Il Sistema Operativo

Informatica B

Francesco Trovò

3 Dicembre 2021

francesco1.trovo@polimi.it

Introduzione al Sistema Operativo

Il Sistema Operativo

- Il Sistema Operativo (SO) è uno strato **software** che **nasconde** agli utenti i **dettagli dell'architettura hardware** del calcolatore
- Fornisce diverse **funzionalità** ad **alto livello** che facilitano **l'accesso alle risorse** del calcolatore
- Supporta **l'esecuzione dei programmi** applicativi **definendo una macchina virtuale**, cioè un modello ideale del calcolatore, sollevando il software applicativo dal compito di gestire i limiti delle risorse disponibili

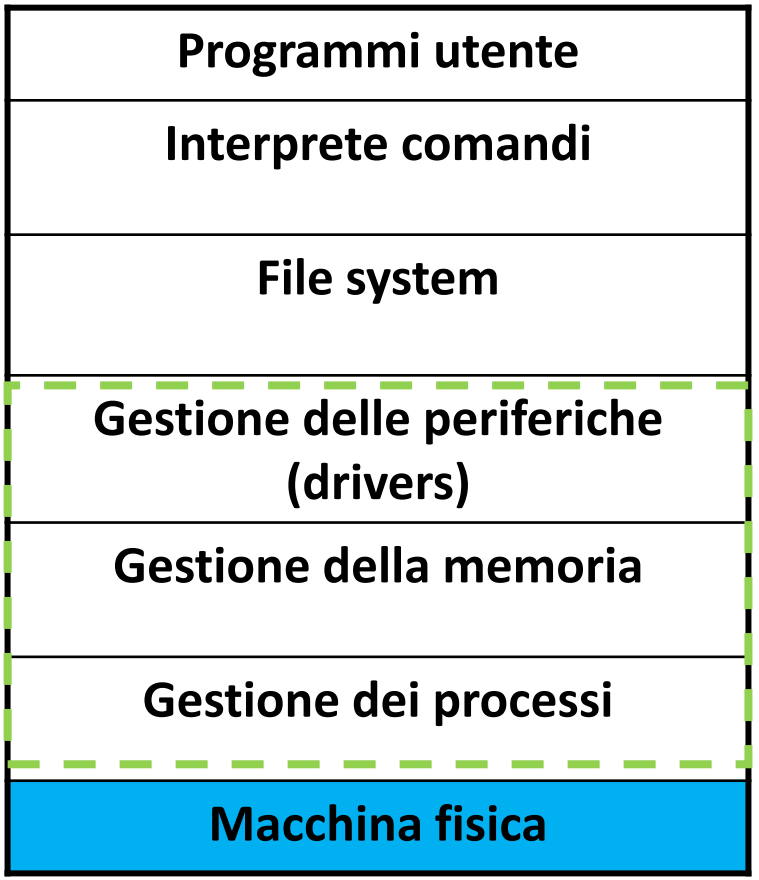
Tipi di Sistema Operativo

In generale, gli SO si possono dividere in:

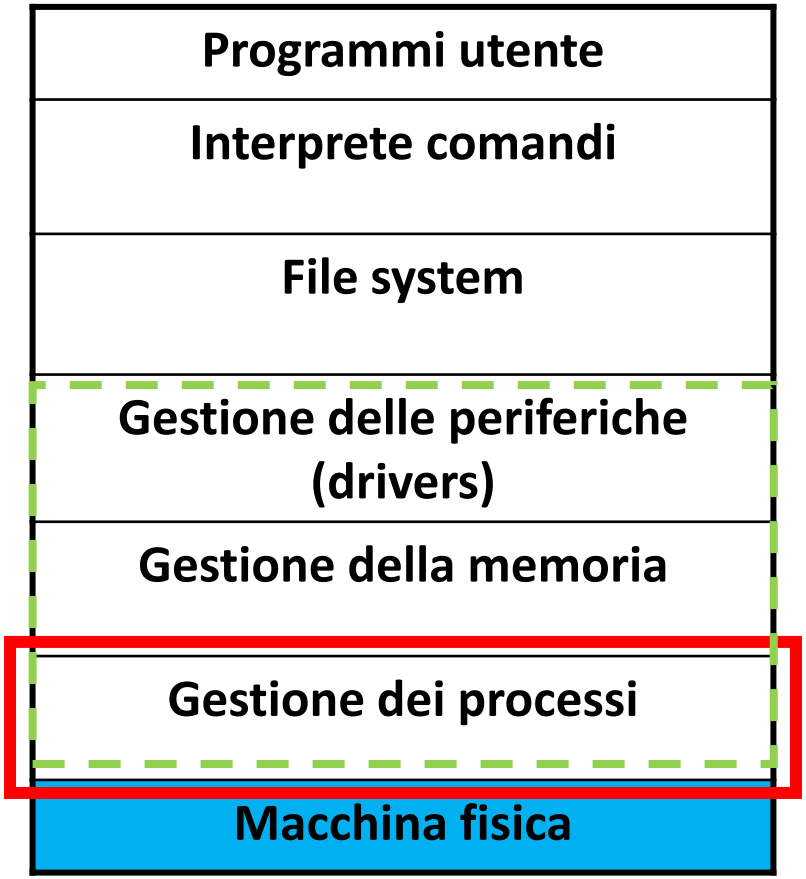
- **Monoutente** e **monoprogrammato**
  - Esecuzione un solo programma applicativo alla volta
  - Viene utilizzato da un solo utente per volta
  - Esempio: DOS
- **Monoutente** e **multiprogrammato** (**multitasking**)
  - Consente di eseguire contemporaneamente più programmi applicativi
  - Esempio: Windows 95
- **Multiutente**
  - Consente l'utilizzo contemporaneo da parte di più utenti
  - E' inerentemente multiprogrammato
  - Esempio: Linux e recenti versioni di Windows

Architettura del Sistema Operativo

- Il SO è tipicamente organizzato a **strati**
- Ciascun **strato** gestisce una risorsa del calcolatore
- Le principali funzionalità offerte sono:
  - La gestione dei processi
  - La gestione della memoria
  - La gestione delle periferiche
  - La gestione del file system
  - La gestione della rete
  - La gestione dell'interfaccia utente
- Le **prime tre** funzionalità sono indispensabili per il funzionamento del sistema e pertanto **costituiscono il nucleo del SO (Kernel)**



Gestione dei Processi

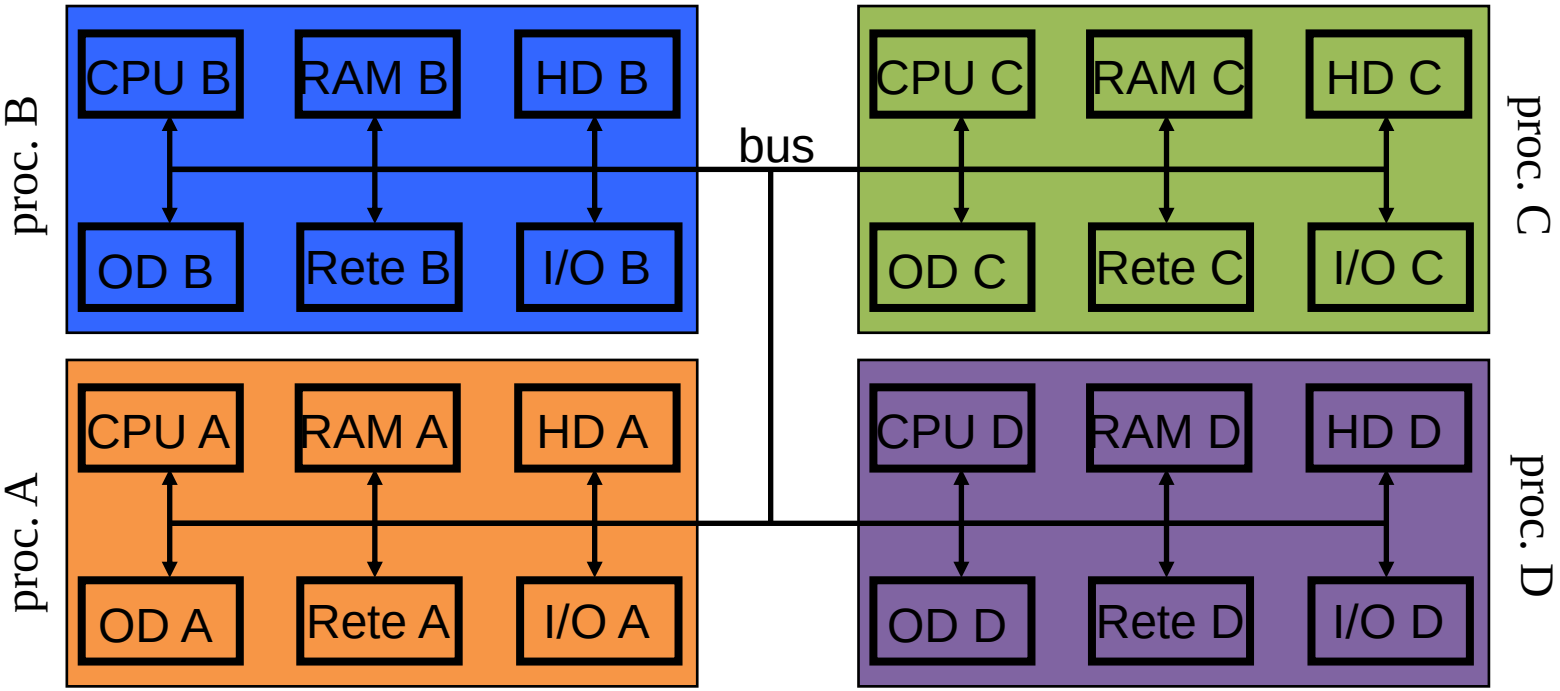


Kernel del SO: Gestione dei Processi

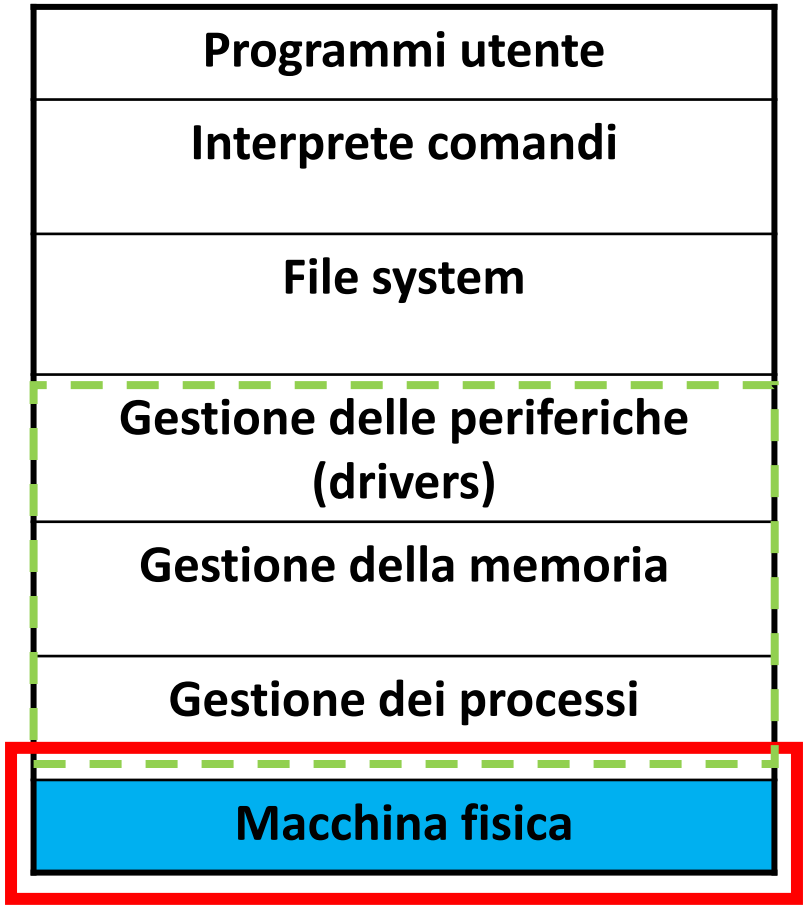
- Il SO si occupa di **gestire l'esecuzione** concorrente di più **programmi** (quantomeno nei sistemi multitasking)
- I programmi durante la loro esecuzione danno luogo a uno o più **processi**
- Il SO offre ad **ogni processo** una **macchina virtuale** interamente **dedicata** a esso
- Ogni processo opera come se fosse l'unico in esecuzione sulla macchina, avendo quindi disponibilità esclusiva delle risorse
- La il tempo di calcolo della **CPU** del calcolatore (o delle CPUs nei sistemi multiprocessore) **viene suddivisa** in maniera opportuna fra i programmi da eseguire

Le Macchine Virtuali

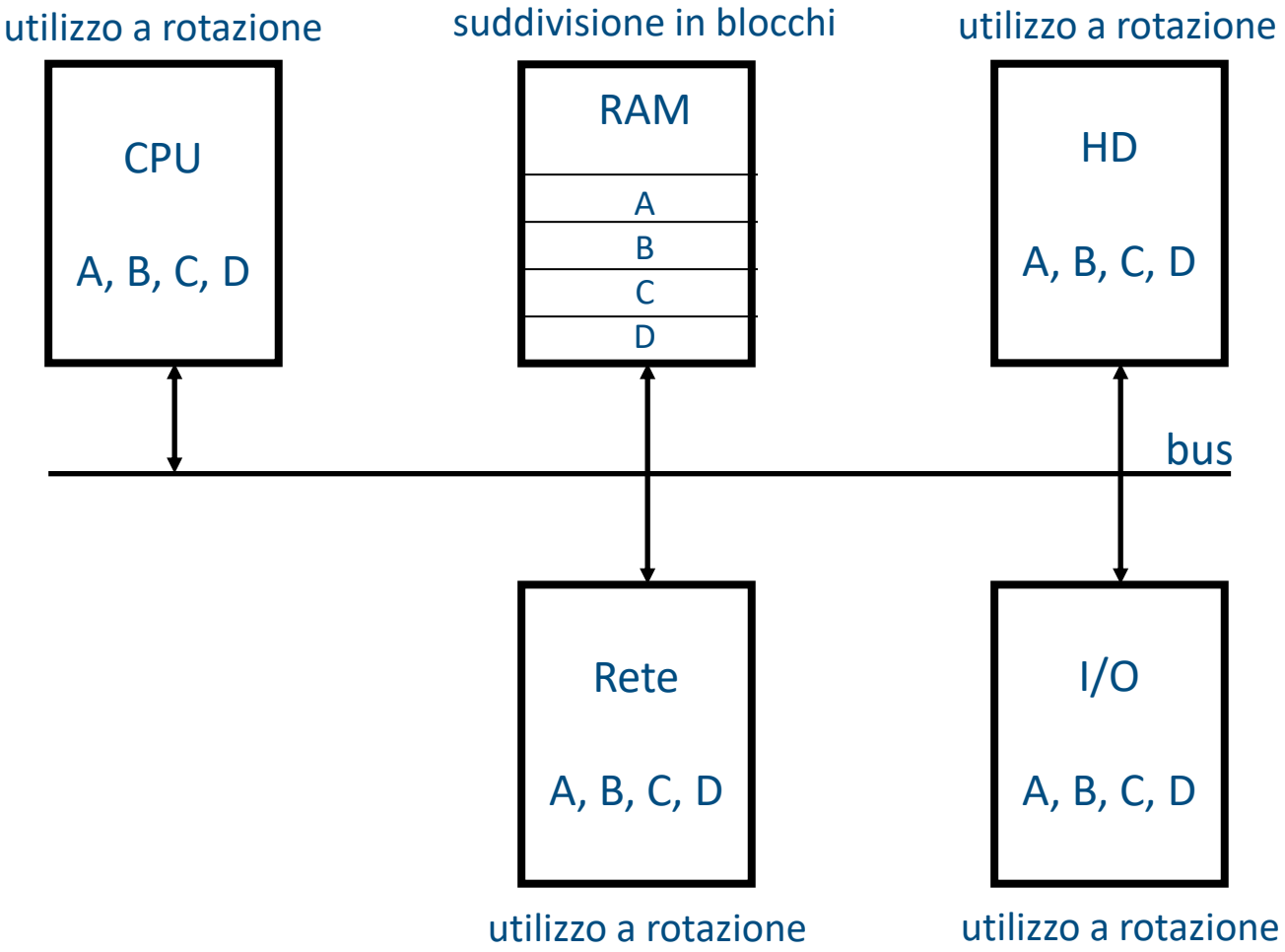
- Il SO permette di gestire **più processi simultaneamente**
- Rende quindi visibile ad ogni processo **una macchina virtuale** ad esso interamente dedicata e quindi con risorse proprie



