tmp

u6-s1-memoria

Sistemi Operativi

Unità 6: La memoria

# Organizzazione delle memoria

Martino Trevisan
Università di Trieste
Dipartimento di Ingegneria e Architettura

---

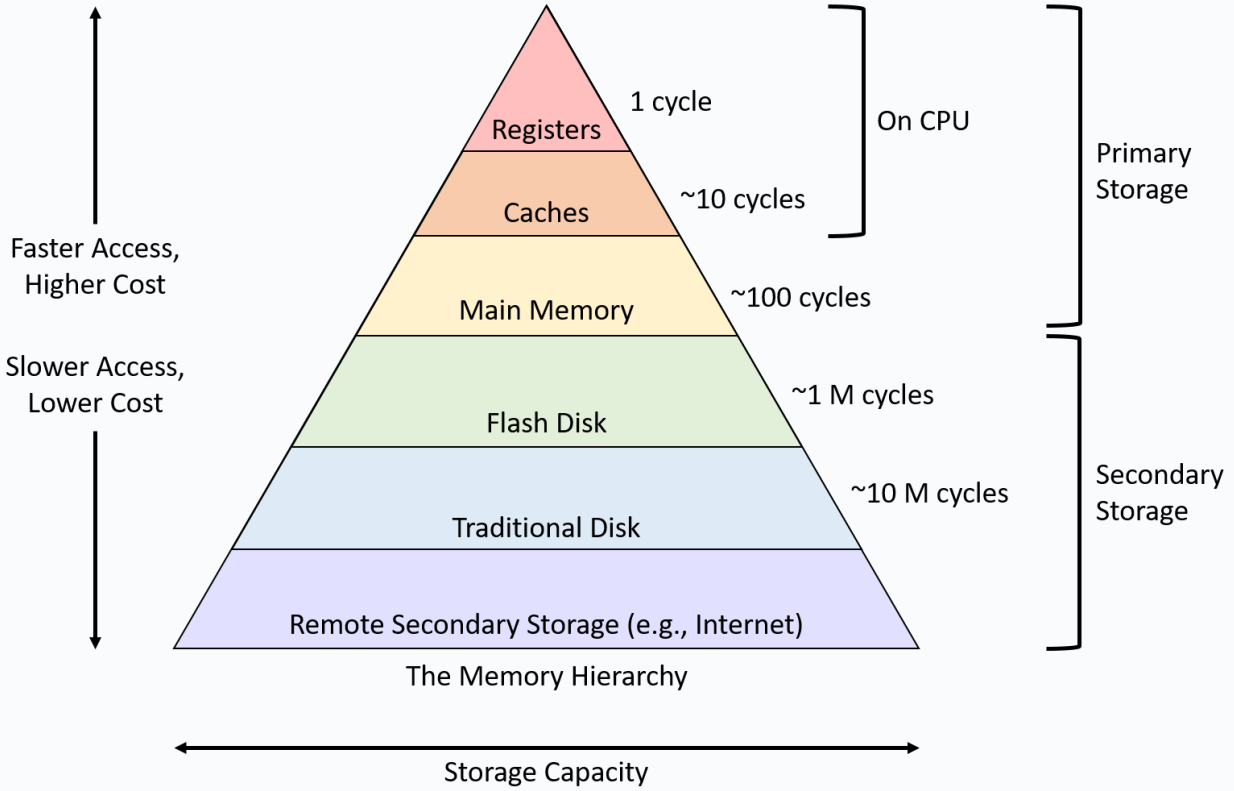
## Argomenti

1. Memoria nei sistemi ad processore
2. Approcci storici
3. La memoria virtuale
4. Rimpiazzamento delle pagine
5. Layout della memoria
6. Loader, librerie e pagine condivise
7. Gestione della memoria in Bash

---

# Richiamo alla Memoria dei Sistemi a Processore

I sistemi ad processore possiedono molte *memorie*




Il compito del SO è gestire l'utilizzo della memoria da parte dei processi, con gli obbiettivi di:

- *Massimizzazione delle prestazioni*: usare la memoria più veloce possibile
- *Isolamento tra processi*: evitare problemi di sicurezza e stabilità
- *Facilita per il programmatore*: si vorrebbe che il SO fosse *trasparente* per chi programma

---

# Approcci Storici per la Memoria

## Pre-S.O.

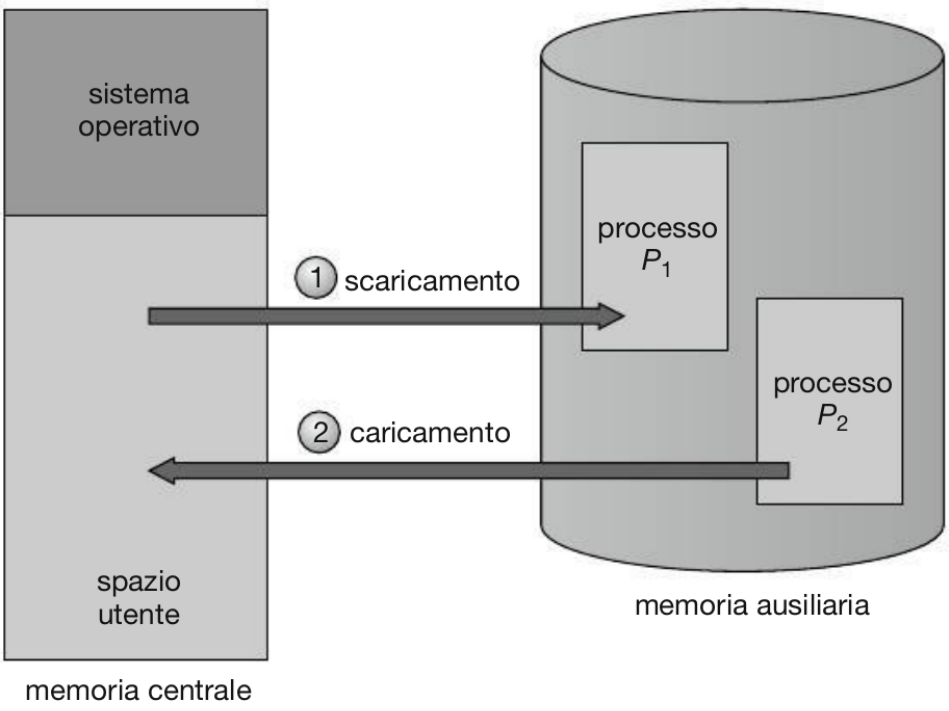
Inizialmente, non vi era SO: l'elaboratore eseguiva un programma per volta

- Un programma poteva accedere a qualsiasi locazione di memoria
  *Nota:* ancora non esisteva la memoria virtuale e la MMU, ecc...

## Sostituzione Totale

Un SO che sostituisce completamente la memoria principale del programma in esecuzione ad ogni *Context Switching*

- L'approccio più *fallimentare* e *lento*: ho troppo tempo sprecato!

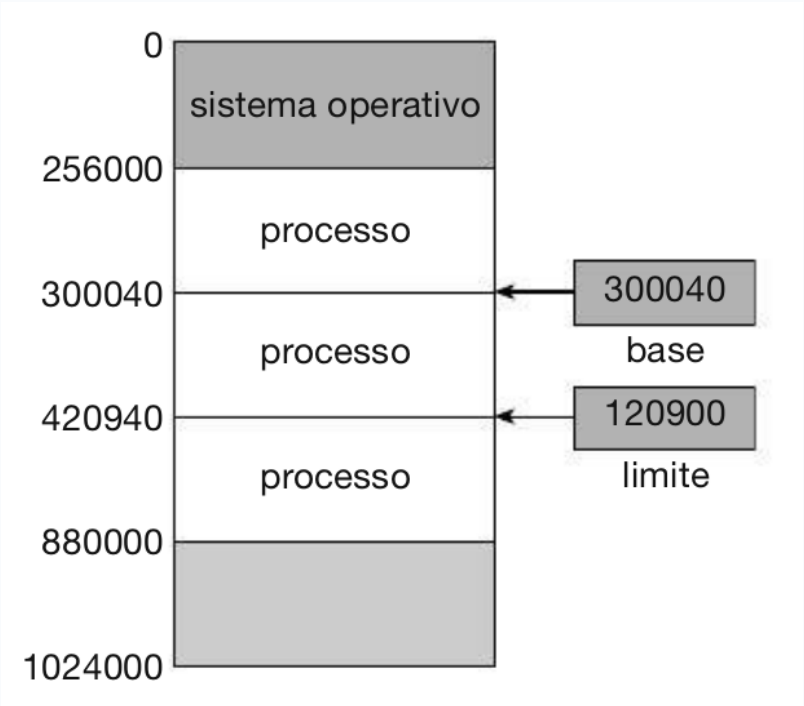



## Approccio Base e Limit

Introdotta negli *anni '70-'80*

**IDEA.**

- Più processi condividono la memoria
- Hanno il permesso di accedere a una sola zona di memoria
- Le *"fette"* sono identificate da due numeri
  - Ogni processo ha una sua *"fetta"*




Molto importante, ebbe un grande successo

- Nasce il concetto di *Indirizzo Logico o Virtuale*

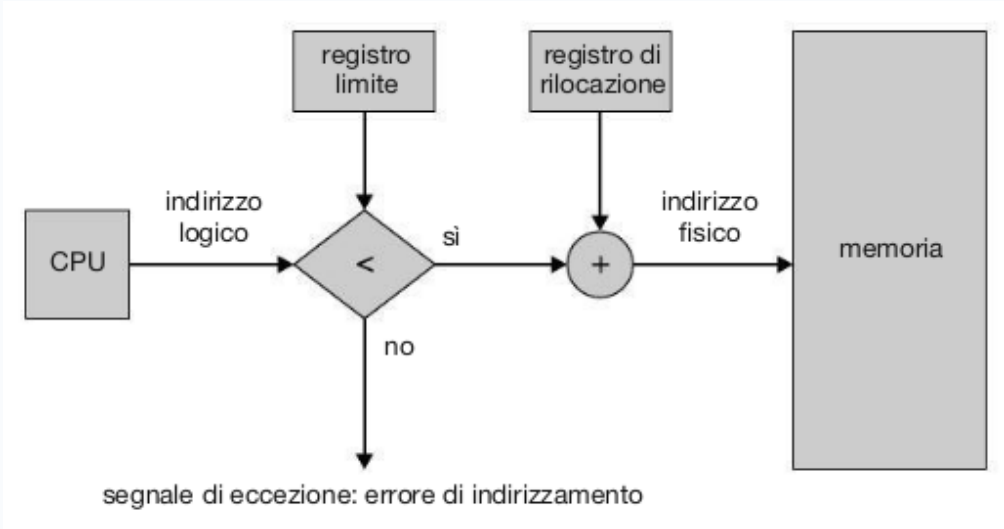
    - Utile per la *sicurezza* della memoria
- La CPU ha i registri *Base* e *Limit*, settati dal SO
- Essa permette ai processi di emettere solo indirizzi consentiti

**Pro:**

- Permette di avere *più processi*
- *Sicuro*: un processo non può accedere a memoria di altri

**Contro:**

- Allocazione *contigua*: si rischia spreco di memoria
    - Rischio la *frammentazione esterna*
- Poca flessibilità




## Approccio a memoria segmentata

Come Base + Limit, ma ogni processo ha a disposizione più *segmenti*
Solitamente, ogni segmento ha scopi diversi:

- Segmento di *Codice*
- Segmento di Dati (Variabili Globali e Costanti)
- Segmento di Stack (Variabili di funzioni)
  Sostanzialmente questa è una *generalizzazione* dell'approccio a memoria segmentata, migliorandola (quasi!) in ogni aspetto

