Giuseppe Pelagatti

Programmazione e Struttura del sistema operativo Linux

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

Parte N: Il Nucleo del Sistema Operativo

cap. N5 – Lo Scheduler

N.5 Lo Scheduler

1. Caratteristiche generali dello Scheduler

Il comportamento del SO è fortemente caratterizzato dalle politiche adottate per decidere quali processi eseguire e per quanto tempo eseguirli (***politiche di scheduling***).
Il componente del SO che realizza le politiche di scheduling è detto Scheduler. Il comportamento dello Scheduler è orientato a garantire le seguenti condizioni:

- che i processi più importanti vengano eseguiti prima dei processi meno importanti
- che i processi di pari importanza vengano eseguiti in maniera equa; in particolare ciò significa che nessun processo dovrebbe attendere il proprio turno di esecuzione per un tempo molto superiore agli altri.

Lo Scheduler è un componente critico nel funzionamento di un Sistema Operativo e molta ricerca viene svolta per migliorarne le prestazioni.

2. Concetti introduttivi

Dati n processi che devono essere eseguiti, la politica **Round Robin** consiste nell'assegnare ad ogni processo uno stesso ***quanto*** di tempo o ***timeslice*** in ordine circolare. Tutti i processi vengono quindi schedulati per lo stesso tempo a turno. Questa politica è equa e garantisce che nessun processo rimanga bloccato per sempre (starvation).
Lo scheduler interviene in certi momenti per determinare quale processo mandare in esecuzione. La scelta del processo da mandare in esecuzione avviene nell'ambito di tutti i processi in stato di PRONTO esistenti nel sistema.
Dati 2 processi P e Q, lo scheduler deve calcolare una grandezza che determini quale dei 2 scegliere; dato che il termine priorità è usato con diversi significati particolari, noi useremo per questa grandezza il termine ***diritto di esecuzione***. Lo scheduler sceglie quindi nell'ambito dei processi pronti quello che possiede il maggiore diritto di esecuzione.
I momenti in cui è significativo eseguire questa scelta sono:

- ogni volta che un processo si autosospende, per scegliere il prossimo processo da eseguire
- ogni volta che un processo viene risvegliato, perchè si estende l'insieme dei processi pronti; in questo caso se il nuovo processo diventato pronto ha un diritto di esecuzione superiore a quello corrente, il diritto di esecuzione diventa “diritto di preemption”, cioè causa la sospensione del processo corrente (si ricorda però cha la preemption è dilazionata fino al prossimo momento di ritorno a modo U)
- ogni volta che il processo in esecuzione è gestito con politica Round Robin e il suo timeslice è esaurito

3. Requisiti dei processi

Non tutti i processi hanno gli stessi requisiti relativamente allo scheduling. Ad un primo livello possiamo distinguere i processi nelle seguenti categorie:

1. Processi **Real-time (in senso stretto).** Questi processi devono soddisfare dei vincoli di tempo estremamente stringenti e devono quindi essere schedulati con estrema rapidità, ***garantendo in ogni caso di non superare un preciso vincolo di ritardo massimo***; un esempio di processo di questo tipo è il controllo di un aereo o di un impianto.
2. Processi **semi-Real-time** (**soft Real time**). Questi processi, pur richiedendo una relativa rapidità di risposta, non richiedono la garanzia assoluta di non superare un certo ritardo massimo. Un esempio di questo tipo è la scrittura di un CD o la esecuzione di un file Audio, che deve essere svolta producendo i dati con una certa continuità; se talvolta questa continuità non viene garantita si verificano dei disturbi, evento relativamente accettabile (almeno in confronto a una caduta di aereo o a un’esplosione di impianto). Tuttavia, il processo che scrive un CD deve avere un certo livello di precedenza che gli permetta normalmente di produrre i dati in tempo quando servono.

3. Processi **Normali**. Sono tutti gli altri processi. Questi processi possono avere diversi comportamenti:
    - **processi I/O bound** sono i processi che si sospendono spesso perchè hanno bisogno di I/O (ad esempio, un Text editor)
    - **processi CPU bound** sono invece i processi che tendono ad usare molto la CPU, perchè si autosospendono raramente (ad esempio, un Compilatore)

4. Classi di Scheduling

Per supportare le diverse categorie di processi lo Scheduler realizza diverse politiche di scheduling; ogni politica di scheduling è realizzata da una **Scheduler Class**. Nel descrittore di un processo il campo

```
constant struct sched_class * sched_class
```

contiene un puntatore alla struttura della scheduler class deputata a gestirlo.

Lo Scheduler è l'unico gestore delle `runqueue` – per questo motivo le altre funzioni devono chiedere allo scheduler di eseguire operazioni sulla `runqueue`. Questa scelta permette di organizzare le `runqueue` in maniera adatta alle diverse classi di scheduling senza dover modificare il resto del sistema.

Quando viene invocata una funzione dello scheduler (cfr. capitolo precedente, "interazione con lo scheduler") tale funzione svolge poche funzioni preliminari e poi invoca la corrispondente funzione della scheduler class alla quale il task appartiene. Ad esempio, per i processi normali attualmente la scheduler class ha la struttura (semplificata) di figura 1, definita nel file `sched_fair.c` (che contiene l'implementazione del Completely Fair Scheduler – CFS – descritto più avanti).

```
1106 static const struct sched_class fair_sched_class = {
1107         .next                   = &idle_sched_class,
1108         .enqueue_task           = enqueue_task_fair,
1109         .dequeue_task           = dequeue_task_fair,
1112         .check_preempt_curr     = check_preempt_wakeup,
1114         .pick_next_task         = pick_next_task_fair,
1115         .put_prev_task          = put_prev_task_fair,
1122         .set_curr_task          = set_curr_task_fair,
1123         .task_tick              = task_tick_fair,
1124         .task_new               = task_new_fair,
1125 };
```

Figura 1

Se quindi una funzione del nucleo invoca `enqueue_task`, all'interno di questa funzione verrà invocata la funzione corrispondente della sched_class del processo in esecuzione p nel modo seguente:

```
p->sched_class->enqueue_task
```

e, se, ad esempio, la `sched_class` del processo è `fair_sched_class`, la funzione effettivamente eseguita sarà `enqueue_task_fair`. In questo modo è possibile aggiungere nuove classi di scheduling al sistema senza doverlo modificare eccessivamente.
Ogni classe può a sua volta implementare più di una politica.

