Gestione della memoria: i servizi

Pietro Braione

Reti e Sistemi Operativi – Anno accademico 2025-2026

Argomenti

- Il problema dell’allocazione della memoria
- Spazio di indirizzamento virtuale
- Le API POSIX per la gestione della memoria

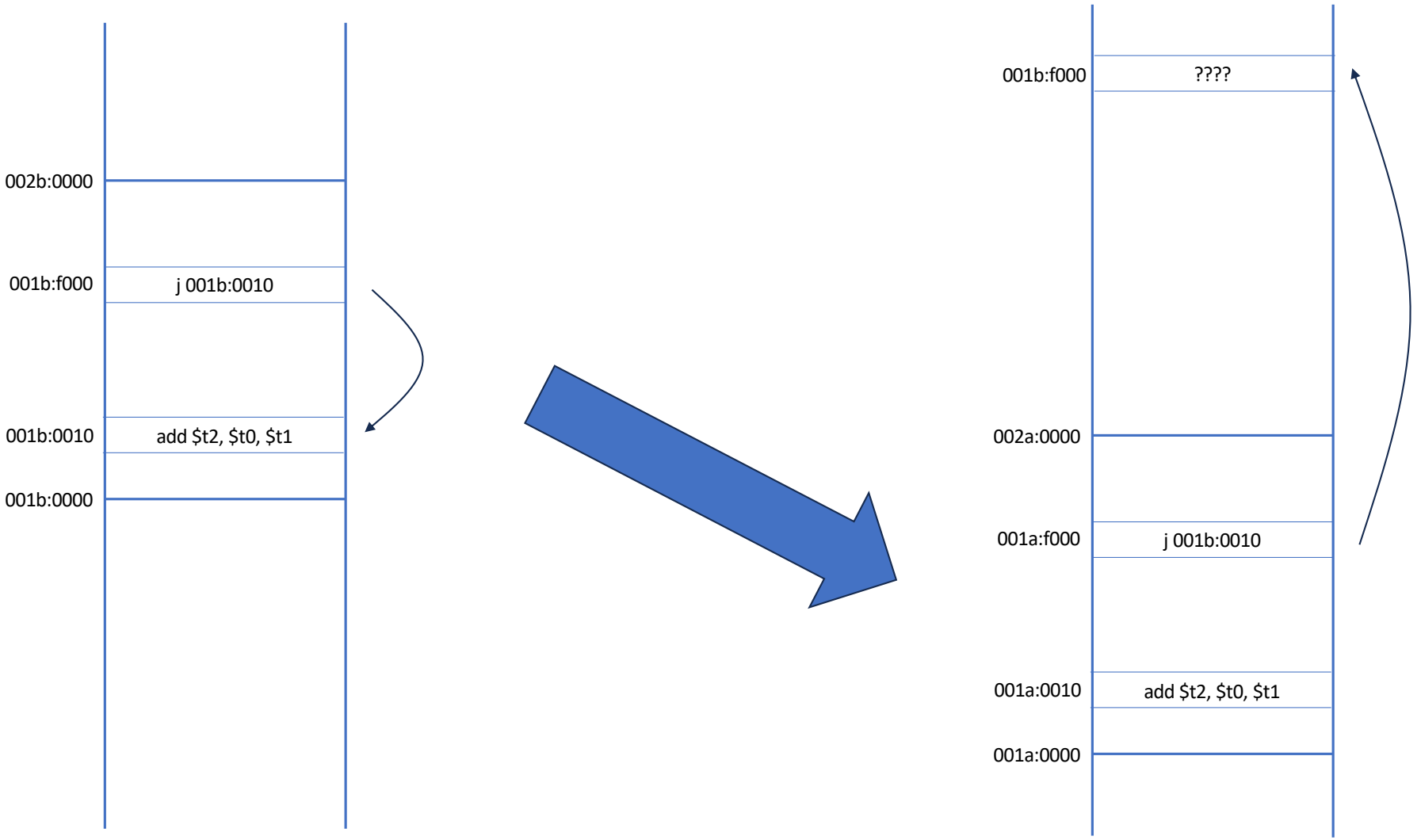
Il problema dell’allocazione della memoria

