Parte IV: Gestione della Memoria

- Memoria Centrale (cap. 9)
- Memoria Virtuale (cap. 10)

9 Memoria Centrale (gestione della…)

- Introduzione
- Binding degli indirizzi
- Spazio degli indirizzi logici e fisici
- Allocazione contigua dei processi in RAM
- Paginazione della memoria primaria

Introduzione

- Abbiamo visto che i moderni SO tentano di massimizzare l'uso delle risorse della macchina, e in primo luogo l'utilizzo della CPU.
- Questo si ottiene mediante le due tecniche fondamentali del multi-tasking e del time-sharing, che richiedono di tenere in memoria primaria contemporaneamente più processi attivi.
- Il SO deve decidere come allocare lo spazio di RAM tra i processi attivi, in modo che ciascun processo sia pronto per sfruttare la CPU quando gli viene assegnata.

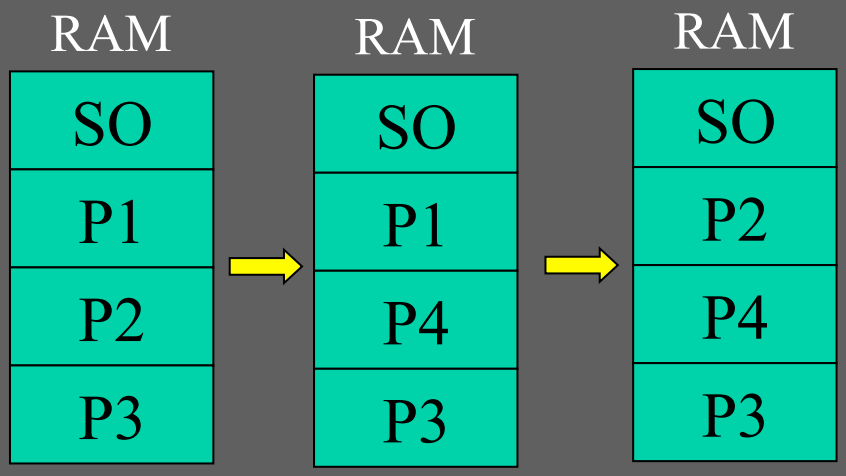
Introduzione

- Supponiamo però che, ad un certo punto, la RAM sia completamente occupata da 3 processi utente, P1, P2 , P3 (per semplicità assumiamo che a tutti i processi venga assegnata una porzione di RAM della stessa dimensione).
- Un nuovo processo P4 viene fatto partire, è immediatamente pronto per usare la CPU, ma non c'è più spazio per caricare il suo codice in RAM, che si può fare?
- Ovviamente si potrebbe aspettare la terminazione di uno dei 3 processi già in RAM, ma supponiamo che uno dei tre processi (diciamo P2) sia temporaneamente in attesa di compiere una lunga operazione di I/O (per cui non userà la CPU a breve)

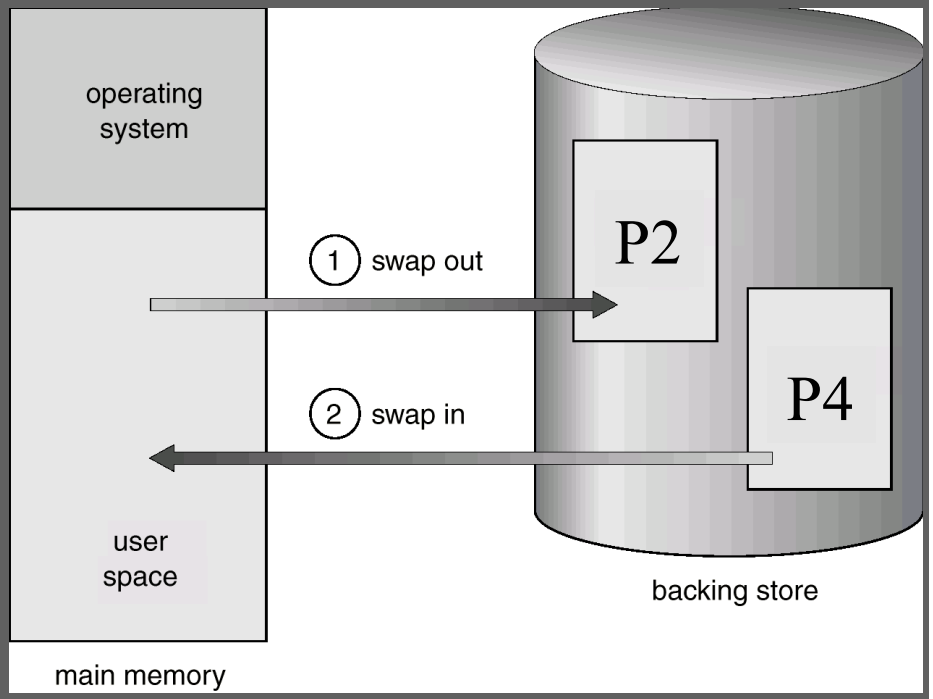
Introduzione

- Il SO potrebbe decidere di spostare temporaneamente P2 sull'hard disk per far posto a P4, che così può concorrere all'uso della CPU.
- Che cosa viene spostato sull'hard disk? **l'immagine** di P2: il codice (anche se, come capiremo meglio più avanti, questo si può anche evitare), i dati e lo stack del processo
- Dopo un po' P1 termina e libera una porzione di RAM. Il SO potrebbe riportare P2 in RAM (ma ora nello spazio che era stato inizialmente assegnato a P1)
- Questa tecnica viene chiamata **swapping** (**avvicendamento di processi**). L'area del disco in cui il SO copia temporaneamente un processo viene detta **area di swap.**

Introduzione



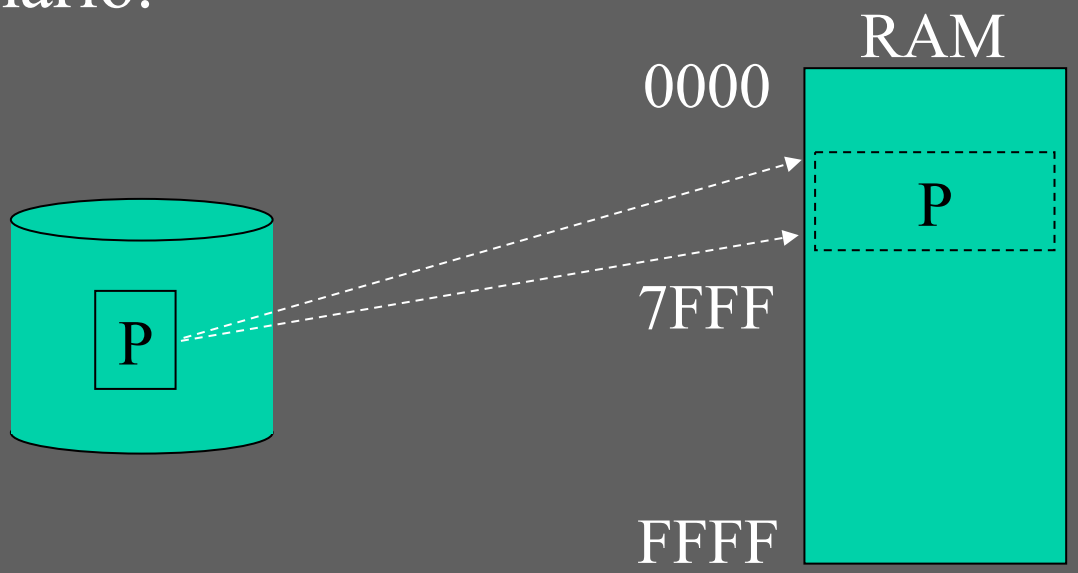
(fig. 9.19 modificata)



- Lo Swapping è raramente usato nei moderni sistemi operativi perché troppo inefficiente, ma l'esempio mette in luce un problema fondamentale nella gestione della memoria primaria: P2 contiene istruzioni che usano indirizzi di memoria primaria: **funziona ancora correttamente quando viene spostato da un'area di RAM ad un'altra?**

Introduzione

- Perché un programma possa essere eseguito, il suo codice deve trovarsi in memoria primaria (ma rivedremo questa affermazione quando parleremo della memoria virtuale)
- Quindi, quando il SO riceve il comando di esecuzione di un programma, deve recuperare il codice del programma dalla memoria secondaria, e decidere in quale porzione della memoria primaria sistemarlo.
  ossia, a partire da quale indirizzo di RAM.
  La cosa non è banale, infatti...



9.1.2 Binding (associazione) degli indirizzi

- Un programma sorgente usa (tra l'altro) dati (variabili) e istruzioni di controllo del flusso di computazione.
- Quando il programma viene compilato e caricato in Memoria Primaria (MP) per essere eseguito, ad ogni variabile è associato l'indirizzo di una locazione di memoria che ne contiene il valore.
- Alle istruzioni di controllo del flusso di esecuzione del programma (ossia i salti condizionati e incondizionati) è associato l'indirizzo di destinazione del salto.
- L'operazione di associazione di variabili e istruzioni agli indirizzi di memoria è detta **binding degli indirizzi**

9.1.2 Binding degli indirizzi

- In altre parole, ad ogni variabile dichiarata nel programma viene fatto corrispondere l'indirizzo di una cella di memoria di RAM in cui verrà memorizzato il valore di quella variabile.
- L'accesso alla variabile, in lettura e scrittura, corrisponde alla lettura e scrittura della cella di memoria il cui indirizzo è stato "legato" (con l'operazione di binding) alla variabile.
- Le istruzioni di salto, che permettono di implementare costrutti come *if-then-else*, *while*, ecc. Sono associate agli indirizzi in RAM dove si trova l'istruzione con cui prosegue l'esecuzione del programma se il salto viene eseguito.

9.1.2 Binding degli indirizzi

- Ad esempio, un istruzione C come:

  counter = counter + 1;

- alla fine diventerà qualcosa del tipo:

  load(R1, 10456)
  Add(R1, #1);
  store(R1, 10456)

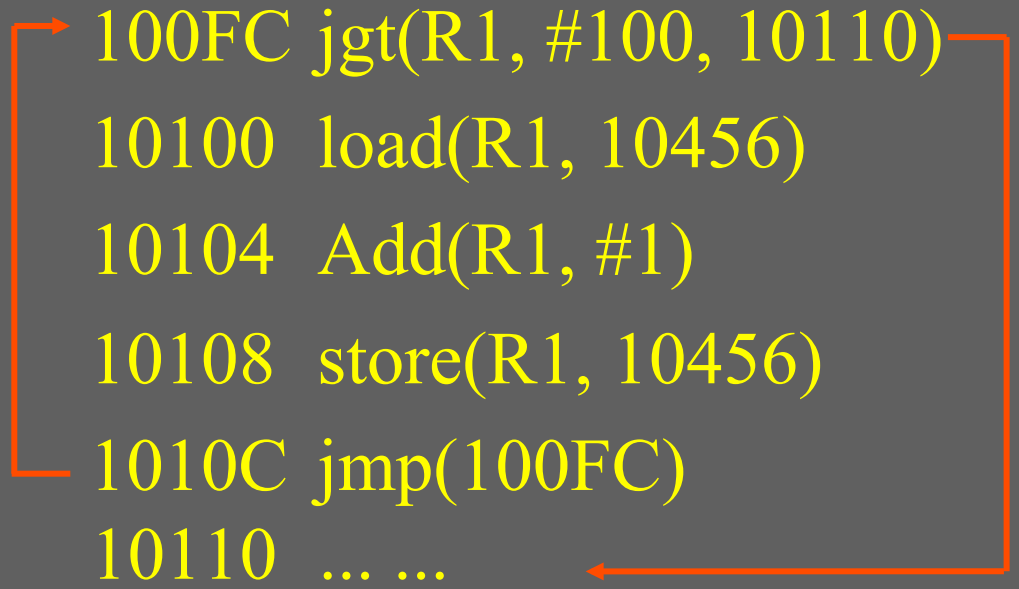
- **10456** è l'indirizzo della cella di memoria che contiene il valore della variabile "counter". L'indirizzo 10456 è stato associato alla variabile "counter" durante la fase di binding degli indirizzi.

9.1.2 Binding degli indirizzi

- Analogamente, un istruzione C come:

  while (counter <= 100) counter++;

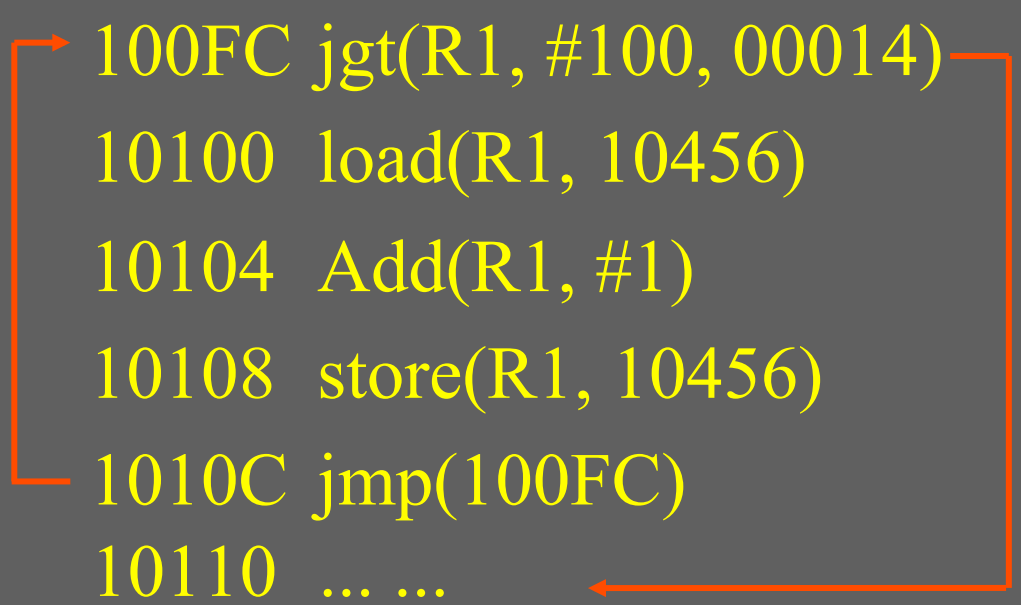
- alla fine diventerà qualcosa del tipo:



// jump if greater than

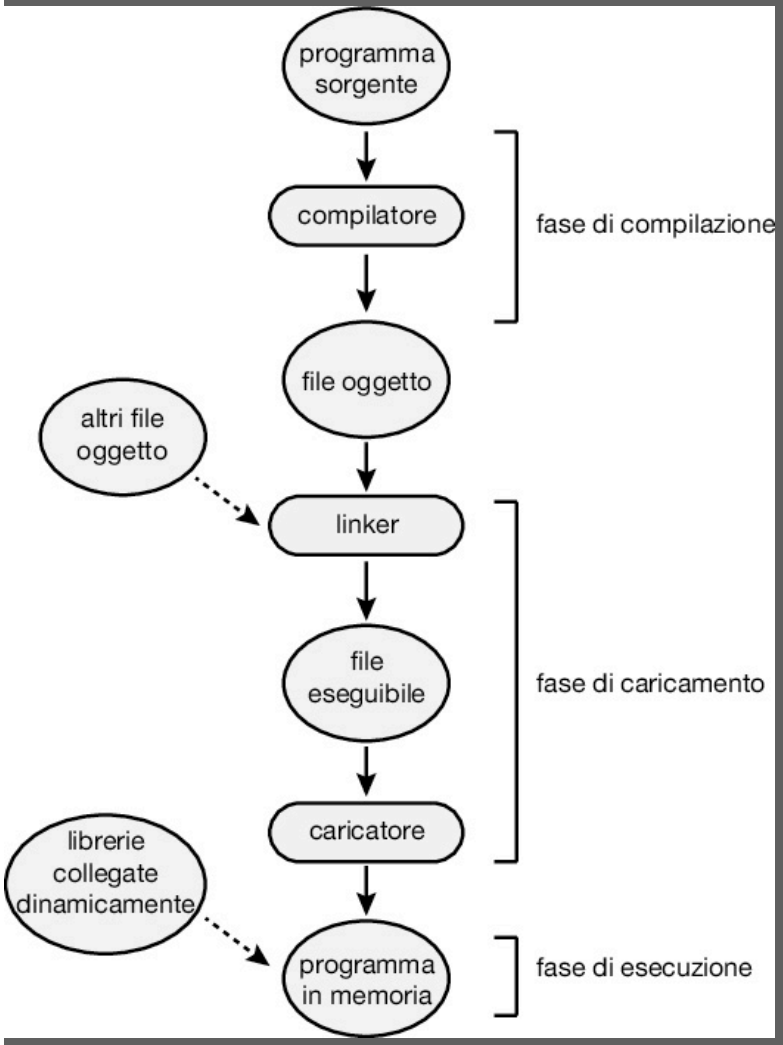
9.1.2 Binding degli indirizzi

- La destinazione dei salti può anche essere specificata rispetto all'indirizzo di istruzione del salto stesso, e dunque il *while* della slide precedente potrebbe anche essere tradotto in assembler così:



// jump if greater than

9.1.2 Binding degli indirizzi



Perché un programma sorgente possa essere eseguito deve passare attraverso varie fasi (fig. 9.3).

Il binding degli indirizzi avviene in una di queste fasi:

-- compilazione
-- caricamento (in RAM)
-- esecuzione

9.1.2 Binding degli indirizzi: quando?

1. **In fase di compilazione**
   - viene generato **codice assoluto o statico.**
   - Il compilatore deve conoscere l'indirizzo della cella di RAM a partire dal quale verrà caricato il programma, in modo da effetture il binding degli indirizzi (che avviene, appunto, in fase di compilazione).
   - Se il SO deve scaricare temporaneamente il processo che usa quel codice in Memoria Secondaria (MS), come nell'esempio visto a inizio capitolo, quando lo ricarica in RAM deve rimetterlo esattamente dove si trovava prima. (Oppure?)
   - Questa gestione è ovviamente molto rigida

prog1:

```
int var1 = 5;
if (var1%2 == 0) var1++;
var1 = var1 * 2;
```

compilazione

```
load R1,10C0
jmp_if_odd R1,10B8
add R1,#1
mul R1,#2
store R1,10C0
 /* four byte int. var */
```

prog1 sarà caricato qui: 10AC

loading

RAM:

```
10A0 ... ... ... ... ...
10A4 ... ... ... ... ...
10A8 ... ... ... ... ...
10AC load R1,10C0
10B0 jmp_if_odd R1,10B8
10B4 add R1,#1
10B8 mul R1,#2
10BC store R1,10C0
10C0 /* four byte int. var */
10C4 ... ... ... ... ...
10C8 ... ... ... ... ...
10CC ... ... ... ... ...
```

9.1.2 Binding degli indirizzi: quando?

2. In fase di caricamento in RAM

- viene generato **codice staticamente rilocabile.**
- Il compilatore associa ad istruzioni e variabili degli **indirizzi relativi rispetto all'inizio del programma, che inizia da un ipotetico indirizzo 0 virtuale**
- Gli indirizzi assoluti finali vengono generati in fase di caricamento del codice in MP in base all'indirizzo di MP a partire dal quale è caricato il codice.
- Il binding degli indirizzi quindi, avviene in fase di caricamento del programma in RAM: se il processo che usa quel codice viene tolto dalla RAM, si può caricarlo in una posizione diversa solo rieffettuando la fase di caricamento (ma è più efficiente che ricompilare tutto)

prog1:

```
int var1 = 5;
if (var1%2 == 0) var1++;
var1 = var1 * 2;
```

compilazione

prog1 sarà caricato qui: 10AC

loading

prog1 parte da uno 0 virtuale

```
0000 load R1, 14
0004 jmp_if_odd R1, C
0008 add R1,#1
000C mul R1,#2
0010 store R1,14
0014 /* four byte int. var */
```

10AC+C=10B8

RAM:

```
10A0 ... ... ... ... ...
10A4 ... ... ... ... ...
10A8 ... ... ... ... ...
10AC load R1,10C0
10B0 jmp_if_odd R1,10B8
10B4 add R1,#1
10B8 mul R1,#2
10BC store R1,10C0
10C0 /* four byte int. var */
10C4 ... ... ... ... ...
10C8 ... ... ... ... ...
10CC ... ... ... ... ...
```

9.1.2 Binding degli indirizzi: quando?

3. In fase di esecuzione

- viene generato **codice dinamicamente rilocabile**
- Il codice in esecuzione usa sempre e solo indirizzi relativi
- la trasformazione di un indirizzo relativo in uno assoluto viene fatta nell'istante in cui viene eseguita l'istruzione che usa quell'indirizzo.
- E' necessario opportuno supporto HW per realizzare questo metodo senza perdita di efficienza
- si parla di **binding dinamico degli indirizzi**

prog1:

```
int var1 = 5;
if (var1%2 == 0) var1++;
var1 = var1 * 2;
```

compilazione

loading

prog1 parte da uno 0 virtuale

```
0000 load R1,14
0004 jmp_if_odd R1, C
0008 add R1,#1
000C mul R1,#2
0010 store R1,14
0014 /* four byte int. var */
```

RAM:

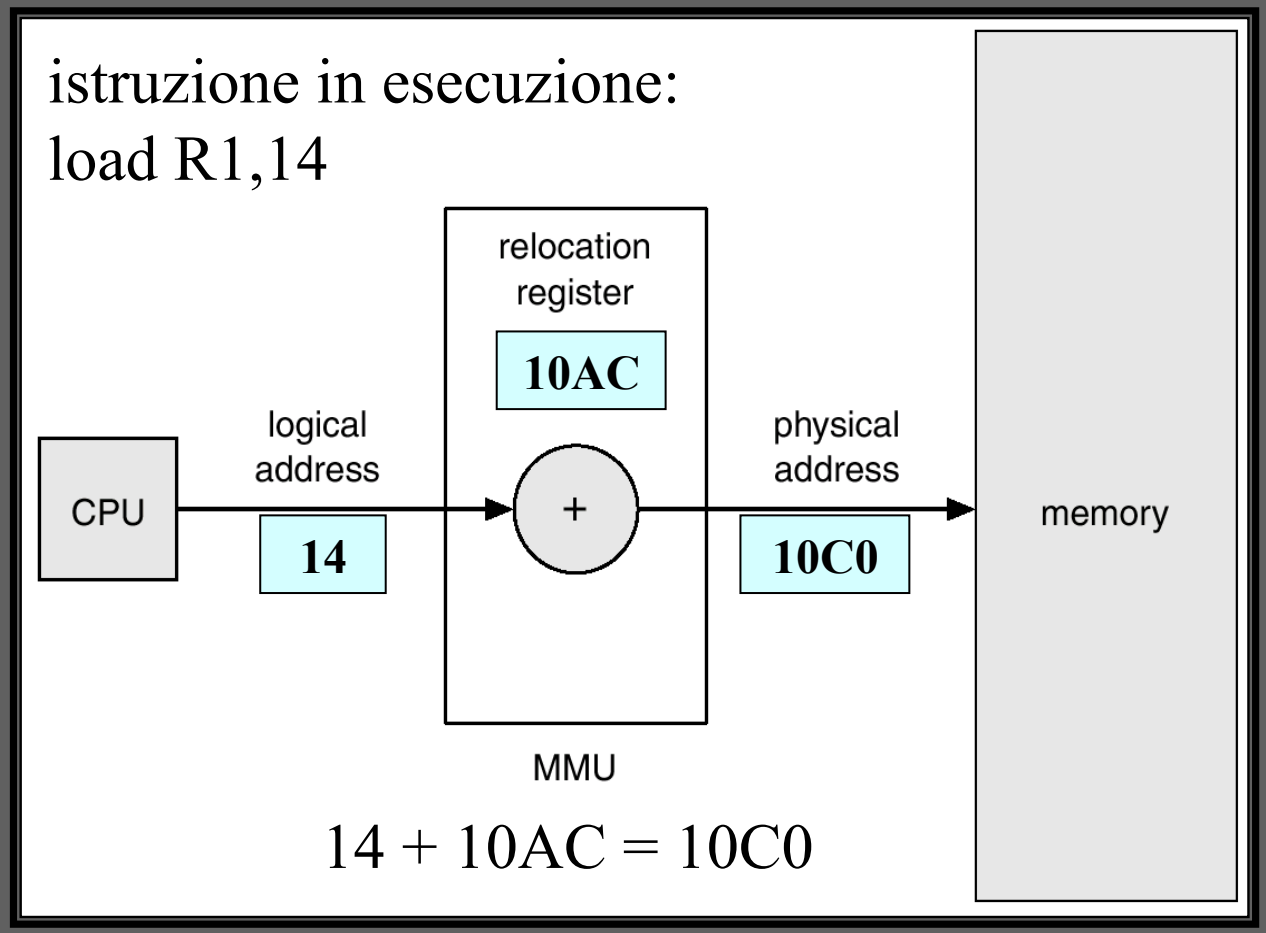
```
10A0 ... ... ... ... ...
10A4 ... ... ... ... ...
10A8 ... ... ... ... ...
10AC load R1,14
10B0 jmp_if_odd R1, C
10B4 add R1,#1
10B8 mul R1,#2
10BC store R1,14
10C0 /* four byte int. var */
10C4 ... ... ... ... ...
10C8 ... ... ... ... ...
10CC ... ... ... ... ...
```