Attualmente le 3 classi più importanti supportate

1. `SCHED_FIFO` (First IN First Out)
2. `SCHED_RR` (Round Robin)
3. `SCHED_NORMAL`

L'ordine di queste 3 classi è un ordine anche dei diritti di esecuzione; un processo della classe `SCHED_FIFO` ha un diritto di esecuzione sempre superiore a quello di un processo di una delle due altre classi e un processo della classe `SCHED_RR` ha un diritto di esecuzione sempre superiore a un processo della classe `SCHED_NORMAL`.

La funzione `schedule( )` invoca la funzione `pick_next_task( )`, che a sua volta invoca le funzioni `pick_next_task( )` specifiche di ogni singola classe in ordine di importanza delle classi per selezionare un nuovo task; `schedule( )` poi procede al context switch utilizzando le funzioni della classe appropriata per togliere dall'esecuzione il task corrente (prev) e mettere in esecuzione il nuovo (next).

```
schedule( ){
       …
       struct tsk_struct * prev;
       prev = CURR;
       if (CURR->stato = ATTESA) //elimina CURR dalla RQ

       //invoca la funzione di scelta del prossimo task
              next = pick_next_task(rq, prev);
       //se next è diverso da prev, esegui il context switch
              if (next != prev) context_switch(prev, next);
       TIF_NEED_RESCHED = 0;
}

pick_next_task(rq, prev){
       for(ciascuna classe di scheduling, in ordine di importanza decrescente)
       {
           //invoca funzione di scelta del prossimo task per la classe in esame
             next = class->pick_next_task(rq, prev);
             if (next != NULL) return next; //appena trovi un task, ritorna
       }
       //pick_next_task restituisce sempre un puntatore valido:
         //a un task PRONTO con diritto di esecuzione massimo, se esiste
         //a prev, se non ci sono altri task pronti e il suo stato non è ATTESA
         //a IDLE, se nessuno dei 2 casi precedenti è praticabile
}
```

5. Scheduling dei Processi soft Realtime

Le classi `SCHED_FIFO` e `SCHED_RR` sono utilizzate per supportare i processi soft RT (Linux non supporta i processi RT in senso stretto, perchè non è in grado di garantire il non superamento di un ritardo massimo). In queste due classi il concetto fondamentale è quello di priorità statica.
A ogni processo di queste classi viene attribuita una priorità detta statica, perchè è attribuita all'inizio e non varia mai. I valori delle priorità statiche appartengono all'intervallo da 1 a 99 (99 è la più alta).

La priorità statica è memorizzata in `task_struct` nel campo `static_prio`.

Scheduling SCHED_FIFO
Un task di questa categoria viene eseguito senza alcun limite di tempo. Se esistono diversi task di questa categoria, sono schedulati in base alla priorità statica – un task con maggiore priorità ha un diritto di esecuzione maggiore di uno a priorità più bassa e può quindi causarne la preemption.

Scheduling SCHED_RR
I task in questa categoria sono simile ai precedenti, però nell'ambito della stessa fascia di priorità sono schedulati con una politica Round Robin; pertanto se esistono diversi task in questa categoria allo stesso livello di priorità, ognuno di loro viene eseguito per un timeslice.

6. Scheduling dei processi normali (SCHED_NORMAL)

6.1. Aspetti generali

I processi normali vengono presi in considerazione dallo Scheduler solo se non ci sono processi delle classi FIFO e RR in stato di PRONTO per l'esecuzione. L'attuale scheduler dei processi normali è chiamato ambiziosamente **Completely Fair Scheduler** (**CFS**), perchè ambisce a simulare il seguente meccanismo ideale: dati N task, dedicare una CPU di potenza 1/N ad ogni task. Naturalmente il meccanismo ideale non è realizzabile con un numero di CPU inferiore a N, quindi in pratica la CPU (che qui supporremo unica per semplificare il discorso) deve essere assegnata per un **quanto** di tempo ad ogni task.

I problemi fondamentali che lo scheduler deve risolvere nel far questo sono:

1. *determinare ragionevolmente la durata dei quanti* (quanti troppo lunghi abbassano la responsività del sistema, quanti troppo corti causano troppo sovraccarico per i numerosi context switch)
2. permettere di *assegnare dei pesi ai processi*, in modo che ai processi più importanti sia assegnato più tempo che a quelli meno importanti
3. permettere ai processi che stanno a lungo in stato di ATTESA di tornare rapidamente in esecuzione quando vengono risvegliati

L'ultimo requisito è legato al seguente problema generato dallo scheduling basato sulla politica Round Robin: in assenza di meccanismi correttivi i processi I/O bound tenderebbero a funzionare molto male; ad esempio, si considerino un Editor di testo e un Compilatore. Ambedue sono processi normali. Tuttavia, dato che l'Editor è molto interattivo (I/O bound) tenderebbe ad eseguire per poco tempo e poi a sospendersi, mentre il compilatore (CPU bound) ogni volta che va in esecuzione tenderebbe ad eseguire per l'intero tempo a sua disposizione

6.2 Meccanismo fondamentale

Per semplificare la presentazione iniziale del meccanismo del CFS assumiamo che tutti i processi abbiano lo stesso peso, precisamente il peso assegnato per default dal sistema. Tale peso è definito nel sistema dalla costante `NICE_0_LOAD`, ma per semplicità lo indicheremo qui come **`L0`**. Inoltre per il momento supponiamo che i task non si autosospendano mai (non eseguano wait)
Quindi per il momento valgono le ipotesi seguente:

t.LOAD = L0 per tutti i task t presenti nella runqueue
i task non eseguono mai wait

