

Università di Ferrara
Laurea Triennale in Informatica
A.A. 2021-2022
Sistemi Operativi e Laboratorio

10. Gestione della Memoria

Prof. Carlo Giannelli

`http://www.unife.it/scienze/informatica/insegnamenti/
sistemi-operativi-laboratorio`

`http://docente.unife.it/carlo.giannelli`

`https://ds.unife.it/people/carlo.giannelli/`

Multiprogrammazione e gestione memoria

Obiettivo primario della **multiprogrammazione** è
l'uso efficiente delle risorse computazionali:

- ❑ efficienza nell'uso della CPU
- ❑ velocità di risposta dei processi
- ❑ ...

Necessità di mantenere più processi in memoria centrale:
SO deve gestire la memoria in modo da consentire la
presenza contemporanea di più processi

Caratteristiche importanti:

- Velocità
- Grado di multiprogrammazione
- Utilizzo della memoria
- Protezione

Gestione della memoria centrale

A livello hw:

ogni sistema è equipaggiato con **un unico spazio di memoria accessibile direttamente da CPU e dispositivi**

Compiti di SO

- ❑ **allocare** memoria ai processi
- ❑ **deallocare** memoria
- ❑ **separare** gli spazi di indirizzi associati ai processi (**protezione**)
- ❑ realizzare i **collegamenti (binding)** tra gli **indirizzi logici** specificati dai processi e le corrispondenti **locazioni nella memoria fisica**
- ❑ **memoria virtuale**: gestire spazi di indirizzi logici di dimensioni superiori allo spazio fisico

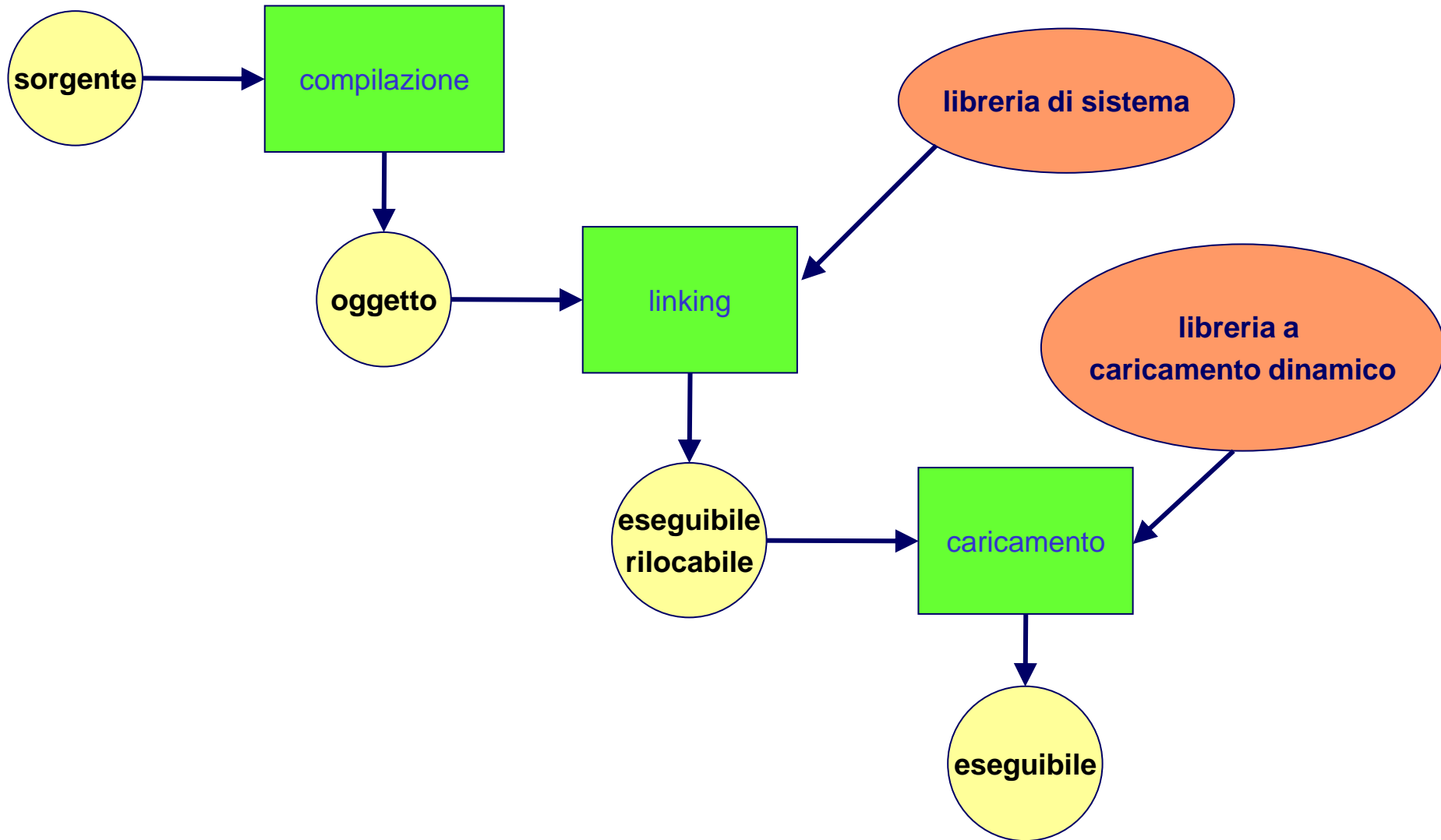
Accesso alla memoria

Memoria centrale:

- ❑ vettore di celle, ognuna univocamente individuata da un indirizzo
- ❑ operazioni fondamentali sulla memoria: **load/store dati e istruzioni**
- ❑ **Indirizzi**
 - **simbolici**: riferimenti a celle di memoria nei programmi in forma sorgente mediante nomi simbolici
 - **logici**: riferimenti a celle **nello spazio logico di indirizzamento del processo**
 - **fisici**: **riferimenti assoluti** delle celle in memoria a **livello HW**

Quale relazione tra i diversi tipi di indirizzo?

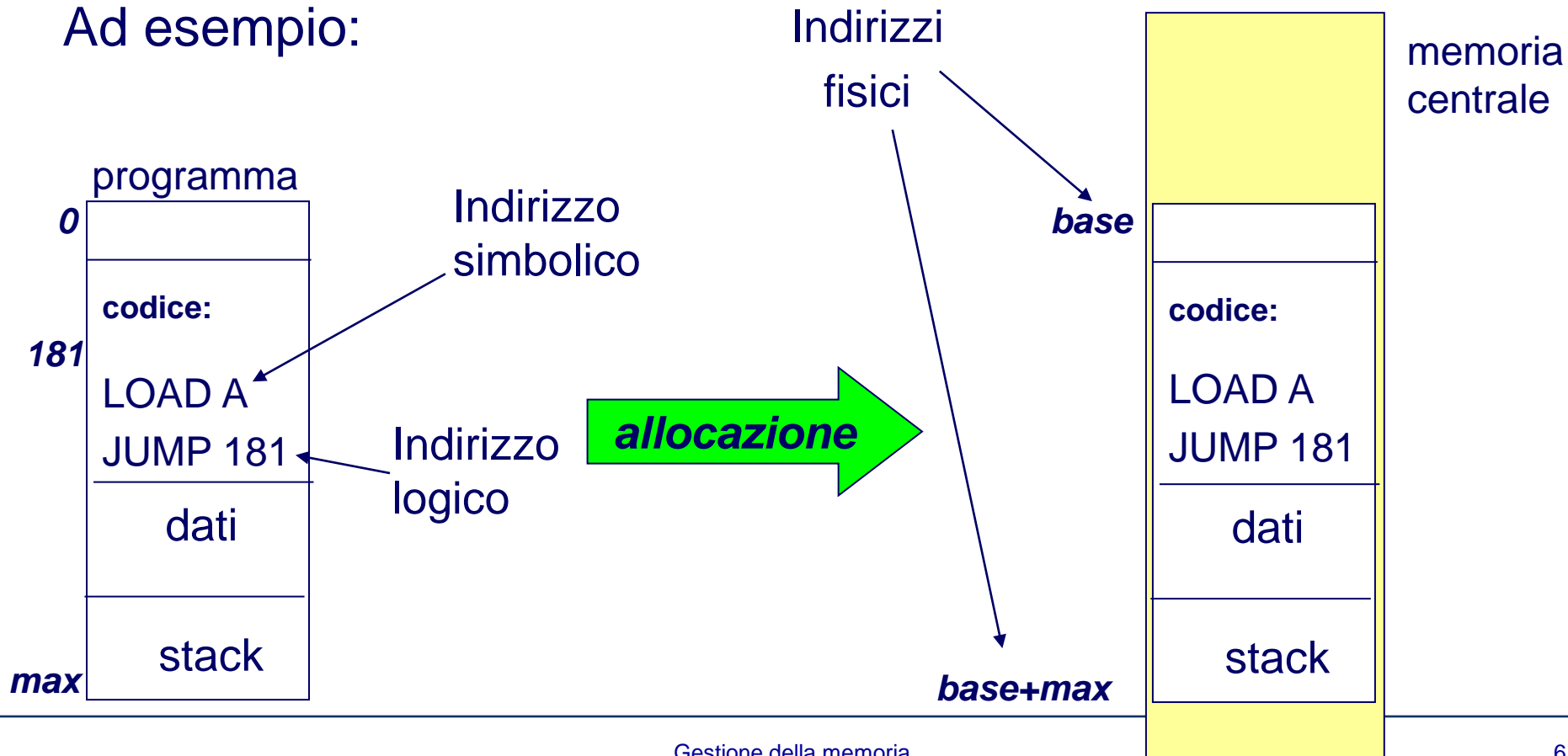
Fasi di sviluppo di un programma



Indirizzi simbolici, logici e fisici

Ogni processo dispone di un **proprio spazio di indirizzamento logico** $[0, \text{max}]$ che viene allocato nella memoria fisica.

Ad esempio:



Binding degli indirizzi

Ad ogni **indirizzo logico/simbolico** viene fatto corrispondere un **indirizzo fisico**: l'associazione tra indirizzi relativi e indirizzi assoluti viene detta **binding**

Binding può essere effettuato:

❑ staticamente

- **a tempo di compilazione.** Il compilatore genera degli **indirizzi assoluti** (esempio: file .com DOS)
- **a tempo di caricamento.** Il compilatore genera degli **indirizzi relativi** che vengono convertiti in indirizzi assoluti dal **loader (codice rilocabile)**

❑ dinamicamente

- **a tempo di esecuzione.** Durante l'esecuzione un processo può essere spostato da un'area all'altra

Caricamento/collegamento dinamico

Obiettivo: ottimizzazione della memoria

Caricamento dinamico

- ❑ in alcuni casi è possibile caricare in memoria una funzione/procedura a runtime **solo quando avviene la chiamata**
- ❑ **loader di collegamento rilocabile:** carica e collega dinamicamente la funzione al programma che la usa
- ❑ la funzione può essere usata da più processi simultaneamente. Problema di **visibilità** → compito SO è concedere/controllare:
 - **l'accesso di un processo allo spazio di un altro processo**
 - **l'accesso di più processi agli stessi indirizzi**

Overlay

In generale, la memoria disponibile può non essere sufficiente ad accogliere codice e dati di un processo

Soluzione a **overlay** mantiene in memoria istruzioni e dati:

- che vengono utilizzati **più frequentemente**
 - che sono necessari nella **fase corrente**
-
- ❑ codice e dati di un processo vengono suddivisi (dal programmatore?) in **overlay** che vengono caricati e scaricati dinamicamente (dal gestore di overlay, di solito esterno al SO)

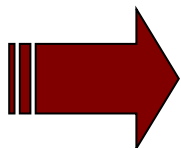
Overlay: esempio

Assembler a 2 passi: produce l'eseguibile di un programma assembler, mediante 2 fasi sequenziali

1. Creazione della tabella dei simboli (passo 1)
2. Generazione dell'eseguibile (passo 2)

4 componenti distinte nel codice assembler:

- ❑ **Tabella dei simboli** (ad es. dim 20KB)
- ❑ **Sottoprogrammi comuni** ai due passi (ad es. 30KB)
- ❑ **Codice passo 1** (ad es. 70KB)
- ❑ **Codice passo 2** (ad es. 80KB)



spazio richiesto per l'allocazione integrale dell'assembler
è quindi di 200KB

Overlay: esempio

Hp: spazio libero in memoria di 150KB

Soluzione: 2 overlay da caricare in sequenza (passo 1 e passo 2); caricamento/scaricamento vengono effettuati da una parte aggiuntiva di codice (**gestore di overlay**, dimensione 10KB) aggiunta al codice dell'assembler

<i>Tabella dei simboli</i>	20KB
<i>Sottoprogrammi comuni</i>	30KB
<i>Gestore overlay</i>	10KB
<i>Codice del Passo 1</i>	70KB

Occupazione complessiva: 130KB

<i>Tabella dei simboli</i>	20KB
<i>Sottoprogrammi comuni</i>	30KB
<i>Gestore overlay</i>	10KB
<i>Codice del Passo 2</i>	80KB

Occupazione complessiva: 140KB

Tecniche di allocazione memoria centrale

Come vengono allocati codice e dati dei processi in memoria centrale?

Varie tecniche

❑ **Allocazione Contigua**

- a partizione singola
- a partizioni multiple

❑ **Allocazione non contigua**

- paginazione
- segmentazione

Allocazione contigua a partizione singola

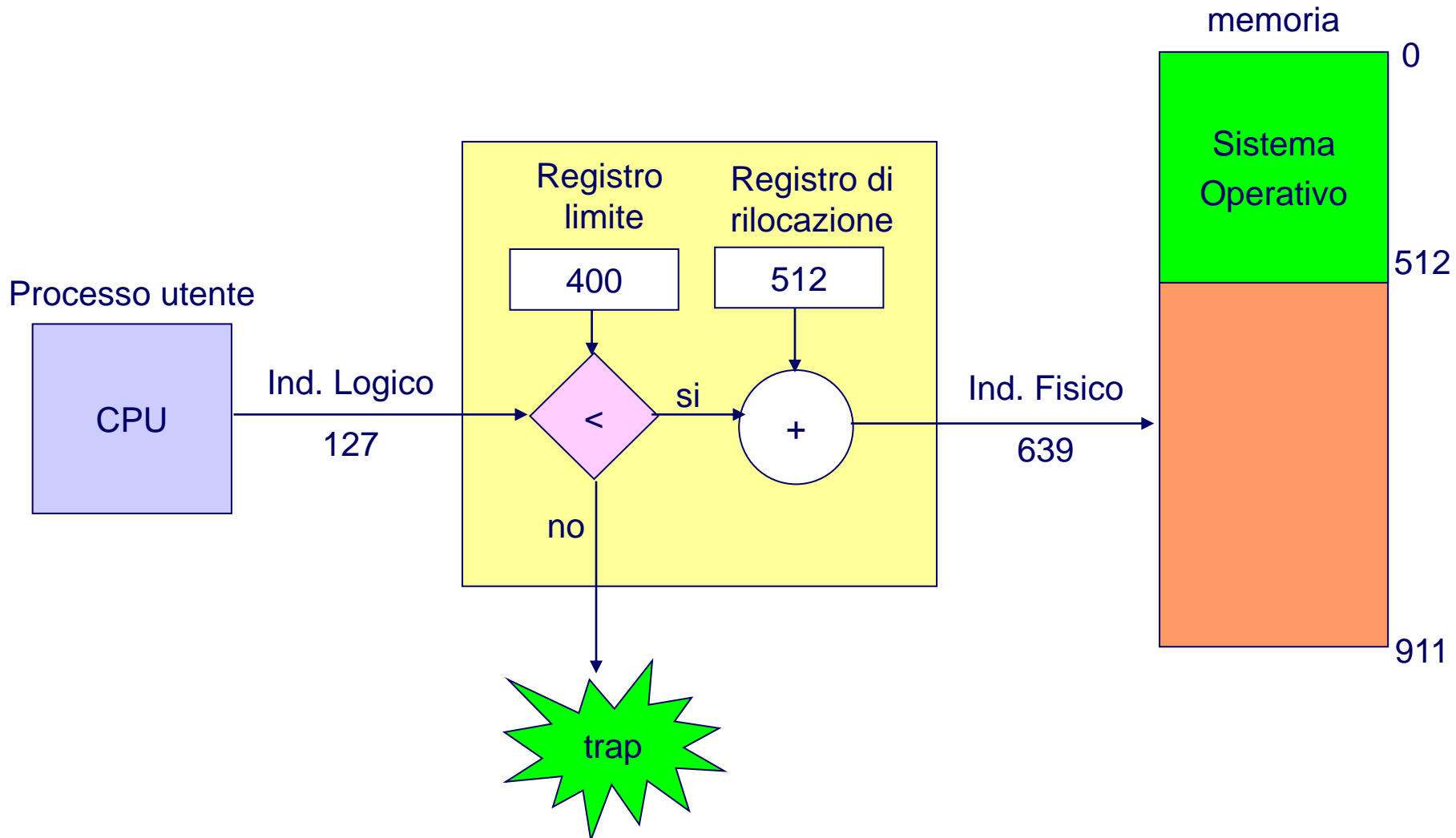
Primo approccio molto semplificato: la parte di **memoria disponibile** per l'allocazione dei processi di utente **non è partizionata**:

- ❑ **un solo processo alla volta** può essere allocato in memoria: non c'è multiprogrammazione

Di solito:

- ❑ SO risiede nella **memoria bassa** $[0, \text{max}]$
- ❑ necessità di **proteggere codice e dati di SO** da accessi di processi utente:
 - uso del **registro di rilocalizzazione** ($\text{RL} = \text{max} + 1$) per garantire la correttezza degli accessi

Allocazione contigua a partizione singola



Allocazione contigua: partizioni multiple

Multiprogrammazione → necessità di **proteggere** codice e dati di ogni processo

Partizioni multiple: ad ogni processo caricato viene associata **un'area di memoria distinta (partizione)**

- ❑ partizioni **fisse**
- ❑ partizioni **variabili**

Allocazione contigua: partizioni multiple

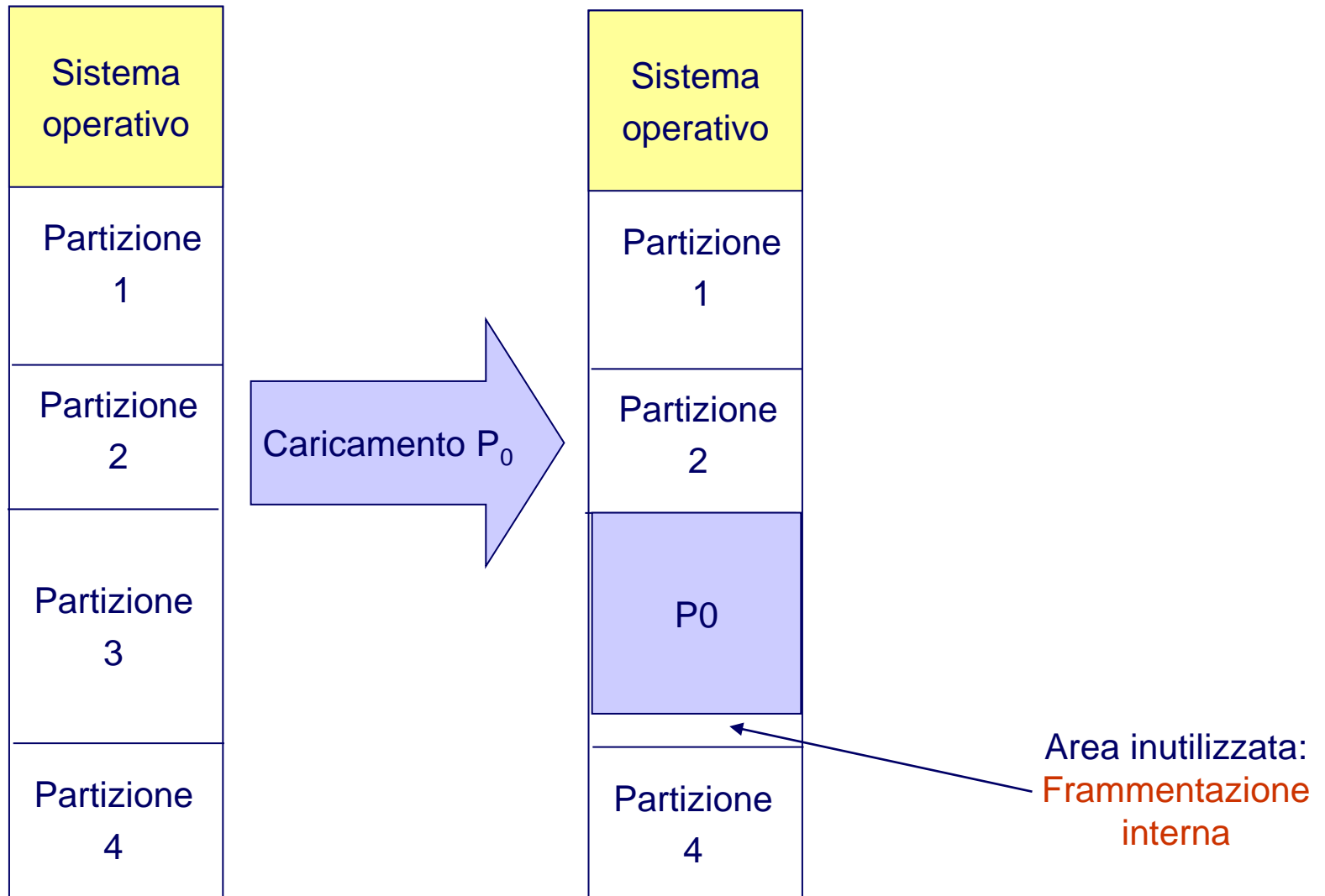
Partizioni fisse (MFT, Multiprogramming with Fixed number of Tasks): dim di ogni partizione fissata a priori

- ❑ ogni partizione ha dimensione prefissata, eventualmente diversa da partizione a partizione
- ❑ quando un processo viene schedulato, SO cerca una partizione libera di dim sufficiente

Problemi:

- ❑ **frammentazione interna**; sottoutilizzo della partizione
- ❑ **grado di multiprogrammazione limitato** al numero di partizioni
- ❑ **dim massima** dello spazio di indirizzamento di un processo limitata da dim della **partizione più estesa**

Partizioni fisse



Partizioni variabili

Partizioni variabili (MVT, Multiprogramming with Variable number of Tasks): ogni partizione allocata dinamicamente e dimensionata in base a dim processo da allocare

- quando un processo viene schedulato, SO cerca un'area sufficientemente grande per allocarvi dinamicamente la partizione associata

Partizioni variabili

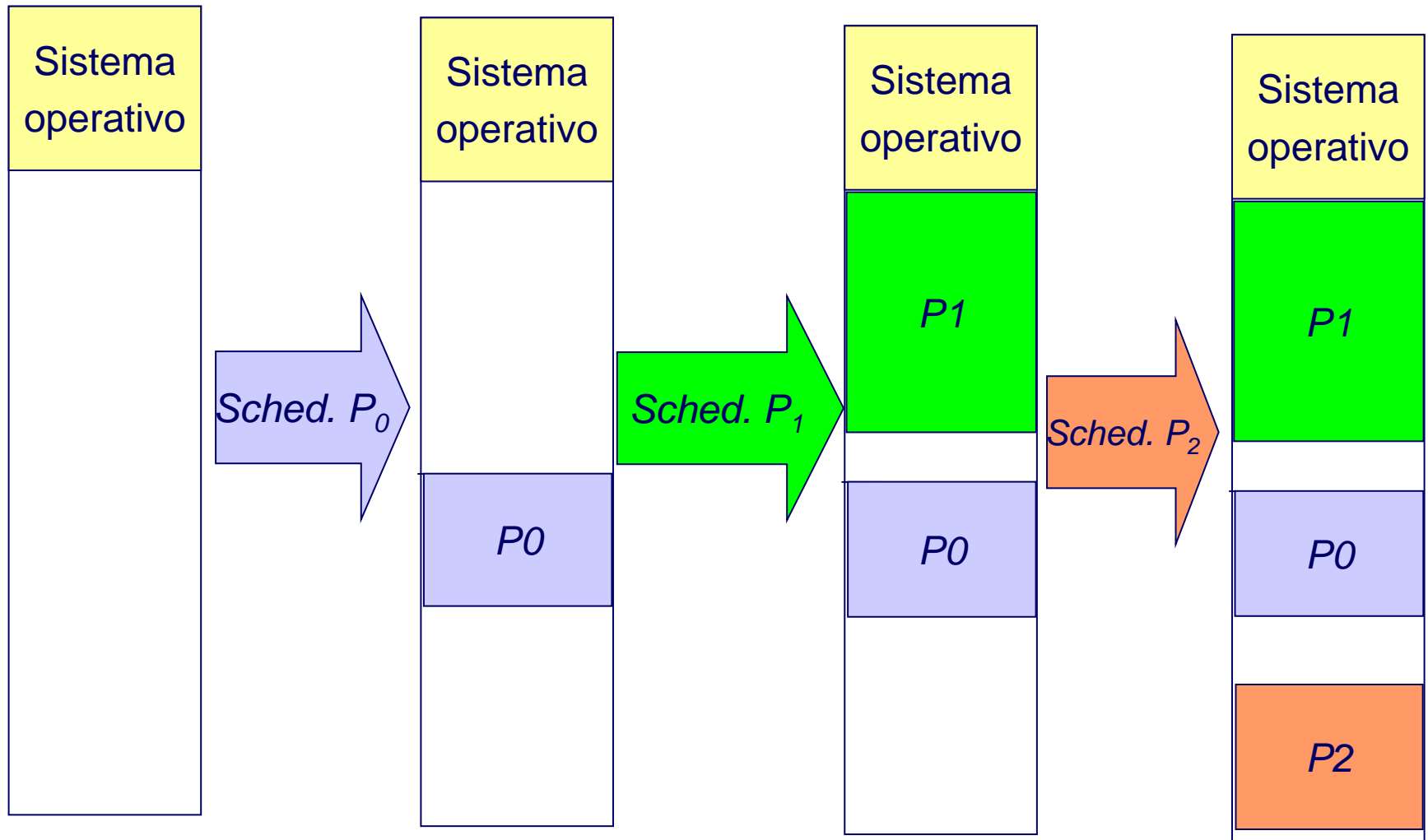
Vantaggi (rispetto a MFT):

- elimina **frammentazione interna** (ogni partizione è della esatta dimensione del processo)
- **grado di multiprogrammazione variabile**
- dimensione massima dello spazio di indirizzamento di ogni processo limitata da dim spazio fisico

Problemi:

- scelta dell'area in cui allocare: **best fit, worst fit, first fit, ...**
- **frammentazione esterna** - man mano che si allocano nuove partizioni, la memoria libera è sempre più frammentata
 - → **necessità di compattazione periodica**

Partizioni variabili



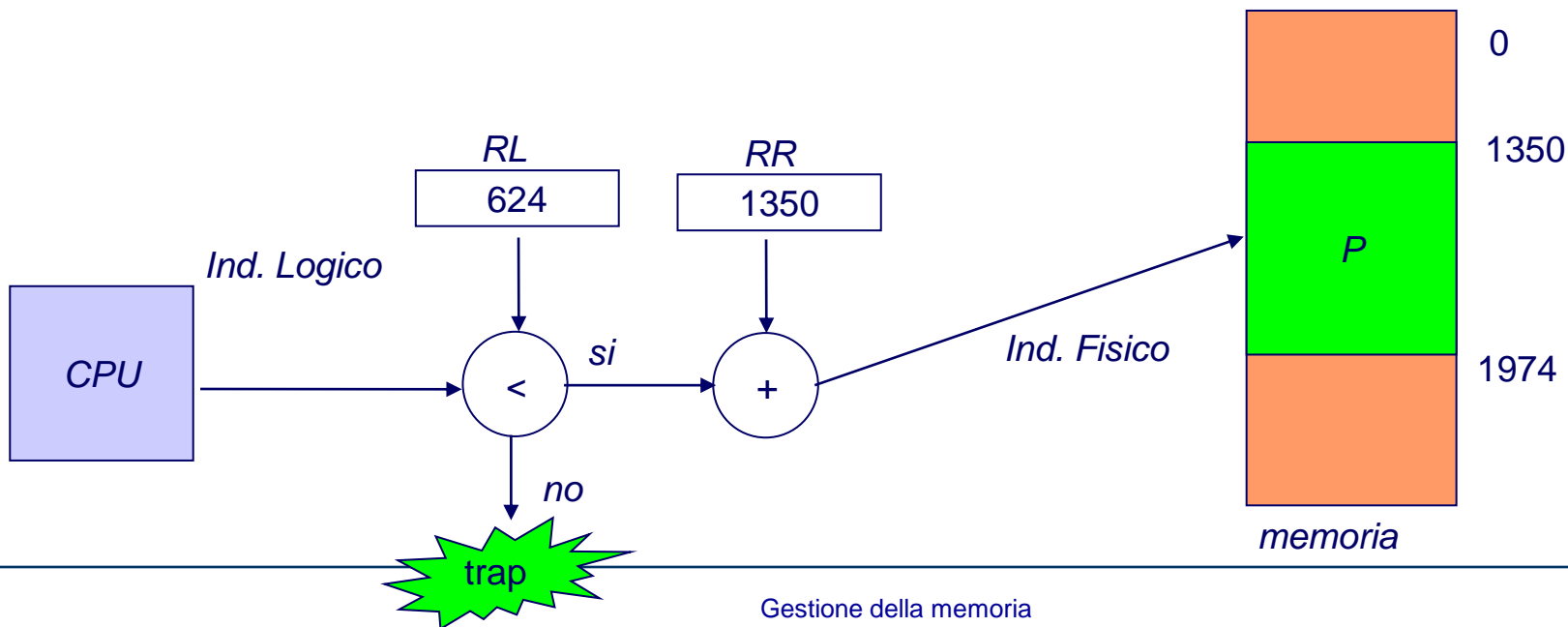
Partizioni & protezione

Protezione realizzata a livello HW mediante:

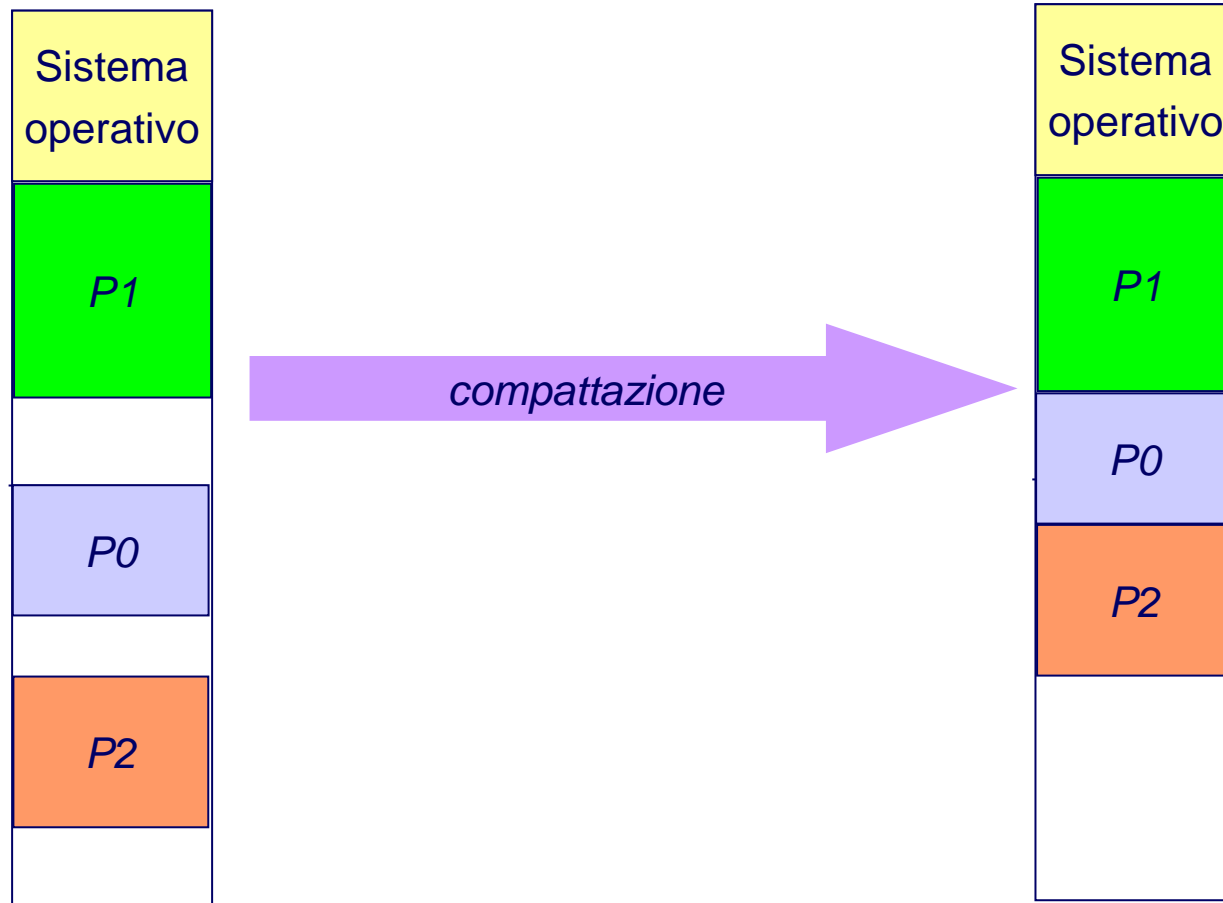
- **registro di rilocazione RR**
- **registro limite RL**

Ad ogni processo è associata una coppia di valori $\langle V_{RR}, V_{RL} \rangle$

Quando un processo P viene schedulato, il dispatcher carica RR e RL con i valori associati al processo $\langle V_{RR}, V_{RL} \rangle$



Compattazione



Problema: possibile crescita dinamica dei processi
➡ **mantenimento dello spazio di crescita**

Paginazione

Allocazione contigua a partizioni multiple: il problema principale è la **frammentazione esterna**

Allocazione non contigua → paginazione

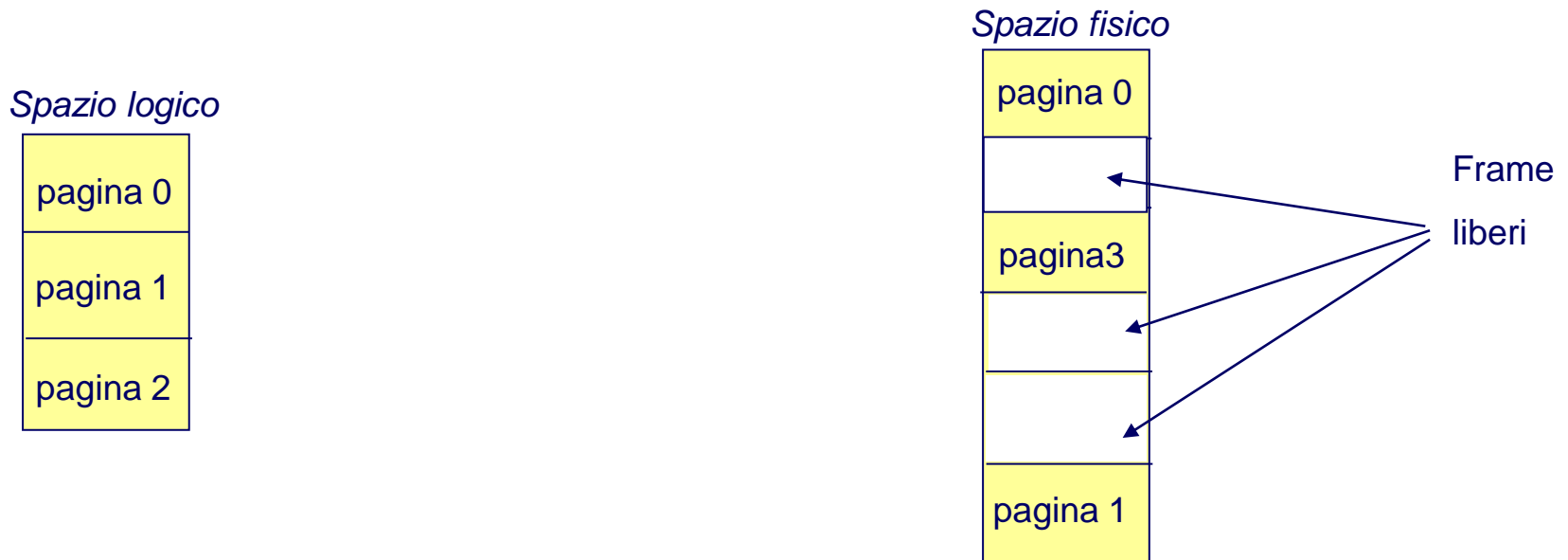
- ❑ **eliminazione frammentazione esterna**
- ❑ **riduzione forte di frammentazione interna**

Idea di base: **partizionamento spazio fisico di memoria in pagine (frame) di dim costante** e limitata (ad es. 4KB) sulle quali mappare **porzioni** dei processi da allocare

Paginazione

- **Spazio fisico:** insieme di frame di dim D_f costante prefissata
- **Spazio logico:** insieme di pagine di dim uguale a D_f

ogni pagina logica di un processo caricato in memoria viene mappata su una pagina fisica in memoria centrale



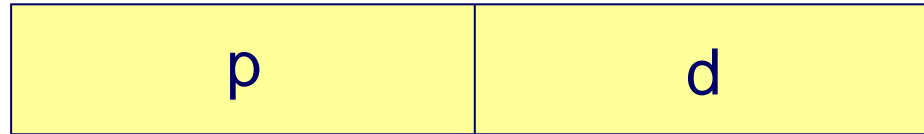
Paginazione

Vantaggi

- ❑ Pagine logiche contigue possono essere allocate su pagine fisiche non contigue: ***non c'è frammentazione esterna***
- ❑ Le pagine sono di dim limitata: ***frammentazione interna*** per ogni processo ***limitata dalla dimensione del frame***
- ❑ È possibile caricare in memoria un ***sottoinsieme delle pagine logiche di un processo*** (*vedi memoria virtuale nel seguito*)

Supporto HW a Paginazione

Struttura dell'indirizzo logico (m bit):



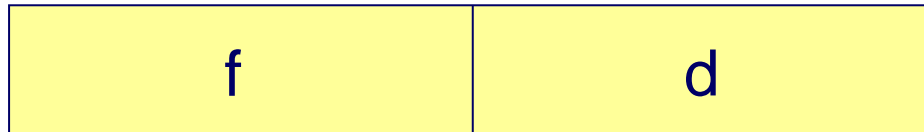
- **p** numero di pagina logica ($m-n$ bit)
- **d** offset della cella rispetto all'inizio della pagina (n bit)

Hp: indirizzi logici di m bit (n bit per offset, e $m-n$ per la pagina)

- dim massima dello spazio logico di indirizzamento $\rightarrow 2^m$
- dim della pagina $\rightarrow 2^n$
- numero di pagine $\rightarrow 2^{m-n}$

Supporto HW a Paginazione

Struttura dell'indirizzo fisico:



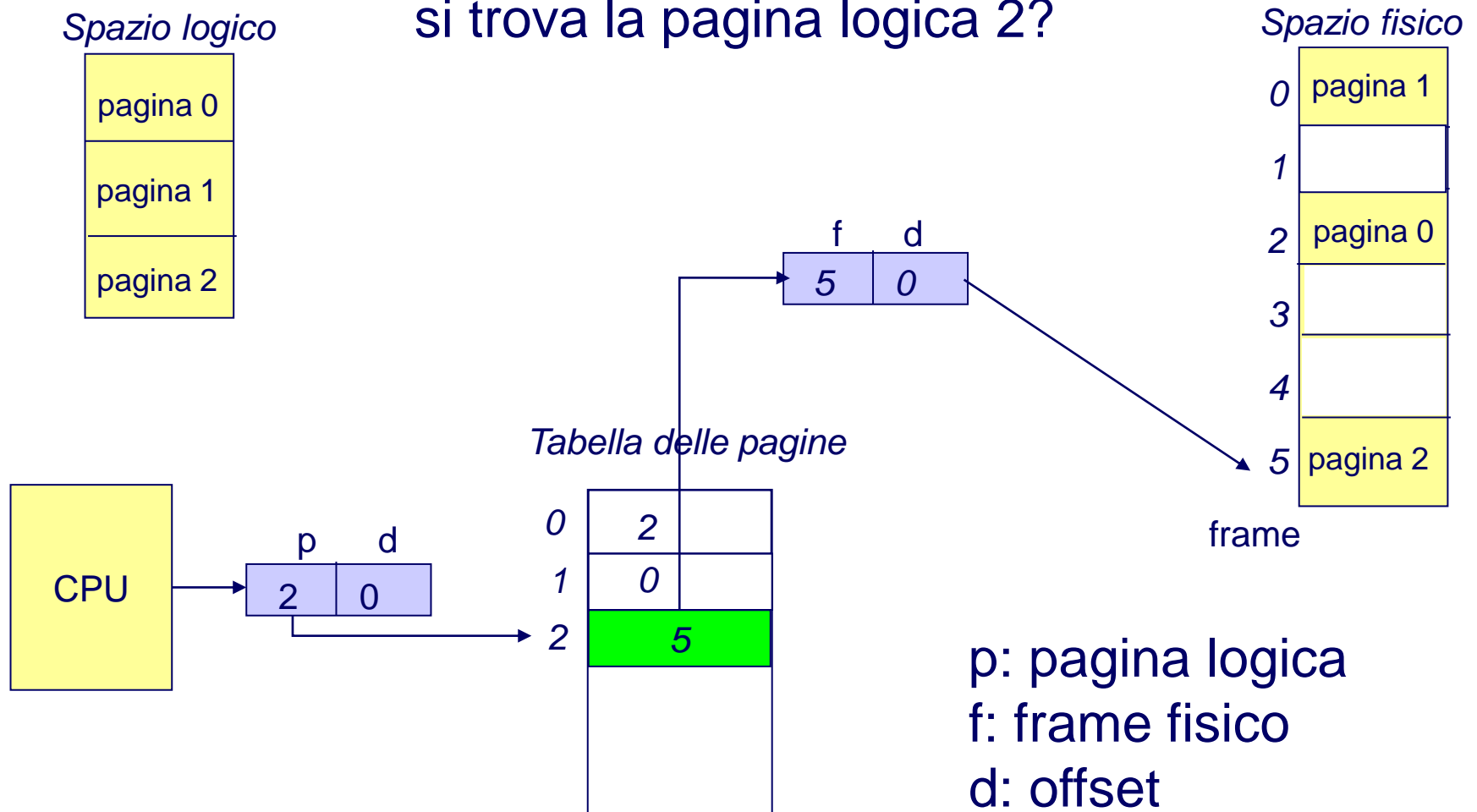
- **f** numero di frame (pagina fisica)
- **d** offset della cella rispetto all'inizio del frame

Binding tra indirizzi logici e fisici può essere realizzato mediante **tabella delle pagine** (associata al processo):

- ❑ **a ogni pagina logica associa la pagina fisica** verso la quale è realizzato il mapping

Supporto HW a paginazione: tabella delle pagine

In quale frame fisico
si trova la pagina logica 2?



Realizzazione della tabella delle pagine

Problemi da affrontare

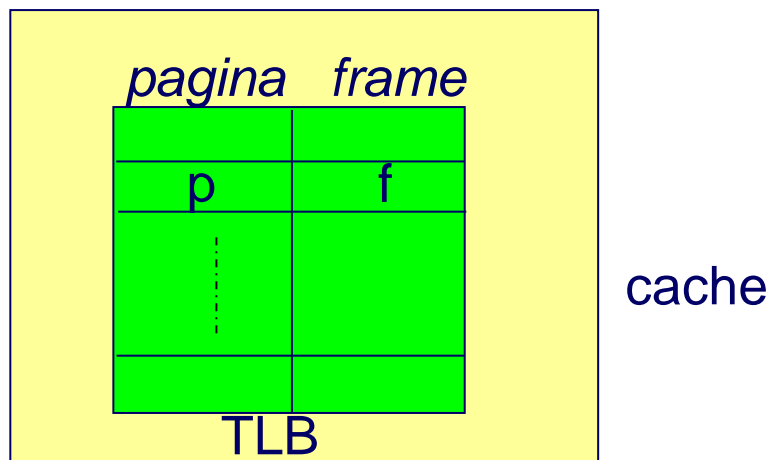
- ❑ **tabella** può essere molto **grande**
- ❑ **traduzione** (ind.logico → ind. fisico) deve essere il **più veloce possibile**

Varie soluzioni

1. Su **registri di CPU**
 - accesso veloce
 - **cambio di contesto pesante**
 - **dimensioni limitate** della tabella
2. In **memoria centrale**: registro PageTableBaseRegister (PTBR) memorizza collocazione tabella in memoria
 - 2 accessi in memoria per ogni operazione (load, store)
3. Uso di **cache**: (**Translation Look-aside Buffers, TLB**) per velocizzare l'accesso

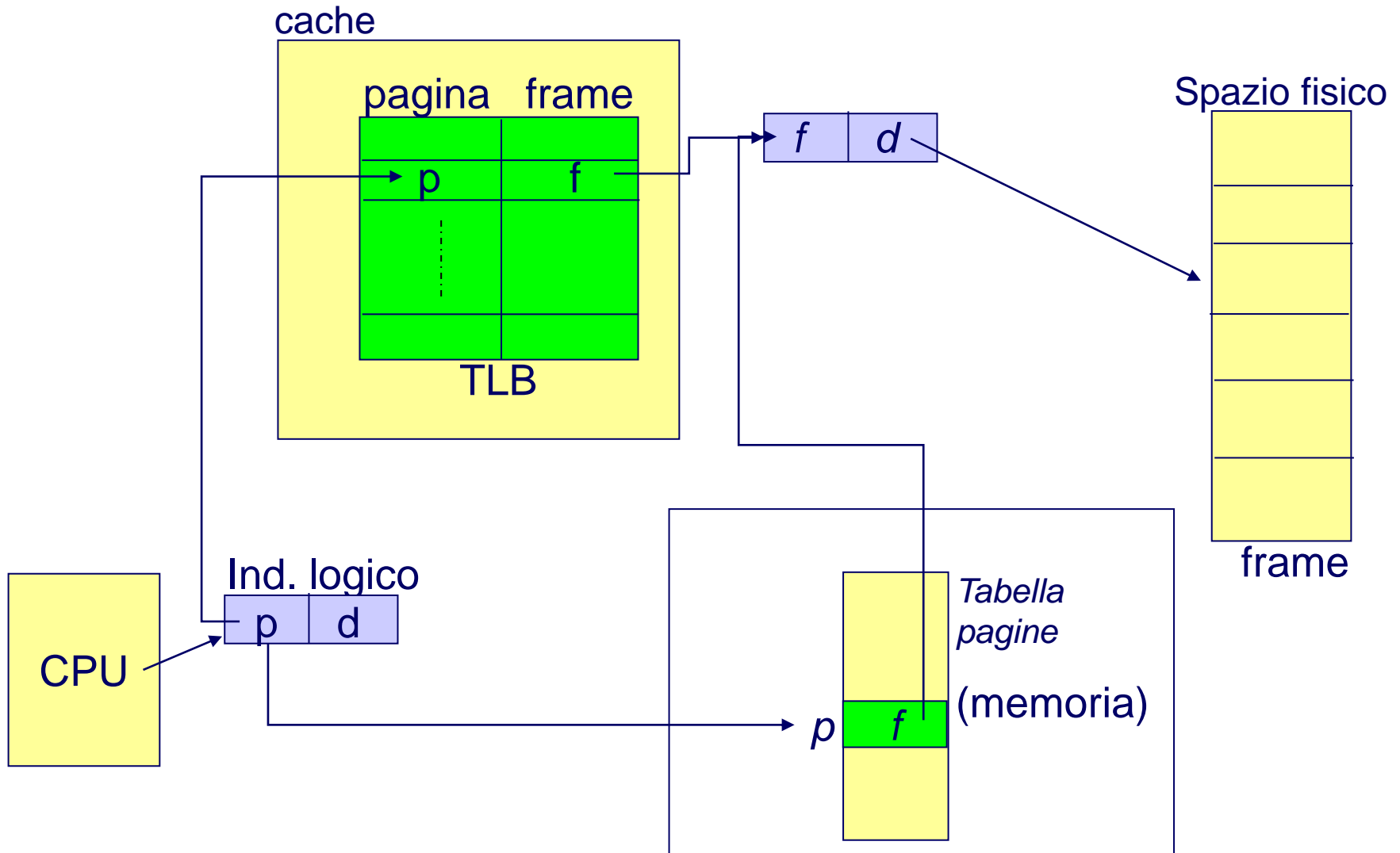
Translation Look-aside Buffers (TLB)

- Tabella delle pagine è allocata **in memoria centrale**
- **Una parte della tabella** delle pagine (di solito, le pagine accedute più di frequente o più di recente) è **copiata in cache: TLB**



Se la coppia (p,f) è già presente in cache l'accesso è veloce; altrimenti SO deve trasferire la coppia richiesta **dalla tabella delle pagine** (in memoria centrale) **in TLB**

Supporto HW a paging: tabella pagine con TLB



Gestione TLB

- TLB inizialmente vuoto
- mentre l'esecuzione procede, viene **gradualmente riempito con indirizzi pagine già accedute**

HIT-RATIO: percentuale di volte che una pagina viene trovata in TLB

- Dipende da **dimensione TLB** (Intel486: **98%**)

Paginazione & protezione

La tabella delle pagine

- ❑ ha **dimensione fissa**
- ❑ non sempre viene utilizzata completamente

Come distinguere gli elementi significativi da quelli non utilizzati?

