CAPITOLO 8

GESTIONE DELLA MEMORIA

Obiettivi del S.O.:

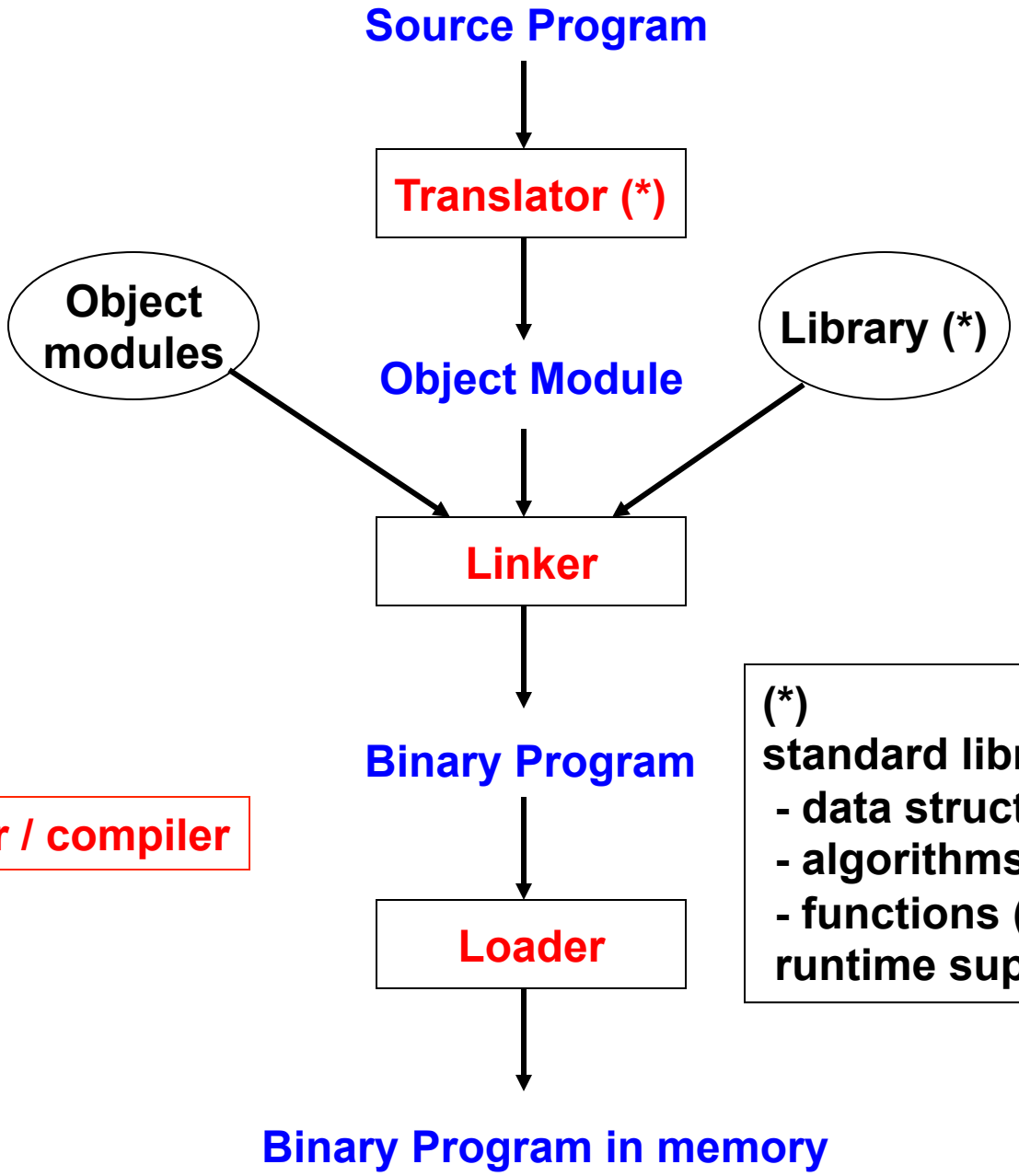
1. Allocare più processi in memoria, per avere parallelismo (*) e migliorare la performance del sistema. (Nei sistemi batch questo non avveniva.)
2. Proteggere la memoria dei processi, impedendo ai processi di accedere alle aree degli altri processi.

Il punto 1 viene rafforzato se il S.O. offre la **memoria virtuale**, cioè la possibilità di avere parte delle aree testo-dati-stack-heap dei processi in memoria e parte su disco.

(*) parallelismo tra uso di CPU e I/O nel caso di uniprocessor, parallelismo anche tra uso di CPU nel caso di multiprocessor.

8.1 - Traduzione, linking, loading: alcune precisazioni

Traduzione, linking, loading di un programma sorgente:



(*) assembler / compiler

(*)
standard library:
- data structures (queues,…)
- algorithms (sorting,…)
- functions (square root)
runtime support

- **Source program**: programma in assembly, C, C++,….

- **Object module**: programma in linguaggio macchina con riferimenti a funzioni / dati di altri object module e/o librerie. Le istruzioni ed i dati dell'area dati sono allocati ad indirizzi di memoria. L'indirizzo da cui far partire il programma può essere un parametro passato al traduttore, altrimenti è 0.

- **Binary program**: programma eseguibile. Gli indirizzi di memoria assegnati dal traduttore possono essere stati **rilocati** (cambiati) dal linker per evitare overlapping con gli indirizzi degli altri moduli / librerie.

- **Binary program in memoria**: programma eseguibile in memoria. Il loader può aver rilocato gli indirizzi di memoria generati dal linker per evitare overlapping di programmi in memoria.

Esempio di un programma **P** in assembly:

```
        START   500             /*il programma ha origine all'indirizzo 500*/
        ENTRY   TOTAL           /* simbolo definito nel presente modulo*/
                                /* e "visibile" da altri moduli */
        EXTRN   MAX, ALPHA      /* simboli definiti in altri moduli*/
        READ    A               /* simbolo definito nel presente modulo*/
LOOP    ..                      /* LOOP è una label */
        ..
        MOVE    R1,  ALPHA      /*R1 è il registro 1*/
        BC      ANY, MAX        /*ANY codifica una condizione del CC*/
        ..
        ..
        BC      LT, LOOP        /*LT codifica una condizione del CC*/
        STOP
A       DS      1               /* definizione del simbolo A */
TOTAL   DS      1               /* definizione del simbolo TOTAL */
        END
```

Abbiamo assunto un linguaggio assembly in cui le istruzioni hanno la forma seguente:

[label] opcode operand1 operand2

Assumiamo anche che in linguaggio macchina un'operazione occupi 4 byte:

- **opcode** occupa 2 cifre, dove 09=READ, 04=MOVE, 06=BC.
- **operand1** occupa 1 cifra. Solitamente è un registro, ma nella **BC** l'operando codifica un valore possibile del **C**ondition **C**ode, per esempio **ANY**, **LT**. Assumiamo che in linguaggio macchina il codice decimale di **ANY** sia **6** e di **LT** sia **1**.
- **operand2** occupa 5 cifre.

Assumiamo che gli indirizzi riguardino parole di 4 byte.

Riportiamo sulla destra alcune istruzioni macchina per **P** generate dall'assembler:

```
        START   500
        ENTRY   TOTAL           ……
        EXTRN   MAX, ALPHA      ……
        READ    A               09 0 00700 /*09 = READ*/
LOOP    ..                      /* NB: l'indirizzo è 00504 (500 + 4) */
        ..
        MOVE    R1,  ALPHA      04 1 00000  /*04 = MOVE, 1 = R1*/
        BC      ANY, MAX        06 6 00000 /* 06 = BC, 6 = ANY*/
                /* i simboli ALPHA e MAX saranno risolti dal linker*/
        ..
        BC      LT, LOOP        06 1 00504 /* 06 = BC, 1 = LT */
        STOP
A       DS      1               /* supponiamo indirizzo 00700 */
TOTAL   DS      1
        END
```

Assumiamo di avere un altro modulo **P'** con le definizioni di **MAX** e di **ALPHA**:

```
START   200
ENTRY   ALPHA
ENTRY   MAX
.......
.......
```

Supponiamo che questo modulo abbia in totale 400 parole.

Il linker non può caricare **P'** all'indirizzo 200 e **P** all'indirizzo 500: ci sarebbe un overlapping tra gli indirizzi 500 e 599.

Assumiamo che **P'** venga caricato all'indirizzo 200 e gli indirizzi di **P** vengano rilocati, ponendo **P** all'indirizzo 600.

Alcune istruzioni macchina per **P** modificate dal linker:

```
        START   500
        ENTRY   TOTAL           ……
        EXTRN   MAX, ALPHA      ……
        READ    A               09 0 00800 /* modifica */
LOOP    ..                      /*NB: l'indirizzo è 604 (500 + 4 + 100)*/
        ..
        MOVE    R1,  ALPHA      04 1 00201  /* 201 = 200+1*/
        BC      ANY, MAX        06 6 00202 /* 202 = 200+2 */
                /* i simboli ALPHA e MAX sono stati risolti dal linker*/
        ..
        BC      LT, LOOP        06 1 00604 /* modifica */
        STOP
A       DS      1               /* il nuovo address è 00800 */
TOTAL   DS      1
        END
```

Entrambi **P** e **P'** potrebbero essere rilocati dal loader. Per esempio, se la memoria da 200 a 299 non è libera, il loader potrebbe rilocarli aggiungendo 100 byte ad ogni address.

Alcune istruzioni macchina per **P** modificate dal loader:

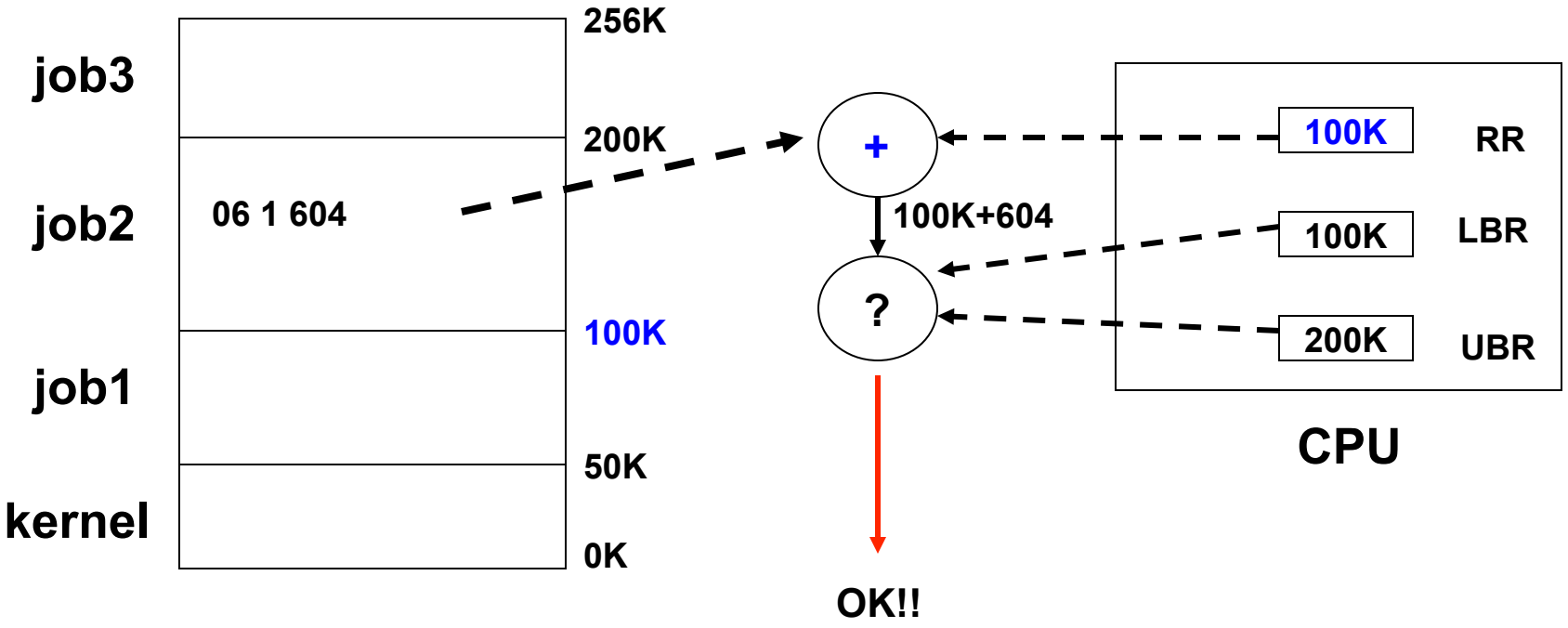
```
        START   700              /* modifica */
        ENTRY   TOTAL            ……
        EXTRN   MAX, ALPHA       ……
        READ    A                09 0 00900 /* modifica */
LOOP    ..                       /*NB: l'indirizzo è 704 (500+4+100+100)*/
        ..
        MOVE    R1,  ALPHA       04 1 00301  /* 301 = 200+1+100*/
        BC      ANY, MAX         06 6 00302 /* 302 = 200+2 +100*/

        ..
        BC      LT, LOOP         06 1 00704 /* modifica */
        STOP
A       DS      1                /* il nuovo address è 00900 */
TOTAL   DS      1
        END
```

Osservazioni:

- Nell'esempio precedente abbiamo visto un esempio di **rilocazione statica** da parte del loader: il loader modifica gli indirizzi del programma prima che il programma venga eseguito.

- Con un opportuno supporto hardware, per esempio con l'impiego di un Relocation Register e della logica necessaria a sommare agli indirizzi il valore del Relocation Register, possiamo avere **rilocazione dinamica** (vedi slide seguente, argomento comunque già affrontato nel Capitolo 4).

Esempio già visto nel Capitolo 4 con impiego del RR ed anche dei registri LBR e UBR:



Anche il linking può essere dinamico e non statico come nell'esempio precedente.

Questo è il caso, ad esempio, delle DLL.

Con il linking dinamico, il linker statico sostituisce un riferimento ad un simbolo definito in un modulo esterno con la logica per effettuare il calcolo dell'indirizzo durante l'esecuzione del programma.

8.2 - Modello di allocazione della memoria

Durante l'esecuzione del programma, vengono allocati due tipi di dati:

- le variabili il cui scope è associato a **blocchi**, **funzioni**, **procedure**. Queste variabili vengono allocate sullo **stack**, a causa della natura last-in-first-out della loro allocazione / deallocazione.

- i **dati creati dinamicamente** con i costrutti appositi, quali **malloc** e **calloc** nel linguaggio C oppure **new** in C++ e Java. Parleremo di dati PCD (**P**rogram **C**ontrolled **D**ynamic data). Vengono allocati nell'area di **heap**.

```
#include <stdio.h>
int a=5; /*variabile globale - area dati*/
int b=6; /*variabile globale - area dati*/

main(  ){
    int x = 10;   /*variabile locale*/
      /*(punto 1)*/
    int y = f1(x,b); /*variabile locale*/
      /*(punto 5)*/
    printf("ecco il valore di y: %d\n",y);
}

int f1(int s, int t){
    int u = a+s+t; /*variabile locale*/
      /*(punto 2)*/
    int v = f2(u); /*variabile locale*/
      /*(punto 4)*/
    return v;
}

int f2 (int h){
    int k = h+15; /*variabile locale*/
      /*(punto 3)*/
    return k;
}
```

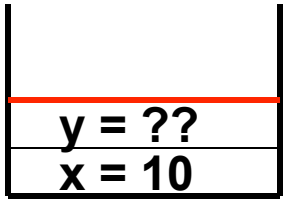
Esempio di programma in C già visto nel Cap. 4. Nella prossima slide vedremo la foto dello stack quando il programma si trova nei punti segnati in rosa.

Ogni frame contiene, oltre a componenti che trascuriamo:

- Parametri attuali,
- Variabili locali,
- Return Address (RA),
- Pointer al frame della procedura chiamante.

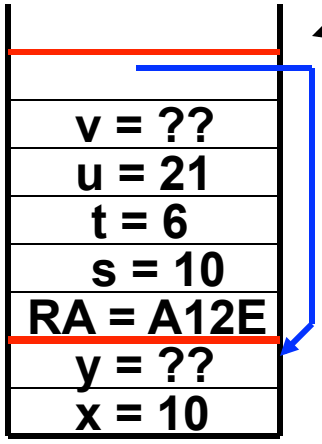
Assunzioni: il programma compilato è tale che:

- l'istruzione di main che segue la chiamata a f1 si trova all'indirizzo A12E,
- l'istruzione di f1 che segue la chiamata a f2 si trova all'indirizzo A39F.



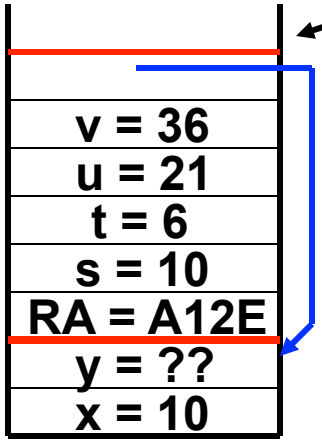
stack top

(1)



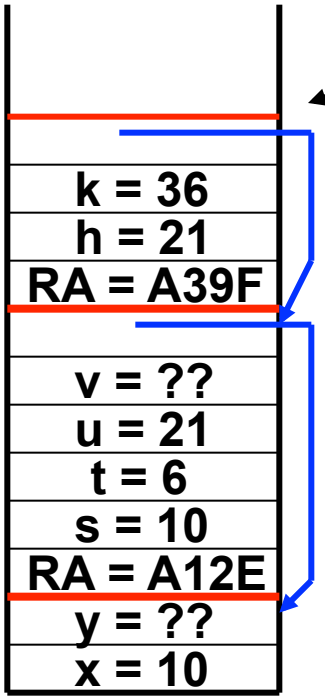
stack top

(2)



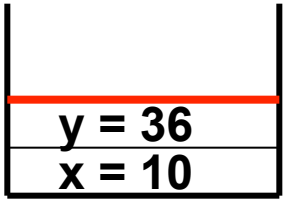
stack top

(4)



stack top

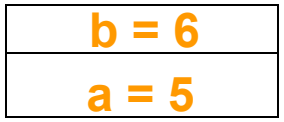
(3)



stack top

(5)

Area dati:

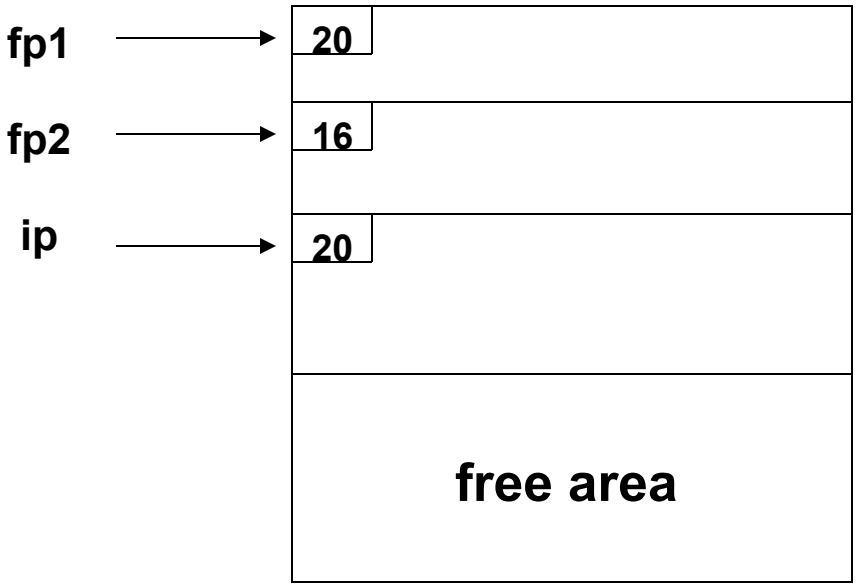


N.B. il valore delle variabili non inizializzate è indicato con "??". In realtà un valore di default viene assegnato, dipende dal linguaggio.

