

Architettura di un SO

Funzionalità di un SO

- Virtualizzazione delle risorse hardware
 - file system
 - processi
 - periferiche astratte
 - ...

Sono tutti concetti che permette di ottenere una virtualizzazione e un'estensione delle risorse hardware della macchina fisica.

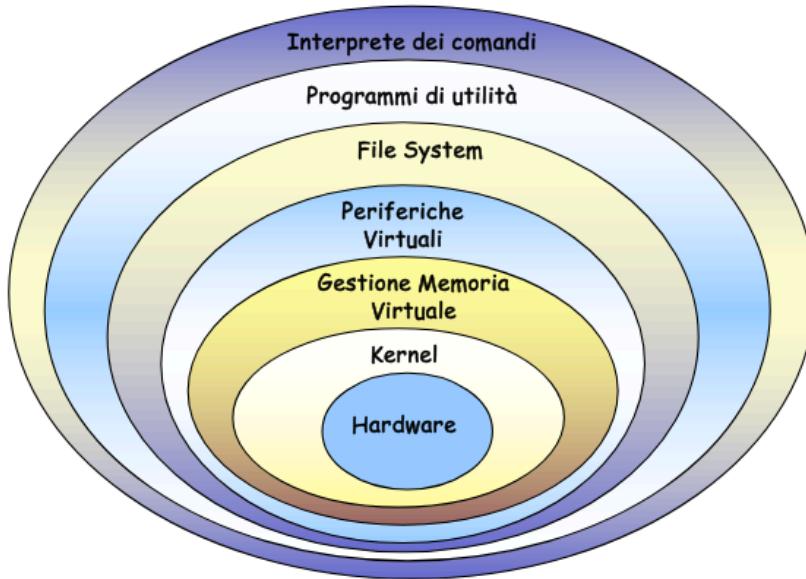
Oltre a creare un'astrazione per la macchina fisica, il SO fornisce anche un API di sistema che permette alle applicazioni utente di operare sull'hardware delegando le operazioni a quest'ultimo che interagisce direttamente con l'hardware.

Queste API garantiscono un accesso uniforme rispetto tutte le componenti hardware che possono essere anche molto eterogenee tra loro.

- Gestione e coordinamento

Il SO ha il compito di arbitrare le richieste dei programmi utente

- meccanismi di protezione → per evitare azioni illecite da parte di processi malfunzionanti
- meccanismi per la comunicazione inter-processo
- gestore delle risorse → scheduling



- Tutti questi strati mi **permettono di astrarre** le risorse hardware le quali sono gestite direttamente dal Kernel, che fa da ultimo intermediario tra il livello software e quello hardware.
- Quindi tutti gli strati che si contrappongono tra l'hardware e i programmi di utilità definiscono una **macchina virtuale**.
- Tutti gli strati intermedi che definiscono un'astrazione potrebbero essere inclusi all'interno del Kernel, questo infatti dipende dall'architettura del SO.

Ogni strato serve a nascondere la complessità dello strato sottostante e a fornire servizi più semplici e potenti allo strato superiore.

Kernel

Il Kernel è quella parte del SO che risiede in memoria principale.

Contiene funzionalità fondamentali del SO.

A livello **kernel**, la macchina virtuale realizzata dal SO (dal punto di vista delle singole applicazioni utente):

- possiede tante unità centrali quanti sono i processi attivi nel sistema (ovvero processori virtuali). Ogni processo ha l'illusione di avere per se un processore dedicato, indipendentemente dal numero reale di processori fisici;
- non possiede meccanismi di interruzione. I processi vedono un flusso di esecuzione continuo e non devono gestire le interruzioni hardware perché sono intercettate dal kernel che provvede alla gestione;

questo è possibile per il context switch che viene fatto all'atto della sospensione del processo per la gestione da parte del SO delle interrupt. Nel momento in cui è terminata la ISR per gestire l'interrupt il processo torna ad eseguire sulla CPU partendo dallo stato in cui è stato sospeso.

Quindi per il processo la gestione delle interruzioni è trasparente.

- possiede istruzioni di sincronizzazione e scambio di messaggi tra processi che operano su processori virtuali.

Tali meccanismi di sincronizzazione e scambio di messaggi avvengono tramite l'utilizzo di syscalls.

Gestione della memoria

A livello della **gestione della memoria**, la macchina virtuale realizzata dal SO (dal punto di vista dei processi):

- consente di far riferimento a **spazi di indirizzamento virtuali**.

Questi creano un'astrazione per i processi, illudendo questi di avere a disposizione l'intera memoria centrale per se.

- gestisce la protezione, ovvero il gestore della memoria verifica che non ci siano interferenze tra i vari processi, quindi che lo spazio di indirizzamento del singolo processo sia isolato e non accessibili da altri processi.

Quindi due processi potrebbero utilizzare due indirizzi virtuali uguali ma questi saranno mappati in indirizzi fisici distinti nel momento in cui i processi andranno ad accedere a questi.

- consente di rendere trasparente la posizione effettiva dei dati/istruzioni per i processi. Infatti i dati che un processo intende leggere e modificare possono risiedere temporaneamente in memoria di massa in casi particolari.

Questi casi particolari corrispondono ad esempio alla saturazione della memoria centrale. In tali casi alcune pagine vengono swappate in una porzione della memoria secondaria utilizzata per mantenere pagine della memoria centrale che non vengono subito utilizzate.

Gestione delle periferiche

A livello **gestione delle periferiche**, la macchina virtuale realizzata dal SO (dal punto di vista dei processi):

- dispone di periferiche dedicate ai singoli processi.

Illude quindi i processi di avere a disposizione tutte le risorse disponibili;

- maschera le caratteristiche fisiche delle periferiche.

I processi non conoscono quali risorse compongono effettivamente l'hardware poiché il SO crea un'astrazione di queste rendendo il loro accesso uniforme.

Quindi nel caso di un lettura o scrittura il processo che la richiede non sa effettivamente le caratteristiche fisiche dell'hardware su cui sta scrivendo o leggendo.

A conoscere ciò sono i driver specifici che implementano effettivamente le operazioni di lettura e scrittura specifiche per il determinato hardware in questione.

- gestisce parzialmente i malfunzionamenti delle periferiche.

File system

Permette di trasformare un enorme contenitore di bit disordinati, memoria secondaria, in un archivio ordinato.

Quindi gli strati superiori vedranno un'organizzazione ordinata della memoria.

Al livello **file system**, la macchina virtuale realizzata dal SO (dal punto di vista del processo):

- gestisce blocchi di informazioni su memoria di massa strutturati logicamente;

Quindi delle sequenze di byte memorizzate in memoria di massa identifica le sequenze che rappresentano cartelle e file;

- ne controlla gli accessi;

Per ogni accesso verifica se il processo richiedente ha i permessi necessari per leggere o scrivere su tale blocco di memoria.

Questo perché ogni file ha associato dei metadati. Tra questi metadati ci sono le informazioni riguardanti i permessi di accesso da parte dei processi;

- ne gestisce l'organizzazione.

Gestisce la geometria della memoria di massa e implementa i metodi che devono essere utilizzare per accedere correttamente ai dati.

Architettura dei sistemi operativi

Il SO è un programma di notevole **complessità** e **dimensione**. Infatti tutti i **layer** che abbiamo visto prima, necessari per garantire la virtualizzazione dell'hardware e una sua gestione efficiente, fanno **parte del SO**.

È quindi fondamentale applicare, durante il suo progetto e la sua realizzazione, le più sofisticate **tecniche proprie dell'ingegneria del software**, al fine di garantire un risultato che goda di tutte le **proprietà che garantiscano la qualità di un sistema software**: correttezza, modularità, facilità di manutenzione, efficienza di esecuzione, etc.

Per questo sono stati proposti, nel tempo, **vari modelli strutturali** cui fare riferimento **per organizzare i componenti** durante le fasi di progetto, realizzazione e test del sistema.

sistemi operativi monolitici

I primi sistemi operativi erano costituiti da un solo programma (**sistemi monolitici**) senza una particolare suddivisione dello stesso in moduli.

Il sistema operativo era **costituito da un insieme di procedure di servizio** a ciascuna delle quali corrispondeva una **chiamata al sistema**.

Normalmente le procedure erano scritte in **linguaggio assembler** per consentire un più efficiente accesso alle risorse hardware della macchina.

Questo tipo di approccio al progetto del sistema poteva essere **adeguato soltanto nel caso di sistemi molto semplici**, come nel caso dei primi sistemi operativi e, successivamente, come nel caso di semplici **sistemi non multiprogrammati**, per esempio il sistema operativo MS-DOS.

Col crescere della complessità tipica dei moderni sistemi multiprogrammati questo approccio ha rapidamente mostrato tutti i propri limiti.

sistemi operativi modulari

Un valido approccio utilizzato per affrontare la complessità di un sistema è quello di fare riferimento a **tecniche di modularizzazione** in modo tale da suddividere il sistema in componenti (moduli), **ciascuno destinato a fornire una delle funzionalità del sistema**.

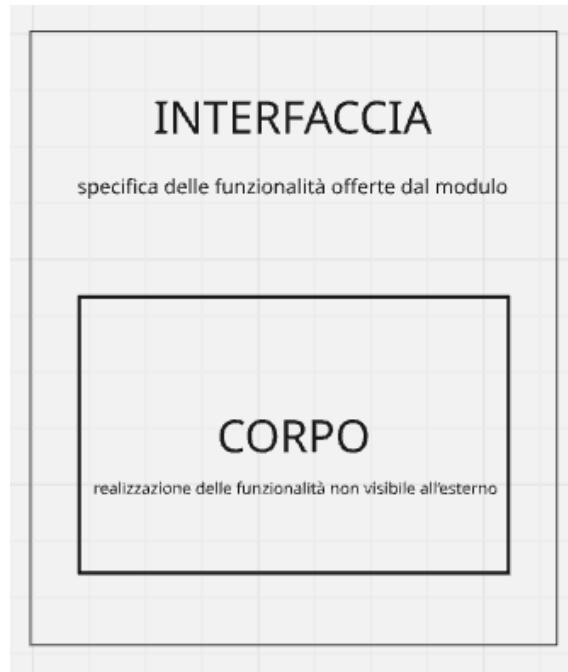
Realizzato usualmente in un **linguaggio di alto livello seguendo i criteri tipici della programmazione strutturata (sistemi modulari)**.

In base ai criteri della programmazione strutturata, ogni modulo è caratterizzato da una ben precisa interfaccia, che specifica la funzionalità offerta dal modulo, e un corpo contenente l'implementazione del modulo, non visibile all'esterno.

In questo modo ogni modifica che veniva apportata ai moduli non influenzava il funzionamento degli altri, a meno che la modifica non era fatta sull'interfaccia offerta dal modulo.

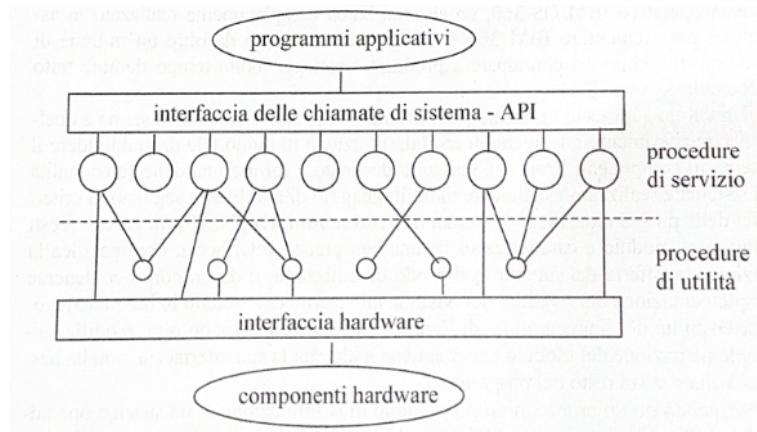
Il primo esempio di strutturazione di un sistema operativo fu quello di che si ottenne modificando la struttura dei primi sistemi monolitici in modo tale da identificare i vari moduli componenti e

dettagliando con cura le rispettive interfacce.



I vari moduli furono distinti in due categorie:

- le procedure di servizio offerte dal sistema ai programmi applicativi tramite chiamate di sistema;
- le procedure di utilità, utilizzate dalle prime ma non direttamente visibili ai processi.



Ovviamente ogni chiamata di sistema provocava un cambio di contesto da modalità utente a modalità kernel.

Questo tipo di struttura è anche quella adottata nel sistema Unix.

In questo caso fanno parte del sistema:

- sia le tipiche **componenti di un sistema operativo**, invocate tramite le chiamate di sistema, eseguite in **stato privilegiato** e identificate globalmente con il termine *kernel*

- sia l'**insieme dei programmi di utilità del sistema** costituiti dalla shell, dai compilatori, dai caricatori, dai linker e dalle librerie di sistema, eseguiti in stato non privilegiato come i normali programmi utente.

Nonostante gli indubbi vantaggi indotti dall'uso di tecniche di modularità, la complessità dei sistemi operativi è andata progressivamente crescendo, richiedendo quindi ulteriori paradigmi di progetto in grado di affrontare in modo più idoneo la crescente complessità.

sistemi operativi a livelli gerarchici di astrazione

Consistono in SO modulari, organizzati in una struttura gerarchica.

Uno degli aspetti fondamentali della tecnica dell'organizzazione gerarchica consiste nel **ridurre il numero di possibili interconnessioni** fra i moduli di un sistema, semplificando quindi sia la fase di progetto sia quella di verifica.

Ovvero si punta ad ottenere un'organizzazione in moduli che punti a diminuire le dipendenze tra questi e quindi aumentare il livello di coesione.

Tale tecnica che permette questo tipo di organizzazione di un sistema operativo è possibile applicarla seguendo due possibili paradigmi complementari.

- top-down, livelli di raffinamento successivi.

Consiste nello scomporre il sistema software nelle sue funzionalità principali, astraendo i dettagli implementativi. Successivamente ciascuna di queste funzionalità veniva a sua volta scomposta in altre funzionalità che la comprendessero astraendo sempre i dettagli implementativi; e così via.

Fino a che non si arrivi ad un livello di funzionalità elementare da cui può iniziare l'implementazione.

- bottom-up.

Consiste nel partire dal basso, quindi a stretto contatto con l'hardware.

Implementare delle funzionalità aggiuntive che non sono disponibili in hardware e da queste funzionalità si parte ad implementarne che le utilizzino.

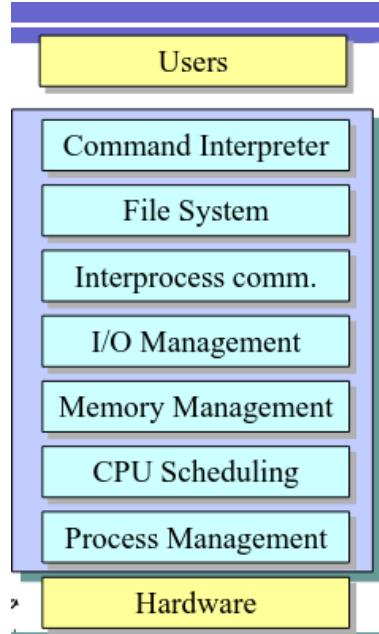
Quindi si arriva ad un livello che non è più necessario comunicare direttamente con l'hardware grazie alle funzionalità elementari che sono state implementate in principio.

Questo quindi permette di implementare funzionalità che astraggono sempre di più il livello hardware.

Un utilizzo combinato di questi due paradigmi permette di realizzare **sistemi operativi strutturati in moduli organizzati gerarchicamente in diversi livelli di astrazione**, in modo tale che i **moduli**

realizzati a un certo livello utilizzano esclusivamente le **funzionalità offerte dai moduli di livello più basso** e forniscano le loro funzionalità ai moduli di livello più alto.

Possiamo intendere questa struttura come se ogni livello definisse una nuova macchina astratta, le cui funzionalità sfruttano quelle offerte dalla macchina astratta di livello inferiore.



sistemi operativi a microkernel

La necessità di proteggere dall'acceso diretto l'hardware da parte dei programmi utente che avrebbero potuto causare dei danni irreparabili si è introdotto il concetto dei livelli di esecuzione:

- kernel mode
- user mode

Questo però ha portato il sistema ad essere più difficilmente modificabile ed estensibile.

Modifiche o estensioni potrebbero risultare necessarie in vari casi:

- si deve aggiungere un driver in seguito alla connessione di un nuovo dispositivo
- si intende modificare alcune scelte relative alla gestione di una o più risorse al fine di rendere il sistema più adatto ai requisiti imposti dai programmi applicativi specifici che andranno in esecuzione su questo.

Per dare una soluzione a questo tipo di problemi è stata proposta una soluzione nota col nome di **struttura a microkernel**.

Il microkernel implementa solo i **meccanismi essenziali** a discrezione del progettista.

Tutte le **strategie per la gestione** delle risorse sono implementate all'esterno del kernel, da processi di sistema che girano in user mode.

Quindi sono facilmente modificabili ed estensibili.

- Driver
- Memory management
- CPU scheduling

Ovviamente è inevitabile che tutti questi programmi non abbiano la necessità di operare direttamente sulle risorse hardware, quindi si utilizzano comunque delle componenti del SO che girano in kernel mode per espletare alcune funzioni. → context switch

Questa struttura permette di aumentare l'affidabilità e la sicurezza dei sistemi perché gli eventuali guasti non si propagano nel kernel, ma rimangono in user mode.

I gestori delle risorse sono particolari processi di sistema, spesso indicati come **server** (file server, terminal server, printer server, ...).

Quando un processo applicativo deve usare una risorsa, deve interagire con il processo server gestori di quella risorsa attraverso i meccanismi di comunicazione forniti dal microkernel.

La comunicazione interprocesso è la caratteristica svantaggiosa di questa struttura perché provoca una perdita delle performance.

Il motivo principale è dovuto al maggior numero di context switch necessari per espletare un singolo servizio ad un processo client.

Infatti utilizzando il comando `time` in UNIX, che misura il tempo che un processo trascorre in user-mode, in kernel-mode e l'attesa nel caso di operazioni di IO, si sono notate le perdite di performance.

Esempio di architetture modulari: linux

Il kernel Linux è monolitico perché è un unico binario che viene caricato in memoria centrale.

Nel tempo è stato introdotto il meccanismo dei moduli caricabili che ha permesso quindi di estendere le funzionalità del kernel senza la necessità che questo venga ricompilato ogni volta.

Quindi possiamo dire che il kernel linux ha un'architettura monolitica e modulare perché implementa il meccanismo dei moduli caricabili che possono essere collegati e scollegati dal kernel a runtime.

I comandi per caricare o scaricare un modulo dal kernel sono:

- `insmod`
- `rmmod`

Il rischio che si corre con questa struttura è che i moduli che possono essere collegati al kernel girano nello stesso spazio di indirizzamento del kernel, quindi se questi hanno un bug grave possono mandare in crash l'intero sistema (kernel panic).

In generale, il progetto del kernel linux ha cercato di prendere i vantaggi dei due mondi: monolitico e flessibilità modulare.

Windows architecture

Struttura modulare per renderlo più flessibile. Questo sistema è basato su un'architettura a microkernel ibrida.

- Non è puramente un'architettura a microkernel, anche se la maggior parte delle funzionalità sono eseguite all'esterno del microkernel.
- Tutti i moduli esterni sono caricati dinamicamente e quindi posso essere rimossi, aggiornati, o sostituiti senza riscrivere o ricompilare tutto il sistema.

Scheduler

Con il termine *short term scheduler* si intende quella funzione del nucleo che ha il compito di gestire l'allocazione della CPU ai processi che si trovano nella coda dei processi pronti.

In generale, lo *scheduler* è quella parte del SO preposta all'assegnazione di risorse a favore dei processi richiedenti.

La selezione tra i processi richiedenti (coda dei relativi PCB) nella coda può avvenire mediante differenti criteri, a seconda **dell'algoritmo di scheduling** che implementa uno schedulatore.

Questo è un compito fondamentale per tutti i sistemi multiprogrammati in cui c'è la necessità di un efficiente gestione delle risorse condivise disponibili per tutti i processi attivi e che eseguono in modo concorrente.

Inoltre gli scheduler sono i componenti più caratterizzanti del sistema operativo, infatti da questa si possono identificare le caratteristiche e gli obiettivi per cui è stato progettato.

Dal punto di vista dell'esecuzione dei processi abbiamo diversi scheduler ognuno che si occupa di svolgere un particolare compito:

- *long term scheduling* o di job
- *medium term scheduling* o di swap
- *short term scheduling* o della CPU

Scheduling a lungo termine

Con **scheduling a lungo termine** si intende quella funzione del sistema operativo che, in un sistema di tipo batch, sceglie tra tutti i programmi caricati in memoria di massa per essere eseguiti, quelli da trasferire in memoria centrale e da inserire nella coda dei processi pronti.

Quindi controlla il **grado di multiprogrammazione** del sistema.

Le sue scelte, in genere, vengono fatte in modo da equilibrare la presenza in memoria centrale di processi caratterizzati da prevalenza di operazioni di elaborazione (CPU bound) e di processi caratterizzati da prevalenza di operazioni di ingresso e uscita (I/O bound).

In modo da distribuire il carico in modo ottimale su tutte le risorse disponibili.

Troppi processi I/O bound → la coda dei processi pronti è sempre quasi vuota

Troppi processi CPU bound → i dispositivi I/O sarebbero poco utilizzati

I possibili criteri su cui si può basare la scelta dello scheduler a lungo termine sono:

- FIFO
- Priorità
- Tempo di esecuzione stimato
- Requisiti di I/O → processi I/O bound
- Tempo presunto di CPU → processi CPU-bound

Scheduling a medio termine

Lo **scheduling a medio termine** rappresenta invece quella funzione del sistema operativo che ha il compito di trasferire temporaneamente processi dalla memoria centrale alla memoria di massa (*swap-out*) e viceversa (*swap-in*).

Questa funzione è necessaria, in caso di RAM prossima alla saturazione, di liberare parte della memoria centrale necessaria ad altri processi già presenti o per rendere possibile il caricamento di nuovi processi.

Il suo obiettivo è quello di **migliorare l'efficienza** nell'utilizzo della risorsa **memoria**.

Se questa gestione dei trasferimenti di processi verso e da la memoria di massa non è fatta in modo efficiente si verificherebbero molte operazioni di I/O per spostare i processi tra le memorie.

- ad esempio nel caso in cui due processi si trovino a collaborare e quindi l'esecuzione dell'uno dipende dall'esecuzione dell'altro;
- supponiamo di avere implementato due processi che operano su una memoria condivisa secondo il paradigma produttori consumatori
- se il produttore venisse sempre swappato in memoria di massa, allora questo poi dovrà esser reintrodotto in memoria centrale per permettere al consumatore di consumare la risorsa.

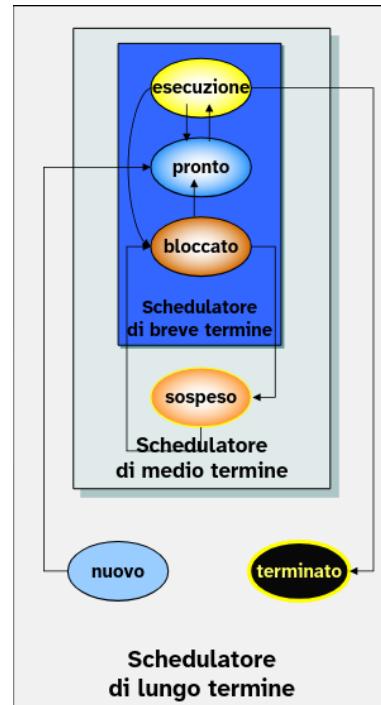
Entrambe le funzioni di long term scheduling e medium term scheduling vengono eseguite dal sistema operativo con una frequenza nettamente inferiore a quella della short term scheduling.

Quindi possono risultare meno costose in termini di tempo di esecuzione e quindi non inficiare molto sulle performance del sistema.

Invece la funzione di short term scheduling è eseguita molto frequentemente, ad ogni context switch, in modo da rendere l'esecuzione di più processi fluida, dando l'impressione che siano eseguiti in parallelo all'utente.

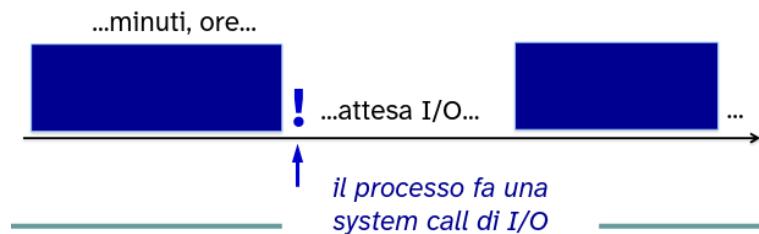
Quindi deve essere una funzione su cui porre l'attenzione per la ottimizzazione, poiché inficia molto sulle performance generali del sistema.

Possiamo dire che tra gli scheduler è quello più caratterizzante del SO.

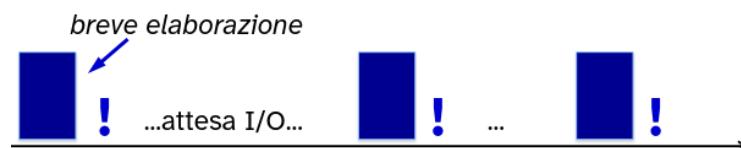


Processi CPU bound e I/O bound

- Processi **CPU-bound**
 - poche chiamate di sistema
 - tendono ad occupare la CPU per lunghi periodi (se il SO non li interrompe)
 - tipico delle applicazioni batch, calcolo numerico



- Processi **I/O-bound**
 - fanno frequenti chiamate di sistema
 - usano brevemente la CPU, per poi mettersi subito in attesa di I/O
 - tipico dei programmi **interattivi** (es. browser, editor di testo, ...)



Scheduler a breve termine

Lo scheduler a breve termine, conosciuto anche con il nome di **dispatcher**, è lo scheduler che viene attivato più frequentemente nel sistema.

Ha il compito di **scegliere a quale tra i processi pronti** assegnare la CPU.

Lo scheduler a breve termine viene invocato all'occorrenza di un evento che comporta la sospensione del processo in esecuzione.

Tali eventi sono per esempio:

- interruzioni del Clock (timer)
- interruzioni di I/O
- system call
- segnali (ad esempio semafori su cui erano sospesi per la cooperazione)

L'obiettivo dell'algoritmo del **Dispatcher** è quello di allocare il processore in maniera tale da ottimizzare uno o più aspetti del comportamento del sistema.

I criteri più utilizzati sono:

- User-oriented

Si riferiscono ai comportamenti del sistema così come **percepiti dall'utente o da un processo**.

→ Ad esempio il tempo di risposta.

- System-oriented

L'obiettivo è quello di **utilizzare in modo efficiente il processore**.

Quindi massimizzare l'utilizzo della risorsa processore.

→ Ad esempio migliorare il throughput di esecuzione.

Ovviamente il comportamento del sistema comprende diverse caratteristiche possibili che possono essere ottimizzate.

Ma molte di queste fanno in conflitto.

Infatti enfatizzando le caratteristiche del sistema operativo che portano a massimizzare il throughput del processore si attenuano le caratteristiche che si riferiscono alla percezione dell'utente o dei processi utente, come ad esempio il tempo di risposta.

Quindi non esiste un algoritmo universale a cui aspirare per ottenere il meglio in tutti i campi possibili. Ma esistono i migliori algoritmi rispetto agli obiettivi di progetto scelti per la realizzazione del SO.

Parametri User-oriented

- **Tempo di turnaround:** intervallo di tempo che trascorre dall'istante in cui un processo è ammesso nel sistema all'istante della sua terminazione.
Tiene conto del **tempo effettivo di esecuzione e del tempo speso in attesa delle risorse** (CPU inclusa).
- **Tempo di risposta:** indica il tempo che trascorre dall'istante della ammissione nel sistema (quindi quando entra nella coda dei processi pronti per la prima volta) all'istante in cui fornisce la prima risposta.
- **Deadlines:** nel caso in cui si possa specificare per ogni processo il termine di completamento (deadline), l'algoritmo dello scheduler a breve termine deve seguire una metrica che porta a **massimizzare la percentuale scadenze raggiunte**.
- **Tempo di servizio:** è il tempo di esecuzione “puro” del processo, cioè quanto tempo serve al processo per terminare se avesse la CPU tutta per sé, senza interruzioni o attese.
- **Slowdown** è il rapporto tra il *tempo di turnaround* e il *tempo di servizio*.

Misura quanto un processo è stato rallentato dal sistema rispetto al tempo che avrebbe impiegato se fosse stato eseguito da solo, senza attese.

Un valore pari ad **1** indica che il processo ha avuto esecuzione immediata (nessuna attesa).

Parametri System-oriented

- **Throughput:** misura la **produttività** di un sistema in termini di **numero di processi terminati per unità di tempo**.

Questo parametro **dipende** fortemente dalla **lunghezza media di un processo** ma è anche **influenzato** dalla particolare **politica adottata per la schedulazione**.

- **Utilizzo della CPU:** rappresenta la percentuale di tempo per cui il processore risulta occupato.

L'obiettivo è di massimizzare la percentuale d'uso della CPU nell'unità di tempo.

- **Fairness:** misura quanto i processi attivi nel sistema, che quindi sono pronti ad eseguire o sono bloccati in attesa di un evento asincroni, sono **trattati equamente in termini di schedulazione**.

Questo parametro esclude qualsiasi concetto di priorità dei processi all'interno del sistema.

→ evita che ci siano situazioni di starvation per un processo.

Esiste un algoritmo perfetto?

- Naturalmente NO!

Ovviamente i parametri descritti che riguardano l'User-oriented e il System-oriented sono interdipendenti tra loro ma alcuni sono nettamente in contrasto.

Ad esempio ottenere una politica *fair* non mi garantisce una massimizzazione dell'*utilizzo della CPU* o il *deadlines*.

Mentre ottenere una politica che va verso la minimizzazione del *turnaround time* porta anche un miglioramento del *tempo di risposta*.

Il progetto e l'implementazione di una politica di scheduling implica sempre un compromesso tra vari requisiti contrastanti.

La scelta dovrà essere fatta tenendo conto “per cosa dovrà essere utilizzato il sistema”.

Utilizzo delle priorità

Lo scheduler può scegliere i processi in base alla loro priorità.

Le priorità assegnate ai processi possono essere:

- **Statiche**, se non si modificano durante il periodo di vita del processo nel sistema.
- **Dinamiche**, se durante il loro ciclo di vita i processi possono modificare la loro priorità in base ad alcuni parametri come: tempo di CPU o tempo di I/O.

I processi sono tipicamente raggruppati in **classi di priorità**

- L'algoritmo di scheduling dovrà scegliere un processo pronto che appartiene alla classe di priorità più alta.

Starvation

L'utilizzo di un algoritmo di scheduling a priorità può indurre situazioni di **attesa indefinita** di processi a priorità più bassa (**starvation**).

Per far fronte a queste situazioni si utilizzano schemi di priorità dinamiche.

Come ad esempio, utilizzare un aumento graduale della priorità dei processi che si trovano in attesa nel sistema da lungo tempo → **Aging**.

