

## SCHEDULING DEI PROCESSI

Come ogni sistema *time sharing* (a condivisione di tempo), Linux dà la magica impressione di eseguire contemporaneamente diversi processi passando da uno all'altro in brevissimo tempo. La commutazione di processo è già stata trattata (vedi "Processi"); qui viene descritto lo scheduling (letteralmente "mettere in fila", ovvero "pianificazione"), che riguarda la scelta di quando eseguire la commutazione e quale processo eseguire. Il riferimento è all'architettura 80x86, ai sistemi che usano il modello di memoria UMA (Uniform Memory Access) e ad un valore di tick di 1 ms.

### Politica di scheduling

L'algoritmo di scheduling nei sistemi Unix tradizionali deve raggiungere alcuni obiettivi contrastanti: tempi di risposta rapidi dei processi, buon rendimento dei job in background, evitare la "starvation" (ovvero l'inedia, cioè l'impossibilità di ottenere le risorse per l'esecuzione), conciliazione delle necessità di processi ad alta e bassa priorità. L'insieme delle regole usate per determinare quando e come selezionare un processo per l'esecuzione è chiamato *politica di scheduling*.

Lo scheduling è basato sulla tecnica del time sharing: molti processi vengono eseguiti con "moltiplicazione a divisione di tempo" (time multiplexing), perché il tempo di CPU è diviso in "slice" (quanti di tempo, letteralmente "fette"), una per ogni processo eseguibile<sup>1</sup>. Chiaramente un processore può eseguire un solo processo alla volta. Se il processo corrente non ha terminato quando si esaurisce il suo quanto di tempo, ha luogo una commutazione di processo. Il time sharing si basa sugli interrupt del timer ed è trasparente ai processi: non è richiesto codice aggiuntivo nei programmi.

La politica di scheduling è basata anche sull'ordinamento dei processi in base alla priorità. A volte si impiegano algoritmi complessi per calcolare la priorità, ma il risultato finale è lo stesso: ad ogni processo è associato un valore che indica allo scheduler quanto è adatto ad essere eseguito.

In Linux la priorità è dinamica. Lo scheduler tiene traccia di ciò che fa ogni processo e adatta periodicamente la sua priorità; in questo modo, processi che non sono stati eseguiti per lungo tempo vengono "accelerati" incrementandone la priorità. Come pure processi che sono stati eseguiti a lungo possono essere penalizzati riducendo la loro priorità.

Riguardo allo scheduling, i processi sono divisi in I/O-dipendenti e CPU-dipendenti. I primi fanno largo uso dei dispositivi di I/O e trascorrono molto tempo in attesa delle operazioni di I/O; i secondi eseguono applicazioni con elaborazioni che fanno un uso intensivo della CPU.

Una classificazione alternativa distingue tre classi di processi:

*Interattivi*: questi interagiscono costantemente con gli utenti, per cui trascorrono molto tempo in attesa della pressione di un tasto o del movimento del mouse. Quando ricevono l'input, devono essere riattivati in fretta o l'utente avrà la sensazione che il sistema non risponde. Il ritardo medio della risposta deve essere tra 50 e 150 millisecondi e la variabilità del ritardo deve essere limitata, altrimenti l'utente avrà la sensazione di un sistema incostante. Tipiche applicazioni interattive sono le shell, gli editor di testo, le applicazioni grafiche.

---

<sup>1</sup> Si ricorda che i processi fermati e sospesi non possono essere scelti dall'algoritmo di scheduling per essere eseguiti.

*Batch*<sup>2</sup> o *non interattivi*: non interagiscono con l'utente e spesso girano in background. Non richiedono rapidi tempi di risposta e vengono spesso penalizzati dallo scheduler. Tipiche applicazioni batch sono i compilatori, i motori di database, i programmi di calcolo scientifici.

*Real-time*<sup>3</sup>: hanno requisiti di scheduling stringenti. Non devono mai essere bloccati da processi a bassa priorità e devono rispondere in tempi brevi e con scostamenti minimi. Sono tipicamente le applicazioni audio e video, i controller di robot e i programmi che raccolgono dati da sensori.

Le due diverse classificazioni sono in qualche modo indipendenti: un processo batch può essere dipendente da I/O (es. server di database) o dalla CPU (es. programma di renderizzazione delle immagini). Mentre i programmi real-time sono riconosciuti esplicitamente dall'algoritmo di scheduling di Linux, non è facile distinguere tra interattivi e batch. Lo scheduler di Linux 2.6 implementa un sofisticato algoritmo euristico che si basa sul comportamento passato del processo per decidere se il processo deve essere considerato batch o interattivo. Naturalmente tenderà a favorire gli interattivi rispetto ai batch. I programmatori possono variare la priorità di scheduling per mezzo delle chiamate di sistema elencate in tabella:

<b>Chiamata di sistema</b>	<b>descrizione</b>
nice()	Cambia la priorità statica di un processo
getpriority()	Restituisce la massima priorità statica di un gruppo di processi
setpriority()	Imposta la priorità statica di un gruppo di processi
sched_getscheduler()	Restituisce la politica di scheduling
sched_setscheduler()	Imposta la politica di scheduling e la priorità real-time di un processo
sched_getparam()	Restituisce la priorità real-time di un processo
sched_setparam()	Imposta la priorità real-time di un processo
sched_yield()	Rilascia volontariamente il processore senza bloccarsi
sched_get_priority_min()	Restituisce la minima priorità real-time per una data politica
sched_get_priority_max()	Restituisce la massima priorità real-time per una data politica
sched_rr_get_interval()	Restituisce il quanto di tempo per la politica Round Robin
sched_setaffinity()	Imposta la maschera di affinità delle CPU per un processo
sched_getaffinity()	restituisce la maschera di affinità delle CPU per un processo

### **Preemption dei processi**

I processi in Linux sono soggetti a preemption (pre-rilascio). Quando uno di essi è nello stato TASK\_RUNNING, il kernel controlla se la sua priorità dinamica è superiore a quella del processo corrente. Se è così, l'esecuzione di current viene sospesa e viene chiamato lo scheduler per selezionare un altro processo (di solito quello divenuto eseguibile). Un processo è soggetto a preemption anche quando si

2 Il termine risale all'epoca delle schede perforate, nella quale i programmatori non avevano accesso diretto agli elaboratori ma preparavano off-line i programmi consegnandoli poi ad un amministratore di sistema che li accodava a quelli già in esecuzione.

3 Risposta in tempo reale significa che la correttezza del risultato dipende dal tempo di risposta.

esaurisce il suo quanto di tempo; il flag `TIF_NEED_RESCHED` della struttura `thread_info` viene attivato e al termine dell'esecuzione del gestore di interrupt timer viene chiamato lo scheduler.

Si consideri il caso in cui solo due programmi, un editor di testo e un compilatore, sono in esecuzione. Il primo è interattivo ed ha una priorità dinamica superiore all'altro. Però viene spesso sospeso, nelle pause di riflessione tra le varie immissioni di dati dell'utilizzatore; inoltre, l'intervallo tra due pressioni dei tasti è un tempo abbastanza lungo. Quando un tasto viene premuto, viene inviato un interrupt e il kernel risveglia l'editor. Inoltre verifica se la sua priorità è superiore a quella di current (del compilatore in questo caso), per cui imposta il flag `TIF_NEED_RESCHED`, determinando così l'attivazione dello scheduler al termine dell'esecuzione del gestore di interrupt. Lo scheduler seleziona l'editor e avvia la commutazione di processo; l'editor riprende rapidamente l'esecuzione e il carattere inviato viene visualizzato sullo schermo. Dopo che il carattere è stato processato, l'editor si sospende in attesa di un'altra pressione di un tasto e il compilatore riprende l'esecuzione.

Da notare che un processo soggetto a preemption rimane nello stato `TASK_RUNNING`; semplicemente, non usa più la CPU. Il kernel 2.6 stesso è soggetto a preemption, per cui anche un processo in modalità del kernel può essere sostituito da un altro.

### **Quanto deve durare un “quanto”?**

La durata del quanto di tempo di esecuzione è un elemento critico per le prestazioni del sistema: non deve essere né troppo lunga né troppo breve. Se la durata media è troppo breve, il sovraccarico del sistema dovuto alle commutazioni di processo diventa troppo elevato. Se, ad esempio, una commutazione richiede 5 millisecondi (ms), e il quanto vale anch'esso 5 ms, il 50 % del tempo di CPU è speso per le commutazioni.

Se la durata è eccessiva, i processi non sembrano più eseguiti contemporaneamente. Se, ad esempio, il quanto è di 5 secondi, ogni processo progredisce per 5 secondi, poi rimane in attesa per un periodo di (5 x numero di processi del sistema) secondi. Spesso si pensa che un quanto lungo degradi i tempi di risposta dei processi interattivi, ma di solito questo è falso. Questi hanno sempre priorità elevata, per cui sostituiscono rapidamente i processi batch, indipendentemente dalla durata del quanto.

In certi casi però la durata eccessiva del quanto degrada la reattività del sistema. Si consideri il caso di due utenti che inseriscono due comandi nelle loro shell: uno lancia un processo batch che richiede CPU e l'altro una applicazione interattiva. Entrambe le shell fanno il fork di un nuovo processo delegandogli l'esecuzione del comando ricevuto. Per ipotesi, i due nuovi processi hanno inizialmente la stessa priorità (Linux non conosce in anticipo il tipo di processo che viene eseguito). Lo scheduler seleziona il batch per primo, per cui l'altro deve attendere un quanto di tempo prima di essere eseguito; se questo tempo è lungo, il sistema appare bloccato all'utente dell'applicazione interattiva.

La scelta del quanto è sempre un compromesso; la regola di Linux è scegliere la durata più lunga possibile compatibilmente con la reattività del sistema.

### **L'algoritmo di scheduling**

L'algoritmo usato nella prime versioni di Linux era molto semplice e lineare: il kernel scandiva la lista dei processi eseguibili, calcolava la loro priorità e sceglieva il migliore. Lo svantaggio maggiore di questo algoritmo è che il tempo necessario per la scelta dipende dal numero di processi, per cui si rivela troppo oneroso per i sistemi che ne gestiscono migliaia.

