

03 - GESTIONE MEMORIA

MEMORIA VIRTUALE E PAGINAZIONE

slide 1

Il concetto di memoria virtuale

Il concetto di memoria virtuale nasce dalla necessità di **separare** i concetti di:

- **spazio di indirizzamento virtuale**
- **dimensione effettiva della memoria fisica**

Lo **spazio di indirizzamento virtuale** è definito dal numero di parole indirizzabili e dipende esclusivamente dal numero di bit **dell'indirizzo** definito dall'architettura del set di istruzioni (ISA) e **non** dalla dimensione della memoria fisica effettivamente disponibile in un calcolatore.

- se N sono i bit di indirizzo del processore, allora 2^N è il numero massimo di byte indirizzabili o spazio di indirizzamento virtuale

La **dimensione della memoria fisica** è sempre **minore** o **uguale** al suo spazio di indirizzamento -- che può aumentare a seguito di espansione di memoria, sempre entro i limiti dello spazio di indirizzamento.

Memoria virtuale: indirizzi virtuali

- Gli indirizzi a cui fa riferimento il programma in esecuzione sono **indirizzi virtuali**
- Gli indirizzi virtuali sono quelli generati da linker (a partire dall'indirizzo 0): sono quindi **indirizzi rilocabili**
- Lo **spazio di indirizzamento virtuale** è quello definito dalla lunghezza degli indirizzi virtuali:
 - Ad esempio con 32 bit di indirizzo => lo spazio di indirizzamento virtuale è di 232 Byte (4 G Byte)

Dimensione virtuale di un programma

È possibile definire la **dimensione virtuale di un programma** (espressa in termini di indirizzi virtuali):

- Somma delle aree di programma effettivamente presenti
- **Dimensione iniziale**: è quella calcolata dal linker dopo la generazione del codice eseguibile;
- **In fase di esecuzione**: la dimensione virtuale **può variare dinamicamente a causa** di:
 - modifica delle dimensioni della **pila** dovuta a chiamate a funzioni;
 - modifica delle dimensioni dello **heap** dovuta alla gestione delle strutture dati dinamiche (allocazione dinamica della memoria);
 - creazione di aree per la mappatura di **librerie dinamiche** o **aree condivise** con altri processi.

Memoria virtuale: indirizzi fisici

- La memoria principale (Main Memory) effettivamente presente nel calcolatore è chiamata **memoria fisica** e i suoi indirizzi sono detti **indirizzi fisici**.
- In un calcolatore è presente **un meccanismo dinamico di traduzione automatica degli indirizzi virtuali in indirizzi fisici**.
- Il meccanismo di traduzione degli indirizzi da virtuali a fisici è completamente **trasparente** al programmatore.
- Il processore genera un **indirizzo virtuale**, che viene tradotto automaticamente da una combinazione di moduli hardware e software in un **indirizzo fisico**, utilizzato per accedere alla memoria fisica: **meccanismo di traduzione dell'indirizzo (memory mapping)**.
- Il meccanismo di gestione della memoria virtuale più usato è la **rilocalizzazione dinamica tramite paginazione**.

Vantaggi della memoria virtuale

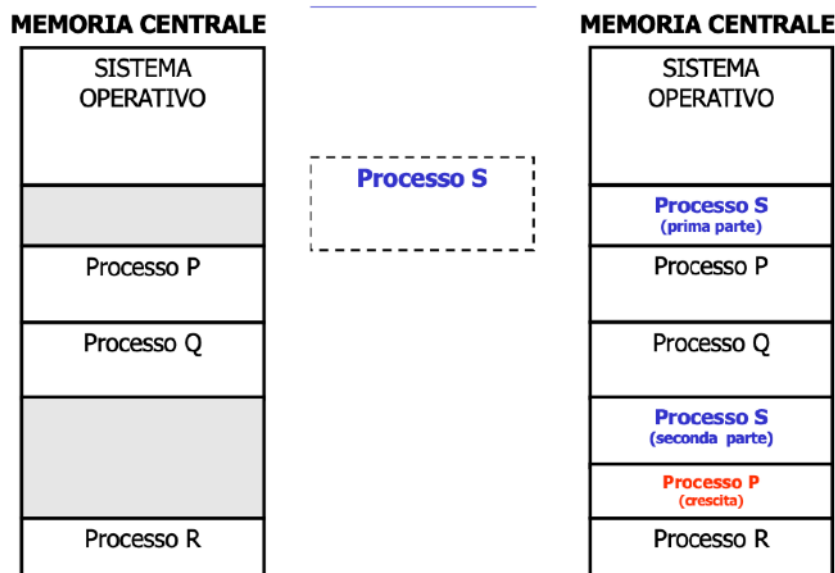
1. Il programmatore può scrivere i propri programmi pensando di avere a disposizione l'intero spazio di indirizzamento virtuale **indipendentemente** dalle dimensioni effettive della memoria fisica.
2. Sia il SO che più processi possono risiedere contemporaneamente in memoria virtuale indipendentemente dalle dimensioni effettive della memoria fisica
3. La dimensione di memoria virtuale di un singolo processo e/o di più processi può essere **maggiore** della dimensione della memoria fisica (che potrebbe essere insufficiente a contenere la memoria virtuale di tutti i processi esistenti).
4. Questo implica che generalmente un **processo non risiede completamente in memoria fisica**
 - il meccanismo di caricamento in memoria fisica delle parti del programma e la traduzione degli indirizzi è trasparente al programmatore.

Memoria virtuale e rilocalizzazione dinamica

- Avere degli indirizzi virtuali implica che ogni programma eseguibile generato dal linker sia in formato rilocabile, cioè con indirizzi che partono dall'indirizzo 0
- Inoltre, al momento dell'esecuzione il programma viene caricato (opportunamente) in memoria con indirizzi virtuali
- Il meccanismo di **rilocalizzazione dinamica** cioè di traduzione automatico degli indirizzi virtuali in indirizzi fisici, è **attivato durante l'esecuzione** del programma ad ogni riferimento a memoria ed è trasparente al programmatore, al compilatore e al linker
- Questa traduzione automatica consente il caricamento di un programma in una qualsiasi locazione della memoria fisica.

Principi base della paginazione

- La memoria virtuale viene suddivisa in porzioni di lunghezza fissa dette **pagine virtuali** (ad esempio di 4 K Byte).
- La memoria fisica viene anch'essa suddivisa in **pagine fisiche della stessa dimensione delle pagine virtuali**.
- Le pagine virtuali possono essere **residenti** o **non residenti** in memoria fisica.
- Le pagine virtuali da eseguire vengono caricate in memoria in altrettante pagine fisiche, scelte arbitrariamente e **non necessariamente contigue**.
- La paginazione ha l'effetto di ridurre il fenomeno della **frammentazione della memoria** cioè della presenza di zone di memoria libere (e piccole) inframmezzate a zone di memoria utilizzate.
- È possibile gestire facilmente **la crescita dinamica della memoria di un processo durante l'esecuzione**.



Paginazione: pagine virtuali

- Lo spazio di indirizzamento virtuale di ogni programma è **lineare (indirizzi virtuali contigui)** ma suddiviso in un numero intero di **pagine virtuali di dimensione fissa**:
 - Tipiche dimensioni di pagina: da 512 Byte a 64 K Byte (esempio: 4 K Byte)
- L' **indirizzo virtuale** può essere visto come composto da:

Numero Pagina Virtuale (NPV)	Spiazzamento (offset) nella pagina
------------------------------	------------------------------------

- Esempio: spazio di indirizzamento virtuale 64 K Byte (16 bit di indirizzo virtuale) con pagine da 512 Byte (9 bit di indirizzo per offset nella pagina)
 - Numero pagine = $64 \text{ K Byte} / 512 \text{ Byte} = 216 / 29 \text{ pagine} = 27 = 128 \text{ pagine virtuali} \rightarrow 7 \text{ bit di indirizzo per NPV}$



Paginazione: pagine fisiche

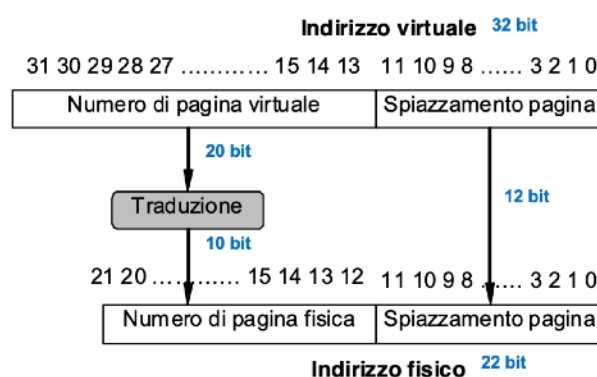
- Lo spazio di indirizzamento fisico (ossia della memoria centrale) viene suddiviso in un numero intero di **pagine fisiche** di uguale dimensione delle pagine virtuali
- Ogni pagina della memoria fisica può quindi contenere **esattamente** una pagina dello spazio di indirizzamento virtuale
- Essendo la dimensione di una pagina virtuale uguale alla dimensione di una pagina fisica
 \rightarrow **Offset nella pagina virtuale è uguale all'offset nella pagina fisica**
- L'indirizzo fisico è dato da:

Numero Pagina Fisica (NPF)	Spiazzamento (offset) nella pagina
----------------------------	------------------------------------

Esempio 1

- Spazio di indirizzamento virtuale **4 G Byte** $\rightarrow 2^{32}$ indirizzi virtuali $\rightarrow 32$ bit di indirizzo virtuale
- Dimensione di pagina: **4 K Byte** $\rightarrow 2^{12}$ indirizzi per lo **spiazzamento (offset)** $\rightarrow 12$ bit di indirizzo per offset
- Numero di pagine virtuali = $2^{32} / 2^{12} = 2^{20}$ pagine virtuali $\rightarrow 20$ bit di indirizzo per NPV
- Spazio di indirizzamento fisico **4 M Byte** $\rightarrow 2^{22}$ indirizzi fisici $\rightarrow 22$ bit di indirizzo fisico
- Numero di pagine fisiche = $2^{22} / 2^{12} = 2^{10}$ pagine fisiche $\rightarrow 10$ bit di indirizzo per NPF

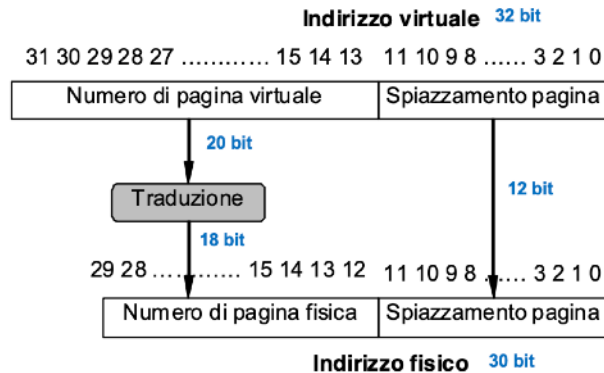
Traduzione da indirizzo virtuale in indirizzo fisico



Esempio 2

- Spazio di indirizzamento virtuale **4 G Byte** → 2^{32} indirizzi virtuali → **32 bit** di indirizzo virtuale
- Dimensione di pagina: **4 K Byte** → 2^{12} indirizzi per lo *spiazzamento (offset)* → **12 bit** di indirizzo per offset
- Numero di pagine virtuali = $2^{32} / 2^{12} = 2^{20}$ pagine virtuali → **20 bit** di indirizzo per NPV
- Spazio di indirizzamento fisico **1 G Byte** → 2^{30} indirizzi fisici → **30 bit** di indirizzo fisico
- Numero di pagine fisiche = $2^{30} / 2^{12} = 2^{18}$ pagine fisiche → **18 bit** di indirizzo per NPF

Traduzione da indirizzo virtuale a indirizzo fisico



Paginazione: Tabella delle Pagine

- Il meccanismo di traduzione o corrispondenza tra pagine virtuali e pagine fisiche è realizzata da una **Tabella delle Pagine (TP)** associata ad ogni processo
 - Esiste una **Tabella delle Pagine** per ogni processo in esecuzione
 - La Tabella delle Pagine fa parte del salvataggio di contesto di un processo (e concettualmente appartiene al descrittore di processo)
- La **TP** deve contenere una **riga per ogni pagina virtuale del processo** che riporta il numero della pagina fisica corrispondente
- In questa ipotesi, il numero di pagina virtuale (NPV) può essere utilizzato come **indice** (indirizzo) nella tabella delle pagine del processo (**tabella indicizzata**)
- Oppure la tabella può essere **associativa** sul contenuto del campo **NPV** associato al corrispondente **NPF** oppure più precisamente sulla coppia **(PID, NPV)** associata al corrispondente **NPF**.

Memoria virtuale di P

Numero di pagina	Contenuto delle pagine
0x00000	AAAA
0x00001	BBBB
0x00002	CCCC
0x00003	DDDD

Tabella Pagine di P

NPV	NPF
0x00000	0x00004
0x00001	0x00005
0x00002	0x00006
0x00003	0x00007

MEMORIA FISICA

Numero di pagina	Contenuto delle pagine
0x00000	S.O.
0x00001	S.O.
0x00002	S.O.
0x00003	S.O.
0x00004	AAAA
0x00005	BBBB
0x00006	CCCC
0x00007	DDDD
0x00008	RRRR
0x00009	SSSS
0x0000A	TTTT
0x0000B	UUUU
0x0000C	VVVV
0x0000D	non usata
0x0000E	non usata
0x0000F	non usata

Memoria virtuale di Q

Numero di pagina	Contenuto delle pagine
0x00000	RRRR
0x00001	SSSS
0x00002	TTTT
0x00003	UUUU
0x00004	VVVV

Tabella Pagine di Q

NPV	NPF
0x00000	0x00008
0x00001	0x00009
0x00002	0x0000A
0x00003	0x0000B
0x00004	0x0000C

Condivisione delle pagine

- I diversi processi possono **condividere delle pagine** (tipicamente di codice o di libreria, ma anche di dati, tramite opportuni meccanismi messi a disposizione dal S.O.)
 - le pagine condivise da diversi processi sono **presenti una sola volta nella memoria fisica**
- Per ogni pagina condivisa da più processi, esiste una riga (cioè un **NPV**) nella corrispondente tabella delle pagine del processo
- I valori di **NPF** relativi alle pagine condivise sono identici nelle righe delle tabelle delle pagine di ogni processo che le condivide.
- Esempio:

Tabella Pagine di P

NPV	NPF
0x00000	0x00004
0x00001	0x00005
0x00002	0x00006
0x00003	0x00007

Tabella Pagine di Q

NPV	NPF
0x00000	0x00004
0x00001	0x00005
0x00002	0x0000A
0x00003	0x0000B
0x00004	0x0000C

Protezione delle pagine

- Il meccanismo di paginazione consente di rilevare, durante l'esecuzione, un accesso a zone di memoria che non appartengono allo spazio di indirizzamento virtuale del processo in esecuzione.
 - questo avviene quando viene generato un numero di pagina virtuale (NPV) che non esiste nella tabella (completa) delle pagine del processo. In questo caso, la MMU (o il Sistema Operativo stesso) generano un **interrupt di violazione di memoria**
- È possibile associare ad ogni pagina virtuale di un processo alcuni **bit di protezione**, che definiscono le modalità di accesso (diritti) consentite:
 - Lettura (**R**)
 - Scrittura (**W**)
 - Esecuzione (**X**) per pagine di codice (istruzioni da leggere ed eseguire)
- I diritti di accesso risultano particolarmente significativi nel caso di **pagine condivise**
- In presenza di violazioni di un diritto di accesso si genera un **interrupt di violazione di memoria**

Gestione delle pagine residenti e non residenti in memoria

- Talvolta il numero di pagine virtuali dei processi eccede il numero di pagine disponibili nella memoria fisica.
- Non tutte le pagine virtuali di un processo sono **residenti** in memoria fisica e alcune devono risiedere su disco.
- Se l'accesso alla memoria appartiene ad una pagina **non residente** in memoria → si genera un **interrupt di segnalazione di errore (page fault)** e il processo viene sospeso in attesa che la pagina venga caricata da disco in memoria fisica.



