Esecuzione con pagine mappate ma non materializzate in RAM

mercoledì 17 maggio 2023 12:19

Esecuzione con pagine mappate ma non materializzate in RAM

Entriamo nei dettagli della memoria virtuale.

La memoria virtuale è uno schema di gestione della memoria - quindi uno schema di gestione dei nostri address space dove le nostre applicazioni software possono svolgere le loro attività - supportato da tutti i sistemi operativi convenzionali, e l'idea della memoria virtuale risiede esattamente in questo:
***È possibile, con la memoria virtuale, che soltanto un sottoinsieme delle pagine mappate sia presente in RAM*.**
Quindi noi virtualizziamo come se tutte queste pagine mappate siano presenti in RAM, perché di fatto sono utilizzabili dal software, però nella realtà non lo sono, NON SONO effettivamente materializzate in RAM.

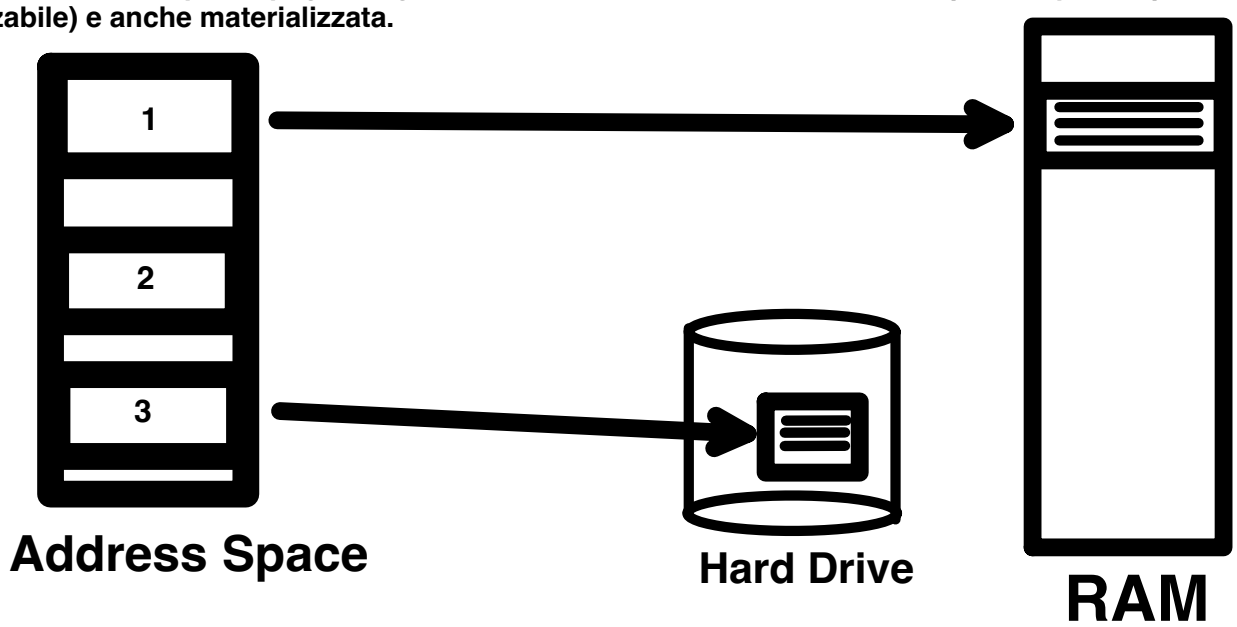
- le cause della non presenza in RAM di pagine mappate sono due
  - ✓ le pagine non sono mai state materializzate
  - ✓ le pagine sono state materializzate ma poi sono state portate fuori RAM

Le pagine che non sono mai state materializzate magari sono pagine che sono state mmappate - quindi l'applicazione ha richiesto di volerle utilizzare - ma ancora non ha mai acceduto. Magari sono pagine anonime e verranno materializzate in RAM quando andremo ad accedere.
Le pagine materializzate e poi portate fuori dalla RAM hanno il concetto legato allo "SWAP-OUT" che noi possiamo eseguire, delle informazioni che caratterizzano un address space. Quindi un address space suddiviso in pagine è tale per cui che alcune di queste pagine che sono correntemente materializzate in RAM, vengono tolte dalla RAM.

- pagine materializzate ma portate fori RAM si dicono ‘swapped-out’
- rispetto ad approcci di swapping tradizionali, la memoria virtuale permette di avere processi parzialmente swappati fuori RAM e parzialmente presenti in RAM

La memoria virtuale è molto interessante, perché in realtà quando noi poi lavoriamo in uno schema di paginazione - quindi l'address space è suddiviso in pagine - essa ci permette di avere processi che sono parzialmente swappati fuori dalla RAM e altre parzialmente presenti in RAM.
Se noi abbiamo la gestione a livello paginato, quindi l'idea che un contenitore è suddiviso in zone differenti l'una dall'altra, mantenere o non mantenere queste zone in memoria, le possiamo attuare per zona (quindi per pagina).

Lo scenario della memoria virtuale, ci fa vedere l'address space in questo modo qua:
Supponiamo che la prima pagina logica sia visibile all'interno della RAM da qualche parte, quindi è una pagina mappata (utilizzabile) e anche materializzata.



