Architettura dei Calcolatori e Sistemi Operativi

Il Nucleo del Sistema Operativo

N5 - Lo Scheduler

06.01.2016

Funzioni Generali dello *Scheduler* – I

definizione

- il *SO* è caratterizzato dalle *politiche* adottate per decidere ***quali task eseguire e per quanto tempo ciascuno – politiche di scheduling***
- il componente del *SO* che realizza le politiche di *scheduling* è detto ***scheduler***

comportamento dello scheduler

- lo *scheduler* ha un comportamento orientato a garantire le condizioni seguenti:
  - che i task più importanti vengano eseguiti prima di quelli meno importanti
  - che i task di pari importanza vengano eseguiti in maniera equa (senza privilegi)
  - e in particolare, che nessun task debba attendere un turno di esecuzione per molto più tempo degli altri task
- lo *scheduler* è un componente critico nel funzionamento di un sistema operativo ed è oggetto di molta ricerca per migliorarne le caratteristiche e le prestazioni

Funzioni Generali dello *Scheduler* – II

politica di scheduling equa

- per un sistema multi-programmato, la gestione dell'esecuzione dei task più immediata, semplice e ragionevolmente equa, è la politica ***round robin***
- dati $N \geq 1$ task di pari importanza, la politica di *scheduling round robin* (*a turno*) assegna un uguale *quanto* di tempo (*timeslice*) a ciascun task circolarmente
- la politica *round robin* è *equa* (*fair*) e garantisce che un task non resti fermo indefinitamente, cioè che non vada incontro a *morte per inedia* (*starvation*)

gestione della runqueue

- lo *scheduler* interviene in certi momenti per determinare quale task (ri)mettere in esecuzione, e contestualmente toglie un altro task dall'esecuzione
- la scelta del task da (ri)mettere in esecuzione avviene tra tutti i task in stato di *PRONTO* esistenti nel sistema, cioè tra tutti quelli nella *runqueue*
- il task scelto è quello che in quel momento ha il ***diritto di esecuzione*** maggiore

Funzioni Generali dello *Scheduler* – III

Tre situazioni in cui lo *scheduler* deve scegliere un task corrente

1. quando un task si *autosospende* e lascia l'esecuzione, poiché bisogna trovare un altro task da eseguire (il processore va sempre assegnato a un task – al limite al task *idle*)
2. quando un task in stato di *ATTESA* viene *risvegliato* da parte di un altro task e così si amplia l'insieme dei task in stato di *PRONTO*, poiché:
   - il task risvegliato potrebbe avere un diritto di esecuzione maggiore di quello corrente
   - se questo è il caso, il maggiore diritto di esecuzione si traduce in *diritto di preemption*, cioè causa la sospensione forzata del task corrente e la sua sostituzione con quello risvegliato
   - però si ricordi che l'attuazione effettiva della *preemption* richiede una commutazione di contesto ed è differita fino al prossimo ritorno a modo U !
3. quando il task correntemente in esecuzione è gestito con politica di *scheduling* di tipo *round robin* e il *quanto di tempo* (*timeslice*) assegnato a tale task *scade*

Task e Requisiti di Schedulazione – I

I task possono avere requisiti di *scheduling* molto diversificati per applicazione e si si possono distinguere i task nelle tre categorie seguenti

1. task ***real-time*** (in ***senso stretto*** o ***hard real-time***): devono soddisfare vincoli di tempo molto stringenti e dunque vanno schedulati con grande rapidità
2. task ***semi-real-time*** (***soft real-time***): possono reagire con una discreta rapidità, ma non garantiscono di non superare un ritardo max fissato
3. task ***normali***: tutti gli altri task, divisi in queste due sotto-categorie principali:
   - ***task I/O bound***: si autosospendono frequentemente poiché hanno bisogno di dati di *I/O*, come per esempio un programma di *videoscrittura* (*text editor*)
   - ***task CPU bound***: tendono a usare la *CPU* per la maggior parte del loro tempo poiché si autosospendono raramente, come per esempio un *compilatore* (*compiler*)

Task e Requisiti di Schedulazione – II

- per gestire ciascuna categoria di task secondo le rispettive caratterstiche distintive, lo *scheduler* realizza varie **politiche di *scheduling***
- ciascuna politica è realizzata da una ***classe di scheduling*** diversa (***scheduler class***)
- nel descrittore di un task il campo seguente contiene un puntatore alla struttura della classe di *scheduling* deputata a gestirlo

```
const struct sched_class * sched_class
```

- lo *scheduler* è l'unico gestore dei task in stato di *PRONTO*, cioè della *runqueue*
- per questo motivo tutte le altre funzioni del *SO* e in particolare quelle del nucleo devono chiedere allo *scheduler* di eseguire operazioni sulla *runqueue*
- la struttura interna di una classe di *scheduling* è piuttosto complessa (vedi dopo)

Classe di *Scheduling* (*Scheduler Class*)

```
// descrittore (semplificato) della classe di scheduling e inizializzazione
static const struct sched_class fair_sched_class = {
    .next                = &idle_sched_class,
    .enqueue_task        = enqueue_task_fair,
    .dequeue_task        = dequeue_task_fair,
    .check_preempt_curr  = check_preempt_wakeup,
    .pick_next_task      = pick_next_task_fair,
    .put_prev_task       = put_prev_task_fair,
    .set_curr_task       = set_curr_task_fair,
    .task_tick           = task_tick_fair,
    .task_new            = task_new_fair,
} \* sched_class *\
```