Swap file

La memoria virtuale di un processo non è contenuta completamente nella memoria fisica durante l'esecuzione

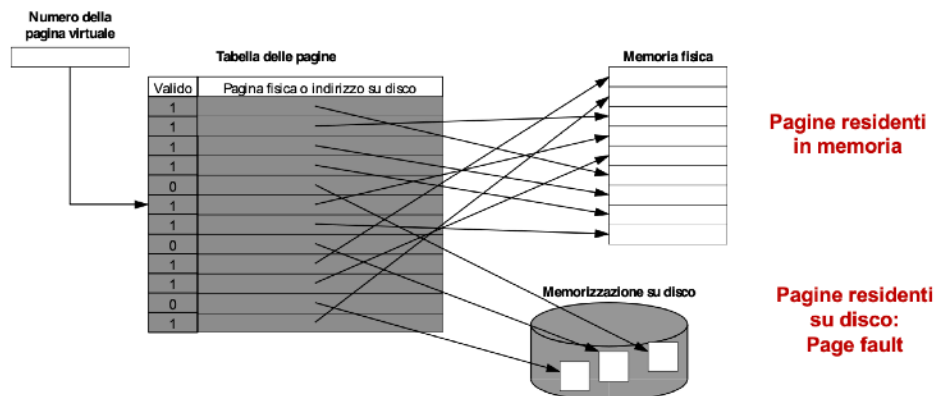
→ è la parte non contenuta in memoria fisica deve risiedere su disco

Necessità di spazio su disco per ogni processo per contenere (tutte) le pagine

virtuali: **swap file** (gestito dal SO)

- alcune pagine virtuali hanno già una immagine su disco, ad es. le pagine di codice presenti nel file eseguibile che non vengono mai modificate
- alcune pagine dati esistono solo nella memoria fisica

Tabella delle pagine: pagine residenti in memoria e su disco

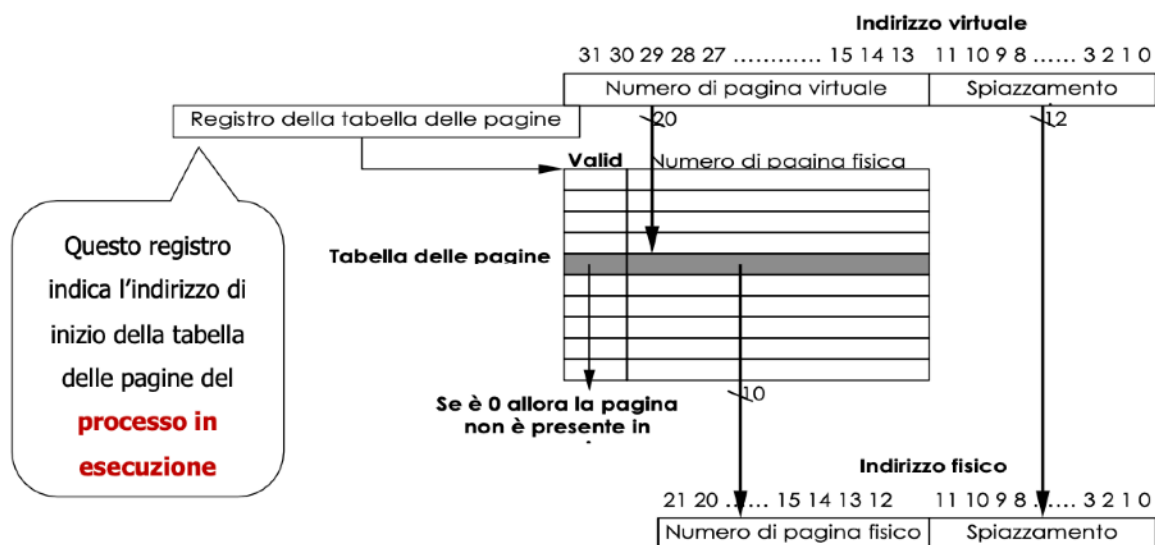


Bit di validità

Nella Tabella delle Pagine si deve indicare se la pagina richiesta è residente in memoria fisica introducendo il **bit di validità (valid bit)**:

- **Se il bit è a 1 la pagina è residente in memoria:** occorre leggere in tabella il numero di pagina fisica e lo si affianca come bit più significativi all'offset ottenendo l'indirizzo fisico di memoria.
- **Se il bit è a 0 si verifica un errore di pagina (page fault)** e il processo viene sospeso in attesa che la pagina sia caricata da disco. Se necessario, una pagina già residente viene scaricata su disco per liberare una pagina fisica che possa contenere la nuova pagina virtuale.

Il bit di validità indica se la pagina corrispondente è residente in memoria

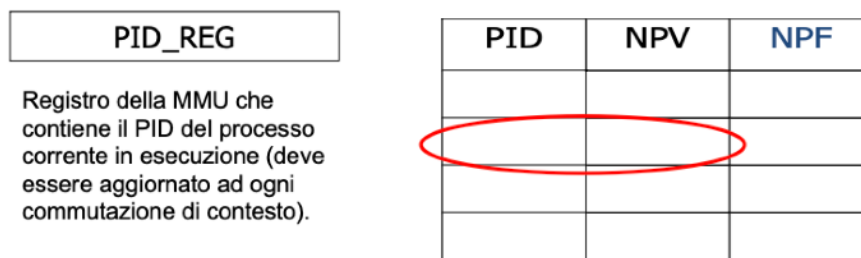


Memory Management Unit

- **Problema:** La Tabella delle Pagine per processo è una struttura dati del SO residente in memoria che può assumere dimensioni anche molto grandi (che dipendono dal numero delle pagine per processo che variano dinamicamente).
- In pratica ogni accesso a memoria richiederà **2 accessi**:
 1. Un accesso alla Tabella delle Pagine per ottenere l'indirizzo fisico;
 2. Un accesso all'indirizzo fisico del dato

→ *Necessari dei meccanismi HW per **accelerare** la traduzione tra indirizzi virtuali e indirizzi fisici delle pagine (NPV e NPF).*

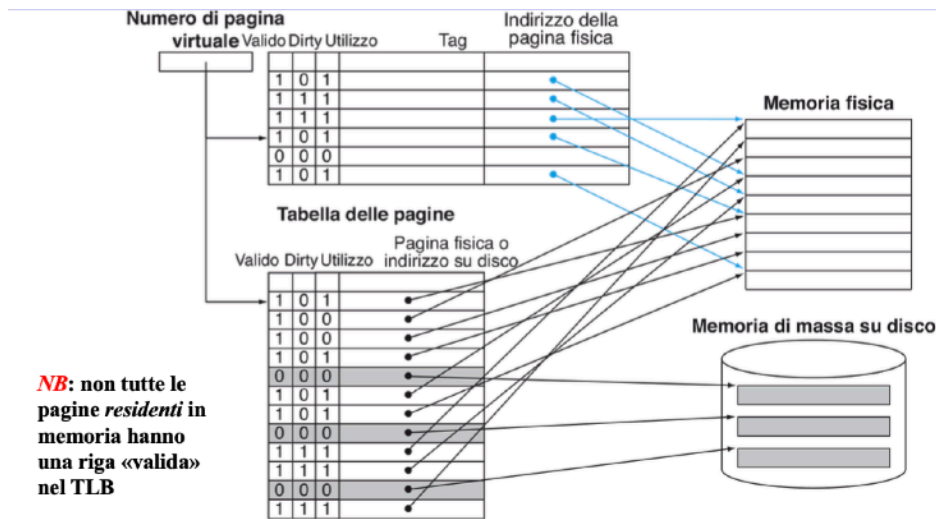
- Per accelerare la traduzione da indirizzi virtuali a fisici si introduce un'unità hardware specializzata: la **Memory Management Unit (MMU)**.
- La **MMU** contiene una **memoria cache (Translation Lookaside Buffer)** che contiene una copia di una parte della tabella delle pagine di un processo (e cioè gli indirizzi delle **pagine recentemente utilizzate**).
- Essendo il **TLB** una cache della tabella delle pagine sarà molto veloce e di dimensioni ridotte:
 - Il valore di NPV non può quindi essere utilizzato come indice del TLB ma sarà memorizzato come tag (**cache completamente associativa**);
 - per distinguere le pagine virtuali di ciascun processo, un tag della tabella deve contenere il **PID** del processo stesso



- Il **TLB** viene realizzato come **cache completamente associativa**:
 - la selezione di una riga della memoria **non** avviene tramite indirizzo (indice) ma come **associazione sul contenuto** di opportuni campi (tag) della riga stessa
 - il tag utilizzato è la **coppia (PID, NPV)** per ottenere il numero di pagina fisica corrispondente (NPF).

TLB

- Il TLB è una memoria cache assegnata al processo in esecuzione per tener traccia delle traduzioni (NPV, NPF) utilizzate più di recente: Lookaside Translation Buffer (TLB)
- Contenuto del TLB:
 - **Tag o etichetta:** NPV
 - **Dati:** NPF
 - **Valid bit (bit di validità)** → indica se la riga della tabella è valida
 - **Dirty bit** → indica se la pagina fisica è stata modificata
 - **Access bit (bit di accesso)** → indica se la pagina è stata utilizzata di recente, serve per attuare la politica di sostituzione LRU (approssimata)



- Il **TLB** è assegnato al processo in esecuzione => deve essere **flushed** (tramite bit di validità) ad ogni commutazione di contesto tra processi.
- Il flush del **TLB** impone la necessità di salvare lo stato del **Dirty bit** (che altrimenti andrebbe perso) della pagina nella TP del processo.
- Poiché il **TLB** contiene una copia solo una parte della TP di un processo → è possibile che avvenga l'accesso ad una pagina virtuale non presente nella tabella associativa (**vero TLB Miss**).
- Per ridurre la probabilità di **TLB Miss**, è necessario che la dimensione del TLB sia almeno uguale al numero **R** di pagine del processo residenti in memoria (in questo caso un **TLB Miss** corrisponderebbe ad un **page fault**).

Accesso a memoria fisica tramite TLB: funzionamento

1. Richiesta di traduzione indirizzo virtuale
2. Ricerca **NPV** in **TLB**
 - se **esiste** (**TLB hit**): legge NPF dal TLB e costruisce l'indirizzo fisico aggiungendo l'offset → pone Access bit (bit di utilizzo) a 1 e Dirty bit a 1 solo se si tratta di una scrittura
 - se **non esiste** (**TLB miss**):
 1. **Page Fault** (se bit di validità = 0 nella TP) la pagina **NON** risiede in memoria → il processo viene messo in attesa che la pagina venga caricata da disco.
 2. **Vero TLB Miss** (se bit di validità = 1 nella TP) la pagina è residente in memoria fisica → carica **NPF** nel TLB dalla TP presente in memoria e si ripete l'accesso

Gestione delle pagine virtuali non residenti in memoria

Durante l'esecuzione di un processo

- un certo numero di pagine virtuali è caricato in memoria in altrettante pagine fisiche (**pagine residenti in memoria fisica**)
- durante un accesso a memoria, in caso di **page fault**, tramite un interrupt il controllo passa al sistema operativo
- il processo in esecuzione va in attesa e il sistema operativo deve
 - rintracciare su disco la pagina virtuale richiesta: il S.O. utilizza per questo le TP "complete" che contengono, per le pagine fuori memoria, il riferimento alla posizione su disco
 - trovare spazio in memoria per caricare (allocare) la pagina richiesta. Questa operazione può causare la deallocazione da memoria di un'altra pagina aggiornando la TP (e TLB) dei processi coinvolti
 - caricare la pagina da disco a memoria
 - far rieseguire l'istruzione che aveva generato il page fault

PROGETTO DI UN SISTEMA DI MEMORIA VIRTUALE

1. Strategia di caricamento delle pagine

- a. su richiesta (on demand)
 - b. working set
2. **Politica di sostituzione:** nel caso di page fault, se la memoria fisica non ha pagine disponibili, occorre decidere come scegliere la pagina da sostituire (deallocare) per far posto alla nuova pagina
3. **Dimensione delle pagine** (unità trasferita tra memoria principale e memoria secondaria)

1.a Caricamento di pagine su richiesta

- Nel caso di esecuzione di un nuovo processo:
 - La tabella delle pagine del processo ha tutti i Valid bit a 0 (nessuna pagina si trova in memoria) e anche il TLB ha tutti i Valid bit a 0.
 - Quando la CPU cerca di accedere alla prima istruzione del codice si verifica un **page fault** e la prima pagina di codice viene caricata in memoria e registrata nella TP e nel TLB.
 - Ogni volta che la CPU genera un indirizzo di una pagina (codice o dati) non ancora allocata in memoria, si verifica un errore di pagina (**page fault**) la pagina viene caricata in memoria fisica e registrata nella TP e nel TLB.
- Metodo chiamato di **paginazione su richiesta (on demand paging)**: *pagine allocate in memoria fisica solo quando necessarie.*

1.b Caricamento di pagine basato su Working Set

- **Metodo basato sul principio di località degli accessi**
 - La maggior parte dei programmi non accede al suo spazio di indirizzamento in modo uniforme ma gli accessi tendono a raggrupparsi in un numero limitato di pagine
- **Working set di ordine k di un programma:**
 - insieme di pagine referenziate durante gli ultimi k accessi alla memoria.
Se **k** è sufficientemente grande, il Working set di un programma varia molto lentamente per il principio di località
 - Se si mantengono in memoria le **k** pagine accedute più recentemente (Working set di ordine **k**), per il principio di località è probabile che il prossimo accesso sia all'interno di tali pagine, cioè che non si verifichi un page fault.
- **Il numero R di pagine residenti in memoria fisica di ogni processo è ottenuto da una stima del Working set (in configurazione).**
- La scelta del parametro **R** determina la frequenza dei page fault:
 - Se scelgo **R** grande: avrò pochi page fault per processo
 - Se scelgo **R** piccolo: possono convivere molti processi in memoria ma avrò più page fault per processo
- A partire dall'inizio dell'esecuzione di un programma, tramite **on demand paging**, è possibile caricare il Working set in memoria (dopo **R** page fault)

2. Politica di sostituzione delle pagine

- **Casuale**
- **Least recently used (LRU):** il SO cerca di sostituire la pagina meno utile nel prossimo futuro, ossia quella **utilizzata meno di recente**, che ha minor probabilità di appartenere al working set attuale (principio di località)

- **FIFO (First In First Out)**: si sostituisce sempre la pagina *caricata meno di recente* indipendentemente da quando si è fatto riferimento a questa pagina

2. Politica di sostituzione delle pagine e bit di controllo

Per gestire in modo efficiente la **scelta della pagina fisica** in cui caricare la pagina che ha causato il page fault (**politica di sostituzione della pagina**), ogni riga della tabella delle pagine deve prevedere altri **2 bit di controllo**:

- **bit di accesso**: azzerato quando la pagina viene posta in memoria, viene **posto a 1** ogni volta che si accede alla pagina fisica corrispondente e azzerato periodicamente dal SO (serve per aggiornare le informazioni necessarie alla gestione della sostituzione della pagine con algoritmo LRU – Least Recently Used)
- **bit di modifica**: azzerato quando la pagina viene posta in memoria, viene posto a 1 quando si accede in **scrittura** ad una parola della pagina fisica corrispondente (serve per gestire la ricopiatura della pagina in memoria su disco, quando questa viene scaricata). Ovviamente le pagine che al momento della deallocazione hanno il bit di modifica a 0 non richiedono la ricopiatura su disco perché non sono state modificate.