9.1.2 Binding degli indirizzi: quando?

- **In fase di esecuzione (cont.)**
  - un opportuno **registro di rilocazione** viene usato per trasformare un indirizzo relativo nel corrispondente indirizzo assoluto **durante l'esecuzione delle istruzioni**.
  - Il registro di rilocazione contiene l'indirizzo di partenza dell'area di RAM in cui è caricato il programma in esecuzione.
  - La **memory management Unit (MMU)** si occuperà di trasformare gli indirizzi relativi in assoluti, usando il registro di rilocazione, per accedere alle celle di RAM indirizzate dalle istruzioni

9.1.2 indirizzi dinamicamente rilocabili

- Per spostare i programmi da un'area di RAM ad un'altra ora basta cambiare l'indirizzo scritto nel registro di rilocazione (fig. 9.5 modificata)



9.1.2 indirizzi dinamicamente rilocabili

- **In fase di esecuzione (cont.)**
  - Lo spostamento del processo da un area all'altra della MP è realizzabile senza problema.
  - Il SO deve solo ricordarsi dell'indirizzo della locazione di MP a partire dalla quale è memorizzato il processo
- Questo è l'approccio adottato in quasi tutti i SO moderni (in realtà, come vedremo, ne viene adottata una variante molto sofisticata, ma il concetto di base è questo)

9.1.2 Binding degli indirizzi

- Riassumendo, quindi:
- nel **codice statico** gli indirizzi nascono assoluti
- nel **codice staticamente rilocabile** gli indirizzi nascono relativi e vengono trasformati in assoluti in fase di caricamento del programma in RAM
- nel **codice dinamicamente rilocabile** gli indirizzi nascono relativi e rimangono tali anche quando il programma viene caricato in RAM ed eseguito

9.1.3 Spazio degli indirizzi Logici e Fisici

- Consideriamo codice dinamicamente rilocabile (d'ora in poi faremo sempre riferimento a codice dinamicamente rilocabile, se non indicato diversamente). Ogni indirizzo usato nel codice è riferito ad un ipotetico indirizzo 0 (zero): l'indirizzo della prima istruzione di cui è formato il codice.

```
0000  load R1,14
0004  jmp_if_odd R1,C
0008  add R1,#1
000C  mul R1,#2
0010  store R1,14
0014/* four byte int. var */
```

9.1.3 Spazio degli indirizzi Logici e Fisici

- Qualsiasi indirizzo usato in quel codice, che potrà essere:
  - *l'indirizzo di una cella di memoria contenente una delle variabili del codice* oppure
  - *l'indirizzo specificato in una istruzione di salto del codice, e che trasferisce il controllo ad un'altra istruzione del codice*
- avrà un valore compreso tra 0 e l'indirizzo *max* dell'ultima "cella di memoria" occupata dal programma (codice e dati).
- Questo insieme di indirizzi prende il nome **di spazio di indirizzamento logico** o **virtuale** del programma o **spazio degli indirizzi logici** (o **virtuali**) del programma.

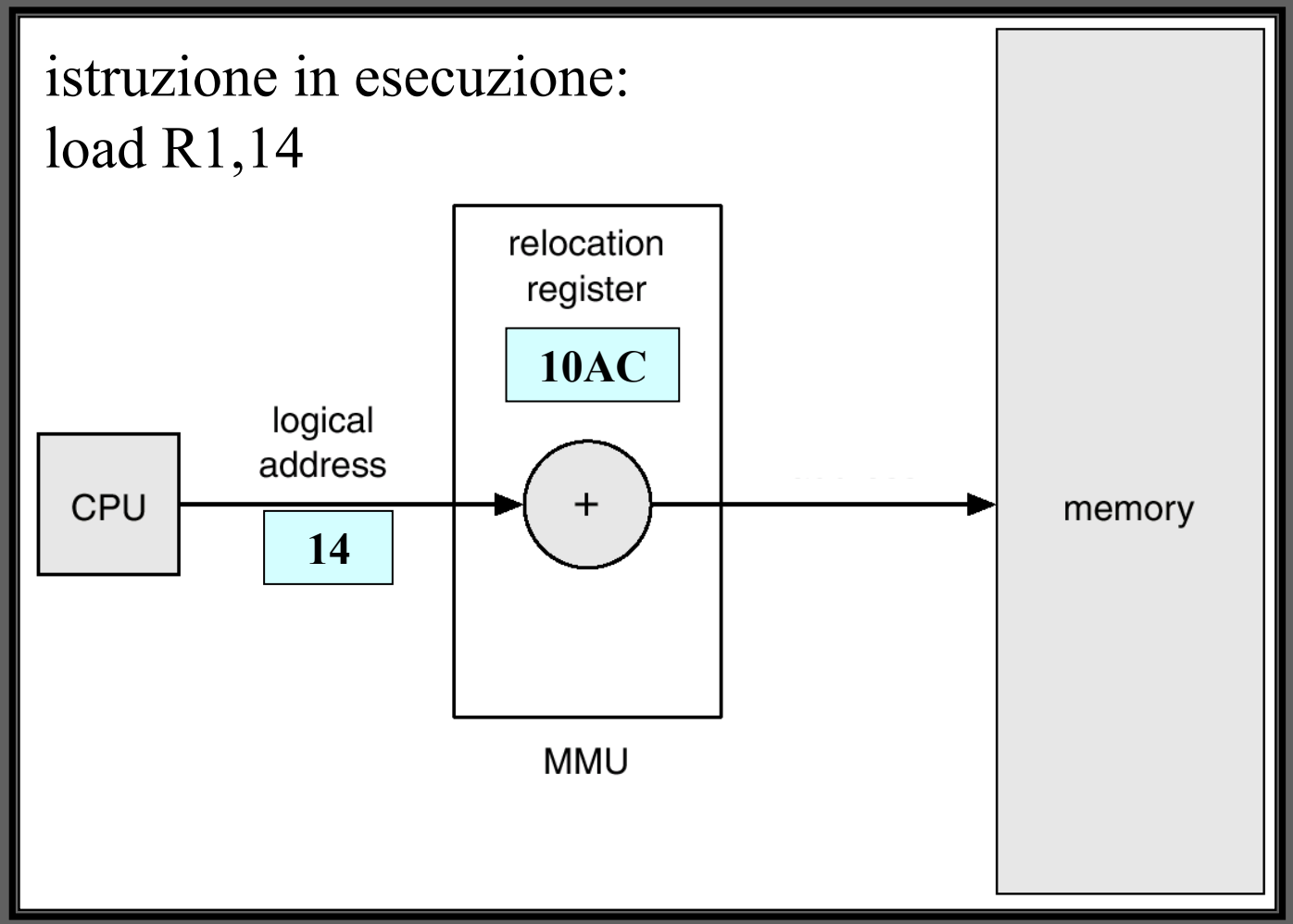
9.1.3 Spazio degli indirizzi Logici e Fisici

- Il codice viene caricato in RAM ed eseguito, e siccome è dinamicamente rilocabile, gli indirizzi usati nel codice rimangono relativi, cioé continuano a fare riferimento all'ipotetico indirizzo 0 della prima cella di memoria occupata dal codice.

- Per coerenza con la terminologia del lucido precedente, invece di parlare di indirizzi relativi parliamo di:

- **Indirizzi logici** (o **virtuali**): indirizzi generati dalla CPU (in termini approssimativi, indirizzi che "escono" dalla CPU per raggiungere la memoria RAM, attraverso la MMU).

9.1.3 Spazio degli indirizzi Logici e Fisici

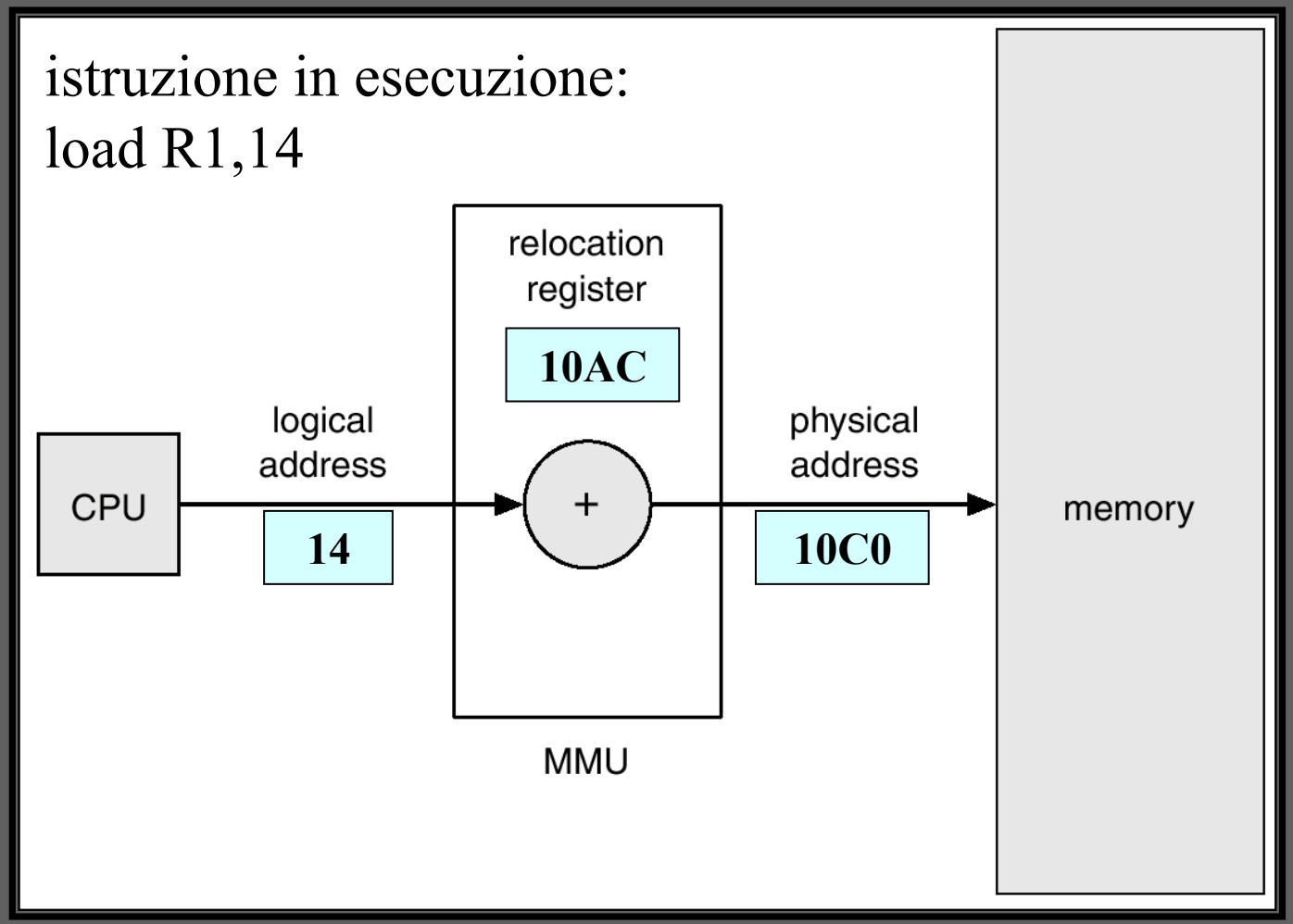


9.1.3 Spazio degli indirizzi Logici e Fisici

- Il programma è caricato dal SO in RAM a partire da una cella di memoria con indirizzo diverso da 0, dunque per poter indirizzare correttamente la RAM, gli indirizzi logici usati nell'istruzione in esecuzione, devono prima passare attraverso il registro di rilocazione, in modo da venire trasformati in:

- **Indirizzi fisici:** indirizzi caricati nel ***memory address register*** **(MAR)** per indirizzare una cella della memoria centrale (in termini approssimativi, un indirizzo che esce dalla MMU e arriva alla RAM, identificando una ben precisa cella di memoria RAM)

9.1.3 Spazio degli indirizzi Logici e Fisici



9.1.3 Spazio degli indirizzi Logici e Fisici

- Ma allora, analogamente a come abbiamo definito lo spazio di indirizzamento logico (o virtuale) del programma come l'insieme degli indirizzi logici usati dal programma, possiamo definire:

- Lo **spazio di indirizzamento fisico** del programma: l'insieme degli indirizzi fisici usati dal programma e dai suoi dati (che evidentemente dipende dall'area di memoria in cui il SO ha caricato il programma stesso).

9.1.3 Spazio degli indirizzi Logici e Fisici

- Dunque, per un programma per cui è stato generato codice dinamicamente rilocabile abbiamo due tipi di indirizzi:
  - gli indirizzi logici del programma, che spaziano nell'intervallo tra 0 e *max*
  - Gli indirizzi fisici, che spaziano nell'intervallo tra r + 0 ed r + *max,* se r è l'indirizzo di memoria a partire dal quale è stato caricato il programma in RAM.
- Il programma utente genera solo indirizzi logici e "pensa" che il processo sia eseguito nelle posizioni da 0 a *max*. invece, gli indirizzi logici sono mappati in indirizzi fisici prima di poter indirizzare correttamente la RAM.

spazio degli indirizzi fisici del programma: da 10AC a 10C3

spazio degli indirizzi logici del programma: da 0000 a 0017 (perché 0017?)

```
0000 load R1,14
0004 jmp_if_odd R1,C
0008 add R1,#1
000C mul R1,#2
0010 store R1,14
0014 /* four byte int. var */
```

RAM:

```
10A0 ... ... ... ... ...
10A4 ... ... ... ... ...
10A8 ... ... ... ... ...
10AC load R1,14
10B0 jmp_if_odd R1,C
10B4 add R1,#1
10B8 mul R1,#2
10BC store R1,14
10C0 /* four byte int. var */
10C4 ... ... ... ... ...
10C8 ... ... ... ... ...
10CC ... ... ... ... ...
```

9.1.3 Spazio degli indirizzi Logici e Fisici

- In realtà, le espressioni
  - *spazio di indirizzamento logico* (o *virtuale*) e
  - *spazio di indirizzamento fisico*
- vengono usate non tanto in riferimento ad un singolo programma, ma principalmente nei riguardi di una architettura nel suo complesso.

9.1.3 Spazio degli indirizzi Logici e Fisici

- Consideriamo ad esempio uno specifico computer che possa essere dotato **al massimo** di 64 Kbyte di RAM (più propriamente, 65536 byte). Per quel computer, in maniera del tutto equivalente possiamo dire che:

1. Il computer può indirizzare 2^{16} byte di RAM
2. gli indirizzi dei byte della RAM vanno da 0000 a FFFF esadecimale (cioè da 0 a $2^{16} - 1$)
3. l'indirizzo di un qualsiasi byte della RAM è scritto su 16 bit (abbiamo bisogno di 16 bit per indirizzare uno qualsiasi dei byte della RAM)

9.1.3 Spazio degli indirizzi Logici e Fisici

- Ossia, possiamo dire che **lo spazio di indirizzamento fisico** di quel computer è scritto su 16 bit, o che va da 0000 a FFFF (esadecimale), o che è di 64 Kbyte.

- Infatti, un qualsiasi programma caricato nella RAM di quel computer, ed in particolare un programma che occupi completamente la RAM del computer (ignoriamo per un momento la presenza del sistema operativo) non potrà generare altro che indirizzi fisici che vanno da un minimo di 0000 ad un massimo di FFFF.

9.1.3 Spazio degli indirizzi Logici e Fisici

- Consideriamo ora un compilatore che generi codice dinamicamente rilocabile per quel computer e supponiamo che usi, ad esempio, 12 bit per scrivere un indirizzo logico.
- Allora, lo spazio di indirizzamento logico di un qualsiasi programma di quel computer sarà al massimo di 2^{12} byte, ossia 4 Kbyte.
- Ovviamente, un programma di quel computer potrà usare uno spazio degli indirizzi logici più piccolo di 4 Kbyte, ma nessun programma (codice + dati) potrà usare uno spazio maggiore di 4 Kbyte (riuscite a vedere perché?)

9.1.3 Spazio degli indirizzi Logici e Fisici

- Diciamo dunque che **lo spazio di indirizzamento logico** a disposizione dei programmi di quel computer è scritto su 12 bit, o che va da 0000 a 0FFF (esadecimale), o è di 4 Kbyte.
- Nel seguito, parlando di spazio di indirizzamento logico o fisico, ci riferiremo a quello della macchina, e non a quello dei singoli programmi che vi girano sopra. Ma in modo equivalente, potremo parlare di un programma che occupa tutto lo spazio di indirizzamento logico della macchina.
- Domanda: in un sistema reale, ha senso che la dimensione dello spazio di indirizzamento fisico sia diverso dalla dimensione dello spazio di indirizzamento logico?

9.1.3 Spazio degli indirizzi Logici e Fisici

- Certo, e anzi questo è il caso più frequente. I processori moderni a 64 bit hanno spazi di indirizzamento fisico molto grandi, in diverse architetture può andare da 2^{40} a 2^{64} byte. Ossia vengono usati da 40 a 64 bit per scrivere un indirizzo fisico.
- Perché non si usano sempre 64 bit per un indirizzo fisico? Sostanzialmente perché sono fin troppi…
- Oltretutto, è molto improbabile che un computer abbia una quantità effettiva di RAM pari al massimo indirizzabile dal suo processore (anche "solo" 2^{40} byte = 1 Terabyte = 1000 Gigabyte!)

9.1.3 Spazio degli indirizzi Logici e Fisici

- Diversi sistemi operativi, e le relative applicazioni, usano gli spazi di indirizzamento virtuali che i progettisti ritengono più adeguati, anche qui con valori che vanno, ad esempio, da 2^{48} a 2^{64} byte. Ossia vengono usati da 48 a 64 bit per scrivere un indirizzo virtuale.

- Nel caso più generale, per un computer moderno vale:

  $|RAM| \neq |physical\ addr.\ space| \neq |virtual\ addr.\ space|$

  e, di solito:

  $|RAM| < |physical\ addr.\ space| < |virtual\ addr.\ space|$

9.1.3 Spazio degli indirizzi Logici e Fisici

- In realtà, vincoli architetturali e dimensionali di vario tipo limitano ulteriormente la quantità di RAM di cui un normale computer (ed in particolare un portatile) può essere dotato, per cui di solito vale anche:

$$|RAM|_{effettiva} \leq |RAM|_{max} << |phis.\ space| < |virt.\ space|$$

(attenzione, non sempre |phisical space| < |virtual space|, ad esempio negli Intel core i7 lo spazio di indirizzamento fisico è scritto su 52 bit, e quello logico su 48 bit).

9.1.3 Spazio degli indirizzi Logici e Fisici

- Se un certo sistema ha uno spazio di indirizzamento virtuale di X byte, questo vuol sostanzialmente dire che in quel sistema possiamo scrivere un programma il quale, insieme ai dati indirizzati da quel programma, occupa in tutto X byte (in altre parole, gli indirizzi virtuali usati in quel programma vanno da 0 a X-1)

- Ma il programma può girare su una macchina in cui:

  |RAM| < |phisical addr. space| < X?

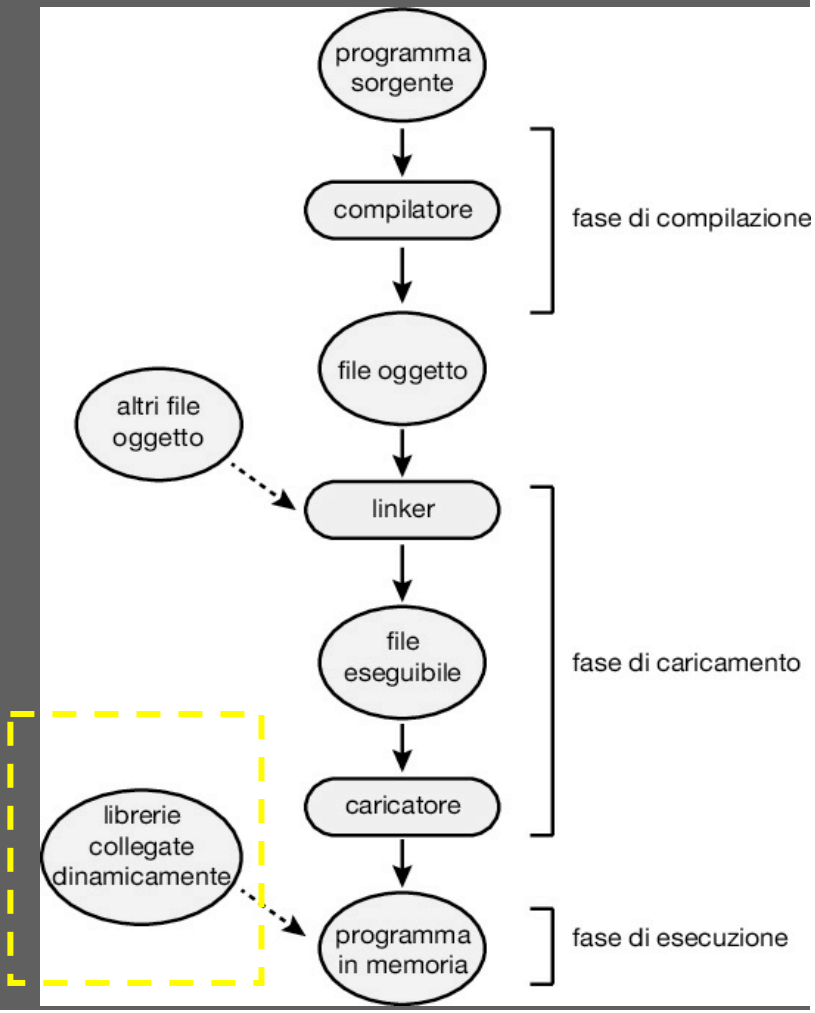
- Torneremo su questa domanda nel capitolo sulla memoria virtuale

9.1.5 Le Librerie

- Nella figura 9.3 che mostra i passi necessari per trasformare un programma sorgente in un eseguibile, vedete indicate anche una operazione che riguarda le **librerie** usate dal programma

- Una libreria è una collezione di subroutine di uso comune messe a disposizione dei programmatori per sviluppare software

- Ad esempio, una libreria matematica metterà a disposizione funzioni per eseguire calcoli matematici, come la funzione sqrt(x) della libreria matematica del C che permette di calcolare la radice quadrata di un numero



9.1.5 Le Librerie

- Le librerie sono comode, perché mettono a disposizione "pezzi di codice" usati comunemente, e che quindi non devono essere riscritti ogni volta da ogni programmatore che ne abbia bisogno.
- Notate che "libreria" è la traduzione sbagliata, ma ormai entrata nell'uso, di "library".
  Ne esistono fondamentalmente di due tipi, ma il primo tipo e ormai obsoleto e non è quasi più usato:

1. **Librerie statiche:** le cui subroutine vengono collegate al codice del programma principale (come un qualsiasi altro modulo oggetto) dal compilatore o dal loader (caricatore) e diventano parte dell'eseguibile.