NOTA BENE

i campi sono puntatori a funzione inizializzati alle versioni *fair* delle funzioni che realizzano il meccanismo *CFS* completo – vedi per esempio più avanti la versione *CFS* della funzione *pick_next_task*

- per mettere un task in stato di *PRONTO*, cioè per eseguire la funzione *enqueue*, il nucleo invoca *p->sched_class->enqueue_task* ed esegue la funzione *enqueue_task_fair*
- è piuttosto facile aggiungere al *SO* nuove classi di *scheduling* o aggiornare quelle esistenti

APPROFONDIMENTO

Politiche di *Scheduling* Fondamentali

Attualmente le tre classi di *scheduling* più importanti del *SO* Linux sono le seguenti:

nome della classe	***politica propria della classe***	***diritto vs le altre classi***
SCHED_FIFO	First In First Out	massimo
SCHED_RR	Round Robin	medio
SCHED_NORMAL	la più complessa (vedi dopo)	minimo

- il *SO* Linux gestisce i task secondo la politica propria della classe di appartenenza:

  FIFO task eseguiti per intero non appena selezionati

  RR task eseguiti a turno in modo strettamente circolare

  NORMAL task gestiti dinamicamente in tempo virtuale (vedi dopo)

- il rapporto tra i diritti di esecuzione di due task in classi differenti è il seguente:

  i task di classe *FIFO* hanno sempre precedenza su tutti quelli delle altre due classi

  i task di classe *RR* hanno sempre precedenza su tutti quelli della classe *NORMAL*, ma danno sempre precedenza a tutti quelli della classe *FIFO*

  i task di classe *NORMAL* danno sempre precedenza a tutti quelli delle altre due classi

Pseudo-codice di *schedule* – Struttura Generale – I

- la funzione ***schedule ( )*** invoca la funzione ***pick_next_task*** per scegliere nella *runqueue* il prossimo task corrente, poi se occorre procede alla *commutazione di contesto* (*context switch*) tramite la macro ***context_switch*** (qui non espansa)

```
schedule (  ) {
   ...
   struct tsk_struct * prev, next;
   prev = CURR;
   if (prev->stato == ATTESA) {
      // togli il task corrente prev dalla runqueue rq
   } /* if */
   // invoca la funzione di scelta del prossimo task e passa rq, prev = CURR
   next = pick_next_task (rq, prev);
   // se non ci sono task pronti nella classi con diritto maggiore (FIFO e RR)
   // il task next viene restituito dalla funzione pick_next_task_fair di CFS
   if (next != prev) { // confronta il task corrente prev e quello scelto next
       // se next è diverso da prev esegui la commutazione di contesto a next
      context_switch (prev, next); // inclusione della macro context_switch
   } /* if */
} /* schedule */
```

scheduling della classe *FIFO*, *RR* e *NORMAL* secondo le regole rispettive

il task *CURR* è andato in *ATTESA* ed esce dalla *runqueue*

se occorre effettua la commutazione di contesto tramite la macro apposita

Pseudo-codice di *schedule* – Struttura Generale – II

- la funzione *pick_next_task* ( ) scandisce le classi di *scheduling* nell'ordine di importanza *FIFO*, *RR* e *NORMAL*, e invoca la funzione *pick_next_task* specifica della classe per scegliere nella *runqueue* il prossimo task corrente, restituendolo

```
pick_next_task (rq, prev) {
   ...
   struct tsk_struct * next;
   for (ciascuna classe di scheduling, in ordine di importanza decrescente) {
      // invoca la funzione di scelta del prossimo task per la classe in esame
      next = class->pick_next_task (rq, prev); // class è la var del ciclo for
      if (next != NULL) {
         // quando nella classe in esame trovi un task, restituiscilo e termina
         return next;
      } /* if */
   } /* for */
   // pick_next_task restituisce sempre un puntatore valido, in questo modo:
   // - a un task PRONTO con diritto di esecuzione massimo, se ne esiste uno
   // - al task prev, se non ce ne sono di pronti e prev non ha stato = ATTESA
   // - al task IDLE, se nessuno dei due casi precedenti è praticabile
} /* pick_next_task */
```

qui la funzione *schedule* ( ) passa il task corrente *CURR*

scheduling delle classi

vedi la **struct** sched_class

Scheduling dei Task *Soft Real-Time* – I

- le classi *SCHED_FIFO* e *SCHED_RR* sono usate per i task di tipo *soft real-time*
  - il *SO* Linux non supporta i processi *RT* in *senso stretto* (*hard real-time*)
  - motivo: non è in grado di garantire il non superamento di un ritardo max
- per queste due classi il concetto fondamentale è quello di *priorità statica*
- a ciascun task di queste due classi viene attribuita alla creazione una *priorità* detta *statica*, perché è assegnata all'inizio e poi (solitamente) non varia più
  - ✓solitamente un task figlio eredita la priorità statica del task padre
  - ✓però si può cambiare la priorità statica di un task (con comandi di amministratore)
- i valori di priorità statica appartengono all'intervallo 1 - 99 (con 99 priorità max)
- la priorità statica del task è memorizzata in *task_struct* nel campo *static_prio*