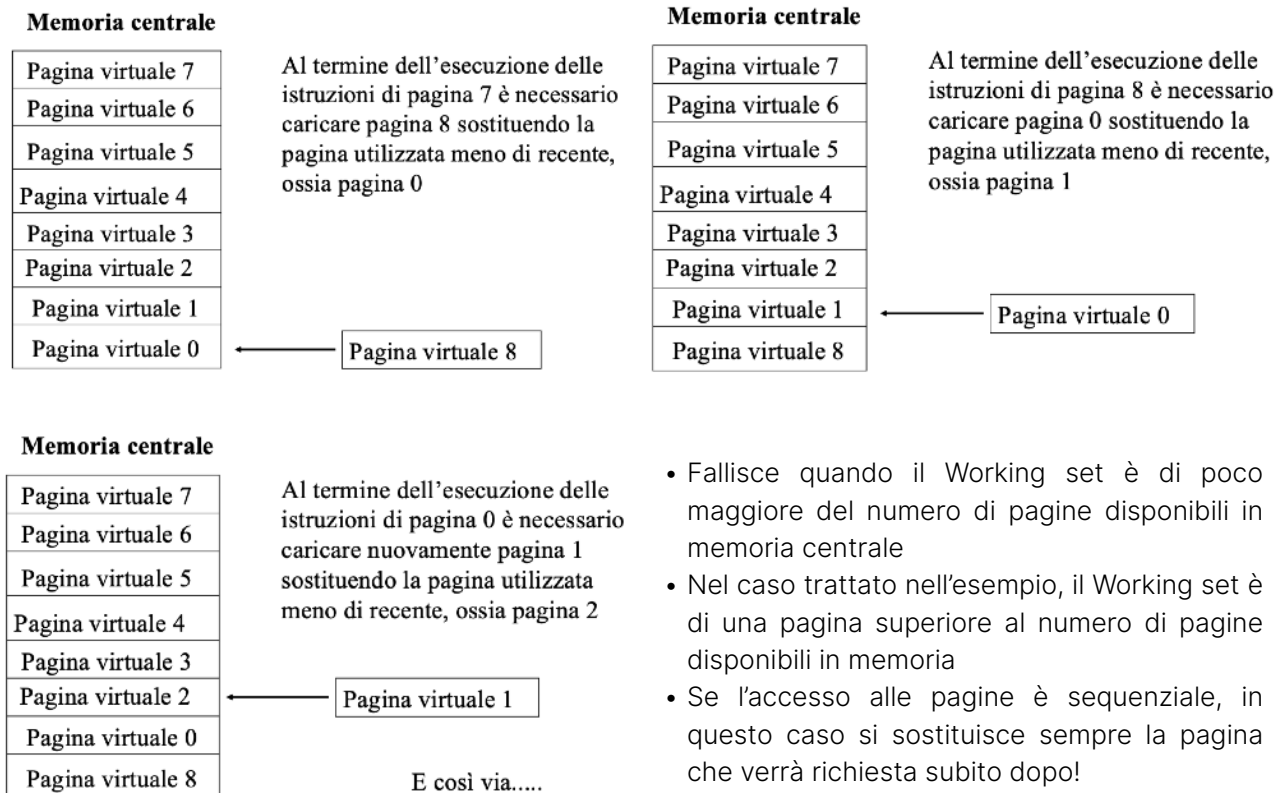
Algoritmo LRU (Least Recently Used)

- Utilizza bit di controllo che riportano le informazioni sugli **accessi alle pagine (lettura/scrittura)**
- La gestione più semplice prevede un **bit di accesso** per ogni riga della tabella delle pagine:
 - il bit viene **posto a 1** quando la pagina viene acceduta e azzerato periodicamente da SO
 - in corrispondenza dell'azzeramento periodico, il SO controlla tali bit e incrementa una **variabile di conteggio interna (grado di non accesso alla pagina)**, per tutte le pagine con bit di accesso a 0
 - viene sostituita la pagina con bit di accesso a 0 e valore di conteggio più alto (grado più alto di non-accesso alla pagina)
- Ha buone prestazioni in media, ma esistono situazioni particolari in cui l'algoritmo LRU ha prestazioni pessime (ad esempio nel caso di working set di poco maggiore rispetto al numero di pagine disponibili in memoria fisica).

esempio di applicazione

- Esecuzione di un ciclo che si estende su 9 pagine virtuali
- La memoria può contenere solo 8 pagine
- Si assuma che le prime 8 pagine virtuali (da 0 a 7) siano in memoria.
- Al termine dell'esecuzione della pagina 7 è necessario scegliere una pagina da scaricare per caricare la pagina 8
- Scelta della pagina da sostituire mediante algoritmo LRU
- La pagina utilizzata meno di recente è la pagina 0 → viene sostituita con la pagina 8
- Al termine dell'esecuzione delle istruzioni che appartengono alla pagina 8 è necessario ricominciare l'esecuzione del ciclo da pagina 0, ossia dalla pagina appena sostituita → errore di pagina!
- La pagina 0 andrà a sostituire la pagina 1, utilizzata meno di recente: ma questa sarà la pagina da ricaricare subito dopo!!

Fallimento dell'algoritmo LRU:



- Fallisce quando il Working set è di poco maggiore del numero di pagine disponibili in memoria centrale
- Nel caso trattato nell'esempio, il Working set è di una pagina superiore al numero di pagine disponibili in memoria
- Se l'accesso alle pagine è sequenziale, in questo caso si sostituisce sempre la pagina che verrà richiesta subito dopo!

Algoritmo di sostituzione FIFO





- Si mantiene l'informazione del momento in cui la pagina è stata **caricata** in memoria
- Si associa un contatore ad ogni pagina **fisica** e inizialmente i contatori sono **posti a 0**
 - al termine della gestione di un errore di pagina i contatori delle pagine fisiche presenti in memoria sono incrementati di **1**
 - il contatore della pagina appena caricata in memoria viene posto a **0**
- In caso di errore di pagina, la **pagina da sostituire** (dealloca) è quella con il **valore del contatore più alto**
 - Il valore di contatore più alto significa che si tratta della pagina che ha assistito a più errori di pagina e quindi si trova in memoria da più tempo rispetto alle altre
- Anche l'algoritmo FIFO non ha un buon comportamento nel caso in cui il working set è di poco maggiore del numero di pagine disponibili in memoria
- Non ci sono algoritmi che danno buoni risultati in questo caso
- Un programma che genera frequenti errori di pagina si dice in **trashing**

Sostituzione e scrittura di una pagina in memoria

- Nel caso di **scrittura**, la modifica avviene **solo in memoria fisica** e non viene riportata immediatamente nella pagina presente su disco (i tempi di accesso al disco molto elevati rispetto ai tempi di accesso alla memoria)
- In caso di **deallocazione** di una pagina in memoria fisica, è necessario verificare se la pagina è stata o meno modificata (bit di modifica)
 - Solo se la pagina da dealloca è stata modificata → è necessario copiarla su disco prima di sostituirla con un'altra pagina
 - Le pagine contenenti istruzioni non possono essere modificate quindi non devono mai essere copiate su disco.

- Le pagine di un processo contenenti dati e/o allocate dinamicamente, quando vengono deallocate, devono essere salvate nello **swap file**, nel quale viene salvata l'immagine delle pagine quando vengono scaricate
- Nel TLB si associa un bit (**bit di modifica o dirty bit**) ad ogni pagina fisica per indicare se la pagina è stata o meno modificata.
- Il bit di modifica viene azzerato quando la pagina viene allocata in memoria e **posto a 1** ogni volta che viene scritta una parola di tale pagina.
- Quando una pagina deve essere deallocata, il SO verifica il bit di modifica per stabilire se la pagina debba essere copiata su disco o no.

3. Dimensionamento delle pagine: frammentazione

- Ad ogni programma viene sempre assegnato un numero intero di pagine, però possono rimanere aree di memoria assegnate ma non occupate:
 - Programma + dati = 26.000 Byte
 - Pagine di 4kByte (4096 Byte)
 - Numero di pagine: $26000/4096 \approx 6.34$
→ necessarie 7 pagine, ma l'ultima pagina è occupata per 1424 Byte e verranno sprecati 2672 Byte
- Per pagine di n Byte la quantità media di spazio di memoria sprecato nell'ultima pagina sarà di n/2 Byte, questo porterebbe a ridurre le dimensioni della pagina per minimizzare lo spreco
- Pagine piccole →  numero di pagine
-  dimensioni della tabella delle pagine
-  dimensioni della MMU
-  trasferimenti da disco

3. Dimensionamento delle pagine: trasferimenti da disco

Pagine piccole riducono l'efficienza dei trasferimenti dal disco:

- Pagine più piccole richiedono più trasferimenti
- **Tempo di ricerca** della pagina su disco + ritardo di rotazione ~ 10 msec
- Tempo di trasferimento di una pagina di 1KByte a 10MB/sec richiede circa 0.1 msec
- Il peso del tempo di trasferimento è di due ordini di grandezza inferiore al tempo di ricerca

Recap: Il concetto di memoria virtuale

- Il concetto di memoria virtuale nasce dalla necessità di separare i concetti di:
 - **spazio di indirizzamento**
 - **dimensione effettiva della memoria fisica**
- Lo **spazio di indirizzamento** è definito dal numero di parole indirizzabili e dipende esclusivamente dal numero di bit dell'indirizzo e non dal numero di parole di memoria fisica effettivamente disponibili
 - se N sono i bit di indirizzo, allora 2^N è il numero massimo di byte indirizzabili
- La **dimensione della memoria fisica** è pari al numero di Byte che la costituiscono (e può variare a seguito di espansione di memoria):
 - poiché la memoria fisica deve essere tutta indirizzabile, la sua dimensione è sempre minore o uguale al suo spazio di indirizzamento
 - ad esempio quando si espande la memoria su un PC, si aumenta la dimensione della memoria fisica, entro i limiti dello spazio di indirizzamento.

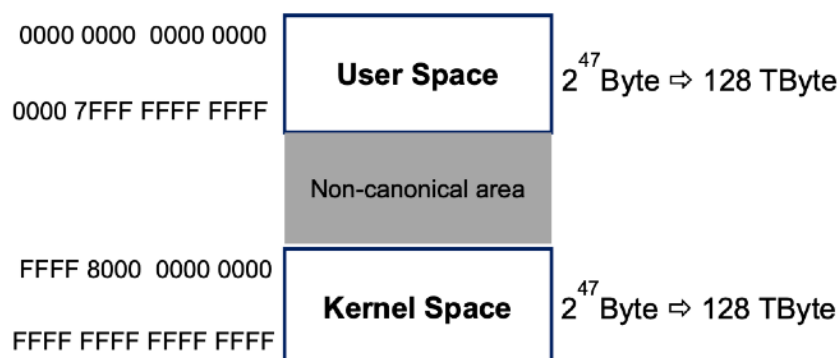
Recap: Memoria virtuale: indirizzi virtuali

- Gli indirizzi a cui fa riferimento il programma in esecuzione sono **indirizzi virtuali**
- Gli indirizzi virtuali sono quelli generati da linker (a partire dall'indirizzo 0): sono quindi **indirizzi rilocabili**
- Lo spazio di indirizzamento virtuale è quello definito dalla lunghezza degli indirizzi virtuali
- In un programma è possibile definire la **dimensione virtuale** (cioè espressa in termini di indirizzi virtuali):
 - **iniziale**: è quella calcolata dal linker dopo la generazione del codice eseguibile
 - **durante l'esecuzione**: la dimensione virtuale di un programma può variare in fase di esecuzione a causa della modifica delle dimensioni dello stack dovuta a chiamate a sottoprogrammi e a causa della modifica delle dimensioni dello heap dovuta alla gestione delle strutture dati dinamiche (allocazione dinamica della memoria).

Spazio di indirizzamento virtuale dell'architettura Intel x64

- Architettura x64 ha uno spazio di indirizzamento virtuale potenziale di 2^{64} Byte con indirizzi virtuali da 64 bit → 16 cifre esadecimali
- Attualmente lo spazio di indirizzamento virtuale utilizzabile è limitato a 2^{48} Byte → 256 TByte: indirizzi virtuali da 48 bit

Lo spazio di indirizzamento virtuale utilizzabile pari a 2^{48} Byte → 256 TByte **suddiviso in due sottospazi di modo U ed S ambedue da 2^{47} Byte → 128 TByte**



Paginazione nell'architettura Intel x64

- Nell'architettura x64, tutta la memoria (Kernel Space e User Space) è **paginata**, compresa la Tabella delle Pagine.
- Dimensione di pagina: **4K Byte** → **2¹² Byte** → **12 bit per lo spiazamento (offset)**
- Numero di pagine dello spazio di indirizzamento virtuale = $2^{48} / 2^{12} = 2^{36}$ pagine virtuali → **36 bit** per indicare il Numero di Pagina Virtuale (NPV)



- La **Tabella delle Pagine (TP)** per ogni processo è molto grande (2³⁶ pagine) ed è sempre residente in memoria fisica e deve mappare tutto lo spazio di indirizzamento del processo (User e Kernel Space) pari a 256 TByte

KERNEL SPACE

Struttura dello spazio di indirizzamento virtuale del kernel

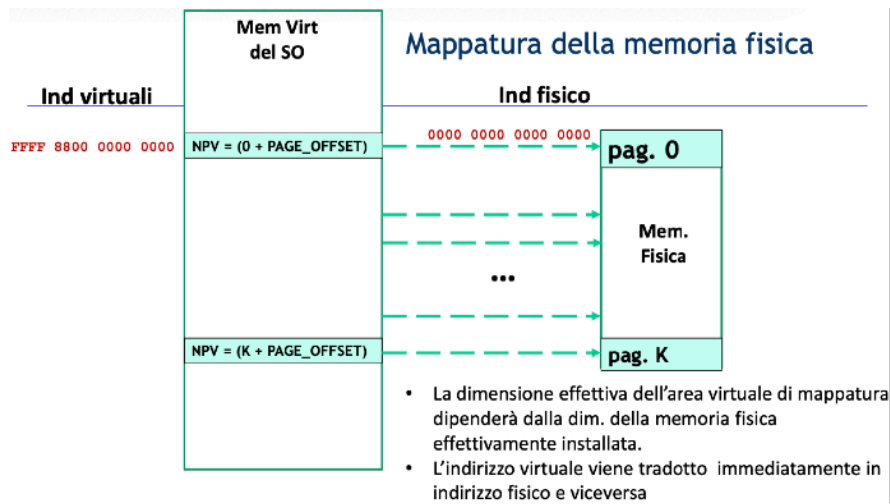


Lo spazio virtuale del kernel è suddiviso in **5 aree** di memoria virtuale principali:

- **Mappatura della memoria fisica:** 2⁴⁶ Byte = 64 TeraByte, (per permettere al codice del Kernel di accedere direttamente a indirizzi fisici)
- **Dati dinamici del kernel:** 32 TeraByte
- **Mappatura della memoria virtuale:** area utilizzata per ottimizzare particolari configurazioni discontinue della memoria (e non verrà trattata da noi) 1 TB
- **Moduli a caricamento dinamico:** 1,5 GigaByte
- **Codice e dati** del sistema operativo: 0,5 GigaByte (512 MegaByte)

Area	Costanti simboliche per indirizzi iniziali	Indirizzo iniziale	Dimensioni
Mappatura memoria fisica	PAGE_OFFSET	FFFF 8800 0000 0000	64TB
Dati dinamici del kernel	VMALLOC_START	FFFF C900 0000 0000	32TB
Mappatura della memoria virtuale	VMEMMAP_START	FFFF EA00 0000 0000	1TB
Moduli a caricamento dinamico	MODULES_VADDR	FFFF FFFF A000 0000	1,5GB
Codice e dati	START_KERNEL_MAP	FFFF FFFF 8000 0000	0,5GB

Nota: tra queste aree esistono spazi non utilizzati e lasciati per usi futuri



Mappatura del kernel space

- Il kernel space e la tabella delle pagine del kernel sono paginate ma senza ridondanza nell'allocazione delle pagine fisiche che sono condivise per tutti i processi.
- In pratica tutte le tabelle delle pagine del kernel di ogni processo puntano alla stessa (ed unica) struttura fisica condivisa relativa al kernel space.

USER SPACE

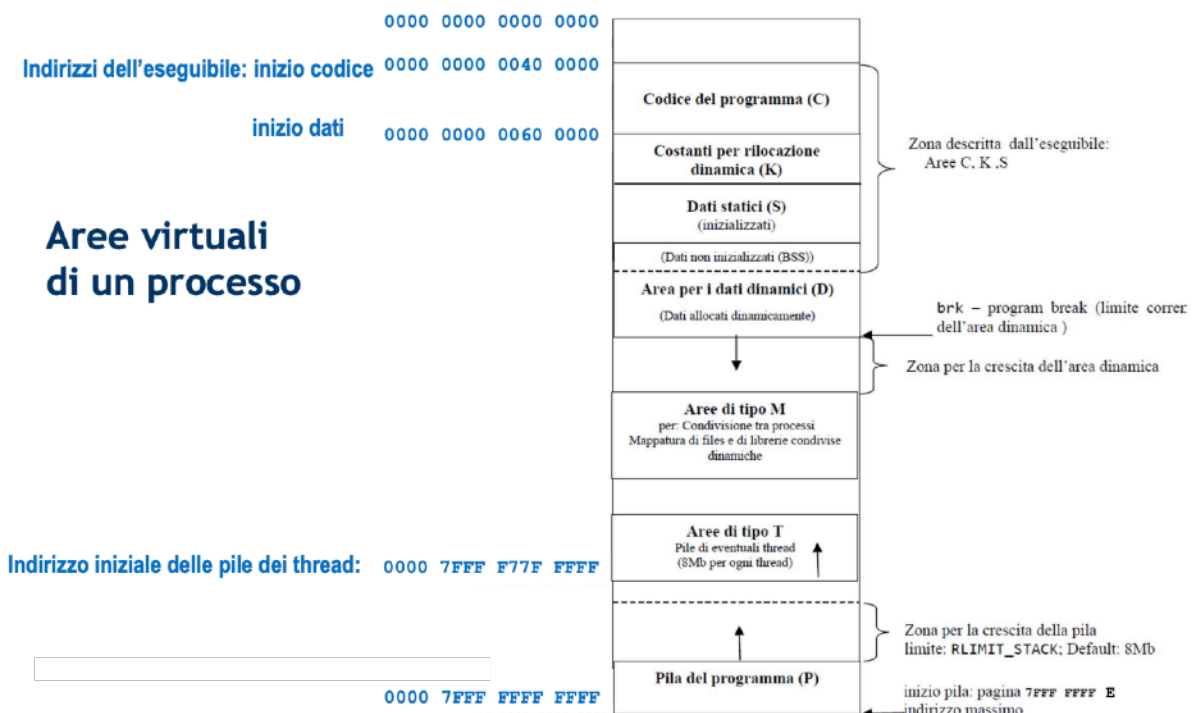
Struttura spazio di indirizzamento virtuale di un processo utente

- La memoria virtuale di un processo Linux non viene considerato un unico spazio di indirizzamento lineare ad uso indifferenziato, ma viene suddivisa in **aree di memoria virtuale (Virtual Memory Area)** indipendenti ed omogenee: **codice**, **dati statici**, **dati dinamici (heap)**, **memoria condivisa**, **librerie dinamiche e pila**.
- Vantaggi delle **VMA**:
 - Distinguere diversi diritti di accesso delle aree: **READ_ONLY (RO)**, **READ_WRITE (RW)**, **EXECUTE (X)**;
 - Consentire una crescita **dinamica** e indipendente di alcune aree come la pila e i dati dinamici;
 - Gestire un'area di memoria **condivisa** tra processi diversi e librerie dinamiche (shared libraries e dynamic linked libraries)
- Poiché la memoria virtuale si appoggia sul concetto di **paginazione**:
 - ogni **VMA** è costituita da un numero intero di pagine virtuali consecutive e con caratteristiche di accesso alla memoria omogenee
 - ogni **VMA** è delimitata da: **NPV_iniziale** e un **NPV_finale**
- Le aree dinamiche (come heap e pila) sono inizialmente piccole quando il programma viene lanciato in esecuzione, ma devono poter crescere **dinamicamente** durante l'esecuzione del programma.

