Giuseppe Pelagatti

Programmazione e Struttura del sistema operativo Linux

Appunti del corso di
Architettura dei Calcolatori e Sistemi Operativi (AXO)

versione 2016

Parte M: La gestione della Memoria

cap. M3 – Gestione della memoria fisica

M.3 Gestione della memoria Fisica

1. Introduzione

La memoria fisica costituisce una risorsa particolarmente critica nei sistemi e la sua gestione ha un grande effetto sulle prestazioni complessive. Purtroppo la teoria in questo campo non è in grado di fornire modelli praticabili e funzionanti nella varietà di contesti in cui opera Linux, quindi sfortunatamente lo sviluppo degli algoritmi di Linux ha dovuto realizzarsi empiricamente, senza un adeguato supporto teorico.

Per questo motivo si tratta anche di un'area nella quale il sistema tende ad essere modificato continuamente ed è difficile mantenere una descrizione aggiornata di ciò che fa. In questo capitolo si descrivono alcuni dei principi di funzionamento di questo sottosistema senza tentare di affrontare i numerosissimi dettagli ed adattamenti empirici che sono stati realizzati nel corso della lunga sperimentazione di uso di Linux.

L'allocazione della memoria può essere suddivisa a 2 livelli:

- allocazione a grana grossa - fornisce ai processi e al sistema le grossa porzioni di memoria sulle quali operare
- allocazione a grana fine - alloca strutture piccole nelle porzioni gestite a grana grossa

Ad esempio, la funzione `malloc( )` permette di allocare in maniera fine dei dati in uno spazio più grande detto heap; la allocazione di spazio allo heap può essere fatta tramite una richiesta `brk( )` al sistema, che costituisce un'allocazione a grana grossa.

La gestione della memoria di cui ci occupiamo è quella a grana grossa, la cui unità di allocazione è la pagina o un gruppo di pagine contiguo.

L'allocazione a grana fine per i processi, cioè la gestione dello heap, è fatta dalle routine del sistema runtime e della libreria del linguaggio C e quindi non riguarda il SO.

L'unica allocazione a grana fine che riguarda il SO, ma che noi non tratteremo, costituisce l'analogo della funzione `malloc( )` per il SO stesso; le corrispondenti funzioni si chiamano `kmalloc()` e `valloc()` nel kernel.

Terminologia: storicamente il termine *swapping* si riferiva alla scaricamento di un intero processo, invece *paging* a quello di una pagina. Linux implementa esclusivamente paging, ma nel codice e nella documentazione utilizza il termine swapping.
E' importante tenere distinte l'allocazione di memoria virtuale da quella fisica: quando un processo esegue una `brk` il suo spazio virtuale viene incrementato ma non viene allocata memoria fisica; solo quando il processo va a scrivere nel nuovo spazio virtuale la necessaria memoria fisica viene allocata.

2. Comportamento di LINUX nella allocazione e deallocazione della memoria

Linux cerca di sfruttare la disponibilità di memoria RAM il più possibile.
Gli usi possibili della RAM sono fondamentalmente i seguenti:

1. tenere in memoria il sistema operativo stesso
2. soddisfare le richieste di memoria dei processi
3. tenere in memoria i blocchi e le pagine letti dal disco: dato che i dischi sono di ordini di grandezza più lenti della RAM conviene sempre tenere in memoria i dati letti dal disco in vista di una sua possibile riutilizzazione successiva

Terminologia: le aree di memoria in cui vengono contenuti blocchi letti dal disco sono tradizionalmente chiamate *buffer*, ma nel codice sorgente e nella documentazione delle versioni recenti di Linux sono chiamate *cache* o *disk cache*, in quanto memoria di transito dal disco (da non confondere con le cache Hardware che costituiscono memorie di transito tra la RAM e la CPU; quest'ultime sono totalmente gestite dall'HW e ignorate dal Sistema Operativo).
Esistono diverse disk cache nel sistema, specializzate per diverse funzioni che vedremo trattando il File System, ma la più importante è la *Page Cache*. Questo argomento verrà ripreso trattando il Filesystem.

In base alle precedenti considerazioni Linux si comporta (inizialmente) nel modo seguente nella gestione della memoria fisica:

- 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 stesso vengono soddisfatte con la massima priorità
- quando un processo richiede memoria, questa gli viene allocata con liberalità, cioè senza particolari limitazioni
- tutti i dati letti dal disco vengono conservati indefinitamente per poter essere eventualmente riutilizzati

In sistemi relativamente scarichi, come sono spesso i PC di uso personale dotati di molta RAM, questo comportamento può durare a lungo, ma ovviamente prima o poi con l'aumento del carico la memoria RAM disponibile può ridursi al punto da richiedere interventi di riduzione delle pagine occupate (**page reclaiming**).

Diremo che una pagina viene **scaricata** se vengono svolte le seguenti operazioni:

- se la pagine è stata letta da disco e non è stata mai modificata (ad esempio, una pagina di codice eseguibile oppure una pagina contenente blocchi di file letti e non modificati) la pagina viene semplicemente resa disponibile per un uso diverso
- se la pagina è stata modificata, cioè se il suo Dirty bit è settato (ad esempio, una pagina contenente nuovi dati da scrivere su un file oppure dati di un processo che sono stati scritti), prima di rendere la pagina disponibile per altri usi la pagine deve essere scritta su disco

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

1. le pagine di page cache non più utilizzate dai processi (ref_count = 1) vengono scaricate; se questo non è sufficiente
2. alcune pagine utilizzate dai processi vengono scaricate; se anche questo non è sufficiente
3. un processo viene eliminato completamente (killed)

Si tenga presente che il sistema deve intervenire prima che la riduzione della RAM sotto una soglia minima renda impossibile qualsiasi intervento, mandando il sistema in blocco.

Possiamo analizzare questo comportamento utilizzando il comando `free`. Il comando free fornisce i dati in Kb per default, ma con le memorie molto grandi disponibili oggi è più leggibile utilizzarlo con l'opzione -m, che fornisce i dati in Megabyte; si faccia però attenzione che in questo caso è presente un arrotondamento che causa alcune mancanze di quadratura.
Le seguenti informazioni sono state prodotte eseguendo il comando `>free -m` su un sistema appena avviato e quindi fortemente scarico:

	total	used	free	shared	buffers	cached
Mem:	1495	749	745	0	25	305
-/+ buffers/cache:		418	1077			
Swap:	1531	0	1531			

Il significato delle diverse colonne della prima riga è il seguente:

1. total: è la quantità di memoria disponibile (è praticamente tutta la memoria fisica installata)
2. used: è la quantità di memoria utilizzata
3. free: è la quantità di memoria ancora utilizzabile (ovviamente, used+free=total)
4. shared non è più usato
5. buffers e cached: è la quantità di memoria utilizzata per i buffer/cache

La seconda riga contiene i valori della prima riga depurati della parte relativa a (buffer + cached); dato che lo spazio allocato ai buffer/cache può essere ridotto a favore dei processi, questo dato indica quanto spazio può essere ulteriormente richiesto per i processi.
In pratica questo significa che nel sistema su un totale di 1495Mb in realtà sono disponibili per i processi 1077Mb ottenibili sommando ai 745 liberi anche i 330 dei buffer/cache (si tenga conto di quanto detto sull'arrotondamento).
Infine, la terza riga indica lo spazio totale, utilizzato e libero sul disco per la funzione di swap.

Partendo da questa situazione possiamo eseguire alcune prove basate sull'esecuzione di programmi che caricano variamente il sistema. Nelle prove seguenti si è proceduto nel modo seguente:

1. è stato eseguito un programma P che ha causato dei cambiamenti nell'assetto della memoria

2. dopo la terminazione del programma è stato utilizzato il comando free; pertanto quando il comando viene eseguito le pagine del processo che ha eseguito P sono state rilasciate

Il primo programma ha eseguito una lunga scrittura di file. La seguente figura mostra che il sistema ha tenuto in memoria i blocchi da scrivere aumentando quindi il valore di buffer/cache (**928**), rendendo apparentemente la memoria libera pericolosamente scarsa (71), ma in realtà la memoria utilizzabile dai processi è rimasta quasi inalterata (1000). Si noti che i buffer allocati per il file sono rimasti occupati anche dopo la terminazione del processo (il sistema infatti non sa che il file non verrà utilizzato da altri processi).

	total	used	free	shared	buffers	cached
Mem:	1495	1424	71	0	**15**	**913**
-/+ buffers/cache:		495	1000			
Swap:	1531	0	1531			

A conferma di quanto detto, eseguiamo, partendo dalla situazione appena creata, un processo che consuma molta memoria; per forzare il consumo di memoria il programma è stato eseguito disabilitando lo swapping delle pagine. Dopo la sua esecuzione il comando free indica che le pagine di cache sono state scaricate per allocarle al processo (il valore 0 in total nella terza riga indica che lo swapping è stato disabilitato); quando il processo è terminato, le pagine sono ritornate libere (il sistema è ritornato a una situazione molto simile a quella iniziale.

	total	used	free	shared	buffers	cached
Mem:	1495	542	953	0	**1**	**40**
-/+ buffers/cache:		499	**995**			
Swap:	0	0	0			

Infine consideriamo un processo che richiede memoria in maniera inarrestabile, anche questo eseguito con swap disabilitato. Il programma è mostrato in figura 2.1; esso entra in un ciclo infinito nel quale continua a richiedere blocchi da 1Mb stampando un output per ogni allocazione; la sua esecuzione fornisce il seguente risultato:

```
Allocated    1 MB
...
Allocated 935 MB
Killed
```

Il processo è riuscito ad allocarsi 935 Mb, poi è stato eliminato, cioè si è applicato il terzo livello di riduzione del carico di memoria indicato sopra. La funzione di Linux che esegue questa operazione è chiamata significativamente **Out Of Memory Killer** (**OOMK**). La quantità di memoria che il processo si è allocato si è avvicinata pericolosamente alla quantità disponibile (935 + il codice e gli altri dati del processo contro i 995 liberi o potenzialmente liberabili).

