Università di Roma Tor Vergata
Corso di Laurea triennale in Informatica

Sistemi operativi e reti

A.A. 2018-2019

Pietro Frasca

Lezione 15

Martedì 27-11-2018

Prevenzione dinamica

- Le tecniche di prevenzione dinamica si basano su algoritmi in grado di verificare, in base allo stato corrente di allocazione delle risorse e alle richieste dei processi, se l'assegnazione di risorse dovute ad una nuova richiesta da parte di un processo può portare a una situazione di stallo.
- Un noto algoritmo di prevenzione dinamica, ideato da Dijkstra, è l'**algoritmo del banchiere** (per una certa analogia al comportamento del banchiere) .
- L'algoritmo risulta molto riduttivo per essere usato nei SO di uso generale in quanto è basato sui seguenti vincoli:
  1. il sistema operativo può gestire un numero fisso di processi e un numero fisso di risorse. Inoltre i processi devono dichiarare inizialmente il numero massimo di risorse di cui hanno bisogno durante la loro esecuzione.
  2. I processi possono richiedere nuove risorse mantenendo le unità già in loro possesso.
  3. Tutte le risorse assegnate a un processo sono rilasciate quando il processo termina la sua esecuzione.

- Gli algoritmi di prevenzione dinamica si basano sul concetto di ***stato sicuro***.
- Lo stato del sistema si dice ***sicuro*** se è possibile trovare una sequenza **Ph-Pj..Pk** con cui assegnare le risorse ai processi, detta **sequenza sicura**, in modo tale che tutti i processi possano usare le risorse che richiedono e terminare.
- Se, in un determinato istante, le risorse che un processo **Pi** richiede non sono disponibili, allora **Pi** si blocca fino a che tutti i processi che lo precedono nella sequenza liberino un numero sufficiente di risorse necessarie a **Pi**.
- Se non esiste una sequenza sicura allora lo stato del sistema è detto ***non è sicuro***. Uno stato non sicuro **può portare** a una condizione di deadlock.
- L'algoritmo deve quindi consentire l'allocazione delle risorse ai processi solo quando le allocazioni portano a stati sicuri.

- Consideriamo, ad esempio, il caso di un sistema con tre processi P1, P2, P3 in cui siano disponibili 15 unità di **un solo tipo** di risorsa e che sia noto il massimo numero di risorse che ciascun processo può richiedere: 12 per P1, 3 per P2 e 11 per P3. Lo **stato iniziale** del sistema può essere così rappresentato:

	R1
P1	0
P2	0
P3	0

Risorse allocate

	R1
P1	12
P2	3
P3	11

Risorse richieste

R1
15

Risorse totali

R1
15

Risorse libere

- Se dopo un certo periodo di tempo sono state assegnate 8 unità a p1, 2 a p2 e 11 a P3, lo stato in cui il sistema si trova può essere così descritto:

	R1
P1	8
P2	2
P3	3

Risorse allocate

	R1
P1	4
P2	1
P3	8

Risorse richieste

R1
15

Risorse totali

R1
2

Risorse libere

- Vediamo se questo **stato è sicuro,** verificando se, a partire da questo stato, esiste una sequenza sicura.

	R1
P1	8
P2	2
P3	3

Risorse allocate

	R1
P1	4
P2	1
P3	8

Risorse richieste

R1
15

Risorse totali

R1
2

Risorse libere

- Si può notare che:
  1. il processo **P2** può allocare la risorsa richiesta, potendo quindi completare la sua esecuzione e liberare le sue 3 risorse che aveva allocato, portando quindi a 4 le risorse disponibili.

	R1
P1	8
P2	**3**
P3	3

Risorse allocate

	R1
P1	4
P2	**0**
P3	8

Risorse richieste

R1
15

Risorse totali

R1
1

Risorse libere

- Quindi quando P2 termina la stato di allocazione è il seguente:

	R1
P1	8
P2	0
P3	3

Risorse allocate

	R1
P1	4
P2	0
P3	8

Risorse richieste

R1
15

Risorse totali

R1
4

Risorse libere

2. A questo punto, il processo **P1** può ora ottenere tutte le 4 risorse ancora necessarie e terminare, portando a 12 le risorse disponibili che consentono al processo **P3** di terminare.

- Lo stato indicato è quindi uno ***stato sicuro*** in quanto partendo da esso esiste la ***sequenza sicura*** **(P2, P1, P3).**

- Altre sequenze potrebbero far passare il sistema da uno stato sicuro a uno stato non sicuro.

	R1
P1	8
P2	2
P3	3

Risorse allocate

	R1
P1	4
P2	1
P3	8

Risorse richieste

R1
15

Risorse totali

R1
2

Risorse libere

- Se, ad esempio, il processo **P3** chiede e otteniene una risorsa, il sistema in questo caso passerebbe in **uno stato che non è sicuro**.

	R1
P1	8
P2	2
P3	**4**

Risorse allocate

	R1
P1	4
P2	1
P3	**7**

Risorse richieste

R1
15

Risorse totali

R1
1

Risorse libere

- Infatti, l'unica risorsa rimasta libera può soddisfare soltanto la richiesta del processo P2 consentendogli di terminare l'esecuzione e liberare le 3 risorse in suo possesso.