**Pro:**

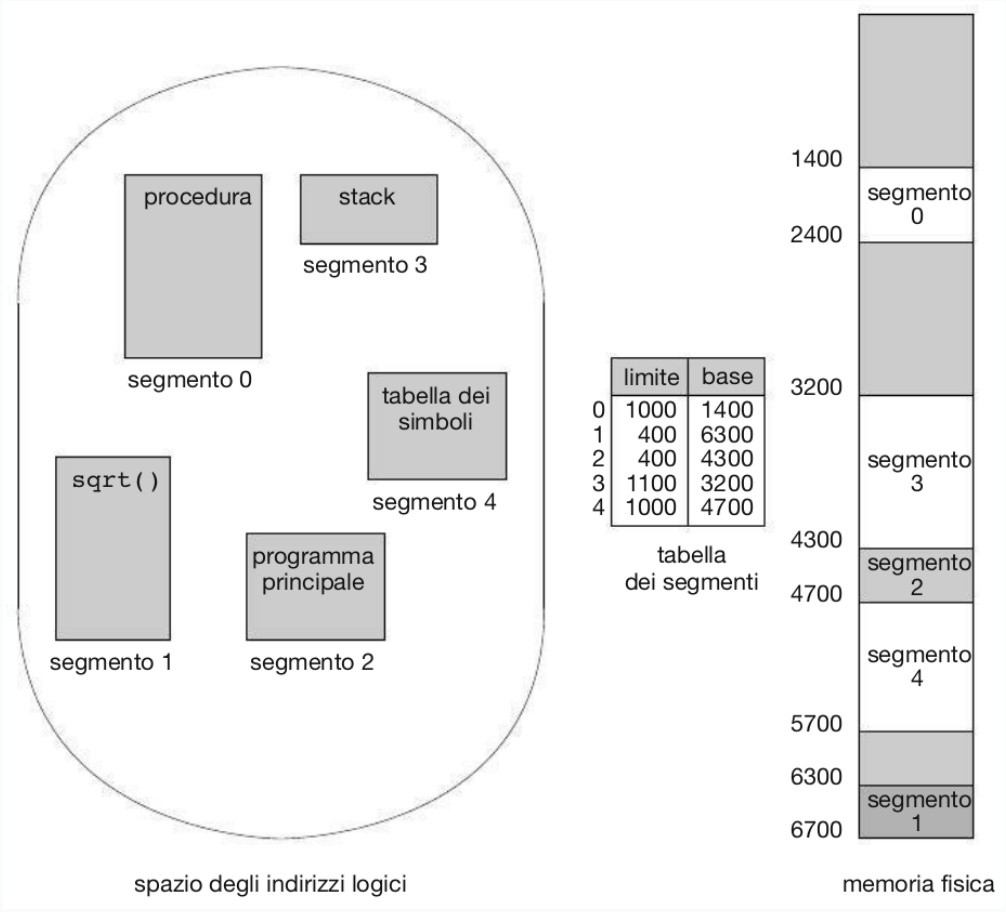
- Abbastanza *flessibile*
- *Semplice* da implementare
- Veloce

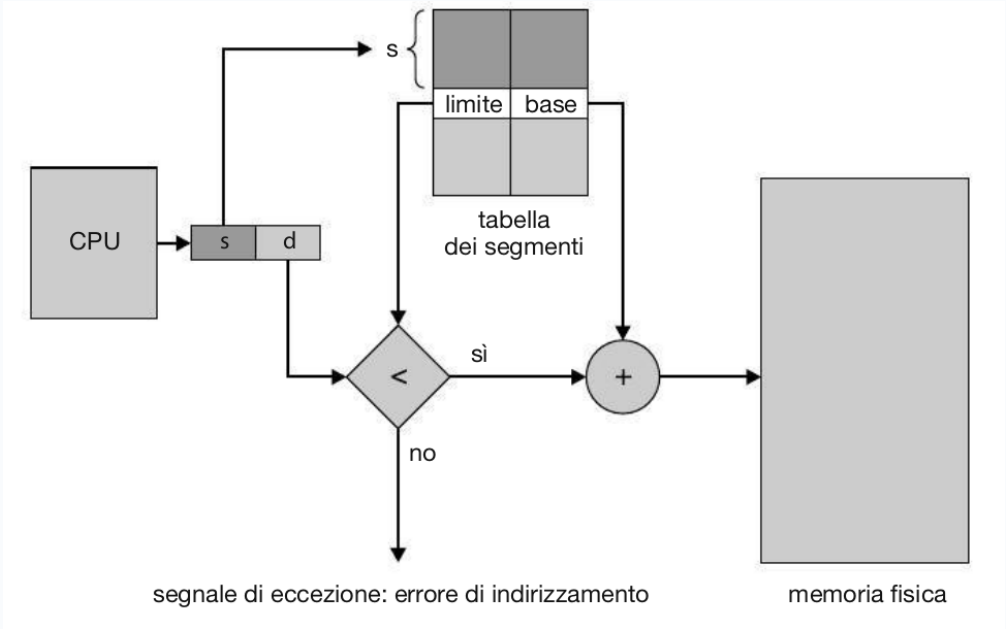
**Contro:**

- Segmenti di lunghezza diversa sono problematici da gestire

- Introducono *frammentazione* come in *Base + Limit*
  - Il limite principale non è stato superato

E' stato comunque molto usato negli anni '80 e '90




## Approccio a paginazione

Vediamo l'approccio moderno.

**IDEA.**
Un processo emette *Indirizzi Virtuali*

- Un modulo hardware detto *Memory Management Unit li traduce in Indirizzi Fisici*

Lo spazio degli indirizzi virtuali è diviso in blocchi di lunghezza fissa dette *pagine*

- Una tabella mappa la posizione delle pagine dallo spazio virtuale a quello fisico

In generale abbiamo che:

- Il processo vede uno *spazio virtuale* che è diviso in *pagine di grandezza dimensione fissa*
- La *tabella delle pagine* indica come sono state allocate nella memoria fisica
- La *MMU* effettua la traduzione
- Il *SO* imposta/programma la MMU
- Tipicamente lo spazio degli indirizzi virtuali è più grande di quello degli indirizzi fisici
    - *Esempio.* Il programma può emettere indirizzi su 1024 pagine, ma in memoria ce ne stanno solo 80. Se non basta, vedremo cosa succede; per ora si *"spera"* che tutto basti
- Il programmatore non deve sapere quanta memoria ha il sistema; gestisce tutto il sistema

**Pro:**

- No frammentazione (ho *pagine* con dimensioni fisse)
- Flessibile
- E' lo standard *de-facto* *
- Utilizzato in tutti i moderni processori e SO

**Contro:**

- Richiede un Hardware veloce, che assiste tutto il processo (in particolare deve avere anche la *MMU*); tuttavia è l'*unico* difetto superabile.

**Figura:** (*Livello circuitale*)

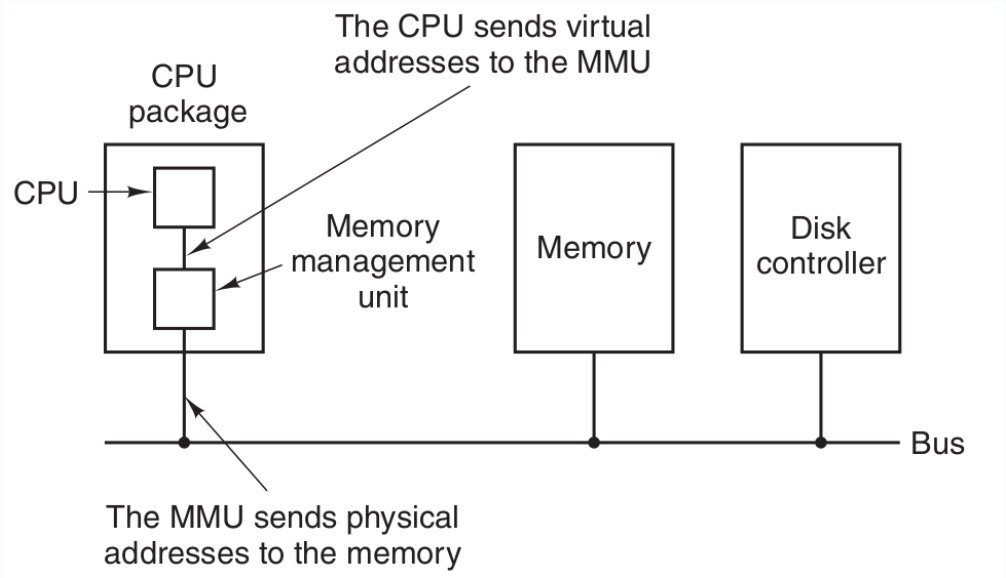
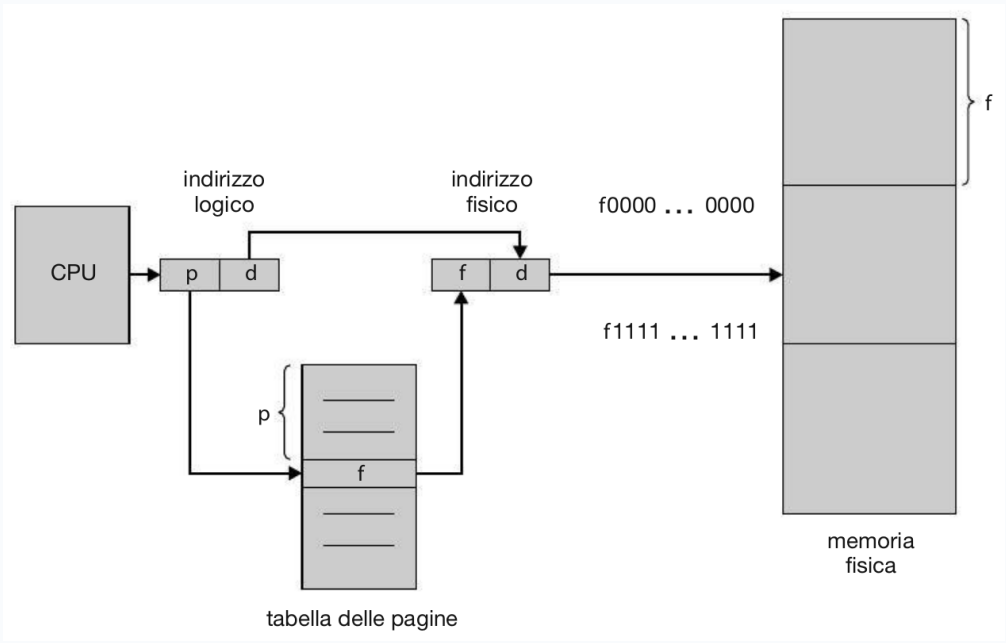
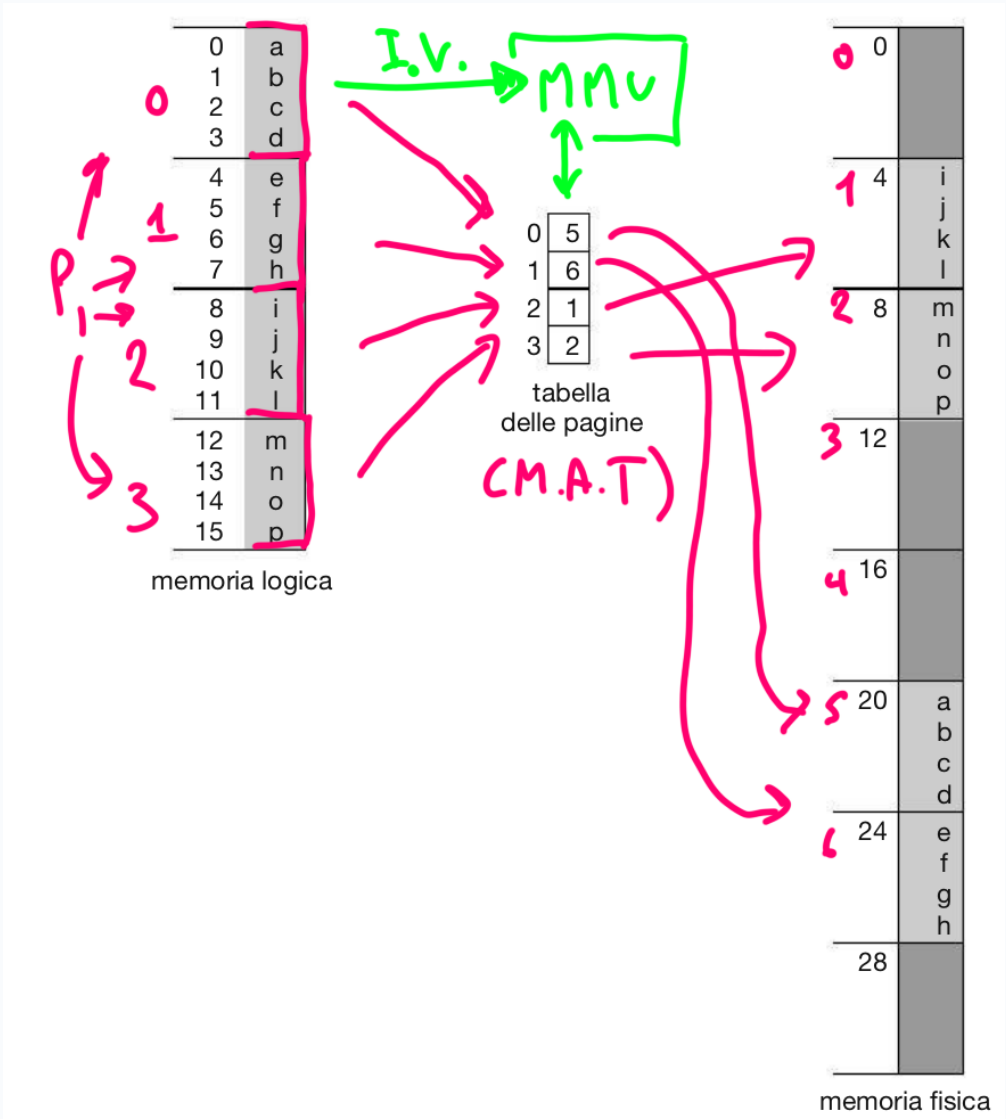
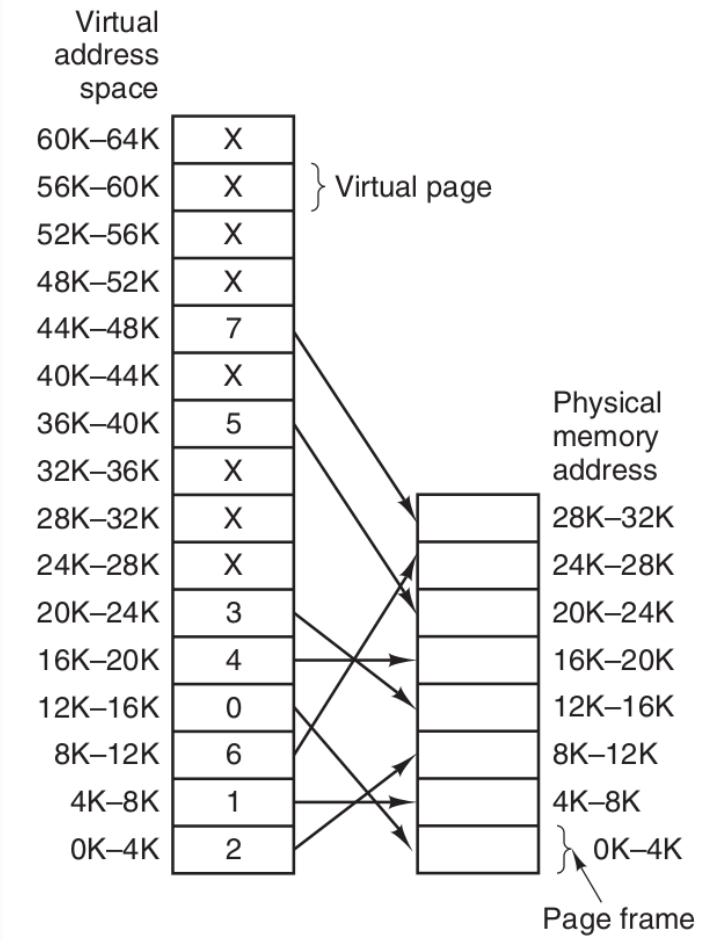


