# Lo Scheduler

un algoritmo *CONCORRENTE*

# Funzioni Generali dello *Scheduler* – I

*definizione di scheduler*

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

*comportamento dello scheduler*

- lo *scheduler* ha un comportamento orientato a garantire le condizioni seguenti
  - i *task* più importanti vengano eseguiti prima di quelli meno importanti
  - i *task* di pari importanza vengano eseguiti in maniera equa, cioè senza privilegi
  - nessun *task* attenda un turno di esecuzione molto più a lungo degli altri *task*
- lo *scheduler* è un componente critico per il buon funzionamento di un *SO*
- la ricerca per migliorare proprietà e prestazioni dello *scheduler* è costante

# Funzioni Generali dello *Scheduler* – II

## *politica di scheduling equa*

- per un sistema di tipo 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 ne toglie un altro dall'esecuzione
- il *task* da (ri)mettere in esecuzione è scelto tra tutti quelli in stato di *PRONTO* presenti nel sistema, cioè tra tutti quelli presenti nella *runqueue*
- il *task* scelto è quello che in quel momento ha il ***diritto di esecuzione*** maggiore

# Funzioni Generali dello *Scheduler* – III

*di base ci sono tre casi dove lo scheduler deve scegliere un task corrente*

1. quando un *task* si *autosospende* e va in stato di *ATTESA* (o quando *termina*), poiché 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 l'insieme dei *task* in stato di *PRONTO* viene ampliato, 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 sostituendolo con un altro task

   ATTENZIONE: non necessariamente sostituendolo con il task appena risvegliato !

   ma si ricordi che l'attuazione di *preemption* richiede *commutazione di contesto* (*context switch*) ed è differita fino al ritorno a modo U !
3. quando *scade* il *quanto di tempo* (*timeslice*) assegnato al *task corrente*
   (vale se i task sono gestiti con la politica di scheduling *round robin*)

# *Task* e Requisiti di Schedulazione – I

- i *task* possono avere requisiti di *scheduling* molto diversificati per applicazione
- a un primo esame si possono distinguere i *task* nelle tre categorie seguenti
- *task* ***real-time*** (in ***senso stretto*** o ***hard real-time***): devono soddisfare vincoli di tempo stringenti e dunque vanno (ri)messi in esecuzione con grande rapidità
- *task* ***semi-real-time*** (***soft real-time***): possono reagire con una discreta rapidità, ma non garantiscono di non superare un ritardo massimo fissato
- *task* ***normali***: tutti gli altri *task*, divisi in queste due sotto-categorie principali

  ***task I/O bound*** *(vincolati allo I/O*): si autosospendono frequentemente poiché hanno bisogno di dati di *I/O*, come per esempio un programma di *videoscrittura* (*text editor*)

  ***task CPU bound*** (*vincolati alla CPU*): 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* (`struct task_struct`) il campo seguente contiene un puntatore alla struttura della classe di *scheduling* deputata a gestirlo

```
constant 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 si devono rivolgere allo *scheduler* per 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 fair di CFS e sua inizializzazione
static const struct sched_class fair_sched_class = {
   // nome funzione    = punt. a funzione in versione fair
   .next               = &idle_sched_class
   .enqueue_task       = enqueue_task_fair
   .dequeue_task       = dequeue_task_fair
   .check_preempt_curr = check_preempt_wakeup  // vedi dopo
   .pick_next_task     = pick_next_task_fair   // vedi dopo
   .put_prev_task      = put_prev_task_fair
   .set_curr_task      = set_curr_task_fair
   .task_tick          = task_tick_fair        // vedi dopo
   .task_new           = task_new_fair         // vedi dopo
} /* sched_class */
```

**NOTA BENE**

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

- per (ri)mettere un *task t* in stato di *PRONTO*, cioè per chiamare la funzione *enqueue* e inserire *t* in *runqueue*, il *SO* invoca *t* → *sched_class* → *enqueue_task* e così esegue la funz. *enqueue_task_fair*
- nota: è piuttosto facile aggiungere al *SO* nuove classi di *scheduling* o aggiornare le classi 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 di classe *NORMAL*,
  ma danno sempre precedenza a tutti quelli di classe *FIFO*