Sotto queste ipotesi il meccanismo base ha una logica molto semplice. Sia **NRT** il numero di task presenti nella runqueue a un certo istante:

- viene determinato un periodo **PER** di schedulazione durante il quale tutti i task della runqueue possono essere eseguiti, se non si autosospendono
- ad ogni task viene assegnato un quanto di tempo **Q** = PER/NRT

I task vengono mantenuti in una coda ordinata **RB** e il funzionamento dello scheduler può essere schematizzato nel modo seguente:

1. viene estratto da RB (e reso CURR) il primo task della coda
2. CURR viene eseguito fino a quando scade il suo quanto di tempo Q
3. CURR viene sospeso e reinserito in RB in fondo alla coda
4. torna al passo 1

In pratica i task sono eseguiti a turno per esattamente PER/NRT ms; si osservi che il periodo di schedulazione deve essere considerato come una finestra che scorre nel tempo; non c'è una suddivisione rigida del tempo in periodi, ma in ogni istante si può fare riferimento all'inizio di un nuovo periodo di schedulazione. In altre parole, in un momento qualsiasi si può asserire che entro i prossimi PER ms tutti i task verranno eseguiti.

ATTENZIONE: da un punto di vista sintattico nelle formule si è spesso utilizzata la notazione p.c (selezione di un campo c di una struttura p) anche quando sarebbe necessario utilizzare p->v (perché p è un puntatore alla struttura che contiene c); questo semplifica la lettura, ma è scorretto se interpretato rigorosamente come codice C.

Controllo del periodo di schedulazione PER

Il periodo di schedulazione varia dinamicamente con l'aumento o la diminuzione del numero di task presenti nella runqueue, non può quindi essere fissato rigidamente, ma è necessario cercare una mediazione tra i seguenti aspetti:

- un periodo troppo lungo può ritardare eccessivamente l'esecuzione di un processo
- un periodo troppo corto può produrre quanti eccessivamente corti al crescere di NRT, portando a commutazioni di contesto troppo frequenti

L'impostazione attualmente adottata da Linux consiste nel basare il calcolo di PER su due parametri di controllo (`SYSCTL` parameter) modificabili dall'amministratore del sistema:

- **LT** (latenza) – default: 6ms
- **GR** (granularità) – default: 0,75ms

Il periodo è calcolato con la seguente formula:

PER = MAX(LT, NRT * GR)

In questo modo il quanto di un processo è LT/NRT fino a quando NRT*GR è minore della latenza; quando NRT*GR supera il valore LT (per i valori di default questo accade con NRT>8) il periodo viene allungato in modo da evitare che il quanto scenda sotto il valore di granularità GR.

Adattamento del quanto ai pesi dei task

La formula Q = PER/NRT vista sopra non tiene conto del peso dei task; in realtà il valore del quanto di tempo t.Q assegnato a un task è proporzionale al suo peso rispetto al peso di tutti i task. Il calcolo del quanto utilizza le seguenti due variabili:

- RQL (rqload) è la somma dei pesi di tutti i task presenti sulla runqueue
- LC (load_coeff) il rapporto tra il peso di un task e RQL;

LC = t.LOAD/RQL , quindi anche t.LOAD = LC*RQL

Il quanto assegnato a un task t è calcolato nel modo seguente:

t.Q = PER * t.LC

Si osservi che nell'ipotesi che tutti i task abbiano peso uguale (L0) queste formule si riducono alla Q = PER/NRT indicata all'inizio; infatti

Q = PER * LC =PER * (L0/(NRT*L0)) = LT/NRT

Il Virtual Time

Per mantenere l'ordinamento dei task nella coda RB il CFS usa il concetto di di **Virtual Runtime** (**VRT**). Il VRT *è una misura del tempo di esecuzione consumato da un processo, basato sulla modificazione del tempo reale in base a opportuni coefficienti*, in modo che la decisione su quale sia il prossimo task da eseguire possa basarsi semplicemente sulla scelta del processo con VRT minimo; dato che RB è ordinato in base ai VRT dei task, il prossimo task da mettere in esecuzione sarà il primo di RB e viene indicato con **LFT** (leftmost) (si ricorda che il task in esecuzione non è contenuto in RB ma puntato dalla variabile CURR). Quando un task termina l'esecuzione viene reinserito nella coda RB nella posizione che gli compete in base al nuovo valore del VRT che possiede.

Consideriamo un task che ha eseguito per **DELTA** nsec e che abbandona l'esecuzione; lo scheduler incrementa in quel momento il suo tempo di esecuzione **SUM** e il suo VRT; mentre il tempo di esecuzione viene semplicemente incrementato di DELTA:

SUM = SUM + DELTA

l'incremento del VRT viene corretto con un coefficiente **VRTC** (`vrt_coeff`) nel modo seguente:

VRTC = L0/peso del task

VRT = VRT + DELTA * VRTC

Si noti che VRTC è inversamente proporzionale a LC; infatti

L0/peso del task = L0 / (LC*RQL) = (L0/RQL) * 1/LC; quindi, dato che L0/RQL è una costante (se non cambia il numero di task)

L'effetto del coefficiente VRTC è di far crescere il VRT dei processi più pesanti più lentamente del VRT dei processi più leggeri. In condizioni di funzionamento stabile (cioè un numero di processi costante, con tutti i processi che consumano tutto il loro quanto) *la crescita del VRT di tutti i processi in un periodo è la stessa*; infatti l'effetto di un quanto più lungo è esattamente compensato da una crescita più lenta del VRT, come si può vedere dalla seguente calcolo dell'incremento di VRT per un processo che ha appena eseguito per tutto il suo quanto:

deltaVRT = DELTA *VRTC = Q*VRTC = (PER * LC) * ((L0/RQL) * 1/LC) = PER*L0/RQL

Come si vede, deltaVRT non dipende dal peso del processo.

