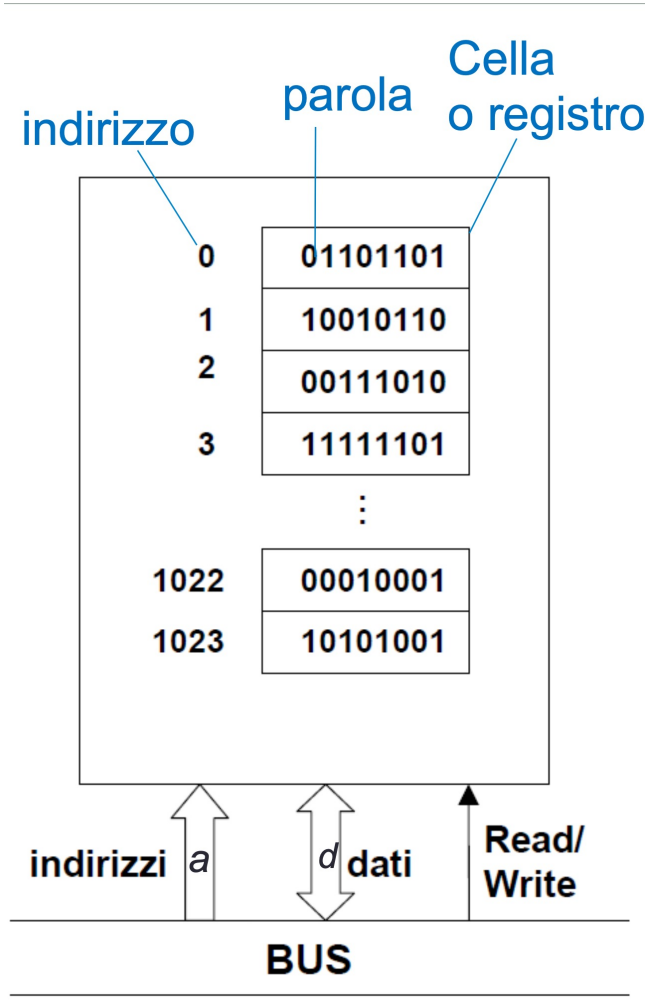
Gestione Memoria

Parte I

Introduzione

- La memoria è un vettore di parole o byte, ogni byte ha un proprio indirizzo.
- La CPU preleva le istruzioni dalla memoria sulla base del contenuto del registro PC (Program Counter). Tali istruzioni possono determinare ulteriori letture (LOAD) o scritture (STORE) in specifici indirizzi di memoria
- Tipico ciclo istruzione:
  - Istruzione prelevata dall amemoria; decodificata (eventuale prelievo di operandi dalla memoria); eseguita sugli operandi; I risultati possono essere salvati nella memoria
- In generale, il programma risiede in un disco in forma di un file binario eseguibile. Il programma deve essere caricato in memoria per essere eseguito.



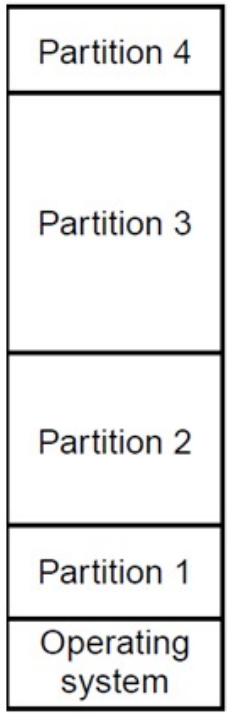
Allocazione memoria

Allocazione memoria

- Quattro concetti base:
  - Allocazione contigua: tutto lo spazio assegnato ad un processo deve essere formato da celle consecutive
  - Allocazione non contigua: posso assegnare ad un processo celle non adiacenti tra di loro
    - La MMU deve essere in grado di supportare questo approccio
  - Allocazione statica: un processo deve mantenere la propria area di memoria dal caricamento alla terminazione
  - Allocazione dinamica: un processo può essere spostato all'interno della memoria durante l'esecuzione

Allocazione a partizioni Multiple Fisse

- La memoria è divisa in un numero fisso di aree dette partizioni di dimensioni diverse
  - Ogni partizione è definita da una coppia base-limite
  - Il numero di partizioni definisce il grado di multiprogrammazione
- Bisogna sapere la dimensione del processo prima di caricarlo
- Nel context switch SO carica
  - Nel registro rilocazione l'indirizzo base della partizione
  - Nel registro limite la dimensione del processo



PARTIZIONI FISSE

PARTIZIONI FISSE

Supponiamo che ogni partizione abbia dimensione 40 K

Ogni partizione può contenere un **solo** processo

Vantaggi: **gestione semplice** da parte del sistema operativo, poiché si deve tenere traccia solo di quali partizioni sono libere

Svantaggi:

1) Frammentazione interna
2) Non si possono eseguire processi con dimensione superiore a quella della partizione (nel nostro caso non si possono eseguire processi dimensione superiore a 40 K)

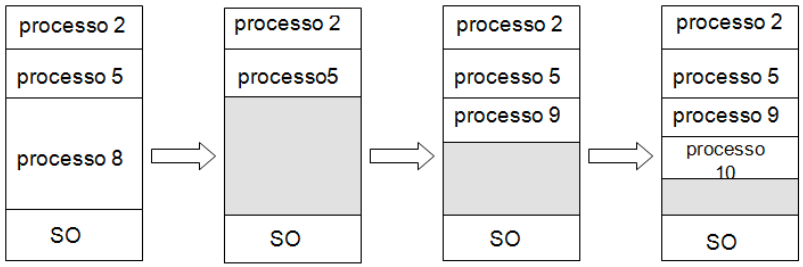
P1	30 K
P2	20 K
P3	5 K
P4	45 K

Le partizioni sono fisse ma possono avere dimensione diversa.

