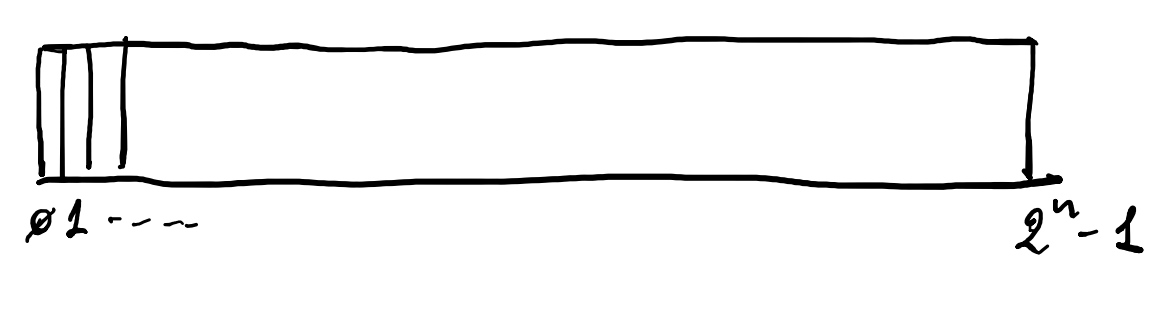
# SC - 21/5/2025

mercoledì 21 maggio 2025 10:20

## GESTIONE DELLA MEMORIA

Ogni processo vede ed usa la memoria come se fosse uno spazio lineare indirizzabile completamente a sua disposizione.



Un problema fondamentale per il SO è come gestire la memoria fisica disponibile in modo che la stessa possa essere acceduta concorrentemente da TUTTI i processi attivi

① Deve Tenere Traccia di Tutte le locazioni di memoria in uso

② Considerare nuove richieste di utilizzo della memoria allocando gli spazi liberi rimanenti

Per indirizzare la memoria (purtroppo) esistono due metodi:

POTENZE DI 10

|  |  |  |
|---|---|---|
| $10^3$ byte | Kilobyte | KB |
| $10^6$ " | megabyte | MB |
| $10^9$ " | giga " | GB |
| $10^{12}$ " | tera " | TB |
| $10^{15}$ " | peta " | PB |
| $10^{18}$ | exa " | EB |
| $10^{21}$ | zetta " | ZB |
| $10^{24}$ | yotta " | YB |

POTENZE DI 2

|  |  |  |
|---|---|---|
| $2^{10}$ byte | Kibibyte | KiB |
| $2^{20}$ " | Mibibyte | MiB |
| $2^{30}$ " | Gibibyte | GiB |
| ⋮ | ⋮ | ⋮ |

## ALLOCAZIONE DELLA MEMORIA

Immaginiamo che il SO sia dotato di un "allocatore" rappresentato da una struttura dati che registra quali parti della memoria fisica sono libere/occupate.

I programmi possono fare richieste all'allocatore con tre primitive

- p = alloca (n) : restituisce l'indirizzo iniziale p di un blocco di memoria contiguo di dimensione pari ad almeno n byte

  => in C è implementata da malloc

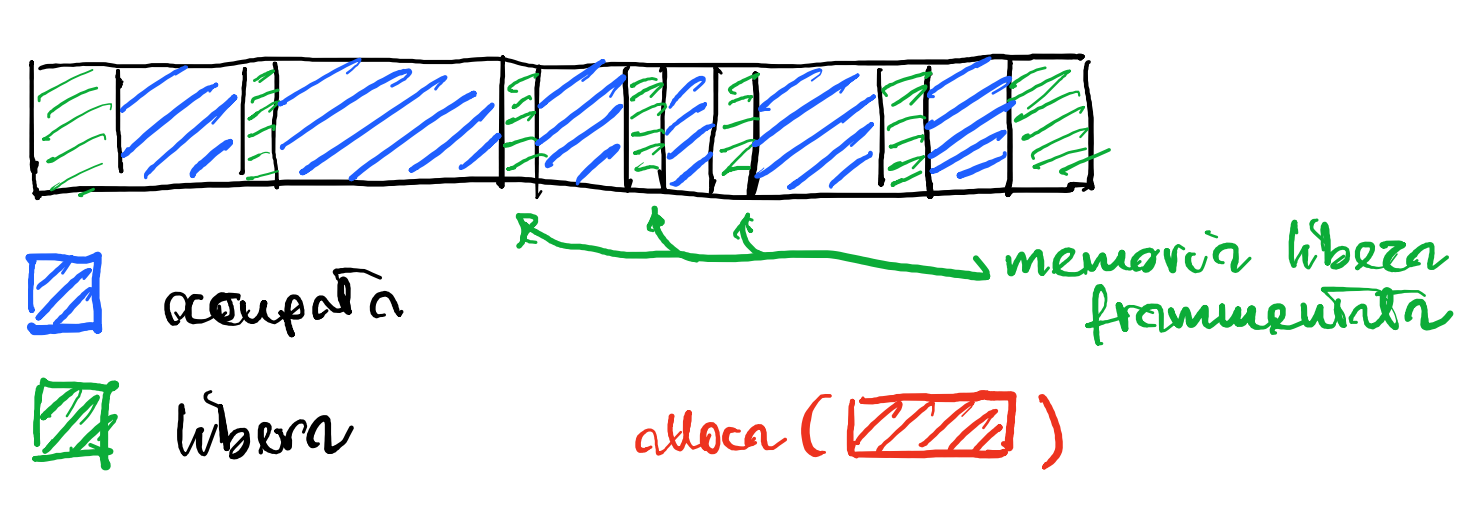
- p' = ridimensiona (p, n) : ridimensiona un blocco di memoria precedentemente allocato a partire da p perché abbia una nuova dimensione di almeno n bytes.
  ATTENZIONE: il nuovo blocco allocato potrebbe essere raggiungibile ad un nuovo indirizzo p'.

- dealloca (p) : libera lo spazio di memoria che era stato precedentemente allocato a partire da p

  => in C è implementata da free()

## FRAMMENTAZIONE DELLA MEMORIA

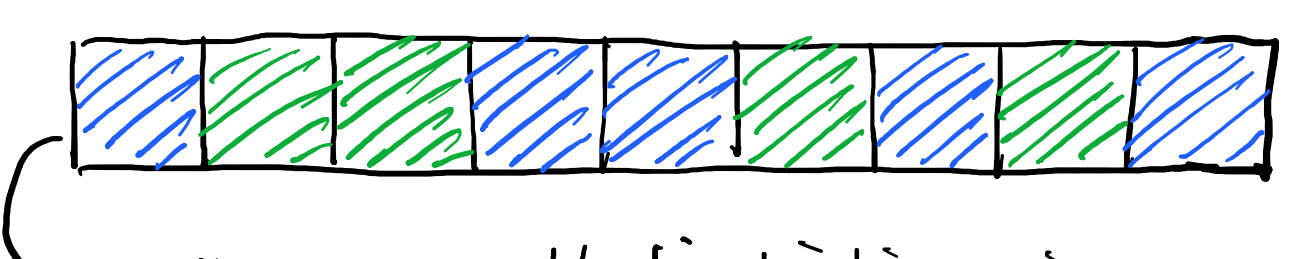
E' il problema per cui in memoria nessuno spazio libero contiguo è grande a sufficienza per soddisfare una richiesta di allocazione


memoria libera frammentata
occupata
libera
alloca ( )

Esistono due Tipi di frammentazione

FRAMMENTAZIONE INTERNA: sono presenti spazi di memoria non utilizzata in blocchi allocati.