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

# Pseudo-codice di *schedule*

se occorre toglie da *runqueue* il task corrente, sceglie il *prossimo task corrente* da *runqueue* tramite *pick_next_task*, e se occorre *commuta contesto* tramite macro *context_switch*

```
schedule (  ) { // funz. di scheduling: sceglie il prox task e commuta contesto
   ...
   struct tsk_struct * prev, next
   prev = CURR  // punta a task corr.
   if (prev->stato == ATTESA || prev->stato == TERMINATO) {
      // 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
      CURR->START = NOW            // istante corrente salvato per prox tick
   } /* if */ // altrimenti il task corr. resta in esec. per un altro turno
   TIF_NEED_RESCHED = 0  // la richiesta di scheduling è stata servita
} /* schedule */
```

var iabili locali

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

*scheduling* della classe *FIFO, RR* e *NORMAL*

qui c'è un *return* (istruzione *RET*) che (ri)porta il flusso di esecuzione al nuovo task corrente scelto

se occorre effettua la commutazione di contesto tramite la macro apposita

# Pseudo-codice di *pick_next_task*

scandisce le classi di *scheduling* nell'ordine di importanza *FIFO*, *RR* e *NORMAL*,
e sceglie il prox *task* corrente tramite la funzione *pick_next_task* *specifica della classe*

```
pick_next_task (rq, prev ) { // funzione di selezione del prossimo task corrente
   ...
   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 questi tre modi:
   // - al task PRONTO che ha diritto di esecuzione max, se ci sono task PRONTI
   // - al task prev (cioè CURR), se non ce ne sono e prev non si è autosospeso
   // - al task IDLE, se nessuno dei due casi precedenti risulta praticabile
} /* pick_next_task */
```

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

var iabile locale

*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 alla creazione del *task* e poi non varia più
  solitamente un *task* clonato (figlio) eredita la priorità statica del *task* padre
  però si può cambiare la priorità statica di un *task* (con comandi di *admin*)
- 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, cioè esegue *wait_event*, *sys_wait* o *sys_nanosleep*
  fino alla terminazione naturale, cioè esegue *sys_exit* o *sys_exit_group*
- 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 *i* sono eseguiti in *round robin*
  ciascun *task* viene eseguito per un *quanto di tempo* a turno circolarmente
- per ciascun livello di priorità *i* esiste una coda di *task* gestita in *round robin*
- i task in coda *i*-esima vanno in esecuzione solo se la coda $(i + 1)$-esima è vuota

# *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 coda di *task* da gestire in modo *CFS* mirando all'obiettivo ideale indicato sopra

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

- lo *scheduler CFS* affronta questi tre problemi fondamentali per il suo obiettivo
  1. *determinare ragionevolmente la durata del quanto di tempo* (fissa o variabile)

     un quanto lungo riduce la responsività del sistema agli eventi esterni (p. es. I/O)

     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 *round robin* per gestire i *task uniformemente*
- *CFS* aggiunge a questa base certi raffinamenti semplici, rapidi da calcolare ed efficaci, per considerare le caratteristiche *individuali* di ciascun *task*

# Meccanismo Base di *CFS* – Durata del Quanto

- si chiama *runqueue* l'insieme dei *task PRONTI* e del *task corrente* (in esecuzione)
- il *numero di task* nella *runqueue* a un certo istante è denotato con $NRT \geq 1$

  *NRT* (*Number of Runnable Tasks*) = num. di *task PRONTI* (ma non in esec.) + 1 (il *task CURR*)
- a ogni *task* si assegna un *peso*, detto *LOAD*, che quantifica l'importanza del *task*
- ecco le ipotesi semplificatrici di partenza per illustrare il meccanismo base di *CFS*

  *t.LOAD = 1* per ogni *task t* presente nella *runqueue* (ossia *i task PRONTI* e *il task corrente*)

  nessun *task* si autosospende o termina (non chiama *wait_event*, *sys_wait*, *sys_nanosleep* o *sys_exit*)
- si stabilisce un *periodo di scheduling* $PER > 0$ durante cui tutti i *task* presenti nella *runqueue* verranno eseguiti, in un qualche ordine
- a ogni *task* si assegna un *uguale quanto di tempo* $Q > 0$, di durata calcolata così

$$Q = PER / NRT$$

e ovviamente vale $PER = \sum_{\forall\, task \text{ in } runqueue} Q = NRT \times Q$