9.1.5 Le Librerie

- Il problema delle librerie statiche è che se N programmi usano le stesse subroutine di libreria, ciascun programma ne incorporerà una copia, duplicando così più volte lo stesso codice (sull'hard disk, e soprattutto in RAM, se quei programmi vengono eseguiti contemporaneamente)
- Inoltre, il codice di una libreria statica viene caricato in RAM anche se, per qualsiasi ragione, il programma che usa quella libreria non chiama mai (in una particolare esecuzione del codice) nessuna delle subroutine della libreria, che quindi occupa inutilmente spazio in RAM.

9.1.5 Le Librerie

2. **Librerie dinamiche:** vengono caricate in RAM solo nel momento in cui (e quindi solo se) il programma che le usa chiama una delle subroutine della libreria di cui ha bisogno. Ossia, vengono caricate in RAM a run-time.

- Un programma che usa una libreria dinamica specifica solamente, all'interno del suo codice, il nome della subroutine di libreria che vuole chiamare in quel punto.
- Se (e solo quando) il programma che sta usando la CPU deve eseguire la subroutine, il SO carica la libreria che la contiene nello spazio di memoria assegnato a quel processo per permetterne l'esecuzione del codice.

9.1.5 Le Librerie

- Di solito il SO cerca le librerie dinamiche in cartelle di sistema predefinite, ma il programma che usa una libreria può specificare ulteriori directory in cui quella libreria può essere presente.
- Un vantaggio fondamentale delle librerie dinamiche è che il loro codice può essere facilmente condiviso tra più processi che le invocano, evitando così inutile duplicazione di codice in RAM. Per questa ragione, le librerie dinamiche sono anche dette **librerie condivise**.
- Un altro vantaggio è che nuove versioni delle librerie dinamiche possono sostituire le vecchie, senza che i programmi che le usano debbano essere ricompilati.

9.1.5 Le Librerie

- In ambiente Unix (e Linux, e Solaris) le librerie dinamiche usano normalmente l'estensione **.so** (shared object), e si trovano di solito nella directory "/lib"

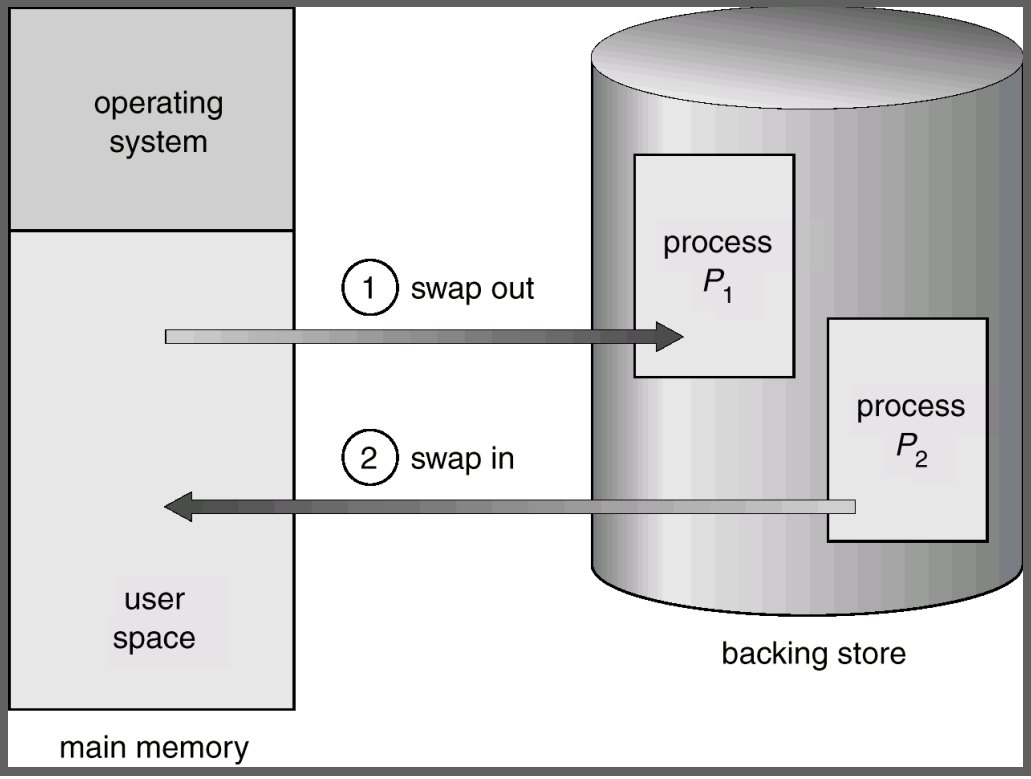
- In ambiente Windows, le librerie dinamiche hanno la estensione **.DLL** (Dynamic Link Library), e molte stanno nella cartella "C:\WINDOWS\system32".

Tecniche di gestione della memoria primaria

- vedremo ora le principali tecniche di gestione della MP, dalle più semplici alle più complesse. Alcune di queste non sono più in uso (o quasi), ma ci permettono di capire concetti più sofisticati a partire da idee semplici:
  - Swapping
  - allocazione contigua a partizioni multiple fisse
  - allocazione contigua a partizioni multiple variabili
  - Paginazione
  - Paginazione a più livelli

Avvicendamento di Processi (swapping)

- Conosciamo già l'idea: salvare in memoria secondaria l'immagine di un processo *non in esecuzione* (**swap out**) e ricaricarla (**swap in**) *prima* di dargli la CPU



Swapping

- Lo swapping permette di avere attivi più processi di quanti ne possa contenere lo spazio della MP:
- Alcuni processi sono mantenuti temporaneamente in un'area dell'hard disk detta **area di swap** (ad uso esclusivo del SO)
- Se il processo "**swappato**" viene ricaricato in una diversa area di MP, dobbiamo usare codice dinamicamente rilocabile

Swapping

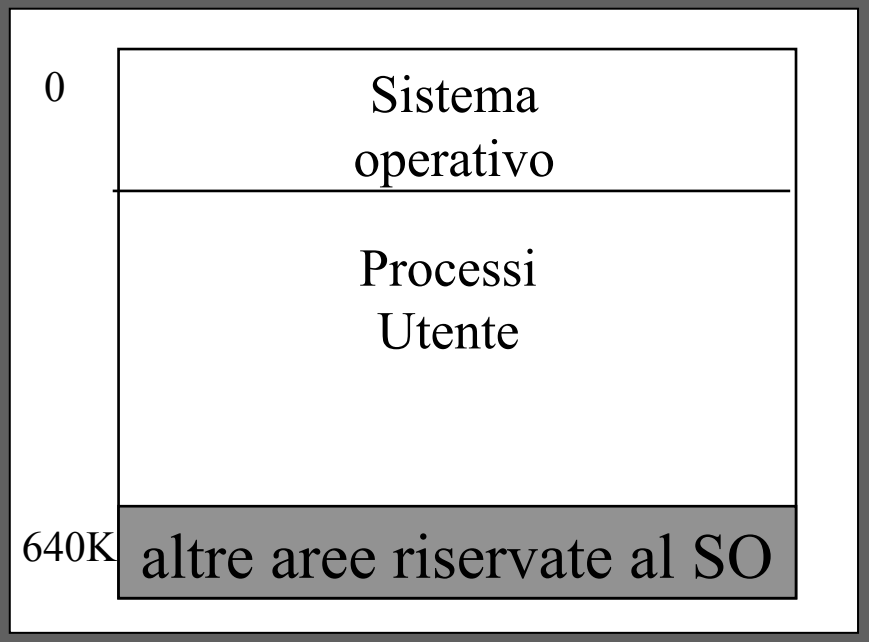
- La maggior parte del tempo di swap è tempo speso per copiare codice e dati di un processo dall'hard disk alla RAM e viceversa.
- I tempi richiesti sono nell'ordine dei millisecondi, quindi con ritardi altissimi, se consideriamo che in un millisecondo il singolo core di una CPU moderna può eseguire milioni di istruzioni
- Dunque, l'overhead che ne risulta è generalmente inaccettabile!

Swapping

- Eccetto casi molto rari, dunque, lo swapping di interi processi non è sostanzialmente più usato nei moderni SO.
- ma l'idea di fondo dello swapping rimane valida: usare parte della memoria secondaria per estendere le dimensioni della memoria primaria, e permettere così l'esecuzione di più processi di quanti ne potrebbe ospitare la sola RAM.
- Riprenderemo quest'idea quando parleremo della memoria virtuale (cap. 10).

9.2 Allocazione contigua della Memoria Primaria

- In un qualsiasi computer, la memoria principale è divisa in due partizioni, assegnate rispettivamente al SO e ai processi.
- Il SO si colloca nella stessa area di memoria puntata dal vettore delle interruzioni (cap. 1), che di solito è allocato nella parte "bassa" della memoria

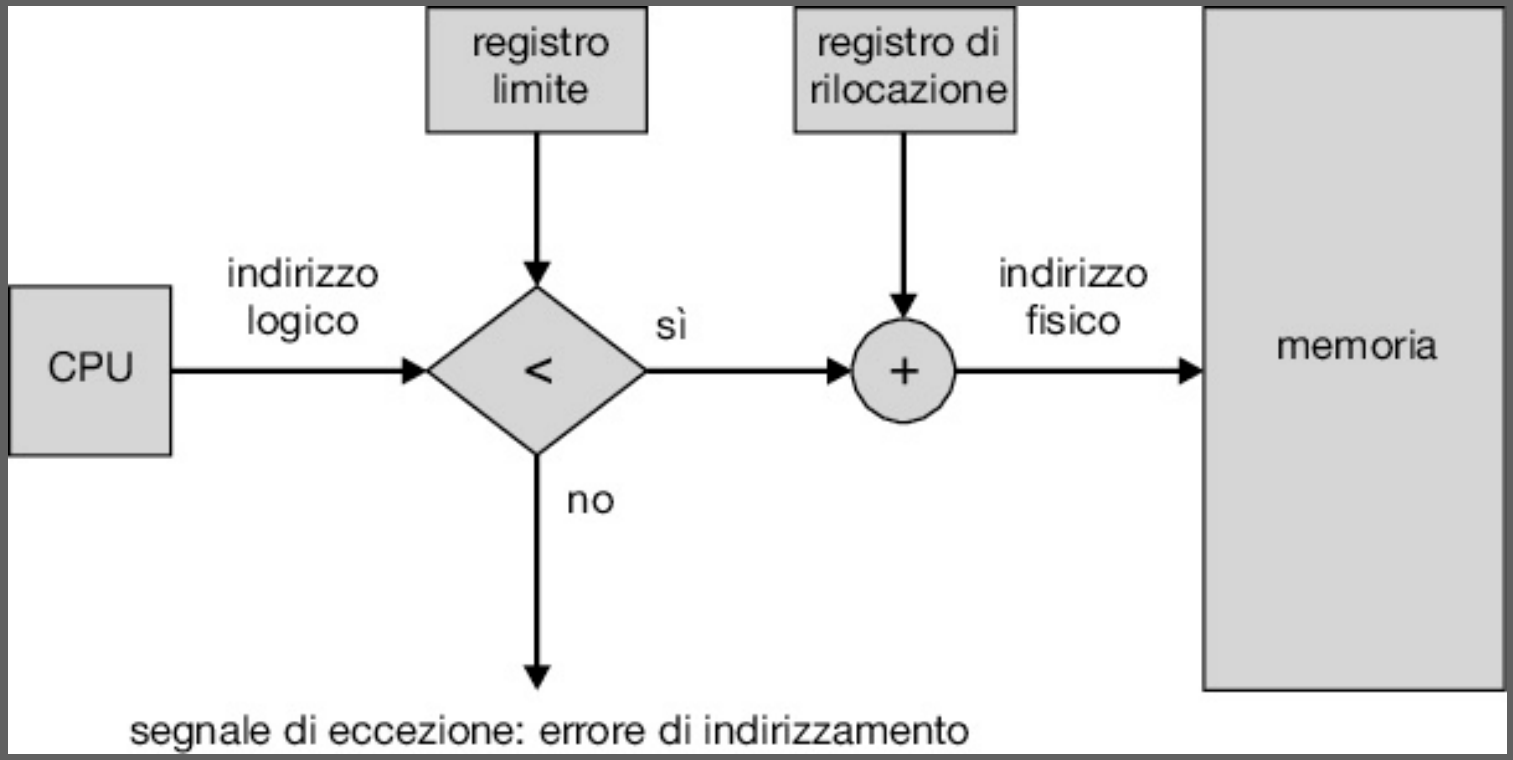


9.2.1 Protezione della memoria

- Nel caso più semplice, l'area non assegnata al SO viene occupata da un solo processo (era questa la soluzione adottata nei primi sistemi operativi MS-dos).
- La protezione della MP in questo caso coincide con la protezione delle aree di memoria del SO
- Un **registro limite** (vedi cap. 1) viene inizializzato dal SO: ogni indirizzo logico usato dal processo utente deve essere inferiore al valore scritto nel registro limite.
- Poiché assumiamo l'uso di codice dinamicamente rilocabile, il registro di rilocazione viene poi usato per generare l'effettivo indirizzo fisico.

9.2.1 Protezione della memoria

- R. di rilocazione = 100.040; R. limite = 74.600
- Gli indirizzi fisici corretti vanno da 100.040 a 174.640 (fig. 9.6)



9.2.2a Allocazione a partizioni multiple fisse

- La memoria è divisa in partizioni di dimensione fissa (non necessariamente tutte uguali)
- Ogni partizione contiene un unico processo, dall'inizio alla fine dell'esecuzione
- Il numero delle partizioni stabilisce il grado di multiprogrammazione
- Quando un processo termina, il suo posto può essere preso da un altro processo

9.2.2a Allocazione a partizioni multiple fisse

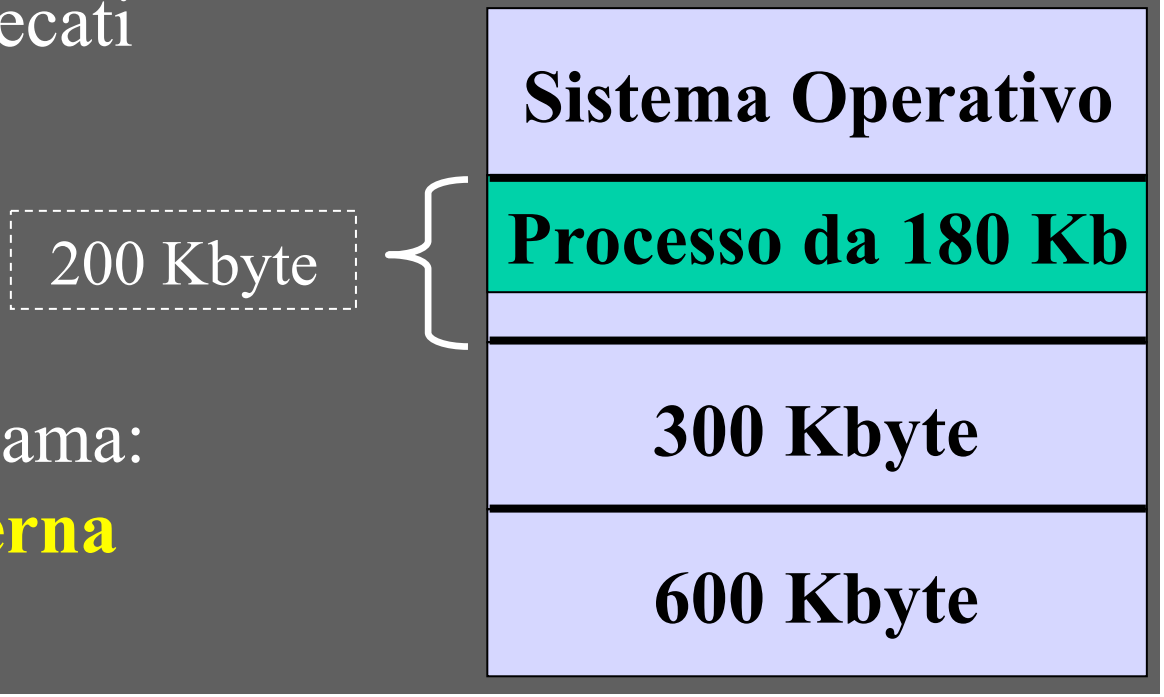
- Il meccanismo dei registri limite e di rilocazione può essere usato per proteggere le varie partizioni da accessi proibiti.
- Al context switch, il dispatcher carica nel registro di rilocazione l'indirizzo di partenza della partizione del processo a cui viene data la CPU
- E carica nel registro limite la dimensione di quella partizione (o del processo contenuto nella partizione?)

9.2.2a Allocazione a partizioni multiple fisse

- Questa tecnica era adottata, ad esempio, nel SO IBM OS/360 (notate quindi che per funzionare correttamente aveva bisogno di una CPU dotata di registro di rilocazione e registro limite).
- Attualmente non è più in uso per gestire i normali processi utente nei SO moderni, perchè presenta troppi svantaggi. Infatti:
- Il grado di multiprogrammazione è limitato dal numero di partizioni previste. Si potrebbe usare la tecnica dello swapping, ma come abbiamo visto, questo produce un eccessivo overhead.

9.2.2a Partizioni multiple fisse: la frammentazione interna

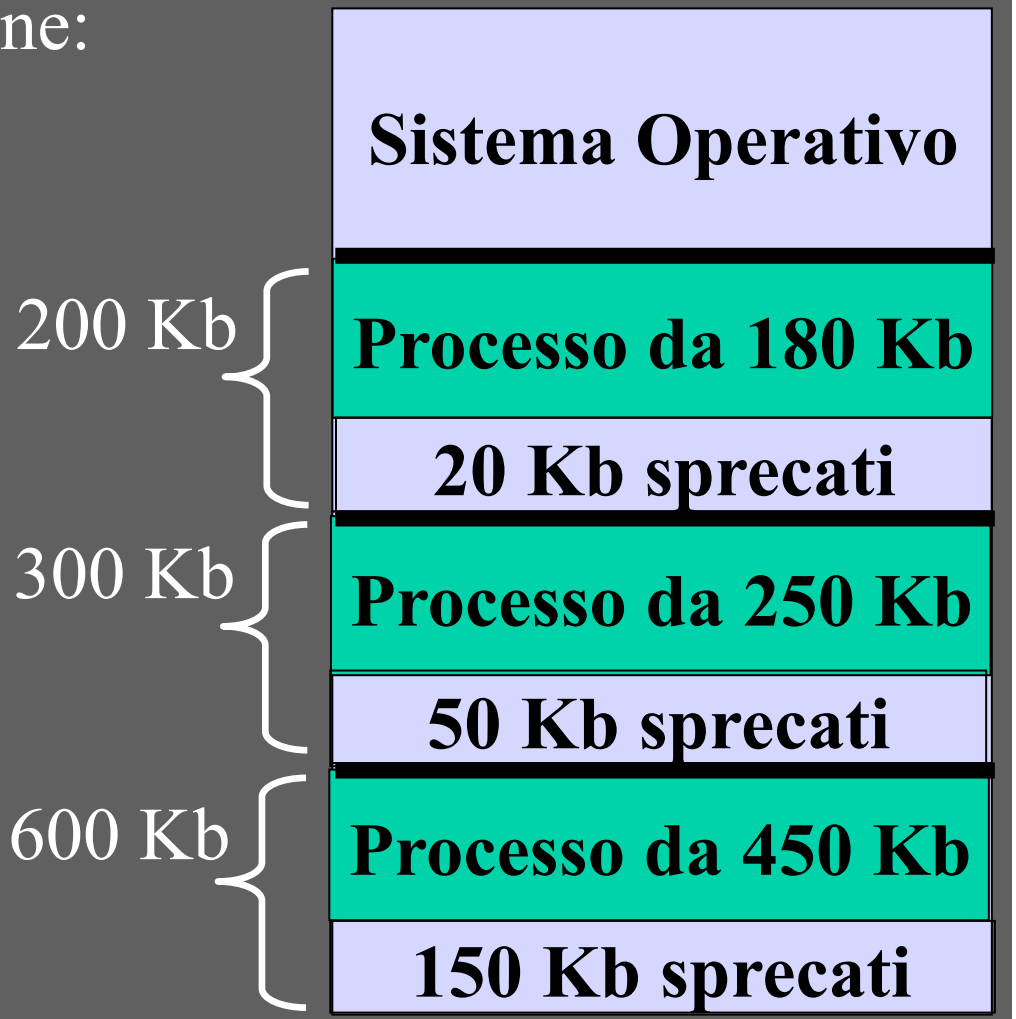
- Inoltre, difficilmente un processo messo in una partizione ha esattamente la dimensione di quella partizione.
  Di conseguenza, la parte che rimane viene sprecata.
  Nell'esempio, nella partizione da 200Kbyte
  20 Kbyte vengono sprecati



- Questo problema si chiama:
  Frammentazione Interna

9.2.2a Partizioni multiple fisse: la frammentazione esterna

- Considerate questa situazione:
- Ci sarebbe spazio per un processo da 20+50+150=220 Kbyte, ma le tre aree libere di RAM non sono contigue, e questa porzione di memoria rimane sprecata.
- Questo problema si chiama: **Frammentazione Esterna**

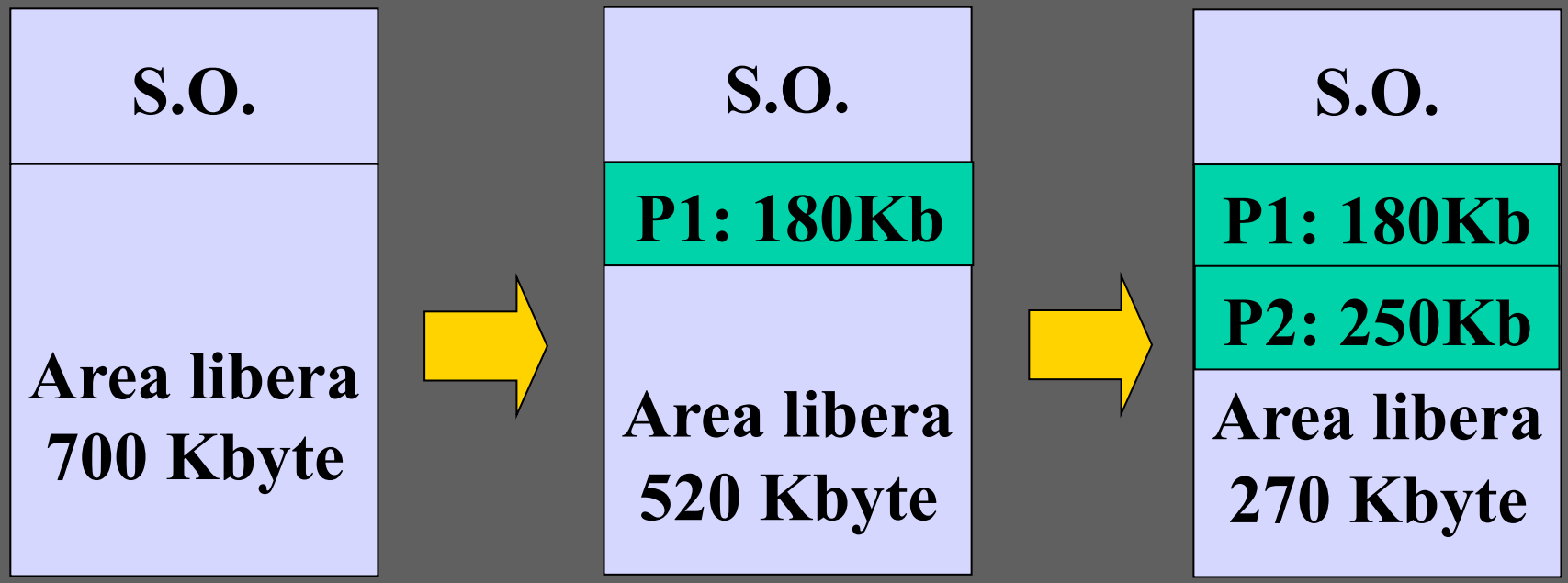


9.2.2a Allocazione a partizioni multiple fisse

- L'allocazione a partizioni fisse ha anche altri problemi:
- Che succede se arriva un processo più grande della partizione più grande?
- Notate che se si aumenta la dimensione media delle partizioni, aumenta anche la frammentazione interna, e diminuisce il grado di multiprogrammazione

9.2.2b Allocazione a partizioni multiple variabili

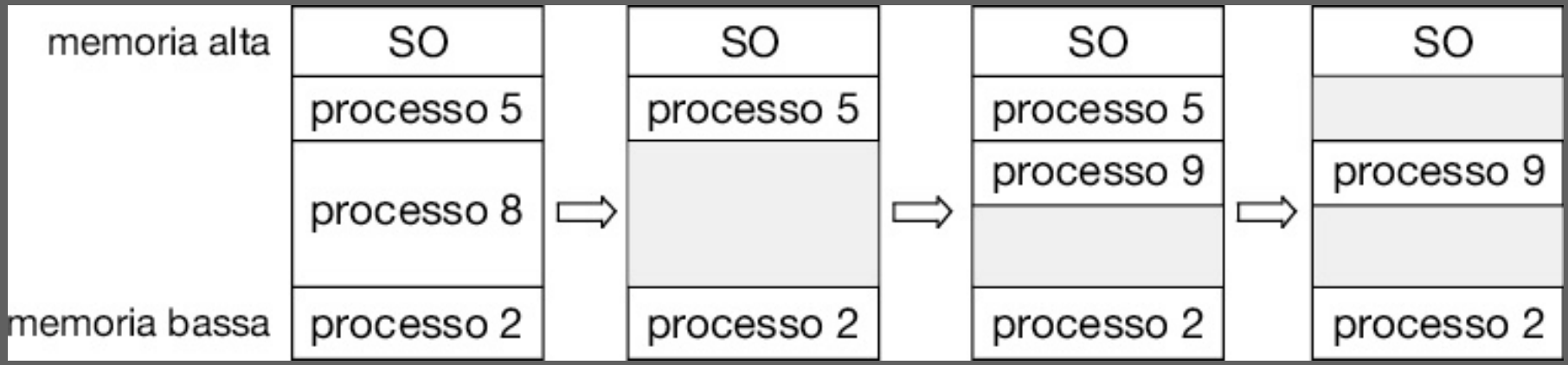
- Ma allora, perché non usare partizioni a dimensione variabile? Un processo riceve una quantità di memoria esattamente pari alla sua dimensione.