	R1
P1	8
P2	**3**
P3	4

Risorse allocate

	R1
P1	4
P2	**0**
P3	7

Risorse richieste

R1
15

Risorse totali

R1
0

Risorse libere

- A questo punto nessun altro processo può terminare: P1 non può ottenere le 4 risorse di cui ha bisogno non essendo queste disponibili e quindi deve essere sospeso; analogamente P3 non può ottenere le 7 risorse e quindi anche esso viene sospeso. Si giunge quindi ad una **situazione di stallo**.

Nell’esempio, per evitare lo stallo, a partire dal precedente stato sicuro, la risorsa richiesta da P3 non deve essere ad esso allocata anche se disponibile.

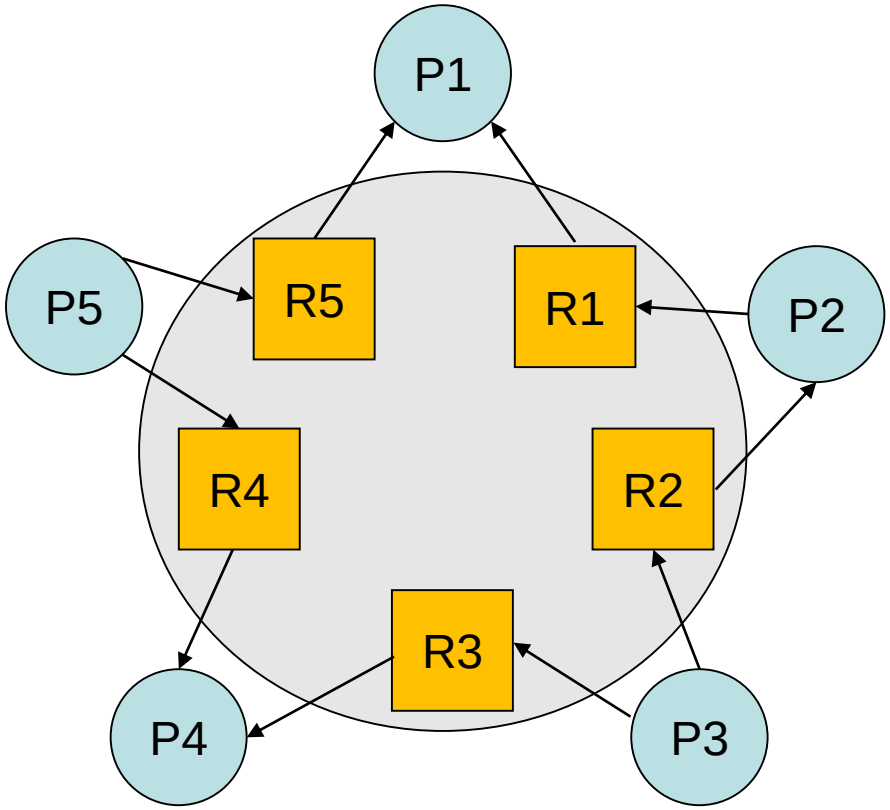
Rilevamento dei blocchi critici

- Se non si prendono adeguati provvedimenti di prevenzione statica o dinamica è possibile che si verifichino situazioni di stallo, coinvolgendo un certo numero di processi e di risorse.
- Spesso si ricorre solo alla rilevazione e alla eliminazione del blocco critico, senza ricorrere ad alcuna tecnica di prevenzione, come nel caso di Windows e Unix.
- L'algoritmo di **rilevazione** viene eseguito dal SO periodicamente con una frequenza che dipende dal tipo di applicazioni o quando ad esempio il grado d'uso della CPU scende sotto una certa soglia in quanto una condizione di blocco critico può rendere inefficienti le prestazioni del sistema.
- L'eliminazione dello stallo si può ottenere con differenti tecniche, la più semplice ed estrema consiste nel fare terminare tutti i processi coinvolti.

- Una soluzione meno drastica consiste nel far terminare uno alla volta i processi coinvolti e liberando via via le risorse da esso allocate, fino a giungere all'eliminazione dello stallo. In questo caso è possibile usare politiche di selezione dei processi da far terminare, basate ad esempio sulla priorità, sul tempo di cpu utilizzato, sul numero di risorse allocate, etc.
- L'implementazione di tali strategie può portare ad un alto overhead per il SO, in quanto dopo ogni terminazione forzata occorre verificare di nuovo se c'è ancora una situazione di stallo.

Problema dei cinque filosofi

- Il problema dei 5 filosofi a cena è un esempio che mostra un problema di sincronizzazione tra thread (o processi). Cinque filosofi stanno cenando in un tavolo rotondo. Ciascun filosofo ha il suo piatto di spaghetti e una bacchetta a destra e un bacchetta a sinistra che condivide con i vicini. Ci sono quindi solo cinque bacchette e per mangiare ne servono 2 per ogni filosofo. Immaginiamo che durante la cena, un filosofo trascorra periodi in cui mangia e periodi in cui pensa, e che ciascun filosofo abbia bisogno di due bacchette per mangiare, e che le bacchette siano prese una alla volta. Quando possiede due bacchette, il filosofo mangia per un po' di tempo, poi lascia le bacchette, una alla volta, e ricomincia a pensare.
- Il problema consiste nel trovare un algoritmo che eviti sia lo stallo (deadlock) che l'attesa indefinita (starvation).
- Lo stallo può verificarsi se ciascuno dei filosofi acquisisce una bacchetta senza mai riuscire a prendere l'altra. Il filosofo F1 aspetta di prendere la bacchetta che ha in mano il filosofo F2, che aspetta la bacchetta che ha in mano il filosofo F3, e così via (condizione di attesa circolare).