FRAMMENTAZIONE ESTERNA: quando una richiesta di allocazione di dimensione n fallisce perché Tutti i blocchi di memoria libera hanno dimensione < n
ATTENZIONE: la frammentazione esterna si può verificare solo se l'allocatore assegna blocchi di dimensione variabile



se alloco sempre blocchi di dimensione m:

⊕ Non ho fram. esterna
⊖ Aumento la fram. interna
⊖ Potrei non riuscire a soddisfare le richieste per n > m

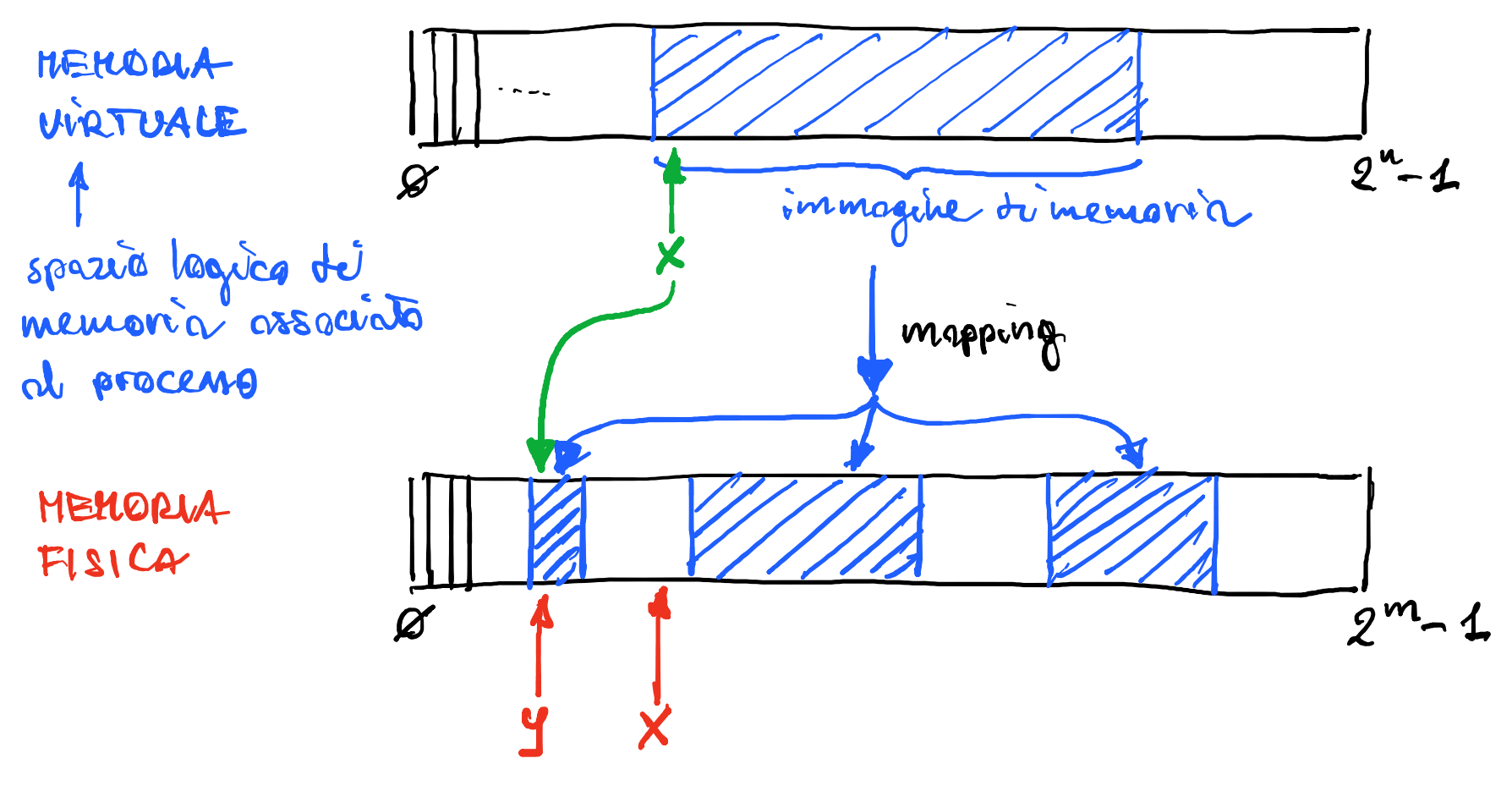
L'allocazione della memoria è tipicamente implementata a due livelli

① Un primo allocatore rende disponibile la memoria fisica ai processi

② Un secondo allocatore lavora a livello del singolo processo per rispondere alle richieste di allocazione dinamica della memoria.

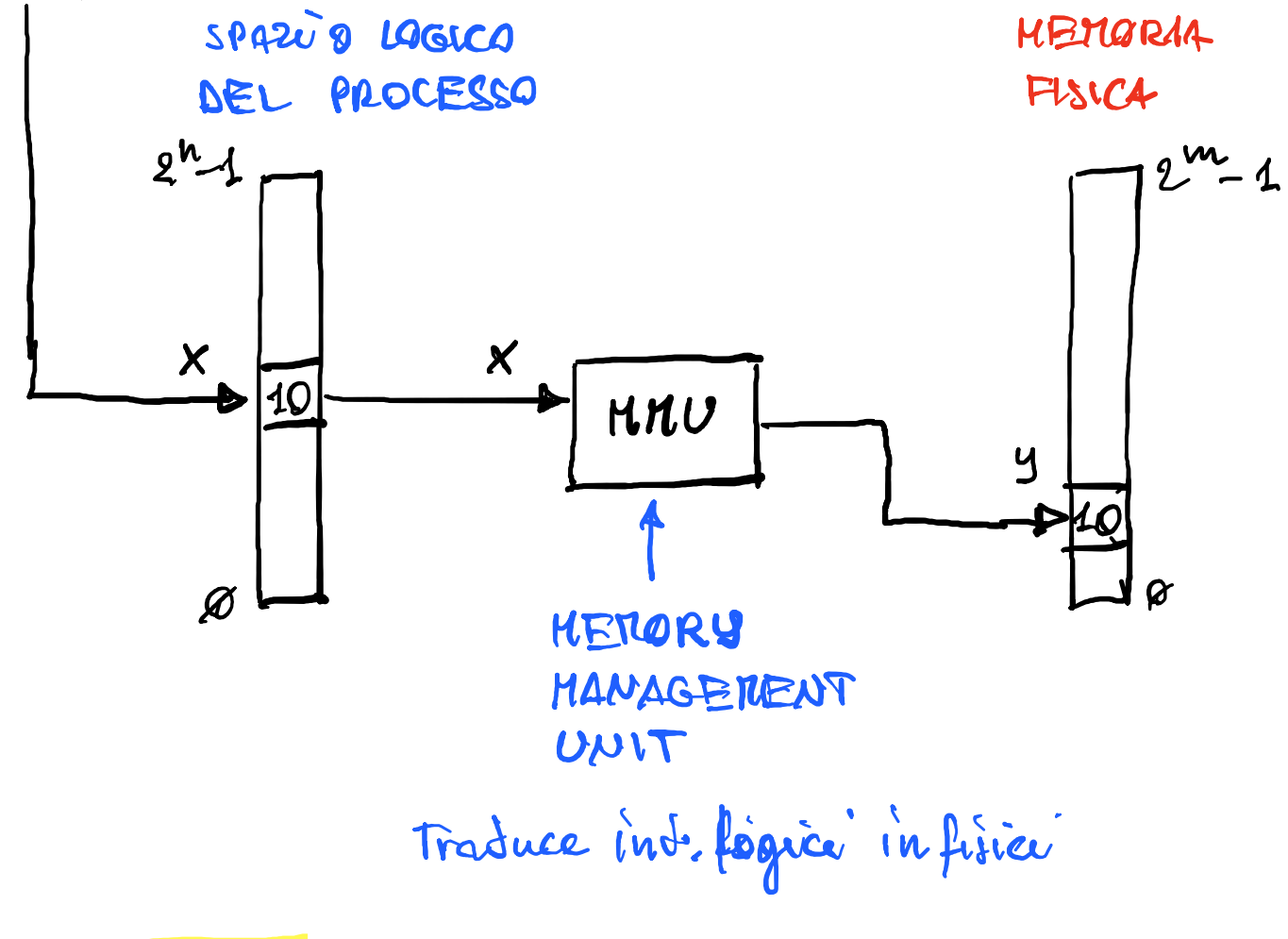
## MEMORIA FISICA E MEMORIA VIRTUALE

La memoria virtuale è una astrazione Tramite cui ogni processo accede ad uno spazio di memoria a lui dedicato caratterizzato da INDIRIZZI LOGICI.
Il processo utilizza gli indirizzi logici come se fossero indirizzi fisici. E' il SO a preoccuparsi della Traduzione logico<->fisico.


