O(1) Scheduler in Linux 2.5.7

Salvatore D'Angelo

Copyright © 2004 Salvatore D'Angelo koala.gnu@tiscali.it

Table of Contents

Introduzione **1**

1 Priorità dei processi **2**

2 Policy di scheduling **5**

3 Strutture dati **6**

4 Funzioni di Utilità **8**

- 4.1 task_rq_lock 8
- 4.2 task_rq_unlock 8
- 4.3 enqueue_task 9
- 4.4 dequeue_task 9
- 4.5 effective_prio 9
- 4.6 activate_task 10
- 4.7 deactivate_task 10
- 4.8 resched_task 11
- 4.9 wait_task_inactive 11
- 4.10 kick_if_running 11
- 4.11 nr_running 12
- 4.12 nr_context_switches 12
- 4.13 sched_exit 12

5 Bilanciamento di carico in ambiente SMP **13**

Introduzione

L'obiettivo di questo articolo è quello di illustrare le politiche di scheduling adottati nel kernel di Linux nelle versioni 2.5.x in seguito ad alcune patches introdotte da Ingo Molnar.

Nello scheduler delle precedenti versioni accadeva che se c'erano n processi attivi che avevano consumato tutto il quanto di tempo a loro disposizione, il kernel faceva una scansione di questi processi per ricalcore la nuova priorità dinamica. Questo approccio chiaramente aveva un costo proporzionale al numero di processi presenti nella run queue, cioè O(n).

L'approccio proposto da Ingo Molnar è leggermente diverso e punta ad evitare questa inutile scansione mantenendo, se non addirittura migliorando, tutte le altre caratteristiche del precedente scheduler.

Ingo ha proposto un modello di scheduler in cui, per ogni CPU esiste una runqueue avente due code di priorità, una per i thread attivi e una per quelli spirati (cioè che hanno terminato il proprio quanto di tempo). L'aggiornamento della priorità dinamica non avviene più alla fine, quando tutti i processi hanno esaurito il proprio quanto di tempo, bensì man mano che essi avanzano. Quando non esistono più processi con un quanto di tempo maggiore di 0, viene fatto uno swap tra le due code che, da un punto di vista computazionale, ha il costo di un semplice scambio tra due puntatori (cioè un costo costante).

Le successive sezioni illustrano in dettaglio sia la teoria su cui si appoggia il modello presentato da Molnar, sia gli aspetti implementativi. In particolare, verranno analizzati i seguenti punti:

- priorità statica e dinamica di un processo;
- policies di scheduling;
- processi real time;
- processi interattivi;
- bilanciamento di carico in ambiente SMP.

1 Priorità dei processi

La priorità di un processo è un numero che esprime l'urgenza con cui esso deve essere eseguito. Task ad altissima priorità (0-99) devono garantire tempi di risposta molto elevati e in Linux vengono chiamati "processi real time". Al contrario, processi a bassa priorità (100-139) sono detti "processi normali".

In Linux minore è il valore di priorità di un processo, maggiore è la priorità con cui deve essere eseguito. Le seguenti due costanti esprimono gli upper limits per le priorità di queste due categorie di processi.

- `MAX_RT_PRIO = 100`
- `MAX_PRIO = 140`

In Linux, ad ogni processo si associa una priorità statica (modificabile solo attraverso le system call setpriority o nice) e una priorità dinamica che varia in base all'evoluzione del processo nel corso della sua esecuzione. Ad esempio, un thread che ha passato molto del suo tempo in stato di sleep sicuramente fino a quel momento ha fatto un uso ridotto della CPU per cui esso dovrà essere premiato con un incremento della priorità dinamica. Questo è utile soprattutto per i threads interattivi che, spendono la maggior parte del proprio tempo ad aspettare che l'utente inserisce dati attraverso i device di input.

Lo standard Posix definisce la priorità statica di un processo come un valore compreso nel range [-20..19] e che chiameremo "nice value".

