memoria centrale

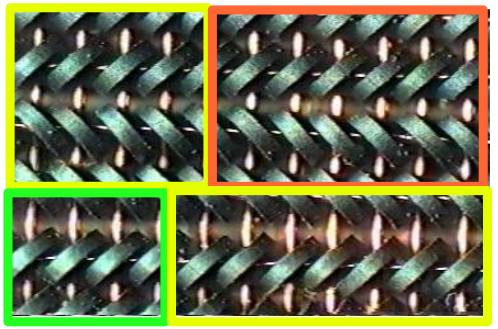
capitolo 8 del libro (VII ed.)

Cristina
Baroglio

Introduzione

- La memoria principale, come la CPU, è una risorsa condivisa fra i processi
- È necessario imporre dei meccanismi di gestione
- Alcuni metodi vengono messi a disposizione già a livello HW
- Altri metodi sono realizzati a un livello di astrazione superiore, fra questi:
  - paginazione della memoria
  - segmentazione della memoria
- La scelta del metodo da adottare dipende da diversi fattori, *in primis* l'architettura considerata

ripartiamo dal livello 0 (HW) e pian piano costruiamo le sovra-strutture necessarie



Core Memory - RAM - 1951

SOVRASTRUTTURA
suddivisione fra i processi

Livello Hardware

- RAM: unica memoria di grandi dimensioni direttamente accessibile alla CPU
- la CPU preleva le istruzioni da eseguire (operazione fetch) e i dati da elaborare (load) dalla RAM e memorizza in essa i risultati dell'elaborazione (store). Ciclo:
  - fetch: individua tramite program counter la prossima istruzione e caricala nell'instruction register
  - decode: decodifica l'operazione tramite un codice operativo
  - execute: individua e carica i dati richiesti ed esegui l'operazione
- **NB**: la CPU riceve ed utilizza un flusso di indirizzi di memoria, non sa come questi siano generati né a cosa servano
- Concentriamoci quindi per ora sui **singoli indirizzi**

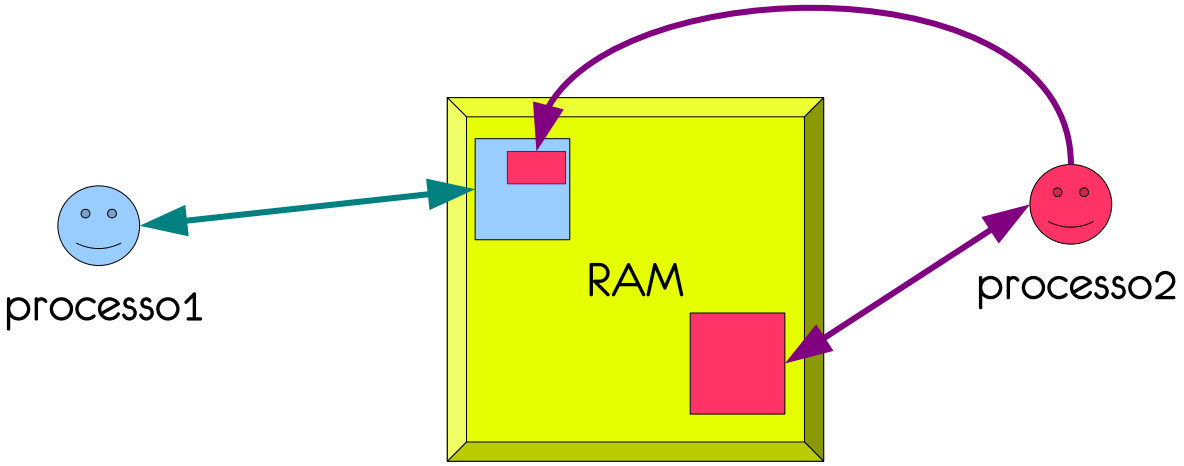
Schema generale: livello 0

sono incorporati nella CPU, ciò consente di effettuare una o più operazioni che usano solo i registri per ogni ciclo di clock

La RAM è accessibile via bus
talvolta l'accesso alla RAM richiede diversi cicli di clock durante i quali la CPU è in attesa

Per consentire un uso migliore della CPU spesso si frappone un buffer (cache) fra RAM e CPU. Scopo della cache: mediare la lentezza di interazione con la RAM mettendo a disposizione dati "probabilmente utili"

Memoria e processi



Processo 1 e Processo 2 usano ciascuno una porzione di RAM …

… e se per errore Processo 2 scrivesse un proprio dato sopra a un'istruzione di Processo 1?

Occorre attuare meccanismi di protezione

La RAM è condivisa da vari processi, per ogni processo la RAM contiene il suo codice (completo o parziale) e i dati da elaborare. È necessario far sì che un processo non vada a sovrascrivere il/i codice/dati di un altro (specie se quest'ultimo è un processo del SO!!!)

In altri termini occorre definire meccanismi che consentono di **assegnare a ogni processo un'area di memoria separata**

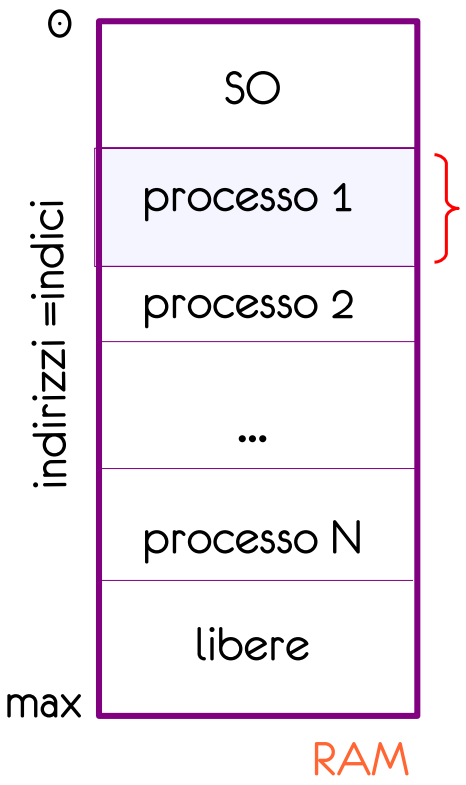
Processi e spazi degli indirizzi

- Un processo in esecuzione è caricato in RAM (codice, dati)
- L'esecuzione di un processo comporta la produzione di indirizzi (di istruzioni, di dati)
- La CPU produce un flusso di indirizzi che consentono al processo l'accesso a elementi contenuti in RAM
- Occorre controllare che un processo acceda solo agli indirizzi appartenenti alla porzione di RAM ad esso assegnata. Esempio di codice che produce errori di accesso in RAM:

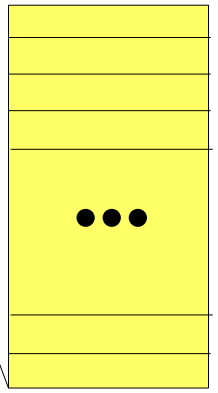
```
for (i=0 ;  ;  i++) mio_array[i] = 0;
```

- Vi sono diversi modi di realizzare il mapping processo-RAM, per cominciare consideriamo il più semplice

Registro base e limite 1/3



porzione di RAM del processo 1



indice *j* (indirizzo) della prima parola di memoria

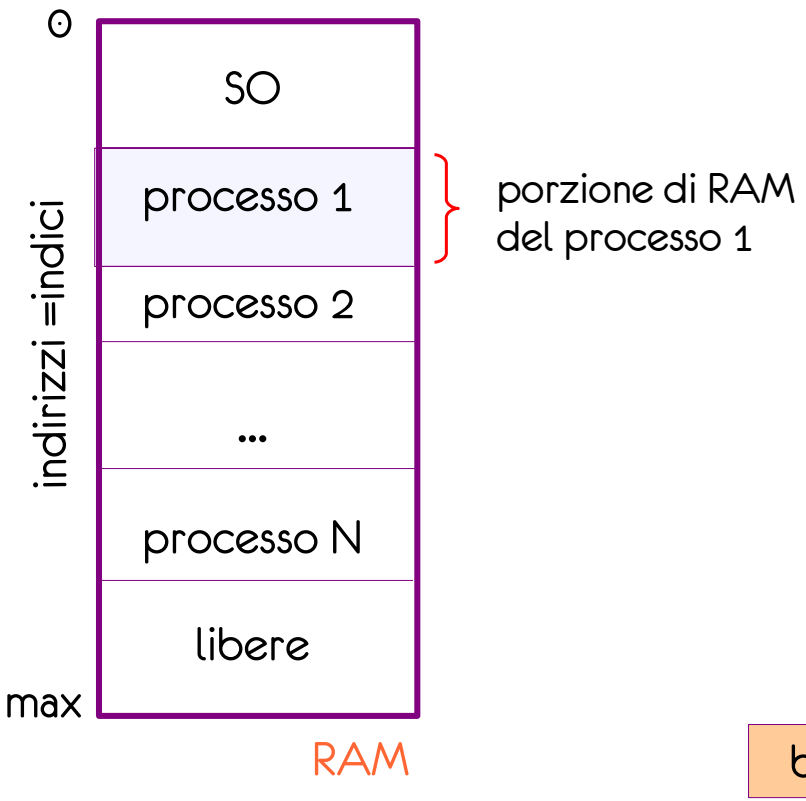
indice *k* (indirizzo) dell'ultima parola di memoria

Una porzione di RAM è, a basso livello, una porzione di un array

Per identificare una porzione di un array ho bisogno di due indici: l'indice del primo elemento e l'indice dell'ultimo elemento.

Posso conservare tale coppia di valori per ciascun processo

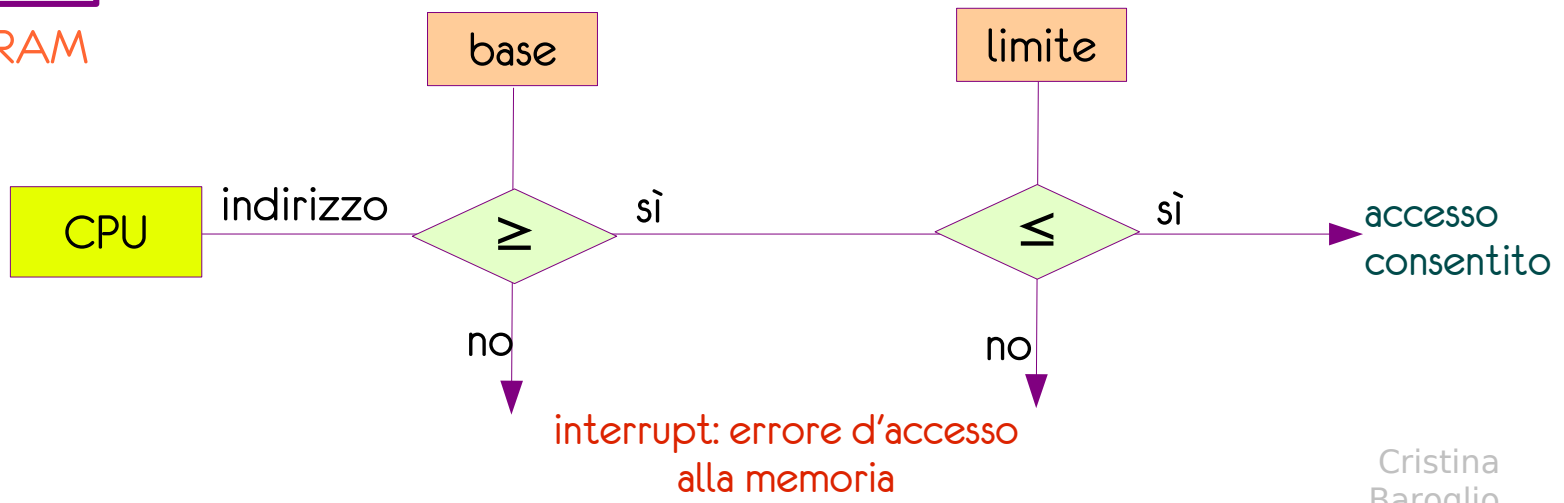
Registro base e limite 2/3



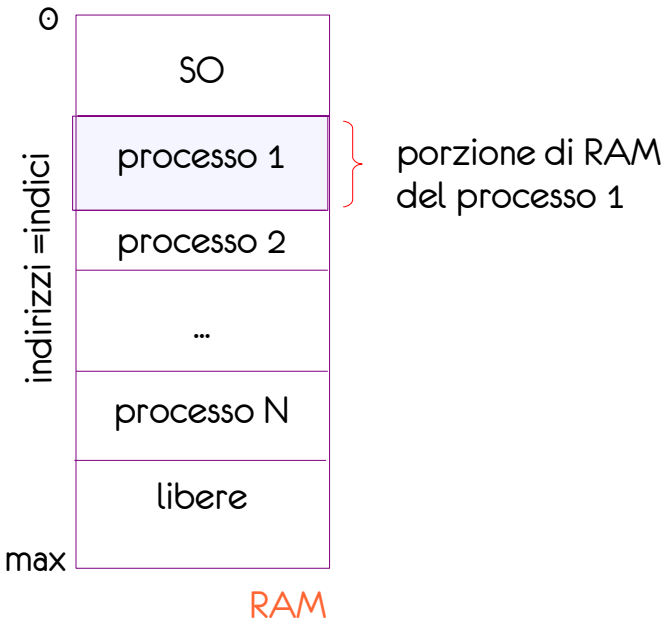
Il SO (e solo il SO) può eseguire un'istruzione che carica tali valori in due registri (registro base e registro limite)

Meccanismo di controllo:

Prodotto un indirizzo la CPU lo confronta con il registro base e il registro limite



Registro base e limite 3/3

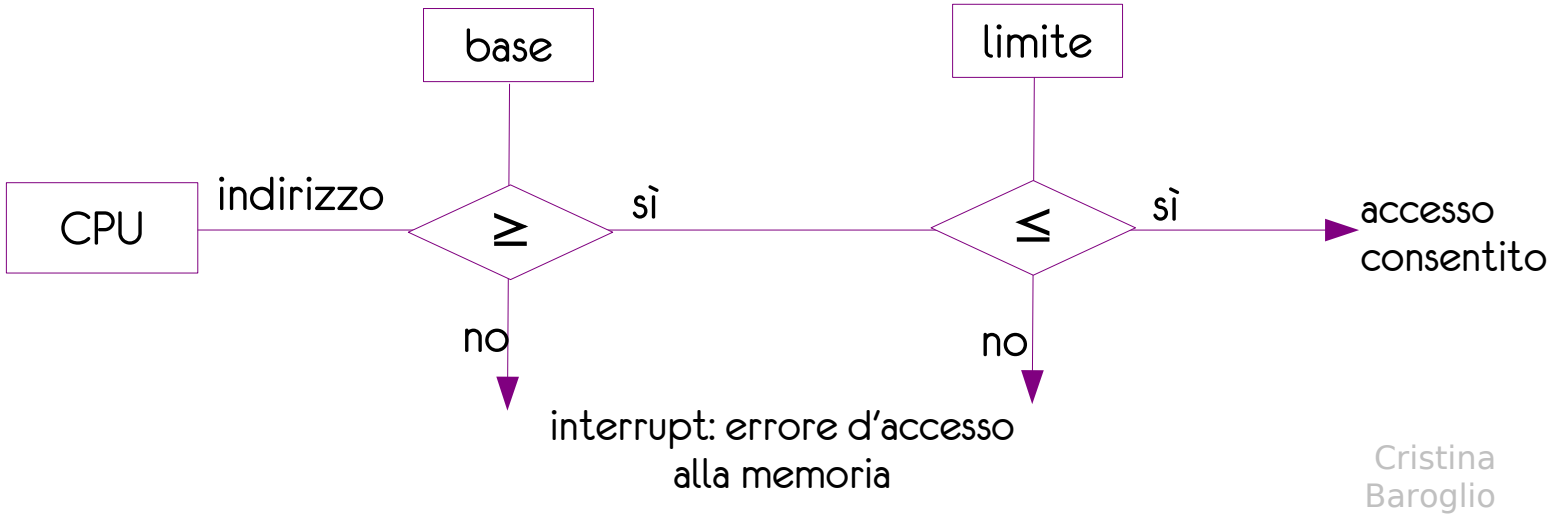


Il SO (e solo il SO) può eseguire un'istruzione che carica tali valori in due registri (registro base e registro limite)

A quale porzione di RAM può accedere il SO?
Tutta, perché per esempio si occupa di caricare in RAM i processi pronti da eseguire

Meccanismo di controllo:

Prodotto un indirizzo la CPU lo confronta con il registro base e il registro limite



Binding fra variabili e indirizzi

```
...
void inizializzaRandom(grafo *G) {
    int numN, i, j;
    printf("\n Quanti nodi vuoi creare
    scanf("%d", &numN);

    (*G).nodes = (nodo *) malloc(sizeo
    (*G).numN = numN;
    (*G).edges = (connessioni)malloc(s

    for (i=0; i<numN; i++) {
        (*G).nodes[i].dato = i; /
        (*G) nodes[i] visitato = 0;
                                co*)malloc(sizeof(arco) * numN);
    }

....
```

uso nomi di variabili

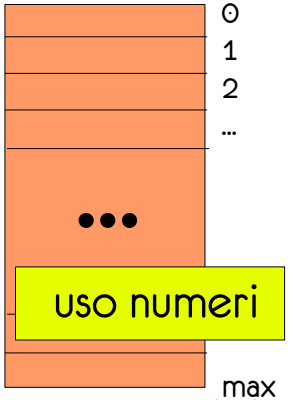
un processo esegue un programma scritto in un linguaggio di programmazione

all'interno del programma si dichiarano e si utilizzano delle **variabili** (i, j, G, numN, ...) e delle **procedure** (somma, cerca, ...)

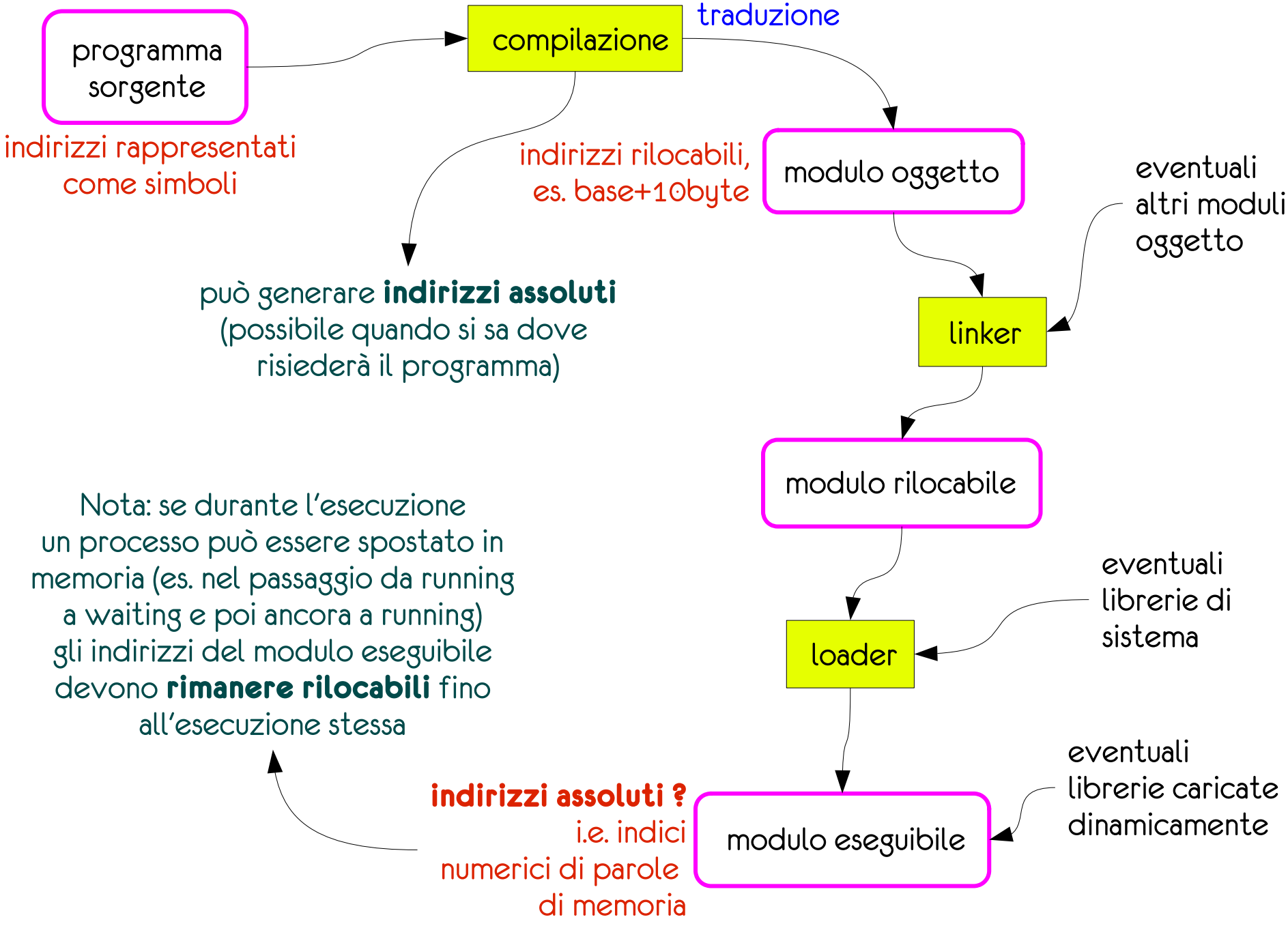
a livello di RAM si utilizzano degli indirizzi cioè degli **indici** di parole di memoria

quali passaggi ci sono nel mezzo?

e se ho più esecuzioni contemporanee dello stesso programma?