MEMORIA VIRTUALE
spazio logico di memoria associato al processo
0
$2^n - 1$
immagine di memoria
x
mapping
MEMORIA FISICA
0
$2^m - 1$
y
x

Il meccanismo di gestione della memoria virtuale permette l'accesso in memoria fisica da parte di un processo attraverso la Traduzione automatica di indirizzi logici in indirizzi fisici

movl $10, (%esp)

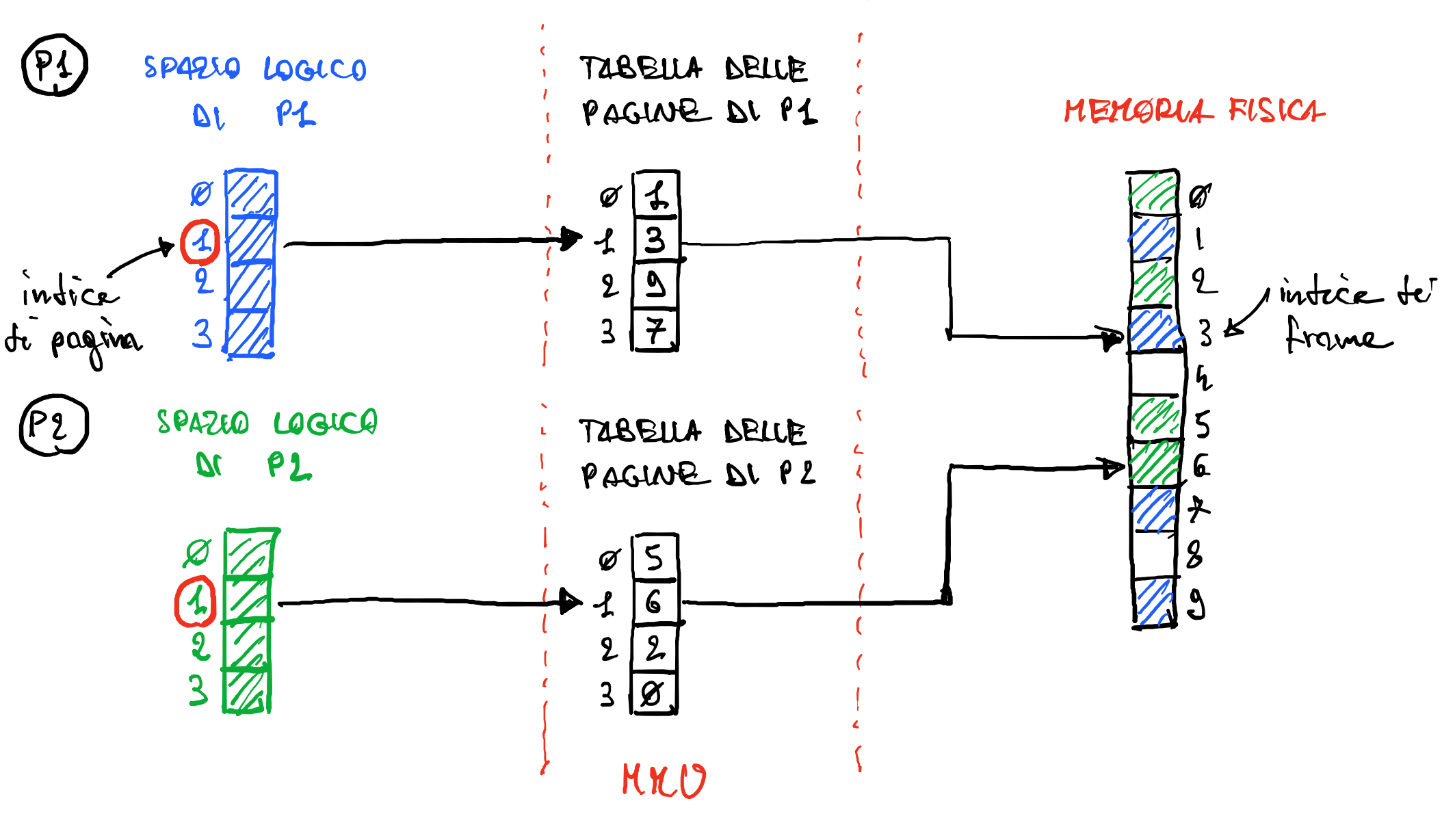

SPAZIO LOGICO DEL PROCESSO
$2^n - 1$
x
10
0
x
MMU
MEMORY MANAGEMENT UNIT
MEMORIA FISICA
$2^m - 1$
y
10
0

Traduce ind. logici in fisici

## PAGINAZIONE DELLA MEMORIA

Con la paginazione lo spazio di indirizzamento logico/fisico viene suddiviso in blocchi di dimensione fissa chiamati:

- "pagine" se nella memoria virtuale
- "frame" " " " fisica


P1
SPAZIO LOGICO DI P1
0
1
2
3
indice di pagina
TABELLA DELLE PAGINE DI P1
0 1
1 3
2 9
3 7
MEMORIA FISICA
0
1
2
3
4
5
6
7
8
9
indice di frame
P2
SPAZIO LOGICO DI P2
0
1
2
3
TABELLA DELLE PAGINE DI P2
0 5
1 6
2 2
3 0
MMU