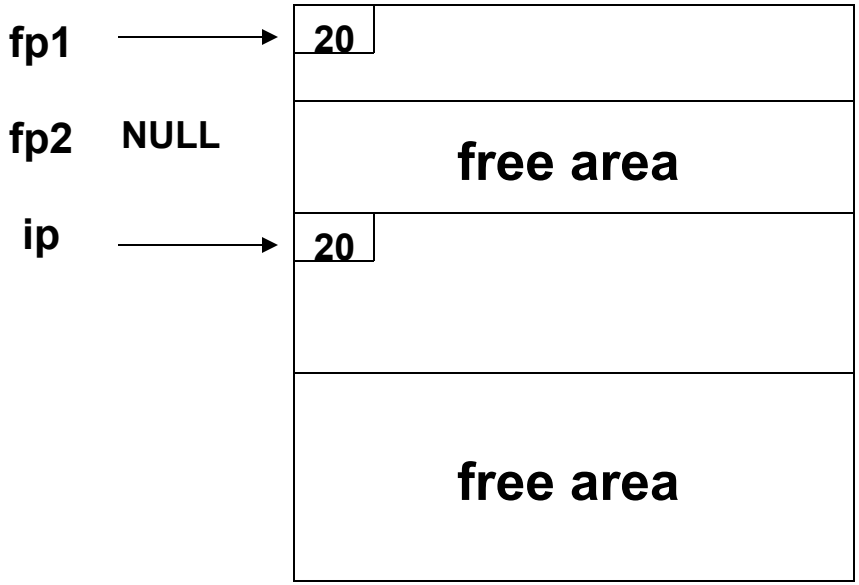
Assumiamo architettura a 16 bit: gli indirizzi sono rappresentabili con 4 cifre esadecimali.
Esempio:
A12E = 1010 0001 0010 1110.

Esempio di uso di dati PCD in C

```
.....
float *fp1, fp2; /* pointers a float */
int *ip;              /* pointer a int */
fp1= (float *) calloc(5,sizeof(float));
 /*lo heap allocator alloca nello heap un'area di memoria per 5 float e
    restituisce un pointer a tale area*/
fp2= (float *) calloc(4,sizeof(float));
  /*lo heap allocator alloca nello heap un'area di memoria per 4 float
    e restituisce un pointer a tale area*/
ip  = (int *) calloc(10, sizeof(int));
  /*lo heap allocator alloca nello heap un'area di memoria per 10 int e
    restituisce un pointer a tale area*/
free(fp2);
  /*lo heap allocator libera l'area di memoria puntata dal pointer fp2*/
.....
```



"Foto" dello heap dopo le tre calloc.
Ogni area allocata ha un campo contenente la dimensione: serve per facilitare la deallocazione



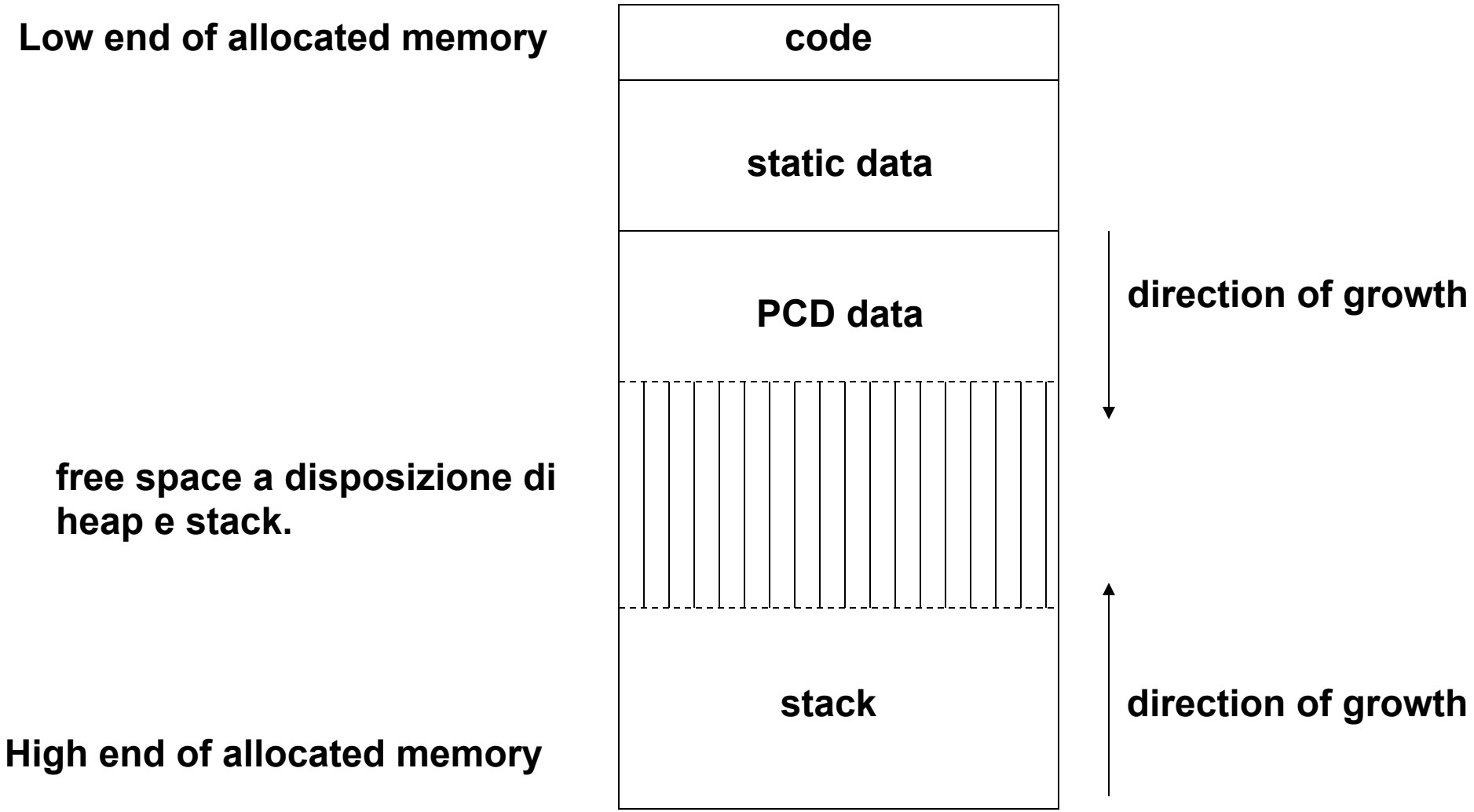
"Foto" dello heap dopo la free.
Il pointer **fp2** è ora nullo e l'area di memoria che puntava è diventata libera

Abbiamo usato il termine "heap allocator". Cosa intendiamo?

- Se il linguaggio offre i costrutti per allocazione/deallocazione di memoria nello heap, il linguaggio deve offrire delle routine per gestire lo heap che vengono invocate da tali costrutti.
- Queste routine fanno parte del **R**un**T**ime **S**upport (**RTS**) del linguaggio, si trovano nella runtime library e vengono collegate al programma dal linker.
- Possiamo chiamare heap allocator/deallocator le routine che si occupano di allocare/deallocare memoria nello heap.

N.B. Il RTS del linguaggio può contenere anche altre funzioni. Esempio: in Java il RTS comprende il **garbage collector**, che libera la memoria allocata nello heap che non ha più puntatori che la riferiscono.

Modello di allocazione della memoria usato dal S.O.:
-il codice ed i dati statici sono allocati in un'area di dimensione statica,
-l'area di heap e lo stack condividono un'area di dimensione statica, ma le due aree hanno dimensione variabile e "crescono" in direzione opposta.

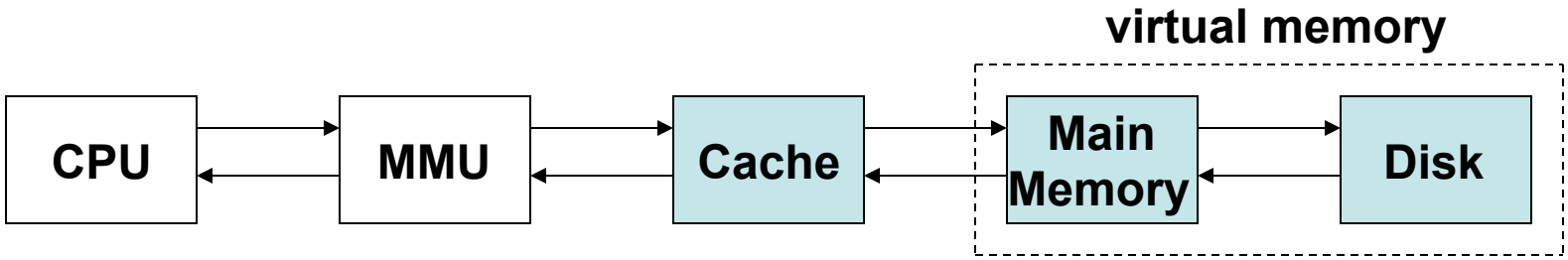


8.3 - Gerarchia di memoria

- Nel mondo ideale l'unità di memoria avrebbe una velocità tale da non rallentare la CPU quando la CPU accede per leggere le istruzioni o per leggere/scrivere i dati.

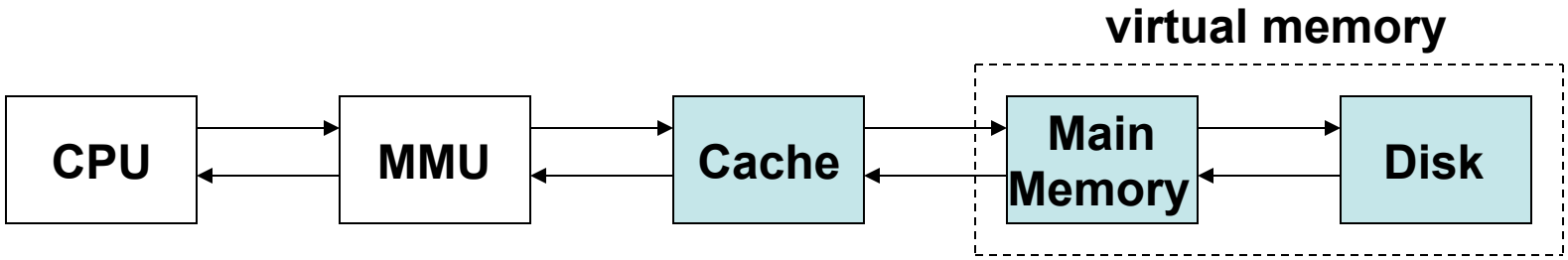
- Le unità di memoria pù veloci sono però le più costose: i sistemi di computazione offrono una gerarchia di unità di memoria con velocità diverse: le unità più veloci e costose sono quelle di minor capacità.

- I programmi contengono riferimenti alla aree testo, dati, stack, heap: l'assunzione è che queste aree siano fisicamente su un'unica unità di memoria. In altre parole, il programmatore ragiona come se ci fosse un'unica unità di memoria, che chiameremo main memory.

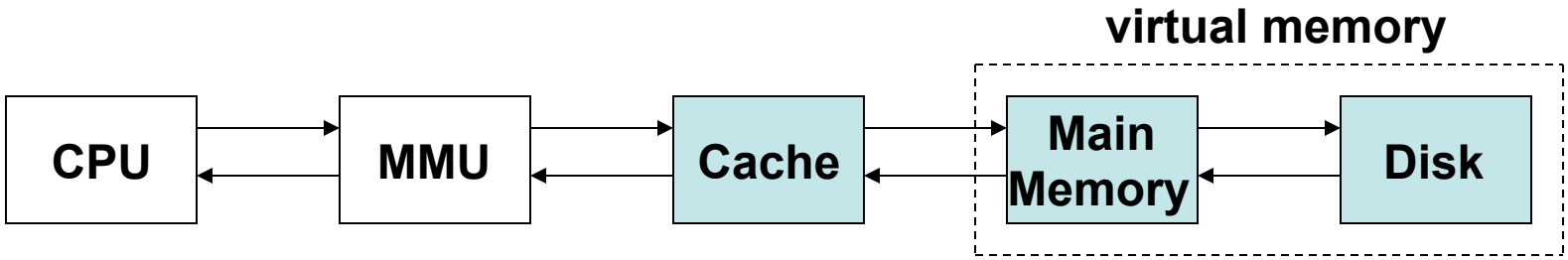


- La CPU accede alla gerarchia di memoria tramite la MMU (Memory Management Unit), che traduce gli **indirizzi logici** in **indirizzi fisici**.

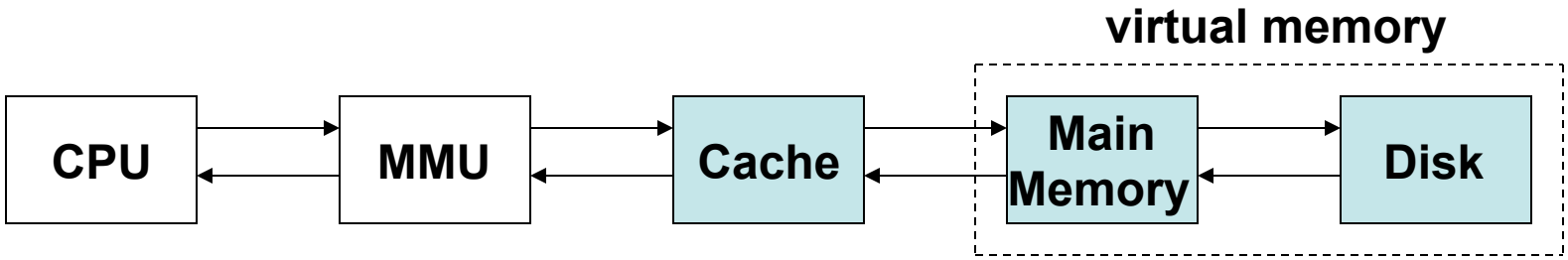
- Il codice di un programma contiene riferimenti alle aree testo, dati, stack, heap che sono assunte essere in main memory: gli indirizzi logici lavorati dalla MMU e gli indirizzi fisici prodotti della MMU sono indirizzi della main memory.



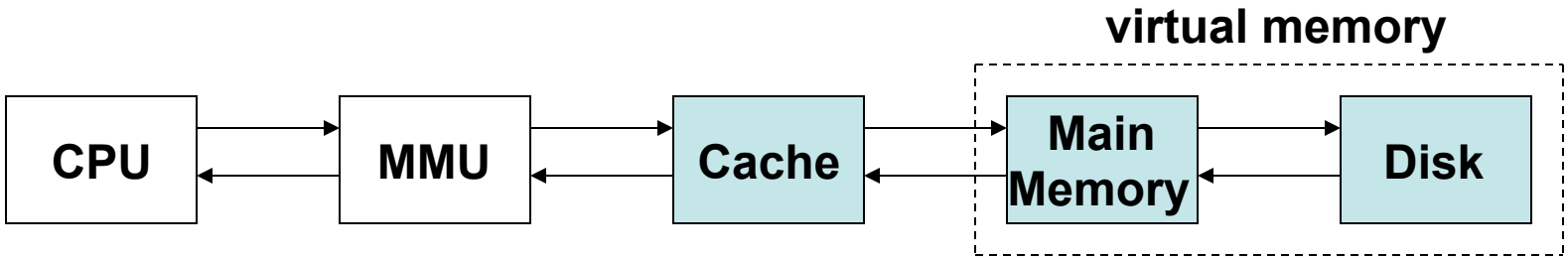
- Il byte che si trova all'indirizzo fisico della main memory generato dalla MMU viene in realtà cercato nella cache. Se non viene trovato, allora è necessario trasferirlo dalla main memory alla cache.
- Gli aggiornamenti effettuati sulla cache devono prima o poi avere un corrispettivo in main memory: servono strategie per trasferire byte dalla cache alla main memory.
- Per ragioni di efficienza, i trasferimenti tra cache e main memory non riguardano singoli byte, ma blocchi di memoria (cache block, cache line).



- Lo schema in figura è semplificato: di norma abbiamo più livelli di cache, esempio:
  - un livello di cache L1 sul chip della CPU
  - un livello di cache L2 connesso alla CPU
  - un livello di cache L3 sul chip della memoria.



- Osservazione: la gestione della cache viene effettuata dall'hardware, non dal S.O..
- Il S.O., consapevole dell'esistenza della cache, può impiegare tecniche per avere una buona performance della cache relativamente agli accessi alle strutture dati del S.O..
- Il S.O., consapevole dell'esistenza della cache, può favorire gli switch tra thread del medesimo processo quando possibile. (Ciò perché i thread condividono aree testo e dati.)



- Se il sistema offre la **memoria virtuale**, una parte delle aree di memoria del processo non si trovano nella main memory ma sul disco. Gli indirizzi fisici generati dalla MMU sono virtuali.
- Se l'indirizzo virtuale proposto dalla CPU non ha un corrispondente indirizzo reale in main memory, il byte deve essere trasferito dal disco alla main memory.
- Gli aggiornamenti effettuati sulla main memory devono prima o poi avere un corrispettivo nel disco: servono strategie per trasferire byte dalla main memory al disco.
- Per ragioni di efficienza, i trasferimenti tra main memory e disco non riguardano singoli byte, ma blocchi di memoria (pagine, segmenti,…).

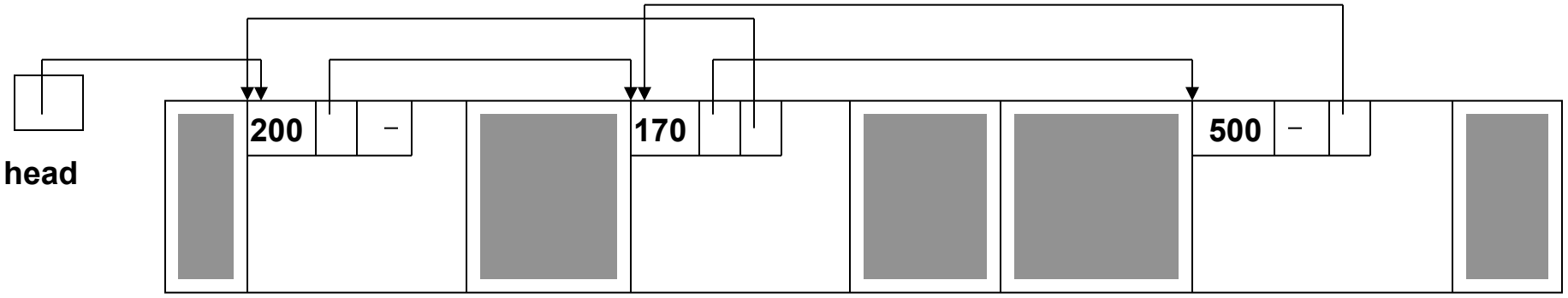
8.4 – RIUSO DELLA MEMORIA

L'allocazione della memoria è un problema che possiamo vedere a due livelli:

- il S.O. deve allocare memoria per i processi;
- il RTS deve allocare memoria per i dati PCD.

In entrambi i casi è necessario riutilizzare la memoria liberata.