Attenzione: Lo studio della memoria avverrà **disabilitando la ASLR (Address Space Layout Randomization)**

AREE VIRTUALI DI UN PROCESSO

- **Codice (C)**: contiene le istruzioni e anche le costanti definite all'interno del codice (ad esempio, le stringhe da passare alle printf)
- **Costanti per rilocalizzazione dinamica (K)**: area destinata a contenere dei parametri determinati in fase di link per il collegamento con le librerie dinamiche
- **Dati statici (S)**: area destinata a contenere i dati inizializzati allocati per tutta la durata di un programma
- **Dati dinamici (D)**: area destinata a contenere i dati allocati dinamicamente su richiesta (tramite `malloc`)
 - il limite corrente di quest'area è indicato dalla variabile **brk (Program break)** contenuta nel descrittore del processo
 - contiene anche gli eventuali dati non inizializzati definiti nell'eseguibile (**Block Started by Symbol - BSS**)
- **Aree per Memory-Mapped files (M)**: queste aree permettono di mappare un file (o librerie) su una porzione di memoria virtuale di un processo in modo che il file possa essere letto o scritto come se fosse un array di byte in memoria. Utilizzate per:
 - **Librerie dinamiche (shared libraries o dynamic linked libraries)**:
 - codice non viene incorporato staticamente nel programma eseguibile
 - vengono caricate in memoria durante l'esecuzione del programma in base alle esigenze del programma stesso
 - sono caricate mappando il loro file eseguibile su una o più aree virtuali di tipo M
 - possono essere condivise tra diversi programmi, che mappano lo stesso file della libreria su loro aree virtuali
 - **Memoria condivisa**: tramite la mappatura di un file su aree virtuali di diversi processi si ottiene la realizzazione di un meccanismo di condivisione della memoria tra tutti i processi che mappano tale file; in questo caso le aree potranno essere in lettura o lettura e scrittura.
- **Pile dei Thread (T)**: aree utilizzate per le pile dei thread
- **Pila (P)**: area di pila di modo U del processo



Crescita delle aree dati dinamiche

- Lo **stack cresce automaticamente** ad ogni attivazione di funzione, senza istruzioni o funzioni di sistema esplicite.
- La memoria dati dinamica (**heap**) **cresce tramite invocazione esplicita di servizi di sistema**: la funzione malloc del C può chiamare al suo interno i servizi di sistema brk() e sbrk().
- Il servizio di sistema LINUX che **incrementa** lo heap è la funzione:
 - * **void sbrk (int incremento)**
 - che incrementa l'area dati dinamici di un numero di pagine pari all'intero superiore di (incremento/dimensione della pagina) e restituisce l'indirizzo iniziale della nuova area;
 - Il servizio modifica l'indirizzo massimo dell'area di heap arrotondandolo ad un limite di pagina;
 - **sbrk(0)** restituisce il valore corrente della cima dell'area dati dinamica (heap).

VMA mappate su file

Una **VMA** può essere **mappata su un file** detto "**backing store**"

- il file è definito in vm_area_struct da:
 - **struct file * vm_file** → individua il file utilizzato come **backing store**
 - **unsigned long vm_pgoff** → page offset all'interno del file

Nota: Alle aree **C**, **K** e **S** viene associato il **file eseguibile .exe** come **backing store** con offset usato per indicare il punto dell'eseguibile in cui inizia il corrispondente segmento codice, dati, ecc.

esempio: Mappa di memoria di un processo in esecuzione cat /proc/NN/maps (NN è il pid del processo)

start-end page	perm	offset	device	i-node	file name
00400 - 00401 c0	r-xp	000000	08:01	394275	.../user.exe
00600 - 00601 k0	r--p	000000	08:01	394275	.../user.exe
00601 - 00602 s0	rw-p	001000	08:01	394275	.../user.exe
7fff f7a1 c - 7fff f7bd 0	r-xp	000000	08:01	271666	.../libc-2.15.so
7fff f7bd 0 - 7fff f7dc f	---p	1b4000	08:01	271666	.../libc-2.15.so
7fff f7dc f - 7fff f7dd 3	r--p	1b3000	08:01	271666	.../libc-2.15.so
7fff f7dd 3 - 7fff f7dd 5	rw-p	1b7000	08:01	271666	.../libc-2.15.so
...					
(altre aree M)					
...					
7fff fffd d-7fff ffff e	rw-p				[stack]

commenti: Al momento dell'exec di un programma eseguibile **LINUX costruisce la struttura delle aree virtuali del processo in base alla struttura definita dall'eseguibile**.

- l'area di pila è stata allocata con una dimensione iniziale di **34 pagine (144 Kbyte)** – l'area di pila è **anonima**, quindi non ha un file associato; l'indicazione [stack] è solo un commento aggiunto dal comando maps
- tutti i file coinvolti risiedono sullo stesso dispositivo 08:01, ma l'eseguibile del programma e quello della libreria libc sono diversi ed hanno quindi diverso **i-node**.
- l'area D è assente; viene creata solo in presenza di dati statici non inizializzati nell'eseguibile (BSS)

Algoritmo di gestione dei Page Fault

Ogni volta che la **MMU** genera un interrupt di Page Fault su un indirizzo virtuale viene attivata una routine detta **Page Fault Handler (PFH)**:

- `if` (NPV non appartiene alla memoria virtuale del processo)
Il processo viene abortito e viene segnalato **Segmentation Fault**
- `else if` (NPV appartiene alla memoria virtuale del processo ma l'accesso non è legittimo perché viola le protezioni)
Il processo viene abortito e viene segnalato **Segmentation Fault**
- `else if` (l'accesso è legittimo, ma NPV non è residente in memoria fisica)
Invoca la routine che deve caricare in memoria fisica la pagina virtuale NPV dal file di backing store (**demand paging**)

MECCANISMO GENERALE DELLE VMA

- le VMA sono classificate in base a 2 criteri:
 - Le VMA possono essere **mappate su file** oppure **anonime (ANONYMOUS)**
 - Le VMA possono essere di tipo **SHARED** oppure **PRIVATE**
- tutte le 4 combinazioni sono supportate in Linux, ma noi **non** considereremo la combinazione ANONYMOUS | SHARED:
 - VMA MAPPATE su FILE: PRIVATE or SHARED;**
 - VMA ANONIME (implicitamente PRIVATE)**

VMA MAPPATE su FILE	PRIVATE	SHARED
VMA ANONIME	PRIVATE	

→ consideriamo 3 tipi di VMA: **MAP_SHARED**, **MAP_PRIVATE**, **MAP_ANONIMOUS** (implicitamente PRIVATE)

Creazione di VMA e allocazione fisica delle pagine

- La creazione di una **VMA** consiste esclusivamente nell'allocazione dello spazio virtuale associato **senza alcuna allocazione di pagine fisiche**.
 - La creazione della **VMA** predispone anche la porzione di **Tabella delle Pagine (TP)** necessaria a rappresentare le pagine virtuali della **VMA** – inizialmente per ogni pagina virtuale la corrispondente **PTE** indicherà che la pagina non è allocata fisicamente.
- => Operazioni che **modificano esclusivamente la memoria virtuale** di un processo:
- creazione o eliminazione di VMA
 - estensione o riduzione di una VMA esistente
- L'allocazione delle pagine fisiche avviene solamente quando un processo **legge** o **scrive** una pagina virtuale (**demand paging**) e nella **PTE** associata alla pagina virtuale NPV sarà inserito il numero di pagina fisica allocata **NPF**
- => Operazioni che **allocano anche la memoria fisica**:
- lettura e scrittura

mmap – creazione di VMA

```
#include <sys/mmap.h>
void *mmap(void *addr, size_t length, int prot, int flags, int fd, off_t offset);
```

mmap restituisce il puntatore all'area allocata

- addr**: permette di suggerire l'indirizzo virtuale iniziale dell'area (cioè il SO sceglierà l'area il più vicino possibile all'indirizzo suggerito); – se non specificato, il SO assegna un indirizzo autonomamente;
- length** dimensione dell'area;
- prot** protezione: **PROT_READ**, **PROT_WRITE**, **PROT_EXEC**

- **flags** con 3 opzioni: **MAP_SHARED** , **MAP_PRIVATE**, **MAP_ANONIMOUS**
- **fd** descrittore del file da mappare
- **offset** indica la prima pagina dell'area all'interno del file (multiplo di **PAGESIZE**)

esempio di uso mmap per creazione di VMA v1 di un processo P mappata su file F

```
#include <sys/mmap.h>
#define PAGESIZE 1024*4
char * base;
fd = open (".F", O_RDWR);      // apre il file F
base = mmap(NULL, PAGESIZE*3, PROT_READ, MAP_SHARED, fd, PAGESIZE);
```

Nota: La differenza di comportamento tra le aree di tipo **MAP_SHARED** e **MAP_PRIVATE** emerge solo nelle operazioni di **scrittura**; pertanto negli esempi relativi ai casi di sola **lettura**, quanto mostrato con riferimento ad aree di tipo **MAP_SHARED** sarebbe valido anche se le aree fossero state dichiarate **VMA_PRIVATE**.

Aree mappate su file - sola lettura

- Un file in Linux è considerato una sequenza di byte; considerare un file diviso in pagine è solo un modo di indirizzare le posizioni dei byte all'interno di una pagina;
- **Esempio: VMA v1** di un processo **P** mappata su un file **F** a partire dalla seconda pagina (pagina 1) – si noti che le pagine della VMA e le corrispondenti pagine del file devono essere **consecutive**
 - **vm_file** indica il descrittore del file;
 - **vm_pgoff** indica il **page offset** cioè la prima pagina della VMA all'interno del file



Esempio 1 - lettura VMA SHARED create da due processi e mappate sullo stesso file

```
#define PAGESIZE 1024*4
char * base;
unsigned long      mapaddress1 = 0x100000000;
unsigned long      mapaddress2 = 0x200000000;

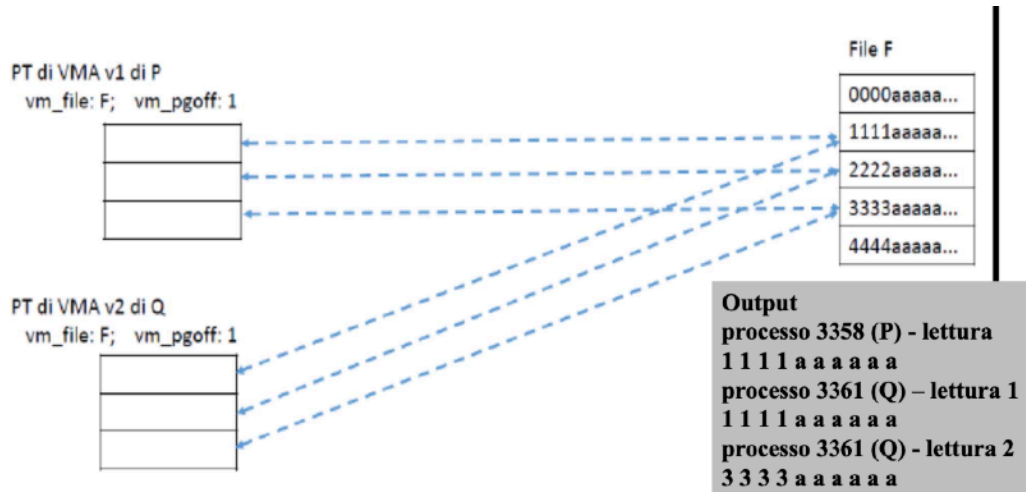
/* Creati due processi con VMA SHARED a indirizzi virtuali diversi
mappate sullo stesso file condiviso fd con offset = PAGESIZE.
Ambedue i processi P e Q leggono la prima pagina.
Il processo Q legge anche la terza pagina */

pid = fork();
if (pid == 0){          // processo figlio P
    fd = open(".F", O_RDWR);
    base = mmap(mapaddress1, PAGESIZE*3, PROT_READ, MAP_SHARED, fd, PAGESIZE);
    // lettura (10 caratteri da ind. base) nella prima pagina
}
```

```

...
pid = fork();
if (pid == 0){           //processo figlio Q
    fd = open("./F", O_RDWR);
    base = mmap(mapaddress2, PAGESIZE*3, PROT_READ, MAP_SHARED, fd, PAGESIZE);
    // lettura 1 (10 caratteri da ind. base) nella prima pagina
    // lettura 2 (10 caratteri da ind. base + 2*PAGESIZE) in terza pag.
}

```



Mappe dei due processi:

```

00400000-00401000  r-xp  00000000  08:01  396306  .../user.exe
00601000-00602000  r--p  00001000  08:01  396306  .../user.exe
00602000-00603000  rw-p  00002000  08:01  396306  .../user.exe
10000000-100003000 rw-s  00001000  08:01  396260  .../F
...

```

a) Processo P

```

00400000-00401000  r-xp  00000000  08:01  396306  .../user.exe
00601000-00602000  r--p  00001000  08:01  396306  .../user.exe
00602000-00603000  rw-p  00002000  08:01  396306  .../user.exe
20000000-200003000 rw-s  00001000  08:01  396260  .../F
...