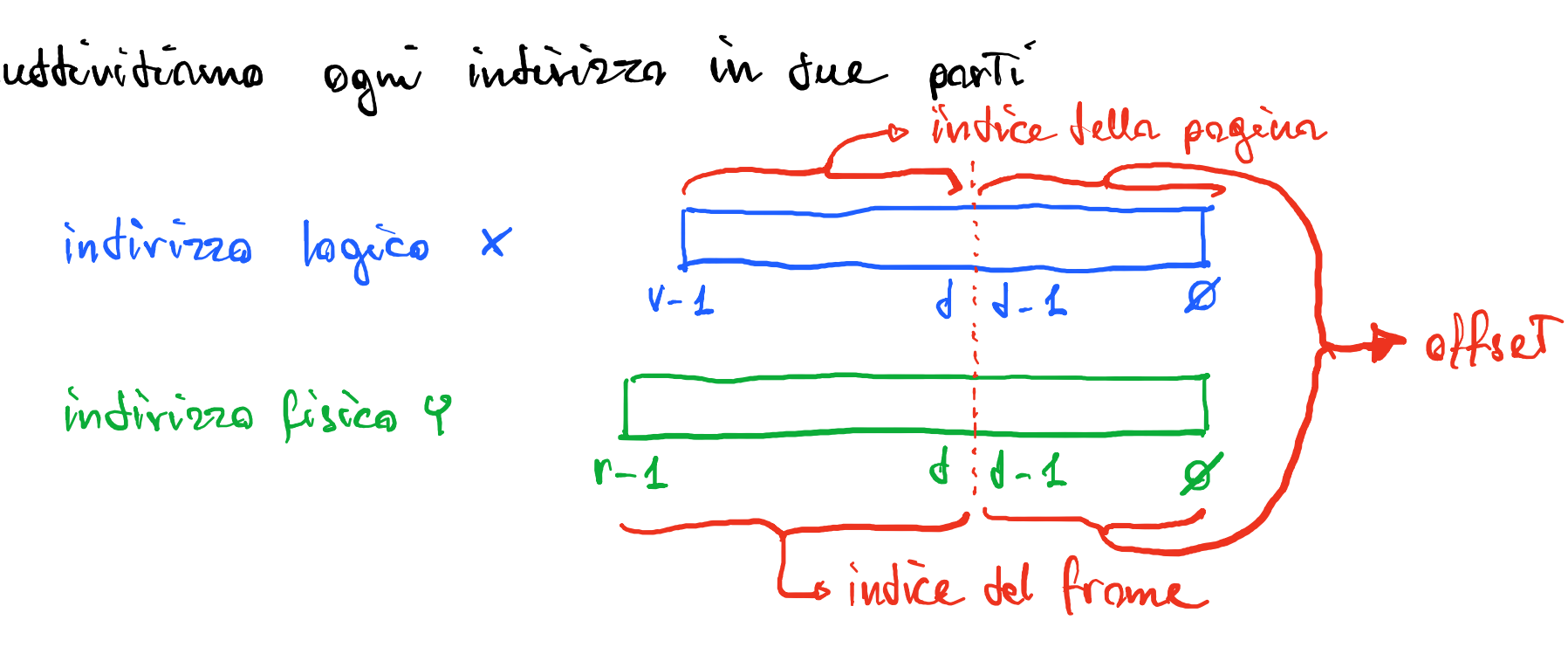
Schema finale di Traduzione degli indirizzi da logici a fisici

Supponiamo:

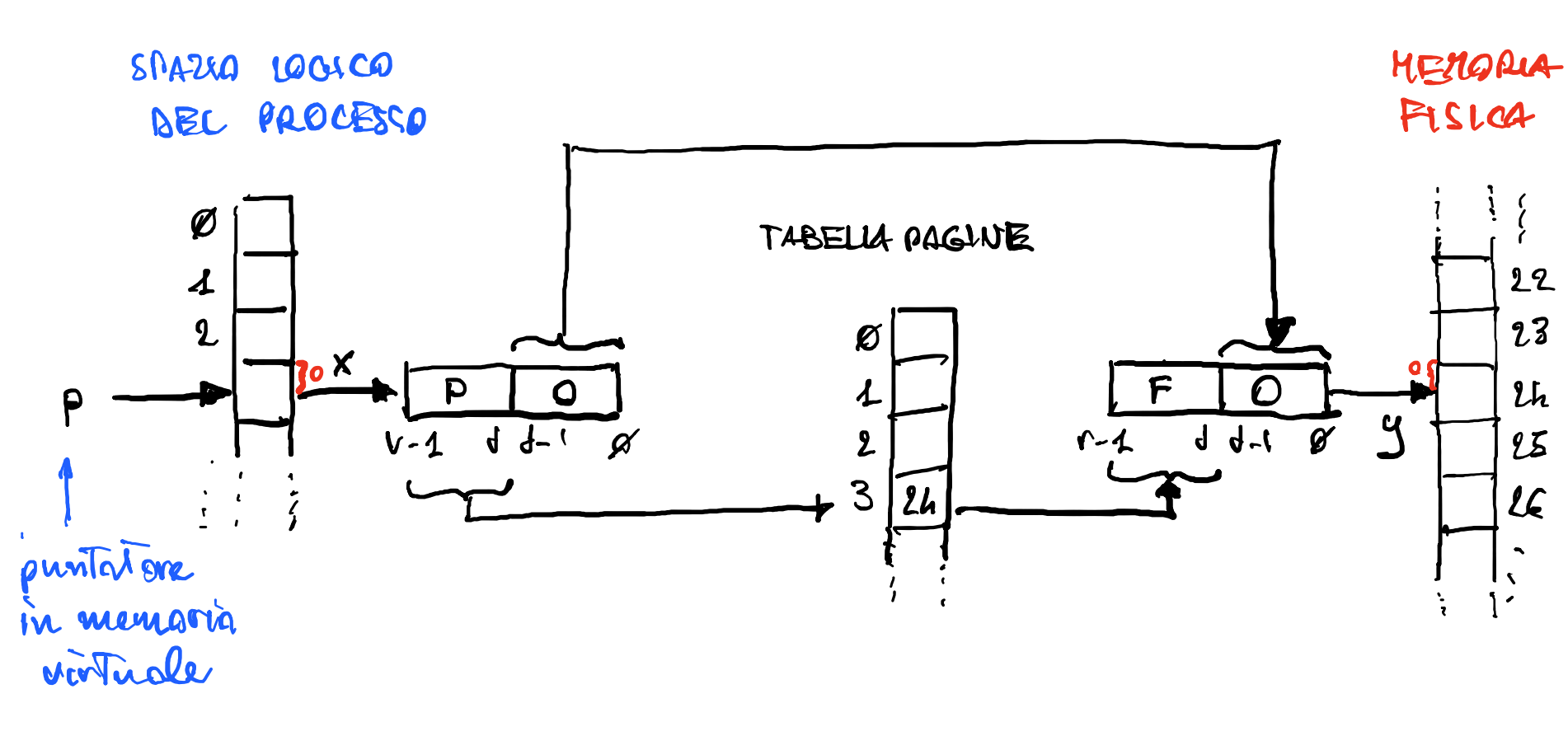
- Memoria virtuale con $2^v$ byte disponibili (es $v = 32 \rightarrow 4GB$ RAM)
- Memoria fisica con $2^r$ byte " (es $r = 33 \rightarrow 8GB$ RAM)
- Pagine di dimensione $2^d$ byte (es.: $d = 12 \rightarrow$ pagine di 4 KB)

Come Traduciamo indirizzi logici a v in ind. fisici a r bit.

Suddividiamo ogni indirizzo in due parti


indice della pagina
indirizzo logico x
v-1
d
d-1
0
offset
indirizzo fisico y
r-1
d
d-1
0
indice del frame