EPISODIO esemplificativo dei problemi che può causare la starvation:

L'IBM 7094, un computer progettato per applicazioni scientifiche e tecnologia su larga scala, ha avuto il più eclatante problema di starvation.

Questo computer è stato utilizzato dalla NASA e per operazioni militari ed è stato introdotto nel 1962 e ha terminato il suo servizio nel 1973.

Quando è terminato il servizio gli utenti trovarono un processo a bassa priorità che fu ammesso nel sistema nel 1976 non è mai terminato perché nella coda dei processi pronti si presentavano ogni volta processi a priorità maggiore.

In questi casi la soluzione è quella di implementare la tecnica di Aging che consiste nel aumentare gradualmente la priorità di un processo in attesa sulla coda dei processi pronti rispetto ad alcuni parametri, come il tempo di attesa per ottenere la risorsa CPU.

Classificazione relativa al momento in cui interviene lo scheduling

Una prima classificazione fra gli algoritmi di scheduling è relativa alla scelta di quali siano gli eventi in seguito ai quali lo scheduler deve intervenire:

- Si indicano come algoritmi di scheduling senza diritto di revoca (*non preemptive*) tutti quelli che prevedono l'intervento dello scheduler esclusivamente quando il processo in esecuzione libera spontaneamente la CPU, o perché termina la propria esecuzione o perché si sospende in attesa del verificarsi di un dato evento asincrono.
- Invece, vengono indicati come algoritmi con il diritto di revoca (*preemptive*) quelli che prevedono l'intervento dello scheduler anche per decidere di revocare la CPU al processo in esecuzione al fine di allocarla ad un altro processo in attesa sulla coda dei processi pronti.

Per esempio a causa dell'arrivo di un segnale di interruzione che indica lo scadere del quanto di tempo assegnato al processo, oppure perché entra in coda dei processi pronti un processo ritenuto “urgente”.

In entrambi i casi il processo in esecuzione viene forzato a rilasciare la risorsa CPU preventivamente e inserito nella coda dei processi pronti.

Gli algoritmi *non preemptive* sono sicuramente quelli più semplici e riducono il numero delle volte che lo scheduler deve intervenire e quindi il **numero di cambi di contesto** fra processi. → Richiedono un minor *overhead* di sistema.

Allo stesso tempo sono anche quelli che non offrono una flessibilità in termini di strategie di scheduling.

Deadlock

Deadlock è un fenomeno che può presentarsi nelle **applicazioni concorrenti** che porta a un blocco permanente di processi in **competizione per risorse condivise**.

→ può portare a crash, situazioni di stallo, etc. (da evitare completamente)

Nei sistemi operativi **general-purpose**, si ammette l'esistenza di deadlock → non si implementano tecniche per evitare il problema ma si rileva nel momento in cui si presenta e si cerca di risolverlo.

Deadlock: definizione e generalità

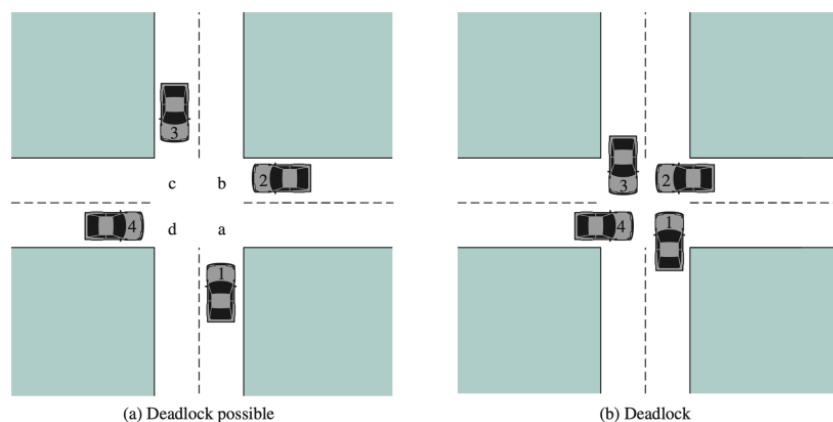
Indica una situazione di **blocco permanente** di un gruppo di processi in competizione per le risorse di sistema (che sono limitate → motivo principale per cui si deve gestire la competizione).

Deadlock è un problema complesso e di rilievo, che può provocare gravi malfunzionamenti.

A seconda dello scopo per cui è progettato, un sistema operativo adotta una **gestione** diversa per controllare il deadlock.

Ad esempio, per i sistemi **real-time** evitare il deadlock è fondamentale, quindi avranno una gestione più rigida che tende a prevenire tali fenomeni a differenza di altri tipi di sistemi operativi, come **general-purpose**, che potranno tendere a non rilevarli e risolverli a posteriori.

Esempio: attraversamento di un incrocio



Sono presenti diversi quadranti identificati dalle lettere **a**, **b**, **c** e **d**; che possono rappresentare le risorse critiche.

- Ogni auto ha bisogno di attraversare due quadranti;
- Ogni auto rappresenta un processo di un sistema operativo.

Ogni si inserisce nel rispettivo primo quadrante, ma per completare la curva deve ottenere l'accesso al quadrante alla loro destra (o sinistra) occupato da un'altra auto, e per quest'ultima vale lo stesso ragionamento.

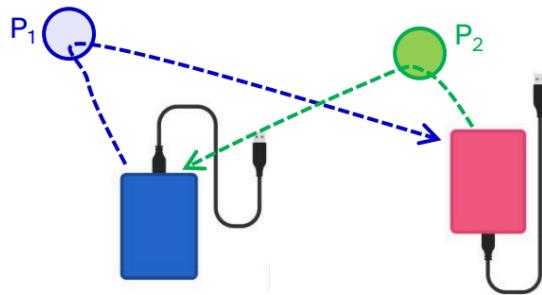
Facendo lo stesso ragionamento si crea un situazione di stallo (se nessun'auto indietreggia).

Esempio: copia di un file

- un sistema ha 2 dischi esterni
- P_1 e P_2 copiano un grosso file da un disco all'altro;
- si suppone sia necessaria la **mutua esclusione**.

Quindi ciascun processo acquisisce uno dei due dischi per leggere e successivamente richiedono l'accesso all'altro.

Ma l'altro disco si ritrova già acquisito. Quindi i due processi attendono a vicenda che l'altro rilasci la risorse.



→ si crea la cosiddetta **attesa circolare** che è la manifestazione del deadlock.

Il problema del deadlock

Una condizione **necessaria** affinché nelle **applicazioni concorrenti** si verifichi un deadlock è che siano presenti almeno due semafori **mutex**, ovvero entrambi inizializzati ad 1.

Quindi quando è presente la **mutua esclusione**.

P₁

```
wait (mutex1)  
<inizio uso disco 1>  
...
```

P₂

```
wait (mutex2)  
<inizio uso disco 2>  
...
```

wait (mutex2)

<inizio uso disco 2>

...

signal (mutex2)

...

signal (mutex1)

wait (mutex1)

<inizio uso disco 1>

...

signal (mutex1)

...

signal (mutex2)

I due processi potrebbero sospendersi entrambi su le `wait()` evidenziate.

Questo è un classico esempio di attesa circolare che si presenta se avviene la seguente sequenza di azioni:

- P1 esegue `wait(mutex1)` e acquisisce la prima risorsa critica;
- P2 esegue `wait(mutex2)` e acquisisce la seconda risorsa critica;
- P1 ha bisogno della seconda risorsa critica, quindi esegue `wait(mutex2)` e si blocca;
- P2 ha bisogno della prima risorsa critica, quindi esegue `wait(mutex1)` e si blocca.

Nel caso in cui l'esecuzione dei due processi segue questa sequenza, i due processi rimangono bloccati

Questa situazione però **non è detto** che accada a ogni esecuzione dell'applicazione concorrente.

→ **non** è un fenomeno **deterministico**, ovvero non significa che questo si manifesti ogni volta che si hanno le condizioni adatte.

In alcuni casi il deadlock dipende da come i processi si alternano sulla CPU.

Per questo motivo il deadlock può manifestarsi **saltuariamente**, in base alla **velocità relativa di esecuzione dei processi**.

La velocità relativa di esecuzione dei processi è un modo per descrivere come i processi avanzano nel tempo l'uno rispetto l'altro, questo avanzamento non è deterministico perché durante l'esecuzione possono:

- essere interrotti dal kernel (preemption),
- essere sospesi in attesa di un'operazione di I/O (la cui durata non è sempre la stessa),
- ricevere più o meno tempo di CPU in base alla politica di scheduling adottata,
- essere ritardati da cache miss.

Come si potrebbe risolvere questa situazione?

Un terzo dovrebbe gestire questa concorrenza, andando ad esempio a killare un processo in modo che questo liberi la risorsa detenuta a favore dell'altro.

Deadlock \neq Starvation

attesa infinita **attesa indefinita**

Deadlock e starvation sono due concetti totalmente diversi che possono essere confusi.

- Con starvation identifichiamo una situazione di attesa **indefinita**. Tale fenomeno è molto legato al concetto di priorità.
- Con deadlock identifichiamo una situazione di attesa **infinita**, in nessun modo i processi possono uscire da questa situazione, al contrario della starvation. Tale fenomeno è molto legato al concetto di mutua esclusione.

Grafo di assegnazione delle risorse

Il grafo di assegnazione delle risorse serve a modellare in modo formale lo stato delle risorse e dei processi di un sistema, per poter **capire se esiste il rischio di deadlock**.

Tramite questa formalizzazione possiamo implementare algoritmi in grado di rilevare una possibile manifestazione di deadlock tra i processi in esecuzione.

Un grafo è un insieme di vertici (o nodi) V e un insieme di archi E .

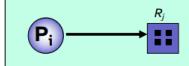
- V è partizionato in **due tipi**:
 - $P = \{P_1, P_2, \dots, P_n\}$ è l'insieme costituito da tutti i **processi** nel sistema.
 - $R = \{R_1, R_2, \dots, R_n\}$ è l'insieme costituito da tutti i tipi di **risorse** nel sistema.
- **Arco di richiesta** (arco orientato) $P_i \rightarrow R_j$, P_i chiede l'accesso a R_j .
- **Arco di assegnazione** (arco orientato) $R_j \rightarrow P_i$, R_j è assegnata al processo P_i .

Ogni risorsa può avere più istanze.

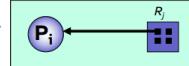
- Processo 

- Tipo di risorsa con 4 istanze 

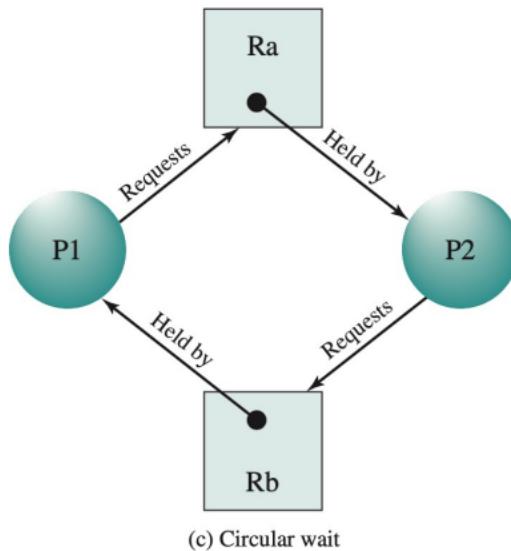
- P_i richiede un'istanza di R_j



- P_i possiede un'istanza di R_j



Una condizione **sufficiente** per la possibile (perché dipende sempre dalla velocità relativa di esecuzione) manifestazione di un deadlock è un **ciclo** nel grafo di assegnazione.

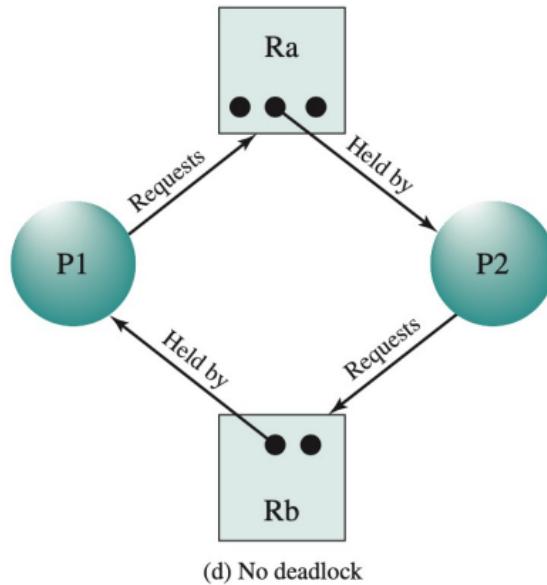


In questo caso P_1 richiede R_a la cui unica istanza è detenuta da P_2 che richiede a sua volta R_b la cui unica istanza è detenuta da P_1 .

Nel momento in cui è presente un ciclo del genere possiamo essere **sicuri** che tra questi processi è **possibile** che avvenga una situazione di deadlock.

Invece possiamo considerare che **si verifica** il deadlock se i processi partono da una condizione iniziale in cui hanno già in possesso le rispettive risorse e facciano una richiesta per l'altra, quando nessuno dei due processi ha terminato.

Ovviamente nel caso in cui le risorse avessero più istanze non ci sarebbe un problema, perché i due processi accederebbero a istanze diverse e quindi non si violerebbe la **mutua esclusione**.



→ Questo **non** significa che avere più istanze **risolva** il problema, perché basta che si aggiungano altri processi al grafo con una particolare configurazione che la situazione di deadlock potrebbe accadere.

OSSERVAZIONI:

- Un ciclo è una condizione sufficiente per la possibile condizione di deadlock.
 - Se il grafo **non contiene cicli** ⇒ non si verificano situazioni di stallo
- Se il grafo **contiene un ciclo** ⇒ si potrebbe verificare una situazione di stallo, la cui possibilità diminuisce con il numero di istanze per ogni risorsa.

Quindi la possibilità che ci sia un deadlock esiste ma non è detto che si verifica, perché tutto dipende anche dalla velocità relativa di esecuzione di ogni processo.

Un caso che possiamo considerare UTOPICO è quando ogni risorsa ha tante istanze quanti siano i processi che potenzialmente possono richiederla → impossibile proprio perché non sappiamo il numero di processi che potenzialmente è troppo elevato e le risorse sono limitate.

Linux cosa fa? Quando rileva un deadlock tenta di **eliminare** (a posteriori) questa condizione andando a terminare uno dei processi scatenanti, la cui scelta dipende da delle metriche.

Quindi accetta che questa situazione può verificarsi e nel momento in cui è tale risolve il problema mediante una sua politica di gestione.

Metodi per la gestione dei deadlock

Esistono diversi approcci per gestire la situazione di deadlock che consistono principalmente in prevenirli (a priori) o rilevarli (a posteriori).

1. Prevenzione dei deadlock (**PREVENTION**):

rendere **impossibile** il verificarsi delle **condizioni di deadlock**, ma al costo di un basso utilizzo delle risorse.

Questo approccio tenta di annullare le condizioni che possono causare un deadlock → evitano che si creino dei cicli nel grafo di assegnazione delle risorse.

Ma nel fare questo **limitano l'utilizzo delle risorse non sfruttandole a pieno**, quindi rallentando il sistema.

2. Evitare i deadlock (**AVOIDANCE**):

le condizioni per il deadlock sono consentite, quindi si ammette l'esistenza delle condizioni tali per cui esso può avvenire, ma il sistema **evita di entrare** in uno stato di deadlock.

Evita la condizione **analizzando ogni richiesta** di risorse prima di concederla, e **accettandola solo** se non porta il sistema in uno “**stato non sicuro**”.

3. Rilevazione del deadlock (**DETECTION**):

Si permette al sistema di entrare in uno stato di deadlock, per poi risolvere il problema (**ripristino il sistema**).

Quindi si ammette che ci possa esser la condizione di deadlock, non si fa nulla per evitarla, ma nel momento in cui questa si verifica il sistema tenta di tornare in uno stato sicuro.

Ovvero si gestisce il deadlock solo dopo che questo si verifica.

La maggioranza dei sistemi operativi general-purpose, inclusi UNIX e Windows, **non dispone di una soluzione generale ed efficiente** al problema del deadlock.

Poiché tutte le politiche di gestione elencate hanno grandi problemi per cui non possono essere adottati.

NOTA: se creo due processi e faccio in modo che questi vadino in deadlock, il sistema operativo non fa nulla. Sarà un problema demandato al programmatore offrire una soluzione alla specifica situazione.

Condizioni per il deadlock

Le condizioni **necessarie** sono:

- **Mutua esclusione** → un processo per alla volta può usare la risorsa.
- **Impossibilità di prelazione** → una risorsa può esser rilasciata **solo volontariamente** dal processo che la possiede, al termine della sua esecuzione.

Quindi non esistono casi in cui a un processo venga prelazionata una risorsa a favore di un altro che la richiede.

- **Possesso e attesa** → un processo che possiede almeno una risorsa, **attende di acquisire ulteriori risorse** già possedute da altri processi.

Invece le condizioni **necessarie e sufficienti** sono (tutte quelle necessarie + attesa circolare):

- **Mutua esclusione**
- **Impossibilità di prelazione**
- **Possesso e attesa**
- **Attesa circolare** → che abbiamo visto essere una **condizione necessaria e sufficiente**.

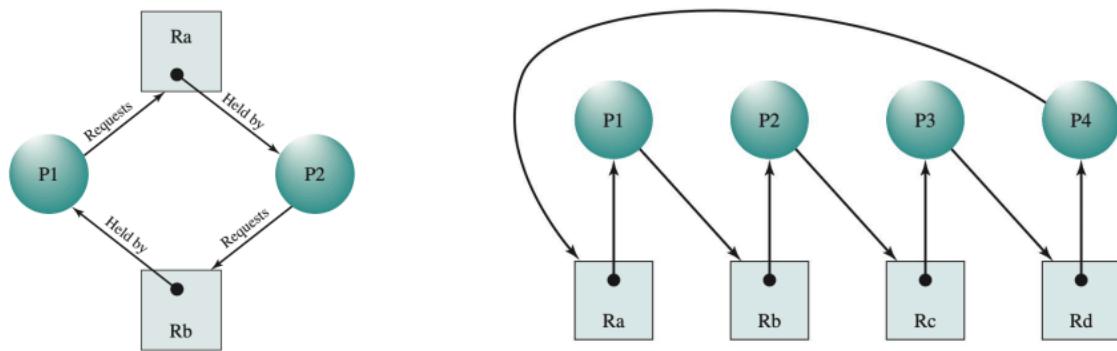
Affinché ci sia attesa circolare devono necessariamente esser verificate tutte le condizioni necessarie. Quindi possiamo dire che se è presente un'attesa circolare allora automaticamente è presente una condizione di deadlock.

(Attesa circolare è una conseguenza di un ciclo nel grafo di assegnazione, ma non è detto che si verifichi)

Attesa circolare è la condizione che si verifica nel momento in cui

- esiste un insieme $\{P_0, P_1, P_2, \dots, P_n\}$ di processi in attesa, tali che:
 - P_0 è in attesa per una risorsa che è posseduta da P_1
 - P_1 è in attesa per una risorsa che è posseduta da P_2
 - ...
 - P_{n-1} è in attesa per una risorsa che è posseduta da P_n
 - P_n è in attesa per una risorsa che è posseduta da P_0

Ovvero quando nel grafo di assegnazione si crea un ciclo di attesa perché non ci sono abbastanza istanze di risorse disponibili per cui, almeno uno, di questi processi non ha bisogno di attendere.



Deadlock PREVENTION

Nella deadlock **prevention**, si evita il deadlock **invalidando** una delle quattro condizioni necessarie e sufficienti.

Svantaggi nell'utilizzo di questo tipo di gestione del deadlock:

1. mancato uso di risorse che sono disponibili;
2. esecuzione rallentata dei processi. → la politica di gestione costringe a processi ad attendere anche quando non è necessario
3. non è possibile alcun tipo di cooperazione tra processi

Mutua esclusione

La mutua esclusione è imposta dalle caratteristiche della risorsa e spesso non è rimovibile a meno che non si serializzi l'esecuzione dei processi, quindi non si necessita di un mutex.

In questo caso però peggiorano molto le performance perché non si sfrutta più la concorrenza tra i processi.

- Può essere rilassata in alcuni casi di risorse condivisibili come, ad esempio, le risorse read-only.
- Comporta costi maggiori → il sistema deve garantire che una risorsa critica (che potrebbe generare inconsistenze) non sia accessibile da più processi che ne fanno richiesta.

Quindi che la risorsa è posseduta da un solo processo per volta.

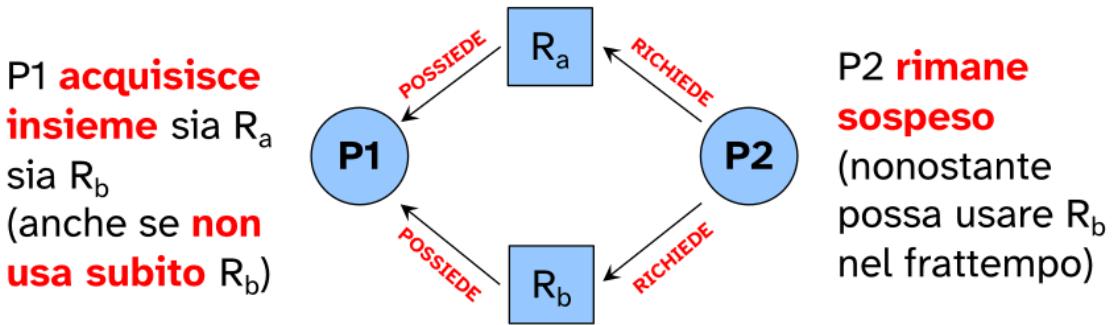
Possesso e attesa

Eliminando questa condizione si forza un processo a **richiedere una risorsa solo quando non ne possiede altre** (es. all'avvio richiede tutte le risorse necessarie alla sua esecuzione).

In questo caso si deve implementare una sorta di dichiarazione per ogni processo delle risorse che utilizza.

Tale dichiarazione deve contenere tutte le risorse necessarie che verranno bloccate per tutta l'esecuzione del processo. Quindi si può capire che è molto inefficiente come soluzione perché le risorse vengono bloccate per tutta l'esecuzione anche se il processo le utilizza in una piccola parte.

→ Approccio soggetto a **starvation**, perché potrebbe esistere un processo che è sempre in esecuzione e utilizza una risorsa che non potrà mai essere utilizzata da altri processi.



- In questo caso se P1 non termina mai → P2 non potrà mai terminare la propria esecuzione.

Impossibilità di prelazione

Rilassando il vincolo di impossibilità di prelazione se un processo già possiede alcune risorse, e ne richiede un'altra che non gli può esser allocata immediatamente, allora **rilascia tutte le risorse possedute**.

Quindi non si mette in attesa per la singola risorsa che richiede, mantenendo il possesso di quelle già allocate, ma libera tutte le risorse e si mette in attesa.

Tale processo quindi non si metterà in attesa per la sola risorsa in più richiesta ma anche per tutte le altre che possedeva e che ha rilasciato.

→ il processo verrà eseguito nuovamente solo quando può riottenere il possesso sia delle **vecchie che delle nuove risorse**.

Attesa circolare

Si stabilisce a priori un **ordinamento totale** tra tutte le risorse.

E si richiede che ogni processo richieda le risorse seguendo l'ordine prestabilito.

Quindi se un processo ha bisogno di utilizzare un certo numero di risorse deve richiedere l'accesso a queste nell'ordine prestabilito nonostante tale ordine non sia quello delle operazioni che effettua su queste.

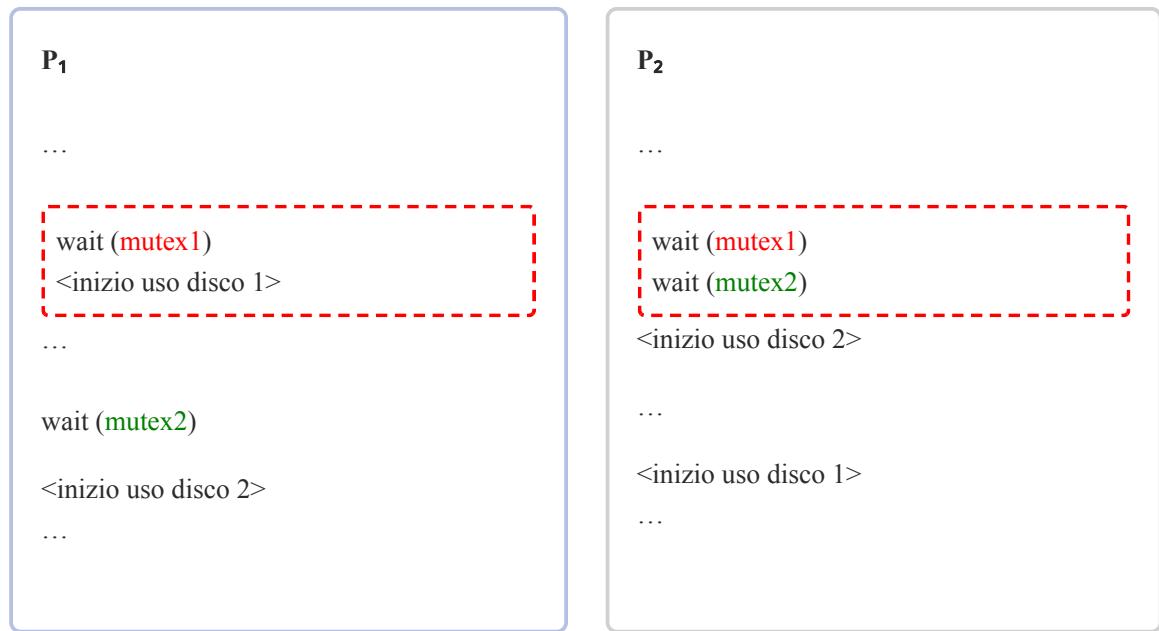
Nell'esempio successivo P2 chiede l'accesso prima a "disco 1" poi a "disco 2" nonostante operi inizialmente solo su "disco 2".

→ provoca una perdita delle performance perché un processo potrebbe possedere una risorsa per un tempo che è molto superiore rispetto al tempo effettivo nel quale opera su tale risorsa.

ESEMPIO:

Ordine imposto:

1. disco 1
2. disco 2



- In questo caso, supponendo che P2 faccia per **primo** la prima richiesta delle risorse:

- P1 è **impossibilitato a usare “disco 1”** anche se P2 sta usando “disco 2”.
 - Si è imposto un ordine di acquisizione delle risorse (a discapito dell’efficienza).

Questo tipo di approccio per la gestione PREVENTION non permette l’implementazione di una cooperazione (come anche gli altri approcci) tra processi.

→ implementando il problema produttori consumatori otteniamo che i produttori producono sempre fino a che non terminano. Solo dopo la terminazione dei produttori i consumatori potranno accedere ai dati prodotti. → comportamento non richiesto per l’implementazione.

Deadlock AVOIDANCE

Nella gestione AVOIDANCE il sistema decide **a tempo di esecuzione** se una richiesta di una risorsa può portare a un deadlock (**prevenzione dinamica**).

- **nessun vincolo a priori** delle risorse
- se lo stato attuale delle risorse è **rischioso**, un algoritmo **rifiuta la richiesta** di allocazione

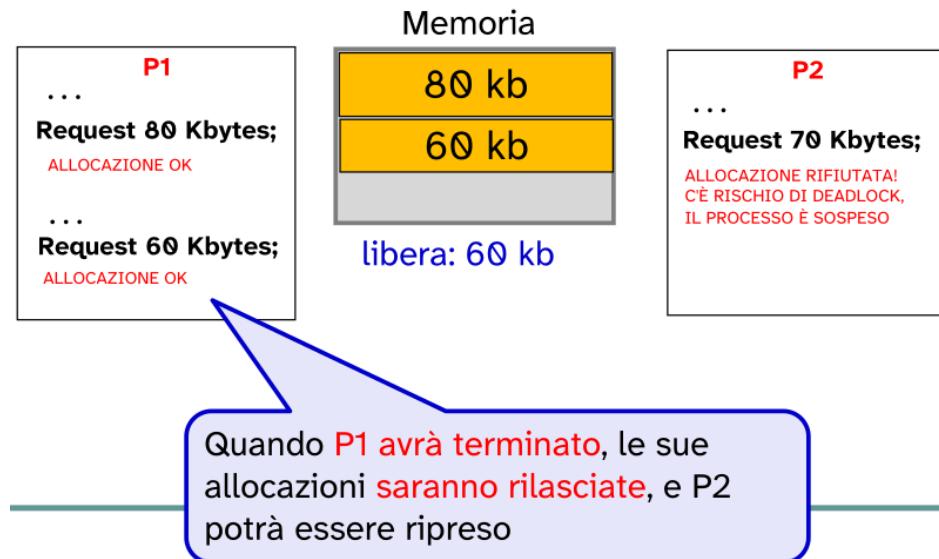
Quindi si accetta la possibilità di incorrere in un deadlock, non eliminando le condizioni necessarie, ma si cerca di evitarlo valutando lo stato in cui si trova il sistema ogni volta che viene effettuata una richiesta (a *run-time*).

Quindi istante per istante, possedendo la storia precedente del grafo delle assegnazioni delle risorse, un algoritmo valuta se **successivamente** a una certa richiesta da parte di un processo porta l'applicazione a un deadlock.

Quindi l'algoritmo deve essere in grado di fare una **sorsa di predizione sull'andamento dell'esecuzione** dei processi negli istanti successivi a una qualsiasi richiesta per una risorsa.

Quindi possiamo considerarla come una prevenzione, che **non è più statica** come per la gestione PREVENTION, ma **dinamica**.

Anche questa come soluzione al deadlock è molto complicata perché richiede diverse assunzioni: come quella di riuscire a prevedere le eventuali richieste di un processo se questo non le dichiara a priori.



La memoria totale è 200 kb.

- P2 non avrà accesso alla risorsa nel momento in cui la richiede perché il processo P1 è già in esecuzione e possiede già una istanza della stessa risorsa.

Però il motivo per cui viene rifiutata la richiesta di P2 è perché guardando la storia di P1 si nota che affinché questo possa terminare dovrà ottenere un'altra istanza della risorsa.

Nel momento in cui richiederà un'altra istanza della risorsa non ci sarà più spazio per l'istanza richiesta da P2

- Caso in cui avviene il deadlock → P2 riceva la risorsa (supponendo che successivamente ne richieda un'altra da 80 kb).

→ Quindi la memoria in possesso sarà 80 + 70 = 150kb.

→ Il processo P1 si sospende perché ne richiede 70kb e allo stesso modo si sospende P2 perché ne richiede, come detto, 80kb.

→ Entrambi i processi sono sospesi in attesa che l'altro rilasci la risorsa: deadlock.

Nell'esempio un algoritmo ha valutato questa situazione conoscendo la storia di allocazione dei processi.

Quindi è necessario supporre che per ogni processo si deve dichiarare la **storia di allocazione**, altrimenti non si possono fare previsioni sull'andamento dell'esecuzione.

Questo produce un **overhead** elevato sullo scheduler e in generale sul kernel. Ma da una soluzione per non entrare nel deadlock.

Linux implementa ciò? No perché non si conosce a priori la storia di allocazione di ogni processo → è impossibile prevedere un deadlock in questo modo.