L'algoritmo di Linux 2.6 è molto più sofisticato. E' scalabile, perché la scelta avviene in un tempo costante, indipendente dal numero di processi, e nei sistemi multiprocessore, ogni CPU ha la propria runqueue, cioè la propria coda di processi eseguibili. Inoltre ha un miglior sistema di identificazione dei processi interattivi e di quelli batch. Di conseguenza, gli utenti di sistemi con grossi carichi di elaborazione percepiscono una migliore reattività delle applicazioni.

Lo scheduler può sempre trovare un processo eseguibile: esiste sempre almeno il processo swapper, che ha PID 0 e viene eseguito solo quando non ci sono altri processi eseguibili. Nei sistemi multiprocessore, ogni CPU ha un proprio swapper. Ogni processo in Linux appartiene ad una delle seguenti classi di scheduling:

**SCHED\_FIFO:** processi real-time FIFO (first in, first out). Quando lo scheduler avvia un processo, lo lascia al suo posto nella coda di esecuzione. Se nessun altro tra i real-time ha priorità superiore, il processo continua ad usare la CPU finché vuole.

**SCHED\_RR:** processi real-time in Round Robin. Quando lo scheduler avvia un processo, lo colloca in fondo alla coda di esecuzione. Questo assicura omogeneità di tempo di CPU per i processi con la stessa priorità.

**SCHED\_NORMAL:** processi comuni.

L'algoritmo si comporta in maniera diversa a seconda se il processo è comune o real-time.

### Schedulazione di processi comuni

Ognuno di essi ha la propria *priorità statica*, un valore usato dallo scheduler per ordinare i processi comuni; il valore varia da 100 (priorità massima) a 139 (minima). Un figlio eredita la priorità statica del padre. Però l'utente proprietario può cambiarla passando alle chiamate di sistema `nice()` e `setpriority()` un valore nice.

### Quanto di tempo base

La priorità statica determina il *quanto di tempo base* di un processo, cioè il tempo di esecuzione assegnato dopo aver esaurito quello precedente. I due valori sono legati dalla relazione seguente:

$$\begin{aligned} \text{quanto base (in millisecondi)} &= (140 - \text{priorità statica}) \times 20 && \text{se priorità} < 120 && \text{oppure} \\ &= (140 - \text{priorità statica}) \times 5 && \text{se priorità} \geq 120 && (1) \end{aligned}$$

Quindi, quanto più alta è la priorità (più basso il suo valore numerico), tanto più lungo è il quanto. La tabella riassume i valori per diverse priorità: il significato dei campi "interactive delta" e "sleep time threshold" (soglia di inattività) sarà chiarito in seguito.

Descrizione	Priorità statica	Valore nice	Quanto base	Interactive delta	Sleep time threshold
Priorità massima	100	-20	800 ms	-3	299 ms
Priorità alta	110	-10	600 ms	-1	499 ms
Priorità di default	120	0	100 ms	+2	799 ms
Priorità bassa	130	+10	50 ms	+4	999 ms
Priorità minima	139	+19	5 ms	+6	1199 ms

### Priorità dinamica e tempo medio di attesa

Oltre alla priorità statica, ogni processo ne possiede una *dinamica*, che va da 100 (maggiore) a 139 (minore); è il valore controllato dallo scheduler quando deve selezionare il candidato all'esecuzione. E' correlato alla priorità statica dalla seguente relazione empirica:

$$\text{priorità dinamica} = \max(100, \min(\text{priorità statica} - \text{bonus} + 5, 139)) \quad (2)$$

Il bonus è un valore che va da 0 a 10; un valore < 5 è una penalizzazione che abbassa la priorità dinamica, mentre un valore > 5 è un premio che la aumenta. Il bonus dipende dalla storia passata del processo ed è correlata al *tempo medio di attesa* (*average sleep time*). Questo è il numero medio di nanosecondi trascorso dal processo in attesa e non deriva da una media del tempo trascorso. Infatti l'attesa nello stato TASK\_INTERRUPTIBLE contribuisce in modo diverso da quella nello stato TASK\_UNINTERRUPTIBLE alla media; inoltre l'attesa media diminuisce mentre il processo è in esecuzione. Infine, il suo valore non può essere superiore a 1 secondo.

La corrispondenza fra tempo medio di attesa e bonus è mostrata in tabella (il significato del campo granularità sarà chiarito in seguito).

Tempo medio di attesa	Bonus	granularità
Tra 0 e 100 ms	0	5120
Tra 100 e 200 ms	1	2560
Tra 200 e 300 ms	2	1280
Tra 300 e 400 ms	3	640
Tra 400 e 500 ms	4	320
Tra 500 e 600 ms	5	160
Tra 600 e 700 ms	6	80
Tra 700 e 800 ms	7	40
Tra 800 e 900 ms	8	20
Tra 900 e 1000 ms	9	10
1 secondo	10	10

Il tempo medio di attesa è usato dallo scheduler anche per stabilire se un processo deve essere considerato interattivo o batch; è considerato interattivo se:

$$\text{priorità dinamica} \leq 3 \times \text{priorità statica} / 4 + 28 \quad (3)$$

equivalente a:

$$\text{bonus} - 5 \geq \text{priorità statica} / 4 - 28$$

L'espressione  $\text{priorità statica} / 4 - 28$  è chiamata *delta interattivo*; alcuni valori tipici sono stati elencati in una tabella precedente. Si nota che è più facile per un processo a priorità elevata diventare interattivo. Ad

esempio, un processo con priorità statica massima (100) è considerato interattivo quando il suo bonus supera 2, cioè quando il tempo medio di attesa supera 200 ms; invece uno con priorità statica minima (139) non è mai considerato interattivo, perché il bonus è sempre inferiore a 11, cioè al valore richiesto per raggiungere un delta interattivo di 6. Un processo con priorità statica standard (120) diventa interattivo con un tempo medio di attesa di 700 ms o più.

### Processi attivi o expired

Anche se i processi con priorità statica maggiore ottengono quanti di tempo più lunghi, essi non devono bloccare completamente quelli a priorità inferiore. Per evitare "l'inedia" (starvation), quando un processo termina il suo quanto, può essere sostituito da un processo a bassa priorità che non ha esaurito il proprio. Per far questo, lo scheduler mantiene due gruppi distinti di processi eseguibili:

*processi attivi*: non hanno terminato il loro quanto di tempo e possono essere eseguiti;

*processi expired*: hanno terminato il loro quanto e non possono essere eseguiti fino a che non sono stati eseguiti tutti gli attivi.

Lo schema è un po' più complesso, perché lo scheduler tenta di migliorare le performance dei processi interattivi. Un processo batch attivo che termina il suo quanto diventa sempre expired. Un processo interattivo attivo che termina il suo quanto di solito rimane attivo: lo scheduler ripristina il suo quanto e lo lascia nel gruppo degli attivi. Lo sposta nel gruppo degli expired se il più vecchio di questi ha già atteso a lungo oppure se un processo expired ha priorità statica maggiore del processo interattivo. Di conseguenza, se il gruppo degli attivi rimane vuoto, i processi expired hanno un'occasione.

### Scheduling dei processi real-time

Ad ognuno di essi è associata una *priorità real-time*, il cui valore va da 1 (priorità massima) a 99 (minima). Lo scheduler favorisce sempre un processo a priorità maggiore; in altre parole, un processo impedisce l'esecuzione di tutti quelli a priorità inferiore finché rimane eseguibile. Contrariamente a quelli comuni, i real-time sono sempre considerati nel gruppo degli attivi. L'utente può modificare la priorità real-time per mezzo delle chiamate di sistema `sched_setparam()` e `sched_setscheduler()`. Se vari processi real-time hanno la stessa priorità, lo scheduler sceglie quello che viene prima nella coda di esecuzione della CPU.

Un processo real-time è sostituito da un altro solo se si verifica una delle seguenti condizioni:

- il processo è soggetto a preemption da parte di uno a priorità real-time maggiore;
- esegue un'operazione bloccante e viene messo in attesa (nello stato `TASK_INTERRUPTIBLE` o `TASK_UNINTERRUPTIBLE`);
- viene arrestato (stato `TASK_STOPPED` o `TASK_TRACED`) o ucciso (`EXIT_ZOMBIE` o `EXIT_DEAD`);
- rilascia volontariamente la CPU chiamando `sched_yield()`;
- è un processo real-time Round Robin (`SCHED_RR`) e ha esaurito il quanto di tempo.

Le chiamate di sistema `nice()` e `setpriority()`, applicate a quest'ultimo tipo di processi, non cambiano la priorità real-time, ma piuttosto la durata del quanto base. Infatti questo intervallo di tempo non dipende dalla priorità real-time, ma da quella statica, in accordo con la formula (1) vista in precedenza.

## Strutture dati usate dallo scheduler

La lista dei processi collega tutti i descrittori di processo, mentre le runqueue collegano i descrittori dei processi eseguibili, cioè quelli nello stato TASK\_RUNNING, eccetto lo swapper.