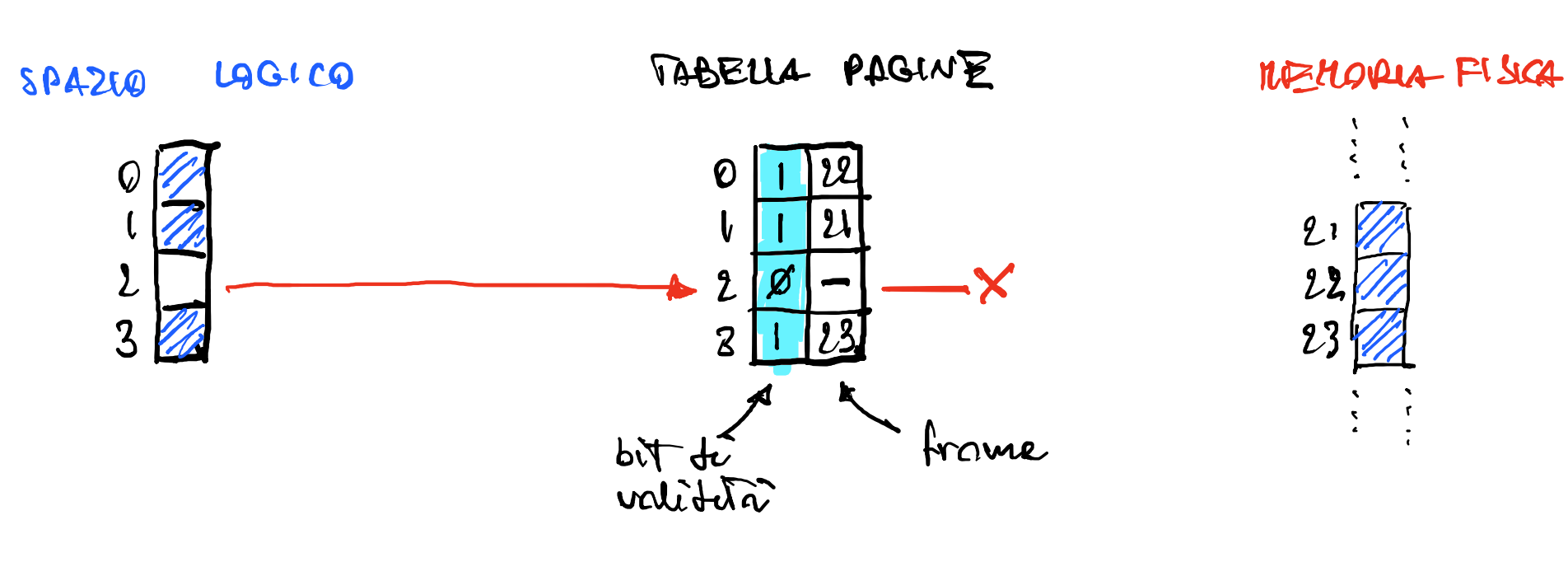
Per ottenere y da x si Traduce l'indice di pagina nel corrispondente indice di frame, e si completa mantenendo inalterato l'offset.


SPAZIO LOGICO DEL PROCESSO
0
1
2
p
puntatore in memoria virtuale
x
P
O
v-1
d d-1
0
TABELLA PAGINE
0
1
2
3
24
F
O
r-1
d d-1
0
y
MEMORIA FISICA
22
23
24
25
26

## PAGINAZIONE CON BIT DI VALIDITA'

Ad ogni riga (pagina) nella Tabella delle pagine aggiungiamo un bit di validità:

0 pagina inutilizzata (non valida)
1 " in uso (valida)


SPAZIO LOGICO
0
1
2
3
TABELLA PAGINE
0 1 22
1 1 21
2 0 —
3 1 23
bit di validità
frame
MEMORIA FISICA
21
22
23

## SWAPPING

A volte i dati presenti in pagine valide possono rimanere inutilizzati a lungo.
Se il sistema non ha più frame liberi disponibili può trasferire i frame poco utilizzati su disco (SWAP-OUT)

Quando un processo accede ad un indirizzo in un frame oggetto di swap-out:

① la CPU genera una interruzione "page fault"

② il Kernel seleziona nella memoria fisica un frame "vittima"

③ viene fatto swap-out del frame vittima

④ nel frame vittima viene caricata la pagina a cui vuole accedere il processo che ha causato il page fault (SWAP-IN)

⑤ vengono aggiornate in MMU le Tabelle delle pagine di entrambi i processi

⑥ il processo interrotto riprende eseguendo di nuovo l'operazione causa del page fault

**PAGINAZIONE CON VALIDITA E SWAPPING**

⊕

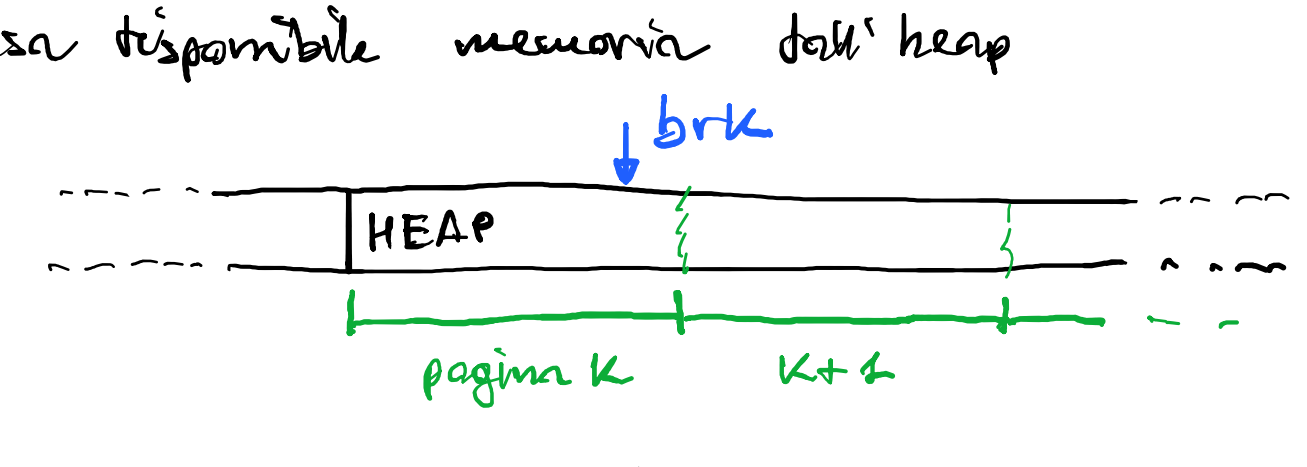
- Tutti i processi usano uno stesso spazio di indirizzamento
- E' possibile mappare su memoria fisica le sole pagine in uso
- E' virtualmente disponibile memoria per un numero illimitato di processi

⊖

- Trashing
- gestione della memoria complessa
- non supportato da Tutte le piattaforme

## ALLOCAZIONE DELLA MEMORIA

Quando richiediamo di allocare memoria con malloc(x) ci viene resa disponibile memoria dall'heap


brk
HEAP
pagina k
k+1

brk: puntatore che definisce lo spazio massimo allocabile da malloc. E' modificabile Tramite la syscall "sbrk"

Le primitive malloc e free partizionano lo spazio nell'heap in Tre parti:

① Spazio allocato ed in uso al processo

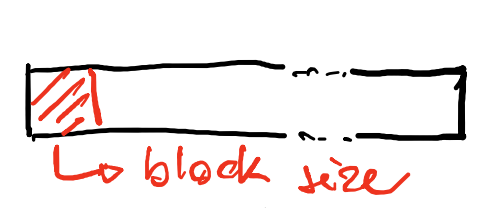
② Spazio libero potenzialmente allocabile

③ Spazio occupato da metadati

Una strategia comune per la gestione dell'heap prevede che questo sia allocato in blocchi. Ogni blocco è una regione dell'heap contigua che ①, ② ③.
Non ci può essere intersezione Tra i blocchi

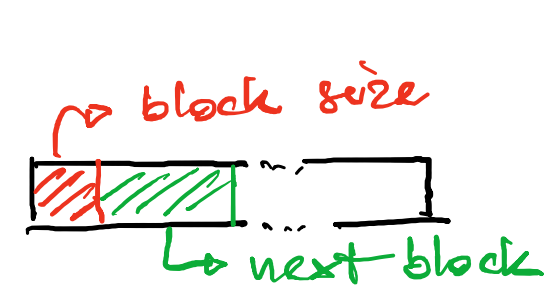
Ogni blocco contiene un header

SE IL BLOCCO E' IN USO



- dimensione del blocco (block size) espressa con 4 byte

SE IL BLOCCO E' LIBERO


block size
next block

- block size
- puntatore al blocco successivo (8 byte)

I blocchi liberi sono organizzati in una lista puntata dalla variabile globale free-list

Per consentire una corretta gestione dei blocchi liberi, ogni richiesta di allocazione alloca ALMENO 12 byte.