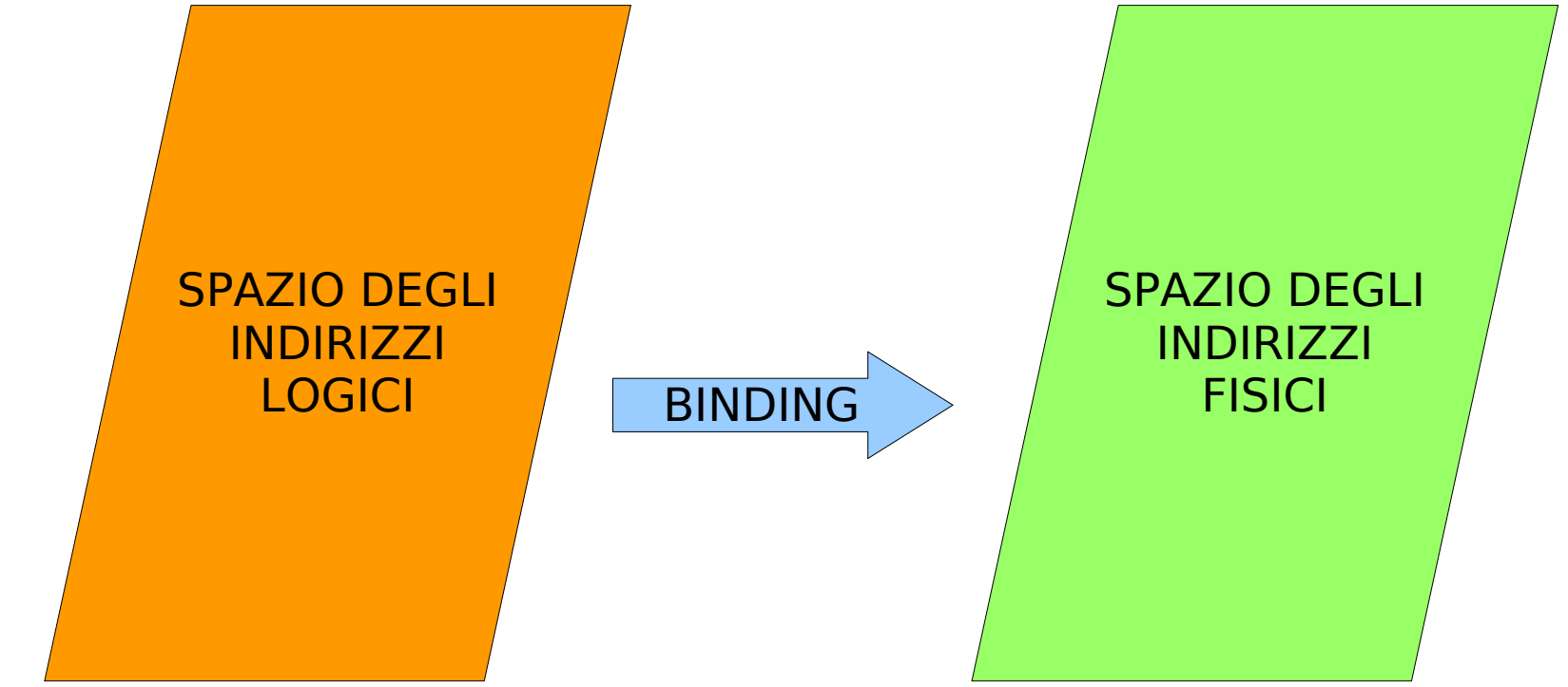
uso numeri


programma sorgente
compilazione
traduzione
indirizzi rappresentati come simboli
indirizzi rilocabili, es. base+10byte
modulo oggetto
eventuali altri moduli oggetto
può generare **indirizzi assoluti** (possibile quando si sa dove risiederà il programma)
linker
modulo rilocabile
Nota: se durante l'esecuzione un processo può essere spostato in memoria (es. nel passaggio da running a waiting e poi ancora a running) gli indirizzi del modulo eseguibile devono **rimanere rilocabili** fino all'esecuzione stessa
eventuali librerie di sistema
loader
indirizzi assoluti ? i.e. indici numerici di parole di memoria
modulo eseguibile
eventuali librerie caricate dinamicamente

Indirizzi logici e fisici

- un indirizzo prodotto dalla CPU è detto indirizzo logico
- d'altro canto ogni parola di memoria ha un proprio indirizzo fisico

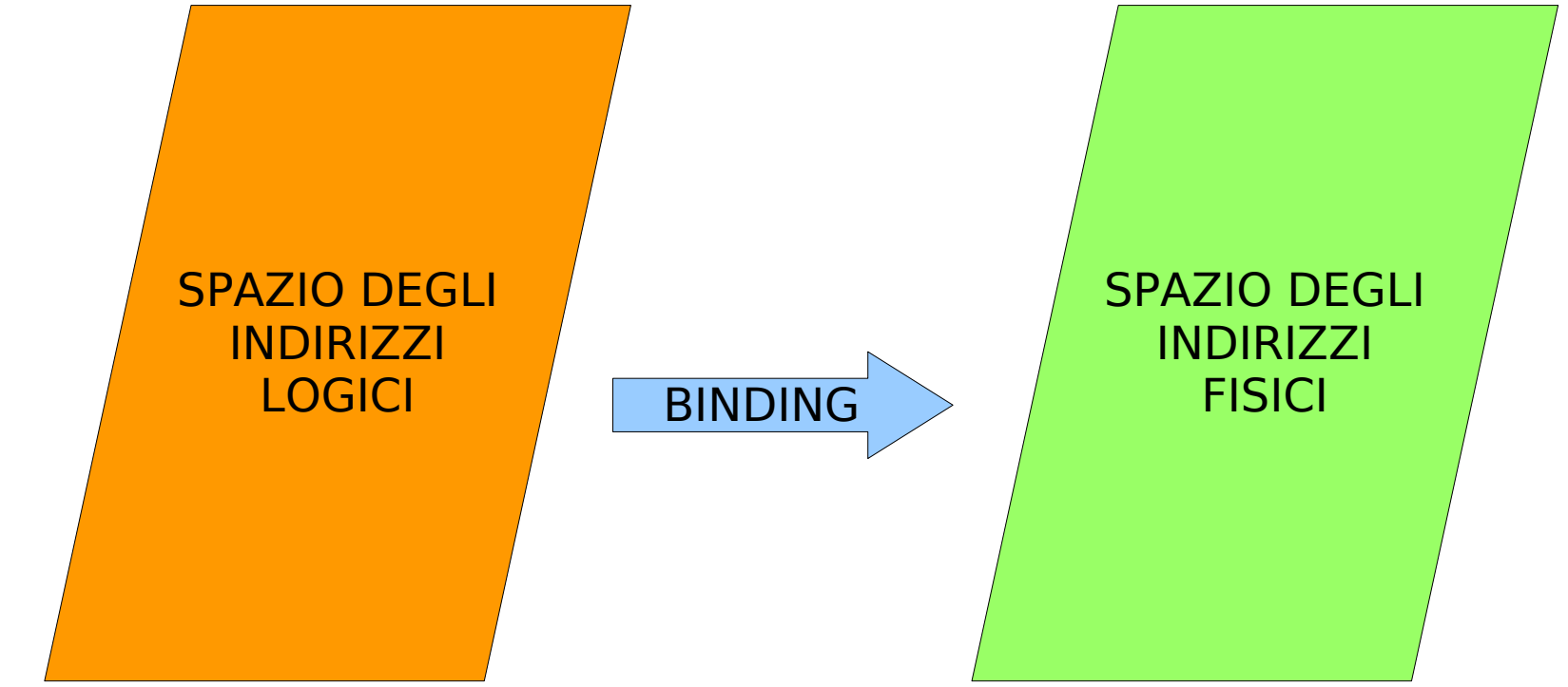
Spazi degli indirizzi di un processo



Indirizzi prodotti dalla CPU

Indirizzi effettivi in RAM

Spazi degli indirizzi di un processo

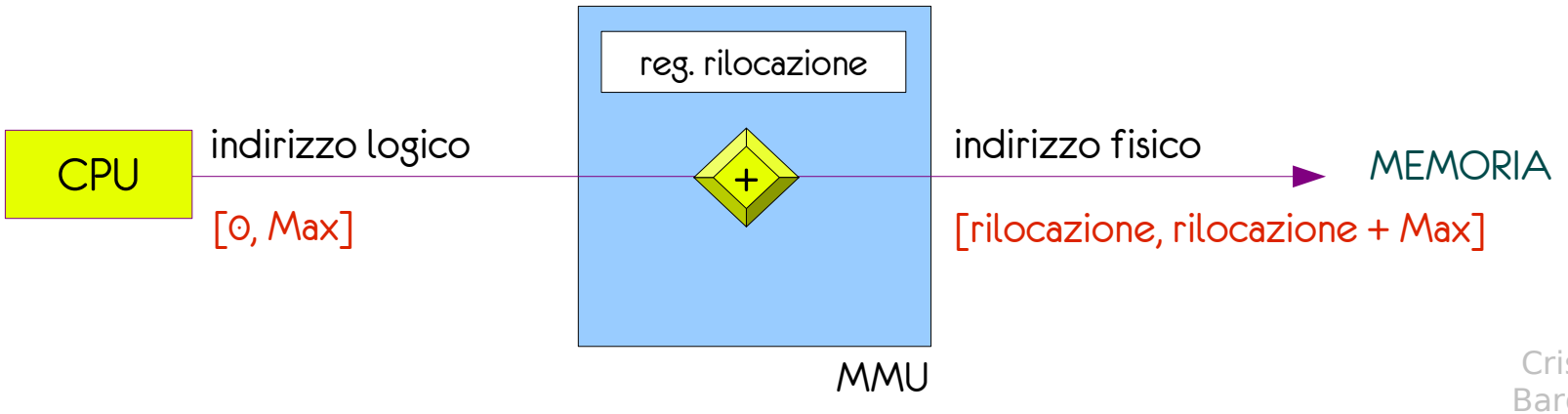


Indirizzi prodotti dalla CPU

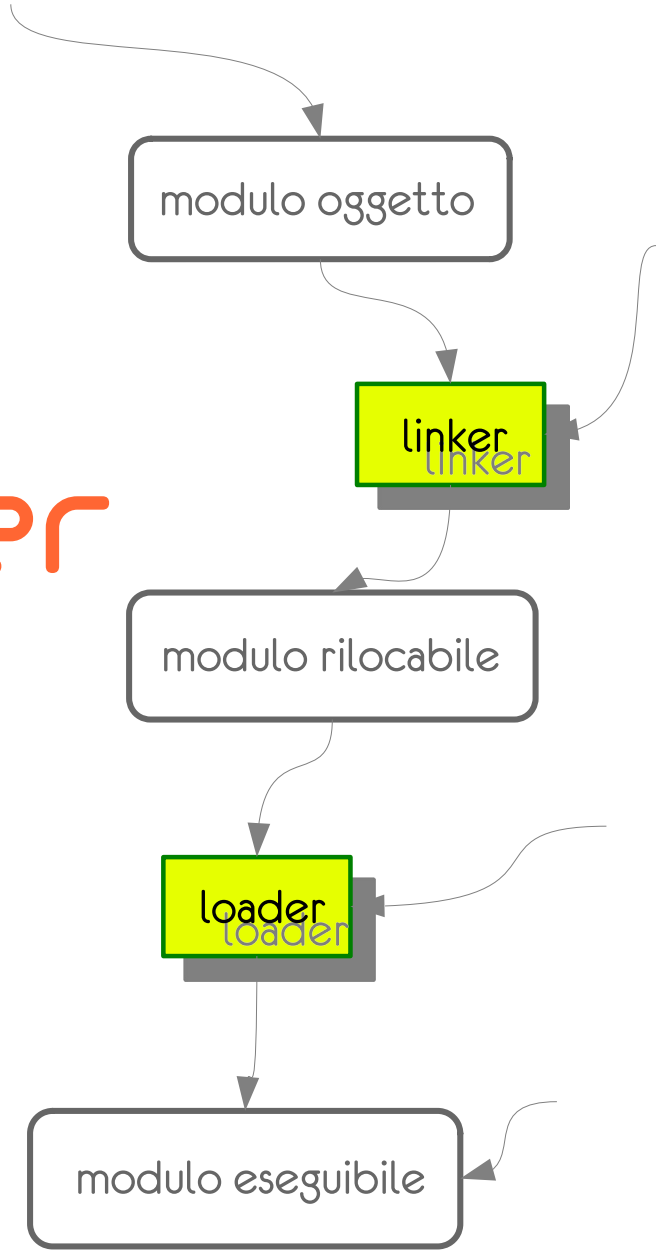
Indirizzi effettivi in RAM

Indirizzi logici e fisici: binding

- **Binding**: mapping dallo spazio degli indirizzi logici di un processo allo spazio dei suoi indirizzi fisici
- quando il binding viene fatto a tempo di compilazione o di caricamento:
  - indirizzo logico = fisico
- quando il binding viene fatto a tempo di esecuzione:
  - La corrispondenza deve essere calcolata: **lo spazio degli indirizzi logici ≠ dallo spazio degli indirizzi fisici**
- in questo caso il binding è a carico dell'MMU (memory management unit). L'MMU può essere realizzato in molti modi il più semplice è una generalizzazione del meccanismo basato sul registro base
- NB: la conversione è fatto SSE serve, cioè SSE si deve accedere alla memoria (lettura/scrittura)



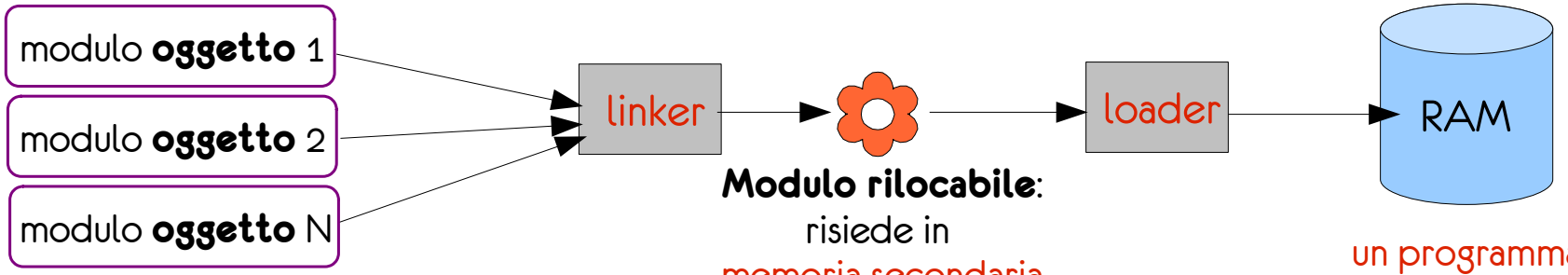
linker e loader



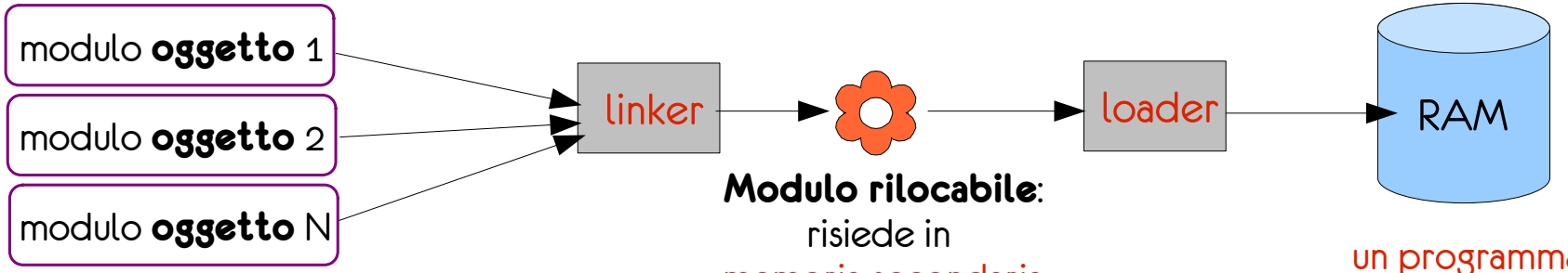
Cristina
Baroglio

Linking e loading

- linking: processo di composizione dei moduli che costituiscono un programma; associa ai nomi (di variabili o procedure), utilizzati da ciascun modulo e non definiti in esso, le corrette definizioni
- loading: copia un programma eseguibile (o parte di esso) nella RAM
- linking e loading sono statici quando precedono l'esecuzione
- linking e loading sono dinamici quando sono svolti durante esecuzione



Linking, loading e RAM

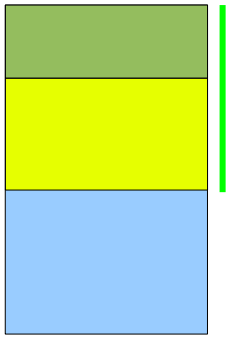


Modulo rilocabile:
risiede in
memoria secondaria

alcuni moduli corrispondono
a librerie di procedure

un programma
può essere ese-
guito solo se è
caricato in RAM

Approccio tradizionale



inserisco una copia del
codice di ogni libreria

eseguibile = composizione di
- file oggetto contenente il main
- file oggetti corrispondenti a librerie

L'intero file risultante è caricato in RAM:

1) una libreria può essere caricata **molte volte**, una copia per ogni programma che la include
2) carico anche le procedure **che non uso**

SPRECO !!!