**Figura:** (*Livello logico*)






---

# La memoria virtuale

E' il naturale effetto *della memoria paginata*

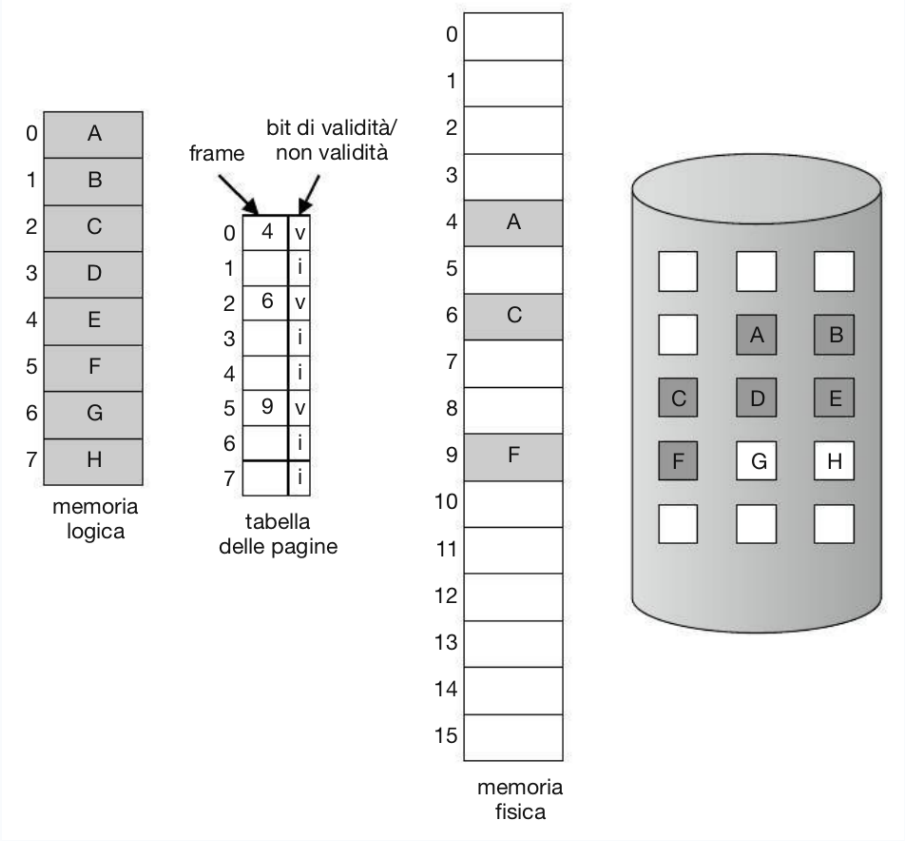
- Il processo vede uno spazio di *indirizzi virtuale*, *mappato sulla memoria fisica*

**Architettura x86-64**:

- Pagine da 4KB
  - *Offset da 12 bit* $(\log_2(4000) \approx 12)$
- Indirizzi Virtuali su 64bit, ma solo 48 significativi utilizzati
  - Memoria virtuale di *256TB*
- La MMU traduce ad indirizzi fisici di 48bit
  - Ma la memoria fisica è sempre *molto* più piccola; di solito abbiamo fino a 32GB per un computer personale.

## Swap

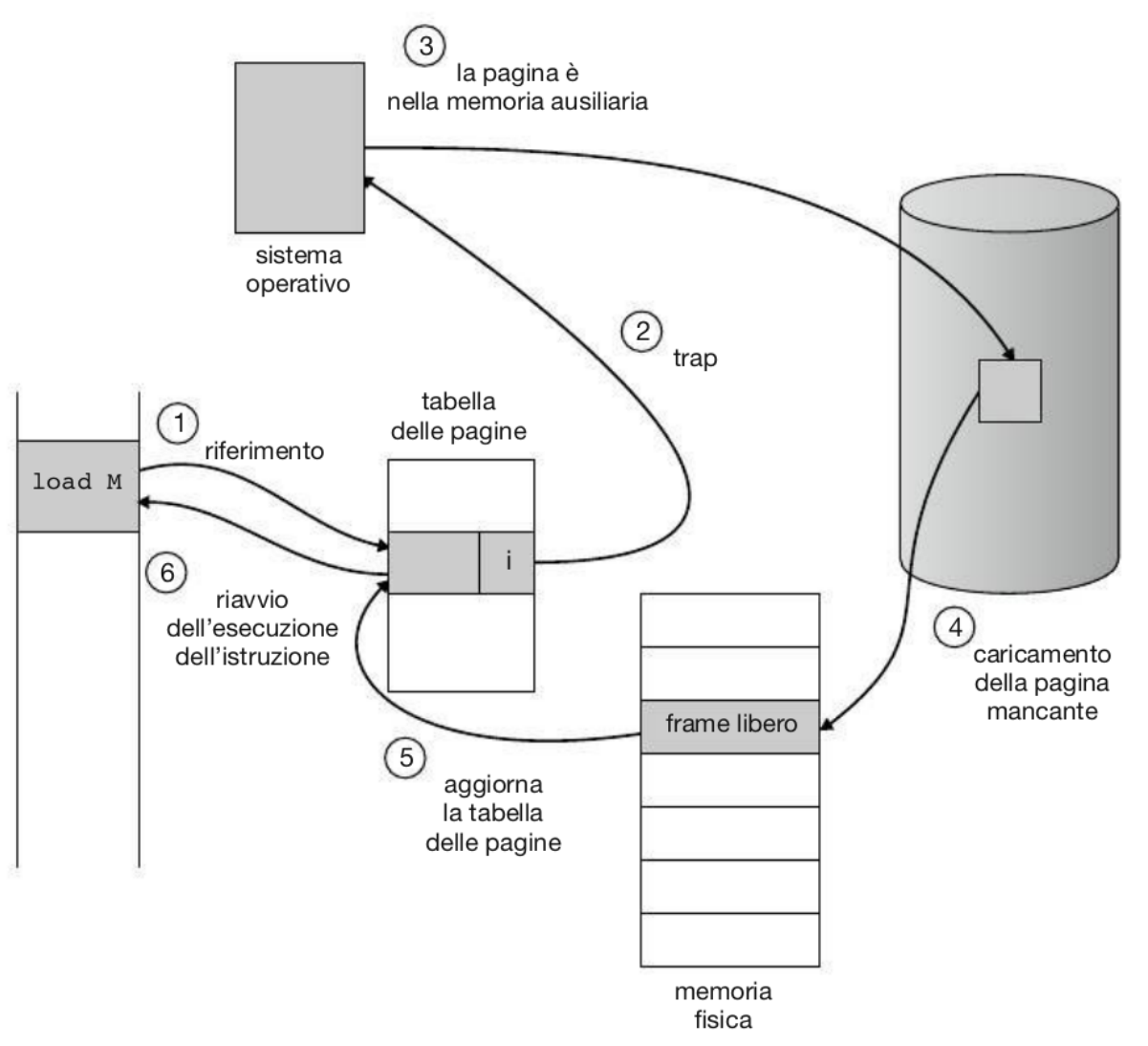
Se le pagine non stanno tutte in memoria, si mettono su disco.
Lo spazio su disco che contiene le pagine non in memoria si chiama *Swap*




## Page Fault

Se il processo *emette un indirizzo di una pagina che non è in memoria*, si verifica un *Page Fault*:

1. La MMU avverte il SO
2. Il SO *interrompe il processo*
3. Il SO *carica* la pagina (o la crea) da disco
4. Il SO *reimposta* la MMU
5. Il processo *riprende*
   Questo è un processo molto *lento*, quindi da *minimizzare*!




---

# Rimpiazzamento delle pagine

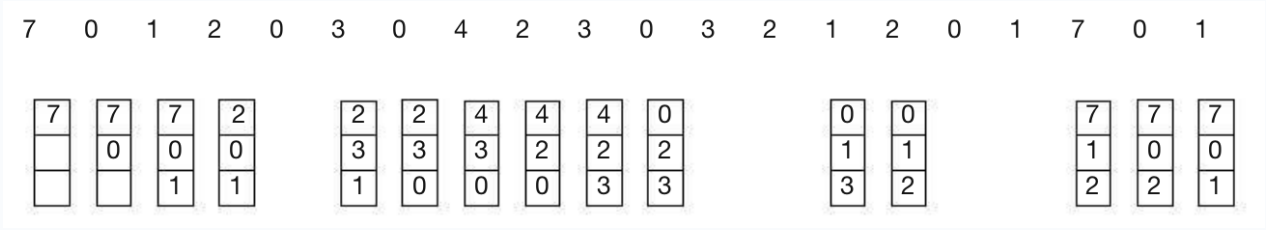
La memoria fisica è *sempre più piccola* di quella virtuale

Se è piena di pagine utilizzate da processi attivi, il SO deve *scegliere* quale pagine eliminare e salvare su disco

Esistono diversi *algoritmi di rimpiazzamento* per effettuare ciò in maniera furba

## Algoritmo FIFO

Rimuovo dalla memoria *la pagina caricata da più tempo*

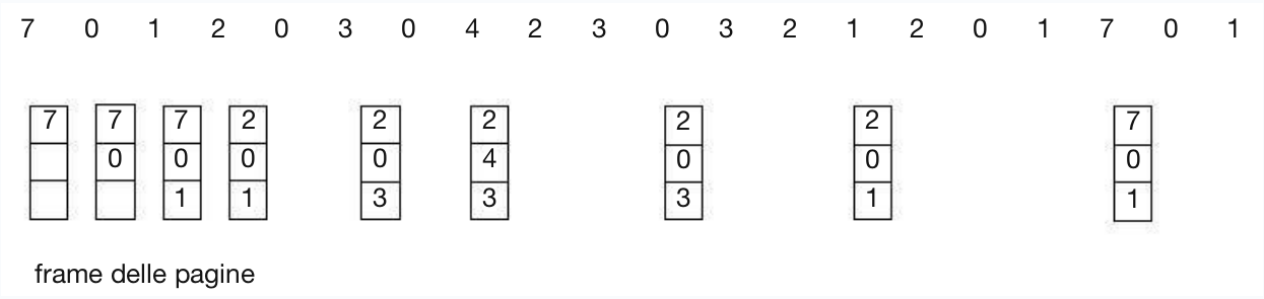



- Semplice, ma inefficace

- *Ragionamento analogico:* A casa non ho abbastanza spazio per mettere nuove cose; quindi prendo la cosa più *vecchia*, come il *divano* e la metto in cantinato.