p = malloc (n) : cerca a partire da free_list un blocco libero di dimensione $s \geq n+4 \geq 12$. Se esiste lo toglie dalla lista. La scelta del blocco può seguire diverse strategie

① FIRST FIT : seleziona il primo blocco adatto

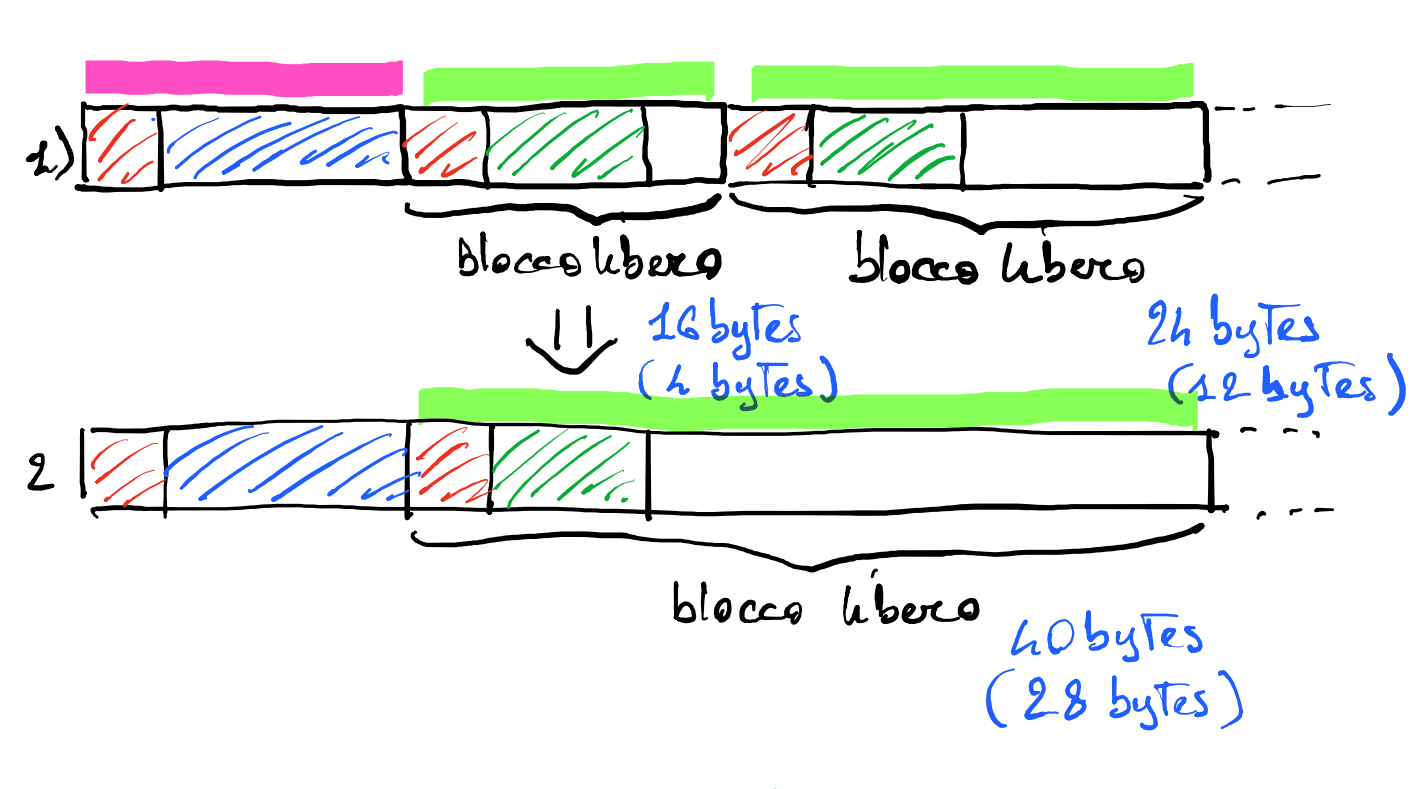
② BEST FIT : " " blocco più piccolo adatto (riduce framm. interna)

Se non esiste un blocco adatto espande l'heap di n+4 byte e crea in questo spazio un nuovo blocco

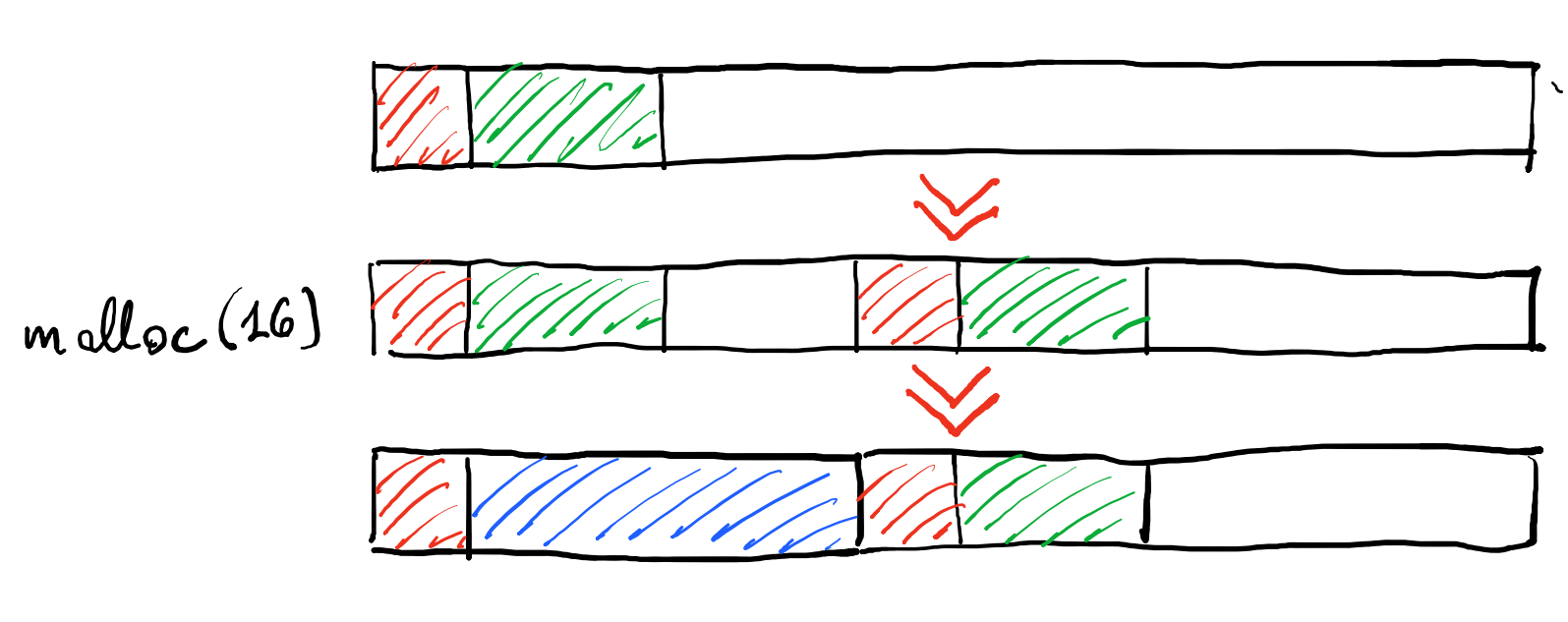
free(p) : aggiunge il blocco puntato da p-4 in Testa a free_list

## POSSIBILI MIGLIORAMENTI

- Fusione di blocchi liberi (adiacenti)


1)
Blocco libero
blocco libero
16 bytes
(4 bytes)
24 bytes
(12 bytes)
2
blocco libero
40 bytes
(28 bytes)