Scheduling dei Task *Soft Real-Time* – II

classe di scheduling SCHED_FIFO

- quando un task entra in esecuzione, viene eseguito senza limite di tempo
  fino a quando si autosospende (esegue *wait_event*)
  fino alla terminazione naturale (esegue *sys_exit*)
- se ci sono due o più task pronti, si sceglie quello a priorità statica maggiore

classe di scheduling SCHED_RR

- due o più task allo stesso livello di priorità statica sono eseguiti in *round robin*
- pertanto ogni task viene eseguito per un *quanto di tempo* a turno circolarmente
  nota: ciascun livello di priorità ha una sua coda di task gestita in *round robin*

Scheduling dei Task di Classe *NORMAL* (*CFS*) – I

Lo *scheduler* Linux candida all'esecuzione i task di classe *NORMAL* solo se in stato di *PRONTO* non ci sono task delle classi *FIFO* e *RR*, che li precedono sempre

- lo *scheduler* Linux per i task di classe di *scheduling SCHED_NORMAL* è chiamato:

> ***Completely Fair Scheduler* (*CFS*)**

- lo *scheduler CFS* ambisce a raggiungere il seguente *obiettivo ideale*, per ogni *CPU*:

> *dati $N \geq 1$ task tutti assegnati a una CPU di potenza 1, dedicare a ciascun task una CPU «virtuale» di potenza 1 / N*

- in pratica la *CPU* va assegnata a ciascun task per un opportuno ***quanto*** di tempo
- se il sistema è multi-processore (o multi-core) ciascuna *CPU* ha una sua *runqueue* da gestire in modo *CFS* cercando di conseguire l'obiettivo ideale indicato sopra

Scheduling dei Task di Classe *NORMAL* (*CFS*) – II

Per raggiungere il suo obiettivo lo *scheduler CFS* deve

1. ***determinare ragionevolmente la durata del quanto di tempo*** (fissa o variabile):

   un quanto lungo riduce la responsività del sistema

   un quanto breve sovraccarica il sistema, per troppe commutazioni di contesto

2. ***assegnare un certo peso a ciascun task***, in modo che ai task più importanti sia dato più peso e dunque più tempo di esecuzione che a quelli meno importanti

3. ***permettere*** a un task rimasto a lungo in stato di *ATTESA* di ***tornare rapidamente in esecuzione*** quando viene risvegliato, ma senza favorirlo troppo

- *CFS* è costituito da una base essenzialmente di tipo *round robin* per gestire i task *uniformemente*, alla quale si aggiungono certi raffinamenti (*correttivi*), semplici, rapidi da calcolare ed efficaci, per considerare le caratteristiche *individuali* di ciascun task

Meccanismo Base di *CFS* – Durata del Quanto

- a ogni task si assegna un ***peso*** (*LOAD*) iniziale, che quantifica l'importanza del task
- la costante di sistema *NICE_0_LOAD*, qui denotata con ***L0***, definisce il peso iniziale

- ipotesi semplificatrici di partenza per illustrare il meccanismo base di *CFS*:

  t.LOAD = L0 per tutti i task *t* presenti nella *runqueue*

  nessun task si autosospende (nessuno esegue *wait_event*)

- il *numero di task* nella *runqueue* a un certo istante è denotato con $NRT \geq 1$
- si stabilisce un *periodo di scheduling* *PER* durante cui tutti i task della *runqueue* possono essere eseguiti se non si autosospendono
- a ogni task si assegna un uguale *quanto di tempo* *Q* la cui durata è calcolata così:

$$Q = PER \,/\, NRT$$

Meccanismo Base di *CFS* – Gestione della *runqueue*

- i task vengono mantenuti in una coda ordinata chiamata *RB*, e il funzionamento dello *scheduler CFS* è schematizzabile nel modo seguente:
  1. il task in *testa* a *RB* viene estratto e diventa corrente (*CURR*)
  2. il task *CURR* viene eseguito fino a quando *scade* il quanto *Q*
  3. il task *CURR* viene sospeso e reinserito in *fondo* a *RB*
  4. si torna al passo 1

- in pratica i task sono eseguiti a turno per esattamente *PER / NRT ms*
- il periodo di schedulazione *PER* è una sorta di *finestra scorrevole* nel tempo:
  non c'è alcuna suddivisione rigida del tempo in intervalli disgiunti consecutivi
  si può considerare ogni istante come l'inizio di un nuovo periodo di schedulazione
- osservando il sistema a partire da un istante casuale per un intervallo di durata pari a *PER ms*, tutti i task vengono eseguiti, ciascuno per un quanto *Q* di tempo

Meccanismo Base di *CFS* – Periodo di *Scheduling*

- il periodo di *scheduling PER* varia *dinamicamente* con il crescere o diminuire del numero di task *NRT* presenti nella *runqueue*, poiché:
  - un periodo troppo lungo può ritardare troppo l'esecuzione di un task
  - un periodo troppo corto può produrre quanti troppo brevi al crescere di *NRT*
- attualmente il *SO* Linux determina il periodo di *scheduling PER* tramite due parametri di controllo (parametri *SYSCTL*) modificabili dall'amministratore di sistema:

LT	*latenza*	default 6 *ms*	*durata minima del periodo PER*
GR	*granularità*	default 0,75 *ms*	

- il periodo di *scheduling PER* è calcolato secondo la formula seguente:

$$PER = max\ (\ LT,\ \ NRT \times GR\ )$$

- se la latenza *LT* è maggiore di $NRT \times GR$, il quanto Q è $> GR$, altrimenti è $= GR$
- di default, con 8 o meno task il periodo *PER* ha il valore fisso minimo di 6 *ms*