Per farlo, è possibile tener traccia della memoria libera in una **free list**. Esempio di double linked free list:



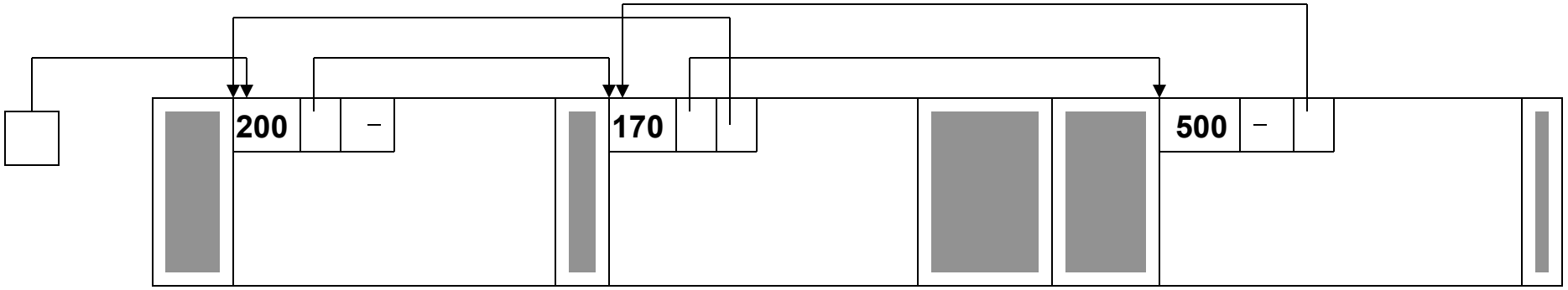
Usando la free list, abbiamo almeno 3 tecniche per allocare un'area di **k** byte:

- **first fit**: si cerca la prima area libera di dimensione **>=k**. Se l'area ha dimensione **d**, si allocano **k** byte e l'area rimanente di **d-k** byte rimane nella free list;

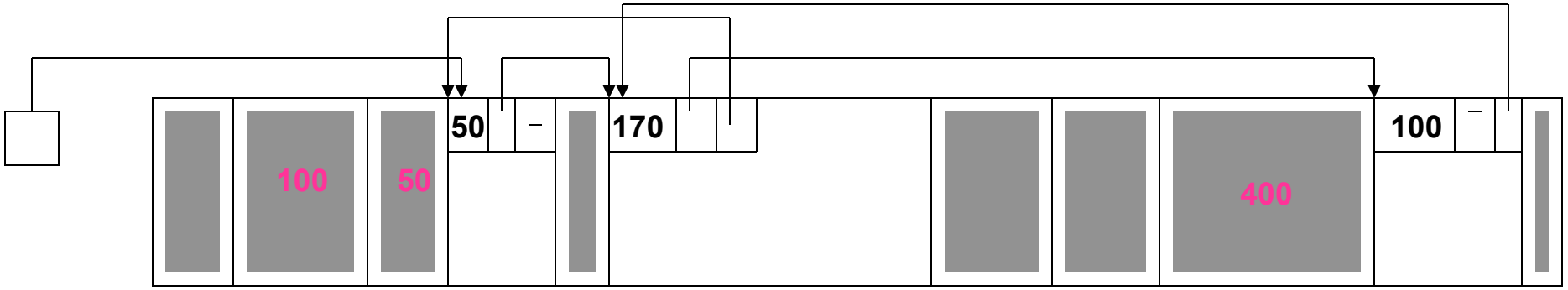
- **best fit**: si cerca l'area libera più piccola tra quelle di dimensione **>=k**. Se l'area ha dimensione **d**, si allocano **k** byte e l'area rimanente di **d-k** byte rimane nella free list;

- **next fit**: si adotta la first fit partendo, però, dal punto in cui era stata effettuata l'ultima allocazione.

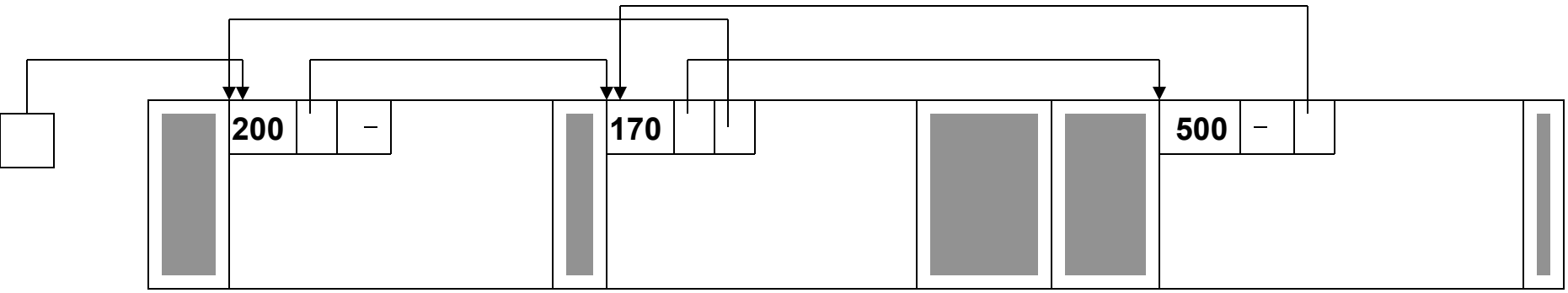
Esempio first fit – situazione iniziale:



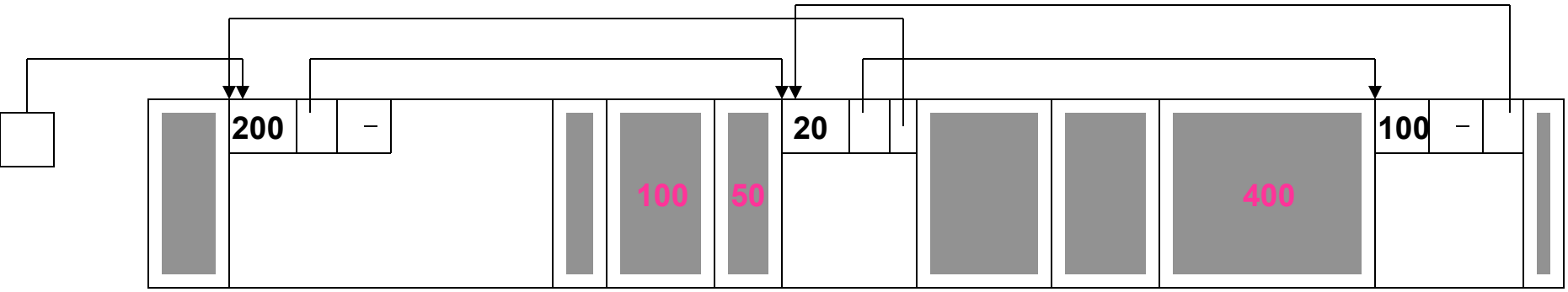
Supponiamo di allocare tre aree da 100, 50, 400 byte:



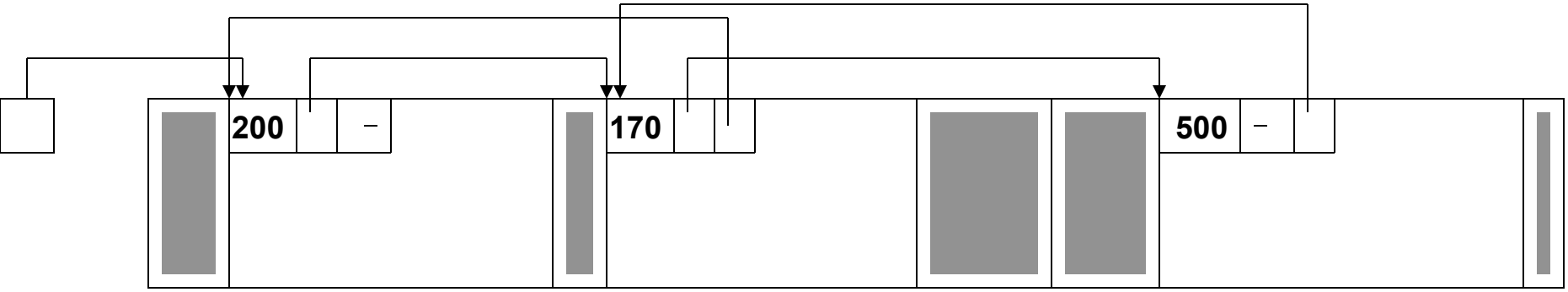
Esempio best fit – situazione iniziale:



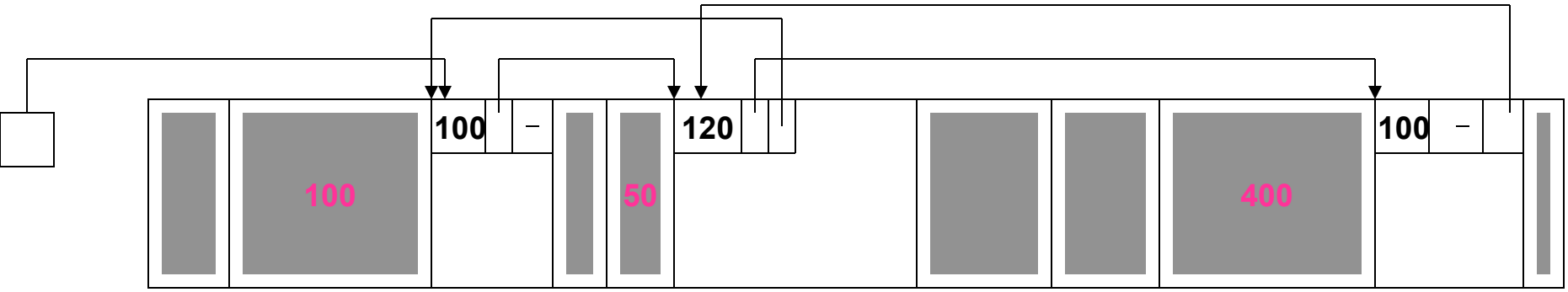
Supponiamo di allocare tre aree da 100, 50, 400 byte:



Esempio next fit:



Supponiamo di allocare tre aree da 100, 50, 400 byte:



Osservazioni:

- Nel caso della first fit un'area libera può spezzarsi più volte. C'è la tendenza al formarsi di aree libere di dimensione piccola che rischiano di essere inutili.

- Nel caso della best fit vengono preservate la aree di dimensioni maggiore, ma a lungo andare il problema si presenta ugualmente. Inoltre la gestione della lista, oppure la ricerca, sono costose.

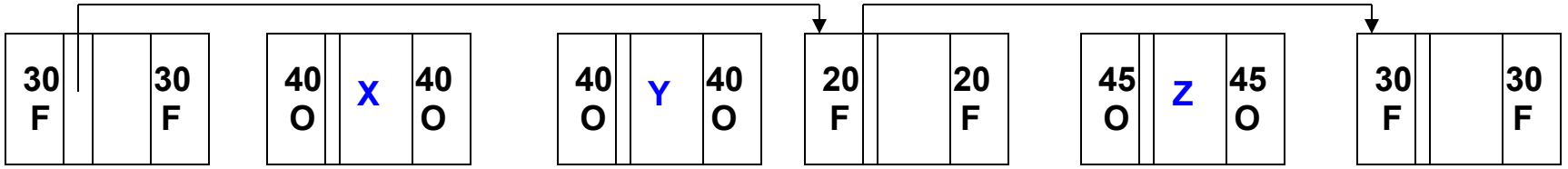
- La next fit è un compromesso.

- Nella pratica, la first fit e la next fit sono più performanti della best fit.

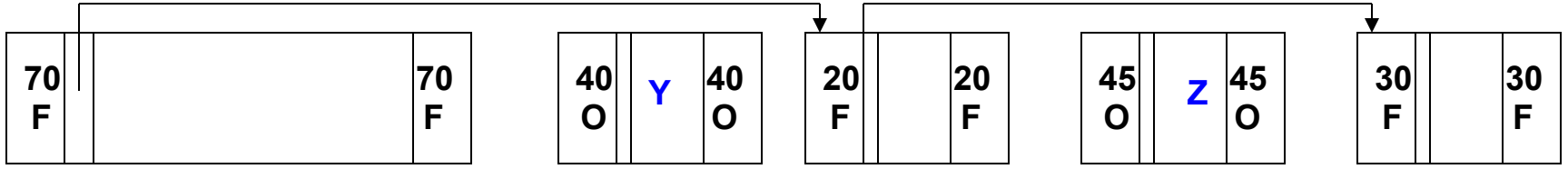
Definizione: l'esistenza di aree di memoria non utilizzabili in un sistema di computazione è detta **frammentazione**.

Tecniche per prevenire/contrastare la frammentazione:

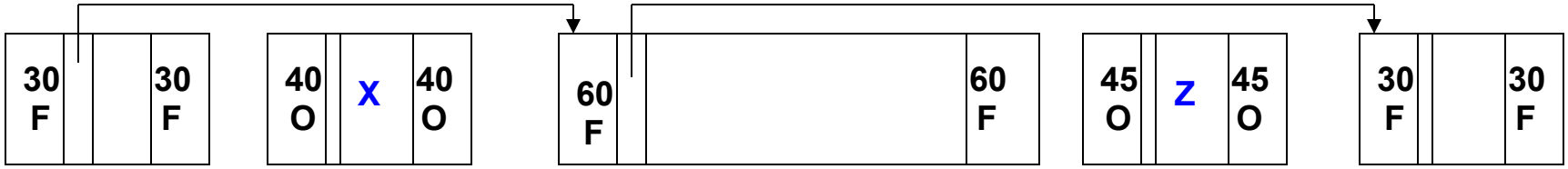
- **boundary tags** (esempio nella prossima slide):
  - all'inizio ed alla fine di ogni area c'è un tag, contenente la coppia (stato free/occupied, dimensione)
  - le aree libere sono collegate da una catena di pointer
  - quando un'area occupata viene liberata, se è possibile la si unisce ad un'area libera confinante (oppure a due).
- **memory compaction** (esempio nella slide successiva):
  - le aree libere sono collegate da una catena di pointer
  - a determinati intervalli di tempo tutte le aree libere vengono unite in un'unica area libera.
- **powers-of-two allocator** (non li vedremo).
- **buddy systems** (non li vedremo).



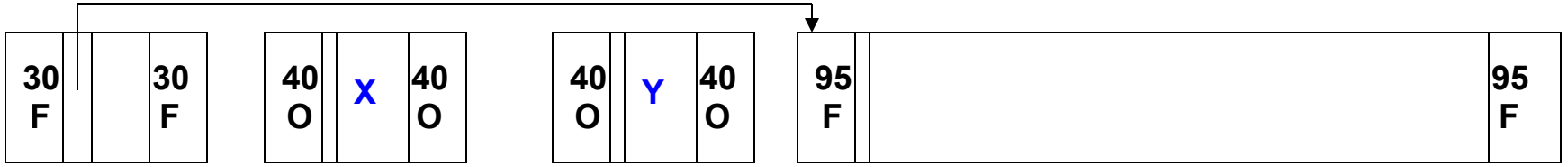
Se si libera l'area X, la si unisce alla free area che la precede:

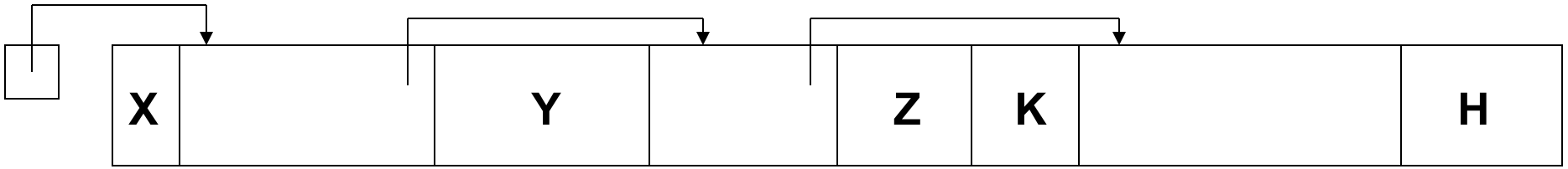


Se, invece, si libera l'area Y, la si unisce alla free area che la segue:

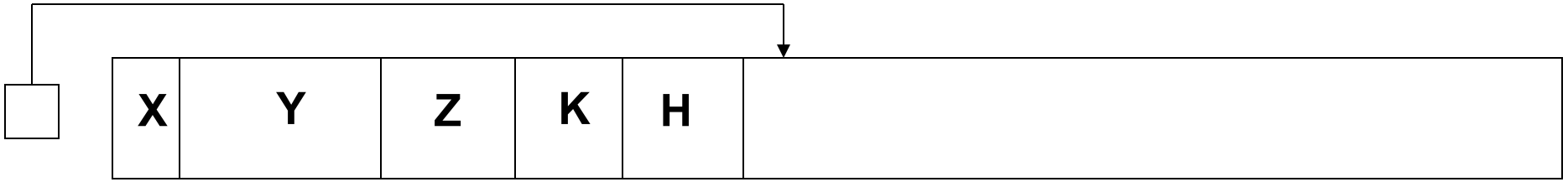


Infine, se si libera l'area **Z**, la si unisce alla free area che la precede ed a quella che la segue:





Dopo il compattamento:



E' richiesta la rilocazione dei programmi cui le aree occupate sono allocate.

Ragionevole solo se la rilocazione è semplice, esempio se abbiamo il registro RR, perché in tal caso è sufficiente modificarne il valore.

8.5 - ALLOCAZIONE DELLA MEMORIA

- Si parla di **allocazione contigua** della memoria quando ogni processo è allocato in una singola area di memoria contigua.

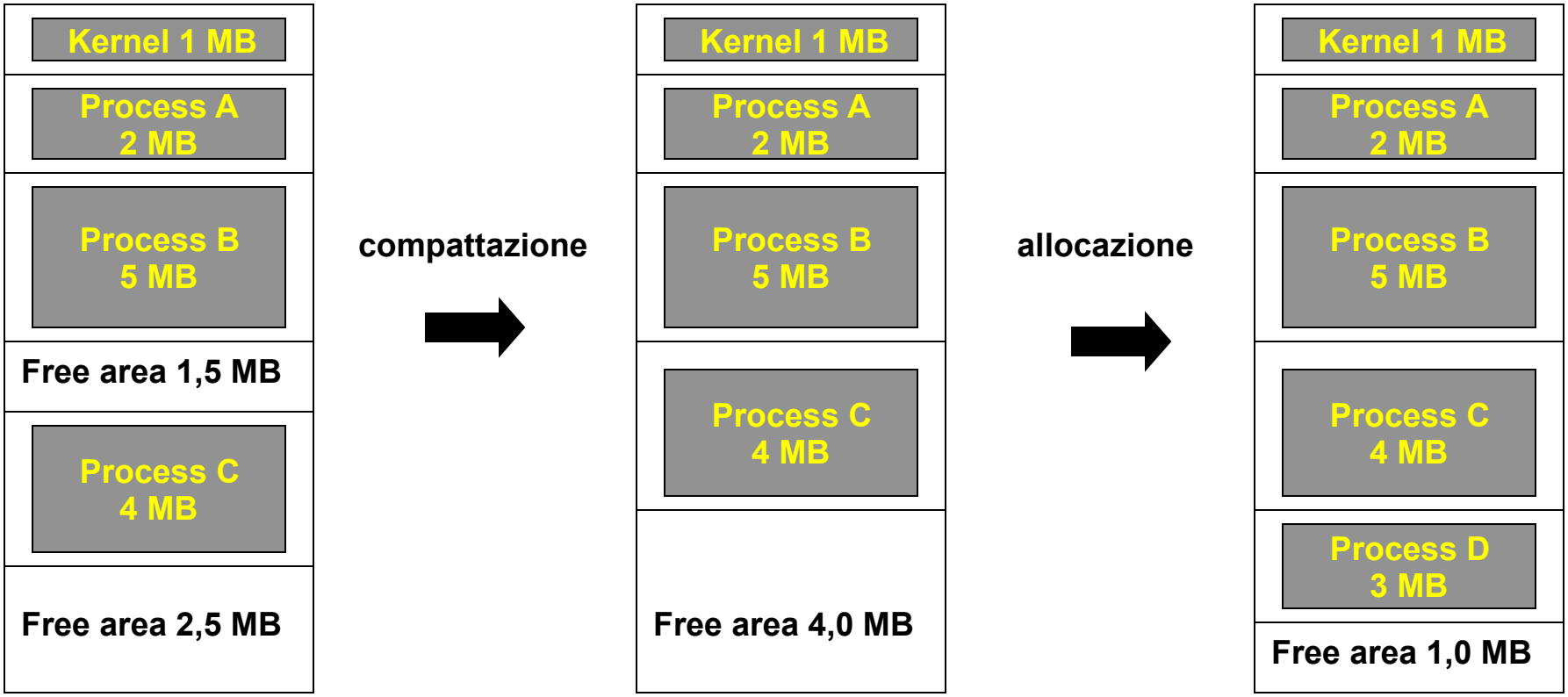
- La protezione della memoria è semplice: è sufficiente disporre dei registri LBR / UBR.