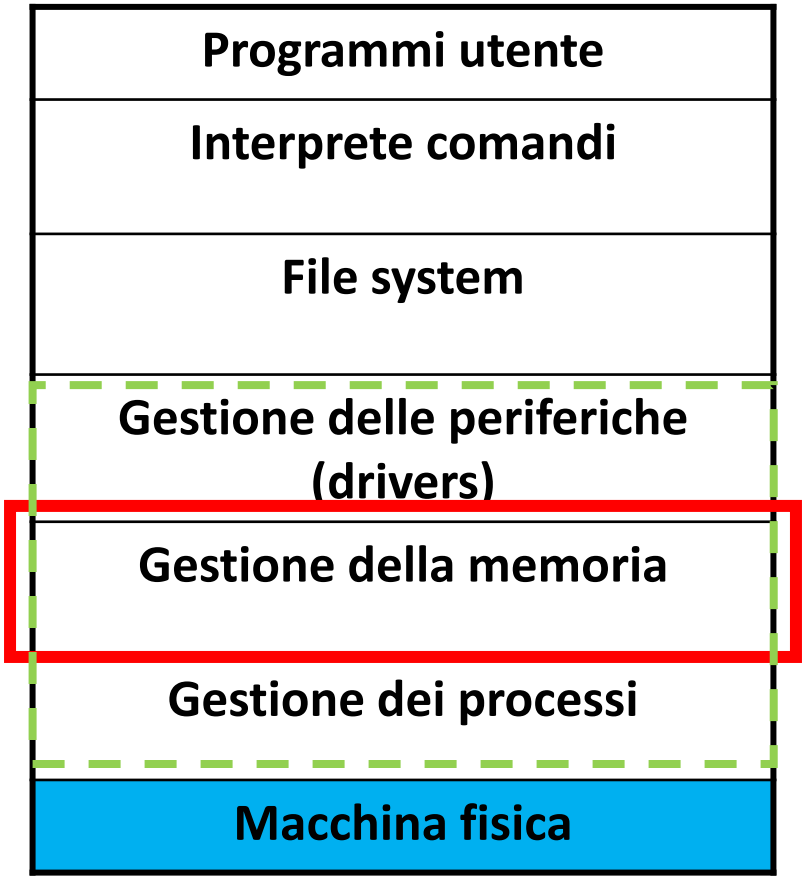
La Macchina Reale



Il Sistema Operativo e la Macchina Reale



Gestione della Memoria

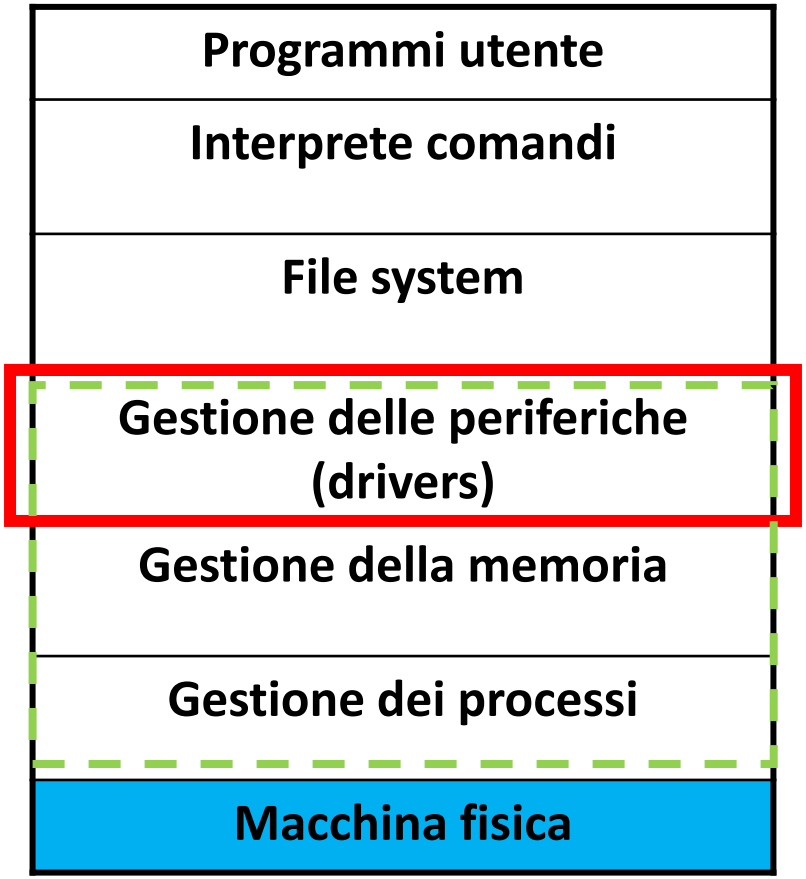


Gestione della Memoria

- La gestione concorrente di molti programmi applicativi comporta la **presenza di molti programmi in MM** (i programmi in esecuzione ed i dati ad essi relativi)
- Il **Gestore della memoria del SO** offre a ogni programma in esecuzione la visione di una **memoria virtuale**, che può avere dimensioni maggiori di quella fisica
- Per gestire la memoria virtuale e suddividerla tra i vari processi, il SO dispone di diversi meccanismi:
  - Rilocazione
  - Paginazione
  - Segmentazione

Vedremo nel dettaglio la Rilocazione e la Paginazione

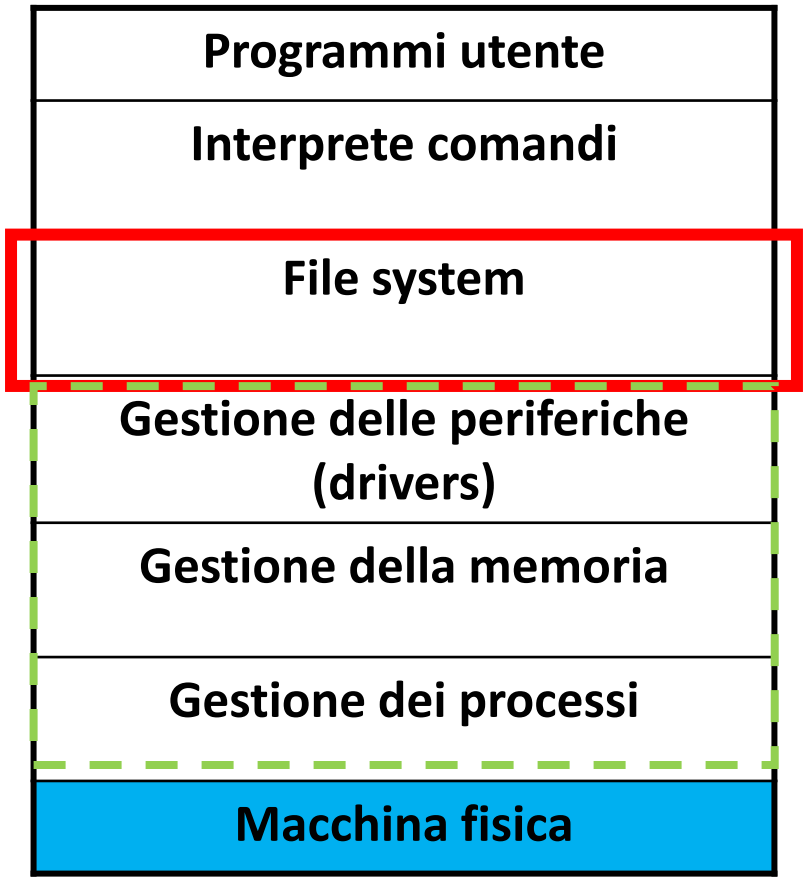
Gestione delle Periferiche



Gestione delle Periferiche: i Driver

- I **driver** sono meccanismi software a cui è affidato il compito **di trasferire dati da e verso le periferiche**
- Consentono ai programmi applicativi di leggere o scrivere i dati mediante **istruzioni di alto livello** che **nascondono la struttura fisica delle periferiche** (e.g., nel sistema Unix le periferiche sono viste come file speciali)
- Danno all'utente l'impressione che la periferica sia interamente dedicata all'utente

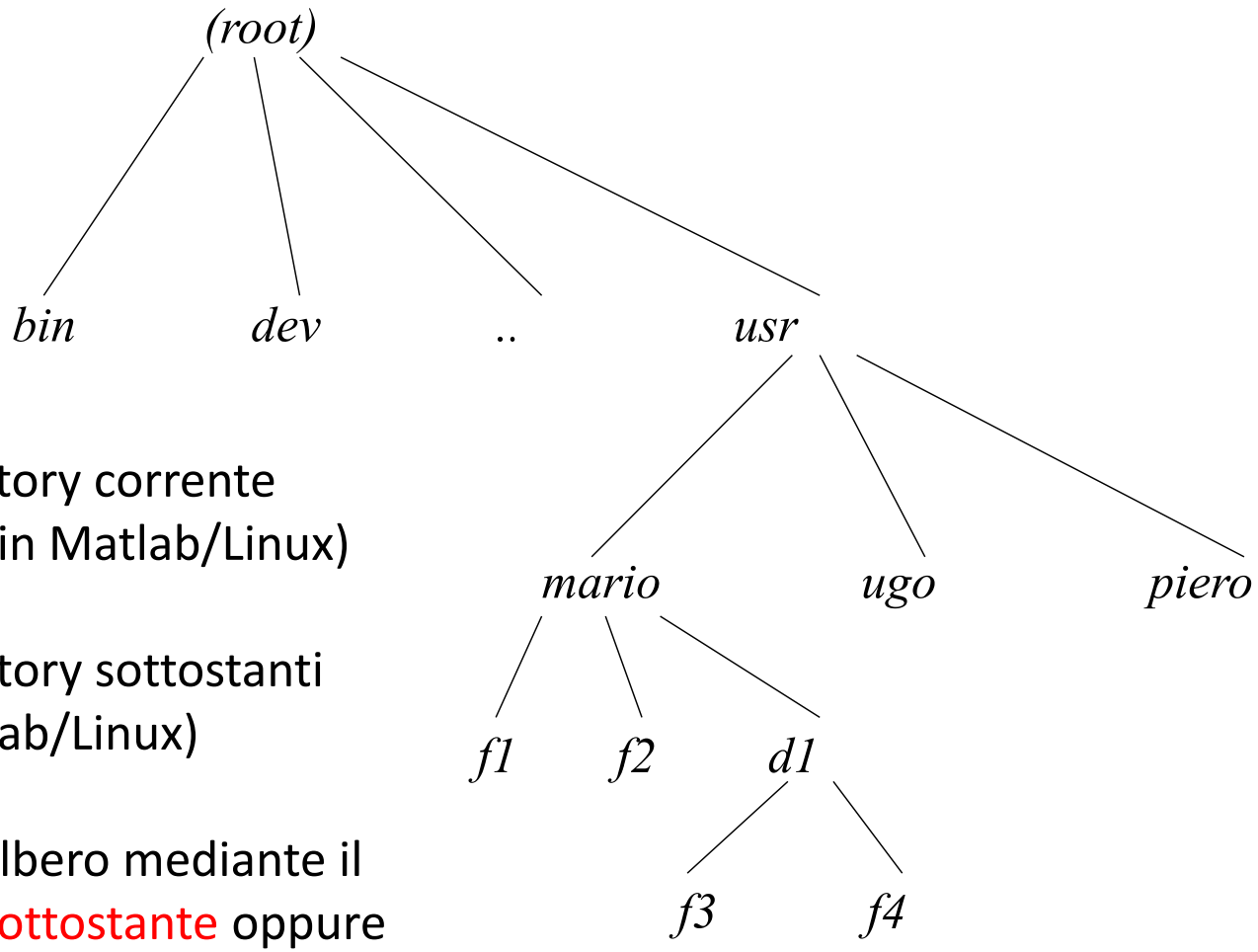
Gestione del File System



Gestione del File System

- Il SO si occupa di gestire i **file** sulla **memoria di massa**:
  - Creare/eliminare un file
  - Assegnare/modificare il nome
  - Collocarlo in una posizione nella memoria di massa
  - Accedervi in lettura e scrittura
- Il tutto mediante istruzioni di alto livello che permettono di ignorare i dettagli di come il file viene scritto
- Gestione dei file **indipendente dalle caratteristiche fisiche** della **memoria di massa**, ad esempio HD, SSD, USB drive
- I file vengono inclusi all'interno di *directory* (o *cartelle*)
- Tipicamente sono organizzati ad albero

La Struttura ad Albero



Per visualizzare la directory corrente digitare solo cd o pwd (in Matlab/Linux)

Per visualizzare le directory sottostanti digitare dir o ls (in Matlab/Linux)

Ci si può spostare nell'albero mediante il comando cd DirectorySottostante oppure cd .. per salire di un livello nell'albero

Organizzazione dei File

- A ciascun utente è normalmente associata una directory specifica, detta ***home directory***
- Il livello di ***protezione*** di un file indica quali operazioni possono essere eseguite da ciascun utente
- Ciascun file ha un ***pathname*** (o nome completo) che include l'intero cammino dalla radice dell'albero

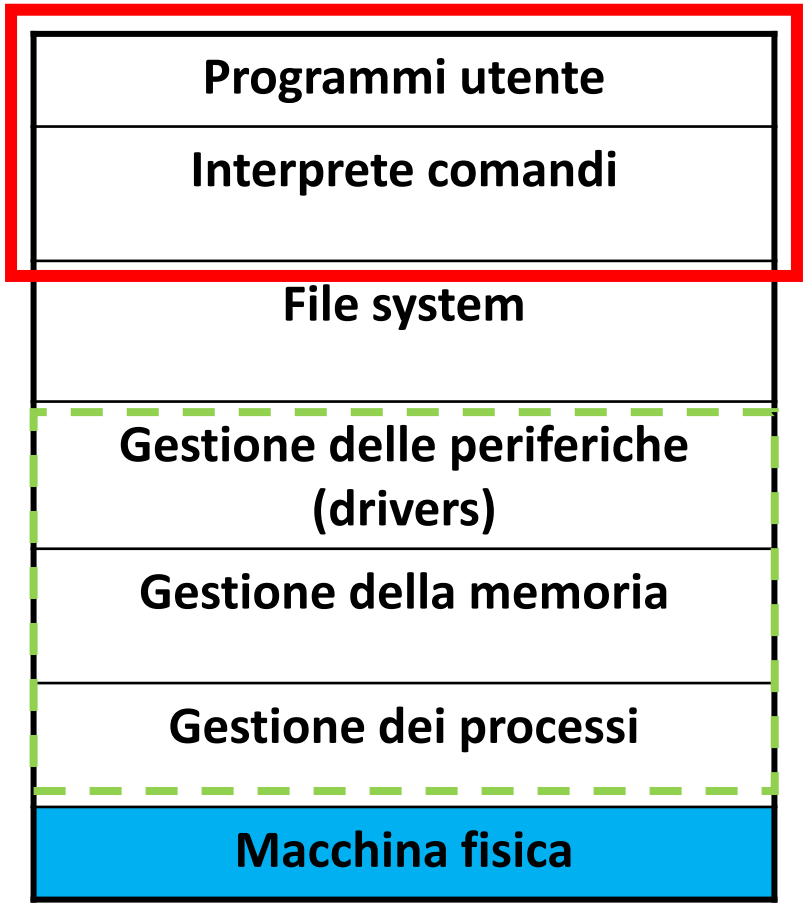
Es. C:\Windows

C:\Users\GiacomoBoracchi\Documents

C:\Users\GiacomoBoracchi\Documents\MATLAB

- Il ***contesto* di un utente** all'interno del file system è la directory in cui correntemente si trova

Gestione dell'Interfaccia Utente



Gestione dell'Interfaccia Utente

- Il SO fornisce un interprete dei comandi inseriti dall'utente attraverso la tastiera o il mouse
- L'interfaccia utente può essere
  - Testuale (esempio: DOS)
  - Grafica (esempio: Windows)
- Consente l'inserimento di diversi comandi:
  - Esecuzione di programmi applicativi
  - Operazioni sulle periferiche
  - Configurazione dei servizi del SO
  - Operazioni sul file system (creazione, rimozione, copia, ricerca)

Il Sistema Operativo ed i Processi

Che Cosa è un Processo?

- Processo ≠ programma
- Un processo viene generato durante l'**esecuzione di un programma**
- Un processo è composto da:
  - **codice eseguibile** (il programma stesso)
  - **dati dell'esecuzione** del programma
  - informazioni relative allo **stato** del processo
- Un processo è quindi un'entità **dinamica**, mentre il programma è un'entità **statica**
- I processi vengono eseguiti dal **processore**
  - uno alla volta nelle architetture a processore singolo
  - vedremo il Round-robin come politica di scheduling

Programmi e Processi

Lo stesso **programma** può avere **più processi associati**:

- Un **programma** può essere **scomposto** in varie parti e ognuna di esse può essere associata a un diverso processo
- Lo stesso **programma** può essere associato a diversi processi quando esso viene **eseguito più volte**, anche simultaneamente

I Processi e il Sistema Operativo

- Anche il **SO** è **implementato** tramite **processi**
- Il **SO** è **garante** che i **conflitti** tra i **processi** siano controllati e **gestiti** correttamente
- I processi possono essere eseguiti in due modalità:
  - **Kernel** (o **Supervisor**) **mode**: la CPU può eseguire qualsiasi istruzione, iniziare qualsiasi operazione I/O, accedere a qualsiasi area di memoria, etc.
  - **User mode**: certe istruzioni, che alterano lo stato globale della macchina, non sono permesse, e.g., operazioni I/O, accesso a certe aree di memoria, etc.
- Il SO viene **eseguito in modalità privilegiata** (kernel mode o supervisor), così da avere processi in grado di controllare altri processi eseguiti in modalità user

Chiamate al Supervisor

I processi utente per eseguire operazioni privilegiate come:

- accesso a file
- accesso a periferiche
- operazioni di I/O
- accesso ad altre risorse

invocano il supervisor tramite chiamate di sistema (System Call)

Chiamate al Supervisor (cnt)

Perché usare la **modalità supervisor** (i.e., privilegiata)?