Infine, per motivi che emergeranno più avanti, lo scheduler mantiene nella runqueue una variabile **VMIN** (`vrtmin`) che rappresenta il VRT minimo tra tutti i task presenti nella runqueue; questa variabile è aggiornata continuamente in base alla regola:

VMIN = MIN(CURR.VRT, LFT.VRT) //provvisoria

dove CURR.VRT è il VRT del task in esecuzione che viene anch'esso aggiornato continuamente. ***Questa formula dovrà essere modificata*** per rispondere ad alcuni problemi discussi più avanti.

Possiamo adesso interpretare lo pseudocodice seguente, relativo alla funzione `tick` invocata periodicamente in base agli interrupt del clock, dove:

- NOW è l'istante corrente
- START è l'istante in cui un task viene messo in esecuzione
- PREV è il valore di SUM al momento in cui un task viene messo in esecuzione

```
tick() {
      //aggiornamento dei parametri di CURR:
            DELTA  = NOW – CURR->START;
            CURR->SUM = CURR->SUM + DELTA;
            CURR->VRT = CURR->VRT + DELTA * VRTC;
            CURR->START = NOW;
      //aggiornamento di VMIN della RQ
            VMIN = MIN(CURR->VRT,LFT->VRT)); //questa formula verrà
            corretta
      //controllo se è scaduto il quanto di tempo
            if ((CURR->SUM – CURR->PREV) >  Q) resched( );
}
```

Esempio 1

In Tabella 1 è simulata l'evoluzione del sistema partendo da una situazione con 3 task di peso uguale a **`L0`** (il peso in questi esempi viene indicato tra perantesi dopo il nome del task) fino al termine di un periodo di schedulazione (PER = 6ms)

Per ogni istante di tempo è mostrata l'azione svolta a partire dall'istante precedente e le variazioni dei parametri globali della runqueue e dei parametri di ogni singolo task. Dove le celle sono vuote significa che è ancora valido il valore precedente.

La condizione iniziale è caratterizzata dai seguenti aspetti:

- esistono 3 task con tutti i coefficienti LC, VRTC identici
- anche i quanti di tempo sono quindi identici (2 ms = PER/NRT)
- i 3 task sono in esecuzione da tempi diversi (diversi SUM) ma i loro VRT sono abbastanza simili

- il task t1 è quello corrente; siamo all'inizio della sua esecuzione, quindi DELTA = 0;
- VMIN è il valore di t1.VRT, cioè di CURR

	RUNQUEUE		**TASKS**			
T				t1 (L0)	t2 (L0)	t3 (L0)
init	NRT	3	LC	1/3	1/3	1/3
	PER	6	VRTC	1	1	1
0	RQL/L0	3	Q	2	2	2
	CURR	t1	SUM	50	30	10
	RB	t2,t3	DELTA	0	0	0
	VMIN	1000	VRT	1000	1000,5	1001
exe	NRT		LC	1/3		
	PER		VRTC	1		
2	RQL/L0		Q	2		
	CURR		SUM	52		
	RB		DELTA	2		
	VMIN	1000,5	VRT	1002		
csw	NRT		LC			
	PER		VRTC			
	RQL/L0		Q			
	CURR	t2	SUM			
	RB	t3, t1	DELTA			
	VMIN	1000,5	VRT			
exe	NRT		LC		1/3	
4	PER		VRTC		1	
	RQL/L0		Q		2	
	CURR		SUM		32	
	RB		DELTA		2	
	VMIN	1001	VRT		1002,5	
csw	NRT		LC			
	PER		VRTC			
	RQL/L0		Q			
	CURR	t3	SUM			
	RB	t1,t2	DELTA			
	VMIN	1001	VRT			
exe	NRT		LC			1/3
6	PER		VRTC			1
	RQL/L0		Q			2
	CURR		SUM			12
	RB		DELTA			2
	VMIN	1002	VRT			1003
csw	NRT		LC			
	PER		VRTC			
	RQL/L0		Q			
	CURR	t1	SUM			
	RB	t2, t3	DELTA			
	VMIN	1002	VRT			

Tabella 1

La colonna di sinistra indica le azioni svolte e lo scorrere del tempo. I tipi di azione utilizzati sono:

- exe: significa che il task CURR ha eseguito dall'istante precedente fino al tempo indicato; ad esempio exe 2 significa che il task ha eseguito per 2 ms a partire dall'istante iniziale; più avanti exe 4 significa che il task ha eseguito per 2 ms dall'istante 2 all'istante 4
- csw: significa context switch; si suppone che il tempo necessario per l'esecuzione del context switch sia trascurabile in questo contesto, quindi il tempo rimane costante

Per capire i valori di VMIN riportati bisogna considerare che all'inizio dell'esecuzione di un task tipicamente il VMIN è uguale al VRT dello stesso task (CURR); tale valore cresce durante l'esecuzione e a un certo punto raggiunge il valore di LFT; da questo momento VMIN rimane costante e uguale a LFT.VRT.

In Tabella 2 si mostra lo stesso esempio in forma più compatta, accorpando alcune azioni in un unico gruppo di linee.