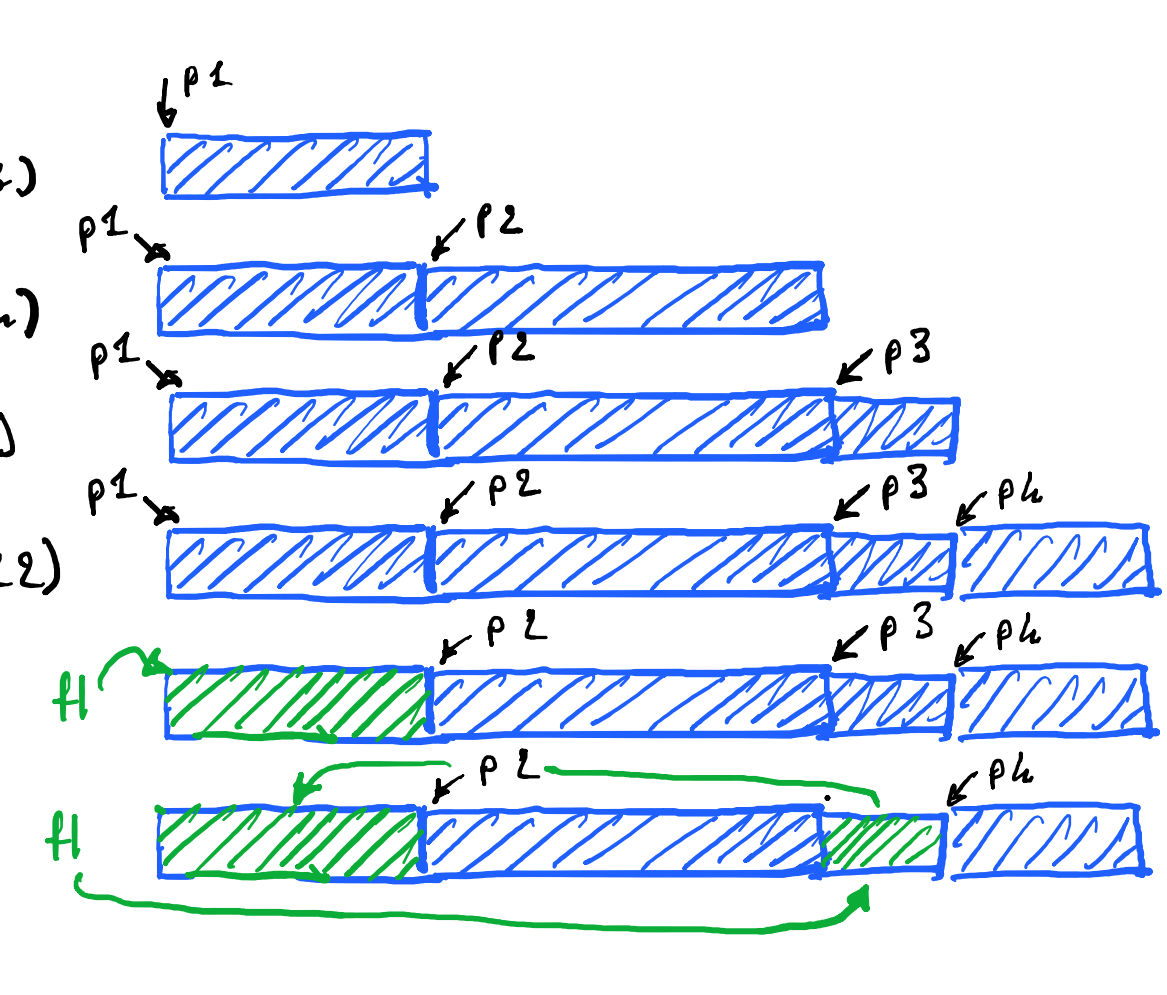
- Partizionamento di blocchi liberi: se un blocco è oggetto di allocazione per una sua parte ridotta, si genera un nuovo blocco con la parte rimanente



- Liste di blocchi liberi separate per dimensioni diverse
- Allineamento

Esempio:

p1 = malloc (16)
p2 = malloc (24)
p3 = malloc (8)
p4 = malloc (12)
free (p1)
free (p3)