### La struttura runqueue

La struttura runqueue è la struttura più importante dello scheduler di Linux 2.6; ogni CPU ne possiede una propria; tutte le runqueue sono memorizzate nella variabile per-CPU runqueues. La macro this\_rq() restituisce l'indirizzo della runqueue della CPU locale, mentre la macro cpu\_rq(n) restituisce l'indirizzo della runqueue della CPU numero n.

```
struct runqueue {
    spinlock_t lock;

    /* nr_running and cpu_load should be in the same cacheline because
       remote CPUs use both these fields when doing load calculation.*/
    unsigned long nr_running;          /* numero di processi eseguibili nella lista */
#ifdef CONFIG_SMP
    unsigned long cpu_load;           /* fattore di carico della CPU basato sul numero */
#endif                               /* di processi in coda */
    unsigned long long nr_switches;   /* n° di commutazioni di processo eseguite dalla CPU */

    /* This is part of a global counter where only the total sum over all CPUs matters.
       A task can increase this counter on one CPU and if it got migrated afterwards it may
       decrease it on another CPU. Always updated under the runqueue lock:*/

    unsigned long nr_uninterruptible; /* n° di processi che erano in lista e ora sono nello stato
                                         TASK_UNINTERRUPTIBLE */
    unsigned long expired_timestamp;   /* tempo di inserimento del più vecchio processo nella
                                         lista dei processi expired*/
    unsigned long long timestamp_last_tick; /* timestamp dell'ultimo interrupt */
    task_t *curr;                     /* puntatore al descrittore del processo corrente */
    task_t *idle;                     /* puntatore al processo idle della CPU */
    struct mm_struct *prev_mm;        /* contiene il descrittore di memoria del processo
                                         rimpiazzato in una commutazione */
    prio_array_t *active;             /* puntatore alla lista dei processi attivi */
    prio_array_t *expired;           /* puntatore alla lista dei processi expired*/
    prio_array_t arrays[2];          /* i due gruppi di processi */
    int best_expired_prio;            /* la priorità statica più alta tra i processi expired*/
    atomic_t nr_iowait;              /* n° dei processi che erano in lista e ora attendono il
                                         completamento di un'operazione di I/O su disco */
#ifdef CONFIG_SMP
    struct sched_domain *sd;         /* punta al dominio base di scheduling della CPU */
#endif
};
```

```

/* For active balancing */
int active_balance;
int push_cpu;
task_t *migration_thread;
struct list_head migration_queue;
#endif
#ifdef CONFIG_SCHEDSTATS
/* latency stats */
struct sched_info rq_sched_info;
/* sys_sched_yield() stats */
unsigned long yld_exp_empty;
unsigned long yld_act_empty;
unsigned long yld_both_empty;
unsigned long yld_cnt;
/* schedule() stats */
unsigned long sched_noswitch;
unsigned long sched_switch;
unsigned long sched_cnt;
unsigned long sched_goidle;
/* pull_task() stats */
unsigned long pt_gained[MAX_IDLE_TYPES];
unsigned long pt_lost[MAX_IDLE_TYPES];
/* active_load_balance() stats */
unsigned long alb_cnt;
unsigned long alb_lost;
unsigned long alb_gained;
unsigned long alb_failed;
/* try_to_wake_up() stats */
unsigned long ttwu_cnt;
unsigned long ttwu_attempts;
unsigned long ttwu_moved;
/* wake_up_new_task() stats */
unsigned long wunt_cnt;
unsigned long wunt_moved;
/* sched_migrate_task() stats */
unsigned long smt_cnt;
/* sched_balance_exec() stats */
unsigned long sbe_cnt;
#endif
};

```

I campi più importanti sono quelli correlati alle liste dei processi eseguibili. Ognuno di essi appartiene ad una sola runqueue. Finché rimane nella stessa runqueue, può essere eseguito solo dalla CPU a cui appartiene la coda. In alcuni casi però i processi possono essere trasferiti ad altre runqueue.

Il campo `arrays` è un array di due strutture `prio_array_t`, ognuna delle quali rappresenta un gruppo di processi eseguibili e comprende 140 elementi iniziali di liste a doppio collegamento, una per ogni possibile valore di priorità, una mappa di bit di priorità e un contatore dei processi inclusi nel gruppo.

Il campo `active` della `runqueue` punta a una delle due strutture `prio_array_t` contenute in `arrays`: il gruppo di processi corrispondente è quello dei processi attivi. Il campo `expired` punta all'altra struttura: il gruppo è quello dei processi `expired`.

Periodicamente, il ruolo delle due strutture cambia, e i processi attivi diventano scaduti e viceversa: per operare questo scambio, lo scheduler scambia semplicemente il contenuto dei campi `active` ed `expired`.

## Descrittore di processo

Ogni descrittore di processo contiene diversi campi collegati allo scheduling:

```
volatile long state;           /* stato corrente del processo */
struct thread_info *thread_info; /* thread_info ->flags contiene TIF_NEED_RESCHED */
                                /* thread_info->cpu contiene il numero della CPU a cui appartiene la
                                coda*/
int prio;                      /* priorità dinamica del processo */
int static_prio;               /* priorità statica del processo */
struct list_head run_list;     /* puntatore agli elementi precedente e successivo della lista */
prio_array_t *array;          /* puntatore al gruppo prio_array_t della runqueue */
unsigned long sleep_avg;       /* tempo medio di attesa del processo */
unsigned long long timestamp;  /* timestamp dell' ultimo inserimento del processo in coda */
                                /* o dell'ultima commutazione di processo che lo ha coinvolto */
unsigned long long last_ran;   /* timestamp dell'ultima commutazione che ha coinvolto il processo */
int activated;                 /* codice di condizione usato quando il processo viene risvegliato */
unsigned long policy;          /* classe di schedulazione: SCHED_NORMAL, SCHED_RR o SCHED_FIFO */
cpumask_t cpus_allowed;       /* mappa di bit delle CPU che possono eseguire il processo */
unsigned int time_slice;       /* tick rimasti nel quanto del processo */
unsigned int first_time_slice; /* flag impostato a 1 se il processo non ha esaurito il quanto */
unsigned long rt_priority      /* priorità real_time del processo */
```

Quando viene creato un nuovo processo, `sched_fork()` chiamata da `copy_process()`, imposta il quanto di tempo sia di `current` (il genitore) che di `p` (il figlio) in questo modo:

```
p->time_slice = (current->time_slice + 1) >> 1;
current->time_slice >>= 1;
```

Il numero di tick rimasti al genitore viene diviso in due metà, una per il figlio e una per il genitore; questo serve ad impedire che gli utenti acquisiscano un tempo di CPU illimitato con la seguente tecnica: il genitore crea un figlio che esegue il suo stesso codice, poi si uccide; il figlio otterrebbe un intero quanto di tempo prima dello scadere di quello del genitore, e così via. Il trucco non funziona perché il kernel non premia il `fork`. Allo stesso modo, un utente non può appropriarsi del processore creando molti processi in `background` da una shell o aprendo molte finestre sul desktop grafico. Più in generale, un processo non può ap-

propriarsi delle risorse (a meno che non sia schedulato come processo real-time) creando molti figli con fork successivi.

Se il genitore ha solo un tick residuo, la divisione porterebbe a `current->time_slice = 0`; in questo caso `copy_process()` riporta `current->time_slice` a 1.

La funzione `copy_process()` inizializza altri campi del figlio legati allo scheduling:

```
p->first_time_slice = 1;
p->timestamp = sched_clock();
```

`first_time_slice` è impostato a 1 perché il figlio non ha ancora esaurito il suo primo quanto (se un processo termina o ne avvia un altro durante il suo primo quanto, il genitore è premiato con il quanto rimanente del figlio). Il valore di `timestamp` è inizializzato con il valore restituito da `sched_clock()`, cioè il contenuto del registro TSC a 64 bit.

## Funzioni usate dallo scheduler

Lo scheduler si basa su alcune funzioni fondamentali:

`scheduler_tick()`: mantiene aggiornato il contatore `current->time_slice`;

`try_to_wake_up()`: risveglia un processo in attesa;

`recalc_task_prio()`: aggiorna la priorità dinamica del processo;

`schedule()`: seleziona un nuovo processo per l'esecuzione;

`load_balance()`: mantiene bilanciate le runqueue di un sistema multiprocessore.

### `scheduler_tick()`

```
void scheduler_tick(void)
{
    int cpu = smp_processor_id();
    runqueue_t *rq = this_rq();
    task_t *p = current;

    rq->timestamp_last_tick = sched_clock();
    if (p == rq->idle) {
        if (wake_priority_sleeper(rq))
            goto out;
        rebalance_tick(cpu, rq, SCHED_IDLE);
        return;
    }
    /* Task might have expired already, but not scheduled off yet */
    if (p->array != rq->active) {
        set_tsk_need_resched(p);
    }
}
```

```

        goto out;
    }
    spin_lock(&rq->lock);
    /*
     * The task was running during this tick - update the
     * time slice counter. Note: we do not update a thread's
     * priority until it either goes to sleep or uses up its
     * timeslice. This makes it possible for interactive tasks
     * to use up their timeslices at their highest priority levels.
     */
    if (rt_task(p)) {
        /*RR tasks need a special form of timeslice management */
        if ((p->policy == SCHED_RR) && !p->time_slice) {
            p->time_slice = task_timeslice(p);
            p->first_time_slice = 0;
            set_tsk_need_resched(p);
            /* put it at the end of the queue: */
            requeue_task(p, rq->active);
        }
        goto out_unlock;          /* FIFO tasks have no timeslices. */
    }
    if (!p->time_slice) {
        dequeue_task(p, rq->active);
        set_tsk_need_resched(p);
        p->prio = effective_prio(p);
        p->time_slice = task_timeslice(p);
        p->first_time_slice = 0;

        if (!rq->expired_timestamp)
            rq->expired_timestamp = jiffies;
        if (!TASK_INTERACTIVE(p) || EXPIRED_STARVING(rq)) {
            enqueue_task(p, rq->expired);
            if (p->static_prio < rq->best_expired_prio)
                rq->best_expired_prio = p->static_prio;
        } else
            enqueue_task(p, rq->active);
    } else {
        /*
         * Prevent a too long timeslice allowing a task to monopolize
         * the CPU. We do this by splitting up the timeslice into
         * smaller pieces.
         * Note: this does not mean the task's timeslices expire or
         * get lost in any way, they just might be preempted by
         * another task of equal priority. (one with higher

```

```

* priority would have preempted this task already.) We
* requeue this task to the end of the list on this priority
* level, which is in essence a round-robin of tasks with
* equal priority.
* This only applies to tasks in the interactive
* delta range with at least TIMESLICE_GRANULARITY to requeue.
*/
if (TASK_INTERACTIVE(p) && !((task_timeslice(p) -
    p->time_slice) % TIMESLICE_GRANULARITY(p)) &&
    (p->time_slice >= TIMESLICE_GRANULARITY(p)) &&
    (p->array == rq->active)) {
    requeue_task(p, rq->active);
    set_tsk_need_resched(p);
}
}
out_unlock:
    spin_unlock(&rq->lock);
out:
    rebalance_tick(cpu, rq, NOT_IDLE);
}

```

La funzione è attivata ad ogni tick per svolgere le seguenti operazioni:

1 – Memorizza nel campo `timestamp_last_tick` della runqueue locale il valore corrente del TSC in nanosecondi, ottenuto da `sched_clock()`.

2 – Controlla se `current` è il processo swapper: se è così, esegue i seguenti passi:

- se la coda contiene altri processo oltre a swapper, imposta il flag `TIF_NEED_RESCHED` di `current` per forzare lo scheduling. Se il kernel supporta la tecnologia hyper-threading, una CPU potrebbe essere inattiva anche se nella sua runqueue ci sono processi eseguibili, se tali processi hanno una priorità molto inferiore rispetto a quella di un processo già in esecuzione su un'altra CPU logica associata con la CPU fisica.
- Salta al punto 7 (non occorre aggiornare il quanto dello swapper).