[Video Prof. Terulli]

Allocazione a partizioni multiple variabili e indirizzamento rilocabile

- Partizioni fisse:
  - Processi piccoli causano spreco di spazio
  - Processi grandi non posso essere eseguiti
    - Meno multiprogrammazione
    - Più frammentazione interna

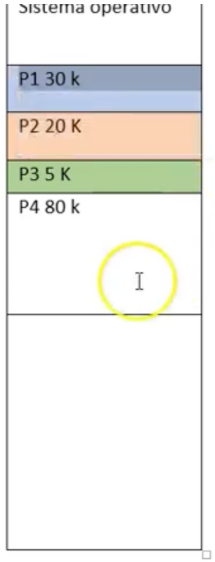


- Allora usiamo partizioni variabili!
- Quando un processo arriva il gestore della memoria cerca una partizion libera abbastanza grande per il processo

PARTIZIONI VARIABILI

Non esistono partizioni prestabilite.

Man mano che arrivano i processi verrà creata una partizione della stessa dimensione del processo stesso



P1	30 K
P2	20 K
P3	5 K
P4	80 k
P5	25 k

Vantaggi:

1) Si creano partizioni esattamente della stessa dimensione del processo, quindi non si crea frammentazione interna
2) Ci possono essere molti più processi rispetto alla gestione a partizioni fisse
3) Partizioni attigue possono essere unite per creare una partizione più grande

Svantaggi:

1) Gestione più complessa da parte del S.O.; infatti per ogni partizione bisogna conoscere la sua posizione, la sua dimensione e se è libera oppure no
2) C'è frammentazione esterna
3) Un processo più grande di tutta la memoria disponibile non può essere eseguito

[Video Prof. Terulli]

Allocazione memoria continua – partizioni variabili - algoritmi

SO usa uno degli algortimi per trovare una partizione adatta al processo e gestire la memoria

Partizioni variabili soffrono di…

… frammentazione esterna

Quando I processi finiscono di creano dei buchi

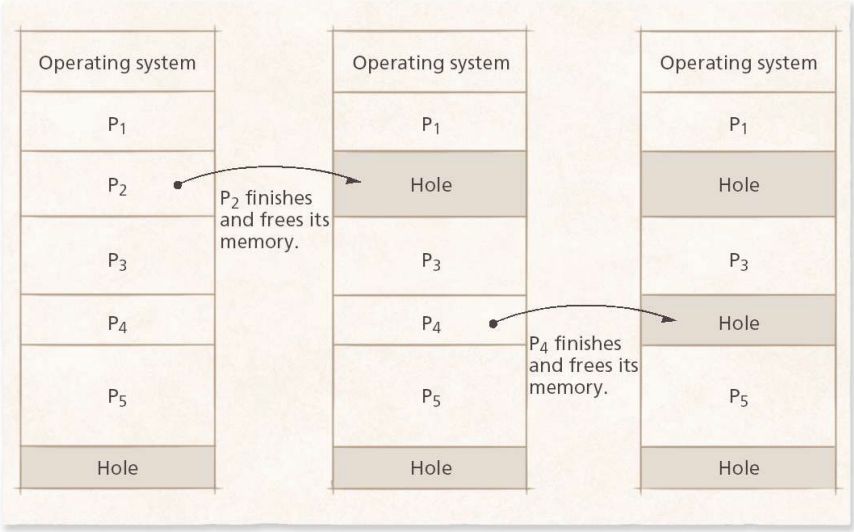
Worst-fit, best-fit, first-fit non risolvono la frammentazione esterna… ma tendono a produrre buchi sempre più piccoli

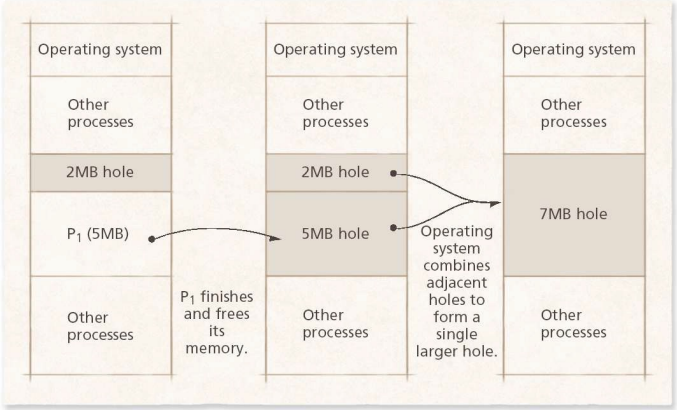
Coalescenza: unisco blocchi adiacenti rimasti non allocati (molto costoso e non recupera rapidamente memoria)

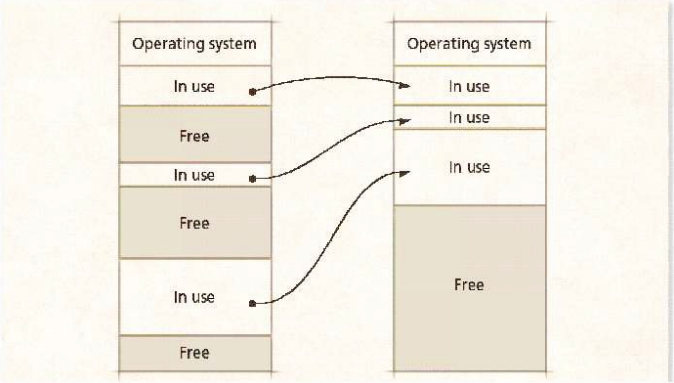
Compattamento: unisce tutti I buchi in un singolo blocco e sposta tutti I blocchi occupati

Ancora non posso caricare un processo in memoria se memoria non ha abbastanza spazio…

… allocazione non contigua!