P1
P5
R5
R1
P2
R4
R2
R3
P4
P3

La situazione di starvation può verificarsi indipendentemente dal deadlock se uno dei filosofi non riesce mai a prendere entrambe le bacchette.

- La soluzione qui riportata evita il verificarsi dello stallo evitando la condizione di attesa circolare, imponendo che i filosofi con indice dispari prendano prima la bacchetta alla loro destra e poi quella alla loro sinistra; viceversa, i filosofi con indice pari (considerando 0 pari) prendano prima la bacchetta che si trova alla loro sinistra e poi quella alla loro destra.

```
#include <pthread.h>
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
```

```c
#define NUMFILOSOFI 5
#define CICLI 100

typedef struct{
    int id;
    pthread_t thread_id;
    char nome[20];
  } Filosofo;

/* Le bacchette sono risorse condivise, quindi ne gestiamo
   l'accesso in mutua esclusione mediante l'uso di mutex*/
pthread_mutex_t bacchetta[NUMFILOSOFI];

/* sospende per un intervallo di tempo random l'esecuzione del
   thread chiamante */
void tempoRnd(int min, int max) {
  sleep(rand()%(max-min+1) + min);
}
```

```
void *filosofo_th(void *id){
  Filosofo fil=*(Filosofo *)id;
  int i;
  for (i=0; i<CICLI; i++){
    printf("Filosofo %d: %s sta pensando \n",fil.id+1,fil.nome);
    tempoRnd(3, 12);
    printf("Filosofo %d: %s ha fame\n", fil.id+1,fil.nome);
    /* condizione che elimina l'attesa circolare */
    if (fil.id % 2){
      pthread_mutex_lock(&bacchetta[fil.id]);
      printf("Filosofo %d: %s prende la bacchetta destra (%d)\n",
        fil.id+1,fil.nome,fil.id+1);
      tempoRnd(1,2);
      pthread_mutex_lock(&bacchetta[(fil.id+1)%NUMFILOSOFI]);
      printf("Filosofo %d: %s prende la bacchetta sinistra
        (%d)\n", fil.id+1, fil.nome,(fil.id+1)%NUMFILOSOFI+1);
     }
```

```
    else{
      pthread_mutex_lock(&bacchetta[(fil.id+1) % NUMFILOSOFI]);
      printf("Filosofo %d: %s prende la bacchetta sinistra
       (%d)\n", fil.id+1, fil.nome,(fil.id+1)%NUMFILOSOFI+1);
      tempoRnd(1,2);
       pthread_mutex_lock(&bacchetta[fil.id]);
       printf("Filosofo %d: %s prende la bacchetta destra
        (%d)\n",fil.id+1,fil.nome,fil.id+1);
    }
    printf("Filosofo %d: %s sta mangiando \n", fil.id+1,
      fil.nome);
    tempoRnd(3, 10);
    pthread_mutex_unlock(&bacchetta[fil.id]);

    printf("Filosofo %d: %s posa la bacchetta destra (%d)\n",
      fil.id+1, fil.nome,fil.id+1);
    pthread_mutex_unlock(&bacchetta[(fil.id+1) % NUMFILOSOFI]);

    printf("Filosofo %d: %s posa la bacchetta sinistra (%d)\n",
      fil.id+1, fil.nome, (fil.id+1)%NUMFILOSOFI+1);
  } //ciclo for

}
```

```
int main(int argc, char *argv[]){
  int i;
  char nome[][20]={"Socrate","Platone","Aristotele","Talete",
   "Pitagora"};
  Filosofo filosofo[NUMFILOSOFI];
  srand(time(NULL));

  /* inizializza i mutex */
  for (i=0; i<NUMFILOSOFI; i++)
    pthread_mutex_init(&bacchetta[i], NULL);

  /* crea e avvia i threads */
  for (i=0; i<NUMFILOSOFI; i++){
    filosofo[i].id=i;
    strcpy(filosofo[i].nome,nome[i]);
    if (pthread_create(&filosofo[i].thread_id, NULL, filosofo_th,
       &filosofo[i]))
      perror("errore pthread_create");
    }
```