	RUNQUEUE		**TASKS**			
T				t1 (L0)	t2 (L0)	t3 (L0)
init	NRT	3	LC	1/3	1/3	1/3
	PER	6	VRTC	1	1	1
0	RQL/L0	3	Q	2	2	2
	CURR	t1	SUM	50	30	10
	RB	t2,t3	DELTA	0	0	0
	VMIN	1000	VRT	1000	1000,5	1001
exe	NRT		LC	1/3		
2	PER		VRTC	1		
csw	RQL/L0		Q	2		
	CURR	t2	SUM	52		
	RB	t3, t1	DELTA	2		
	VMIN	1000,5	VRT	1002		
exe	NRT		LC		1/3	
4	PER		VRTC		1	
csw	RQL/L0		Q		2	
	CURR	t3	SUM		32	
	RB	t1,t2	DELTA		2	
	VMIN	1001	VRT		1002,5	
exe	NRT		LC			1/3
6	PER		VRTC			1
csw	RQL/L0		Q			2
	CURR		SUM			12
	RB		DELTA			2
	VMIN	1002	VRT			1003

Tabella 2

Esempio 2.
Consideriamo ora il caso in cui i task hanno pesi diversi. Per semplicità i pesi sono espressi in funzione di L0.
In Tabella 3 si mostra un esempio simile al precedente, ma con pesi diversi per i 3 task:

- t1.LOAD = 1 * L0
- t2.LOAD = 1,5 *L0
- t3.LOAD = 0,5 *L0

RQL è rimasto invariato.
Si nota:

- i pesi indicati producono un forte squilibrio nell'esecuzione dei task; il quanto di t2 è il triplo del quanto di t1
- il VRT dei 3 task cresce invece di quantità identiche (2 ms), quindi l'ordinamento di esecuzione si mantiene inalterato, consolidando il vantaggio dei tsk più pesanti, che ad ogni ciclo beneficiano di un quanto più lungo
- viene comunque garantito al task più leggero di eseguire almeno un volta per ogni periodo

	RUNQUEUE		TASKS			
T				t1 (L0)	t2 (1,5 L0)	t3 (0,5 L0)
init	NRT	3	LC	1/3	1/2	1/6
	PER	6	VRTC	1	2/3	2
0	RQL/L0	3	Q	2	3	1
	CURR	t1	SUM	50	30	10
	RB	t2,t3	DELTA	0	0	0
	VMIN	1000	VRT	1000	1000,5	1001
exe	NRT		LC	1/3		
2	PER		VRTC	1		
csw	RQL/L0		Q	2		
	CURR	t2	SUM	52		
	RB	t3, t1	DELTA	2		
	VMIN	1000,5	VRT	1002		
exe	NRT		LC		1/2	
5	PER		VRTC		2/3	
csw	RQL/L0		Q		3	
	CURR	t3	SUM		33	
	RB	t1,t2	DELTA		3	
	VMIN	1001	VRT		1002,5	
exe	NRT		LC			1/6
6	PER		VRTC			2
csw	RQL/L0		Q			1
	CURR	t1	SUM			11
	RB	t2,t3	DELTA			1
	VMIN	1002	VRT			1003

Tabella 3

Possiamo ora analizzare lo pseudocodice della funzione `pick_next_task_fair( )` che sceglie il nuovo task di classe `SCHED_NORMAL` (si ricorda che questa funzione viene invocata sole se le altre classi di scheduling non hanno restituito un task da mettere in esecuzione).

```
pick_next_task_fair(rq){
   if (LFT != NULL) {
       //RB non è vuoto
       CURR = LFT;
       //elimina LFT da RB ristrutturando la coda opportunamente
       CURR->PREV = CURR->SUM; //salva in PREV il valore di SUM
   }
   else {
      //il RB è vuoto

      if (CURR == NULL) // CURR è stato eliminato perchè in ATTESA
         CURR = IDLE;
      //altrimenti CURR non viene modificato
   }
   //a questo punto CURR può essere uguale a LFT precedente,
   //oppure a IDLE,
   //oppure al precedente CURR (cioè non è stato modificato)

   return CURR;
}
```

La logica è evidente; si tenga presente che CURR è ancora il task che dovrebbe essere sostituito nell'esecuzione. Prima di sostituirlo il valore di SUM viene salvato in PREV. Se ci sono task pronti, scegli LFT, altrimenti, se CURR non è andato in attesa, prosegui la sua esecuzione; se anche questo non è possibile, esegui IDLE.

6.3 Gestione di wait e wakeup

L'idea di base del CFS per soddisfare il requisito 3, cioè garantire una risposta tempestiva dopo un wakeup si basa sul meccanismo fondamentale appena visto: dato che un processo in ATTESA non esegue, il VRT resta indietro rispetto agli altri, quindi al risveglio andrà in una posizione favorevole nel RB.

`wakeup` di un processo **tw**

Il risveglio di un processo lo fa reinserire nella runqueue e quindi modifica il parametro NRT il peso totale della runqueue (RQL); per conseguenza tutti i parametri dipendenti da NRT e RQL devono essere rivalutati.
Quando un processo viene risvegliato è possibile che il suo VRT sia molto basso (perchè è in ATTESA da lungo tempo) oppure sia ancora relativamente alto (ATTESA breve).
Il nuovo VRT che gli viene assegnato è:

```
tw.VRT = MAX(tw.VRT, (VRTMIN – LT/2) )
```

In base a questa assegnazione il processo risvegliato parte con un valore di VRT che lo candida ad essere eseguito nel prossimo futuro, ma non gli da un credito tale da distorcere completamente il sistema (come accadrebbe assegnando direttamente tw.VRT a un task che è stato in attesa per interi secondi).
Tipicamente, se il processo ha fatto un'attesa molto breve gli viene lasciato il suo VRT.

Inoltre, la formula precedente mostra che un task risvegliato può avere un VRT inferiore a VMIN e quindi al prossimo aggiornamento di VMIN da parte della funzione `tick`, VMIN scenderebbe. Dato che è opportuno che *VMIN cresca in maniera monotona*, la formula vista prima per l'assegnamento di VMIN nella funzione `tick` deve essere modificata:

```
VMIN = MAX(VMIN,  MIN( CURR.VRT,  LFT.VRT));
```

In questo modo se esiste un task con VRT inferiore a VMIN, il suo valore non produce un assegnamento di VMIN.