Internamente lo scheduler converte il nice value in priority (e viceversa) usando le formule:

```
NICE_TO_PRIO(nice) = MAX_RT_PRIO + nice + 20;
PRIO_TO_NICE(priority) = priority - MAX_RT_PRIO - 20;
```

L'associazione tra un processo e le relative priorità è rappresentata nel kernel attraverso due campi nel descrittore di processo, come mostrato qui di seguito.

```
struct task_struct {
        int prio;               // priorità dinamica
        int static_prio;        // priorità statica
        ...
}
```

Dato un thread T, il suo nice value è dato dalla relazione:

```
TASK_NICE(T) = PRIO_TO_NICE(T.static_prio)
```

Per motivi storici che evitiamo di riportare in questa sede, a volte si preferisce normalizzare la priority di un processo nel range [0..39] anzichè in [100..139] e tale priorità viene chiamata "User Priority".

```
TASK_USER_PRIO(T) = USER_PRIO(T1.static_prio)
```

dove:

```
USER_PRIO(priority) = priority - MAX_RT_PRIO
```

E' chiaro che il max valore ammesso dalla "User Priority" è uguale a MAX_PRIO normalizzata nel range [0..39].

```
MAX_USER_PRIO = USER_PRIO(MAX_PRIO)
```

Un processo T rimane in esecuzione finchè non viene interrotto da eventi esterni oppure termina il suo timeslice (o "quanto di tempo"). Normalmente un thread che termina il suo timeslice viene messo nella coda dei threads spirati in attesa che tutti gli altri processi terminino il proprio quanto di tempo. Ciò non accade, però, per i thread interattivi i quali, dovendo garantire tempi di risposta immediati all'utente, vengono messi di nuovo in coda in base ad una politica round robin. A questo punto ci si pone la seguente domanda: ma come si fa a capire quando un thread è interattivo?

Definiamo l'interattività di un thread come una funzione f booleana suriettiva tale che per ogni coppia (static priority, dynamic priority) associa un valore di 1 (thread interattivo) o 0 (thread non interattivo).

In particolare, dato un thread T avremo che:

```
TASK_INTERACTIVE(T) = T.prio <= T.static_prio - DELTA(T)
```

dove:

```
DELTA(T) = (TASK_NICE(T) * (MAX_USER_PRIO*PRIO_BONUS_RATIO/100) / 40) +
           INTERACTIVE_DELTA
PRIO_BONUS_RATIO  = 25
INTERACTIVE_DELTA = 2
```

L' intera espressione dice che **DELTA(T)** è pari al 25% del nice value del thread + un valore costante **INTERACTIVE_DELTA**.Il seguente diagramma mostra come varia l'interattività di un thread in funzione del nice value del thread e della sua priorità dinamica. I valori 1 e 0 indicano l'interattività del thread.

```
   Priorità statica
      |
  -20 | 1  1  1  1  1  1  1  1  1  0  0
  -10 | 1  1  1  1  1  1  1  0  0  0  0
    0 | 1  1  1  1  0  0  0  0  0  0  0
   10 | 1  1  0  0  0  0  0  0  0  0  0
   19 | 0  0  0  0  0  0  0  0  0  0  0
      |__________________________________ Priorità dinamica
       -5 -4 -3 -2 -1  0  1  2  3  4  5
```

E' facile osservare come un thread di bassa priorità (nice = 19) difficilmente verrà considerato interattivo, al contrario un thread ad alta priorità viene considerato interattivo in un'ampio range di priorità dinamica. Si osservi come, per ogni variazione **DELTA(T)** della priorità statica, l'interattività di un thread cresce linearmente (**INTERACTIVE_DELTA**) rispetto alla priorità dinamica.

Ogni thread, durante la sua esecuzione, può essere interrotto o da un evento esterno o dal kernel se il suo quanto di tempo è spirato. Il quanto di tempo di un thread varia da 10ms a 300 ms in base al suo nice value. Il valore di default è pari a 150ms.