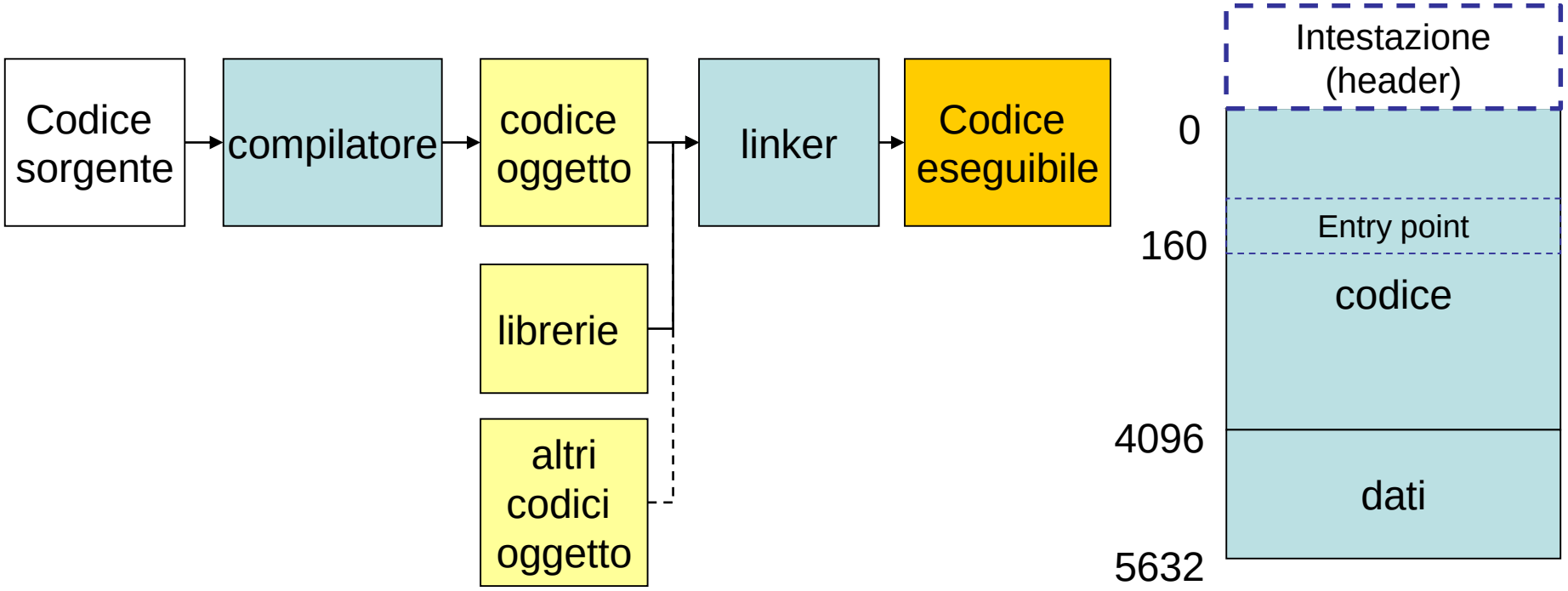
```
/* il thread main attende che i filosofi terminino */
  for (i=0; i<NUMFILOSOFI; i++)
    if (pthread_join(filosofo[i].thread_id, NULL))
      perror("errore pthread_join");
  return 0;
}
```

Gestione della memoria

- La memoria principale è una risorsa importante che va gestita in modo efficiente. Il componente che gestisce la memoria è detto ***gestore della memoria***. Il suo compito è tenere traccia di quali parti di memoria sono in uso, allocare la memoria ai processi in base alle loro richieste e liberarla quando i processi terminano.
- Analizzeremo varie tecniche di gestione della memoria dalle più semplici alle più complesse che dipendono sia dall'architettura del calcolatore, in particolare del processore, sia dall'architettura software del SO.
- Prima di mostrare le tecniche introduciamo alcuni concetti e definizioni basilari.

Creazione di un file eseguibile

- Un programma è costituito da un in insieme di moduli scritti mediante un linguaggio (anche più di uno) di programmazione (**codice sorgente**). Ciascun modulo sorgente viene compilato (dal compilatore) producendo il modulo oggetto il quale viene collegato con eventuali altri moduli oggetto e/o librerie dal **linker**, producendo il **codice eseguibile** che viene caricato in memoria dal **caricatore**.



Lo **spazio virtuale di un processo**: memoria necessaria a contenere il suo codice, i dati su cui operare e lo stack (compreso l'heap).