Eseguendo lo stesso processo con lo swapping abilitato, esso arriva ad allocarsi 2490 Mb prima di essere eliminato dal OOMK. Sommando la dimensione dello swap file ai 935MB della prova precedente si ottiene 2466, che spiega abbastanza bene questo valore (quasi il doppio della dimensione dell'intera memoria fisica).
A questo punto il comando free fornisce il seguente risultato.

	total	used	free	shared	buffers	cached
Mem:	1495	259	1235	0	0	43
-/+ buffers/cache:		215	1280			
Swap:	1531	321	1210			

Dato che il comando free è stato dato dopo la terminazione del programma, *i 321 blocchi in swap appartengono a processi diversi, che sono stati obbligati a cedere pagine al nuovo processo*. Un processo vorace di memoria può quindi buttare fuori memoria altri processi.

Infine, per vedere il file di swap pieno modifichiamo il programma in modo che si allochi 2440 Mb (in base alla prova precedente il OOMK non dovrebbe quindi intervenire) e alla fine si ponga in sleep per un

certo tempo, ***in modo da poter eseguire il commando free con il processo ancora vivo***. Il risultato mostra che il file di swap si è riempito fino quasi al limite della sua capacità. Si osservi che la cache è stata molto svuotata, ma non azzerata e che sono rimaste comunque delle pagine libere.

	total	used	free	shared	buffers	cached
Mem:	1495	1430	65	0	0	10
-/+ buffers/cache:		1419	75			
Swap:	1531	**1529**	2			

Aumentando ulteriormente le pagine richieste anche questo processo viene eliminato dal OOMK

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

/* funzione che alloca 1Mb ad ogni iterazione del ciclo while */
void alloc_memory( ){
        int mb = 0;
        char* buffer;
        while((buffer=malloc(1024*1024)) != NULL) {
        /* la funzione memset(s, c, n) riempie n bytes dell'area s con il valore c
        *  la sua invocazione è necessaria per forzare l'allocazione di memoria fisica
        *  perché malloc alloca solo la memoria virtuale  */
        memset(buffer, 0, 1024*1024);
                mb++;
                printf("Allocated %d MB\n", mb);
        }
}

int main(int argc, char** argv) {
        int i, j, k, stato;
        pid_t pid;
        alloc_memory();
   return 0;
}
```

Figura 2.1

3. Allocazione della memoria

L'unità di base per l'allocazione della memoria è la pagina, ma, se possibile, Linux cerca di allocare blocchi più grandi di pagine contigue e di mantenere la memoria il meno frammentata possibile. Questa scelta può apparire inutile, perché la paginazione permette di operare su uno spazio virtuale continuo anche se le corrispondenti pagine fisiche non lo sono, ma ha svariate motivazioni, ad esempio:

- 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 libera risulta più compatta

L'allocazione a blocchi di pagine contigue si basa sul seguente meccanismo:

- La memoria è suddivisa in grandi blocchi di pagine contigue, la cui dimensione è sempre una potenza di 2, detta **Ordine** del blocco (una costante MAX_ORDER definisce la massima dimensione dei blocchi)
- Per ogni ordine esiste una lista che collega tutti i blocchi di quell'ordine
- Le richieste di allocazione indicano l'ordine del blocco che desiderano
- Se un blocco dell'ordine richiesto è disponibile, questo viene allocato, altrimenti
    - un blocco doppio viene diviso in 2 una volta

- I due nuovi blocchi sono detti **buddies** (buddy significa compagno) l'uno dell'altro. Uno viene allocato, l'altro è libero. Questa relazione viene rappresentata dalle strutture dati utilizzate per l'allocazione
- Se necessario, la suddivisione procede più volte.
- Quando un blocco viene liberato il suo buddy viene analizzato e, se è libero, i due vengono riunificati, ricreando un blocco più grande

All'inizio, quando la memoria è molto libera, vengono costruite poche liste con blocchi molto grandi; progressivamente si alimentano le liste di blocchi più piccoli dovute agli "scarti" delle divisioni in 2, ma la ricostruzione dei blocchi quando vengono liberati tende a mantenere sotto controllo la frantumazione della memoria.

4. Deallocazione della Memoria fisica – determinazione delle pagine da scaricare

Fino a quando la quantità di RAM libera è sufficientemente grande Linux alloca la memoria senza svolgere particolari controlli. Questa scelta è orientata ad utilizzare il meglio possibile tutta la RAM disponibile. La memoria richiesta dai processi e dalle disk cache cresce quindi continuamente; in particolare:

- quella destinata ai processi viene rilasciata quando un processo termina, quindi cresce a causa dell'aumento del numero di processi o della crescita delle pagine allocate a ogni processo, ma poi può decrescere spontaneamente
- quella destinata alle disk cache cresce sempre, perché le disk cache non hanno un criterio per stabilire che un certo dato su disco non verrà più riutilizzato

In questo modo a un certo punto può accadere che la quantità di memoria libera scenda a un livello critico (low memory) che porta a far intervenire la procedura di liberazione di pagine **PFRA (Page Frame Reclaiming Algorithm)**. Si noti che PFRA ha bisogno di allocare memoria essa stessa, quindi, per evitare che il sistema raggiunga una saturazione della memoria non più risolvibile e vada in crash, l'attivazione di PFRA deve avvenire prima di tale momento, ovvero il livello di low memory deve esssere tale da garantire il funzionamento di FPRA. Inoltre, per ridurre il numero di attivazioni di PFRA, PFRA cerca di riportare il numero di pagine libere a un livello superiore al minimo, che chiameremo **maxFree**.
Il meccanismo utilizzato da PFRA per scegliere quali pagine liberare si basa sui principi di LRU, cioè sulla scelta delle pagine meno utilizzate recentemente.

Il problema della liberazione delle pagine può essere suddiviso in sottoproblemi ben distinti:

1. la scelta delle pagine da liberare, che trattiamo in questo capitolo, e che a sua volta può essere suddiviso in
    1. determinazione di ***quando*** e ***quante*** pagine è necessario liberare
    2. determinazione di ***quali*** pagine liberare; il meccanismo utilizzato da PFRA per scegliere quali pagine liberare si basa sui principi di LRU, quindi sulla scelta delle pagine non utilizzate da più tempo, e può essere a sua volta suddiviso in 2 sottoproblemi:
        1. ***mantenimento dell'informazione*** relativa all'accesso alle pagine
        2. ***scelta delle pagine meno utilizzate*** in base a tale informazione
2. il meccanismo di ***swapping***, cioè l'effettivo scaricamento delle pagine con liberazione della memoria (***swapout***) e l'eventuale ricaricamento di pagine swappate in memoria (***swapin***); questo argomento sarà trattato nel prossimo capitolo.

4.1 Determinazione di *quando* e *quante* pagine è necessario liberare

Per affrontare questo argomento iniziamo con la definizione dei seguenti parametri:

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

Il PFRA è invocato nei casi seguenti:

1. Invocazione diretta da parte di un processo che richiede **requiredPages** pagine di memoria se **`(freePages - requiredPages) < minFree`**, cioè se l'allocazione delle pagine richieste porterebbe la memoria fisica sotto il livello di guardia
2. Attivazione periodica tramite **`kswapd`** (`kernel swap daemon)`,una funzione che viene attivata periodicamente e invoca PFRA se **`freePages < maxFree`**

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