- **la frammentazione interna scompare? Vediamo…**

9.2.2b Allocazione a partizioni multiple variabili

- Quando un processo termina lascia un “buco” in RAM, e un altro processo può occupare una parte dello spazio liberato:



- Durante il normale lavoro del SO, in cui vecchi processi terminano e nuovi processi nascono, si formano nella RAM buchi sparsi sempre più piccoli e difficili da riempire... (fig. 9.7)

9.2.2b Allocazione a partizioni multiple variabili

- Per gestire l'allocazione a partizioni multiple variabili Il SO deve ovviamente tenere nota di tutti i buchi liberi e occupati, e aggiornare l'informazione ogni volta che nasce o termina un processo.

- Quando un processo deve essere caricato in RAM, il SO deve assegnargli una partizione di dimensione sufficiente, se ce n'è una...

- Quale strategia usare per scegliere una partizione da assegnare ad un processo?

9.2.2b Allocazione a partizioni multiple variabili

- **First Fit:** scegli *la prima partizione abbastanza grande* da poter ospitare il processo
- **Best Fit:** scegli *la più piccola partizione abbastanza grande* da poter ospitare il processo
- **Worst Fit:** scegli *la partizione più grande*
- Sperimentalmente, si è visto che Worst Fit è quello che funziona peggio in termini di spreco della MP. Best Fit e First Fit sono simili, per cui tanto vale usare First Fit (perchè?).

9.2.3 Allocazione a partizioni multiple variabili: la frammentazione

- Il problema della **frammentazione esterna** rimane: col tempo si formano tanti piccoli buchi liberi non contigui che non possono ospitare alcun processo
- Statisticamente, da 1/3 a 1/2 della MP può venire sprecato in questo modo
- Ma c'è anche un problema di **frammentazione interna**: costa troppo tenere traccia dei buchi molto piccoli, e quindi si agganciano ad una partizione adiacente. Come conseguenza, si crea uno spreco nascosto

9.2.3 Allocazione a partizioni multiple variabili: la compattazione

- Come possiamo recuperare i buchi di RAM inutilizzati? **compattando la memoria**: spostiamo le partizioni occupate dai processi in modo che queste siano tutte contigue fra di loro. Si forma così un unico grande buco libero
- Il compattamento richiede **rilocazione** dei processi, sia del codice che dei dati. Questo quindi è possibile solo se si usa codice dinamicamente rilocabile
- La compattazione può richiedere molto tempo e lavoro al SO. Durante la compattazione il sistema è inusabile

9.2.3 Allocazione a partizioni multiple variabili: la compattazione

- Ci sono diversi modi di compattare la memoria principale:
- spostare tutti i processi ad un estremo della memoria
- spostare i processi in buchi già esistenti verso gli estremi della memoria (statisticamente si sposta meno memoria)
- Spostare solo quanto necessario per fare entrare un nuovo processo che altrimenti non avrebbe spazio sufficiente.
- La tecnica della allocazione a partizioni multiple variabili e la relativa compattazione era usata ad esempio nei sistemi PDP della Digital Equipment Corp. degli anni ‘70

9.2.3 Allocazione a partizioni multiple variabili: la compattazione

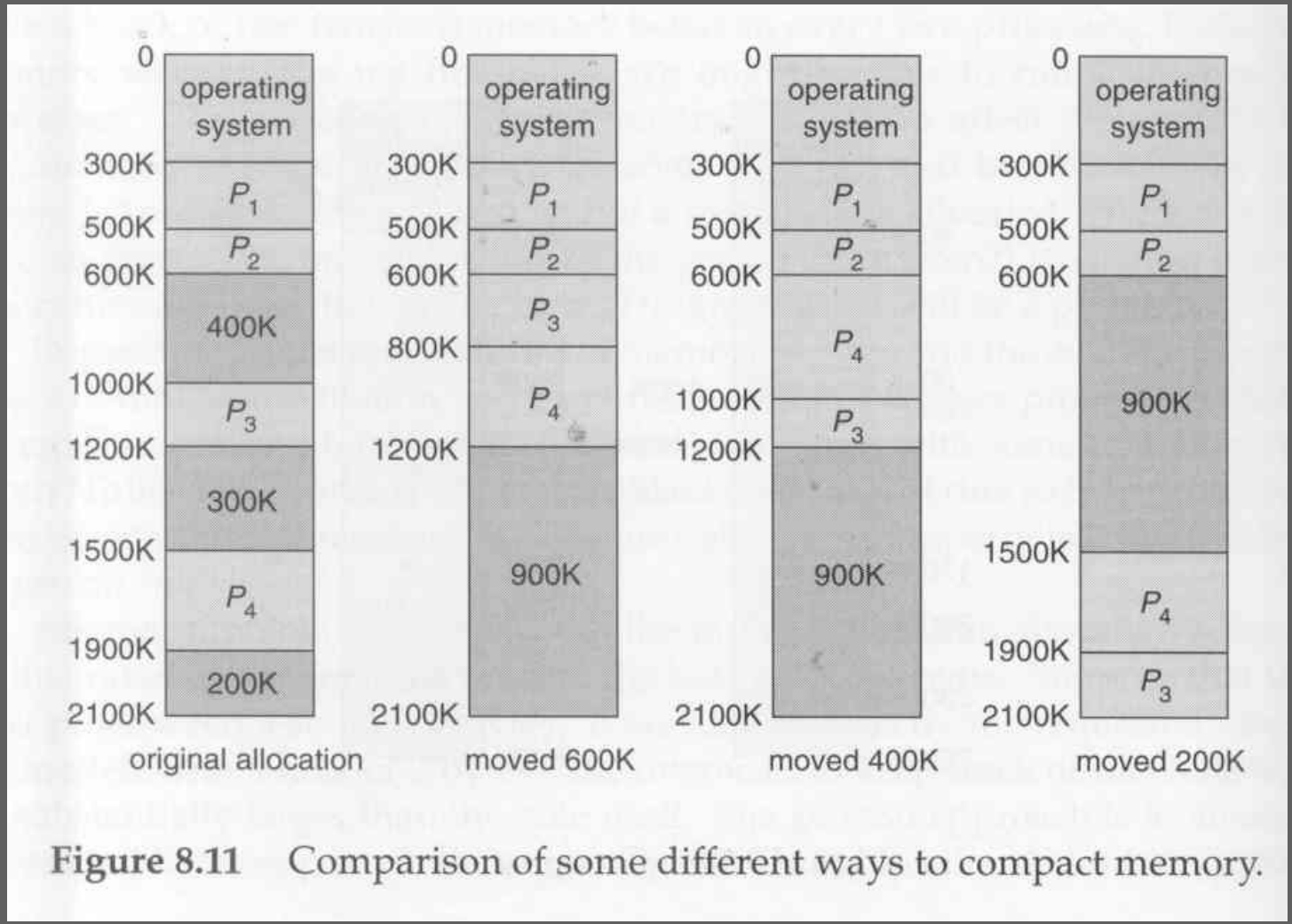


Figure 8.11 Comparison of some different ways to compact memory.

9.3 Paginazione della memoria

- L'allocazione contigua della memoria principale presenta quindi diversi problemi.
- L'alternativa è ammettere che l'area di memoria allocata ad un processo possa essere in realtà suddivisa in tanti pezzi **non contigui** fra loro
- Se tutti i "pezzi" hanno la stessa dimensione allora il termine esatto per indicare questa tecnica è: **paginazione della memoria (primaria)**

9.3.1 Paginazione: metodo di base

- La Memoria Primaria (o più propriamente, lo spazio di indirizzamento fisico, ma d'ora in poi consideriamo i due concetti come equivalenti) è divisa in "pezzi" detti **frame** (o **pagine fisiche**), tutti della stessa dimensione (noi useremo sempre il termine *frame*)
- La dimensione è sempre una potenza di due (512, ..., 8192 byte). Come vedremo, questo semplifica tantissimo la gestione del meccanismo.
- Lo spazio di indirizzamento logico viene sempre visto dal processo come uno spazio contiguo (es.: indirizzi da 000000 a 111111), ma è diviso in **pagine** di dimensione identica ai frame

prog1:

0	0000 load R1,14 0004 jmp_if_odd R1,C
1	0008 add R1,#1 000C mul R1,#2
2	0010 store R1,14 0014 /* four byte int. var */

lo spazio di indirizzamento logico di *prog1* va da 0000 a 0017. Lo spazio di indirizzamento fisico complessivo va da 0000 a 0037. Macchine diverse possono naturalmente avere una diversa quantità di RAM, e quindi uno spazio fisico diverso. Notate che pagine e frame sono numerati consecutivamente.

RAM:

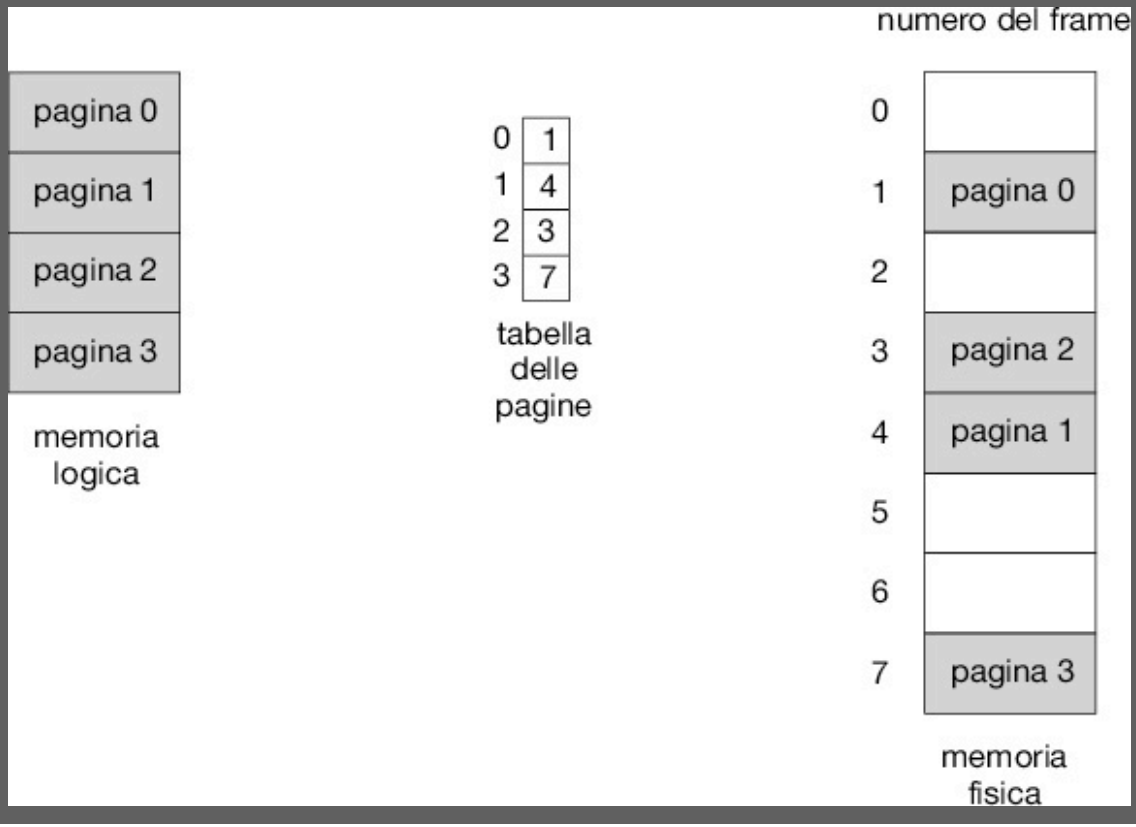
0000 ... ... 0004	0
0008 ... ... 000C	1
0010 ... ... 0014	2
0018 ... ... 001C	3
0020 ... ... 0024	4
0028 ... ... 002C	5
0030 ... ... 0034	6

9.3.1 Paginazione: metodo di base

- Per poter eseguire un processo che occupa *x* pagine, il SO cerca *x* frame liberi in cui caricare le pagine
- **Gli x frame possono non essere adiacenti**, e le pagine possono essere caricate nei frame in un qualsiasi ordine
- Ad ogni processo è associata una **tabella delle pagine** (o **Page Table** o **PT**): un array le cui entry contengono i numeri dei frame in cui il SO ha caricato in RAM le pagine del processo.
- Il SO deve anche tenere traccia dei frame liberi in memoria primaria, così potrà usarli per scriverci dentro le pagine di un nuovo processo appena nato nel sistema.

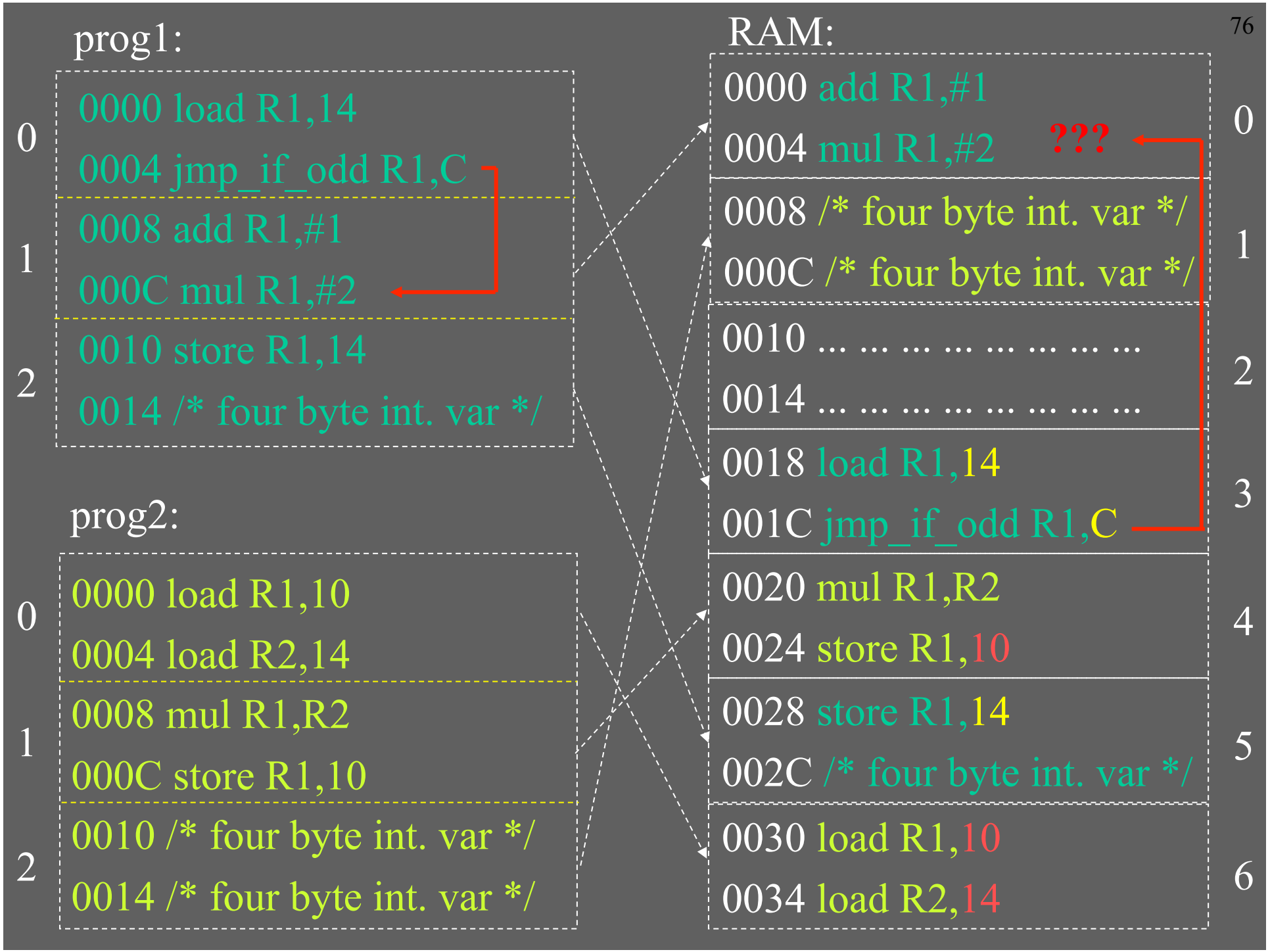
9.3.1 Paginazione: metodo di base

- Per ogni processo, l'indice di ogni entry della sua tabella delle pagine corrisponde al numero di una pagina del processo, e l'entry contiene il numero del frame in cui è stata memorizzata quella pagina (fig. 9.9):



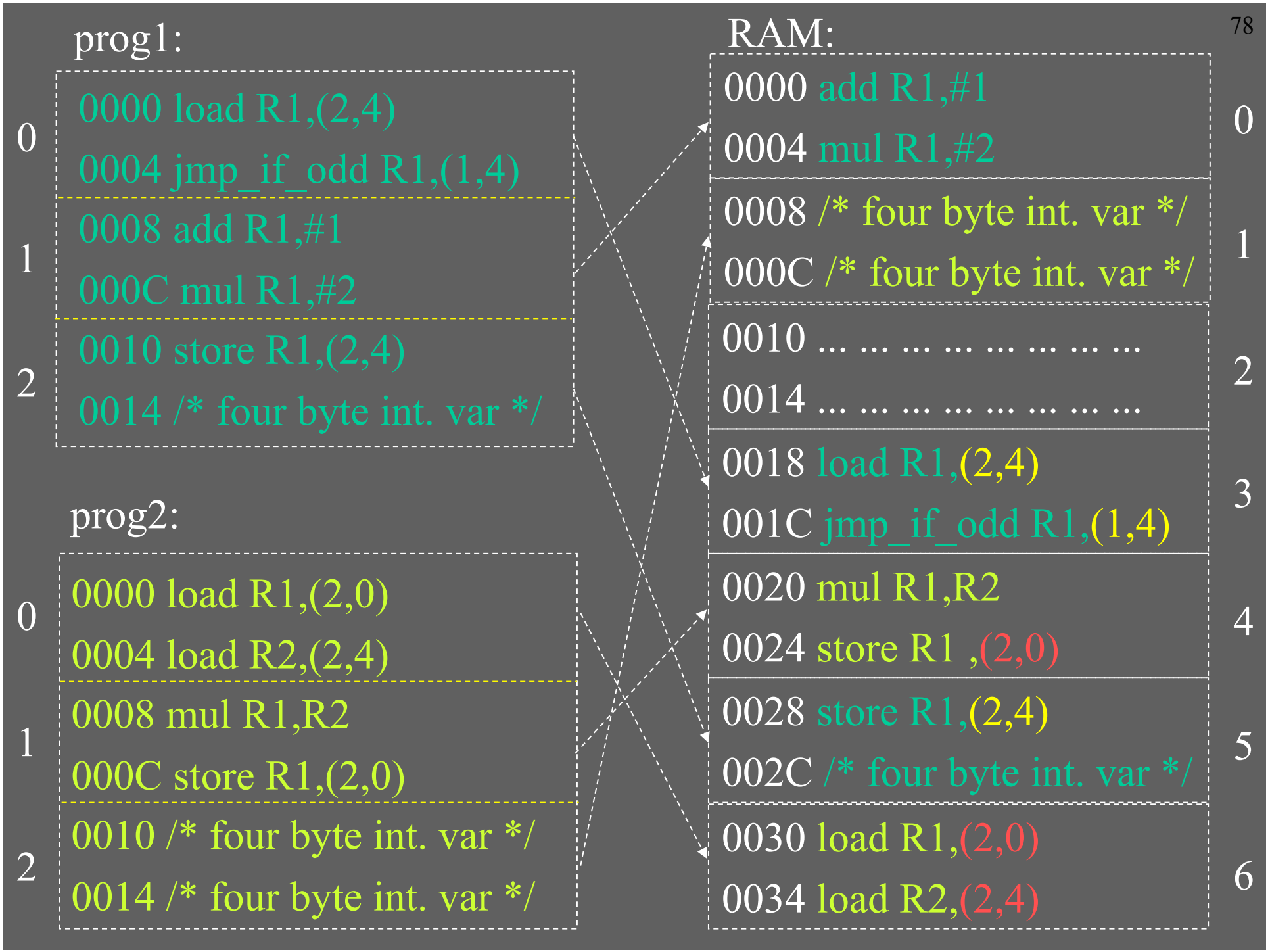
9.3.1 Paginazione: metodo di base

- Il problema è che però adesso gli indirizzi relativi, una volta che il programma viene trasferito in RAM, sembrano non funzionare più! Infatti, non esiste più un indirizzo in RAM a partire dal quale è memorizzato il programma.
- Guardate ad esempio l'istruzione "jmp_if_odd R1,C" di prog1: una volta in RAM, dovrebbe specificare un salto all'indirizzo 0004. Come possiamo calcolare tale indirizzo di destinazione del salto? E analogo problema si presenta per gli indirizzi delle variabili
- Per far funzionare tutto, dobbiamo ripensare gli indirizzi relativi (o logici, o virtuali): non più come indirizzi "lineari" all'interno dello spazio di indirizzamento logico del programma (ossia da 0 all'indirizzo dell'ultimo byte usato dal programma e dai suoi dati).