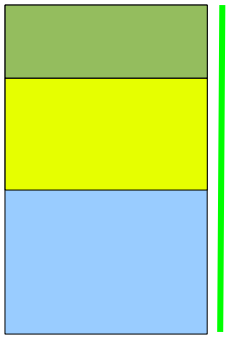
Linking e loading dinamici

- linking/loading: sono detti dinamici quando sono effettuati nella fase di esecuzione
- loading dinamico: una procedura è caricata in RAM quando occorre la sua prima invocazione (al suo primo utilizzo)
- linking dinamico: il collegamento del codice di una procedura al suo nome è effettuato alla sua prima invocazione. In questo caso il linker statico aggiunge solo uno *stub* della procedura in questione

Loading dinamico

- tutte le procedure risiedono in memoria secondaria sotto forma di codice rilocabile
- il codice di una procedura viene caricato nella RAM solo quando la procedura viene chiamata (per la prima volta)

- vantaggio rispetto a caricare un'intera libreria: si occupa meno RAM
- nota: occorre che il SO fornisca gli strumenti per realizzare librerie a caricamento dinamico



carico il codice di una procedura solo quando richiamata

In RAM solo una parte di programma:

- una procedura può essere caricata molte volte come parte di programmi diversi
- però carico solo le procedure che uso

eseguibile = composizione di
- file oggetto contenente il main
- rif. ai codici rilocabili

maggiore efficienza

Linking dinamico

- Rimanda il collegamento reale di una libreria alla fase di esecuzione
- dopo la compilazione, il linker statico arricchisce il “nucleo” del programma aggiungendo gli *stub* relativi alle procedure appartenenti alle librerie dinamiche usate
- **stub** = codice di riferimento, ha la seguente funzione
  - durante l'esecuzione, lo stub verifica se il codice della procedura è già stato caricato nella RAM:
    - se sì, sostituisce se stesso con l'indirizzo della procedura in questione
    - se no, causa il caricamento del codice della procedura e poi procede con la sostituzione come nel caso precedente
- **NB**: non importa se la procedura di libreria è stata caricata da un altro processo, tutti i processi fanno riferimento alla stessa copia del codice, la libreria risulta condivisa

Aggiornamento librerie condivise

- Il linking dinamico ha un notevole vantaggio:
- se aggiorno una libreria dinamica, automaticamente tutti i programmi che usano la libreria faranno riferimento alla nuova versione anche senza ricompilare (linking e compilazione sono spesso eseguiti entrambi dal compilatore)!
- se il linking fosse solo statico dovrei invece aggiornare il collegamento della libreria al programma, prima di utilizzare il medesimo

allocazione della ram

capitolo 8 del libro (VII ed.), da 8.3

Cristina
Baroglio

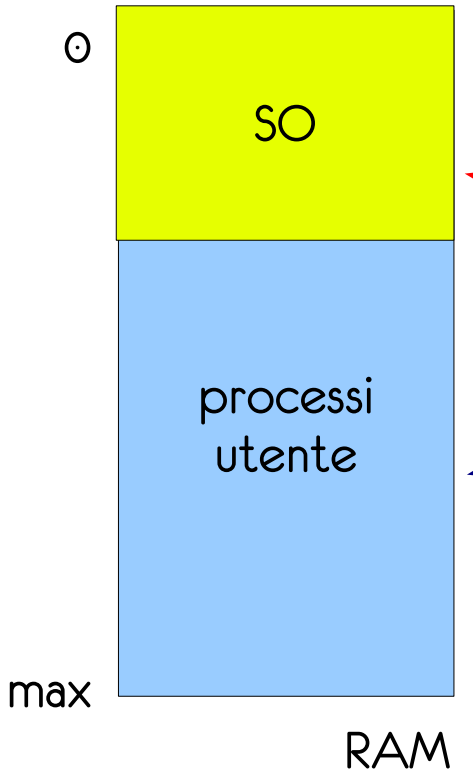
Sommario

- Lasciamo il Livello 0 per salire d'astrazione, passiamo al Livello 1
- Affronteremo ora il problema della gestione della memoria principale e, in particolare, della scelta dell'area da assegnare a un processo (quale memoria, quanta memoria, organizzata come)
- Tre approcci:
  1) Allocazione contigua
  2) Paginazione
  3) Segmentazione
- Ogni approccio fa riferimento a un modello di rappresentazione, che richiede apposite strutture dati e meccanismi di gestione

Allocazione contigua

- Allocazione contigua
  - rilocazione e protezione
  - partizioni multiple
  - frammentazione

Introduzione



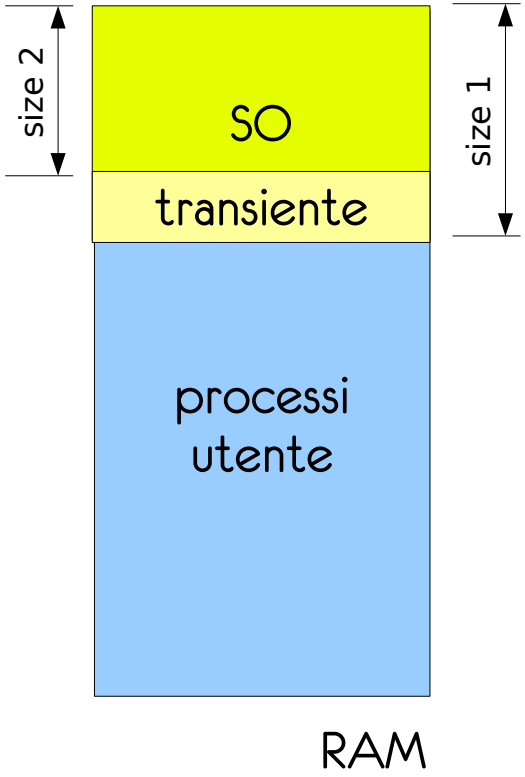
Nel modello ad allocazione contigua si suddivide la RAM in due parti:

- una riservata al SO, posizionata solitamente in memoria bassa (la posizione dipende dalla posizione del vettore delle interruzioni)
- l'altra riservata ai processi utente

occorre proteggere la partizione di memoria riservata al SO da letture/scritture ad opera di processi utente. Inoltre devo proteggere in modo analogo le aree di RAM riservate ai diversi processi utente

Ciò è facilmente realizzabile utilizzando un registro di rilocazione per la conversione di indirizzi logici in indirizzi fisici

Introduzione



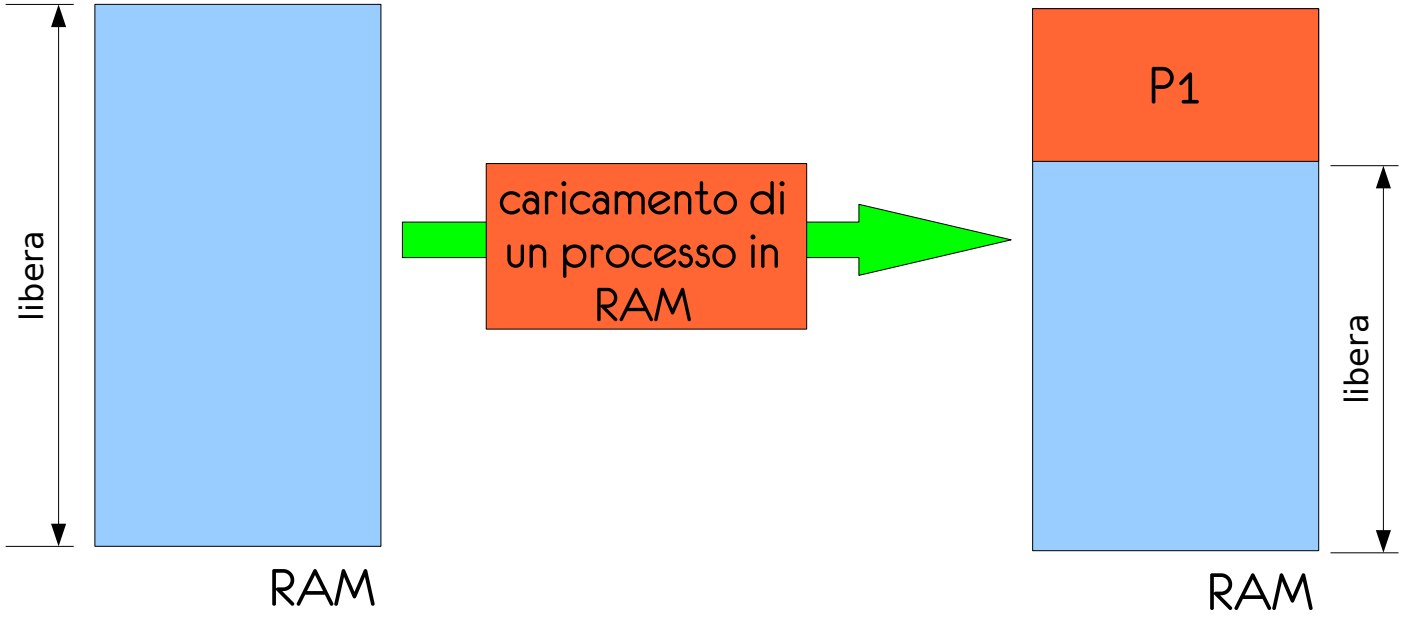
Il codice di SO caricato può a sua volta essere suddiviso in un nucleo di base sempre necessario e una parte che può essere utile o meno a seconda della circostanze (codice transiente).

Il codice transiente può essere rimosso dalla RAM quando non serve e aggiunto quando serve

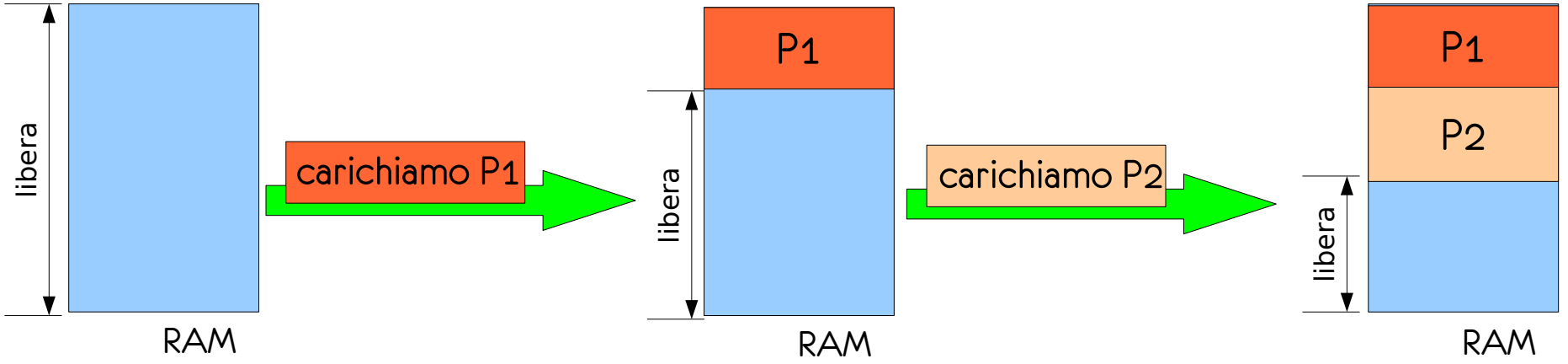
In queste circostanze occorre poter modificare la partizione riservata al SO

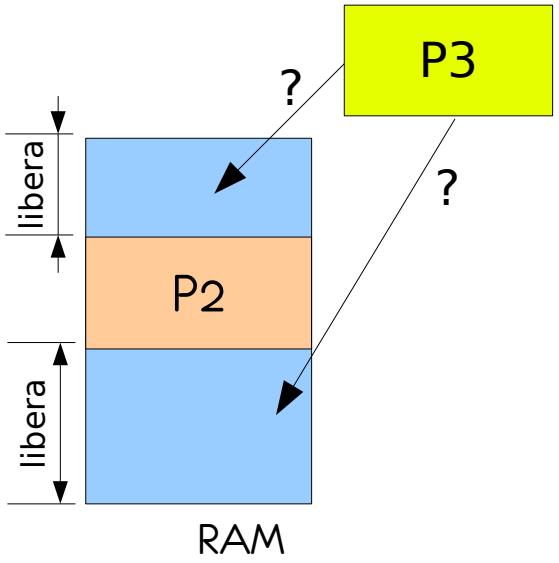
Allocazione a partizioni multiple

- Vediamo ora come funziona il meccanismo di allocazione della memoria ai processi nel modello ad allocazione contigua
- Lo schema seguito si chiama “a partizioni multiple”
- All'inizio tutta la RAM (esclusa la porzione per il SO) è libera:



Allocazione a partizioni multiple





P1 è terminato

In generale il SO deve mantenere una lista delle porzioni libere
Una porzione libera è detta "buco"

Un nuovo processo può essere caricato in RAM SSE esiste un buco abbastanza grande per contenere la sua immagine

In generale sarà possibile scegliere fra diverse opzioni ...

Criteri di scelta

- **Best-fit**: scelgo la porzione più piccola fra quelle adeguate a contenere l'immagine del processo
- **First-fit**: scelgo la prima porzione sufficientemente grande, trovata scandendo la lista dei buchi liberi
- **Worst-fit**: scelgo la porzione più grande fra quelle libere

Criteri di scelta

- **Qual'è la migliore?** Per capirlo occorre introdurre la nozione di frammentazione della memoria
- Di per sé per **frammentazione** si intende lo spezzettamento della memoria in tante parti. Si dice che si ha:
  - frammentazione esterna, se queste parti sono abbastanza grandi da essere utilizzabili (es. per contenere un processo)
  - frammentazione interna, se sono molto piccoli, praticamente inutilizzabili. In questo caso il frammento viene unito alla partizione precedente.

Criteri di scelta

- La frammentazione è un problema.
- L’analisi statistica mostra che con il first-fit, ogni N blocchi di memoria allocati si perde uno spazio pari a 0.5*N blocchi a causa della frammentazione (regola del 50%)
- In pratica 1/3 della memoria risulta inutilizzabile

RAM ALLOCATA RAM INUTILIZZABILE

Criteri di scelta

- In generale worst-fit è la strategia peggiore,
- First-fit e best-fit non sono sempre l'uno meglio dell'altro però computazionalmente first-fit è una tecnica meno costosa

Combattere la frammentazione

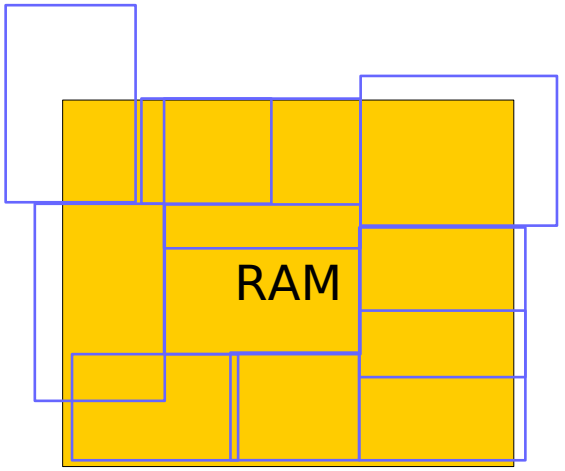
- È possibile combattere la frammentazione attuando di tanto in tanto una politica di compattamento: spostare le immagini dei processi in memoria dimodoché risultino contigue
- Il compattamento è applicabile solo se il binding fra indirizzi logici e fisici è effettuato a tempo di esecuzione

allocazione contigua, binding e rilocazione

Cristina
Baroglio

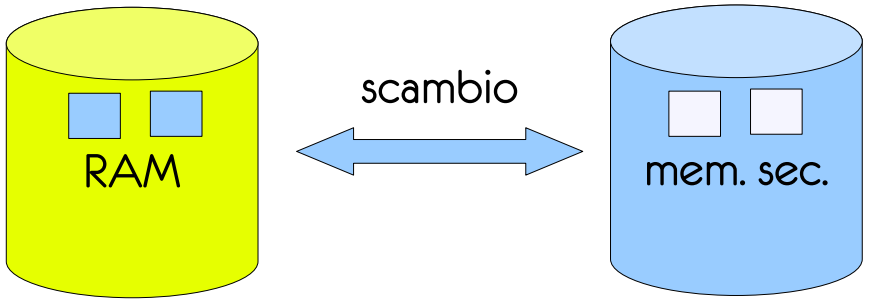
Swapping

- Scheduling di medio termine (già accennato)
- La RAM ha dimensione limitata
- Può succedere che i processi running e ready siano così tanti da richiedere complessivamente una quantità di memoria maggiore di quella offerta dalla RAM

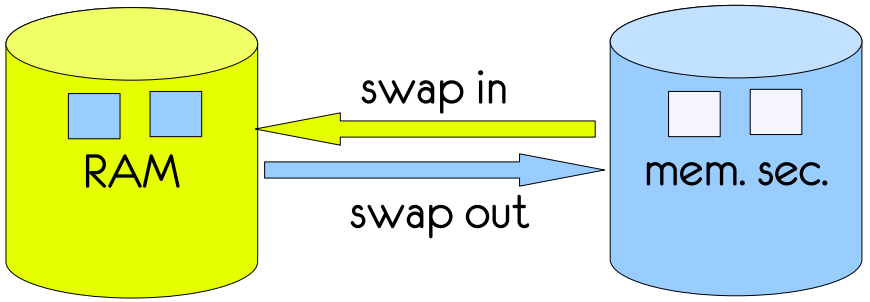


Swapping

- Scheduling di medio termine (già accennato)
- Soluzione: mantenere una parte dei processi ready in memoria secondaria ed effettuare di tanto in tanto lo swapping (lo scambio) fra processi in RAM e processi in memoria secondaria



Swapping



- swap in: carico l'immagine di un processo ready da memoria secondaria (anche detta backing store) in RAM
- swap out: scarico l'immagine di un processo che non è in esecuzione in memoria secondaria
- per motivi di efficienza è importante che i processi in testa alla ready queue siano conservati nella RAM, gli altri possono essere conservati in memoria secondaria

Swapping e binding

- L'immagine di un processo può fare swap-out e dopo un po' essere ricaricata in RAM (es. round-robin)
- In questo caso posso ricollocarla in una porzione qualsiasi della RAM?

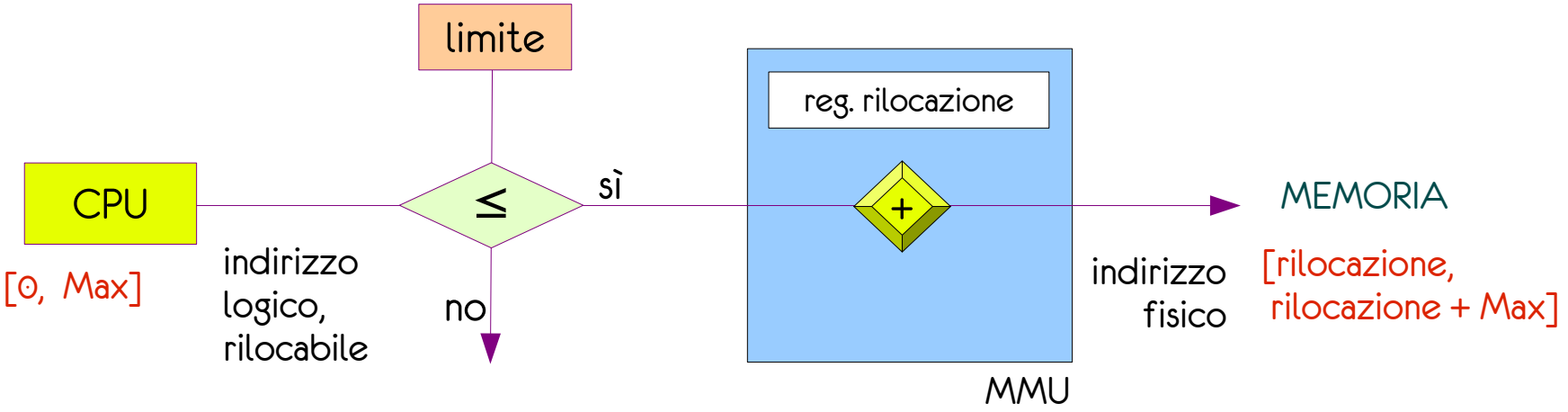
Swapping e binding

- La collocazione dipende dal **quando** viene effettuato il binding delle variabili:
  - se il **codice non è rilocabile** allora l'immagine del processo dovrà rioccupare la stessa sezione di RAM
  - se è **rilocabile** (in particolare, se il binding è dinamico) questo non è necessario

---

- Si può implementare la rilocazione se la RAM è gestita secondo il modello dell'allocazione contigua?
- Come avviene il binding?

Rilocazione e protezione



registro limite e registro di rilocazione vengono caricati durante il context switch
il contenuto del registro di rilocazione può variare nel tempo

L'uso dei registri limite e di rilocazione è possibile solo se l'HW dispone di queste strutture ⇒ la realizzazione dell'approccio a memoria contigua può essere realizzato solo previa presenza di un adeguato supporto HW

qualche dettaglio sullo swapping

Tempo di swapping

- Il tempo necessario al completamento dello swapping è dato dal tempo di swap-out + tempo di swap-in
- dipende dalla dimensione delle immagini dei processi coinvolti e dal tempo di trasferimento da/a memoria secondaria
- **Esempio**: se il tempo di trasferimento è pari a 1MB/sec, di quanto tempo ho bisogno per trasferire un processo con un'immagine da 100KB?

$$100\text{KB}/(1\text{MB}/sec) = (100\text{KB}/1000\,KB)\,sec = 0.1\,sec = 100\,msec$$

- di solito si usano i millisecondi come unità di misura
- ...