Meccanismo Completo di *CFS* – valutare il singolo Task

- il meccanismo base si fonda su un quanto *Q* costante e sulla politica *round robin*
- si tratta di un punto di partenza per realizzare uno *scheduler fair* (*equo*) effettivo
- ma il meccanismo effettivo deve tenere conto di due aspetti rilevanti di *fairness*:
  - la durata del quanto di tempo deve dipendere dal *peso effettivo* assegnato al task
  - il tempo di esecuzione va misurato *virtualmente* secondo il *comportamento* del task
- il meccanismo *CFS* completo è dunque un po' più complesso, ma non troppo
- l'aspetto più importante di *CFS* è quello relativo alla misura virtuale del tempo
- tale misura virtuale (ri)calcola certe variabili per ogni task, in tre circostanze:
  1. a ogni impulso del *real-time clock*, cioè nella funzione *task_tick* ( ) dello *scheduler*
  2. a ogni risveglio del task, cioè nella funzione *wake_up* del nucleo
  3. alla creazione del task, per inizializzarle, cioè insieme al servizio *sys_clone*
- la decisione su quale task mettere in esecuzione viene poi presa dalla funzione *schedule* ( ) quando viene chiamata

Durata del Quanto in Funzione del Peso del Task

- la formula $Q = PER / NRT$ non tiene conto dei pesi dei task, considerandoli uguali
- bisogna valutare il **peso relativo dello specifico task *t*** rispetto a tutti i task

$t.LOAD$ — peso dello specifico task *t* (si può assegnare con comandi di admin)

$$RQL = \sum_{\forall \text{ task } t \text{ in } runqueue} t.LOAD$$

somma dei pesi di tutti i task nella *runqueue* (*rqload*)

$$t.LC = t.LOAD / RQL$$

rapporto tra il peso dello specifico task *t* e *RQL* (*load_coeff*)

o equivalentemente $t.LOAD = t.LC \times RQL$

- la durata del quanto di tempo *Q* di uno specifico task *t*, denotata con ***t.Q***, dipende allora dal task *t* ed è proporzionale al peso di *t* rispetto al peso di tutti i task

$$t.Q = PER \times t.LC$$

- se tutti i task hanno lo stesso peso, cioè si fissa $t.LOAD = L0$, si riottiene il quanto $Q = PER / NRT$ uguale per tutti i task illustrato prima in tale ipotesi, poiché si ha:

$$RQL = NRT \times L0 \quad \text{e dunque} \quad Q = PER \times LC = PER \times \left(L0 / (NRT \times L0)\right) = PER / NRT$$

Virtual RunTime – Concetto

- lo *scheduler CFS* usa il ***Virtual RunTime*** (***VRT***) per ordinare i task nella *runqueue*

il VRT è una misura virtuale del tempo di esecuzione consumato da un processo, basata sulla modifica del tempo reale tramite coefficienti opportuni

- la decisione su quale sia il prossimo task da mettere in esecuzione si basa semplicemente sulla scelta del task con ***VRT minimo*** tra quelli nella *runqueue*
- la *runqueue* è costituita dal puntatore *CURR* al (descrittore del) task corrente e dalla coda *RB* ordinata in base ai *VRT* dei task (il task corrente non si trova in *RB*)
  - ✓ c'è sempre un task corrente, mentre la coda *RB* può essere vuota (raramente ...)
  - ✓ il prossimo task da eseguire è il primo in *RB* e si indica con *LFT* (*leftmost task*), salvo che la coda *RB* sia vuota, nel qual caso il task corrente continua l'esecuzione
- il ***VRT del task corrente*** viene ricalcolato a ogni *tick* del *real-time clock* del sistema
- quando il task corrente termina l'esecuzione, viene reinserito in *RB* nella posizione che gli compete in base al valore di *VRT* assunto durante l'esecuzione

Virtual RunTime – (ri)Calcolo

Base dell'algoritmo di (ri)calcolo del *Virtual Time VRT* di un task:

variabili ausiliarie	***significato***
SUM	*tempo totale (reale) di esecuzione del task*
DELTA	*durata dell'ultimo quanto consumato dal task (DELTA ≤ Q)*
VRTC	*coefficiente di correzione di VRT (vrt_coeff)*
VRTC = L0 / LOAD	*rapporto tra L0 (peso default) e peso del task*

stanno nella funzione *tick*

(ri)calcolo del Virtual RunTime VRT	***significato***
$SUM = SUM + DELTA$	*tempo reale (fisico) di esecuzione del task corrente*
$VRT = VRT + \underbrace{DELTA \times VRTC}_{\Delta\, VRT}$	*tempo virtuale (cioè corretto con VRTC) di esec. del task*

- il coefficiente *VRTC* fà crescere i *VRT* dei task più pesanti meno dei *VRT* di quelli più leggeri, in modo da non avvantaggiare troppo i primi a scapito dei secondi

Meccanismo Completo di *CFS* – Commento

- se il numero di task è costante e ogni task consuma tutto il quanto *Q* assegnatogli, cioè se vale *DELTA* = *Q*, in un periodo *PER* i *VRT* di tutti i task crescono di una stessa quantità

$$\Delta\, VRT = DELTA \times VRTC = Q \times (L0 \,/\, LOAD) = (PER \times LC) \times \big((L0 \,/\, (LC \times RQL)\big) = PER \times L0 \,/\, RQL$$

- in tale caso l'incremento $\Delta\, VRT$ del *virtual runtime* non dipende dal peso (*LOAD*) del task

se Δ VRT è indipendente dal task, ordinare la runqueue per VRT e scegliere il task con VRT minimo equivale a gestire la runqueue in round robin !