Il problema dell'allocazione della memoria

- Perché un programma possa andare in esecuzione esso deve avere a disposizione:
  - Il processore, per eseguire il codice
  - La memoria centrale, per memorizzare il codice e i dati sul quale il codice opera
- Solo nei sistemi operativi più semplici un solo programma alla volta è in memoria: nei moderni sistemi operativi molti programmi sono contemporaneamente in memoria in uno stesso istante
- Secondo la terminologia precedentemente introdotta: più immagini di più processi sono presenti contemporaneamente nella memoria centrale
- Il sistema operativo deve, pertanto, allocare porzioni di memoria centrale ai diversi processi in funzione delle necessità di tali processi

Lo spazio di indirizzamento

- Ogni processo ha a disposizione un proprio **spazio di indirizzamento**, ossia un insieme di indirizzi di memoria che può usare
- Nei primi sistemi operativi tale spazio di indirizzamento era il range di indirizzi di memoria centrale che veniva assegnato al processo
  - Ad esempio, se l'immagine di un certo processo avesse avuto dimensione 1 MB e fosse stata caricata in memoria centrale dall'indirizzo 001B:0000…
  - …il suo spazio di indirizzamento sarebbe stato 001B:0000 – 002B:0000
- Questo però non permette di caricare lo stesso programma in zone diverse di memoria!
- Consideriamo ad esempio un compilatore che deve compilare un programma C in linguaggio macchina...

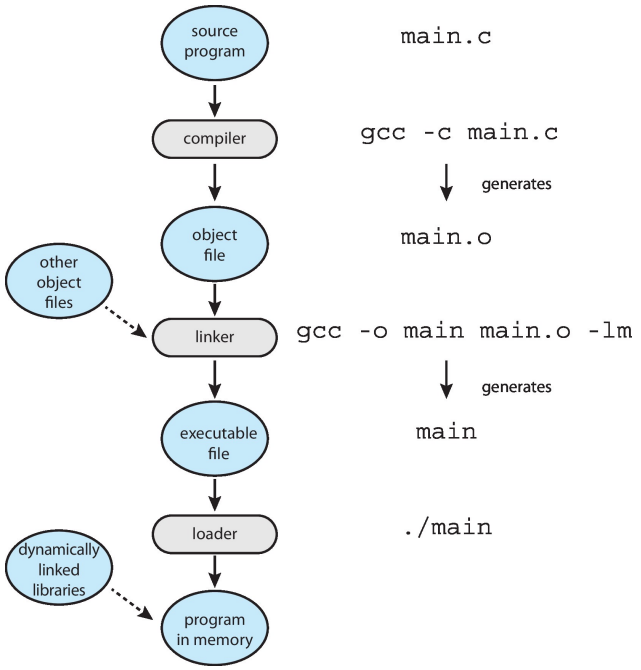

002b:0000
001b:f000
j 001b:0010
001b:0010
add $t2, $t0, $t1
001b:0000
001b:f000
????
002a:0000
001a:f000
j 001b:0010
001a:0010
add $t2, $t0, $t1
001a:0000

Associazione degli indirizzi

- Un sistema operativo moderno di regola carica uno stesso programma, in momenti diversi, in diverse aree di memoria (dove trova spazio)
- Come fa, pertanto, un'istruzione macchina di un programma a far riferimento ad una certa locazione di memoria, se il suo indirizzo non è noto a priori ma dipende da dove il programma viene caricato?
- Prima possibilità: **position-independent code (PIC)**, ossia il codice macchina deve usare solo modi di indirizzamento relativi
- Seconda possibilità: vengono tradotti gli indirizzi assoluti negli indirizzi corretti in funzione dell'indirizzo di caricamento
- Questa operazione di «traduzione» è detta di **associazione** (**binding**) **degli indirizzi**

Loader e linker

- Un programma sorgente è compilato in un file oggetto che deve poter essere caricato a partire da qualsiasi locazione di memoria fisica (**file oggetto rilocabile**)
- I **linker**, o linkage editor, combinano più file oggetto (diversi file sorgente + librerie) per formare un file eseguibile
- I **loader** si occupano di caricare in memoria i file eseguibili nel momento in cui devono essere eseguiti



Librerie dinamiche

- I loader (o ulteriori linker dinamici) effettuano il linking delle **librerie dinamiche**
- Le librerie dinamiche vengono collegate quando il programma è caricato o durante l'esecuzione del programma stesso
- Vantaggio delle librerie dinamiche: possono essere condivise tra diversi programmi, riducendo le dimensioni dei programmi stessi e risparmiando memoria
- Ulteriore vantaggio: se aggiorno la libreria, non devo ricompilare i programmi che la usano