- **Bit di validità**: ogni elemento contiene un bit
 - se è a 1, **entry valida** (pagina appartiene allo spazio logico del processo)
 - se è 0, **entry non valida**
- **Page Table Length Register**: registro che contiene il **numero degli elementi validi** nella tabella delle pagine

In aggiunta, per ogni entry della tabella delle pagine, possono esserci uno o più **bit di protezione che esprimono le modalità di accesso alla pagina** (es. read-only)

Paginazione a più livelli

Lo spazio logico di indirizzamento di un processo può essere molto esteso:

- ❑ **elevato numero di pagine**
- ❑ **tabella delle pagine di grandi dimensioni**

Ad esempio:

- ❑ per ipotesi, indirizzi di 32 bit → spazio logico di 4GB
dimensione pagina 4KB (2^{12})
- ❑ la tabella delle pagine dovrebbe contenere $2^{32}/2^{12}$
elementi → 2^{20} elementi (circa 1M)

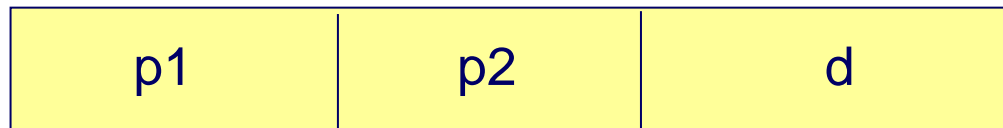
Paginazione a più livelli: allocazione non contigua anche della tabella delle pagine → si applica ancora la tecnica di paginazione alla tabella delle pagine!

Esempio: paginazione a due livelli

Vengono utilizzati **due livelli di tabelle delle pagine**

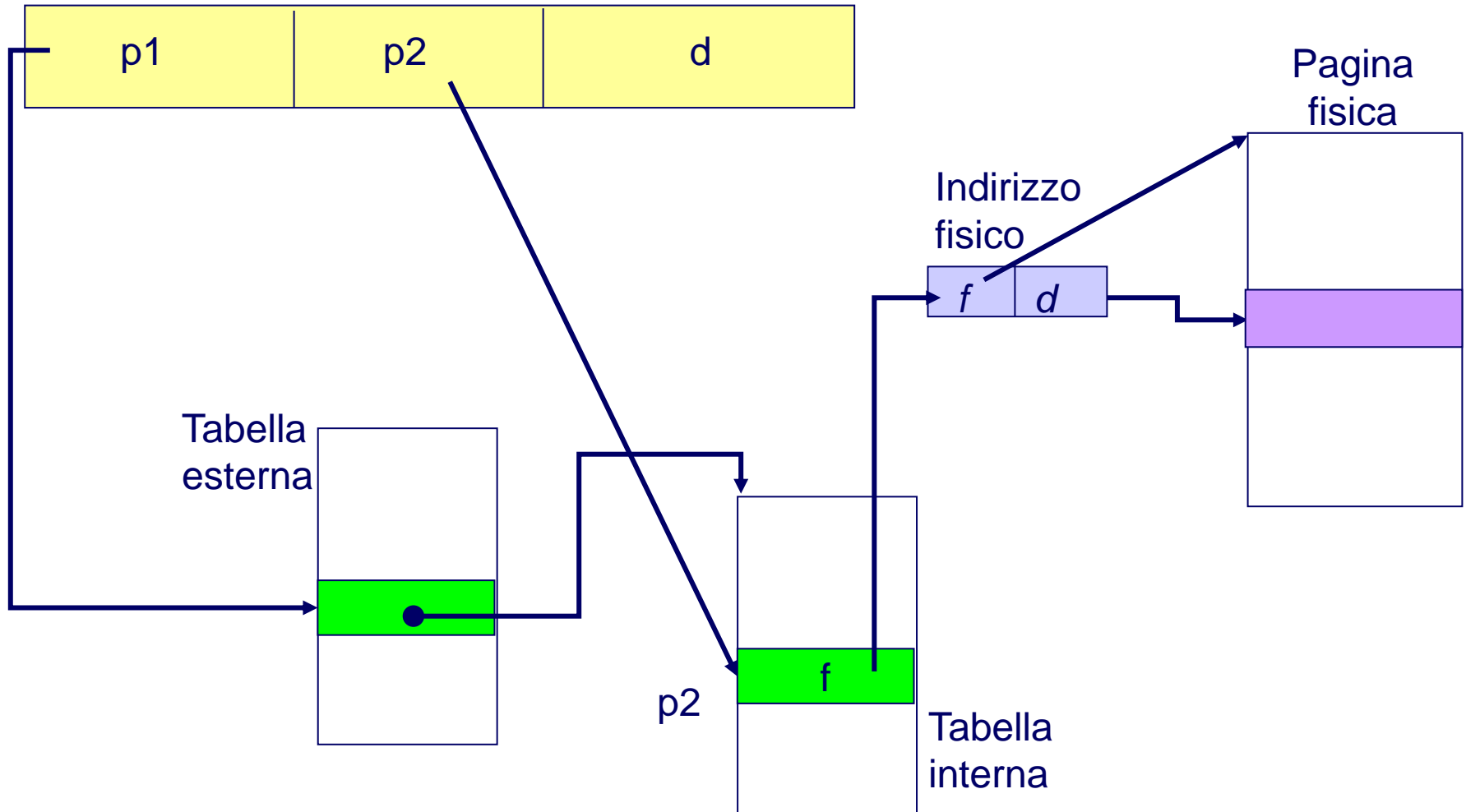
- **primo livello (tabella esterna)**: contiene gli indirizzi delle tabelle delle pagine collocate al secondo livello (tabelle interne)

Struttura dell'indirizzo logico:



- ❑ **p1** indice di pagina nella tavola esterna
- ❑ **p2** offset nella tavola interna
- ❑ **d** offset cella all'interno della pagina fisica

Esempio: paginazione a due livelli



Paginazione a più livelli

Vantaggi

- possibilità di **indirizzare spazi logici di dimensioni elevate** riducendo i problemi di allocazione delle tabelle
- possibilità di mantenere **in memoria soltanto le pagine** della tabella **che servono**

Svantaggio

- **tempo di accesso più elevato**: per tradurre un indirizzo logico sono necessari più accessi in memoria (ad esempio, 2 livelli di paginazione → 2 accessi)

Tabella delle pagine invertita

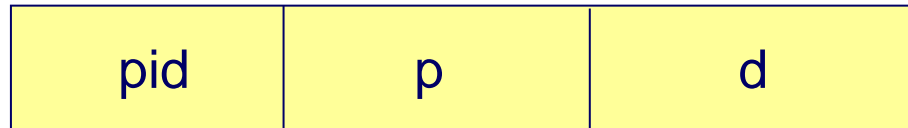
Per limitare l'occupazione di memoria, in alcuni SO si usa **un'unica struttura dati globale** che ha un elemento per ogni frame:

tabella delle pagine invertita

Ogni elemento della tabella delle pagine invertita **rappresenta un frame (indirizzo pari alla posizione nella tabella)** e, in caso di frame allocato, contiene:

- **pid**: identificatore del processo a cui è assegnato il frame
- **p**: numero di pagina logica

La struttura dell'indirizzo logico è, quindi:



d è l'offset all'interno della pagina

Tabella delle pagine invertita

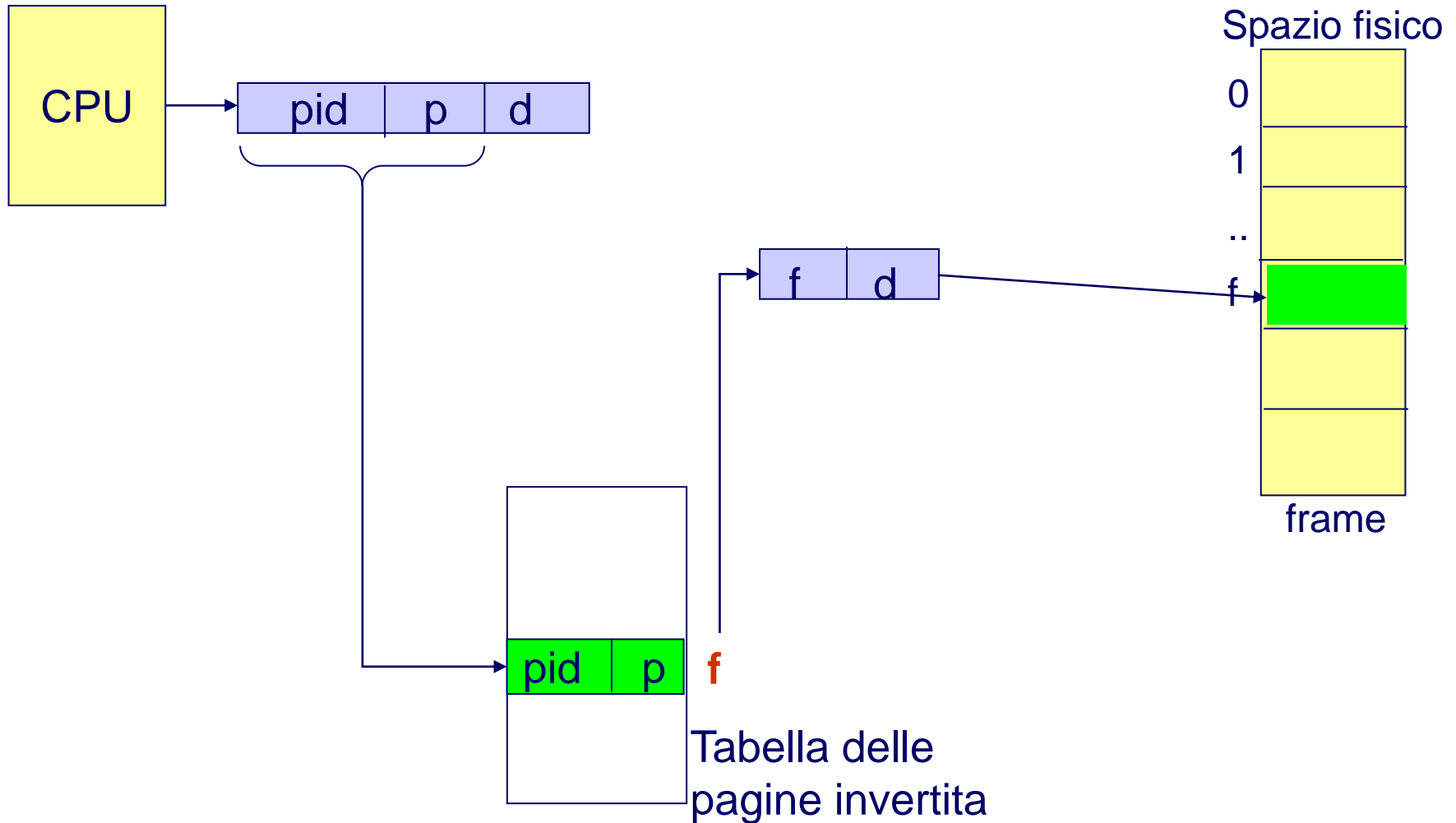


Tabella delle pagine invertita

Per tradurre un indirizzo logico $\langle \text{pid}, p, d \rangle$:

- ricerca nella tabella dell'elemento che contiene la coppia $(\text{pid}, p) \rightarrow$ l'indice dell'elemento trovato rappresenta il **numero del frame allocato alla pagina logica p**

Problemi

- **tempo di ricerca** nella tabella invertita
- difficoltà di realizzazione della **condivisione di codice tra processi (rientranza)**: come associare un frame a più pagine logiche di processi diversi?