In generale, valgono le seguenti definizioni:

```
MIN_TIMESLICE =  10 * HZ / 1000   ( 10 ms)
MAX_TIMESLICE = 300 * HZ / 1000   (300 ms)

TASK_TIMESLICE(T) = MIN_TIMESLICE +
                    (MAX_TIMESLICE - MIN_TIMESLICE) *
                    ((MAX_PRIO - T.static_prio - 1)/39)
```

La seguente tabella mostra il quanto di tempo per un generico thread al variare del suo nice value.

```
Niceness        Quanto di tempo
-------------------------------
 19               10.00 ms
 18               17.43 ms
 17               24.87 ms
 16               32.30 ms
 15               39.74 ms
 14               47.17 ms
 13               54.61 ms
...             .........
-18              285.12 ms
-19              292.56 ms
-20              300.00 ms
```

2 Policy di scheduling

Con il termine di policy si intende la politica che lo scheduler applica nello scegliere il prossimo thread da eseguire. Ad esempio, lo scheduler può decidere se un thread può essere prelazionato oppure, data la sua importanza, lasciarlo in esecuzione per un tempo considerevolmente lungo.

Linux applica fondamentalmente tre tipi di policy per lo scheduling di threads:

- `SCHED_FIFO`
- `SCHED_RR`
- `SCHED_OTHER`

di cui le prime due sono riservate a processi real time (la cui priorità statica varia da 0 a 99), mentre l'ultima è riservata a processi Posix convenzionali.

Se un thread viene schedulato con policy **SCHED_FIFO**, allora significa che è un processo real time ad altissima priorità che deve essere eseguito riducendo la sua prelazione il più possibile. Il thread rimane quindi in esecuzione a patto che non ci siano thread a priorità maggiore.

La policy **SCHED_RR** sempre relativa a processi real time applica una politica "time - sharing", consente di prelazionare un thread qualora il suo timeslice si esaurisce.

Infine la policy **SCHED_OTHER**, relativa a processi convenzionali, applica anch'essa una politica di time-sharing tra i threads, con la differenza che i processi sono a bassa priorità.

3 Strutture dati

Prima di illustrare le strutture dati che lo scheduler utilizza per la scelta dei threads che di volta in volta dovrà eseguire, bisogna ricordare due importanti proprietà che lo scheduler deve garantire:

- Nessuna CPU deve rimanere inattiva se ci sono threads da eseguire;
- evitare il più possibile che un thread salti da una CPU all'altra.

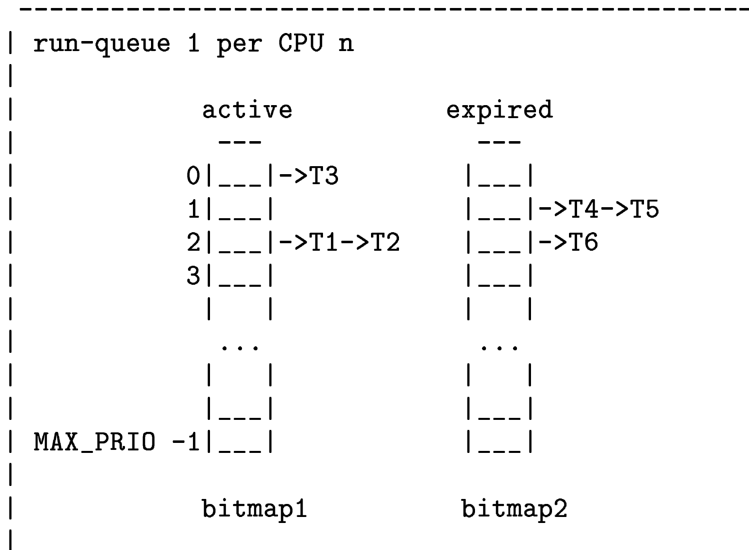
Proprio queste due condizioni hanno determinato la scelta delle strutture dati che ora andremo ad analizzare.

Lo scheduler utilizza una run queue per ogni CPU. Ciascuna run queue è costituita da due code di priorità: una per i threads attivi (active), cioè quelli che non hanno ancora esaurito il proprio quanto di tempo, e una per i threads spirati (expired), cioè quelli che hanno esaurito il proprio quanto di tempo.

Inizialmente, i threads vengono messi nella prima coda e non appena uno di essi termina il suo quanto di tempo, viene spostato nella seconda coda a meno che non si tratti di un thread interattivo.

