Sicurezza del Software

Riccardo Aziani

Ottobre 2024

Indice

Capitolo 1

Spazio di memoria

Non è immediato definire cosa sia la memoria, perché è necessario scegliere un **livello di astrazione** a cui fare riferimento: per un programmatore, la memoria potrebbe essere un insieme di variabili tipate; invece, per un ingegnere elettronico potrebbe essere un insieme di celle elettroniche.

A seconda del livello di astrazione la memoria può essere definita in modi diversi.

1.1 Astrazioni di memoria

Per quanto riguarda le astrazioni a livello software, ci sono pricipalmente tre livelli.

1.1.1 Memoria fisica

È il livello più basso di astrazione. Ha una struttura semplice: è una **lunga sequenza di byte** dove ogni byte ha un **indirizzo**; il primo byte ha indirizzo 0, il secondo 1, e così via.

È possibile leggere o scrivere dei byte in questa memoria usando delle istruzioni del processore (che parlerà effitivamente con i banchi di RAM).

Problematiche dell'utilizzo della memoria fisica

Questo strato di memoria non è adatto alle esigenze dei sistemi moderni, come ad esempio la programmazione, perché?

Immaginiamo di avere due programmi in esecuzione nello stesso momento sulla stessa macchina; entrambi questi programmi devono fare delle operazioni sulla memoria, operando quindi su certi indirizzi di memoria.

Se questi programmi fossero scritti scegliendo gli stessi indirizzi, finirebbero per *"pestarsi i piedi"*. Da una parte è problematico far convivere applicazioni diverse, assicurandosi che utilizzino indirizzi di memoria diversi; dall'altra ci sono problemi di sicurezza nell'usare direttamente la memoria fisica: un'applicazione malevola potrebbe andare a toccare la memoria di un'altra applicazione influenzandone il comportamento.

Per queste ragioni nessuna applicazione moderna utilizza direttamente la memoria fisica, ma una sua astrazione chiamata *memoria virtuale*.

1.1.2 Memoria virtuale

È una astrazione della memoria fisica realizzata dal sistema operativo, che è l'unico componente che lavora effitivamente sulla memoria fisica.

A livello di *contenuto*, segue la stessa struttura della memoria fisica: è una sequenza di byte indirizzabili.

Però, a differenza della memoria fisica, **ogni processo di sistema ha uno spazio di memoria indipendente**: ogni programma ha la sua memoria **indipendente** ed **isolata** dalla memoria virtuale degli altri processi.

In questo modo, si evita che i processi *"si pestino i piedi"* o che uno vada a toccare la memoria dell'altro.

Mapping

Per realizzare questa astrazione si utilizzano dei mapping: ogni area della memoria virtuale è mappata a un'area della memoria fisica. Questo processo è realizzato dal sistema operativo.

Ne consegue che è possibile avere lo stesso indirizzo virtuale in due processi diversi, mappati a due indirizzi fisici diversi.

La memoria virtuale non è completamente accessibile, ma solo alcune aree che effitivamente servono sono mappate.

Flag di protezione

La memoria virtuale fornisce delle **funzionalità di protezione degli accessi**: si può marcare ciascuna area della memoria virtuale come solo lettura, leggibile/scrivibile, controllare se può contenere codice che può essere eseguito, e così via.

Si può dunque controllare il tipo di accesso che è consentito fare a determinate aree di memoria virtuale.

1.1.3 Dati tipati (variabili)

L'astrazione dei tapi tipati viene realizzata dai linguaggi di programmazione sopra la memoria virtuale. Quando dichiaro una variabile, il linguaggio di programmazione si tiene un pezzettino di memoria virtuale per contenere quella variabile.

Quando scrivo (ad esempio) un intero in quella variabile, il linguaggio di programmazione converte quell'intero in una sequenza di byte per poi metterlo in memoria.

Quando va a leggere quell'intero, viene letta la sequenza di byte dalla memoria per poi riconvertirlo in un intero.

1.2 Spazio virtuale Linux userspace

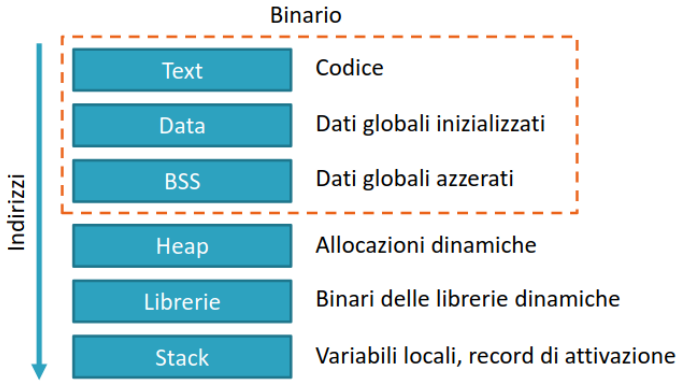
Andiamo a vedere com'è fatto la memoria virtuale di un processo utente su Linux; il kernel avrà uno spazio virtuale diverso.