- La rilocazione dei processi è semplice: è sufficiente disporre del registro RR.

- Allocare in memoria un processo che richiede **k** byte è possibile solo se è disponibile un'area di memoria dimensione **>=k**.

  Sono pertanto necessarie le tecniche per prevenire / contrastare la frammentazione della memoria appena trattate nella Sezione 8.4.

Esempio. Supponiamo di dover allocare il processo D da 3,0 MB quando aree libere da >= 3,0 MB non sono disponibili. Usando la compattazione è semplice:



- L'allocazione contigua può essere integrata con lo **swapping** (vedi Sezione 4.4 e 4.7):
  - alcuni processi sono in memoria
  - altri processi sono sullo **swap device** (porzione di disco).

- I processi in memoria waiting o ready possono essere **swapped out**. Questo compito spetta ad un processo di sistema (swapper - processo 0 - nel caso di UNIX V).

- I processi sullo swap device ready possono essere **swapped in**. Questo compito spetta ad un processo di sistema (swapper - processo 0 - nel caso di UNIX V).

- Se si dispone del registro RR il processo swapped out potrà, in seguito, essere facilmente swapped in anche in un'area di memoria diversa.

- Si parla di **allocazione non contigua** della memoria quando ogni processo può essere allocato in più aree di memoria non adiacenti.

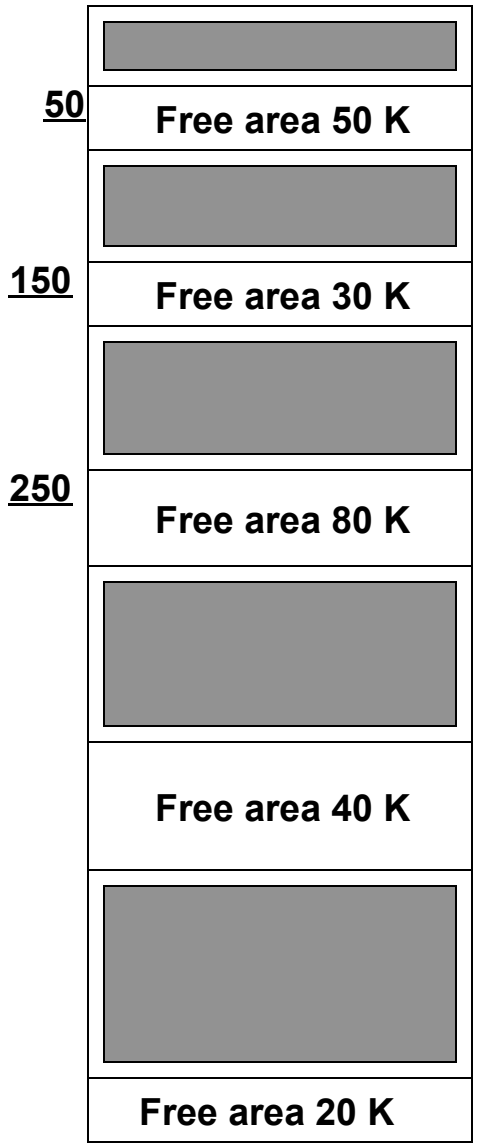
- Ogni porzione del processo che viene allocata in un'area contigua è detta **componente**.

- Allocare in memoria un processo che richiede **k** byte non necessita di una singola area di memoria di dimensione **>=k**.

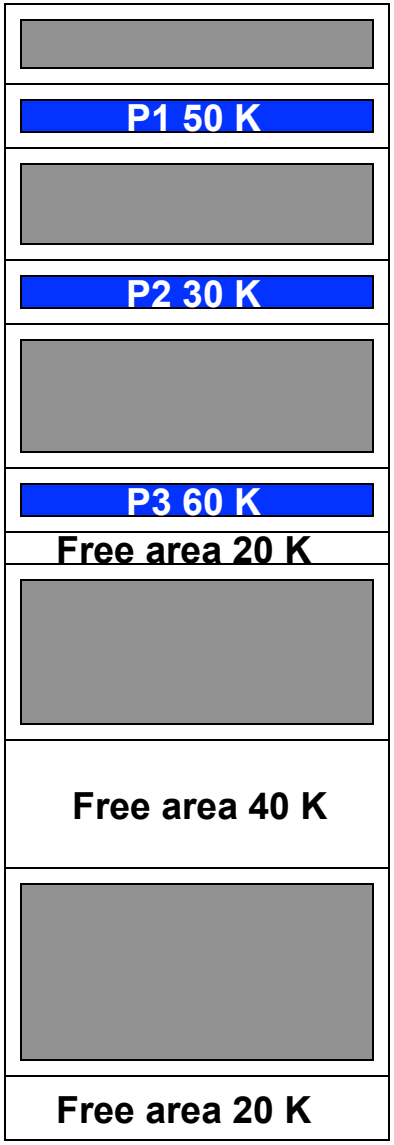
- Il problema della frammentazione è risolto, oppure limitato (vedremo meglio in seguito).

- La protezione della memoria e la traduzione da indirizzi logici ad indirizzi fisici è più complicata rispetto al caso dell'allocazione contigua (non bastano RR, LBR, UBR).

Allocazione non contigua - idea - 1:

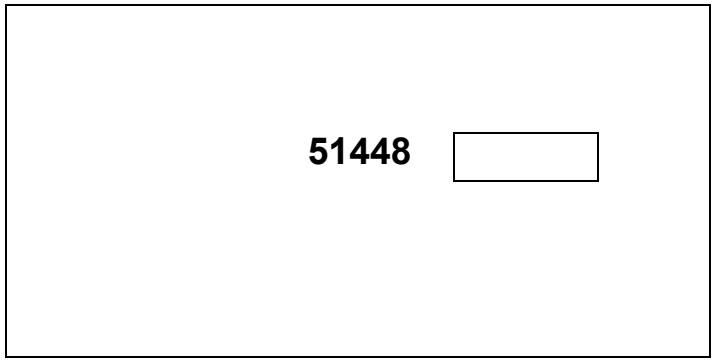


Allocazione di un processo P da 140 K: P viene diviso in 3 componenti: P1,P2,P3 da 50K, 30K, 60K.



Allocazione non contigua – idea - 2:

Assumiamo che la variabile **x** sia all'indirizzo logico **51448** del processo P della slide precedente:



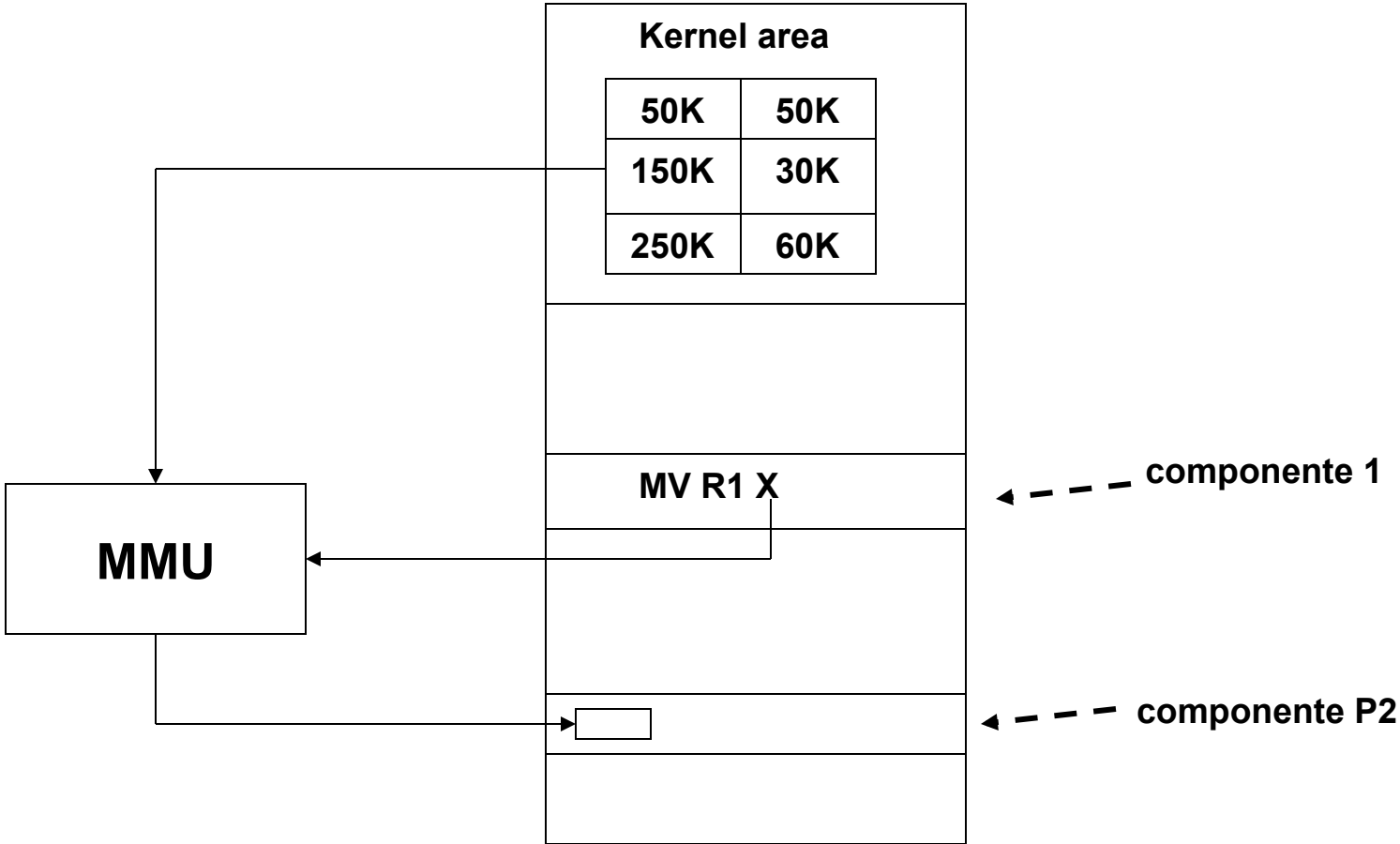
Se un'istruzione fa riferimento ad **x**, qual è l'indirizzo fisico che l'istruzione deve considerare?

- La componente **P1** ha dimensione **51200** byte (**50 K**).
- La componente **P2** ha dimensione **30 K**.
- L'indirizzo logico **51448** si trova pertanto nella componente **P2**, con offset **248** dall'indirizzo iniziale di **P2**, cioè **150 K** .

Allocazione non contigua – idea - 3:

Come tradurre l'indirizzo logico in indirizzo fisico?

Ci pensa la MMU, sfruttando le informazioni relative all'allocazione di **P** memorizzate dal S.O.:



Allocazione non contigua – idea - 4:

La MMU ha a disposizione una tabella in cui, per ogni componente, sono registrati l'indirizzo iniziale e la dimensione.

Partendo da un indirizzo logico, la MMU determina un indirizzo fisico, vale a dire:

- la componente
- l'offset rispetto all'indirizzo iniziale della componente.

Abbiamo due tecniche principali per l'allocazione non contigua:

- paginazione
- segmentazione

8.6- PAGINAZIONE e SEGMENTAZIONE

Paginazione - 1:

- La memoria è divisa in 2^f **page frame**, ciascuno di capacità 2^s byte, dove 2^{f+s} è la dimensione della memoria.

- Ognuno dei 2^{f+s} byte della memoria è individuato da un indirizzo che consiste in **f+s** bit:
  - gli **f** bit più significativi (leftmost) individuano il page frame
  - gli **s** bit meno significativi (rightmost) individuano l'**offset** nel page frame.

- Esempio con f=22 e s=10
  indirizzo **0000000000000000010100000110000**:
  page frame **10**, offset **48**.

Paginazione - 2:

- Ogni processo è logicamente diviso in **pagine** aventi le medesime dimensione dei page frame. Se il processo ha dimensione **d**, le pagine sono $\lfloor d/2^s \rfloor$.

- Ognuno dei **d** byte dello spazio del processo è individuato da un indirizzo che consiste in **l+s** bit:

  - gli **l** bit più significativi (leftmost) individuano la pagina

  - gli **s** bit meno significativi (rightmost) individuano l'offset nella pagina.

- Non è detto che **l <= f**.

Paginazione – 3:

- Allocare un processo in memoria significa associare un page frame ad ogni sua pagina. Non è richiesto che i page frame siano adiacenti, altrimenti la paginazione non avrebbe senso.

- Non esiste frammentazione, fermo restando che ad ogni processo di dimensione **d** vengono allocati $\mathbf{|d/2^s| - d}$ byte "di troppo", cioè l'ultimo page frame è usato solo in parte.

Paginazione – 4:

- Per ogni processo il S.O. mantiene una **page table**, che tiene traccia del page frame in cui è allocata ogni pagina.

  La page table contiene $|d/2^s|$ sequenze da **f** bit, dove la sequenza numero **n** individua il page frame assegnato alla pagina **n**, per $0 <= n <= |d/2^s|-1$.

  In pratica, il numero di pagina indicizza la page table.

- Dato un indirizzo logico di **l+s** bit, la MMU lo converte nell'indirizzo fisico di **f+s** bit determinato come segue:

  - il valore degli **f** bit leftmost si trova nella entry della page table di indice **l**

  - il valore degli **s** bit rightmost dell'indirizzo logico corrisponde al valore degli **s** bit rightmost dell'indirizzo fisico.

  Di fatto, la MMU fa una sostituzione di bit e non fa calcoli.

Paginazione – Esempio:

Siano **s = 10**, **f= 22**. (Le pagine hanno dimensione 1K).

Assumo un processo di dimensione 4000 byte.

Il processo ha 4 pagine. (96 byte "sprecati".)

Se vengono allocate nei page frame **3**, **20**, **5**, **9**, la page table per P ha la forma seguente:

00……000011
00……010100
00……000101
00……001001

Assumiamo che il processo abbia l'istruzione **MV R1 2100**. L'indirizzo **2100** (**2K+52**) corrisponde all'offset **52** della terza pagina, cioè la pagina numero **2**. La MMU accede alla terza riga della page table ed usa i 22 bit per formare l'indirizzo fisico, che sarà **5K + 52.**

Paginazione – 6:

- Il S.O. deve tener traccia dei page frame liberi per allocare i processi. Di norma vengono organizzati in una **free frame list**.

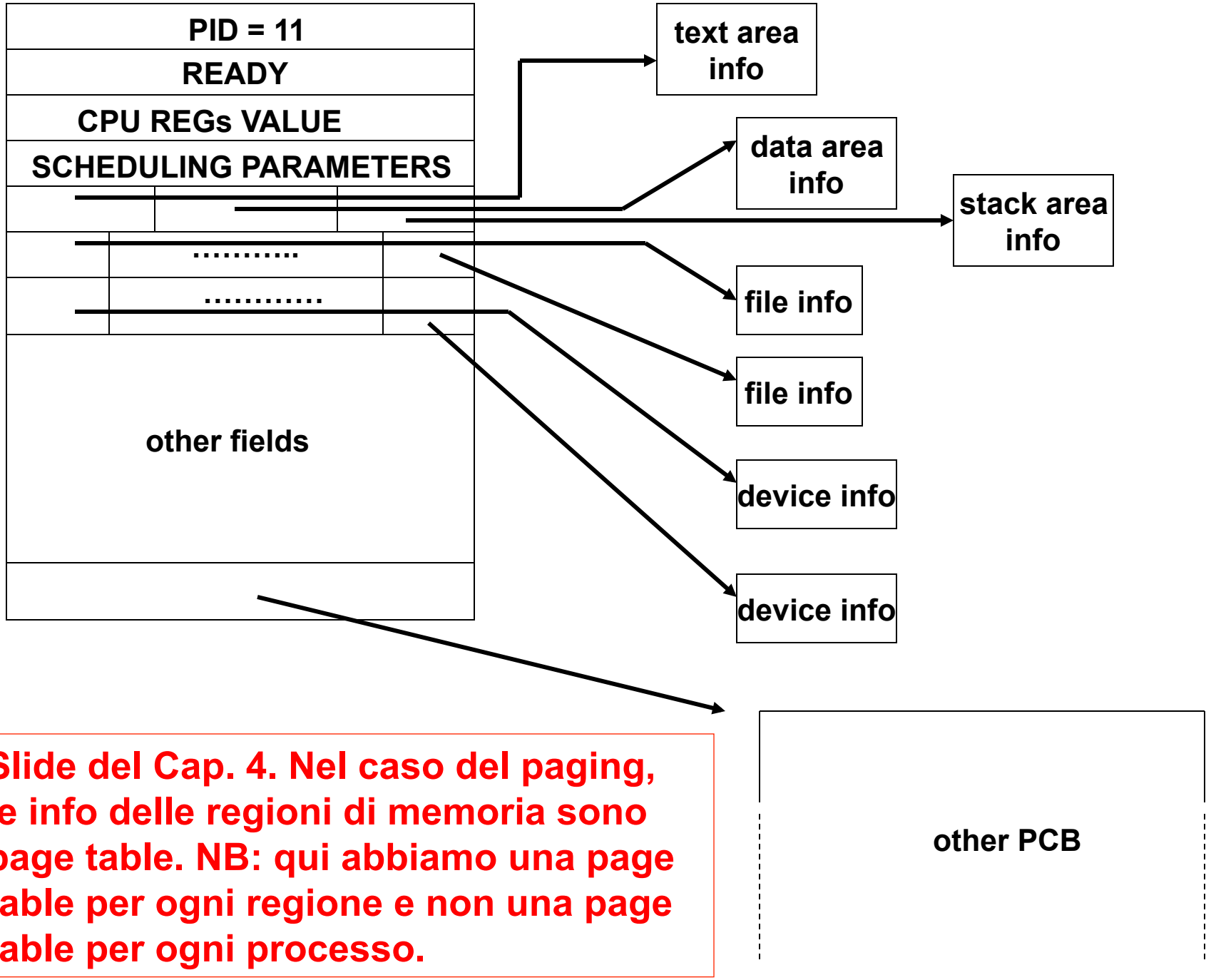
- La free frame list è una struttura del kernel cui i processi non possono accedere.