Organizzazione della memoria in segmenti

La segmentazione si basa sul **partizionamento dello spazio logico degli indirizzi di un processo in parti (segmenti)**, ognuna caratterizzata da nome e lunghezza

- **Divisione semantica per funzione:** ad esempio
 - codice
 - dati
 - stack
 - heap
- Non è stabilito un ordine tra i segmenti
- Ogni segmento allocato in memoria in modo contiguo
- Ad ogni segmento SO associa un intero attraverso il quale lo si può riferire

Segmentazione

Struttura degli indirizzi logici: ogni indirizzo è costituito dalla coppia **<segmento, offset>**

- ❑ **segmento:** numero che individua il segmento nel sistema
- ❑ **offset:** posizione cella all'interno del segmento

Supporto HW alla segmentazione

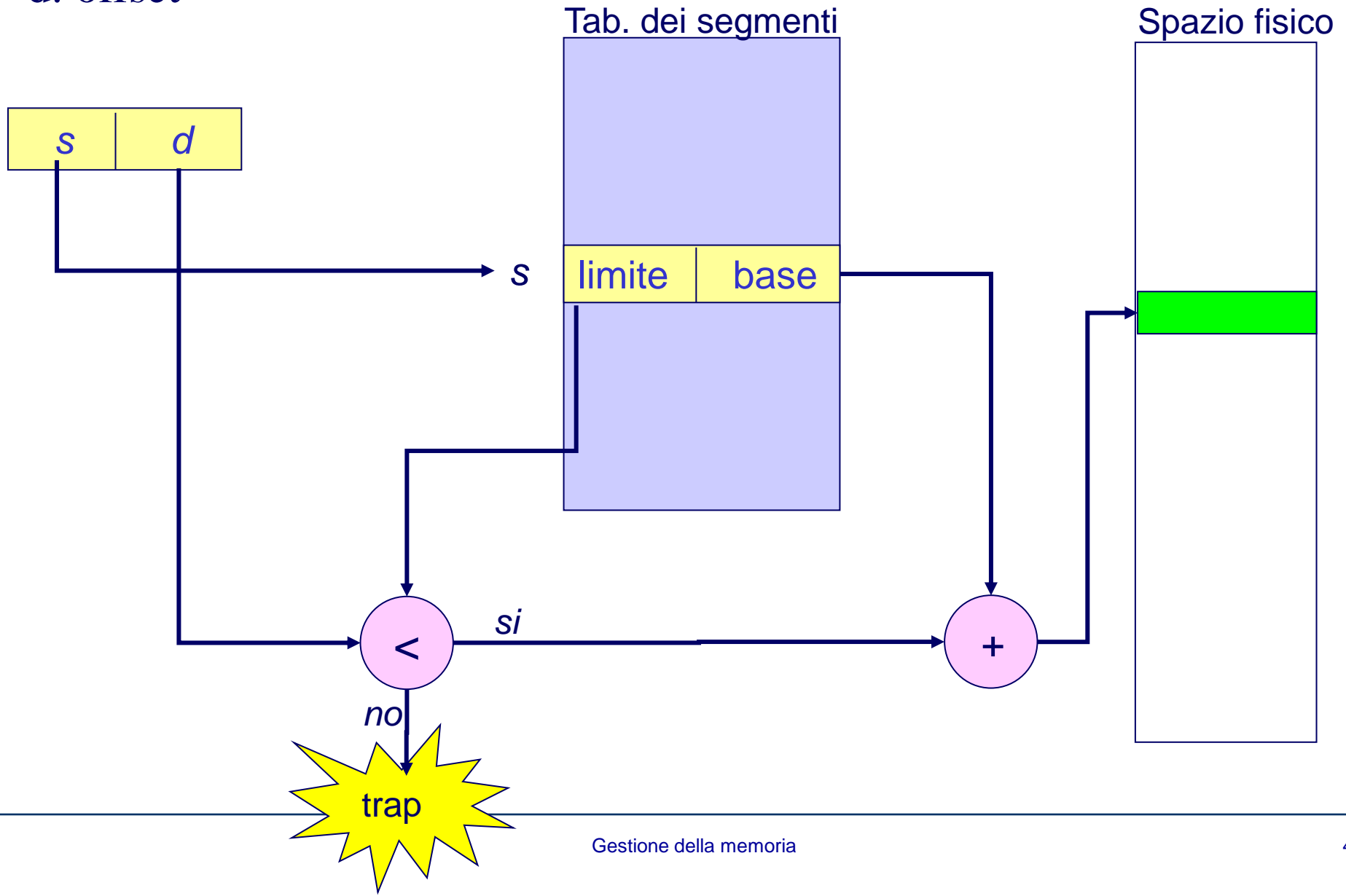
Tabella dei segmenti: ha una entry per ogni segmento che ne descrive l'allocazione in memoria fisica mediante la coppia **<base, limite>**

- ❑ **base:** indirizzo prima cella del segmento nello spazio fisico
- ❑ **limite:** indica la dimensione del segmento

Segmentazione

s: # segmento

d: offset



Realizzazione della tabella dei segmenti

Tabella globale: possibilità di **dimensioni elevate**

Realizzazione

- Su **registri** di CPU
- In **memoria**, mediante **registri base** (Segment Table Base Register, STBR) e **limite** (Segment Table Length Register, STLR)
- Su **cache** (solo l'insieme dei segmenti usati più recentemente)

Segmentazione

Estensione della tecnica di **allocazione a partizioni variabili**

- ❑ partizioni variabili: 1 segmento/processo
- ❑ **segmentazione: più segmenti/processo**

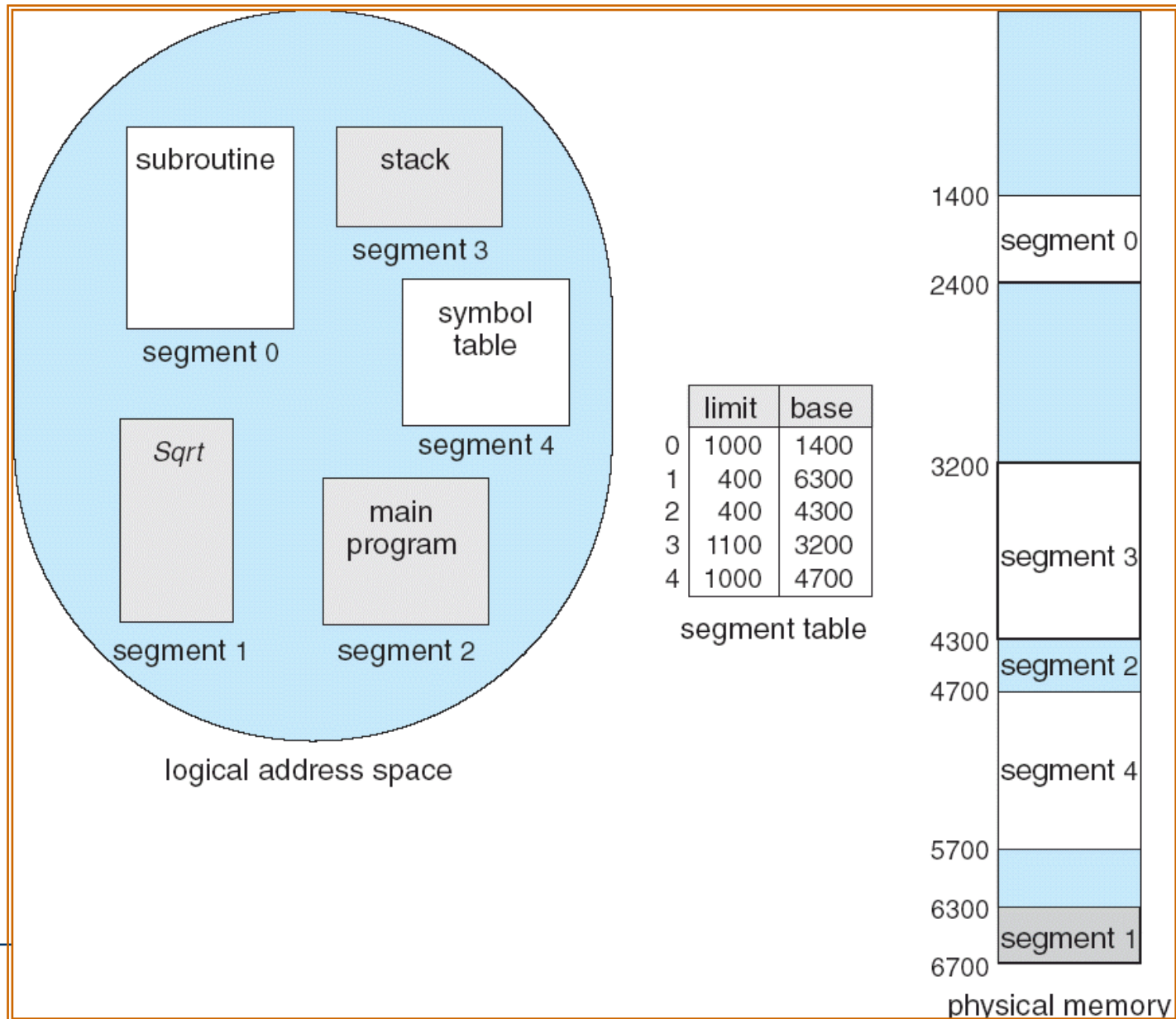
Problema principale:

- ❑ come nel caso delle partizioni variabili,
frammentazione esterna

Soluzione: allocazione dei segmenti con tecniche

- ❑ **best fit**
- ❑ **worst fit**
- ❑ ...

Esempio di segmentazione



Segmentazione paginata

Segmentazione e paginazione possono essere combinate (ad esempio Intel x86):

- ❑ spazio logico segmentato (specialmente per motivi di protezione)
- ❑ ogni segmento suddiviso in pagine

Vantaggi:

- ❑ **eliminazione della frammentazione esterna** (ma introduzione di frammentazione interna...)
- ❑ non necessario mantenere in memoria l'intero segmento, ma è possibile caricare **soltanto le pagine necessarie** (vedi memoria virtuale nel seguito)

Strutture dati:

- ❑ tabella dei segmenti
- ❑ una tabella delle pagine per ogni segmento

Ad esempio, segmentazione in Linux

Linux adotta una gestione della memoria basata su **segmentazione paginata**

Vari tipi di segmento:

- **code** (kernel, user)
- **data** (kernel, user)
- **task state segments** (registri dei processi per il cambio di contesto)
- ...

I segmenti sono paginati con **paginazione a tre livelli**

Memoria virtuale

La **dimensione della memoria** può rappresentare un vincolo importante, riguardo a

- dimensione dei processi
- grado di multiprogrammazione

Può essere desiderabile un sistema di gestione della memoria che:

- ❑ consenta la presenza di **più processi** in memoria (ad es. partizioni multiple, paginazione e segmentazione), **indipendentemente dalla dimensione dello spazio disponibile**
- ❑ svincoli il **grado di multiprogrammazione** dalla dimensione effettiva della memoria

→ **Memoria virtuale**

Memoria virtuale

Con le tecniche viste finora

- ❑ **l'intero spazio logico** di ogni processo è **allocato in memoria**

oppure

- ❑ **overlay, caricamento dinamico**: si possono allocare/deallocare parti dello spazio di indirizzi
→ **a carico del programmatore**

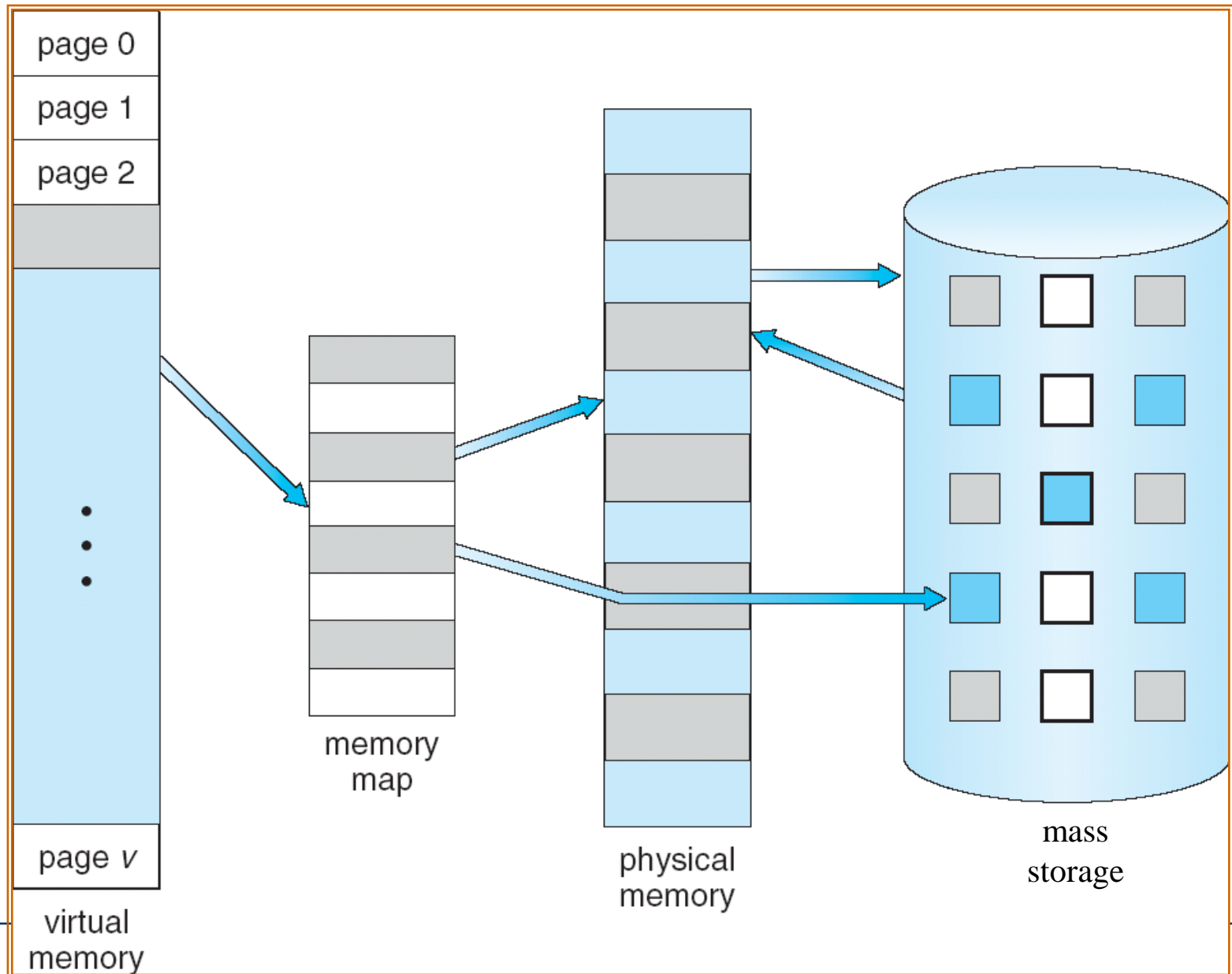
Memoria Virtuale: è un metodo di gestione della memoria che consente **l'esecuzione di processi non completamente allocati** in memoria

Memoria virtuale

Vantaggi:

- ❑ **dimensione spazio logico degli indirizzi non vincolata** dall'estensione della memoria
- ❑ **grado di multiprogrammazione indipendente** dalla dimensione della memoria fisica
- ❑ **efficienza**: caricamento di un processo e swapping hanno un costo più limitato (meno I/O)
- ❑ **astrazione**: il programmatore non deve preoccuparsi dei vincoli relativi alla dimensione della memoria

Memoria virtuale più ampia di memoria fisica



Paginazione su richiesta

Di solito la memoria virtuale è realizzata mediante tecniche di **paginazione su richiesta**:

- ❑ **tutte le pagine di ogni processo risiedono in memoria di massa**; durante l'esecuzione alcune di esse vengono **trasferite, all'occorrenza**, in memoria centrale

Pager: modulo del SO che realizza i **trasferimenti delle pagine da/verso memoria secondaria/ centrale** ("swapper" di pagine)

- ❑ **paginazione su richiesta (o "su domanda")**: **pager lazy ("pigro")** trasferisce in memoria centrale una pagina soltanto **se ritenuta necessaria**

Paginazione su richiesta

Esecuzione di un processo può richiedere swap-in del processo

- ❑ **swapper**: gestisce i trasferimenti di **interi processi** (mem.centrale \leftrightarrow mem. secondaria)
- ❑ **pager**: gestisce i trasferimenti di singole pagine

Prima di eseguire swap-in di un processo:

- ❑ **pager può prevedere** le pagine di cui (**probabilmente**) il processo avrà bisogno **inizialmente** \rightarrow **caricamento**

HW necessario:

- ❑ tabella delle pagine (con PTBR, PTLR, e/o TLB, ...)
- ❑ **memoria secondaria** e strutture necessarie per la sua gestione (uso di dischi veloci)

Paginazione su richiesta

Quindi, in generale, una pagina dello spazio logico di un processo:

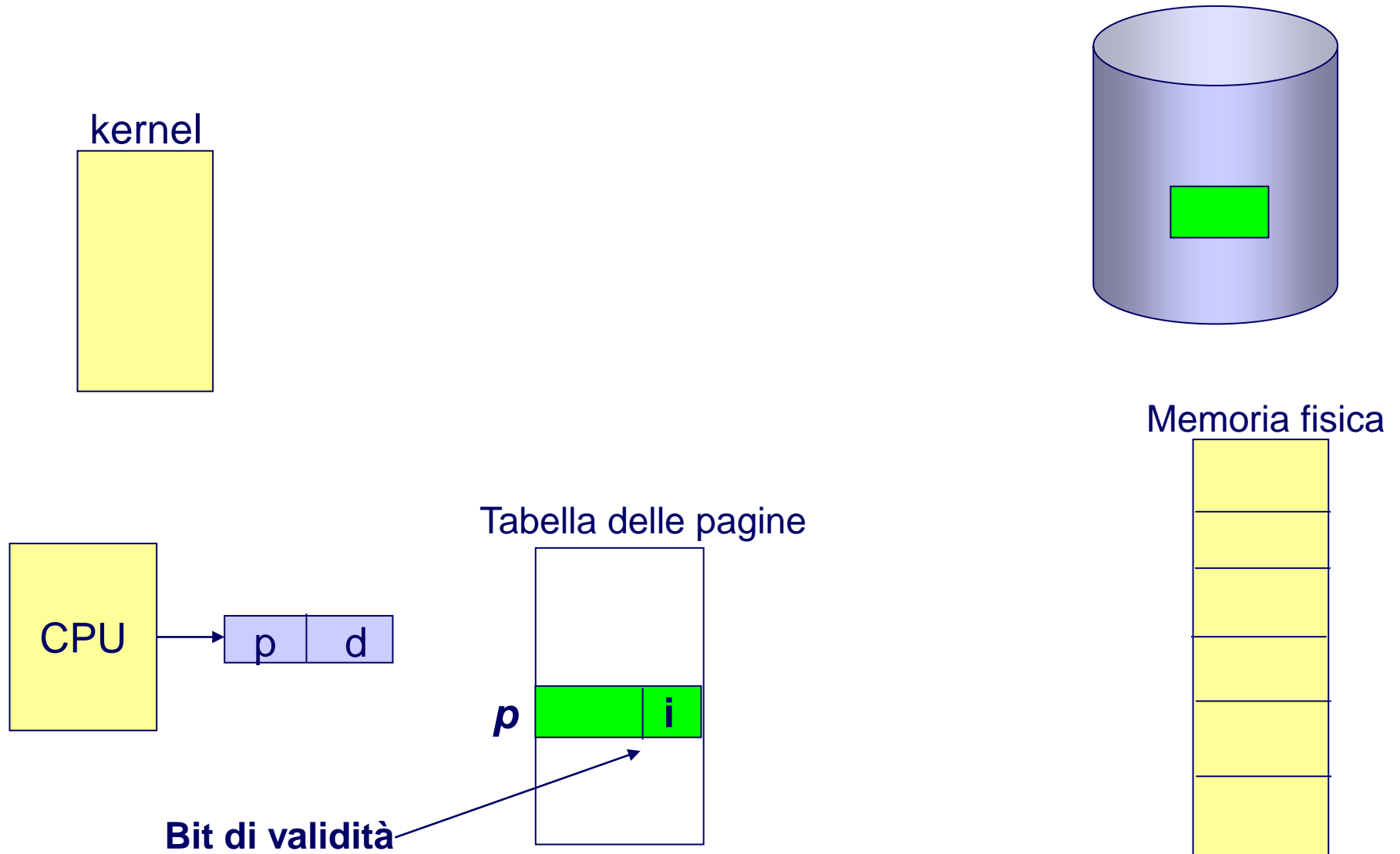
- può essere ***allocata in memoria centrale***
- può essere ***in memoria secondaria***

Come distinguere i due casi?

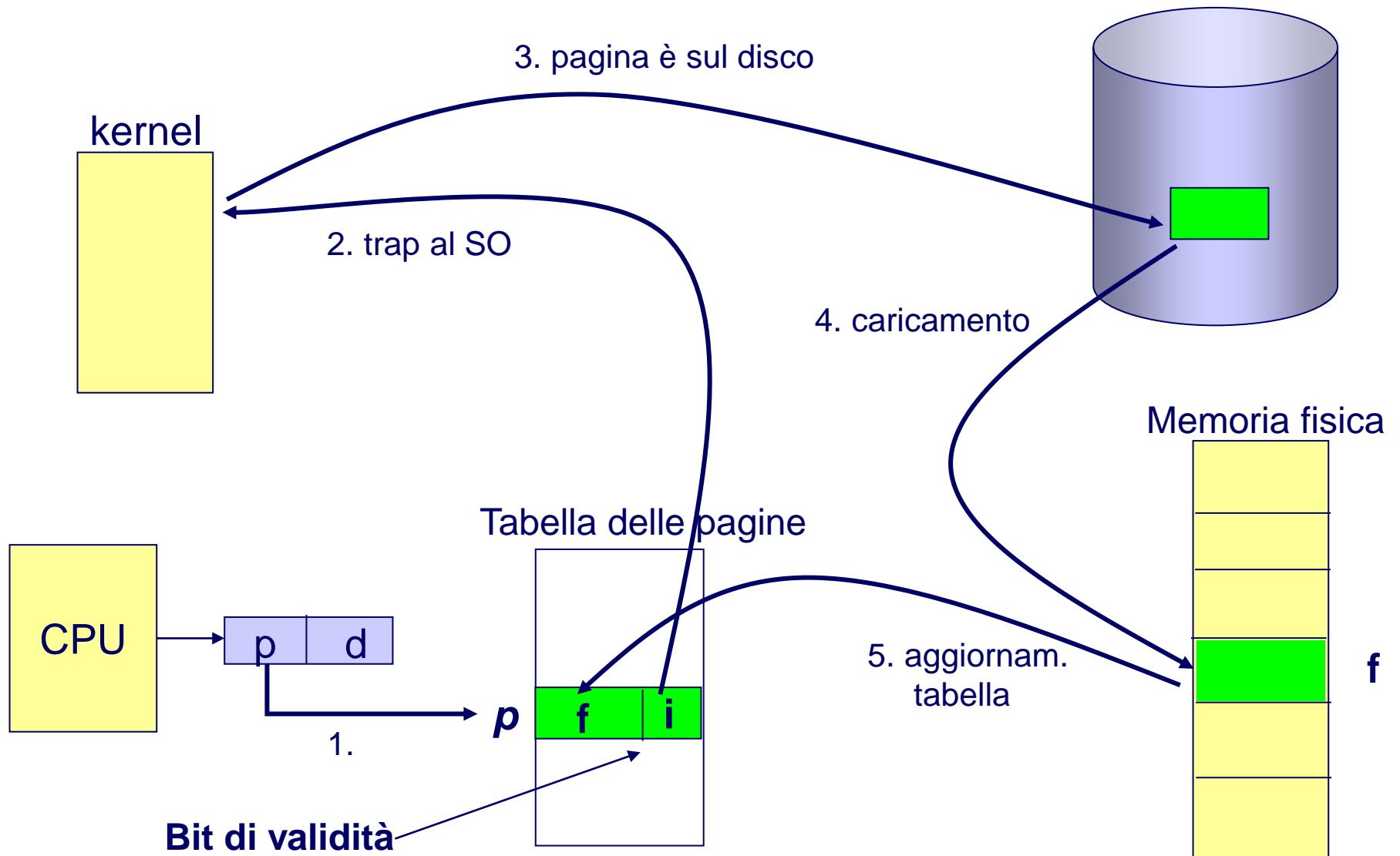
La tabella delle pagine contiene **bit di validità**:

- se la pagina è ***presente in memoria centrale***
- se è ***in memoria secondaria*** oppure è ***invalida*** (\notin spazio logico del processo)
→ **interruzione al SO (*page fault*)**

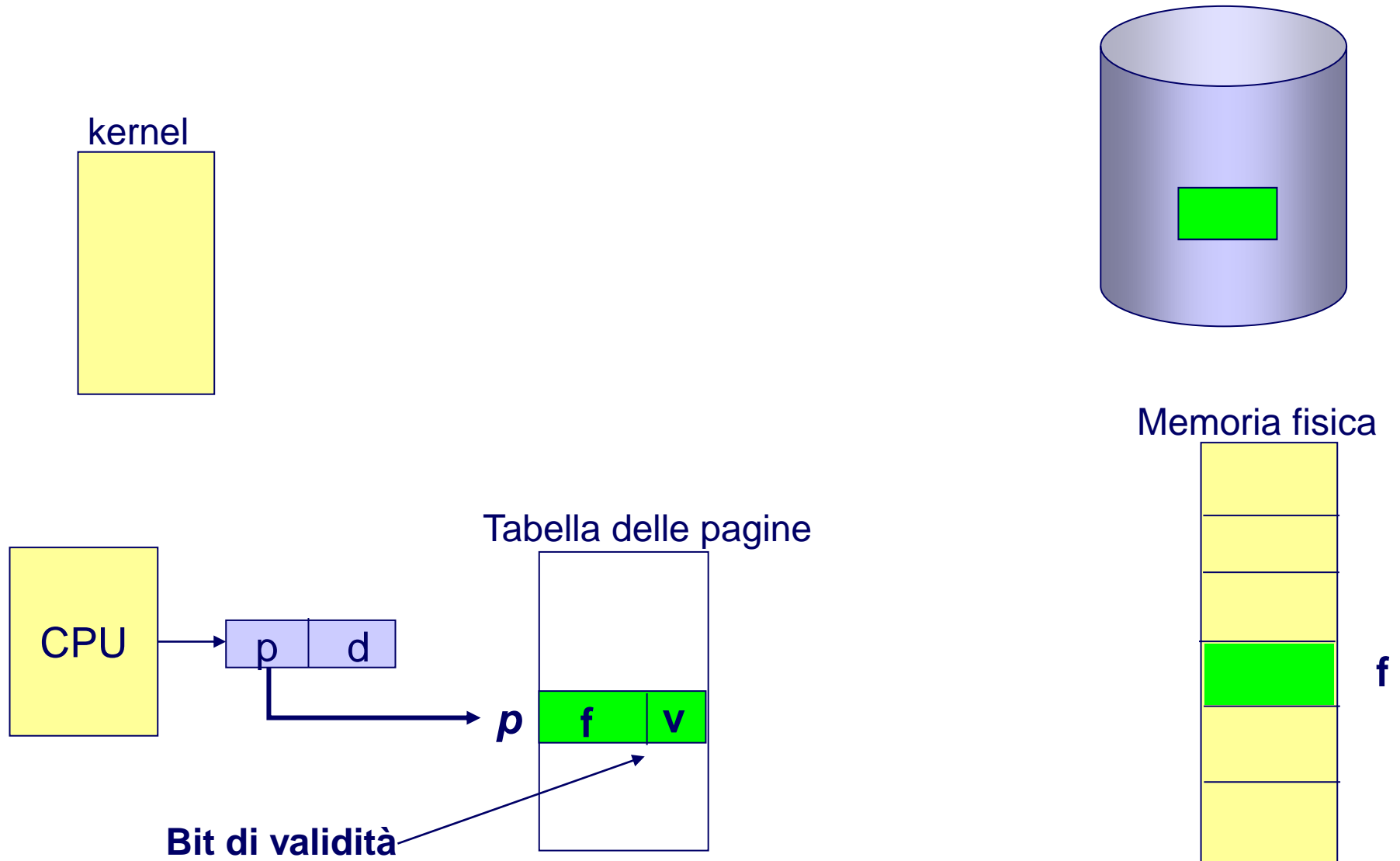
Paginazione su richiesta: page fault



Paginazione su richiesta: page fault



Page fault: situazione finale



Trattamento page fault

Quando kernel SO riceve l'interruzione dovuta al page fault

1. **Salvataggio del contesto di esecuzione** del processo (registri, stato, tabella delle pagine)
2. **Verifica del motivo del page fault** (mediante una tabella interna al kernel)
 - **riferimento illegale (violazione delle politiche di protezione)** → terminazione del processo
 - **riferimento legale**: la pagina è in memoria secondaria
3. **Copia della pagina** in un frame libero
4. **Aggiornamento della tabella delle pagine**
5. **Ripristino del processo**: esecuzione dell'istruzione interrotta dal page fault

kernel

Paginazione su richiesta: sovrallocazione

In seguito a un page fault:

- se è necessario caricare una pagina in memoria centrale, ***può darsi che non ci siano frame liberi***

sovrallocazione

Soluzione

→ **sostituzione** di una pagina P_{vitt} (**vittima**) allocata in memoria con la pagina P_{new} da caricare:

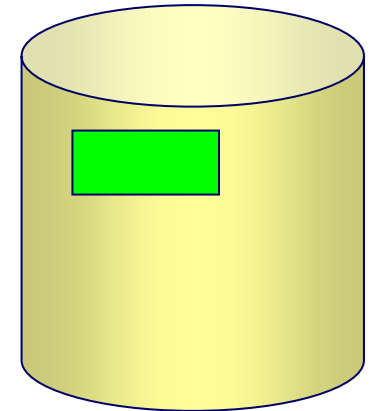
1. Individuazione della vittima P_{vitt}
2. Salvataggio di P_{vitt} su disco
3. Caricamento di P_{new} nel frame liberato
4. Aggiornamento tabelle
5. Ripresa del processo

Sostituzione di pagine

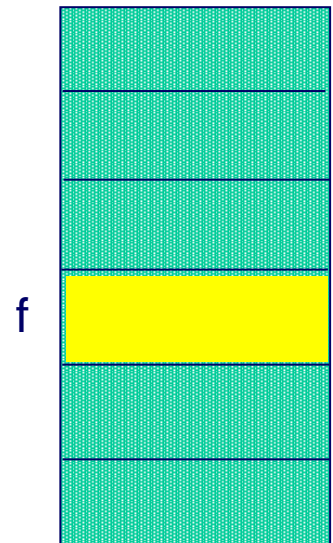
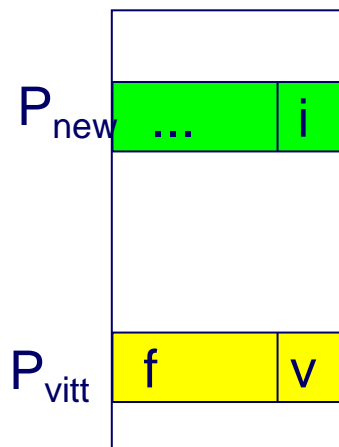
Memoria compl. allocata (sovrallocazione):

sostituire P_{vitt} con la pagina P_{new} da caricare

1. Individuazione della vittima P_{vitt}
2. Salvataggio di P_{vitt} su disco
3. Caricamento di P_{new} nel frame liberato
4. Aggiornamento tabelle
5. Ripresa del processo



Ind. logico

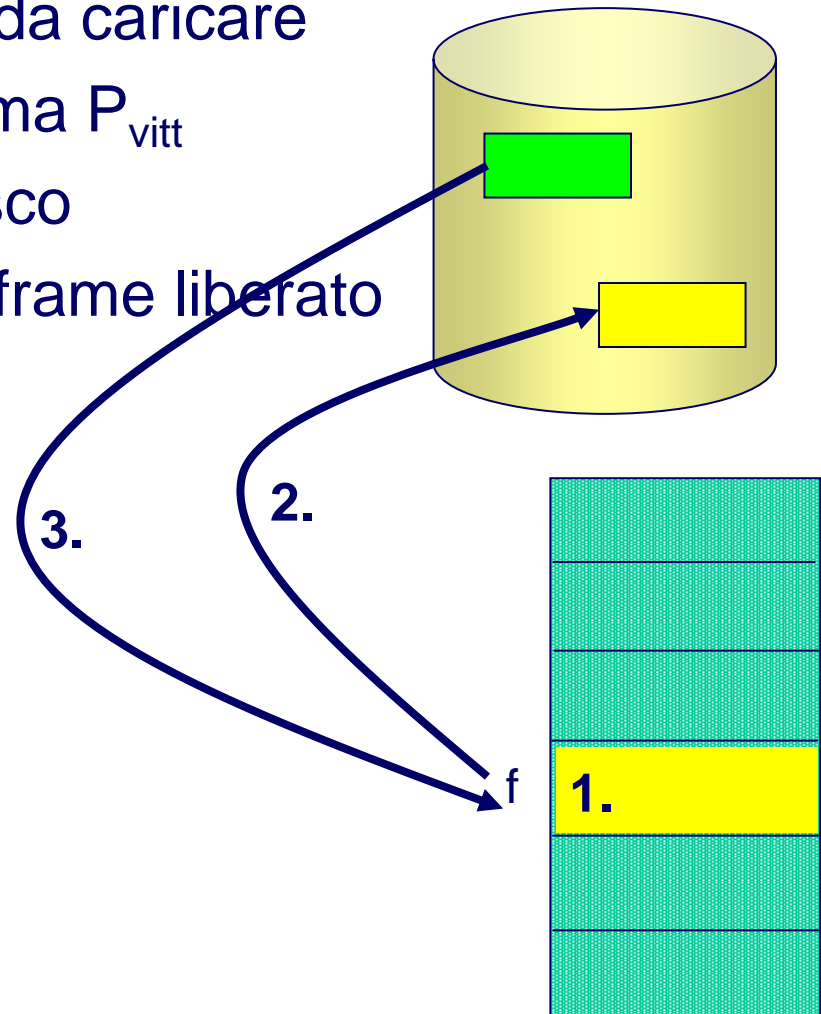
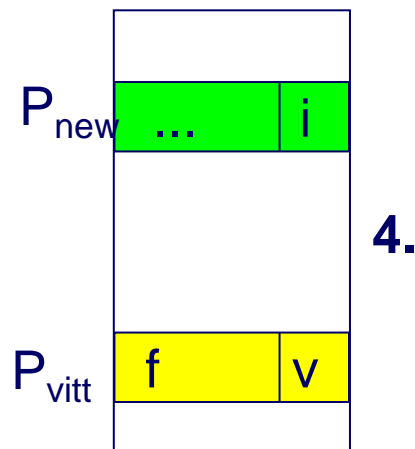