- Un processo A non deve poter scrivere messaggi su un terminale non associato allo stesso processo A
- Un processo A non deve poter leggere caratteri immessi da un terminale non associato allo stesso processo A
- Un processo non deve poter sconfinare al di fuori del proprio spazio di memoria:
  - per non accedere allo spazio di memoria associato a un altro processo, modificando codice e dati di quest'ultimo
  - per non occupare tutta la memoria disponibile nel sistema, bloccandolo e rendendolo così inutilizzabile da altri processi
- La condivisione di risorse (dischi, CPU, etc.) deve essere tale da cautelare i dati di ogni utente

Lo Stato di un Processo

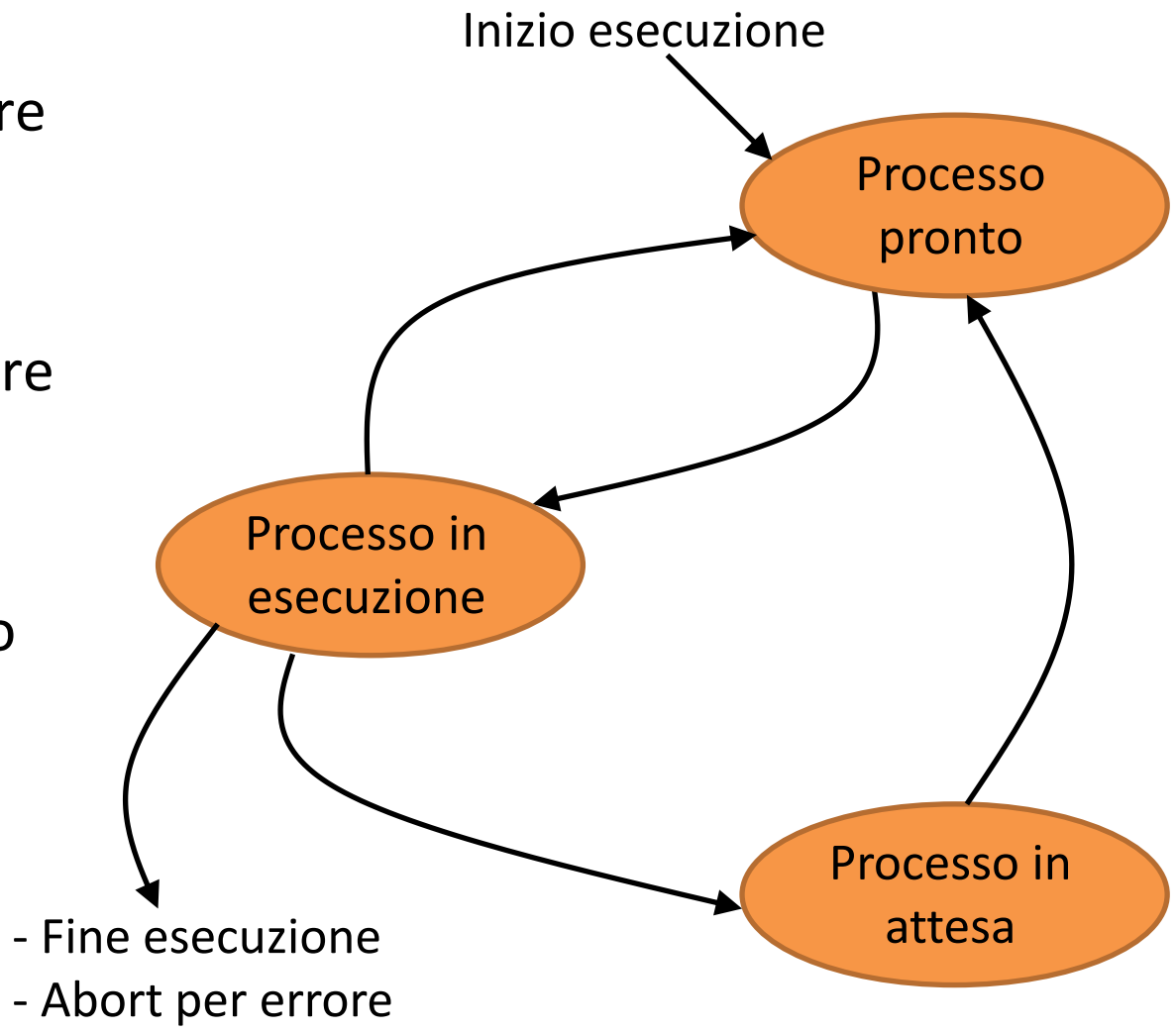
Lo stato del processo può essere distinto fra:

stato ***interno*** e stato ***esterno***

- Lo **stato interno** indica:
  - la **prossima istruzione del programma** che deve essere eseguita
  - i **valori delle variabili e dei registri** utilizzati dal processo
- Lo **stato esterno** indica se il processo è:
  - in **esecuzione** (il processore è assegnato al processo)
  - **pronto** per l'esecuzione, e quindi in attesa di accedere alla CPU, quando il SO lo deciderà
  - in **attesa** di un evento, ad es. la lettura da disco o l'inserimento di dati da tastiera

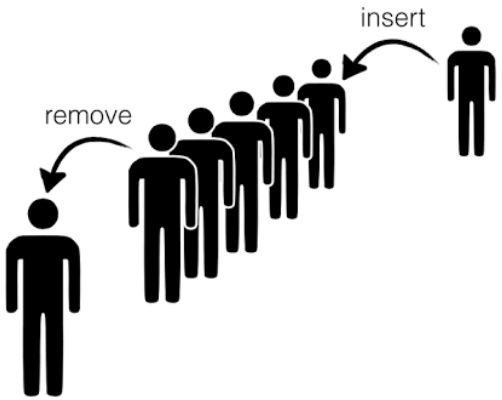
Transizioni tra Stati del Processo

- ***In esecuzione***: assegnato al processore ed eseguito da esso
- ***Pronto***: può andare in esecuzione, se il gestore dei processi lo decide
- ***In attesa***: attende il verificarsi di un evento esterno per andare in stato di *pronto*



Algoritmi di Scheduling: Approccio FIFO

- La schedulazione avviene in modo semplice
  - Il primo processo viene mandato in esecuzione
  - Quando termina è il turno del secondo processo



- Vantaggio: semplice da realizzare
- Svantaggi: inefficiente e inutilizzabile su un sistema interattivo (viene usato nelle workstation per il calcolo scientifico)

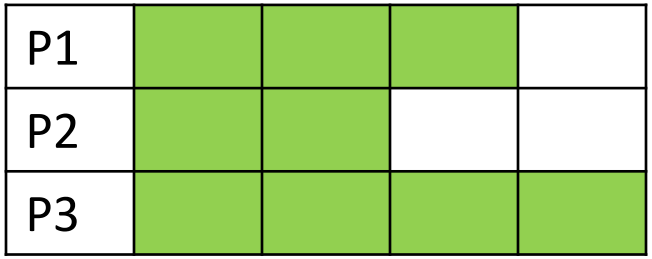
Algoritmi di Scheduling: Approccio Ideale

- Il minimo tempo medio di attesa si ottiene eseguendo prima i processi la cui esecuzione è più rapida
- Il problema è che non si conosce *a priori* quanto un programma rimarrà in esecuzione
- Si corre il rischio di non vedere eseguiti i processi troppo lunghi

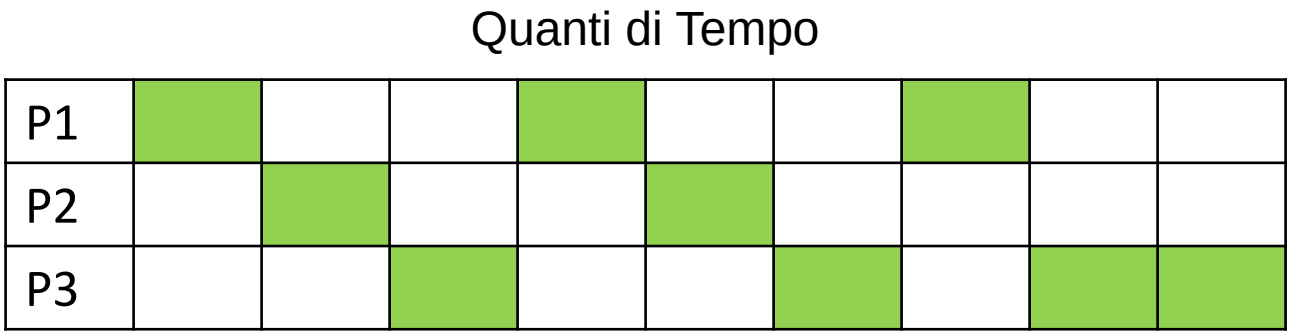
Algoritmi di Scheduling: Round Robin

- Suddivide la CPU tra i vari processi in base ad un **quanto di tempo** scandito dal clock di sistema

Es. Tre processi P1, P2, P3 con tempo di esecuzione:

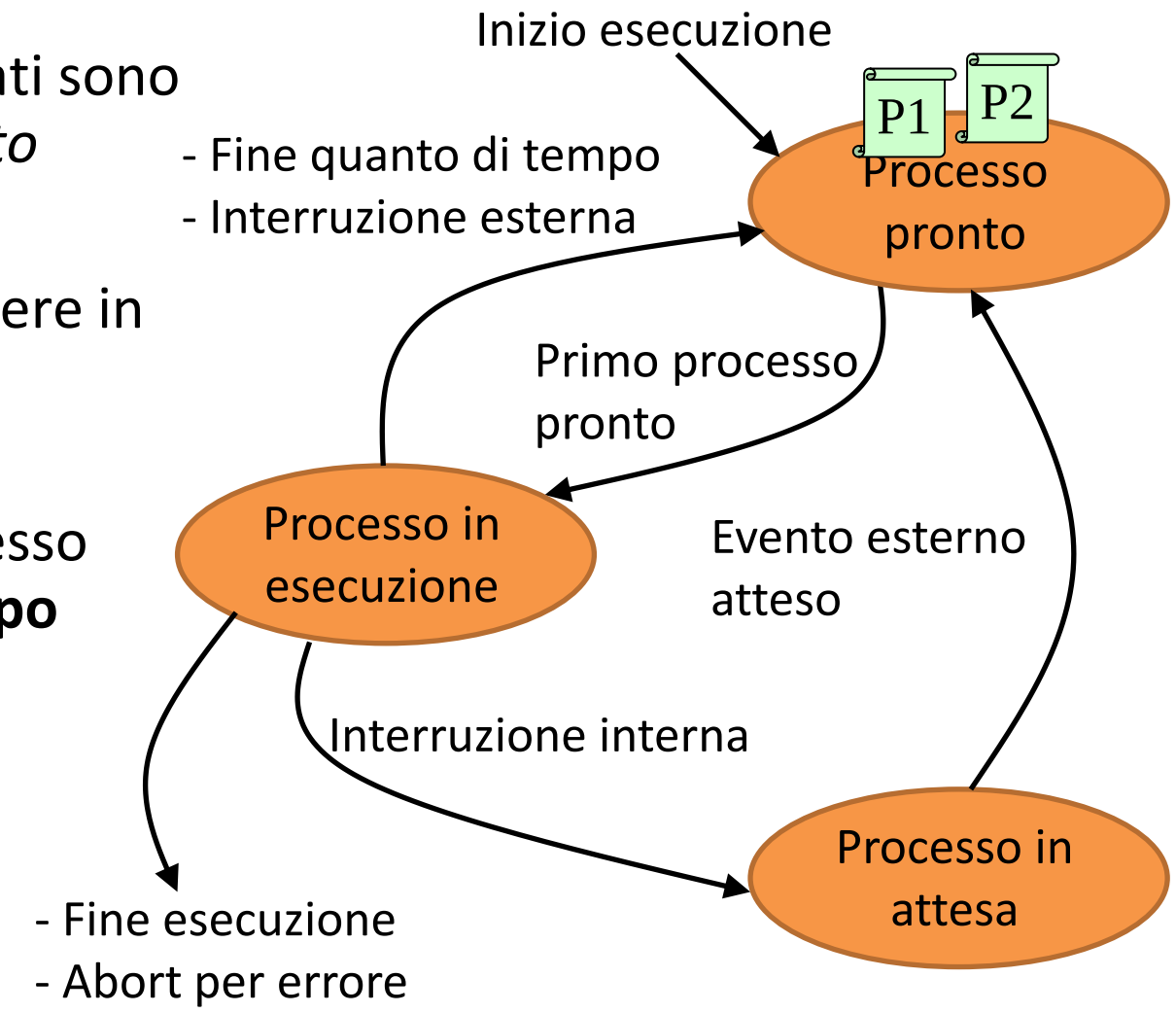


Scheduling con round robin:



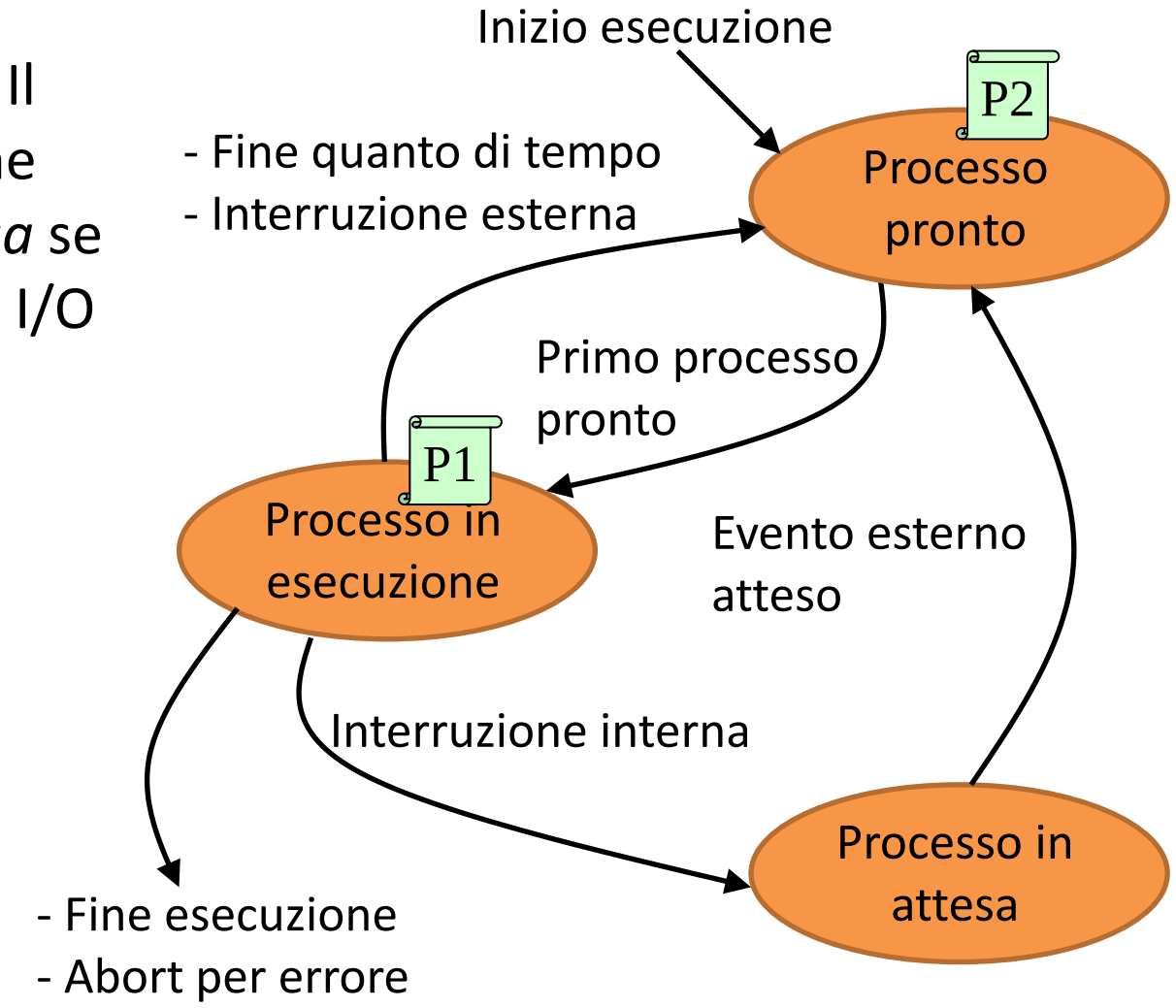
Esempio: Round Robin (1)

- I processi appena creati sono messi in stato di *pronto*
- Il **kernel** decide quale processo **pronto** mettere in stato **di esecuzione**
- Il **kernel** assegna il processore a un processo per un **quanto di tempo**



Esempio: Round Robin (2)

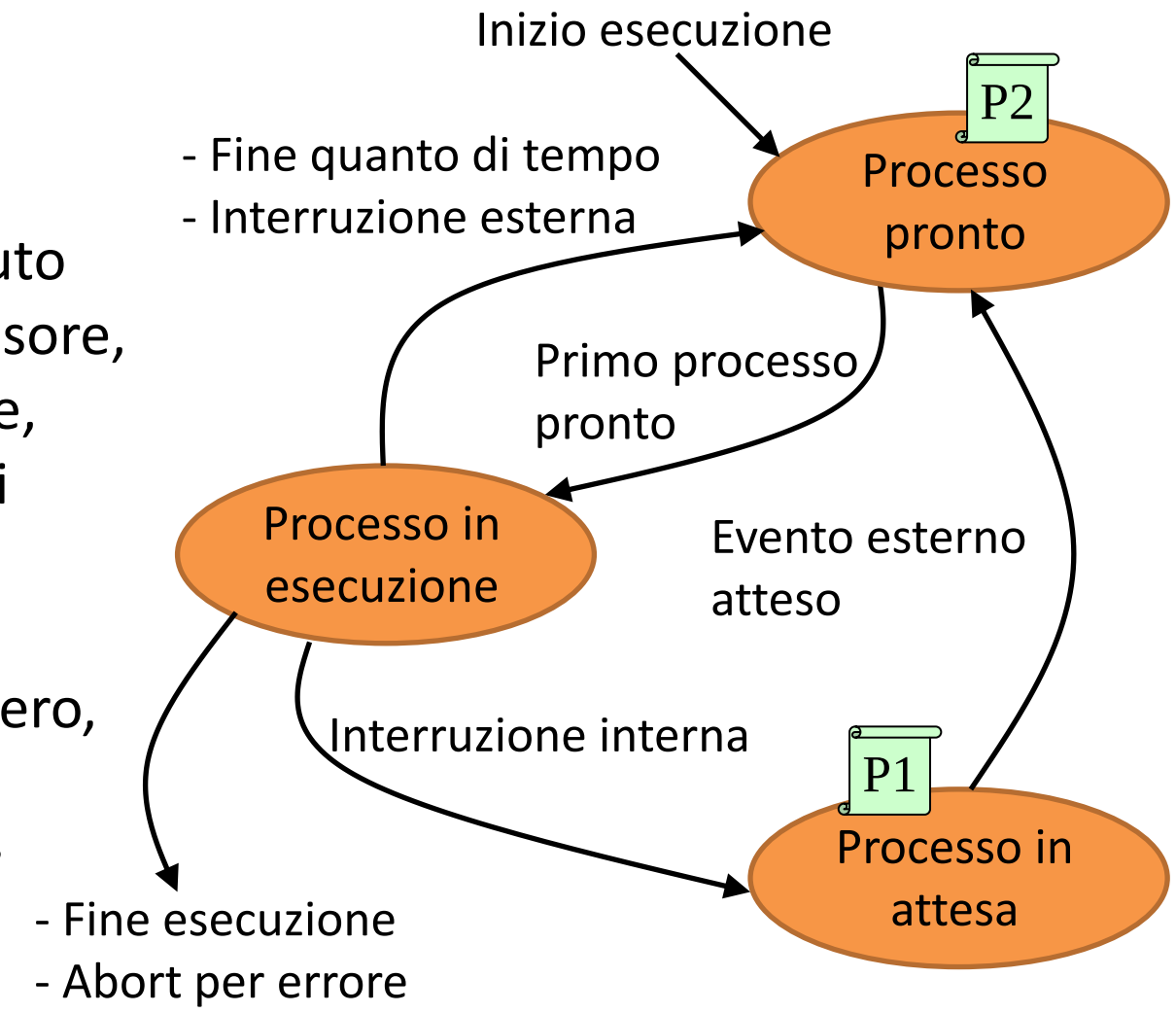
- **Interruzione interna**: Il processo in esecuzione passa in stato di *attesa* se richiede operazioni di I/O