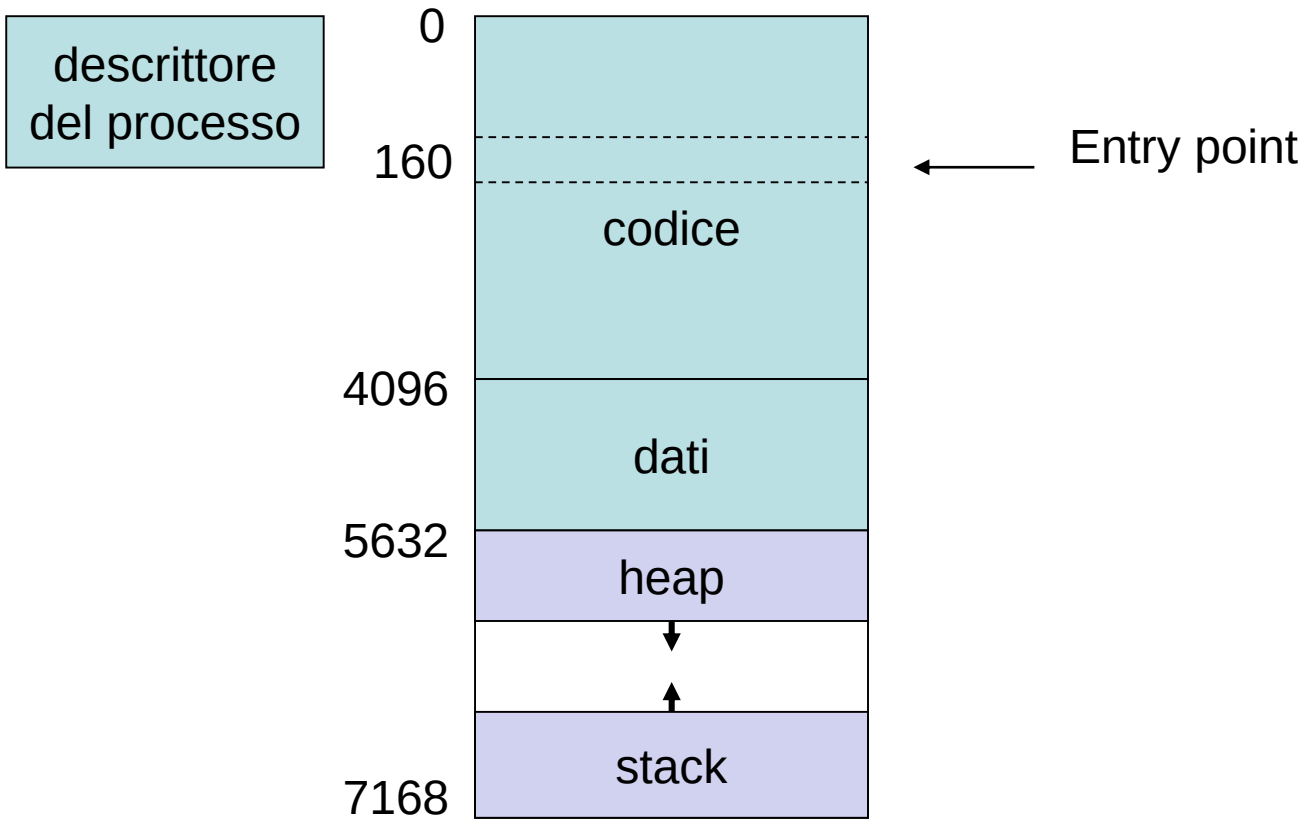


Immagine di un processo

Organizzazione della spazio virtuale

- **Spazio virtuale unico**

  Inizialmente, consideriamo il caso in cui lo spazio virtuale del processo sia costituita da **un unico spazio** di indirizzi virtuali (indirizzi compresi tra 0 e 7168).

  Supponiamo che il linker generi indirizzi contigui per tutti i moduli che compongono il programma.

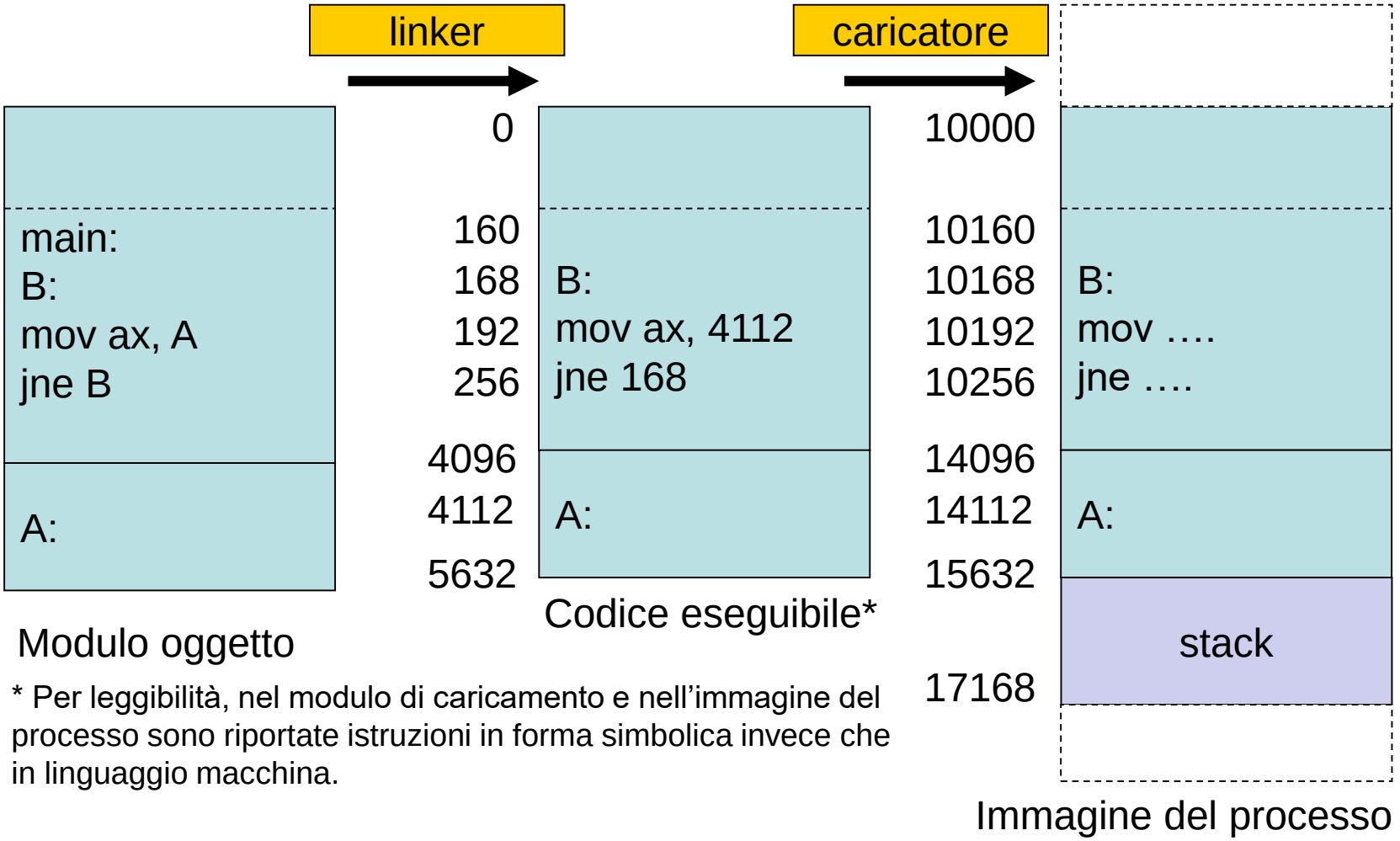
  In particolare la struttura del processo visto è composta da tre parti, dette **segmenti**, distinte:

  - **segmento del codice (text)**
  - **segmento dati**
  - **segmento dello stack (non consideriamo l'heap)**

Rilocazione statica e dinamica

- Generalmente, lo spazio virtuale di un processo sarà caricato in memoria fisica a partire da qualsiasi indirizzo **l** diverso da **0**.
- Pertanto distinguiamo gli indirizzi di memoria virtuale di un processo detti ***indirizzi virtuali (o logici)***) dagli indirizzi di memoria fisica (***indirizzi fisici***).
- Ad esempio, nel caso di un processo con un immagine come nella figura precedente, l'insieme di tutti gli indirizzi virtuali del processo (***spazio degli indirizzi virtuali***) è dato dai valori compresi tra 0 e 7168 (7 KB). Nel caso in cui il processo fosse caricato in memoria a partire dall'indirizzo **10000**, gli indirizzi fisici ad esso associati (***spazio degli indirizzi fisici***) sarebbero compresi tra **10000** e **17168** (10000+7168).
- I due spazi di indirizzi non sono indipendenti ma sono legati da una funzione detta **funzione di rilocazione.**

Generazione della memoria virtuale e dell'immagine del processo



* Per leggibilità, nel modulo di caricamento e nell'immagine del processo sono riportate istruzioni in forma simbolica invece che in linguaggio macchina.

$$If = f(Iv)$$

- La funzione di rilocazione fa corrispondere ad ogni indirizzo virtuale **Iv** un indirizzo fisico **If**.
- Un esempio di funzione di rilocazione è data da:

$$If = Iv + I0$$

dove **I0** è una costante che indica l'indirizzo di base.

- Nel esempio mostrato la funzione di rilocazione è la seguente:

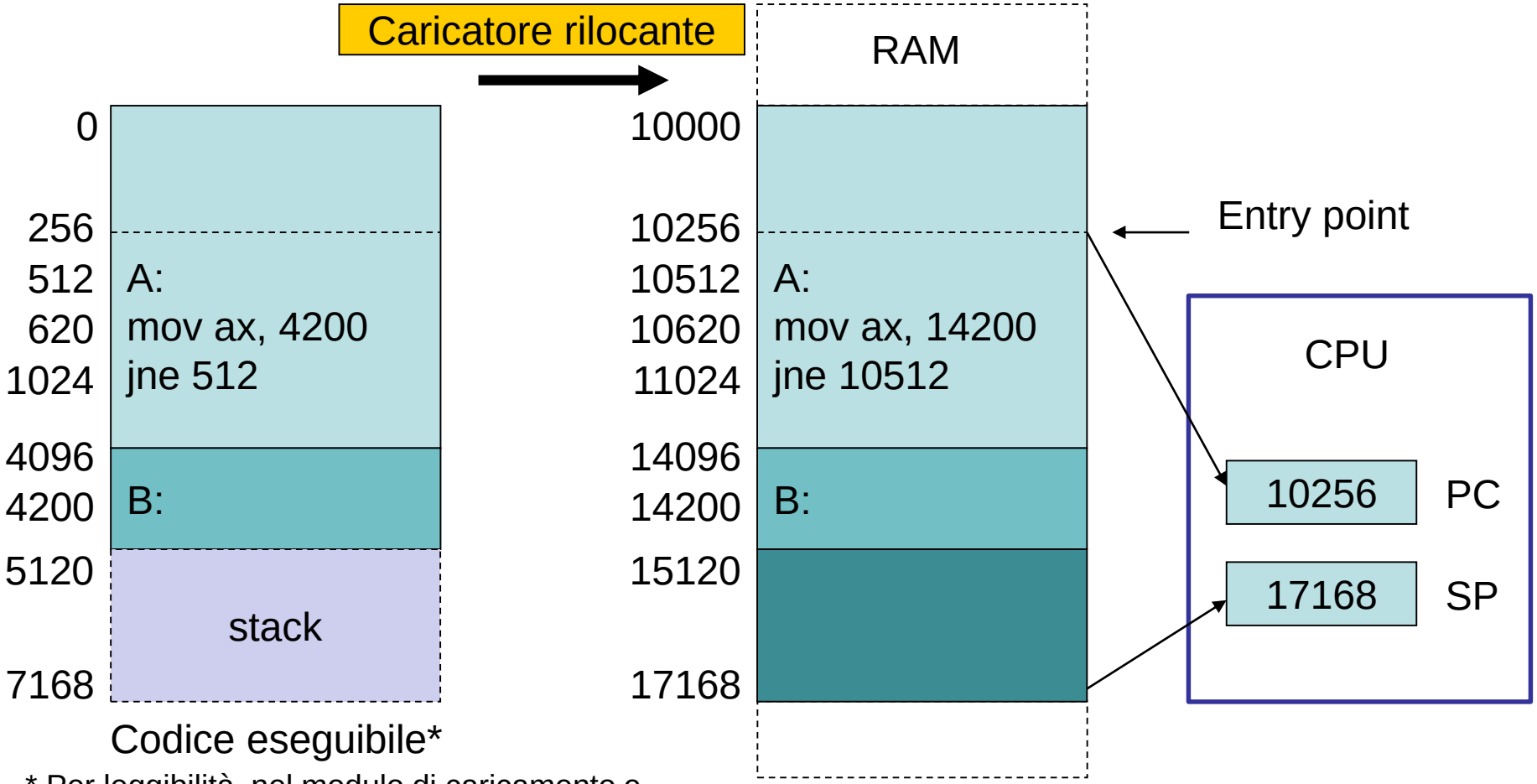
$$If = Iv + 10000$$

- Le tecniche per effettuare la rilocazione degli indirizzi sono di due tipi: **rilocazione statica** e **rilocazione dinamica.**