Sostituzione di pagine

Memoria compl. allocata (sovrallocazione):

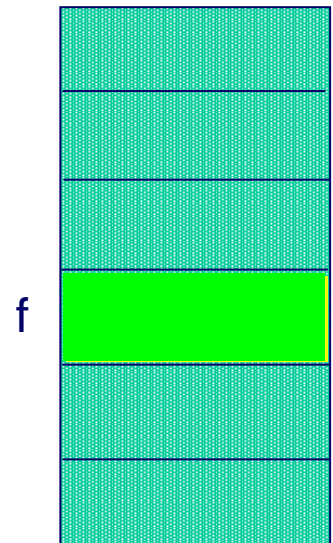
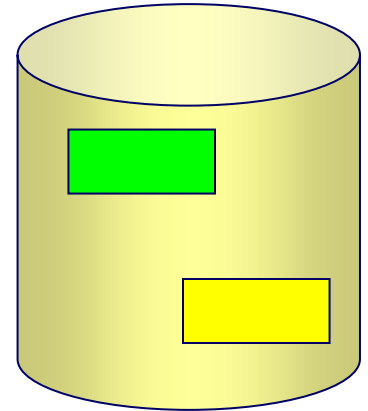
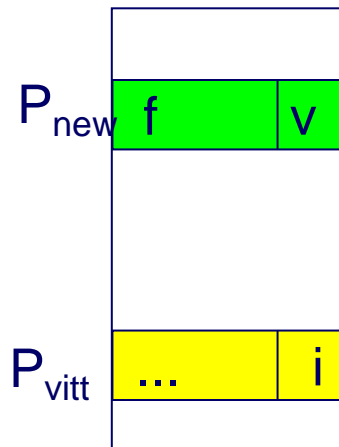
sostituire P_{vitt} con la pagina P_{new} da caricare

1. Individuazione della vittima P_{vitt}
2. Salvataggio di P_{vitt} su disco
3. Caricamento di P_{new} nel frame liberato
4. Aggiornamento tabelle
5. Ripresa del processo



Sostituzione di pagine

situazione finale



Sostituzione di pagine

In generale, la sostituzione di una pagina **può richiedere 2 trasferimenti da/verso il disco**:

- per scaricare la vittima
- per caricare la pagina nuova

Però è possibile che la **vittima non sia stata modificata** rispetto alla **copia residente sul disco**; ad esempio:

- pagine di codice (read-only)
- pagine contenenti dati che non sono stati modificati durante la permanenza in memoria

➤ In questo caso la **copia della vittima sul disco può essere evitata**:

→ introduzione del **bit di modifica (dirty bit)**

Dirty bit

Per rendere **più efficiente il trattamento del page fault** in caso di **sovrallocazione**

- ❑ si introduce in ogni elemento della tabella delle pagine un **bit di modifica (dirty bit)**
 - se settato, la pagina ha subito **almeno un aggiornamento** da quando è caricata in memoria
 - se a 0, la pagina **non è stata modificata**
- ❑ algoritmo di sostituzione esamina il bit di modifica della vittima:
 - esegue **swap-out della vittima solo se il dirty bit è settato**

Algoritmi di sostituzione della pagina vittima

La finalità di ogni algoritmo di sostituzione è **sostituire quelle pagine** la cui **probabilità** di essere accedute a **breve termine è bassa**

Algoritmi

- **LFU (Least Frequently Used)**: sostituita la pagina che è stata usata **meno frequentemente** (in un intervallo di tempo prefissato)
 - è necessario associare **un contatore degli accessi ad ogni pagina**
 - la vittima è quella con minimo valore del contatore

Algoritmi di sostituzione

- **FIFO**: sostituita la pagina che è **da più tempo caricata in memoria** (indipendentemente dal suo uso)
 - è necessario memorizzare la **cronologia dei caricamenti in memoria**
- **LRU (Least Recently Used)**: di solito preferibile per principio di località; viene sostituita la pagina che è stata usata **meno recentemente**
 - è necessario registrare la **sequenza degli accessi** alle pagine in memoria
 - **overhead**, dovuto **all'aggiornamento della sequenza** degli accessi per ogni accesso in memoria

Algoritmi di sostituzione

Implementazione LRU: necessario registrare la **sequenza temporale di accessi** alle pagine

Soluzioni

- **Time stamping:** l'elemento della tabella delle pagine contiene un campo che rappresenta **l'istante dell'ultimo accesso alla pagina**
 - **Costo della ricerca** della pagina vittima
- **Stack: struttura dati tipo stack** in cui ogni elemento rappresenta una pagina; l'accesso a una pagina provoca lo spostamento dell'elemento corrispondente al top dello stack → **bottom contiene la pagina LRU**
 - gestione può essere costosa, ma **non c'è overhead di ricerca**

Algoritmi di sostituzione: LRU approssimato

Spesso si utilizzano **versioni semplificate** di LRU introducendo, al posto della sequenza degli accessi, **un bit di uso** associato alla pagina:

- ❑ al momento del caricamento è **inizializzato a 0**
- ❑ quando la pagina viene **acceduta**, viene **settrato**
- ❑ **periodicamente**, i bit di uso vengono **resettati**

→ viene sostituita una **pagina con bit di uso==0**; il criterio di scelta, ad esempio, potrebbe inoltre considerare il **dirty bit**:

- **tra tutte le pagine non usate di recente** (bit di uso==0), ne viene scelta una **non aggiornata** (dirty bit==0)

Località dei programmi

Si è osservato che un processo, in una certa fase di esecuzione:

- ❑ usa solo un **sottoinsieme relativamente piccolo delle sue pagine logiche**
- ❑ sottoinsieme delle **pagine effettivamente utilizzate varia lentamente** nel tempo

- **Località spaziale**

- ❑ alta probabilità di accedere a **locazioni vicine (nello spazio logico/virtuale) a locazioni appena accedute** (ad esempio, elementi di un vettore, codice sequenziale, ...)

- **Località temporale**

- ❑ alta probabilità di accesso a **locazioni accedute di recente** (ad esempio cicli) → vedi algoritmo LRU

Working set

In alternativa alla paginazione su domanda, **tecniche di gestione della memoria che si basano su pre-paginazione:**

- si **prevede** il set di pagine di cui il processo da caricare ha bisogno per la prossima fase di esecuzione

working set

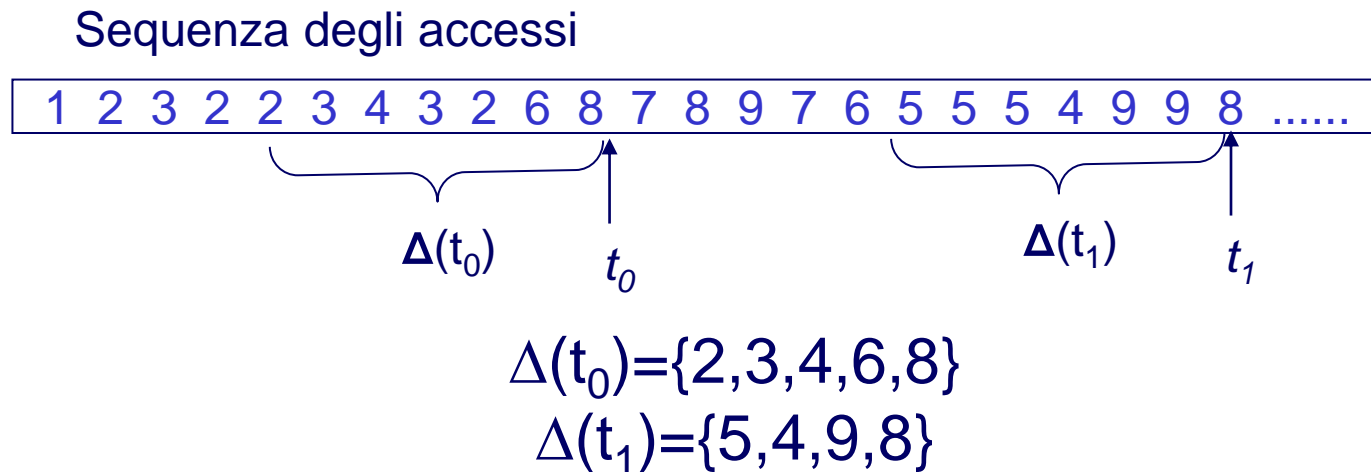
working set può essere individuato **in base a criteri di località temporale**

Working set

Dato un intero Δ , il working set di un processo P (nell'istante t) è l'insieme di pagine $\Delta(t)$ indirizzate da P nei più recenti Δ riferimenti

- Δ definisce la “finestra” del working set

Ad esempio, per $\Delta = 7$



Prepaginazione con working set

- Caricamento di un processo consiste nel caricamento di un **working set iniziale**
- SO mantiene **working set** di ogni processo aggiornandolo dinamicamente, in base al principio di **località temporale**:
 - ❑ all'istante t vengono **mantenute le pagine usate dal processo nell'ultima finestra $\Delta(t)$**
 - ❑ le altre pagine (esterne a $\Delta(t)$) **possono essere sostituite**

Vantaggio

- ❑ **riduzione del numero di page fault**

Working set

Il parametro Δ caratterizza il working set, esprimendo
l'estensione della finestra dei riferimenti

- Δ **piccolo**: working set **insufficiente a garantire località (alto numero di page fault)**
- Δ **grande**: allocazione di pagine non necessarie

Ad ogni istante, data la dimensione corrente del working set **WSS_i** di ogni processo P_i , si può individuare

$$D = \sum_i WSS_i \text{ richiesta totale di frame}$$

Se m è il numero totale di frame

- $D < m$: può esserci spazio per allocazione nuovi processi
- $D > m$: **swapping di uno (o più) processi**

Un esempio: gestione della memoria in UNIX (prime versioni)

In UNIX spazio logico **segmentato**:

- ❑ nelle **prime versioni** (prima di BSDv3), **allocazione contigua dei segmenti**
 - **segmentazione pura**
 - **non c'era memoria virtuale**
- ❑ in caso di difficoltà di allocazione dei processi **swapping dell'intero spazio degli indirizzi**
- ❑ **condivisione di codice**: possibilità di evitare trasferimenti di codice da memoria secondaria a memoria centrale → minor overhead di swapping

Un esempio: gestione della memoria in UNIX (prime versioni)

Tecnica di **allocazione contigua** dei segmenti:

- **first fit** sia per l'allocazione in memoria centrale, che in memoria secondaria (swap-out)

Problemi

- frammentazione esterna
- stretta influenza dim spazio fisico sulla gestione dei processi in multiprogrammazione
- crescita dinamica dello spazio → possibilità di riallocazione di processi già caricati in memoria

UNIX: swapping

In assenza di memoria virtuale, ***swapper*** ricopre un ruolo chiave per la ***gestione delle contese di memoria*** da parte dei diversi processi:

- ❑ periodicamente (ad esempio nelle prime versioni ogni 4s) lo ***swapper*** viene attivato per provvedere (***eventualmente***) a ***swap-in e swap-out*** di processi
 - **swap-out:**
 - processi inattivi (sleeping)
 - processi “ingombranti”
 - processi da più tempo in memoria
 - **swap-in:**
 - processi piccoli
 - processi da più tempo ***swapped***

La gestione della memoria in UNIX (versioni attuali)

Da BSDv3 in poi:

- ❑ **segmentazione paginata**
- ❑ **memoria virtuale** tramite **paginazione su richiesta**

L'allocazione di ogni segmento **non è contigua**:

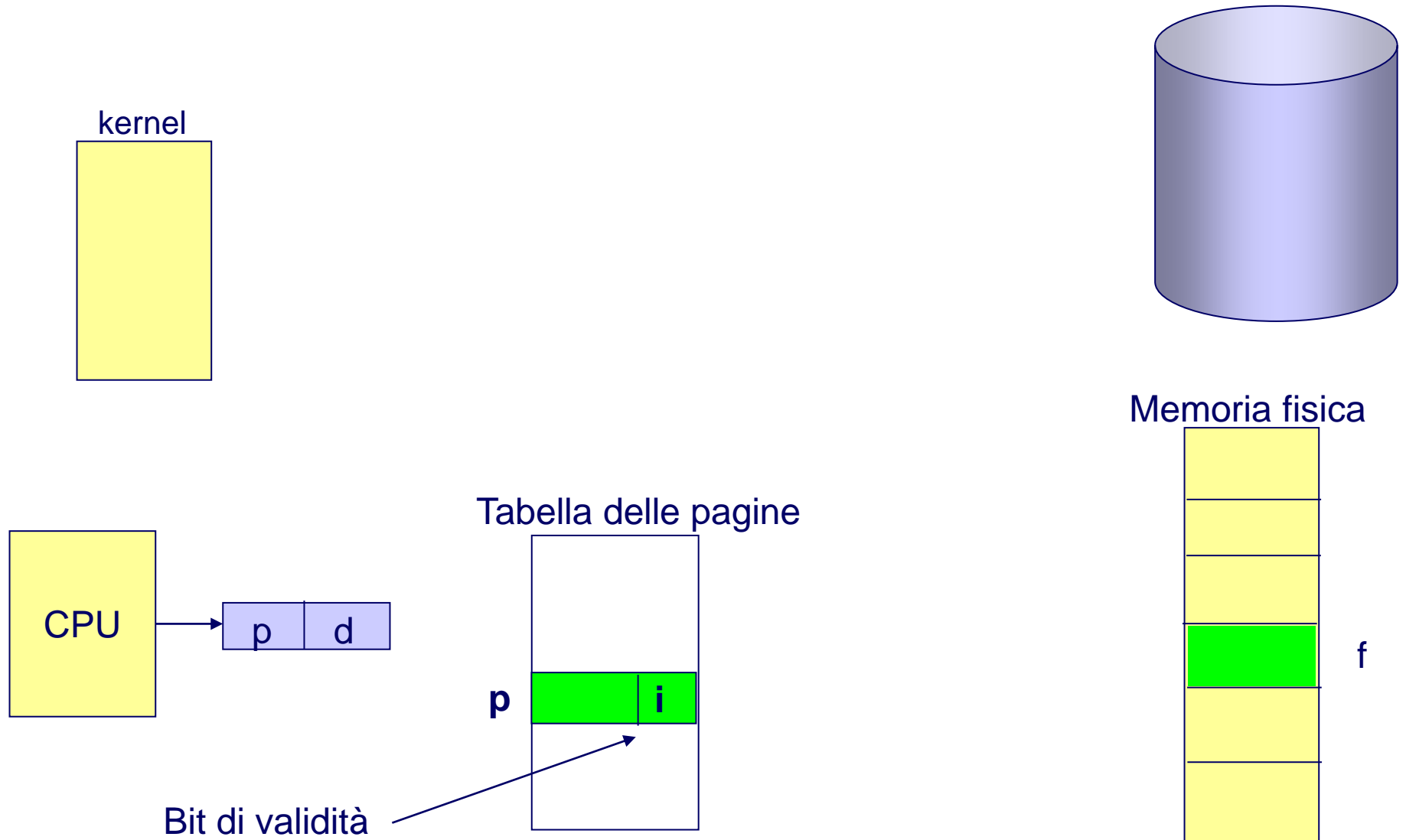
- ❑ si risolve il problema della frammentazione esterna
- ❑ frammentazione interna trascurabile (pagine di dimensioni piccole)