In questo caso lo *scheduler CFS* completo è realizzato a partire da una base di tipo *round robin*

- con tale base si ha una politica *equa* (*fair*) se tutti i task si comportano in modo uniforme:

  il numero di task è costante nel tempo (il parametro *NRT* è costante)
  i task consumano l'intero quanto di tempo senza interruzione (non si autosospendono mai)

Il meccanismo *CFS* completo con quanto *Q* e tempo virtuale *VRT* dipendenti dal task modifica la politica *round robin* adattandola ai vari task, ma senza snaturarla (vedi dopo)

APPROFONDIMENTO

Meccanismo Completo di *CFS* – Coda Ordinata *RB*

- struttura della coda ordinata *RB* che contiene i task in stato di *PRONTO*
  - la coda *RB* è *ordinata* secondo valori *crescenti* del parametro *VRT* dei task
  - il task in testa alla coda è chiamato task *LFT* (*leftmost*) e ha il *VRT minimo*
- operazioni sulla coda *RB* – inserimento ordinato ed estrazione di un task
  - un nuovo task viene sempre inserito a partire dal fondo della coda e avanza nella coda fino a trovare un task con *VRT uguale o minore*, oppure fino a diventare *LFT* (quando ha *VRT* minimo)
  - l'unico task estraibile dalla coda è il task *LFT*, cioè quello in testa e che ha sempre *VRT minimo*
- *se i VRT dei task vengono sempre incrementati di quantità uguali (p. es. 1), la coda RB contiene dalla testa (LFT) solo task con VRT = n eventualmente seguiti da task con VRT = n + 1, fino in fondo*
- *dunque quando il task LFT con VRT = n viene scelto e tolto da RB, diventa corrente, incrementa VRT a n + 1, poi lascia l'esecuzione e rientra in RB, esso si riposiziona esattamente in fondo alla coda*
- *pertanto, come detto prima, se i task sono costanti e non si autosospendono mai, allora la runqueue di CFS con il meccanismo VRT funziona complessivamente come round robin puro*
- *la coda RB è un «Red-Black tree», cioè una struttura dati che logicamente funziona come una coda ordinata ma in realtà è organizzata ad albero, ed è usata per l'efficienza dell'inserimento ordinato*

APPROFONDIMENTO

Virtual RunTime – Tempo Virtuale Minimo

- insieme al *VRT* del task corrente, lo *scheduler* ricalcola una variabile *VMIN* (*vrtmin*) che rappresenta il *VRT minimo* tra i *VRT* di tutti i task presenti nella *runqueue*
- come si vedrà, *VMIN* serve per riallineare i *VRT* dei task risvegliati, che dopo un'attesa lunga hanno un *VRT* molto arretrato rispetto ai task rimasti in *runqueue*
- siccome rappresenta un tempo, la variabile *VMIN* deve crescere monotonicamente
- versione *provvisoria* della formula per (ri)calcolare *VMIN*, assai intuitiva:

sta nella funz. *tick*

VMIN = min (CURR.VRT, LFT.VRT) // *LFT* è il primo task di *RB* (in *RB* ha *VRT* minimo)

- dato che la coda *RB* è sempre ordinata per *VRT* crescenti a partire dalla testa, basta prendere il *VRT* minimo tra quello di *CURR* e quello del task in testa a *RB*
- ma la formula sopra tratta *erroneamente* certi casi di risveglio, e verrà *emendata*

APPROFONDIMENTO

Funzione *tick* ( ) con *Virtual RunTime*

```
\\ NOW istante corrente e CURR puntatore al task corrente
\\ START istante in cui un task va in esecuzione
\\ PREV  valore di SUM quando un task va in esecuzione
\\ DELTA è una variabile locale della funzione tick
task_tick_fair ( ) {
   // CFS - ricalcolo dei parametri di CURR
   DELTA = NOW – CURR->START;
   CURR->SUM = CURR->SUM + DELTA;
   CURR->VRT = CURR->VRT + DELTA * CURR->VRTC;
   CURR->START = NOW;
   // ricalcolo di VMIN della runqueue (ancora provvisorio)
   VMIN = min (CURR->VRT, LFT->VRT); // formula provvisoria
   // controllo di scadenza del quanto di tempo di CURR
   if ((CURR->SUM – CURR->PREV) > CURR->Q) resched ( );
 } /* tick */
```

ricalcolo dei parametri del *virtual runtime* del task *CURR*

da completare e chiarire più avanti

poni TIF_NEED_RESCHED a 1

NOTA BENE

in pratica il *VRT* del task viene (ri)calcolato a ogni *tick* del *real-time clock*

il valore dell'intervallo *DELTA* è (ri)calcolato come tempo intercorso tra due (ri)calcoli consecutivi del *VRT* del task (*DELTA* può valere meno di un quanto)

però alla fine del quanto il *VRT* del task accumula la durata dell'intero quanto

Esempio 1 – Task senza Attesa

tempi in *ms* – *LT* e *GR* hanno valori di default (*PER* = 6)