  "Perde pezzi" quando si alloca memoria ai processi, "riacquista pezzi" quando i processi liberano memoria.

- La page table di un processo è una struttura dati del kernel cui il processo non può accedere.

  - Se fosse nel PCB allora i PCB non potrebbero avere tutti la solita dimensione.

  - E' in un'area di memoria verso cui il PCB ha un pointer.



Slide del Cap. 4. Nel caso del paging, le info delle regioni di memoria sono page table. NB: qui abbiamo una page table per ogni regione e non una page table per ogni processo.

Segmentazione - 1:

- Ogni processo è logicamente diviso in **segmenti** che corrispondono ad entità logiche, quali, ad esempio, funzioni, strutture dati, oggetti.

- Ogni segmento di ogni processo ha la propria dimensione e, pertanto, NON può avere un corrispondente segment frame in memoria.

- Ognuno dei byte dello spazio del processo è individuato da un indirizzo composto da due dati:
  - il numero del segmento
  - l'offset nel segmento.

Segmentazione – 2:

- Allocare un processo in memoria significa associare un'area di memoria libera ad ogni suo segmento. Non è richiesto che tali aree siano adiacenti.

- E' possibile la frammentazione, perché i segmenti possono avere dimensione maggiore delle dimensioni delle aree di memoria libere.

  Però, la frammentazione in assenza di segmentazione è più probabile.

- La segmentazione è più adatta a facilitare la condivisione di moduli di codice, strutture dati, oggetti che a prevenire la frammentazione.

Segmentazione – 3:

- Per ogni processo il S.O. mantiene una **segment table**, che tiene traccia di due dati:

  - l'indirizzo dell'area di memoria cui è allocato il segmento

  - la dimensione del segmento.

- Dato un indirizzo logico che consiste del numero di segmento **n** e dell'offset **o**, la MMU lo converte nell'indirizzo fisico calcolato come segue:

  - l'indirizzo iniziale del segmento che si trova nella entry **n** della segment table viene sommato all'offset **o**.

  N.B.: nel caso del paging non veniva effettuata nessuna somma, ma una semplice sostituzione di bit. Ciò era più veloce.

Segmentazione – Esempio:

Assumo un processo con 3 segmenti: **main**, **search**, **update**, di dimensione **2K**, **3K**, **4K**.

Se vengono allocati agli indirizzi **100K**, **50K**, **80715**, la segment table avrà la forma seguente:

2K	**100K**
3K	**50K**
4K	**80715**

Assumiamo che il processo abbia l'istruzione **call update, f1**, dove **f1** è una funzione nel segmento **update** che si trova all'offset **52**.

La MMU accede alla terza riga della segment table ed aggiunge l'indirizzo iniziale del segmento, cioè **80715**, a **52**, dopo aver controllato che **52** sia minore della dimensione del segmento, cioè **4K.**

Segmentazione - 5:

- Non ha senso avere una segment frame list, non essendoci segment frame.

- Il kernel deve tener traccia di tutte le aree libere e della loro dimensione, usando una free area list.

- Esattamente come la page table, la segment table di un processo è una struttura dati del kernel cui il processo non può accedere.

  - Se fosse nel PCB allora i PCB non potrebbero avere tutti la solita dimensione.

  - E' in un'area di memoria verso cui il PCB ha un pointer.

Segmentazione + paginazione:

- I segmenti sono entità logiche divise in pagine che hanno la stessa dimensione dei page frame.

- Ogni entry della segment table contiene un pointer alla page table del segmento.

- Si hanno i vantaggi di entrambe le tecniche.

8.7 – MEMORIA VIRTUALE

- Intuizione: parliamo di **memoria virtuale** quando il S.O. crea l'illusione che il sistema abbia più memoria di quella effettiva.

- L'obiettivo è di rendere i processi indipendenti dalla capacità di memoria del sistema.

- L'implementazione si basa sull'uso del disco.

- Formalmente: la memoria virtuale è la gerarchia di memoria che consiste nella memoria e nel disco e che consente ai processi di operare quando in memoria è presente solo una porzione del loro spazio indirizzabile.

- La memoria virtuale si basa sulla paginazione (demand paging) e/o segmentazione.

Demand paging – idea (1):

- la memoria è divisa in page frame ed i processi sono costituiti da pagine come spiegato nella Sezione 7.6;
- l'intero spazio logico dei processi è memorizzato su un **paging device**, cioè un disco oppure una porzione di disco;
- l'area del paging device allocata per un processo viene detta **swap space** del processo;
- quando un processo inizia l'esecuzione, viene allocato solo un page frame per la pagina che contiene la prima istruzione;

Demand paging – idea (2):

- quando il processo tenta di accedere ad una pagina cui non è assegnato nessun page frame, la pagina viene copiata dallo swap space in un page frame libero;

- quando un processo modifica una pagina, l'immagine della pagina sullo swap space è obsoleta;

- se servono page frame, il S.O. può liberare i page frame associati alle pagine dei processi, in tal caso le immagini di tali pagine sullo swap space vanno aggiornate.

Il demand paging si basa su tre concetti fondamentali:

- **page fault**: eccezione che viene sollevata dalla MMU quando un processo tenta di accedere ad una pagina che non è associata a nessun page frame;

- **page-in**: in seguito al page fault, il S.O. (page fault handler) copia la pagina dallo swap space in un page frame libero.

  Se non esistono page frame liberi, il S.O. deve liberarne uno;

- **page-out**: quando il S.O. libera un page frame, se la pagina associata è stata modificata dopo l'ultimo page-in, deve essere copiata sullo swap space;

- **page replacement**: consiste nel liberare un page frame e nel successivo page-in di una pagina diversa.

Osservazioni:

- il page fault è un esempio di evoluzione dell'hardware dettata dallo sviluppo dei S.O.;

- i page-in e page-out costituiscono il **page I/O**, che è distinto dal program I/O in quanto è trasparente al programma, cioè non è controllabile dal programmatore;

- i processi che generano page fault vengono messi in waiting, pertanto vedono degradare la loro velocità di avanzamento;

- i page I/O possono dar luogo a letture/scritture su disco, che daranno luogo ad I/O interrupt che degraderanno la velocità di avanzamento di altri processi.

Ogni entry della page table deve contenere i seguenti campi:

- **validity bit**: vale 1 se la pagina è in un page frame, 0 altrimenti;
- **frame #**: numero del page frame associato alla pagina, significativo solo se **validity bit** = 1;
- **prot** (protection info): codifica dei permessi per accedere alla pagina in lettura/scrittura;
- **ref** (reference info): campo che codifica l'informazione circa i recenti accessi alla pagina, utile per scegliere quali pagine devono subire lo swap out;
- **mod bit** (modified bit): vale 1 se la pagina è stata modificata dopo l'ultimo page-in e, pertanto, richiederà il page-out quando il frame verrà liberato, 0 altrimenti;
- **disk address**: indirizzo della pagina nello swap device.

Traduzione da parte della MMU di un indirizzo logico di **l+s** bit, dove gli **l** bit leftomost costituiscono il page # e gli **s** rightmost bit l'offset:

1. gli **l** leftmost bit determinano la entry della page table da utilizzare, come già visto nella Sezione 8.6;

2. se **validity bit** = **0** viene sollevato un page fault:

   - il page fault handler usa il valore del campo **disk address** per determinare il blocco dello swap device da leggere (il valore viene passato con le modalità viste nel Cap. 3);

   - il page fault handler deve accedere alla free frame list per assegnare un frame libero alla pagina;

   - dopo aver copiato la pagina dallo swap device nel page frame, il page fault handler deve settare il **validity bit** a 1, deve settare il **frame #** ed andare al punto 3;

3. se **validity bit** = **1** l'indirizzo fisico viene determinato concatenando gli **f** bit del **frame #** con gli **s** bit rightmost dell'indirizzo logico.

Traduzione con uso del TLB (1):

- La traduzione appena descritta prevede due cicli di memoria per ogni accesso alla memoria:
  - il primo ciclo per accedere alla page table;
  - il secondo ciclo per l'accesso vero e proprio.

- Per risparmiare uno dei due cicli, la MMU può usare un **Translation Look-aside Buffer (TLB)**, cioè una **memoria associativa** contenente coppie (page #, frame #).

- Le memorie associative sono costose: il TLB contiene la coppia (page #, frame #) solo per alcune pagine.

Traduzione con uso del TLB (2):

Dato l'indirizzo logico page #, offset, la MMU accede al TLB:

1. se nel TLB c'è la coppia (page #, frame #), la MMU costruisce l'indirizzo fisico frame #, offset;

2. se non c'è nessuna coppia (page #, frame #) per il page # considerato, si parla di **TLB miss** e la MMU usa la page table in memoria come descritto in precedenza.

   In questo caso, la coppia (page #, frame #) viene aggiunta al TLB, eliminando eventualmente una coppia esistente.

   Non precisiamo:

   - i criteri possibili per la scelta della coppia da eliminare dal TLB;

   - gli eventuali supporti hardware necessari ad implementare tali criteri.

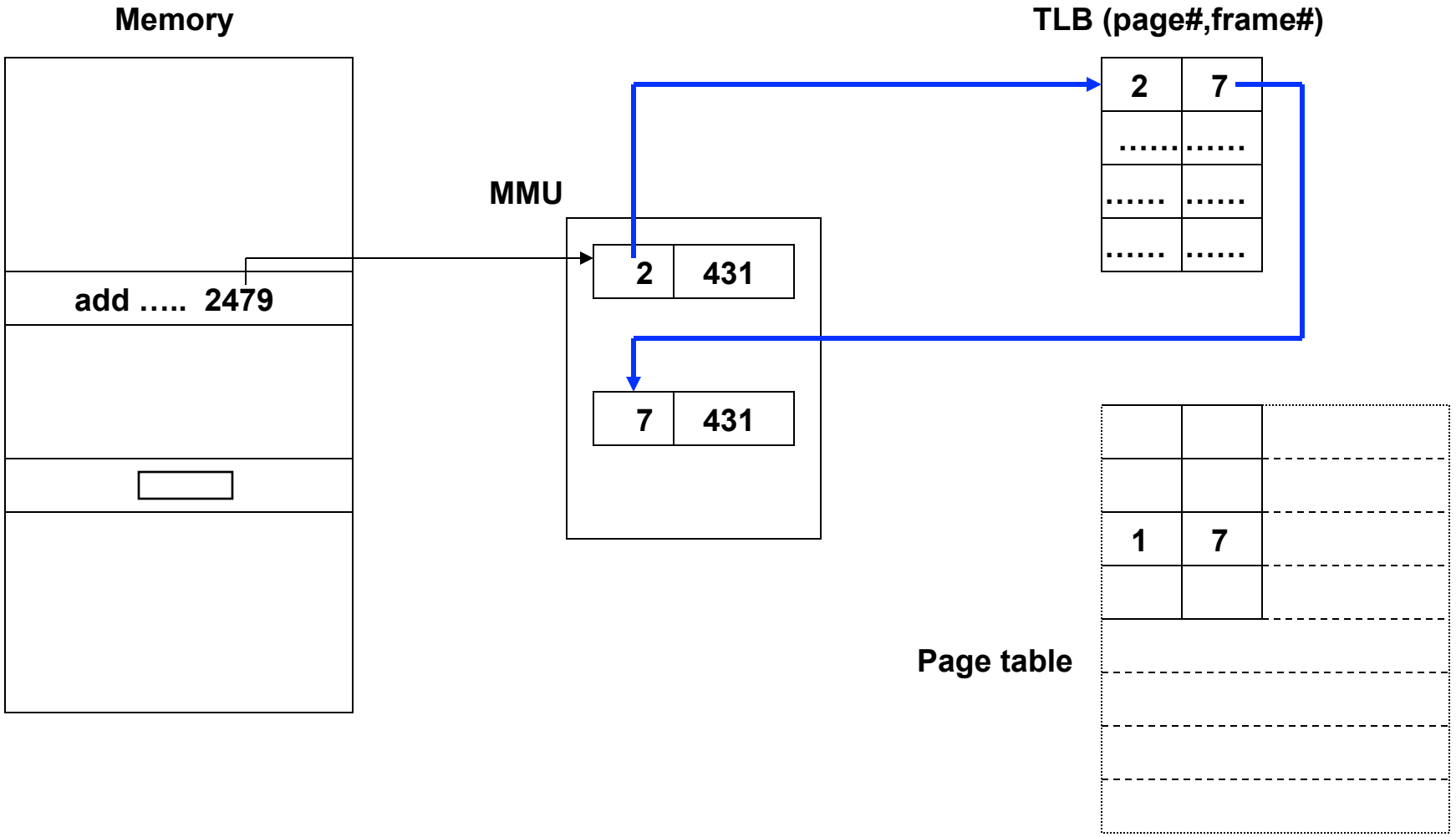
Riassumendo, la MMU genera l'indirizzo fisico a partire dall'indirizzo logico nel modo seguente:

1. usando la entry del TLB dedicata alla pagina, se esiste;

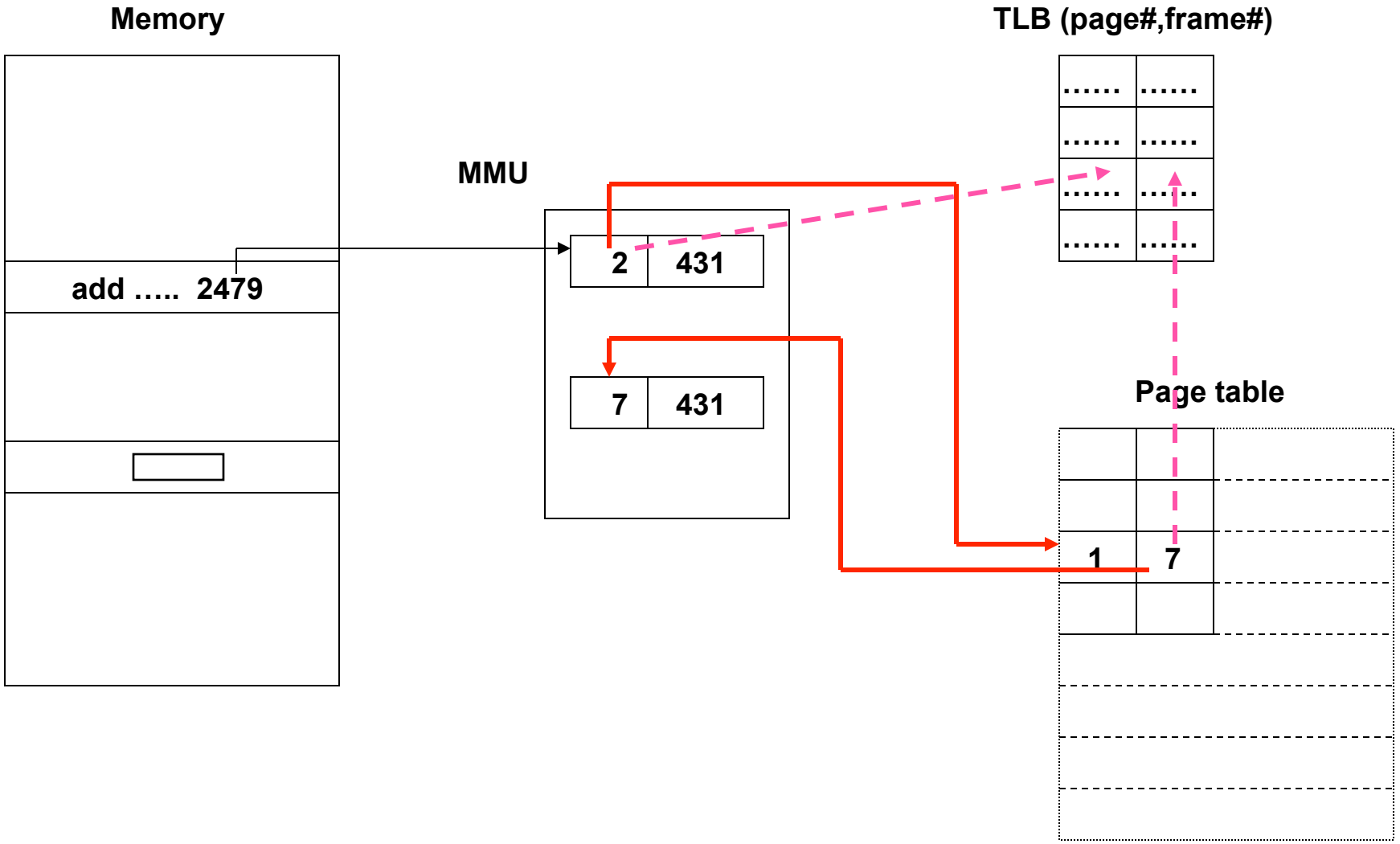
2. sfruttando la entry relativa alla pagina sulla page table, se tale entry ha validity bit 1 e se il caso 1) non è applicabile;

3. generando un page fault se i casi 1) e 2) non sono applicabili.

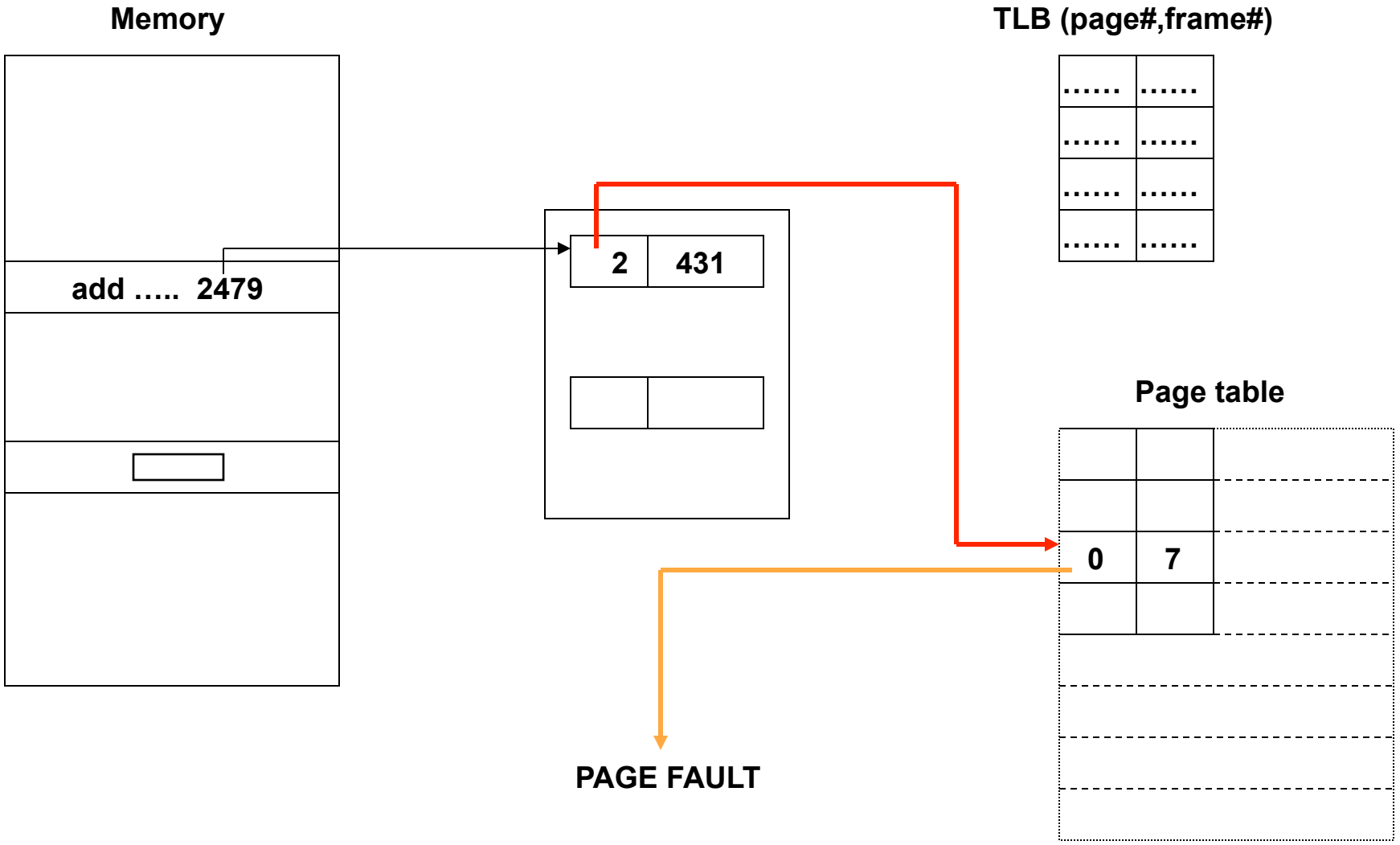
Vediamo un esempio:



Assumiamo che un processo acceda all'indirizzo 2479: page 2, offset 431. Questa slide mostra il lavoro della MMU nel caso in cui la coppia (2,frame#) sia presente nel TLB. In particolare, frame# vale 7, quindi gli l bit più significativi dell'indirizzo fisico assumono il valore 7, ed i 10 bit meno significativi mantengono il valore 431.



