Giuseppe Pelagatti

Programmazione e Struttura del sistema operativo Linux

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

Parte M: La gestione della Memoria

cap. M4 – Gestione della memoria fisica

M.4 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 letti dal disco: dato che i dischi sono di ordini di grandezza più lenti della RAM conviene sempre tenere in memoria un blocco letto 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 nelle versioni recenti di Linux c'è stata una convergenza verso l'uso di una cache organizzata a pagine, detta **Page Cache**.

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. molte pagine di cache vengono scaricate; se questo non è sufficiente
2. alcune pagine dei 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 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 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 le seguenti motivazioni:

- la memoria è acceduta anche dai canali DMA in base a indirizzi fisici, non virtuali; se un buffer di memoria supera la dimensione della pagina deve essere costituito da pagine contigue
- è comunque preferibile usare pagine contigue fisicamente per motivi legati sia alle cache che alla latenza di accesso alle RAM
- 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

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 (se non ci sono processi come quello di figura 1)
- 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 la 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.
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.

Invocazione del PFRA
Il PFRA è invocato nei casi seguenti:

1. Invocazione diretta tramite la funzione `try_to_free_pages` da parte di un processo
2. Attivazione periodica tramite `kswapd` – `kernel swap daemon`

Il `kernel swap daemon, kswapd,` è una funzione che viene attivata periodicamente ogni volta che scatta il "kernel swap timer" e controlla se il numero di pagine libere nel sistema è diventato troppo basso; se tale condizione si è verificata, `kswapd` attiva il PFRA.

L'attivazione diretta si verifica 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; processi di questo tipo richiedono pagine di memoria prima che `kswapd` le liberi.

Sia `kswapd` che `try_to_free_pages` invocano la stessa funzione del PFRA, quindi la trattazione seguente si applica ad ambedue i casi.

Le liste Least Recently Used (LRU list)
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 e alle Buffer/Cache:

- ***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).

Possiamo suddividere il funzionamento di PFRA in due parti:

1. lo scaricamento effettivo di pagine appartenenti alla lista inattiva, a partire dalla coda
2. lo spostamento delle pagine da una lista all'altra per tenere conto degli accessi alla memoria.

Scaricamento delle pagine della lista inattiva
La funzione di scaricamento deve determinare anzitutto quante pagine scaricare (algoritmo empirico). Non conviene scaricare una sola pagina, perché si causerebbe probabilmente una riattivazione quasi immediata del PFRA. Gli algoritmi utilizzati per questa funzione sono di natura empirica e non vengono trattati.

Le pagine vengono scaricate partendo dalla coda della inactive, che contiene le pagine più invecchiate.

Per eseguire lo swap out la funzione deve trovare il file di swap e la posizione al suo interno in cui scrivere la pagina; questo avviene utilizzando il puntatore `mm→swap_address` presente nella struttura `mm_struct` del processo.
Quando una pagina viene swappata nella sua PTE viene inserito, al posto del riferimento alla memoria fisica, un puntatore alla posizione nel file di swap in cui è stata salvata.

Spostamento delle pagine tra le due liste
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 il Working Set di tutti i processi*.

Rispetto alla struttura di LRU classico abbiamo 2 differenze fondamentali e una serie di aggiustamenti secondari.
La prima differenza fondamentale è che la politica di sostituzione è globale invece di essere applicata ad ogni singolo processo.
La seconda 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 nella PTE della pagina, 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 periodica Controlla_liste 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 pongono in testa alla active con ref=1 le nuove pagine caricate da un processo

- 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 1
Nelle 2 liste le lettere maiuscole indicano ref=1, quelle minuscole ref=0.
In ogni riga viene mostrato lo stato prodotto da un'attivazione di Controlla_liste in base allo stato del bit di accesso, indicato nella colonna "pagine accedute".

Attivazioni	**pagine accedute**	**Active**	**Inactive**
0 (iniziale)		A B C D	x y
1	A D Y	A D b c	Y x
2	A D Y	A D y	B C x
3	A D Y	A D Y	b c x

A questo punto nella lista inactive ci sono le pagine meno accedute, in ordine di "invecchiamento".

Si noti che la pagina c 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 C l'avrebbe riportata nella active.

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 inattiva (ref=0 nella inactive) richiedono 2 valutazioni positive per ritornare nella active.

Esempio 2
Utilizziamo la stessa rappresentazione dell'esempio precedente, ma distinguendo le pagine in base ai processi.
Indichiamo le pagine dei processi con la seguente convenzione.

- Processo P: pagine p1, p2 …
- Processo Q: pagine q1, q2 ….
- ecc

Ipotizziamo anche che il sistema richieda di scaricare almeno 4 pagine quando il numero totale di pagine supera 12.

Stato iniziale: esistono un processo P e un processo Q; Q ha eseguito una exec, quindi tutte le sue pagine sono state invalidate.

Sono indicate in rosso le pagine che vengono caricate da disco.

Attivazioni	**In esecuzione**	**pagine accedute**	**Active**	**Inactive**
0 (iniziale)	Q		P1,P2,P3	p4,p5,p6,p7
1	Q	q1,q2,q3,q4	**Q1,Q2;Q3,Q4**,p1,p2,p3	p4,p5,p6,p7
2	Q	q1,q2,q3,q4	Q1,Q2;Q3,Q4	P1,P2,P3, p4,p5,p6,p7
3	Q	q3,q4,q5,q6	**Q5,Q6**,Q3,Q4,q1,q2	p1,p2,p3, p4,p5,p6,p7
numero pagine > 12, invoca scaricamento			Q5,Q6,Q3,Q4,q1,q2	p1,p2,p3
4	Q	q3,q4,q5,q6	Q5,Q6,Q3,Q4	Q1,Q2, p1,p2,p3
commutazione di contesto				
5	P	p1,p2,p5,p7	**P5,P7,** q5,q6,q3,q4	q1,q2,P1,P2,p3
6	P	p1,p2,p5,p7	P5,P7,p1,p2	Q5,Q6,Q3,Q4,q1,q2,p3
7	P	p1,p2,p5,p7	P5,P7,P1,P2	q5,q6,q3,q4,q1,q2,p3

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.

Interferenza tra Gestione della Memoria e Scheduling
L'esempio 2 mostra che 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.