VRTC = L0/LOAD

VRT = VRT + DELTA x VRTC

ATTIVITÀ E TEMPO	RUNQUEUE		TASK – *LOAD = L0* per *t1*, *t2* e *t3*			
			L0=1/3	*t1*	*t2*	*t3*
INIT 0	NRT	3	LC	1 / 3	1 / 3	1 / 3
	PER	6	VRTC	1	1	1
	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 2	NRT	VALORI COME PRIMA	LC	1 / 3	VALORI COME PRIMA	VALORI COME PRIMA
	PER		VRTC	1		
	RQL / L0		Q	2		
	CURR		SUM	52		
	RB		DELTA	2		
	VMIN	1000,5	VRT	1002		
CSW *t1* → *t2*	NRT	VALORI COME PRIMA	LC	**CONTEXT SWITCH** tempo trascurabile		
	PER		VRTC			
	RQL / L0		Q			
	CURR	t2	SUM			
	RB	t3, t1	DELTA			
	VMIN	1000,5	VRT			
EXE 4	NRT	VALORI COME PRIMA	LC	VALORI COME PRIMA	1 / 3	VALORI COME PRIMA
	PER		VRTC		1	
	RQL / L0		Q		2	
	CURR		SUM		32	
	RB		DELTA		2	
	VMIN	1001	VRT		1002,5	
CSW *t2* → *t3*	NRT	VALORI COME PRIMA	LC	**CONTEXT SWITCH** tempo trascurabile		
	PER		VRTC			
	RQL / L0		Q			
	CURR	t3	SUM			
	RB	t1, t2	DELTA			
	VMIN	1001	VRT			
EXE 6	NRT	VALORI COME PRIMA	LC	VALORI COME PRIMA	VALORI COME PRIMA	1 / 3
	PER		VRTC			1
	RQL / L0		Q			2
	CURR		SUM			12
	RB		DELTA			2
	VMIN	1002	VRT			1003
CSW *t3* → *t1*	NRT	VALORI COME PRIMA	LC	**CONTEXT SWITCH** tempo trascurabile		
	PER		VRTC			
	RQL / L0		Q			
	CURR	t1	SUM			
	RB	t2, t3	DELTA			
	VMIN	1002	VRT			

Tab. 1 – esecuzione *EXE* e commutazione di contesto *CSW* separate

ATTIVITÀ E TEMPO	RUNQUEUE		TASK			
			L0 = 1/3	*t1* *LOAD = L0*	*t2* *LOAD = L0*	*t3* *LOAD = L0*
INIT 0	NRT	3	LC	1 / 3	1 / 3	1 / 3
	PER	6	VRTC	1	1	1
	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 2 CSW *t1* → *t2*	NRT		LC	1/3		
	PER		VRTC	1		
	RQL / L0		Q	2		
	CURR	t2	SUM	52		
	RB	t3, t1	DELTA	2		
	VMIN	1000,5	VRT	1002		
EXE 4 CSW *t2* → *t3*	NRT		LC		1 / 3	
	PER		VRTC		1	
	RQL / L0		Q		2	
	CURR	t3	SUM		32	
	RB	t1, t2	DELTA		2	
	VMIN	1001	VRT		1002,5	
EXE 6 CSW *t3* → *t1*	NRT		LC			1 / 3
	PER		VRTC			1
	RQL / L0		Q			2
	CURR	t1	SUM			12
	RB	t2, t3	DELTA			2
	VMIN	1002	VRT			1003

Tab. 2 – come tab. 1 ma *EXE* e *CSW* sono riunite

Esempio 2 – Task senza Attesa ma con Pesi Differenti

i task hanno pesi (*LOAD*) differenti
tutti i task sono sempre in *runqueue*

$$RQL = L0 + 1{,}5 \times L0 + 0{,}5 \times L0$$
$$= 1/3 + 1/2 + 1/6 = 1$$

i *VRTC* e i quanti di tempo *Q* dipendono dai task

i pesi assegnati squilibrano l'esecuzione dei task: il quanto di *t2* è triplo rispetto a *t1*

invece i *VRT* dei tre task crescono di quantità identiche, cioè 2 *ms*, pertanto l'ordine di esecuzione non muta e avvantaggia i task più pesanti che a ogni ciclo beneficiano di un quanto più lungo

comunque al task più leggero viene garantito di andare in esecuzione almeno una volta per ogni periodo di *scheduling*

tempi in *ms* – *LT* e *GR* hanno valori di default (*PER* = 6)

ATTIVITÀ E TEMPO	RUNQUEUE		TASK			
			L0 = 1/3	***t1*** ***LOAD = L0***	***t2*** ***LOAD = 1,5 × L0***	***t3*** ***LOAD = 0,5 × L0***
INIT 0	NRT	3	LC	1 / 3	1 / 2	1 / 6
	PER	6	VRTC	1	2 / 3	2
	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 2 CSW *t1 → t2*	NRT		LC	1 / 3		
	PER		VRTC	1		
	RQL / L0		Q	2		
	CURR	t2	SUM	52		
	RB	t3, t1	DELTA	2		
	VMIN	1000,5	VRT	1002		
EXE 5 CSW *t2 → t3*	NRT		LC		1 / 2	
	PER		VRTC		2 / 3	
	RQL / L0		Q		3	
	CURR	t3	SUM		33	
	RB	t1, t2	DELTA		3	
	VMIN	1001	VRT		1002,5	
EXE 6 CSW *t3 → t1*	NRT		LC			1 / 6
	PER		VRTC			2
	RQL / L0		Q			1
	CURR	t1	SUM			11
	RB	t2, t3	DELTA			1
	VMIN	1002	VRT			1003