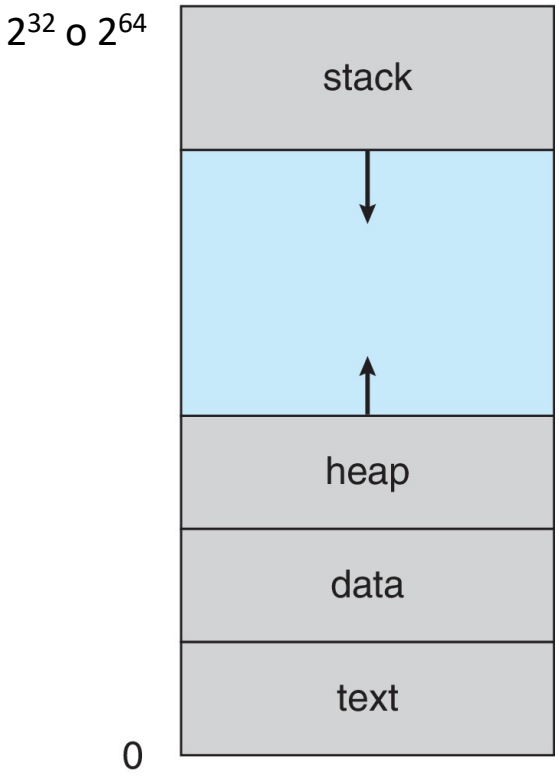
Associazione degli indirizzi: varianti

- L'associazione degli indirizzi può essere fatta in tre momenti diversi:
  - **In compilazione**: il linker, a partire dall'indirizzo di caricamento, effettua il binding e genera **codice assoluto**
  - **In caricamento**: il linker genera **codice rilocabile** e il loader, a partire dall'indirizzo di caricamento, effettua il binding al momento del caricamento in memoria del codice
  - **In esecuzione**: il binding viene effettuato dall'hardware dinamicamente mentre il codice viene eseguito
- Vantaggi e svantaggi:
  - In compilazione: soluzione semplice, ma se cambia l'indirizzo di caricamento il codice va ricompilato (si possono, ad esempio, avere n versioni per n diversi indirizzi di caricamento)
  - In caricamento: permette di variare liberamente l'indirizzo di caricamento da esecuzione ad esecuzione, ma è una soluzione lenta che difficilmente permette di rilocare (spostare) l'immagine di un processo *durante* la sua esecuzione; inoltre l'eseguibile deve contenere delle opportune tabelle che indichino le istruzioni macchina da modificare
  - In esecuzione: soluzione rapida che permette di rilocare l'immagine di un processo anche durante la sua esecuzione, e di proteggere la memoria centrale non assegnata ad un processo, ma richiede il supporto dell'hardware
- Il binding in esecuzione è quello usato per le applicazioni in tutti i sistemi operativi moderni; il binding in compilazione può essere usato per alcuni eseguibili speciali, come il kernel o i bootloader, di cui si sa a priori l'indirizzo di caricamento

Spazio di indirizzamento virtuale

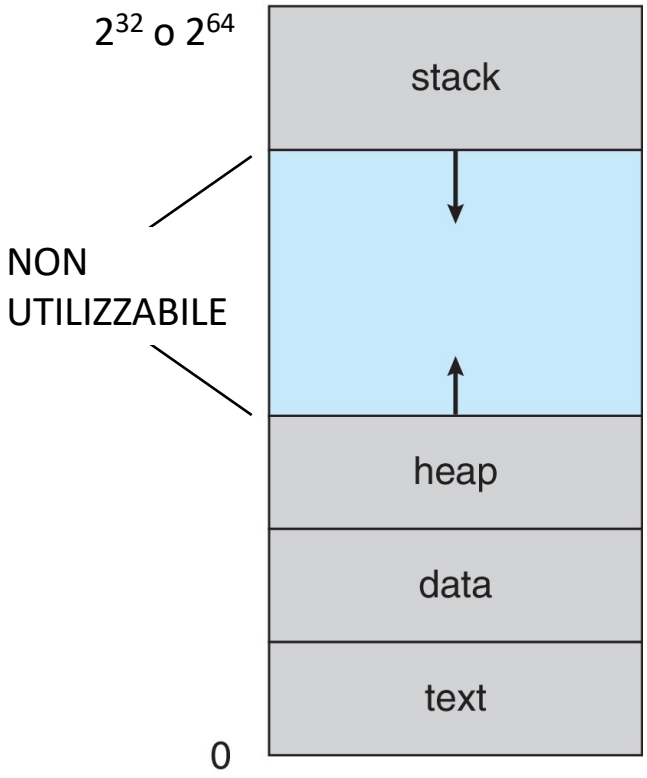
Spazio di indirizzamento virtuale (1)