Per decidere se il nuovo processo deve causare un rescheduling vengono valutati due aspetti:

1. se il processo appartiene a una classi di scheduling superiore,
2. se il suo VRT è inferiore al VRT del processo corrente

La seconda valutazione viene però modificata con un correttivo che serve ad evitare che un processo che esegue attese brevissime causi commutazioni di contesto troppo frequenti; la formula applicata è contenuta nel seguente statement della funzione `check_preempt_curr(tw …)`, invocata da `wakeup():`

```
if ( (tw->schedule_class = RR) or
    ((tw->vrt + WGR * tw->load_coeff) <  CURR->vrt ) ) resched();
```

dove WGR (wakeup granularity) è un parametro di configurazione (SYSCTL) con default 1ms.

Esempio 3

Consideriamo la stessa situazione iniziale dell'esempio 1 (Tabella 2), ma ora il task t1 si pone in ATTESA dopo 0,7 ms di esecuzione.

In Tabella 4 è mostrato cosa accade al momento della wait:

1. vengono ricalcolati tutti i parametri generali, perché ora NRT=2 e RQL=2*L0;il quanto dei task diventa 3ms
2. viene messo in esecuzione t2

Dopo 2,5 ms di attesa il task t1 viene risvegliato. Vengono ricalcolati tutti i parametri generali.

A questo punto viene calcolato il valore di VRT da assegnare al task risvegliato t1:

t1.VRT = MAX(1000,7 , (1001 – 3)) = 1000,7

Infine viene valutata la necessità di rischedulare; dato che

t1.VRT + WGR * t1.LC = 1001 + (1*0,33) = 1001,33 < t2.VRT = 1003 → **resched**

	RUNQUEUE		TASKS			
T				t1 (L0)	t2 (L0)	t3 (L0)
init	NRT	3	LC	1/3	1/3	1/3
	PER	6	VRTC/L0	1	1	1
0	RQL/L0	3	Q	2	2	2
	CURR	t1	SUM	50	30	10
	RB	t2,t3	DELTA	0	0	0
	VMIN	1000	VRT	1000	1000,5	1001
t1.exe	NRT		LC	1/3		
0,7	PER		VRTC	1		
	RQL/L0		Q	2		
	CURR	t1	SUM	50,7		
	RB	t2,t3	DELTA	0,7		
	VMIN	1000,5	VRT	1000,7		
t1.wait	NRT	**2**	LC		**1/2**	**1/2**
resched	PER	6	VRTC		1	1
t2.exe	RQL/L0	**2**	Q		**3**	**3**
3,2	CURR	t2	SUM		32,5	
	RB	t3	DELTA		2,5	
	VMIN	1001	VRT		1003	
t2.wup t1;	NRT	**3**	LC	**1/3**	**1/3**	**1/3**
t2.resched	PER	6	VRTC	1	1	1
csw t2–> t1	RQL/L0	**3**	Q	**2**	**2**	**2**
	CURR	**t1**	SUM	50,7		
	RB	t3,t2	DELTA	**0**		
	VMIN	1001	VRT	1000,7		

Tabella 4

Esempio 4
Consideriamo la stessa situazione dell'esempio precedente, con il task t1 che si pone in ATTESA dopo 1 ms di esecuzione.
In questo caso però il task t1 viene risvegliato 0,6 ms dopo che è andato in ATTESA.
In Tabella 5 è mostrato cosa accade.

Viene calcolato il valore di VRT da assegnare al task risvegliato t1:

t1.VRT = MAX(1001, (1001 – 3)) = 1001

Infine viene valutata la necessità di rischedulare; dato che

t1.VRT + WGR * t1.LC = 1001 + (1*0,33) = 1001,33 > t2.VRT = 1001,1 → **non si invoca resched**

Questo esempio mostra l'effetto di WGR, perchè senza WGR avremmo avuto

t1.VRT = 1001 < t2.VRT = 1001,1 → **resched**

	RUNQUEUE		TASKS			
T				t1 (L0)	t2 (L0)	t3 (L0)
init	NRT	3	LC	1/3	1/3	1/3
	PER	6	VRTC/L0	1	1	1
0	RQL/L0	3	Q	2	2	2
	CURR	t1	SUM	50	30	10
	RB	t2,t3	DELTA	0	0	0
	VMIN	1000	VRT	1000	1000,5	1001
exe	NRT		LC	1/3		
1	PER		VRTC	1		
	RQL/L0		Q	2		
	CURR	t1	SUM	51		
	RB	t2,t3	DELTA	1		
	VMIN	1000,5	VRT	1001		
t1.wait	NRT	**2**	LC		**1/2**	**1/2**
t2.exe	PER	6	VRTC		1	1
0,6	RQL/L0	**2**	Q		**3**	**3**
	CURR	t2	SUM		30,6	
	RB	t3	DELTA		0,6	
	VMIN	1001	VRT		1001,1	
t2.wup t1;	NRT	**3**	LC	**1/3**	**1/3**	**1/3**
	PER	6	VRTC	1	1	1
(nessun	RQL/L0	**3**	Q	**2**	**2**	**2**
resched,	CURR	t2	SUM	51	30,6	
prosegue	RB	t3, t1	DELTA	1	0,6	
t2)	VMIN	1001	VRT	**1001**	1001,1	

Tabella 5

6.4 Creazione e cancellazione di task

Sia la cancellazione che la creazione di nuovi task comportano il ricalcolo dei parametri dovuto al cambiamento del numero di processi.
Per la cancellazione l'unico ulteriore aspetto che riguarda lo scheduler è la necessità di rescheduling.
Nel caso della creazione è invece necessario determinare il VRT iniziale da assegnare al processo; la regola adottata è la seguente:

tnew.VRT = VMIN + tnew.Q x tnew.VRTC

In base a questa assegnazione un processo nuovo (che ha tempo di esecuzione SUM=0) parte con un valore di VRT omogeneo agli altri processi. A differenza del caso di wakeup, al nuovo processo viene assegnato un valore di VRT che non lo posizionerà all'inizio della coda, ma comunque in modo da entrare nel periodo di scheduling che inizia con la sua creazione.
Questa assegnazione spiega perchè i valori di VRT sono generalmente molto più grandi di quelli di SUM nei processi, considerando che VMIN cresce monotonamente.
La necessità di rescheduling è valutata esattamente nello stesso modo del caso di wakeup; `sys_clone()` invoca `check_preempt_curr(tnew …)` e quindi viene eseguito lo stgatement:

```
if ( (tnew->schedule_class = RR) or
    ((tnew->vrt + WGR * tnew->load_coeff) <  CURR->vrt ) ) resched();
```