Presupposto: queste tecniche richiedono di **conoscere in anticipo** tutte le richieste che un processo può fare nell'arco della sua esecuzione.

Questa è un'assunzione molto pesante, un caso più semplice di utilizzo è quello che: ogni processo **dichiara il numero massimo** di istanze di risorse di cui può avere bisogno.

→ Tali istanze però potrebbero non essere utilizzate subito e quindi tolte ad altri processi che potrebbero sfruttarle immediatamente.

Approcci

Abbiamo due diversi approcci per questo tipo di gestione:

1. **Process initiation Denial:** all'avvio di un nuovo processo (rifiuto l'esecuzione del processo)

Non si avvia un processo se le sue richieste potrebbero portare ad un deadlock

2. **Resource Allocation Denial:** al momento di una richiesta di allocare una risorsa. (il processo esegue ma possono essere vietate le richieste nonostante la disponibilità corrente è valida)

Si consente l'avvio, ma le richieste di allocazione possono essere rifiutate se possono portare a deadlock.

Entrambi questi approcci si basano sulla costruzione di diverse strutture algebriche:

sia n = numero di processi,

e m = numero di tipi di risorse

- **Resource** = $R = (R_1, \dots, R_m)$

Risorse totali nel sistema. R_i è il numero di istanze presenti nel sistema per la risorsa i -esima.

- **Available** = $V = (V_1, \dots, V_m)$

Numero di istanze per ogni risorsa non allocate ad alcun processo. V_i rappresenta il # di istanze della risorsa R_i non ancora allocate.

- **Claim** = $C = \text{matrice } n \times m$

C_{ij} = richiesta del processo P_i per la risorsa R_j

- **Allocation** = $A = \text{matrice } n \times m$

A_{ij} = allocazione corrente al processo P_i della risorsa R_j

Process Initiation Denial

- La matrice C (di richiesta) indica **il numero massimo di richieste** per ogni processo (righe), di una certa risorsa (colonne).
- Deve essere fornita prima dell'avvio dei processi

Quindi questa matrice verrà aggiornata e ricalcolata ogni volta che un processo termina o inizia la sua esecuzione.

$$\mathbf{C} = \begin{pmatrix} \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot \end{pmatrix} \begin{matrix} \text{processi} \\ (\text{Claim}) \\ \text{risorse} \end{matrix}$$

Un processo P_{n+1} viene eseguito solo se:

$$R_j \geq C_{(n+1)j} + \sum_{i=1}^n C_{ij} \quad \text{for all } j$$

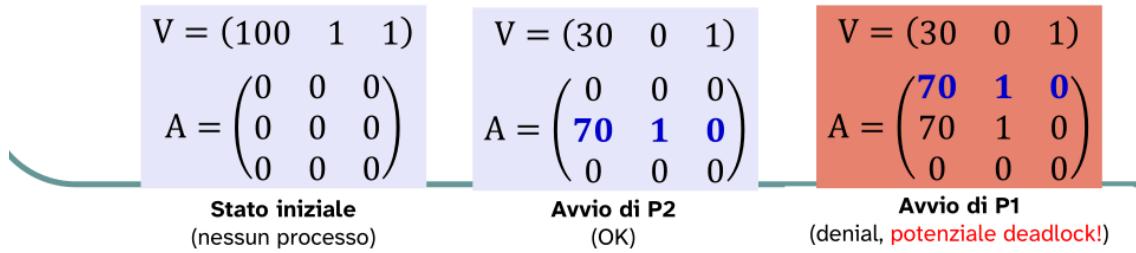
Cioè un processo viene eseguito se il **numero massimo di richieste di tutti i processi** (sommatoria) **più quelle del nuovo processo** (C_{ij}) per la risorsa j -esima è minore del numero di istanze della richiesta R_j .

Questo deve esser calcolato per ogni tipo di risorsa → per ogni colonna di C .

ESEMPIO:

- Tre tipi di risorse:
 - 100 MB di memoria
 - 1 file di log
 - 1 porta seriale
- Claims:
 - P1: 70 MB di memoria, 1 porta seriale
 - P2: 70 MB di memoria, 1 porta seriale
 - P3: 50 MB di memoria, 1 file di log

Risorse
$R = (100 \ 1 \ 1)$
Claim
$C = \begin{pmatrix} 70 & 1 & 0 \\ 70 & 1 & 0 \\ 50 & 0 & 1 \end{pmatrix}$



In questo esempio P1 non viene eseguito fin tanto che P2 non termina la sua esecuzione (almeno).

Il motivo per cui non viene eseguito è proprio il **vincolo** imposto da process initiation denial:

- nel momento in cui P1 tenta di esser eseguito.

Per MB di memoria:

$$100 (R1) \geq 70 (C11) + 70 (C21) + 0 (C31) = 140 \Rightarrow \text{non verificato}$$

Per la porta seriale:

$$1 (R2) \geq 1 (C12) + 1 (C22) + 0 (C32) = 2 \Rightarrow \text{non verificato}$$

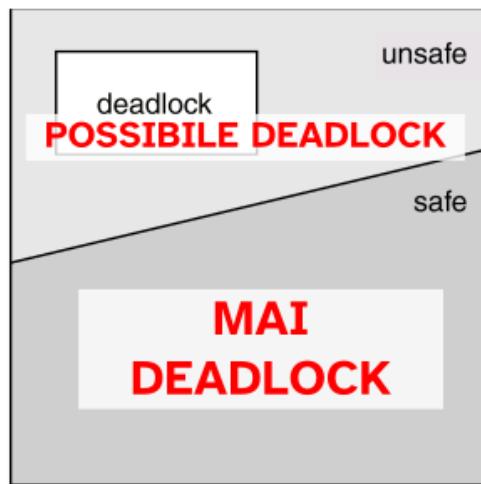
Resource Allocation Denial

Tale approccio viene chiamato anche **algoritmo del banchiere**

→ viene eseguito ad ogni tentativo di allocazione;

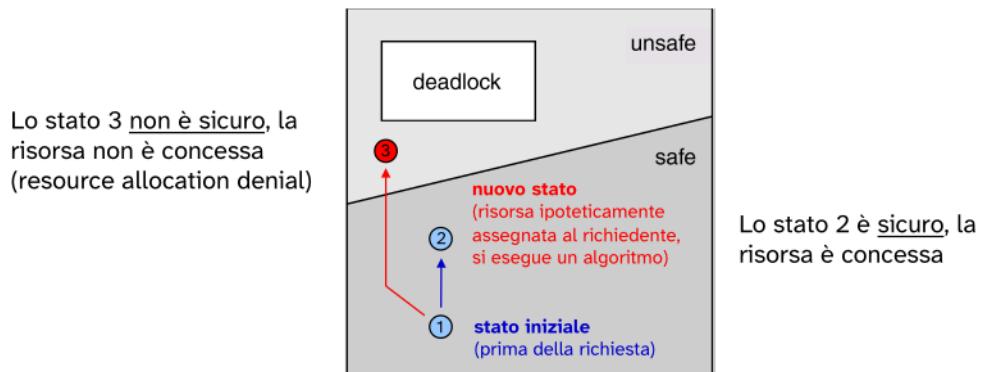
- se l'allocazione può portare ad uno stato “non-sicuro” viene rifiutata.

L'algoritmo fa in modo che lo stato del sistema (risorse e processi) **non sia mai uno stato non sicuro**



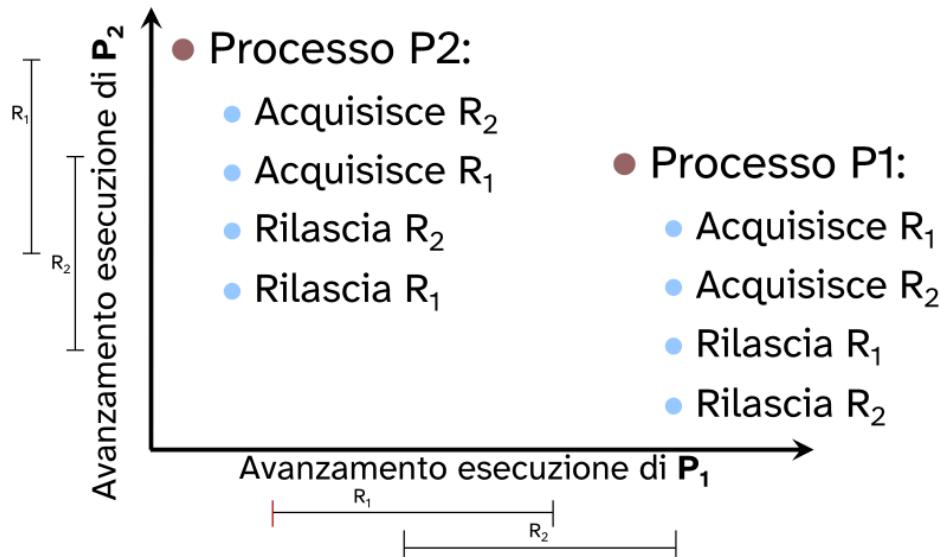
La **strategia** consiste nel trovare una sequenza di esecuzione *safe*.

La sicurezza di uno stato dipende dalle risorse disponibili, e dalle richieste di tutti i processi nel sistema.



Quindi l'obiettivo è fare in modo di determinare una sequenza di esecuzione che non faccia mai entrare lo stato del sistema nella porzione del piano *unsafe*.

ESEMPIO:

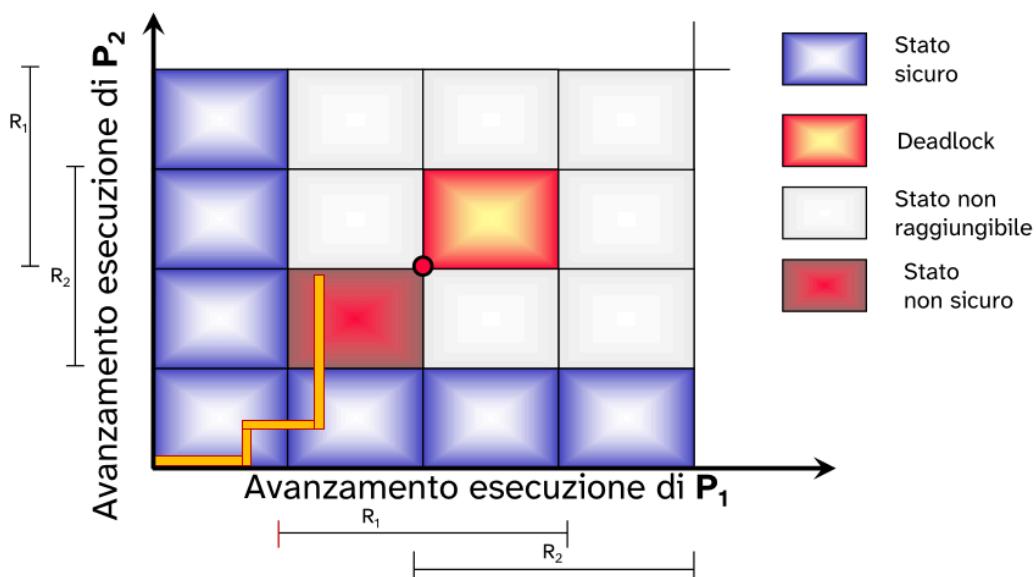


In questo esempio si può vedere la sequenza delle operazioni di richiesta e rilascio delle risorse R1 e R2 per i processi P1 e P2.

Si nota subito che questa configurazione **potrebbe** portare ad uno stato non sicuro (deadlock) a run-time (dipende dalla velocità relativa di esecuzione).

Infatti non è detto che possa esserci un deadlock, perché come vediamo l'algoritmo del banchiere ha come risultato due sequenze sicure per lo stato iniziale.

→ quindi lo stato iniziale è uno stato sicuro, ciò significa che l'esecuzione potrebbe arrivare a terminare senza il manifestarsi di **deadlock** (se viene seguita una delle sequenze sicure).



- Il modo in cui sono stati eseguiti i due processi hanno portato lo stato ad essere **non sicuro**, in questo caso è inevitabile il deadlock.
- Entrambi i processi resteranno in attesa l'uno dell'altro.

Sequenza sicura

Il sistema è in uno **stato sicuro** se, partendo da questo stato, **esiste un sequenza sicura** di esecuzione di tutti i processi nel sistema.

Tale sequenza è una sequenza di **esecuzione “ipotetica”** dei processi nel sistema che porta al processo richiedente di una risorsa a terminare la propria esecuzione. (es. Pa, Pb, Pc, ...)

(che porta a tutti i processi del sistema a terminare)

Affinché uno stato sia sicuro è sufficiente che esista almeno una sequenza sicura.

L'algoritmo del banchiere prevede proprio di trovare la sequenza sicura che verifichi lo stato corrente, se al processo che ha richiesto una risorsa la ottiene.

Se esiste almeno una sequenza sicura,

tale che il processo richiedente possa terminare dopo una serie di terminazioni di altri processi.

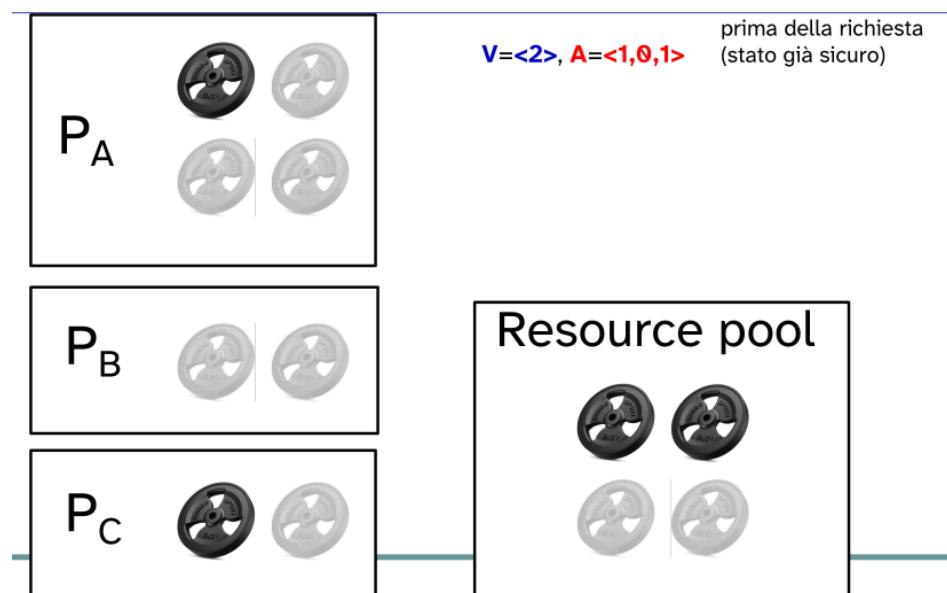
(tale che tutti i processi del sistema possano terminare dopo una serie finita di terminazioni di altri processi)

→ Allora lo stato è **sicuro** e la richiesta viene accettata.

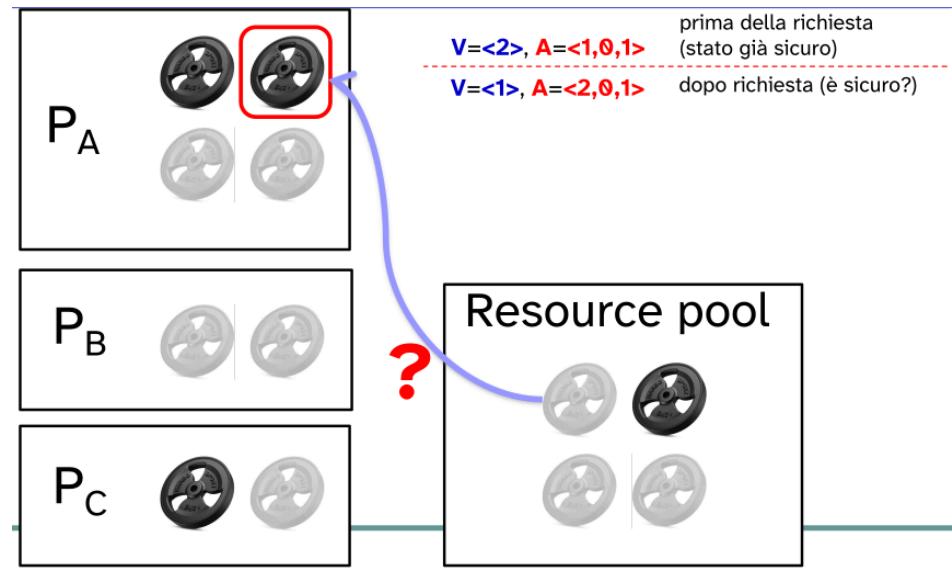
→ Altrimenti se **non esiste** lo stato non sarà sicuro e quindi la richiesta viene **rifiutata**; il processo si mette in **attesa** e verrà riattivato solo nel momento in cui la sua richiesta porti in uno stato sicuro.

workflow

- Si parte da uno stato che si suppone esser sicuro.



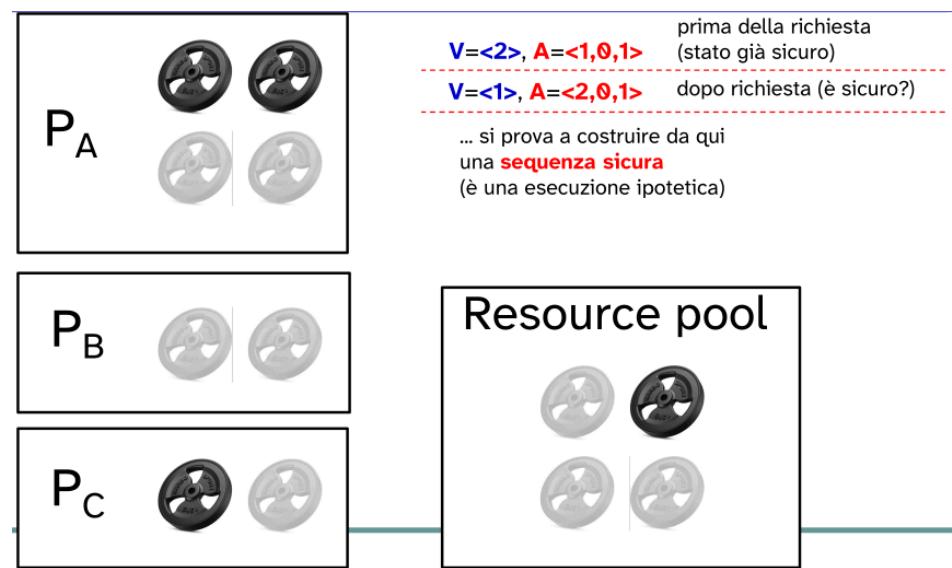
- Un processo Pa fa una richiesta di istanze di risorsa.



- Dopo tale richiesta si verifica che lo stato sia sicuro, supponendo che la richiesta sia stata accettata.

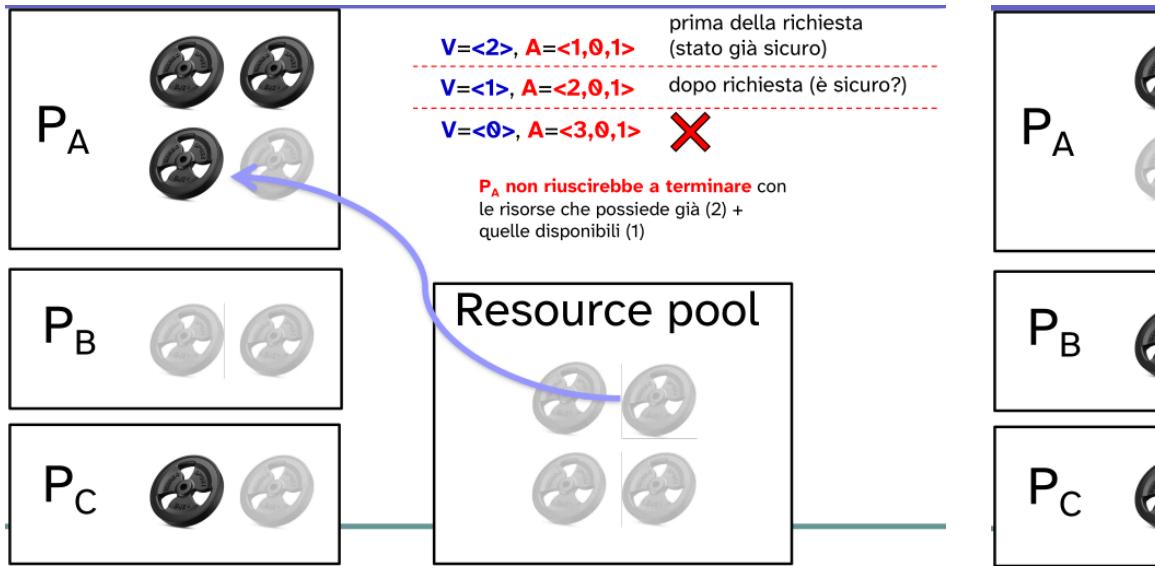
→ come avviene tale verifica:

- si tenta di trovare una sequenza sicura partendo dallo stato in cui la richiesta di Pa sia stata accettata.

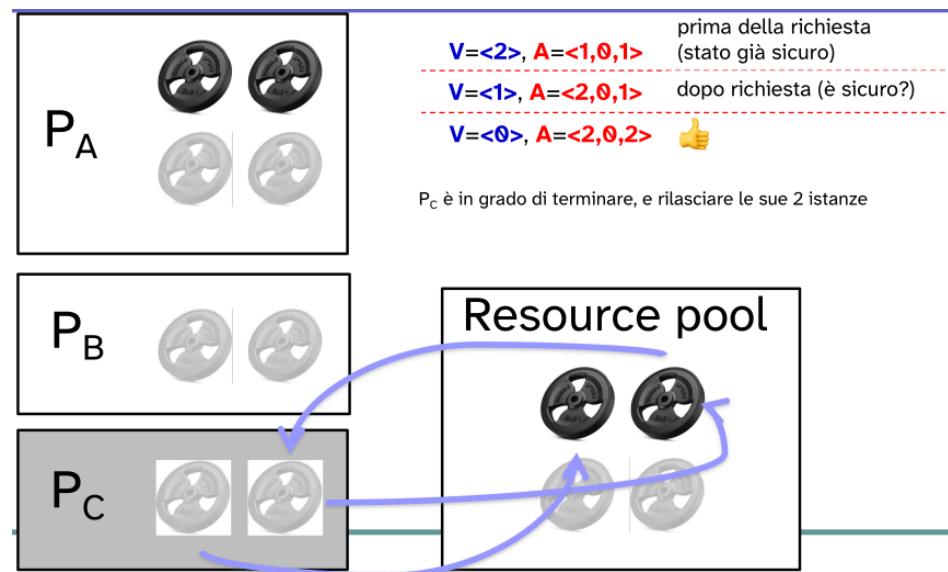


- si considera **ogni processo Pi** in esecuzione che necessita dello stesso tipo di risorse (Pa,Pb,PC);
- si verifica se il processo Pi possa terminare, con **le risorse disponibili rimanenti**, considerando che la richiesta di Pa sia stata accettata.

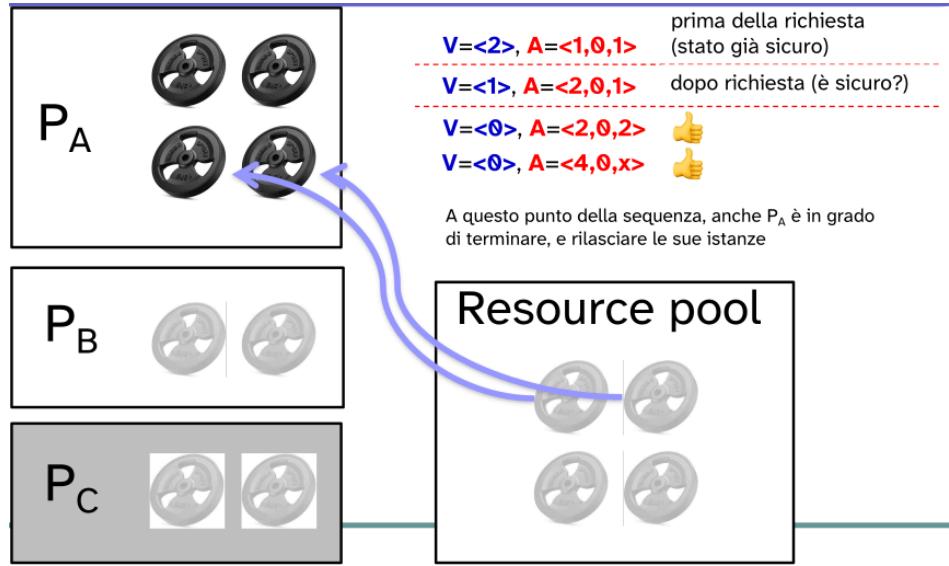
Quindi si suppone di assegnare al processo P_i tutte le risorse del suo **claim**, se disponibili.
Se c'è disponibilità si suppone che il processo termini e rilasci le risorse possedute;



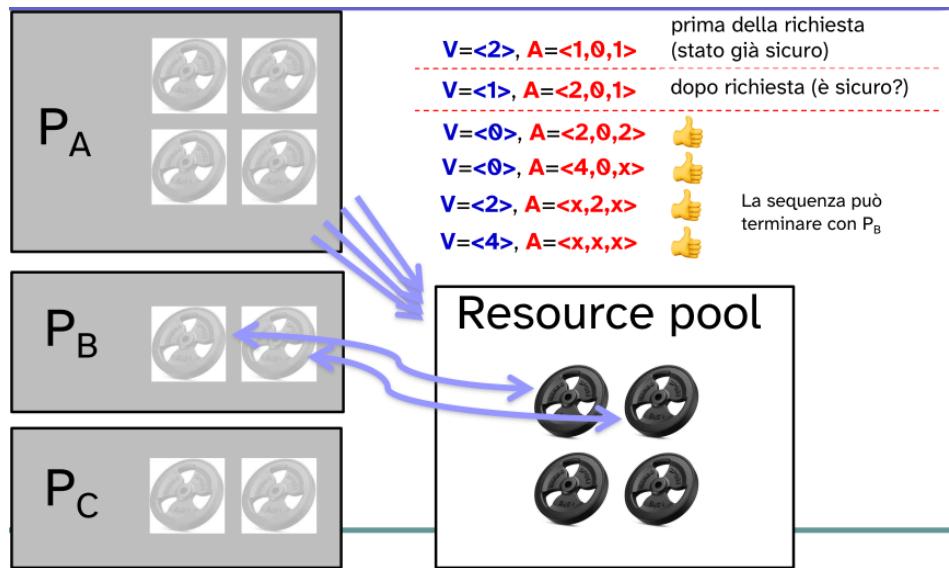
- se il processo i -esimo riesce a terminare allora potrà essere aggiunto alla **sequenza sicura**; altrimenti si passa al prossimo processo.



- P_C è in grado di terminare quindi viene aggiunto alla sequenza sicura di esecuzione.
- Si suppone che dopo la sua terminazione **rilasci** tutte le risorse possedute.
→ in modo da permettere ad altri processi di terminare.



- P_A riesce a terminare, viene aggiunto alla sequenza sicura.

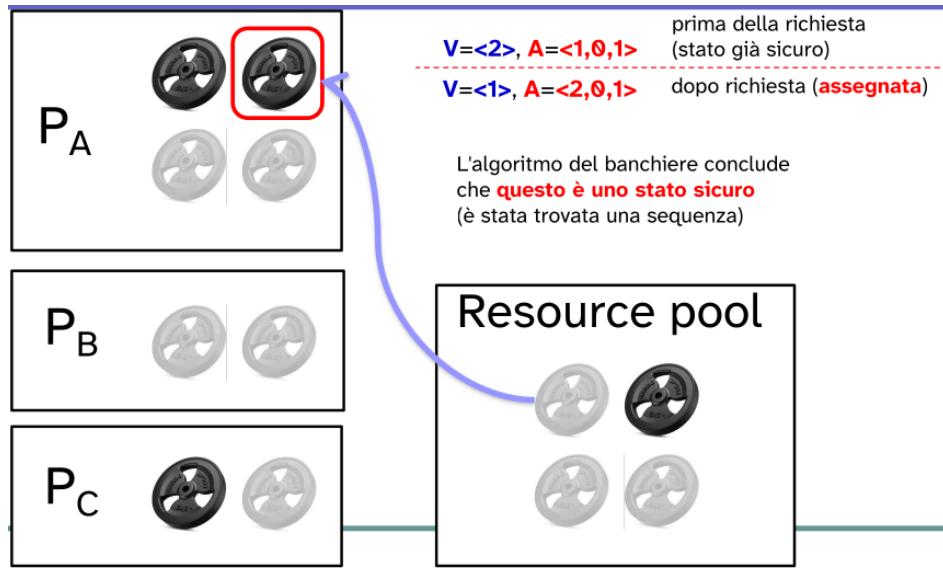


- Anche P_B riesce a terminare.
→ Si è trovata una sequenza sicura di esecuzione in cui tutti i processi terminano a valle dell'allocazione di risorse a P_A.

Si itera questo procedimento fin quando:

- il prossimo processo a far parte della sequenza sicura è il processo richiedente, P_A.
- A questo punto **termina l'algoritmo con una sequenza sicura di esecuzione**.

⇒ la richiesta viene accettata perché lo **stato**, dopo l'allocazione delle risorse al processo P_A, è **sicuro**.



oppure

- se per un'iterazione **non** si trova alcun processo in **grado di terminare completamente**, supponendo di assegnargli tutte le risorse del **claim (se disponibili)**.
- Allora il risultato dell'algoritmo è che non esiste una sequenza sicura.

⇒ la richiesta di Pa viene **rifiutata** perché lo stato successivo se la richiesta fosse accettata sarebbe **non sicuro**.

Il processo rimane in attesa fin quando la propria richiesta non porti in uno stato sicuro.

caratteristiche di una sequenza sicura

La sequenza sicura (P_1, P_2, \dots, P_n) è un ordine di esecuzione dei processi, tale che:

- include **tutti i processi attualmente attivi** nel sistema;
- ogni processo P_i esegue **completamente**, dopo che tutti i processi precedenti P_j , $j < i$, abbiano a loro volta **eseguito per intero** e nell'ordine della sequenza;
- Ogni processo P_i ottiene tutte le risorse del suo **claim**, e le **rilascia tutte** al termine della sua esecuzione;
- ogni processo P_i usa una **quantità di risorse non superiore** alla somma di:
 - risorse **disponibili** nello stato S
 - risorse **rilasciate** dei processi **precedenti** nella sequenza, P_j con $j < i$

considerazioni

Per un corretto funzionamento è necessario che l'algoritmo sia sempre **eseguito ad ogni tentativo di allocazione** di risorse.

In ogni momento in cui lo stato cambia, si verifica se esiste almeno una sequenza sicura.

Intuitivamente, l'algoritmo garantisce sempre che **esista almeno una exit strategy** che evita il deadlock.

Questa sequenza sicuro non è detto che sia l'ordine effettivo con cui eseguiranno i processi.

problematiche:

- è richiesto che sia **noto preventivamente** il numero **massimo** di risorse che ogni processo utilizzerà;
- i processi che vengono analizzati dell'algoritmo devono essere indipendenti (non è prevista la sincronizzazione).