Esempio: Round Robin (3)

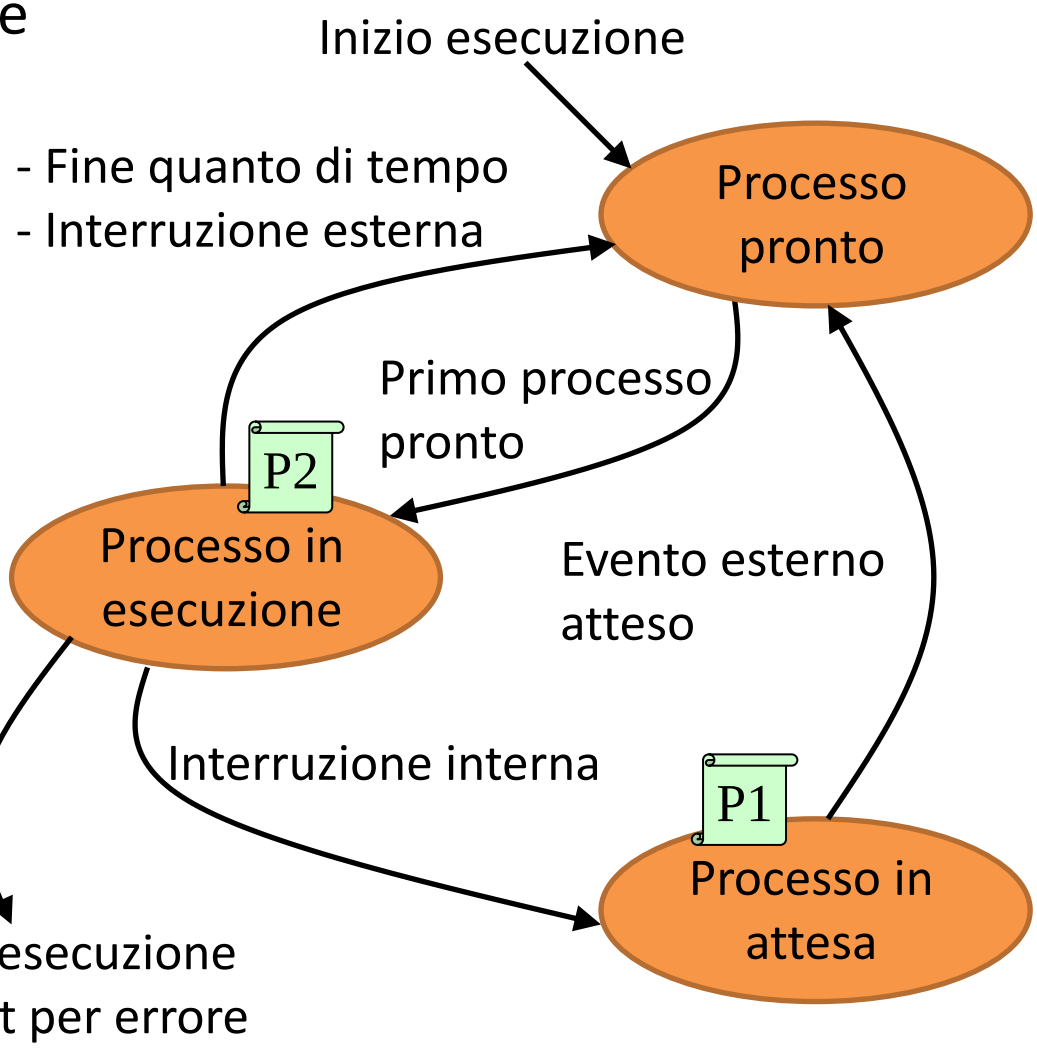
- **Cambiamento di contesto**:
  salva il *contesto* di P1, copia il contenuto dei registri del processore, il codice in esecuzione, etc. nel descrittore di processo

- Il processore è ora libero, un altro processo passerà in *esecuzione*



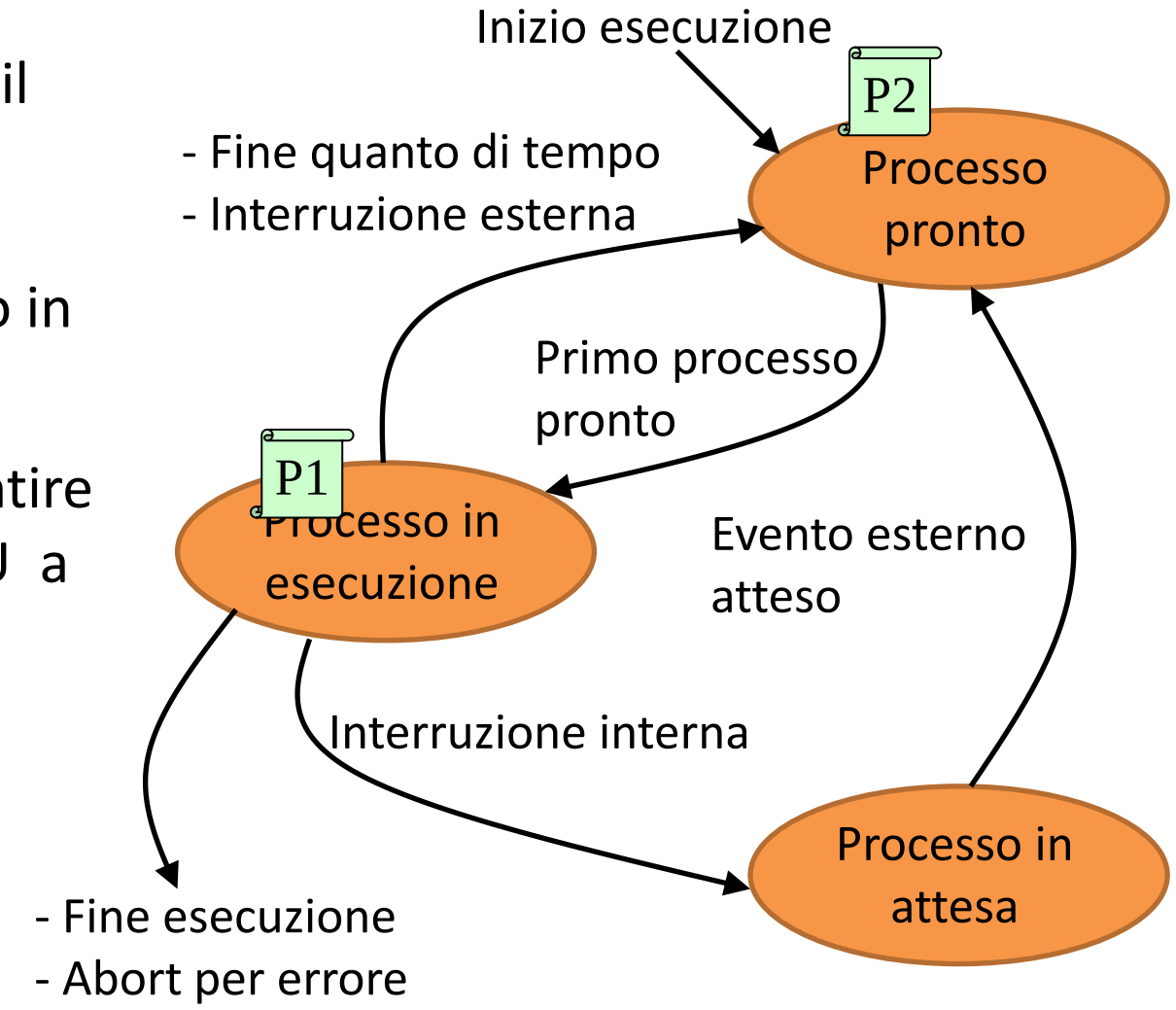
Esempio: Round Robin (4)

- Quando l'operazione di I/O è finita viene generata un'***interruzione esterna***
- Il processo in esecuzione viene **interrotto**
- Il kernel esegue il ***gestore delle interruzioni*** che esegue le azioni opportune
  - load dati da periferiche
  - **Riporta P1 *pronto***
- Il kernel sceglie quale processo mandare in esecuzione



Esempio: Round Robin (5)

- ***Pre-emption***: quando il quanto di tempo è scaduto, il nucleo interrompe il processo in esecuzione
- Il kernel cerca di garantire un uso equo della CPU a tutti i processi



La gestione del Quanto di Tempo

Il quanto di tempo è gestito da una particolare interruzione, generata dall'orologio di sistema:

- **a una frequenza definita**, il dispositivo che realizza **l'orologio** di sistema genera **un'interruzione**
- se **il quanto di tempo non è scaduto quando il processo in esecuzione termina,** il processore prosegue nell'esecuzione

Se invece il **quanto di tempo è scaduto** viene invocata una particolare funzione del kernel (**preemption**) che:

1. cambia lo stato del processo da esecuzione a pronto
2. salva il contesto del processo
3. attiva una particolare funzione del kernel (**change**) che esegue una commutazione di contesto e manda in esecuzione uno dei processi in stato di pronto

Algoritmi di Scheduling: Round Robin con Priorità

- Su linux si fa un round robin con diverse code di attesa in base alla priorità del processo

Es. L'attesa con priorità al pronto soccorso rispetto al codice (verde, giallo, rosso)

Riepilogando

Un processo in esecuzione può:

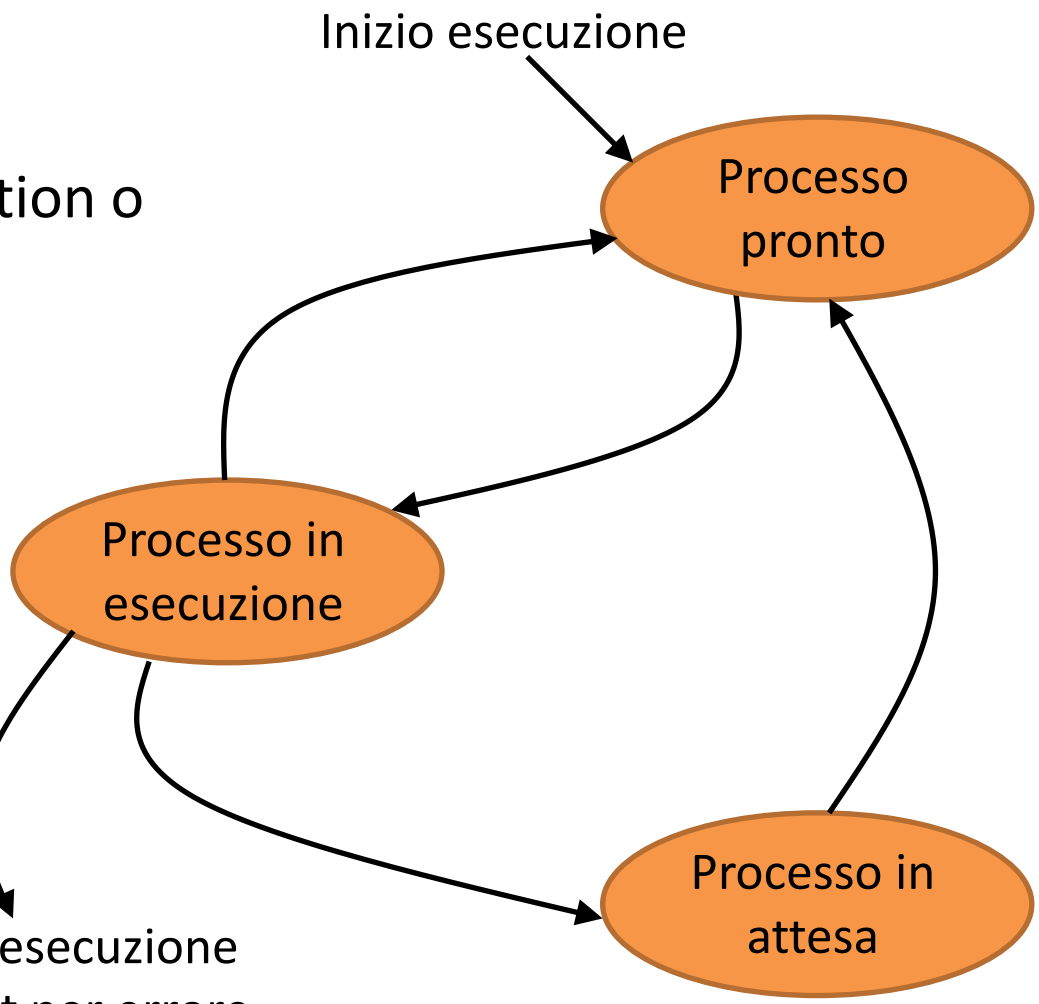
- terminare
- tornare in pronto (preemption o interruzione esterna)
- andare in attesa (SVC)

Un processo pronto può:

- andare in esecuzione (per andare in attesa deve fare una SVC, quindi essere eseguito)

Un processo in attesa può

- andare in pronto (no esecuzione subito)



Esercizio

- I seguenti processi in coda con relativa durata in millisecondi, il quanto di tempo dura di 20 ms:

  p_1 30 ms, p_2 15 ms, p_3 60 ms, p_4 45 ms

  - Assumendo che nessuno di questi richieda una system call e che i processi in stato di pronto vengano gestiti con una FIFO (first in first out), visualizzare la sequenza di esecuzione secondo uno schema Round Robin specificando
    - il tempo totale impiegato
    - il numero di context switch effettuati
    - il tempo medio utilizzato per processo

Esempio di esecuzione processi in Round Robin

- I seguenti processi in coda con relativa durata in millisecondi, il quanto di tempo dura di 20 ms:

  p_1 30 ms, p_2 15 ms, p_3 60 ms, p_4 45 ms

  Verranno eseguiti nel seguente ordine:

		Tempo dall'inizio
1.	p_1 (interrotto dopo 20 ms, ne rimangono altri 10 ms)	0
2.	p_2 (termina dopo 15 perché dura meno di 20 ms)	20
3.	p_3 (interrotto dopo 20 ms, ne rimangono altri 40 ms)	35
4.	p_4 (interrotto dopo 20 ms, ne rimangono altri 25 ms)	55
5.	p_1 (termina dopo 10, necessitava di meno di 20 ms)	75
6.	p_3 (interrotto dopo 20 ms, ne rimangono altri 20 ms)	85
7.	p_4 (interrotto dopo 20 ms, ne rimangono altri 5 ms)	105
8.	p_3 (termina dopo 20 ms, necessitava di 20 ms)	125
9.	p_4 (termina dopo 5 ms, necessitava meno di 20 ms)	145
		150

5 Contex switches

- Il tempo medio «perso» è la media dei tempi in cui ciascun processo finisce e la sua durata effettiva:

  [(85-30) + (35-15) + (145-60) + (150-45)]/4 = 66,25 ms

Esercizio

- Siano P e Q due processi lanciati su un sistema monoprocessore
- P contiene una **`scanf()`**, mentre Q non comporta alcuna chiamata al supervisor
- Dire se ciascuna delle seguenti affermazioni é vera o falsa, giustificando le risposte

  «Il processo P potrebbe terminare senza mai essere mai essere nello stato *in attesa*»

FALSO: dal momento che contiene una **`scanf()`** dovrà necessariamente effettuata una supervisor call e il suo stato diverrà “in attesa”

Esercizio

- Siano P e Q due processi lanciati su un sistema monoprocessore
- P contiene una **`scanf()`**, mentre Q non comporta alcuna chiamata al supervisor
- Dire se ciascuna delle seguenti affermazioni é vera o falsa, giustificando le risposte

«Se il processo Q viene lanciato prima di P allora Q termina sicuramente prima di P»

FALSO: non è possibile sapere quale processo terminerà prima a priori dal momento che ad ogni processo è garantito un solo quanto di tempo alla volta

Esercizio

- Siano P e Q due processi lanciati su un sistema monoprocessor
- P contiene una **`scanf()`**, mentre Q non comporta alcuna chiamata al supervisor
- Dire se ciascuna delle seguenti affermazioni é vera o falsa, giustificando le risposte