Esempio 2 (tab. 3) – task con pesi differenti e loro effetti rispetto a es. 1

Pseudo-codice di *pick_next_task_fair* ( ) – Classe *NORMAL CFS*

- la funzione *pick_next_task_fair* ( ) della classe di *scheduling NORMAL* seleziona il prossimo task da eseguire secondo le regole di *CFS*, basate sull'uso di *VRT* – si prende il task con *VRT* minore: *LFT,* o *CURR* se non c'è *LFT,* o *IDLE* se non c'è altro

```
pick_next_task_fair (rq, prev) {
   ...
   struct tsk_struct * next;
   // scelta del prossimo task secondo le regole della classe di scheduling NORMAL
   if (LFT != NULL) {
      // la coda RB dei task pronti non è vuota (LFT è il task con VRT minore)
      next = LFT;
      // togli LFT dalla coda RB dei task pronti e ristrutturala opportunamente
      next->PREV = next->SUM; // inoltre salva il valore di SUM nel campo PREV
   } else {
      // la coda RB dei task pronti è vuota (caso raro, ma succede – c'è solo CURR)
      if (prev == NULL) { // il task CURR è stato eliminato perché andato in ATTESA
         next = IDLE;
      } /* if */
   } /* if */
   // ora il task next può essere LFT o CURR se non c'è LFT o IDLE se non c'è altro
   return next; // restituisci il task scelto come prossimo da eseguire e termina
} /* schedule */
```

scheduling classe *NORMAL*

qui la funzione *pick_next_task* ( ) passa il task corrente *CURR*

LFT diventa il task *CURR*

IDLE diventa il task *CURR*

rientra a *schedule*

Virtual RunTime – Attesa e Risveglio di un Task – I

- quando la funzione *wake_up* risveglia un task *tw*, deve ricalcolare il *VRT* di *tw*
  naturalmente *wake_up* viene eseguita da un task DIVERSO dal task *tw* risvegliato !
- come prima, si dovrebbe prendere il min *VRT* tra quelli di *tw* e della testa di *RB*
- però *wake_up* deve evitare che *tw.VRT* diminuisca troppo e che di conseguenza il task *tw* sia troppo favorito, come può accadere se esso ha atteso molto a lungo
- ecco come (ri)calcolare il *VRT* di un task *tw* risvegliato (due formule *concorrenti* !):

sta in *wake_up*

$tw.VRT = max\ (tw.VRT,\ VMIN - LT\ /\ 2)$ // *LT* è la latenza (param. *SYSCTL*)

sta in *tick*

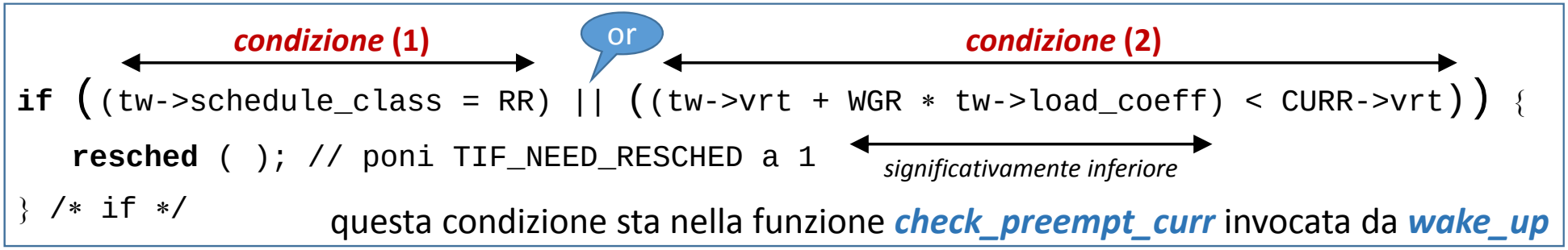
formula provvisoria

$VMIN = max\ (VMIN,\ min\ (CURR.VRT,\ LFT.VRT))$ // formula **definitiva** di *VMIN*

- il task *tw* risvegliato parte con un valore di *VRT* che lo candida all'esecuzione nel prossimo futuro, ma senza che gli sia dato credito eccessivo rispetto a tutti gli altri
- per la nuova formula *VMIN* non può decrescere, come è logicamente necessario
- tipicamente, se il task ha fatta un'attesa molto breve gli viene lasciato il suo *VRT*

Virtual RunTime – Attesa e Risveglio di un Task – II

- risvegliando un task *tw*, si richiede lo *scheduling* se la condizione (1) o (2) è verificata:
  1. il task risvegliato è in una classe di *scheduling* con diritto di esecuzione *maggiore*
  2. il *VRT* del task risvegliato è (*significativamente*) *inferiore* al *VRT* del task corrente
- la condizione (2) ha un coefficiente correttivo detto *WGR* (*wakeup granularity*) affinché un task con attese brevissime non possa causare *context switch* troppo frequenti
- ecco la condizione completa di *preemption* che va valutata, con il coefficiente *WGR*:



- per default il coefficiente *WGR* (è un parametro di configurazione *SYSCTL*) vale 1 *ms*

Esempi 3 e 4 – Task con Attesa e Risveglio

tempi in *ms* – *LT* e *GR* hanno valori di default (*PER* = 6)