Ogni coda di priorità (active o expired) è costituita da MAX_PRIO buckets ciascuno dei quali punta ad una lista di thread aventi la stessa priorità statica.

Il seguente diagramma mostra l'organizzazione in memoria di tali strutture su un'ipotetica CPU *n*



E' facile osservare che ad ogni coda di priorità è associata una bitmap il cui i-esimo bit i = 1 se e solo se la lista dell'i-esimo bucket non è vuota.

Ma quante words sono necessarie per allocare un'intera bitmap costituita da **MAX_PRIO** bits?

Ricordiamo innanzittutto che su un'architettura a 32 bits una word è costituita da 4 bytes, mentre su un'architettura a 64 bits una word è costituita da 8 bytes.

Da un punto di vista del compilatore il tipo long rappresenta la word sulle varie architetture, quindi sizeof(long)=4 su macchine a 32 bits e sizeof(long)=8 su macchine a 64. L'espressione:

```
(MAX_PRIO+8) / 8
```

indica il numero minimo di bytes necessari per incpsulare la bitmap, mentre l'espressione:

```
(((MAX_PRIO+8) / 8) + sizeof(long)-1)/sizeof(long)
```

indica il numero minimo di word necessarie a contenere la bitmap.

Linux definisce ciascuna coda di priorità (active o expired) nel modo seguente:

```
BITMAP_SIZE = (((MAX_PRIO+8) / 8) + sizeof(long)-1)/sizeof(long)

struct prio_array {
    int nr_active;                      // numero di threads
                                        // presenti nella coda
    unsigned long bitmap[BITMAP_SIZE];  // bitmap
    list_t queue[MAX_PRIO];             // coda a priorità
}
```

il tipo list_t rappresenta una generica lista doppiamente linkata.

Le runqueue dello scheduler, invece, sono definite nel modo seguente:

```
struct runqueue {
         spinlock_t lock;
         unsigned long nr_running;
         task_t *curr, *idle;
         prio_array_t *active, *expired, arrays[2];
}

static struct runqueue runqueues[NR_CPUS] __cacheline_aligned;
```

Si osservi come il numero delle runqueue è pari al numero delle CPUs. Per ciascuna runqueue, inoltre, esiste un mutex che viene utilizzato per evitare race conditions tra thread su tale risorsa, c'è poi un campo nr_running che indica il numero di thread attualmente associati alla runqueue. I campi `curr` e `idle` puntano, rispettivamente, al thread correntemente in esecuzione e all'idle thread.

Infine, come già anticipato sopra ci sono due code di priorità (`arrays`) che, alternativamente, rappresentano la coda active e expired.

Come vedremo in seguito, quando tutti i threads hanno esaurito il proprio quanto di tempo, verrà fatto uno swap tra i valori di active e expired in modo tale che la vecchia coda expired diventi quella active e, viceversa, quella active diventi expired.

4 Funzioni di Utilità

In questo capitolo introduciamo alcune funzioni di utilità dello scheduler necessari per aggiungere/rimuovere threads alle/dalle runqueues, attivare/disattivarne i locks e altro ancora.

4.1 task_rq_lock

Dato in input un thread, acquisisce il lock sul mutex della runqueue in cui esso è presente restituendo in output la coda con lock attivo.

```
static inline runqueue_t *task_rq_lock(task_t *p, unsigned long *flags)
```

`p` *thread appartenente alla runqueue cui va acquisito il mutex*

`flag` *usato per il lock della runqueue*

Pseudo Codice:

```
ll:
   disabilita thread preeption;
   rq = runqueue in cui il thread T  presente;
   lock rq;
   if (rq non è più la runqueue di T) {
        unlock rq;
        abilita thread preeption;
        goto ll;
  }
  return rq
```

L'if è necessario perchè dall'istruzione di assegnamento prima del lock al momento effettivo del lock è probabile che il thread abbia cambiato runqueue, cioè sia in esecuzione su un'altra CPU.