«Una volta lanciato Q rimarrà sempre nello stato *in esecuzione*»

FALSO: non è possibile affermarlo con certezza: se Q dovesse terminare prima dello scadere del quanto di tempo allora rimarrà sempre nello stato “in esecuzione”, viceversa sarà posto nello stato di “pronto”. Potrebbe inoltre subentrare l’interruzione esterna

Esercizio

- Si consideri un sistema monoprocessore con i 4 processi in stato di pronto:

  p_1 75ns, p_2 40ns, p_3 20ns, p_4 40ns
- La CPU adotta una politica di tipo Round Robin con quanto di tempo 30ns, nessun processo esegue chiamate al supervisor
- Si assuma che i processi siano in ordinati in una FIFO da p_1 a p_4, tuttavia p_3 e p_4 hanno priorità alta, mentre p_1 e p_2 hanno priorità normale e a parità di priorità viene mandato in esecuzione il processo avanti nella FIFO

Si descriva la sequenza di esecuzione riportando:

- dopo quanti ns termina ogni processo a partire dall'inizio
- quanti context switch richiede l'intero processo
- il tempo di attesa medio per ogni processo

Soluzione

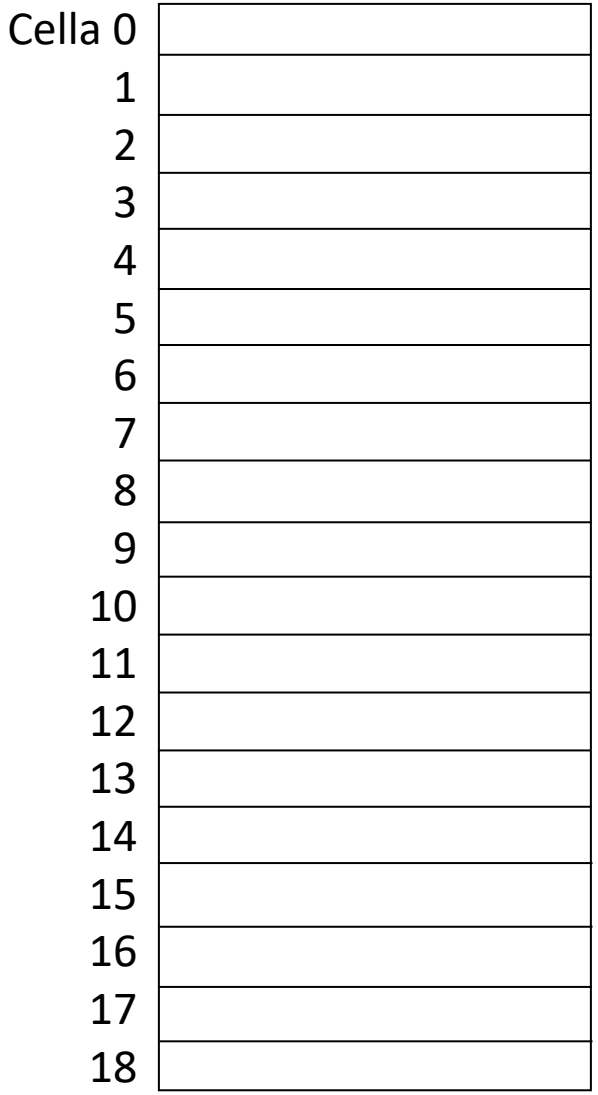
Processo in Esecuzione	Durata Esecuzione	Tempo trascorso	Termina/CS
p_3	20	20	Termina
p_4	30	50	CS
p_4	10	60	Termina
p_1	30	90	CS
p_2	30	120	CS
p_1	30	150	CS
p_2	10	160	T
p_1	15	175	T

Tempo medio attesa: $\frac{1}{n}\sum_i^n(T_i - D_i)$ dove D_i è la durata prevista del processo P_i, T_i è l'istante in cui termina P_i
ovvero ((175 - 75) + (160 -40) + (60 -40) + (20 - 20)) / 4 = 80ns

Il Sistema Operativo e la Gestione della Memoria

Gestione della Memoria

- E’ un **modello lineare**
- La memoria è una **sequenza di parole** (celle) numerate da 0 fino a un valore massimo M
- Il numero che identifica ogni cella è detto **indirizzo**
- La dimensione della cella dipende dal tipo di calcolatore (per noi sarà di 8 bit, ossia un byte)
- E’ una risorsa da **suddividere** tra i vari programmi, come la CPU



Spazio di Indirizzamento

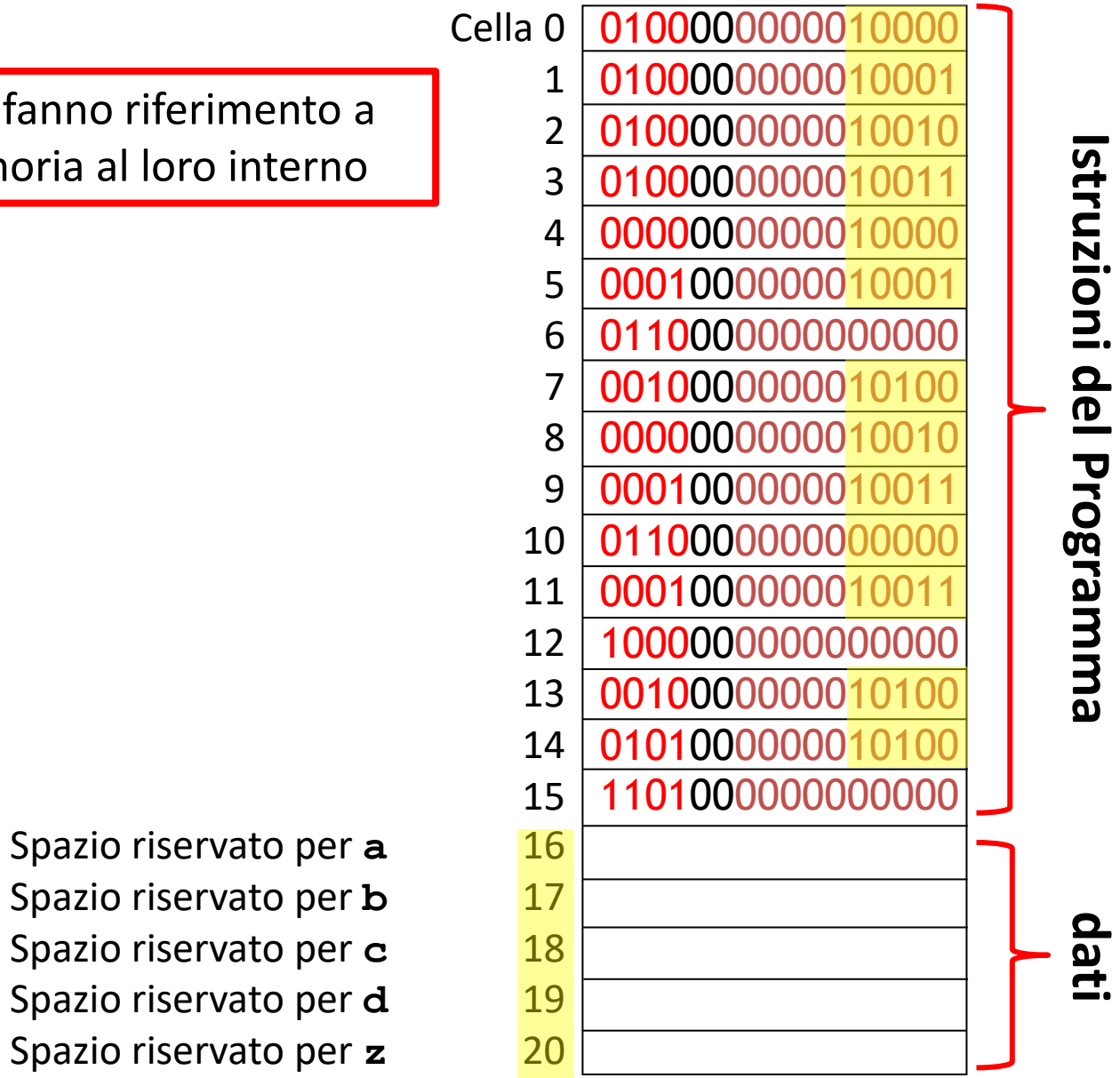
- Lo spazio di indirizzamento è **il numero massimo** di **indirizzi** possibili della memoria
- **Dipende** dalla lunghezza in **bit degli indirizzi**
- Se gli **indirizzi** sono **lunghi N bit**, lo spazio di indirizzamento è di **$M = 2^N$ celle**
- Tutte le celle devono essere indirizzabili, quindi

  Dimensione Memoria Utilizzabile ≤ Spazio Indirizzamento

- Le dimensioni della memoria sono generalmente espresse in:
  - KB (Kilobyte) = 2^{10} byte
  - MB (Megabyte) = 2^{20} byte
  - GB (Gigabyte) = 2^{30} byte

La Memoria nei Programmi

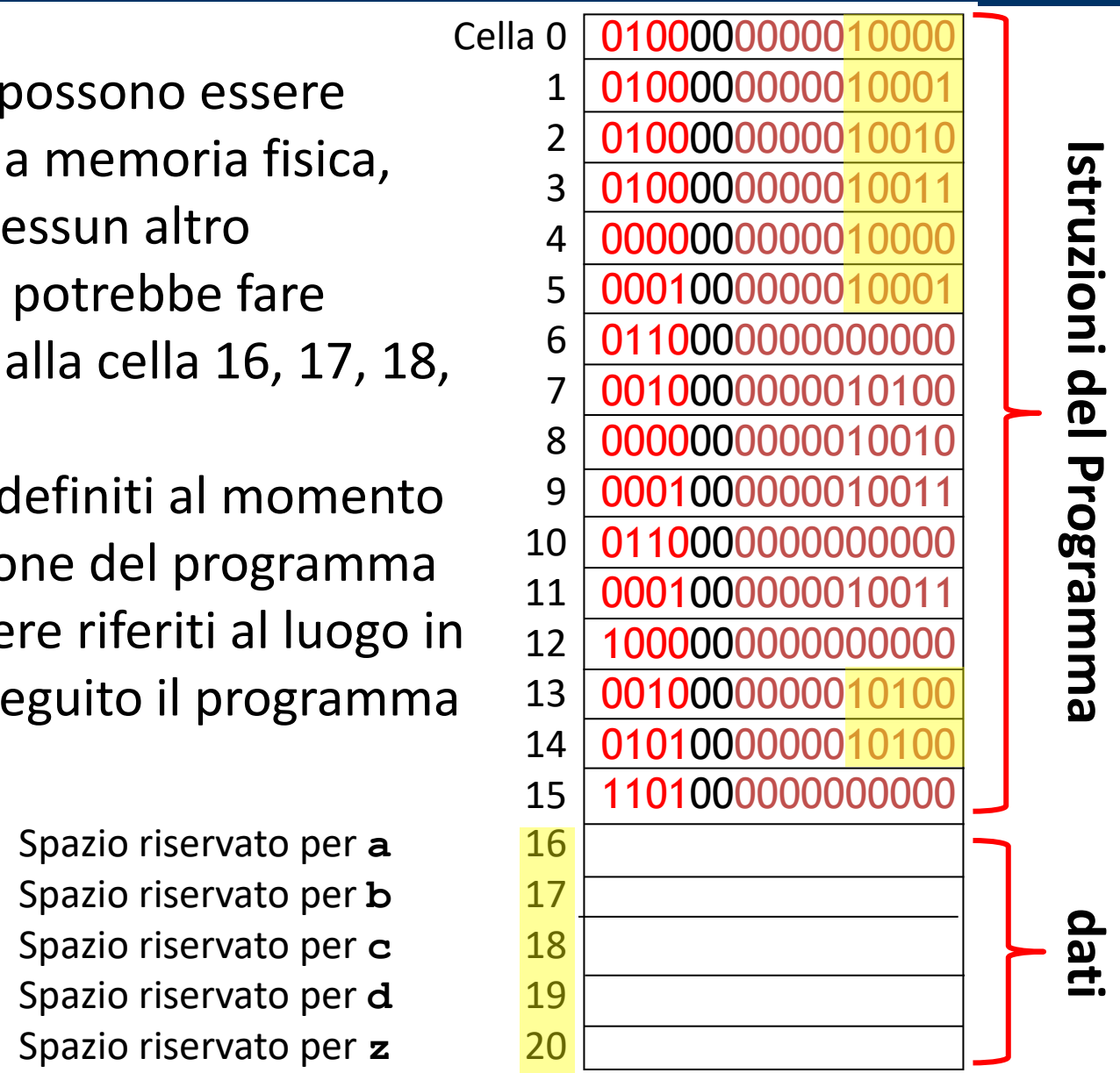
I programmi fanno riferimento a celle di memoria al loro interno



Programma in Memoria Centrale

Questi non possono essere indirizzi nella memoria fisica, altrimenti nessun altro programma potrebbe fare riferimento alla cella 16, 17, 18, 19 e 20

Gli indirizzi definiti al momento della creazione del programma devono essere riferiti al luogo in cui viene eseguito il programma



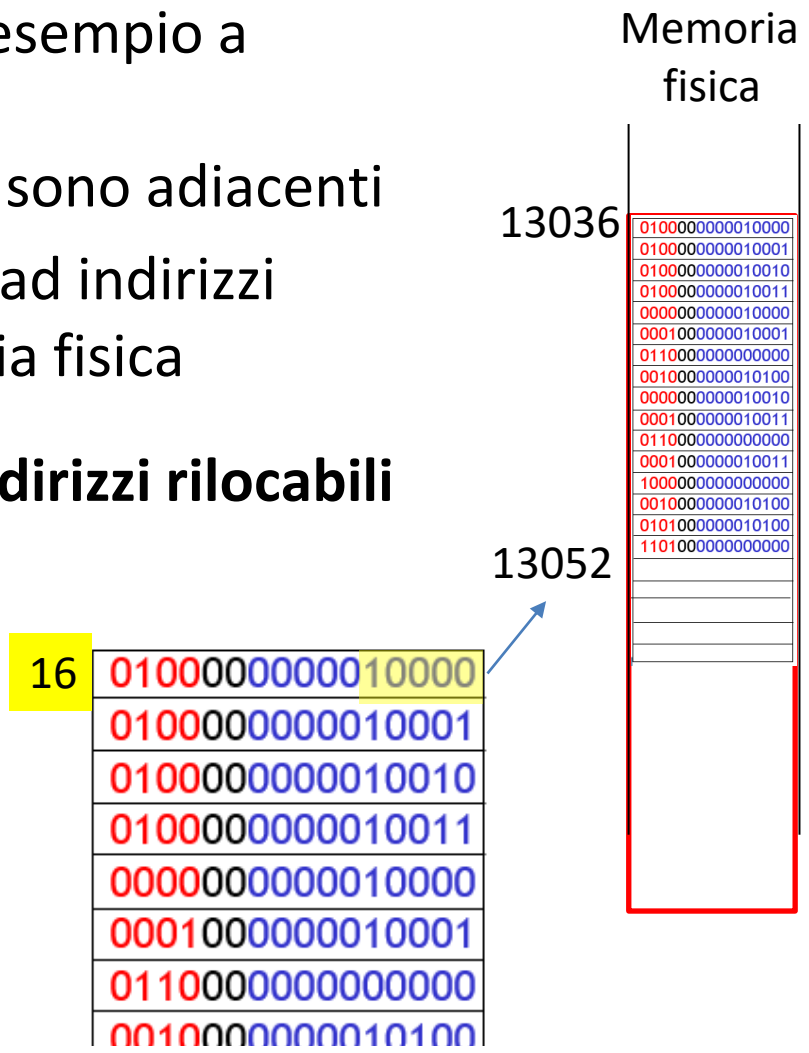
Indirizzi Rilocabili

Quando il programma viene eseguito viene caricato nella memoria fisica (nell'esempio a partire dalla cella 13036)

- Tutte le celle del programma sono adiacenti
- Il programma fa' riferimento ad indirizzi diversi da quelli della memoria fisica

Ogni processo fa' riferimento ad **indirizzi rilocabili** che partono da zero

La **rilocazione dinamica** è un meccanismo che associa ad ogni indirizzo logico quello fisico corrispondente