Altrimenti il problema si complicherebbe eccessivamente

→ si dovrebbe tener conto che un processo possa terminare solo se termini prima un altro processo;

- deve esser presente un numero predeterminato e costante di risorse da allocare;
- Tutti i processi devono rilasciare le risorse possedute prima di terminare.

Deadlock DETECTION

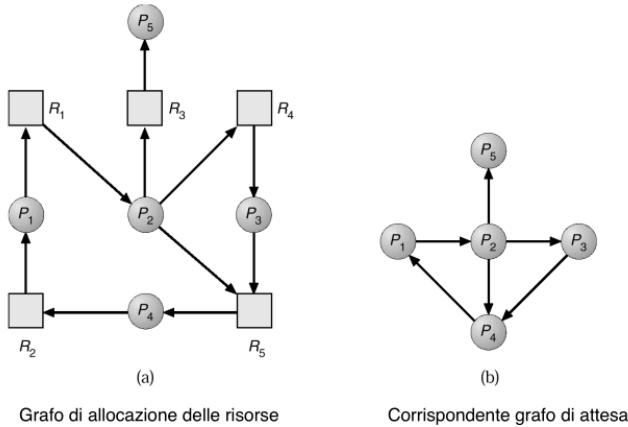
- Non vincola le richieste alle risorse, consente il verificarsi del deadlock.
- Il sistema esegue un algoritmo **per il rilevamento dell'attesa circolare**:
 - periodicamente;
 - ad ogni richiesta;
 - quando il grado di uso della CPU è basso.
- In caso affermativo, il sistema applica un algoritmo di **ripristino** (recovery).

La strategia di detection sfrutta il **grafo di attesa**, che è un grafo costruito da quello di assegnazione delle risorse.

Tale grafo rappresenta l'attesa che un processo ha rispetto un altro processo.

- Ogni nodo è un processo.

- Gli archi indicano che un processo è in attesa che un altro processo rilasci la propria risorsa.
- Periodicamente viene aggiornato e chiamato l'algoritmo per la **ricerca di eventuali cicli di attesa**.
 - Tale algoritmo richiede un numero di operazioni dell'ordine di n^2 , dove n è il numero di vertici (processi).



Strategie di ripristino:

- si **uccidono tutti i processi** in uno stato di deadlock
- si esegue un **checkpoint** di uno stato precedente al deadlock e si fanno **ripartire i processi**
- si **uccide un processo alla volta** fino a quando il deadlock non esiste più
- si **prelazionano le risorse** ai processi bloccati fino a quando il deadlock non esiste più

Nel caso in cui la strategia prevede l'aborto di un processo in esecuzione, si possono utilizzare diverse metriche per decidere quale tra quelli interessati:

- minor tempo di CPU consumato fino a quel momento
- minor numero di linee di output prodotte finora
- maggior tempo stimato per la terminazione
- minor numero di risorse allocate finora
- minore priorità

Approach	Resource Allocation Policy	Different Schemes	Major Advantages	Major Disadvantages
Prevention	Conservative; undercommits resources	Requesting all resources at once	<ul style="list-style-type: none"> • Works well for processes that perform a single burst of activity • No preemption necessary 	<ul style="list-style-type: none"> • Inefficient • Delays process initiation • Future resource requirements must be known by processes
		Preemption	<ul style="list-style-type: none"> • Convenient when applied to resources whose state can be saved and restored easily 	<ul style="list-style-type: none"> • Preempts more often than necessary
		Resource ordering	<ul style="list-style-type: none"> • Feasible to enforce via compile-time checks • Needs no run-time computation since problem is solved in system design 	<ul style="list-style-type: none"> • Disallows incremental resource requests
Avoidance	Midway between that of detection and prevention	Manipulate to find at least one safe path	<ul style="list-style-type: none"> • No preemption necessary 	<ul style="list-style-type: none"> • Future resource requirements must be known by OS • Processes can be blocked for long periods
Detection	Very liberal; requested resources are granted where possible	Invoke periodically to test for deadlock	<ul style="list-style-type: none"> • Never delays process initiation • Facilitates online handling 	<ul style="list-style-type: none"> • Inherent preemption losses

Gestione della memoria

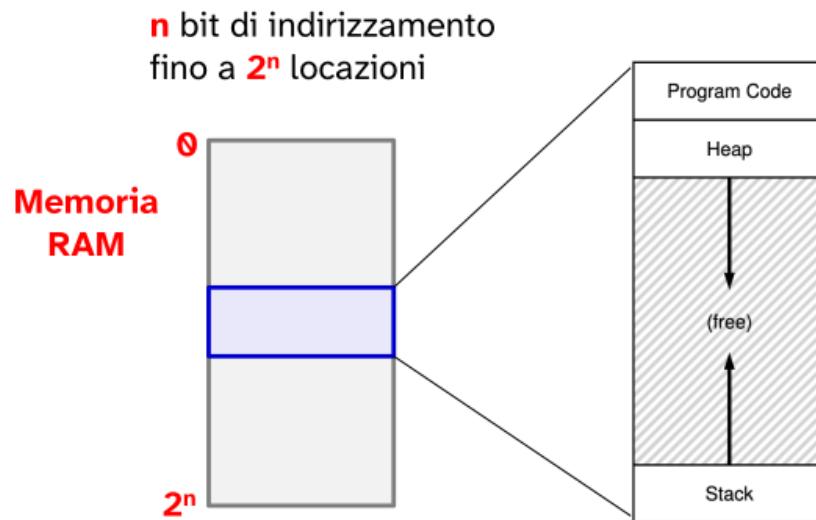
La **memoria principale** costituisce, insieme la CPU, una delle risorse per realizzare l'**astrazione di processo**.

Il processo dispone di una **area di memoria** ad esso **riservata** (non accessibile da altri processi). Tale area di memoria (*memoria virtuale*) illude il processo facendogli credere di avere a disposizione l'**intera memoria principale**.

Questa memoria riservata permette a ciascun processo di avere una **vista indipendente e continua** della memoria, anche se, fisicamente, la memoria principale è condivisa tra più processi e l'allocazione dell'immagine di questi non è detto che sia continua.

Il sistema operativo, come per il processore, attraverso l'utilizzo dei **PCB** garantisce il principio di **isolamento** tra processi, ovvero è necessario assicurarsi che l'esecuzione di uno non interferisca con quella di un altro.

Quindi fisicamente i processi non devono accedere ad aree di memoria fisiche in cui è contenuta l'immagine di un altro processo.



L'immagine del processo è descritta all'interno del PCB dello specifico processo.

In questa struttura sono contenute anche informazioni **su dove si trovano** le diverse parti dell'immagine **in memoria centrale**.

I processi non accedono mai direttamente alla memoria principale tramite indirizzi. Lavorano invece su una propria **memoria virtuale** (riservata), esso non utilizza indirizzi di memoria fisici ma bensì **virtuali**.

Tali **indirizzi virtuali** hanno senso solo nella visione della memoria del processo → non corrispondono a veri e propri indirizzi della memoria principale.

ESEMPIO:

Quando utilizziamo indirizzi all'interno di un codice non stiamo realmente utilizzando indirizzi fisici ma indirizzi virtuali.

Infatti quando stampiamo gli indirizzi di memoria delle variabili che utilizza un processo non stiamo realmente stampando indirizzi di memoria, ma indirizzi di memoria logici.

```
#include <stdio.h>
#include <stdlib.h>

int main() {
    int x = 3;
    printf("location of stack: %p\n", &x);
    char * p = malloc(1024);
    printf("location of heap : %p\n", p);
    printf("location of code : %p\n", main);
}
```

```
location of stack: 0x7fff691aea64
location of heap : 0x1096008c0
location of code : 0x1095afe50
```

Tali indirizzi virtuali che abbiamo stampato vengono tradotti da un componente hardware (MMU - Memory Management Unit) per eseguire le istruzioni desiderate.

Il kernel, a differenza di un Hypervisor, astrae l'hardware per consentire l'esecuzione di un processo in modo indipendente.

La posizione (**indirizzi**) di codice e dati nella memoria di un processo è un'**astrazione**. → indirizzi virtuali.

La posizione effettiva in memoria fisica è **gestita dal sistema operativo**.

Aspetti e parametri caratterizzanti la gestione della memoria

- **Supporto Hardware** per la gestione della memoria, in particolare l'isolamento tra i processi. (MMU)
- **Organizzazione logica** della memoria virtuale, ovvero le diverse sezioni che fanno parte della memoria virtuale di ogni processo.

- **Organizzazione fisica** della memoria principale, ovvero descrive come è allocata in memoria principale l'immagine di ogni singolo processo.

Tale organizzazione è necessaria per capire dall'indirizzo virtuale a quale indirizzo fisico si sta effettivamente accedendo.

- **Dimensione della memoria virtuale** che può essere più **grande della memoria fisica disponibile**.

Quindi il sistema operativo deve fare in modo di emulare la memoria fisica utilizzando la memoria secondaria, per permettere l'esecuzione dei processi che utilizzano più memoria di quanta ne sia disponibile.

- **Rilocazione:**

La rilocazione è un meccanismo che permette di caricare l'immagine di un processo in memoria principale.

Esistono due principali tipi di rilocazione:

- statica: gli indirizzi vengono definiti al momento della compilazione o caricamento del programma e non possono essere cambiati.
- dinamica: gli indirizzi possono essere modificati durante l'esecuzione del programma, ad esempio quando il programma viene riattivato e ritorna nella memoria principale dallo swap.

- **Organizzazione dello spazio virtuale**

- spazio virtuale unico: tutta la memoria di un processo è trattata come un'unica area.
- spazio virtuale segmentato: la memoria del processo è suddivisa in **segmenti** (ad esempio, codice, dati, stack) che vengono gestiti separatamente.

- **Allocazione**

Si riferisce alla maniera con cui il sistema operativo assegna la memoria principale ai processi.

- contigua: i dati di un processo sono collocati in un blocco contiguo di memoria.
- non contigua: i dati di un processo possono essere sparsi in diverse aree della memoria, come avviene nella **paginazione**.

- **Caricamento**

Il caricamento riguarda come i processi vengono caricati in memoria:

- tutto insieme: il processo viene caricato tutto in una volta nella memoria.
- a domanda: solo le parti del processo necessarie vengono caricate in memoria, quando l'esecuzione ne richiede l'accesso.

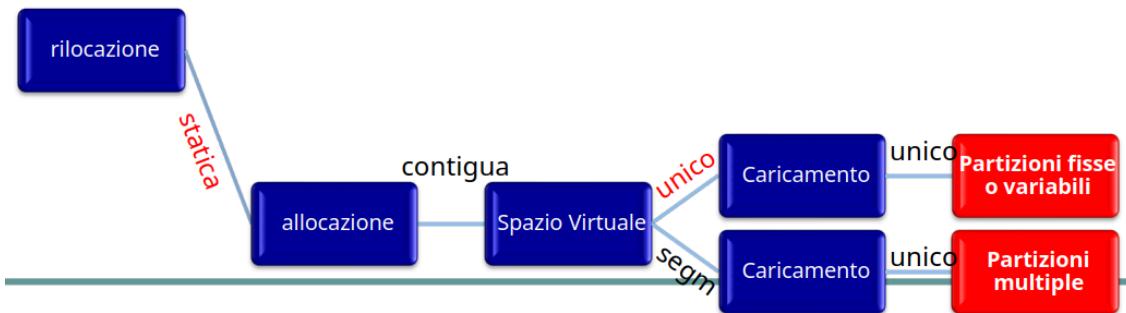
Tecniche di gestione della memoria

Rilocazione significa allocare l'immagine del processo nella memoria principale, può avvenire in diversi modi.

Questa rilocazione può essere statica o dinamica.

Nei primi sistemi, il posizionamento del codice e dati è **fisso**. → con assi abbimo inserito in posizioni fisse il codice e i dati.

Il processo è caricato in RAM tutto insieme come un **unico blocco** (allocazione contigua).



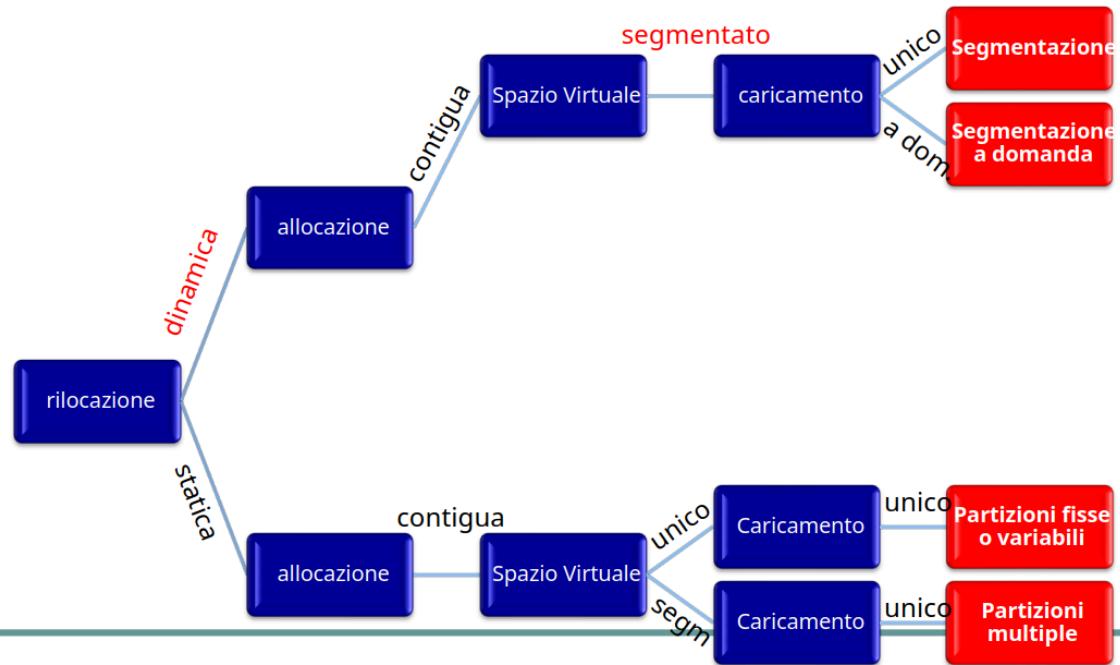
Statica perché era più semplice.

Con l'introduzione di hw più evoluti si è permesso di configurare nella CPU il posizionamento di codice e dati, a **tempo di esecuzione**.

→ ad esempio sfruttando una base e un offset per determinare l'indirizzo fisico.

Con il supporto dei compilatori, la memoria del processo è divisa in blocchi (segmenti) che possono essere **gestiti in maniera separata e indipendente dal SO**.

Ad esempio il SO protegge il segmento dedicato al codice dando il permesso di sola lettura all'area di memoria a cui sarà destinato.



Ulteriori evoluzioni hw hanno permesso di **caricare** l'immagine del processo (o un suo segmento) **non più per intero**, ma in **piccole porzioni (pagine)** (caricamento a domanda) **sparpagliate** in memoria fisica (allocazione non contigua).

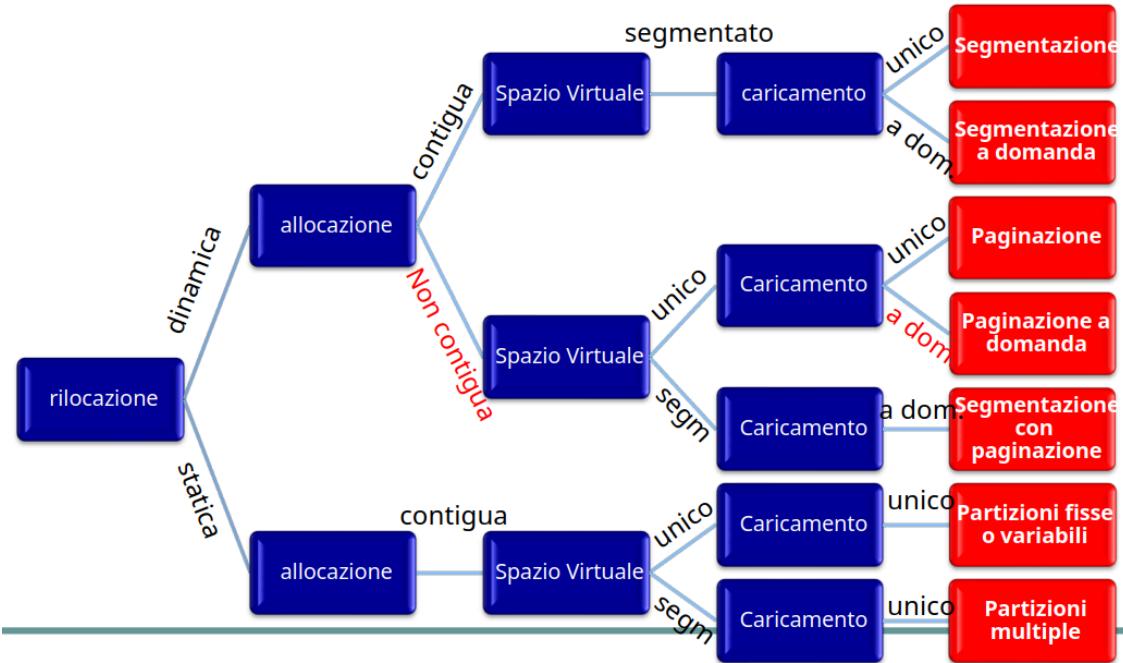
Quindi è stata introdotta l'allocazione di tipo **non contigua** grazie allo sviluppo dell'hardware MMU.

Che vantaggio mi da l'allocazione di tipo non contigua?

→ uso efficiente dell'area di memoria in termini di **flessibilità e dinamicità**; i processi non devono essere necessariamente allocati in un blocco contiguo di memoria.

→ attenua il **problema della frammentazione**, perché non ho bisogno di trovare una area di memoria di dimensione tale da contenere interamente l'immagine del processo. Ci sono due tipi principali di frammentazione:

- frammentazione interna: si verifica quando la memoria allocata per il processo è **più grande di quella effettivamente necessaria**.
- frammentazione esterna: si verifica quando ci sono **molti piccoli spazi liberi sparsi** nella memoria fisica, ma **nessuno** di questi è **abbastanza grande da ospitare un nuovo processo**.



- **Partizioni fisse:** memoria suddivisa in blocchi di **dimensione fissa**. Facile da gestire ma potrebbe causare **frammentazione interna**.
- **Partizioni variabili:** la memoria virtuale viene allocata in blocchi di **dimensioni variabili** a seconda delle necessità del processo. Riduce la frammentazione interna ma può comunque portare a **frammentazione esterna**.
- **Partizioni multiple:** una combinazione di partizioni fisse e variabili.
- **Segmentazione a domanda:** caricamento dinamico di segmenti di processi solo quando necessari. Ottimizza l'uso della memoria, ma può causare ritardi nei processi se la memoria virtuale deve essere frequentemente caricata dal disco.

Rilocazione

Modalità di allocazione dei dati e del codice di un processo in memoria fisica.

L'associazione di **istruzioni** e **dati** ad indirizzi di memoria (fisica) si può compiere in ognuna delle sequenti fasi:

- **compilazione:** se nella fase di compilazione si conosce dove il processo risiederà nella memoria, allora può inserire direttamente gli **indirizzi fisici nel codice oggetto**.
 - si genera così **codice assoluto**: gli indirizzi nel codice oggetto sono indirizzi fisici reali della RAM. Quindi codice e dati saranno allocati sempre in tali posizioni ad ogni esecuzione.
 - potrebbe essere un problema se il SO decidesse di caricare il programma in un'altra zona della memoria. Sarebbe necessaria una ricompilazione perché tutti gli indirizzi sarebbero **sbagliati**.

Quindi in questo caso gli indirizzi sarebbero fissati per sempre (fino ad una successiva ricompilazione)

→ nessuna **flessibilità**;

- **caricamento**: se nella fase di compilazione non è possibile sapere in che punto della memoria risiederà il processo, il compilatore deve generare **codice rilocabile**.

Quindi il **compilatore non inserisce** nel codice oggetto **indirizzi fisici** ma dei valori relativi che il loader dovrà sistemare.

Quando il processo viene caricato in memoria, il **loader sceglie un indirizzo di partenza** e scorre il codice oggetto aggiustando tutti gli altri indirizzi relativi.

→ dopo il caricamento, lo spazio di indirizzamento del processo non può più esser spostato, altrimenti andrebbero aggiornati tutti gli indirizzi fisici;

- **esecuzione**: se durante l'esecuzione il processo può essere spostato da un segmento di memoria ad un altro, è necessario che si ritardi l'associazione degli indirizzi fino alla fase di esecuzione. (rilocazione dinamica)

→ Qui entriamo in merito della **memoria virtuale** che permette al SO di:

- spostare un processo in RAM;
- fare swapping su disco;
- caricare pezzi di processo dove capita (dove si ha disponibilità) → utilizzo efficiente della memoria.

Per permettere ciò **non conviene** fissare gli indirizzi fisici **né a compilazione, né al caricamento**.

Allora cosa accade:

- nel codice non vengono inseriti indirizzi fisici ma **logici**;
- ogni volta che la CPU fa un accesso a memoria, interviene la **MMU** che traduce gli indirizzi logici nei corrispettivi indirizzi fisici a run-time.

Se il processo viene spostato, basta cambiare il **modo** con cui la MMU traduce gli indirizzi.

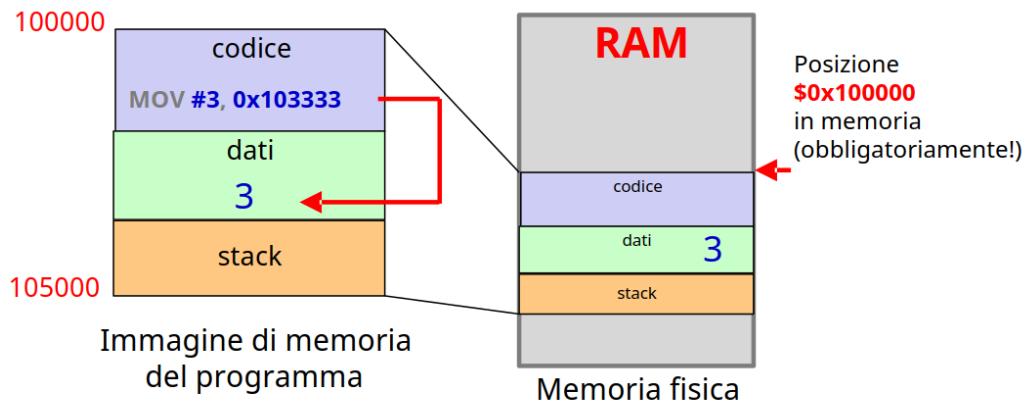
Quindi per realizzare questo schema sono necessarie specifiche caratteristiche dell'architettura → **MMU**

statica

La rilocazione statica stabilisce gli indirizzi di codice e dati **al momento della compilazione o caricamento**.

La posizione del codice e dei dati **non può più essere modificata** a tempo di esecuzione.

→ nella programmazione in assembly si decide a priori dove il codice viene posizionato in memoria centrale.

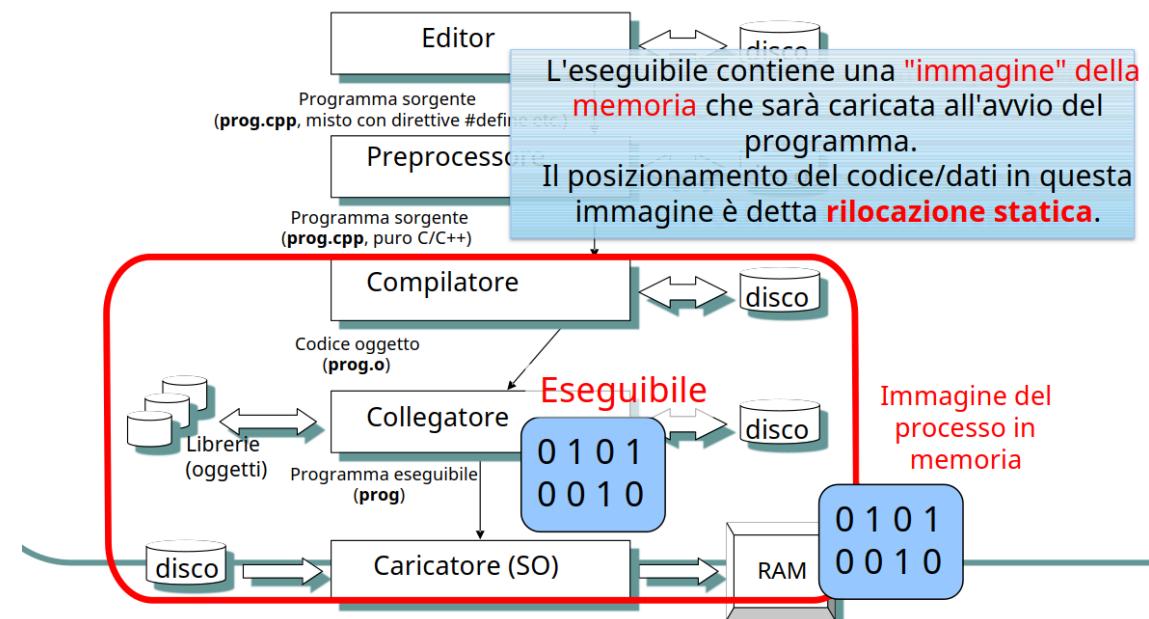


Il problema di questo tipo di rilocazione si presenta nel **context switch**: il processo quando viene prelazionato e inserito nella coda dei processi pronti oppure swappato

→ nel momento della sua successiva esecuzione deve trovarsi sempre nella stessa area di memoria, altrimenti tutti gli indirizzi fisici su cui lavora sarebbero sbagliati.

Questo è un grande **onere computazionale** affidato al sistema operativo.

Quindi con una rilocazione statica è il compilatore o il caricatore ad inserire degli indirizzi fisici all'interno rispettivamente del codice oggetto o del codice eseguibile.



Questi indirizzi non potranno esser cambiato al meno che il codice non venga ricompilato oppure ricaricato.

dinamica

Con questo approccio, dinamico, l'indirizzo di codice/dati nella immagine (*virtuale*) **non corrisponde** alla loro posizione effettiva in RAM (*indirizzo fisico*).

L'effettiva posizione dello spazio di indirizzamento del processo è **scelta da SO**, al caricamento iniziale del processo, o anche durante la sua esecuzione.

→ richiede un supporto hardware che permetta a tempo di esecuzione di tradurre gli indirizzi virtuali in indirizzi fisici.

Infatti la rilocazione di tipo dinamico si introduce un distinzione fondamentale:

- **Indirizzo virtuale:**

indirizzo **acceduto dal programma** durante l'esecuzione.

- **Indirizzo fisico:**

indirizzo visto dall'unità di memoria, **posizione effettiva** del dato/istruzione.

Quando otteniamo **segmentation fault** significa che l'indirizzo virtuale che stiamo utilizzando, erroneamente, non è associato ad un indirizzo fisico facente parte dell'immagine del processo nella memoria fisica → l'indirizzo fisico che stiamo deferenziando non è mappato nella memoria virtuale.

→ la traduzione da indirizzo virtuale a indirizzo fisico ha dato questo problema.

Vedremo che il fault può essere **ripristinabile**, perché il segmento/pagina non sono in memoria fisica ma sono swappati in memoria di massa. → ci troviamo nella situazione di un caricamento a domanda.

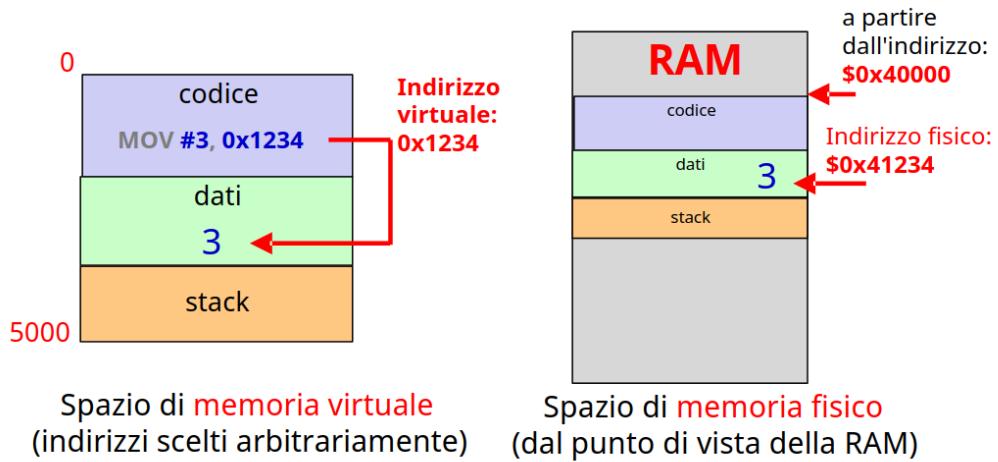
Il processore vede solo indirizzi logici (virtuali)

→ **tradotti dall'hardware** che ha la visibilità della memoria fisica, ovvero MMU.

MMU rende trasparente l'accesso del processore agli indirizzi fisici.

Se non fosse così il processore **sarebbe più performante** (vedrebbe direttamente gli indirizzi fisici), ma perderei tutta la parte di virtualizzazione.

Infatti come abbiamo visto, aprendo il file eseguibile od oggetto, non sono presenti indirizzi fisici ma unicamente indirizzi virtuali che solitamente partono tutti da **0x0**.



Vantaggi della rilocazione dinamica

La rilocazione dinamica consente lo **swapping**

- un processo è temporaneamente **sospeso** e trasferito in **memoria secondaria**;
- il processo potrà poi essere ri-caricato in un'**area di memoria differente**, in base alla situazione attuale della memoria.

MMU (Memory Management Unit)

MMU è un componente hardware della CPU che ha il compito di rendere **trasparente l'accesso al processore** alla memoria fisica tramite una traduzione di **indirizzi virtuali** (utilizzati dal processore) in **indirizzi fisici**.

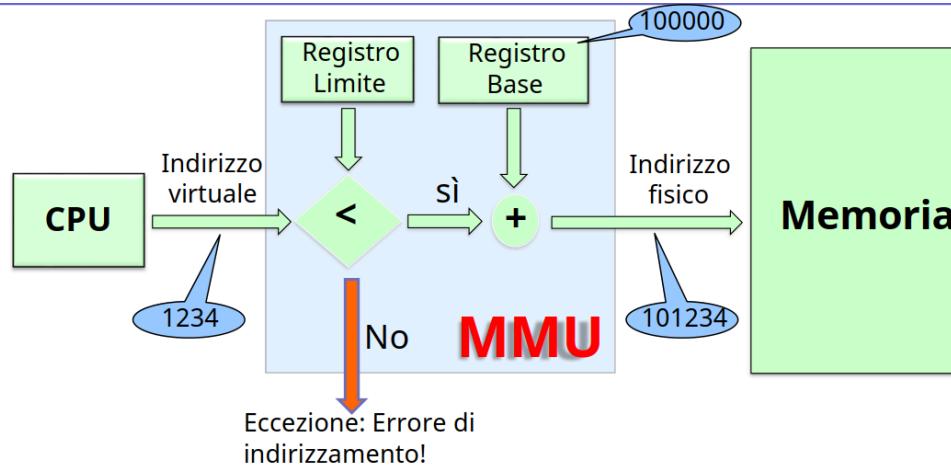
ESEMPIO DI RILOCAZIONE DINAMICA:

Caso basilare (obsoleto):

- spazio virtuale unico → viene trattato come un'unica area.
- allocazione contigua → viene allocato in memoria tutto insieme.