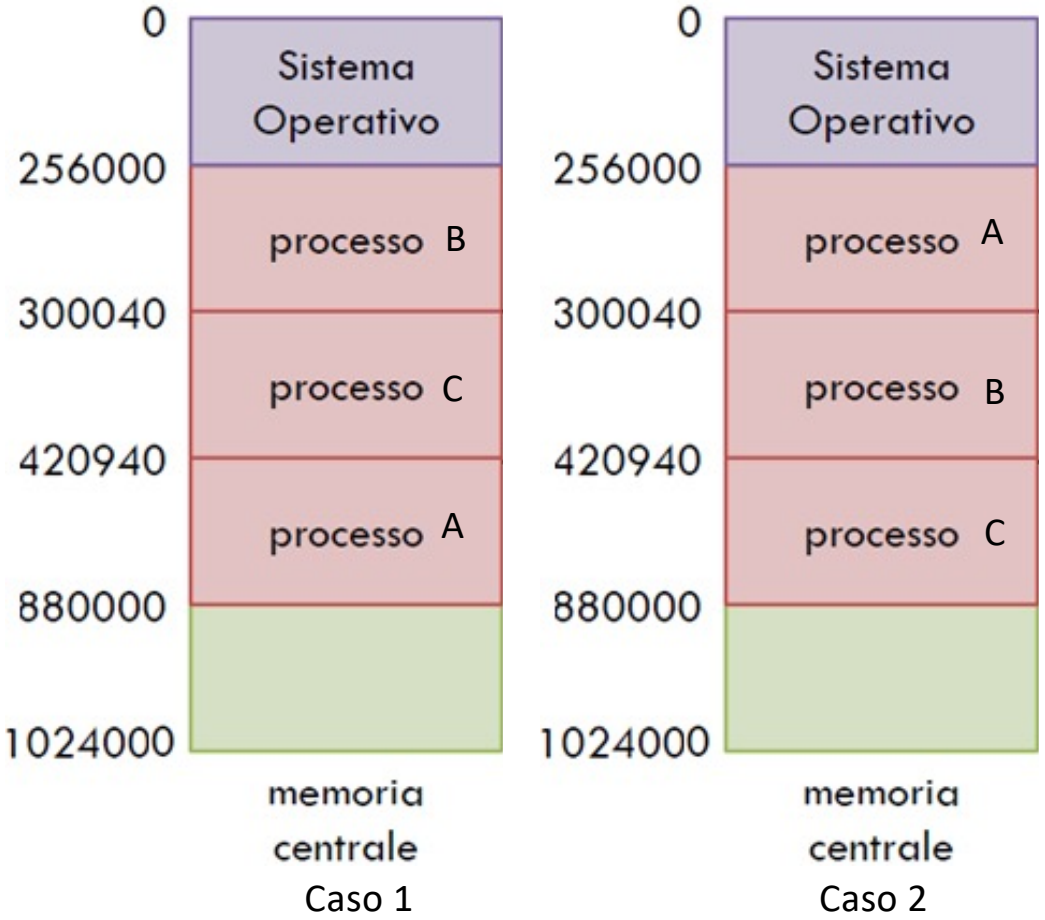
Algoritmi per partizioni variabili

- *First-fit: assegna il primo blocco libero abbastanza grande per contenere lo spazio richiesto*
  - *facile da implementare e basso overhead*
- *Best-fit: assegna il più piccolo blocco libero abbastanza grande*
  - Bisogna scandire tutta la lista dei buchi (Maggiore overhead) – se non ordinate per dimensione
  - Si produce il buco più piccolo residuo
- *Worst-fit: assegna il più grande blocco libero.*
  - *Si deve scandire la lista*
  - *Si produce il più grande buco residuo che può essere utile per allocazioni successive*

Worst-fit è il peggiore per tempi e uso della memoria. First-fit e best-fit sono paragonabili per uso della memoria ma il primo è più veloce