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

- tutti i *task* in stato di *PRONTO* (tranne *CURR*) sono tenuti in una *coda* chiamata *RB*
- il funzionamento generale dello *scheduler CFS* è riassumibile così (passi da 1 a 4)
  1. il *task* in *testa* a *RB* viene estratto e diventa *t*ask corrente (*CURR*)
  2. il *task CURR* viene eseguito fino a quando *scade* il suo quanto *Q*
  3. il *task CURR* viene tolto da esecuzione e reinserito in *fondo* a *RB*
  4. si torna al passo 1
- in pratica ogni *task* è eseguito a turno per esattamente *PER / NRT ms* (millisecondi)
- il periodo di *scheduling PER* è una *finestra scorrevole* (*sliding window*) 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 *scheduling*
- 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 (finestra) di *scheduling PER* varia *dinamicamente* con il crescere o diminuire del numero *NRT* di *task* 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* | durata minima del quanto *Q* (se quanti tutti uguali) |

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

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

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

# Meccanismo Completo di *CFS* – valutare il singolo *Task*

- il meccanismo base si fonda su un quanto *Q uguale* e sulla politica *round robin*
- si tratta di un punto di partenza per realizzare uno *scheduler equo* (*fair*) effettivo
- il meccanismo effettivo deve considerare due aspetti di *equità* (*fairness*) rilevanti
  - 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*
- dunque il meccanismo completo di *CFS* è 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
  - a ogni impulso del *real-time clock*, cioè nella funzione *task_tick* dello *scheduler*
  - a ogni risveglio del *task*, cioè nella funzione *wake_up* (e sue varianti) del nucleo
  - alla creazione del *task*, per inizializzarle, cioè insieme al servizio di *SO sys_clone*
- la decisione su quale *task* (ri)mettere in esecuzione viene poi presa dalla funzione *schedule* quando viene chiamata, nelle circostanze già viste studiando il nucleo

# 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*, così

  $t.LOAD$ — peso dello specifico *task t* (ereditato dal padre o assegnato da *admin*)

  $RQL = \sum_{\forall task\ t\ \text{in}\ runqueue} t.LOAD$ — somma dei pesi di tutti i *task* nella *runqueue* (*rqload*) – è $> 0$

  $t.LC = t.LOAD / RQL$ — coeff. di peso: rapporto tra il peso del *task t* e *RQL* (*load_coeff*)
- la durata del quanto di tempo *Q* di uno specifico *task t*, denotata con *t.Q*, dipende da *t* ed è proporzionale al rapporto tra il peso di *t* e il peso della *runqueue*, così

  $t.Q = PER \times t.LC$ e ovviamente vale $\sum_{\forall task\ t\ \text{in}\ runqueue} t.Q = PER$
- se tutti i *task* in runqueue hanno lo stesso peso *LOAD*, si riottiene il quanto *PER / NRT* uguale per tutti i *task* illustrato prima in tale ipotesi, poiché si ha

  $RQL = NRT \times LOAD, \quad LC = LOAD / RQL = 1 / NRT \quad \text{e} \quad Q = PER \times LC = PER / NRT$

# *Virtual RunTime* – Concetto

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

*VRT è una misura virtuale del tempo di esecuzione consumato da un task, basata sulla modifica del tempo reale tramite un coefficiente di correzione*

- 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 PRONTI* (il *task CURR* non si trova in *RB*)
- c'è sempre un *task* corrente, mentre la coda *RB* può essere vuota (ma 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