prog1:
0
0000 load R1,14
0004 jmp_if_odd R1,C
1
0008 add R1,#1
000C mul R1,#2
2
0010 store R1,14
0014 /* four byte int. var */
prog2:
0
0000 load R1,10
0004 load R2,14
1
0008 mul R1,R2
000C store R1,10
2
0010 /* four byte int. var */
0014 /* four byte int. var */
RAM:
0000 add R1,#1
0004 mul R1,#2
???
0
0008 /* four byte int. var */
000C /* four byte int. var */
1
0010 ... ...
0014 ... ...
2
0018 load R1,14
001C jmp_if_odd R1,C
3
0020 mul R1,R2
0024 store R1,10
4
0028 store R1,14
002C /* four byte int. var */
5
0030 load R1,10
0034 load R2,14
6

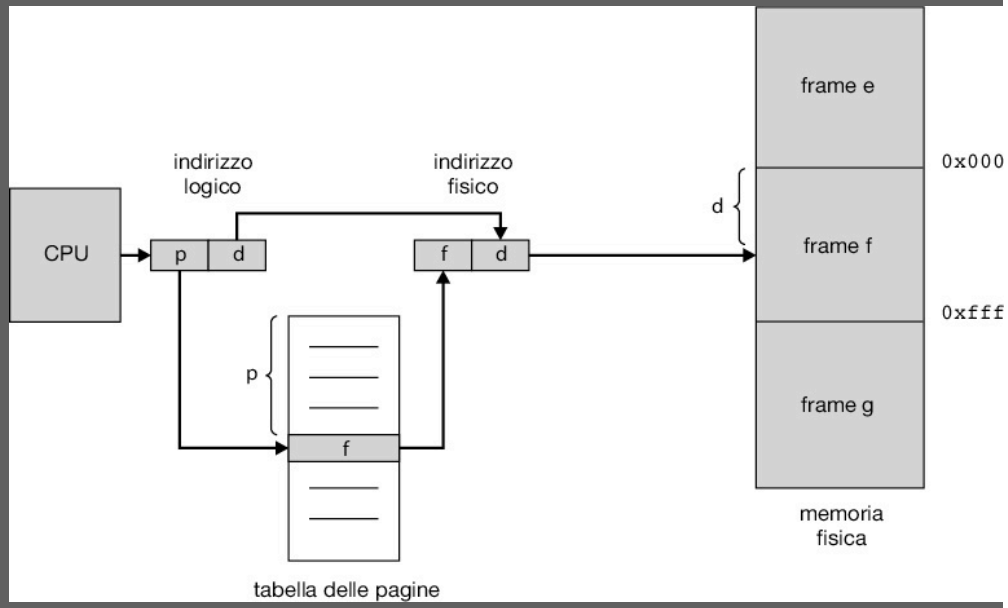
9.3.1 Paginazione: metodo di base

- Gli indirizzi logici diventano delle coppie di valori, in cui:
- il primo elemento della coppia specifica il **numero della pagina** all'interno della quale si trova la cella di memoria che vogliamo indirizzare;
- il secondo elemento specifica la posizione (o **offset**) della cella di memoria che vogliamo indirizzare rispetto ad un ipotetico indirizzo 0 (zero): l'indirizzo del primo byte della pagina specificata dal primo elemento della coppia.
- Quindi un indirizzo logico ha ora la forma: **(pagina, offset)**
- **Ma come vedremo più avanti, sotto opportune condizioni gli indirizzi logici lineari e gli indirizzi logici specificati come coppie di valori COINCIDONO.**


prog1:
0
0000 load R1,(2,4)
0004 jmp_if_odd R1,(1,4)
1
0008 add R1,#1
000C mul R1,#2
2
0010 store R1,(2,4)
0014 /* four byte int. var */
prog2:
0
0000 load R1,(2,0)
0004 load R2,(2,4)
1
0008 mul R1,R2
000C store R1,(2,0)
2
0010 /* four byte int. var */
0014 /* four byte int. var */
RAM:
0
0000 add R1,#1
0004 mul R1,#2
1
0008 /* four byte int. var */
000C /* four byte int. var */
2
0010 ... ...
0014 ... ...
3
0018 load R1,(2,4)
001C jmp_if_odd R1,(1,4)
4
0020 mul R1,R2
0024 store R1 ,(2,0)
5
0028 store R1,(2,4)
002C /* four byte int. var */
6
0030 load R1,(2,0)
0034 load R2,(2,4)

9.3.1 Paginazione: traduzione degli indirizzi

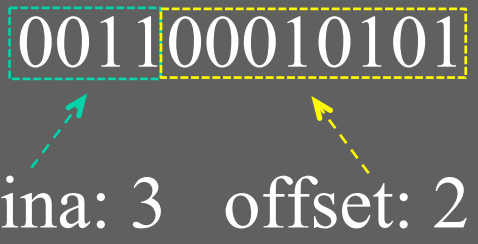
- Un indirizzo logico viene tradotto in uno fisico così:
- Il numero di pagina **p** viene usato come indice nella page table del processo per sapere in quale frame **f** è contenuta quella pagina



- Una volta noto il frame f che contiene la pagina p, l'offset **d** ("d" da displacement) può essere applicato a partire dall'inizio del frame per indirizzare il byte specificato dalla coppia **p,d** (Fig. 9.8: architettura di paginazione)

9.3.1 Paginazione: metodo di base

- Vediamo più in dettaglio: ogni informazione all'interno del computer, e quindi anche un indirizzo logico, in definitiva non è che una sequenza di bit. Ad esempio: 001100010101
- Ma ora abbiamo deciso di dividere un indirizzo logico in due parti, e se abbiamo a disposizione 12 bit in tutto, dobbiamo decidere quanti usarne per specificare il numero di una pagina e quanti per l'offset all'interno della pagina.
- Ad esempio, decidiamo di usare 4 bit per scrivere il numero di pagina e i restanti 8 per l'offset all'interno della pagina.
- Ecco allora che l'indirizzo logico 001100010101 non è altro che la notazione binaria di:

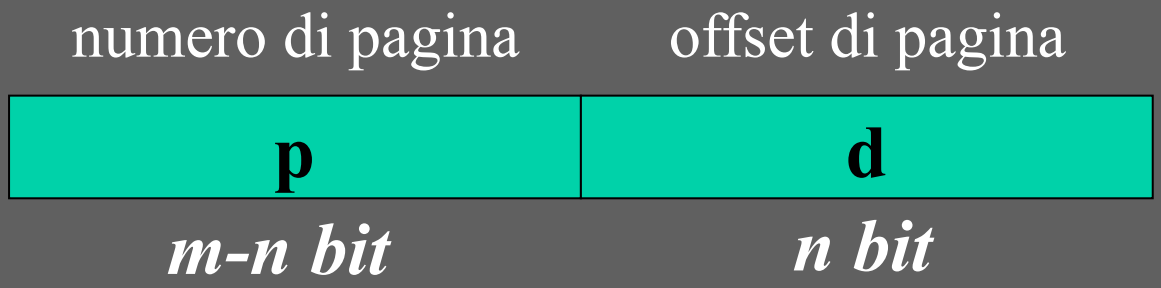


9.3.1 Paginazione: metodo di base

- La scelta del numero di bit da usare per scrivere il numero di pagina e l'offset di un indirizzo logico dipendono dall'hardware su cui dovrà girare il SO, che impone:
- **Il numero di bit su cui va scritto un indirizzo logico** (poniamo ad esempio ***m*** bit)
- **La dimensione di un frame, e quindi di una pagina** (poniamo ad esempio 2^n byte).
- Ma quindi, dovremo usare ***n*** bit per scrivere l'offset all'interno di una pagina/frame, e il numero di bit usati per scrivere il numero di una pagina sarà pari a ***m-n*** bit.
- Notate che a questo punto è stabilita la dimensione dello spazio di indirizzamento logico: $2^{(m-n)} \times 2^n$ byte

9.3.1 Paginazione: metodo di base

- Facciamo un esempio: una macchina permette di usare frame della dimensione di 2^{12} byte = 4096 byte (dunque $n = 12$). Questa sarà anche la dimensione di una pagina nello spazio di indirizzamento logico del sistema
- Se la macchina mette a disposizione $m = 22$ bit per scrivere un indirizzo logico, allora $m-n = 10$.
- Allora, lo spazio di indirizzamento logico della macchina sarà di (2^{10} pagine) × (2^{12} byte) = 4 megabyte



9.3.1 Paginazione: metodo di base

- Qui si può vedere uno dei tanti casi di interazione tra il sistema operativo e l'hardware sottostante. È l'hardware a decidere la dimensione dei frame, e il numero di bit su cui scrivere un indirizzo logico, e il sistema operativo "deve adeguarsi".
- In questo modo (lo capiremo meglio tra poco) la traduzione degli indirizzi da logici a fisici può essere fatta direttamente a livello hardware in maniera molto efficiente.
- In altre parole, alcune porzioni di un sistema operativo vengono sempre scritte tenendo conto dello specifico hardware su cui girerà il sistema, in modo da sfruttarne al meglio le caratteristiche e limitare l'overhead introdotto dal sistema operativo stesso.

9.3.1 Paginazione: metodo di base

- Se non altro, i processori moderni permettono di solito al progettista del sistema operativo di decidere tra diversi valori possibili la dimensione dei frame (e quindi delle pagine) che, una volta scelta, rimane fissa.
- Ad esempio, i processori ARM usati negli iPhone e negli iPad permettono di scegliere tra 4KB, 16KB, 64KB, 1MB e 16MB come dimensione di un frame.
- La famiglia dei processori Intel dal Pentium fino ai core i9 permette di scegliere tra 4KB e 4MB
- Gli Intel Itanium-2 permettono di scegliere tra 4KB, 8KB, 64KB, 256KB, 1MB, 4MB, 16MB o 256MB.

9.3.1 Paginazione: metodo di base

- Ora definiamo più precisamente l'operazione di traduzione degli indirizzi logici in indirizzi fisici.
- Un **indirizzo logico** è formato da due parti (p,d):
- **Numero di Pagina (p)** – viene usato come indice per selezionare la entry della Page Table in cui si trova il *numero del frame* in cui è caricata la pagina.
- **Offset di Pagina (d)** – viene usato all'interno del frame selezionato al passo precedente per *spostarsi* nel punto esatto del frame specificato dall'indirizzo logico.

8.4.1 Paginazione: metodo di base

- Ma possiamo riformulare il lucido precedente in modo leggermente diverso ma equivalente:
- Un **indirizzo logico** è formato da due parti (p,d):
- **Numero di Pagina (p)** – viene usato come indice per selezionare la entry della Page Table in cui si trova *l'indirizzo base* del frame in cui è caricata la pagina.
- **Offset di Pagina (d)** – viene *sommato* all'indirizzo base del frame per generare l'indirizzo fisico.
- Ma che differenza c'è tra *il numero del frame*, menzionato nel lucido precedente, e *l'indirizzo base* del frame? E quindi qual è il modo giusto di usare l'offset: *spostarsi* o *sommare*?

page table di prog1: contiene
i *numeri dei frame*

0	3
1	0
2	5

page table di prog2: contiene
gli *indirizzi base dei frame*

0	0030
1	0020
2	0008

RAM:

0000 add R1,#1	0
0004 mul R1,#2	
0008 /* four byte int. var */	1
000C /* four byte int. var */	
0010	2
0014	
0018 load R1,(2,4)	3
001C jmp_if_odd R1,(1,4)	
0020 mul R1,R2	4
0024 store R1,(2,0)	
0028 store R1,(2,4)	5
002C /* four byte int. var */	
0030 load R1,(2,0)	6
0034 load R2,(2,4)	

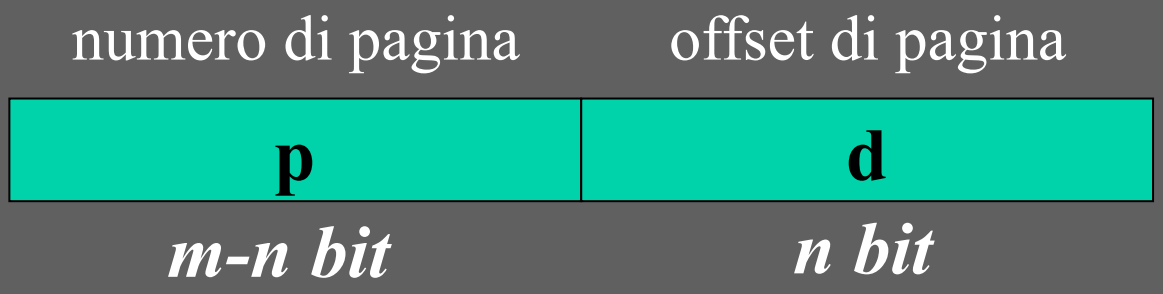
9.3.1 Paginazione: traduzione degli indirizzi

- Ora possiamo mettere assieme tutto quello detto fin'ora e vedere come:

1. Gli indirizzi logici all'interno dello spazio di indirizzamento logico possono essere interpretati indifferentemente come **valori lineari** oppure come **coppie (pagina, offset)**
2. Ciascuna entry di una tabella delle pagine può contenere equivalentemente *un numero di frame* oppure *l'indirizzo di partenza di un frame* (come vedremo fra poco, per ragioni puramente pratiche si sceglie di memorizzare *il numero del frame*)

9.3.1 Paginazione: traduzione degli indirizzi

- Ecco qui la doppia natura degli indirizzi logici, che sono allo stesso tempo **valori lineari** e **coppie (pagina, offset)**
- Consideriamo infatti uno spazio di indirizzamento logico di dimensione 2^m byte
- Assumiamo che la dimensione di un pagina sia 2^n byte
- Allora dato un indirizzo logico lineare scritto su *m* bit:
  - gli *m-n* bit più significativi indicano il numero di pagina
  - gli *n* bit meno significativi indicano l'offset



9.3.1 Esempio

- Supponiamo di avere uno spazio di indirizzamento logico di 16 byte: gli indirizzi logici sono quindi scritti usando $m = 4$ bit ($2^4 = 16$).
- Ogni pagina ha dimensione 2^2=4 byte, con n=2.
- Gli indirizzi logici vanno da 0000 a 1111, ma il programma (istruzioni + dati) **abcdefghilmn** occupa solo tre pagine.
- Ogni istruzione/dato di questo esempio occupa esattamente un byte

pagina 0	00 00	a
	00 01	b
	00 10	c
	00 11	d
pagina 1	01 00	e
	01 01	f
	01 10	g
	01 11	h
pagina 2	10 00	i
	10 01	l
	10 10	m
	10 11	n

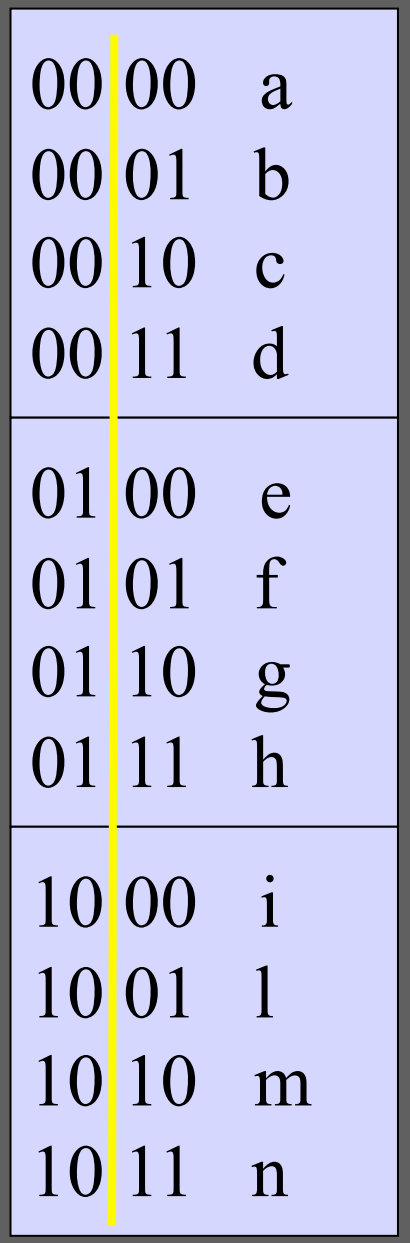
9.3.1 Esempio

- Qui lo spazio di indirizzamento logico del programma è diviso in pagine, ma gli indirizzi di istruzioni e dati (appunto, lo spazio di indirizzamento logico del programma) sono valori consecutivi che vanno da 0000, dove si trova la prima istruzione del programma, a 1011, dove si trova l'ultima istruzione/dato del programma.

pagina 0	00 00	a
	00 01	b
	00 10	c
	00 11	d
pagina 1	01 00	e
	01 01	f
	01 10	g
	01 11	h
pagina 2	10 00	i
	10 01	l
	10 10	m
	10 11	n

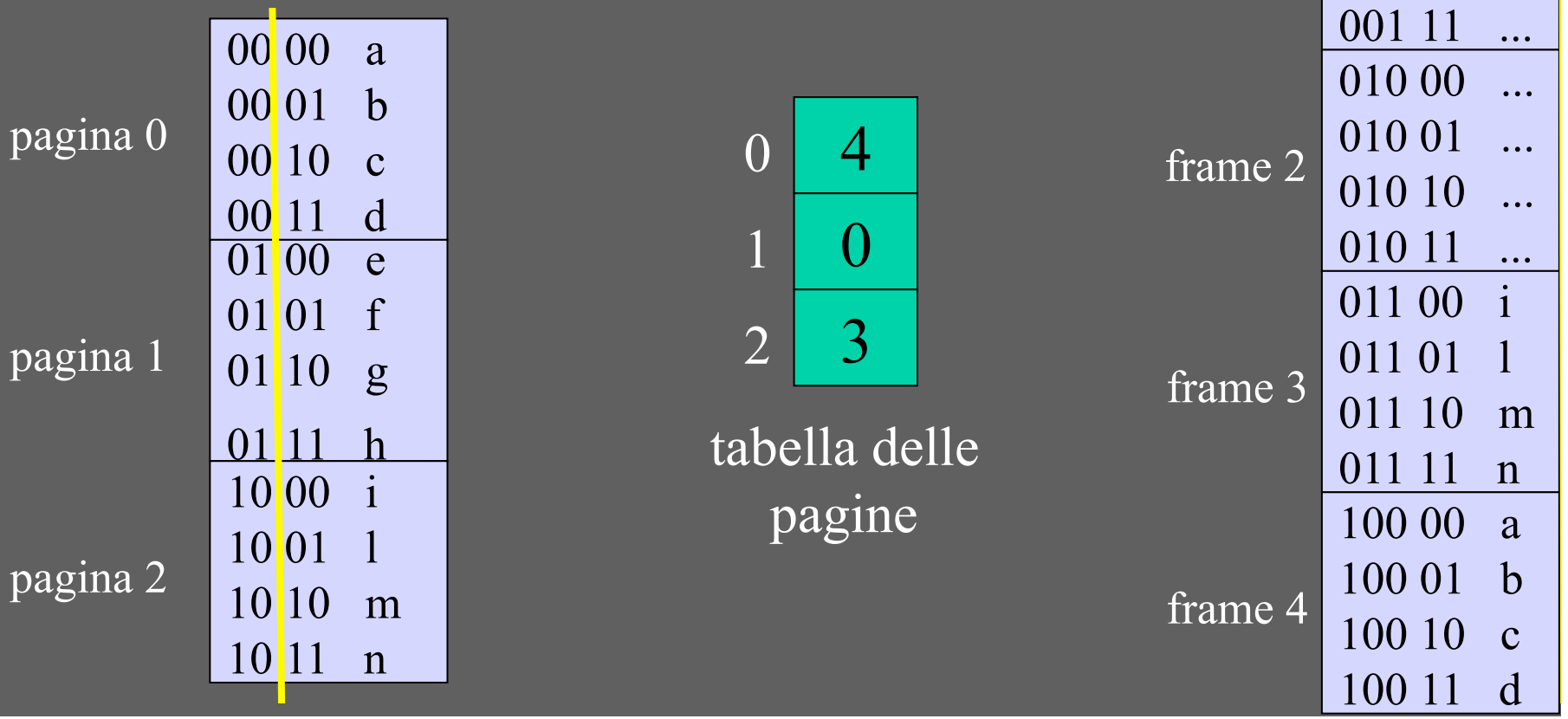
9.3.1 esempio

- Ma, notate: gli $m - n$ ($4 - 2 = 2$) bit più significativi di ogni indirizzo, non sono altro che il numero della pagina corrispondente
- i restanti n (cioè 2) bit meno significativi, non sono altro che l'offset all'interno di ogni pagina



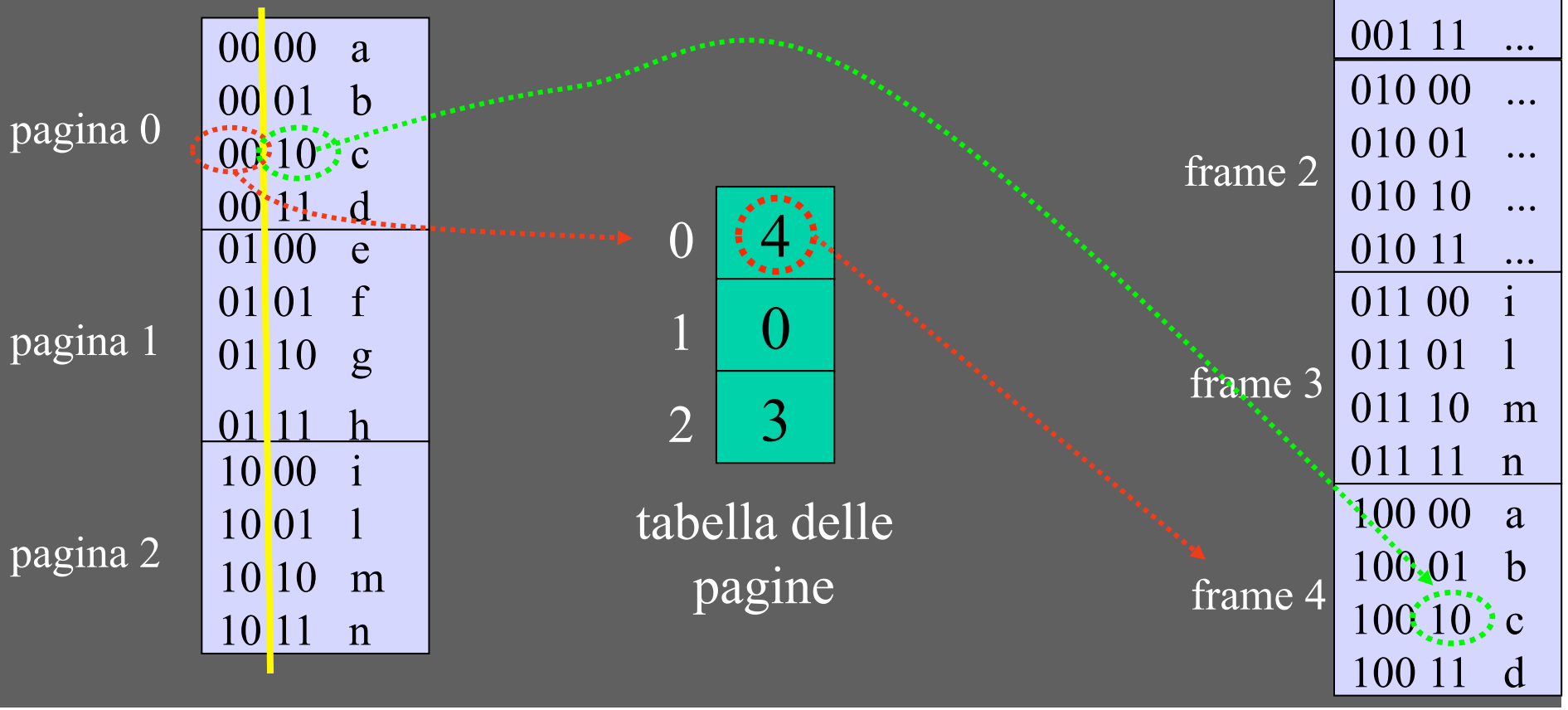
9.3.1 esempio

- Il programma viene caricato in RAM. Usiamo 5 bit per un indirizzo fisico, per cui lo spazio di indirizzamento fisico sarebbe di 2^5=32 byte, ma la nostra macchina ha solo 20 byte effettivi di RAM, 5 frame con indirizzo da 00000 a 10011 (fig. 9.10 modificata)