ATTIVITÀ E TEMPO	RUNQUEUE		TASK			
			L0 = 1/3	*t1* LOAD = L0	*t2* LOAD = L0	*t3* LOAD = L0
INIT 0	NRT	3	LC	1 / 3	1 / 3	1 / 3
	PER	6	VRTC	1	1	1
	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 0,7	NRT		LC	1 / 3		
	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_event t1.schedule csw t1 → t2 t2.exe 3,2	NRT	**2**	LC	ATTESA (non è in *runqueue*)	**1 / 2**	**1 / 2**
	PER	6	VRTC		1	1
	RQL / L0	**2**	Q		**3**	**3**
	CURR	t2	SUM		32,5	
	RB	t3	DELTA		2,5	
	VMIN	1001	VRT		1003	
t2.wake_up t1 condizione di *check_pr* vera t2.resched csw t2 → t1	NRT	**3**	LC	**1 / 3**	**1 / 3**	**1 / 3**
	PER	6	VRTC	1	1	1
	RQL / L0	**3**	Q	**2**	**2**	**2**
	CURR	t1	SUM	50,7		
	RB	t3, t2	DELTA	0		
	VMIN	1001	VRT	100,7		

Esempio 3 (tab. 4) – il task *t1* va in attesa dopo 0,7 *ms* di esecuzione e si risveglia dopo 2,5 *ms* di attesa (in rosso le variazioni di parametri)

ATTIVITÀ E TEMPO	RUNQUEUE		TASK			
			L0 = 1/3	*t1* LOAD = L0	*t2* LOAD = L0	*t3* LOAD = L0
INIT 0	NRT	3	LC	1 / 3	1 / 3	1 / 3
	PER	6	VRTC	1	1	1
	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 1	NRT		LC	1 / 3		
	PER		VRTC	1		
	RQL / L0		Q	2		
	CURR	t1	SUM	51		
	RB	t2, t3	DELTA	1		
	VMIN	1000,5	VRT	1001		
t1.wait_event t1.schedule csw t1 → t2 t2.exe 1,6	NRT	**2**	LC	ATTESA (non è in *runqueue*)	**1 / 2**	**1 / 2**
	PER	6	VRTC		1	1
	RQL / L0	**2**	Q		**3**	**3**
	CURR	t2	SUM		30,6	
	RB	t3	DELTA		0,6	
	VMIN	1001	VRT		1001,1	
t2.wake_up t1 condizione di *check_pr* falsa t2 prosegue senza csw	NRT	**3**	LC	**1 / 3**	**1 / 3**	**1 / 3**
	PER	6	VRTC	1	1	1
	RQL/L0	**3**	Q	**2**	**2**	**2**
	CURR	t2	SUM		30,6	
	RB	t3, t1	DELTA		0,6	
	VMIN	1001	VRT		1001,1	

Esempio 4 (tab. 5) – il task *t1* va in attesa dopo 1 *ms* ma si risveglia dopo solo 0,6 *ms* di attesa (in rosa le differenze rispetto all'esempio 3)

Virtual RunTime – Creazione e Terminazione di un Task

- se un task termina (*sys_exit*), occorre rifare immediatamente lo *scheduling*
- se un task *tnew* viene creato (*sys_clone*), occorre inizializzare il suo *VRT*, così:

$$tnew.VRT = VMIN + tnew.Q \times tnew.VRTC$$ // inizializzazione – in ***sys_clone***

- il nuovo task *tnew* parte con un valore di *VRT* circa allineato a quelli degli altri task
- la condizione completa di *preemption* è la stessa valutata per il risveglio del task:

```
if ((tw->schedule_class = RR) || ((tw->vrt + WGR * tw->load_coeff) < CURR->vrt)) {
   resched ( ); // poni TIF_NEED_RESCHED a 1
} /* if */
```

questa condizione sta nella funzione ***check_preempt_curr*** invocata da ***sys_clone***

- però a differenza di ciò che succede risvegliando un task, il *VRT* iniziale del nuovo task creato non è tale da posizionare il task all'inizio della coda *RB*
- tuttavia il nuovo task creato è posizionato in *RB* in modo da andare certamente in esecuzione durante il periodo di *scheduling PER* che inizia con la sua creazione

APPROFONDIMENTO

Assegnamento del Peso a un Task

- l'assegnamento di un peso a un task si basa sul parametro *nice_value* del task
- l'utente può assegnare un *nice_value* diverso a ciascun task (comando *nice* di *shell*)
- il *nice_value* va da -20 (max priorità, poca gentilezza – valori negativi assegnabili solo dall'amministratore) a $+19$ (min priorità, molta gentilezza), con valore di default pari a 0
- a pari priorità statica e politica di *scheduling*, un task con *nice_value* maggiore ottiene in proporzione meno tempo di *CPU* di uno con *nice_value* minore
- il *nice_value* di un task *t* viene convertito nel peso del task *t*, cioè in *t.LOAD*
- la regola di conversione è approssimata dalla formula esponenziale seguente:

$t.LOAD = 1024 / 1{,}25^{\,nice\_value}$ // assegna peso – in ***sys_clone*** o in ***(re)nice***

- ma la regola effettiva usata dal *SO* Linux non ha forma analitica, bensì tabulare
- l'amministratore può cambiare dinamicamente il *nice_value* (comando *renice* di *shell*)

APPROFONDIMENTO