Consideriamo un'altra pagina logica utilizzabile, quindi mappata nell'address space, quindi l'applicazione ha richiesto al sistema operativo di voler utilizzare quella zona dell'address space, ma quella zona non è mai stata acceduta, quindi quella pagina non esiste in RAM.
Ma se non è mai esistita in RAM non può esistere né tantomeno all'interno della swap area. Ovvio e scontato.
Ma poi sotto abbiamo una terza pagina che, non è più all'interno della RAM, ma è all'interno della swap area. Quindi è su un hard drive, cioè è stata portata fuori dalla RAM e caricata all'interno di quella zona.
Qui abbiamo un address space in cui qualcosa è presente in RAM, qualcosa non è mai esistito neanche in RAM e che quindi non esiste nemmeno all'interno di un hard drive (non essendo mai stato in RAM), e qualcosa che è esistito in RAM ma poi è stato swappato fuori.

Gestire un address space secondo queste regole ha dei vantaggi.

- Permettere l'esecuzione di processi il cui contenitore logico (quindi lo spazio di indirizzamento) eccede le dimensioni della memoria di lavoro.
Schemi in cui il contenitore O è tutto in RAM O è tutto fuori RAM, chiaramente non permettono questo, perché per essere completamente swapped in ram, un contenitore deve essere di taglia adeguata alla taglia della RAM.
Ma con lo scenario appena disegnato qui, stiamo mandando in esercizio un'applicazione in cui la pagina 1 è correntemente in RAM, quindi il software che è li presente è utilizzabile dall'applicazione perché è in RAM, la pagina 2 non c'è perché non è mai stata utilizzata sino ad ora, mentre la pagina 3 è stata portata fuori. Non fa niente. Mandiamo comunque in esercizio l'applicazione per esempio, nel caso in cui adesso - per questa applicazione -, serve ritoccare solo la zona 1, e questo è un vantaggio enorme.

- Permettere di aumentare il numero dei processi che possono essere mantenuti contemporaneamente in memoria.
Perché manteniamo solo un sottoinsieme delle pagine mappate all'interno dell'address space, e quindi questo ci permette altresì di aumentare il grado di multiprogrammazione.

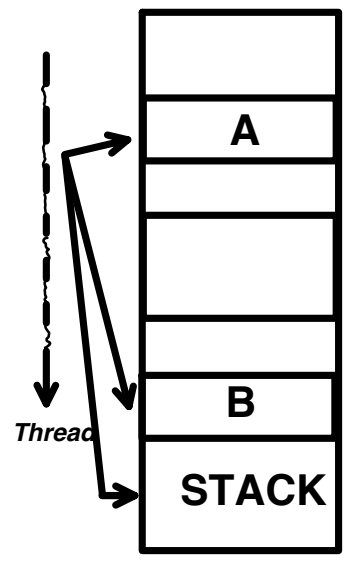
- Permettere la riduzione del tempo necessario alle operazioni di swapping.
L'applicazione disegnata è tale per cui, di questa applicazione abbiamo swappato solo la pagina 3, e questo ha comportato meno tempo rispetto al dover eseguire lo swap out di tutte le pagine logiche, se fossero stato presenti correntemente nella

memoria.
La stessa situazione è analoga per riportare un'applicazione - o parti dell'applicazione - all'interno della RAM. Chiaramente se avevamo portato fuori memoria solo una porzione di queste pagine che, originariamente, erano all'interno della memoria, riportarle in memoria è un lavoro minore rispetto a dover riportare in memoria tutte le pagine dell'applicazione.
Quindi riduciamo anche il numero delle operazioni di I/O che andiamo ad eseguire quando dobbiamo effettuare un'operazione di Swap Out e Swap In.

Detto questo, qual è il motivo per cui su tutti i sistemi operativi convenzionali abbiamo la memoria virtuale, quindi abbiamo una gestione dell'address space esattamente come descritta nel disegno sopra?

Motivazioni per l'efficacia

Quando noi eseguiamo il software abbiamo una località spaziale ed una località temporale. Quindi abbiamo un address space di alcune pagine, un thread che sta eseguendo qualcosa, e magari in questo momento il thread sta eseguendo alcune parti di software che sono nella pagina A, e alcuni dati nella pagina B.

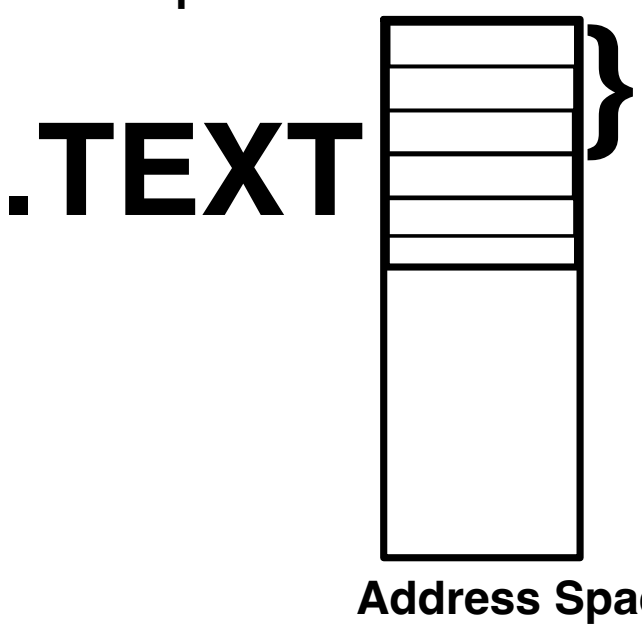


Address Space

Poi magari starà utilizzando qualcosa che è nella stack area, quindi in un'altra pagina sotto.