Tempo di swapping

- Supponendo identici tempo di swap-out e tempo di swap-in complessivamente occorreranno circa 200 msec
- il “circa” è dovuto al tempo necessario a posizionare la testina del disco

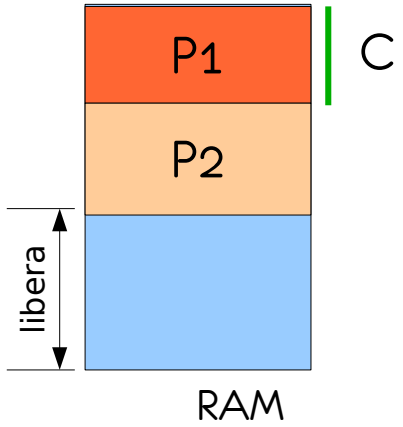
Commenti

- Un processo può essere oggetto di swapping SSE non ha in atto operazioni di I/O perché le operazioni di I/O non possono essere effettuate su variabili residenti in memoria secondaria
- **Esempi**
  - Unix (prime versioni): di base lo swapping era disabilitato, si attiva solo quando il carico del sistema è molto elevato
  - Windows 3.1: lo swapping era attivato solo a carico elevato ed era effettuato manualmente dall'utente

fine introduzione sulla gestione della RAM

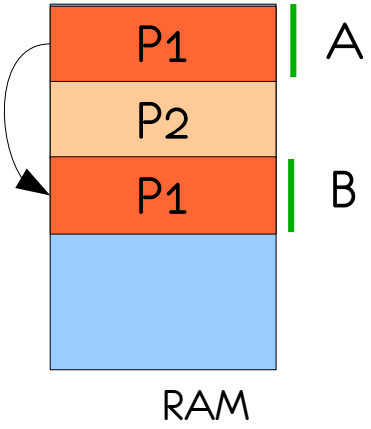
Paginazione della memoria

- La paginazione è un meccanismo di gestione della RAM alternativo all'allocazione contigua
- detto “**spazio degli indirizzi di un processo**” l'insieme di tutti gli indirizzi a cui il processo ha accesso, carattestica fondamentale della paginazione è che essa consente allo spazio degli indirizzi fisici di un processo di non essere contiguo



allocazione contigua

sp. indirizzi di P1 = C



paginazione

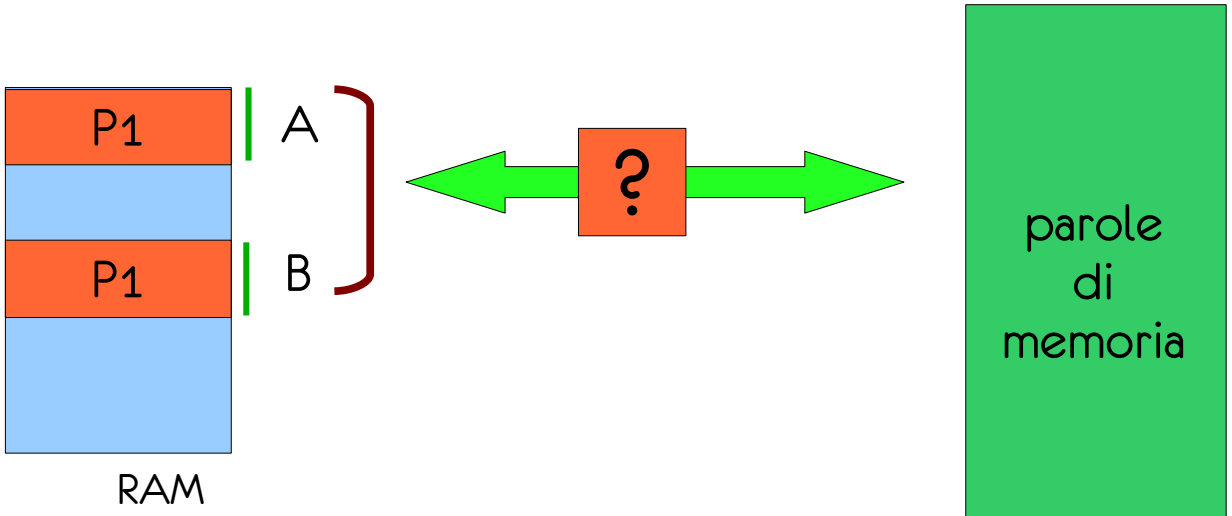
sp. indirizzi di P1 = A ∪ B

È UN VANTAGGIO?

Paginazione

- È un vantaggio
- Consente di far (de)crescere in modo dinamico lo spazio riservato a un processo, semplicemente (togliendo) aggiungendo delle pagine
- Quindi per esempio posso mantenere in RAM solo una porzione del codice di un processo, aggiungendo via via altre parti utili, con riferimento all'esecuzione corrente
- La gestione del codice transiente del SO diventa molto più semplice e naturale
- Paginazione e architettura di una macchina sono strettamente correlate: la paginazione è possibile solo avendo un opportuno supporto hardware
- Vediamo ora le strutture necessarie per realizzare questo modello ...

Pagine e strutture di supporto



paginazione

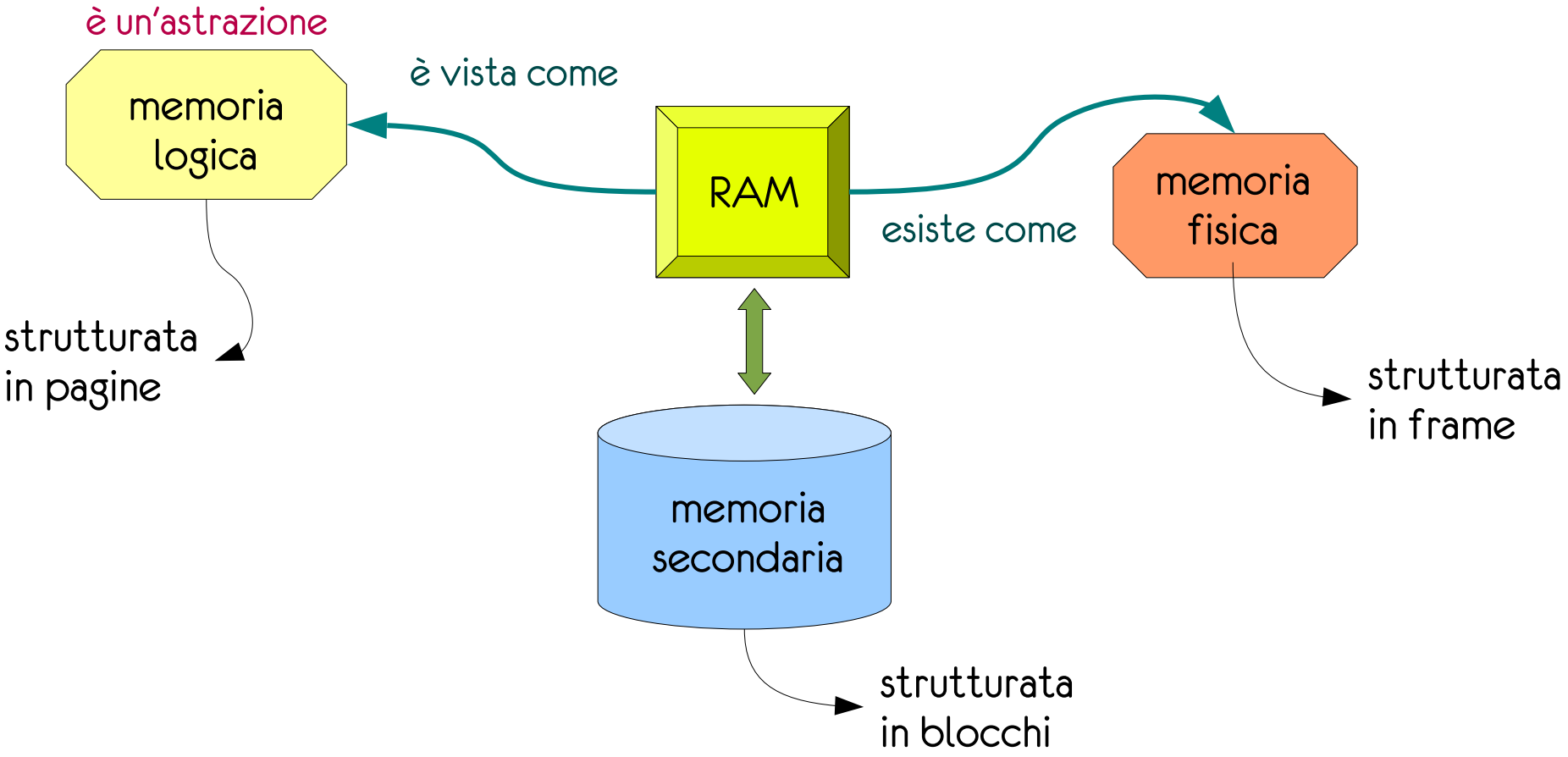
sp. indirizzi di P1 = A ∪ B

(1) Come posso associare porzioni di processo a porzioni di RAM?

(2) Come posso organizzare le diverse porzioni in un tutt'uno?

Vedere la RAM come array non è più così comodo ...

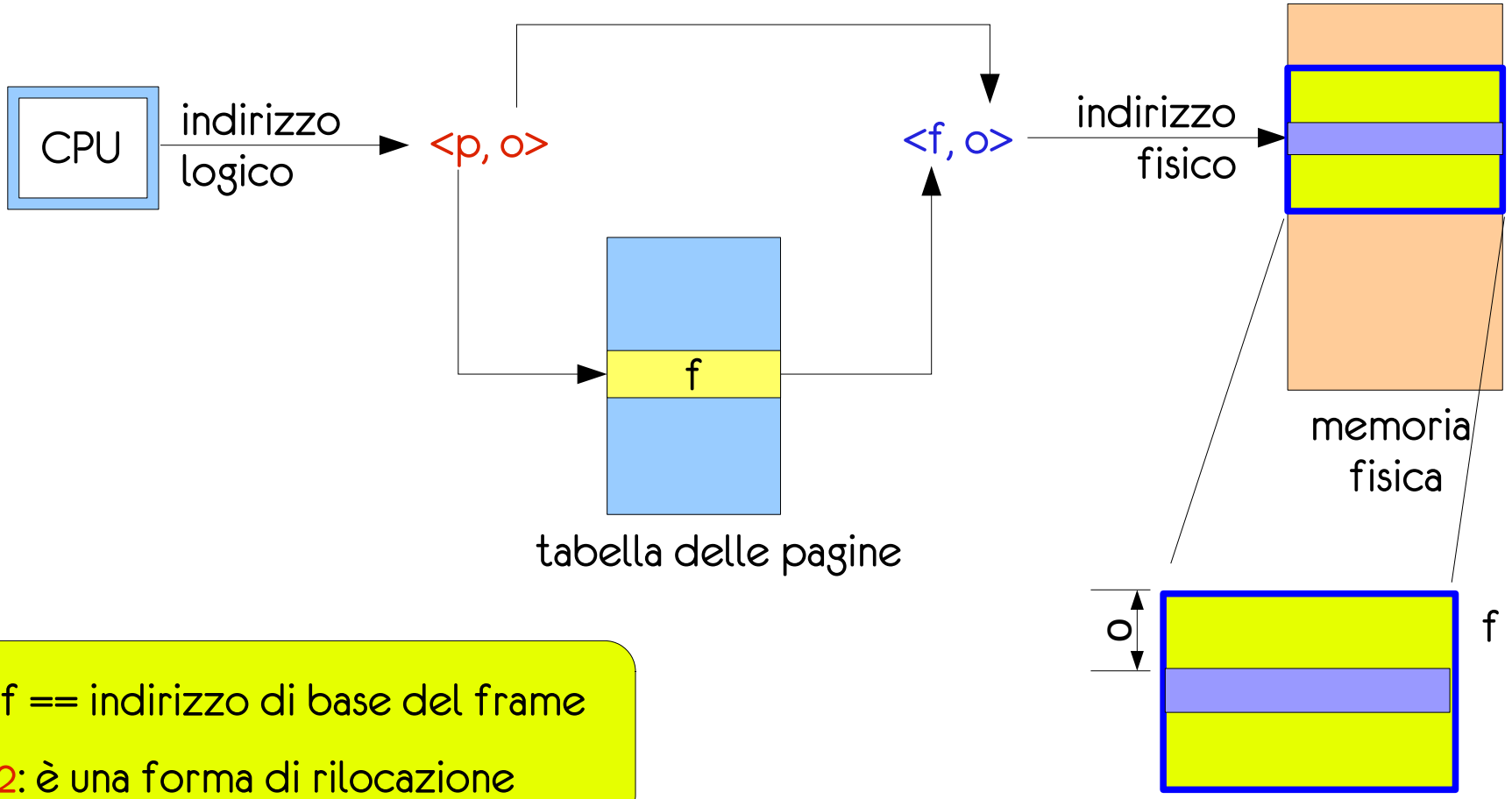
Pagine, frame e blocchi



blocchi, frame e pagine sono tutti termini che fanno riferimento a porzioni di memoria di uguali dimensioni ma appartenenti a viste/elementi diversi

Pagine e frame

Il primo tipo di struttura di cui abbiamo bisogno serve a fare il binding fra indirizzi logici e indirizzi fisici. In questo contesto un indirizzo logico è una coppia <numero_di_pagina, offset>, un indirizzo fisico è una coppia data a <id_frame, offset>



Nota: f == indirizzo di base del frame

Nota 2: è una forma di rilocazione

Indirizzi logici

- La dimensione è la stessa per tutte le pagine ed è definita dall'architettura; è una potenza di 2 normalmente compresa fra 512 byte e 16 MB
- Supponiamo che la dimensione di una pagina sia 2^n e la dimensione della memoria logica sia 2^m, in quante pagine sarà suddivisa la memoria logica?

$$2^m / 2^n = 2^{m-n}$$

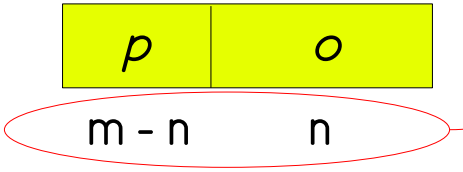
- quanti bit servono per rappresentare un numero di pagina?

$$m - n$$

- quanti bit occorrono quindi per rappresentare lo scostamento all'interno di una pagina?

$$n$$

- quindi un indirizzo logico:



Nota: è giusto che la somma faccia m?

Indirizzi logici: esempio

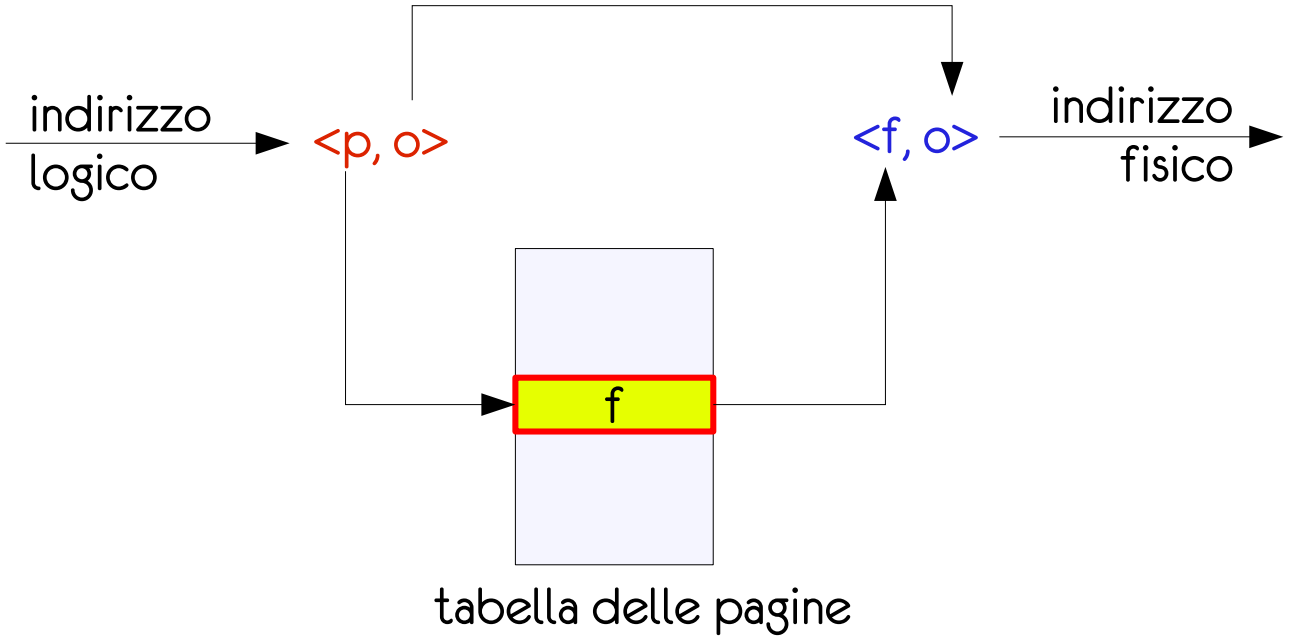
- Supponiamo di avere pagine di dimensione 512 byte e una RAM di dimensione 1MB, quante pagine avremo? Quanti bit occorrono per rappresentare indirizzi logici in questo contesto?
- Innanzi tutto devo riportarmi a potenze di 2 nella stessa unità di misura:
  - pagina: 512 byte = 2^9 byte
  - memoria logica: 1MB = 2^{20} byte
- Ora possiamo fare i conti:
  - numero di pagine necessarie: $2^{20} / 2^9 = 2^{20-9} = 2^{11}$
  - per rappresentare il numero di pagina occorrono 11 bit
  - per rappresentare l'offset occorrono): 9 bit
- num bit per le pagine + num bit x l'offset = 11 + 9 = 20