Rilocazione statica

- Tecnica più semplice. Tutti gli indirizzi virtuali sono rilocati prima che il processo inizi la sua esecuzione.
- La rilocazione è eseguita dal caricatore (***caricatore rilocante***) durante la fase di caricamento, modificando tutti gli indirizzi da virtuali a fisici che sono ottenuti sommando a ciascun indirizzo virtuale l'indirizzo iniziale di caricamento.
- Nel momento della creazione del processo nel registro **PC** è caricato l'indirizzo fisico dell'**entry point** del programma e nello **SP** l'indirizzo fisico relativo alla **base dello stack**.
- La figura seguente mostra l'immagine di un processo rilocata a partire dall'indirizzo 10000.

Rilocazione statica



* Per leggibilità, nel modulo di caricamento e nell'immagine del processo sono riportate istruzioni in assembler invece che in linguaggio macchina.

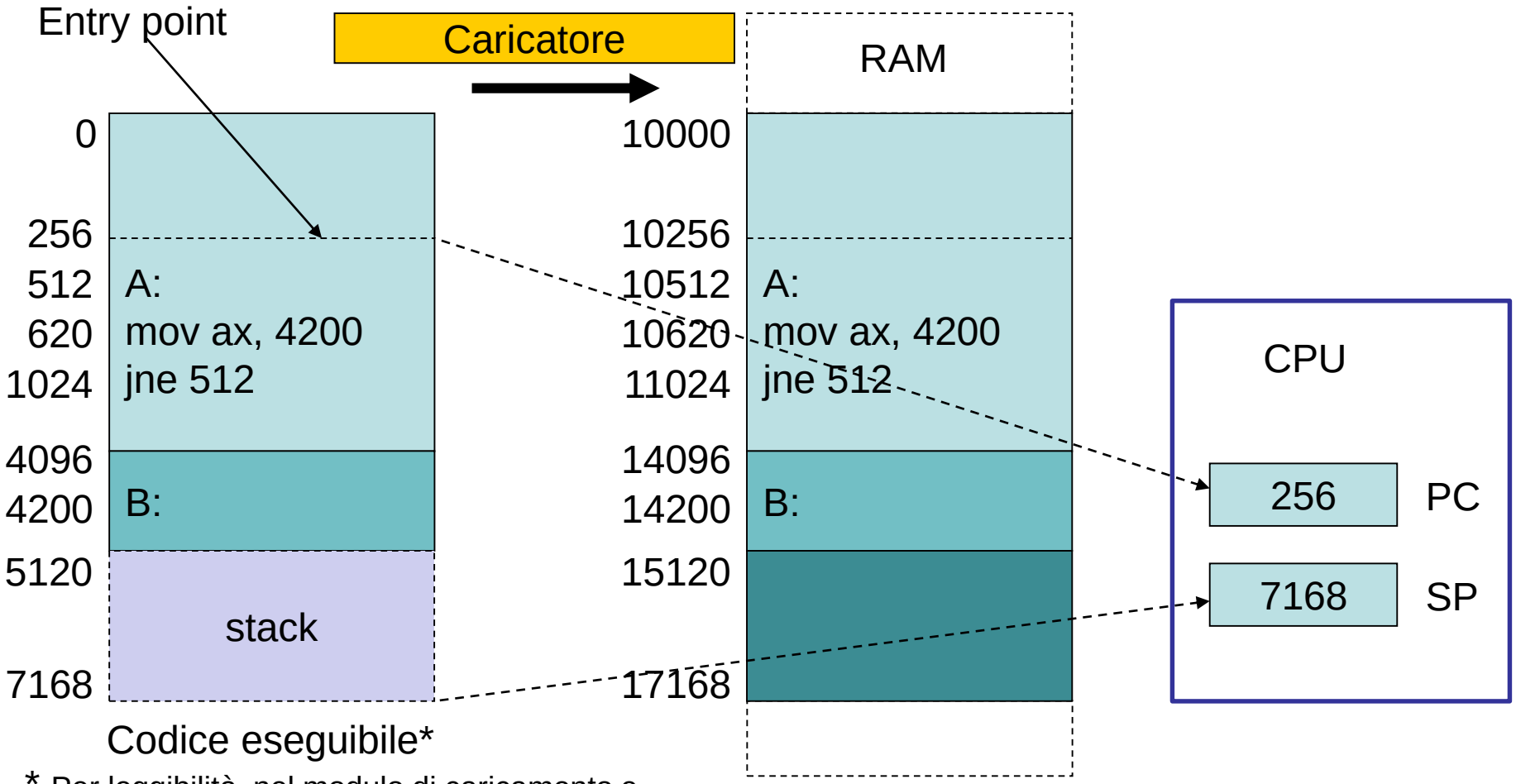
Rilocazione dinamica

- Con la rilocazione dinamica, la traduzione degli indirizzi da virtuali a fisici avviene durante l'esecuzione del processo.
- Il caricatore trasferisce in memoria direttamente il contenuto del codice eseguibile contenente gli indirizzi virtuali, senza modificarli.
- Nei registri **PC (**program counter) e **SP** (stack pointer) sono caricati i valori degli indirizzi virtuali relativi all'entry point del programma e della base dello stack.
- L'architettura di un processore che consente di effettuare la rilocazione dinamica deve avere un supporto hardware (spesso chiamato **MMU** (**Memory Management Unit**)) che implementa la funzione di rilocazione.