```

b) Processo Q

Le TP (parziali) dei due processi:

```

[ 2792.343262] VMA start address 000100000000 ===== size: 3
[ 2792.343263] 000100000 :: 000071816 :: P,R
[ 2792.343264] 000100001 :: 000000000 :: ,
[ 2792.343265] 000100002 :: 000000000 :: ,

```

a) Processo P

```

[ 2792.347357] VMA start address 000200000000 ===== size: 3
[ 2792.347358] 000200000 :: 000071816 :: P,R
[ 2792.347359] 000200001 :: 000000000 :: ,
[ 2792.347360] 000200002 :: 000071818 :: P,R

```

b) Processo Q

La prima pagina virtuale è condivisa fisicamente (stesso NPF)

→ Esiste quindi un meccanismo che ha permesso al processo Q di accorgersi che la prima pagina era già stata caricata in memoria fisica

→ La terza pagina è allocata in memoria fisica solo per il processo Q

PAGE CACHE

- La **Page Cache** è un meccanismo per evitare che un processo rilegga da disco una pagina già caricata in memoria fisica.
 - **Vantaggio:** risparmio nell'allocazione di memoria fisica e del tempo necessario per copiare il contenuto della pagina dal file system in memoria fisica.
- Per **Page Cache Index** si intende un meccanismo efficiente per la ricerca di una pagina in base al **descrittore di pagina** che contiene la coppia **<identificatore file, offset>** sul quale è mappata, e altre informazioni, tra cui **ref_count** (contatore dei riferimenti alla pagina fisica).
- Si noti che **ref_count** è uguale al numero di processi che mappano la pagina + 1 in modo da mantenere l'allocazione della pagina fisica anche quando tutti i processi che la utilizzano terminano;

Crescita e decrescita della page cache

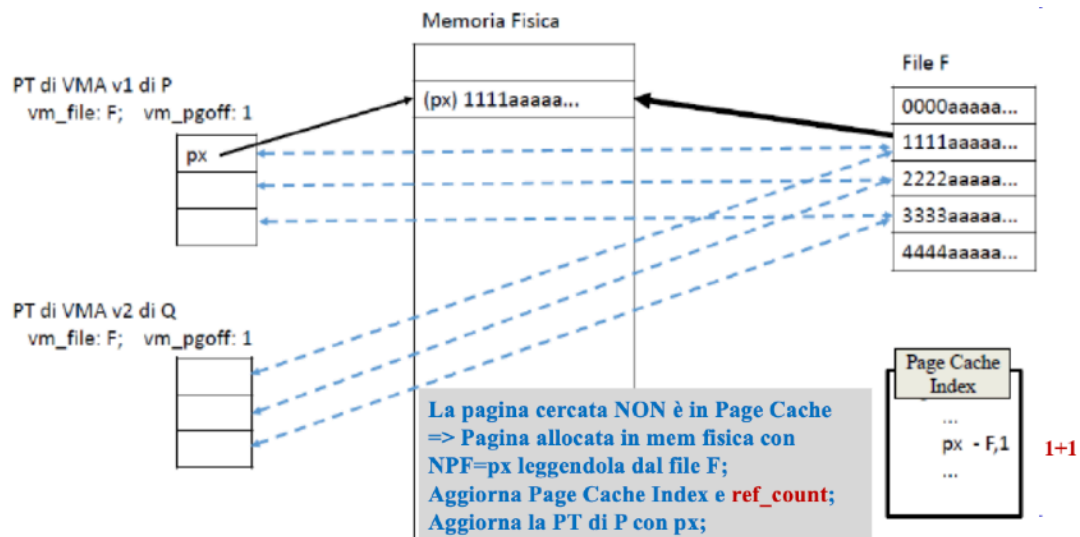
- **Obiettivo:** mantenere in memoria le pagine lette da disco il più a lungo possibile, perché qualche processo potrebbe volerle accedere in futuro trovandole già in memoria e risparmiando così costosi accessi a disco.
- Le pagine caricate nella **Page Cache** non vengono liberate neppure quando tutti i processi che le utilizzavano non le utilizzano più, ad esempio perché sono state scritte e quindi duplicate (tramite meccanismo COW) oppure addirittura perché i processi sono terminati (exit).
- L'eventuale liberazione di pagine della **Page Cache** avviene solo nel contesto della generale politica di **gestione della memoria fisica**, a fronte di nuove richieste di pagine fisiche da parte dei processi quando la memoria fisica è quasi piena

Accesso alla page cache

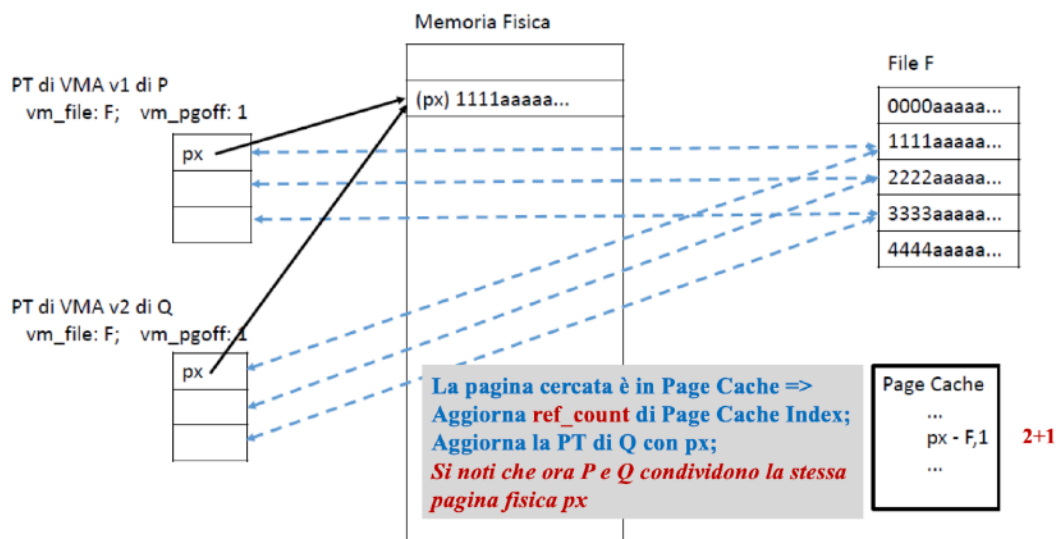
Quando un processo richiede di accedere ad una pagina virtuale mappata su un file, il sistema svolge le seguenti operazioni (con riferimento all'esempio 1):

- determina il descrittore di pagina costituito dalla coppia **<F, 1>** dove:
 - **F** è il file indicato nella **VMA**;
 - il **page offset** (offset in pagine) è la somma dell'offset della **VMA** rispetto al file (1, nell'esempio) sommato all'offset dell'indirizzo di pagina richiesto rispetto all'inizio della **VMA** cioè 0 nella prima lettura, perché la variabile base punta alla prima pagina della **VMA**;
- verifica se la pagina è già indicizzata nella Page Cache → la pagina virtuale viene semplicemente mappata su tale pagina fisica; aggiorna **ref_count** di Page Cache Index; aggiorna la PT del processo;
- altrimenti alloca una pagina fisica in memoria e vi carica la pagina leggendola dal file; aggiorna Page Cache Index e aggiorna **ref_count**; aggiorna la PT del processo;

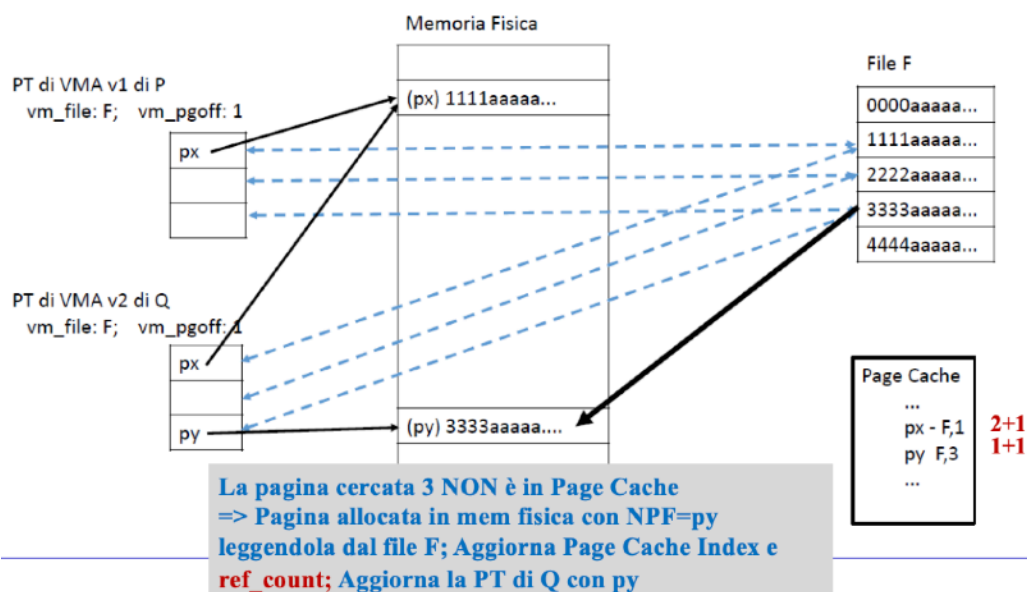
esempio 1 - il processo P esegue la 1° lettura alla 1° pagina



esempio 1 - il processo Q esegue la 1° lettura alla 1° pagina



esempio 1 - il processo Q esegue la 2° lettura alla 3° pagina



SCRITTURA IN VMA SHARED

I dati vengono scritti sulla pagina fisica condivisa tramite la Page Cache, quindi:

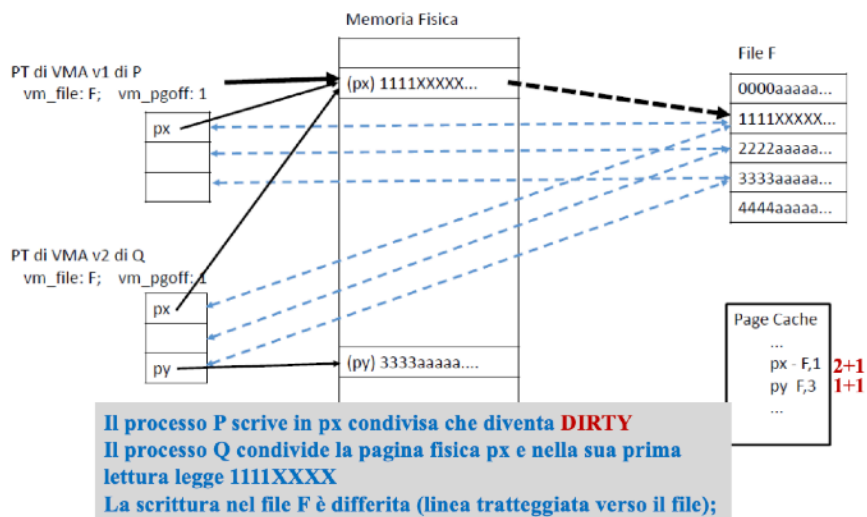
1. la pagina fisica modificata è marcata **"dirty"**
2. tutti i processi che mappano tale pagina fisica condivisa vedono **immediatamente** le modifiche
3. successivamente la pagina modificata verrà riscritta sul file;
 - non è indispensabile riscrivere subito su file la pagina, in quanto i processi che la condividono vedono le modifiche;
 - in questo modo **la pagina verrà scritta su disco una volta sola** anche se venisse aggiornata più volte in memoria fisica;

esempio 2 - scrittura in VMA SHARED

- Modifichiamo l'esempio 1 in modo che il processo P invece di leggere, **scriva nella prima pagina** dei caratteri ad X dopo i primi 4 uno:
1 1 1 1 X X X X X
- Ovviamente la VMA deve essere abilitata in **scrittura**; pertanto dobbiamo modificare l'invocazione di `mmap`:

```
base = mmap(mapaddress1, PAGE_SIZE*3, PROT_READ|PROT_WRITE, MAP_SHARED, fd, PAGE_SIZE);
```

esempio 2 - scrittura in VMA SHARED da parte di P, lettura da parte di Q



SCRITTURA IN VMA PRIVATE: MECCANISMO COPY-ON-WRITE (COW)

Per realizzare questo meccanismo, detto **Copy-on-write (COW)**, è necessario intercettare le **scritture** su pagine di **VMA PRIVATE**:

- la protezione delle pagine di una **VMA PRIVATE** **scrivibile** deve essere posta inizialmente in lettura **R** (non scrivibile) in modo che l'eventuale **scrittura** in una pagina generi un **Page Fault** per violazione di protezione;
- il gestore di Page Fault scoprirà questa situazione che richiederà **l'allocazione di una nuova pagina fisica**.

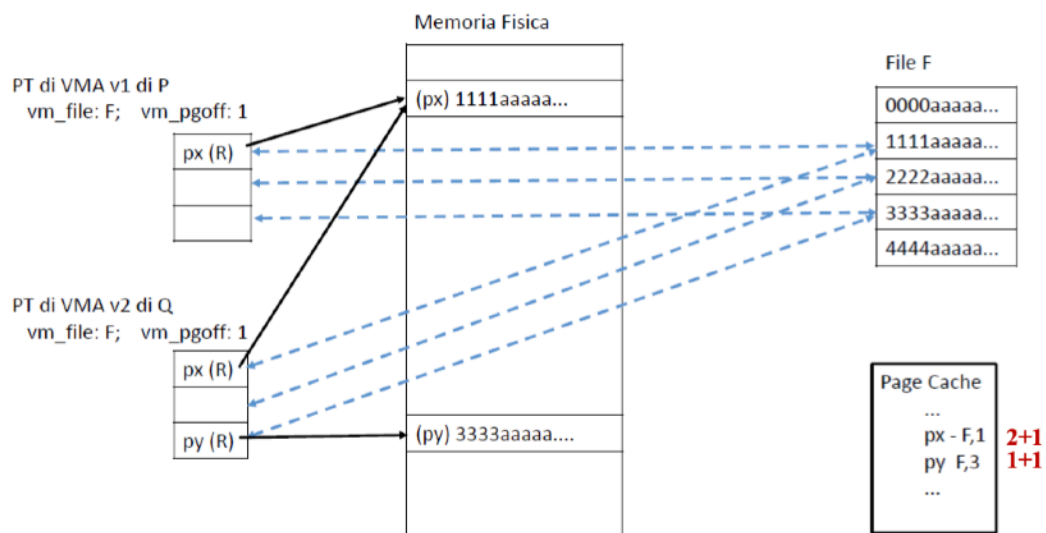
- La scrittura in una pagina virtuale di una VMA di tipo **PRIVATE** allocata in una pagina fisica PFx si basa sul meccanismo di COW:
 - la pagina **PFx** viene **duplicata** allocando **una nuova pagina fisica PFz PRIVATA per P**;
 - la nuova pagina non risulta più mappata sul file F;
 - la nuova pagina non appartiene alla Page Cache e non risulta più condivisa dagli eventuali processi che la condividevano;
 - la scrittura effettuata dal processo P viene applicata solamente alla pagina fisica privata PFz ;
 - la pagina fisica originale PFx ed il file F rimangono inalterati;

esempio 3 - scrittura in VMA PRIVATE

Modifichiamo l'esempio creando la VMA del processo P di tipo PRIVATE come segue:

```
base = mmap(mapaddress1, PAGE_SIZE * 3, PROT_READ|PROT_WRITE, MAP_PRIVATE, fd, PAGE_SIZE);
```

esempio 3 - situazione dopo la creazione delle VMA, la prima lettura di P e le due letture di Q

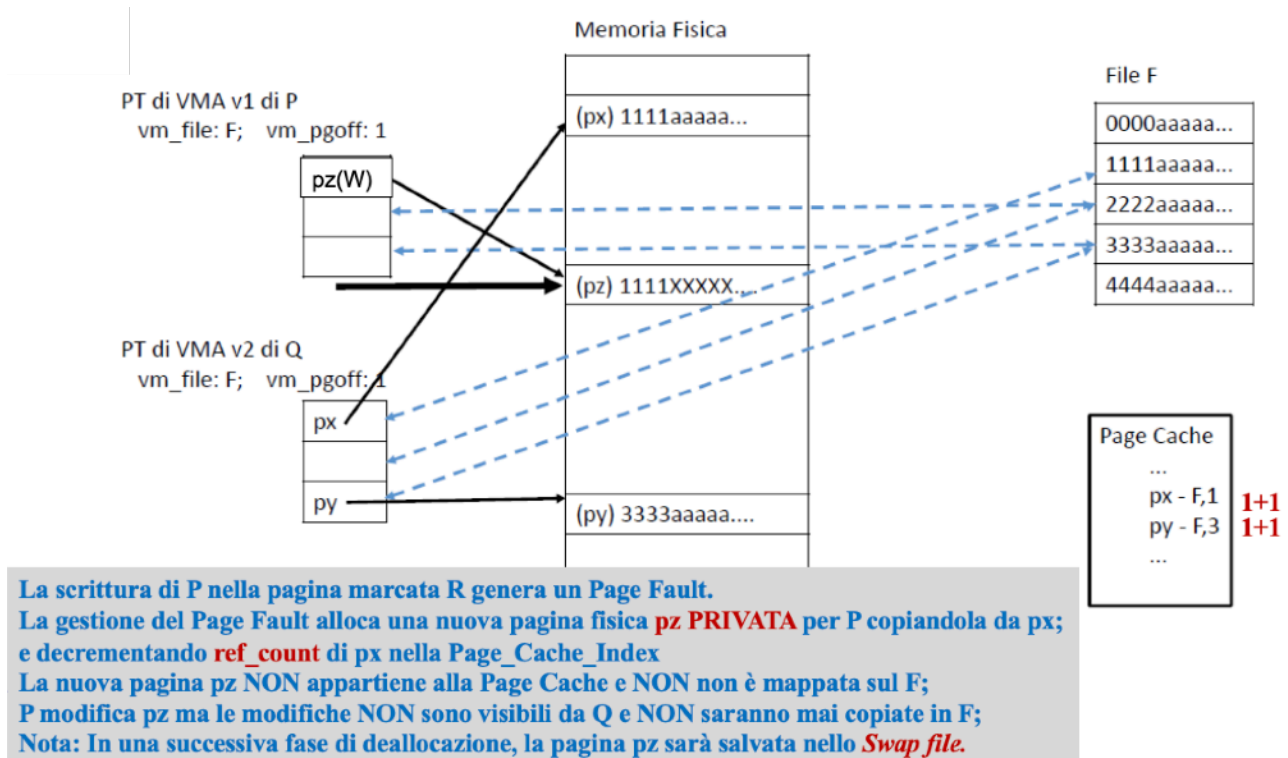


Si noti che in fase di lettura, il comportamento delle **VMA_PRIVATE** è lo stesso delle **VMA_SHARED** (infatti **px** è inizialmente **condivisa** e la protezione delle pagine è posta a **R**)

esempio 3 - scrittura in VMA PRIVATE

- Supponiamo che P scriva il valore 1 1 1 1 X X X X nella prima pagina privata e che il processo Q rilegga le pagine 1 e 3 DOPO la scrittura di P:
- L'output prodotto dai processi è in questo caso:
 - processo P – scrittura di P del valore 1111XXXX nella prima pagina privata
 - processo Q - prima lettura di Q nella prima pagina
1 1 1 1 a a a a a → **il processo Q non osserva l'effetto della scrittura effettuata da P nella prima pagina privata**
 - processo Q - seconda lettura di Q nella terza pagina
3 3 3 3 a a a a a

esempio 3 - situazione dopo la scrittura di P nella 1° pagina privata



Algoritmo del Page Fault Handler con meccanismo Copy on Write

```

if (NPV non appartiene alla memoria virtuale del processo)
    il processo è abortito e viene segnalato un "Segmentation Fault"
else if (NPV è allocata in pagina PFx, ma l'accesso non è legittimo perché viola le protezioni) {
    if (violazione di accesso in scrittura a pagina con protezione R di una VMA scrivibile){
        if (ref_count(di PFx) > 1) {
            copia PFx in una pagina fisica privata PFz;
            decrementa ref_count di PFx in Page Cache;
            assegna NPV a PFz e scrivi in PFz}
        else { abilita NPV in scrittura };    //pagina utilizzata SOLO da questo processo
    }
    else if (l'accesso è legittimo, ma NPV non è allocata in memoria)
        invoca la routine che deve caricare in memoria la pagina virtuale NPV dal file di backing
        store (demand paging)
}
    
```