3 – controlla se `current->array` punta al gruppo dei processi attivi. Se non è così, il processo ha esaurito il quanto ma non è stato ancora sostituito: attiva `TIF_NEED_RESCHED` per forzare lo scheduling, poi salta al punto 7.

4 – Acquisisce lo spin-lock.

5 – Decrementa il contatore `time_slice` e controlla se il quanto è esaurito. Le operazioni sono diverse in base alla classe di scheduling del processo e verranno illustrate più avanti.

6 – Rilascia lo spin lock.

7 – Chiama la funzione `rebalance_tick()` per controllare che le CPU abbiano circa lo stesso numero di processi eseguibili.

### **Aggiornamento del time\_slice di un processo real-time**

Se `current` è un processo real-time FIFO, la funzione non ha nulla da fare; infatti `current` non può essere soggetto a preemption da parte di processi a priorità inferiore o uguale, per cui non ha senso aggiornare il suo `time_slice`.

Se `current` è un processo real-time Round Robin, `scheduler_tick()` decrementa il suo `time_slice` e controlla il quanto: se è esaurito, effettua alcune operazioni per assicurare che `current` venga sostituito. Per prima cosa ripristina il valore di `time_slice` chiamando `task_timeslice()`. Questa funzione considera la priorità statica del processo e restituisce il quanto base secondo la formula (1). Poi viene azzerato `first_time_slice`: questo flag viene attivato da `copy_process()` nell'ambito della chiamata di sistema `fork()`, e deve essere azzerato quando il primo quanto viene consumato.

La funzione chiama poi `set_tsk_need_resched()` per attivare il flag `TIF_NEED_RESCHED`, che comporta la sostituzione di `current` con un altro processo real-time con priorità maggiore o uguale, se esiste.

L'ultima operazione consiste nello spostare il descrittore all'ultimo posto della coda dei processi attivi che corrisponde alla priorità di `current`. In questo modo il processo non viene selezionato per l'esecuzione fino a che non hanno ottenuto la CPU gli altri processi real-time di pari priorità. Questo è il significato della politica Round Robin.

### **Aggiornamento del time\_slice di un processo convenzionale**

In questo caso `scheduler_tick()` compie le seguenti azioni:

1 – Decrementa `current->time_slice`.

2 – Controlla il suo valore: se il quanto è esaurito:

- chiama `dequeue_task()` per rimuoverlo dal gruppo `this_rq()->active`;
- chiama `set_tsk_need_resched()` per attivare `TIF_NEED_RESCHED`;
- aggiorna la priorità dinamica: `current->prio = effective_prio(current)`; questa funzione legge i campi `static_prio` e `sleep_avg` di `current` e calcola la priorità dinamica secondo la formula (2);
- ripristina il valore di `time_slice` e azzerava `first_time_slice`;
- se il campo `expired_timestamp` della runqueue locale è zero (e cioè il gruppo dei processi expired è vuoto), scrive nel campo il valore del tick corrente;
- inserisce `current` nella lista dei processi attivi o in quella dei processi expired: la macro `TASK_INTERACTIVE` restituisce 1 se il processo è riconosciuto come interattivo usando la formula (3). La macro `EXPIRED_STARVING` controlla se il primo processo nella coda di quelli expired ha dovuto attendere più di 1000 tick volte il numero dei processi nella coda degli eseguibili più uno: se è così la macro restituisce 1. La macro restituisce 1 anche se la priorità statica di `current` è maggiore di quella di ogni processo expired.

3 - Se il quanto non è terminato, controlla se `time_slice` è troppo lungo: la macro `TIMESLICE_GRANULARITY` restituisce il prodotto del numero di CPU del sistema e una costante proporzionale al bonus di `current`. Il

quanto di processi interattivi con alta priorità statica viene diviso in varie parti di dimensioni pari a `TIMESLICE_GRANULARITY` in modo che non monopolizzino la CPU.

### **try\_to\_wake\_up()**

```
static int try_to_wake_up(task_t * p, unsigned int state, int sync)
{
    int cpu, this_cpu, success = 0;
    unsigned long flags;
    long old_state;
    runqueue_t *rq;
#ifdef CONFIG_SMP
    unsigned long load, this_load;
    struct sched_domain *sd;
    int new_cpu;
#endif
    rq = task_rq_lock(p, &flags);
    schedstat_inc(rq, ttwu_cnt);
    old_state = p->state;
    if (!(old_state & state))
        goto out;
    if (p->array)
        goto out_running;
    cpu = task_cpu(p);
    this_cpu = smp_processor_id();
#ifdef CONFIG_SMP
    if (unlikely(task_running(rq, p)))
        goto out_activate;

    new_cpu = cpu;
    if (cpu == this_cpu || unlikely(!cpu_isset(this_cpu, p->cpus_allowed)))
        goto out_set_cpu;
    load = source_load(cpu);
    this_load = target_load(this_cpu);
    /*
     * If sync wakeup then subtract the (maximum possible) effect of
     * the currently running task from the load of the current CPU:
     */
    if (sync)
        this_load -= SCHED_LOAD_SCALE;

    /* Don't pull the task off an idle CPU to a busy one */
    if (load < SCHED_LOAD_SCALE/2 && this_load > SCHED_LOAD_SCALE/2)
        goto out_set_cpu;
```

```

new_cpu = this_cpu; /* Wake to this CPU if we can */
/*
 * Scan domains for affine wakeup and passive balancing
 * possibilities.
 */
for_each_domain(this_cpu, sd) {
    unsigned int imbalance;
    /*
     * Start passive balancing when half the imbalance_pct
     * limit is reached.
     */
    imbalance = sd->imbalance_pct + (sd->imbalance_pct - 100) / 2;

    if ((sd->flags & SD_WAKE_AFFINE) &&
        !task_hot(p, rq->timestamp_last_tick, sd)) {
        /*
         * This domain has SD_WAKE_AFFINE and p is cache cold
         * in this domain.
         */
        if (cpu_isset(cpu, sd->span)) {
            schedstat_inc(sd, ttwu_wake_affine);
            goto out_set_cpu;
        }
    } else if ((sd->flags & SD_WAKE_BALANCE) &&
        imbalance*this_load <= 100*load) {
        /*
         * This domain has SD_WAKE_BALANCE and there is
         * an imbalance.
         */
        if (cpu_isset(cpu, sd->span)) {
            schedstat_inc(sd, ttwu_wake_balance);
            goto out_set_cpu;
        }
    }
}
new_cpu = cpu; /* Could not wake to this_cpu. Wake to cpu instead */
out_set_cpu:
    schedstat_inc(rq, ttwu_attempts);
    new_cpu = wake_idle(new_cpu, p);
    if (new_cpu != cpu) {
        schedstat_inc(rq, ttwu_moved);
        set_task_cpu(p, new_cpu);
        task_rq_unlock(rq, &flags);
        /* might preempt at this point */
    }

```

```

        rq = task_rq_lock(p, &flags);
        old_state = p->state;
        if (!(old_state & state))
            goto out;
        if (p->array)
            goto out_running;

        this_cpu = smp_processor_id();
        cpu = task_cpu(p);
    }
out_activate:
#endif /* CONFIG_SMP */
    if (old_state == TASK_UNINTERRUPTIBLE) {
        rq->nr_uninterruptible--;
        /*
         * Tasks on involuntary sleep don't earn
         * sleep_avg beyond just interactive state.
         */
        p->activated = -1;
    }
    /*
     * Sync wakeups (i.e. those types of wakeups where the waker
     * has indicated that it will leave the CPU in short order)
     * don't trigger a preemption, if the woken up task will run on
     * this cpu. (in this case the 'I will reschedule' promise of
     * the waker guarantees that the freshly woken up task is going
     * to be considered on this CPU.)
     */
    activate_task(p, rq, cpu == this_cpu);
    if (!sync || cpu != this_cpu) {
        if (TASK_PREEMPTS_CURR(p, rq))
            resched_task(rq->curr);
    }
    success = 1;
out_running:
    p->state = TASK_RUNNING;
out:
    task_rq_unlock(rq, &flags);
    return success;
}

```

La funzione risveglia un processo inserito in una *wait queue* o in attesa di un segnale, cambiando il suo stato in `TASK_RUNNING` e inserendolo nella runqueue della CPU locale. Riceve come parametri:

- il puntatore (p) al descrittore del processo da risvegliare;
- una maschera (state) degli stati di processo che possono essere risvegliati;
- un flag (sync) che impedisce al processo risvegliato di esercitare la preemption sul processo corrente della CPU.

La funzione compie le seguenti operazioni:

1 – Chiama `task_rq_lock()` per disabilitare gli interrupt locali e acquisire il lock della runqueue `rq` della CPU su cui è stato eseguito per l'ultima volta il processo (e che può essere diversa dalla CPU locale); il numero di tale CPU è conservato nel campo `p->thread_info->cpu`.

2 – Controlla se lo stato del processo `p->state` è compreso nella maschera ricevuta come argomento; se non lo è, termina.

3 - Se il campo `p->array` non è NULL, il processo è già inserito in una coda: salta al punto 8.

4 – Nei sistemi multiprocessore controlla se il processo va spostato ad un'altra CPU. La funzione sceglie una runqueue in base ad alcune regole euristiche:

- se una CPU è inattiva, sceglie la sua runqueue; la preferenza è per la CPU dell'ultima esecuzione, poi per la CPU locale, in questo ordine;
- se il carico di lavoro della CPU dell'ultima esecuzione è inferiore a quello della CPU locale, sceglie la prima;
- se il processo è stato eseguito di recente, sceglie la runqueue dell'ultima esecuzione (per sfruttare la cache);
- se spostare il processo sulla CPU locale aumenta il bilanciamento dei carichi, lo fa.

A questo punto la funzione ha identificato la CPU target e, di conseguenza, la runqueue in cui inserire il processo.

5 – Se il processo è nello stato `TASK_UNINTERRUPTIBLE`, decrementa il campo `nr_uninterruptible` della runqueue target e imposta a -1 il campo `p->activated`. Più avanti ne viene spiegato il significato.