In questo caso il TLB non ha la coppia (2,frame#), pertanto la MMU deve accedere alla page table, dove la entry numero 2 ha validity bit = 1 e frame# = 7. Anche in questo caso gli l bit più significativi assumono il valore 7. La coppia (2,7) viene inserita in una entry del TLB, come indicato dalle frecce tratteggiate.



In questo caso il TLB non ha la coppia (2,frame#), pertanto la MMU deve accedere alla page table, dove, però, la entry numero 2 ha validity bit = 0. La MMU genera un page fault.

Un'osservazione relativa al TLB:

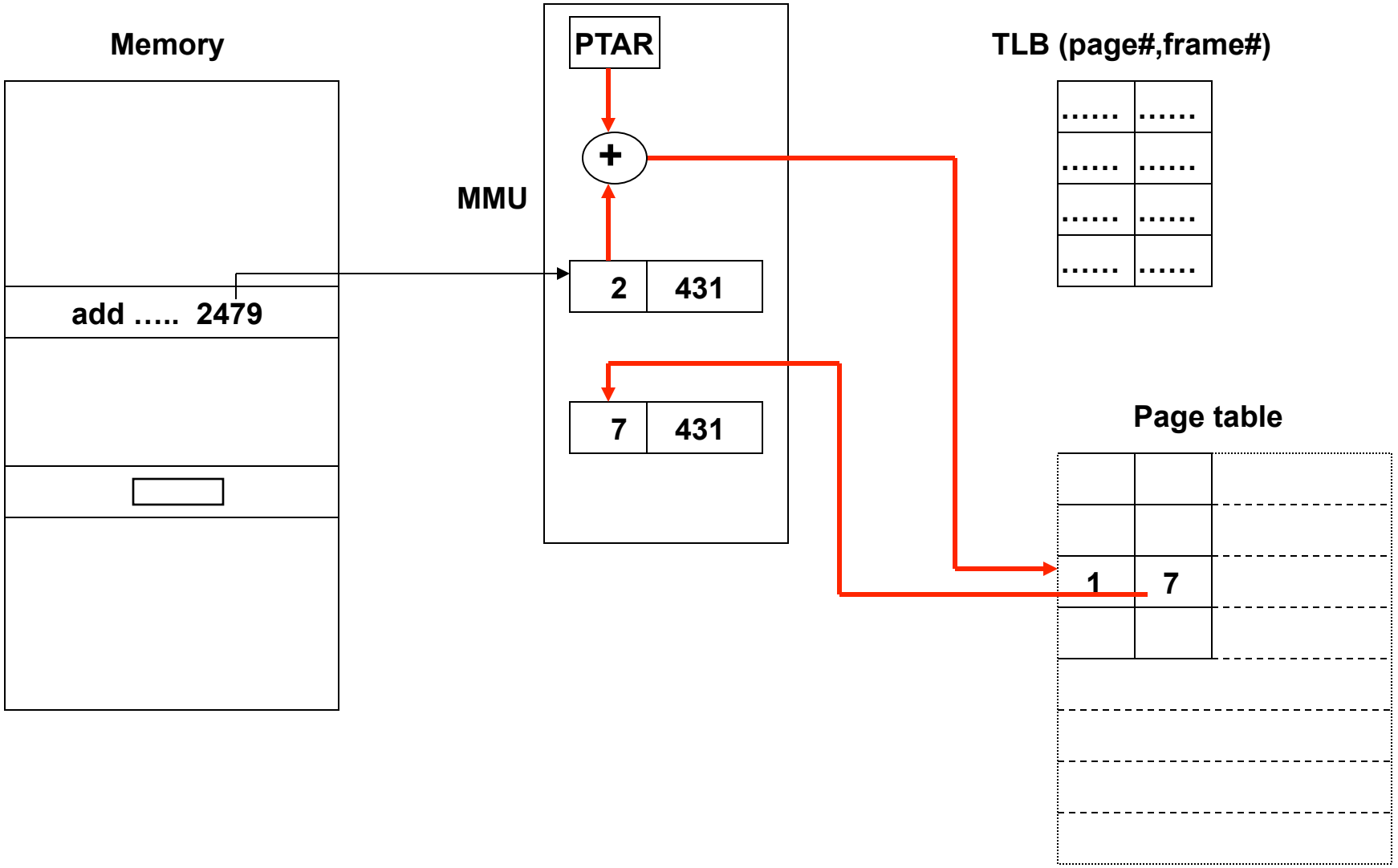
- quando avviene un context switch, il TLB va azzerato per evitare che il processo schedulato acceda alla memoria del precesso che ha perso il processore;

- il processo schedulato avrà il TLB "vuoto": questo è un elemento che introduce overhead.

In realtà siamo stati un po' imprecisi:

- abbiamo ragionato come se la MMU accedesse ad una generica page table, ma, in realtà, la MMU accede alla page table del processo running;
- in memoria sono presenti le page table di tutti i processi;
- per implementare facilmente questo aspetto l'architettura può offrire il **Page Table Address Register (PTAR)**, contenente l'indirizzo della page table del processo running;
- il valore da assegnare al PTAR è memorizzato nel PCB del processo;

Raffiniamo l'esempio di Pag. 75 nella prossima slide:



L'indirizzo della entry della page table va sommato all'indirizzo iniziale della page table. Per essere precisi, la MMU accede al seguente indirizzo:

valore del PTAR + dimensione delle entry della page table * numero pagina

Protezione della memoria - 1:

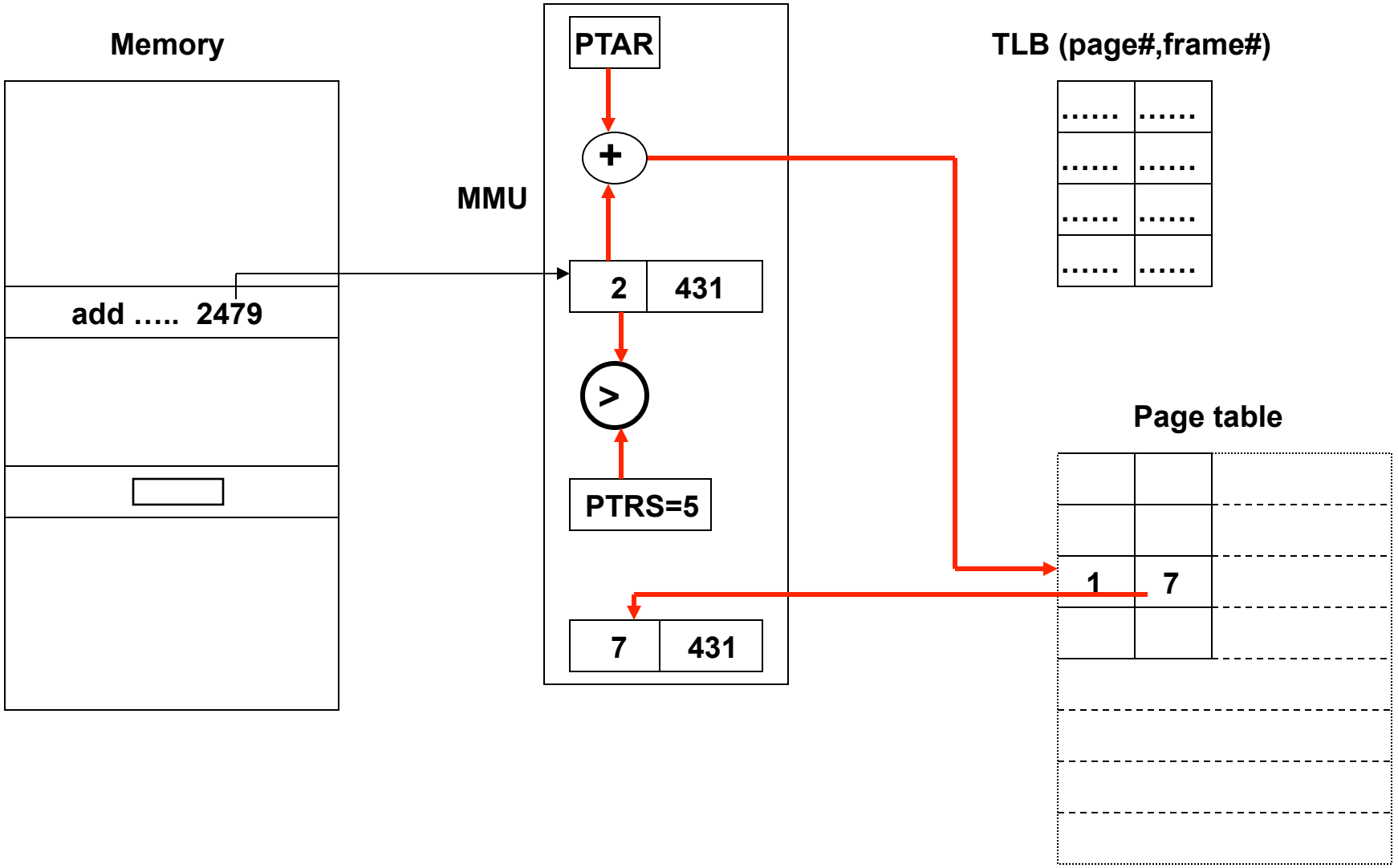
- se un processo tenta di accedere ad un indirizzo al di fuori del proprio spazio logico, viene generata un **memory protection exception**.

- questo si verifica quando gli **l** bit leftmost dell'indirizzo logico hanno un valore **k** e la pagina **k** non esiste, in quanto il processo ha pagine **0,1,…,h** con **h<k**;

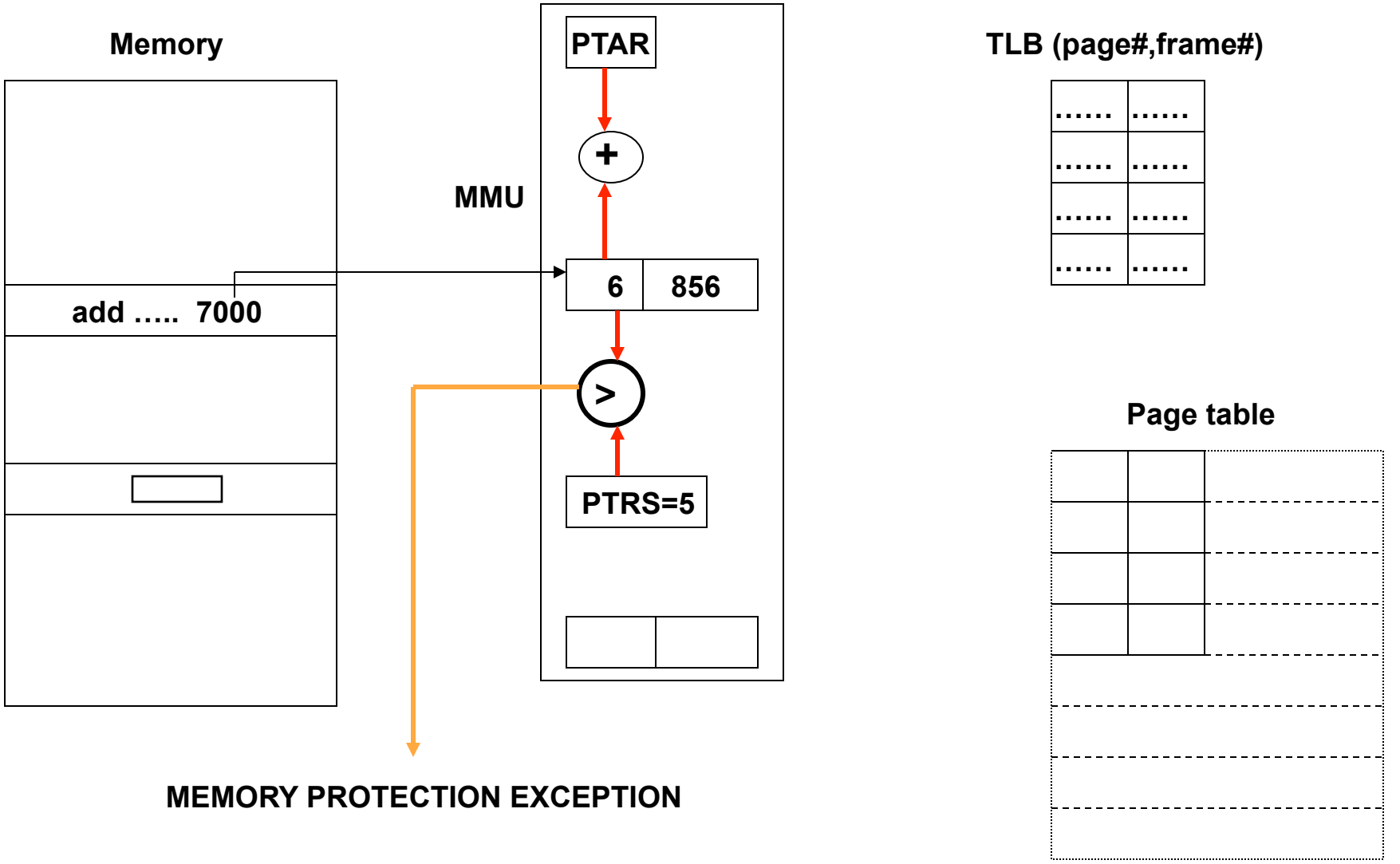
- il controllo può essere fatto facilmente se l'architettura offre un **Page Table Size Register (PTSR)** che memorizza il numero dell'ultima pagina del processo running.

- il PTSR deve essere memorizzato nel PCB del processo.

Raffiniamo l'esempio di Pag. 79 nelle prossime due slide:



Il numero di pagina è <= al valore del PTRS: nessuna violazione.



Il numero di pagina è maggiore del PTRS: la pagina non esiste, viene sollevata l'eccezione.

Protezione della memoria - 2:

- il campo **prot** della page table codifica le protezioni in lettura/ scrittura delle pagine;

- se l'architettura fornisce un adeguato supporto, il valore del campo può essere confrontato con il tipo di accesso che viene effettuato;

- se il tipo di accesso non è consentito, la MMU genera una **memory protection exception**.

Principio di località: un indirizzo logico generato eseguendo un'istruzione ha probabilità elevata di essere vicino agli indirizzi logici generati eseguendo le istruzioni più recenti.

Perché il principio è veritiero:

- le istruzioni di branch sono solitamente non più del 10-20% del totale, quindi i processi eseguono prevalentemente istruzioni memorizzate l'una adiacente all'altra;
- gli accessi alle strutture dati sono spesso effettuati in istruzioni ravvicinate.

In base al principio di località, per minimizzare i page fault, quando è necessario effettuare lo swap out di una pagina conviene scegliere tra le pagine che sono state visitate meno di recente.

Osservazione 1:

aumentando i page frame allocati ad ogni processo:

(+) diminuisce la probabilità di avere page fault (+ efficienza)

(-) si hanno meno processi in memoria (- efficienza)

Va trovato un equilibrio.

Se il numero di page frame allocati ad un processo è troppo basso:

- i page fault sono frequenti;
- cresce il page I/O;
- cresce il numero di context switch.

Si parla di **trashing**.

Osservazione 2: diminuendo la size **s** delle pagine:

(+) la memoria sprecata, mediamente **s/2** byte, diminuisce

(+) il numero di page fault diminuisce, a parità di memoria allocata al processo. Infatti è più probabile generare un indirizzo che si trovi in una pagina allocata in un page frame

(-) il numero di entry nelle page table è maggiore

(-) la dimensione del **frame #** è maggiore

(-) lo swap device perde efficienza se usa blocchi di dimensione ridotta

Anche qui va trovato un equilibrio.

Come anticipato a Pag. 69, la page table ha due campi che servono per il page replacement:

- **mod** - si tratta di un semplice bit:
  - impostato a 0 al momento del page in;
  - aggiornato ad 1 quando il page frame viene modificato.

  Quando il page frame viene liberato, se il bit **mod** vale 1 allora è necessario il page out.

- **ref** - si tratta di un semplice bit:
  - impostato ad 1 al momento del page in;
  - aggiornato a 0 ad intervalli regolari;
  - aggiornato ad 1 quando la pagina viene visitata.

  Se **ref** vale 1 allora il page frame è stato visitato di recente, ed in base al principio di località non è un buon candidato ad essere liberato se servono page frame liberi.

8.8: DETTAGLI IMPLEMENTATIVI DEL PAGING

Superpagine.

Chiamiamo **TLB reach** il prodotto page size * TLB entries.

In pratica, è "la memoria coperta dal TLB".

La dimensione delle RAM cresce rapidamente, la dimensione delle memorie associative no: il rapporto TLB reach / capacità RAM tende a diminuire, il numero di TLB miss tende ad aumentare, con due conseguenze:

- gli accessi alla memoria che sfruttano il TLB sono pochi
- gli accessi alla cache sono rallentati dai TLB miss e dalla necessità di accedere alla page table in memoria.

Soluzione possibile: uso delle **superpagine**.

- Una superpagina è come una pagina, però ha size pari a $\mathbf{2^n}$ * size delle pagine, per un **n** opportuno.

- L'architettura offre superpagine di size $\mathbf{2^n}$ * size delle pagine per alcuni **n**.

- Più pagine contigue con accessi frequenti vengono promosse a superpagina dal S.O.. Parliamo di **promotion**.

- Una TLB entry può riferisi ad una pagina o ad una superpagina. Se una TLB entry si riferisce ad una superpagina, il TLB reach si alza.

- Se alcune pagine della superpagina non hanno più accessi, la superpagina viene smembrata. Parliamo di **demotion**.

- Non indaghiamo ulteriormente ……

Organizzazione delle page table.

Se i processi hanno page table di dimensioni ampie, una porzione significativa di memoria viene usata per gestire la memoria virtuale e non per mantenere lo spazio indirizzabile dei processi.