- ecco la *forma base* dell'algoritmo di (ri)calcolo del *Virtual Time VRT* di un *task*

| ***variabili ausiliarie*** | ***significato*** |
|---|---|
| *SUM* | *tempo REALE totale (dalla clonazione) di esec. del task* |
| *DELTA* | *tempo REALE dal tick precedente di esecuzione del task* |
| $VRTC = 1 / LOAD$ | *coefficiente di correzione di VRT* (*vrt_coeff*) |

stanno nella funzione *task_tick_fair*

| ***(ri)calcolo del Virtual RunTime VRT*** | ***significato*** |
|---|---|
| $SUM = SUM + DELTA$ | *tempo REALE totale (da clonazione) di esec. del task* |
| $VRT = VRT + \underbrace{DELTA \times VRTC}_{\Delta\, VRT}$ | *tempo VIRTUALE (corretto con VRTC) di esecuzione* |

incremento di *VRT* per ogni *tick* del *real-time clock*

- 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* usa tutto il quanto *Q* assegnatogli, cioè se vale $\Delta\, VRT = Q \times VRTC$, in un periodo *PER* i *VRT* di tutti i *task* crescono di una stessa quantità

  $$\Delta\, VRT = Q \times VRTC = (PER \times LC) \times (1 \,/\, LOAD) = PER \times (LOAD \,/\, RQL) \times (1 \,/\, LOAD) = PER \,/\, RQL$$

- in tale caso l'incremento $\Delta\, VRT$ del *virtual runtime* non dipende dal peso (*LOAD*) del *task*
- il vantaggio di un quanto *Q* più lungo è compensato da una crescita di *VRT* più lenta

*se $\Delta$ 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 !*

- dunque 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* in stato di *PRONTO* *non varia* nel tempo, cioè il parametro *NRT* è costante
  i *task* consumano l'intero quanto di tempo senza interruzione, cioè *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)

# 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 è una struttura dati che logicamente opera come una coda ordinata, ma in realtà è organizzata come un albero per l'efficienza dell'inserimento ordinato*

# *Virtual RunTime* – Tempo Virtuale Minimo

- insieme al *VRT* del *task*, lo *scheduler* (ri)calcola una variabile *VMIN* (*vrtmin*) che rappresenta il *VRT* di valore *minimo* tra i *VRT* di tutti i *task* 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*
- poiché rappresenta un tempo, la variabile *VMIN* deve crescere monotonicamente
- ecco una versione *provvisoria* della formula per (ri)calcolare *VMIN*, assai intuitiva

sta in *task_tick*

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

- poiché 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 LFT* in testa a *RB*
- ma la formula sopra tratta *erroneamente* certi casi di risveglio, e andrà *emendata*

*APPROFONDIMENTO*

# Funzione *task_tick_fair* di *CFS* con *Virtual RunTime*

variabili globali e campi del descrittore di *task*

```
// NOW istante di tempo corrente, START istante precedente salvato
// PREV tempo reale di esec. quando il task è diventato corrente
// SUM e VRT tempi reale e virtuale di esecuzione del task
task_tick_fair ( ) { // (ri)calcolo di SUM e VRT
```

var loc

```
   // CFS – (ri)calcolo dei parametri di CURR
   // DELTA tempo passato dal tick precedente
   DELTA = NOW – CURR->START  // durata tick
   CURR->SUM = CURR->SUM + DELTA
   CURR->VRT = CURR->VRT + DELTA * CURR->VRTC
   CURR->START = NOW  // salva per prox tick
```

(ri)calcolo dei parametri del *virtual runtime* del *task CURR*

periodo *task_tick* ≈ 0,01 *ms*

da completare e chiarire più avanti

```
   // 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 ( )
} /* task_tick_fair */
```

poni *TIF_NEED_RESCHED* a 1

per il calcolo di *PREV* vedi *pick_next_task_fair*

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*, ossia tra due *tick* consecutivi (*DELTA* può valere meno di un quanto)

però alla fine del quanto il *VRT* del *task* accumula la durata (virtuale) del quanto