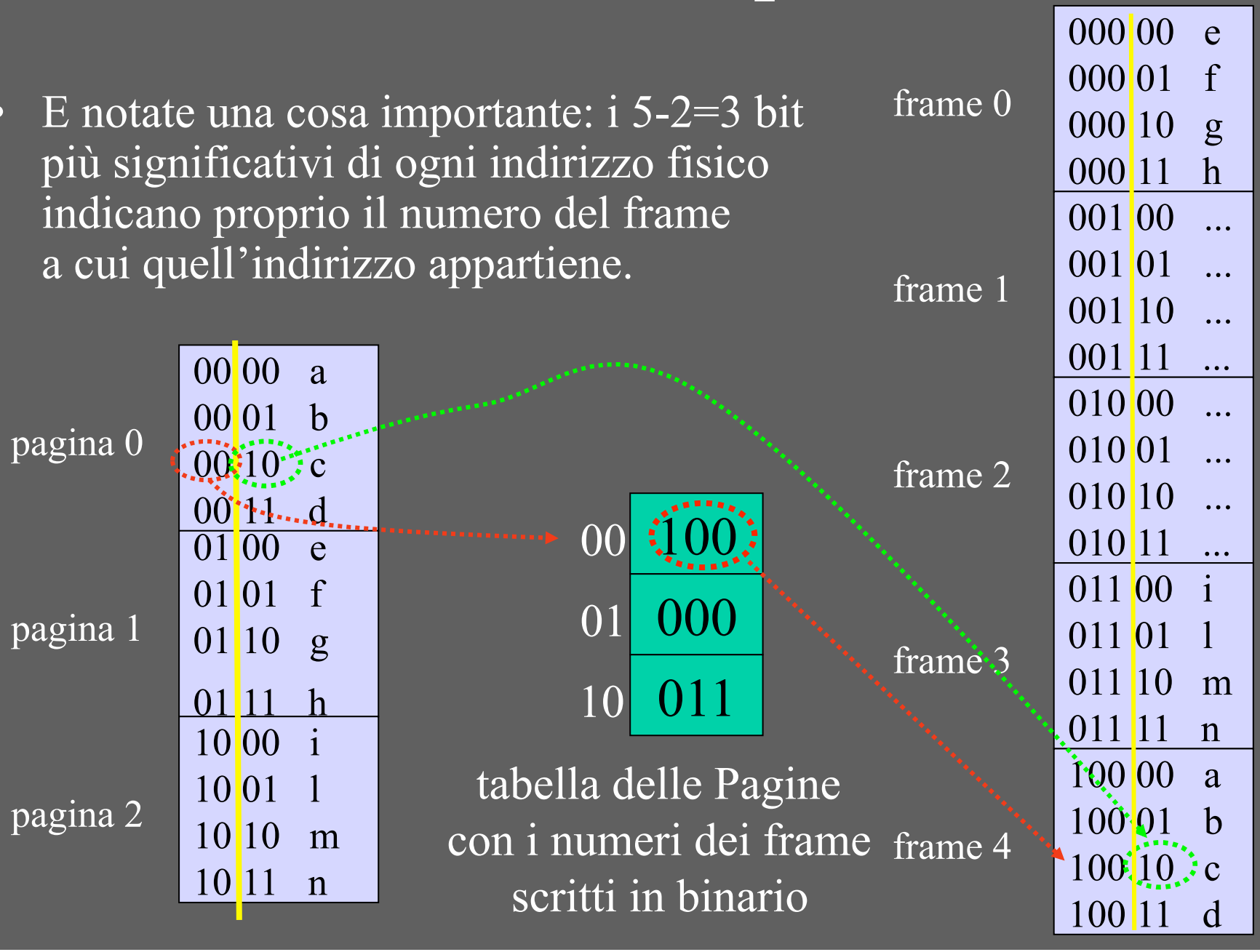
9.3.1 esempio

- L'indirizzo logico 0010 viene convertito nell'indirizzo fisico 10010: il numero di pagina 0 viene usato per indirizzare la tabella delle pagine e sapere che la pagina 0 si trova nel frame 4. L'offset 10 viene applicato a partire dall'inizio del frame 4 (riguardate la figura 9.8)



9.3.1 esempio

- E notate una cosa importante: i 5-2=3 bit più significativi di ogni indirizzo fisico indicano proprio il numero del frame a cui quell'indirizzo appartiene.



9.3.1 Paginazione: traduzione degli indirizzi

- Tutto funziona correttamente perchè le pagine/frame hanno una dimensione |P| che è una potenza di 2. ($|P|=2^n$)
- Allora, in ogni pagina/frame l'offset va da una configurazione di tutti *zeri* a una di tutti *uno*:
  - offset valido per una pagina: $\underbrace{00...00}_{n \text{ bit}}$ → 11...11
- Inoltre, l'indirizzo di partenza di ogni pagina/frame deve avere per forza gli *n* bit meno significativi tutti a 0.

9.3.1 Paginazione: traduzione degli indirizzi

- Quindi, gli indirizzi logici di partenza di un insieme di pagine in cui è suddiviso lo spazio di indirizzamento logico di un programma avranno sempre una forma in cui gli *n* bit meno significativi sono tutti uguali a 0:

  00..00 00..00 inizio della pagina 0

  00..01 00..00 inizio della pagina 1

  00..10 00..00 inizio della pagina 2

  00..11 00..00 inizio della pagina 3

  n bit meno significativi

- E lo stesso vale per i frame nello spazio di indirizzamento fisico

9.3.1 Paginazione: traduzione degli indirizzi

- Allora, se rimuoviamo gli *n* bit meno significativi di un indirizzo logico fatto di *m* bit, gli *m* – *n* bit che rimangono non fanno altro che contare (in binario) le pagine in cui è suddiviso lo spazio di indirizzamento logico:

  00..00 = pagina 0

  00..01 = pagina 1

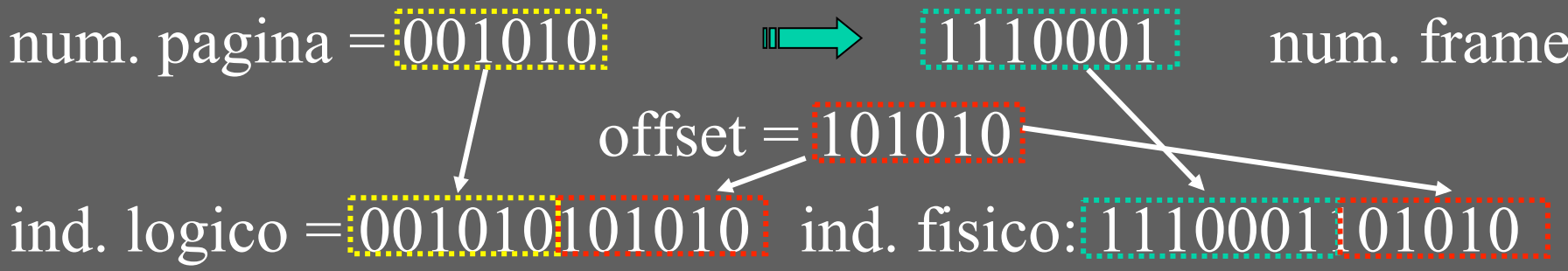
  00..10 = pagina 2

  00..11 = pagina 3

- E lo stesso vale per i frame nello spazio di indirizzamento fisico, eventualmente usando un numero diverso di bit per contare i frame

9.3.1 Paginazione: traduzione degli indirizzi

- Una volta recuperato il numero del frame che contiene una certa pagina, l'indirizzo fisico (ad esempio scritto su k bit), può essere ricostruito usando il numero del frame per i $k - n$ bit più significativi dell'indirizzo, a cui attacchiamo gli n bit meno significativi dell'offset rispetto a quel frame (o alla pagina corrispondente, dato che hanno la stessa dimensione)



9.3.1 Paginazione: traduzione degli indirizzi

- Notate che, nel calcolare l'indirizzo fisico a partire da quello logico, la vera operazione che stiamo facendo è di sommare all'indirizzo di partenza del frame:

$$111000\underbrace{1000000}_{\text{bit meno significativi}}$$

l'offset 101010:

$$\begin{array}{r} 1110001000000 + \\ \underline{\qquad\qquad 101010} = \\ 1110001101010 \end{array}$$

9.3.1 Paginazione: traduzione degli indirizzi

- Ma se pagine e frame hanno una dimensione pari a una potenza di due, allora:

1. non c'è bisogno di fare effettivamente la somma tra l'indirizzo di base del frame e l'offset (il che richiederebbe tempo), e:
2. Non c'è bisogno di memorizzare in ogni entry di una page table l'indirizzo di base del frame, ma solo il numero del frame, dato che gli n bit meno significativi dell'indirizzo sono sempre tutti a 0 (il che fa risparmiare spazio)

- Per generare un indirizzo fisico basta attaccare insieme il numero del frame e l'offset, un'operazione che si può fare molto velocemente a livello hardware.

9.3.1 Paginazione: traduzione degli indirizzi

- In altre parole una operazione di somma come:

$$\begin{array}{r} 1110001000000 + \\ 101010 = \\ \hline 1110001101010 \equiv \end{array}$$

- Viene eseguita più semplicemente, e direttamente dall'hardware su cui gira il sistema operativo come:

1110001 "attaccato a" 101010

- Se non usiamo potenze di due per le pagine, possiamo far funzionare tutto lo stesso, ma non in maniera così semplice ed efficiente.

9.3.1 Paginazione: alcune osservazioni

- La paginazione separa nettamente lo spazio di indirizzamento logico da quello fisico.
- Il programma “vede” la memoria come uno spazio di indirizzamento contiguo che incomincia sempre all’indirizzo logico 0.
- In realtà, il programma è “sparpagliato” in tanti pezzettini (i frame) nella memoria fisica, insieme ad altri programmi

9.3.1 Paginazione: vantaggi

- **La paginazione implementa automaticamente una forma di protezione dello spazio di indirizzamento.**

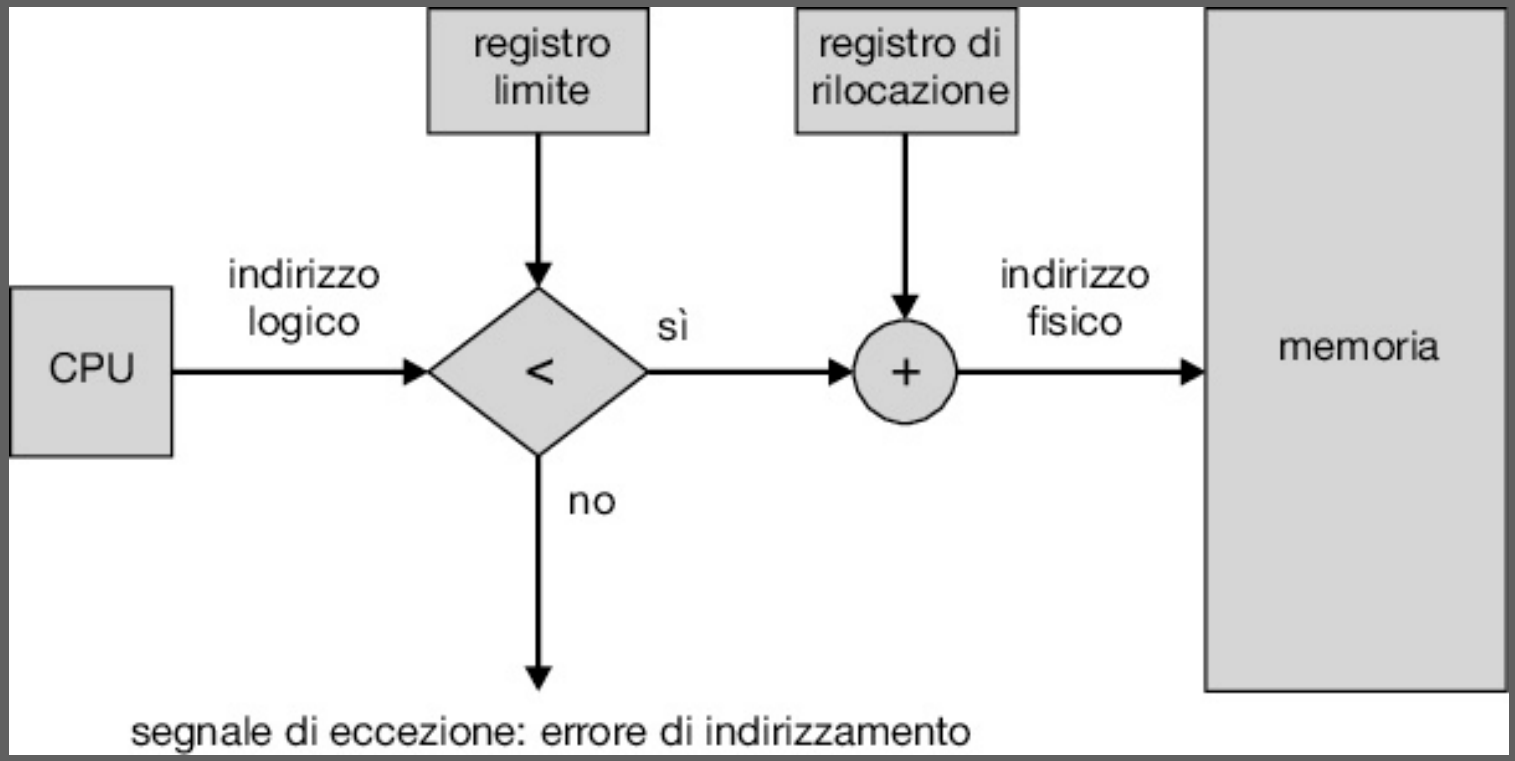
- Un processo può indirizzare solo i frame contenuti nella sua tabella delle pagine.
-
- Quei frame contengono le pagine che appartengono al processo stesso, perché è il sistema operativo a costruire la tabella delle pagine di ogni processo attivo.

9.3.1 Paginazione: vantaggi

- **la paginazione evita la frammentazione esterna**: qualsiasi frame libero può essere usato per contenere una pagina
- La frammentazione interna rimane, ed è in media di mezza pagina per processo (l'ultima pagina del processo può non occupare completamente il frame che la contiene)
- **La paginazione è una forma di rilocazione dinamica**, in cui ad ogni pagina corrisponde un diverso valore del registro di rilocazione: l'indirizzo di partenza del frame che contiene quella pagina

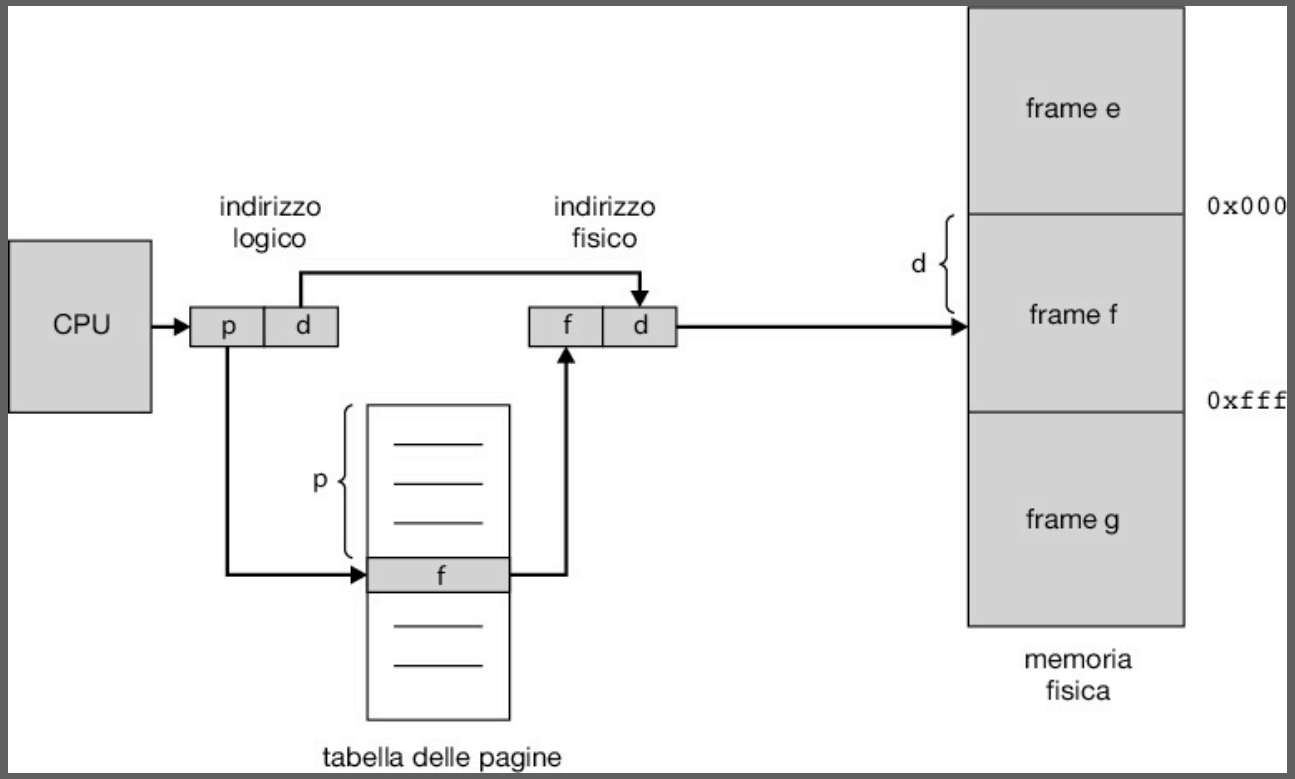
Rilocazione dinamica attraverso i registri limite e di rilocazione

- Ricordate: prima si controlla che l'indirizzo logico usato non ecceda lo spazio di indirizzamento del processo, e poi si calcola l'indirizzo fisico (fig. 9.6)



Rilocazione dinamica di ogni singola pagina attraverso la page table

- Qui il controllo attraverso il registro limite non serve più: in ogni indirizzo logico, l'offset non può che specificare una posizione all'interno del frame corrispondente (fig. 9.8)



9.3.1 Paginazione: dimensione delle pagine

- Storicamente, la dimensione delle pagine è aumentata col tempo. Attualmente si usano pagine di 4096, 8192, 16384 byte, e si può arrivare anche a 256 Mega byte.

- pagine più grandi producono maggiore frammentazione interna, ma permettono di avere delle tabelle delle pagine più corte (come vedremo più avanti questo è preferibile).

9.3.1 Paginazione: svantaggi

- Ad ogni processo è associata una tabella delle pagine, che occupa spazio in RAM, e Il SO deve mantenere anche una **frame table** che descrive lo stato di allocazione della memoria fisica:
  - quali frame sono liberi
  - quali frame sono occupati e da quale pagina di quale processo
- Ad ogni context switch, il SO deve "attivare" la tabella delle pagine del processo a cui viene assegnata la CPU, questo richiede un certo lavoro e aumenta il tempo necessario al cambio di contesto (e non è il problema più grave…).

9.3.2 Supporto Hardware alla paginazione

- Dato che ogni accesso alla MP passa attraverso il sistema di paginazione, il meccanismo di traduzione degli indirizzi da logici a fisici deve essere efficiente

- Il problema fondamentale riguarda la gestione della tabella delle pagine:

- Ogni indirizzo logico che esce dalla CPU deve passare attraverso una entry della PT. L'accesso alla PT deve quindi essere veloce.

9.3.2 Supporto Hardware

- Se il numero di pagine per processo è piccolo, la PT può essere memorizzata all'interno di appositi registri della CPU
- Ad esempio, il PDP-11 usava 8 registri per la PT. Gli indirizzi fisici erano a 16 bit, per cui la MP aveva una dimensione di 64 Kbyte! (divisi in 8 frame da 8 Kbyte)
- ma nei computer moderni è normale usare PT da migliaia di elementi. Non possiamo metterli tutti nei registri della CPU!

9.3.2 Supporto Hardware

- Siamo costretti a tenere la PT di ogni processo in MP: consumando spazio (ecco perché è meglio avere pagine grosse...)

- Ma c'è un problema più grave: Per leggere un dato all'indirizzo logico I, occorre prima accedere alla RAM per recuperare il numero del frame e calcolare l'indirizzo fisico, e poi usarlo per leggere il dato vero e proprio.

- **Il numero di accessi alla MP RADDOPPIA!** Questo è inaccettabile, perché leggere un dato/istruzione in RAM costa più di 100 cicli di clock, mentre costa meno di 5 cicli di clock se si trova in un registro o nella cache L1 della CPU

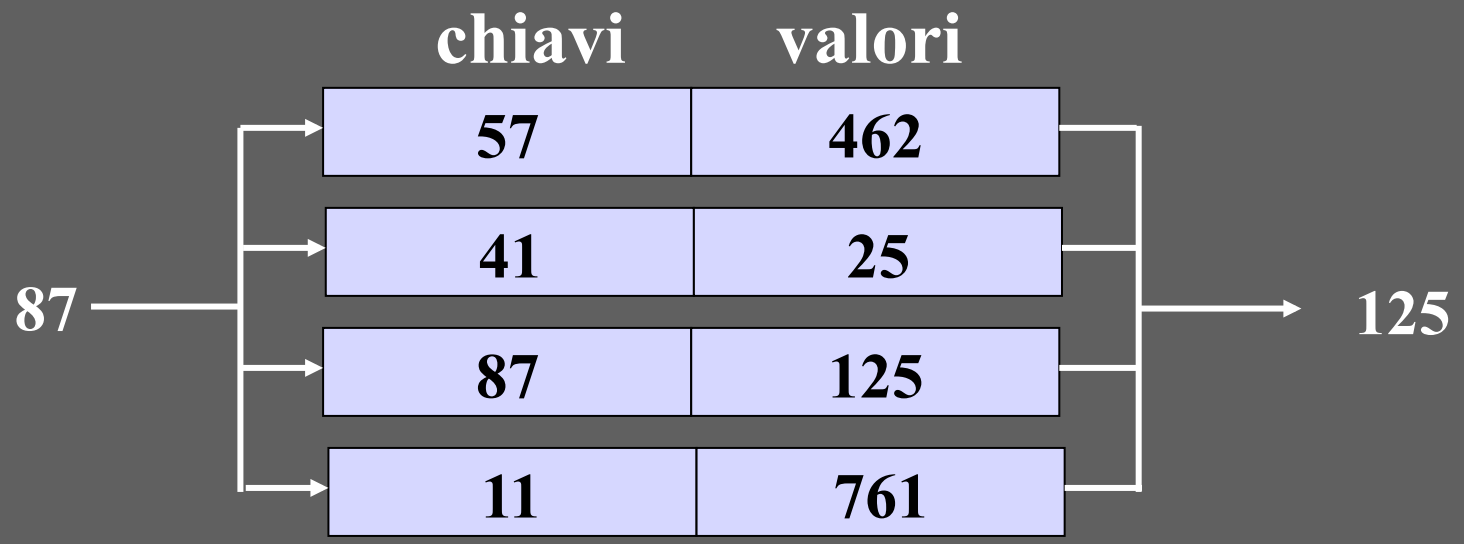
9.3.2 Supporto Hardware

- La soluzione consiste nell'usare una tecnica di caching della PT mediante una **memoria associativa** messa a disposizione dalla CPU detta **translation look-aside buffer (TLB)**: ecco un altro esempio di come i progettisti hardware possono dotare i processori di dispositivi che facilitano e rendono più efficiente il lavoro del SO

- Inoltre, il SO ha a disposizione anche un registro che contiene l'indirizzo di partenza della PT del processo attivo: il **page-table base register (PTBR).** Al context switch basta modificare il PTBR per "attivare" la PT del processo che viene mandato in CPU.