Quando viene **lanciato un eseguibile** (binario), viene creato un processo e quell'eseguibile viene **caricato in memoria virtuale**.

Il binario contiene varie parti, come quella in cui è contenuta il codice o quella in cui sono contenuti i dati; tutte queste informazioni vengono caricate in memoria.

Altre aree interessanti in memoria virtuali sono:

- **heap:** contiene tutte le allocazioni dinamiche (*malloc() in C, new() in C++*)
- **librerie** esterne importate nel programma
- **stack:** è dove vengono memorizzate le variabili locali e tutta una serie di informazioni utilizzate per il flusso di controllo del programma, come gestire le chiamate a funzione



Il **binario** è posizionato ad indirizzi relativamente **bassi**, mentre lo **stack** è l'area che sta ad indirizzi più **alti**.

1.2.1 Un esempio

In Figura 1.1 viene mostrato un programma che stampa quattro indirizzi:

- indirizzo della funzione main, quindi del codice macchina del programma
- indirizzo di una variabile locale
- indirizzo di una locazione sull'heap
- indirizzo di stack

```c
int var_global;

int main()
{
    int var_stack;

    int *ptr_heap = malloc(sizeof(int));

    printf("main       @ %p\n", &main);
    printf("var_global @ %p\n", &var_global);
    printf("ptr_heap   = %p\n", ptr_heap);
    printf("var_stack  @ %p\n", &var_stack);

    getchar();

    return 0;
}
```

Figura 1.1: Programma C

In Figura 1.2 viene mostrato il risultato dell'esecuzione del programma.s

```
~/cc21/o/ss1_demos ❯ bin/address_space
main       @ 0x401146
var_global @ 0x404038
ptr_heap   = 0x1cec2a0
var_stack  @ 0x7ffcaa7adbd4
```

Figura 1.2: Esecuzione del programma C

In Linux è possibile vedere il layout della memoria virtuale di un processo usando il *file system* `'proc'`: `/proc` ha una *sub-directory* per ogni processo del sistema (nominata con l'id del processo nel sistema).

Ciò che viene fatto in Figura 1.3 è stato usare `pgrep` per trovare l'id associato al processo *address_space* (il programma C).

In ogni linea:

- il primo numero è l'indirizzo di partenza in esadecimale, e dopo '-' c'è l'indirizzo di fine
- dopo ci sono i flag di protezione (`r`, `rw`, `x`)

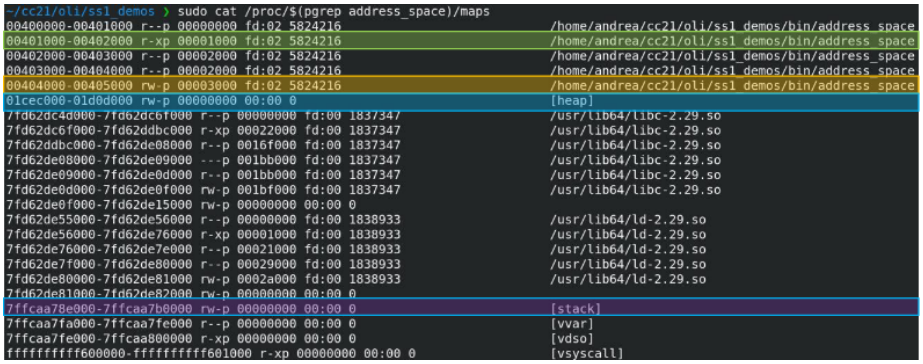


Figura 1.3: Layout della memoria virtuale del processo

L'indirizzo del ***main*** è leggibile ed eseguibile.

La ***variabile globale*** è leggibile e scrivibile; lo stesso vale per l'***heap***.

Dopo ci sono una serie di mapping di *libc* e *ld*, una serie di librerie standard caricate in tutti i processi. Se il processo utilizza delle liberie aggiuntive saranno caricate in questa zona.

Infine, negli indirizzi più alti, si trova lo ***stack***.