Per risparmiare memoria esistono due strategie:

- uso delle **Inverted Page Table (IPT)**;
- **doppia paginazione**.

L'IPT ha una entry per ogni frame, con due campi:

- l'indicazione che il frame è libero, oppure
- la coppia (process id, page#), se il frame è occupato.

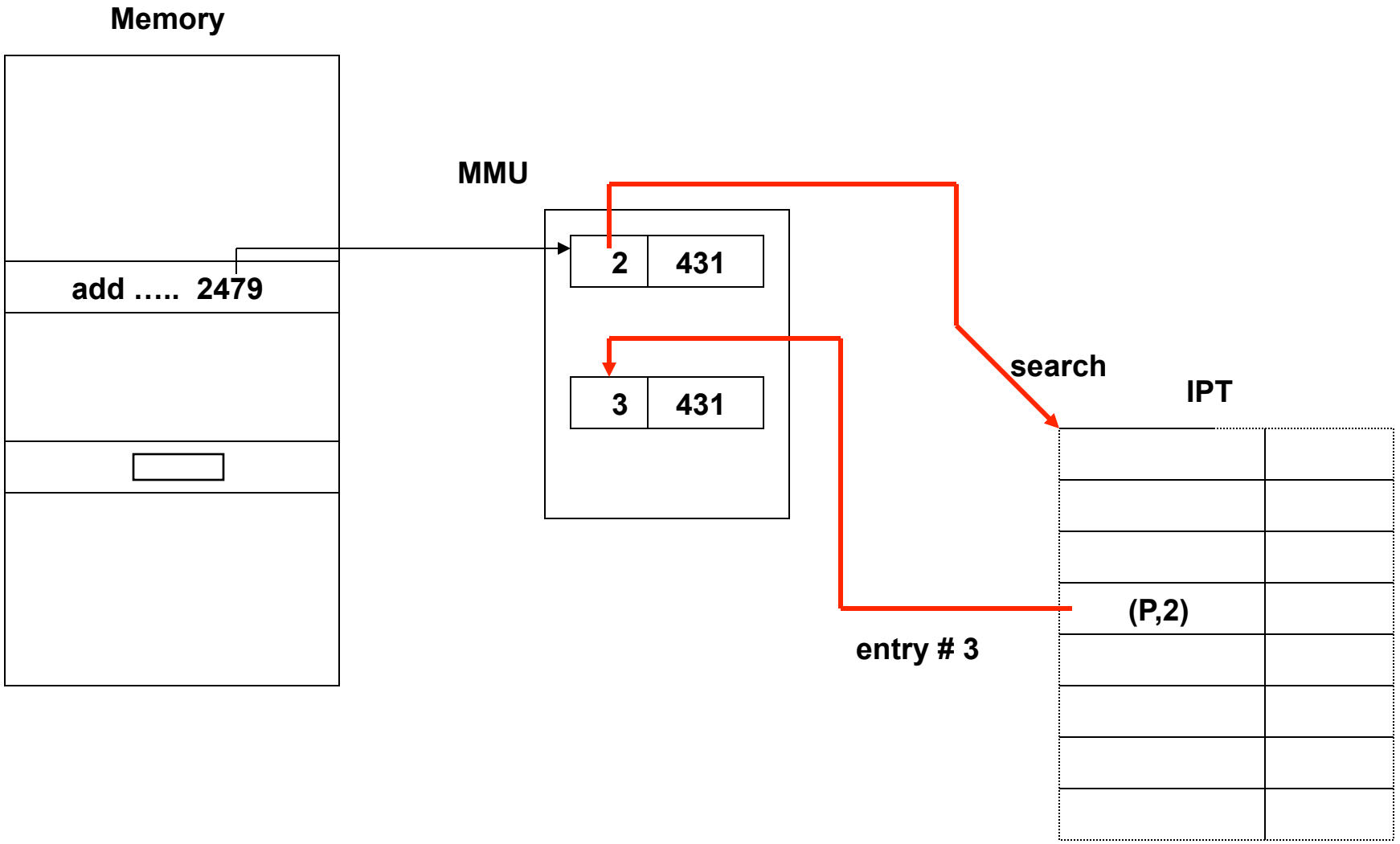
Vantaggio: la size dell'IPT dipende dall'architettura (size della memoria e delle pagine) e non dal numero di processi e dal loro numero di pagine.

Svantaggio: La MMU deve cercare nell'IPT la entry che contiene la coppia (process id, page#) per poter sostituire il page# con il frame# e costruire l'indirizzo fisico.

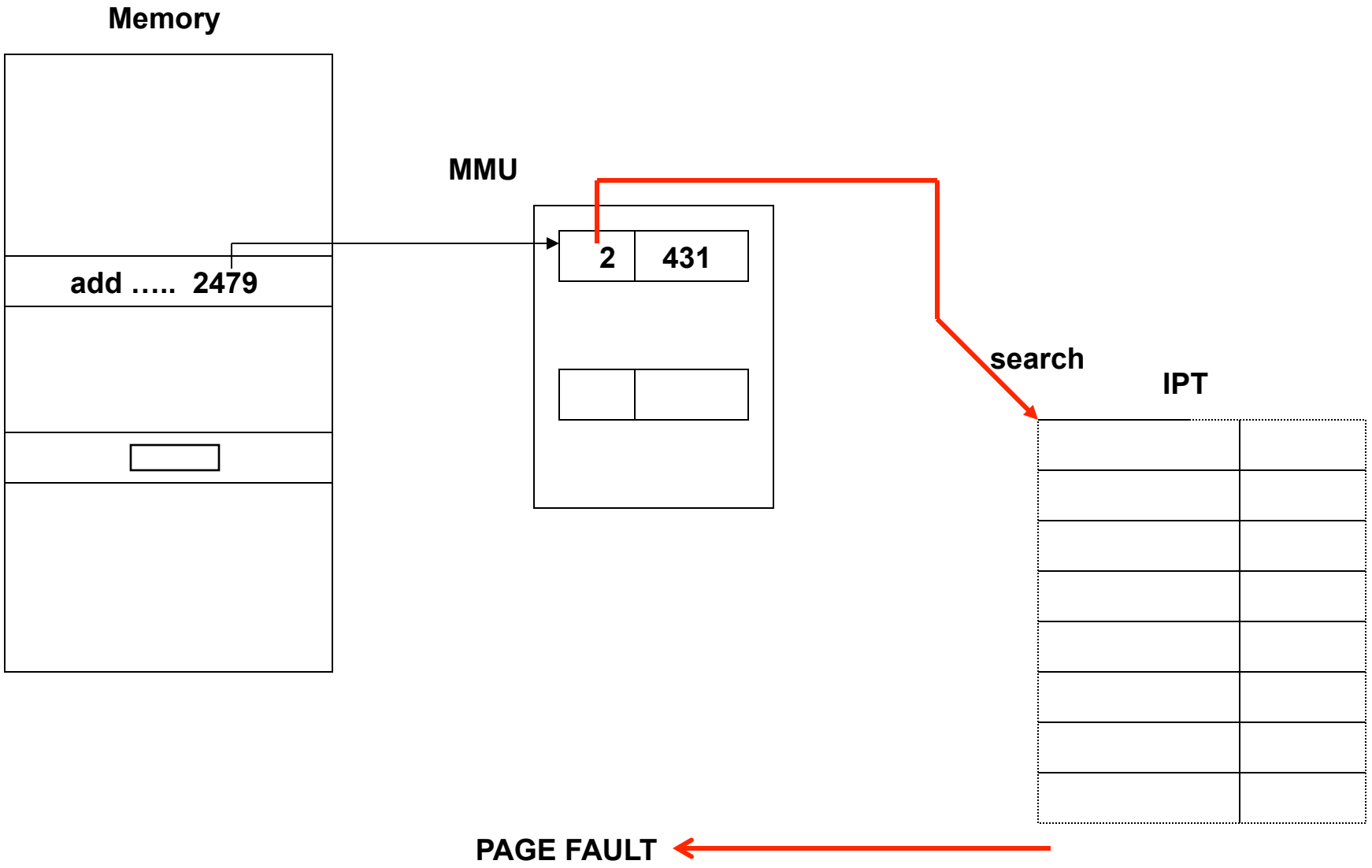
La ricerca è costosa.

NB: il frame# coincide con il numero della entry.

Due esempi nelle prossime due slide.



Assumiamo che il processo P sia in esecuzione. La MMU deve cercare la coppia (P,2). La coppia si trova nella entry 3: il frame associato alla coppia (P,2) è il frame numero 3.



Assumiamo che il processo P sia in esecuzione. La MMU deve cercare la coppia (P,2). La coppia non si trova nell'IPT: abbiamo un page fault.

Per evitare la ricerca sull'IPT, si può usare una **funzione hash**:

- ogni entry dell'IPT ha almeno 4 campi:
- - stato libero/occupato
  - coppia (process id, page #);
  - frame #, che non è un indice, come invece accadeva nel caso precedente;
  - pointer, per costruire una linked list.

  Possono esserci ulteriori campi, che qui non precisiamo;

- si assume un numero primo **a** maggiore del numero di frame;

- si assume una **hash table** contenente **a** pointers verso l'IPT;

- si considera la funzione hash **h: x → x MOD a**

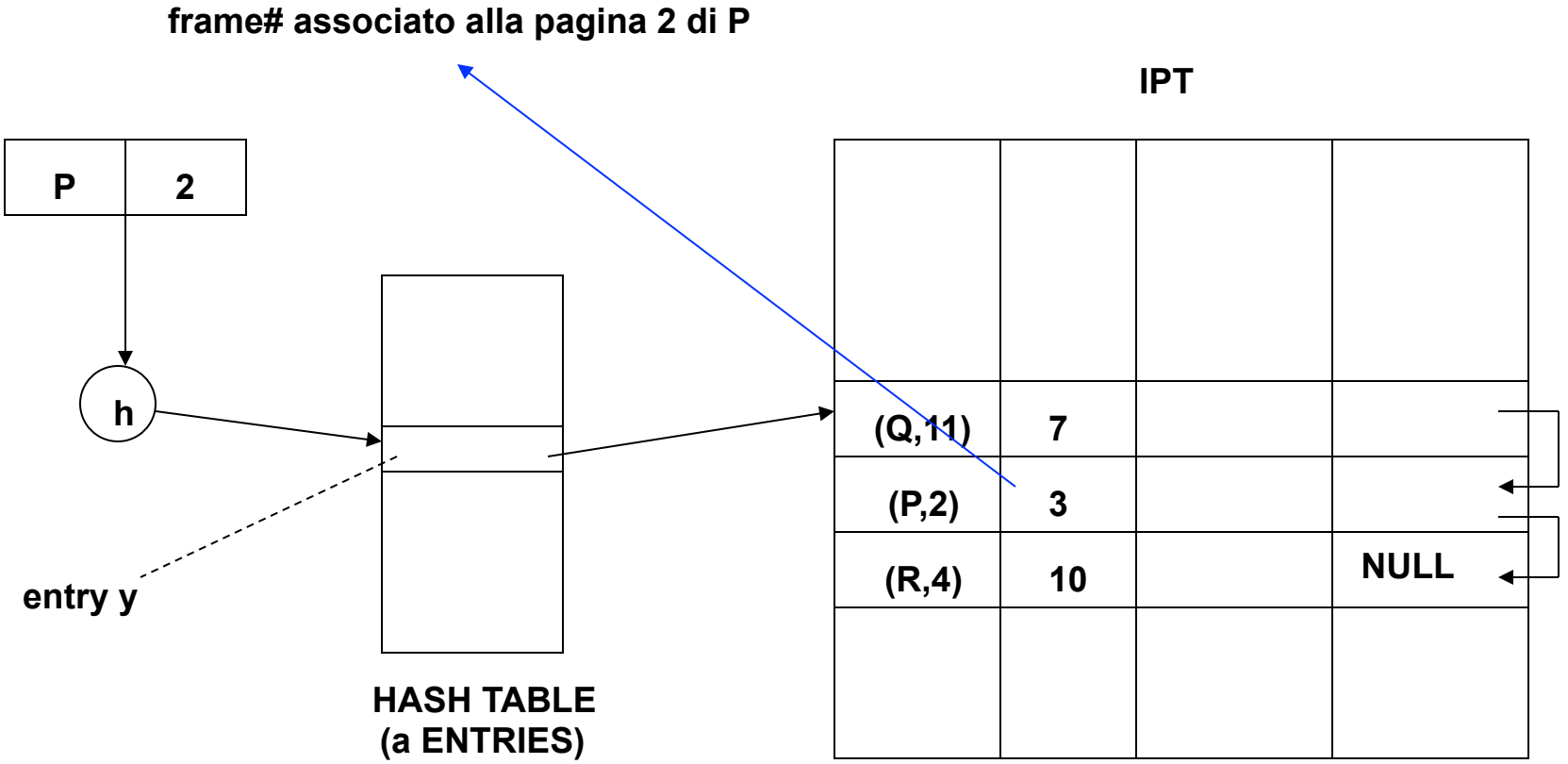
  NB. se **h(x) = y** allora **0<=y<=a-1**, ed esiste la entry numero **y** nella hash table;

- concatenando un process id **P** ed un page # **p** otteniamo una stringa di bit interpretabile come intero **x** che può essere usato come argomento della funzione **h**;

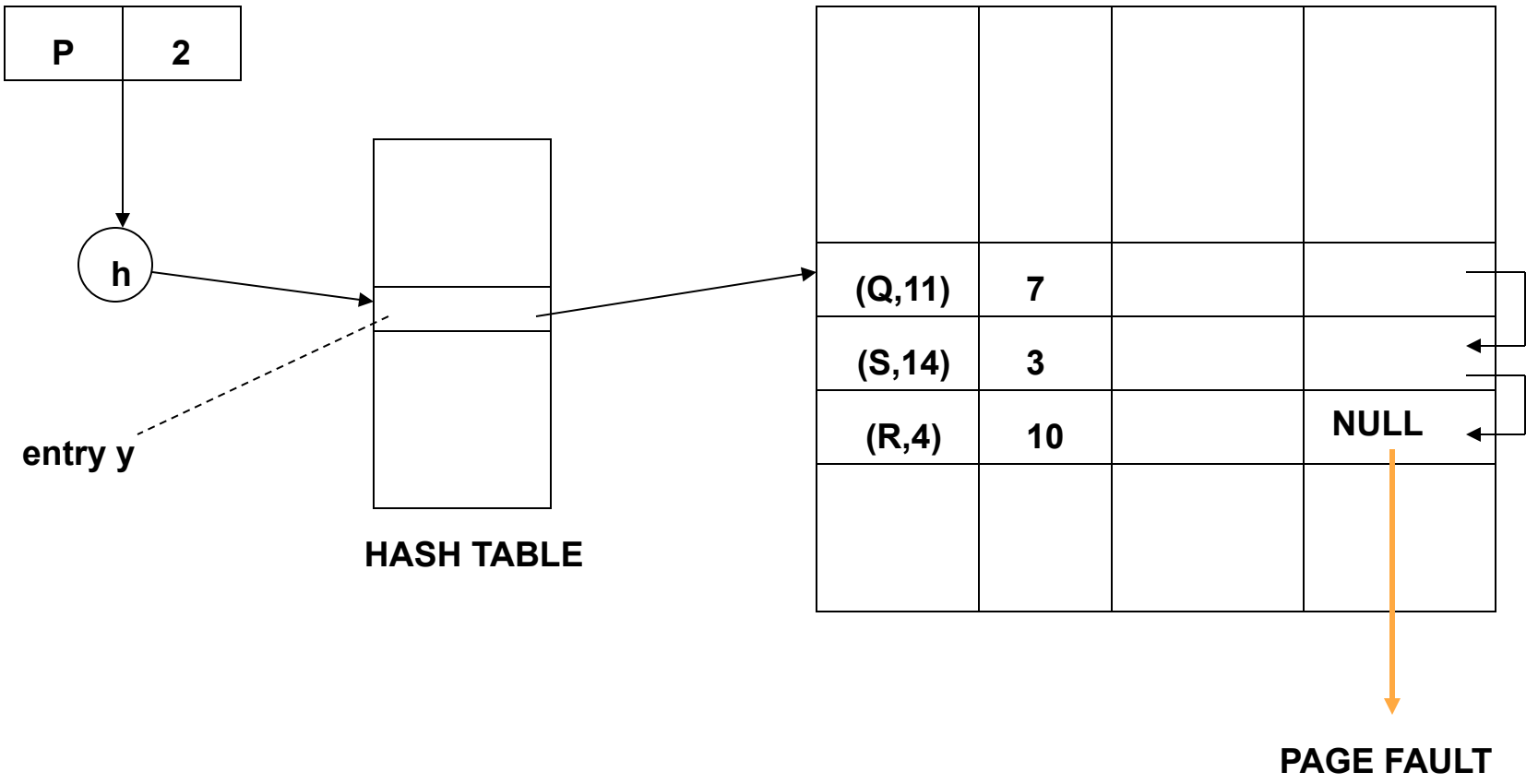
- per ogni **0<=y<=a-1,** le coppie **(Q,q)** che appaiono nell'IPT e tali che **h(Qq)=y** sono concatenate in una linked costruita con i pointers dell'IPT;

- per ogni **0<=y<=a-1**, la entry **y** della **hash table** punta alla prima entry dell'IPT contenente una coppia **(Q,q)** tale che **h(Qq)=y**.

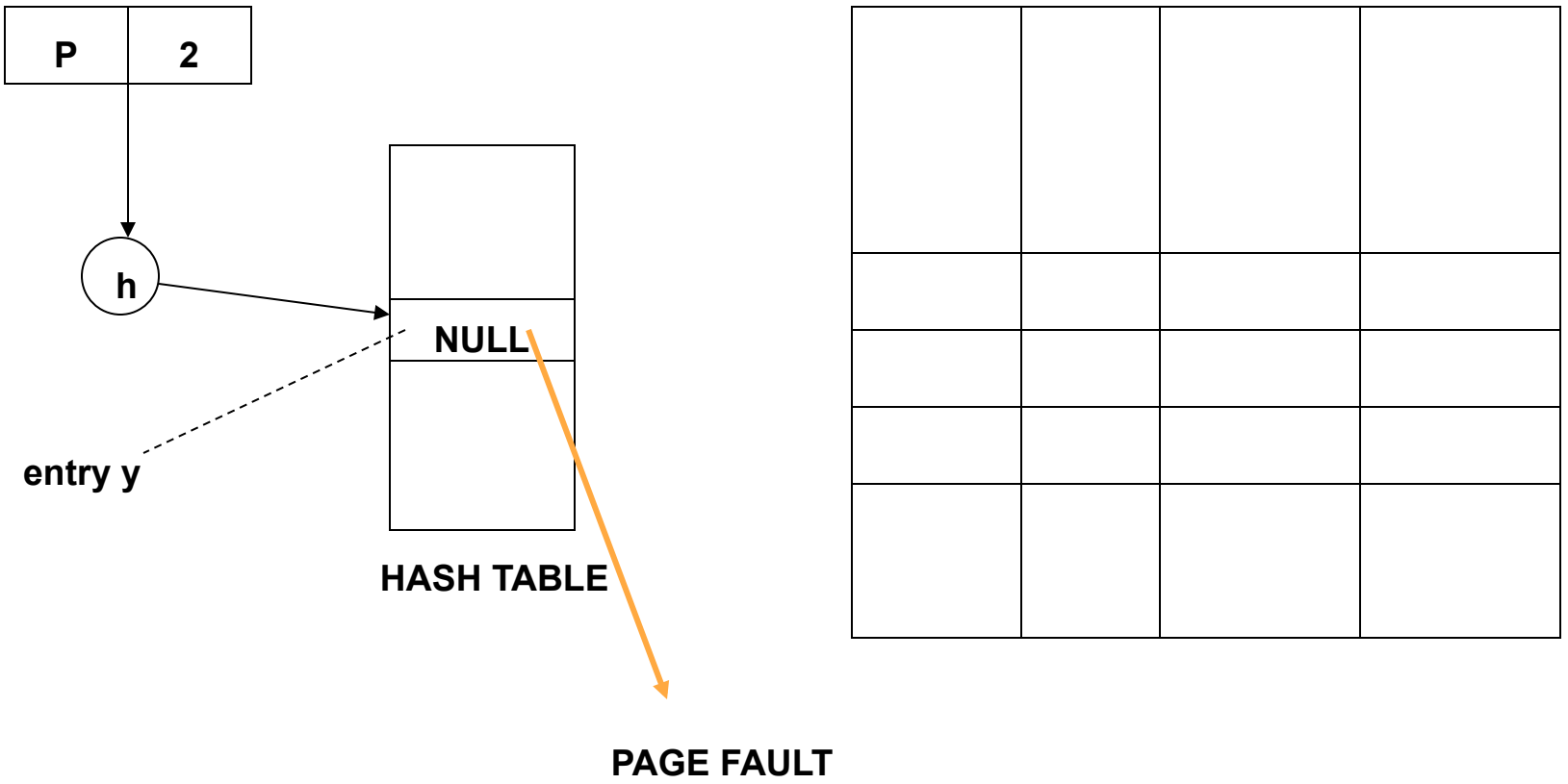
Vediamo nelle seguenti 3 slide come sfruttare la hash table.