4.2 task_rq_unlock

Data in input una runqueue il cui mutex è stato acquisito da task_rq_lock, questa funzione provvede a rilasciare tale lock.

```
static inline void task_rq_unlock(runqueue_t *rq, unsigned long *flags)
```

`rq` *coda cui va rilasciato il lock*

`flag` *usato per l'unlock della runqueue*

Pseudo Codice

```
unlock rq;
abilita task preeption;
```

4.3 enqueue_task

Aggiunge un thread a una coda di priorità (active o expired).

`static inline void enqueue_task(struct task_struct *p, prio_array_t *array)`

`p` *thread da aggiungere*

`array` *coda di priorità a cui va aggiunto il thread*

Pseudo Codice

```
aggiungi p in coda alla lista array->queue[p->prio];
attiva il bit p->prio nella bitmap array->bitmap;
incrementa il numero di threads presenti nella coda di priorità;
```

4.4 dequeue_task

Rimuove un thread da una coda a priorità (active o expired).

`static inline void dequeue_task(struct task_struct *p, prio_array_t *array)`

`p` *thread da rimuovere*

`array` *coda di priorità da cui va rimosso il thread*

Pseudo Codice

```
decrementa il numero di threads presenti nella coda di priorità;
rivuovi p dalla lista array->queue[p->prio];
if (lista array->queue[p->prio]  vuota)
   azzera il bit p->prio della bitmap array->bitmap;
```

4.5 effective_prio

Come anticipato sopra, un thread interattivo che spende la maggior parte del proprio tempo in stato di sleep, va premiato con uno speciale bonus che va sommato alla priorità dinamica in modo tale che tale thread al successivo wake up abbia maggiori probabilità di essere eseguito. Questo chiaramente ha un effetto positivo sui tempi di risposta che l'utente riceve dal sistema.

Per ogni thread esiste un tempo medio di sleep che varia nel range [0 ... MAX_SLEEP_AVG]. Tale valore verrà poi scalato nel range [-5 ... +5]. la cui ampiezza è il 25% (PRIO_BONUS_RATIO) dell'ampiezza del range delle user priority ([0 ... 39]).

```
bonus = MAX_USER_PRIO*PRIO_BONUS_RATIO*p->sleep_avg/MAX_SLEEP_AVG/100 -
MAX_USER_PRIO*PRIO_BONUS_RATIO/100/2;
```

dove p->sleep_avg rappresenta appunto il tempo medio di sleep del thread p.

Calcolato tale bonus, la priorità dinamica sarà uguale alla priorità statica meno questo valore, controllando, per, che essa sia sempre compresa nel range [MAX_RT_PRIO ... MAX_PRIO-1].

`static inline void effetctive_prio(struct task_struct *p)`

`p` *thread di cui va calcolata l'effettiva priorità dinamica*

Pseudo Codice

```
bonus = MAX_USER_PRIO*PRIO_BONUS_RATIO*p->sleep_avg/MAX_SLEEP_AVG/100 -
        MAX_USER_PRIO*PRIO_BONUS_RATIO/100/2;
prio = p->static - bonus;
if (prio < MAX_RT_PRIO)
    prio = MAX_RT_PRIO;
if (prio > MAX_PRIO-1)
    prio = MAX_PRIO-1;
    return prio
```

4.6 activate_task

Quando un processo viene risvegliato, viene invocata questa funzione che aggiorna il suo tempo medio di sleeping (se il thread era in stato di sleep), ne calcola l'effettiva priorità dinamica utilizzando la funzione effective_prio vista sopra e aggiunge il thread alla coda attiva della runqueue in input.

`static inline void activate_task(task_t *p, runqueue_t *rq)`

`p` *thread da riattivare*

`rq` *runqueue che lo ospiterà nella sua coda active*

Pseudo Codice

```
sleep time = ora corrente - p->sleep_timestamp;
if (p non è un thread real time e torna da uno stato di sleep)
    // aggiorno tempo medio di sleep
    p->sleep_avg += sleep_time;
    if (p->sleep_avg > MAX_SLEEP_AVG)
        p->sleep_avg = MAX_SLEEP_AVG;
    p->prio = effective_prio(p);
    enqueue_task(p, coda attiva di rq);
    incrementa il numero di thread presenti in rq;
```

`p->sleep_timestamp` contiene il timestamp dell'istante in cui il thread era entrato in stato di sleep. Tale timestamp conterrà l'ora corrente se il thread non era in stato di sleep e in tale condizione lo sleep time è nullo.