```
toFree = maxFree - freePages + requiredPages
```

In condizioni di carico leggero la differenza tra **minFree** e **maxFree** permette al kswapd di mantenere sufficiente memoria libera, causando rare attivazioni dirette di PFRA, ma in situazioni di forte carico, quando `kswapd` non è in grado di tenere il passo con le richieste di memoria da parte di un processo che la consuma voracemente, l'attivazione diretta diventa più frequente.

4.2 Determinazione di *quali* pagine liberare

Lo scopo fondamentale di PFRA è di selezionare una pagina occupata da liberare. Dal punto di vista di PFRA i tipi di pagine sono:

1. Pagine non scaricabili
    a. pagine statiche del SO dichiarate non scaricabili
    b. pagine allocate dinamicamente dal S.O
    c. pagine appartenenti alla sPila dei processi
2. Pagine mappate sul file eseguibile dei processi che possono essere scaricate senza mai riscriverle (codice, costanti)
3. Pagine che richiedono l'esistenza di una Swap Area su disco per essere scaricate
    a. pagine dati
    b. pagine della uPila
    c. pagine dello Heap
4. Pagine che sono mappate su un file: pagine appartenenti ai buffer/cache

Si tenga presente che la scelta della pagina da liberare costituisce uno degli algoritmi più delicati nel funzionamento del SO.

4.2.1 Mantenimento dell'informazione relativa all'accesso alle pagine

Il meccanismo utilizzato da PFRA si basa sui principi di LRU, ma ne adatta molti aspetti alla effettiva implementabilità e alle esigenze emerse sperimentalmente dall'impiego in molti contesti diversi.

Esistono 2 liste globali, dette LRU list, che collegano tutte le pagine appartenenti ai Processi:

- ***active list***: contiene tutte le pagine che sono state accedute recentemente e non possono essere scaricate; in questo modo PFRA non dovrà scorrerle per scegliere un pagina da scaricare
- ***inactive list***: contiene le pagine inattive da molto tempo che sono quindi candidate per essere scaricate

In ambedue le liste le pagine sono tenute in ordine di invecchiamento (approssimato, come vedremo) tramite spostamento delle pagine da una lista all'altra per tenere conto degli accessi alla memoria.

L'obiettivo dello spostamento delle pagine all'interno e tra le due liste è di *accumulare le pagine meno utilizzate in coda alla inactive, mantenendo nella active le pagine più utilizzate di recente (Working Set) di tutti i processi*.

Rispetto alla struttura di LRU teorico abbiamo una differenza fondamentale e una serie di aggiustamenti secondari.

La differenza fondamentale è dovuta all'impossibilità di implementare rigorosamente LRU con l'hardware disponibile. Dato che l'x64 non tiene traccia del numero di accessi alla memoria, Linux realizza un'approssimazione basata sul **bit di accesso A** presente nel TLB, che viene posto a 1 dal x64 ogni volta che la pagina viene acceduta, e può essere azzerato esplicitamente dal sistema operativo.

La seguente descrizione dell'algoritmo è estremamente semplificata rispetto a quello reale, che è anche soggetto a modifiche frequenti nell'evoluzione del sistema, ma ne riflette la logica generale.

Ad ogni pagina viene associato un flag, **ref** (referenced), oltre al bit di accesso **A** settato dall'Hardware. 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 Controlla_liste, attivata periodicamente da kswapd, che rappresenta una sintesi semplificata rispetto alle funzioni reali del sistema ed esegue una scansione di ambedue le liste:

1. **Scansione della active dalla coda** spostando eventualmente alcune pagine alla inactive
2. **Scansione della inactive dalla testa** (escluse le pagine appena inserite provenienti dalla active) spostando eventualmente alcune pagine alla active

Ogni pagina viene quindi processata una sola volta.
Alla funzione Controlla_liste dobbiamo aggiungere:

- funzioni che caricano nuove pagine:
  - le pagine richieste da un processo vengono poste in testa alla active con ref=1
  - le pagine di un nuovo processo appena creato possiedono nelle LRU list lo stesso ref di quelle del padre e sono poste in testa alla active o inactive nello stesso ordine in cui vi compaiono quelle del processo padre
- funzioni che eliminano dalle liste pagine non più residenti (swapped out) oppure definitivamente eliminate (exit del processo)

La funzione periodica Controlla_liste esegue le seguenti operazioni sulle pagine delle due 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

Esempio/Esercizio 1
Negli esercizi per indicare sinteticamente le pagine con ref=1 le scriviamo in lettere maiuscole, quelle con ref=0 in lettere minuscole.
Si consideri il seguente stato iniziale delle liste:

active: `PP4, PP3, PP2, PC0` inactive: `pp1, pp0`

Mostrare l'evoluzione delle liste per 3 attivazioni di kswapd sapendo che le pagine accedute (cioè con bit A=1) ad ogni attivazione sono sempre solo `pc0, pp4, pp1`

Soluzione

Attivazioni	**Active**	**Inactive**
stato iniziale	`PP4, PP3, PP2, PC0`	`pp1, pp0`
1	`PP4, PC0, pp3, pp2,`	`PP1, pp0`
2	`PP4, PC0, pp1,`	`PP3, PP2, pp0`
3	`PP4, PC0, PP1`	`pp3, pp2, pp0`

Si noti che la pagina PP2, inizialmente con A settato, ha impiegato 3 cicli per finire (quasi) in fondo alla inactive; durante questi 3 cicli non è mai stata acceduta. Se fosse stata acceduta durante questi 3 cicli avrebbe recuperato posizioni; ad esempio nel ciclo 2 un accesso a PP2 l'avrebbe riportata nella active. Le liste hanno raggiunto uno stato stabile, nel senso che ulteriori esecuzioni di kswapd con gli stessi accessi a pagina non le modificherebbe ulteriormente.

L'algoritmo raggiunge dunque l'obiettivo di accumulare le pagine meno utilizzate in fondo alla inactive, ma con una certa lentezza che permette di tenere conto di più di una singola valutazione del bit di accesso.

In generale, pagine molto attive (ref=1 nella active) richiedono 2 valutazioni negative prima di passare alla inactive e pagine inattive (ref=0 nella inactive) richiedono 2 valutazioni positive per ritornare nella active.

Attenzione: la simulazione precedente è stata svolta ipotizzando che lo stato iniziale del TLB fosse già conforme all'elenco di pagine accedute, come possiamo vedere in Figura 4.1a, che mostra lo stato del TLB iniziale oltre alle liste LRU; in figura 4.1b è mostrato lo stesso esempio partendo da un diverso stato del TLB – come si vede in questo caso sono state necessarie 4 esecuzioni di kswapd per raggiungere lo stato "stabile", perché la prima attivazione è servita a modificare il TLB in base agli accessi.

Esempio 1 bis (con TLB)

Stato iniziale (con TLB come nell'esempio precedente)

```
____STATO del TLB_____________________________________________________
      Pc0 : 01 -  0: 1:   ||   Pp0 : 02 -  1: 0:   ||
      Pp1 : 03 -  1: 1:   ||   Pp2 : 04 -  1: 0:   ||
      Pp3 : 05 -  1: 0:   ||   Pp4 : 06 -  1: 1:   ||
            -----         ||         -----         ||
            -----         ||         -----         ||
LRU ACTIVE:   PP4, PP3, PP2, PC0,         LRU INACTIVE: pp1, pp0,

Accessi solo a pc0, pp4, pp1
LRU ACTIVE:   PP4, PC0, pp3, pp2,         LRU INACTIVE: PP1, pp0,
LRU ACTIVE:   PP4, PC0, pp1,              LRU INACTIVE: PP3, PP2, pp0,
LRU ACTIVE:   PP4, PC0, PP1,              LRU INACTIVE: pp3, pp2, pp0,
```

(a)

Stato iniziale (con TLB diverso dall'esempio precedente)

```
____STATO del TLB_____________________________________________________
      Pc0 : 01 -  0: 1:   ||   Pp0 : 02 -  1: 0:   ||
      Pp1 : 03 -  1: 1:   ||   Pp2 : 04 -  1: 1:   ||
      Pp3 : 05 -  1: 1:   ||   Pp4 : 06 -  1: 1:   ||
            -----         ||         -----         ||
            -----         ||         -----         ||
LRU ACTIVE:   PP4, PP3, PP2, PC0,         LRU INACTIVE: pp1, pp0,

Accessi solo a pc0, pp4, pp1
____STATO del TLB_____________________________________________________
      Pc0 : 01 -  0: 1:   ||   Pp0 : 02 -  1: 0:   ||
      Pp1 : 03 -  1: 1:   ||   Pp2 : 04 -  1: 0:   ||
      Pp3 : 05 -  1: 0:   ||   Pp4 : 06 -  1: 1:   ||
            -----         ||         -----         ||
            -----         ||         -----         ||
LRU ACTIVE:   PP4, PP3, PP2, PC0,         LRU INACTIVE: PP1, pp0,
LRU ACTIVE:   PP4, PC0, pp3, pp2, pp1,    LRU INACTIVE: pp0,
LRU ACTIVE:   PP4, PC0, PP1,              LRU INACTIVE: PP3, PP2, pp0,
LRU ACTIVE:   PP4, PC0, PP1,              LRU INACTIVE: pp3, pp2, pp0,
```

(b)

Figura 4.1

Nell'esercizio 1 abbiamo ipotizzato che una serie di esecuzioni di kswapd trovassero le stesse pagine accedute. Per rappresentare situazioni più dinamiche e integrare il meccanismo di controllo delle liste LRU con gli altri aspetti della memoria definiamo il seguente tipo di evento composto:

Evento: **Read(lista pagine lette) - Write(lista pagine scritte) - N kswapd**

Questo tipo di evento viene interpretato nel modo seguente:

1. Ripeti per N volte le operazioni di lettura/scrittura/kswapd
2. esegui un'ultima volta le sole operazioni di lettura/scrittura

Nota bene:

- se N=0 vengono eseguite una volta le operazioni di lettura/scrittura, con le regole già viste nel capitolo relativo allo spazio virtuale dei processi (M2)
- se la lista di operazioni è vuota, viene eseguito N volte kswapd

Questa interpretazione serve a rendere univoco l'effetto sul TLB: il TLB finale avrà il valore lasciato dall'ultima esecuzione delle operazioni di lettura/scrittura, senza kswapd.
Se durante le operazioni di lettura/scrittura vengono allocate pagine in memoria, queste vengono aggiunte in testa alla lista active con ref=1 nell'ordine in cui vengono allocate.

Esempio/Esercizio 2
Si consideri il seguente stato iniziale del TLB e delle lrulist:

```
LRU ACTIVE:   PP5, PP4, PP3, PP2, PP1, PC2, PP0, PC0,     LRU INACTIVE: -
STATO del TLB
      Pc0 : 0 -  0: 1:   ||   Pp0 : 1 -  1: 1:   ||
      Pc2 : 2 -  0: 1:   ||   Pp1 : 3 -  1: 1:   ||
      Pp2 : 4 -  1: 1:   ||   Pp3 : 5 -  1: 1:   ||
      Pp4 : 6 -  1: 1:   ||   Pp5 : 7 -  1: 1:   ||
            VOID         ||         VOID         ||
```

Si mostri lo stato delle lrulist e del TLB dopo ognuno dei seguenti eventi:

Evento 1: Read(pc2) - Write(pp1,pp2,pp3,pp4,pp5) - 1 kswapd ====
Evento 2: Read(pc2) - Write(pp1,pp2,pp3) - 1 kswapd
Evento 3: Read(pc1) - Write(pp1) - 1 kswapd
Evento 4: Read(pc1) - Write(pp1) - 1 kswapd

Soluzione: vedi figura 4.3

Ogni volta che interviene kswapd lo stato del TLB da osservare è quello prodotto dall'evento precedente; nell'esempio, considerando la pagina Pc0:

- il primo intervento di kswapd azzera il bit di accesso in TLB, ma non modifica la posizione di Pc0 nella lista, perché il bit di accesso era 1;
- tutti i successivi accessi a Pc0 trovano il bit di accesso azzerato e quindi spostano progressivamente Pc0 verso la coda della inactive
- dopo 4 interventi di kswapd Pc0 è finita in fondo alla inactive

Come già detto la descrizione dell'algoritmo è molto semplificata rispetto a quello reale; l'algoritmo reale tra l'altro cerca di mantenere un certo rapporto tra la dimensione della active e quella della inactive.