Paginazione e rilocazione

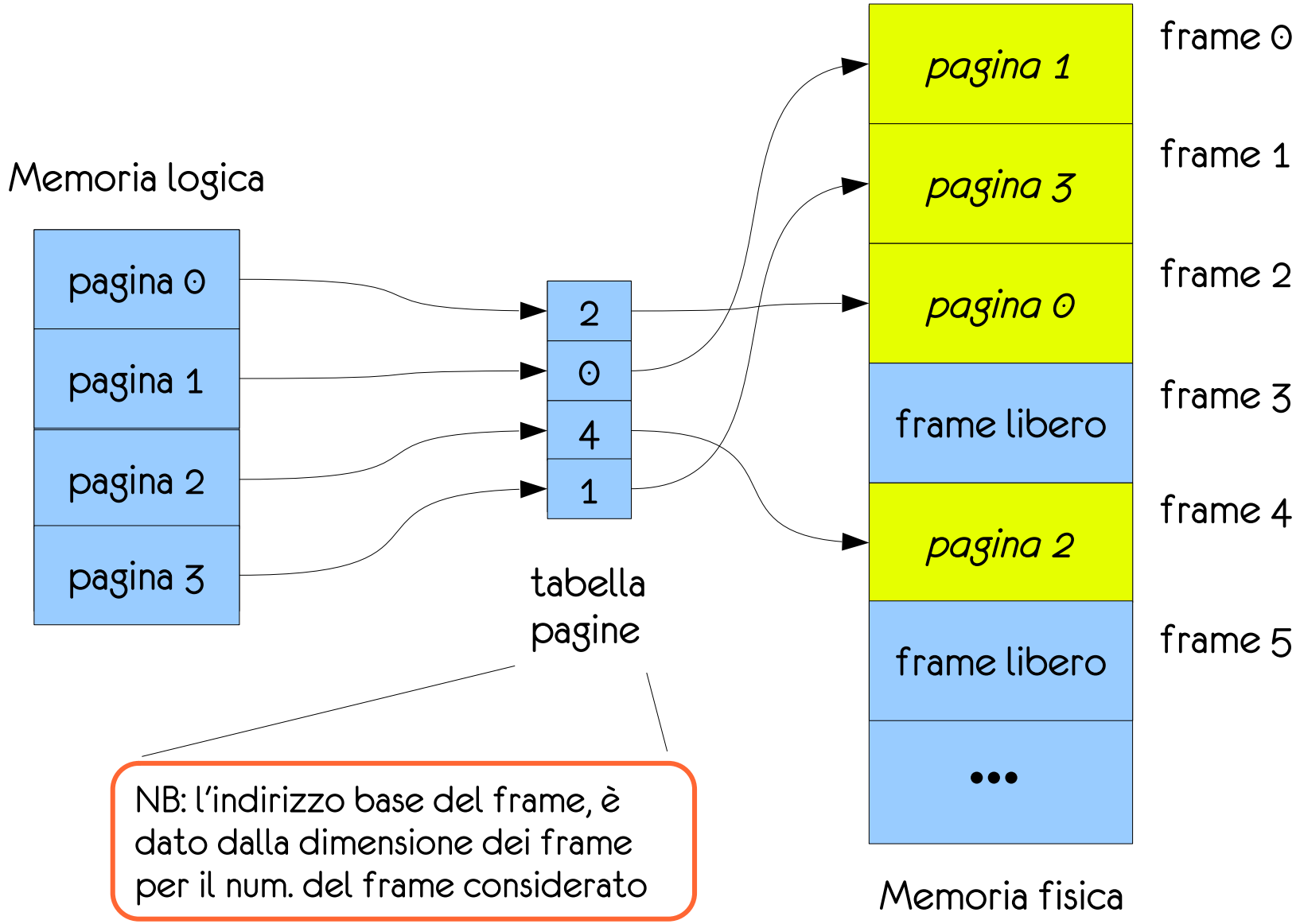
- Cosa vuol dire “rilocare il codice” in questo contesto?

consentire l'accesso a una pagina, indipendentemente dal frame in cui è caricata

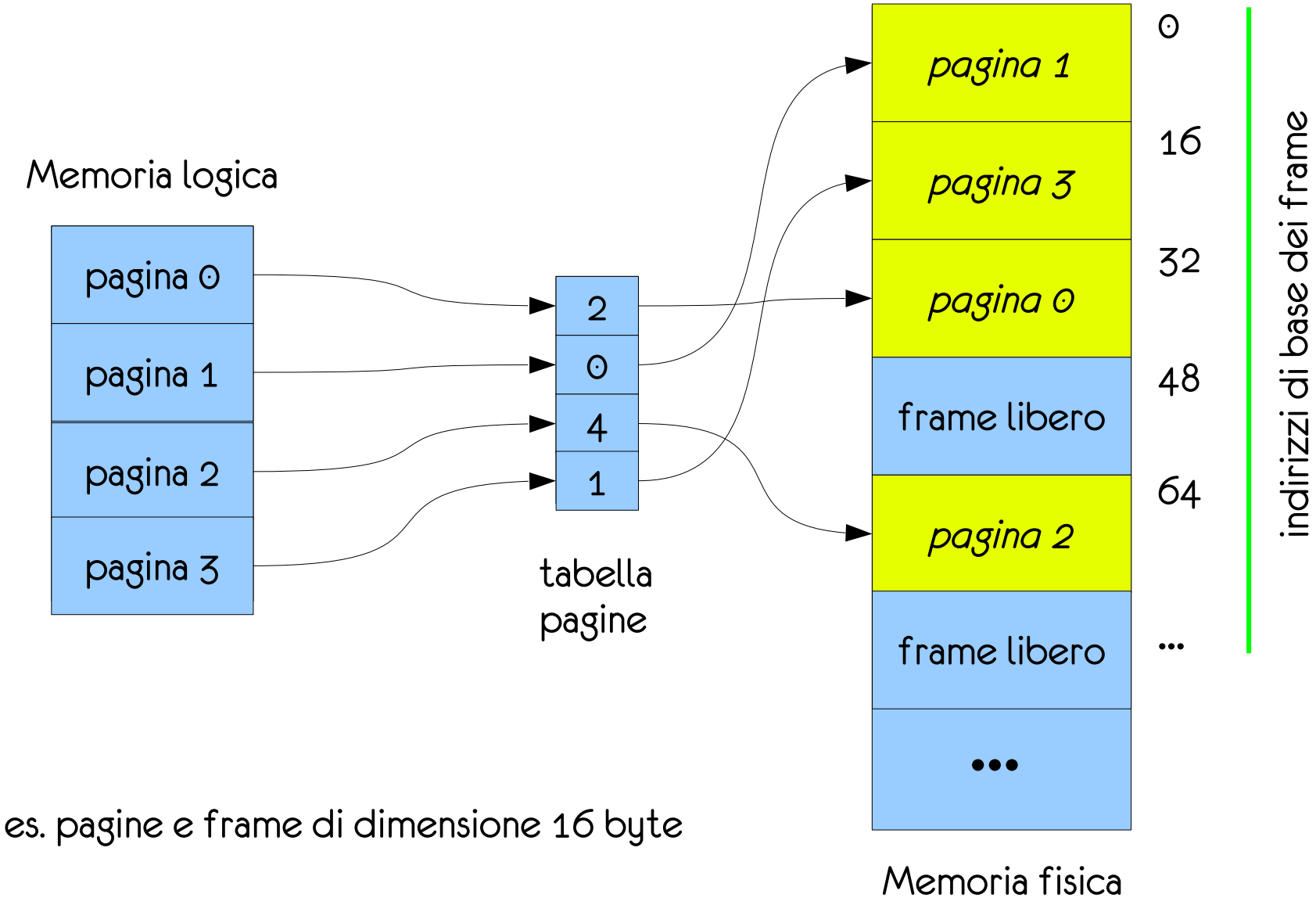
- Si può realizzare la rilocazione? Sì, il registro di rilocazione è sostituito dalla entry nella tabella delle pagine, corrispondente a p. Il valore f individua l'indirizzo di inizio del frame



Esempio di paginazione

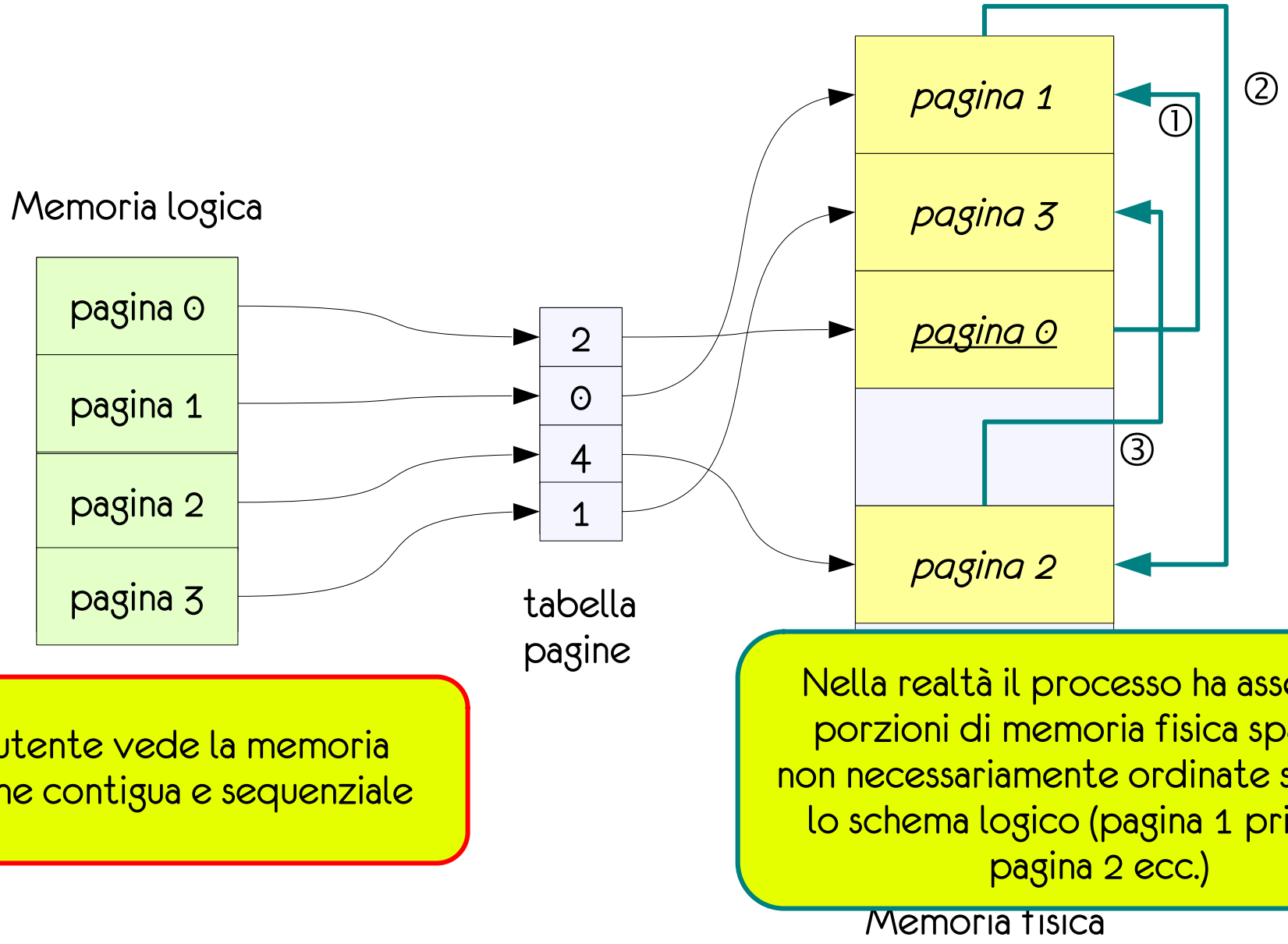


Esempio di paginazione



es. pagine e frame di dimensione 16 byte

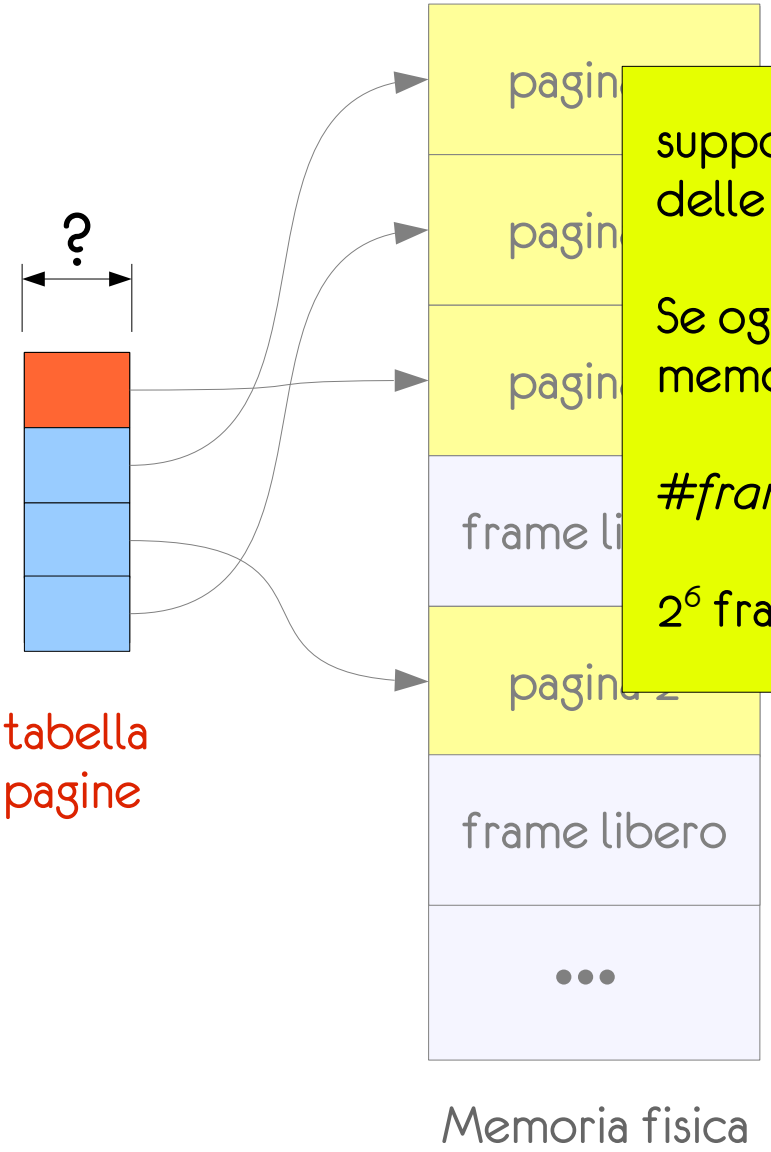
Esempio di paginazione



L'utente vede la memoria come contigua e sequenziale

Nella realtà il processo ha assegnate porzioni di memoria fisica sparse e non necessariamente ordinate secondo lo schema logico (pagina 1 prima di pagina 2 ecc.)

Qualche conto



supponiamo che un elemento della tabella delle pagine sia grande 4 byte

Se ogni frame ha dimensione 4 KB, a quanta memoria fisica riusciamo ad accedere?

*#frame indirizzabili * dimensione di un frame*

2^6 frame * 4 KB = 2^2 * 2^6 KB = 2^8 KB = 256 KB

Paginazione e frammentazione

- La paginazione elimina il problema della frammentazione esterna però permane il problema della frammentazione interna
- Ogni processo può avere allocate un numero di pagine diverso, quante di queste presenteranno frammentazione interna?
- Soltanto l'ultima perché raramente la dimensione di un processo sarà un multiplo della dimensione di una pagina, quindi in generale avrò bisogno di *N pagine più un pezzetto* per ciascun processo
-

Paginazione e frammentazione

- **Frammentazione interna**: in media possiamo dire che avremo ½ pagina non utilizzata per ogni processo
- Dimensione ottimale delle pagine: occorre trovare un compromesso fra limitare il problema della frammentazione e ridurre i costi dell'I/O:
  - **pagine piccole**: riducono la frammentazione
  - **pagine grandi**: migliori quando occorre trasferire da/a memoria secondaria grosse quantità di dati (carico una pagina invece di tante in sequenza)

Indirizzi logici

- La dimensione è la stessa per tutte le pagine ed è definita dall'architettura; è una potenza di 2 normalmente compresa fra 512 byte e 16 MB
- Supponiamo che la dimensione di una pagina sia 2^n e la dimensione della memoria logica sia 2^m, in quante pagine sarà suddivisa la memoria logica?

$$2^m / 2^n = 2^{m-n}$$

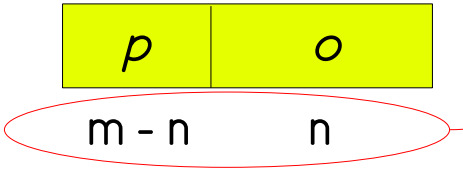
- quanti bit servono per rappresentare un numero di pagina?

$$m - n$$

- quanti bit occorrono quindi per rappresentare lo scostamento all'interno di una pagina?

$$n$$

- quindi un indirizzo logico:



Nota: è giusto che la somma faccia m?

Indirizzi logici: esempio

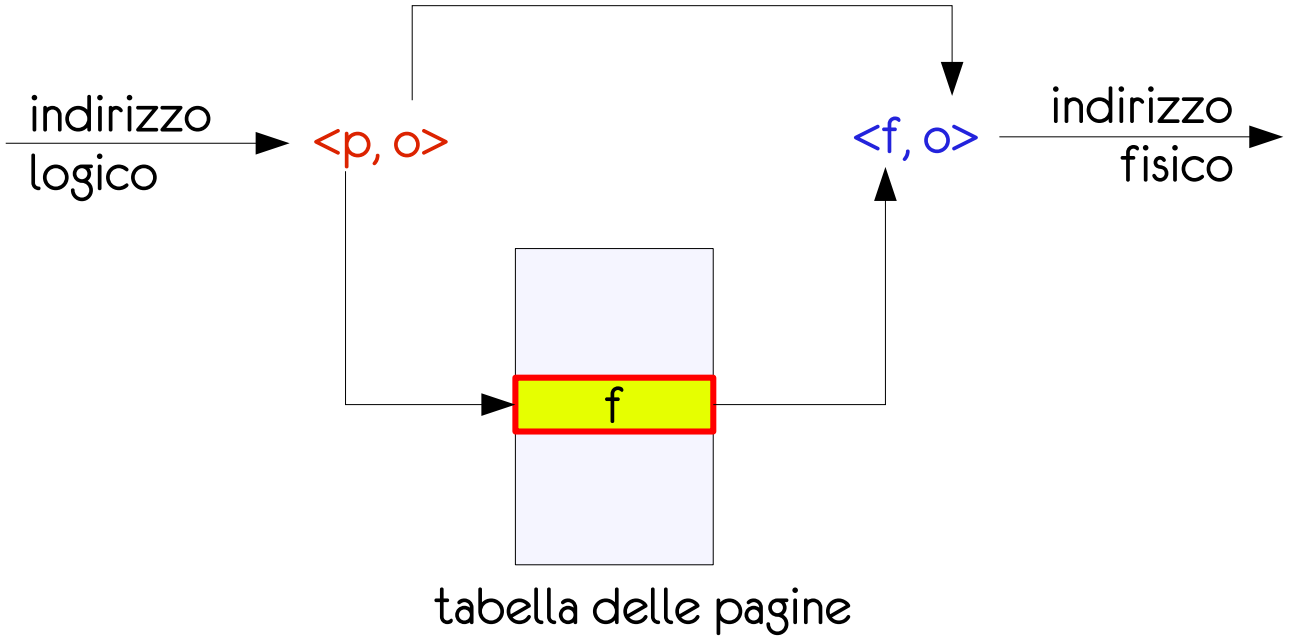
- Supponiamo di avere pagine di dimensione 512 byte e una RAM di dimensione 1MB, quante pagine avremo? Quanti bit occorrono per rappresentare indirizzi logici in questo contesto?
- Innanzi tutto devo riportarmi a potenze di 2 nella stessa unità di misura:
  - pagina: 512 byte = 2^9 byte
  - memoria logica: 1MB = 2^{20} byte
- Ora possiamo fare i conti:
  - numero di pagine necessarie: $2^{20} / 2^9 = 2^{20-9} = 2^{11}$
  - per rappresentare il numero di pagina occorrono 11 bit
  - per rappresentare l'offset occorrono): 9 bit
- num bit per le pagine + num bit x l'offset = 11 + 9 = 20