## Algoritmo Ottimo

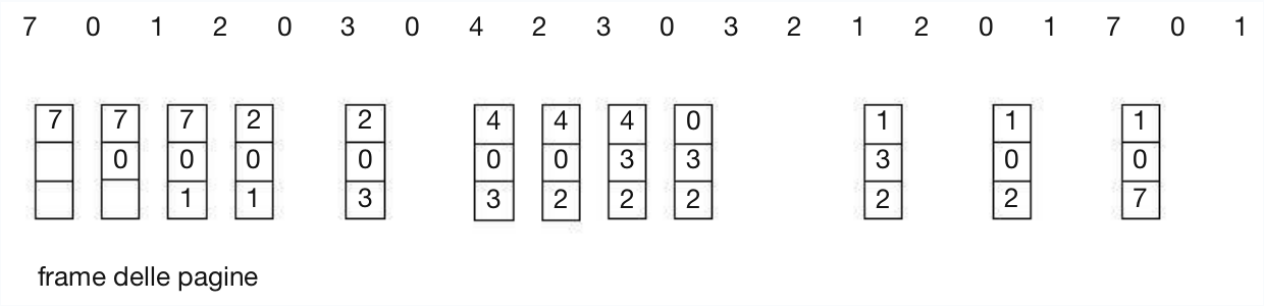
Rimuovo la pagina che non mi servirà per più tempo *nel futuro*




- Ottimo, ma *impossibile prevedere il futuro* (purtroppo)
  - *Ragionamento analogico:* A maggio tolgo gli scii, a dicembre tolgo la tavola da surf. Purtroppo impossibile da *"automatizzare"* ragionamenti del genere in una maniera rigorosa

## Algoritmo Least Recently Used

Rimuovo la pagina che *non viene usata da più tempo*




- Semplice, efficace.
- Serve collaborazione della *MMU* per tenere traccia di accessi
- Usato quasi sempre (con varianti)
- Non si fa differenza tra processi: le pagine sono uguali

---

# Layout della memoria

Un processo può accedere a *qualsiasi locazione di memoria nello spazio degli indirizzi virtuali*

- Lo spazio degli indirizzi virtuali è *diviso in pagine*
- Se la pagina è in *memoria*, la MMU traduce in *indirizzo fisico*
- Se la pagina *non è in memoria*, il SO la creerà/preleverà *da disco*

## Organizzazione della Memoria

Abbiamo molti modi per *organizzare* l'accesso agli indirizzi virtuali.

Un programma che accede a indirizzi "casuali" non è efficiente

- Utilizzo di pagine e memoria sarebbe *molto penalizzato*

Storicamente si cominciavano a usare indirizzi a partire *da quelli "bassi"*:

- Si inizia a utilizzare indirizzo `00 00 00 00`, poi `00 00 00 01`
- Così si riempie una pagina completamente, poi se inizia a usare un'altra

Ci sono diverse convenzioni, che dipendono da architettura dell'elaboratore e OS. Noi vediamo quella per *Linux*

## Convenzione Linux

Attualmente, si usano *sia indirizzi all'estremo alto che all'estremo basso*

- La memoria può crescere in due direzioni
- Posso avere due zone di memoria che crescono a seconda dell'esigenza del programmatore

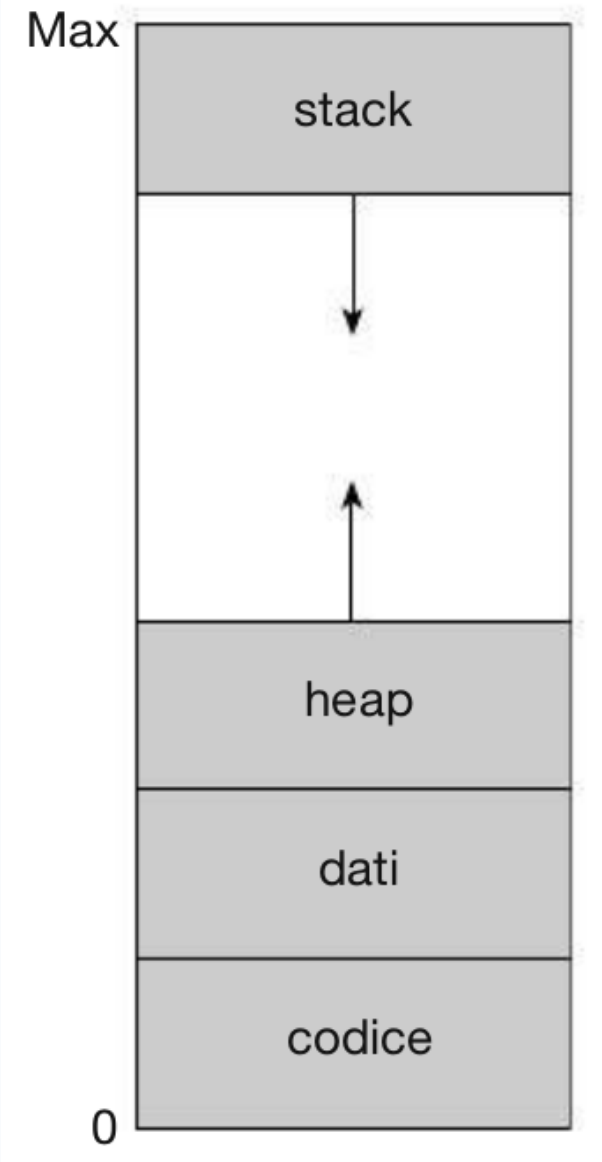
Principalmente abbiamo *direzioni*:

- **Heap** $(\xi)$: cresce dal basso verso l'alto. Usato dal programmatore per allocare memoria quando gli serve
- **Stack** $(\sigma - \rho)$: cresce dall'alto verso il basso. Usato dal compilatore per posizionare le variabili delle funzioni

Un processo può accedere a *qualsiasi locazione di memoria*.
Per convenzione e prestazioni si preferisce iniziare gli *estremi*

Ci sono 4 *zone di memoria*:

- Codice $(\zeta)$ - *Dimensione fissata*
- Dati $(\delta)$- *Dimensione fissata*
- Stack $(\sigma - \rho)$ - *Dimensione variabile dall'alto al basso*
- Heap $(\xi)$ - *Dimensione variabile dal basso all'alto*




Adesso andiamo ad approfondire le parti varie.

## Codice $\zeta$

Il SO copia il codice del programma dal disco *verso gli indirizzi più bassi*

- Il codice deve obbligatoriamente trovarsi *in memoria*
- Il registro *Program Counter* della CPU punta a un indirizzo in questo range

Questa parte della memoria è *Read Only* (imposta dalla *CPU*): un programma non può modificare se stesso (senno ci sarebbero casini assurdi)

## Dati $\delta$

Gli indirizzi immediatamente *maggiori* del codice, sono usati per le *variabili globali*

- Il compilatore usa questi indirizzi per le variabili globali
- Le variabili globali *inizializzate* vengono riempite direttamente dal SO quando viene avviato il processo
    - Per questo quando inizializziamo gli *array*, devono avere *dimensione nota*.

- Le altre contengono tutti `'\0'` (ovvero hanno valore $0$)

## Heap $\xi$

Usato per la *Memoria Dinamica*

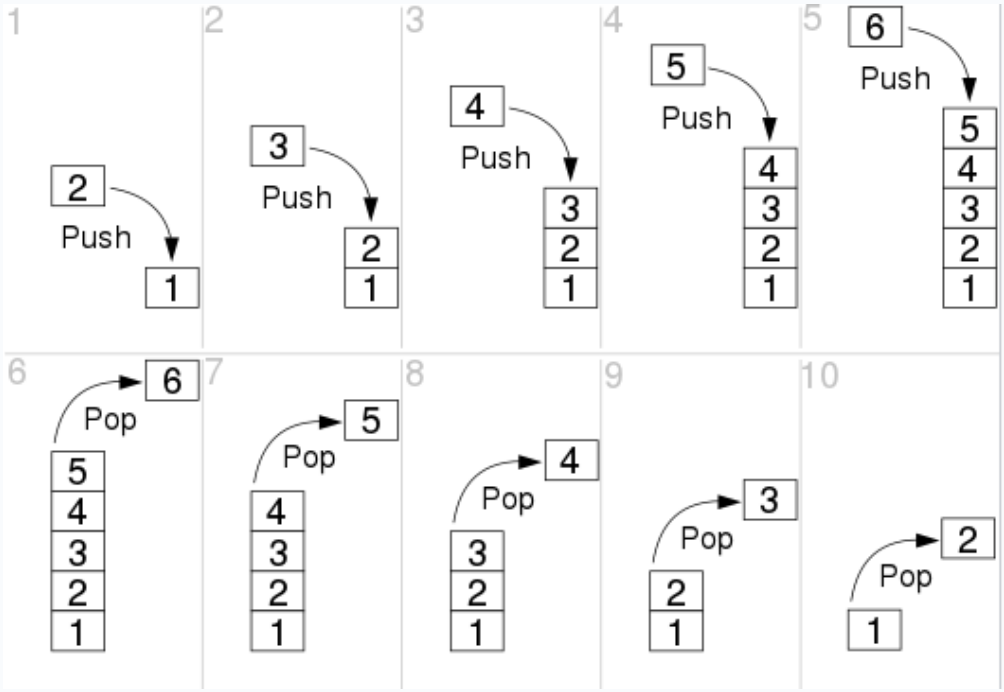
- Il programmatore può aver bisogno di *memoria la cui dimensione non viene prevista* in fase di programmazione
- Gestita tramite le funzioni di libreria *malloc* e *free*
- Vedremo *in seguito*

## Stack $\sigma - \rho$

Usato per le variabili relative a funzioni: argomenti e variabili interne

- Come dice, questa zona è gestita come se fosse una *pila*
  I dati vengono:
- Impilati per essere aggiunti: *Push* (aggiungo stack)
- Tolti dalla pila quando devono essere usati: *Pop* (rimuovo uno stack)
  Si comportano come una *specie di molla*, che va su e giù

I dati vengono aggiunti e tolti dalla cima della pila come nella seguente figura:




- Abbiamo una struttura del tipo *last in, first out* (LIFO)

## Operazioni di Push e Pop

Concetto pratico per gestire le *funzioni*!
Quando viene chiamata una funzione, si aggiunge un blocco allo stack contente (*Push*):

- Indirizzo di *ritorno*
- *Parametri* copiati (passaggio per valore)

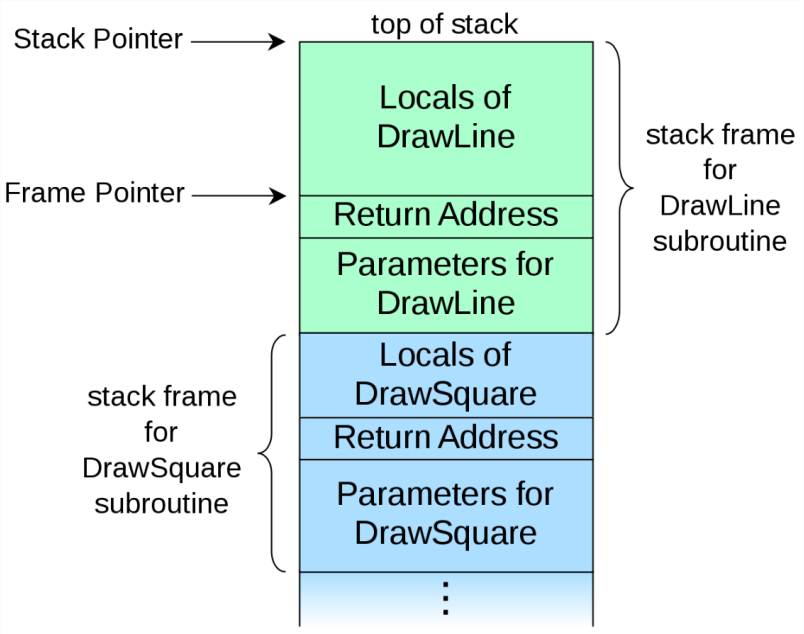
- *Variabili locali* (si cerca il più *vicino*, preferibilmente nello stack stesso)

Quando la funzione ritorna, il blocco si elimina (*Pop*)