```
=== Evento 1: Read(pc2) - Write(pp1,pp2,pp3,pp4,pp5) - 1 kswapd ====
LRU ACTIVE:   PP5, PP4, PP3, PP2, PP1, PC2, PP0, PC0,
LRU INACTIVE:
STATO del TLB
      Pc0 : 0 -  0: 0:   ||   Pp0 : 1 -  1: 0:   ||
      Pc2 : 2 -  0: 1:   ||   Pp1 : 3 -  1: 1:   ||
      Pp2 : 4 -  1: 1:   ||   Pp3 : 5 -  1: 1:   ||
      Pp4 : 6 -  1: 1:   ||   Pp5 : 7 -  1: 1:   ||
            VOID         ||         VOID         ||
=== Evento 2: Read(pc2) - Write(pp1,pp2,pp3) - 1 kswapd ====
LRU ACTIVE:   PP5, PP4, PP3, PP2, PP1, PC2, pp0, pc0,
LRU INACTIVE:
STATO del TLB
      Pc0 : 0 -  0: 0:   ||   Pp0 : 1 -  1: 0:   ||
      Pc2 : 2 -  0: 1:   ||   Pp1 : 3 -  1: 1:   ||
      Pp2 : 4 -  1: 1:   ||   Pp3 : 5 -  1: 1:   ||
      Pp4 : 6 -  1: 0:   ||   Pp5 : 7 -  1: 0:   ||
            VOID         ||         VOID         ||
=== Evento 3: Read(pc1) - Write(pp1) - 1 kswapd ====
LRU ACTIVE:   PC1, PP3, PP2, PP1, PC2, pp5, pp4,
LRU INACTIVE: PP0, PC0,
STATO del TLB
      Pc0 : 0 -  0: 0:   ||   Pp0 : 1 -  1: 0:   ||
      Pc2 : 2 -  0: 0:   ||   Pp1 : 3 -  1: 1:   ||
      Pp2 : 4 -  1: 0:   ||   Pp3 : 5 -  1: 0:   ||
      Pp4 : 6 -  1: 0:   ||   Pp5 : 7 -  1: 0:   ||
      Pc1 : 8 -  0: 1:   ||         VOID         ||
=== Evento 4: Read(pc1) - Write(pp1) - 1 kswapd ====
LRU ACTIVE:   PC1, PP1, pp3, pp2, pc2,
LRU INACTIVE: PP5, PP4, pp0, pc0,
STATO del TLB
      Pc0 : 0 -  0: 0:   ||   Pp0 : 1 -  1: 0:   ||
      Pc2 : 2 -  0: 0:   ||   Pp1 : 3 -  1: 1:   ||
      Pp2 : 4 -  1: 0:   ||   Pp3 : 5 -  1: 0:   ||
      Pp4 : 6 -  1: 0:   ||   Pp5 : 7 -  1: 0:   ||
      Pc1 : 8 -  0: 1:   ||         VOID         ||
```

Figura 4.3

4.2.2 Scaricamento delle pagine della lista inattiva

Le pagine da scaricare vengono determinate in base alle seguenti regole:

1. Prima di tutto vengono scaricate le pagine appartenenti alla PageCache non più utilizzate da nessun processo, in ordine di NPF
2. Poi vengono scelte le pagine virtuali della inactive, partendo dalla coda, ma tenendo conto della condivisione nel modo seguente: una pagina virtuale viene considerata scaricabile *solo se tutte le pagine condivise sono più invecchiate di lei* (ovvero, se una pagina fisica è condivisa tra molte pagine virtuali viene considerata scaricabile solo quando nella scansione della inactive si sono trovate tutte le pagine che la condividono)

Esempio/Esercizio 3
Si consideri il seguente stato iniziale della memoria e delle lrulist. maxFree = 3, minFree = 2.

```
____MEMORIA FISICA____(pagine libere: 3)_____________________________
      00 : <ZP>                ||  01 : Pc1/<XX,1>          ||
      02 : Pp1                 ||  03 : Pp0 D               ||
      04 : <G,0>               ||  05 : <G,1>               ||
      06 : <G,2>               ||  07 : <G,3>               ||
      08 : Pp2                 ||  09 : ----                ||
      10 : ----                ||  11 : ----                ||
LRU ACTIVE:   PP2, PP1, PC1,
LRU INACTIVE: pp0,
```

Mostrare lo stato della memoria e delle lrulist dopo i seguenti 4 eventi consecutivi:

1. il processo P alloca le pagine di pila pp3, pp4 e pp5,
2. il processo P esegue una fork(R),
3. il processo P accede alle pagine pc1, pp1, pp2 e pp4 mentre kswapd interviene 4 volte
4. il processo P esegue sbrk(5) e poi scrive le pagine pd0, pd1, pd2, pd3 (indicare gli NPV delle pagine della inactive che vengono scaricate)

Soluzione
Evento 1: write(pp3, pp4 e pp5)
Le richieste di pagine e le liberazioni avvengono nel seguente ordine:

- pp3 viene allocata in pagina 9 (free=2)
- richiesta una pagine per pp4 → interviene PFRA, required = 1, da liberare = 3 – 2 + 1 = 2, vengono liberate 2 pagine di page cache, cioè le pagine **4 e 5**, **pp4** viene allocata in pagina 4
- **pp5** viene allocata in pagina 5

A questo punto lo stato della memoria e delle lrulist è il seguente:

```
____MEMORIA FISICA____(pagine libere: 2)_____________________________
      00 : <ZP>                ||  01 : Pc1/<XX,1>          ||
      02 : Pp1                 ||  03 : Pp0 D               ||
      04 : Pp4                 ||  05 : Pp5                 ||
      06 : <G,2>               ||  07 : <G,3>               ||
      08 : Pp2                 ||  09 : Pp3                 ||
      10 : ----                ||  11 : ----                ||
LRU ACTIVE:   PP5, PP4, PP3, PP2, PP1, PC1   LRU INACTIVE: pp0,
```

Evento 2: il processo P esegue una fork(R)
Le richieste di pagine e le liberazioni avvengono nel seguente ordine:

- fork condivide inizialmente tutte le pagine tra P ed R
- fork richiede una pagina per la pagina in cima alla pila del processo padre (pp5)
- dato che freePages = 2, viene invocato PFRA con requiredPages = 1 → toFree = 3 – 2 + 1 = 2,
- vengono quindi liberate le pagine 6 e 7 della page cache
- pp5 viene allocata in pagina 6
- tutte le nuove pagine sono inserite progressivamente in testa alla active

Lo stato della memoria a questo punto è il seguente (le lrulist sono inalterate):

```
____MEMORIA FISICA____(pagine libere: 3)_____________________________
      00 : <ZP>                ||  01 : Pc1/Rc1/<XX,1>      ||
      02 : Pp1/Rp1             ||  03 : Pp0/Rp0 D           ||
      04 : Pp4/Rp4             ||  05 : Rp5 D               ||
      06 : Pp5                 ||  07 : ----                ||
      08 : Pp2/Rp2             ||  09 : Pp3/Rp3             ||
      10 : ----                ||  11 : ----                ||

LRU ACTIVE:  RP5, RP4, RP3, RP2, RP1, RC1, PP5, PP4, PP3, PP2, PP1, PC1,
LRU INACTIVE: rp0, pp0
```

Evento 3: read(pc1, pp1, pp2 e pp4), 4 kswapd
4 esecuzioni di kswapd portano in inactive tutte le pagine eccetto quelle accedute; le pagine di R vengono spostate prima di quelle di P, perché le pagine di P sono inizialmente in stato di accedute nel TLB

```
LRU ACTIVE:   PP4, PP2, PP1, PC1,
LRU INACTIVE: pp5, pp3, rp5, rp4, rp3, rp2, rp1, rc1, rp0, pp0
```

Evento 4: sbrk(5), write(pd0, pd1, pd2, pd3) (indicare gli NPV delle pagine della inactive che vengono scaricate)
sbrk(5) non ha nessun effetto sulla memoria, invece la scrittura delle pagine causa una serie di richieste di allocazione nel seguente ordine:

- la pagina pd0 viene allocata in pagina 7 (freePages = 2)
- alla successiva richiesta di una pagina interviene PFRA per liberarne 2
- scansione dalla coda della inactive:
  - pp0 non viene scaricata, perché è in pagina condivisa con pagina meno invecchiata
  - rp0 viene scaricata insieme a pp0 → liberata pagina 3 (freePages = 3)
  - rc1, rp1, rp2, rp3, rp4 non vengono scaricate
  - rp5 viene scaricata, liberando pagina 5 (freePages = 4)
- pd1 viene allocato in pagina 3 (freePages = 3)
- pd2 viene allocato in pagina 5 (freePages = 2)
- alla successiva richiesta di una pagina interviene PFRA per liberarne 2:
  - pp3 viene scaricata insieme a rp3, che è più vecchia, liberando pagina 9 (freePages = 3)
  - pp5 viene scaricata, liberando pagina 6 (freePages = 4)
- pd3 viene allocata in pagina 6 (freePages = 3)

Lo stato della memoria e delle lrulist è ora il seguente:

```
____MEMORIA FISICA____(pagine libere: 3)_____________________________
      00 : <ZP>                ||  01 : Pc1/Rc1/<XX,1>      ||
      02 : Pp1/Rp1             ||  03 : Pd1                 ||
      04 : Pp4/Rp4             ||  05 : Pd2                 ||
      06 : Pd3                 ||  07 : Pd0                 ||
      08 : Pp2/Rp2             ||  09 : ----                ||
      10 : ----                ||  11 : ----                ||
LRU ACTIVE:   PD3, PD2, PD1, PD0, PP4, PP2, PP1, PC1,
LRU INACTIVE: rp4, rp2, rp1, rc1,
```

Complessivamente sono state scaricate le seguenti pagine: Pp0/Rp0 , Rp5, Pp3/Rp3 , Pp5 ; nel prossimo capitolo analizziamo come tale scaricamento è avvenuto.

5. Il meccanismo di 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 Filesystem). 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 Slots***, ognuno di dimensione uguale a una pagina. Indicheremo con **SWPN** gli identificatori dei Page Slot (tali identificatori devono essere in generale delle coppie <SwapArea, PageSlot> ma negli esempi, grazie all'ipotesi di avere una sola Swap Area, il SWPN può essere semplicemente un numero).
La struttura di ogni Swap Area è descritta da un descrittore; in particolare, tale descrittore contiene un contatore per ogni Page Slot detto ***swap_map_counter***; tale contatore verrà utilizzato per tener traccia del numero di PTE che riferiscono la pagina fisica swappata (ovvero il numero di pagine virtuali condivise in tale pagina fisica).

In linea di principio le regole dello swapping sono le seguenti:

- quando PFRA chiede di scaricare una pagina fisica (**swap_out**):
  - viene allocato un Page Slot
  - la pagina fisica viene copiata nel Page Slot e liberata (**operazione su disco**);
  - 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 del Page Slot al posto del NPF e il bit di presenza viene azzerato
- quando un processo accede a una pagina virtuale swappata:
  - si verifica un page fault
  - il gestore del page fault attiva la procedura di caricamento (**swap_in**)
  - viene allocata una pagina fisica in memoria
  - il Page Slot indicato dalla PTE viene copiato nella pagina fisica (**operazione su disco**)
  - la PTE viene aggiornata inserendo l'NPF della pagina fisica al posto del SWPN del Page Slot

5.1 Swap_out

Una volta identificate le pagine da liberare come abbiamo visto nel capitolo precedente il meccanismo di swap_out è semplice. Per ogni pagina che deve essere liberata si tratta di determinare:

1. se la pagina è dirty; in tal caso il suo contenuto deve essere salvato su disco
2. se la pagina è mappata su file in maniera shared oppure è anonima; se è mappata su file deve essere scritta su tale file; se è anonima deve essere salvata nella swap area

Determinare se una pagina è dirty è banale per le pagine fisiche non condivise, ma può essere complesso per le pagine condivise.
Una pagina fisica PFx è dirty se:

- il suo descrittore è marcato D a seguito di un TLB flush (vedi capitolo M2, 4.2)
- una delle pagine virtuali condivise in PFx è contenuta nel TLB ed è marcata D

Negli esempi per rappresentare lo stato delle pagine swappate nella TP è utilizzata, nella posizione corrispondente al NPF, la notazione Sn, dove n è il SWPN del Page Slot. Ad esempio, se un processo ha swappato la pagina d0, nella TP avremo la riga **<d0:s0>**

Esempio/Esercizio 4
Si riconsideri l'evento finale dell'esercizio precedente in cui venivano scaricate le seguenti pagine:
Pp0/Rp0 , Rp5, Pp3/Rp3 , Pp5
Memoria, TLB e PT dei processi prima di tale evento sono

```
____MEMORIA FISICA____(pagine libere: 3)_____________________
      00 : <ZP>                ||  01 : Pc1/Rc1/<XX,1>      ||
      02 : Pp1/Rp1             ||  03 : Pp0/Rp0 D           ||
      04 : Pp4/Rp4             ||  05 : Rp5 D               ||
      06 : Pp5                 ||  07 : ----                ||
      08 : Pp2/Rp2             ||  09 : Pp3/Rp3             ||
      10 : ----                ||  11 : ----                ||
```

```
____STATO del TLB_____________________________________________________
      Pc1 : 01 -  0: 1:   ||   Pp0 : 03 -  1: 0:   ||
      Pp1 : 02 -  1: 1:   ||   Pp2 : 08 -  1: 1:   ||
      Pp3 : 09 -  1: 0:   ||   Pp4 : 04 -  1: 1:   ||
      Pp5 : 06 -  1: 0:   ||         -----         ||
PROCESSO: P  ******************************************************
    PT: <c0 :- -> <c1 :1  R> <k0 :- -> <s0 :- -> <p0 :3  D R> <p1 :2  R>
        <p2 :8  R> <p3 :9  R> <p4 :4  R> <p5 :6  W> <p6 :- ->
PROCESSO: R  ******************************************************
    PT: <c0 :- -> <c1 :1  R> <k0 :- -> <s0 :- -> <p0 :3  D R> <p1 :2  R>
        <p2 :8  R> <p3 :9  R> <p4 :4  R> <p5 :5  W> <p6 :- ->
```

Rappresentare il contenuto della memoria, dello Swap File e delle TP dopo tale evento.

Soluzione

Analizziamo la necessità di swappare le pagine da liberare:

- Pp0/Rp0 , → richiede swap perché marcata D
- Rp5, → richiede swap perché marcata D
- Pp3/Rp3 → richiede swap, perché Pp3 è marcata dirty nel TLB,
- Pp5 → richiede swap, perché marcata dirty nel TLB

Il nuovo stato della memoria, dello Swap File e delle TP dei processi è il seguente

```
____MEMORIA FISICA____(pagine libere: 3)_____________________________
      00 : <ZP>                ||  01 : Pc1/Rc1/<XX,1>      ||
      02 : Pp1/Rp1             ||  03 : Pd1                 ||
      04 : Pp4/Rp4             ||  05 : Pd2                 ||
      06 : Pd3                 ||  07 : Pd0                 ||
      08 : Pp2/Rp2             ||  09 : ----                ||
      10 : ----                ||  11 : ----                ||