VMA di tipo ANONIMO

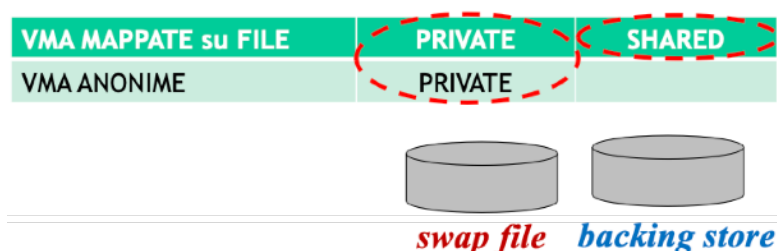
- Il sistema utilizza **aree anonime** per la **pila** e per lo **heap** cioè per l'area dati dinamici dei processi
- Le aree di tipo **ANONIMO non** hanno un file associato
- La definizione di un'area di tipo **ANONIMO non** alloca memoria fisica perché le sue pagine virtuali sono mappate sulla **ZeroPage** (una pagina fisica inizializzata a zero e mantenuta dal SO)
 - La **lettura** di una pagina trova la ZeroPage senza richiedere l'allocazione di alcuna pagina fisica.

- La **scrittura** di una pagina innesca il meccanismo **Copy on Write** che richiede l'allocazione di una nuova pagina fisica (come per le pagine di aree **VMA** di tipo **PRIVATE**)

Distinzione tra Backing Store e Swap File

Quando è necessario **deallocare una pagina fisica che è stata scritta (dirty page)**, il suo contenuto deve essere salvato su un file:

1. Le pagine di tipo **VMA SHARED** sono salvate sul file di **backing store** sul quale sono mappate
2. Le pagine di tipo **VMA ANONIME** (che non hanno un file di backing store) o **VMA PRIVATE** duplicate in memoria fisica a causa di una scrittura a seguito di **COW** (e quindi non più mappate su file di backing store) devono essere salvate su un file denominato **Swap File**.



Applicazione alle VMA di un processo

- I meccanismi visti in questa lezione non sono applicati solo per implementare le VMA generate esplicitamente da un processo tramite `mmap()`, ma sono utilizzati in generale dal sistema operativo per gestire le **VMA dei processi**.
- Possiamo suddividere le **VMA** di un processo nel modo seguente:
 - **VMA mappate sull'eseguibile (C, K e S)**
 - **VMA (M) aree memory mapped** create su richiesta non solo del programma eseguibile, ma anche del S.O. (ad esempio, librerie dinamiche mappate sui rispettivi eseguibili)
 - **VMA anonime**: Dati dinamici (D) Pile dei thread (T), Pila (P)
- Il **demand paging** consiste nell'**allocazione fisica delle pagine virtuali solo quando sono accedute** in lettura o in scrittura: ad esempio, una pagina di codice verrà allocata in memoria fisica solo alla prima lettura;
- **Meccanismo COW** per scrittura in **VMA PRIVATE** (es. librerie condivise) e in **VMA ANONIME** (es. pagine dati dinamici o pagine di pila)

VMA mappate sull'eseguibile (C, K, S)

Le aree **C, K e S** mappate sull'eseguibile sono tutte di tipo **VMA PRIVATE**

- Le aree codice **C** e costanti per rilocalizzazione dinamica **K** non sono scrivibili (la distinzione tra **SHARED/PRIVATE** sarebbe irrilevante) e rimangono condivise.
- Le pagine dei dati statici **S** devono essere di tipo **PRIVATE**, perchè le **scritture** non devono modificare l'eseguibile e non devono essere osservabili tra processi che eseguono lo stesso programma. Le pagine saranno duplicate al momento della scrittura tramite il meccanismo **Copy On Write** (non più condivise con altri processi e con il file eseguibile) e salvate sullo swap file.

start-end page	perm	offset	device	i-node	file name
0040 0 - 0040 1	r-xp	000000	08:01	394275	.../user.exe
0060 0 - 0060 1	r--p	000000	08:01	394275	.../user.exe
0060 1 - 0060 2	rw-p	001000	08:01	394275	.../user.exe
...					

VMA mappate sull'eseguibile (C)

Se due processi eseguono contemporaneamente lo stesso programma, le pagine di codice (C) saranno **condivise**, in quanto il secondo processo troverà tali pagine già lette e presenti nella Page Cache

→ Grazie al principio di ***mantenere in memoria fisica le pagine lette da disco il più a lungo possibile***, è possibile che un processo trovi già nella Page Cache il codice del programma caricato da un precedente processo, **anche se quest'ultimo è già terminato** – dipenderà da quanto la memoria fisica è stata occupata durante l'intervallo tra le esecuzioni dei due processi

VMA per Memory-mapped files (M)

- Linker utilizza VMA per realizzare la condivisione delle pagine fisiche delle librerie condivise (ad esempio, **glibc**)
- Il Linker crea le VMA per le varie librerie condivise utilizzando il tipo **MAP_PRIVATE**; in questo modo:
 - il codice, che non viene mai scritto, rimane sempre **condiviso**
 - solo le pagine sulle quali un processo scrive sono riallocate al processo tramite il meccanismo COW e non più condivise con gli altri processi e con il file

start-end page	perm	offset	device	i-node	file name
...					
7fff f7a1 c - 7fff f7bd 0	r-xp	000000	08:01	271666	.../libc-2.15.so
7fff f7bd 0 - 7fff f7dc f	---p	1b4000	08:01	271666	.../libc-2.15.so
7fff f7dc f - 7fff f7dd 3	r--p	1b3000	08:01	271666	.../libc-2.15.so
7fff f7dd 3 - 7fff f7dd 5	rw-p	1b7000	08:01	271666	.../libc-2.15.so
...					

VMA di pila

- La VMA relativa alla **pila** ha le seguenti caratteristiche:
 - **ANONIMA**
 - dimensionamento iniziale (34 pagine nel sistema sperimentato)
 - flag **GROWSDOWN** attivo: indica che la pila può crescere automaticamente quando viene acceduta l'ultima pagina disponibile (**pagina di growsdown**)
- **le pagine vengono allocate in base alle richieste di accesso (ON DEMAND)**
- la prima pagina virtuale della VMA (pagina di growsdown) **non è mai allocata**, anche quando la pila viene riempita
- **Esiste un meccanismo di crescita automatica della VMA di pila, ma non di decrescita** – quando la pila decresce le pagine rimangono allocate (ma verranno sovrascritte da una successiva ricrescita della pila)

Riassunto delle proprietà della VMA di pila

- Al momento dell'**exec** la VMA di pila viene creata con un certo numero di NPV iniziali non allocate fisicamente eccetto quelle utilizzate immediatamente (tipicamente la prima o le prime due)
- tale area iniziale di pila può essere acceduta in qualsiasi posizione, anche secondo schemi non conformi al funzionamento di una pila
- ogni accesso a NPV non allocate fisicamente ne causa l'allocazione
- l'accesso al NPV iniziale (pagina di growsdown) produce una crescita automatica dell'area
- il tentativo di accedere pagine diverse da quelle esistenti produce un Segmentation fault

VMA dei dati dinamici (D)

- **Dati dinamici (D)**: area destinata a contenere i dati allocati dinamicamente su richiesta (tramite `malloc`)
 - il limite corrente di quest'area è indicato dalla variabile `brk` (**Program break**) contenuta nel descrittore del processo
 - contiene anche gli eventuali dati statici non inizializzati definiti nell'eseguibile (**Block Started by Symbol - BSS**)
- La VMA relative all'area D ha le seguenti caratteristiche:
 - **ANONIMA**
 - dimensione iniziale uguale a quella dei dati statici non inizializzati (BSS) indicati nell'eseguibile
- L'area può crescere grazie al servizio `brk()`
 - la funzione `malloc` ha bisogno di richiedere al SO di allocare nuovo spazio tramite l'invocazione del servizio di sistema `brk()` o `sbrk()`
 - `brk()` e `sbrk()` sono due funzioni di libreria che invocano in forma diversa lo stesso servizio; analizziamo solamente la forma `sbrk`, che ha la seguente sintassi:

`*void sbrk(int incremento)`

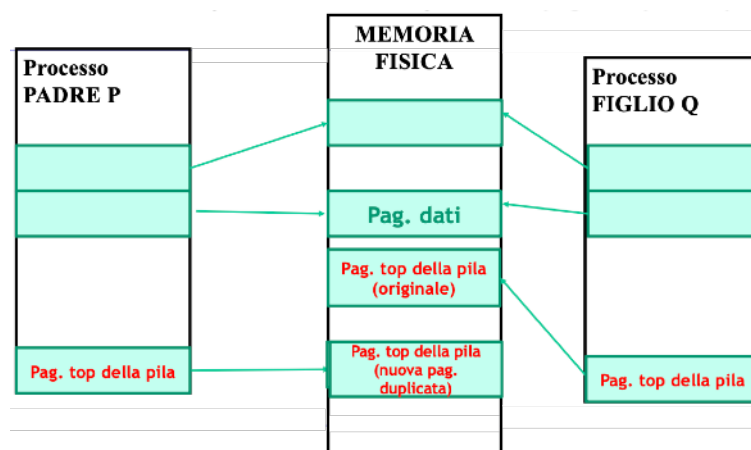
- incrementa l'area dati dinamici del valore incremento, arrotondato ad un limite di pagina
- restituisce un puntatore alla posizione iniziale della nuova area
- `sbrk(0)` restituisce il valore corrente della cima dell'area dinamica.

Evento di fork e gestione della memoria

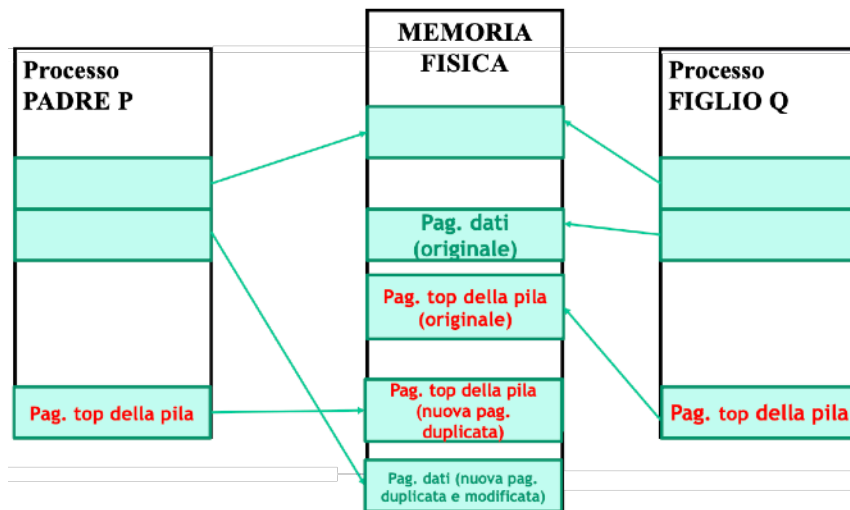
- L'esecuzione di una `fork()` *crea lo spazio virtuale della VMA di Q ma le pagine virtuali presenti in memoria fisica sono CONDIVISE tra processo padre P e figlio Q*
 - quindi inizialmente marcate R (sola lettura)
- **Tranne la pagina al top della pila* che viene duplicata** (tramite meccanismo COW) ed attribuita al padre, mentre la pagina fisica originale è attribuita al processo figlio
- Si assume che il processo padre prosegua l'esecuzione
- Se la pagina dati dinamici era allocata nella ZP, solo allora la pagina dati da scrivere viene effettivamente allocata (le pagine relative alle aree dati sono inizialmente poste con diritto di accesso in sola lettura) con meccanismo di **copy on write per duplicare la pagina che deve essere scritta ed attribuirla al padre**; la pagina fisica originale è attribuita al processo figlio

(*) Che contiene la variabile `pid = fork()` alla quale è assegnato il valore restituito dalla `fork()`.

Memoria subito dopo una fork con duplicazione pagina top della pila



Memoria dopo che il processo padre ha scritto in una pagina dati



Exec e gestione memoria

`exec ()`

- L'esecuzione della "exec" invalida tutta la Tabella delle Pagine relative al processo.
- La struttura delle VMA viene ricostruita in base al contenuto del file eseguibile e vengono predisposte le PTE necessarie nella nuova TP
- In pratica vengono caricate inizialmente in memoria fisica:
 - la pagina di codice contenente la prima istruzione del programma lanciato in esecuzione
 - la prima pagina di pila nella quale sono posti i parametri passati al main – (argv e env)
 - il programma inizia l'esecuzione
- Successivamente, tramite la tecnica di Demand Paging, vengono allocate le pagine di codice e dati del programma lanciato in esecuzione in base ai page fault generati

Creazione di thread e pile dei thread

- I processi leggeri (**thread**) condividono la stessa struttura di aree virtuali e Tabella delle Pagine del processo padre (**thread principale**)
- Assoluta **condivisone** della memoria tra il processo padre e i suoi thread: ogni modifica di memoria eseguita da un **thread** è visibile da tutti gli altri
- Per ogni **thread** viene allocata una pila di dimensione **2048 pagine**, cioè **8 MB (2048 = x800)** seguita da una pagina di interposizione in sola lettura
- La pila del **primo thread** inizia logicamente al **NPV 7FFF F77F F** e si sviluppa verso indirizzi più bassi, quindi la sua VMA è compresa tra
 - 7FFF F6FF F (infatti 6FFF + 800 = 77FF)
 - 7FFF F77F F
- La pila del **secondo thread** inizia logicamente al **NPV 7FFF F6FF E** e si sviluppa verso indirizzi più bassi, quindi la sua VMA è compresa tra
 - 7FFF F67F E
 - 7FFF F6FF E (= 7FFFF6FFF – 1 per pagina di interposizione)

Content switch e gestione della memoria

- L'esecuzione di un **Context Switch** non ha effetti diretti sulla memoria, ma causa uno svuotamento (**flush**) del TLB, perché rende tutte le pagine del processo corrente non utilizzabili nel prossimo futuro.

- Conseguenza del flush del TLB è la necessità di **salvare lo stato del Dirty bit** presente nel TLB per le pagine dirty eliminate (che altrimenti andrebbe perso).
 - **Dirty bit salvato in memoria fisica nei descrittori delle pagine fisiche eliminate dal TLB;**
 - **Dirty bit salvato anche nella Page Table;**
- Successivamente carica le pagine attive di codice e di pila del processo che va in esecuzione

Exit e gestione memoria

- L'esecuzione della **exit** di un processo causa i seguenti effetti sulla gestione della memoria:
 - eliminazione della struttura delle **VMA** del processo
 - eliminazione della **PT** del processo
 - deallocazione delle **NPV** del processo con conseguente deallocazione delle pagine fisiche occupate solamente da tali **NPV** (in assenza di condivisione cioè **ref_count=1**), oppure riduzione del loro **ref_count** (per pagine che potrebbero essere mappate su file o condivise con un altro processo);
 - esecuzione di un **context switch** con conseguente **flush** del TLB
- Nel caso in cui il processo terminato abbia dei *thread* attivi, anche questi devono essere eliminati.