6.5 Riassunto delle notazioni e delle regole del CFS

Notazione

Parametri `SYSCTL`:

LT (`latency`) – default: 6ms
GR (`granularity`) – default: 0,75ms
WGR (`wakeup_granularity`) - default: 1ms;

Altri Parametri globali:

TICK dello scheduler periodico: ogni 0,01 ms
NRT = numero di task in stato di PRONTO –
NOW = istante corrente

Elementi della runqueue RQ:

CURR
VMIN
RB (Red Black Tree – è la coda ordinata in base al VRT dei task)
LFT (è il primo della RB, o leftmost)
IDLE (puntatore al descrittore di idle, che non è inserito in RB)
RQL = somma di tutti i t.LOAD, per tutti i task presenti sulla RQ

Elementi di ogni task t

VRT (virtual time)
SUM: è il tempo reale di esecuzione del task (alla creazione sum=0)
PREV: è il valore di SUM al momento in cui il task è diventato CURR
LOAD è il peso del task t

Coefficienti di peso

t.VRTC = `vrt_coeff`
t.LC = `load_coeff`

Regole

Calcolo dei coefficienti di peso
t.LC = `t.LOAD /` RQL
t.VRTC = L0 / t.LOAD

Calcolo del periodo di schedulazione PER
PER = MAX(LT, NRT * GR)

Calcolo del quanto di un task
t.Q = PER * LC

Aggiornamento periodico (ad ogni TICK e nei momenti di wakeup e creazione)
DELTA = NOW – START
SUM = SUM + DELTA
VRT = VRT + VRT*VRTC
VMIN = MAX(VMIN, MIN(VRT, LFT.VRT))
if (SUM - PREV) > Q --> resched

Wakeup di un processo tw
tw.VRT = MAX (tw.VRT, (VMIN – LT/2))
if ((tw.schedule_class = RR) or ((tw.VRT + WGR * tw.LC) < CURR.VRT)) resched

Creazione di un processo tnew
tnew.VRT = VMIN + tnew.Q* tnew.VRTC
if ((tnew.schedule_class = RR) or ((tnew.VRT + WGR * tnew.LC) < CURR.VRT))
resched

6.6 Il meccanismo di assegnazione dei pesi ai processi

L'assegnazione dei pesi ai processi si basa sul `nice_value`.

Nice value (priorità) e peso dei processi

L'utente può assegnare a un processo un `nice_value`. I `nice_value` vanno da -20 (massima priorità, bassa gentilezza – assegnabili solo dall'amministratore) a +19 (minima priorità, grande gentilezza). Il valore iniziale assegnato per default a un processo è 0. A parità di priorità e di politica di schedulazione, i processi che hanno `nice_value` maggiori ottengono in proporzione meno tempo di CPU rispetto a processi che hanno `nice_value` minori.

I `nice_value` sono trasformati in pesi dei task (t.LOAD); la regola di trasformazione, indicata in Tabella 6 estratta dal file `kernel/sched/sched.h`, corrisponde all'incirca alla seguente formula esponenziale:

$$\textbf{t.LOAD} = (1024/\ 1.25^{\text{nice\_value}})$$

kernel/sched/sched.h

```
/*
 * Nice levels are multiplicative, with a gentle 10% change for every
 * nice level changed. I.e. when a CPU-bound task goes from nice 0 to
 * nice 1, it will get ~10% less CPU time than another CPU-bound task
 * that remained on nice 0.
 *
 * The "10% effect" is relative and cumulative: from _any_ nice level,
 * if you go up 1 level, it's -10% CPU usage, if you go down 1 level
 * it's +10% CPU usage. (to achieve that we use a multiplier of 1.25.
 * If a task goes up by ~10% and another task goes down by ~10% then
 * the relative distance between them is ~25%.)
 */

static const int prio_to_weight[40] = {
 /* -20 */     88761,     71755,     56483,     46273,     36291,
 /* -15 */     29154,     23254,     18705,     14949,     11916,
 /* -10 */      9548,      7620,      6100,      4904,      3906,
 /*  -5 */      3121,      2501,      1991,      1586,      1277,
 /*   0 */      1024,       820,       655,       526,       423,
 /*   5 */       335,       272,       215,       172,       137,
 /*  10 */       110,        87,        70,        56,        45,
 /*  15 */        36,        29,        23,        18,        15,
};
```

Tabella 6

La costante `NICE_0_LOAD` vale quindi 1024, perché è il peso associato al `nice_value` 0.

6.6 Esempi sul sistema reale (Approfondimento)
Utilizzando i nice_value si può osservare il comportamento del sistema reale.

Esempio 1: effetto del nicevalue su 2 processi CPU_bound
Il programma di figura 2a crea 2 thread che eseguono ambedue per 10 volte la funzione cpu_load, che impiega 100ms. Il primo thread tf1 modifica il proprio nice_value a 5, mentre il thread tf2 mantiene il nice_value di default (0).

Il risultato dell'esecuzione, riportato in figura 2b. Come si vede, il thread tf2 riesce ad eseguire la funzione cpu_load il triplo delle volte rispetto a tf1.