MMU in questo caso possiamo rappresentarla come un componente hardware avente due registri speciali: **limite** e **base**.

In questo caso banale la traduzione avviene con la somma tra l'indirizzo virtuale + la base, ovviamente a valle di una verifica sull'indirizzo virtuale (se esiste un indirizzo fisico mappato con tale indirizzo virtuale).



Ovviamente dobbiamo prevedere un modo per modificare i registri dell'MMU, che sono specifici per ogni processo.

Oltre a cambiare valore per ogni processo i due registri devono esser modificati anche ogni volta che un processo passa in esecuzione da che era swappato

→ con la rilocazione dinamica potrebbe ritrovarsi in una posizione della memoria fisica differente alla precedente

Questo modello infatti non è la realtà, è uno schema di funzionamento base dell'MMU.

Caricamento unico e a domanda

Il caricamento in memoria dell'immagine del processo è fatta dal **loader** che è parte del SO che:

- **legge** l'eseguibile (e.g. ELF);
- **alloca** memoria per il processo;
- **mappa** gli indirizzi virtuali negli indirizzi fisici (a seconda della rilocazione utilizzata: statica o dinamica);
- **copia** in RAM le parti necessarie del programma (**a seconda della tipologia di caricamento**);
- **prepara** il processo per la prima esecuzione → ad esempio, crea il PCB.

Nella fase di copia il loader si può comportare in due modi a seconda della tipologia di caricamento implementata:

- Nel **caricamento unico** (tutto insieme)
 - Il loader carica tutta l'immagine del programma in memoria RAM.
- Questo è un approccio tipico dei vecchi sistemi o microcontrollori.
- Nel **caricamento a domanda** (demand loading)
 - Il loader crea solo le mappature virtuali → fisiche.

- Il caricamento effettivo del codice/dati avviene **solo quando necessario**.
- La MMU genera un fault quando la CPU tenta di accedere a una parte del processo che **NON è ancora stata caricata in memoria fisica**.

A questa fault generata (**interruzione sincrona**), il SO chiama un handler che fa una recovery. Ovvero verifica se tale parte di processo a cui vuole accedere la CPU è presente in memoria secondaria e la carica, permettendo al processore di proseguire con l'esecuzione.

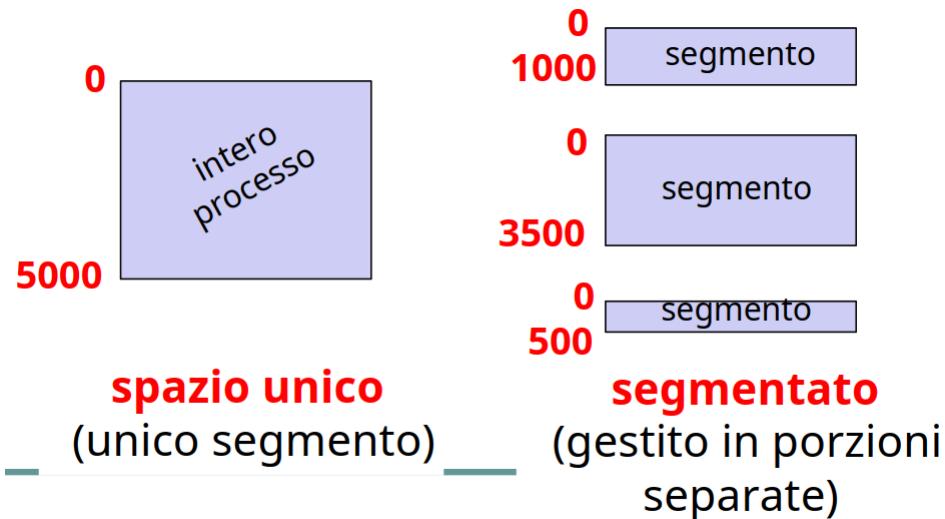
Se tale parte non viene trovata viene rilanciata ancora la fault che in questo caso genera la terminazione del processo → ha tentato di accedere ad un indirizzo di memoria che non fa parte del proprio spazio di indirizzamento.

NOTA: fault è un tipo di interruzione sincrona perché viene chiamata nel momento in cui nella tabella delle pagine o segmenti non è stata mappata la pagina virtuale in una fisica. **NON è asincrona**.

Gestione dello spazio virtuale

Vi sono due possibili approcci per gestire lo **spazio virtuale** degli indirizzi.

- Uno **spazio unico**
(corrispondente all'intero processo)
- Un insieme di **segmenti**
(**segmentazione** → la memoria del processo è gestita in porzioni separate)



Nell'approccio segmentato l'immagine del processo è suddivisa in **porzioni logiche** (segmenti) gestite in modo separato ed indipendente dal sistema operativo.

Ogni segmento può avere una dimensione variabile, in base alla struttura logica del programma (codice, heap, stack, dati).

Vantaggio principale:

La segmentazione facilita la condivisione di aree di memoria fisica tra più processi, mappandole in segmenti distinti del loro spazio di indirizzamento.

- → Esempio:

il segmento di codice (text) di un programma è di sola lettura.

Più processi che eseguono lo stesso programma possono **condividere la stessa copia fisica** del codice, mappandola nel proprio spazio di indirizzamento virtuale all'interno del segmento dedicato all'area testo.

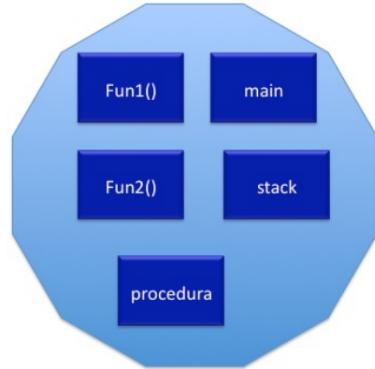
segmentazione

A **tempo di compilazione**, si configura lo spazio virtuale segmentato.

Viene creato un diverso **segmento** per ciascun **modulo** del programma.

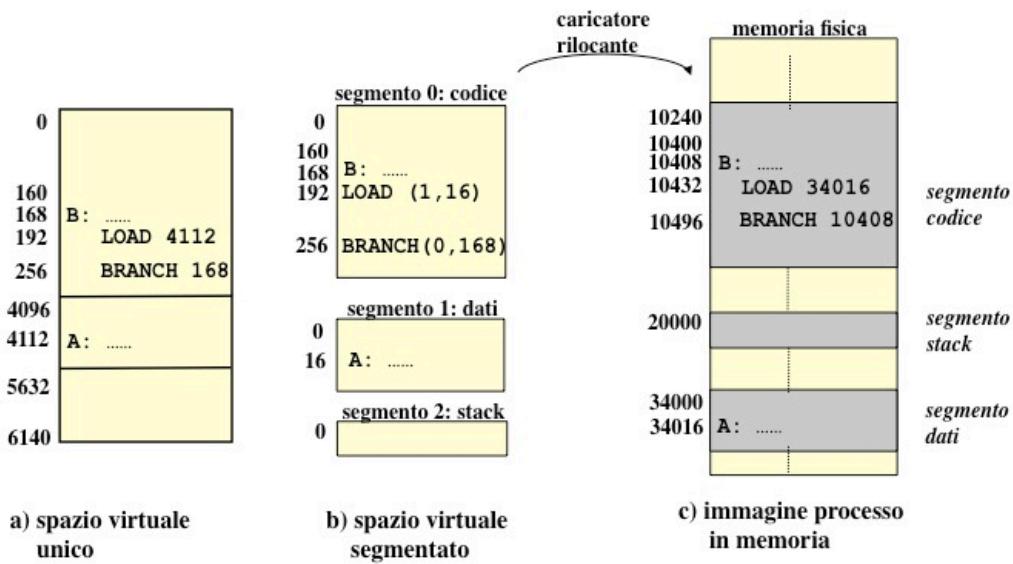
vantaggi della segmentazione

- **Protezione** dei segmenti;
permette una **granularità fine** nella gestione dei permessi sullo spazio di indirizzamento.
Ad ogni segmento può avere **permessi diversi** (lettura, scrittura, esecuzione).
- **Condivisione** dei segmenti;
segmenti come **codice** (text) o le **librerie condivise** possono essere mappate da più processi e condividere un'unica copia fisica, **riducendo l'uso della memoria**.
- **Allocazione indipendente** dei segmenti in memoria fisica → riduce (ma non elimina) il problema della frammentazione.
Ogni segmento può essere **collocato separatamente in punti diversi** della memoria fisica.



Un indirizzo virtuale è una coppia (contiene due informazioni), fatta da:

- un **identificativo** del segmento;
- uno **scostamento** (offset) all'interno del segmento.

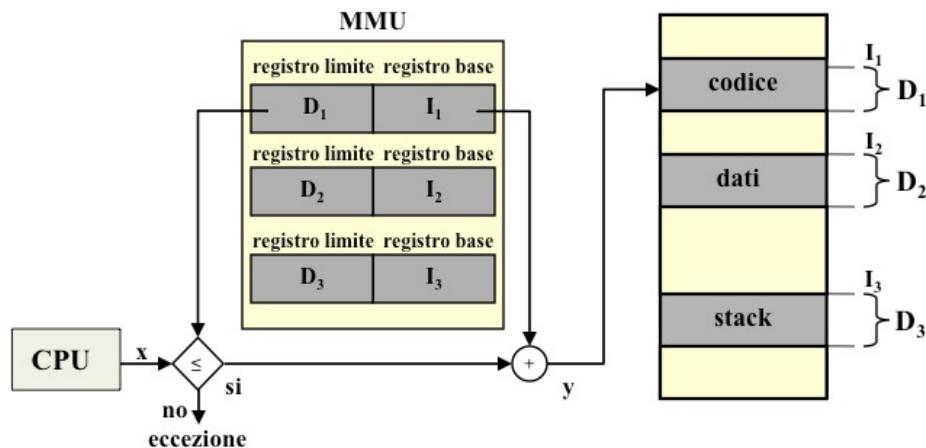


L'MMU è il componente su cui si basa anche la segmentazione.

- Traduce gli indirizzi virtuali dalla forma (*segmento, offset*) in indirizzi fisici.
- Nel caso di **pochi segmenti**, è sufficiente avere nella MMU più coppie di registri base/limite: uno per ogni segmento.

Questo è un **limite fisico** importante perché non mi permetterebbe di avere molti segmenti. → non posso avere un numero infinito di registri.

ESEMPIO SEGMENTAZIONE CASO SEMPLICE:



- in questo caso ogni coppia di registri corrisponde a limite e base per un segmento allocato in memoria fisica
- l'identificativo **sg** servirà per capire a quale tipologia di segmento accedere facendo riferimento alla giusta coppia D, I

- l'offset (`off`) sarà sommato al corretto registro base I se \leq del registro limite D corrispondente ad un segmento, altrimenti la MMU solleverà un'eccezione (segmentation fault).

Nel caso **generale**, quando si vuole supportare un **numero arbitrario di segmenti**, non è possibile avere una coppia **base/limite** nella MMU per ogni segmento di un processo.

Per questo motivo:

- le coppie base/limite non risiedono nella MMU, ma sono memorizzate in **memoria RAM**;
- queste informazioni sono raccolte in una struttura dedicata, chiamata **tabella dei segmenti (segment table)** unica per ogni processo;
- ogni **entry** della tabella dei segmenti contiene i dati relativi a un **segmento del processo**: **indirizzo base, limite e bit di controllo** (permessi rwx).

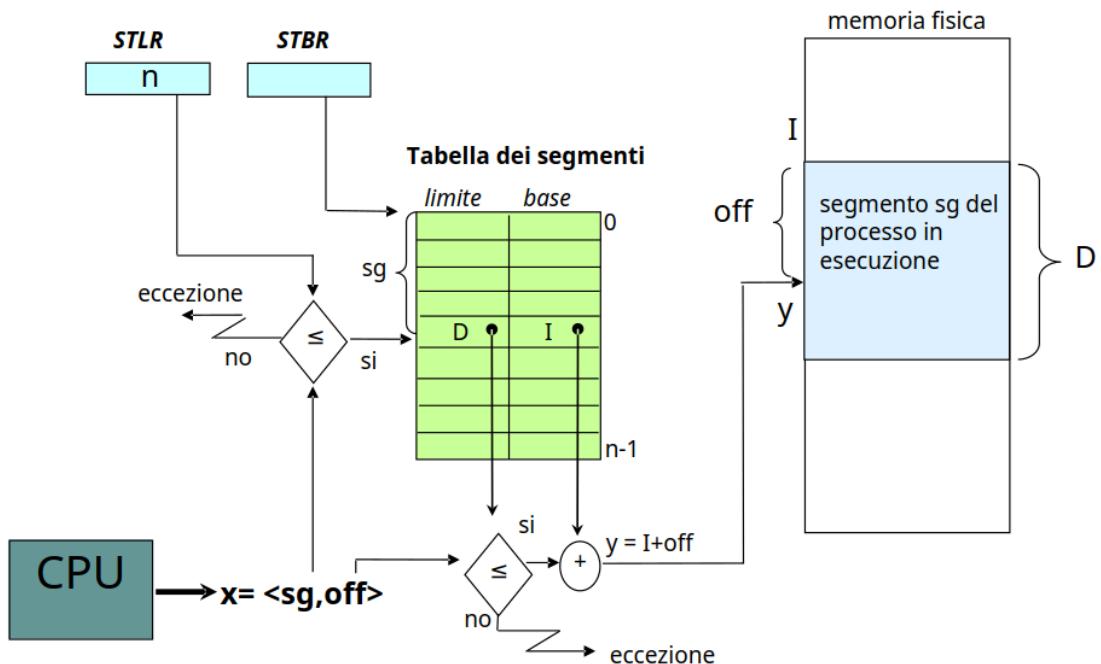
La MMU gestisce la segment table con due appositi **registri**:

- **STBR** (Segment Table Base Register): indirizzo in memoria fisica in cui si trova la tabella dei segmenti.
- **STLR** (Segment Table Limit Register): dimensione della tabella dei segmenti (indica il **numero di segmenti del processo**)

Il SO, all'atto del caricamento in memoria del processo da eseguire, imposterà l'**indirizzo fisico** dell'*entry point* della *segment table* nel registro **STBR**.

Tale indirizzo essendo strettamente legato al processo, è contenuto all'interno del PCB.

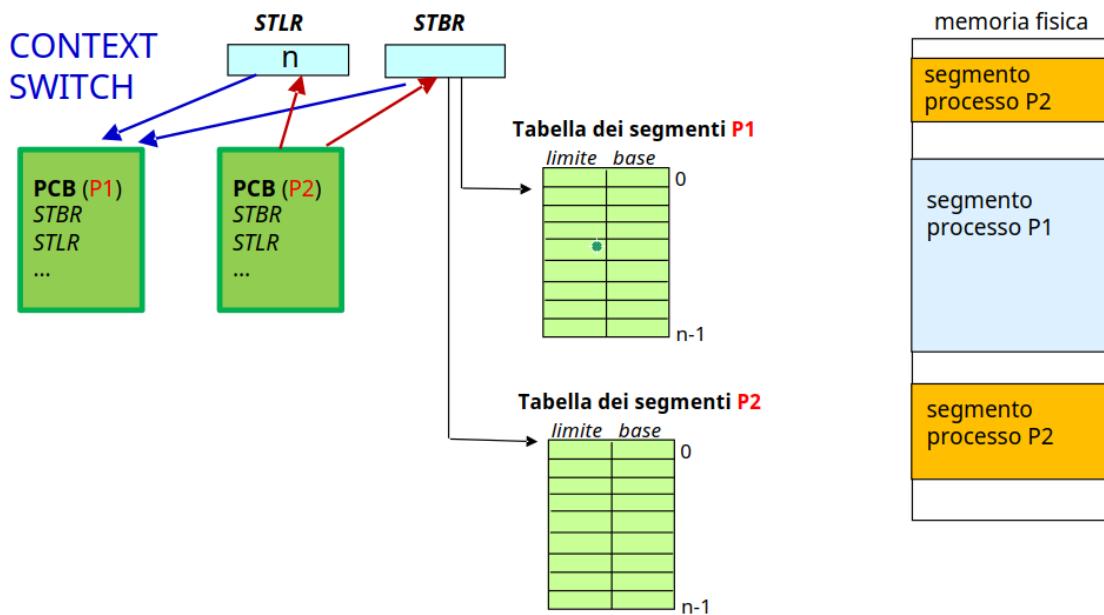
ESEMPIO DI TRADUZIONE UTILIZZANDO LA SEGMENT TABLE:



- **sg** è l'offset per identificare l'entry point relativo ad un segmento.
- Per accedere alla tabella il processore utilizza le informazioni contenute in **STBR** e **STLR** i cui valori sono contenuti all'interno del PCB di ogni processo.

NOTA sulla segment table:

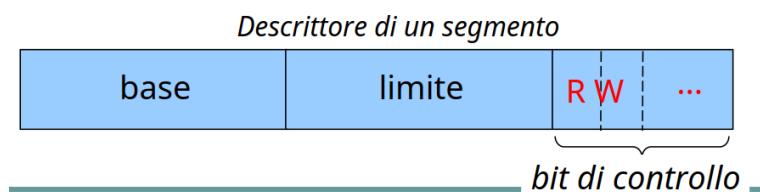
- ogni processo ha una **segment table differente**
- i registri STBR/STLR sono configurati ad ogni **context switch** dei processi
→ durante il context switch il SO carica i valori di STBR/STLR dal **PCB del processo** (sono legati al singolo processo).



protezione

Ogni segmento può avere diversi **permessi di accesso** specificati da bit di controllo contenuti nella segment table.

- Ogni riga della segment table di un processo contiene per ogni entry anche una sequenza di **bit di controllo** per gestire i permessi sull'area di memoria in cui è mappato il segmento.
- MMU produce una **exception** se il programma non rispetta i permessi.



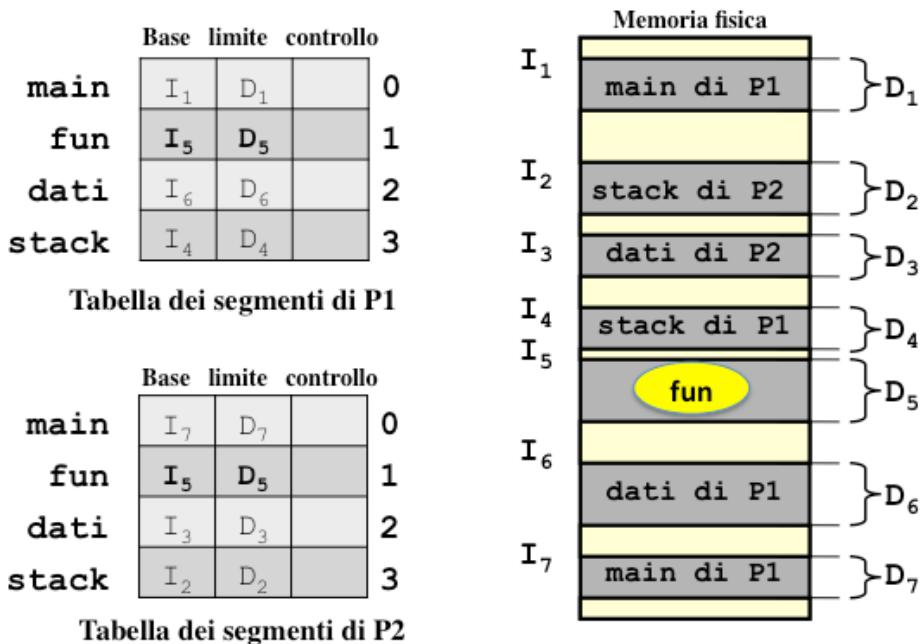
condivisione

La segmentazione consente la **condivisione dei segmenti** tra più processi, allocando in memoria fisica una sola copia del segmento.

Abbiamo due processi con le rispettive tabelle dei segmenti.

→ Se hanno una entry in comune significa che i due processi possono accedere alla stessa area di memoria fisica.

Quindi stanno effettivamente condividendo la copia del segmento di memoria. Questo permette di allocare una sola copia del segmento in memoria.



allocazione

Uno spazio/segmento di memoria virtuale può essere **collocato in memoria fisica** in due possibili modi:

- **allocazione contigua:**

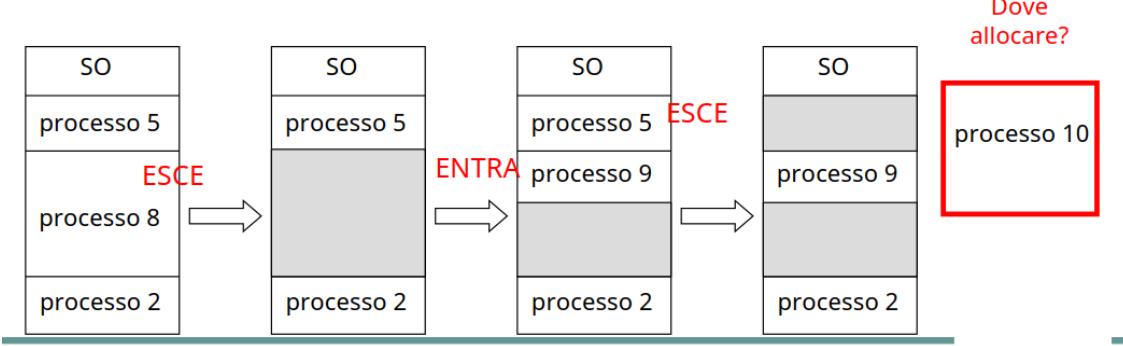
lo spazio/segmento è **copiato per intero**, in un intervallo di memoria fisica agli indirizzi [D;D+I]

- **allocazione non contigua (paginazione)**

allocazione contigua

- Il SO colloca il proprio blocco di memoria virtuale, e quelli dei processi, in intervalli **non-sovrapposti** della memoria fisica
- Quando un processo termina, la memoria fisica occupata si libera, creando un **hole**

- Quando un nuovo processo viene caricato, occorre **cercare un hole sufficientemente grande** da contenerlo
→ compito dello scheduler a medio termine.



Questo è un esempio su come facilmente si può arrivare ad un problema di **frammentazione esterna** della memoria.

Se ci sono **più buchi liberi**, ci sono vari criteri per scegliere dove collocare un segmento:

- **first-fit**: si assegna il **primo** hole sufficientemente grande;
- **best-fit**: si assegna lo hole **più piccolo** tra quelli sufficientemente grandi per contenere lo spazio di indirizzamento del processo;
- **worst fit**: si assegna lo hole più grande.

In generale, gli **schemi a partizione di dimensione variabile** soffrono del problema della frammentazione esterna.

Frammentazione esterna: spazio di memoria perduto sotto forma di spezzoni.

- Lo spazio di memoria totale sarebbe sufficiente per soddisfare una richiesta, **ma non è contiguo**.
- → non si sfrutta a pieno la quantità di memoria totale a disposizione.

Dualmente gli schemi a partizione di dimensione fissa soffrono del problema della frammentazione interna.

Frammentazione interna: è un concetto relativo al singolo segmento, in particolare è legato alla dimensione di questo.

Se i segmenti hanno una dimensione fissa, non è detto che l'immagine di un processo sia un multiplo di questa dimensione. Quindi tale immagine potrebbe occupare un numero di segmenti, ma non tutte le word di questi segmenti avranno un significato.

Ovvero siamo sprecando una porzione dell'ultimo segmento in cui è contenuta l'immagine del processo.



Paginazione

La segmentazione oggi non viene più utilizzata.

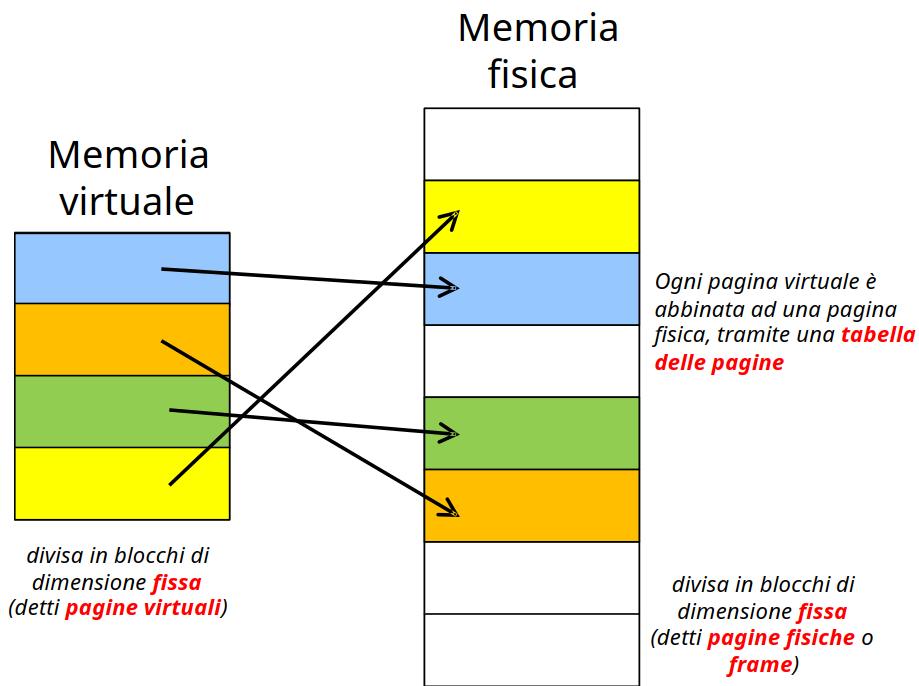
Invece viene utilizzata la paginazione perché permette di eliminare un enorme problema: la frammentazione esterna.

Ovviamente non è perfetto come approccio infatti anche questo introduce un problema, quello di frammentazione interna. → perdo mediamente per ogni pagina una quantità di memoria pari alla metà della dimensione della pagina stessa.

Paginazione:

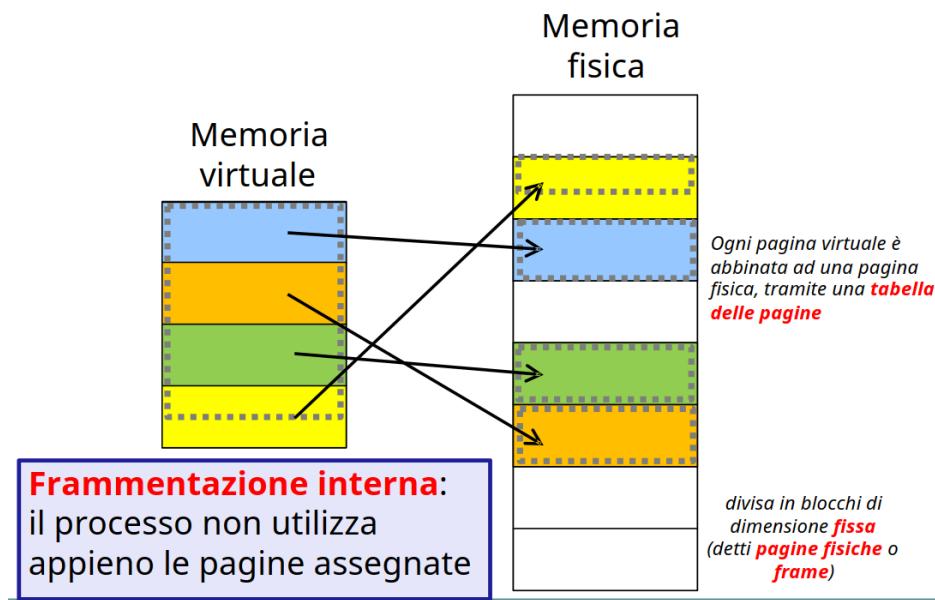
- tecnica di allocazione **non contigua** → se fosse contigua la paginazione non avrebbe senso;
- lo spazio virtuale è diviso in **blocchi di dimensione fissa**;
- evita la frammentazione esterna;
- introduce la frammentazione interna.

A causa del fatto che lo spazio virtuale è suddiviso e quindi caricato in memoria fisica all'interno di blocchi di dimensione **fissa**.



- La **tavella delle pagine** è salvata in RAM e come per la segmentazione l'*entry point* è memorizzato nel PCB di ogni processo.
- Ogni processo ha la propria tabella delle pagine.

Come detto questo approccio è soggetto alla frammentazione interna.



- La pagina associata all'ultima pagina della memoria virtuale non è completamente utilizzata.
→ spreco della memoria, perché non è possibile più utilizzarla fino a quando non viene deallocata.

Cosa accade nel momento in cui si verifica una frammentazione interna:

- spazio di memoria perso per un blocco assegnato ma non utilizzato a pieno
- si verifica se la dimensione del processo non è un multiplo esatto della dimensione dei blocchi

Questo fenomeno è tanto **più trascurabile** quanto **più piccola è assegnata la dimensione** di ogni pagina.

Tipicamente, la **dimensione di pagina** è una potenza di 2, compresa tra 512 byte e 16 MB.

Traduzione degli indirizzi

Essendo l'allocazione non contigua è necessario memorizzare ogni pagina virtuale in che posizione della memoria fisica si trova.

Nel codice, quindi, si utilizzano indirizzi virtuali che devono essere **tradotti dall'MMU** in indirizzi fisici a *run-time*.

Questo hardware è necessario perché nell'approccio di allocazione **non contigua non si conosce a priori dove sarà rilocato il codice**.

→ dobbiamo sempre tener conto che se parliamo di paginazione allora stiamo parlando di allocazione non contigua, altrimenti la paginazione non avrebbe senso.

Quindi quando l'architettura utilizza l'approccio di paginazione, tutti i programmi utilizzano indirizzi virtuali.

La paginazione, inoltre, non è solo una tecnica di allocazione non contigua, ma anche una tecnica per la **gestione della memoria virtuale**. In altre parole, la paginazione permette di creare uno spazio virtuale più grande della memoria fisica, se la gestione del caricamento delle pagine è della tipologia: **a domanda**.

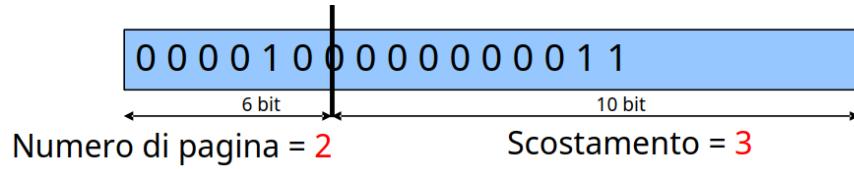
Un indirizzo virtuale contiene la **coppia**:

- **numero di pagina (p)**:
identifica una pagina nella memoria fisica → nel contesto di un processo.
- **scostamento di pagina(d)**:
indica la posizione dell'indirizzo all'interno della pagina.

A differenza della segmentazione, non sono due valori separati, ma sono contenuti entrambi in un **unico valore**.