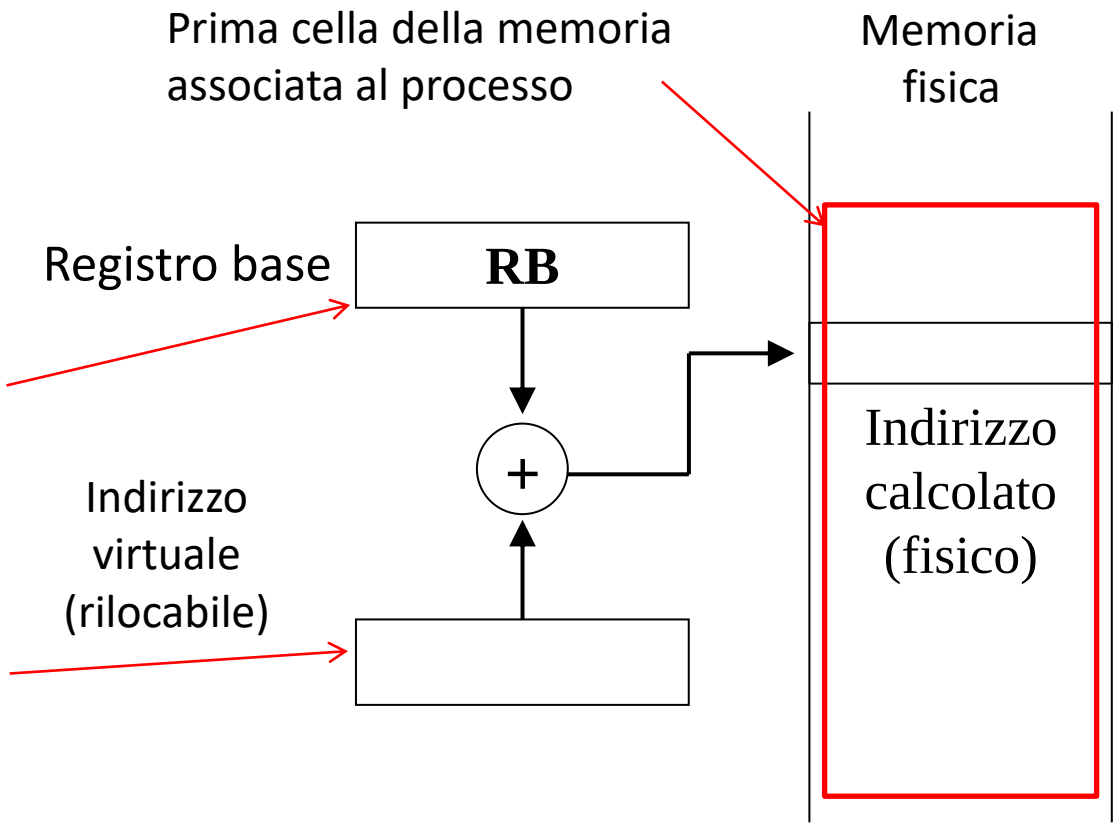


Rilocazione Dinamica

Rilocazione Dinamica

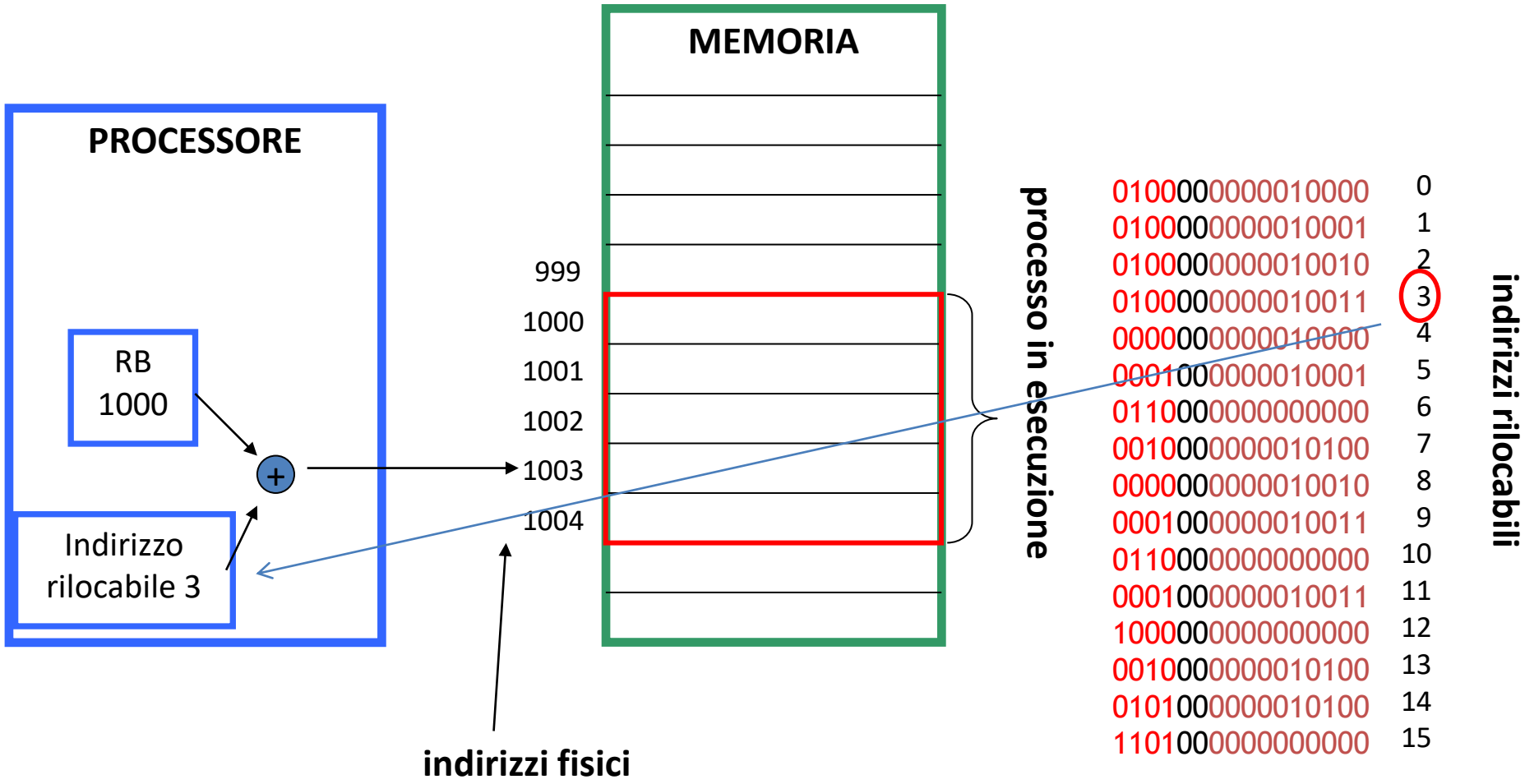
E' un registro della CPU che contiene **l'indirizzo fisico** della **prima cella** in cui è stato caricato il **programma**

l'indirizzo delle **istruzioni** e delle **variabili** specificato nel **codice**



Il processo in esecuzione viene posizionato in un punto della memoria fisica definito dal sistema operativo

Rilocazione Dinamica



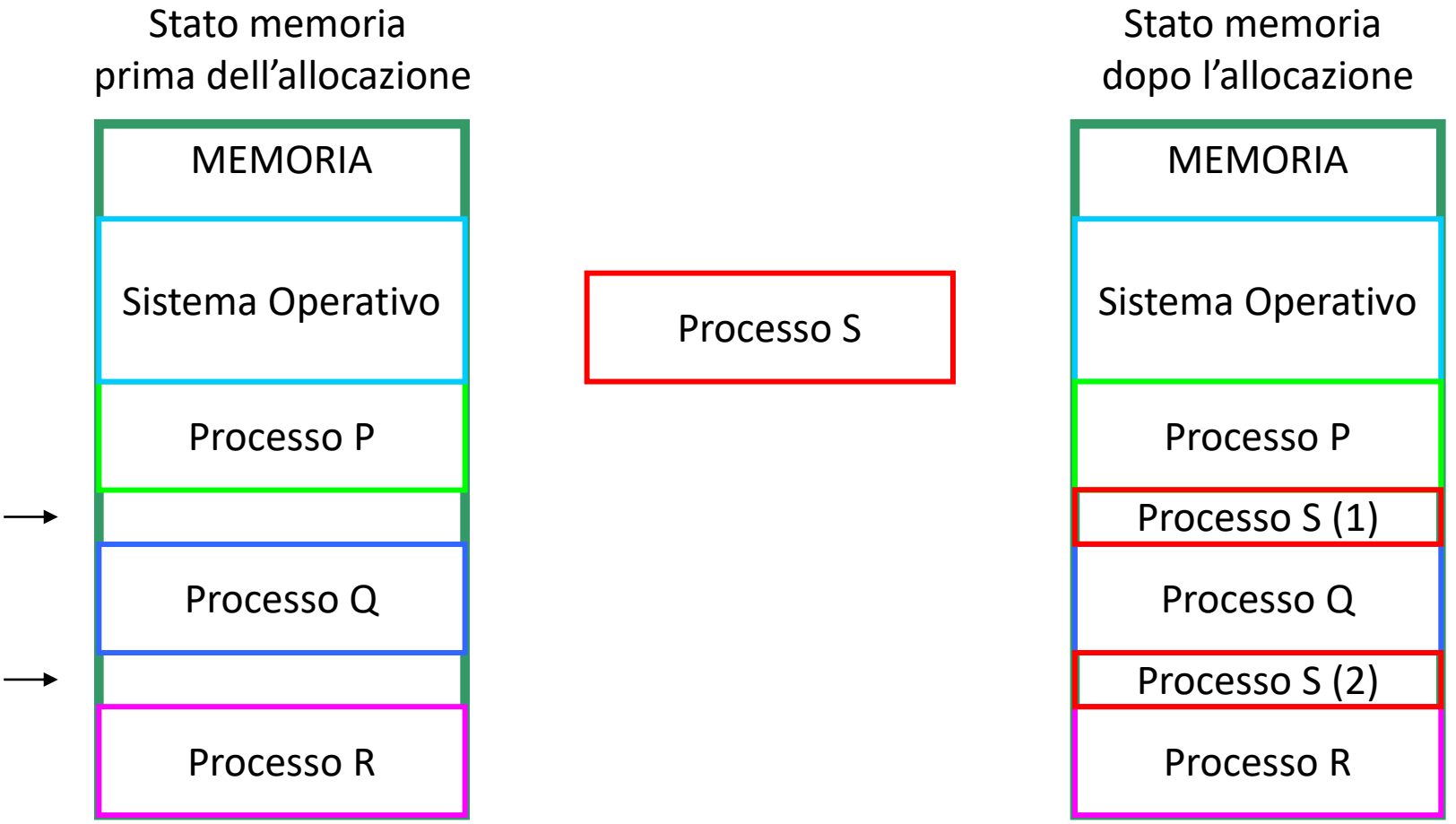
Soluzione al problema della frammentazione

Durante l'utilizzo la memoria fisica potrebbe risultare frammentata:

- Processi che terminano potrebbero lasciare dei buchi
- I buchi potrebbero non essere abbastanza grandi per essere riempiti da altri processi attivati in seguito

Per **evitare la frammentazione** della memoria (spazi vuoti in memoria inutilizzabili) è utile **allocare** i **programmi suddividendoli** in pezzi di dimensione fissata

Soluzione al Problema della Frammentazione

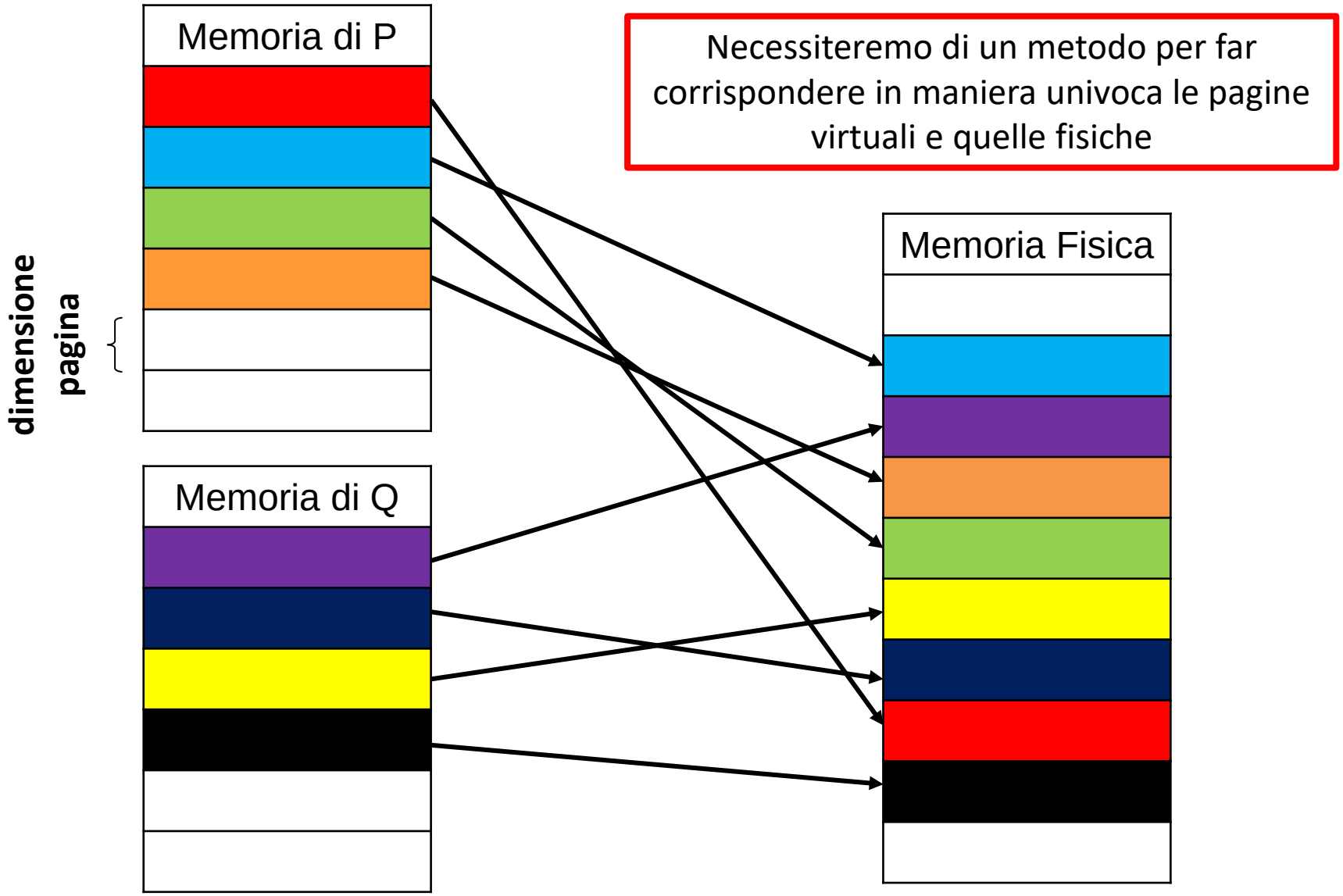


Occorre «spezzare» il processo S per trovare posto nella memoria fisica, ma la rilocazione dinamica vista prima non funziona

Paginazione

- Si **rinuncia** ad avere una **zona contigua** della **memoria fisica** per ciascun processo
- Ogni programma ha visione di una **memoria virtuale**, interamente dedicata a se
- La **memoria virtuale** del programma viene **suddivisa in pagine virtuali**, porzioni di lunghezza fissa (potenza di 2, es: 4K)
- La **memoria fisica** viene suddivisa in **pagine fisiche** della **stessa dimensione**
- **Durante l'esecuzione, le pagine virtuali** vengono **caricate** in altrettante **pagine fisiche**, **non necessariamente contigue**

Paginazione

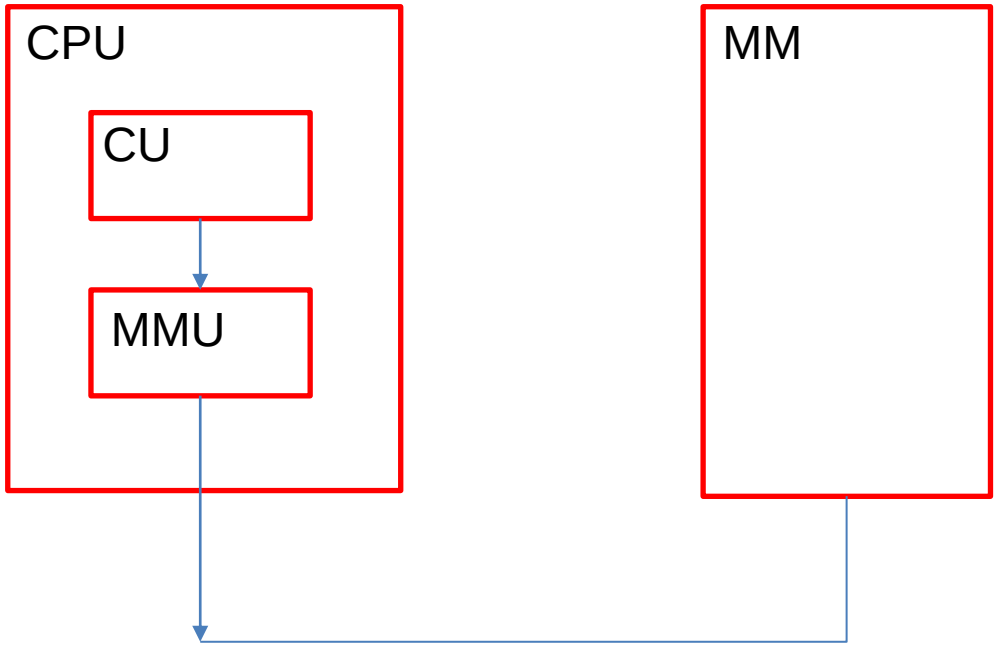


Memoria virtuale vs. fisica

- La memoria effettivamente presente nel calcolatore è la **memoria *fisica*** e gli indirizzi delle sue celle sono detti ***indirizzi fisici***
- I **processi** fanno riferimento ad **indirizzi *virtuali*** che sono nella **memoria *virtuale***
- Gli indrizzi virtuali sono poi tradotti dal sistema operativo in indirizzi fisici, per poter arrivare al dato in memoria