Il valore die pesi associati ai nice_value 0 e 5, come indicato in tabella 6, sono 1024 e 335, e il loro rapporto è 3.05; la prova conferma quindi che il rapporto tra i pesi dei task determina il rapporto tra i relativi tempi di esecuzione

```
void cpu_load(int iterations) {
        long long i,j, k;
        for (k=0; k<iterations; k++){
                //circa 100ms ad ogni iterazione
                for (j=0; j<MAX; j++){  i=j; }
        }
}
void * tfcpu(void * tfarg){
        int k;
        int arg = (int) tfarg;
        if (arg == 1) {             //thread 1 – esegue con nice = 5
                 nice(5);      printf("start tf 1 \n");
        }
        else if (arg == 2) {      //thread 2 – esegue con nice di default = 0
                printf("start tf 2 \n");
        }
        //ciclo eseguito da ambedue i thread – durata 100 ms
        for (k=0; k<10; k++){
                printf("eseguito ciclo %d per tf %d \n", k, arg);
                cpu_load(1);
        }
        printf("tfcpu TERMINATA numero: %d\n", arg);
        return NULL;
}
int main(long argc, char * argv[], char * envp[]) {
        pthread_t t1, t2;
        int pid;
        pthread_create(&t1, NULL, &tfcpu, (void *) 1);
        pthread_create(&t2, NULL, &tfcpu, (void *) 2);
        printf("MAIN - ATTESA \n");
        pthread_join(t1, NULL);
        pthread_join(t2, NULL);
        printf("MAIN PROCESS TERMINATO \n");
        return printed_lines;
}
```

a) codice del programma

```
MAIN - ATTESA                   eseguito ciclo 2 per tf 1
start tf 2                      eseguito ciclo 7 per tf 2
eseguito ciclo 0 per tf 2       eseguito ciclo 8 per tf 2
start tf 1                      eseguito ciclo 9 per tf 2
eseguito ciclo 0 per tf 1       eseguito ciclo 3 per tf 1
eseguito ciclo 1 per tf 2       tfcpu TERMINATA numero: 2
eseguito ciclo 2 per tf 2       eseguito ciclo 4 per tf 1
eseguito ciclo 3 per tf 2       ....
eseguito ciclo 1 per tf 1       eseguito ciclo 9 per tf 1
eseguito ciclo 4 per tf 2       tfcpu TERMINATA numero: 1
eseguito ciclo 5 per tf 2       MAIN PROCESS TERMINATO
eseguito ciclo 6 per tf 2
```

b) risultato dell'esecuzione

Figura 2 – sperimentazione dell'effetto dei nice_value

Esempio 2: lettura dei valori interni al sistema

Nella seguente figura sono mostrati i valori delle variabili VRT, START, SUM e PREV di un task, ottenuti tramite un kernel module apposito.

I valori sono stati campionati facendo eseguire un programma che esegue 3 volte un ciclo che richiede circa 100ms; la prima riga mostra i valori a inizio programma, le seconda a inizio ciclo, e le successive alla fine di ogni iterazione del ciclo.

Il programma è stato rieseguito 3 volte con nice_value 0, 5, 10, ai quali corrispondono i pesi 1024, 335, 110 rispettivamente.

Nei risultati sono indicati anche i valori dei relativi vrt_coeff (VRTC) e la differenza tra i valori iniziale e finale del VRT e di SUM.

Tutti i valori temporali stampati sono in nanosecondi.

Si osserva che il rapporto tra SUM e VRT riflette esattamente il valore di VRTC; altri valori devono essere considerati con una certa tolleranza, per diversi motivi:

- la calibratura della durata del ciclo a 100 ms non è perfetta e non è del tutto stabile; il tempo effettivamente impiegato oscilla di alcuni percento
- nel sistema sono attivi altri task di sistema che interferiscono

Esempio – effetto di nice sui parametri dello scheduler
a) nice = 0 load 1024

[2453.427265] vrt: 29755 043792, start: 2453427 251962, sum: 586721, prev_sum: 586721
[2453.537675]
[2453.537679] vrt: 29852 647989, start: 2453536 559531, sum: 98 190918, prev_sum: 91 031363
[2453.636694]
[2453.636698] vrt: 29949 536024, start: 2453636 018675, sum: 195 078953, prev_sum: 191 411257
[2453.737772]
[2453.737776] vrt: 30047 380845, start: 2453737 174017, sum: 292 923774, prev_sum: 292 923774
[2453.846347]
[2453.846352] vrt: 30148 521595, start: 2453844 359992, sum: 394 064524, prev_sum: 390 213700
delta vrt: 393 ms **delta sum: 394**

b) nice = 5 load 335 (vrt_coeff: 3,1)

[2506.967770] vrt: 30221 851996, start: 2506967 762715, sum: 576577, prev_sum: 576577
[2507.076766]
[2507.076770] vrt: 30521 901471, start: 2507076 334538, sum: 98 737299, prev_sum: 79 506029
[2507.176417]
[2507.176417] vrt: 30818 620481, start: 2507176 416755, sum: 195 808464, prev_sum: 188 394180
[2507.280752]
[2507.280756] vrt: 31121 588611, start: 2507280 012015, sum: 294 924020, prev_sum: 288 610822
[2507.406717]
[2507.406721] vrt: 31424 604156, start: 2507404 016266, sum: 394 055088, prev_sum: 391 630771
delta vrt: 1203 ≈ delta sum * 3,1 **delta sum: 394**

c) nice = 10 load 110 (vrt_coeff: 9,3)

[2935.301661] vrt: 32757 003504, start: 2935301 652169, sum: 604004, prev_sum: 604004
[2935.403602]
[2935.403606] vrt: 33618 675839, start: 2935400 010181, sum: 93 166464, prev_sum: 29 850013
[2935.499936]
[2935.499941] vrt: 34519 023316, start: 2935496 849093, sum: 189 883480, prev_sum: 133 064418
[2935.594520]
[2935.594523] vrt: 35406 688692, start: 2935592 208872, sum: 285 238161, prev_sum: 193 355737
[2935.691773]
[2935.691776] vrt: 36275 207653, start: 2935688 021581, sum: 378 536098, prev_sum: 294 564557
delta vrt: 3518 ≈ delta sum * 9,3 **delta sum: 378**

Figura 3