→ Nel contesto della CPU, ovvero questa vede un unico indirizzo le cui informazioni non le tratta separatamente.

ESEMPIO: indirizzo virtuale a **16bit**: **0x0803**

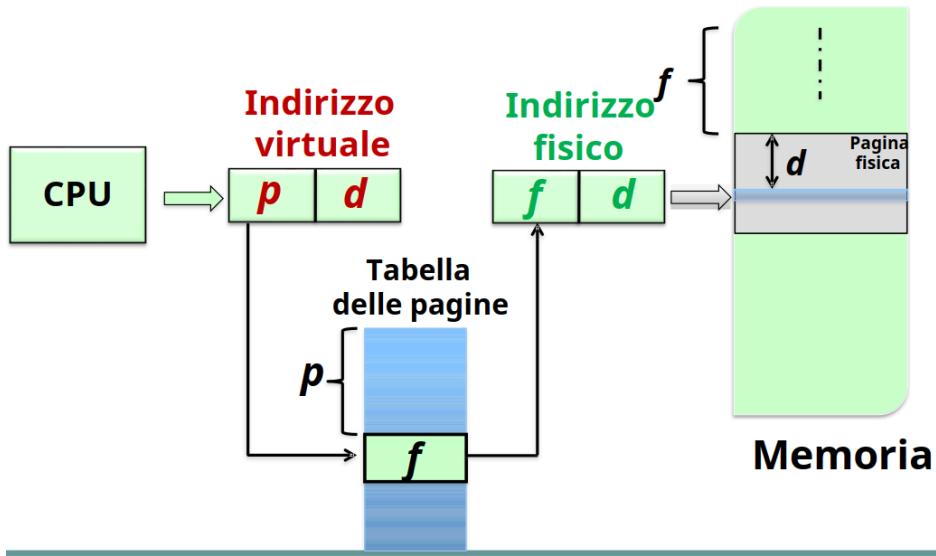


Dimensione delle pagine: $2^{10} = 1024 = 1\text{ kB}$

Numero massimo di pagine: $2^6 = 64$ pagine

Quindi l'indirizzo viene diviso in due campi ognuno dei quali porta con se un'informazione.

Vediamo come avviene la traduzione con l'utilizzo dell'MMU sulla *page table*.



L'MMU valuta in che posizione si trova l'indirizzo base della pagina identificata da *p* e utilizza tale indirizzo *f* sommato all'offset *d* per ottenere la traduzione in indirizzo fisico.

Ovviamente saranno presenti condizioni che bloccano i casi in cui si eccede dalla tabella delle pagine con *p*.

Tabella delle pagine

La tabella delle pagine ha una riga per **ogni pagina virtuale del processo**.

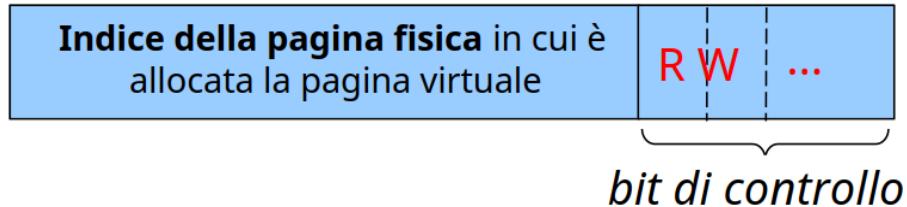
All'interno di questa riga sono contenuti:

- indice della **pagina fisica**,
- **bit di gestione** (permessi di accesso, etc.).

Ovviamente quando si tenta di operare su una pagina l'MMU verifica che il programma non violi i permessi presenti su tale.

→ in caso di violazione solleva un **page fault**.

Elemento della tabella delle pagine

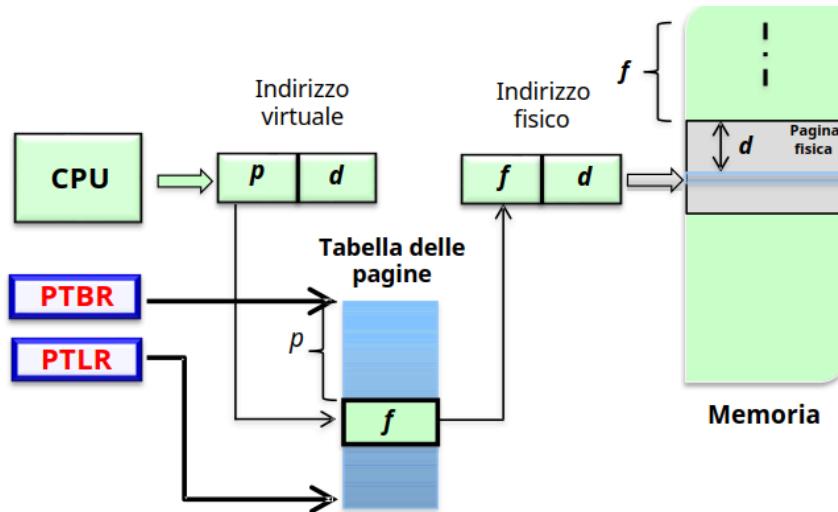


La tabella delle pagine è in **memoria principale**.

La MMU usa 2 registri utilizzarla:

- **PTBR** (Page-Table Base Register): indirizzo fisico della tabella delle pagine in memoria fisica.
- **PTLR** (Page-Table Length Register): dimensione della tabella delle pagine.

Questi due valori sono contenuti all'interno del PCB di ogni processo e vengono caricati in tali registri ogni volta che avviene un **context switch**.



architettura di paginazione con TLB

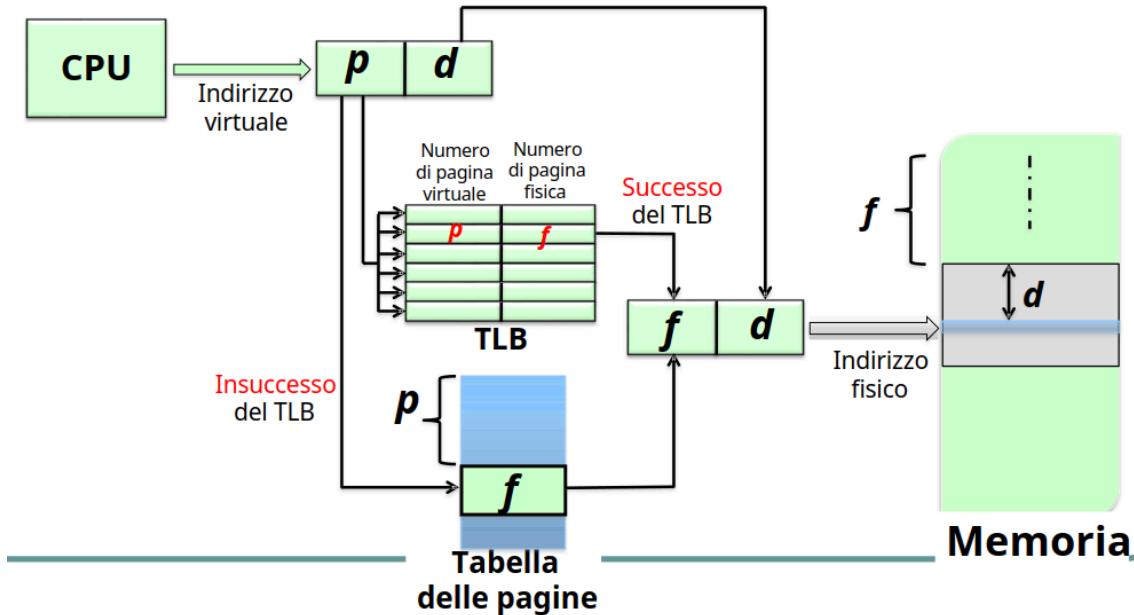
Per accedere ad ogni singolo dato nella memoria quindi servono **due accessi**.

1. per leggere la tabella delle pagine
2. per accedere al dato/istruzione vero e proprio

Questo provoca un **rallentamento** degli accessi a memoria.

Per **migliorare l'efficienza**, si usa una **cache associativa** detta **TLB** (Translation Look-aside Buffer) che si trova all'interno dell'MMU.

La ricerca di un valore in tale cache associativa ha complessità $O(1) \rightarrow$ costante.



- Si possono verificare due situazioni:

- L'accesso alla TLB produce un cache hit, quindi subito ho l'entry point della pagina in memoria fisica, tale operazione richiede nanosecondi.
- L'accesso alla TLB produce un cache miss, quindi la ricerca passa sulla tabella delle pagine, tale operazione richiede decine di nanosecondi.

In ogni caso dovrò fare più di un accesso per ottenere il dato/istruzione nella memoria fisica.

Tempo effettivo di accesso

Tasso di successo (hit ratio, α): percentuale di volte che un numero di pagina virtuale si trova nel TLB.

Supponiamo che:

- Lookup associativo = ε unità di tempo
- Un accesso alla memoria = κ unità di tempo

Allora il **tempo effettivo d'accesso** (*effective access time*):

$$EAT = (\kappa + \varepsilon)\alpha + (2\kappa + \varepsilon)(1 - \alpha) = (2 - \alpha)\kappa + \varepsilon$$

Dove la prima parte indica il tempo in caso di successo mentre la seconda indica il tempo in caso di insuccesso.

Questo *EAT* è il **tempo medio che un sistema impiega per accedere alla memoria**, tendendo conto sia degli **hit cache** che dei **miss cache**.

(supponendo che la pagina sia effettivamente presente in memoria fisica, ovvero che non debba esser caricata dalla memoria di massa)

Dimensione della tabella delle pagine

La scelta della dimensione della pagina influenza molto l'efficienza, per questo è necessario che si trovi un gusto compromesso per sistemi general purpose.

Supponendo di avere un sistema che valuta indirizzi a **32bit**. → Lo spazio totale di indirizzamento: $2^{32} = 4\text{GB}$.

Usando pagine di **1KB** (2^{10}):

- **dimensione** della tabella: $2^{22} = 4\text{MB} \rightarrow !$
- **frammentazione interna** media: $\frac{2^{10}}{2} = 0.5\text{KB} \rightarrow \heartsuit$

Usando pagine di **64KB** (2^{16}):

- **dimensione** della tabella: $2^{16} = 64\text{KB} \rightarrow \heartsuit$
- **frammentazione interna** media: $\frac{2^{16}}{2} = 32\text{KB} \rightarrow !$

Come si può notare la scelta di una dimensione o del numero di pagine influenza l'altra grandezza.

→ bisogna scegliere una **dimensione di pagina** che abbia un buon **compromesso** tra i due valori.
→ Il compromesso è tra **frammentazione e performance**. Perché aumentato il numero di pagine riduco la frammentazione ma diminuisco le performance poiché il SO deve gestire un grande numero di pagine; invece diminuendo il numero di pagine aumento la frammentazione interna media per ogni pagina.

Validità delle pagine virtuali

Raramente un processo usa tutto il suo spazio di indirizzamento virtuale.

→ quantità di memoria usata tipicamente da una applicazione desktop si aggira intorno: $\sim 100\text{MB}$.
(single process)

Lo spazio virtuale che un processo potenzialmente può utilizzare è pari all'intero spazio di indirizzamento: se ho **16bit** per indirizzo → **4GB**

Process Name	User	% CPU	ID	Memory	Disk read total
spotify	giovanni	2,56	1774	354,9 MB	27,4 M
opera --type=renderer --crash	giovanni	1,88	7736	57,7 MB	N
opera	giovanni	0,72	6386	92,0 MB	10,9 M
pipewire-pulse	giovanni	0,60	2466	10,3 MB	N
gnome-system-monitor	giovanni	0,60	17220	67,1 MB	9,3 M
Discord	giovanni	0,60	18279	324,5 MB	13,6 M
spotify	giovanni	0,55	17394	147,0 MB	455,6 M
spotify	giovanni	0,55	17511	44,3 MB	33,3 M
pipewire	giovanni	0,51	2453	7,7 MB	86,0 k
gnome-shell	giovanni	0,38	2734	282,2 MB	91,5 M
Discord	giovanni	0,34	17534	93,3 MB	114,7 M
Discord	giovanni	0,26	18322	9,7 MB	N
code	giovanni	0,09	5726	39,9 MB	N
mdbook	giovanni	0,09	6280	36,1 MB	14,6 M
opera --type=renderer --crash	giovanni	0,09	6636	167,2 MB	6,1 M
spotify	giovanni	0,09	17553	22,1 MB	3,8 M
Discord	giovanni	0,09	17877	14,0 MB	2,7 M
MathWorksServiceHost	giovanni	0,04	2947	55,6 MB	218,8 M
code	giovanni	0,04	5607	67,2 MB	1,6 M
code	giovanni	0,04	5741	502,6 MB	854,0 k
java	giovanni	0,04	5935	329,9 MB	2,8 M
opera	giovanni	0,04	6336	285,1 MB	541,9 M

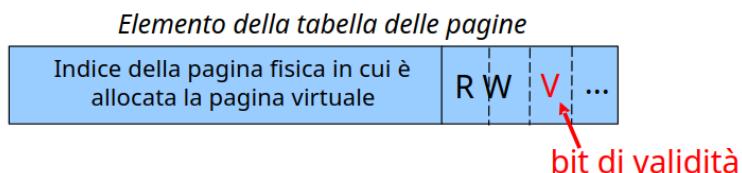
In realtà alcune delle pagine all'interno dello spazio virtuale allocato, non sono veramente utilizzate.

→ tra i bit di controllo possiamo aggiungere un *validity bit*.

Quindi il SO può marcare le pagine **virtuali in uso** usando tale bit nella tabella delle pagine.

il bit di validità so riferisce alle pagine virtuali → infatti ogni entry point della tabella delle pagine identifica una pagina virtuale.

Il bit di validità viene attivato nel momento in cui la pagina è **allocata** dal processo (es. tramite `malloc()`).



Supponendo che all'interno della page table sono contenute tutte le possibili pagine che un processo può utilizzare. → copre l'intero spazio di indirizzamento.

Allora potremmo avere due casistiche nel momenti in cui un processo genera un indirizzo virtuale la cui parte che **identifica il numero** di pagina non è associato a nessun frame fisico.

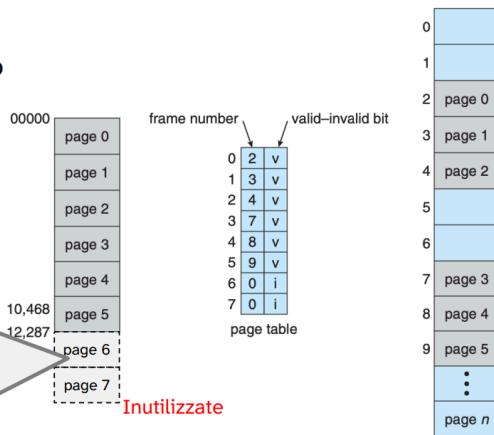
Le cause di questa situazione sono dovute al fatto che:

- la pagina potrebbe esser stata swappata $V = 0$;
- l'indirizzo virtuale non appartiene ad alcuna regione di memoria valida del processo.

Esempio:

- Indirizzi a 14 bit
- Spazio virtuale: 16Kb
- Dim. pagina: 2Kb

- Se il programma tentasse di accedere a queste pagine, la **MMU** genererebbe una **exception**.
- Il SO non uccide subito il processo se quella pagina appartiene al suo spazio di indirizzamento -> cerca di allocarla
- Il SO uccide il processo ("**Segmentation fault**") se non riesce a mappare la pagina virtuale in una pagina (frame) fisica

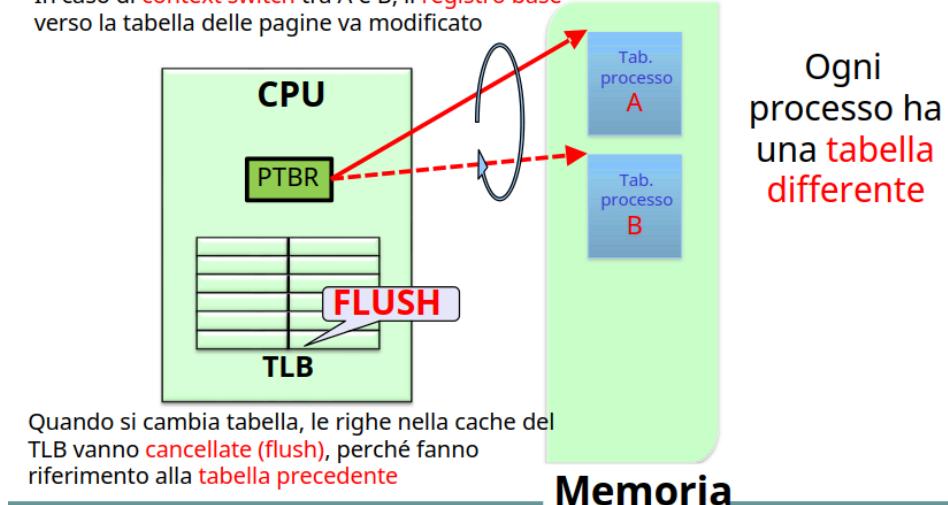


La MMU genera quindi genera una **exception** (page fault).

- Il SO non termina subito il processo se la pagina appartiene al suo spazio di indirizzamento, infatti viene eseguita l'ISR per gestire il page fault che cerca di allocarla dalla memoria secondaria.
 - Il SO termina il processo solo se il risultato del page fault handler non mappa la pagina virtuale in una pagina fisica → perché non è stata trovata nella memoria secondaria.
 - Oppure se l'operazione che il processo tenta di fare su tale area non è valida secondo i permessi descritti su questa.

Cosa accade durante un **context switch**

In caso di **context switch** tra A e B, il **registro base** verso la tabella delle pagine va modificato



- La TLB viene popolata a **run-time** → inizialmente saranno presenti solo page fault che faranno al SO a ricaricare le pagine in memoria principale.

Oltre al bit di validità è presente un ulteriore bit di controllo: il *dirty bit*.

Questo dirty bit è legato al fatto che la pagina è stata scritta durante la sua permanenza in memoria fisica.

Quando un processo **scrive** in una pagina fisica, la MMU setta automaticamente il dirty bit a **1**. Questo bit indica che il contenuto della pagina **non coincide più** con la copia originale presente su disco. → tale informazione ha un implicazione durante lo **swap-out**.

- Se il dirty bit = 1, significa che la pagina contiene modifiche **che devono essere salvate nello swap**, altrimenti andrebbero perse.
- Se il dirty bit = 0, significa che la pagina **non è stata modificata** e che **esiste già una copia valida** della pagina su disco (es. nell'eseguibile del processo, è stata già salvata precedentemente).

Per il meccanismo di coerenza caching, si deve garantire la coerenza tra memoria grande e memoria piccola.

Struttura della tabella delle pagine

Bisogna capire quale struttura sia più adatta a contenere la tabella delle pagine al fine di risolvere diverse problematiche:

Problemi: le tabelle delle pagine:

- hanno grosse dimensioni
- sono numerose (una per ogni processo)
- sono “sparse” (poche pagine valide)

Soluzioni: per ogni problema

- Paginazione gerarchica → evita il problema della **grandezza**
- Tabella delle pagine basata su hash → evita il problema della **numerosità**
- Tabella delle pagine invertita → non si usa più (ideale per vecchi sistemi operativi)
→ per evitare il problema della **sparsità**

Paginazione gerarchica

Suddivisione della tabella delle pagine in parti più piccole, secondo una **organizzazione gerarchica**.

- La MMU divide l'indirizzo di pagina in più parti (p_1, p_2), tante quanti sono i livelli di gerarchia.
- Nella tabella di primo livello, trova l'indirizzo della tabella di secondo livello, e così via.

Nota: la MMU **impiega più tempo per attraversare la tabella gerarchica** (aumentano i tempi di accesso).

Vediamo il motivo di questo tempo aggiuntivo per attraversare la memoria: (2 livelli di gerarchia) indirizzo virtuale: $(p_1, p_2, offset)$.

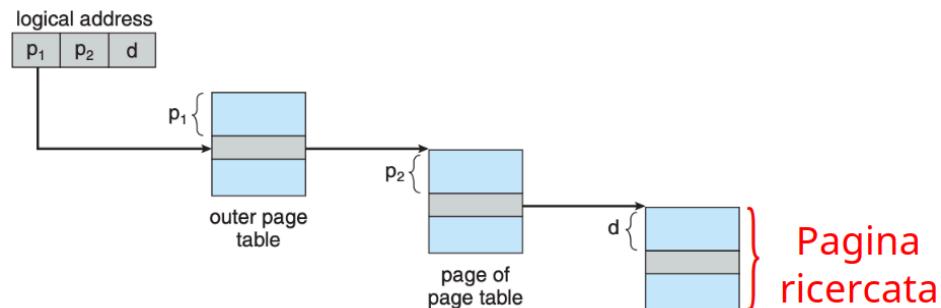
Il flusso sarebbe:

1. La MMU accede alla tabella di primo livello e usando p_1 (**accesso a memoria**) ottiene l'indirizzo base della tabella di secondo livello.
2. La MMU accede alla tabella di secondo livello e usando p_2 (**accesso a memoria**) ottiene il numero di frame corrispondente.
3. Infine, la MMU accede alla memoria principale nel frame trovato e utilizza $offset$ (**accesso a memoria**) per recuperare la casella desiderata.

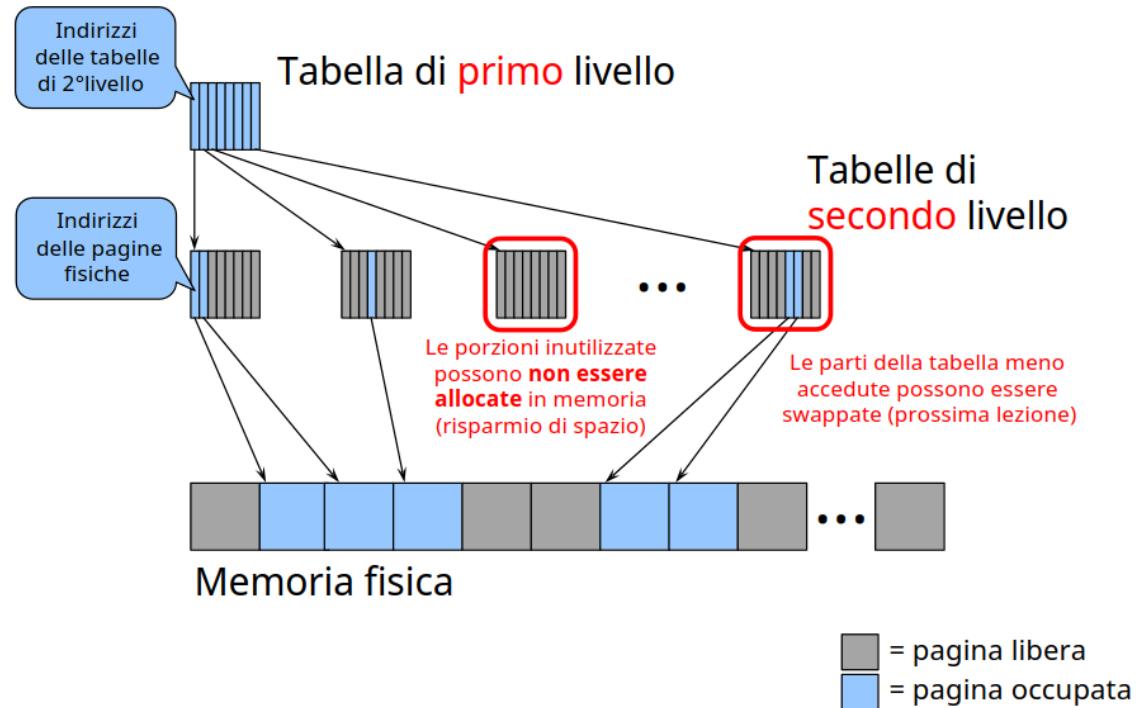
Il motivo dell'aumento del tempo per attraversare la tabella è che la MMU ha bisogno di fare molti **più accessi a memoria fisica** per tradurre un **singolo indirizzo virtuale**.

Nel caso di paginazione semplice (ad un livello) il numero di accessi totali per operare la traduzione di un indirizzo virtuale è pari a 2.

1. Accesso alla page table
 2. accesso alla memoria fisica
-



- Con l'utilizzo di questa struttura si è risolto il problema della grandezza di una tabella delle pagine.



- Linux utilizza 4 livelli gerarchici per la tabella delle pagine di ogni processo.
- Questa struttura serve per **evitare di allocare una struttura grande quanto tutto lo spazio indirizzabile dal processo**.
→ sono presenti delle porzioni (tabelle intermedie) per cui è possibile evitare l'allocazione perché vuote

Le singole tabelle sono più piccole rispetto alla tabella non gerarchica.

ESEMPIO DI PAGINAZIONE A DUE LIVELLI:

Nella paginazione gerarchica, il “numero di pagina” nell’indirizzo virtuale viene **a sua volta suddiviso in più parti** (tante quante sono i livelli di gerarchia).

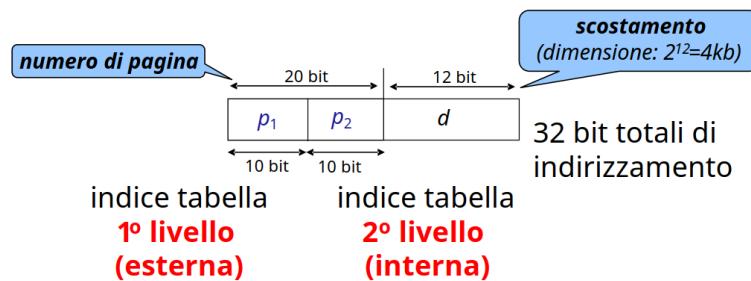


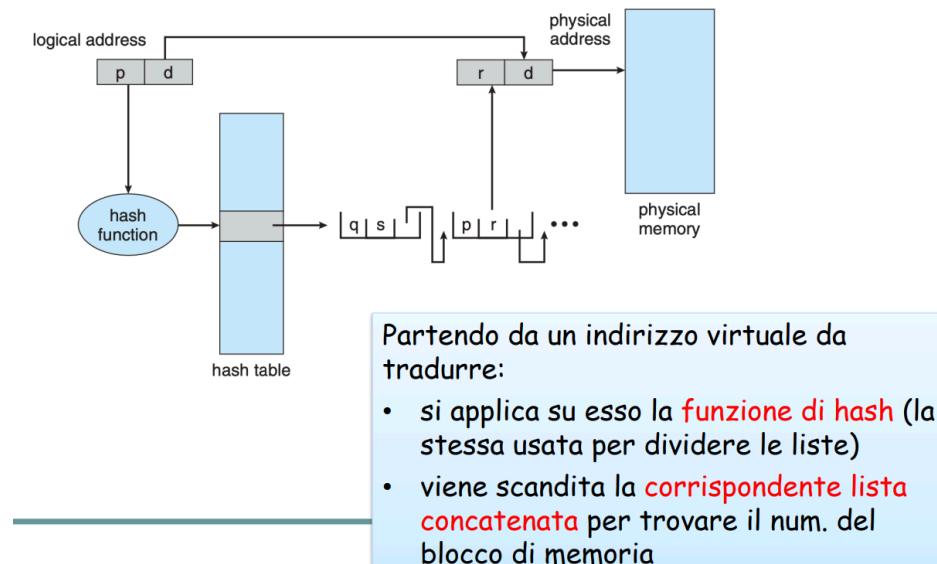
Tabelle delle pagine basate su hash

Le righe della tabella delle pagine sono organizzate utilizzando una **lista concatenata (linked list)**.

- Si memorizzano esclusivamente le righe per le pagine valide.
- Otteniamo quindi un ulteriore **risparmio di memoria**, ma ciò **rallenta la ricerca** (occorre scandire la linked list, ricerca basata sul contenuto).

Per ottimizzare i tempi di ricerca, si dividono le righe su **tante liste concatenate di piccole dimensioni**.

- Una funzione di hash è applicata al numero della pagina virtuale.
- Gli elementi (entry) con lo **stesso valore della funzione di hash** sono collocati nella **stessa lista concatenata**.



- Dopo l'applicazione della funzione di hash sul numero di pagina, che si trovava nell'indirizzo virtuale, si individua una delle M pagine (liste concatenate).
- Dove $M < N$ perché la funzione di hash produce delle collisioni (N è il numero di pagine totali).
- Dopo aver individuato la lista concatenata si cerca all'interno di questa il numero di pagina che identifica la pagina fisica in cui è mappata la pagina virtuale.

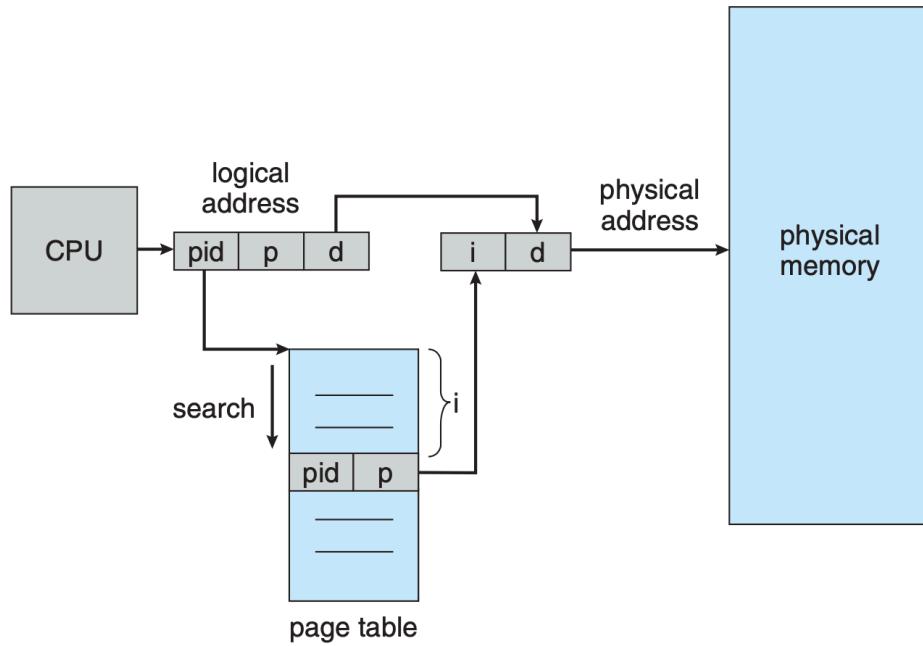
Ovviamente in questo approccio la lunghezza delle liste concatenate è contenuto rispetto al caso in cui abbiamo solo una lista concatenata di tutte le pagine associate ad un processo.

Tabella delle pagine invertita

Negli schemi precedenti esiste **una tabella distinta per ogni processo**.

In questo approccio, **tabella delle pagine invertita**, si hanno queste caratteristiche:

- **Una sola tabella delle pagine comune a tutti i processi**.
- Questa tabella ha un elemento per **ogni pagina fisica**.
- Ogni elemento contiene l'indirizzo **virtuale** della pagina memorizzata **in quella locazione fisica**, con informazioni sul processo detentore della pagina.



Una sola è la tabella delle pagine → globale.

Il numero di righe è pari al numero di pagine fisiche (invece che virtuali).