Strategia complessiva di allocazione della memoria fisica

In generale Linux cerca di sfruttare il più possibile la disponibilità di memoria RAM

- una certa quantità di memoria viene allocata inizialmente al Sistema Operativo e non viene mai deallocata
- le eventuali richieste di memoria dinamica da parte del SO o dei processi vengono gestite con il meccanismo di **demand paging**
- tutti i dati letti dal disco vengono conservati per poter essere eventualmente riutilizzati

	DRAM	NAND Flash	HDD
Access time (w/r)	10/10ns	200/25us	9.5/8.5ms

Allocazione della memoria fisica

L'unità di base per l'allocazione della memoria è la **pagina**, ma Linux cerca di allocare **blocchi di pagine contigue** per mantenere la memoria il meno frammentata possibile

Questa scelta può apparire inutile, perché lo spazio di indirizzamento virtuale permette di operare su una sequenza continua di indirizzi, anche se le corrispondenti pagine fisiche non lo sono, tuttavia:

- la memoria è acceduta anche dai canali DMA in base a indirizzi fisici, non virtuali; se un buffer di DMA supera la dimensione della pagina deve essere costituito da pagine contigue
- la rappresentazione della RAM risulta più compatta, facilitando ulteriori ottimizzazioni delle architetture hardware

Deallocazione della memoria fisica

- Fino a quando la quantità di RAM libera è sufficientemente grande Linux alloca memoria fisica senza svolgere particolari controlli
 - **Quindi la memoria richiesta dai processi e dalle disk cache tende a crescere continuamente**
- La memoria occupata può **decrescere spontaneamente** solamente in due casi:
 - Quando un processo termina (exit)
 - Quando un processo rilascia esplicitamente la memoria
- Può accadere che la quantità di memoria libera scenda sotto un **livello critico (minFree)**
- In questo caso interviene l'algoritmo **PFRA (Page Frame Reclaiming Algoritmo)** che cerca di riportare il numero di pagine libere a un **livello obiettivo** che chiameremo **maxFree**

NOTA: dato che PFRA ha bisogno di allocare memoria per il proprio funzionamento, per evitare che il sistema raggiunga una saturazione della memoria e vada in crash, l'attivazione di PFRA deve avvenire prima di tale momento

Scomposizione del problema di Page Frame Reclaiming

Il problema di **Page Frame Reclaiming** può essere scomposto in:

1. la **scelta delle pagine da deallocare**, che a sua volta può essere suddiviso in
 1. determinazione di **quando** e **quante** pagine è necessario liberare
 2. determinazione di **quali** pagine liberare
2. il meccanismo di **swapping**, cioè l'effettiva deallocazione delle pagine dalla memoria fisica (**swap-out**) e l'eventuale riallocazione in memoria fisica di pagine swappate (**swap-in**)

Strategia complessiva di deallocazione della memoria

Gli interventi di **deallocazione** si svolgono applicando i seguenti tipi di azioni nell'ordine indicato:

1. le pagine di Page Cache non utilizzate dai processi (**ref_count = 1**) vengono deallocate; se questo non è sufficiente
2. alcune pagine utilizzate dai processi vengono deallocate; se anche questo non è sufficiente
3. un processo viene eliminato completamente (**killed**)

Quando una pagina viene **deallocata** vengono svolte le seguenti operazioni:

- se la pagina in memoria è stata letta da disco ma mai modificata, essa viene resa subito disponibile
- se la pagina è stata **modificata**, cioè se il suo **Dirty bit** è settato, prima di rendere la pagina disponibile, essa deve essere scritta su disco

Parametri di PFRA

Parametri di **PFRA** (Page Frame Reclaiming Algorithm):

- **freePages**: numero di pagine di memoria fisica libere
- **requiredPages**: numero di pagine che vengono richieste per una certa attività da parte di un processo (o del SO)
- **minFree**: numero minimo di pagine libere sotto il quale non si vorrebbe scendere (**livello critico**)
- **maxFree**: numero di pagine libere al quale PFRA tenta di riportare freePages (**livello obiettivo**)

PFRA: determinazione di quando e quante pagine deallocare

QUANDO deallocare?

PFRA è invocato nei casi seguenti:

1. **Invocazione diretta** da parte di un processo che richiede **requiredPages** se:
$$(freePages - requiredPages) < minFree$$

cioè se l'allocazione delle pagine richieste porterebbe la memoria fisica sotto il livello critico
2. **Attivazione periodica** tramite **kswapd** (kernel swap daemon), una funzione che viene attivata periodicamente e che invoca PFRA se **freePages < maxFree**

QUANTE pagine deallocare?

PFRA determina il numero di pagine da deallocare (**toFree**) tenendo conto delle pagine richieste con l'obiettivo di riportare il sistema ad aver **maxFree** pagine libere:

$$toFree = maxFree - freePages + requiredPages$$

PFRA: determinazione di quali pagine deallocare

Dal punto di vista di **PFRA** i tipi di pagine sono:

- Pagine non deallocabili (kernel mode):
 - pagine statiche del SO dichiarate non deallocabili
 - pagine allocate dinamicamente dal S.O
 - pagine appartenenti alla sPila dei processi
- Pagine mappate sul file eseguibile dei processi che possono essere deallocate senza mai riscriverle (codice, costanti)
- Pagine (anonime) che richiedono l'esistenza di una Swap Area su disco per essere deallocate
 - pagine dati
 - pagine della uPila
 - pagine dello Heap
- Pagine che sono mappate su un file: pagine appartenenti ai buffer/cache (vedi File System)

- Prima di tutto PFRA dealloca le pagine appartenenti alla PageCache non più utilizzate da nessun processo (**ref_count = 1**), in ordine di NPF.
- Nel caso non fosse sufficiente, l'algoritmo utilizzato da **PFRA** per scegliere quali pagine deallocare si basa sui principi di **LRU**, cioè sulla scelta delle pagine meno recentemente usate.
- Per poter implementare questo comportamento occorre:
 1. Mantenere l'informazione relativa all'accesso delle pagine
 2. Avere un meccanismo efficiente per scegliere le pagine meno recentemente usate

NOTA: L'algoritmo implementato in Linux è più complesso ed efficiente. Tuttavia, l'algoritmo presentato nelle seguenti slide cattura la sua logica fondamentale.

Liste di accesso alle pagine

- Per mantenere l'informazione relativa all'accesso alle pagine, esistono due liste globali, dette **LRU list**, che elencano tutte le pagine appartenenti ai processi e alle Buffer/Cache:
 - **ACTIVE LIST**: contiene tutte le pagine che sono state accedute recentemente e non possono essere deallocate (in questo modo PFRA non dovrà scorrerle per scegliere una pagina da deallocare);
 - **INACTIVE LIST**: contiene le pagine inattive da molto tempo che sono quindi candidate per essere deallocate.
- L'obiettivo dello spostamento delle pagine all'interno e tra le due liste **active** e **inactive** è di **accumulare le pagine meno recentemente utilizzate in coda alla inactive**, mantenendo nella active le pagine più recentemente utilizzate (Working Set) di tutti i processi
- Rispetto al meccanismo di LRU teorico abbiamo una differenza fondamentale dovuta alle limitazioni dell'hardware:
 - x64 non tiene traccia del numero di accessi alla memoria
 - Linux realizza un'approssimazione basata sul **bit di accesso A alla pagina presente nel TLB**:
 - **A viene posto a 1 da x64 ogni volta che la pagina viene acceduta**
 - **A viene azzerato periodicamente dal sistema operativo**

Spostamento delle pagine tra le due liste

- Oltre al bit di accesso **A** settato dall'Hardware, ad ogni pagina viene associato un flag, **ref** (referenced),
- Il flag **ref** serve in pratica a raddoppiare il numero di accessi necessari per spostare una pagina da una lista all'altra
- Definiamo una funzione periodica **Controlla_liste**, attivata da **kswapd**, che esegue una scansione di ambedue le liste:
 1. **Scansione della active list dalla coda** spostando eventualmente alcune pagine alla inactive
 2. **Scansione della inactive list dalla testa** (escluse le pagine appena inserite provenienti dalla active) spostando eventualmente alcune pagine alla active
- Ogni pagina viene quindi processata una sola volta analizzando i due flag **A** e **ref**
- Alla funzione **Controlla_liste** dobbiamo aggiungere:
 - Funzioni che allocano nuove pagine:
 - Richieste da un processo e poste nella lista active con **ref=1**
 - Richieste da un processo figlio appena creato e poste nella lista **active** o **inactive** nello stesso ordine o con lo stesso ref del processo padre;

- Funzioni che eliminano dalle liste pagine non più residenti (**swapped out**) oppure pagine deallocate definitivamente per la terminazione (**exit**) di un processo

Funzione periodica *Controlla_liste*

1. Scansione della active list dalla coda

- se **A = 1**
 - azzera A
 - se (ref = 1) sposta P in testa alla active
 - se (ref = 0) pone ref = 1
- se **A = 0**
 - se (ref = 1) pone ref = 0
 - se (ref=0) sposta P in testa alla inactive con ref = 1

2. Scansione della inactive list dalla testa (escluse le pagine appena inserite provenienti dalla active)

- se **A = 1**
 - azzera A
 - se (ref = 1) sposta P in coda alla active con ref=0
 - se (ref = 0) pone ref = 1
- se **A = 0**
 - se (ref = 1) pone ref = 0
 - se (ref = 0) sposta P in coda alla inactive

Deallocazione delle pagine della lista inattiva

Vengono scelte le pagine virtuali della **inactive**, partendo dalla coda, ma tenendo conto della condivisione nel modo seguente:

- una pagina virtuale viene considerata deallocabile *solo se tutte le pagine condivise sono accedute meno recentemente di lei*
- (ovvero, se una pagina fisica è condivisa tra molte pagine virtuali, viene considerata deallocabile solo quando nella scansione della **inactive** si sono già trovate tutte le pagine che la condividono)

IL MECCANISMO DI SWAPPING

Deallocazione delle pagine da memoria fisica

- Quando è necessario **deallocare una pagina fisica che è stata scritta (dirty page)**, il suo contenuto deve essere salvato su disco:
 1. Se la pagina è mappata su **file di backing store** in modo **SHARED** => deve essere salvate su tale file;
 2. Se la pagina fisica è **ANONIMA** oppure **PRIVATA** e duplicata in memoria fisica a causa di una scrittura a seguito di **COW** (e quindi non più mappata su file) => deve essere salvata sullo **Swap File**
- Una pagina fisica **Pf_x** è **Dirty** se:
 - il suo descrittore è marcato **D** a seguito di un **TLB flush**
 - una delle pagine virtuali condivise in **Pf_x** è presente nel **TLB** ed è marcata **D**

Deallocazione delle pagine - Swapping

- Il meccanismo di **Swapping** richiede che sia definita almeno una **Swap Area su disco** costituita da un file o da una partizione (vedi capitolo sul File System).

- Per semplicità negli esempi ipotizzeremo che esista una sola **Swap Area** di tipo file, che chiameremo anche **Swap File**.

Una **Swap Area** consiste di una sequenza di **Page Slot** di dimensione pari ad una pagina:

- **SWPN (Swap Page Number)** è l'identificatore di un **Page Slot**
- Per ogni **Page Slot** esiste un contatore detto **swap_map_counter** per tener traccia del numero di PTE che si riferiscono alla pagina fisica swappata su disco (ovvero il numero di pagine virtuali che condividono la pagina)

Regole generali di swapping: *Swap_out*

- Quando **PFRA** chiede di **deallocare** una pagina da memoria fisica a disco (**swap_out**):
 - viene allocato un **Page Slot** dello Swap File su disco;
 - la pagina fisica viene copiata nel Page Slot su disco (**scrittura su disco**) e deallocata;
 - allo **swap_map_counter** viene assegnato il numero di pagine virtuali che condividono la pagina fisica;
 - in ogni **PTE** che condivideva la pagina fisica viene registrato il **SWPN** identificatore del **Page Slot** al posto del **NPF** (e il bit di presenza in memoria fisica viene azzerato);
- Negli esercizi lo **SWPN** delle pagine swappate è indicato nella **TP** preceduto da una **S**, per distinguerlo da un normale **NPF**
 - esempio, se un processo ha swappato la pagina **d0** nel **Page Slot 1**, nella **TP** avremo l'associazione **<d0: s1>**

Regole generali di swapping: *Swap_in*

Quando un processo accede ad una pagina precedentemente swappata su disco:

- si verifica un **Page Fault**
- il gestore del **Page Fault** attiva la procedura di **allocazione (swap_in)** della pagina in memoria fisica
- il contenuto del **Page Slot** indicato dalla **PTE** viene copiato da disco nella pagina fisica (**lettura da disco**)
- la **PTE** viene aggiornata inserendo l'**NPF** della pagina fisica al posto del **SWPN** identificatore del **Page Slot**

Ottimizzazione dello Swapping

- Spesso una pagina già deallocata con uno **Swap_out** e sulla quale è stato già eseguito uno **Swap_in**, deve essere di nuovo deallocata (**Swap_out**).
- LINUX tenta di limitare il più possibile i trasferimenti tra disco e memoria fisica nel caso di pagine swappate più volte **ma senza essere state modificate**:
 - In pratica **non** si cancella la pagina dalla Swap Area al momento dello **swap_in**
 - Ad un successivo **Swap_out**, se la pagina non è stata modificata dal momento dello **Swap_in**, non è necessario riscriverla su disco perché già presente;
- Necessario introdurre una **Swap Cache** composta da:
 - l'insieme delle pagine che sono state rilette da Swap File e non sono state modificate
 - alcune strutture ausiliarie (**Swap_Cache_Index**) che permettono di gestirla come se tali pagine fossero mappate sulla Swap Area.
- Una pagina appartiene alla **Swap Cache** se è presente sia in memoria che su Swap Area e se è registrata nel **Swap_Cache_Index**.

Meccanismo della Swap Cache

Si consideri una pagina swappata nella Swap Area:

- Se la pagina viene **letta**, si deve eseguire uno **Swap_in**: la pagina viene copiata dallo Swap File in memoria fisica, **ma la copia nella Swap Area non viene eliminata**
 - La pagina letta viene marcata in sola lettura R.
 - Nel **Swap_Cache_Index** viene inserito un descrittore che contiene i riferimenti sia alla pagina fisica **Pf_x** che al Page Slot **SWPN_x**
 - Finché la pagina viene solamente **letta** questa situazione non cambia e, se la pagina viene nuovamente deallocata, **non è necessario riscriverla su disco**.
- Se la pagina viene **scritta**, si verifica un **Page Fault** che causa le seguenti operazioni:
 - la pagina diventa privata, cioè viene allocata una nuova pagina fisica;
 - la protezione della pagina viene posta a **W**;
 - il contatore **swap_map_counter** viene decrementato (perché la pagina non appartiene più alla Swap Area);
 - se il contatore è diventato **0**, il **Page Slot** viene deallocato dalla **Swap Area**;

per un esempio vedere le slide L11 - M3

Gestione delle liste LRU in caso di Swap-in

Il comportamento delle liste active e inactive in caso di **Swap_in** è il seguente:

- La **NPV** che è stata allocata in memoria fisica (in lettura o scrittura) cioè quella che ha causato lo **Swap_in** → **inserita in testa alla Lista active con ref = 1**;
- Eventuali altre **NPV** condivise con la stessa pagina fisica da parte di altri processi → **inserite in coda alla Lista inactive con ref = 0**.

Interferenza tra Gestione della Memoria e Scheduling

- L'allocazione/deallocazione della memoria interferisce con i meccanismi di scheduling
- Supponiamo che:
 - un processo Q a bassa priorità sia un forte consumatore di memoria
 - contemporaneamente sia in funzione un processo P molto interattivo
- Può accadere che, mentre il processo P è in attesa, il processo Q carichi progressivamente tutte le sue pagine forzando fuori memoria le pagine del processo P (e magari anche dello Shell sul quale Q era stato invocato).
- Quando P viene risvegliato e entra rapidamente in esecuzione grazie ai suoi elevati diritti di esecuzione (VRT basso) si verificherà un ritardo dovuto al caricamento delle pagine che erano state deallocate.