Quindi questo thread utilizzerà per un po' di tempo istruzioni che sono nella pagina A, toccherà dati nella pagina B e toccherà una certa zona della stack area. Tutte le altre pagine non servono per ora, chiaramente dovuto al principio di località.
Per questo thread e per questa applicazione potremmo per un po' mantenere in RAM soltanto queste 3 pagine (A,B,STACK), e quindi questo ci permette di dire che secondo i principi di località la memoria virtuale ci può dare comunque ci può dare un vantaggio prestazionale.
Mantenere anche solo parzialmente le informazioni di un address space all'interno della RAM può essere efficace dal punto di vista prestazionale.

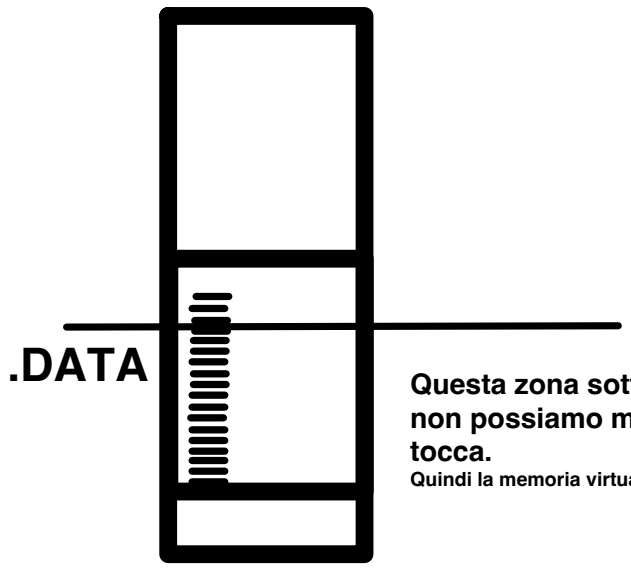
Un secondo vantaggio è quello per cui tante applicazioni all'interno del loro address space hanno comunque sia blocchi di codice che strutture dati che vengono ad essere utilizzate solo rarissimamente. Potremmo avere che, all'interno di un address space, in una zona .TEXT, dove ci sono varie pagine, magari le prime pagine sono utilizzate per implementare funzioni che vengono chiamate soltanto nel momento in cui eventualmente c'è necessità di fare delle cose quando accadono degli eventi molto particolari.



Address Space

Se questi eventi non accadono, queste pagine sono completamente inutilizzate da questa istanza di applicazione.
<u>Ma allora è utile averle in memoria RAM</u>?
Chiaramente la risposta è NO.

Altra cosa interessante è che in molte applicazioni noi abbiamo il sovradimensionamento delle strutture dati (array, tavole, etc..). Quindi dato un address space all'interno della zona DATI abbiamo rappresentato un array enorme di cui magari per questa specifica istanza di applicazione, utilizziamo solo una prima parte. Ma la cosa bella è che questo potrebbe essere anche semplicemente un array di caratteri che noi utilizziamo per ospitare dati dallo std::in e magari sono stringhe sempre corte. Quindi alla fine utilizziamo solo la parte iniziale di questo array.



Address Space

Questa zona sotto, e quindi anche le relative pagine, non sono mai utilizzate. E queste non possiamo mai materializzarle in memoria fisica, fino a che l'applicazione non le tocca.
Quindi la memoria virtuale ci da dei vantaggi importantissimi in tanti contesti.

Vantaggi della memoria virtuale

- permette l'esecuzione di processi il cui spazio di indirizzamento <u>eccede</u> le dimensioni della memoria di lavoro
- permette di <u>aumentare il numero di processi</u> che possono essere mantenuti contemporaneamente in memoria (aumento del grado di multiprogrammazione)
- permette la <u>riduzione del tempo</u> necessario alle operazioni di swapping

Motivazioni per l'efficacia

- principi di località spaziale e temporale (l'accesso ad un indirizzo logico implica, con elevata probabilità, accesso allo stesso indirizzo o ad indirizzi adiacenti nell'immediato futuro)
- esistenza di porzioni di codice per la gestione di condizioni di errore o opzioni di programma di cui rarissimamente viene richiesta l'esecuzione
- sovradimensionamento di strutture dati (array, tavole ...) per le quali viene allocata più memoria di quella necessaria per la specifica istanza del problema

La memoria virtuale ci permette di avere un'esecuzione che prevede l'idea che un AB ha, ad ogni istante di tempo, soltanto alcune pagine realmente materializzate all'interno della RAM.