4.7 deactivate_task

Disattiva un thread togliendolo dalla coda di priorità in cui è attualmente inserito. Implicitamente il thread viene rimosso anche dalla runqueue che attualmente lo contiene.

`static inline void deactivate_task(struct task_struct *p, runqueue_t *rq)`

`p` *thread da disattivare*

`rq` *runqueue da cui il thread va rimosso*

Pseudo Codice

```
decrementa il numero di threads presenti nella run queue;
dequeue_task(p, coda di priorità che lo contiene)
```

4.8 resched_task

Imposta il 3 bit del campo flag nella strutture thread_info presente nel descrittore di processo.

```
struct thread_info {
__u32 flags;
...
}

struct task_struct {
...
struct thread_info *thread_info;
...
}
```

Data la semplicità della funzione evitiamo di riportare ulteriori dettagli.

4.9 wait_task_inactive

Attende finchè il thread in input non diventi inattivo.

```
static inline void wait_task_inactive(struct task_struct *p)
```

`p` *thread da attendere*

Pseudo Codice

```
repeat:

        disabilita preeption threads;
        rq = runqueue associata al thread p
        while (p è in esecuzione)
               aspetta
        rq = task_rq_lock(p, ...);
        if (p è correntemente in esecuzione) {
              // probabilmente un attimo prima del locking p è stato
              // rischedulato, per cui si ricomincia da capo
              task_rq_unlock(rq, ...);
              attiva preeption dei thread;
              goto repeat;
        }
        task_rq_unlock(rq, ...);
        attiva preeption dei threads;
```

4.10 kick_if_running

Questo metodo viene utilizzato affinchè il thread in input, se correntemente in esecuzione su una data CPU, venga rischedulato al più presto. Questa funzione molto utilizzata dal gestore di segnali quando si ha necessità di inviare segnali velocemente a threads che girano in user mode.

`void kick_if_running(task_t * p)`

`p` *thread da rischedulare se correntemente in esecuzione*

Pseudo Codice

```
if (p  correntemente in esecuzione)
   resched_task(p);
```

4.11 nr_running

Restituisce il numero di thread in esecuzione o pronti ad essere eseguiti. Questa funzione non fa altro che restituire la somma del numero di threads presenti in ciascuna runqueue.

4.12 nr_context_switches

Restituisce il numero di context switch avvenuti in tutte le runqueue dello scheduler. Affinchè ciò sia facilmente realizzabile, viene definito un campo contatore all'interno della struttura runqueue:

```
struct runqueue {
...
unsigned long nr_switches;
...
}
```

che viene incrementato ad ogni context switch per quella runqueue.

4.13 sched_exit

Questa funzione viene invocata quando termina il processo in input (a causa di un'invocazione ad exit) al fine di aggiornare alcuni parametri del processo padre. Ad esempio al time slice del padre viene sommato quello del figlio e, se il processo figlio aveva un tempo medio di sleep inferiore al padre, allora questi viene opportunamente aggiornato.

`void sched_exit(task_t * p)`

`p` *thread terminato*

Pseudo Codice

```
disattiva interrupt;
// aggirnamento time slice
current->time_slice += p->time_slice;
if (current->time_slice > MAX_TIMESLICE)
    current->time_slice = MAX_TIMESLICE;
attiva interrupt;

// ricalcolo tempo medio di sleep
if (p->sleep_avg < current->sleep_avg)
    // current->sleep_avg = 75 % di current->sleep_avg +
       25 % di p->sleep_avg
current->sleep_avg = (current->sleep_avg * EXIT_WEIGHT + p->sleep_avg)
                     / (EXIT_WEIGHT + 1)
```

5 Bilanciamento di carico in ambiente SMP

TODO