- I sistemi operativi moderni offrono ai processi un'astrazione detta **spazio di indirizzamento virtuale**, o **virtual address space (VAS**)
- Uno spazio di indirizzamento virtuale è uno spazio di indirizzamento indipendente dagli indirizzi fisici della memoria centrale nella quale l'immagine è caricata
- Tale spazio di indirizzamento tipicamente si estende dall'indirizzo 0 al massimo indirizzo consentito dall'architettura del processore
- Tecniche di associazione degli indirizzi fanno corrispondere lo spazio di indirizzamento virtuale del processo con la regione (o le regioni) di memoria centrale che la sua immagine occupa
- In tal modo compilatore e linker possono lavorare «come se» avessero a disposizione tutta la memoria!



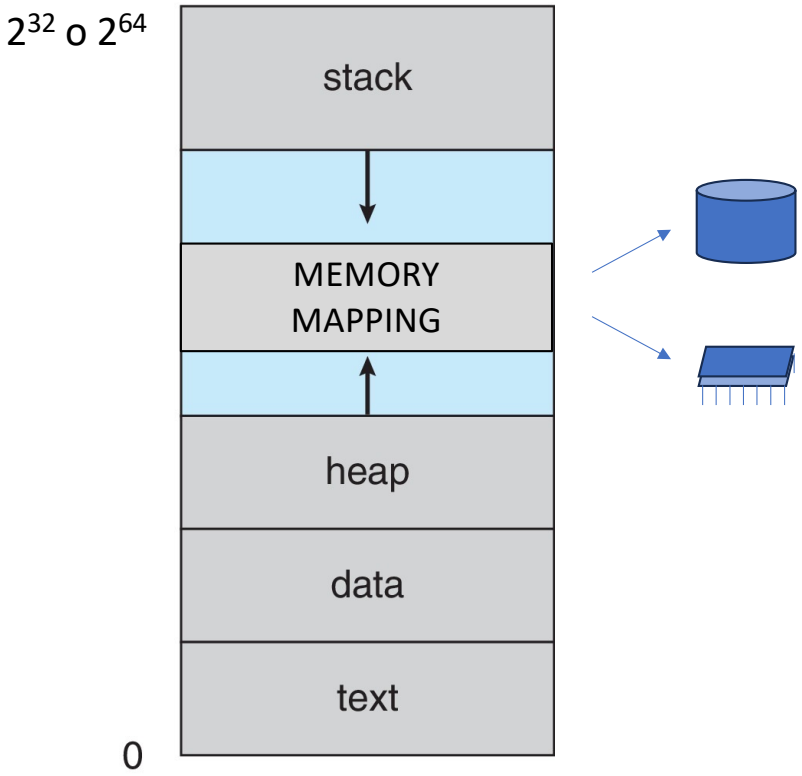
Spazio di indirizzamento virtuale (2)

- Lo spazio di indirizzamento virtuale di un processo è di regola molto più ampio della memoria centrale
- Questo implica che buona parte dello spazio di indirizzamento virtuale non è utilizzabile dal processo perché non è associato a nessuna regione di memoria centrale
- Tale regione inutilizzabile è di solito compresa tra stack e heap
- Stack e heap possono essere dinamicamente estesi e ridotti utilizzando opportune API che mappano della memoria centrale su parte della regione inutilizzabile