6 – Chiama `activate_task()` che esegue le seguenti operazioni:

- chiama `sched_clock()` per avere il timestamp corrente in nanosecondi; se la CPU target non è quella locale, compensa lo scostamento del timer locale usando i timestamp relativi agli ultimi interrupt sulla CPU locale e target:

```
now = (sched_clock() - this_rq()->timestamp_last_tick) + rq->timestamp_last_tick;
```

- chiama `recalc_task_prio()` passandole il puntatore `p` e il timestamp appena calcolato;
- imposta il valore di `p->activated`;
- imposta `p->timestamp` con il valore calcolato prima;
- inserisce il descrittore nel gruppo dei processi attivi:

```
enqueue_task(p, rq->active); rq->nr_running++;
```

7 – Se la CPU target non è quella locale o il flag sync non è attivato, controlla se il nuovo processo eseguibile ha priorità dinamica maggiore di quello corrente della runqueue rq; se è così, chiama `resched_task()` per sostituire `rq->curr`. Nei sistemi uniprocessore questa funzione esegue `set_tsk_need_resched()` per attivare `TIF_NEED_RESCHED`. Nei sistemi multiprocessore controlla se il vecchio valore di `TIF_NEED_RESCHED` era 0, se la CPU target è diversa da quella locale e se il flag `TIF_POLLING_NRFLAG` di `rq->curr` è azzerato (la CPU target non sta testando attivamente `TIF_NEED_RESCHED`); se è così, chiama `smpt_send_reschedule()` per inviare un interrupt interprocessore e forzare un rescheduling sulla CPU target.

8 – Imposta `p->state` a `TASK_RUNNING`.

9 – Chiama `task_rq_unlock()` per sbloccare il lock di rq e riabilitare gli interrupt locali.

10 – Restituisce 1 se il processo è stato risvegliato e 0 in caso contrario.

### **recalc\_task\_prio()**

La funzione aggiorna il tempo medio di attesa e la priorità dinamica di un processo. Riceve come parametri il puntatore p al descrittore e un timestamp now calcolato da `sched_clock()`, ed esegue le seguenti azioni:

1 – Memorizza nella variabile `sleep_time` il risultato di:

$$\min(\text{now} - p \rightarrow \text{timestamp}, 10^9)$$

`p->timestamp` contiene il timestamp della commutazione di processo che ha sospeso p; perciò `sleep_time` contiene il numero di nanosecondi che il processo ha dovuto attendere dall'ultima esecuzione oppure 1 secondo, se il tempo di attesa è stato maggiore.

2 – se `sleep_time` non è maggiore di zero salta al punto 8 senza aggiornare il tempo medio di attesa.

3 – Controlla se il processo non è un thread del kernel, se è stato risvegliato dallo stato `TASK_UNINTERRUPTIBLE` (`p->activate = -1`) e se è stato “sveglio” in modo continuativo dopo una certa soglia di inattività: se le tre condizioni sono soddisfatte, imposta `p->sleep_avg` a 900 tick, un valore empirico che deriva dalla differenza tra il quanto base di un processo standard e il massimo tempo medio di attesa. Poi salta al punto 8.

La soglia di inattività dipende dalla priorità statica e alcuni valori tipici sono stati riportati in una tabella precedente. L'obiettivo di questo valore è di assicurare ad un processo rimasto per lungo tempo in modalità `uninterruptible` - di solito aspettando una operazione di I/O su disco – ottenga un valore di `sleep_avg` abbastanza lungo da poter compiere il proprio lavoro in fretta, ma senza provocare starvation degli altri processi.

4 – Esegue la macro `CURRENT_BONUS` per calcolare il valore del bonus del precedente tempo medio di attesa del processo. Se  $(10 - \text{bonus})$  è maggiore di zero, la funzione moltiplica `sleep_time` per questo valore. Poiché `sleep_time` verrà aggiunto al tempo medio di attesa, minore è il tempo medio di attesa corrente, più rapidamente il processo sarà risvegliato.

5 – Se il processo è nello stato `TASK_UNINTERRUPTIBLE` e non è un thread del kernel:

- controlla se `p->sleep_avg` è maggiore o uguale alla soglia di attesa (`sleep time threshold`); se è così, reimposta a zero la variabile locale `sleep_avg` e salta al punto 6;
- se  $(\text{sleep\_avg} + p \rightarrow \text{sleep\_avg}) \geq \text{sleep time threshold}$ , imposta `sleep_avg = 0` e

```
p->sleep_avg = sleep_time_threshold;
```

Per limitare in qualche modo l'aumento del tempo medio di attesa, la funzione non premia troppo i processi batch che rimangono quiescenti per lungo tempo.

6 – Aggiunge sleep\_time a p->sleep\_avg.

7 – Controlla se p->sleep\_avg supera i 1000 tick (in nanosecondi); se è così imposta il valore a 1000 tick (sempre in nanosecondi).

8 – Aggiorna la priorità dinamica del processo:

```
p->prio = effective_prio(p);
```

## La funzione schedule()

Questa funzione implementa lo scheduler. Il suo obiettivo è di trovare un processo nella runqueue e assegnargli la CPU; viene chiamata o direttamente o in modalità lazy da varie routine del kernel.

```
asmlinkage void __sched schedule(void)
{
    long *switch_count;
    task_t *prev, *next;
    runqueue_t *rq;
    prio_array_t *array;
    struct list_head *queue;
    unsigned long long now;
    unsigned long run_time;
    int cpu, idx;
    /*
     * Test if we are atomic. Since do_exit() needs to call into
     * schedule() atomically, we ignore that path for now.
     * Otherwise, whine if we are scheduling when we should not be.
     */
    if (likely(!current->exit_state)) {
        if (unlikely(in_atomic())) {
            printk(KERN_ERR "scheduling while atomic: "
                "%s/0x%08x/%d\n",
                current->comm, preempt_count(), current->pid);
            dump_stack();
        }
    }
    profile_hit(SCHED_PROFILING, __builtin_return_address(0));
need_resched:
    preempt_disable();
```

```

    prev = current;
    release_kernel_lock(prev);
need_resched_nonpreemptible:
    rq = this_rq();
    /*
     * The idle thread is not allowed to schedule!
     * Remove this check after it has been exercised a bit.
     */
    if (unlikely(prev == rq->idle) && prev->state != TASK_RUNNING) {
        printk(KERN_ERR "bad: scheduling from the idle thread!\n");
        dump_stack();
    }
    schedstat_inc(rq, sched_cnt);
    now = sched_clock();
    if (likely(now - prev->timestamp < NS_MAX_SLEEP_AVG))
        run_time = now - prev->timestamp;
    else
        run_time = NS_MAX_SLEEP_AVG;
    /*
     * Tasks charged proportionately less run_time at high sleep_avg to
     * delay them losing their interactive status
     */
    run_time /= (CURRENT_BONUS(prev) ? : 1);

    spin_lock_irq(&rq->lock);

    if (unlikely(prev->flags & PF_DEAD))
        prev->state = EXIT_DEAD;

    switch_count = &prev->nivcsw;
    if (prev->state && !(preempt_count() & PREEMPT_ACTIVE)) {
        switch_count = &prev->nvcs;
        if (unlikely((prev->state & TASK_INTERRUPTIBLE) &&
            unlikely(signal_pending(prev))))
            prev->state = TASK_RUNNING;
        else {
            if (prev->state == TASK_UNINTERRUPTIBLE)
                rq->nr_uninterruptible++;
            deactivate_task(prev, rq);
        }
    }
    cpu = smp_processor_id();
    if (unlikely(!rq->nr_running)) {
go_idle:

```

```

idle_balance(cpu, rq);
if (!rq->nr_running) {
    next = rq->idle;
    rq->expired_timestamp = 0;
    wake_sleeping_dependent(cpu, rq);
    /*
     * wake_sleeping_dependent() might have released
     * the runqueue, so break out if we got new
     * tasks meanwhile:
     */
    if (!rq->nr_running)
        goto switch_tasks;
}
} else {
    if (dependent_sleeper(cpu, rq)) {
        next = rq->idle;
        goto switch_tasks;
    }
    /* dependent_sleeper() releases and reacquires the runqueue
     * lock, hence go into the idle loop if the rq went
     * empty meanwhile: */
    if (unlikely(!rq->nr_running))
        goto go_idle;
}
array = rq->active;
if (unlikely(!array->nr_active)) {
    /* Switch the active and expired arrays. */
    schedstat_inc(rq, sched_switch);
    rq->active = rq->expired;
    rq->expired = array;
    array = rq->active;
    rq->expired_timestamp = 0;
    rq->best_expired_prio = MAX_PRIO;
} else
    schedstat_inc(rq, sched_noswitch);

idx = sched_find_first_bit(array->bitmap);
queue = array->queue + idx;
next = list_entry(queue->next, task_t, run_list);

if (!rt_task(next) && next->activated > 0) {
    unsigned long long delta = now - next->timestamp;

    if (next->activated == 1)

```

```

        delta = delta * (ON_RUNQUEUE_WEIGHT * 128 / 100) / 128;

        array = next->array;
        dequeue_task(next, array);
        recalc_task_prio(next, next->timestamp + delta);
        enqueue_task(next, array);
    }
    next->activated = 0;
switch_tasks:
    if (next == rq->idle)
        schedstat_inc(rq, sched_goidle);
    prefetch(next);
    clear_tsk_need_resched(prev);
    rcu_qsctr_inc(task_cpu(prev));

    prev->sleep_avg -= run_time;
    if ((long)prev->sleep_avg <= 0)
        prev->sleep_avg = 0;
    prev->timestamp = prev->last_ran = now;

    sched_info_switch(prev, next);
    if (likely(prev != next)) {
        next->timestamp = now;
        rq->nr_switches++;
        rq->curr = next;
        ++*switch_count;

        prepare_arch_switch(rq, next);
        prev = context_switch(rq, prev, next);
        barrier();

        finish_task_switch(prev);
    } else
        spin_unlock_irq(&rq->lock);

    prev = current;
    if (unlikely(reacquire_kernel_lock(prev) < 0))
        goto need_resched_nonpreemptible;
    preempt_enable_no_resched();
    if (unlikely(test_thread_flag(TIF_NEED_RESCHED)))
        goto need_resched;
}

```

## **Chiamata diretta**

schedule() viene chiamata direttamente quando current deve essere bloccato perché la risorsa che deve utilizzare non è disponibile; la routine del kernel che lo deve bloccare agisce così:

- inserisce current nella wait queue appropriata;
- cambia il suo stato in TASK\_INTERRUPTIBLE o in TASK\_UNINTERRUPTIBLE;
- chiama schedule();
- controlla se la risorsa è disponibile; se non lo è torna al secondo punto;
- quando la risorsa torna disponibile, toglie current dalla wait queue.