SWAP FILE: Pp0/Rp0  , Rp5  , Pp3/Rp3  , Pp5  , ----, ----,

PROCESSO: P  ******************************************************
    PT: <c0 :- -> <c1 :1  R> <k0 :- -> <s0 :- -> <d0 :7  W> <d1 :3  W>
        <d2 :5  W> <d3 :6  W> <d4 :- -> <p0 :s0 R> <p1 :2  R> <p2 :8  R>
        <p3 :s2 R> <p4 :4  R> <p5 :s3 W> <p6 :- ->
PROCESSO: R  ******************************************************
    PT: <c0 :- -> <c1 :1  R> <k0 :- -> <s0 :- -> <p0 :s0 R> <p1 :2  R>
        <p2 :8  R> <p3 :s2 R> <p4 :4  R> <p5 :s1  W> <p6 :- ->
```

Nelle TP sono evidenziate le PTE che si riferiscono a pagine swappate (s0, s1, s2 3 s3 sono i primi 4 page slot della swap area secondo le convenzioni indicate sopra). Il motivo per cui pp0, rp0, pp3 e rp3 sono marcate in sola lettura è dovuto al meccanismo di swap in, trattato nel prossimo capitolo.

5.2 Swap_in

Il meccanismo di Swap_In è complicato dal tentativo di limitare il più possibile i trasferimenti tra swap file e memoria nel caso di pagine che vengono swappate più volte senza essere riscritte. Infatti, spesso una pagina sulla quale viene eseguito uno swap_in deve successivamente essere nuovamente scaricata. Se la pagina non è stata mai modificata dal momento dello swap_in, Linux evita di doverla riscrivere su disco; a questo scopo non la cancella dalla Swap Area al momento dello swap_in. Questo risultato è ottenuto tramite una struttura dati simile alla Page Cache, detta ***Swap Cache***.

In effetti, quando una pagina è stata swappata la Swap Area costituisce per tale pagina una specie di backing store; la situazione è simile a quella di una VMA mappata su file, ma vale per pagine singole invece che per intere VMA.

Con Swap Cache si intende (in maniera simile alla PageCache) *l'insieme delle pagine che sono state rilette da Swap File e non sono state modificate e alcune strutture ausiliarie* (**Swap_Cache_Index**) che permettono di gestirla come se tali pagine fossero mappate sulla Swap Area. Una pagina appartiene alle Swap Cache se è presente sia in memoria che su Swap area e se è registrata nel Swap_Cache_Index.

Dato che le pagine che richiedono swapping sono solo le ***pagine anonime*** e che non prendiamo in considerazione la possibilità che le pagine anonime possano appartenere ad aree SHARED, consideriamo solo la gestione mappata in maniera PRIVATE, cioè:

a. quando si esegue uno swap_in la pagina viene copiata in memoria fisica, ma la copia nella Swap Area non viene eliminata; *la pagina letta viene marcata in sola lettura*
b. nel Swap_Cache_Index viene inserito un descrittore che contiene i riferimenti sia alla pagina fisica che al Page Slot