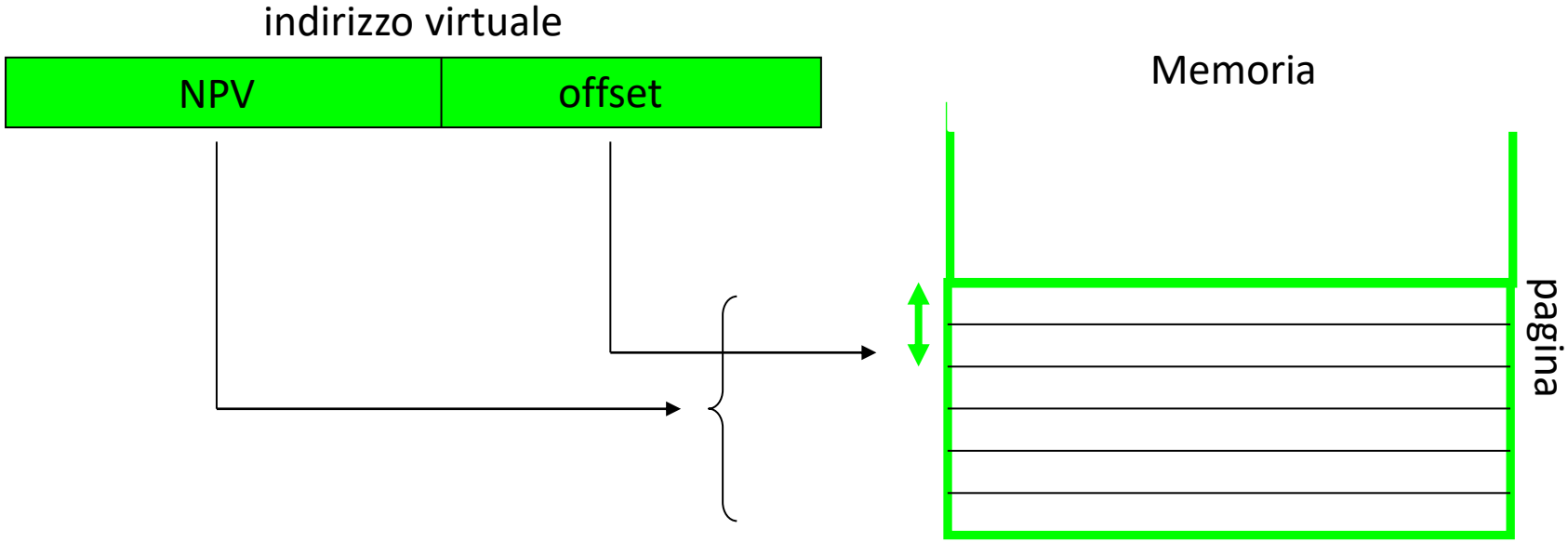
Mapping Pagine Virtuali in Pagine Fisiche

- Una componente della CPU, la Memory Management Unit (**MMU**) gestisce le associazioni tra pagine virtuali e pagine fisiche
- Le richieste di celle di accesso alla memoria vengono tradotte dalla MMU, prima di passare alla Main Memory



Paginazione: Struttura degli indirizzi virtuali

- Un **indirizzo virtuale** è costituito da:
  - un **numero di pagina virtuale** (**NPV**)
  - uno **spiazzamento** (**offset**) all'interno della pagina

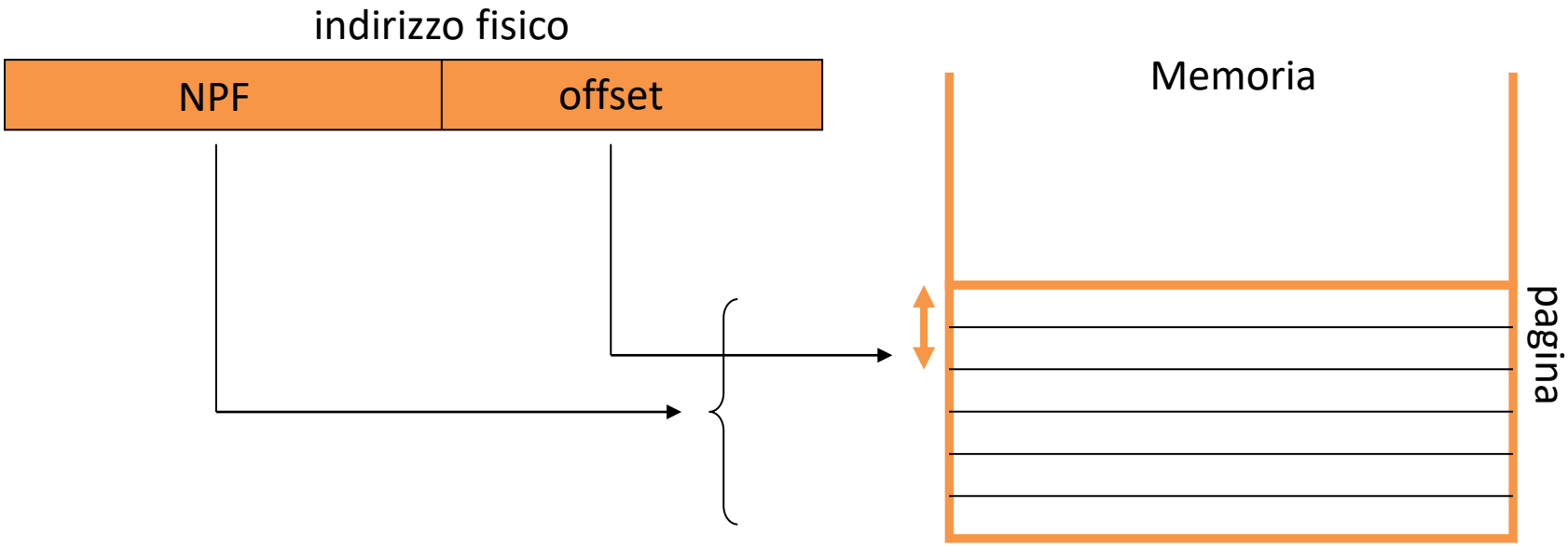


NB: se (come succede sempre in pratica) la dimensione della pagina è una potenza di 2, ***giustapponendo*** numero pagina e offset si calcola la somma base_pagina+offset

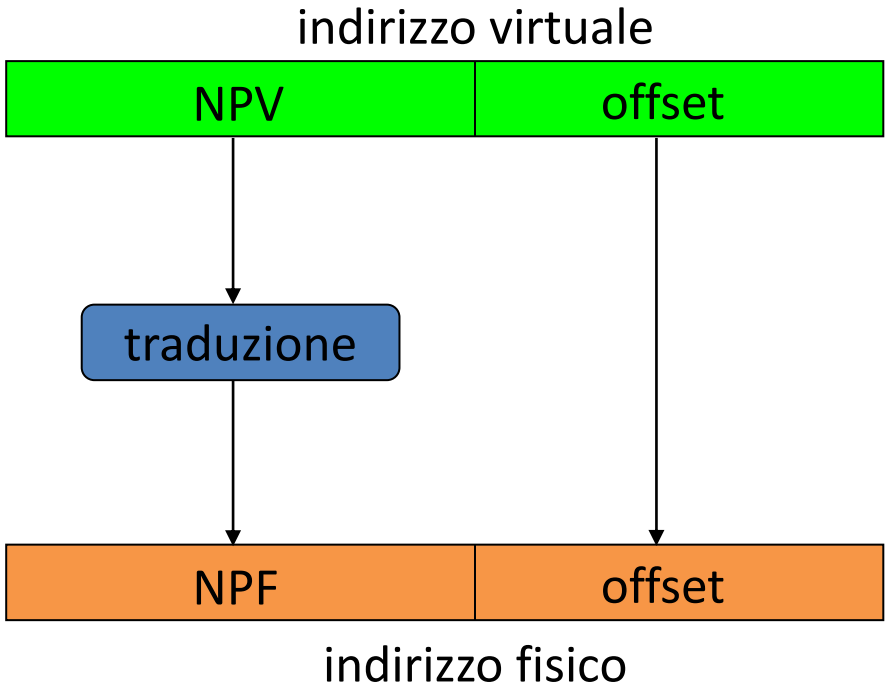
Paginazione: Struttura degli indirizzi fisici

La struttura degli **indirizzi fisici** è:

- un numero di pagina fisica (**NPF**)
- uno spiazzamento (**offset**) all'interno della pagina



Paginazione: Traduzione dal virtuale al fisico



le **pagine virtuali** e quelle **fisiche hanno la stessa dimensione**, quindi **l'offset assume gli stessi valori**

Esempio

- Spazio di indirizzamento virtuale con indirizzi da 32 bit

  ⇨ 2^{32} indirizzi

- Dimensione di pagina: ogni pagina occupa 4K byte

  ⇨ contiene 2^{12} byte

  - **Offset:** numero di bit necessari per indirizzare una cella all'interno di una pagina

    ⇨ $\log_2(\text{dim\_pagina}) = \log_2(4000) \approx 12$

  - **Numero di pagine:** dello spazio di indirizzamento virtuale

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

Esempio

- Spazio di indirizzamento fisico totale 4M parole (o celle)

  ⇨ 2^{22} indirizzi

- Dimensione di pagina fisica (deve essere uguale alla dimensione della pagina virtuale):

  ⇨ ogni pagina ha 4K ⇨ 2^{12} byte

- Spazio di indirizzamento nella pagina

  ⇨ 12 bit (offset)

- Numero di pagine dello spazio di indirizzamento fisico

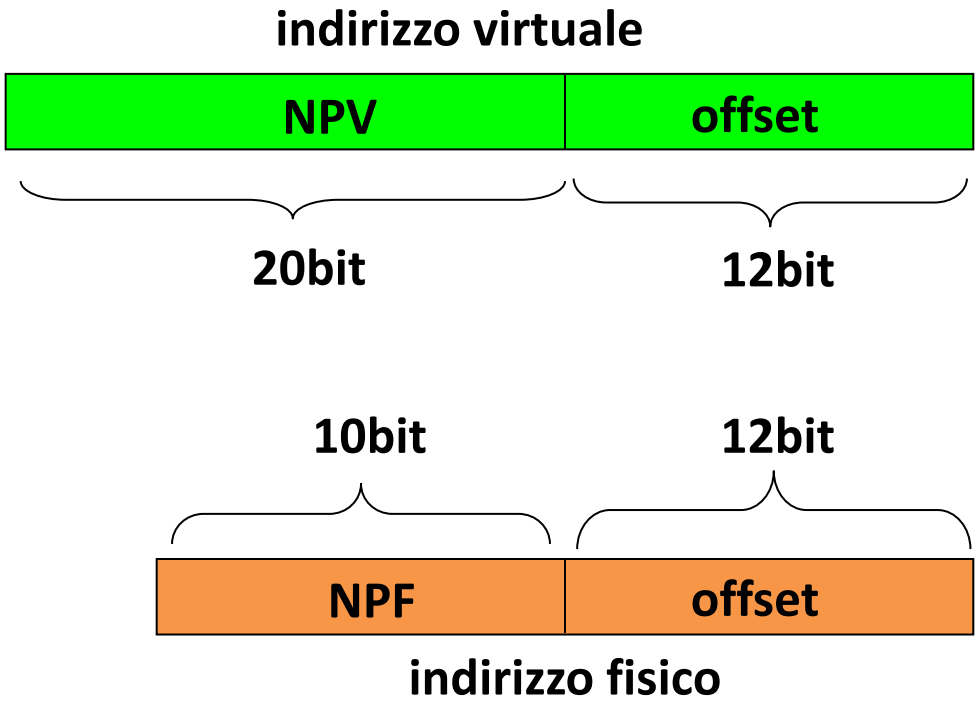
  ⇨ $2^{22} / 2^{12} = 2^{10}$ pagine

- Spazio dedicato ad indirizzo pagina

  ⇨ NPF 10 bit

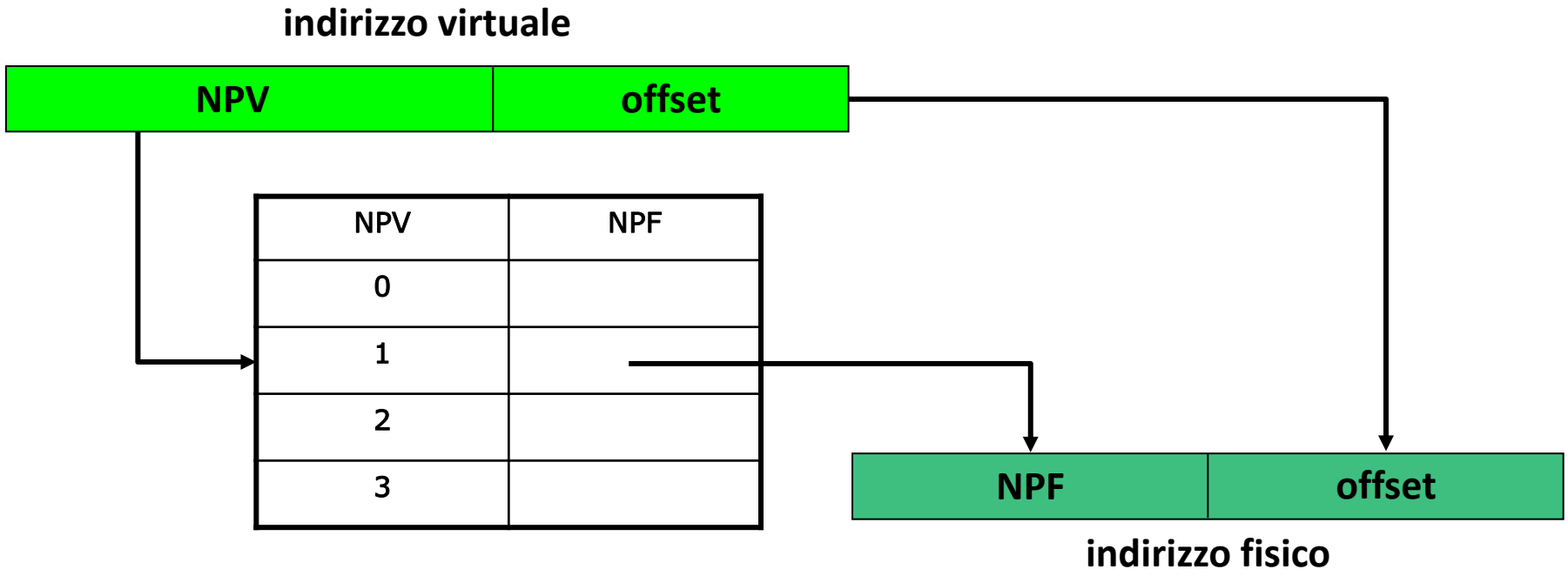
  ⇨ NPV 20 bit

Esempio (segue)



Paginazione: Tabella Delle Pagine

- E’ il meccanismo più semplice tradurre da virtuale a fisico
- E’ possibile fare una tabella delle pagine per ciascun processo



Memory Management Unit

- Per accelerare la **traduzione da NPV a NPF** si ricorre alla Memory Management Unit (**MMU**)
- La **MMU è una memoria particolarmente veloce** (memoria associativa) dalle dimensioni ridotte, contenente solo le **informazioni sulle pagine più utilizzate**
- Visto che gli **NPV** e gli **NPF** si riferiscono alle **pagine di un processo**, ogni volta che il processo in esecuzione cambia la MMU dovrebbe essere tutta riscritta
- Per evitare ciò si **aggiunge una colonna** con **l'identificativo del processo** al quale appartengono le pagine e **un registro** che contiene l'identificativo del processo attualmente in esecuzione

Memoria Virtuale vs. Fisica

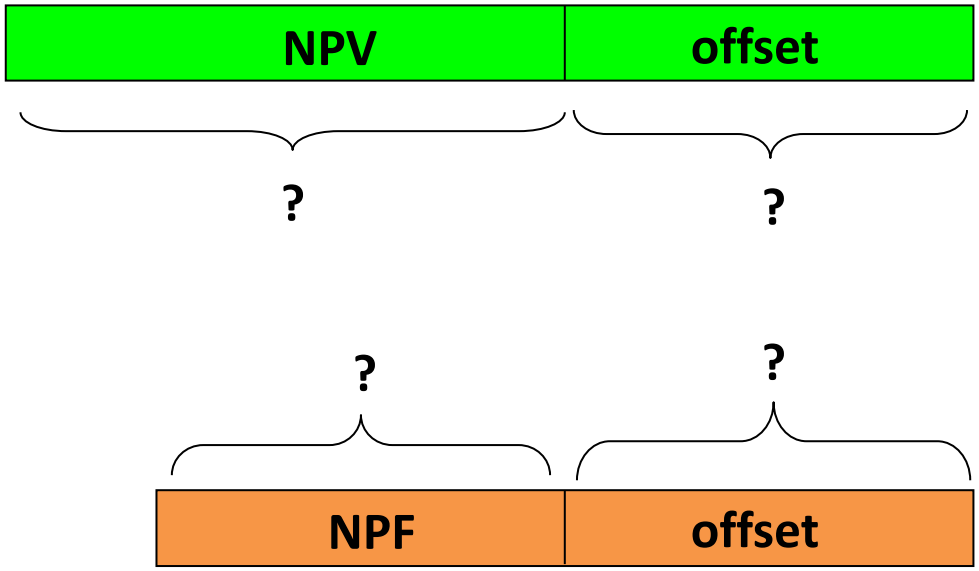
- La memoria virtuale può essere maggiore della memoria fisica
  - Basta prevedere più pagine virtuali che fisiche
  - Le pagine di memoria virtuale che non risiedono nella memoria fisica sono nella memoria di massa
  - Questo permette al SO e ai programmi di avere più processi in memoria di quanta sia la memoria fisica
- La memoria **virtuale** e quella **fisica non coincidono** perché conviene mantenere nella memoria fisica una sola copia di parti di programmi che sono uguali in diversi processi (memoria condivisa)
- La **memoria fisica** può essere **insufficiente** a contenere la **memoria virtuale** di tutti processi

Pagine Residenti e Non Residenti

- **Durante l'esecuzione** di un programma **solo alcune pagine virtuali** sono **caricate** in **pagine fisiche**
- Le pagine virtuali caricate in sono dette **pagine residenti**
- A **ogni accesso** alla memoria **si controlla che all'indirizzo virtuale corrisponda una pagina residente**, altrimenti si produce un interrupt di segnalazione di errore detto **page-fault**
- Il **processo viene sospeso in attesa** che la pagina richiesta venga caricata in memoria, eventualmente scaricando su disco una pagina già residente per liberare lo spazio necessario
- Aumentare la memoria fisica di un PC aumenta il numero di pagine residenti e quindi ne velocizza il funzionamento

Esempio

- Un sistema dotato di memoria virtuale con paginazione è caratterizzato dai seguenti parametri: l'indirizzo logico è di 13 bit e l'indirizzo fisico è di 12 bit; la dimensione delle pagine è di 512 byte
- Definire la struttura dell'indirizzo logico e di quello fisico indicando la lunghezza dei campi che li costituiscono



Esempio

- Un sistema dotato di memoria virtuale con paginazione è caratterizzato dai seguenti parametri: l'indirizzo logico è di 13 bit e l'indirizzo fisico è di 12 bit; la dimensione delle pagine è di 512 byte
- Definire la struttura dell'indirizzo logico e di quello fisico indicando la lunghezza dei campi che li costituiscono

Risposta

- Indirizzo logico: NPV: 4 bit offset logico: 9 bit
- Indirizzo fisico: NPF: 3 bit offset fisico: 9 bit

Esempio

Un sistema dispone di 8 Kbyte di memoria fisica indirizzabile ed è dotato di memoria virtuale con paginazione caratterizzata dai seguenti parametri: l'indirizzo logico è di 15 bit e le pagine sono di 256 byte

Qual è la dimensione della memoria virtuale indirizzabile?