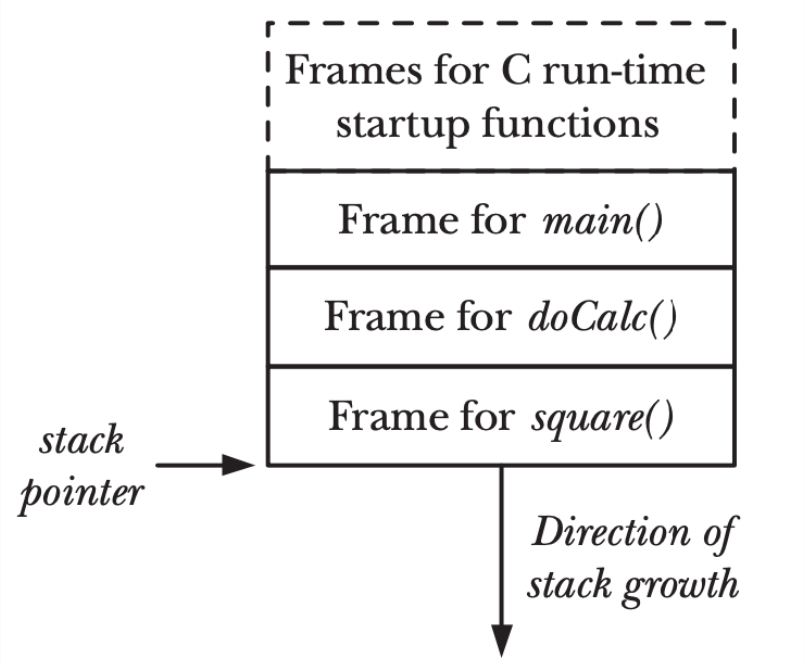
Ogni blocco si chiama *Stack Frame*

- Creato quando la funzione viene invocata
- Cancellato quando la funzione ritorna

**Stack**




**Stack Frame**




---

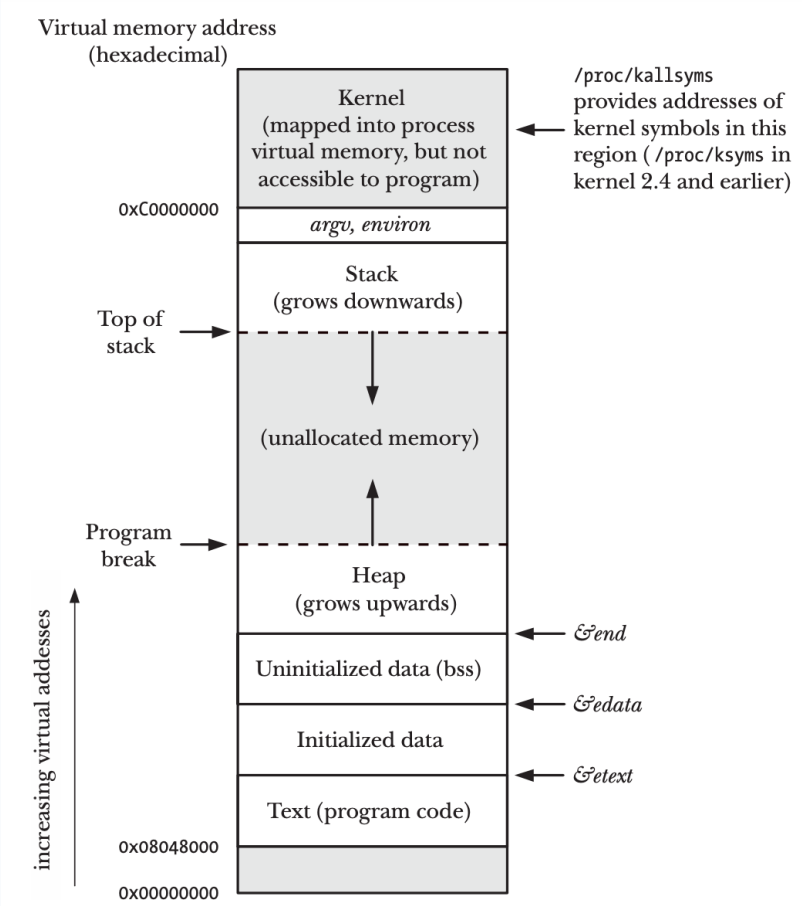
## Stack Frame in Linux

Il layout visto prima é *generico*.

In Linux, precisiamo che ci sono altri costrutti.

Per *data* $\delta$:

- Un segmento per *variabili globali inizializzate* e uno per *quelle non inizializzate*
  Per *stack* $\rho - \sigma$:
- `argc` e `argv` in indirizzi alti
- *Top of Stack*: indirizzo minimo per parte alta
  Per *heap* $\xi$:
- *Program Break*: indirizzo massimo per parte bassa




---

# Loader, librerie e pagine condivise della Memoria

## Loader

Il *Loader* è il componente del SO che *avvia i processi*.
I suoi compiti sono:

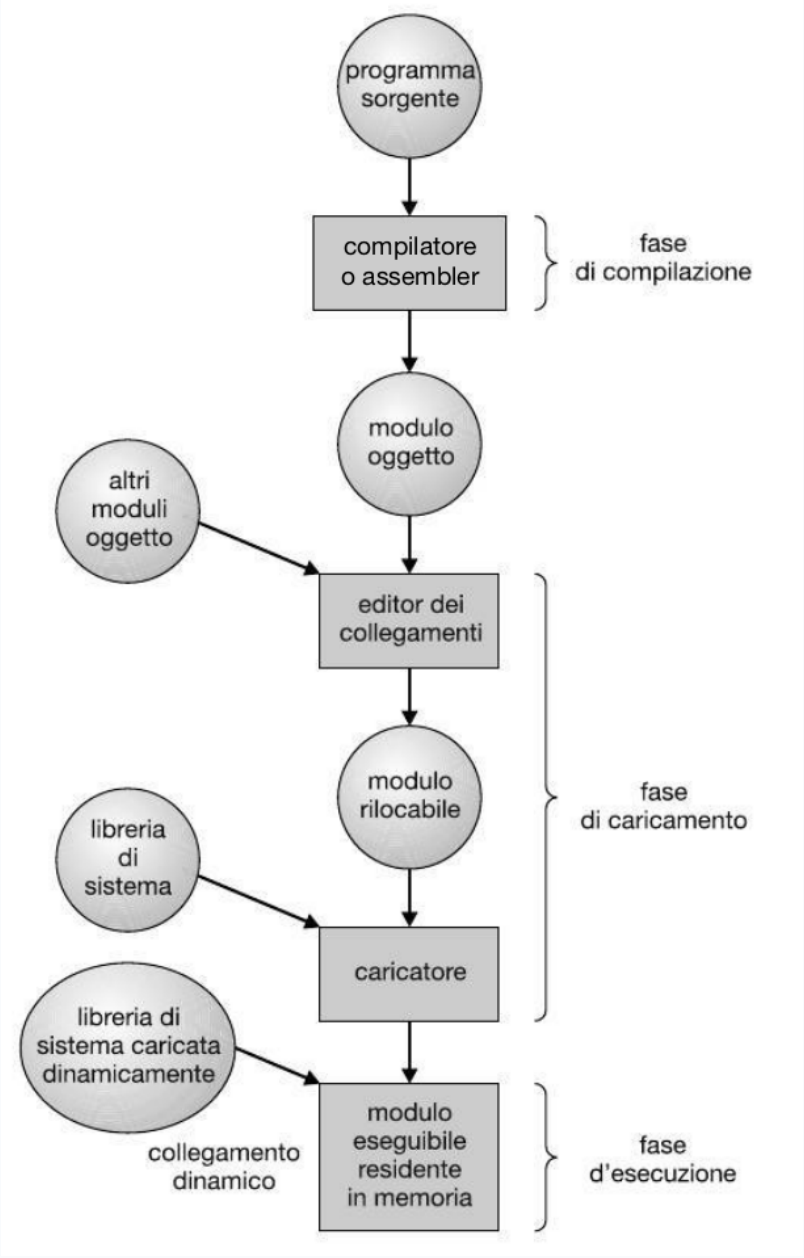
- Verificare che l'utente abbia i *permessi*
- Copia il codice del programma e le variabili globali inizializzate *in memoria* ($\delta$)

- Caricare le *librerie condivise*
- Valorizzare **argc** e **argv** (copiate in alto)
- Avviare il processo dal **main** impostando lo *scheduler* del SO

Il compilatore crea il codice macchina

Il loader lo *carica* e ne *avvia* l'esecuzione

- Deve caricare *anche le librerie di sistema*, se sono usate




## Librerie Condivise

I programmi possono usare *librerie condivise*:

- Offerte *dal SO* per facilitare la chiamata a System Call
- Installate da utente per scopi particolari (e.g., trigonometria)

Le librerie sono codice che gira in *User Mode*

- Non hanno alcun privilegio rispetto al codice utente
- *Non* sono parte del Kernel

Le *librerie condivise* sono *codice eseguibile* in cartelle predefinite del sistema

In **Linux**:

- `/lib`
- `/usr/lib`
- Directory elencate nel file `/etc/ld.so.conf`

In **Windows**:

- `C:\Windows\SYSTEM32`
- Cartella corrente

## Formato ELF Linux

Gli eseguibili in Linux sono in formato ELF (*Executable and Linking Format*)

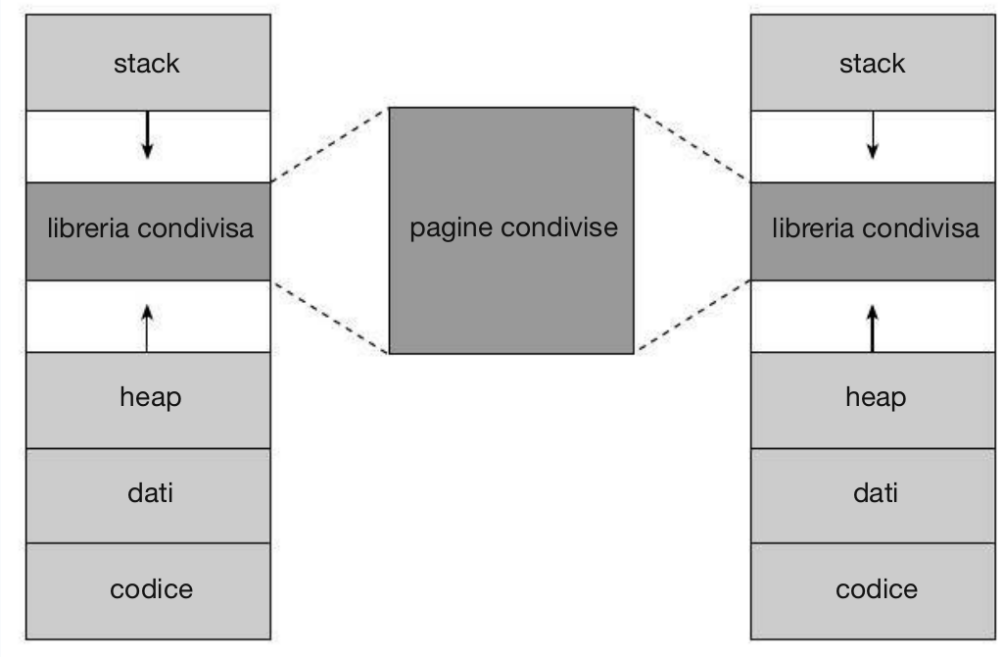
- Oltre il codice, contengono *la lista delle librerie di sistema* che useranno
- Contenute in una posizione predefinita

Le librerie e condivise sono identificate dal nome e dalla versione

---

## Locazione delle Librerie Condivise

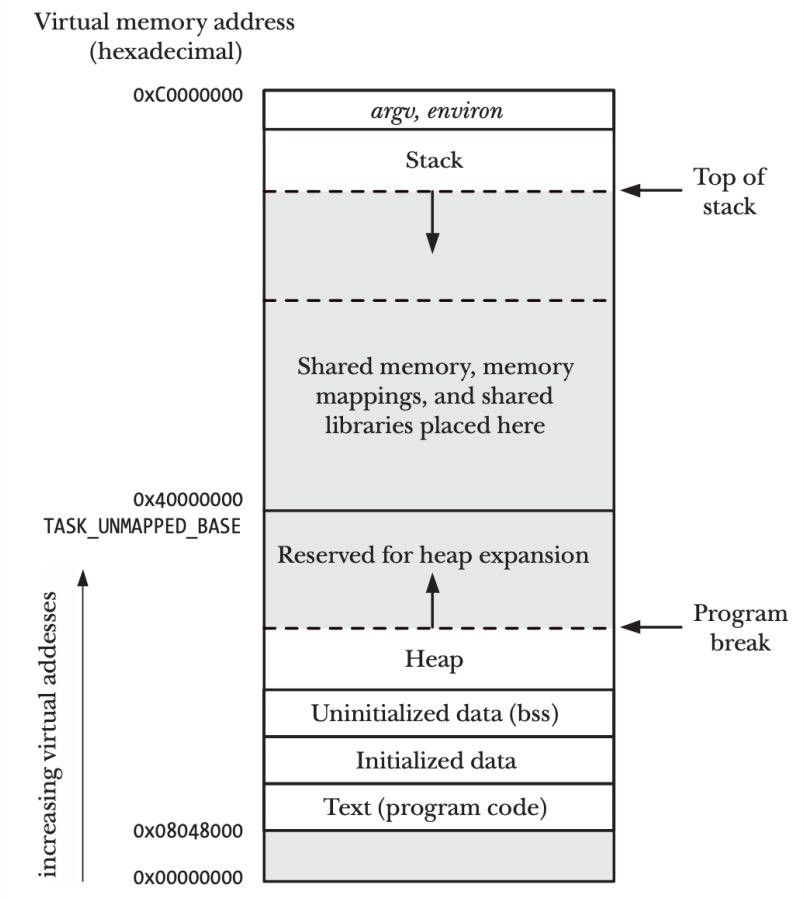
Le librerie sono caricate *dal Loader* in indirizzi intermedi
Se più processi usano la *stessa libreria*, la *pagina viene condivisa* (ovvero gli indirizzi virtuali vengono tradotti negli stessi indirizzi fisici)