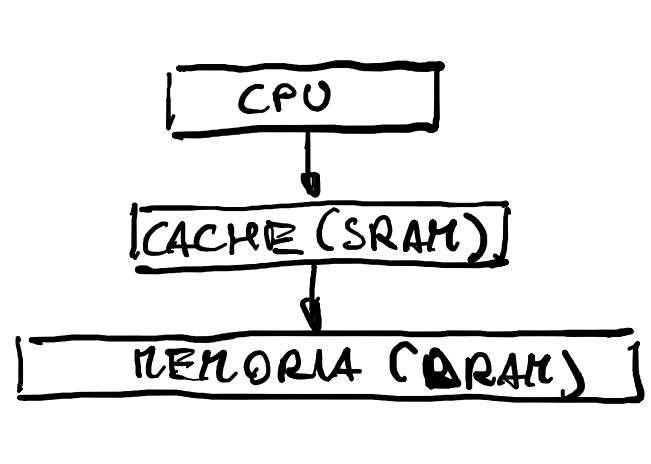

p1
p2
p3
p4
H
H

## CACHE ED USO OTTIMIZZATO DELLA MEMORIA

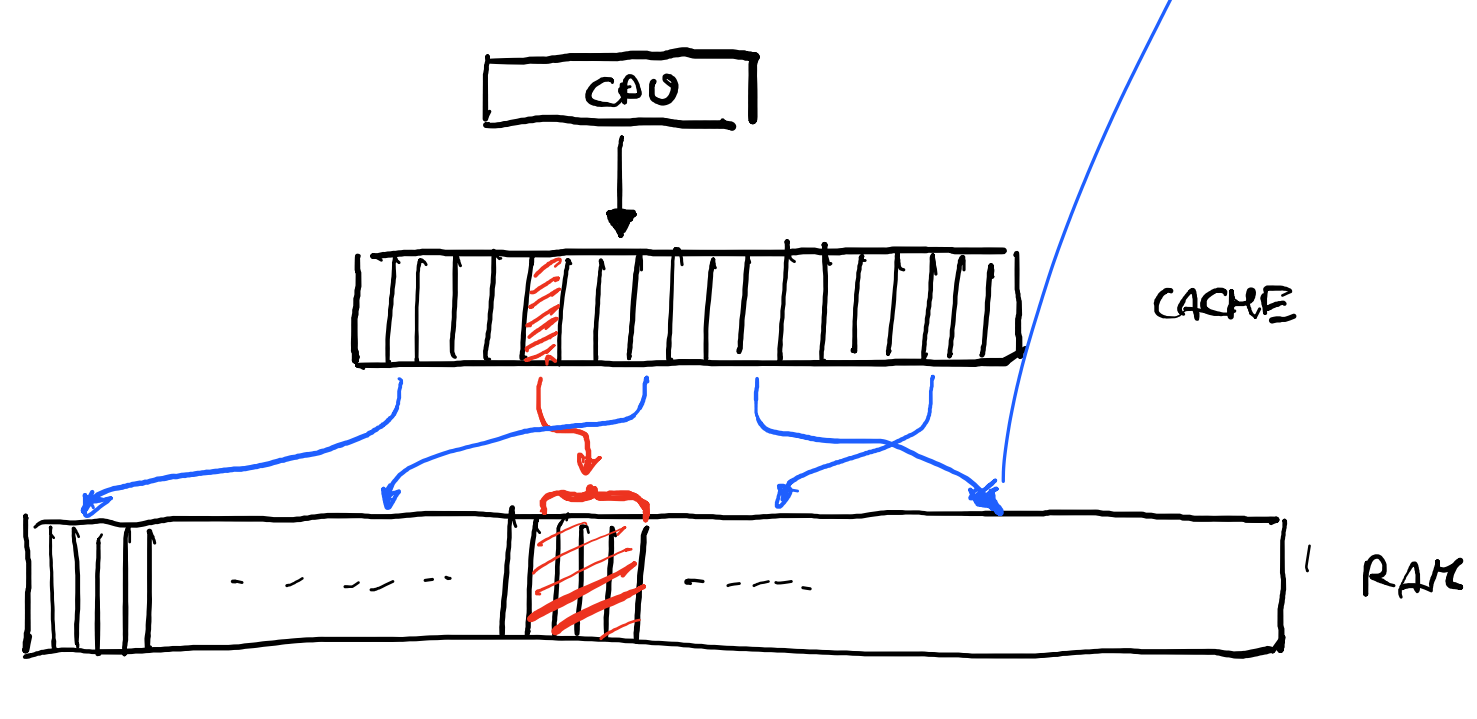
Consideriamo due operazioni

movl $1, %eax → ~1 nsec

movl $1, (%eax) → ~100 nsec


CPU
CACHE (SRAM)
MEMORIA (DRAM)

Una cache è costituita da un insieme di righe di dimensione fissa (es. 64 byte). Queste righe contengono una COPIA di parte dei dati in memoria


CPU
CACHE
RAM

Ogni indirizzo X in RAM è associato ad un blocco con identificativo $\lfloor X/L \rfloor$ dove L è la dimensione delle righe della cache.

Quando la CPU Tenta di accedere a X:

CACHE-HIT: la cache contiene in una delle sue linee una copia del blocco dove si trova x. Il dato viene letto dalla cache.

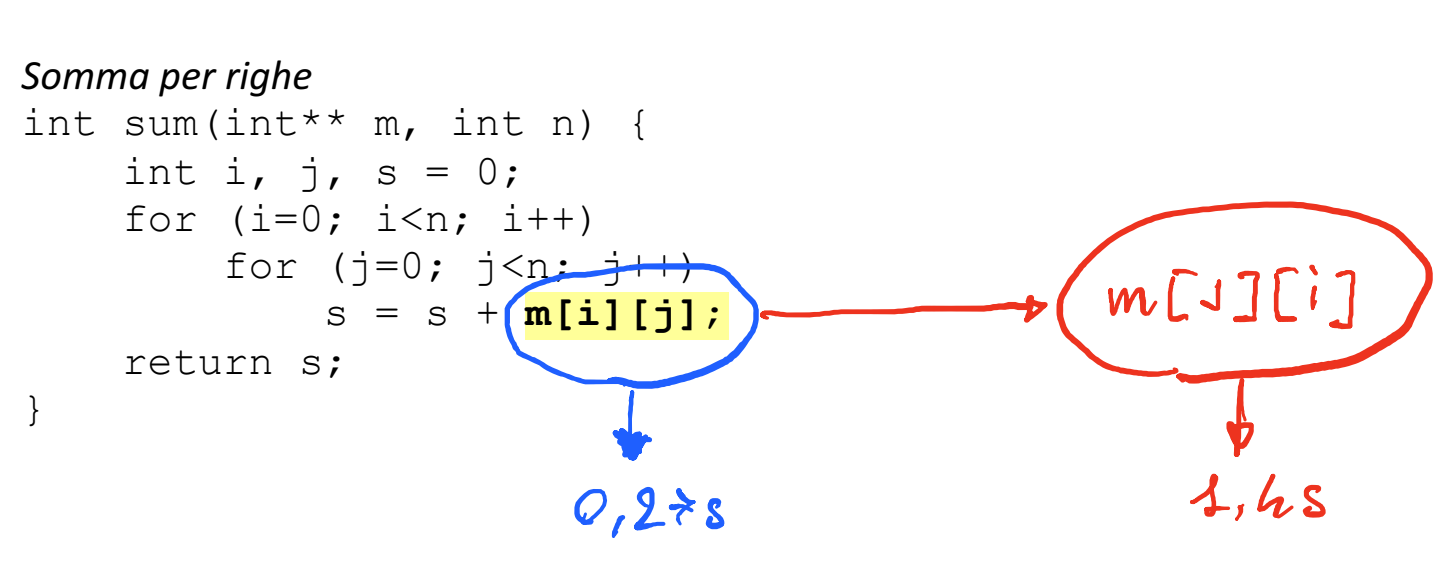
CACHE-MISS: il dato non è presente in cache. Il dato deve essere prima portato in cache con una lettura del blocco dalla RAM

Se la cache è piena, si seleziona una linea vittima.
La selezione si basa su principi di località:

LOCALITA' TEMPORALE: se un processo accede ad un indirizzo x è molto probabile che accederà di nuovo a x nel prossimo futuro

LOCALITA' SPAZIALE: se un processo accede ad un indirizzo x è molto probabile che accederà all'indirizzo x+1.

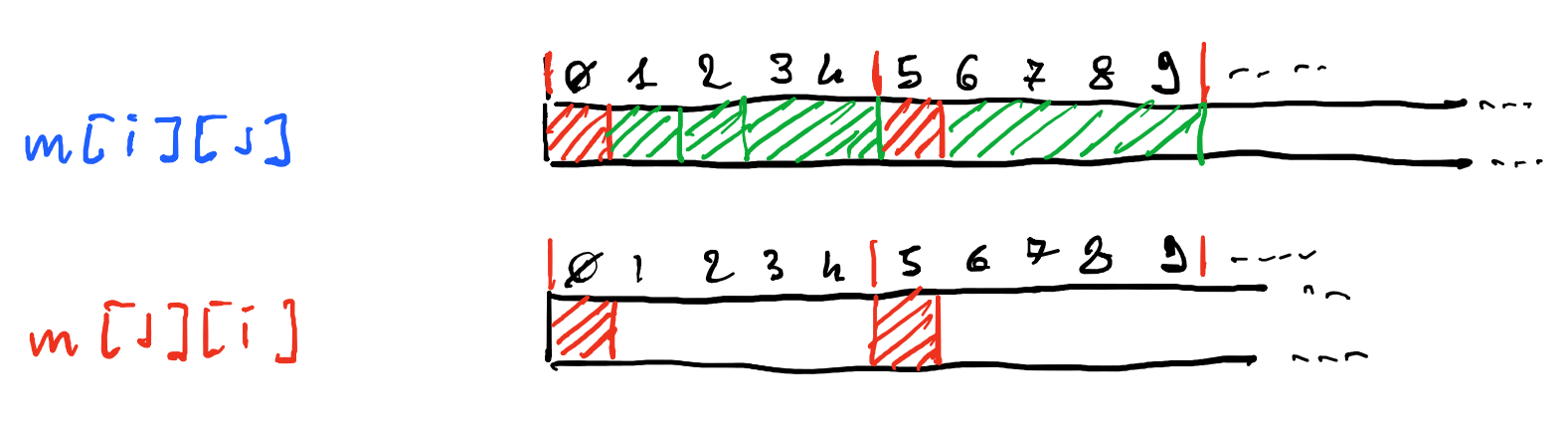
Esempio



*Somma per righe*

```
int sum(int** m, int n) {
    int i, j, s = 0;
    for (i=0; i<n; i++)
        for (j=0; j<n; j++)
            s = s + m[i][j];
    return s;
}
```

m[j][i]
0,27s
1,4s


m[i][j]
0 1 2 3 4 5 6 7 8 9
m[j][i]
0 1 2 3 4 5 6 7 8 9

$\frac{0,27}{1,4} \approx 5x$