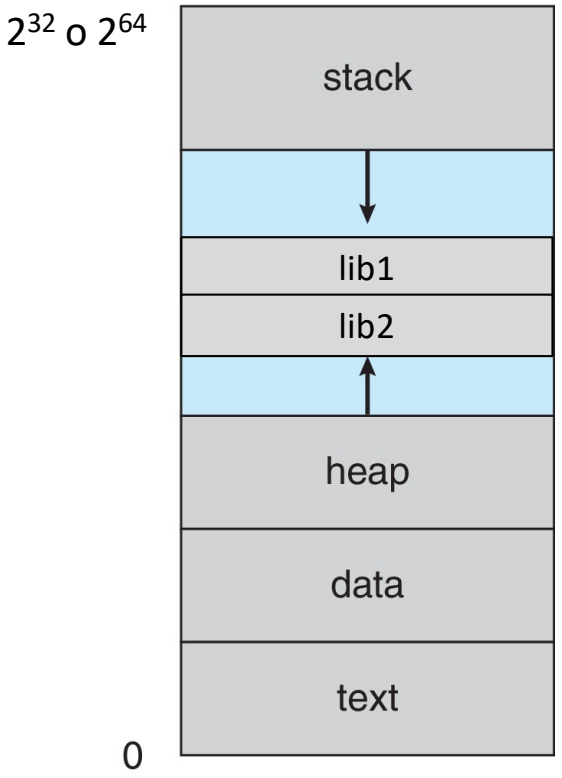
Memory mapping

- In generale, i sistemi operativi mettono a disposizione API per mappare una regione inutilizzabile del VAS su memoria centrale, così che diventi utilizzabile
- Esistono anche API che permettono di mappare una regione del VAS sul contenuto di un file (file mappati in memoria)
- In tal modo l'accesso al file può avvenire utilizzando le istruzioni macchina per accedere alla memoria, anziché le API del filesystem



Librerie dinamiche

- Le librerie dinamiche vengono caricate nella zona tra stack e heap
- Dal momento che possono essere caricate in qualsiasi posizione nello spazio di indirizzamento virtuale devono essere compilate come PIC



Le API POSIX per la gestione della memoria

Perché studiare le API per la gestione della memoria?

- Di norma non dobbiamo usare le API per gestire stack e heap:
  - Lo stack è gestito automaticamente dal sistema operativo
  - Lo heap è gestito di norma dal supporto runtime del linguaggio (new in C++) o dalla sua libreria (malloc in C), che invocano le API per ridurre / espandere lo heap in funzione delle necessità del processo
- Perché allora ci interessa sapere quali sono le API per la gestione della memoria?
  - Ci permettono di avere regioni di memoria con permessi particolari (sola lettura, eseguibili…)
  - Ci permettono di implementare componenti quali allocatori di memoria, compilatori just-in-time... qualora volessimo implementare il nostro nuovo linguaggio di programmazione
  - Ci permettono di utilizzare i file mappati in memoria e la memoria condivisa

API POSIX per la gestione della memoria

- Le API Unix legacy per cambiare la dimensione del segmento dati (che nello standard POSIX comprende le regioni data e heap) sono brk e sbrk
- Tali API sono deprecate in favore dell'API mmap, e incompatibili con questa (ma esistono ancora in diversi OS, ad esempio Linux)
- L'API mmap permette di mappare una regione ancora non utilizzata del VAS su:
  - Memoria centrale, oppure
  - Un file (che viene mappato in memoria), oppure
  - Memoria condivisa

mmap

```
void *mmap(void *addr, size_t length, int prot, int flags, int fd, off_t offset);
```

- *addr* è l'indirizzo virtuale a partire dal quale si vuole effettuare il mapping
- *length* è la lunghezza della regione di memoria da mappare. La memoria si può mappare solo per multipli della dimensione di 4k (4096 bytes) su processori x86.
- *prot* indica quali permessi ha quella regione di memoria, può assumere i valori PROT_READ, PROT_WRITE, PROT_EXEC, or PROT_NONE
- *flags* indica opzioni sul mapping. MAP_FIXED ad esempio indica che la memoria deve essere allocata a partire esattamente dall'indirizzo passato nel parametro addr oppure la chiamata deve fallire
- *fd* è un descrittore di un file o di memoria condivisa; se si intende mappare memoria ordinaria occorre passare MAP_ANONYMOUS tra i flags (estensione non POSIX)
- *offset* viene usato in combinazione con fd per indicare la porzione del file che intendiamo mappare
- `mmap` restituisce l'indirizzo della memoria mappata, oppure la costante MAP_FAILED se la chiamata fallisce

Esempi d'uso mmap e msync

- Per mappare 4 KB di memoria a partire dall'indirizzo virtuale 0xa0000000:

```
void *ptr = mmap(0xa0000000, 4096, PROT_READ|PROT_WRITE,
MAP_ANONYMOUS, 0, 0);
```

- Per mappare, a partire dall'indirizzo virtuale 0xb0000000, 8192 bytes del file /usr/foo a partire dal byte 100:

```
int fd = open("/usr/foo", O_RDWR);
void *ptr = mmap(0xb0000000, 8192, PROT_READ|PROT_WRITE,
MAP_PRIVATE, fd, 100);
```

- Per sincronizzare le modifiche in memoria con il file:

```
int ok = msync(0xb0000000, 8192, MS_SYNC|MS_INVALIDATE);
```