La chiamata diretta è fatta anche dai driver di dispositivo che eseguono lunghe funzioni cicliche; ad ogni ciclo, il driver controlla TIF\_NEED\_RESCHED e, se necessario, chiama schedule() per rilasciare la CPU volontariamente.

### **Chiamata lazy**

Viene effettuata impostando TIF\_NEED\_RESCHED = 1 in current. Dato che il controllo di questo flag è sempre effettuato prima di riprendere l'esecuzione di un processo in modalità utente, schedule() viene sempre chiamata poco dopo.

Esempi tipici di questa modalità di chiamata sono:

- quando current ha esaurito il suo quanto; viene eseguita da scheduler\_tick();
- quando viene risvegliato un processo che ha priorità maggiore di current; viene eseguita da try\_to\_wake\_up();
- quando viene fatta la chiamata di sistema sched\_setscheduler().

### **Azioni compiute da schedule() prima di una commutazione di processo**

L'obiettivo di schedule() è di rimpiazzare current con un altro processo: perciò principalmente la funzione deve impostare la variabile locale next in modo che punti al descrittore del processo scelto per la sostituzione. Se nessun altro ha priorità superiore a current, next punterà a current e non ci sarà commutazione.

schedule() inizia definendo alcune variabili locali, disabilitando la preemption del kernel, ponendo prev = current e rilasciando il big kernel lock eventualmente in possesso di prev. Da notare che non viene cambiato il valore del campo lock\_depth: in questo modo, quando prev riprenderà l'esecuzione, automaticamente acquisirà o meno il big kernel lock in base al valore di depth.

Viene chiamata sched\_clock() per leggere il TSC e convertire il valore in nanosecondi; il timestamp ottenuto viene conservato nella variabile locale now. schedule() calcola poi la durata del quanto usata da prev e la memorizza in run\_time, applicando il solito arrotondamento a 1 secondo. Il valore run\_time viene usato per attribuire al processo il tempo di uso della CPU. Un processo con un lungo tempo medio di attesa viene favorito:

```
run_time /= (CURRENT_BONUS(prev) ? : 1);
```

Va ricordato che `CURRENT_BONUS` restituisce un valore tra 0 e 10 proporzionale al tempo medio di attesa.

La funzione disabilita gli interrupt locali e acquisisce lo spin lock della runqueue. Poi controlla se per caso `prev` è stato ucciso; per riconoscere questo caso controlla il flag `PF_DEAD`.

`schedule()` esamina poi lo stato di `prev`; se non è eseguibile e non ha subito `preemption` in modalità del kernel, deve essere tolto dalla runqueue. Comunque, se il processo ha segnali in attesa non bloccati e il suo stato è `TASK_INTERRUPTIBLE`, imposta lo stato a `TASK_RUNNING` e lo lascia nella runqueue: questa azione non è la stessa cosa che assegnare a `prev` il processore, ma gli fornisce solo un'occasione di essere eseguito. Se non è questo il caso, la funzione `deactivate_task()` rimuove il processo dalla runqueue.

Ora `schedule()` controlla in numero di processi eseguibili rimasti nella runqueue; se ne trova, chiama `dependent_sleeper()`. Nella maggior parte dei casi questa restituisce subito 0. Se però il kernel supporta la tecnologia `hyper-threading`, la funzione controlla se il processo che sta per essere eseguito ha priorità significativamente diversa da un processo fratello già in esecuzione su una CPU logica della stessa CPU fisica; in questo caso particolare `schedule()` rifiuta di scegliere il processo a bassa priorità e al suo posto esegue `swapper`.

Se non esistono processi eseguibili (etichetta `go_idle:`), la funzione chiama `idle_balance()` per spostare alcuni processi da altre runqueue; essa è simile a `load_balance()` già descritta in precedenza. Se anche questa funzione non riesce a fornire processi eseguibili alla coda locale, `schedule()` chiama `wake_sleeping_dependent()` per fare un `rescheduling` dei processi nelle CPU idle (che cioè eseguono il processo `swapper`): questo caso insolito avviene quando il kernel supporta la tecnologia `hyper-threading`, come spiegato in precedenza. Invece nei sistemi uniprocessore oppure quando tutti i tentativi di spostare processi da altre CPU sono falliti, la funzione sceglie `swapper` come `next` e prosegue.

Se `schedule()` ha trovato che nella runqueue ci sono processi eseguibili, deve controllare che almeno uno di essi sia attivo. Se non ne trova, scambia il contenuto delle liste `active` ed `expired`, in modo che tutti i processi `expired` divengano attivi e la lista `expired` ora vuota sia pronta ad accogliere nuovi processi.

A questo punto `schedule()` deve identificare un processo nella lista degli attivi di `prio_array_t`. Cerca prima di tutto nella maschera di bit il primo bit attivato, segno che la corrispondente lista di priorità non è vuota. La funzione usata, `sched_find_first_bit()` si basa sull'istruzione `assembly bsfl` che restituisce l'indice del bit meno significativo attivato in una `word` a 32 bit. L'indice di questo bit individua perciò la lista che contiene il processo da eseguire.

Ora `next` contiene il puntatore al descrittore del processo che sostituirà `prev`; `schedule()` controlla il campo `next->activated`, che codifica lo stato del processo nel momento in cui è stato risvegliato:

Valore	Descrizione
0	Il processo era nello stato <code>TASK_RUNNING</code> .
1	Il processo era nello stato <code>TASK_INTERRUPTIBLE</code> o <code>TASK_STOPPED</code> , ed è risvegliato da una routine di servizio, da una chiamata di sistema o da un thread del kernel.
2	Il processo era nello stato <code>TASK_INTERRUPTIBLE</code> o <code>TASK_STOPPED</code> , ed è risvegliato da un gestore di interrupt o da una funzione differibile.
-1	Il processo era nello stato <code>TASK_UNINTERRUPTIBLE</code> ed è risvegliato.

Se `next` è un processo convenzionale e `activated` è maggiore di 0, lo scheduler aggiunge al tempo medio di attesa di `next` i nanosecondi trascorsi da quando è stato inserito nella runqueue. In altre parole il valore

sleep\_time è incrementato per comprendere anche il tempo speso dal processo nella coda di attesa della CPU.

Da notare che lo scheduler distingue i processi con valori di activated 2 e 1: se è 2, aggiunge tutto il tempo di attesa nella runqueue; se è 1, ne aggiunge solo una parte. Questo perché è più facile che i processi interattivi vengano risvegliati da eventi asincroni che da eventi sincroni.

### Azioni compiute da schedule() per realizzare la commutazione di processo (switch\_tasks:)

Ora il kernel deve accedere alla struttura thread\_info di next, che si trova all'inizio del descrittore del processo. La macro prefetch(next) è un suggerimento alla CPU perché memorizzi nella cache il contenuto dei primi campi del descrittore di next; serve a migliorare le prestazioni perché i dati sono trasferiti parallelamente alle istruzioni seguenti, che non riguardano next.

Prima di rimpiazzare prev, lo scheduler deve fare un po' di lavoro amministrativo: clear\_tsk\_need\_resched() azzerava il flag TIF\_NEED\_RESCHED di prev, nel caso in cui schedule() sia stata chiamata in modalità lazy. Poi registra che la CPU sta per attraversare uno stato quiescente. Quindi decrementa il tempo medio di attesa di prev sommando il quanto di tempo della CPU, poi aggiorna il timestamp. E' possibile che prev e next coincidano, se non esistono altri processi. In questo caso la funzione evita la commutazione (ultimo else della funzione).

Se prev e next sono diversi, viene chiamata context\_switch() per creare lo spazio di indirizzamento di next. Il campo active\_mm del descrittore di processo punta al descrittore di memoria *usato* dal processo, mentre il campo mm punta al descrittore di memoria *posseduto* dal processo. Nei processi normali, i due campi puntano allo stesso indirizzo; invece un thread del kernel non possiede un proprio spazio di indirizzamento e il campo mm è impostato a NULL. La funzione context\_switch() si assicura che se next è un thread del kernel, usi lo spazio di indirizzamento già adoperato da prev:

```
if(!next->mm) {
    next->active_mm = prev->active_mm;
    atomic_inc(&prev->active_mm->mm_count);
    enter_lazy_tlb(prev->active_mm, next);
}
```

Fino a Linux 2.2 i thread del kernel avevano un proprio spazio di indirizzamento. Questa scelta non era ottimale, perché obbligava a cambiare le tabelle di paginazione in ogni caso, mentre un thread del kernel usa solo il quarto GB dello spazio di indirizzamento, la cui mappatura è unica per tutti i processi. Quel che è peggio, alterando il registro cr3 venivano invalidate tutte le entry di TLB, con una penalizzazione significativa delle prestazioni. Ora Linux è molto più efficiente, perché le tabelle di paginazione non sono toccate se next è un thread del kernel. Inoltre, come ulteriore ottimizzazione, schedule() imposta la modalità "lazy" per TLB.

Se invece next è un processo normale, context\_switch() rimpiazza lo spazio di indirizzamento di prev con quello di next:

```
if(next->mm)
    switch_mm(prev->active_mm, next->mm, next);
```

Se prev è un thread del kernel o un processo che sta terminando, context\_switch() salva il suo puntatore al descrittore di memoria nel campo prev\_mm della runqueue, poi resetta prev->active\_mm:

```
if(!prev_mm) {
```

```

    rq->prev_mm = prev->active_mm;
    prev->active_mm = NULL;
}

```

Ora `context_switch()` può chiamare finalmente `switch_to()` per eseguire la commutazione tra `prev` e `next`.

```

switch_to(prev, next, prev);
return prev;

```

### Azioni compiute da `schedule()` dopo la commutazione di processo

Le istruzioni successive alla chiamata di `switch_to()` non vengono eseguite da `next`, ma da `prev` quando, in un momento successivo, viene selezionato di nuovo dallo scheduler per l'esecuzione. A quel punto, `prev` non punta al processo fin qui descritto come `prev`, ma a quel (diverso) processo che sarà sostituito dal nostro attuale `prev` quando verrà eseguito di nuovo dalla CPU.