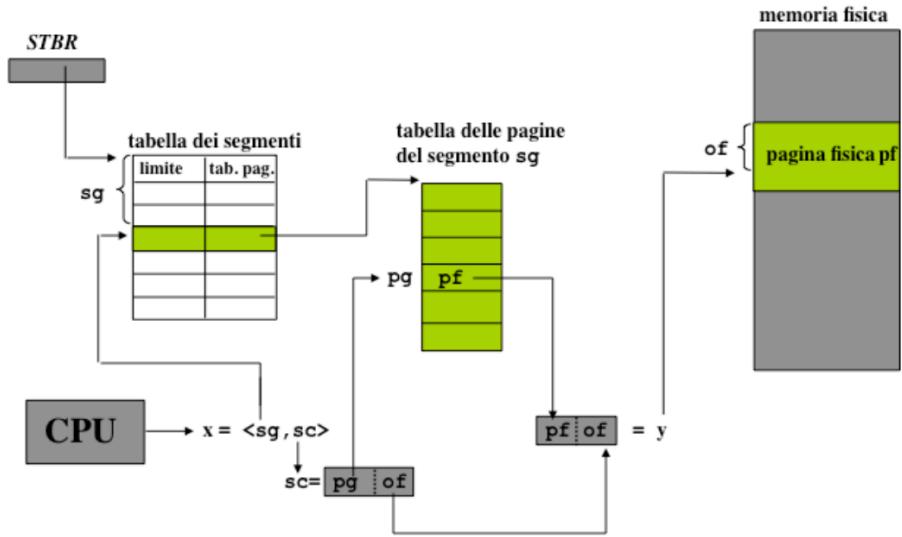
- Ogni entry della tabella contiene il PID, del processo che possiede la pagina fisica, e l'indirizzo virtuale di tale pagina.
- Una volta trovata la riga corrispondente si valuta l'offset rispetto l'indirizzo di base della tabella delle pagine invertita.
Tale offset corrisponde alla prima parte dell'indirizzo fisico della pagina in memoria fisica.

Infatti l'ultima operazione è quella di inserire l'offset nella parte dell'indirizzo virtuale in cui è presente il numero di pagina e il PID.

In questo approccio risparmiamo la numerosità delle pagine, oltre al fatto che queste non sono più sparse nella memoria fisica. → si può sfruttare il principio di località.

Segmentazione paginata

Utilizzata in Linux.



In questa soluzione si utilizza principalmente una paginazione con il supporto hardware.

Ma a livello software si sfruttano tutti i vantaggi della segmentazione:

- condivisione dei segmenti
- granularità fine per l'assegnazione dei permessi → protezione delle aree di memoria
- dimensione variabile dei segmenti

Questa tabella dei segmenti viene risolta in software, mentre la tabella delle pagine viene risolta in hardware sempre dall'MMU.

La **tabella dei segmenti** è unica per ogni processo.

La **tabella delle pagine** è unica per ogni segmento del processo.

Quindi per ogni segmento avremo una tabella delle pagine in cui sono mappate le pagine virtuali corrispondenti a tale segmento in pagine fisiche.

Quando il processore utilizza un indirizzo virtuale $x = \langle sg, sc \rangle$, sg identifica un segmento all'interno della tabella dei segmenti.

Una volta ottenuto l'indirizzo base della tabella delle pagine per il segmento identificato precedentemente, viene utilizzata l'altra parte dell'indirizzo virtuale sc per identificare la pagina fisica.

→ si utilizza sc come offset rispetto l'indirizzo base della tabella delle pagine.

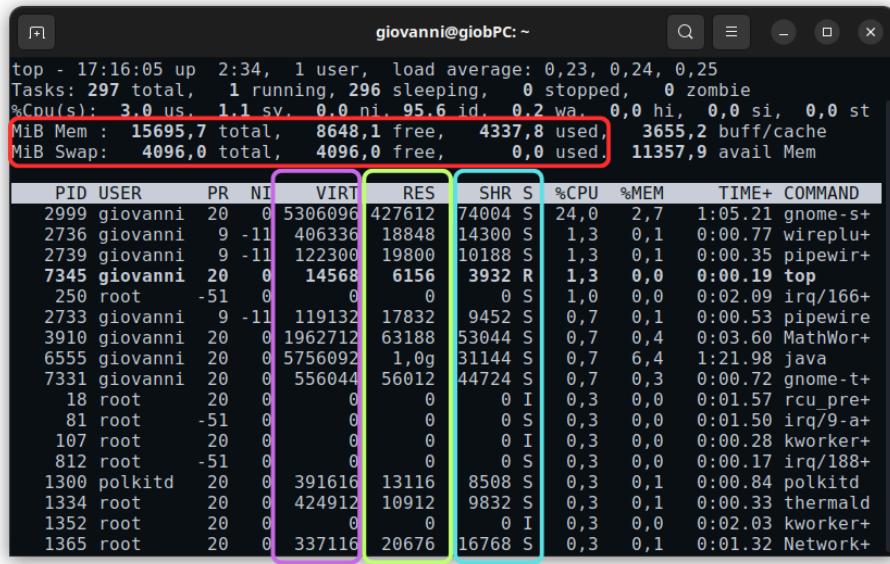
A questo punto, ottenuta l'indirizzo base della pagina fisica, si somma a questo l'offset of per ottenere l'indirizzo fisico a cui il processo fa riferimento.

Ovviamente durante tutto questo processo si devono verificare le condizioni che non causino inconsistenze tra i processi, come le condizioni di limite con **STBL** e **PTLR**.

Quindi nella tabella dei segmenti individuo l'*entry point* che fa riferimento alla tabella delle pagine per quel segmento.

Gestione della memoria in Linux

Con l'uso di `top` possiamo avere info sull'utilizzo di memoria fisica utilizzata e la dimensione degli spazi di indirizzamento di ogni processo.



- **Memoria RAM e SWAP del sistema, il loro utilizzo e la quantità libera**
- **Spazio di indirizzamento utilizzato dal processo**
- **Memoria “residente”, ovvero lo spazio occupato in RAM dal processo**
- **Pagine condivise dai processi (es. codice librerie)**

Linux utilizza una gestione della memoria del tipo **paginata** e **segmentata**.

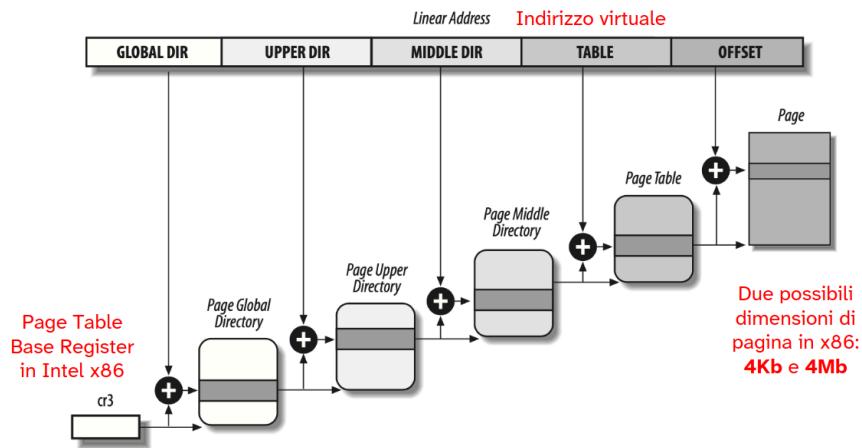
- Paginazione: la memoria è suddivisa in piccole unità chiamate *pagine* che possono essere mappate in memoria fisica.
- Segmentazione: è una tecnica più vecchia, ma non più utilizzata in modo prevalente nelle versioni moderne, ma offre diversi vantaggi rispetto la paginazione che si intendono sfruttare.

Una delle caratteristiche di Linux è la sua **portabilità**, ossia capacità di girare su più architetture diverse. Tale gestione della memoria infatti è fatta per essere compatibile con diverse piattaforme hardware, rendendo il sistema operativo glessibile e adattabile a veri tipi di dispositivi

Infatti tale gestione della memoria è supportata anche per sistemi con grandi quantità di memoria (NUMA) e multi-processore (SMP).

Linux sfrutta la paginazione gerarchica per mantenere le tabelle delle pagine. Questa gerarchia solitamente è su 4 livelli, infatti ogni indirizzo virtuale (o lineare) è suddiviso logicamente in campi ognuno dei quali identificano un offset per la tabella delle pagine di un livello.

ESEMPIO DI TRADUZIONE DI UN INDIRIZZO VIRTUALE:



Ogni pagina ha una dimensione fissata che può esser modificata all'atto della compilazione del kernel.

Tipicamente, le due dimensioni possibili in x86 sono 4KB e 4MB. Per Linux di default ogni pagina ha una dimensione di 4KB.

Per riferirci alla prima tabella delle pagine che si trova al livello più basso sfruttiamo l'indirizzo virtuale mantenuto all'interno del *control register*, che per x86 è il register 3.

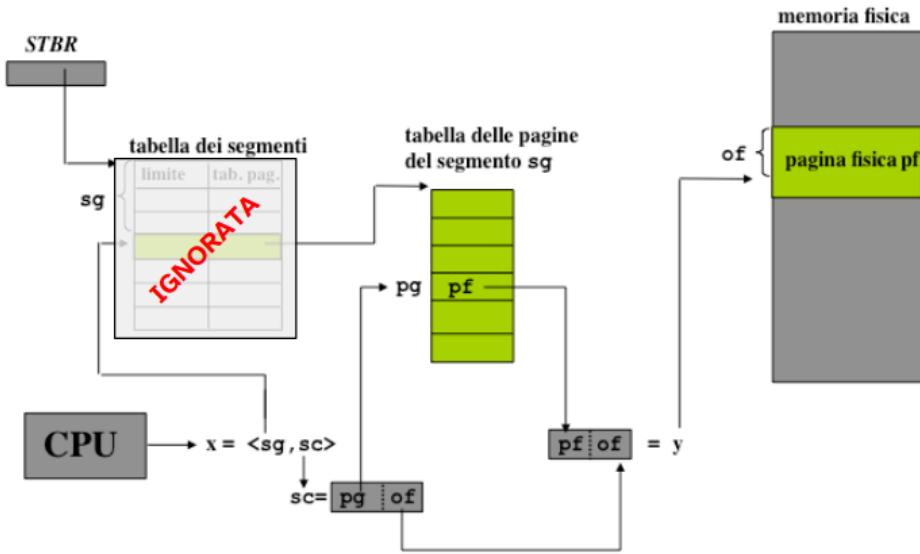
Questo registro viene popolato all'atto del **context switch**, ogni processo ha associata la propria gerarchia di tabelle di pagine. Essendo una caratteristica direttamente legata al processo, tale indirizzo è contenuto all'interno del descrittore, che in Linux è chiamato `task_struct`.

Spazio di indirizzamento virtuale

Le nuove versioni di Linux **non si avvalgono del processore per la segmentazione**, ovvero non utilizzano il **meccanismo di segmentazione** fisico implementato dentro la **MMU**.

Per motivi di:

- portabilità
- efficienza del context switch e traduzione degli indirizzi



Viene quindi utilizzata una paginazione segmentata in cui la segmentazione è implementata via software. In modo da sfruttare i vantaggi di questa.

La memoria è organizzata in **Virtual Memory Areas (VMA)** che consiste nell'unità di virtualizzazione della memoria.

Ogni VMA rappresenta un **blocco contiguo** di **memoria virtuale** e corrisponde ad un segmento.

Quindi possiamo avere VMA, ovvero **pagine virtuali contigue**, contenenti codice, stack, librerie,

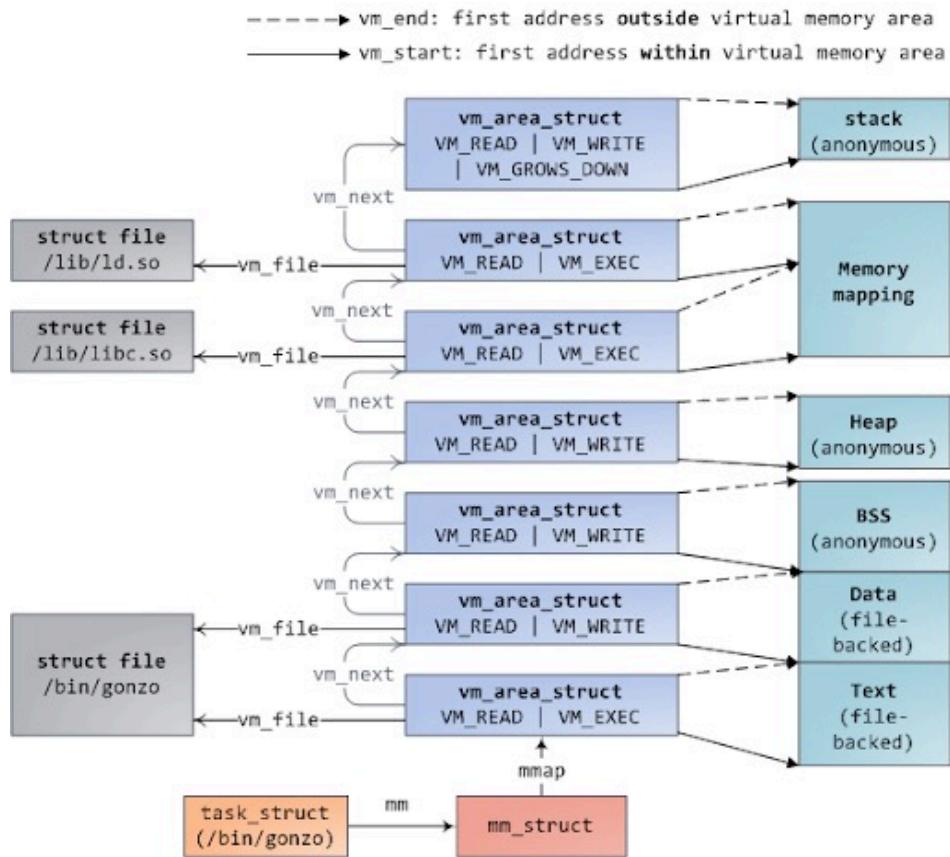
Ogni VMA è **tipizzata** per il **contenuto**, come avviene per i segmenti. Inoltre per ognuna possiamo gestire i permessi, per tipologia di pagina.

Quindi possiamo avere una **maggior granularità** per gestire l'accesso ai blocchi contigui rispetto alla paginazione in cui la granularità era più fine.

Ciò mi permette di proteggere e condividere porzioni dello spazio di indirizzamento in base al suo contenuto.

Struttura di una VMA

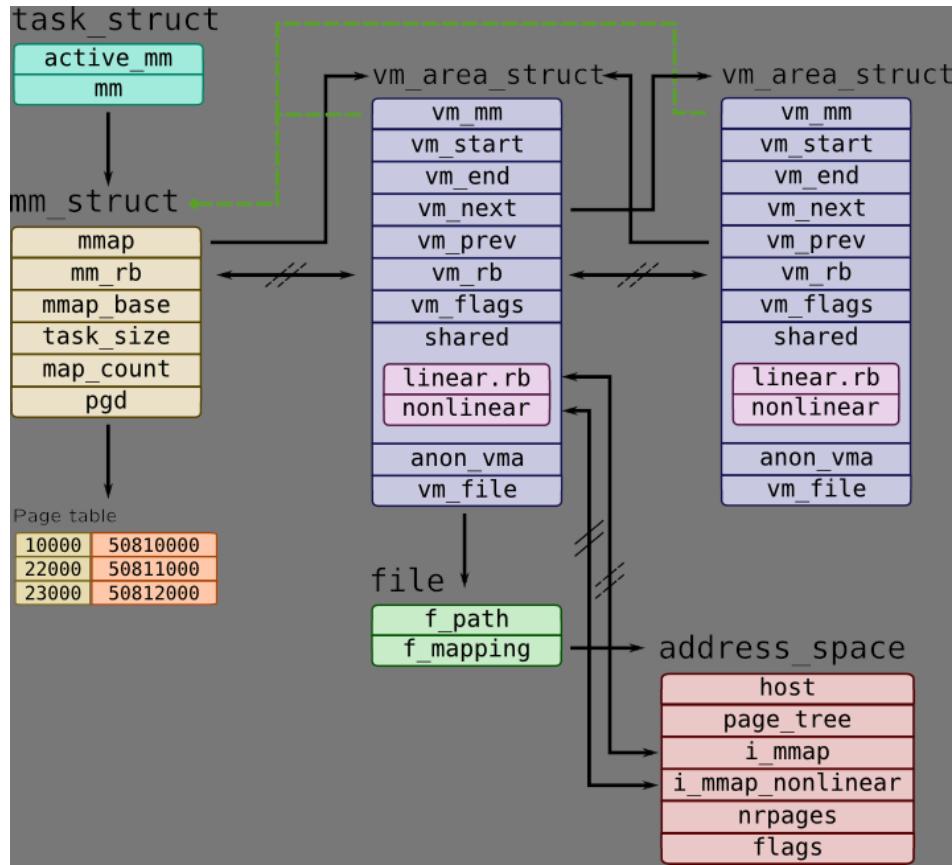
Una VMA è gestita dal kernel mediante una struttura dati che contiene tutte le informazioni necessarie. Il puntatore a tale struttura è contenuto all'interno del descrittore del processo.



- `mm_struct` è la struttura del kernel di Linux che rappresenta la **gestione della memoria virtuale di un processo**.
- Ogni `vm_area_struct` rappresenta una Virtual Memory Area. Ognuna ha un insieme di permessi e la posizione nello spazio di memoria virtuale.
- Ogni `vm_area_struct` è associata ad un blocco contiguo di memoria contenente dati di una specifica categoria.
- Inoltre ogni struttura per le VMA punta alla successiva.

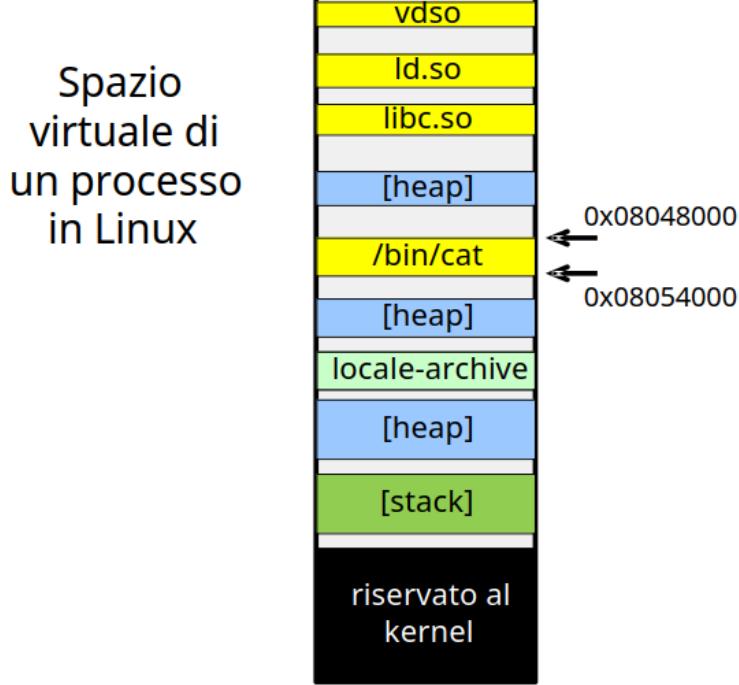
Le pagine `anonymous` sono quelle che non sono associate a nessun file esterno, ovvero in memoria di massa. Quindi all'atto della terminazione del processo, il contenuto di tali aree viene perso.

Invece le pagine abbinate ad un file sul disco sono dette `memory-mapped`



- `vm_file` si riferisce al file presente in memoria di massa, quindi non è contenuto nelle VMA anonime.
- All'interno della `mm_struct`, che rappresenta l'intera memoria virtuale, è presente anche il puntatore alla `page table` che il processo in questione: `gpd`.

ESEMPIO DI SPAZIO VIRTUALE DI UN PROCESSO IN LINUX



- Ogni VMA ha un range e una tipologia che identifica il contenuto.
- Gli indirizzi che vengono utilizzati sono indirizzi lineari o virtuali.
- C'è sempre la presenza, per ogni memoria virtuale di un processo, una **porzione riservata al kernel**.

Possiamo visualizzare la disposizione delle VMA all'interno della memoria virtuale utilizzando le informazioni contenute all'interno dello pseudo-filesystem `proc`.

Utilizzando infatti `cat /proc/<PID>/maps` otteniamo una cosa del genere:

0027e000-0027f000 r-xp 00000000 00:00 [vdso]	Codice
0061a000-00638000 r-xp 00000000 08:02 4872 /lib/ld-2.12.1.so	
00638000-00639000 r-p 0001d000 08:02 4872 /lib/ld-2.12.1.so	
00639000-0063a000 rw-p 0001e000 08:02 4872 /lib/ld-2.12.1.so	
0063c000-007c1000 r-xp 00000000 08:02 5433 /lib/libc-2.12.1.so	
007c1000-007c2000 --p 00185000 08:02 5433 /lib/libc-2.12.1.so	
007c2000-007c4000 r-p 00185000 08:02 5433 /lib/libc-2.12.1.so	
007c4000-007c5000 rw-p 00187000 08:02 5433 /lib/libc-2.12.1.so	
007c5000-007c8000 rw-p 00000000 00:00 0	
08048000-08053000 r-xp 00000000 08:02 12016 /bin/cat	Heap (memoria "anonima")
08053000-08054000 rw-p 0000a000 08:02 12016 /bin/cat	
09607000-09628000 rw-p 00000000 00:00 0 [heap]	
b7672000-b7872000 r-p 00000000 08:02 8857 /usr/lib/locale/locale-archive	File memory-mapped
b7872000-b7873000 rw-p 00000000 00:00 0	
b7884000-b7885000 rw-p 00000000 00:00 0	
bfaaa000-bfabf000 rw-p 00000000 00:00 0 [stack]	

- In questo caso non sono presenti VMA condivise tra processi.
- Tutti gli indirizzi presenti sono sempre virtuali.

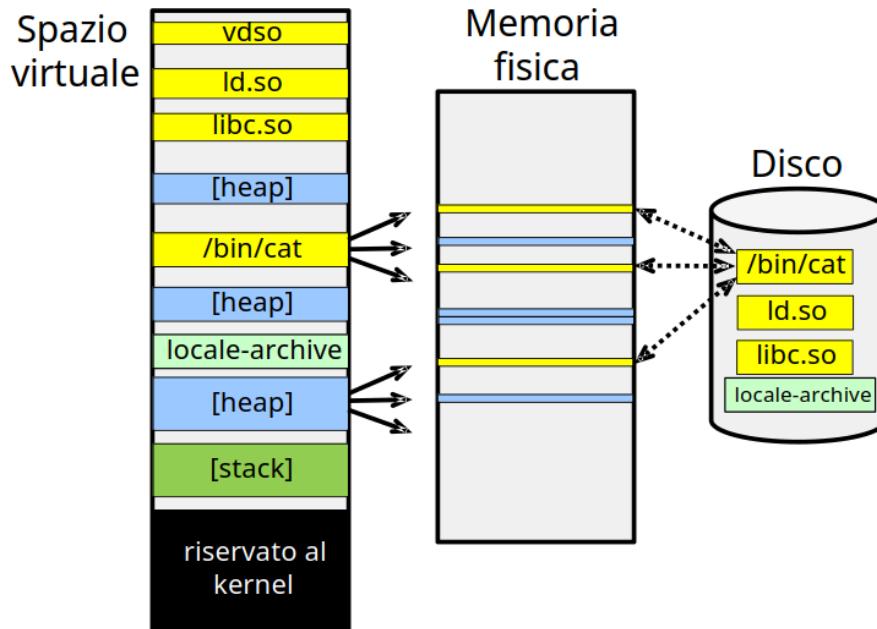
Con VMA condivise tra processi:

```

1  giovanni@giobPC:/proc$ cat 23619/maps
2  5a10863cf000-5a10863d0000 r--p 00000000 103:04 1313762
3  5a10863d0000-5a10863d1000 r-xp 00001000 103:04 1313762
4  5a10863d1000-5a10863d2000 r--p 00002000 103:04 1313762
5  5a10863d2000-5a10863d3000 r--p 00002000 103:04 1313762
6  5a10863d3000-5a10863d4000 rw-p 00003000 103:04 1313762
7  5a10a9e43000-5a10a9e44000 rw-p 00000000 00:00 0
8  7109c6000000-7109c6028000 r--p 00000000 103:04 688600
9  7109c6028000-7109c61b0000 r-xp 00028000 103:04 688600
10 7109c61b0000-7109c61ff000 r--p 001b0000 103:04 688600
11 7109c61ff000-7109c6203000 r--p 001fe000 103:04 688600
12 7109c6203000-7109c6205000 rw-p 00202000 103:04 688600
13 7109c6205000-7109c6212000 rw-p 00000000 00:00 0
14 7109c63c7000-7109c63ca000 rw-p 00000000 00:00 0
15 7109c63e4000-7109c63e5000 rw-s 00000000 00:01 32802
16 7109c63e5000-7109c63e7000 rw-p 00000000 00:00 0
17 7109c63e7000-7109c63e9000 r--p 00000000 00:00 0
18 7109c63e9000-7109c63eb000 r--p 00000000 00:00 0
19 7109c63eb000-7109c63ed000 r-xp 00000000 00:00 0
20 7109c63ed000-7109c63ee000 r--p 00000000 103:04 688597
21 7109c63ee000-7109c6419000 r-xp 00001000 103:04 688597
22 7109c6419000-7109c6423000 r--p 0002c000 103:04 688597
23 7109c6423000-7109c6425000 r--p 00036000 103:04 688597
24 7109c6425000-7109c6427000 rw-p 00038000 103:04 688597
25 7ffc4d086000-7ffc4d0a8000 rw-p 00000000 00:00 0
26 ffffffff600000-fffffffff601000 --xp 00000000 00:00 0
27

```

Vediamo come possono essere mappate tali pagine virtuali in pagine fisiche in memoria fisica:



Come possiamo vedere le pagine virtuali della VMA `/bin/cat` nella memoria fisica non sono allocate in maniera contigua.

Inoltre tali pagine sono `memory-mapped`, quindi è presente una loro copia in memoria di massa.

Lo stesso non si può dire per le pagine fisiche che corrispondono allo **heap** del processo.

// TODO: continua.

Virtualizzazione

Una macchina **virtuale (VM)** è una emulazione (mediante tecniche sw/hw) di una macchina reale.

Ogni macchina virtuale esegue il proprio **sistema operativo** ed applicazioni.

Le VM sono gestite da una **virtual machine monitor (VMM)**, o **hypervisor**. Più macchine virtuali condividono le risorse fisiche della macchina su cui eseguono.

Su una stessa macchina reale possono coesistere diverse macchine virtuale attive che condividono le risorse di questa.

Motivi che hanno portato all'utilizzo delle VM

- Performance
- Flessibilità
- Affidabilità
- Sicurezza

Per ottenere queste qualità sarebbe necessario isolare ogni applicazione in modo che una di queste vada in crash le altre non sarebbero influenzate.

Senza VM, il problema sarebbe che per ottenere questo isolamento usando solo hardware fisico, bisognerebbe comprare una macchina fisica per ogni singolo servizio.

Utilizzare tante macchine fisiche quanti sono i servizi oltre ad essere molto costoso è anche uno spreco di energia e spazio.

Utilizzando invece diverse VM, una per ogni servizio da eseguire, possiamo ottenere la caratteristica di isolamento e inoltre si riducono enormemente gli spazi e il costo necessario.

Questa operazione è detta **consolidation**, ovvero raggruppare più macchine virtuali su un unico server fisico per ottimizzare l'uso delle risorse.

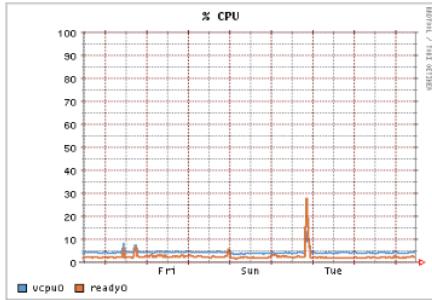
Gestione di sistemi virtualizzati

Le macchine virtuali possono essere velocemente create, configurate, monitorate, migrate...

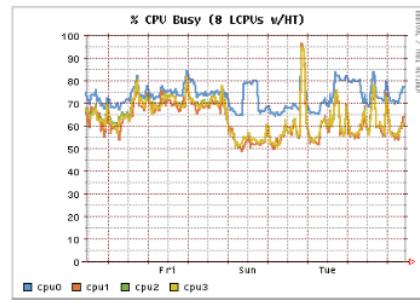
Attraverso strumenti software centralizzati.

Efficienza

Avere più VM su una stessa macchina fisica (**workload consolidation**) permette di sfruttare appieno la capacità dell'hardware e di ridurre i consumi energetici.



Un singolo server



Server consolidati

Flessibilità

Applicazioni **legacy**, basate su SO obsoleti e non più supportati, possono essere eseguite su macchine moderne.

Quindi questo eviterebbe il problema di continuare ad utilizzare macchine legacy unicamente per la loro compatibilità con l'applicazione che è in esecuzione su esse.

Virtualizzazione e cloud computing

Il **cloud computing** permette lo outsourcing (spostare) di VM in centri di calcolo privati o di terze parti (**pay-per-use**).

Quindi invece di eseguire le macchine virtuali sui propri server in azienda, il Cloud Computing permette di spostare la loro esecuzione in centri di calcolo. → Non si paga più per l'hardware ma per l'utilizzo di risorse.

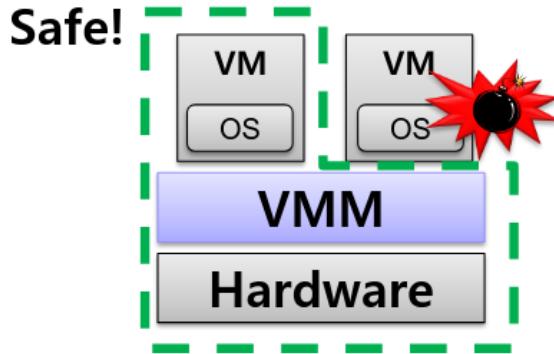
Questo riduce il costo della manutenzione e gestione dei server fisici per le aziende.

Affidabilità e sicurezza

Le macchine virtuali permettono di isolare meglio le applicazioni, questo mi garantisce affidabilità e sicurezza.

- Un'applicazione compromessa o difettosa opera solo sulla VM isolata, e non può interferire con le altre VM su cui sono eseguiti altri servizi.

- Il VMM è l'unico componente **privilegiato** che può gestire l'hardware fisico e le macchine virtuali.



Confronto tra VMM e SO

Il VMM, come il SO, fornisce una **astrazione** della macchina fisica su cui esegue

Il SO è già di per sé un virtualizzatore dell'hardware (di risorse):

- concetto di processo
- concetto di file e directory attraverso il FS
- concetto di memoria virtuale
- socket
- ...

“Container (Fine slide)”

L'astrazione offerta dal SO non è in senso stretto, perché quello che fa effettivamente un virtual machine monitor è quello di emulare completamente l'intera architettura virtuale su un'architettura fisica del tutto diversa.

Posso emulare un'architettura `arm` su una intel `x86` senza problemi, per il tipo di astrazione offerta da un VMM.

