplificazione). Nel momento in cui avviene una Swap In:

- La pagine viene copiata in memoria fisica, marcata in sola lettura come pagina condivisa (dalla swap e dal processo).
- Nella Swap Cache Index viene inserito un descrittore contenente i riferimenti sia alla Pagina Fisica che al Page Slot.

Se poi la pagina viene scritta, si verifica un Page Fault

- La pagina diventa privata (non appartiene più alla Swap Area)
- La sua protezione viene posta a W
- `swap_map_counter--`
- Se `swap_map_counter == 0` il Page Slot viene liberato nella Swap Area.

Successivamente a una Swap In

- la NPV viene inserita in testa alla lista active con `ref = 1`
- altre NPV condivise all'interno della stessa pagina vengono poste in coda alla lista active con `ref = 0` *(penso ci sia un errore: le NPV in swap dovrebbero essere solo private nella nostra semplificazione)*

Chapter 3

Tabella della pagine

3.1 Struttura della memoria virtuale del SO

Gli indirizzi virtuali si estendono da `FFFF 8000 0000 0000` a `FFFF FFFF FFFF FFFF` per un totale di 2^{47}B = 128TB.

- Codice e dati statici: 0.5GB
- Moduli a caricamento dinamici: 1.5GB
- Mappatura della memoria fisica: 64TB (50%)
- Strutture dinamiche
- Mappature memoria virtuale
- altro

3.1.1 Page offset

Il sistema operativo opera su indirizzi virtuali (come gli altri processi), ma deve anche essere in grado di accedere agli indirizzi fisici, ad esempio per gestire la tabella della pagine (una struttura dati HW molto grande). Questo problema è in parte risolto mappando 1:1 la memoria fisica. L'indirizzo iniziale di questa area virtuale è chiamato `PAGE_OFFSET`. Le aree di sPila dei processi sono allocate dal SO in un indirizzo fisico e mappate sul corrispondente indirizzo virtuale per essere utilizzate.

3.2 Paginazione in x64

3.2.1 Tabella della pagine

La MMU (Memory Management Unit) è l'unità di gestione della memoria a livello HW: può essere un componente della CPU o esterno.

L'indirizzo di una pagina virtuale (di 4KB) è suddiviso in 12 bit di offset e 36 bit di NPV (per un totale di 2^{36} pagine virtuali). Ovviamente le pagine virtuali non sono tutte allocate sulla TP: perciò la TP è organizzata come un albero su 4 livelli:

- i 36 bit di NPV sono suddivisi in 4 parti da 9 bit
- ogni gruppo da 9 bit rappresenta l'offset nella **directory**: una tabella le cui right sono chiamate **PTE** (Page Table Entry)

- La dimensione di ogni directory è 4KB: ogni PTE occupa 64 bit.
- L'indirizzo della directory principale è contenuto nel registro `CR3` (Control Register 3) della CPU.

3.2.2 Page Walk

Per tradurre un NPV in NPF:

- accedi a `CR3`
- leggi la PTE indicata dall'offset di primo livello di NPV
- leggi l'indirizzo di base della directory di livello inferiore
- ripeti la procudura per ogni livello

Ad ogni livello, la PTE non contiene solo l'indirizzo del livello inferiore, ma anche dei **flags**, memorizzati nei 12 bit bassi non utilizzati (essendo l'offset della TP 0).

3.2.3 Translation Lookaside Buffer

Essendo che la Page Walk richiede 5 accessi a memoria, l'architettura x64 possiede un TLB in cui conservare le corrispondenze tra NPV e NPF più recenti. Il TLB è gestito dall'MMU in maniera trasparente al software. Se il registro CR3 viene modificato la MMU invalida tutte le PTE del TLB, escluse quelle globali (flag G).

3.2.4 Struttura della TP

Le 4 directory sono:

- **PGD**: Page Global Directory
- **PUD**: Page Upper Directory
- **PMD**: Page Middle Directory
- **PT**: Page Table

3.2.5 Avviamento del SO

Deve esistere un meccanismo che all'avvio del sistema permetta di caricare la TP. Durante l'avvio le funzioni di caricamento accedono direttamente alle memoria fisica (paginazione non attiva).

La tabella delle pagine è infatti memorizzata sulla memoria fisica anch'essa. La tabella della pagine, quindi, rimappa anche se stessa.

3.2.6 TP del Kernel

Non esistendo una TP del SO, il SO viene mappato nella metà superiore dalla TP di ogni processo. Le metà superiori di ogni processo puntano alla stessa sottodirectory, relativa al SO (memorizzata fisicamente una volta sola).

3.3 Dimensione della TP

- una PTE di PMD mappa 512 x 4KB = 2MB
- una PTE di PUD mappa 512 x 2MB = 1GB
- una PTE di PGD mappa 512 x 1GB = 512GB

L'occupazione percentuale di memoria nella TP rispetto alla memoria fisica diminuisce col numero della pagine fisiche utilizzate tendendo a 1/512. Con programmi molto piccoli, tuttavia può essere maggiore di 1.

3.4 TP VMA e thread

Gli stack crescono verso indirizzi bassi, mentre le VMA verso indirizzi alti. La pagina finale di uno stack è la prima pagina di una VMA, e viceversa.

I thread condividono la stessa TP del padre, differendo solo per quanto riguardo lo stack. Ogni thread ha bisogno di 2048 + 1 pagina per lo stack, e quindi 4 PT + 1 pagina.

3.5 Gestione della TP da software

- `mk_pte()` covert in una PTE un numero di pagina fisica e una serie di flag.
- `pdg_alloc()`, `pud_alloc()`, `pmd_alloc()`, `pte_alloc()` allocano e inizializzano intere pagine della TP.
- `pgd_free()`, `pud_free()`, `pmd_free()`, `pte_free()` liberano intere pagine della TP.
- `set_pdg()`, `set_pud()`, `set_pmd()`, `set_pte()` modificano una PTE.

Quando viene richiesto l'accesso a una NPV non ancora mappata sulla TP, il So deve inserire una nuova PTE nella directory PT (se questa già esiste), oppure allocare una nuova PT, e inserire la PTE nella directory PMD. Se questa non esiste la procedura viene ripetuta.

3.6 Approfondimenti

3.6.1 TLB

La TLB è gestita in modo trasparente dall'HW. La TLB flush viene eseguita ogni volta che viene modificato il valore di CR3 (ad esempio durante un context switch). Le pagine di sistema più utilizzate sono marcate come globali, e non vengono perciò eliminate.

3.6.2 Cache

Anche la cache è gestita in modo trasparente dall'HW.

3.6.3 Context switch

La funzione `schedule()`, prima di invocare la macro `switch_to(prev, next)`, invoca la funzione `switch_mm(oldmm, mm, next)`. Questa funzione invoca la macro `load_cr3(next->pgd)` che carica in CR3 il nuovo valore.