Subito dopo la chiamata di `context_switch()` si trova una barriera di ottimizzazione per il codice (la macro `barrier()`); poi viene chiamata la funzione `finish_task_switch()`:

```

mm = this_rq()->prev_mm;
this_rq()->prev_mm = NULL;
prev_task_flags = prev->flags;
spin_unlock(&this_rq()->lock);
if (mm)
    mmdrop(mm);
if(prev_task_flags & PF_DEAD)
    put_task_struct(prev);

```

Se `prev` è un thread del kernel, il campo `prev_mm` della runqueue contiene l'indirizzo del descrittore di memoria affidato a `prev`; `mmdrop()` decrementa il suo contatore d'uso; se diventa 0, la funzione lo rilascia assieme alle tabelle di paginazione e alle regioni di memoria virtuale.

La funzione `finish_task_function()` rilascia anche lo spin lock e abilita gli interrupt locali. Poi controlla se `prev` è uno zombie, e in caso affermativo chiama `put_task_struct()` per liberare il contatore di riferimento del descrittore di processo e cancellare tutti i riferimenti al processo stesso.

Le ultime istruzioni di `schedule()` fanno sì che essa riacquisti il big kernel lock se necessario, riabiliti la preemption del kernel e controlli se qualche altro processo ha il flag `TIF_NEED_RESCHED` attivato. In questo caso tutta la funzione viene eseguita di nuovo dall'inizio, altrimenti termina.

### Bilanciamento delle runqueue nei sistemi multiprocessore

Linux si conforma al modello SMP (Simmetric MultiProcessing); ciò significa che il kernel non ha nessuna preferenza per una CPU rispetto alle altre. Tuttavia i sistemi multiprocessore hanno diverse varianti e lo scheduler si comporta in maniera diversa e seconda dell'hardware. In particolare si possono distinguere tre tipi di sistemi:

*architettura multiprocessore classica*: fino a qualche tempo fa era l'architettura più comune. Le macchine hanno un insieme comune di chip RAM condivisi da tutte le CPU;

*hyper-threading*: il processore gestisce diversi flussi di esecuzione contemporaneamente; ha diverse copie dei registri interni e opera veloci commutazioni fra di esse. Questa tecnologia messa a punto da Intel, consente al processore di sfruttare i cicli macchina per eseguire un altro flusso di esecuzione mentre il primo è in attesa per un'operazione di memoria. Una CPU di questo tipo è vista da Linux come un insieme di diverse CPU logiche.

*NUMA (non uniform memory access)*: le CPU e i chip di memoria sono raggruppati in *nod*i locali (di solito un nodo comprende una CPU e vari chip di RAM). L'arbitro della memoria, cioè il circuito che serializza gli accessi alla memoria, è un collo di bottiglia per i classici sistemi multiprocessore. In un sistema NUMA, quando una CPU accede ad un chip locale, non esiste competizione, per cui l'accesso è rapido; d'altra parte accedere ad una memoria in un altro nodo è un processo lento.

Questi tipi di base sono spesso combinati fra loro. Ad esempio, una scheda madre che supporta due CPU con hyper-threading è vista da Linux come se avesse 4 CPU logiche. Come si è visto in precedenza, *schedule()* sceglie il processo da eseguire prendendolo dalla runqueue della CPU locale, per cui ognuna di esse esegue solo i processi della propria runqueue. D'altra parte ogni processo è contenuto in una sola runqueue. Perciò fino a che rimane eseguibile, è legato di solito ad una sola CPU.

Questa scelta di progetto è di solito positiva per le prestazioni del sistema, perché la cache hardware di ogni CPU contiene i dati propri del processo in esecuzione. A volte però questo legame può creare dei problemi: si consideri un gran numero di processi batch che impegnano a fondo la CPU: se la maggior parte di essi si trova in un'unica runqueue, una delle CPU avrà un carico notevole di lavoro, mentre le altre saranno quasi inattive.

Perciò il kernel periodicamente controlla se il carico delle varie runqueue è bilanciato e, se necessario, sposta alcuni processi da una all'altra. Tuttavia per ottenere le migliori performance, l'algoritmo di bilanciamento deve considerare anche la topologia delle CPU nel sistema. A partire dal kernel 2.6.7 Linux è dotato di un algoritmo basato sul concetto di *domini di scheduling*. Grazie ai domini, l'algoritmo può essere facilmente regolato in base alle caratteristiche dei vari sistemi, anche su quelli più recenti basati su processori multicore.

### Domini di scheduling

Un dominio di scheduling è un gruppo di CPU il cui carico di lavoro deve essere mantenuto bilanciato dal kernel. In generale i domini sono organizzati in modo gerarchico: il dominio di grado più alto, che generalmente raggruppa tutte le CPU del sistema, include domini figli, ognuno dei quali comprende un sottoinsieme di CPU. Grazie all'organizzazione gerarchica, il bilanciamento è svolto in maniera efficiente.

Ogni dominio è diviso, a sua volta, in uno o più gruppi, ognuno dei quali è un sottoinsieme delle CPU del dominio. Il bilanciamento viene fatto anche fra i gruppi del dominio di scheduling. In altre parole, un processo è spostato da una CPU all'altra solo se il carico totale di un gruppo è significativamente diverso da quello degli altri.

(a) – SMP

CPU 0
CPU 1

(b) – SMP con hyper-threading

CPU 0	CPU 1
CPU 2	CPU 3

(c) - NUMA

CPU 0	CPU 1	CPU 4	CPU 5
CPU 2	CPU 3	CPU 6	CPU 7

Caso a: c'è un solo dominio di scheduling per un sistema SMP a 2 CPU. Il dominio comprende 2 gruppi composti ciascuno da una sola CPU.

Caso b: gerarchia a 2 livelli per un sistema a 2 CPU con hyper-threading. Il dominio di grado più alto comprende le 4 CPU logiche, ed è composto da due gruppi. Ogni gruppo corrisponde ad un dominio figlio e comprende una CPU fisica. Il dominio di livello inferiore (o base) include due gruppi, uno per ogni CPU logica.

Caso c: gerarchia a 2 livelli per un sistema NUMA a 8 CPU con due nodi e 4 CPU per nodo. Il dominio di grado più alto è organizzato in due gruppi che corrispondono ai due nodi. Ogni dominio base comprende le CPU entro un singolo nodo e ha 4 gruppi, ognuno dei quali è formato da una singola CPU.

Ogni dominio è rappresentato da un descrittore `sched_domain`, ogni gruppo da un descrittore `sched_group`. Ogni `sched_domain` ha un campo `groups`, che punta al primo elemento di una lista di descrittori di gruppo. Ha anche un campo `parent` che punta al descrittore del dominio di livello superiore, se esiste.

I descrittori `sched_domain` di tutte le CPU fisiche del sistema sono conservati nella variabile per-CPU `phys_domains`. Se il kernel non supporta la tecnologia hyper-threading, questi domini sono al livello inferiore della gerarchia dei domini, e il campo `sd` della `runqueue` punta ad essi: sono perciò i domini di scheduling base. Se invece il kernel supporta l'hyper-threading, i domini di livello inferiore sono contenuti nella variabile per-CPU `cpu-domains`.

### **Funzione `rebalance_tick()`**

Per mantenere bilanciate le `runqueue`, la funzione `rebalance_tick()` viene chiamata da `scheduler_tick()` una volta ogni tick. Riceve come parametri l'indice `this_cpu` della CPU locale, l'indirizzo `this_rq` della `runqueue` locale e il flag `idle` che può avere i valori seguenti:

`SCHED_IDLE`: la CPU è inattiva, e `current` è il processo `swapper`;

`NOT_IDLE`: la CPU è attiva e `current` non è il processo `swapper`.

La funzione determina dapprima il numero di processi nella `runqueue` e aggiorna il valore del suo carico di lavoro medio: per far questo, accede ai campi `nr_running` e `cpu_load` del descrittore di `runqueue`. Poi inizia un ciclo su tutti i domini di scheduling nel percorso dal dominio di base (referenziato dal campo `sd` del descrittore) fino a quello di livello più alto. Ad ogni iterazione decide se è il momento di eseguire `load_balance()`. Il valore di `idle` e quello di alcuni parametri del descrittore `sched_domain` condizionano la frequenza della chiamata di `load_balance()`. Se `idle` vale `SCHED_IDLE`, `load_balance()` viene chiamata spesso (ogni uno o due tick per domini di scheduling corrispondenti a CPU logiche e fisiche). Se `idle` vale `NOT_IDLE`, `load_balance()` viene chiamata più raramente, una volta ogni 10 millisecondi per domini corrispondenti a CPU logiche, e ogni 100 millisecondi per domini corrispondenti a CPU fisiche.

### **Funzione `load_balance()`**

Controlla se un dominio è sbilanciato in modo significativo, o meglio se lo sbilanciamento può essere ridotto spostando alcuni processi dal gruppo più impegnato alla CPU locale. Riceve quattro parametri:

`this_cpu`: indice della CPU locale;

this\_rq: indirizzo del descrittore della runqueue locale;

sd: punta al descrittore del dominio di scheduling da controllare;

idle: vale SCHED\_IDLE o NOT\_IDLE.

La funzione esegue le seguenti operazioni:

1 – Acquisisce lo spin lock this\_rq->lock.

2 – Chiama la funzione find\_busiest\_group() per analizzare il carico del gruppo entro il dominio di scheduling. Restituisce il descrittore sched\_group del gruppo più impegnato, purché non sia quello che include la CPU locale; in quest'ultimo caso, restituisce anche il numero di processi da spostare nella runqueue locale per bilanciare i carichi. Se il gruppo più impegnato non comprende la CPU locale oppure i gruppi sono sostanzialmente bilanciati, restituisce NULL. La procedura non è banale perché la funzione tenta di filtrare le fluttuazioni statistiche del carico di lavoro.

3 – Se find\_busiest\_group() non trova un gruppo che non include la CPU locale significativamente più impegnato degli altri entro il dominio, la funzione rilascia lo spin lock, aggiusta i parametri nel descrittore del dominio di scheduling, per ritardare la prossima chiamata di load\_balance() e termina.

4 – Chiama find\_busiest\_queue() per trovare la CPU più impegnata nel gruppo determinato al punto 2; restituisce l'indirizzo del descrittore busiest della runqueue corrispondente.