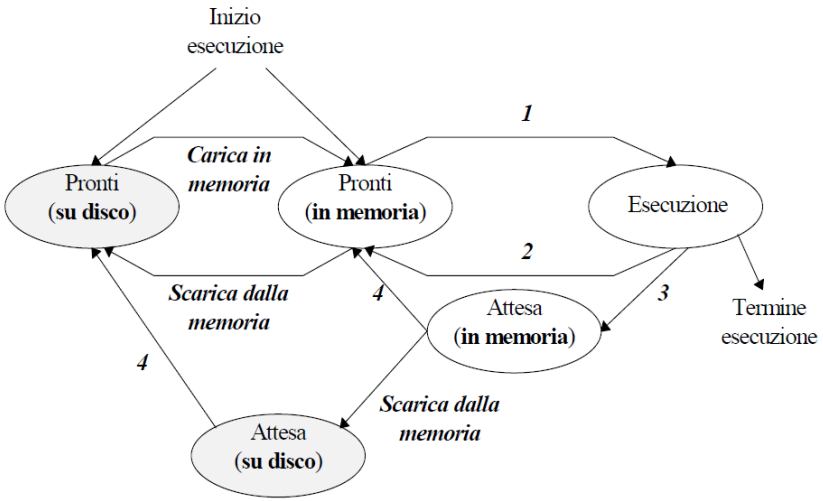
Indirizzamento

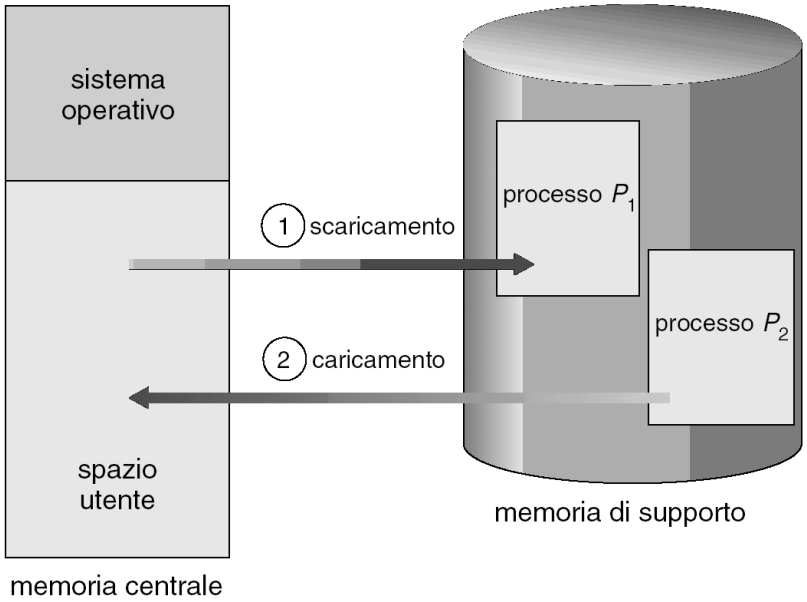
- Caso 1 : Processo A viene assegnato spazio 429040 – 880000
- Caso 2: processo A viene assegnato spazio 256000 – 300040
- Come fa il programmatore a sapere dove risiede A in memoria?
- Quando vengono definiti gli indirizzi delle variabili di un programma?



Swapping

- Permette di gestire più processi di quelli che fisicamente sarebbero caricabili in memoria
- Scheduler a medio termine





Allocazione statica vs dinamica

Binding degli indirizzi

Dobbiamo sapere come avviene il binding degli indirizzi per capire come SO può gestire la memoria

Spazio indirizzi logici e fisici

- Indirizzi generati da CPU sono indirizzi logici mentre un indirizzo per la memoria (l'indirizzo caricato nel registro MAR – Memory Address Register) indica un indirizzo fisico
- Indirizzi logici e fisici vengono associati in modo diverso
  - A tempo di compilazione
  - A tempo di caricamento
  - A tempo di esecuzione

Binding

Ogni processo risiede in memoria, ogni istruzione e dati sono individuate da indirizzi

Nel codice sorgente gli indirizzi sono simbolici

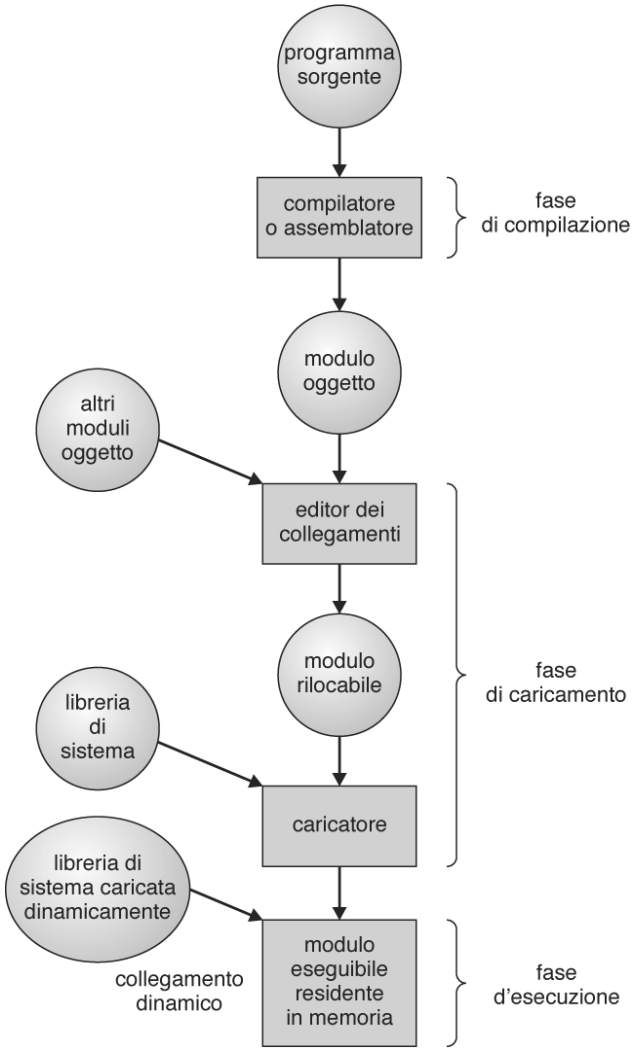
Binding: la fase in cui si associano istruzioni e dati a indirizzi di memoria

Il binding può avvenire in diversi momenti

- Durante la compilazione
- Durante il caricamento
- Durante l'esecuzione

L'indirizzo logico, o indirizzo virtuale, è l'indirizzo utilizzato da un programma in esecuzione per accedere alla memoria. Quando un programma richiede dati o istruzioni, utilizza un indirizzo logico.