9.3.2 Supporto Hardware: TLB

- Funzionamento di una memoria associativa: più coppie di celle **chiave** – **valore**. Specificando in input una chiave, questa viene confrontata **contemporaneamente** con tutte le chiavi presenti, e viene dato in output il valore associato alla chiave specificata. Il dispositivo è molto veloce e costoso, e le dimensioni sono quindi ridotte (64 -- 1024 elementi)

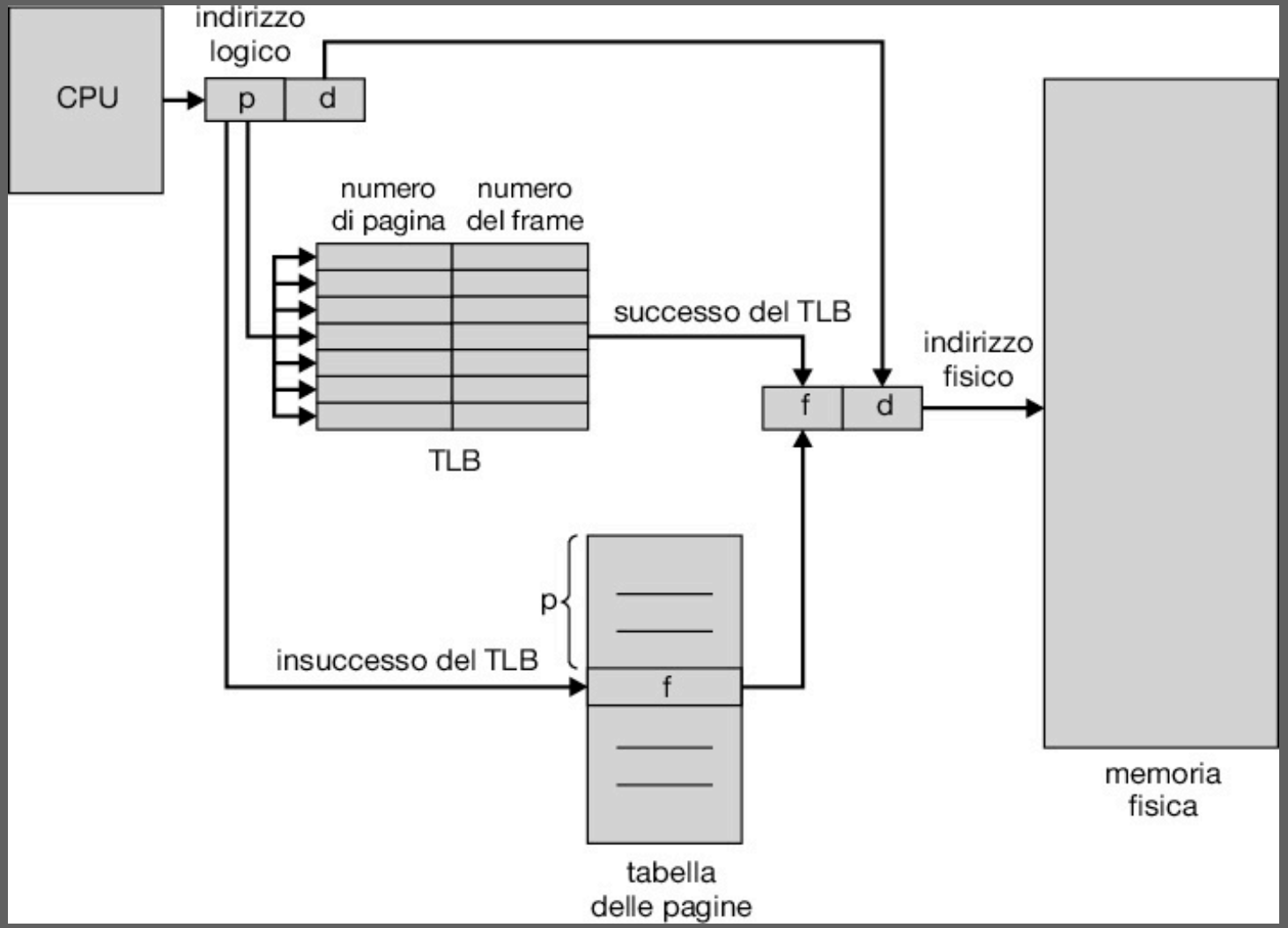


9.3.2 Supporto Hardware: uso del TLB

- Nella TLB viene caricata una porzione della PT (ovviamente, se ci sta può essere caricata tutta): **numero pagina** (chiave) – **frame relativo** (valore).
- quando viene generato un indirizzo logico, il numero di pagina è presentato al TLB per ricavare il frame relativo
- Se la ricerca del numero di pagina del TLB ha successo (**hit**) si ha una penalizzazione circa del 10% del tempo di accesso alla memoria senza paginazione
- Se la ricerca fallisce (**miss**) allora occorre usare la PT in memoria primaria

9.3.2 Supporto Hardware: uso del TLB

- Fig. 9.12: architettura di paginazione con TLB



9.3.2 TLB: hit ratio e prestazioni

- Come cambiano le prestazioni con e senza TLB? Chiamiamo **Hit Ratio:** la % di successi (hit) nell'uso del TLB. Ovviamente, un maggiore hit ratio implica una degradazione minore delle prestazioni.
- Facciamo un esempio supponendo: 1) un TLB perfetto, ossia accedere al TLB non costa tempo; 2) 10 nanosec per accedere alla RAM una volta tradotto l'indirizzo logico in fisico, 3) hit ratio = 80%. Abbiamo allora che il tempo medio di accesso in RAM diventa:

  10 nsec × 0,80 + (10 + 10) nsec × 0,20 = 12 nsec

- Con una degradazione delle prestazioni del 20%

9.3.2 TLB: hit ratio e prestazioni

- Se il TLB avesse invece un hit ratio del 99% avremmo:

  10 nsec × 0,99 + (10 + 10) nsec × 0,01 = 10,1 nsec

- Con una degradazione delle prestazioni del 1%
- Se invece assumiamo che il TLB abbia un tempo di accesso > 0, e sia all'incirca 1 nsec (un decimo del tempo di accesso in RAM), sempre con un hit ratio del 99% abbiamo:

  (10+1) nsec × 0,99 + (10 + 10) nsec × 0,01 = 11,09 nsec

- Con una degradazione delle prestazioni di quasi 11%

9.3.2 Supporto Hardware: uso del TLB

- Quando si verifica un “miss”, la coppia **pagina-frame** mancante viene recuperata attraverso la PT in RAM, ma viene anche copiata nel TLB: i riferimenti successivi alla stessa pagina useranno ma copia nel TLB
- Se il TLB è pieno, una entry va sovrascritta, scegliendone una a caso o la **Least Recently Used**
- Al context switch il TLB va svuotato, e verrà gradualmente popolato con coppie pagina-frame del processo che è entrato in esecuzione

9.3.2 Un caso reale: i TLB del core i7

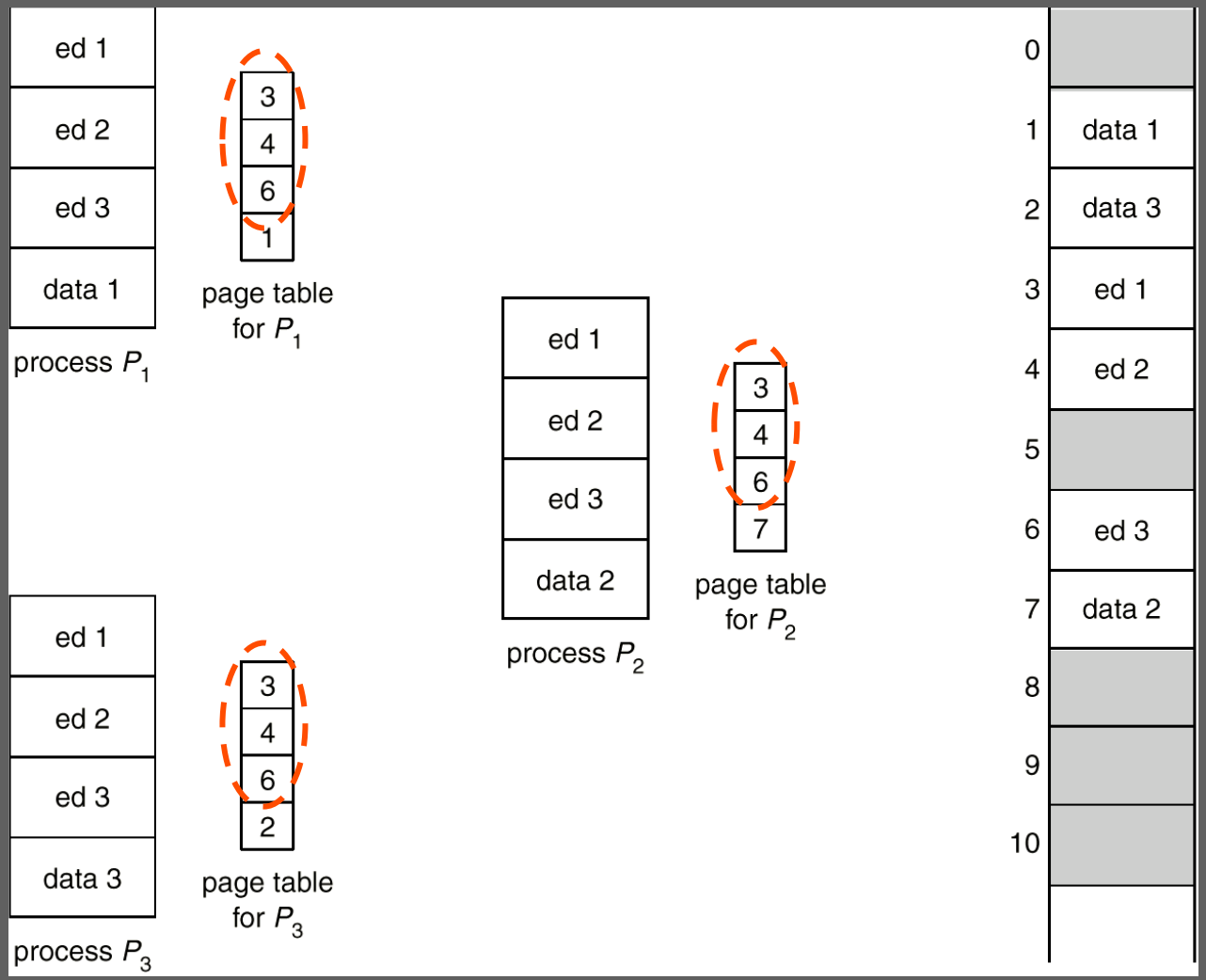
- Come sappiamo, una CPU moderna è dotata di almeno due, ma spesso 3 livelli di cache, che riducono il costo di andare a prelevare dati e istruzioni dalla RAM
- Il TLB è sostanzialmente una cache di indirizzi, e oltre alle normali cache, i processori della famiglia core i7 hanno addirittura due livelli di cache TLB:
  - $L1_{instruction\text{-}address}$(128 entry) + $L1_{data\text{-}address}$(64 entry)
  - L2 512 entry
- In caso di miss su L1, un accesso a L2 costa circa 6 cicli di clock, e in caso di ulteriore miss occorre usare la copia della PT in RAM, sprecando più di cento cicli di clock

9.3.4 Pagine Condivise

- Se due processi eseguono lo stesso codice, è inutile – e soprattutto spreca spazio in RAM – tenerne in MP due copie del codice
- La paginazione facilita la condivisione di codice, perchè questo non cambia durante l'esecuzione (si chiama codice **puro** o **rientrante**)
- Tra le altre cose, una pagina condivisa può essere usata per contenere il codice di una libreria dinamica usata contemporaneamente da più processi

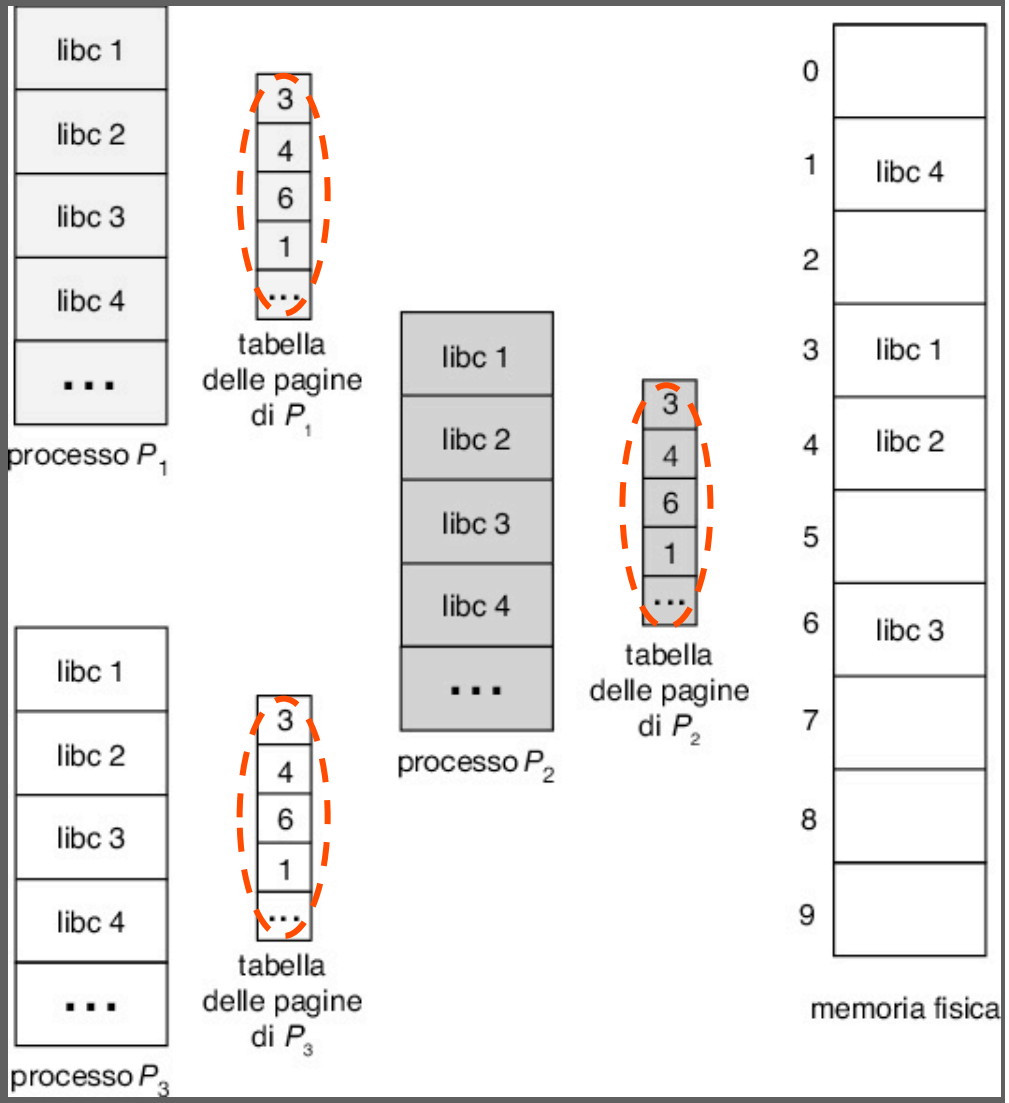
9.3.4 Pagine Condivise

- Condivisione di codice in ambiente paginato:



9.3.4 Pagine Condivise

- Condivisione della libreria standard del C in ambiente paginato (fig. 9.14):



9.4.1 Paginazione a più livelli (paginazione gerarchica)

- Nei moderni calcolatori lo spazio di indirizzi logici può ormai arrivare anche a 2^{64} byte.
- E anche con solo 2^{32} byte di spazio logico, e pagine da 4 Kbyte (2^{12}), la PT può avere anche un milione (2^{20}) di entry.
- Se ogni entry occupa 4 byte, la PT corrispondente occuperà 4 Mbyte e ci vorranno 1024 frame per contenerla!
- Come facciamo ad allocare un'area dati così grande in MP?

(dai numeri qui sopra, possiamo ipotizzare la dimensione massima dello spazio di indirizzamento fisico del sistema?)

9.4.1 Paginazione a più livelli

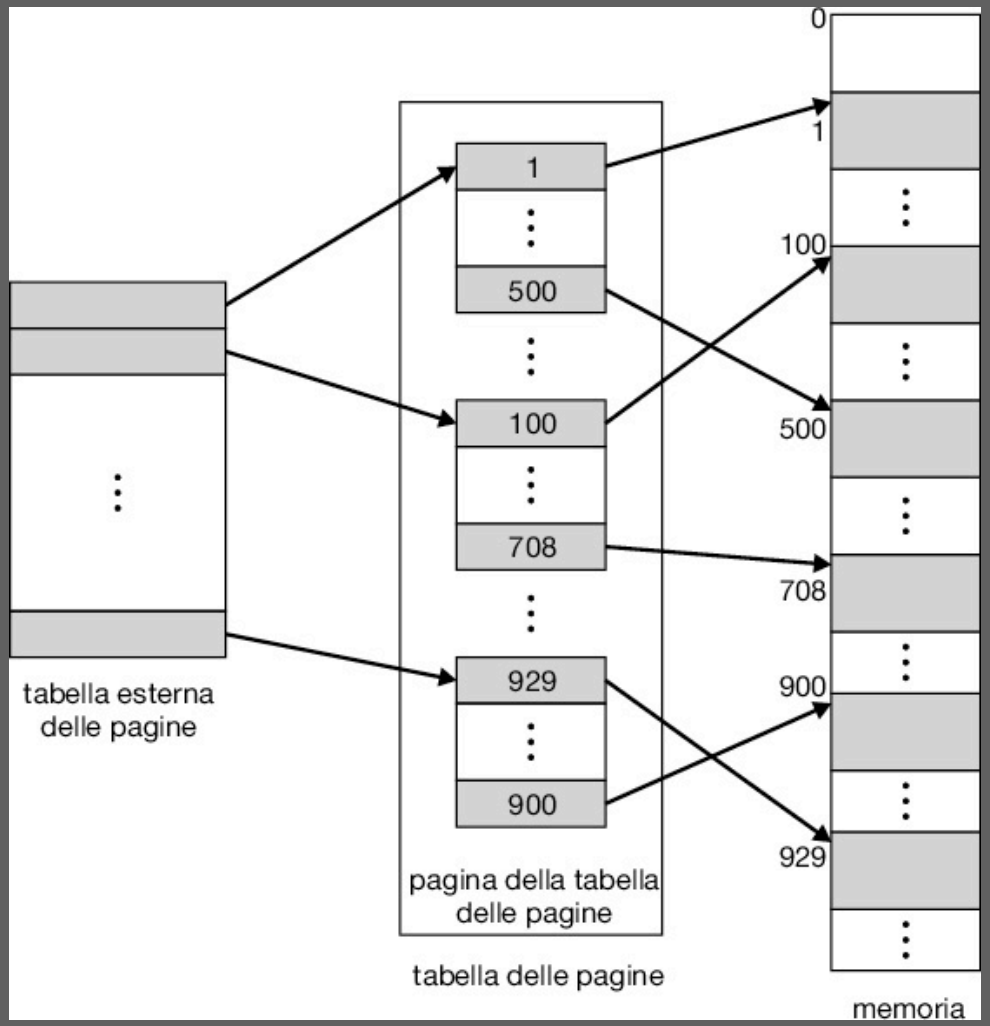
- Usiamo la paginazione per evitare di aver bisogno di aree di MP contigue molto grandi, ma può accadere che sia la PT del processo attivo ad essere molto grande…

- Una possibile soluzione al problema dell'allocazione di PT che possono occupare molto spazio consiste nel paginare le PT, cioè suddividerle in pagine memorizzate in frame non necessariamente adiacenti della memoria principale

- Si parla allora di **paginazione a due livelli:** la PT vera e propria (che ora chiameremo **PT interna**) richiede una **PT esterna**, che ci dice in quali frame sono state memorizzate le pagine della PT interna

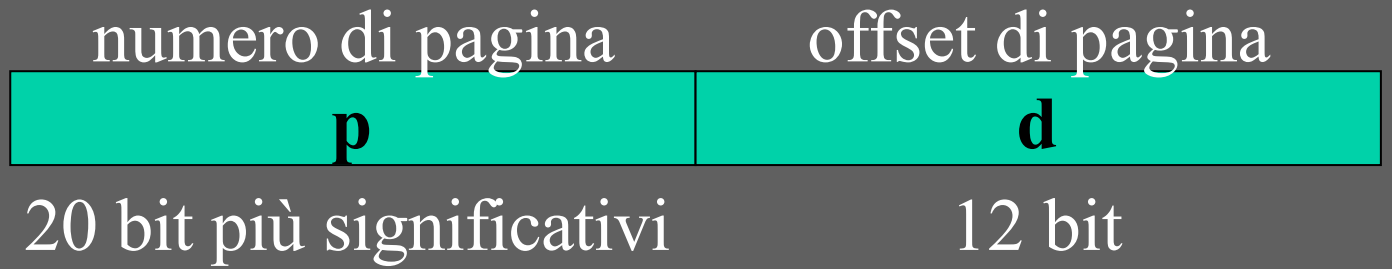
9.4.1 Paginazione a più livelli

- Schema di una tabella delle pagine a due livelli (fig. 9.15):



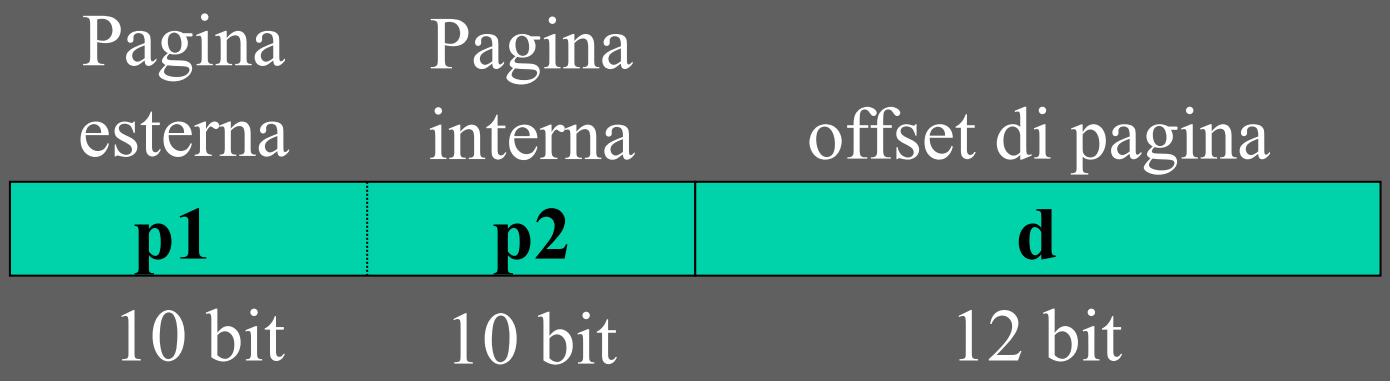
9.4.1 Esempio

- Consideriamo una macchina con 32 bit di spazio di indirizzamento logico e fisico, e con pagine/frame da 4 Kbyte.
- Come sappiamo, un indirizzo logico sarà composto da:
  - 20 bit per il numero della pagina +
  - 12 bit per l'offset all'interno della pagina