Paginazione e rilocazione

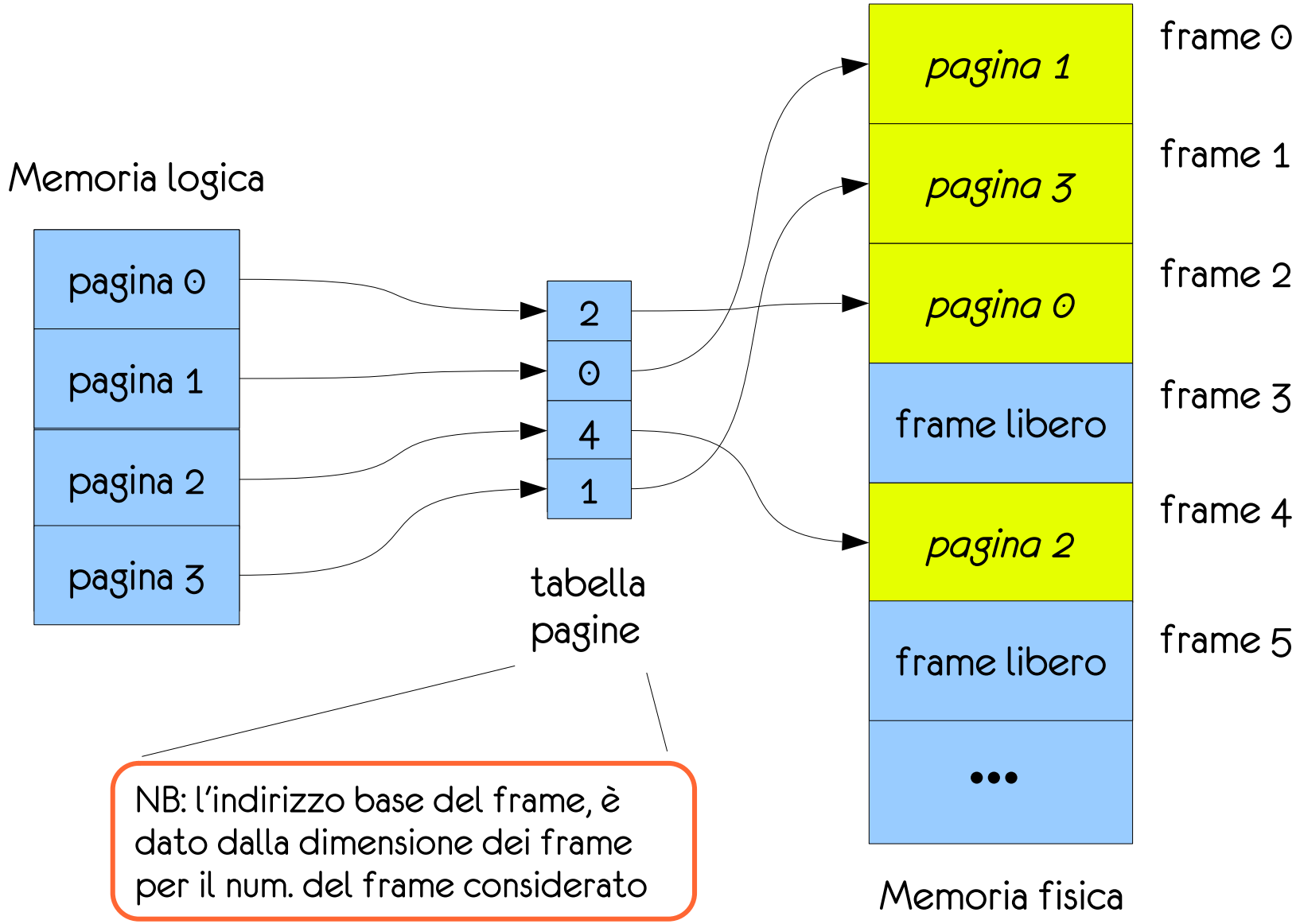
- Cosa vuol dire “rilocare il codice” in questo contesto?

consentire l'accesso a una pagina, indipendentemente dal frame in cui è caricata

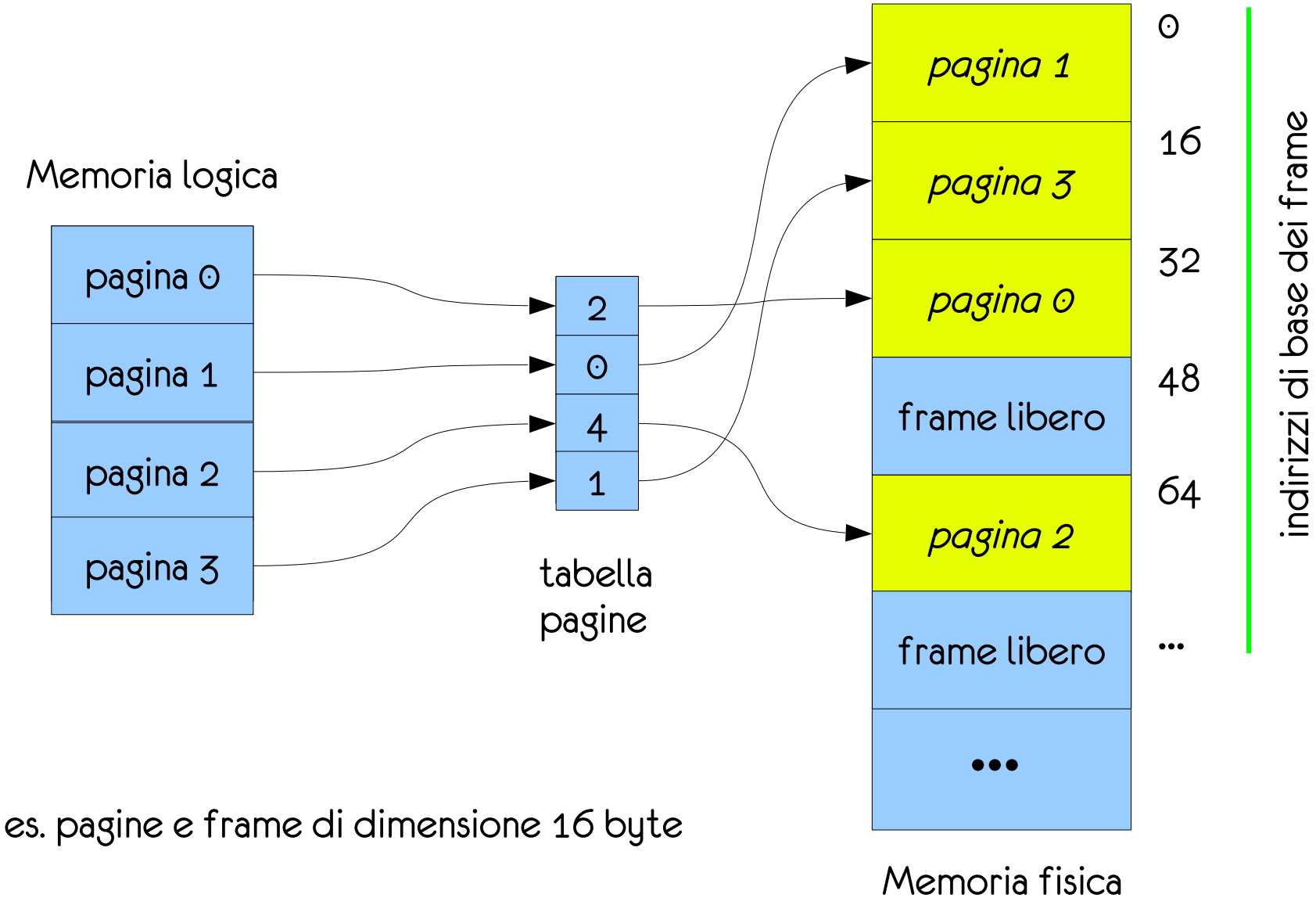
- Si può realizzare la rilocazione? Sì, il registro di rilocazione è sostituito dalla entry nella tabella delle pagine, corrispondente a p. Il valore f individua l'indirizzo di inizio del frame



Esempio di paginazione

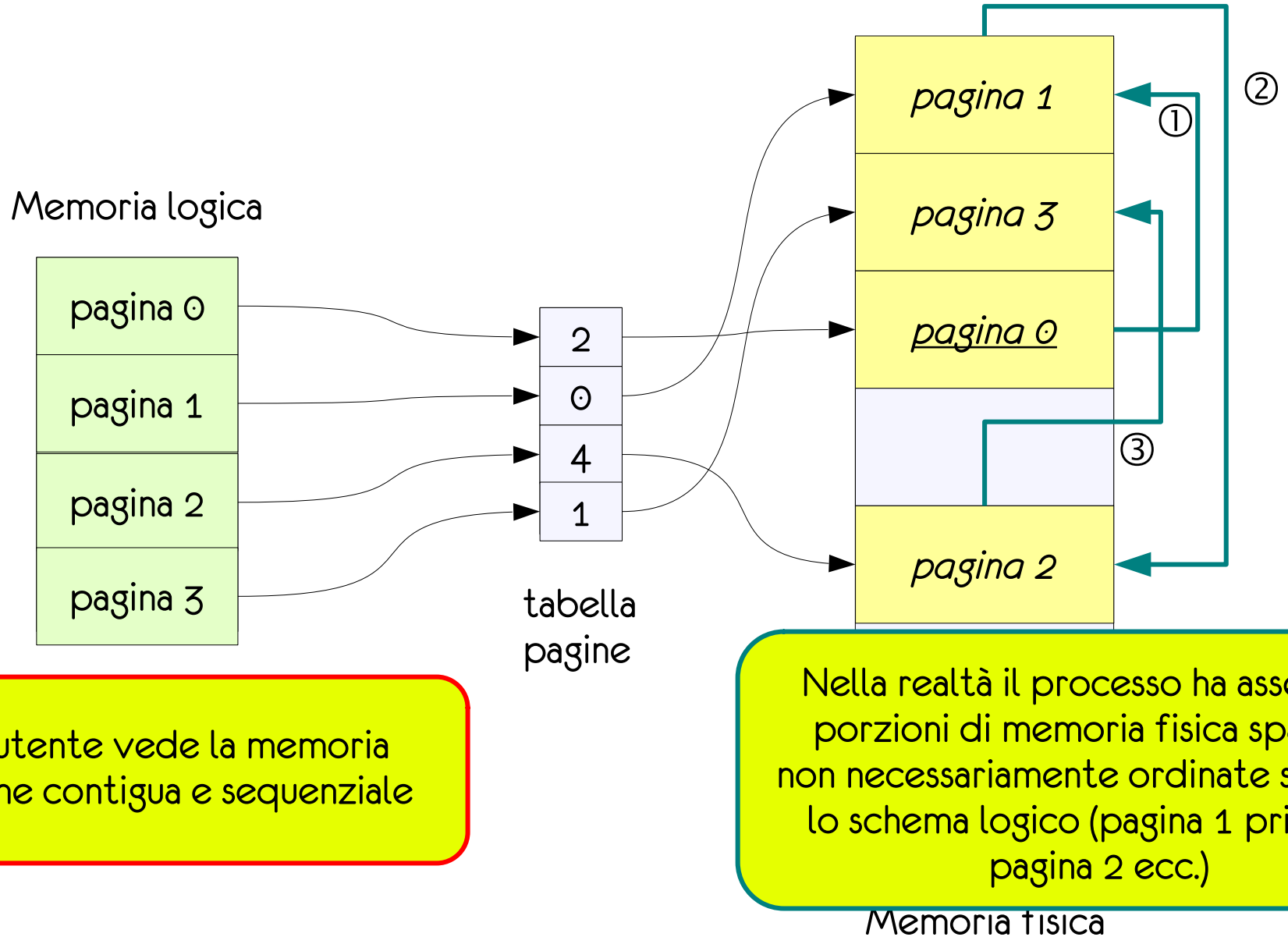


Esempio di paginazione



es. pagine e frame di dimensione 16 byte

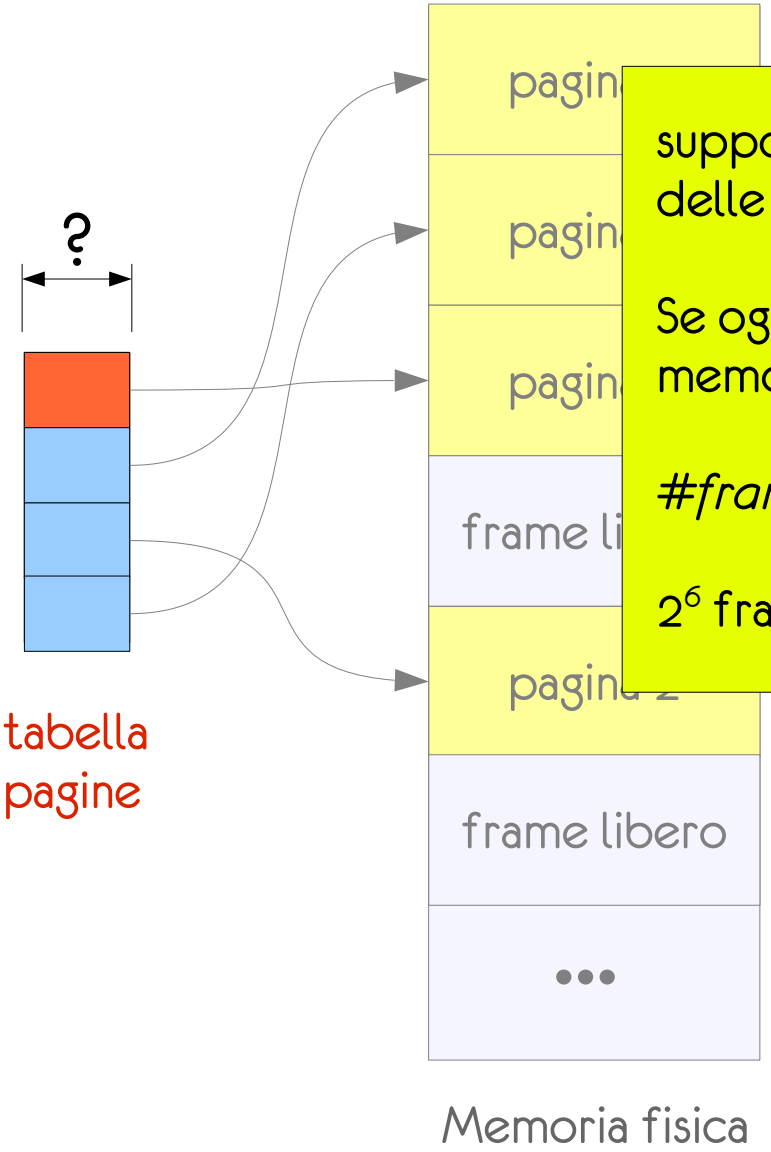
Esempio di paginazione



L'utente vede la memoria come contigua e sequenziale

Nella realtà il processo ha assegnate porzioni di memoria fisica sparse e non necessariamente ordinate secondo lo schema logico (pagina 1 prima di pagina 2 ecc.)

Qualche conto



supponiamo che un elemento della tabella delle pagine sia grande 4 byte

Se ogni frame ha dimensione 4 KB, a quanta memoria fisica riusciamo ad accedere?

*#frame indirizzabili * dimensione di un frame*

2^6 frame * 4 KB = 2^2 * 2^6 KB = 2^8 KB = 256 KB

Paginazione e frammentazione

- La paginazione elimina il problema della frammentazione esterna però permane il problema della frammentazione interna
- Ogni processo può avere allocate un numero di pagine diverso, quante di queste presenteranno frammentazione interna?
- Soltanto l'ultima perché raramente la dimensione di un processo sarà un multiplo della dimensione di una pagina, quindi in generale avrò bisogno di *N pagine più un pezzetto* per ciascun processo
-

Paginazione e frammentazione

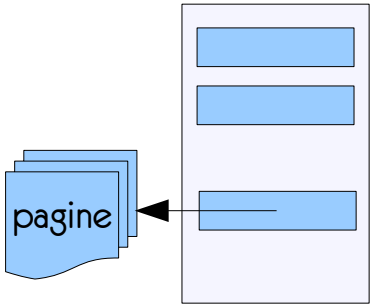
- **Frammentazione interna**: in media possiamo dire che avremo ½ pagina non utilizzata per ogni processo
- Dimensione ottimale delle pagine: occorre trovare un compromesso fra limitare il problema della frammentazione e ridurre i costi dell'I/O:
  - **pagine piccole**: riducono la frammentazione
  - **pagine grandi**: migliori quando occorre trasferire da/a memoria secondaria grosse quantità di dati (carico una pagina invece di tante in sequenza)

Quante tabelle delle pagine?

- ogni processo in RAM ha la propria tabella delle pagine ∈ PCB
- inoltre il SO mantiene (in copia unica) numero e lista dei frame liberi (tabella dei frame). Quando si carica un processo:

```
if (#frame liberi ≥ #pagine del processo) {
     foreach (pagina da caricare) {
        <<crea una entry nella tab. delle pag. del processo>>
        <<nuova entry = num. di un frame libero>>
        <<aggiorna tabella dei frame>>
        <<carica pagina>>
     }
 }
 else ???
      /* per ora diciamo che il processo non verrà caricato */
```

Architettura di paginazione



Il PCB di un processo contiene un riferimento alle pagine del processo stesso (codice e dati su memoria secondaria)

Quando la CPU è assegnata a un processo il dispatcher carica le sue pagine nella RAM impostando i valori della tabella delle pagine del processo e della tabella dei frame

Com'è implementata la tabella delle pagine?

tramite un insieme di registri

PRO: i registri sono ad accesso veloce

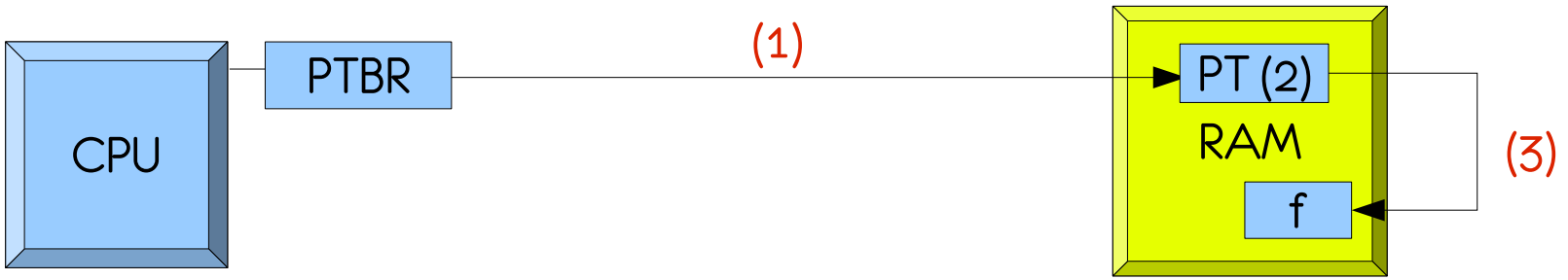
CONTRO: troppo pochi, ne serve uno per ogni pagina del processo

Accettabile se #pagine ≦ 256

viene memorizzata nella RAM si usa un solo registro (PTBR, page table base register) che contiene l'indirizzo di base della tabella

il dispatcher dovrà modificare solo il valore di questo registro

Problema: tempi di accesso



(1) tramite PTBR accedo alla RAM per individuare la page table

(2) tramite la page table costruisco l'indirizzo fisico a cui accede

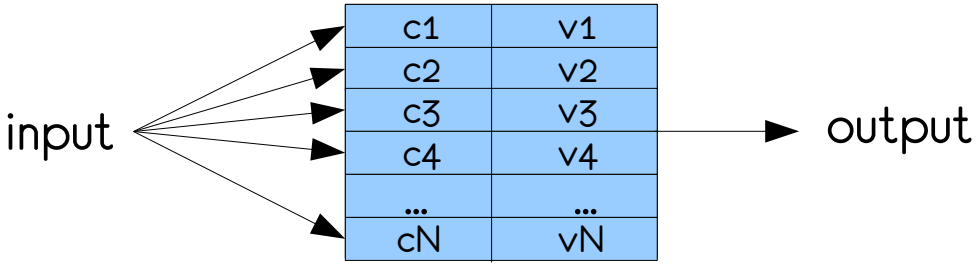
(3) accedo all'indirizzo fisico di interesse

Il numero di accessi alla RAM è raddoppiato perché ogni volta devo prima accedere alla tabella delle pagine (PT) e poi all'indirizzo che mi interessa