Definire la struttura dell'indirizzo logico e di quello fisico indicando la lunghezza dei campi che li costituiscono

- 15bit → 2^15 byte → 32Kbyte
- 256 byte → 2^8 byte → offset = 8bit
- 8kbyte → 2^13 byte → indirizzo fisico = 13 bit
- Indirizzo fisico: NPF 5bit, offset 8bit
- Indirizzo logico: NPV 7bit, offset 8bit

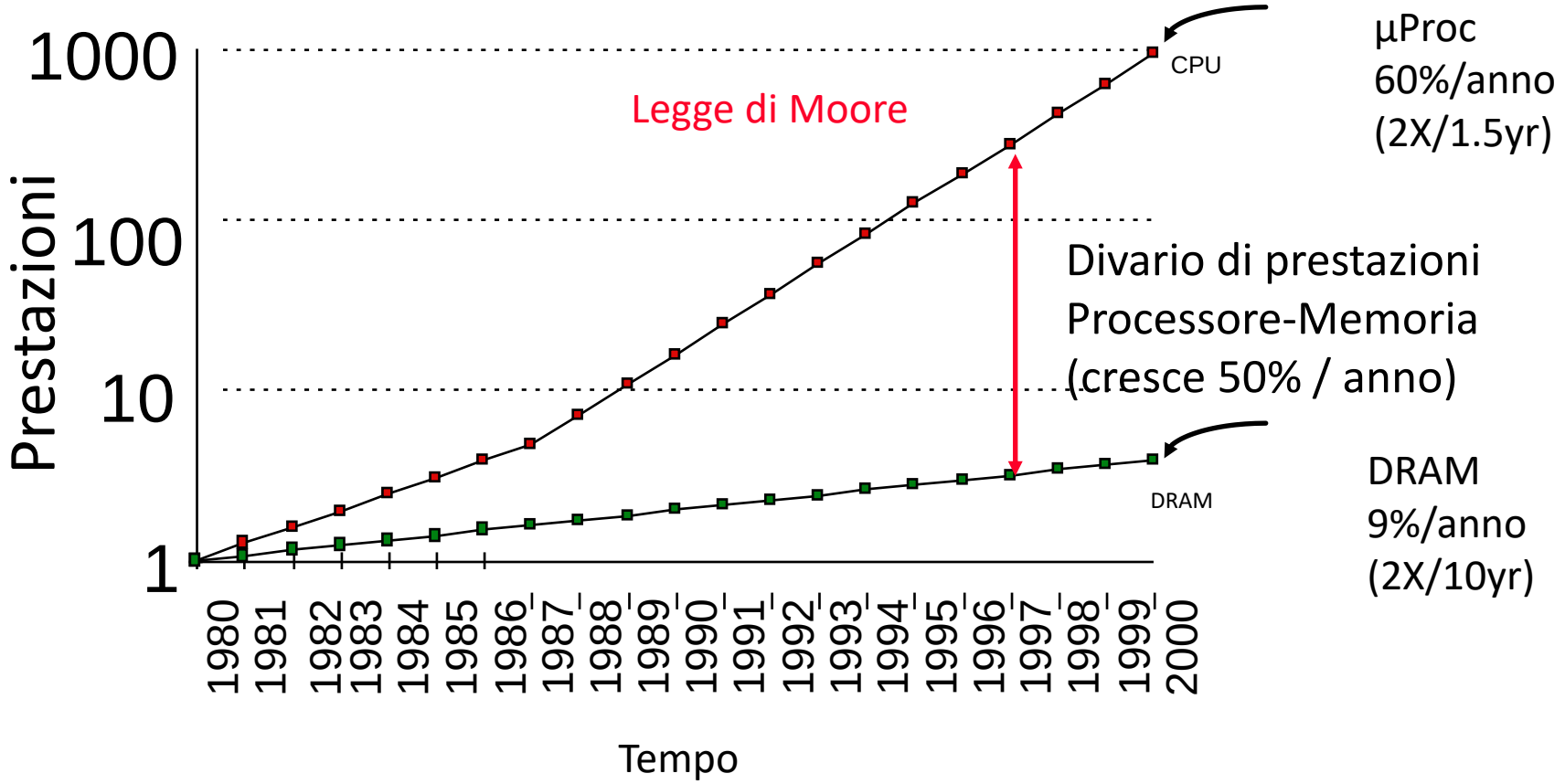
La Memoria Cache

Il Problema della Memoria: Costo vs. Prestazioni

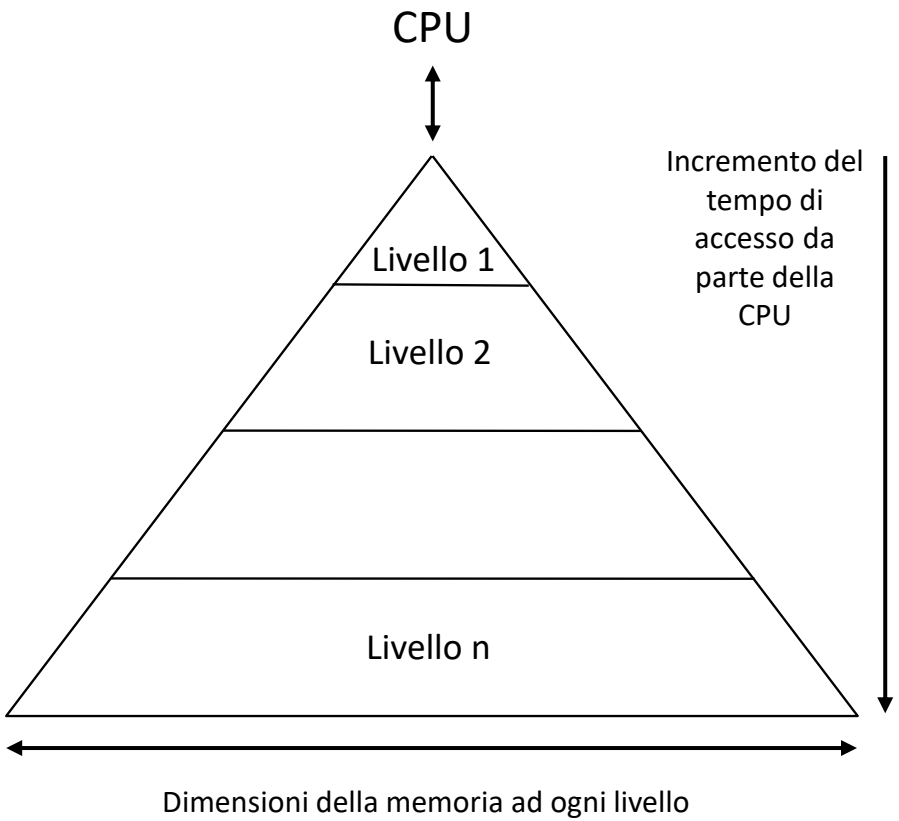
- Il **tempo** di **accesso a dati** e istruzioni in MM è **maggiore** rispetto al tempo per **eseguire istruzioni**:
  - fornire agli utenti una memoria grande e veloce
  - fornire al processore i dati alla velocità con cui è in grado di elaborarli
- Però il tasso di crescita nella velocità dei processori non è stato seguito da quello delle memorie:
  - SRAM: Tempo di accesso 2 - 25ns al costo di $100 - $250 per Mbyte
  - DRAM: Tempo di accesso 60-120ns al costo di $5 - $10 per Mbyte
  - Disco: Tempo di accesso da 10 a 20 million ns al costo di $0.10 – $0.20 per Mbyte

[dati non aggiornati]

Prestazioni di Processori e Cache

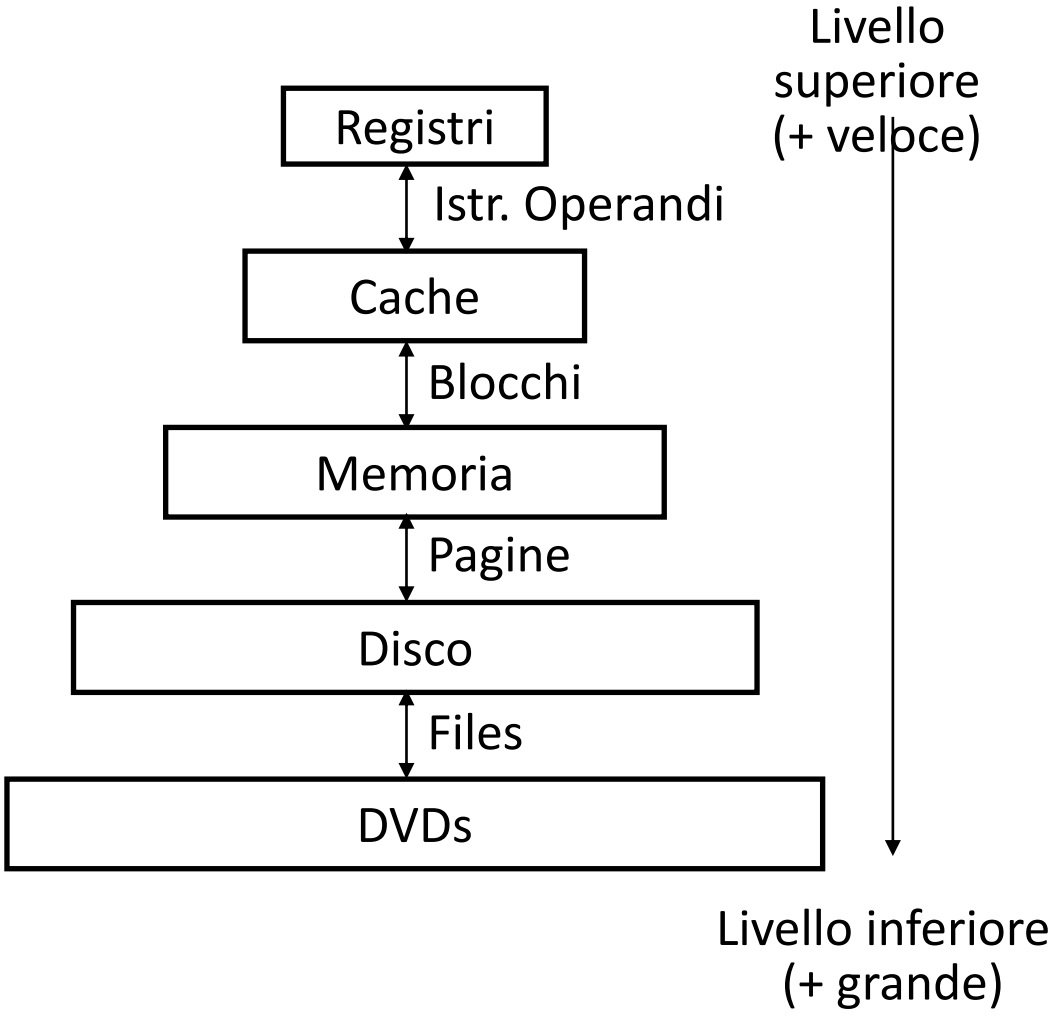


Soluzione: Gerarchia di Memoria



Utilizzare diversi livelli di memoria, con tecnologie diverse in modo da ottenere un buon compromesso costo/prestazioni

Esempio di Gerarchia



Registri CPU: h Bytes, <10s ns

Cache: K Bytes, 10-100 ns
1-0.1 cents/bit

MM: M Bytes, 200ns- 500ns
0.0001-0.00001 cents /bit

Disco: G Bytes, 10 ms
1E-6 – 1E-5 cents/bit

DVDs, infinito, sec-min
1E-11 cent/bit

Gestione della Gerarchia della Memoria

- Un sistema sarà **efficiente** se riesce sempre a **trovare nella parte alta della memoria le informazioni richieste** dai processi in esecuzione
- **È possibile sfruttare l'organizzazione gerarchica della memoria** per incrementarne le prestazioni
- Occorrono **strategie** per **portare le porzioni** di memoria più **utilizzate** nelle **gerarchie più alte** della memoria
  - Occorre **decidere**, durante l'esecuzione del programma, **quali dati mantenere/copiare/rimuovere nei vari livelli della gerarchia di memoria**.
- Queste strategie si basano sul principio di **Località**

Il Principio di Località

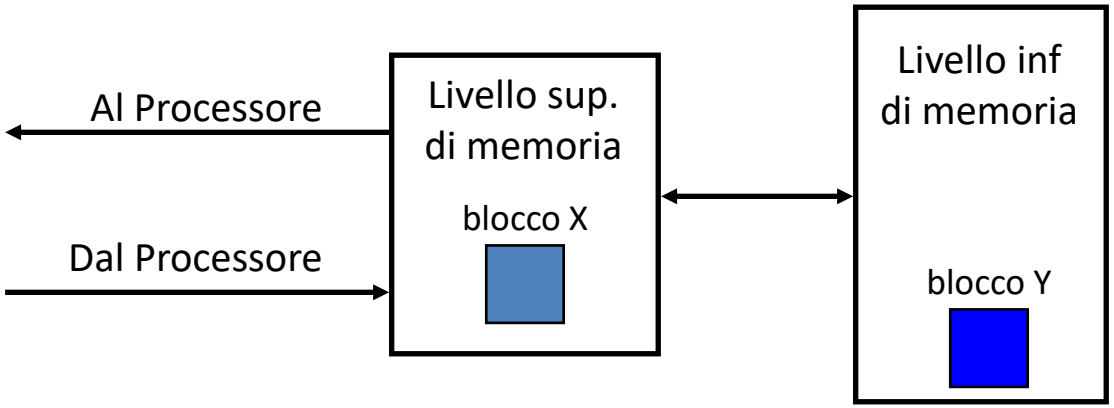
Località: in ogni istante di tempo **un programma accede** a una **parte** relativamente **piccola** del suo **spazio di indirizzamento**

- **Località temporale**: se un dato viene referenziato in un dato istante, è probabile che lo stesso dato venga nuovamente richiesto entro breve
- **Località Spaziale**: se un dato viene utilizzato in un dato istante, è probabile che dati posizionati in celle di memoria adiacenti vengano anch'essi richiesti entro breve

Negli ultimi 15 anni, le tecniche di miglioramento delle prestazioni nell'hardware si sono basate sul principio di località

Gerarchia di Memoria

- Si consideri una gerarchia a due livelli
- Il processore richiede un dato al sistema di memoria:
  - La richiesta viene prima inviata al livello di memoria superiore (più vicino al processore)
  - Se il dato è presente, la richiesta ha successo
  - Se il dato non è presente nel livello superiore (fallimento della richiesta) la ricerca viene effettuata nel livello inferiore



Gerarchia di Memoria: Definizioni

Hit (successo): dati presenti in un blocco del livello superiore (esempio: blocco X)

- **Hit Rate (tasso di successo)**: numero di accessi a memoria che trovano il dato nel livello più alto sul numero totale di accessi
- **Hit Time (tempo di successo)**: tempo per accedere al dato nel livello più alto della gerarchia:

Hit Time = Tempo di accesso alla CACHE + tempo per determinare successo/fallimento della richiesta

Gerarchia di Memoria: Definizioni

Miss (fallimento): i dati devono essere recuperati dal livello inferiore della memoria (Blocco Y)

- **Miss Rate (tasso di fallimento)** = 1 - (Hit Rate)
- **Miss Penalty (tempo di fallimento)**:
  - tempo per determinare il MISS +
  - tempo necessario a sosituire un blocco nel livello superiore +
  - tempo per trasferire il blocco al processore
- tipicamente si ha: Hit Time << Miss Penalty

Tempo medio di accesso in presenza di memoria cache:
HitTime*HitRate + MissRate*MissPenalty
semplicemente la media pesata con le probabilità di successo/falluimento

Cache

- Memoria al livello superiore della gerarchia
  - veloce nei tempi di accesso ma di dimensioni ridotte
- Contiene i dati che il processore richiama **più frequentemente**, per evitare l'accesso alle memorie più lente (che danno luogo a situazioni di stallo)
- Risiede **fisicamente vicino** al processore, possibilmente nello stesso chip
- Alcuni processori hanno due tipi di cache: L1 (interno al processore) ed L2 (esterno al processore)
- Sfruttare il principio di località dei programmi e tenere in memoria cache i dati utilizzati più di recente
- Obiettivo: fornire dati al processore in uno/due cicli di clock

Cache e principio di località

- Le memorie cache sfruttano il principio di **località spaziale trasferendo** dal livello inferiore della gerarchia **più dati di quanti non ne siano stati strettamente richiesti** (blocco o linea di cache)
- La **località temporale** viene sfruttata **nella scelta del blocco da sostituire nella gestione di un fallimento** (es: sostituire il blocco a cui si è fatto accesso meno di recente)

Reminder