# Memoria Condivisa

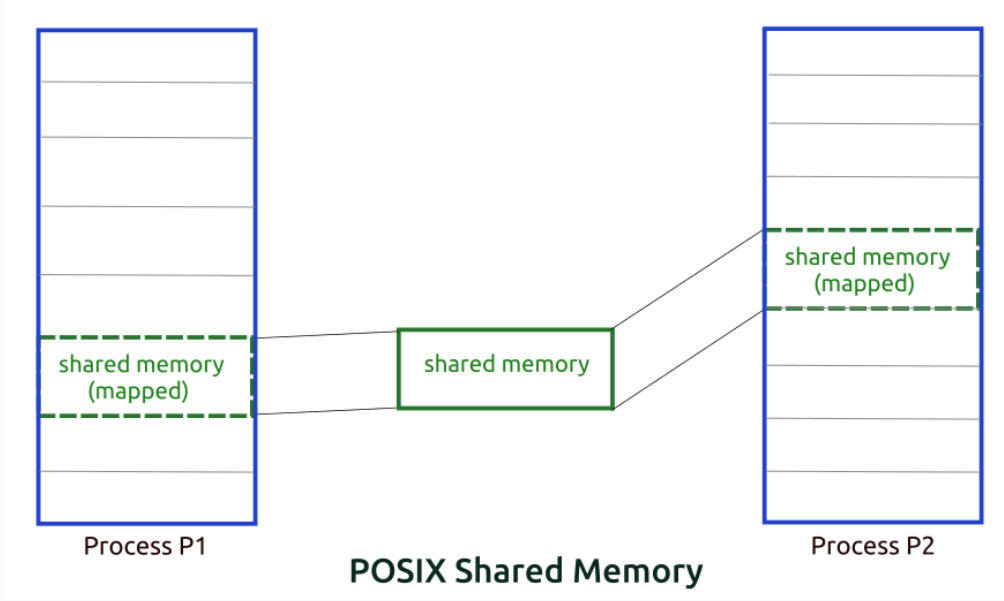
La memoria condivisa tra processi funziona *nello stesso modo*

- Gli *indirizzi intermedi,* tra heap e stack, sono usati per tutto ciò che é condiviso
- Esempi: *memory mapping*, eccetera...



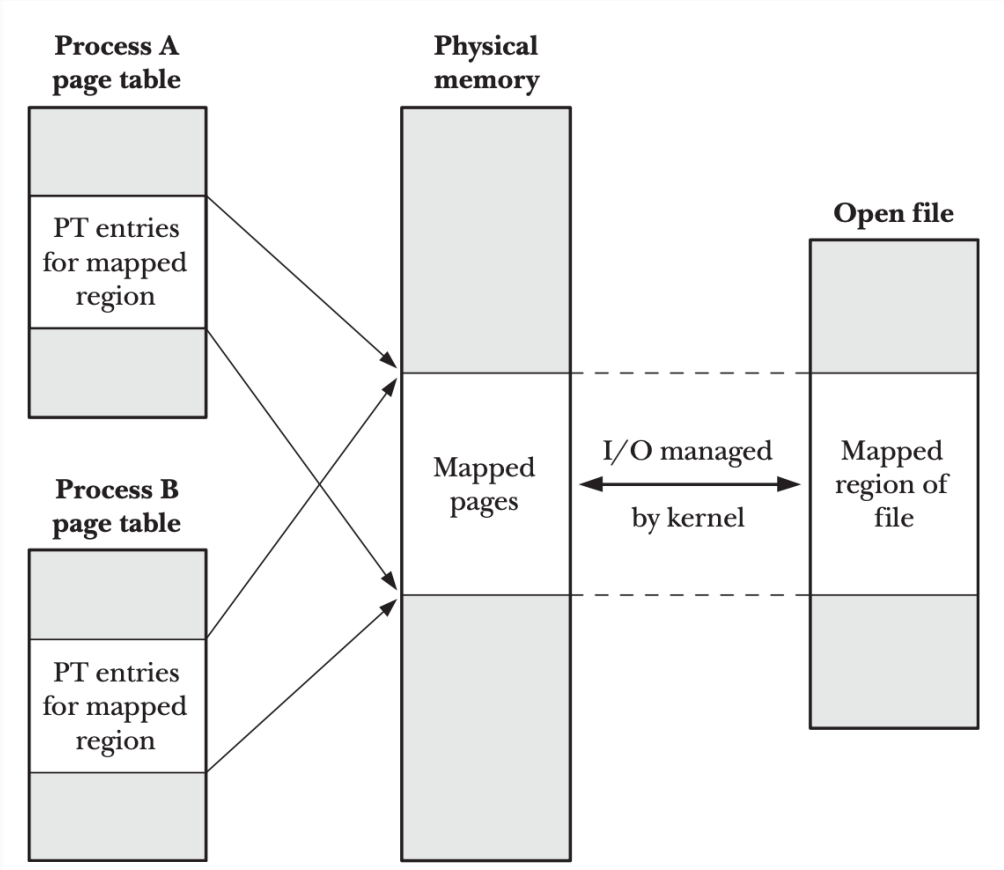

Tutto ciò è possibile *grazie alla MMU*

- Il SO imposta la MMU per *implementare questo schema*
  - Due indirizzi virtuali vengono *tradotti* per puntare alla medesima zona di memoria condivisa



## Esempio di Memoria Condivisa

In caso di **mmap** con *persistenza su file*, il kernel si occupa di allineare la zona di memoria condivisa *su disco*




---

# Gestione della memoria in Bash

Vediamo dei comandi per *gestire la memoria* in Bash.

**Lista.** (*Comandi Bash*)

- **free**: mostra quanta memoria è disponibile/utilizzata/libera nella macchina
- **top**: mostra varie informazioni sui processi
  - Colonna **RES**: *Resident Set Size* quante *pagine* dello spazio virtuale di un processo sono *caricate in memoria fisica*
  - Colonna **VIRT**: *Virtual Set Size* quante *pagine sono state usate* dal processo nella sua storia
  - Colonna **%MEM**: **RES/totale**, ovvero *percentuale di memoria fisica* della macchina contente pagine del processo
- **cat /proc/meminfo**: mostra informazioni dettagliate su memoria della macchina (questa è la fonte del comando **free**
- **ldd eseguibile**: mostra quali librerie condivise esso richiede
- **objdump -p eseguibile**: dissector del formato ELF (*mostra* i campi dell'ELF)

## Domande

Il compito della Memory Management Unit è:
**• Gestire il funzionamento della cache**
**• Allocare zone di memoria**
**• Tradurre gli indirizzi da virtuali a fisici**
**Risposta:** *Tradurre gli indirizzi da virtuali a fisici*

Cosa fa la MMU quando una pagina non è in memoria?
**• La carica**
**• Termina il processo che ha generato l'indirizzo**
**• Avverte il SO**
**Risposta:** *Avverte il SO* (mediante una Page Fault)

La zona di memoria Stack viene utilizzata per:
**• Memorizzare variabili globali**
**• Memorizzare il codice del programma**
**• Contenere le variabili relative alle funzioni**
**• Allocare la memoria dinamica**
**Risposta:** *Contenere le variabili relative alle funzioni*

Un sistema ha pagine da 1KB, indirizzi virtuali da 32bit e fisici da 16bit. Quanti bit sono dedicati all'offset di pagina?
**• 6** **• 10** **• 22**
**Risposta:** *10* (1KB → $2^{10}B$; allora ho 10 bit per l'offset)

Un sistema ha pagine da 1KB, indirizzi virtuali da 32bit e fisici da 16bit. Di quante pagine dispone un processo nello spazio degli indirizzi virtuali?
**• 64** **• 1024** **• circa 4 Milioni**
**Risposta:** *circa 4 Milioni* ($2^{32-10} \approx 4 \cdot 10^6$)

In Linux, il Loader è:
**• Un componente del SO**
**• La funzione principale di un programma**
**• La zona di memoria dove è memorizzato il codice del programma in esecuzione**
**Risposta:** *Un componente del SO*

In Linux, le librerie condivise sono mappate in memoria:
**• In una zona intermedia tra stack e heap**
**• Nella zona di dati**
**• Nella zona di codice**
**• Nello stack**
**Risposta:** *In una zona intermedia tra stack e heap*

u6-s2-memoria-dinamica

## Sistemi Operativi

## Unità 6: La memoria

# La memoria Dinamica

Martino Trevisan
Università di Trieste
Dipartimento di Ingegneria e Architettura

---

## Argomenti

1. Limiti della memoria statica
2. La memoria dinamica
3. La funzione **malloc**
4. La funzione **calloc**
5. La funzione **realloc**
6. La funzione **free**
7. Cenni di funzionamento interno

---

# Limiti della memoria statica

Diamo un po' di *motivazioni* per la *memoria dinamica*.

## Variabili Globali, Locali e Statici

Facciamo prima un ripasso sulle *variabili globali, locali e statici*

**Variabili Globali**
Le variabili globali sono allocate nella segmento di *dati* $\delta$ (Layout della Memoria Virtuale > ^6aeac3).
Il loader *inizializza* il valore

```c
int a = 40; /* Inizializzata dal loader*/
int main(){...}
```

Se non specificato, la variabile è inizializzata a $0$.

```c
int a; /* Inizializzata a 0 */
int main(){...}
```

**Variabili Locali**

Le variabili di funzione sono allocate nello *stack* $\sigma - \rho$ (Layout della Memoria Virtuale > ^b73f4b)

*NON* viene inizializzato il valore! *Possono contenere dati arbitrari*

```c
int f(int a, int b){
   int s = a + b;
   return s;
}
```

Gli argomenti `a` e `b`, la variabile `s` e il valore di ritorno si trovano nello stack

**Variabili Statiche**

le variabili in una funzione con la keyword **static** sono allocate *nel segmento* dati $\delta$ e non nello stack.

Inizializzate dal *loader*.

*Conservano in valore* dopo il termine della funzione.

```c
#include <stdio.h>
int fun(){
  static int count = 0; /* Inizializzata UNA volta sola dal loader all avvio del processo.
                           E NON ogni volta che la funzione viene invocata*/
  count++;
  return count;
}

int main(){
  printf("%d ", fun());
  printf("%d ", fun());
  return 0;
}
```

**Stampa** `1 2`
Sono effettivamente strane, non li useremo tanto

## Problema delle Variabili Globali

**Problema.** (*Variabili globali non determinate a priori*)
Ci sono casi in cui il programmatore *non sà quanti dati deve caricare in memoria*

- Lettura di una struttura dati da file
- Input utente di lunghezza variabile

Con quello che abbiamo visto, in C gli array hanno lunghezza fissa, nota a tempo di compilazione

```c
#define N 50
int v [N];
```

Come soluzione si può optare per la seguente

*Sovradimensionamento*: approccio con cui i programmatori creano vettori o matrici di dimensione molto grande

- Atti a contenere (quasi) ogni possibile input
- Approccio funzionante, ma non risolutivo
- La *Memoria Dinamica* risolve questo problema

Cosa non si può assolutamente fare è il seguente.
In C, *NON* si possono creare array di lunghezza non nota al compilatore
Il seguente codice è *sbagliatissimo!!!!!!*

```c
scanf("%d", &n);
int v[n];
```

## Esempio di Sovradimensionamento

**Esempio di sovradimensionamento:** media di $N$ numeri letti da tastiera

**Nota:** la media di $N$ numeri si può calcolare anche senza tenerli in memoria. Basta tenersi la somma $\Sigma$

```c
#include <stdio.h>
#define MAXN 50 /* Se n>50 il programma non funziona */
int main() {
    int n, i;
    float v[MAXN], s = 0;

    printf("Quanti numeri vuoi leggere? ");
    scanf("%d", &n); /* Se n>50 il programma non funziona */
    printf("Inserisci %d numeri:\n", n);

    for (i=0; i<n; i++)
        scanf("%f", &v[i]); //  &v[i] == v+i
    for (i=0; i<n; i++)
        s += v[i];
    printf("Media: %f\n", s/n);

    return 0;
}
```