Questo indirizzo viene quindi tradotto in un indirizzo fisico dalla CPU e dal sistema operativo, attraverso una serie di meccanismi di gestione della memoria.



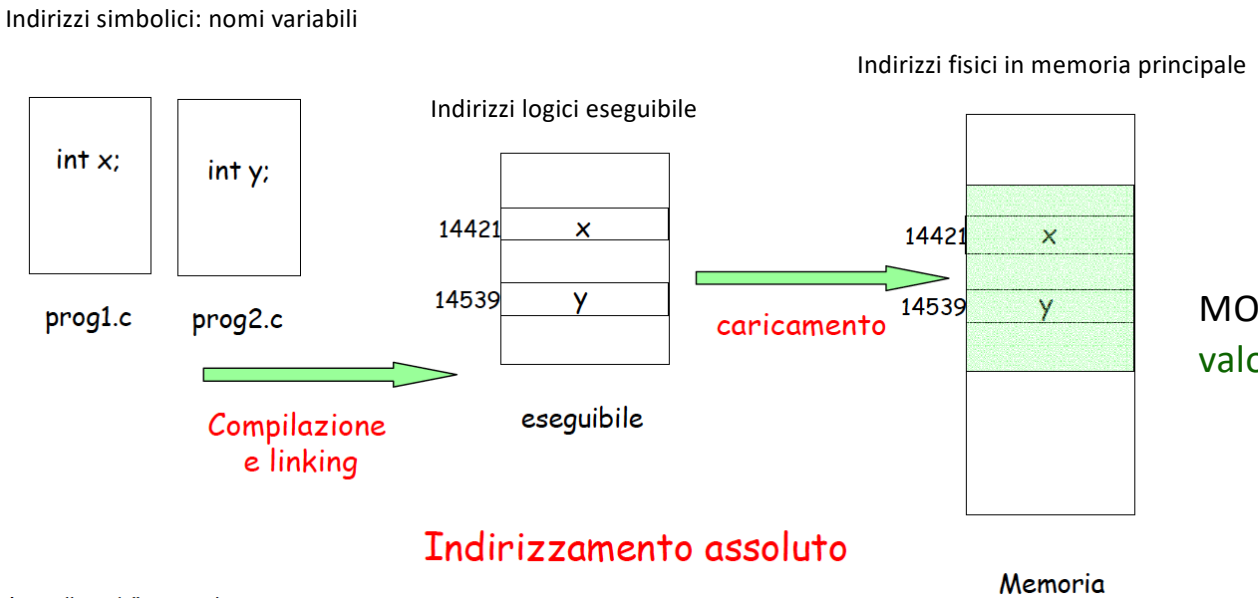
Compilazione: Il codice scritto in linguaggio di alto livello viene tradotto in codice linguaggio macchina

Linking: il codice delle librerie importate viene incluso nel modulo oggetto

Caricamento: il codice binario viene caricato in memoria principale

Esecuzione: il codice viene eseguito dalla CPU che invia richieste di lettura/scrittura alla memoria principale

Binding a tempo di compilazione



MOV 14421 AX ; memorizza il valore di AX all'indirizzo 14421

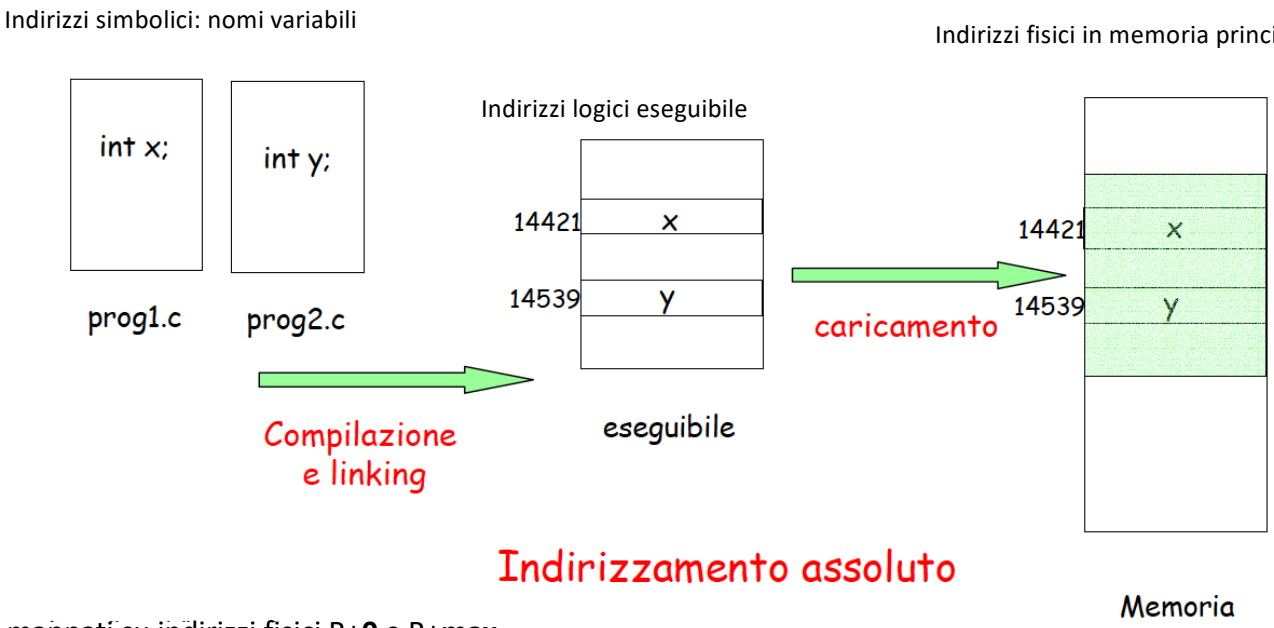
Allocazione statica: la posizione del processo in memoria non può cambiare.