Risorse	Sistema Operativo	VMM
CPU	Processi, Thread	CPU virtuale
Memoria	Memoria virtuale	Memoria virtuale
Disco	File, Directory	Disco virtuale
Rete	Socket	Rete virtuale

Architetture principali di una hypervisor

- TIPO 1

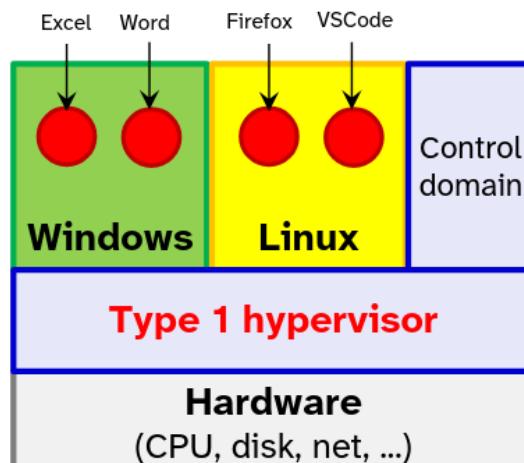
- TIPO 2

1. Il VMM esegue su un “hardware nudo” (**bare-metal virtualization o server virtualization**).

In questa architettura, il software di virtualizzazione è il padrone dell’hardware assoluto.

Questo tipo di hypervisor viene molto utilizzato nei data center.

L’architettura di tipo 1 ha migliori prestazioni perché non c’è un ulteriore layer (livello di indirezione) che divide il VMM dall’hardware, a differenza di ciò che accade nell’architettura di tipo 2



Hypervisor Tipo 1:

2. Il VMM esegue su un SO tradizionale (es. Windows), viene detta **hosted hypervisor**.

Per comunicare con l’hardware il sistema operativo **guest** deve passare per l’hypervisor che a sua volta deve passare per il sistema operativo **host**.

L’hypervisor è considerato per il SO host come un qualsiasi altro processo in esecuzione.

Questo utilizzo facilita l’integrazione tra sistemi operativi, ad esempio, posso copiare file facilmente dal desktop dell’host dentro la macchina virtuale guest.

Ha come svantaggio le performance perché ogni richiesta della VM deve attraversare due strati: l’Hypervisor e poi il sistema operativo host.

Consiste nell’architettura che utilizziamo con VMware Workstation/Fusion, Oracle VM VirtualBox.

In generale un hypervisor deve garantire di virtualizzare:

- la CPU
- la memoria
- l’I/O

Virtualizzazione della CPU

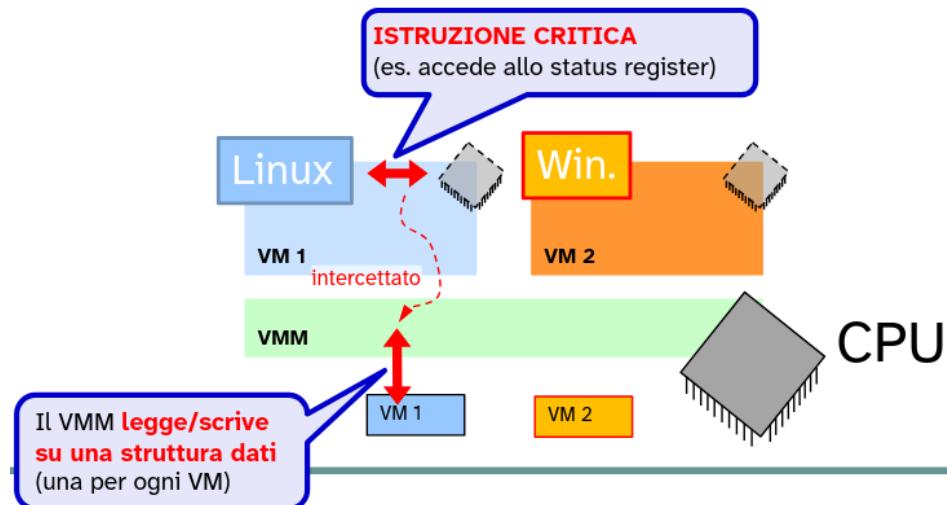
Le prime tecniche di virtualizzazione della CPU consistono nel far credere al kernel della macchina virtuale guest che il **processore virtuale** su cui opera sia fisico.

L'hypervisor deve quindi implementare un mapping 1:1 tra il processore da emulare e quello fisico su cui effettivamente viene eseguito il kernel guest.

Tutto quello che viene eseguito in kernel mode nel guest in realtà non è un vero e proprio kernel mode; in realtà è in user mode rispetto al sistema operativo host.

Quindi se per il sistema operativo host la macchina virtuale non è altro che un semplice processo, deve essere l'hypervisor ad astrarre completamente un macchina fisica per il kernel guest.

L'hypervisor deve quindi intercettare delle “istruzioni critiche” (**sensitive instructions**)



L'hypervisor deve intercettare le istruzioni critiche che necessitano di un grado di privilegio superiore che la VM non ha. C'è la necessità di emulare questa istruzione.

De-privilegging

Primo meccanismo che deve essere implementato all'interno degli hypervisor.

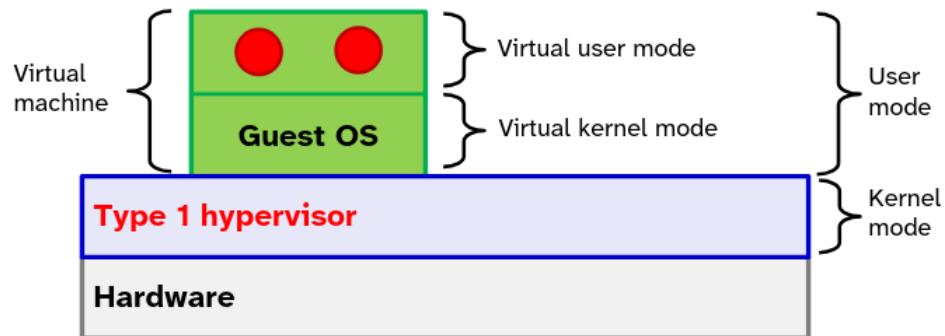
Tale concetto consiste nel “degradare” o togliere i privilegi al SO ospite, spostandolo a un livello di esecuzione inferiore rispetto a quello a cui è abituato a girare, ovvero spostarlo in user mode.

- La VMM (hypervisor) gira nel vero **kernel mode**. È l'unico che ha il controllo diretto e privilegiato dell'hardware fisico
- Il guest SO viene spostato in **user mode** completamente. Quindi quanto il guest SO pensa di operare in kernel mode in realtà di trova ancora in user mode, e chiamiamo questo livello di

privilegio virtual kernel mode

Ovvero:

- VMM → kernel mode
- VM → user mode, dentro cui:
 - → guest SO è in virtual kernel mode
 - → i processi del guest SO sono in virtual user mode

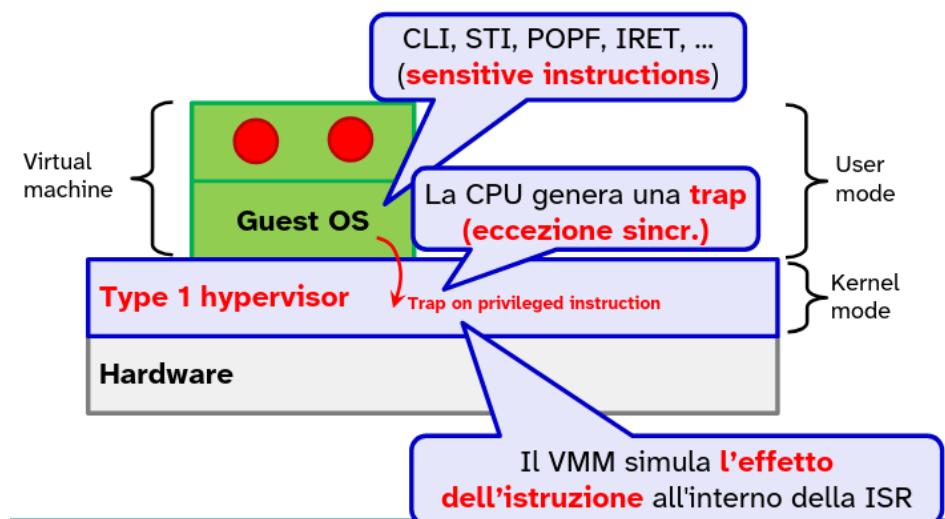


Per permettere al guest SO di operare in kernel mode, un primo meccanismo che si può sfruttare è quello delle trap: nel momento in cui il guest esegue delle istruzioni privilegiate/critiche, allora la CPU virtuale genera una **trap**.

Questo meccanismo è detto: **TRAP-AND-EMULATE**.

Questa trap viene intercettata dall'hypervisor, elabora il tipo di istruzione che ha generato quella trap ed emula l'istruzione privilegiata associata.

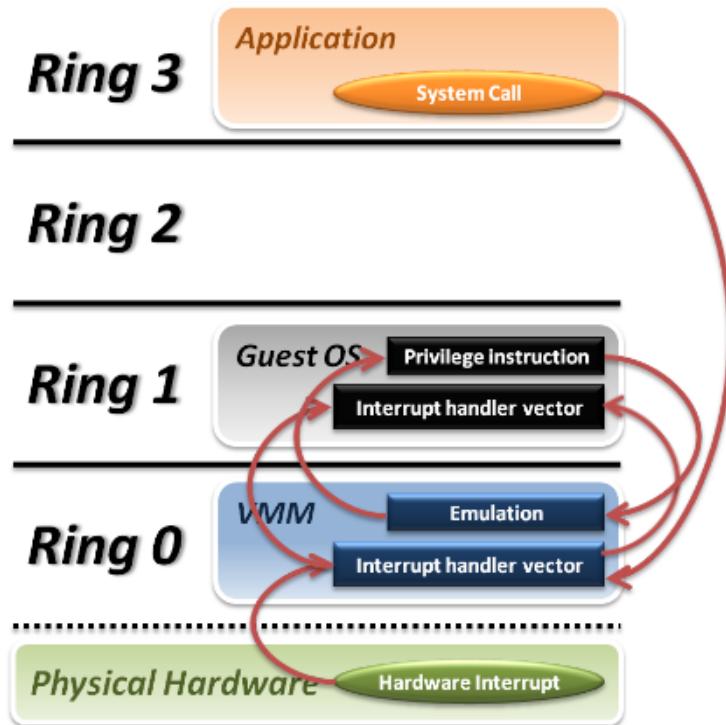
Tutte le istruzioni sensitive non possono eseguire direttamente, ma devono passare per l'hypervisor, e vengono a scatenare una trap → passa il controllo (de-privilegian) all'hypervisor che emula/simula l'effetto dell'istruzione.



Nel caso di CPU intel questa cosa avviene in maniera diversa, perché in queste CPU esistono diversi livelli di privilegio, classificati in Ring.

4 livelli di privilegio, dove Ring 3 ha il più basso livello di privilegio mentre in Ring 0 la VMM.

Cosa accade quando il guest SO vuole eseguire una istruzione sensitive:



- L'applicazione che gira nella VM esegue una system call, questo causa una trap della CPU virtuale che viene gestita dalla VMM. La VMM salta al guest host per far eseguire l'interrupt handler e poi la ISR associata alla trap generata.
- L'interrupt hardware invece sono gestite dalla VMM che eseguono l'ISR e successivamente si salta all'esecuzione dell'ISR del guest SO.
- Invece le istruzioni privilegiate nel guest SO causano il generarsi di una trap che viene intercettata dalla VMM che la gestisce ed emula l'effetto desiderato dal guest SO.

Poniamo in esempio che il guest SO voglia eseguire una **CLI**.

Il VMM simula la **CLI** ponendo a **0** lo **Interrupt Flag** all'interno di una struttura dati dedicata alla VM.

- Lo **Interrupt Flag** nel **registro fisico** della CPU è inalterato
- Le interrupt fisiche sono ancora ricevute dallo VMM

La VMM emula l'effetto della chiamata di sistema, infatti da quel momento in poi la VMM continuerà a ricevere delle interruzioni dalla CPU fisica, ma non le inoltra al guest SO.

Problemi con l'architettura **x86**

L'architettura **x86** tradizionale **non è “virtualizzabile”** con solo il trap-and-emulate.

Il motivo è che molte delle istruzioni sensibili **non generano alcuna trap**

Quindi non è più sufficiente il meccanismo di trap-and-emulate.

Se il guest tenta di eseguire una istruzione sensitive, la CPU **ignora l'istruzione**; tali istruzioni sono dette **“sensibili ma non privilegiate”**.

Questo è stato risolto grazie ad un supporto hardware, cioè sono state introdotte delle istruzioni specifiche per la virtualizzazione.

Ma prima di ciò non era possibile virtualizzare tale architettura.

- Teorema di Popek e Goldberg

Una macchina può essere virtualizzata se ogni istruzione sensitive è privilegiata.

Le istruzioni privilegiate sono quelle che scatenano una *trap* quando eseguite in user mode.

Invece un'istruzione è definita sensitive quanto interagisce direttamente con la configurazione o lo stato dell'hardware sottostante.

Tecniche della virtualizzazione delle CPU

- **Full virtualization**, senza supporto hardware

Utilizza tecniche software complesse come la **Dynamic Binary Translation**. Il VMM legge il binario del Guest SO e lo “riscrive” al volo per renderlo sicuro, senza che il guest SO se ne accorga.

- **Para-virtualization**

il guest SO è sviluppato appositamente per cooperare con il VMM.

Si abbandona l'idea di ingannare il guest SO, è consapevole di essere su una macchina virtuale. Il SO ospite viene riscritto per essere consapevole di girare su una macchina virtuale, quindi invece di provare ad eseguire istruzioni hardware direttamente si interfaccia con la VMM (cooperazione).

- **Full virtualization**, con **supporto hardware**

migliori prestazioni e VMM più semplice.

La CPU stessa ha nuove modalità che permettono al VMM di intercettare le operazioni critiche senza dover riscrivere il codice software.

ESEMPIO: confronto tra **para-virtualization** e **full virtualization**

Il problema da risolvere è lo stesso: il guest SO vuole leggere un registro critico, quello contenente l'indirizzo della **Interrupt Descriptor table**.

Queste due soluzioni ci permettono di farlo ma con approcci differenti:

- **Dynamic Binary Traslation**

Agisce sul **codice binario**.

Il sorgente del sistema operativo non viene toccato, quindi si può utilizzare un qualsiasi sistema operativo, anche Windows.

Il VMM analizza il flusso di istruzioni mentre vengono eseguite, intercettando quelle sensitive.

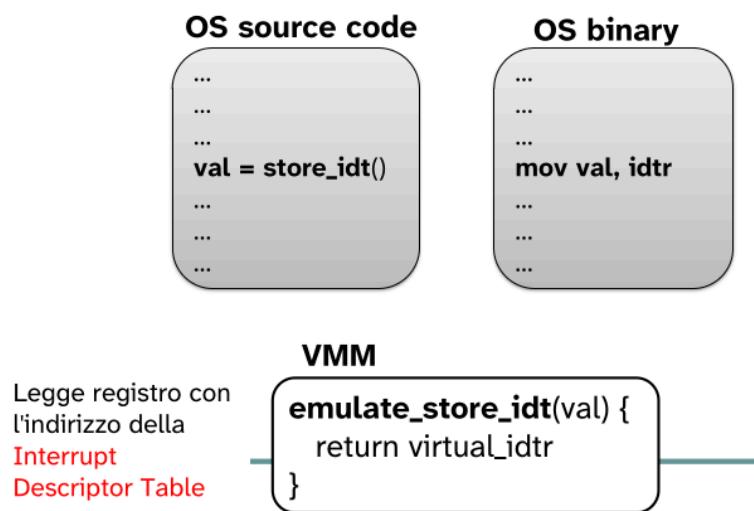
Infatti quando vede `mov val, idtr`, la intercetta e la sostituisce dinamicamente con una chiamata alla sua funzione che emula l'effetto di quella istruzione.

- **Hypercall**

Si agisce sul **codice sorgente**. Gli sviluppatori del sistema operativo modificano il codice: cancellando istruzioni problematiche e inserendo una chiamata esplicita alla VMM, chiamata **Hypercall**.

È una soluzione molto efficiente, ma richiede di poter modificare il codice sorgente del SO e ricompilerlo. (impossibile con Windows)

Quindi consideriamo di trovarci in questa situazione:

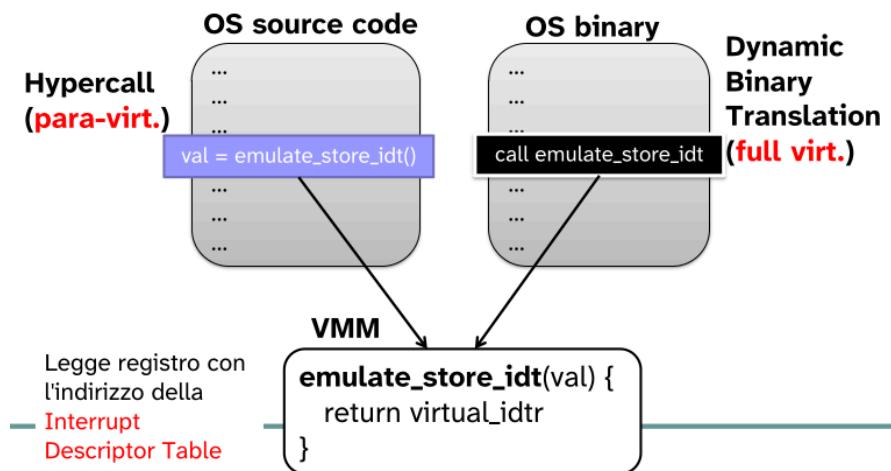


- Codice sorgente
- Codice binario che viene eseguito
- funzione del VMM che emula il comportamento dell'istruzione in esame

Per l'approccio che utilizza la para-virtualization il sorgente deve essere modificato per poter girare su una VM. La modifica consiste nella chiamata alla funzione dal VMM che emula quel comportamento, ovvero `emulate_store_idt()` (Hypercall).

Per l'approccio che utilizza la full virtualization il sorgente rimane invariato, e a tempo di esecuzione abbiamo due principali tecniche che dipendono dalla presenza o meno del supporto hardware.

- Meccanismo software (Dynamic Binary Translation): la VMM sostituisce dinamicamente le istruzioni. Il VMM legge il **codice binario** prima che venga eseguito, individua le istruzioni critiche e le riscrive sostituendole con il codice di emulazione → `emulate_store_idt()`.
- Meccanismo hardware (Intel VT-x / Trap-and-Emulate): in questo caso il codice binario in memoria non viene necessariamente sostituito/riscritto. È la CPU fisica che, quando incontra l'istruzione critica mentre gira la VM (in *non-root mode*), ferma tutto e genera un evento hardware (**VMExit**). Il controllo passa al VMM che esegue l'emulazione e poi restituisce il controllo.



Full virtualization, no supporto hw

Nel caso di una CPU fisica che non supporta la virtualizzazione, in cui non tutte le istruzioni sensitive generano una trap, l'hypervisor di **VMware** introduce tecniche efficienti di full-virtualization per Intel `x86`.

Queste tecniche sono ad esempio la **Dynamic Binary Translation**.

Il codice binario del guest veniva riscritto dinamicamente dall'hypervisor prima di essere eseguito: sostituendo le istruzioni sensibili con un codice di emulazione

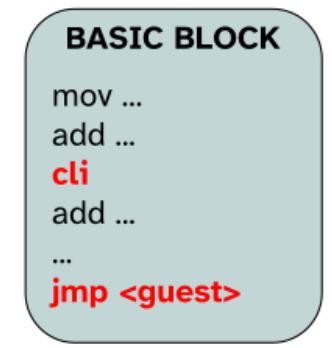
Coprendo il gap della non virtualizzabilità di `x86`.

Oggi con il supporto hardware VMware si è adattata perché è molto più performante.

Dynamic Binary Translation

All'avvio della VM, il VMM analizza a blocchi il codice eseguito dal guest SO.

Ogni blocco, detto **Basic Block** è una breve **sequenza di istruzioni sequenziali** che terminano con una **istruzione di salto**.



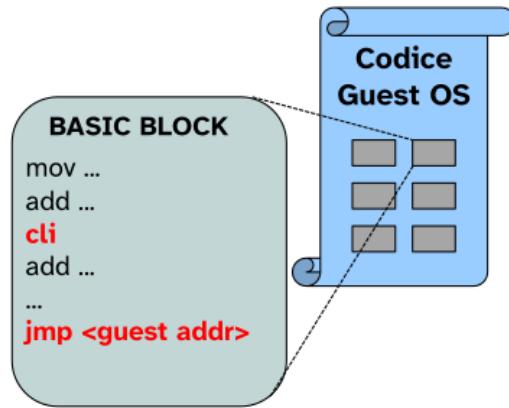
L'Hypervisor quindi va a scandire questi blocchi all'interno del binario del kernel e li analizza e riscrive eventuali istruzioni sensitive.

Il salto finale viene sostituito con una chiamata all'hypervisor per mantenere la catena di traduzioni attiva:

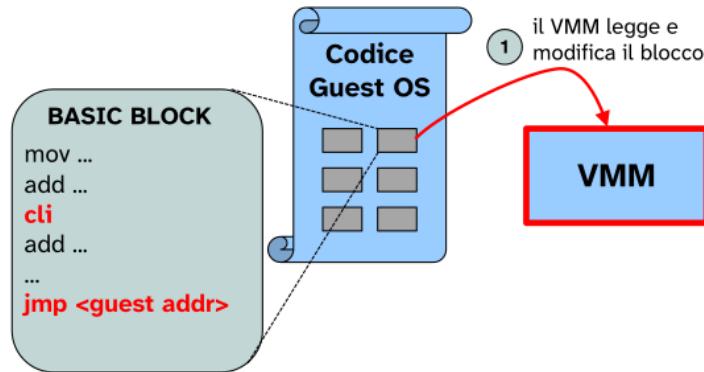
- La CPU esegue il blocco tradotto (sicuro)
- Arriva all'ultima istruzione (salto modificato)
- Il controllo passa al VMM
- Il VMM controlla se il blocco a cui avrebbe saltato la CPU prima della modifica del salto sia già tradotto
 - Se **si**: fa saltare la CPU direttamente alla versione già tradotta e sicura
 - Se **no**: traduce il nuovo blocco, lo salva in memoria cache, e poi fa saltare la CPU lì

In poche parole, il salto finale viene modificato per garantire che la CPU non esegua mai codice originale, ma rimbalzi sempre attraverso il VMM per ottenere il prossimo pezzo di codice tradotto e sicuro.

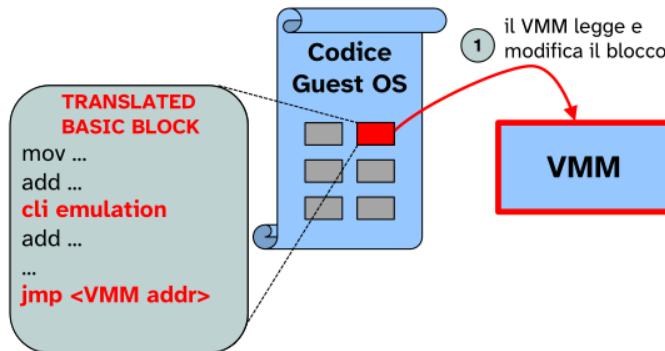
Vediamo cosa accade tutto ciò:



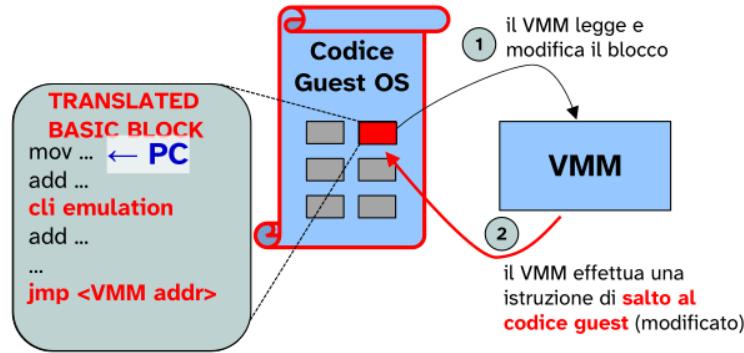
- Codice grezzo che non è stato ancora tradotto dal VMM



- Il VMM legge il basic block e lo modifica



- Dopo la modifica nel basic block viene sostituita l'istruzione sensitive e viene modificato il salto finale per poter permettere al VMM di analizzare il basic block successivo



- Il controllo viene ripristinato al guest SO così che possa eseguire il codice tradotto e sicuro
- Una volta terminata l'esecuzione del blocco l'istruzione di salto riporta il controllo al VMM per continuare con il meccanismo di Dynamic Binary Translation

Para-virtualizzazione

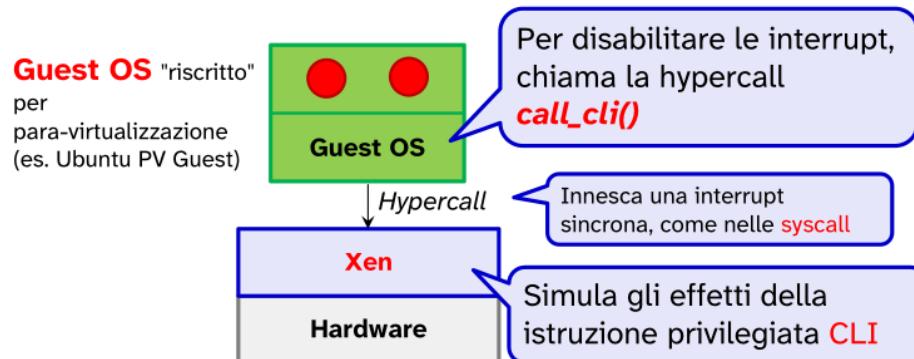
Con questo approccio c'è la necessità di riscrivere il codice sorgente del guest SO in modo che questo cooperi con il VMM.

Il guest SO è consapevole del fatto di esser eseguito su di una macchina virtuale (VM).

- Le istruzioni sensitive nel guest SO sono sostituite da **hypercalls** che fanno riferimento a funzioni del VMM che emulano il comportamento delle istruzioni originarie.

A differenza della full virtualization, si modifica il **codice sorgente del guest SO**, non il codice binario.

La modifica è fatta dal **programmatore** e non dal VMM, quindi i sistemi che possono girare su Hypervisor, che sfruttano questa tecnica di virtualizzazione della CPU, devono essere costruiti ad hoc.



Il guest SO così modificato **non può eseguire sull'hardware fisico**: può eseguire solo in combinazione con il **VMM**.

Inoltre questo approccio non è utilizzabile per sistemi **legacy** oppure **proprietari**, come Windows.

Supporto hardware per la full-virtualization della CPU

Le CPU Intel VT introducono:

- due modalità di esecuzione: **VMX root** e **VMX non-root**;
- **VMCS** (VM Control Structure).

Queste soluzioni sono implementate nell'hardware per evitare tutto l'overhead dovuto alla traduzione dinamica del codice binario e per permettere a qualsiasi guest SO di poter eseguire su una VM.

Hanno come vantaggio:

- semplificano il codice del VMM, buona parte della virtualizzazione è fatta in hardware;
- evita il **ring de-privileging** del guest SO;
- maggiore **efficienza del cambio di contesto** tra VM e VMM;
- garantisce il meccanismo di trap per tutte le istruzioni critiche

Quindi le principali novità sono:

- Modalità di esecuzione (VMX)
 - **VMX Root Mode**: dove gira il VMM. Qui il software ha pieno controllo dell'hardware, esattamente come il kernel di un sistema operativo tradizionale.
 - **VMX Non-Root Mode**: dove gira il Guest SO. In questa modalità, il sistema operativo ospite può girare al suo livello di privilegio “naturale” senza dover essere de-privilegiato. Quindi viene eseguito con il massimo dei privilegi ma in VMX non-root mode.
- Passaggio tra Root mode e Non-Root mode

Il passaggio tra queste due modalità è gestito direttamente dall'hardware tramite eventi specifici:

- **VM Entry**: il passaggio dal VMM al Guest SO. Avviene quando il VMM lancia o riprende l'esecuzione di una VM.
- **VM Exit**: il passaggio inverso, dal Guest SO al VMM. Avviene automaticamente quando il Guest SO tenta di eseguire un'istruzione sensitive (Perché nonostante ha tutti i privilegi, si trova in VMX Non-Root mode).

L'hardware intercetta l'istruzione e restituisce il controllo al VMM affinché possa gestirla.

- **VMCS (Virtual Machine Control structure)**

Per gestire questi passaggi in modo efficiente, intel ha introdotto una struttura dati hardware chiamata **VMCS**. La VMCS agisce come una “scheda cliente” per ogni VM e contiene:

- **Guest state**: lo stato della CPU quando gira la VM. Viene caricato durante la *VM Entry* e salvato durante la *VM Exit*.
- **Host state**: lo stato della CPU quando gira il VMM. Viene caricato durante la *VM Exit*.
- **Control Data**: istruzioni per la CPU fisica su quali eventi devono causare un *VM Exit*

Eventi che possono causare **VM Exit**:

- **Istruzioni sensitive**
 - CPUID
 - RDMSR, WRMSR
 - INVLPG
 - RDPMC, RDTSC
 - HTL, MWAIT, PAUSE
 - VMCALL: nuova istruzione per invocare il VMM
- **Accessi a stato sensitive**
 - MOV DRx: accessi ai debug register
 - MOV CRx: accessi ai control register
 - Task switch: accessi al CR3 (puntatore alla tabella delle pagine)
- **Eccezioni ed eventi asincroni**
 - Page fault, debug exceptions, interrupts, etc.

Definizioni utili

- **Processo:**

Unità base di esecuzione del SO, descrive **l'attività dell'elaboratore relative alla specifica esecuzione di un codice.**

è un'entità dinamica che consiste in: programma da eseguire + stato di esecuzione.

Possiamo avere più processi relativi all'esecuzione di più istanze dello stesso programma.

- **Programma**

Sequenza statica di esecuzione, corrisponde alla codifica in un linguaggio di programmazione comprensibile all'elaboratore, in modo che questo possa essere eseguito.

- **Immagine di un processo**

L'immagine di un processo consiste nell'insieme del codice, area dati globale, stack, heap e il descrittore. Ovvero tutte quelle componenti necessarie all'esecuzione in un contesto di sistema operativo multiprogrammato.

- Stack
- Heap
- Area dati globali
- Codice → rientrante
- PCB
- Stack del kernel → utilizzato dal kernel quando il processo esegue in kernel mode

- **Spazio di indirizzamento**

L'immagine del processo e le risorse da esso possedute costituiscono il suo spazio di indirizzamento.

- **Thread**

Flusso di controllo sequenziale di un processo eseguito in maniera concorrente a questo.

A differenza di un processo non possiede delle risorse private alla sua esecuzione ma le condivide con altri thread dello stesso processo e il processo stesso.

Quello che appartiene ad un thread - processo leggero - sono:

- insieme di registri
- stack

Tutte le altre aree di memoria fanno parte dello spazio di indirizzamento del processo, quindi condivise.