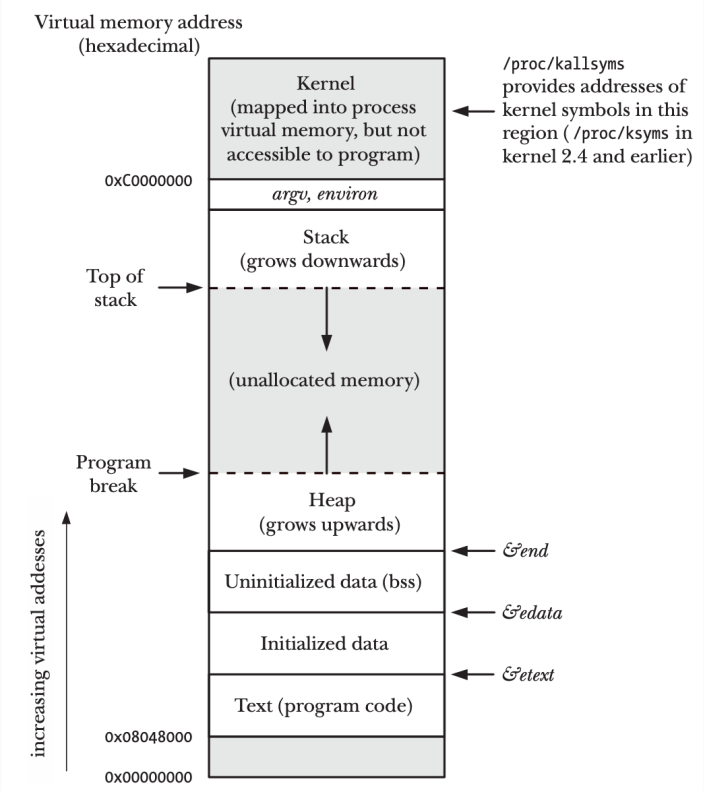
---

## La memoria dinamica

In C è possibile utilizzare la *memoria dinamica* per creare *strutture dati* la cui dimensione non è nota in fase di compilazione

**Uso tipico**: creazione di vettori di lunghezza arbitraria e decisa a *run time*. Esempio: lista linkata

**Funzionamento**: si utilizzano indirizzi virtuali nel segmento *Heap* $\xi$. Esso può crescere durante l'esecuzione del programma. Il *program break* serve per *"stabilire"* il limite degli indirizzi virtuali nello heap $\xi$, e si può andare a modificarlo.




# Manipolazione della Memoria Dinamica in Linux

### In Linux

Per utilizzare la memoria dinamica si utilizzano delle *funzioni di libreria* (`malloc`, `free`, ecc...) per *allocare o liberare blocchi di memoria*.

Le funzioni di libreria utilizzano la System Call `sbrk` che informa il sistema operativo che il processo emetterà indirizzi virtuali *in zone precedentemente* non usate (aumenta o decrementa il *program break*).

- In pratica si informa il SO che l'Heap sta crescendo e il processo accederà a *pagine di memoria virtuale aggiuntive*

### Thread-Safe

Tutte le funzioni di libreria per la memoria dinamica sono *Thread Safe*.

- Possono essere invocate in parallelo da molteplici *thread*
  - *Ma non all'interno di Signal Handler!*
- Internamente mantengono e usano *mutex* (che vedremo) per regolare l'accesso alle strutture dati

# Funzione `malloc`

```c
#include <stdlib.h>
void *malloc(size_t size);
```

Alloca **size** byte di memoria e ritorna il puntatore alla memoria allocata.
La memoria *NON* è inizializzata, può contenere qualsiasi valore
Se l'allocazione *fallisce* (e.g., manca memoria), ritorna **NULL**. Altrimenti ritorna l'indirizzo **void *** da convertire mediante un cast.

**Note** (*Utilizzo*)

1. La **malloc** richiede **size** in byte. Bisogna utilizzare l'operatore **sizeof** per conoscere la dimensione del tipo di variabile da allocare. Ricordiamoci che viene valutata in *fase di compilazione*!
2. Il valore di ritorno è **void ***, ovvero un puntatore senza tipo.
   Per utilizzare la memoria allocata, conviene *assegnarla a un puntatore al tipo desiderato*

**Esempio:**

```c
/* Vogliamo allocare un vettore di float*/
float * v;
/* La lunghezza è determinata a run time */
scanf("%d", &n);
/* I byte da allocare sono n blocchi ognuno lungo quanto un float */
v = malloc(n * sizeof(float)); /* Un void* è assegnato a un float* */
v[0] = 12.2; /* Aritmetica dei puntatori */
```

## La funzione calloc

```c
#include <stdlib.h>
void *calloc(size_t nmemb, size_t size);
```

Simile alla **malloc**. Funzione gemella
Alloca memoria per un array di **nmemb** *elementi* ognuno di **size** byte e ne *ritorna il puntatore*.
La memoria *é inizializzata* a 0.

**Osservazione:** a differenza della **malloc**, la **calloc** riceve **size** e **nmemb** e fa la moltiplicazione internamente. Come mai? Boooh; comunque useremo più la **malloc**

## La funzione realloc

Voglio *modificare* la dimensione della memoria dinamica appena creata con **malloc**.
Come famo?

```c
#include <stdlib.h>
void *realloc(void *ptr, size_t size);
```

*Modifica la dimensione della zona di memoria* puntata da **ptr** a **size** byte.
Il valore di ritorno è il puntatore *alla zone estesa*

- Se comporta un *restringimento* della zona di memoria, i dati in eccesso sono *persi*
- Se comporta un aumento, la zona aggiuntiva *NON* è inizializzata (come con malloc)

**Nota:** **ptr** deve essere stato ottenuto con **malloc** **calloc** o **realloc**. Altrimenti succedono cose grave...

**Osservazione:** se possibile, la **realloc** estende la zona di memoria corrente, e il valore di ritorno è uguale a **ptr**
Se non è possibile, i dati vengono copiati in *una nuova regione*, il cui indirizzo viene ritornato

## La funzione free

```c
#include <stdlib.h>
void free(void *ptr);
```

*Dealloca* (o libera) la zona di memoria indicata da **ptr**.

- Ovviamente **ptr** deve essere stato ottenuto con **malloc** **calloc** o **realloc**
  Se si tenta di liberare più volta una zona di memoria, il comportamento non è definito (ovvero è meglio evitare di far casini, probabilmente si ottengono dei *segmentation Fault*.

## Esempio

**Esercizio:** si scriva un programma che memorizza un numero $N$ di **float** letti da tastiera.
Il numero $N$ è letto da tastiera all'inizio del programma.
Infine il programma ne stampa la media.

```c
#include <stdio.h>
#include <stdlib.h>
int main() {
   int n, i;
   float * v, s = 0;

   printf("Quanti numeri vuoi leggere? ");
   scanf("%d", &n);
   printf("Inserisci %d numeri:\n", n);

   v = malloc (n*sizeof(float)); /* Allocazione. Notare cast implicito da void* a
float* */
   for (i=0; i<n; i++)
      scanf("%f", &v[i]);

   for (i=0; i<n; i++) /* Calcola la somma */
      s += v[i];

   printf("Media: %f\n", s/n);
   free (v); /* Deallocazione */
   return 0;
}
```

---

## Importanza della `free`

Tutte le zone di memoria vanno deallocate tramite la `free`
Se non viene fatto, la memoria è liberata al termine del processo

**Importante!**

- Non deallocare la memoria è sempre un errore!
- Nei programmi che devono essere eseguiti per lungo tempo, la memoria non deallocata causa il cosiddetto *Memory Leak*: a un certo punto, viene allocata tutta la memoria del sistema!
- Per questo in certi contesti *non* è consentito usare la memoria dinamica.

## Errori comuni con la Memoria Dinamica

**Errori comuni:**
Valore di ritorno di `malloc` non assegnato a un puntatore

```c
// Errato
float v = malloc(5*sizeof(float));
float v [10] = malloc(5*sizeof(float));
// Corretto
float * v = malloc(5*sizeof(float));
```

Creare un array la cui dimensione non è nota durante la compilazione

```c
// Errato
float v [n];
// Corretto
float * v = malloc(n*sizeof(float));
```

**Errori comuni:**
Utilizzo errato dell'aritmetica dei puntatori

```c
float * v = malloc(5*sizeof(float));
// Errato
v+2 = 43.5; // v+2 è un puntatore
&(v+2) = 43.5;  // (v+2) è già un puntatore. Usare '&' non ha senso
// Corretto
*(v+2) = 43.5;
v[2] = 43.5;
```

Utilizzo errato nella **scanf**

```c
// Errato
scanf("%f", v[2]);
scanf("%f", *(v+2) );
// Corretto
scanf("%f", &v[2]);
scanf("%f", v+2 );
```

## Esercizi con la Memoria DInamica

**Esercizio:** si scriva una funzione che ritorna un sequenza di $N$ `float` equispaziati tra $a$ e $b$

```c
#include <stdlib.h>
#include <string.h>
float * seq(int N, float a, float b){
    int i;
    float * v;

    v = malloc(N*sizeof(float));
    for (i=0; i<N; i++)
        v[i] = a + (float)i/N*(b-a); /* Cast a float necessario per 'i' */

    return v;
}
```

**Utilizzo:**

```c
int i ;
float * s = seq(10, 2, 5);
for (i=0; i<10; i++)
    printf("s[%d]==%f\n", i, s[i]);
free(s); /* Importante! */
```

**Esercizio:** si scriva una funzione che riceve come argomenti un interno $N$ e un pattern $p$. La funzione ritorna una stringa lunga $N$ che contiene il pattern $p$ ripetutamente.
Esempio: `repeat(5, "ah")` $\rightarrow$ `ahaha`

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

char * repeat(int N, char * pattern){
   int i, l;
   char * s;

   s = malloc((N+1)*sizeof(char));
   s[N] = '\0';
   l = strlen(pattern);

   for (i=0; i<N; i++)
      s[i] = pattern[i%l];

   return s;
}
```

**Utilizzo:** `printf("%s\n", repeat(15, "ciao! ") );` stampa: `ciao! ciao! cia`

---

# Cenni di funzionamento interno della Malloc

Le funzioni `malloc calloc realloc free` sono delle funzioni di libreria: mantengono lo heap $\xi$ con delle *strutture dati*. Vediamo in dettaglio *come* fanno a mantenere questo heap

## System Call sbrk

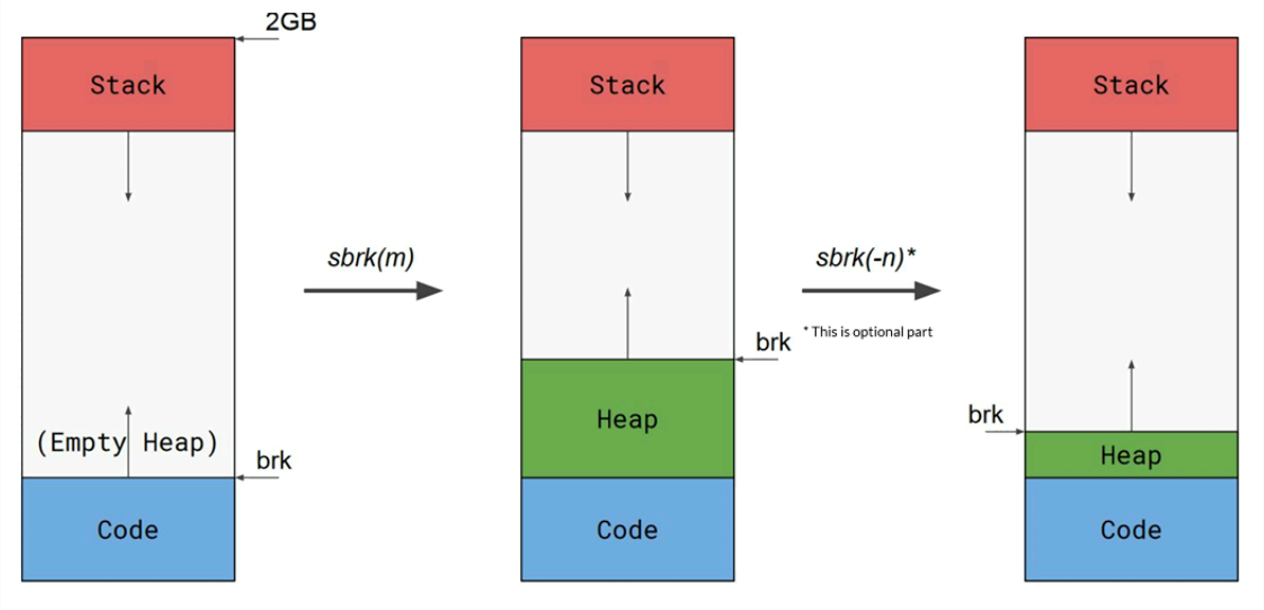
Esse usano la System Call `sbrk`.