Indirizzi logici = indirizzi fisici

Posizione in memoria processo nota a priori, codice assoluto

Semplice e veloce e non richiede hardware speciale ma… con la multiprogrammazione non funziona

Binding a tempo di caricamento (loading)



Il sistema operativo carica il programma in memoria e determina dove posizionarlo.

Gli indirizzi logici generati dal programma vengono modificati sulla base dell'indirizzo di base della memoria in cui il programma è caricato, utilizzando il registro di rilocazione.

Viene creato un mapping **statico** tra indirizzi logici e fisici. Se un programma è caricato all'indirizzo 14000, gli indirizzi logici saranno tradotti aggiungendo 14000 al valore dell'indirizzo logico.

Gli indirizzi logici saranno mappati su indirizzi fisici R+**0** e R+**max**

Ogni processo ha un proprio spazio di indirizzamento logico da **0** a **max** e fa riferimento a questo spazio di indirizzamento (non sa dove sarà effettivamente in memoria)

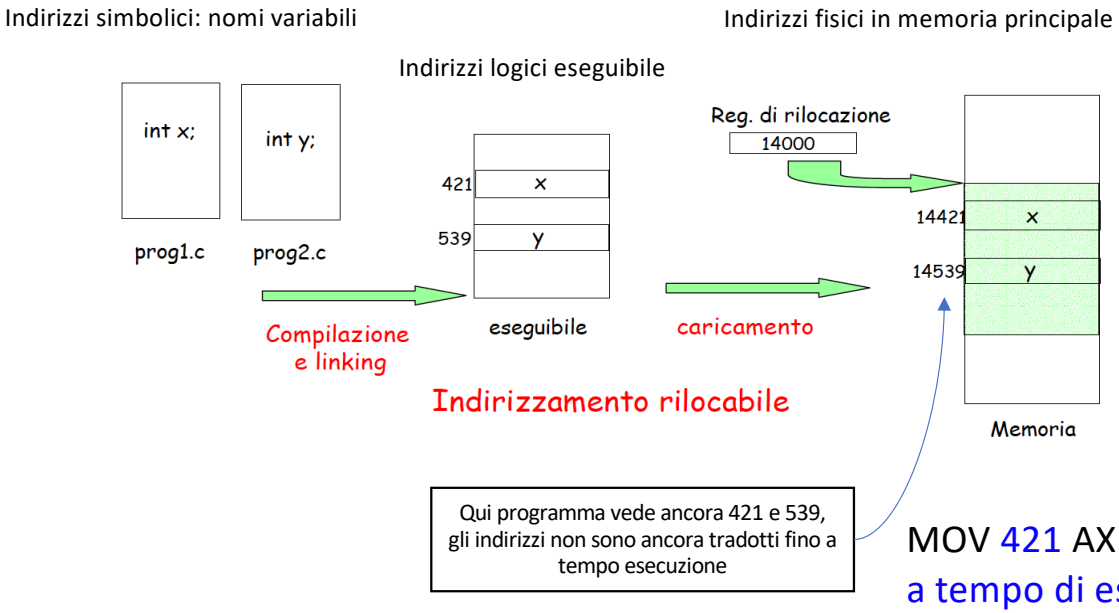
Indirizzi logici ancora uguali indirizzi fisici => indirizzo fisico = indirizzo logico + indirizzo base e.g., 14421 = 421 + 14000

Permette di gestire multiprogrammazione

MOV 14421 AX ;
memorizza il valore di AX
all'indirizzo 14421

Codice rilocabile: Spazio di indirizzi logici e fisici coincide ma se cambio posizione… devo ricaricare, rifare il mapping degli indirizzi…costoso

Binding a tempo di esecuzione (execution)



Anche quando caricato in memoria, il codice vede gli indirizzi logici da **0** a **max** mai quelli fisicia da **r+0** a **r+max**

Indirizzi logici diversi da indirizzi fisici

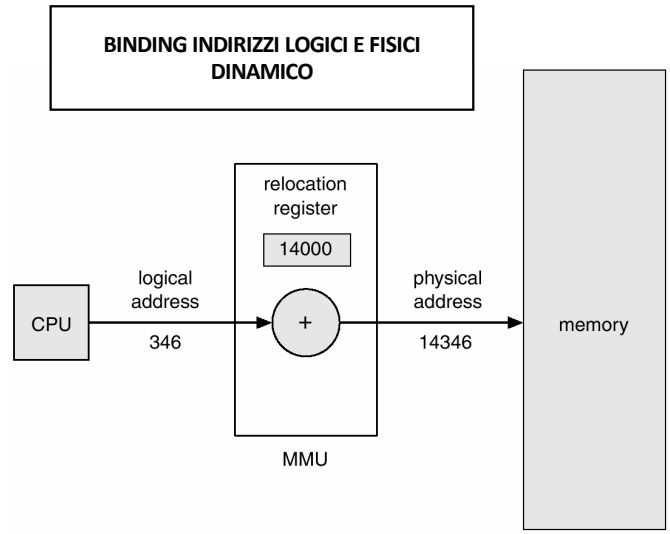
Codice rilocabile e spostabile durante l'esecuzione, il processo può essere spostato da una zona all'altra della memoria durante l'esecuzione