La funzione h applicata alle coppie (P,2), (Q,11) e (R,4) da lo stesso risultato, un valore 0<=y<=a-1. La entry y della hash table ha un pointer ad una linked list nell'IPT, dove la MMU cerca la entry (P,2). Abbiamo comunque una ricerca, ma su una porzione limitata dell'IPT.



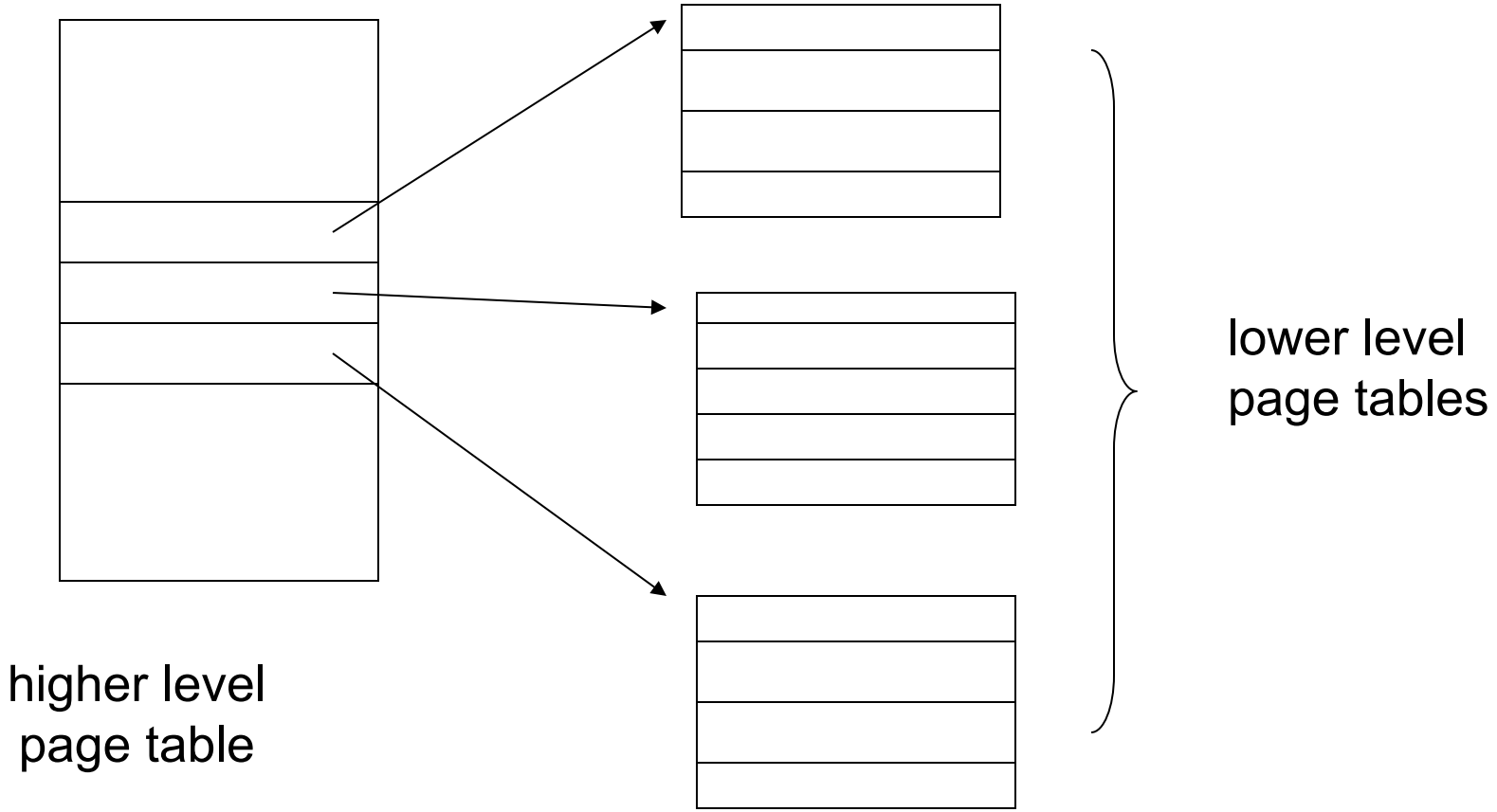
La funzione h applicata alle coppie (P,2), (Q,11), (S,14) e (R,4) da lo stesso risultato, un valore 0<=y<=a-1. La entry y della hash table ha un pointer ad una linked list nell'IPT, dove la MMU cerca la entry (P,2). Non la trova: page fault



La funzione h applicata alla coppia (P,2) da come risultato un valore 0<=y<=a-1.
La entry y della hash table contiene il pointer NULL: page fault

Doppia paginazione.

Per evitare di avere le page table in memoria, le page table vengono a loro volta paginate:

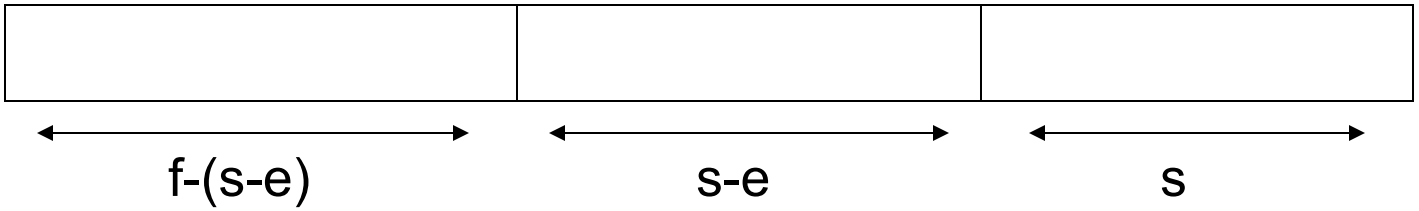


Assumiamo di avere 2^f frame di size 2^s (indirizzi di **f+s** bit);

Assumiamo, per semplificare la presentazione e nel rispetto della pratica, che una page table entry occupi 2^e byte.

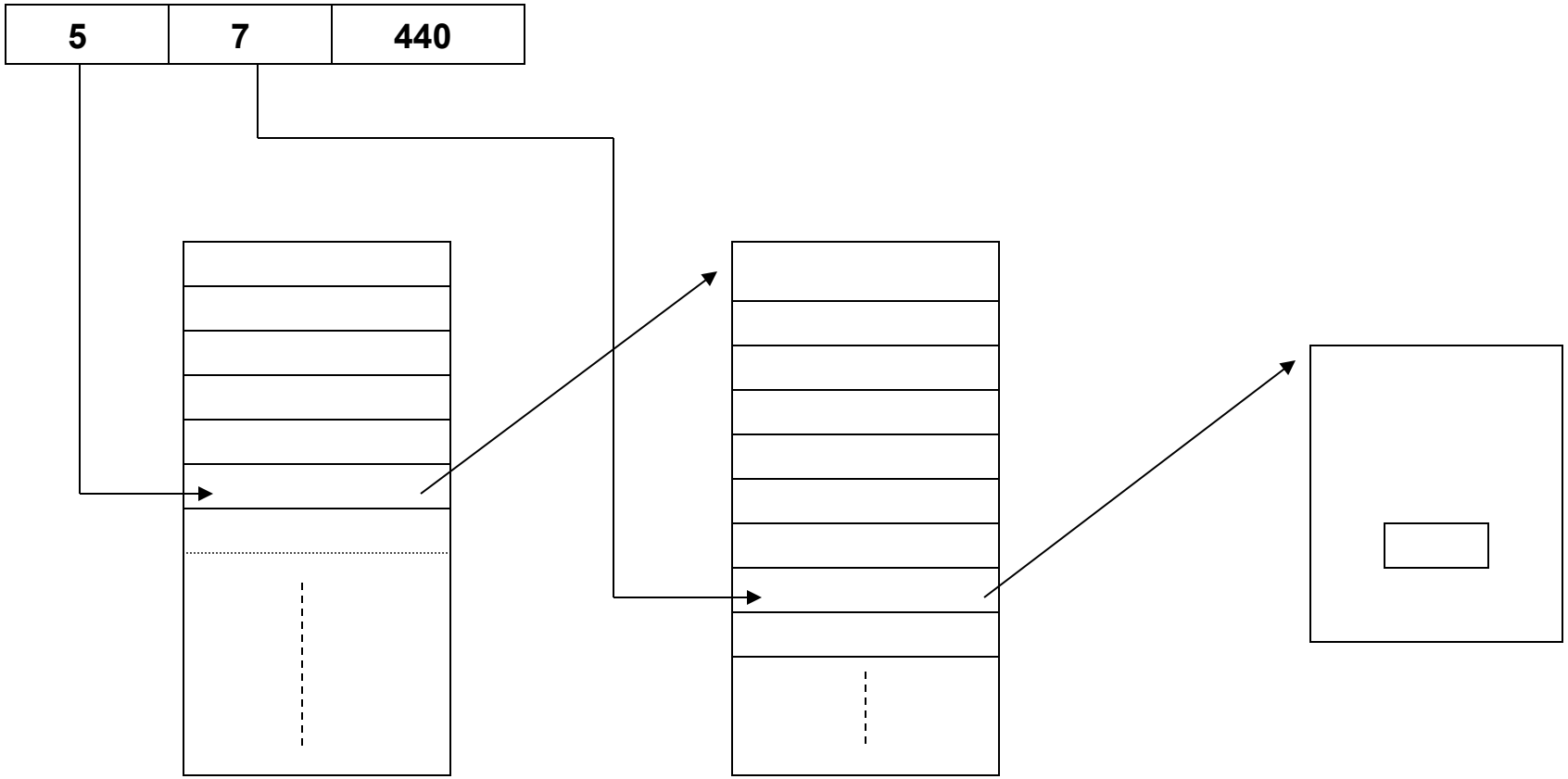
Il numero di entry in una page della page table è 2^{s-e}.

Gli indirizzi assumono il seguente significato:



- **f-(s-e)** bit leftmost: entry nella higher level page table
- **s-e** bit centrali: entry nella lower level page table
- **s** bit rightmost: offset nella pagina

Esempio:



Il valore 5 indica di accedere alla entry 5 della higher level page table.
Qui troviamo il frame# che ospita la page table di lower level da considerare.
Il valore 7 indica di accedere alla entry 7 della lower level page table.
Qui troviamo il frame# che ospita la pagina. Il frame# concatenato all'offset 440 costituisce l'indirizzo fisico.

Due considerazioni:

- La doppia paginazione si può generalizzare: si possono avere **n** livelli di paginazione con **n>2**;

- I page fault possono verificarsi anche perché le page table dei vari livelli non sono in memoria.

8.9: POLITICHE DI PAGE REPLACEMENT

Ricordiamo che, in seguito ad un page fault, se non esistono frame liberi è necessario un page replacement.

Quale frame conviene liberare?

- al fine di minimizzare i page-in conviene liberare un frame che ospita la pagina che ha la minor probabilità di avere accessi in futuro. Purtroppo questo non può essere stabilito con certezza;
- per evitare un page-out conviene liberare un frame che ospita una pagina che non necessita di page out.

Altra domanda: è necessario liberare un frame allocato al processo oppure si può scegliere di liberare un frame allocato ad un altro processo? Risponderemo nella Sezione 8.10.

Supponiamo che il numero massimo di pagine allocate ad un processo **P** sia costante (nella Sezione 8.10 discuteremo l'esistenza di alternative valide), e che sia necessario liberare un frame allocato al processo. Fondamentalmente esistono due politiche teoriche:

- **FIFO page replacement**: viene rimpiazzata la pagina del processo che è in memoria da più tempo;

- **Last Recently Used (LRU) page replacement**: viene rimpiazzata la pagina del processo in memoria che ha avuto l'accesso meno recente. Alla base di questa politica abbiamo il principio di località.

In seguito indichiamo con n_P il numero massimo di pagine allocate ad un processo **P**.

Esempio: assumiamo che $\mathbf{n_P = 2}$ e che **P** acceda alle proprie pagine nel seguente ordine:

0, 1, 0, 2, 0, 1, 2.

Pagine in memoria:

FIFO: **0, 01, 01, 12, 02, 01, 12**

LRU: **0, 01, 01, 02, 02, 01, 12**

Dato n_P il numero massimo di pagine allocate ad un processo **P**, sia $n_{P,K}$ l'insieme delle pagine del processo **P** che sono in memoria dopo **k** accessi.

Ovviamente $n_{P,K} <= n_P$.

Definizione: una politica di replacement ha la **stack property** se $n_P < n'_P$ implica $n_{P,k}$ contenuto in $n'_{P,k}$ per ogni **k.**

E' stato valutato che le politiche di replacement che godono della stack property hanno prestazioni migliori delle politiche che non godono della medesima proprietà.

La politica LRU gode della stack property, la politica FIFO no.

Esempio: Il processo P accede alle pagine nell'ordine:

5, 4, 3, 2, 1, 4, 3, 5, 4, 3, 2, 1, 5

$n_{P,K}$ con n_P**=3** e FIFO**:**

5,54,543, 243, 213, 214, 314, 354, 354, 354, 352, 152, 152

$n_{P,K}$ con n_P**=4** e FIFO**:**

5,54,543,5432,1432,1432,1432,1532,1542,1543,2543,2143,2153

$n_{P,K}$ con n_P**=3** e LRU**:**

5,54,543, 243, 213, 214, 314, 354, 354, 354, 324, 321, 521

$n_{P,K}$ con n_P**=4** e LRU**:**

5,54,543,5432,1432,1432,1432,1435,1435,1435,2435,2431,2531

Implementare le politiche FIFO o LRU richiede:

- supporto hardware
- tempo di computazione.

E' opportuno approssimare. Vediamo due esempi.

LRU approssimata con uso del reference bit:

- **ref** impostato ad 1 al momento del page in;
- **ref** aggiornato a 0 quando il S.O. deve effettuare un page replacement e trova tutti i reference bit impostati a 1;
- **ref** aggiornato ad 1 quando la pagina viene visitata.

Quando deve essere effettuato un page replacement ed è necessario liberare un frame:

- se possibile si sceglie un frame con **ref** = 0. Conviene scegliere un frame con modified bit = 0;
- se tutti i **ref** sono settati ad 1 vengono azzerati e la scelta è arbitraria.

FIFO approssimata: **second chance algorithm**:

- il S.O. mantiene una coda dei frame occupati;
- il S.O. imposta ad 1 il **ref** del frame ogni volta che il frame viene visitato;
- se deve essere liberato un frame, si considera il frame in testa alla coda (cioè il più anziano):

  - se ha **ref** = 0 viene liberato e viene tolto dalla coda;

  - se ha **ref** = 1 il bit viene impostato a 0 ed il frame viene inserito in fondo alla coda (il frame ha una second chance).

8.10: ALLOCAZIONE DELLA MEMORIA AI PROCESSI

Ricordando che

- se un processo ha pochi frame allocati rischia il trashing;
- se i processi hanno troppi frame allocati diminuisce il numero di processi in memoria,

il S.O. deve determinare il numero di frame allocati ad ogni processo.

Abbiamo 2 tipi di allocazione:

- **allocazione fissa**: il numero di frame allocati ad un processo è scelto staticamente e non varia mai;
- **allocazione variabile**: il numero di frame allocati ad un processo può variare durante l'esecuzione.

Abbiamo 2 tipi di replacement:

- **replacement locale**: per allocare un frame ad una pagina del processo P può essere liberato solo un frame allocato a P;
- **replacement globale**: per allocare un frame ad una pagina del processo P può essere liberato un frame allocato a qualsiasi processo.

Abbiamo tre approcci:

1. allocazione fissa, replacement locale:

- la memoria allocata al processo è determinata in base a qualche criterio quando il processo inizia l'esecuzione;
- l'implementazione è facile;
- il page replacement costa poco, perché coinvolge pochi frame;
- se la decisione iniziale non è ottimale si rischia il trashing, oppure di aver allocato memoria non utilizzata al processo.

2. allocazione variabile, replacement globale:

- c'è il rischio che un processo ottenga troppi frame. Esempio: se abbiamo la politica di replacement LRU verranno liberate pagine di processi non running e non pagine del processo running;
- c'è il rischio che un processo in waiting perda tutti i frame, e che abbia numerosi page fault quando verrà schedulato.

3. allocazione variabile, replacement locale:

- si evitano i problemi menzionati nei punti 1) e 2);
- non è facile determinare il numero di frame da allocare al processo durante l'esecuzione.

Esempio di approccio 3: il **modello working set**.

Definizione: dato un parametro **d**, il **working set** di un processo **P** è l'insieme delle pagine visitate nelle ultime **d** istruzioni di **P**.

Notazione:

- **WS(P,t,d)**: working set di **P** al tempo **t** dato il parametro **d**;
- **WSS(P,t,d)**: numero di pagine di **WS(P,t,d)**.

Allocazione:

ad ogni istante **t**, le pagine **WS(P,t,d)** sono tutte in memoria, ed il processo **P** ha allocato **WSS(P,t,d)** frame. Se questo è impossibile, **P** va in waiting.

Motivazioni:

- per il principio di località, la probabilità che **P** acceda ad una pagina in memoria è elevata;
- se **d** è scelto bene, si evita che **P** abbia allocato un numero troppo basso di frame.

Dinamica:

- per ogni processo **P**, **WSS(P,t,d)** varia in funzione di **t**;
- se la somma degli **WSS(P,t,d)** di tutti i processi **P** in memoria è maggiore del numero di frame disponibili, uno o più processi vengono rimossi dalla memoria;
- se c'è un processo **P** con **WSS(P,t,d)** minore del numero di frame liberi, allora le pagine in **WS(P,t,d)** possono essere allocate in memoria.