Out Of Memory Killer (OOMK)

- In sistemi molto carichi il PFRA può non riuscire a risolvere la situazione; in tal caso come estrema ratio deve invocare il OOMK, che seleziona un processo e lo elimina (kill).
- OOMK viene invocato quando la memoria libera è molto poca e PFRA non è riuscito a liberare sufficienti PF.
- La funzione più delicata del OOMK si chiama significativamente **select_bad_process()** ed ha il compito di fare una scelta intelligente del processo da eliminare, in base ai seguenti criteri:
 - il processo abbia molte pagine occupate, in modo che la sua eliminazione dia un contributo significativo di pagine libere
 - abbia svolto poco lavoro
 - abbia priorità bassa (tendenzialmente indica processi poco importanti)
 - non abbia privilegi di root (questi processi svolgono funzioni importanti)
 - non gestisca direttamente componenti hardware, per non lasciarle in uno stato inconsistente

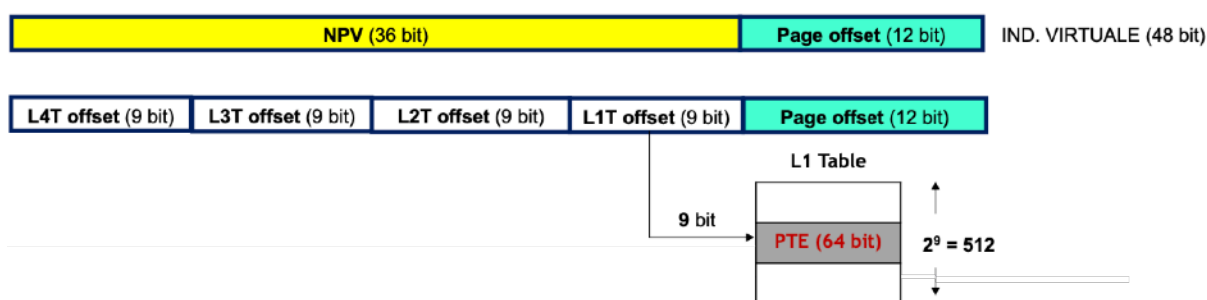
Thrashing

- I meccanismi adottati da Linux sono complessi e il loro comportamento è difficile da predire in diverse condizioni di carico
- La calibratura dei parametri nei diversi contesti può essere uno dei compiti più difficili per l'Amministratore del sistema
- Non è quindi escluso che in certe situazioni si verifichi un fenomeno detto **Thrashing**:
 - il sistema continua a deallocare pagine che vengono nuovamente richieste dai processi,
 - le pagine vengono continuamente scritte e rilette dal disco e nessun processo riesce a progredire fino a terminare e liberare risorse

Per vedere la paginazione nell'architettura Intel x64, vedi inizio appunti

Implementazione della paginazione nell'x64

- Il numero di pagine virtuali (2^{36} pagine) è molto grande: necessario evitare di allocare una **Tabella delle Pagine (TP)** così grande per ogni processo → la TP è organizzata su 4 livelli gerarchici dove i 36 bit di NPV sono suddivisi in 4 gruppi da 9 bit
- Ogni gruppo da 9 bit rappresenta l'offset all'interno di una tabella (directory) contenente 512 righe
- Ogni riga è chiamata **PTE: Page Table Entry** e contiene 64 bit (8 Byte)
- La dimensione di una tabella (directory) è pari a $512 \times 8 \text{ Byte} = 29 \times 23 \text{ Byte} = 4 \text{ K Byte}$ quindi occupa esattamente una pagina

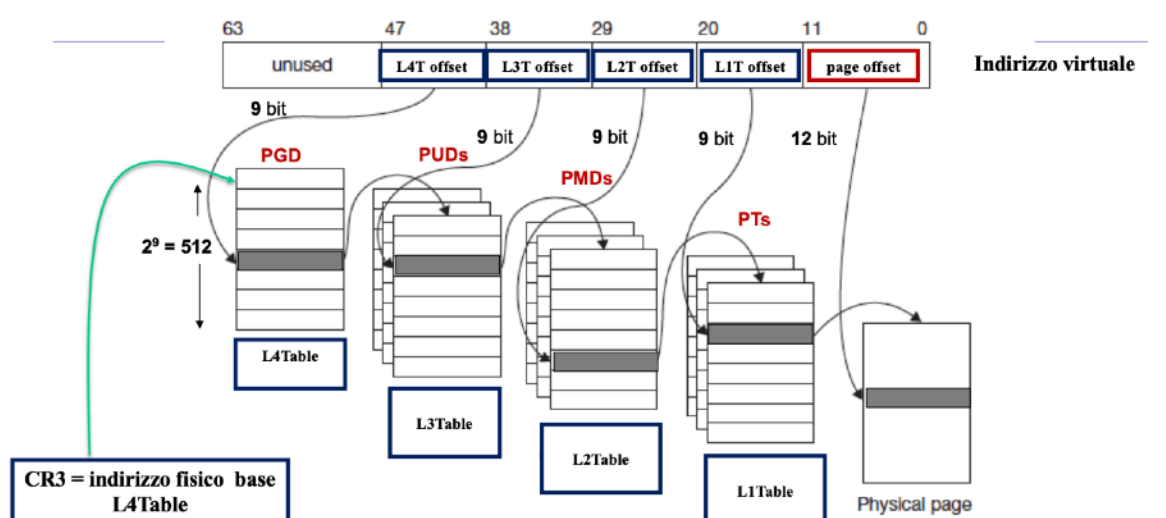


Le tabelle ai vari livelli vengono denominate rispettivamente:

- Livello 4: PGD Page Global Directory
- Livello 3: PUD Page Upper Directory
- Livello 2: PMD Page Middle Directory
- Livello 1: PT Page Table



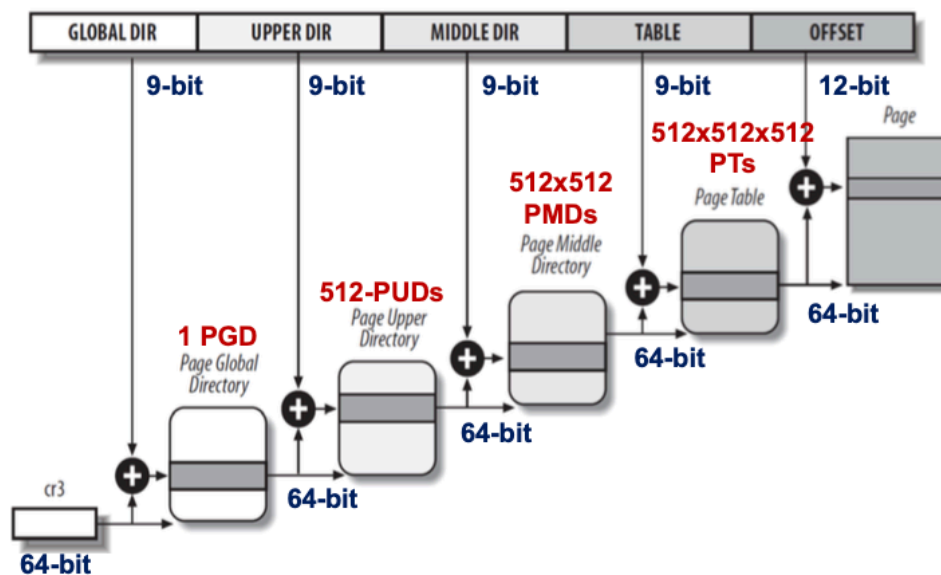
Struttura della tabella delle pagine nell'x64



Page Walk

- L'indirizzo base della Page Global Directory del processo in esecuzione è contenuto nel registro **CR3** del processore.
- Il meccanismo di mappatura di un **NPV** in un **NPF** si basa su **4 livelli di accesso alla pagina**: ad ogni livello si legge una **PTE** che rappresenta l'indirizzo fisico di base della tabella (directory) di livello inferiore.
- Tutti gli indirizzi contenuti nelle entry dei diversi livelli di tabelle (directory) sono **indirizzi fisici**, cioè utilizzati direttamente per accedere alle pagine fisiche.
- Quando il processore deve accedere ad un indirizzo fisico a partire da un indirizzo virtuale, deve attraversare tutta la gerarchia: il **Page Walk** richiede **5 accessi** a memoria fisica per accedere all'indirizzo cercato all'interno della pagina fisica.

Page Walk: richiede 5 accessi a memoria fisica

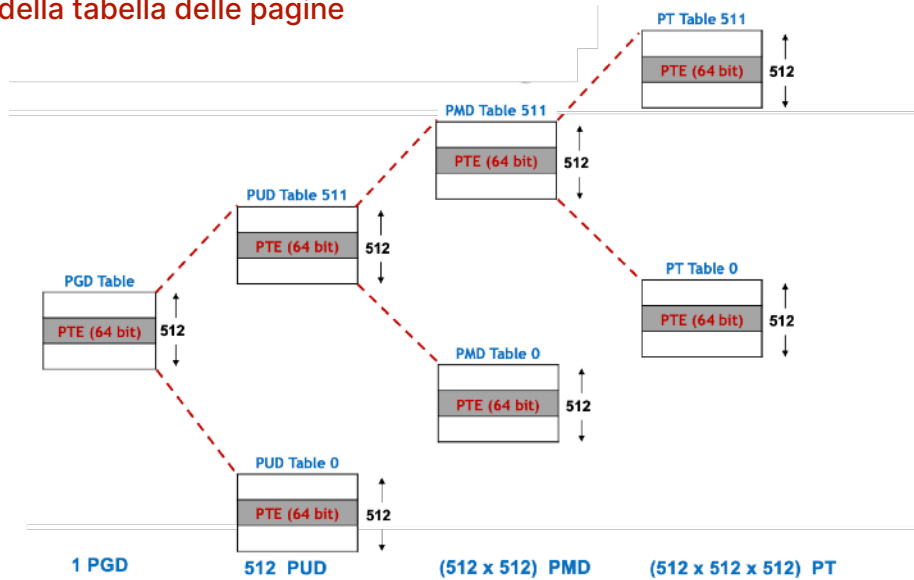


Page Table: Flag

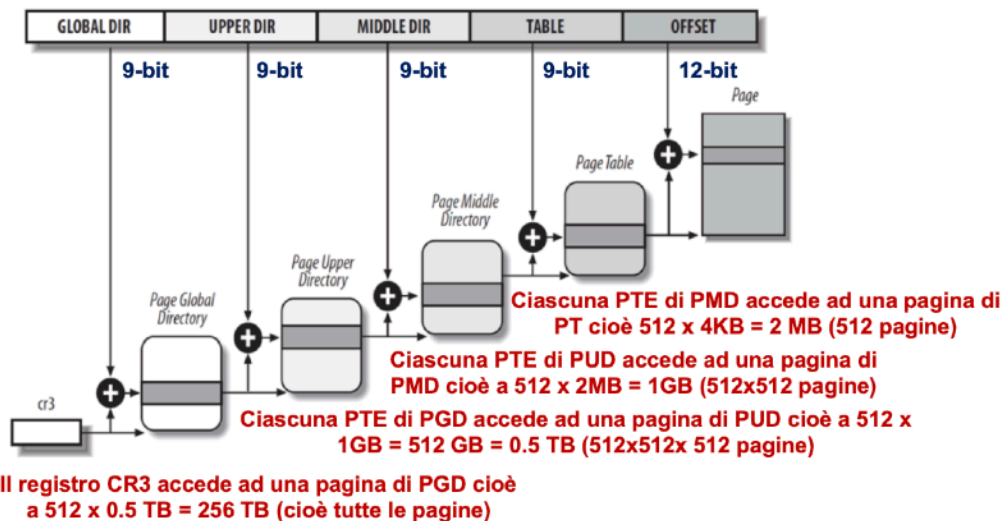
- La PTE di ogni tabella (directory) contiene anche dei flag memorizzati nei 12 bit meno significativi (perché nelle tabelle i bit di offset non sono utilizzati) e nel bit 63 più significativo (non utilizzato)
- Esempio: flag utilizzati per la Page Table (livello 1):

posizione	sigla	nome	interpretazione valori
0	P	present	la PTE ha un contenuto valido
1	R/W	read/write	la pagina è scrivibile: 1=W, 0=ReadOnly
2	U/S	user/supervisor	la pagina appartiene a spazio User: 1=U, 0=S
5	A	accessed	la pagina è stata acceduta (azzerabile da SW)
6	D	dirty	la pagina è stata scritta (azzerabile da SW)
8	G	global page	vedi sotto (gestione TLB)
63	NX	no execute	la pagina non contiene codice eseguibile

Dimensione della tabella delle pagine



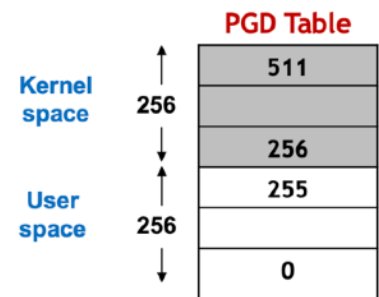
Dimensione di memoria accessibile dalle PTE dei diversi livelli di gerarchia



PGD Table: accesso a Kernel Space e User Space

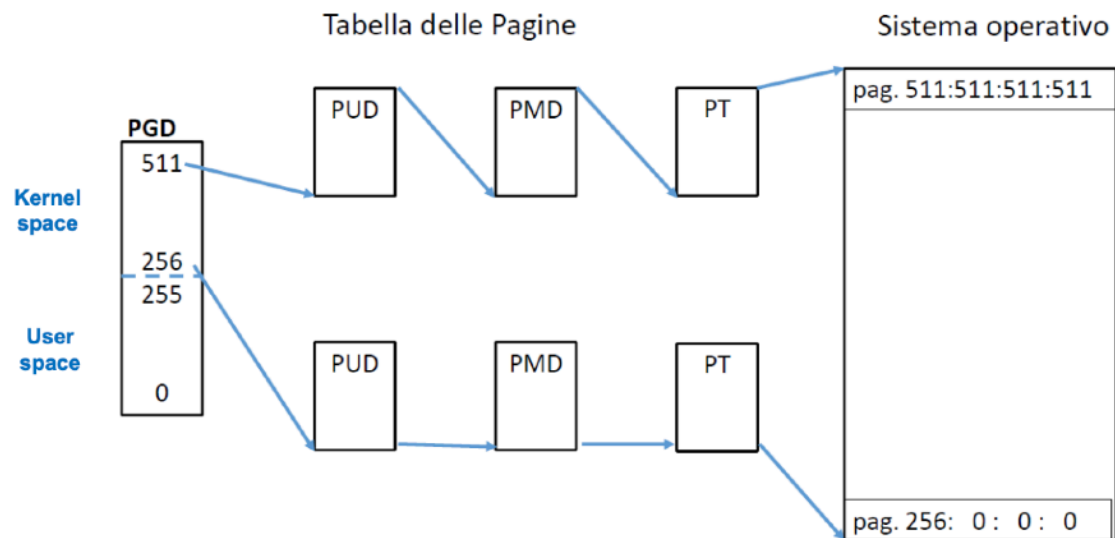
- Ciascuna PTE della Page Global Directory accede ad una pagina di PUD cioè a 0.5 TB
- Metà superiore della PGD Table accede alle tabelle delle pagine del Kernel Space:
256 entries x 0,5 TB = 128 TB Kernel Space

- Si noti che la struttura delle tabelle delle pagine relativa al Kernel Space viene mappata dalla PGD Table di ogni processo, ma **senza ridondanza nell'allocazione di pagine fisiche** cioè tutte le metà superiori della PGD Table di ogni processo puntano alla stessa (unica) struttura fisica (condivisa) relativa al Kernel



- Metà inferiore della PGD Table accede alle tabelle delle pagine dello User Space: 256 entries x 0,5 TB = 128 TB User Space

PGD Table: mappatura del Kernel Space



Paginazione del SO e fase di bootstrap

- Nell'architettura x64, **tutta la memoria (Kernel Space e User Space) è paginata**, quindi anche il SO e la stessa Tabella delle Pagine (TP) sono paginati.
 - La **TP** è puntata dal registro **CR3**, che è unico, quindi non può esistere una **TP** separata per il SO (infatti abbiamo visto che la **PGD Table** mappa sia il Kernel Space sia lo User Space)
- All'avviamento del sistema (**bootstrap**), la **TP** non è ancora inizializzata, quindi la paginazione non è attiva, ma deve esistere un meccanismo che permetta di caricare tale tabella per far partire il sistema
 - Le funzioni di caricamento iniziale funzionano quindi accedendo direttamente alla memoria fisica, senza rilocazione.
 - Quando è stata caricata una porzione della tabella delle pagine adeguata a far funzionare almeno una parte del SO, il meccanismo di paginazione viene attivato e il caricamento completo del Kernel viene terminato.

Page Walk e TLB

- Il processore che genera un indirizzo virtuale, deve attraversare tutta la gerarchia della Tabella delle Pagine (TP) per trovare **l'indirizzo della pagina fisica**:

=> **Page Walk pari a 5 accessi a memoria fisica** per accedere all'indirizzo fisico all'interno della pagina (di cui **4 accessi** a memoria fisica per ogni accesso alle tabelle delle pagine più **1 accesso** dovuto all'offset nella pagina)

- Per velocizzare l'accesso a memoria fisica si introduce un **TLB (Translation Look-aside Buffer)** cioè una cache associativa della TP dove sono copiate le coppie **NPV-NPF** usate più recentemente:
 - La **Memory Management Unit** gestisce il TLB via **hardware** in modo trasparente rispetto al SO
 - Solo nel caso di primo accesso alla pagina (**cold start miss nel TLB**) è necessario percorrere il **Page Walk**.

Sulle slide si trovano gli esempi di:

- **Decomposizione indirizzo iniziale della pila del processo**
- **Pila del processo e primo thread**
- **Dimensione della tabella pagine**
- **Mappatura della memoria fisica**