MOV 421 AX ; memorizza il valore di AX all'indirizzo 14421
a tempo di esecuzione

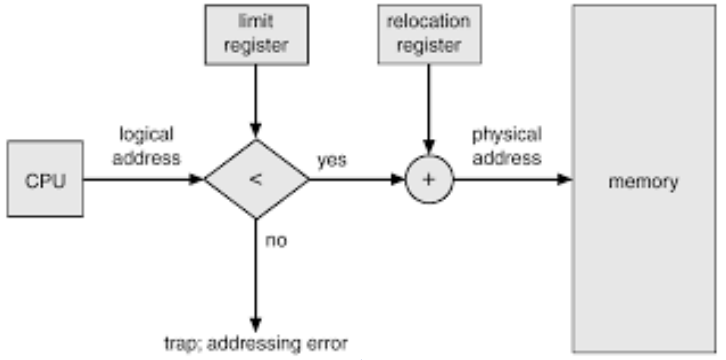
Hardware apposito (MMU - Memory Management Unit): CPU gestisce indirizzi logici/virtuali mentre MMU traduce indirizzi logici in fisici

MMU

- Con registro rilocazione



- Con registro rilocazione e limite



PROTEZIONE MEMORIA
Tra SO e processi utenti
Tra processi utenti e processi utenti

In entrambi i casi gestisco solo allocazione **contigua** della memoria
Ovvero il processo è in memoria in celle consecutive

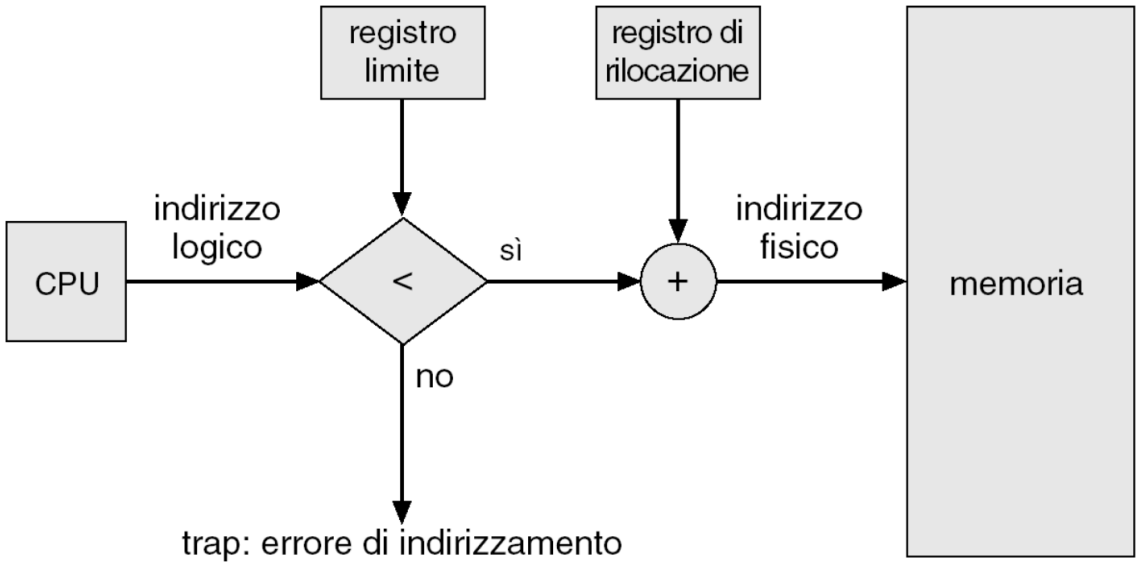
Rilocazione memoria

Il processo non può accedere a locazione di memoria prima di **base** e dopo **base+limite**

Protezione memoria SO e di altri processi utenti

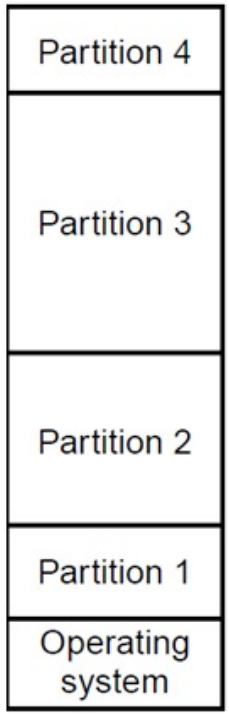
Il dispatcher sceglie uno dei processi nella coda dei pronti secondo l'algortimo di scheduling dei processi e carica I corrispondenti registri di rilocazione e limite (PCB del processo)

Ogni indirizzo prodotto viene confrontato con questi valori



Allocazione a partizioni Multiple Fisse – esempio rilocazione tempo esecuzione

- La memoria è divisa in un numero fisso di aree dette partizioni di dimensioni diverse
  - Ogni partizione è definita da una coppia base-limite
  - Il numero di partizioni definisce il grado di multiprogrammazione
- Bisogna sapere la dimensione del processo prima di caricarlo
- **Nel context switch SO carica**
  - **Nel registro rilocazione l'indirizzo base della partizione**
  - **Nel registro limite la dimensione del processo**



Swapping

- A seconda della fase in cui gli indirizzi logic vengono associati I processi possono essere spostati in zone di memoria diverse
- Fase di compilazione o caricamento: nello stesso punto
  - Rilocazione: se sposto in zona diversa, SO deve rifare il binding…costoso
- Fase di esecuzione: in posti diversi
  - Gli indirizzi vengono ricalcolati al momento dell'esecuzione

Approfondimento su codice assembly e indirizzamento

ISA

- L'**Instruction Set Architecture (ISA)** è un insieme di istruzioni di base che il processore è in grado di eseguire, costituendo il suo linguaggio macchina.