c. finchè la pagina viene solamente letta questa situazione non cambia e, se la pagina viene nuovamente scaricata, non è necessario riscriverla su disco
d. se la pagina viene scritta, si verifica un Page Fault che causa le seguenti operazioni:
    - la pagina diventa privata, cioè non appartiene più alla Swap Area;
    - la sua protezione viene posta a W
    - il contatore swap_map_counter viene decrementato
    - se il contatore è diventato 0 il Page Slot viene liberato nella Swap Area

Per quanto riguarda le liste LRU il comportamento è il seguente:

- la NPV che è stata letta o scritta, cioè quella che ha causato lo Swap_in, viene inserita in tasta alla active con ref = 1;
- eventuali altre NPV condivise all'interno della stessa pagina vengono poste in coda alla inactive con ref = 0;

Esempio.

Le seguenti figure illustrano una possibile sequenza di operazioni. In figura 5.1 è mostrata una possibile situazione iniziale, con 3 processi che condividono una pagina swappata (si pensi ad esempio a una pagina dati dinamici dopo due fork). Nel modello semplificato avremo Pd0/Qd0/Rd0 nello swap file e
<Pd0:sx R> <Qd0:sx R> <Rd0:SWx R> nelle TP.

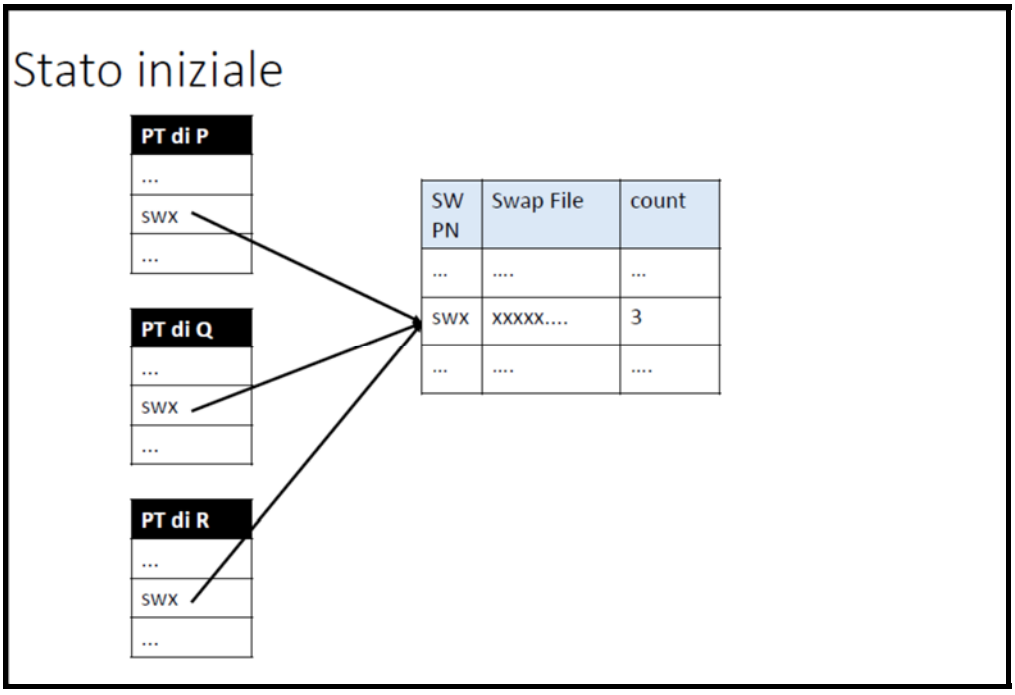

Figura 5.1

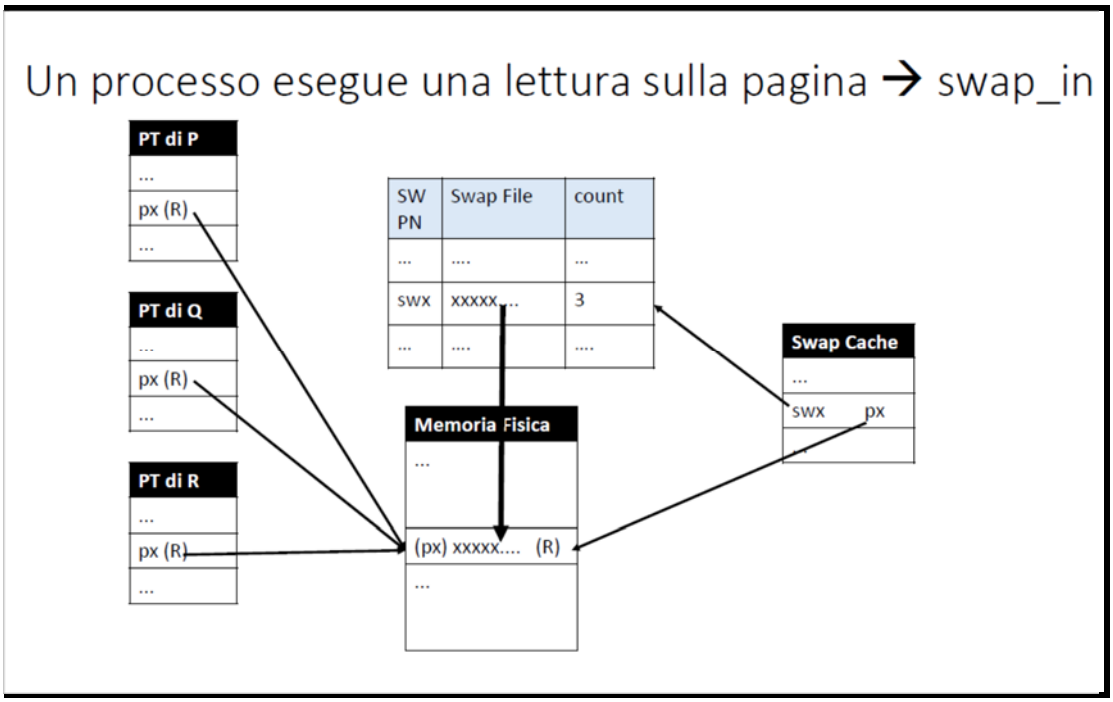

Figura 5.2

In figura 5.2 è mostrato l'effetto di una lettura da parte di un processo. La pagina è stata riletta. Nel modello semplificato abbiamo: Pd0/Qd0/Rd0 sia nello swap file che in memoria, le TP saranno modificate a <Pd0: px R> <Qd0: px R> <Rd0: : px R> ; inoltre abbiamo <swx, px> nella swap cache.

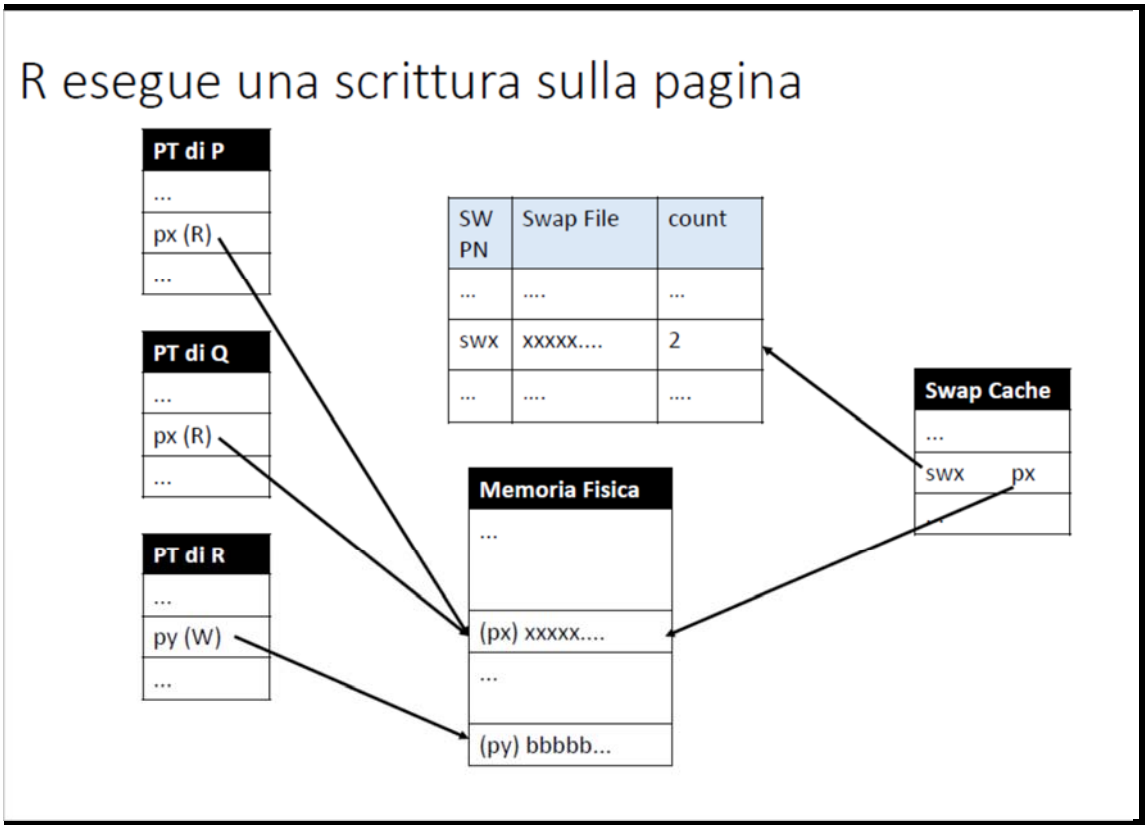

Figura 5.3

R scrive nella pagina. Viene allocata una nuova pagine in memoria. Le TP sono modificate a <Pd0: : px R> <Qd0: px R> <Rd0: py W> ; il contatore swap_map_counter è decrementato a 2

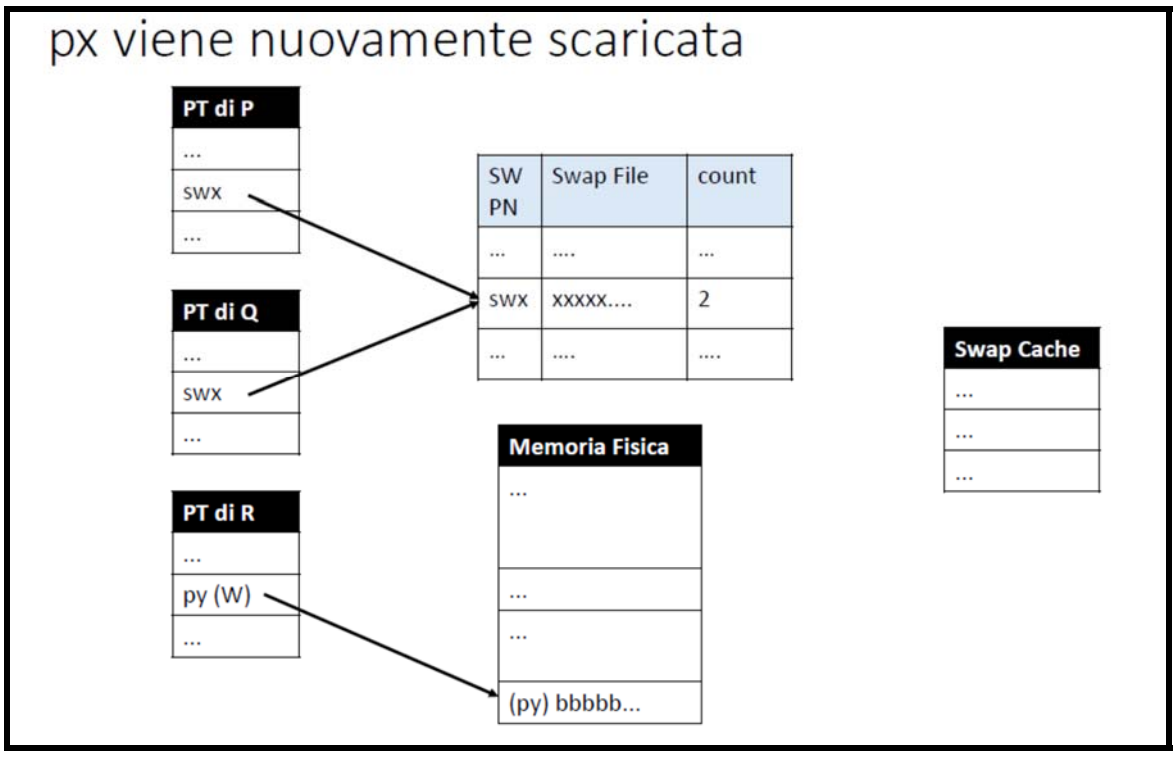

Figura 5.4

px viene nuovamente scaricata, py rimane indipendente. Le TP diventano:
<Pd0:SWx R> <Qd0:SWx R> <Rd0: py W>

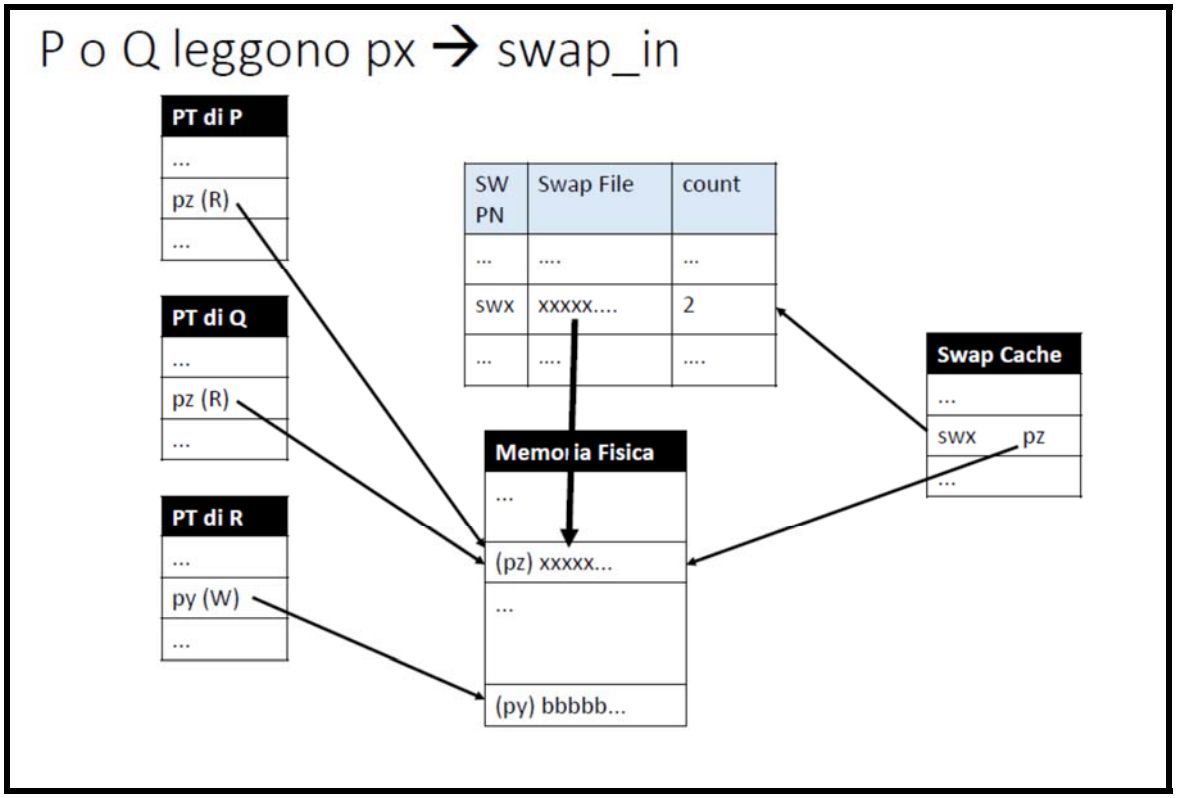

Figura 5.5

Un nuovo accesso a Pd0/Qd0 con conseguente swap_in una nuova pagina fisica (pz). Le TP diventano:
<Pd0: pz R> <Qd0: pz R> <Rd0: py W>

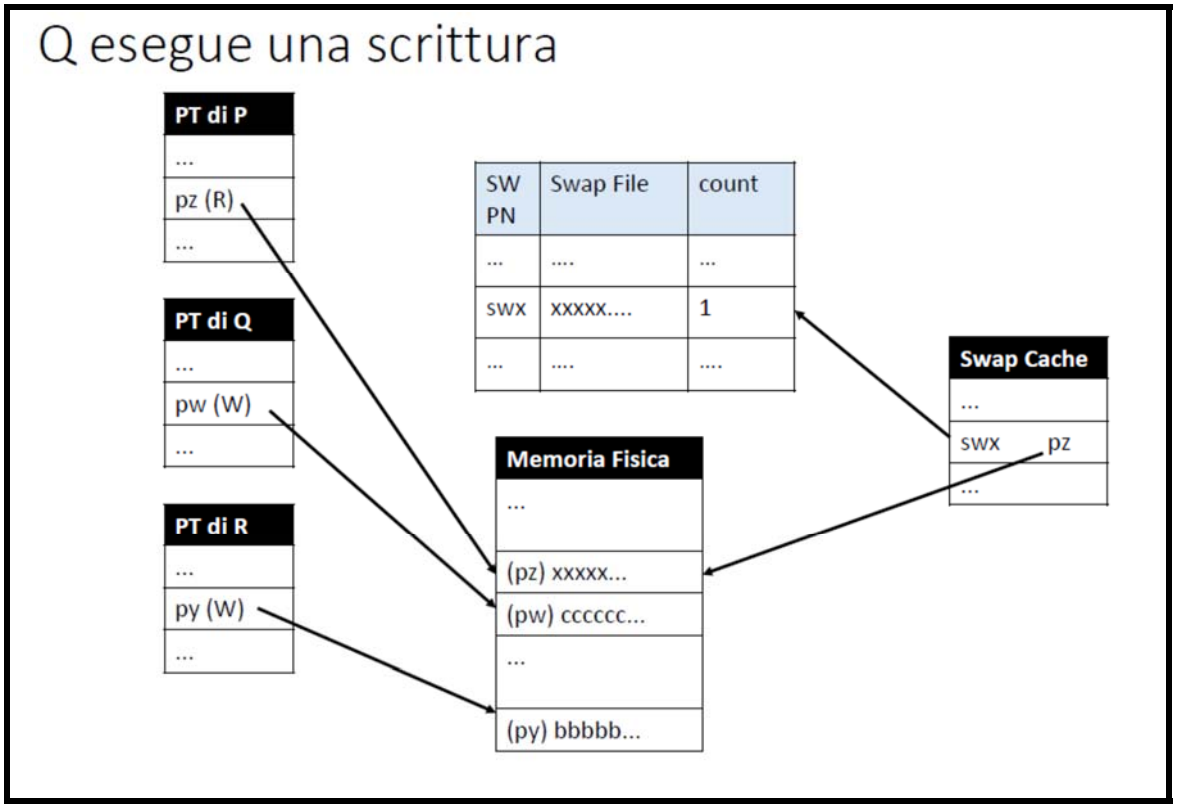

Figura 5.6

La scrittura eseguita da Q decrementa il swap_map_counter a 1. Le TP diventano:
<Pd0: pz R> <Qd0: pw W> <Rd0: py W>

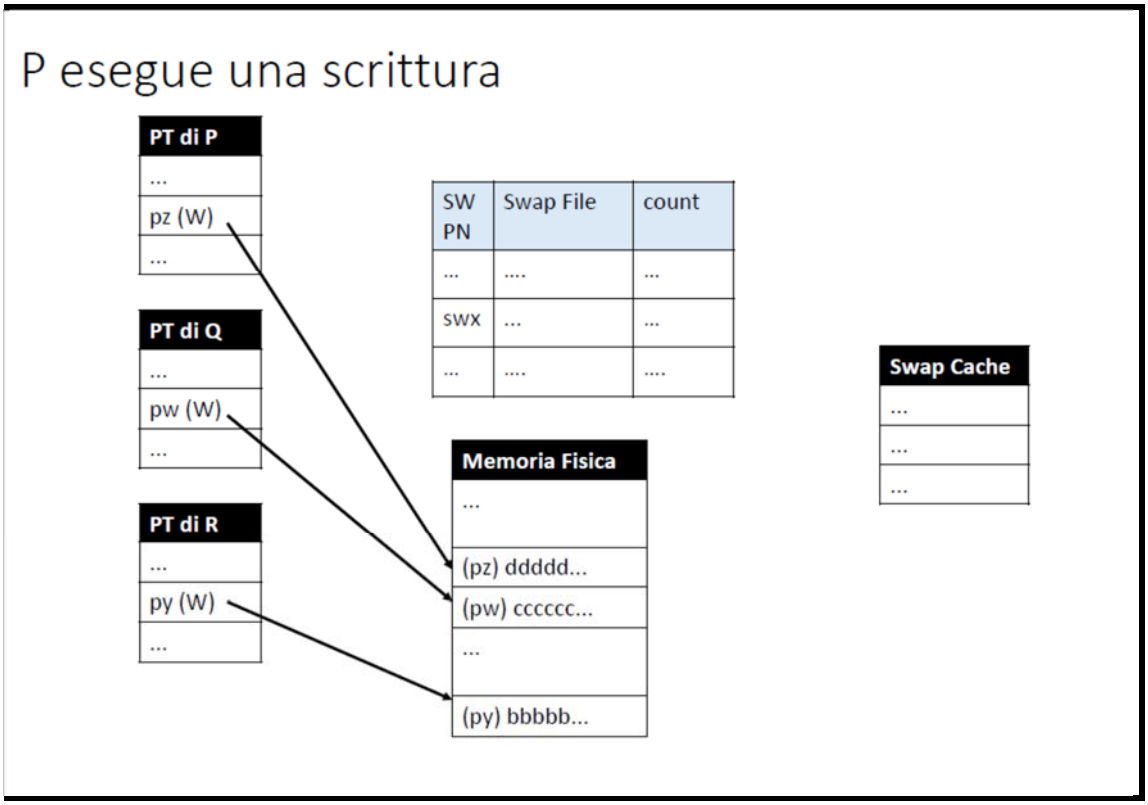

Figura 5.7

Infine, la scrittura da parte di P azzera il swap_map_counter. La pagina pz diventa privata del processo P, il Page Slot viene liberato e la Swap Cache non contiene più riferimenti a questa pagina. Le TP diventano <Pd0: pz W> <Qd0: pw W> <Rd0: py W>

Esempio/Esercizio 5
Si consideri lo stato seguente (derivato dallo stato finale dell'esercizio 4 modificando le LRU list):