Il raddoppio degli accessi è inaccettabile, rallenta troppo l'esecuzione

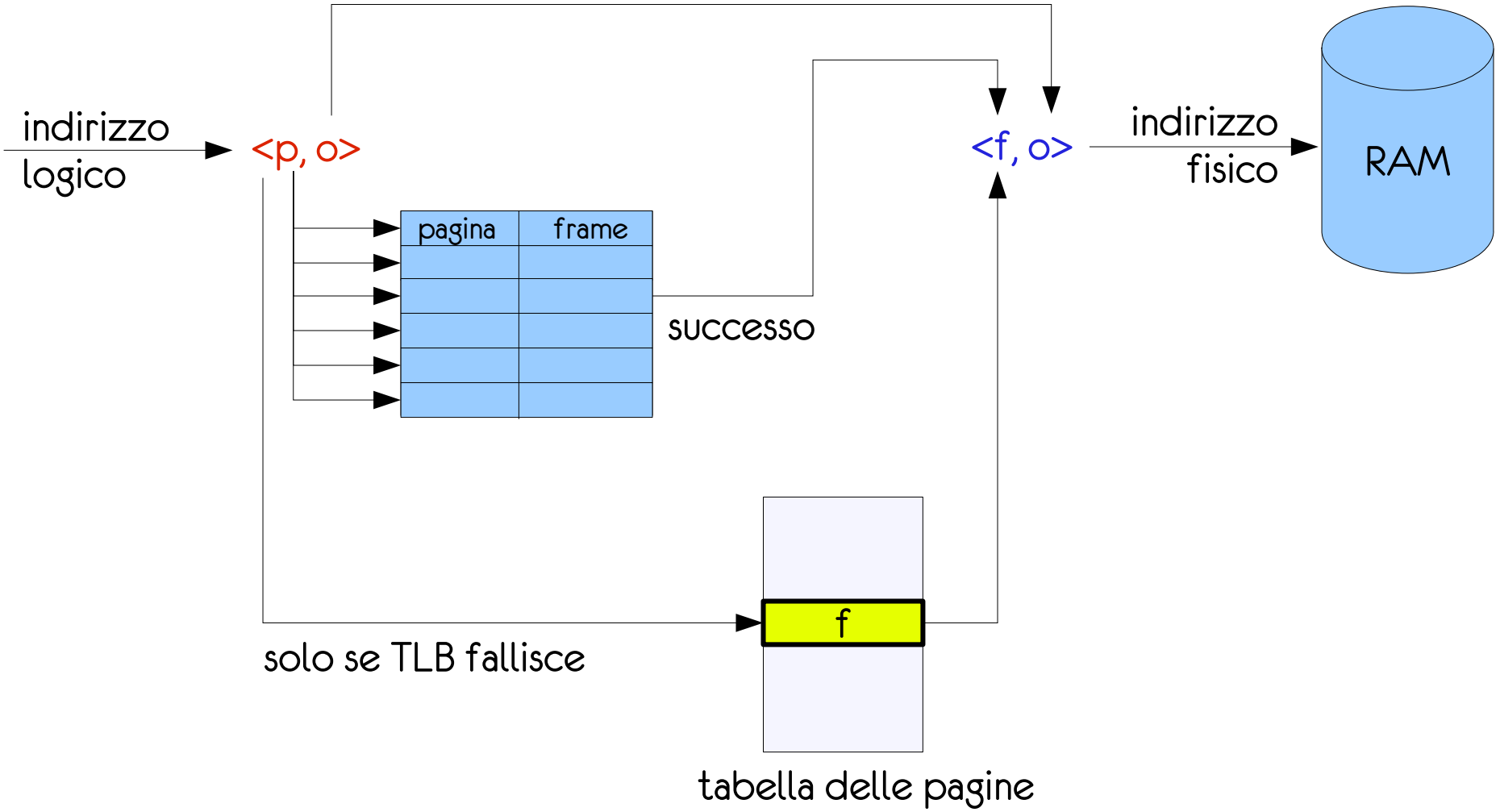
Soluzione: Translation look-aside buffer

TLB (translation look-aside buffer): è una cache molto veloce contenente delle coppie <chiave, valore>. Quando riceve un input lo confronta contemporaneamente con tutte le chiavi. Se trova una chiave corrispondente restituisce il valore associato.



Molte architetture adottano un TLB in associazione alla tabella delle pagine per limitare il problema del doppio accesso alla RAM, chiavi = numeri di pagina, valori = # frame

Schema d'uso


indirizzo
logico
<p, o>
pagina
frame
successo
<f, o>
indirizzo
fisico
RAM
f
solo se TLB fallisce
tabella delle pagine

Hit ratio

- Col termine **hit ratio** si intende la percentuale di successo relativo al reperimento di una pagina tramite TLB (la pagina è accessibile tramite il TLB)
- Dire che l'hit ratio è pari al 92% significa che in 92 casi su 100 il frame è stato individuato attraverso il TLB e solo nei restanti 8 si è dovuto accedere alla tabella delle pagine
- L'hit ratio consente di calcolare il tempo medio effettivo di accesso a una pagina

Hit ratio

Esempio

- Supponiamo che il tempo di accesso al TLB sia di 20 nsec mentre l'accesso alla RAM richiede 100 nsec
- Proviamo a calcolare il tempo medio effettivo necessario per accedere a una pagina nel caso in cui l'hit ratio è pari all'82% e nel caso in cui è pari al 95%

Hit ratio

- **Ipotesi:**
  - tempo di accesso al TLB = 20 nsec
  - tempo di accesso alla RAM = 100 nsec
  - hit ratio = 82% = 0.82
    (oppure 95%=0.95)
  - percentuale di accessi tramite page table = 18% = 0.18 (oppure 5%=0.05)

Hit ratio

- **Calcolo**
  - tempo di accesso a una pagina tramite TLB:
    - T_{tlb} = accesso a TLB + accesso RAM = 20 + 100 nsec = 120 nsec
  - tempo di accesso a una pagina tramite tabella delle pagina:
    - T_{pt} = 2 x accesso RAM = 100 + 100 nsec = 200 nsec
  - 82%: t.di a. medio = (0,82 * T_{tlb}) + (0,18 * T_{pt}) = 0,82 x 120 + 0,18 x 200 = 134,4 nsec
  - 95%: t.di a. medio = (0,95 * T_{tlb}) + (0,05 * T_{pt}) = 0,95 x 120 + 0,05 x 200 = 124 nsec

Aggiornamento del TLB

- Quando si inserisce una nuova coppia nel TLB?
- Quando non trovo una pagina di interesse (*TLB miss*):
  - se c'è spazio nel TLB si inserisce la nuova coppia <pagina, frame>
  - se non c'è spazio, si sostituisce una coppia già presente con quella nuova, di solito viene scelta per la sostituzione la coppia usata meno di recente
- l'idea di fondo è che quando accedo ad una nuova pagina, verosimilmente l'avrò bisogno per un po' di tempo
- NB: alcune delle pagine usate come chiavi nel TLB corrispondono ai processi principali del SO, le loro entry sono considerate non sovrascrivibili -> ho pagine di processi diversi!!!

Context switch

- Il TLB contiene dati (quasi) esclusivamente concernenti il processo running
- Quando si effettua un context switch occorre aggiornare il contenuto del TLB? Vorrei attuare un meccanismo di protezione che impedisca a un processo di accedere alle pagine di un altro

Context switch

- Due possibili soluzioni:
  - Alcuni TLB arricchiscono l'informazione contenuta in essi, aggiungendo un ASID (identificatore univoco dello spazio degli indirizzi di un processo). Un ASID identifica in modo univoco un processo. Uso l'informazione relativa a una pagina nel TLB, solo se l'ASID del processo running corrisponde all'ASID della pagina
  - Soluzione alternativa: il dispatcher svuota il TLB ad ogni context switch

Protezione

- Un aspetto ulteriore concerne la **modalità di accesso** alle pagine, memorizzata nella tabella delle pagine del processo
- Una pagina può essere accessibile in lettura, lettura-scrittura, esecuzione, oppure essere invalida (non appartiene allo spazio degli indirizzi del processo).
- Si possono avere pagine non valide quando la tabella delle pagine di un processo è più piccola dello spazio riservato ad essa nella RAM
- Se un processo cerca di accedere a una pagina non valida, viene generato un interrupt
- La modalità di accesso viene aggiunta alla tabella delle pagine

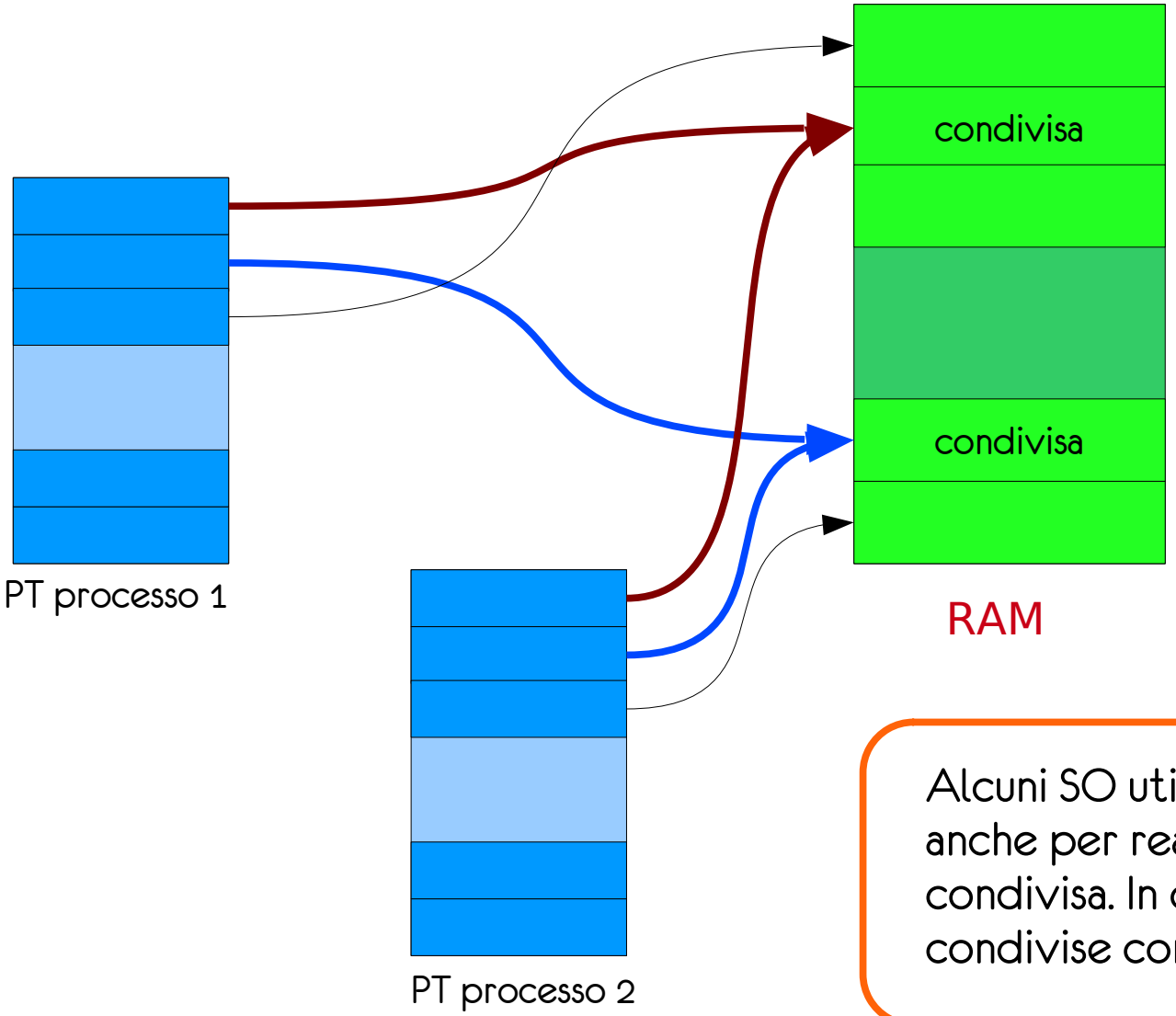
Condivisione delle pagine

- Spesso nei sistemi in time-sharing diversi utenti usano lo stesso programma contemporaneamente
- In un contesto di multi-programmazione in senso lato, succede spesso che più processi utilizzino la stessa libreria contemporaneamente

Condivisione delle pagine

- Supponiamo che un programma di questo tipo (es. compilatore) occupi 200KB, che 4 utenti lo stiano usando, che i loro processi abbiano 1 pagina di dati e che le pagine siano grandi 50KB: complessivamente saranno occupate 20 pagine di RAM: 1000KB
- se fosse possibile far condividere il codice sarebbero invece sufficienti 400KB di RAM (50 x 4 per i dati + 200 per il codice in copia unica)
- NB: solo le pagine di codice possono essere condivise, pagine accessibili in sola lettura. Si parla di **codice rientrante**

Condivisione delle pagine



Alcuni SO utilizzano questo sistema anche per realizzare la memoria condivisa. In questo caso le pagine condivise contengono dati

Struttura della tabella delle pagine

capitolo 8 del libro (VII ed.), da 8.5

Cristina
Baroglio

Elenco

- paginazione multi-livello:
  - struttura gerarchica ad albero
- hash table:
  - spesso usata per architetture oltre i 32 bit
- tabella delle pagine invertita:
  - approccio diverso in cui si ha una sola tabella delle pagine globale anziché una per ciascun processo

Paginazione multi-livello

- La dimensione della tabella delle pagine dipende dalla **dimensione dello spazio degli indirizzi logici**
- Consideriamo:
  - indirizzi a 32 bit (permettono di fare riferimento a 2^{32} parole di memoria),
  - con pagine grandi 4KB (cioè 2^{12})
- in quante pagine verrà suddiviso lo spazio degli indirizzi logici?

Paginazione multi-livello

- In quante pagine verrà suddiviso lo spazio degli indirizzi logici?

  $2^{32} / 2^{12} = 2^{20}$

- cioè la tabella delle pagine potrà avere fino a 2^{20} entry (precisamente 1,048,576)

Paginazione multi-livello

- In quante pagine verrà suddiviso lo spazio degli indirizzi logici?

  $2^{32} / 2^{12} = 2^{20}$

- cioè la tabella delle pagine potrà avere fino a 2^{20} entry (precisamente 1,048,576)
- se ogni entry occupa 4byte, una tabella delle pagine potrà occuperà **4MB** !!!

Paginazione multi-livello

- **Osservazione**:

  nella realtà nessun processo usa l'intero spazio degli indirizzi, ne usa molto meno

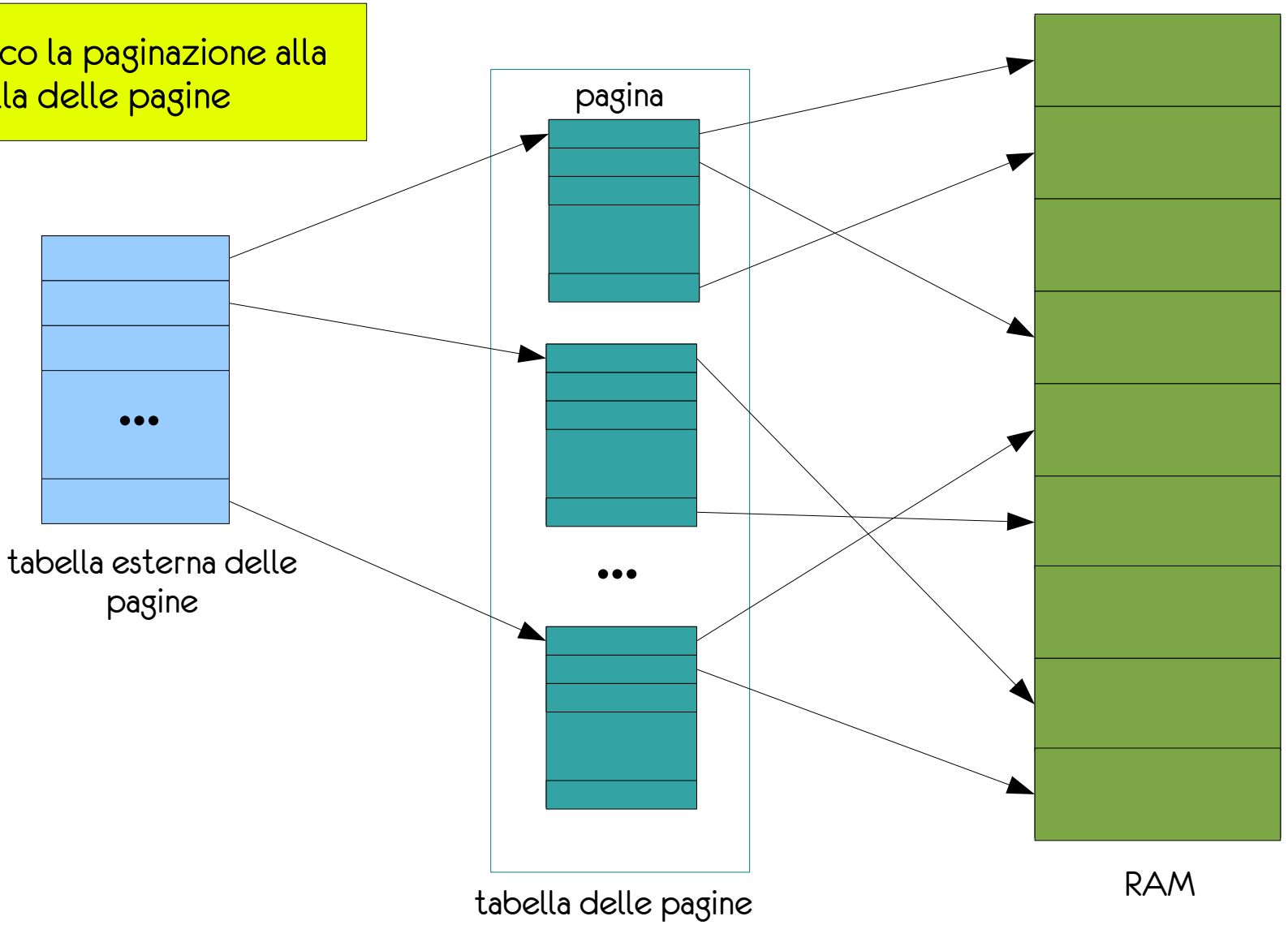
  la maggior parte della tabella sarebbe inutilizzata, tuttavia i processi possono avere tabelle delle pagine molto pesanti

Paginazione multi-livello

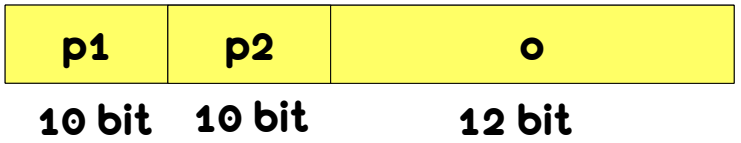
- **Conclusione**: l'allocazione della tabella delle pagine non può essere contigua ma va suddivisa in tante parti organizzate in una struttura
- **Nella paginazione multi-livello si usa un albero**

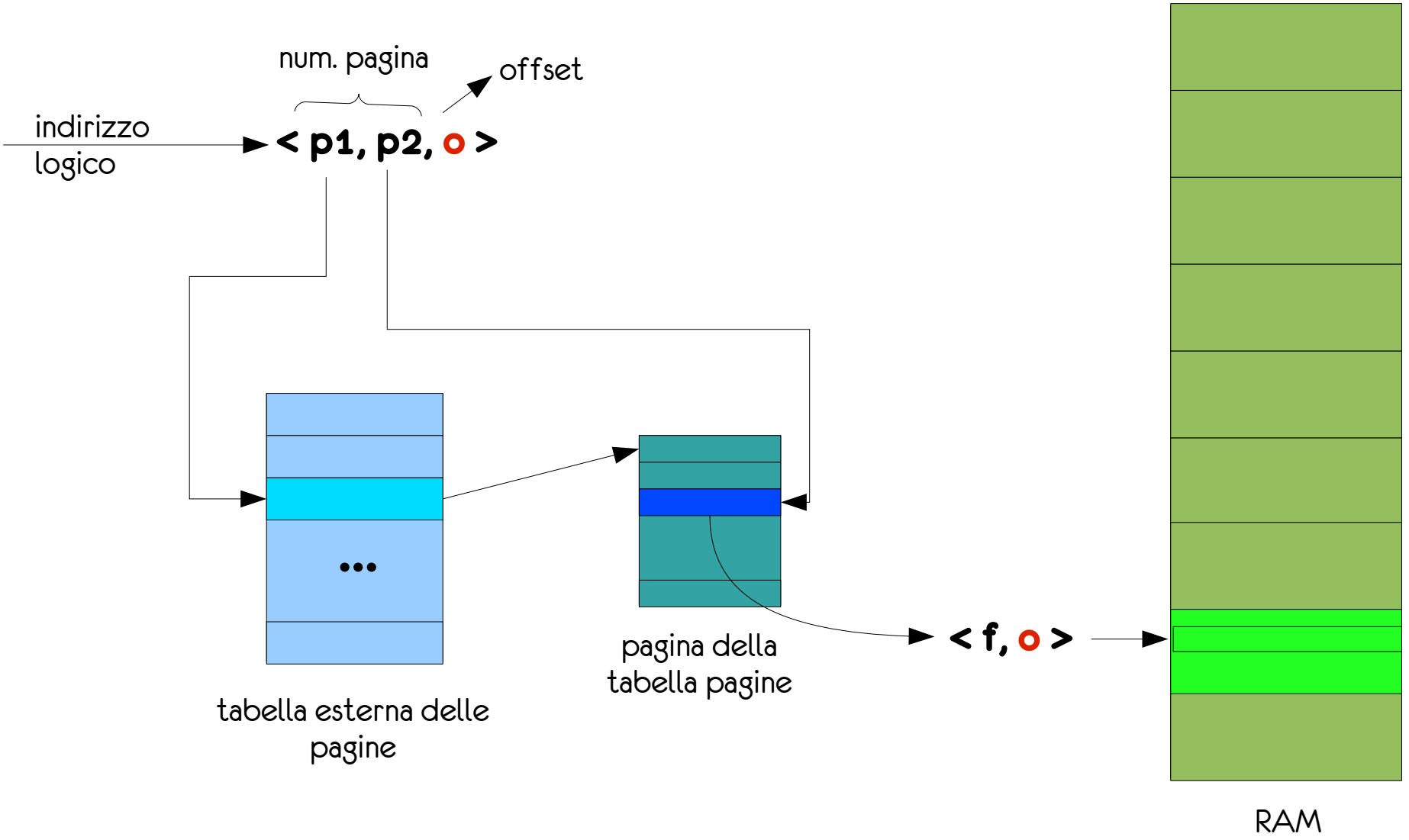
Paginazione a due livelli

applico la paginazione alla tabella delle pagine



Indirizzi logici





Esempi e commenti

- Architettura SPARC (di Sun):
  paginazione a 3 livelli
- Motorola 68030 a 32 bit:
  paginazione a 4 livelli