La gestione della memoria in UNIX (versioni attuali)

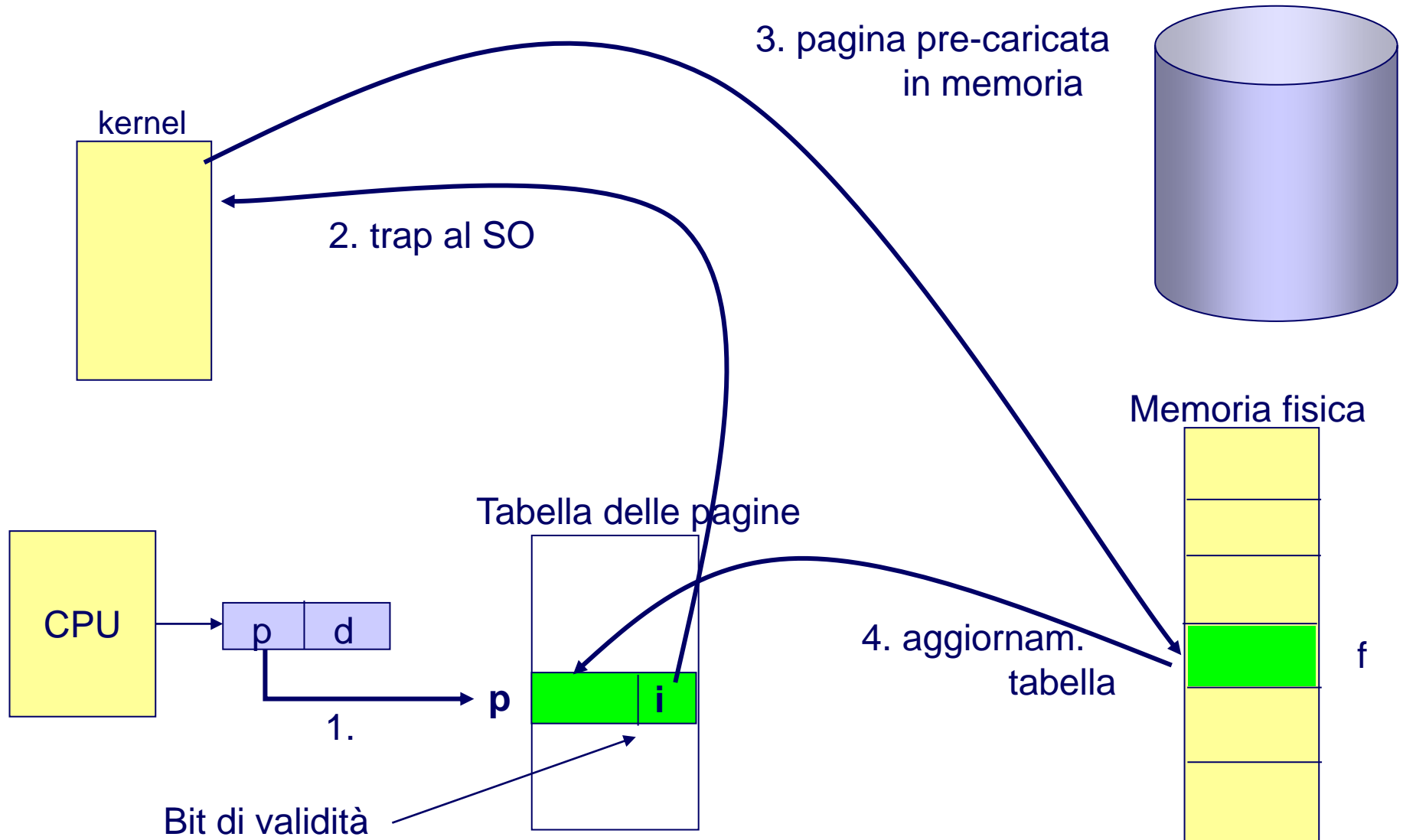
Paginazione su richiesta

- ❑ **pre-paginazione:** uso dei frame liberi per pre-caricare pagine non strettamente necessarie
Quando avviene un page fault, se la pagina è già in un frame libero, basta soltanto modificare:
 - tabella delle pagine
 - lista dei frame liberi
- ❑ **core map:** struttura dati interna al kernel che descrive lo **stato di allocazione dei frame** e che viene consultata in caso di page fault

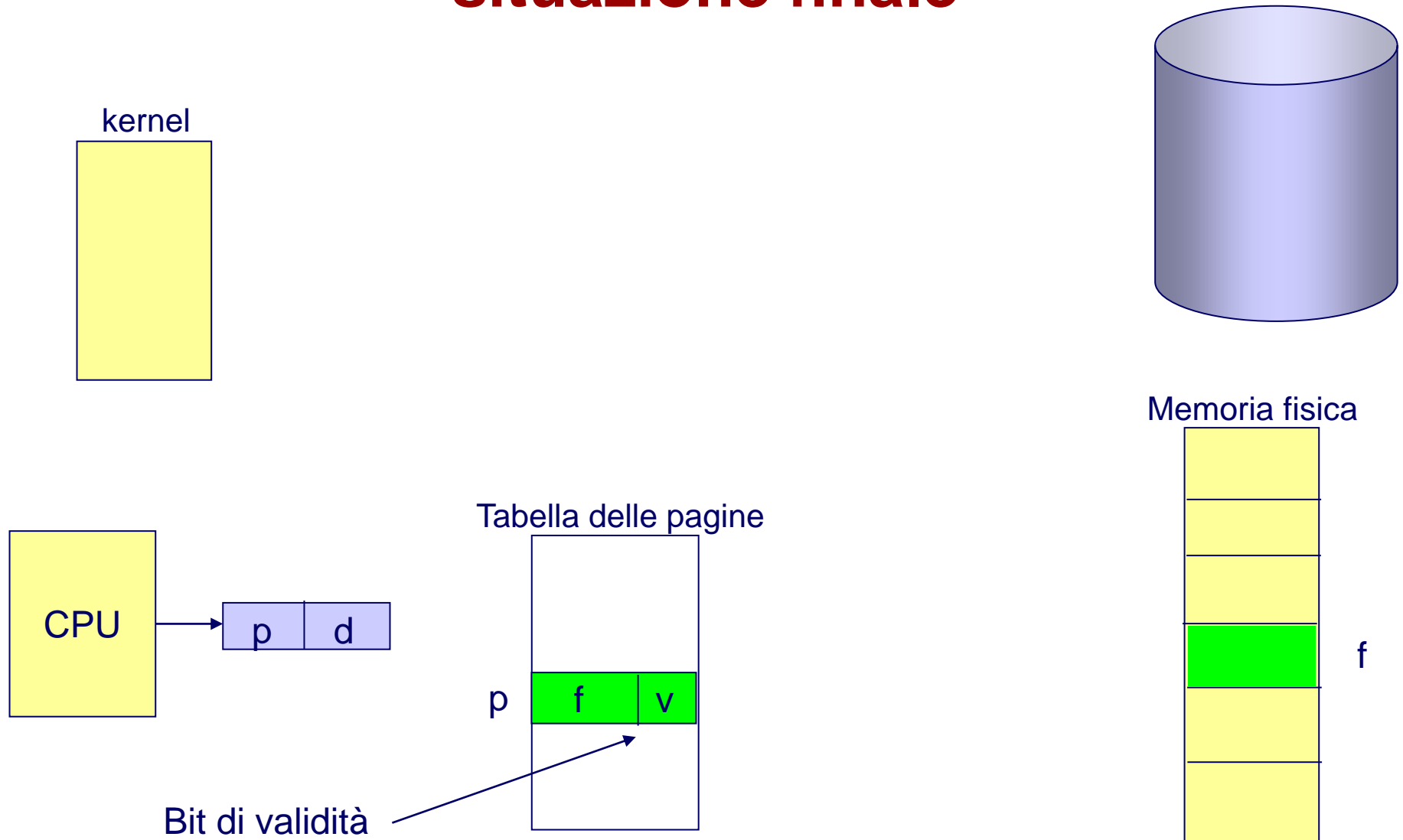
UNIX: page-fault in caso di pre-paginazione



UNIX: page-fault in caso di pre-paginazione



UNIX: page-fault in caso di pre-paginazione situazione finale



UNIX: algoritmo di sostituzione

LRU modificato o algoritmo di seconda chance (BSDv4.3 Tahoe)

ad ogni pagina viene associato un **bit di uso**:

- ❑ al momento del caricamento è **inizializzato a 0**
- ❑ quando la pagina è acceduta, viene **settato a 1**
- ❑ nella fase di **ricerca di una vittima**, vengono esaminati i **bit di uso di tutte le pagine in memoria**
 - se una pagina ha il bit di uso a 1, viene posto a 0
 - se una pagina ha il bit di uso a 0, viene selezionata come vittima

UNIX: algoritmo di sostituzione

Sostituzione della vittima:

- ❑ pagina viene **resa invalida**
- ❑ frame selezionato viene inserito **nella lista dei frame liberi**
 - se c'è **dirty bit**:
 - **solo se dirty bit=1** → pagina va **copiata** in memoria secondaria
 - se non c'è **dirty bit** → pagina va **sempre copiata** in memoria secondaria

L'algoritmo di sostituzione viene eseguito dal pager **pagedaemon (pid=2)**

UNIX: sostituzione delle pagine

Scaricamento di pagine (sostituzione) attivato quando numero totale di frame liberi è ritenuto insufficiente (minore del valore **lotsfree**)

Parametri

- ❑ **lotsfree**: numero minimo di frame liberi **per evitare sostituzione** di pagine
- ❑ **minfree**: numero minimo di frame liberi necessari **per evitare swapping** dei processi
- ❑ **desfree**: numero **desiderato** di frame liberi

lotsfree > desfree > minfree

UNIX: scheduling, paginazione e swapping

Scheduler attiva l'algoritmo di sostituzione se

- ❑ il numero di frame liberi **< lotsfree**

Se sistema di **paginazione** è **sovraccarico**, ovvero:

- numero di frame liberi **< minfree**
- numero medio di frame liberi nell'unità di tempo **< desfree**

→ **scheduler** attiva **swapper** (al massimo ogni secondo)

SO evita che **pagedaemon** usi più del 10% del tempo totale di CPU: attivazione (al massimo) ogni 250ms

Gestione della memoria in Linux

- Allocazione basata su **segmentazione paginata**
- Paginazione **a più (2 o 3) livelli**
- Allocazione contigua dei moduli di codice caricati dinamicamente (non abbiamo visto i meccanismi di caricamento runtime di codice in questo corso...)
- **Memoria virtuale, senza working set**

Linux: organizzazione della memoria fisica

Alcune aree riservate a scopi specifici:

- ❑ **Area codice kernel:** pagine di quest'area sono **locked** (non subiscono paginazione)
- ❑ **Kernel cache:** heap del kernel (**locked**)
- ❑ **Area moduli gestiti dinamicamente:** allocazione mediante algoritmo **buddy list** (allocazione contigua dei singoli moduli)
- ❑ **Buffer cache:** gestione I/O su dispositivi a blocchi
- ❑ **Inode cache:** copia degli **inode** utilizzati recentemente (vedi tabella file attivi)
- ❑ **Page cache:** pagine non più utilizzate in attesa di sostituzione
- ❑ ...

Il resto della memoria è utilizzato per i processi utente

Linux: spazio di indirizzamento

Ad ogni processo Linux possono essere allocati **4GB**, di memoria centrale (in caso di sistema a 32 bit):

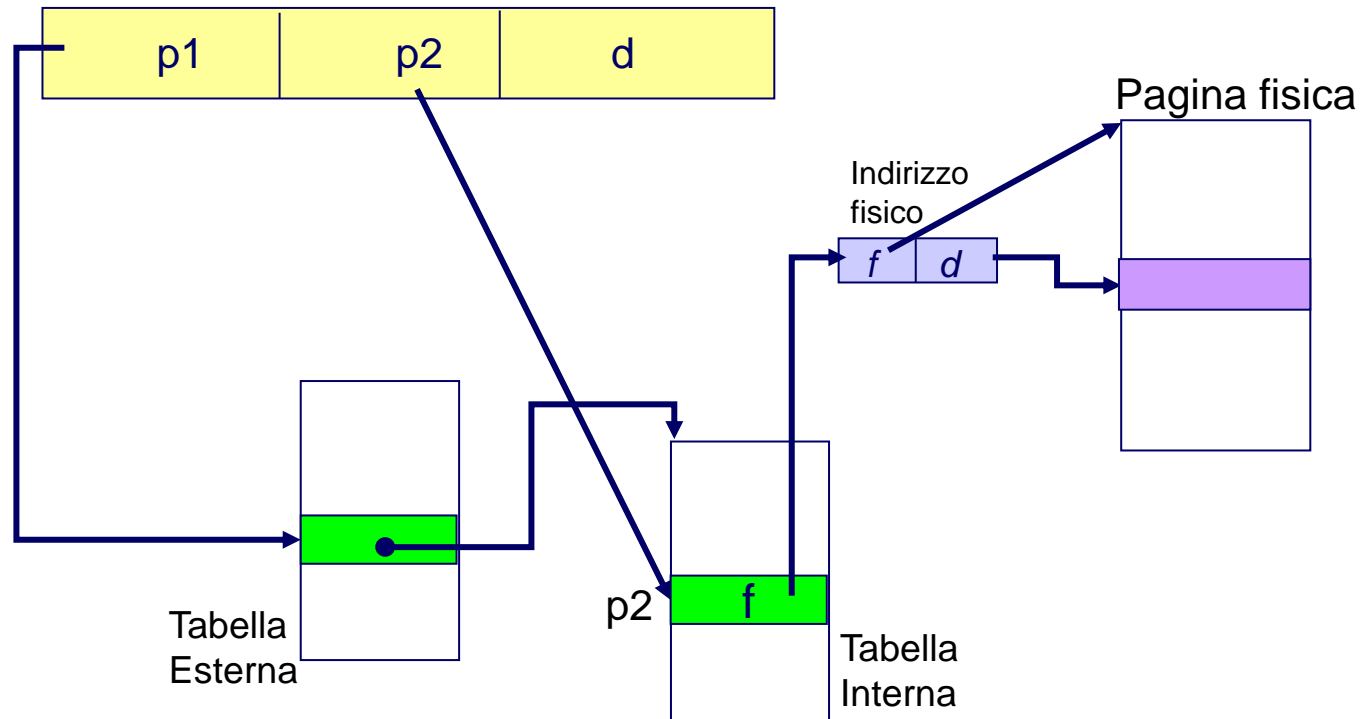
- ❑ 3GB al massimo possono essere utilizzati per lo **spazio di indirizzamento virtuale**
- ❑ 1GB **riservato al kernel**, accessibile quando il processo esegue in kernel mode
- ❑ con architetture a 64 bit, spazio di indirizzamento ≥ 1 TB, paginazione a 4 livelli

Spazio di indirizzamento di ogni processo può essere suddiviso in un insieme di **regioni omogenee e contigue**

- ❑ ogni **regione** è costituita da una **sequenza di pagine accomunate dalle stesse caratteristiche di protezione e di paginazione**
- ❑ ogni pagina ha una **dimensione costante** (4KB su architettura Intel)

Linux: paginazione

- ❑ **paginazione a tre livelli**
- ❑ realizzato per processori Alpha, in alcune architetture i livelli si riducono a 2 (ad esempio Pentium)



Linux: page-fetching e sostituzione

- ❑ **NON** viene utilizzata la tecnica del **working set**
- ❑ viene mantenuto un **insieme di pagine libere** che possano essere utilizzate dai processi (**page cache**)
- ❑ analogamente a UNIX, una volta al secondo:
 - viene controllato che ci siano **sufficienti pagine libere**
 - altrimenti, viene **liberata una pagina occupata**