```
____MEMORIA FISICA____(pagine libere: 3)_____________________________
      00 : <ZP>                 ||  01 : Pc1/Rc1/<XX,1>      ||
      02 : Pp1/Rp1              ||  03 : Pd1                 ||
      04 : Pp4/Rp4              ||  05 : Pd2                 ||
      06 : Pd3                  ||  07 : Pd0                 ||
      08 : Pp2/Rp2              ||  09 : ----                ||
      10 : ----                 ||  11 : ----                ||

SWAP FILE: Pp0/Rp0  , Rp5  , Pp3/Rp3  , Pp5  , ----, ----,

PROCESSO: P  ********************************************************
    PT: <c0 :- -> <c1 :1  R> <k0 :- -> <s0 :- -> <d0 :7  W> <d1 :3  W>
        <d2 :5  W> <d3 :6  W> <d4 :- -> <p0 :s0 R> <p1 :2  R> <p2 :8  R>
        <p3 :s2 R> <p4 :4  R> <p5 :s3 W> <p6 :- ->
PROCESSO: R  ********************************************************
    PT: <c0 :- -> <c1 :1  R> <k0 :- -> <s0 :- -> <p0 :s0 R> <p1 :2  R>
        <p2 :8  R> <p3 :s2 R> <p4 :4  R> <p5 :s1  W> <p6 :- ->
```

Le lrulist siano le seguenti:

```
      LRU ACTIVE:   PD1, PD0, PP4, PC1,
      LRU INACTIVE: pd3, pd2, pp2, pp1, rp4, rp2, rp1, rc1
```

Mostrare lo stato dopo i seguenti eventi:

1. lettura di pp0 e pp5, scrittura di pp3
2. scrittura di pp5

Soluzione.
Evento 1: lettura di pp0 e pp5, scrittura di pp3
Le operazioni si svolgono nell'ordine seguente:

- lettura di pp0 richiede swap_in, la pagina pp0/rp0 viene caricata in pagina 9 (freePages = 2)
- lettura di pp5 → invocazione di PFRA → liberazione di 2 pagine: pp1 e pp2 che vanno in swap area liberando le pagine 2 e 8 (freePages = 4)
- caricamento di pp5 in 2 (freePages = 3)
- swap in di pp3/rp3 in pagina 8 (freePages = 2)
- scrittura di pp3 con COW → invocazione di PFRA → liberazione di 2 pagine: pd2 e pd3 liberando le pagine fisiche 5 e 6 (freePages = 4)
- scrittura di pp3 in pagina 5 (freePages = 3)

Lo stato è il seguente (sono evidenziate le pagine in Swap Cache)

```
PROCESSO: P  ********************************************************
    PT: <c0 :- -> <c1 :1  R> <k0 :- -> <s0 :- -> <d0 :7  W> <d1 :3  W>
        <d2 :s6 W> <d3 :s7 W> <d4 :- -> <p0 :9  R> <p1 :s4 R> <p2 :s5 R>
        <p3 :5  W> <p4 :4  R> <p5 :2  R> <p6 :- ->
PROCESSO: R  ********************************************************
    PT: <c0 :- -> <c1 :1  R> <k0 :- -> <s0 :- -> <p0 :9  R> <p1 :s4 R>
        <p2 :s5 R> <p3 :8  R> <p4 :4  R> <p5 :s1  W> <p6 :- ->
____MEMORIA FISICA____(pagine libere: 3)_____________________________
      00 : <ZP>                ||  01 : Pc1/Rc1/<XX,1>      ||
      02 : Pp5                 ||  03 : Pd1                 ||
      04 : Pp4/Rp4             ||  05 : Pp3                 ||
      06 : ----                ||  07 : Pd0                 ||
      08 : Rp3                 ||  09 : Pp0/Rp0             ||
      10 : ----                ||  11 : ----                ||
SWAP FILE: Pp0/Rp0  , Rp5  , Rp3  , Pp5  , Pp1/Rp1  , Pp2/Rp2  , Pd2  , Pd3,

LRU ACTIVE:   PP3, PP5, PP0, PD1, PD0, PP4, PC1,
LRU INACTIVE: rp4, rc1, rp0, rp3,
```

Evento 2: scrittura di pp5
La scrittura di pp5 elimina pp5 dalla Swap Area (e quindi dalla Swap Cache) rendendo pp5 una pagina privata del processo P, abilitata normalmente in scrittura.

```
PROCESSO: P  ********************************************************
    PT: <c0 :- -> <c1 :1  R> <k0 :- -> <s0 :- -> <d0 :7  W> <d1 :3  W>
        <d2 :s6 W> <d3 :s7 W> <d4 :- -> <p0 :9  R> <p1 :s4 R> <p2 :s5 R>
        <p3 :5  W> <p4 :4  R> <p5 :2  W> <p6 :- ->
____MEMORIA FISICA____(pagine libere: 3)_____________________________
      00 : <ZP>                ||  01 : Pc1/Rc1/<XX,1>      ||
      02 : Pp5                 ||  03 : Pd1                 ||
      04 : Pp4/Rp4             ||  05 : Pp3                 ||
      06 : ----                ||  07 : Pd0                 ||
      08 : Rp3                 ||  09 : Pp0/Rp0             ||
      10 : ----                ||  11 : ----                ||

SWAP FILE: Pp0/Rp0  , Rp5  , Rp3  , ----, Pp1/Rp1  , Pp2/Rp2  , Pd2  , Pd3
, ----, ----,

LRU ACTIVE:   PP3, PP5, PP0, PD1, PD0, PP4, PC1
LRU INACTIVE: rp4, rc1, rp0, rp3,
```

5. Osservazioni finali

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 e che 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 scaricate.

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 nessuna 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 svolgono funzioni importanti)
- non gestisca direttamente componenti hardware, per non lasciarle in uno stato inconsistente
- non sia un Kernel thread

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.