9.4.1 Esempio

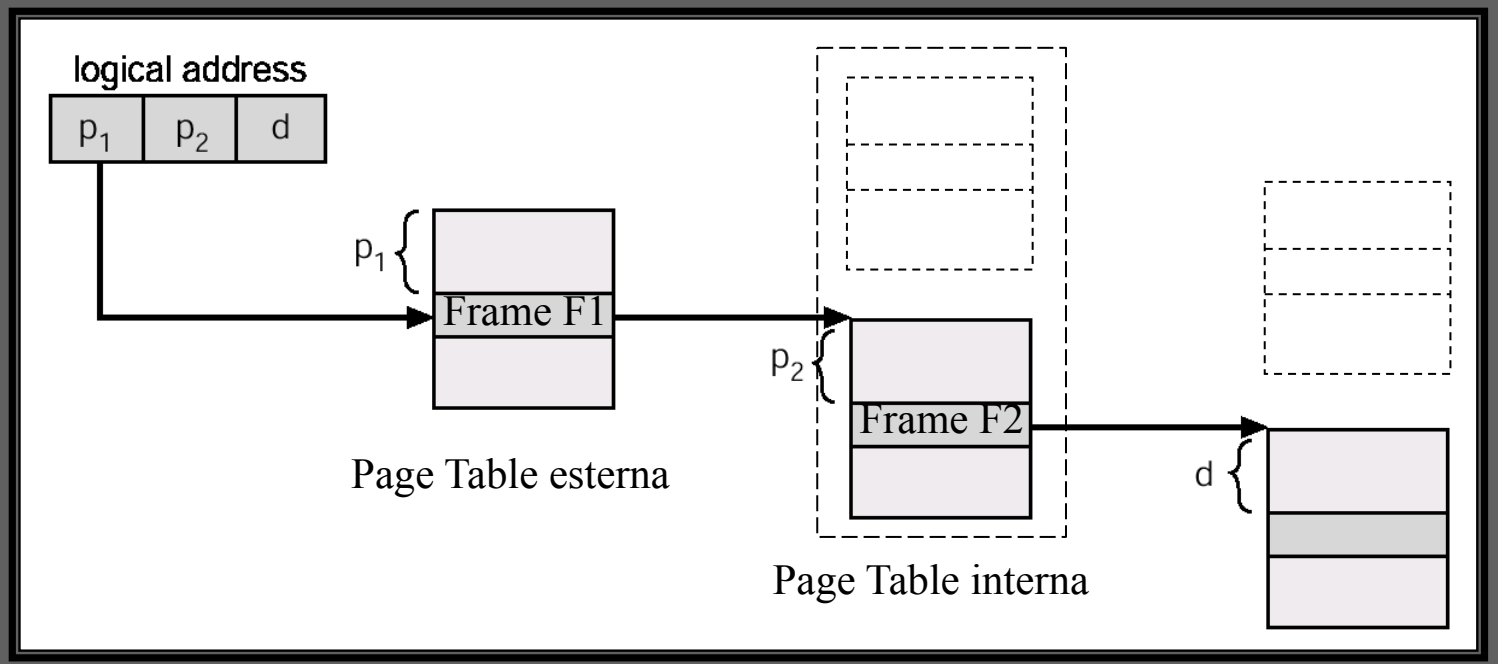
- Per paginare la tabelle delle pagine, il numero di pagina p, scritto su 20 bit sarà a sua volta scomposto come segue:
  - 10 bit più significativi (p1) = entry nella PT esterna che punta al frame F1 che contiene un pezzo della PT interna
  - 10 bit intermedi (p2) = offset nel frame F1



9.4.1 Esempio

- Traduzione degli indirizzi con paginazione a due livelli (fig. 9.16 modificata):

Pagina esterna	Pagina interna	offset di pagina
p1	**p2**	**d**
10 bit	10 bit	12 bit



9.4.1 Esempio

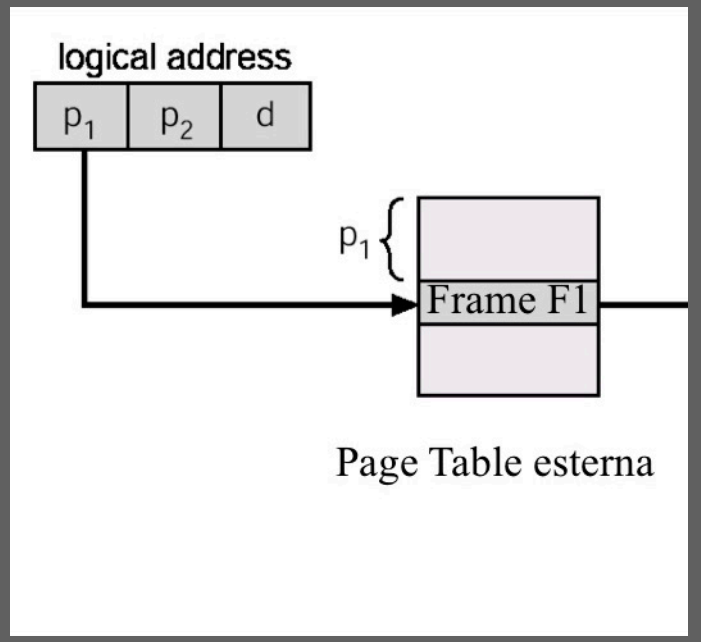
- Per capire il meccanismo numericamente, assumiamo che vengano usati 4 byte per ogni entry delle PT di questo sistema (in realtà basterebbero 20 bit. Perché?)

- Prendiamo la page table più grande, che ha 2^{20} entry, e che ora chiamiamo **PT interna**, e che ha quindi una dimensione di 4×2^{20} byte e occupa esattamente $2^{10} = 1024$ frame.

- Questa PT interna verrà memorizzata in 1024 frame non adiacenti, di cui il SO terrà traccia costruendo una **PT esterna**, che occupa esattamente un frame, dato che ogni entry della PT esterna occupa 4 byte.

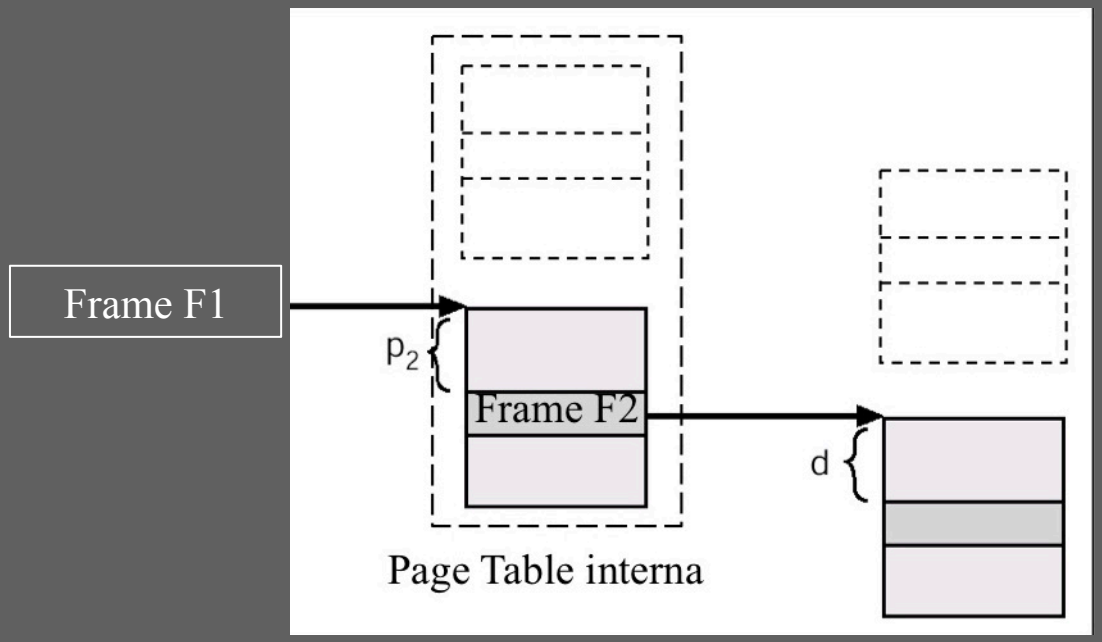
9.4.1 Esempio

- I passi per tradurre un indirizzo logico V scritto su 32 bit in un indirizzo fisico divengono quindi i seguenti:
- I 10 bit più significativi di V (p1) vengono usati per indirizzare una delle 1024 entry della PT esterna (che di fatto è un array con 1024 entry da 4 byte ciascuna).
- In quella entry c'è il numero del frame F1 che contiene una delle pagine in cui è suddivisa la page table interna.



9.4.1 Esempio

- Recuperato F1, con i 10 bit intermedi di V (p2) indirizziamo una delle 1024 entry di F1. In quella entry c'è scritto il numero di un frame F2 che contiene la pagina (notate, il cui numero è dato da p1p2) dell'indirizzo logico V
- L'offset d aggiunto ad F2 genera l'indirizzo fisico



9.4.1 Paginazione a più livelli

- La paginazione a 2 livelli veniva usata, ad esempio, nei vecchi Pentium (lo vediamo più avanti come esempio reale)
- L'architettura VAX (Dec) usava una PT esterna di 4 elementi
- Ma che succede se lo spazio logico è su 64 bit?
- con pagine di 4 Kbyte, 4 byte per ogni entry nelle PT, la PT esterna può occupare fino a 2^{44} byte!!! (provate a verificare questo valore da soli)

9.4.1 Paginazione a più livelli

- Dobbiamo paginare anche la PT esterna!
- in realtà, questo è fatto anche per le architetture a 32 bit. Ad esempio, le architetture SPARC (SUN microsystems) a 32 bit usavano uno schema a 3 livelli. La CPU a 32 bit Motorola 68030 usava una paginazione a 4 livelli
- 4 livelli non sono nemmeno sufficienti per architetture a 64 bit. L'UltraSPARC richiederebbe addirittura 7 livelli di paginazione: se la pagina cercata non è nel TLB, la traduzione dell'indirizzo da logico a fisico richiederebbe di navigare attraverso 7 livelli di pagine in RAM, con un conseguente overhead altissimo

9.4.3 Page Table Invertita (IPT)

- Una soluzione alternativa, adottata in alcune architetture a 64 bit, è chiamata **Tabella delle Pagine Invertita (IPT)**
- Una IPT descrive l'occupazione dei frame della memoria fisica, quindi:
- c'è una sola IPT per tutto il sistema (anzichè una PT per processo), il che è un vantaggio, si spreca meno RAM
- la dimensione della IPT dipende strettamente dalla dimensione della memoria primaria
- **l'indice di ogni entry della IPT corrisponde al numero di un frame in memoria principale**

9.4.3 Page Table Invertita (IPT)

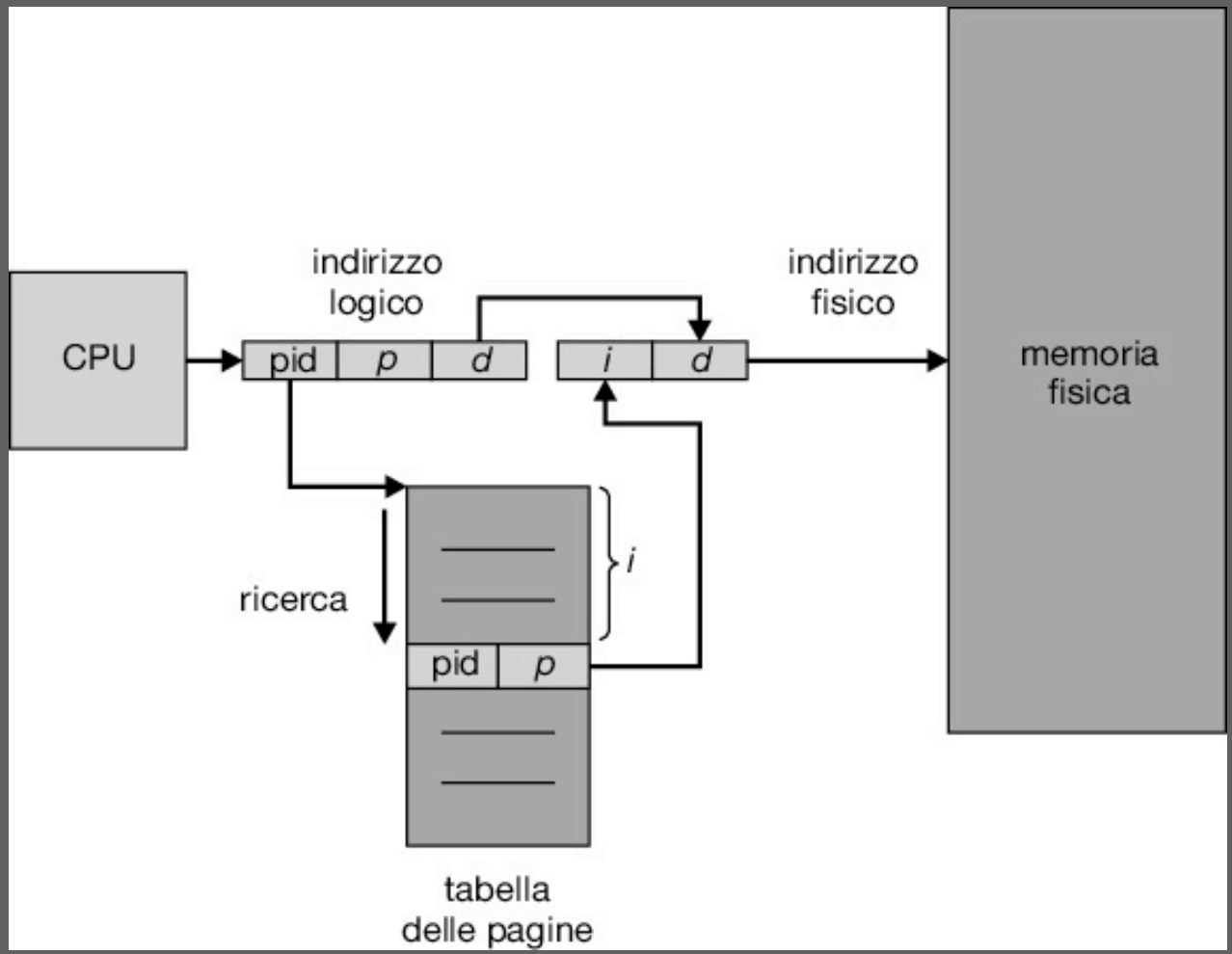
- Ogni entry della IPT contiene una coppia:
  <process-id, page-number> dove:
  - process-id = identificativo del processo che possiede la pagina
  - page-number = numero della pagina contenuta nel frame corrispondente a quella entry

- Ogni indirizzo logico generato dalla CPU è una tripla:
  <process-id, page-number, offset>

- Per generare un indirizzo fisico si cerca nella IPT la coppia <process-id, page-number>. Se la si trova all'*i*-esimo elemento, si genera l'indirizzo fisico <*i*,offset>

9.4.3 Page Table Invertita (IPT)

- Tabella delle pagine invertita (fig. 9.18):



9.4.3 Page Table Invertita (IPT)

- Con l'IPT si risparmia spazio, ma il tempo di traduzione di un indirizzo da logico a fisico può aumentare enormemente
- Infatti, per generare un indirizzo fisico occorre scorrere ogni entry della IPT fino a trovare quella che contiene la coppia <process-id, page-number> dell'indirizzo logico da tradurre
- Se la IPT è tenuta in RAM, ciò può voler dire centinaia o migliaia di accessi in MP prima di trovare l'entry giusta.
- Tuttavia, usando opportune memorie associative per contenere tutta o parte della IPT, la maggior parte degli indirizzi può essere tradotta senza spreco di tempo

9.6.1 Il supporto alla paginazione della microarchitettura IA-32

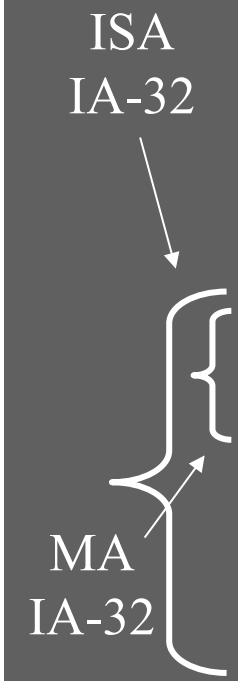
- La paginazione può in teoria essere implementata senza supporto hardware, ma un aiuto dall'hardware è fondamentale se non si vogliono degradare troppo le prestazioni.
- A parte il supporto fondamentale del TLB, tutti i processori moderni forniscono una gamma di facilitazioni per una gestione efficiente dei riferimenti in memoria.
- Un esempio classico è quello della famiglia dei vecchi processori Intel basati sulla microarchitettura IA-32

9.6.1 Microarchitettura IA-32

- La Intel introduce la microarchitettura IA-32 con il 80386 nel 1985, il primo processore a 32 bit, capace di indirizzare 4 Gbyte di memoria (N.B: Dire che un processore è a 32 bit e che può indirizzare 4 Gbyte di memoria sono due concetti distinti!)
- IA-32 però indica anche l'instruction set del processore.
- La microarchitettura IA-32 fu usata dal 1985 (80386) al 1995 (Pentium), e poi sostituita dalla P6 (dal Pentium Pro del '95 al Pentium III del '99) e poi dalla NetBurst (Pentium 4 degli anni 2000)
- Invece, l'instruction set IA-32 fu mantenuto fino al Pentium 4 e ai primi suoi successori immediati

9.6.1 Microarchitettura IA-32

Ecco la lista dei processori Intel a partire dal primo microprocessore della storia, il 4004. Mentre l'instruction set (ISA) IA-32 viene mantenuto fino al Pentium 4, la microarchitettura (MA) IA-32 viene abbandonata a partire dal Pentium Pro (Tanenbaum, Fig. 1.11):



Chip	Date	MHz	Transistors	Memory	Notes
4004	4/1971	0.108	2300	640	First microprocessor on a chip
8008	4/1972	0.108	3500	16 KB	First 8-bit microprocessor
8080	4/1974	2	6000	64 KB	First general-purpose CPU on a chip
8086	6/1978	5–10	29,000	1 MB	First 16-bit CPU on a chip
8088	6/1979	5–8	29,000	1 MB	Used in IBM PC
80286	2/1982	8–12	134,000	16 MB	Memory protection present
80386	10/1985	16–33	275,000	4 GB	First 32-bit CPU
80486	4/1989	25–100	1.2M	4 GB	Built-in 8-KB cache memory
Pentium	3/1993	60–233	3.1M	4 GB	Two pipelines; later models had MMX
Pentium Pro	3/1995	150–200	5.5M	4 GB	Two levels of cache built in
Pentium II	5/1997	233–450	7.5M	4 GB	Pentium Pro plus MMX instructions
Pentium III	2/1999	650–1400	9.5M	4 GB	SSE Instructions for 3D graphics
Pentium 4	11/2000	1300–3800	42M	4 GB	Hyperthreading; more SSE instructions

9.6.1 Microarchitettura IA-32

- Nella microarchitettura IA-32, a scelta del SO che gira sul processore, è possibile usare pagine da 4 Kbyte o da 4 Mbyte. Nel caso di pagine da 4 Kbyte il processore adotta uno schema di paginazione a due livelli.

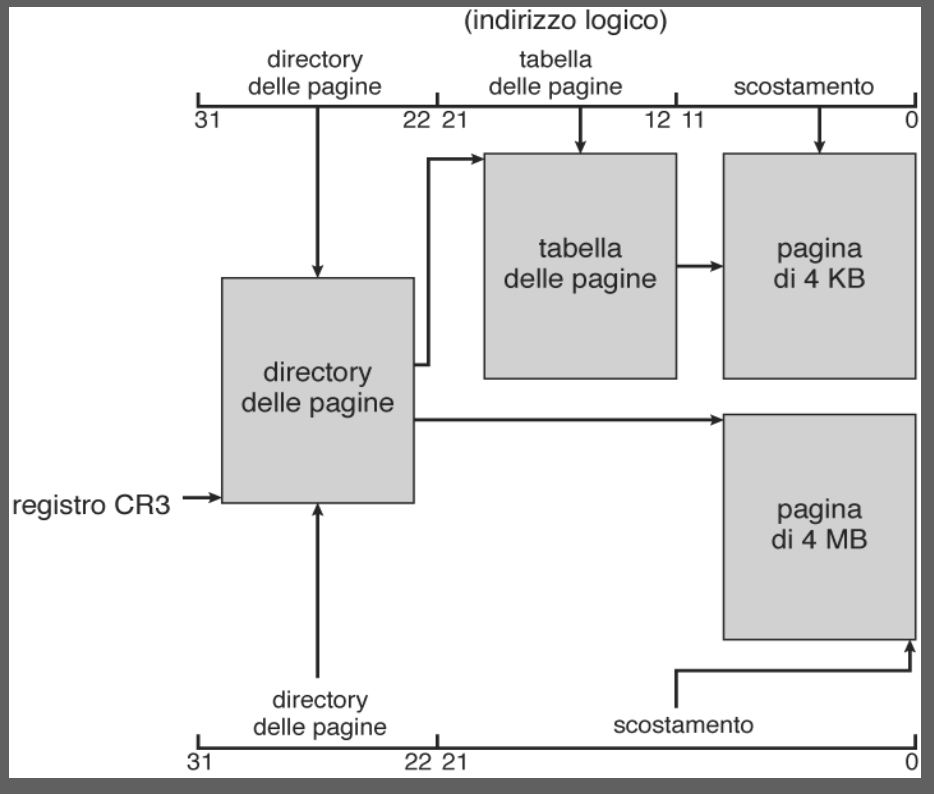
- Questo schema fu mantenuto anche nei processori basati sulle successive microarchitetture P6 e NetBurst, che indirizzavano sempre 4 Gbyte di RAM (in realtà negli ultimi Pentium 4 era possibile indirizzare fino a 64 Gbyte di RAM)

- La traduzione degli indirizzi da logici a fisici avviene nel solito modo visto, attraverso l'unità di paginazione:

9.6.1 Microarchitettura IA-32

- Fig. 9.23: notate come lo schema di traduzione passi da uno a due livelli a seconda che le pagine siano da 4 Mbyte o da 4 Kbyte.
- Il registro CR3, accessibile solo da SO, contiene in ogni istante l'indirizzo di partenza della tabella delle pagine (esterna) del processo

page number		page offset
p_1	p_2	d
10	10	12



9. Conclusioni

- Le tecniche di gestione della MP cercano di aumentare il più possibile il livello di multiprogrammazione:
  - supportando lo swapping e la rilocazione dinamica del codice
  - Limitando la frammentazione
  - favorendo la condivisione del testo fra diversi processi
- Ma queste tecniche rendono il sistema più complesso e introducono delle inefficienze, che devono essere limitate da opportuni supporti hardware.

9. Conclusioni

- Abbiamo infatti visto tecniche di gestione della memoria molto diverse, da quelli semplici per sistemi single user alla paginazione a più livelli
- E in pratica, è il supporto hardware che determina la classe di tecniche implementate dal SO, che deve garantire una corretta gestione della memoria senza introdurre troppo overhead.
- A sua volta, il supporto hardware è progettato pensando al tipo di gestione della MP che si vuole implementare

Per chi vuole approfondire

- Sezione 9.6: le architetture Intel a 32 e 64 bit
- Sezione 9.7: l'architettura ARMv8