- **Compatibilità tra Microarchitetture**: Computer con microarchitetture diverse possono condividere lo stesso instruction set. Ad esempio, l'Intel Pentium e l'AMD Athlon implementano versioni quasi identiche dell'instruction set x86, pur avendo architetture interne molto diverse.
- **Definizione di ISA**: Un'ISA definisce l'insieme di tutti i codici binari (opcode) che rappresentano i comandi eseguibili nativamente da un particolare design di CPU.
- **Relazione tra Istruzioni**: Esiste un rapporto diretto tra istruzioni assembly, ISA e linguaggio macchina:
- 1 istruzione ad alto livello (e.g., in C) corrisponde a N istruzioni ISA, che a loro volta corrispondono a N istruzioni di linguaggio macchina.

Istruzione MOV assembly

- i trasferimenti di dati tra RAM e registri in Assembly sono effettuati utilizzando l'istruzione mov, che consente di leggere e scrivere dati a partire dagli indirizzi di memoria specificati.
- **Scopo**: Copiare dati da una sorgente a una destinazione.
- **Sintassi**: mov destinazione, sorgente
- **Esempi di sorgente**: Registri, memoria, costanti immediate.

Esempio*: mov al, [var] //Carica il valore della variabile var (dalla RAM) nel registro al.

**** Ci sono dei trasferimenti validi e non, non li vediamo (per esempio da memoria a memoria)***

Traduzione a Tempo di Compilazione

- Gli indirizzi sono risolti durante la compilazione del programma.
- L'istruzione diventa specifica per l'indirizzo di memoria assegnato.

```
section .data
   var db 5 ; var ha indirizzo 0x00400000

section .text
   mov al, [0x00400000] ; Indirizzo risolto a tempo di compilazione
```

Il compilatore conosce l'indirizzo di var e sostituisce [var] con 0x00400000. SEMPRE! Se ricompiliamo, potrebbe cambiare.

**** Ci sono dei trasferimenti validi e non, non li vediamo (per esempio da memoria a memoria)***

Traduzione a Tempo di Caricamento

- Gli indirizzi sono risolti quando il programma viene caricato in memoria.
- L’istruzione rimane generica fino al caricamento.

```
section .data
   var db 5 ; var è assegnato un indirizzo durante il caricamento

section .text
   mov al, [var] ; L'indirizzo effettivo viene risolto a tempo di caricamento
```

**** Ci sono dei trasferimenti validi e non, non li vediamo (per esempio da memoria a memoria)***

Traduzione a Tempo di Esecuzione

- Gli indirizzi sono risolti durante l'esecuzione del programma.

```
section .data
   var db 5 ; var può cambiare a runtime

section .text
   ; Supponiamo che si usi un puntatore
   mov bx, offset var ; Carica l'indirizzo di var in BX
   mov al, [bx]       ; Carica il valore di var in AL a tempo di esecuzione
```

**** Ci sono dei trasferimenti validi e non, non li vediamo (per esempio da memoria a memoria)***

Esempio – traduzione indirizzi fase di compilazione

```
int x = 10;  // Variabile x
int y = 4;   // Variabile y
int sum = x + y;  // Somma di x e y
```

Gli indirizzi delle variabili sono fissi e noti al momento della compilazione. Supponiamo che x, y, e sum siano allocati a indirizzi specifici.

Gli offset per x, y, e sum sono stati definiti esplicitamente nel codice (0x0000, 0x0002, e 0x0004).

Le istruzioni di caricamento e somma utilizzano direttamente questi offset fissi.

```
section .data
   myVar db 0            ; Riserva un byte per myVar all'indirizzo
0x00400000
   myConst db 10         ; Riserva un byte per myConst
all'indirizzo 0x00400001

section .text
   global _start

_start:
   mov al, [0x00400001]    ; Carica il valore di myConst (10) nel
registro AL
   add al, 5             ; AL ora contiene 15 (10 + 5)
   mov [0x00400000], al    ; Scrive il valore 15 nella variabile
myVar

   ; Uscita dal programma
   mov eax, 1            ; Chiamata di sistema per uscire
   xor ebx, ebx          ; Codice di uscita 0
   int 0x80              ; Interruzione per invocare il kernel
```

Esempio – traduzione indirizzi fase di caricamento

```
int x = 10;  // Variabile x
int y = 4;   // Variabile y
int sum = x + y;  // Somma di x e y
```

Gli indirizzi delle variabili non sono noti fino a quando il programma viene caricato in memoria. Gli offset sono simbolici e verranno risolti a tempo di caricamento.

Gli offset per x, y, e sum sono dichiarati, ma il compilatore non li risolve in valori assoluti.

Il **loader** si occupa di determinare gli indirizzi effettivi in memoria al momento del caricamento, sostituendo i riferimenti simbolici con gli offset appropriati.

```
section .data
   x dw 10          ; x all'indirizzo offset 0x0000
   y dw 4           ; y all'indirizzo offset 0x0002
   sum dw 0         ; sum all'indirizzo offset 0x0004

section .text
   org 0x100        ; Inizio del programma

_start:
   mov ax, [x]      ; Carica il valore di x (0x0000) in AX
   add ax, [y]      ; Somma il valore di y (0x0002) a AX
   mov [sum], ax    ; Salva il risultato in sum (0x0004)

   ; Uscita dal programma
   mov ax, 0x4C00
   int 0x21         ; Interruzione per terminare il programma
```