```c
void *sbrk(intptr_t increment);
```

Incrementa di `increment` il *data segment*, inteso come unione di *segmento codice, dati e heap* ($\zeta \cup \delta \cup \xi$).
In pratica, informa il SO che l'*heap* si sta *espandendo*.

- Il SO, se necessario, *imposterà* la MMU per accogliere pagine aggiuntive

Chiamare la **sbrk** è di per se sufficiente per poter usare indirizzi virtuali più alti




Tuttavia, per il programmatore sarebbe difficile gestire la memoria dinamica solo usando la **sbrk**

- Dovrebbe tenere traccia di *ogni allocazione e di ogni de allocazione*
- Dovrebbe avere una tecnica per riusare i *buchi* lasciati liberi da una deallocazione
  - Nel momento in cui si fa una nuova allocazione, così da evitare delle eventuali *frammentazioni*.
- Invocare la **sbrk** a ogni allocazione è *inefficiente*
  - Una System Call è lenta (implica un *Context Switch*)
  Che incubo! Meno male che non bisogna reinventarsi la ruota...

Le funzioni di libreria **malloc**, etc., gestiscono tutto ciò per il programmatore

- Utilizzando opportune *strutture dati*

---

## Storia della Malloc

La moderna funzione **malloc** deriva dalla proposta di Doug Lea, professore della *State University of New York at Oswego*

Internamente usa una *linked list* per tenere traccia delle zone occupate.

**Nota Tecnica:** *heap* ha due significati!

1. Una struttura dati che implementa una coda a priorità tramite un albero (*informatica teorica*)
   - Permette di trovare facilmente il massimo di un insieme di numeri
   - Veloce da aggiornare

2. La zona della memoria virtuale dove viene allocata la memoria dinamica (*informatica ingegneristica*) ($\xi$)
   Queste due definizioni non c'entrano per niente a vicenda

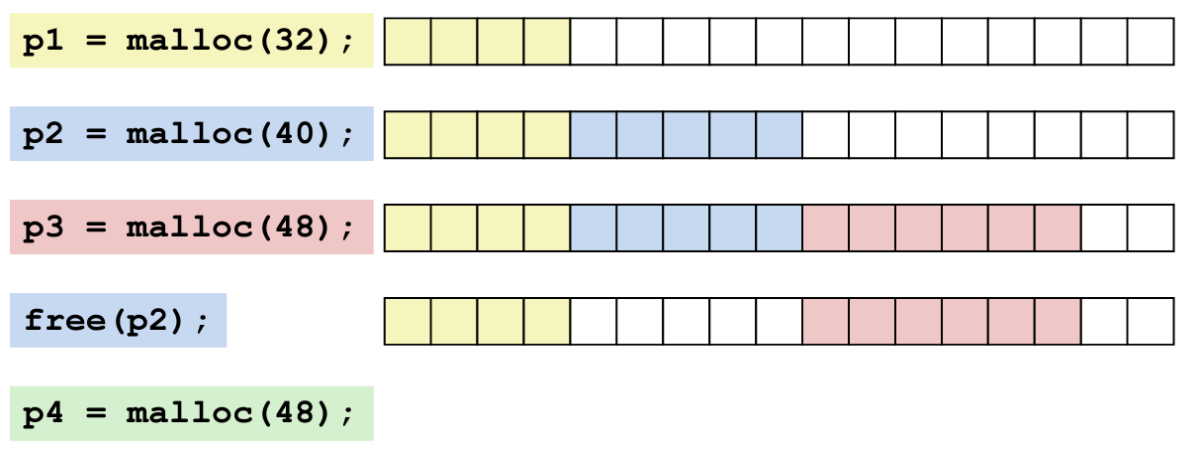
## Funzionamento della Malloc

La **malloc** gestisce *blocchi di grandezza variabile*

- Non c'è nessuna discretizzazione o utilizzo di blocchi di grandezza fissa
- Porta ad avere *frammentazione esterna*: memoria sprecata perchè è una zona contigua troppo piccola per essere allocata

**Esempio.** (*Frammentazione esterna*)
E' possibile che si giunga a situazione come questa:

- Esempio con blocchi di dimensione fissa di 8B




**malloc(48)** potrebbe essere evasa, se la memoria libera fosse contigua. Cosa si fa per risolvere questo problema? Niente, la si accetta com'è.

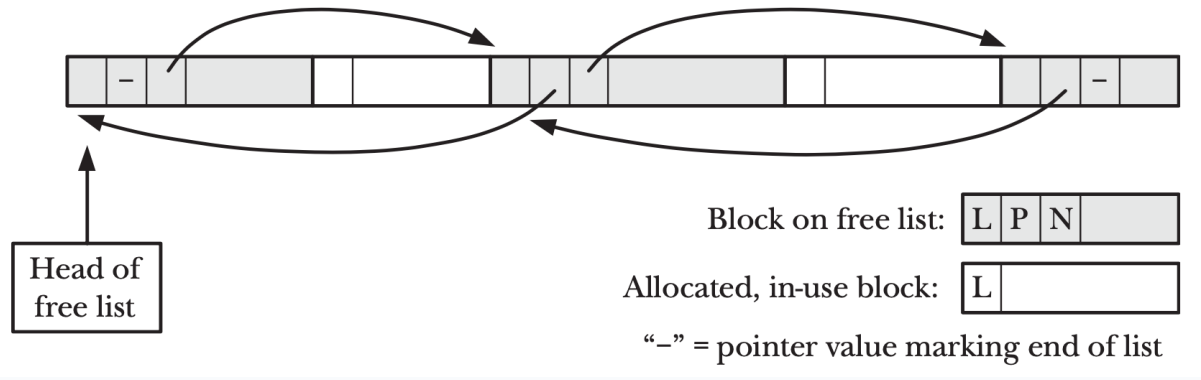
## Struttura Dati: Arena

La **malloc** gestisce indipendentemente *più di una zona di memoria*, dette *Arenas*.

- Le strutture dati sono replicate
- Rende *più efficiente l'utilizzo in contesti multithread*
  - Le funzioni **malloc**, etc., sono Thread Safe
- Evita che diversi thread vengano rallentati aspettando il release di un *lock*
  - I lock sono necessari, ma l'utilizzo di più di un strutture ne diminuisce l'impatto
- Le *Arena* sono praticamente un espediente per la *malloc efficiente* e *sicura*.

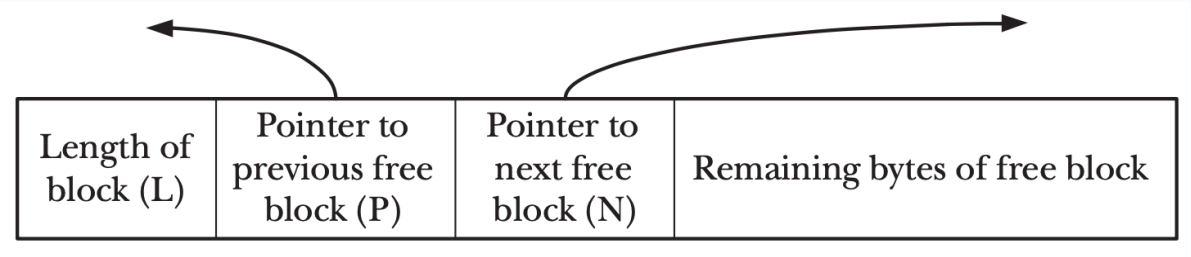
## Struttura Dati: Linked List e Doubly Linked List

1. Una zona di memoria gestita dalla **malloc** é amministrata tramite una *linked list*
2. I segmenti ancora liberi sono una *Doubly linked list*
   - Le zone allocate sono momentaneamente rimosse dalla lista

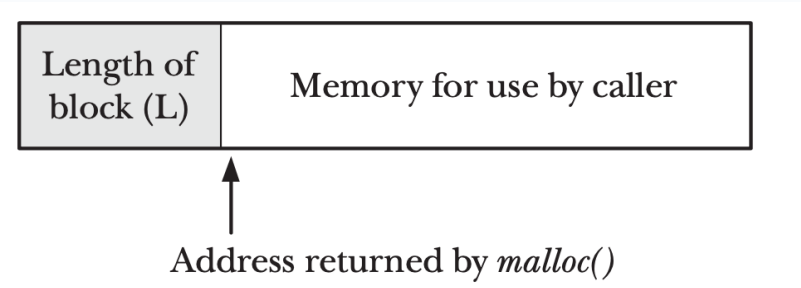



3. Ogni *zona libera o allocata* ha una **struct** nei *primi byte* che fornisce *informazioni su di essa e sui blocchi adiacenti* (struttura interna delle zone di memoria)

- **Zona Libera**




- **Zona Allocata**

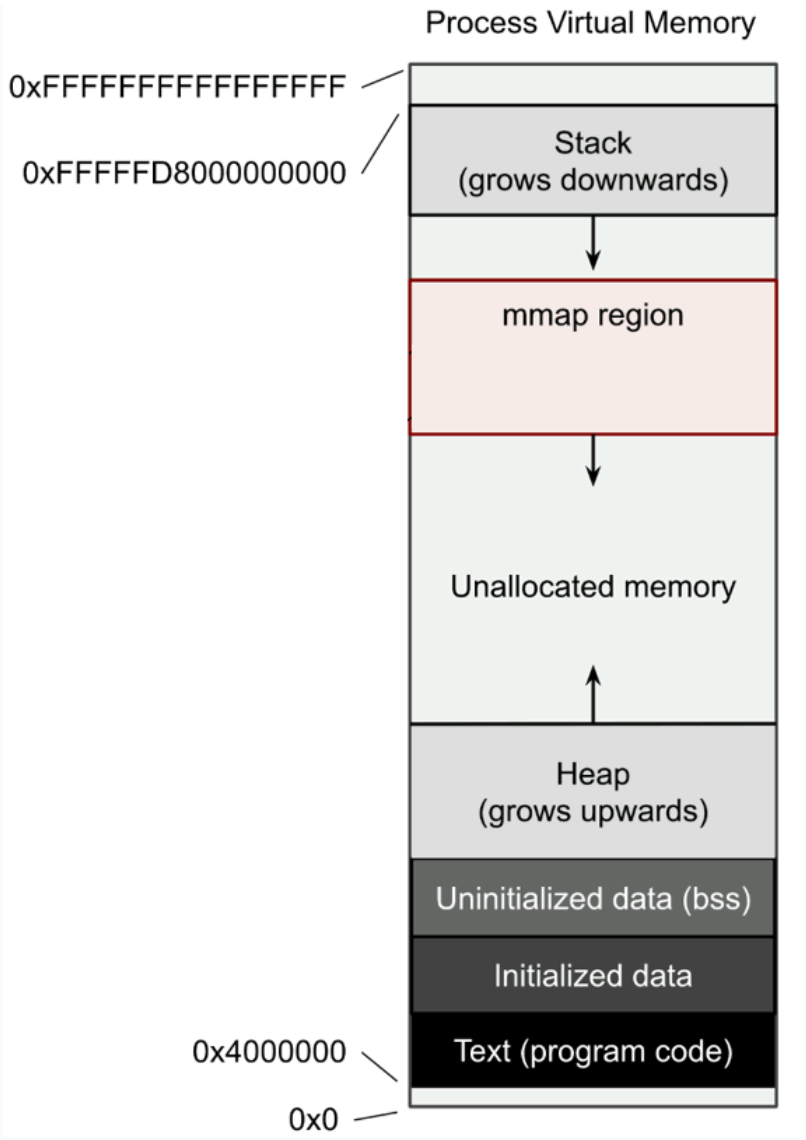



---

## Allocazione di Grandi Regioni di Memoria

In caso la **malloc** debba allocare *grandi regioni di memoria* (tipicamente $> 128\ kB$) usa la System Call **mmap** per allocare una zona di memoria.

- **malloc** chiede una regione di tipo **MAP_ANONYMOUS|MAP_PRIVATE**. Non deve essere condivisa con nessuno (no flag **MAP_SHARED**!)
- Il SO crea *una o più pagine per il processo*
- Le colloca in una posizione *a sua scelta* nello *spazio degli indirizzi virtuali*
- Non sempre ottengo un *puntatore* in $\xi$




---

## Domande

Si consideri il seguente codice C:

```c
int c = 40;
int main(){
    int i;
    static int j;
    ...
}
```

Quali variabili risiedono nello stack?

• **Tutte** • **i e j** • **i**

**Risposta:** *i*

Il seguente codice è corretto in C?

```c
#define size 1024
int i [size];
```

**• Si** **• No**
**Risposta:** *Si*

Si completi il seguente codice C

```c
double * a, int i;
scanf("%d", &i);
a = ...
```

**• float[i]** **• malloc(i);**
**• malloc(i * sizeof(double));**
**• malloc(sizeof(double));**
**Risposta**: **malloc(i*sizeof(double));**

La **malloc**:
**• è una System Call**
**• è utilizzata dalla funzione sbrk**
**• utilizza la System Call sbrk**
**Risposta:** *utilizza la System Call sbrk*