5 – Acquisisce un secondo spin lock busiest->lock. Per prevenire deadlock bisogna agire con cautela; prima viene rilasciato this\_rq->lock, poi i due lock vengono acquisiti incrementando gli indici della CPU.

6 – Chiama move\_tasks() per tentare di spostare alcuni processi da busiest a this\_rq.

7 – Se la funzione non ha successo nel trasferimento, il dominio è ancora sbilanciato. Imposta a 1 il flag busiest->active\_balance e risveglia il thread del kernel *migration* il cui descrittore è memorizzato in busiest->migration\_thread. Questo thread percorre la catena dei domini di scheduling, dal dominio base della coda busiest fino al dominio di livello più alto, in cerca di una CPU inattiva. Se la trova, chiama move\_tasks() per spostare un processo nella runqueue inattiva.

8 – Rilascia gli spin lock.

9 – Termina.

### **Funzione move\_tasks()**

Sposta processi da una runqueue di origine alla coda locale. Riceve 6 parametri:

this\_rq e this\_cpu (runqueue e cpu locali), busiest (descrittore della runqueue origine), max\_nr\_move (numero massimo di processi da spostare), sd (indirizzo del descrittore del dominio di scheduling nel quale viene condotta l'operazione di bilanciamento) e il flag idle (che può assumere anche il valore NEWLY\_IDLE quando la funzione viene chiamata direttamente da idle\_balance()).

La funzione analizza dapprima i processi expired nella runqueue busiest, a partire da quelli a priorità più elevata. Quando tutti questi sono stati analizzati, passa a quelli attivi. Per ogni candidato, chiama can\_migrate\_task() che restituisce 1 se sono soddisfatte tutte le condizioni seguenti:

- il processo non è in esecuzione su CPU remote;

- la CPU locale è inclusa nella maschera di bit `cpus_allowed` del descrittore di processo;
- è verificata almeno una delle seguenti condizioni:
  - la CPU locale è inattiva; se il kernel supporta l'hyper-threading, tutte le CPU logiche devono essere inattive;
  - il kernel ha problemi di bilanciamento, perché ripetuti tentativi di spostamento di processi hanno fallito;
  - il processo da spostare non è *cache hot* (non è stato eseguito di recente su una CPU remota, per cui si può ritenere che la cache non contenga dati a lui correlati).

Se `can_migrate_task()` restituisce 1, `move_tasks()` chiama `pull_task()` per spostare il processo candidato nella runqueue locale. Essa esegue `dequeue_task()` per rimuovere il processo dalla coda, poi esegue `enqueue_task()` per inserirlo nella coda locale e infine, se il processo ha priorità superiore a `current`, chiama `resched_task()` per applicare la preemption al processo corrente della CPU locale.

### Chiamate di sistema correlate allo scheduling

Sono state introdotte diverse chiamate di sistema per consentire ai processi di cambiare priorità e politiche di scheduling. In generale gli utenti possono sempre diminuire il valore della priorità dei loro processi; devono però possedere i privilegi del superuser se vogliono aumentarla o variare la priorità dei processi degli altri utenti.

#### La chiamata di sistema `nice()`

Consente di variare la priorità di base dei processi. Il valore intero contenuto nel parametro `increment` è usato per modificare il campo `nice` del descrittore di processo. Il comando Unix `nice`, che consente agli utenti di avviare programmi con priorità modificata, è basato su questa chiamata di sistema.

La routine di servizio `sys_nice()` gestisce la chiamata di sistema. Anche se `increment` può avere qualunque valore, valori assoluti superiori vengono portati a 40. Tradizionalmente i valori negativi corrispondono a richieste di aumento di priorità e richiedono i privilegi di superuser. In questo caso la funzione chiama `capable()` per verificare se il processo possiede `CAP_SYS_NICE`. In più chiama `security_task_setnice()`. Se risulta che l'utente possiede i privilegi richiesti, `sys_nice()` converte `current->static_prio` nell'intervallo dei valori di `nice`, aggiunge `increment` e chiama `set_user_nice()`. Questa funzione acquisisce il lock della runqueue locale, aggiorna la priorità statica di `current`, chiama `resched_task()` e rilascia il lock. La chiamata di sistema `nice()` viene mantenuta per compatibilità; è stata sostituita da `setpriority()`.

#### `getpriority()` e `setpriority()`

`nice()` agisce solo sul processo che la chiama: altre due chiamate di sistema, `getpriority()` e `setpriority()` agiscono sulla priorità di base di tutti i processi di un dato gruppo. La prima restituisce 20 – `nice` del processo a priorità più elevata del gruppo; la seconda imposta la priorità di base di tutti i processi di un dato gruppo. Il kernel implementa le due chiamate di sistema attraverso `sys_getpriority()` e `sys_setpriority()`. Entrambe operano sullo stesso insieme di parametri:

which: il valore che identifica il gruppo di processi, che può essere:

- `PRIO_PROCESS`: seleziona i processi in base al loro ID (campo pid);
- `PRIO_PGRP`: seleziona i processi in base all'ID di gruppo (campo pgrp);
- `PRIO_USER`: seleziona i processi in base all'ID utente (campo uid);

who: il valore di pid, pgrp o uid, a seconda del valore di which; se è 0, si riferisce al processo corrente;

niceval: la nuova priorità di base (serve solo a `sys_setpriority()`). Va da -20 (massima) a +19 (minima).

Solo i processi con capacità `CAP_SYS_NICE` possono incrementare la loro priorità o modificare quella degli altri. Le chiamate di sistema in generale restituiscono un valore negativo solo se si verifica un errore; per questa ragione `getpriority()` non restituisce un valore tra -20 e +19, ma un valore positivo tra 1 e 40.

### **`sched_getaffinity()` e `sched_setaffinity()`**

Queste due chiamate di sistema restituiscono e impostano rispettivamente la maschera di affinità alla CPU di un processo, cioè la maschera di bit delle CPU che possono eseguire il processo. Essa è conservata nel campo `cpus_allowed` del descrittore di processo.

La routine di servizio della chiamata di sistema `sys_sched_getaffinity()` controlla i descrittori di processo chiamando `find_task_by_pid()`, poi restituisce il valore di `cpus_allowed` in AND con la mappa di bit delle CPU disponibili.

La `sys_sched_setaffinity()` è un po' più complessa. Prima di cercare il descrittore di processo e aggiornare il campo `cpus_allowed`, deve controllare se il processo è inserito nella runqueue di una CPU che non è più presente nella nuova maschera di affinità. In questo caso, il processo va trasferito da una runqueue a un'altra. Per evitare deadlock e race conditions, questa operazione viene compiuta dal thread di migrazione del kernel (ne esiste uno per ogni CPU). Quando deve avvenire uno spostamento da `rq1` a `rq2`, la chiamata di sistema risveglia il thread di migrazione di `rq1` che sposta il processo da `rq1` a `rq2`.

### **Chiamate di sistema correlate ai processi real-time**

C'è un gruppo di chiamate di sistema che permette ai processi di cambiare la politica di scheduling e, in particolare, di diventare real-time. Come al solito un processo deve avere `CAP_SYS_NICE` per poter operare questi cambiamenti.

### **`sched_getscheduler()` e `sched_setscheduler()`**

La prima delle due chiamate di sistema interroga la politica di scheduling del processo pid. Se `pid = 0`, viene restituita la politica di scheduling del processo che ha eseguito la chiamata. La funzione restituisce uno dei valori: `SCHED_FIFO`, `SCHED_RR`, `SCHED_NORMAL` (anche detta `SCHED_OTHER`). La funzione `sys_sched_getscheduler()` chiama `find_process_by_pid()` che trova il descrittore corrispondente al pid e restituisce il valore del campo `policy`.

La seconda chiamata di sistema imposta la politica di scheduling e i parametri associati per il processo pid. Se `pid = 0`, ci si riferisce al processo che ha eseguito la chiamata. La `sys_sched_setscheduler()` chiama

semplicemente `do_sched_setscheduler()` che controlla la validità dei parametri passati (politica di scheduling policy e nuova priorità `param->sched_priority`); controlla anche se il processo ha la necessaria `CAP_SYS_NICE` o i diritti di superuser. Se tutto è ok, rimuove il processo dalla sua runqueue (se eseguibile); aggiorna le priorità statica, dinamica e real-time; lo reinserisce nella coda e infine chiama `resched_task()`.

### **`sched_getparam()` e `sched_setparam()`**

La prima chiamata di sistema restituisce i parametri di scheduling del processo pid; se `pid = 0`, si riferisce a current. La routine di servizio `sys_sched_getparam()`, trova il descrittore di processo associato al pid, memorizza il campo `rt_priority` in una variabile locale di tipo `sched_param` e chiama `copy_to_user()` per copiarlo nello spazio di indirizzamento del processo all'indirizzo specificato dal parametro `param`.

La seconda chiamata di sistema è simile a `sched_setscheduler()`. La differenza è che non permette al processo che ha fatto la chiamata di definire la politica di scheduling. La `sys_sched_setparam()` chiama `do_sched_setscheduler()`.

### **`sched_yield()`**

Questa chiamata di sistema permette ad un processo di rilasciare volontariamente la CPU senza essere sospeso, rimanendo nello stato `TASK_RUNNING`. Lo scheduler lo inserisce nel gruppo degli expired (se è un processo convenzionale) o alla fine della runqueue (se è real-time). Poi viene chiamata `schedule()`. In questo modo altri processi con la stessa priorità dinamica possono essere eseguiti. E' usata principalmente da processi `SCHED_FIFO`.

### **`sched_get_priority_min()` e `sched_get_priority_max()`**

Restituiscono rispettivamente i valori minimo e massimo della priorità statica real-time che possono essere usati dalla politica di scheduling identificata dal parametro `policy`. La prima restituisce 1 se current è real-time, se no 0; la seconda restituisce 99 se current è real-time, se no 0.

### **`sched_rr_get_interval()`**

Questa chiamata di sistema scrive in una struttura contenuta nello spazio di indirizzamento in modalità utente il quanto di tempo per un processo real time in Round Robin identificata da pid. Se è zero, la funzione scrive il quanto del processo corrente.

La routine di servizio corrispondente, `sys_sched_rr_get-interval()` chiama `find_process_by_pid()` per ottenere il descrittore associato a pid. Poi converte il quanto base in secondi e nanosecondi e lo copia nella struttura di destinazione. Per convenzione, il quanto di un processo real-time FIFO è uguale a zero.