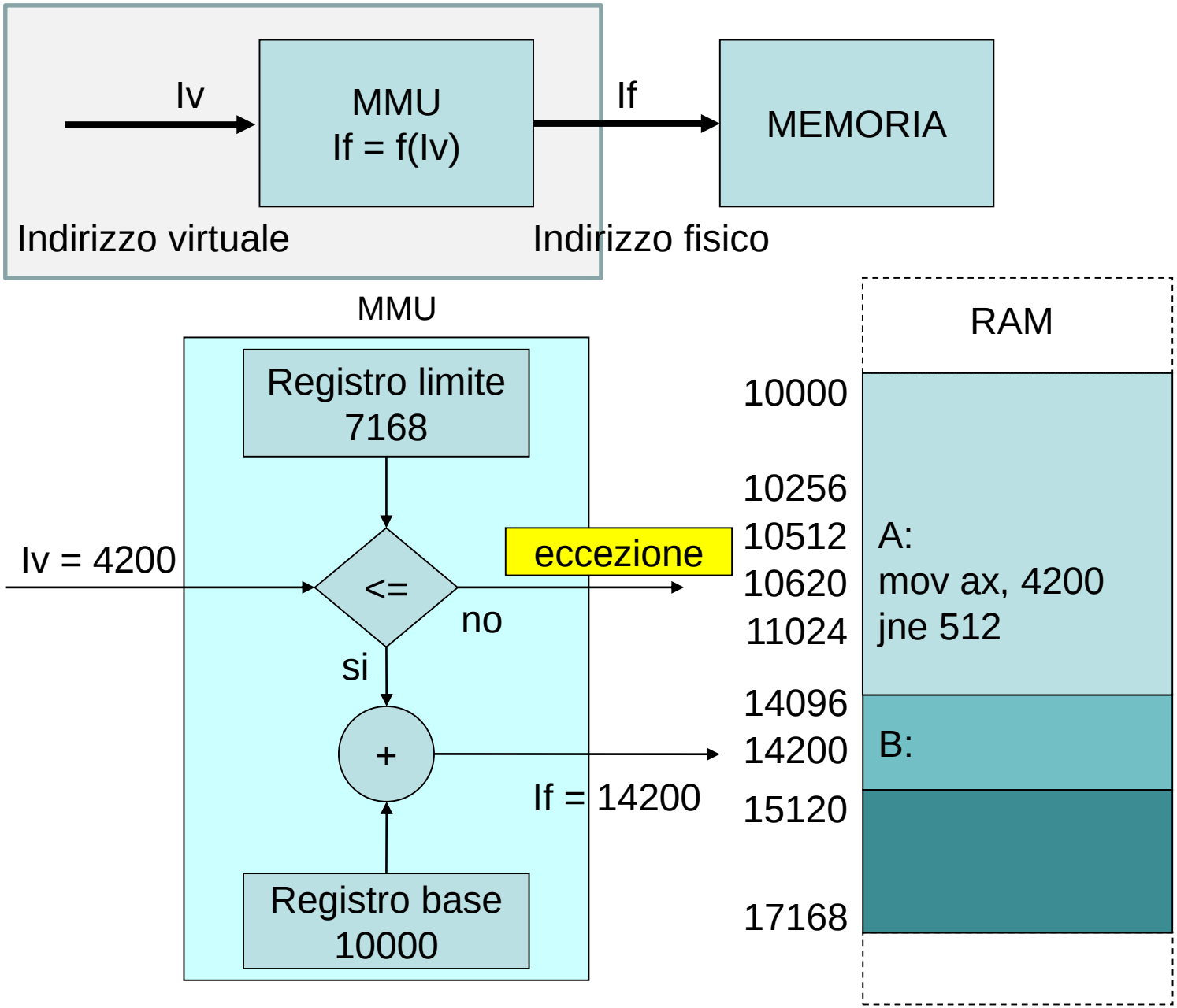
  If = f(Iv)

  traducendo dinamicamente ogni indirizzo virtuale nel corrispondente indirizzo fisico.

Rilocazione dinamica



* Per leggibilità, nel modulo di caricamento e nell'immagine del processo sono riportate istruzioni in assembler invece che in linguaggio macchina.


Iv
MMU
If = f(Iv)
If
MEMORIA
Indirizzo virtuale
Indirizzo fisico
MMU
Registro limite
7168
Iv = 4200
<=
eccezione
no
si
+
If = 14200
Registro base
10000
RAM
10000
10256
10512
10620
11024
14096
14200
15120
17168
A:
mov ax, 4200
jne 512
B:

- Con la rilocazione dinamica, quando ad un processo è revocata la memoria (funzione di swap_out) e successivamente viene riallocata una diversa partizione (funzione di swap_in), in quanto il processo torna nello stato di esecuzione, viene caricato nel registro base l’indirizzo iniziale della nuova area di memoria assegnata al processo.