Esempi e commenti

- La paginazione multilivello non è appropriate per architetture a 64 bit: la tabella esterna risulterebbe troppo grande
- Si dovrebbero aggiungere troppi livelli ulteriori di paginazione (es. UltraSPARC ne avrebbe richiesti 7!) e ciò renderebbe troppo costoso l'accesso alla RAM qualora si verificassa un TLB miss
- Diverse architetture a 64 bit ad alte prestazioni usano tabelle delle pagine inverse (vedremo fra poco) unitamente a tabelle hash

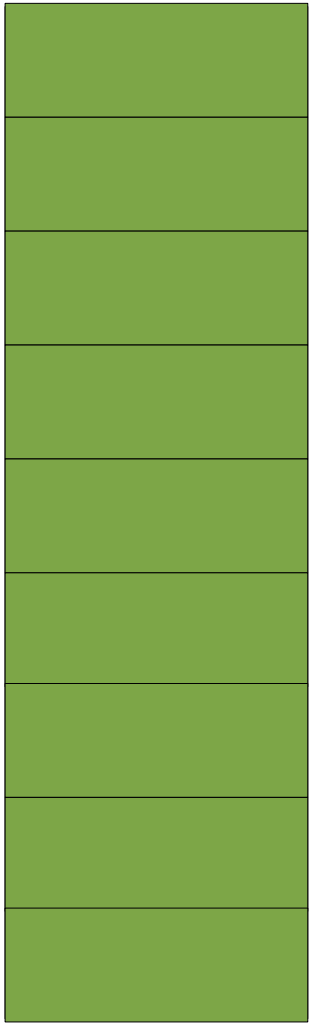
Vantaggi e svantaggi delle tabelle delle pagine

- **Vantaggi**
  - rappresentazione naturale se si adotta un approccio processo-centrico
  - la tabella delle pagine è ordinata in modo tale che sia semplice per il SO identificare il punto in cui si trova l'indirizzo del frame di interesse

Vantaggi e svantaggi delle tabelle delle pagine

- **Svantaggi**
  - una tabella delle pagine può essere grandissima (un elemento per ogni pagina del processo), contenere milioni di elementi e occupare troppo spazio di quella RAM che dovrebbe contribuire a gestire
- **Esistono approcci alternativi?**
- **invertiamo il fuoco e ricominciamo il discorso a partire dalla RAM anziché dai processi**

Tabella invertita



RAM

La RAM è suddivisa in frame, ogni frame può essere libero oppure contenere la pagina di un processo

Posso quindi pensare di mantenere una sola tabella con una entry di questo tipo per ogni frame:

< pid, p >

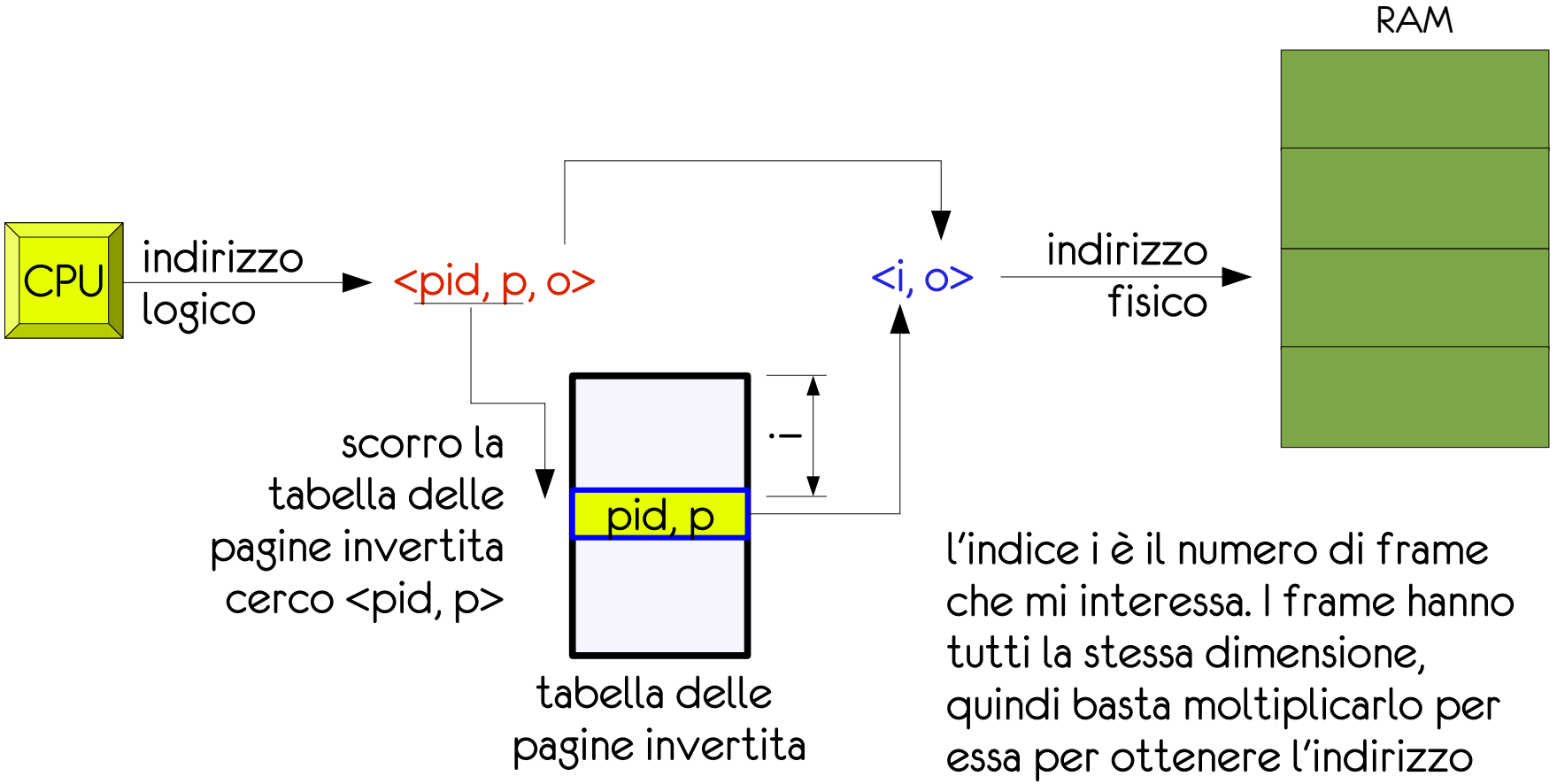
pid = processo, p = pagina

Gli indirizzi logici saranno quindi triple di questo tipo:

< pid, p, o>

pid = processo, p = pagina, o = offset

Struttura



l'indice i è il numero di frame che mi interessa. I frame hanno tutti la stessa dimensione, quindi basta moltiplicarlo per essa per ottenere l'indirizzo base del frame

Vantaggi e svantaggi

- **Vantaggi**
  - rappresentazione più compatta
  - richiede meno spazio in memoria
- **Svantaggi**
  - tempi d'accesso aggravati dalla ricerca in tabella
  - per ovviare a questo limite talvolta la tabella delle pagine invertita è implementata come una hash table talvolta ci si appoggia a un TLB

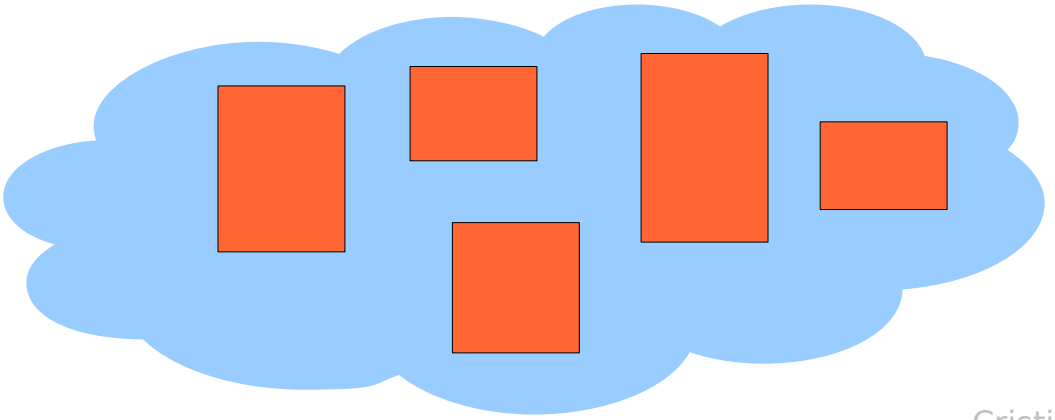
segmentazione

capitolo 8 del libro (VII ed.), da 8.6

Cristina
Baroglio

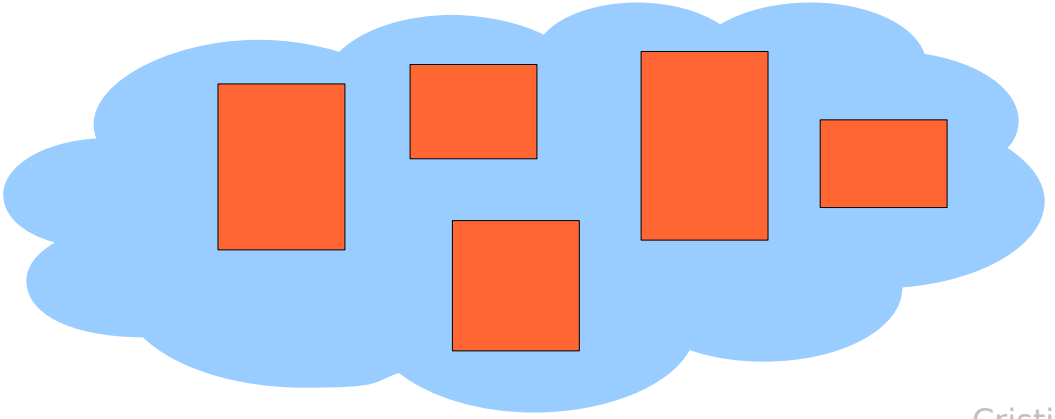
Introduzione

- La paginazione struttura la memoria in un insieme di elementi, il cui contenuto può essere indifferentemente costituito da codice e/o dati
- Questa organizzazione è molto diversa dal modo in cui i *programmatori* vedono i programmi, cioè come strutturate in parti con un preciso *valore funzionale* (il codice, le librerie, lo stack, l'heap)



Introduzione

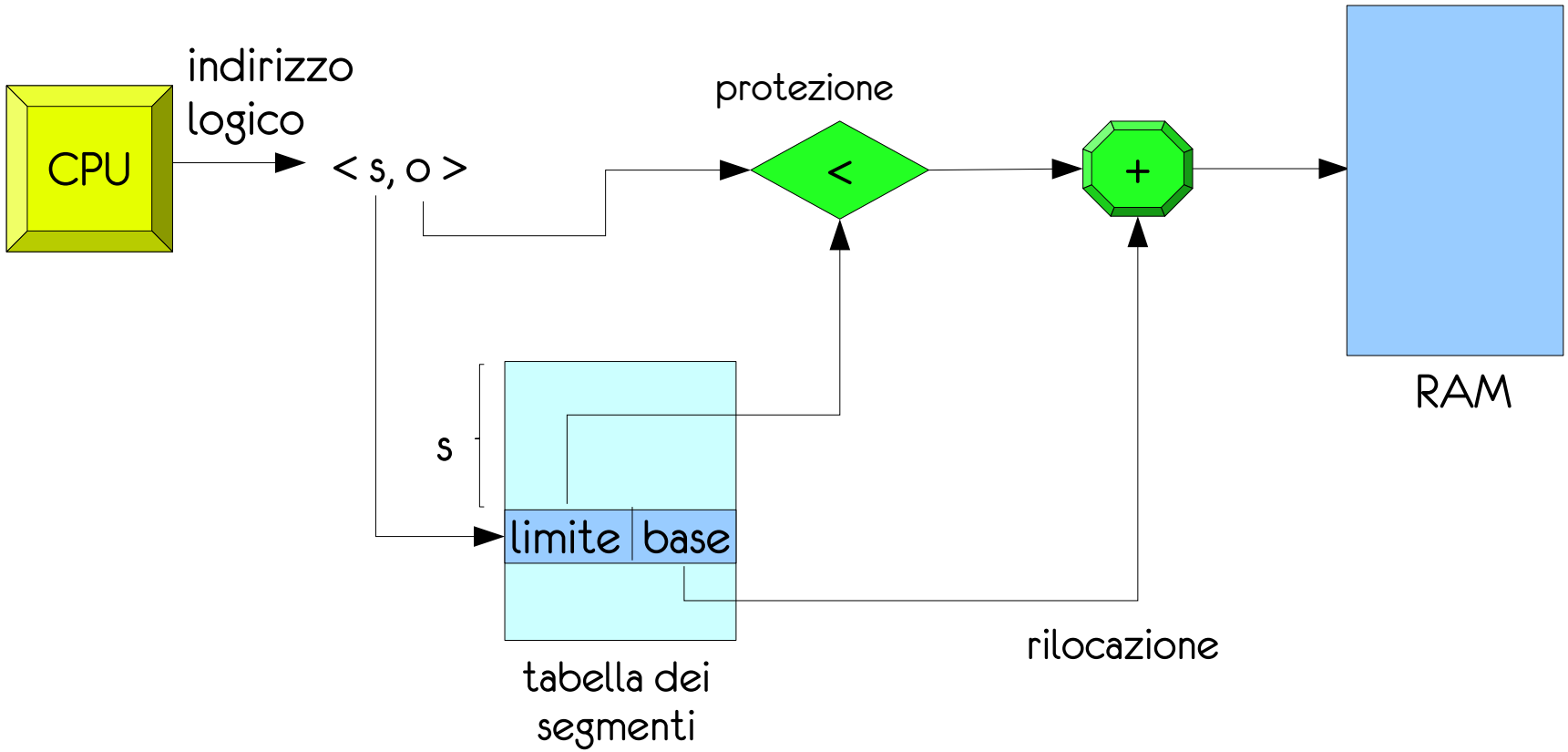
- La segmentazione modella la memoria secondo uno schema più vicino al modo in cui il programmatore vede il programma
- Ogni processo ha una porzione di RAM organizzata come un insieme di segmenti di memoria, di dimensione variabile



Segmenti

- Per adottare questo approccio è necessario che il programma risulti già organizzato in segmenti. Tale strutturazione viene effettuata dal compilatore
- Es. compilatore C può creare questi segmenti per un programma:
  - codice
  - vrb globali
  - stack per ciascun thread
  - heap
  - libreria standard del C
  - le altre librerie avranno assegnati segmenti in un tempo successivo
- Indirizzo logico: **< id_di_segmento, offset >**

Architettura



s = indice della entry relativa al segmento

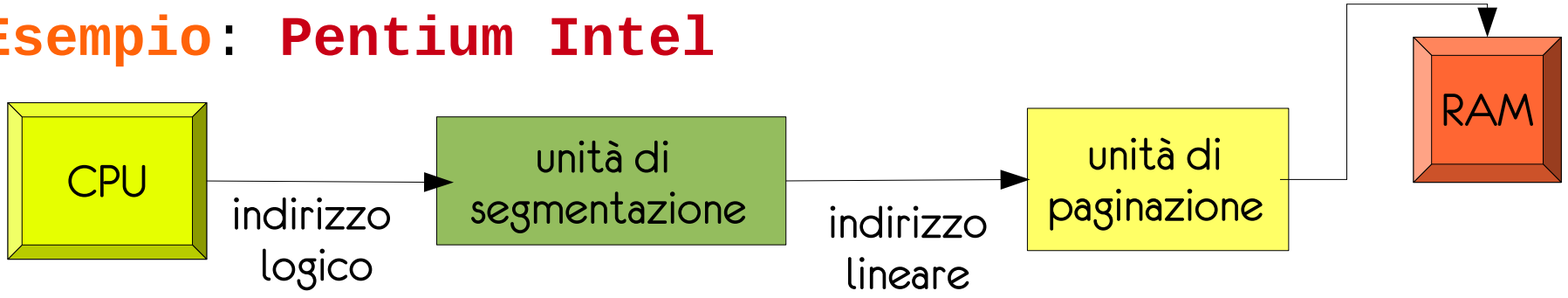
ogni segmento è caratterizzato da due valori: indirizzo di base e indirizzo limite perché la loro dimensione è varia

Commenti

- La segmentazione è spesso più efficiente della paginazione
- I segmenti possono avere dimensione estremamente varia
- Quando lo scheduler a medio o lungo termine carica un processo, deve trovare uno spazio adeguato alle dimensioni dei suoi segmenti.
- I problemi sono simili a quelli dello schema a partizioni multiple, anche se la segmentazione è un meccanismo molto più flessibile
  - occorre gestire la memoria libera in modo dinamico
  - si ha frammentazione esterna
  - possibile attuare tecniche di compattazione per ridurre la frammentazione

paginazione + segmentazione

- Alcune architetture sfruttano una tecnica che si basa sull'unione di paginazione e segmentazione
- **Esempio: Pentium Intel**
-



Ogni segmento è strutturato in pagine

dimensione max segmento 4GB
numero max di segmenti per processo 16K

Il processore ha 6 registri di segmento che permettono a un processo di fare riferimento a 6 segmenti contemporaneamente

8K riservati ∈ tabella locale dei descrittori

8K condivisi ∈ tabella globale dei descrittori

Intel Pentium

indirizzo logico

< segmento, g, o >

id di segmento

offset: 32 bit

segmento globale/locale

indirizzo lineare

ottenuto tramite base e limite del segmento
+ offset: non è ancora un indirizzo fisico

Si applica infine uno schema
di **paginazione a due livelli**

Indirizzo lineare così scomposto:

< p1, p2, offset >

Conclusioni

- Esistono diversi modelli per la gestione della RAM
- La scelta dipende fortemente dall'HW a disposizione
- Ogni HW è progettato in modo tale da supportare uno (o più) modelli specifici
- Nel valutare un modello/HW occorre tenere presente:
  - se consente la rilocazione
  - se consente lo swapping
  - se consente la condivisione
  - se attua meccanismi di protezione

supporti alla multi-programmazione