# Esempio 1 – *Task* senza Attesa

| CONDIZIONI INIZIALI | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| RUNQUEUE - | NRT | \|\| | PER | \|\| | RQL | \|\| | CURR | \|\| | VMIN | | |
| | 3 | \|\| | 6,00 | \|\| | 3,00 | \|\| | t1 | \|\| | 100,00 | | |
| TASKS: | ID | \|\| | LOAD | \|\| | LC | \|\| | Q | \|\| | VRTC | \|\| | SUM | \|\| | VRT |
| CURRENT | t1 | \|\| | 1 | \|\| | | \|\| | | \|\| | | \|\| | 10,00 | \|\| | 100,00 |
| RB TREE | t2 | \|\| | 1 | \|\| | | \|\| | | \|\| | | \|\| | 20,00 | \|\| | 100,50 |
| | t3 | \|\| | 1 | \|\| | | \|\| | | \|\| | | \|\| | 30,00 | \|\| | 101,00 |

| CONDIZIONI INIZIALI | ********** | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| RUNQUEUE - | NRT | \|\| | PER | \|\| | RQL | \|\| | CURR | \|\| | VMIN | | |
| | 3 | \|\| | 6,00 | \|\| | 3,00 | \|\| | t1 | \|\| | 100,00 | | |
| TASKS: | ID | \|\| | LOAD | \|\| | LC | \|\| | Q | \|\| | VRTC | \|\| | SUM | \|\| | VRT |
| CURRENT | t1 | \|\| | 1.0 | \|\| | 0,33 | \|\| | 2,00 | \|\| | 1,00 | \|\| | 10,00 | \|\| | 100,00 |
| RB TREE | t2 | \|\| | 1.0 | \|\| | 0,33 | \|\| | 2,00 | \|\| | 1,00 | \|\| | 20,00 | \|\| | 100,50 |
| | t3 | \|\| | 1.0 | \|\| | 0,33 | \|\| | 2,00 | \|\| | 1,00 | \|\| | 30,00 | \|\| | 101,00 |

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

# Esempio 1 – *Task* senza Attesa – Simulazione (1 evento)

| CONDIZIONI INIZIALI | ********** | | | | | |
|---|---|---|---|---|---|---|
| RUNQUEUE - NRT | PER | RQL | CURR | VMIN | | |
| 3 | 6,00 | 3,00 | t1 | 100,00 | | |
| TASKS: ID | LOAD | LC | Q | VRTC | SUM | VRT |
| CURRENT t1 | 1.0 | 0,33 | 2,00 | 1,00 | 10,00 | 100,00 |
| RB TREE t2 | 1.0 | 0,33 | 2,00 | 1,00 | 20,00 | 100,50 |
| t3 | 1.0 | 0,33 | 2,00 | 1,00 | 30,00 | 101,00 |

1)

| EVENT ********** | TIME | TYPE | CONTEXT | RESCHEDULE | | |
|---|---|---|---|---|---|---|
| | 2,00 | Q_SCADE | t1 | true | | |
| RUNQUEUE - NRT | PER | RQL | CURR | VMIN | | |
| 3 | 6,00 | 3,00 | t2 | 100,50 | | |
| TASKS: ID | LOAD | LC | Q | VRTC | SUM | VRT |
| CURRENT t2 | 1.0 | 0,33 | 2,00 | 1,00 | 20,00 | 100,50 |
| RB TREE t3 | 1.0 | 0,33 | 2,00 | 1,00 | 30,00 | 101,00 |
| t1 | 1.0 | 0,33 | 2,00 | 1,00 | 12,00 | 102,00 |

# Esempio 1 – *Task* senza Attesa – Simulazione (4 eventi)

CONDIZIONI INIZIALI **********

| RUNQUEUE - NRT | PER | RQL | CURR | VMIN |
|---|---|---|---|---|
| 3 | 6,00 | 3,00 | t1 | 100,00 |

| TASKS: | ID | LOAD | LC | Q | VRTC | SUM | VRT |
|---|---|---|---|---|---|---|---|
| CURRENT | t1 | 1.0 | 0,33 | 2,00 | 1,00 | 10,00 | 100,00 |
| RB TREE | t2 | 1.0 | 0,33 | 2,00 | 1,00 | 20,00 | 100,50 |
| | t3 | 1.0 | 0,33 | 2,00 | 1,00 | 30,00 | 101,00 |

1)

| EVENT ********** | TIME | TYPE | CONTEXT | RESCHEDULE |
|---|---|---|---|---|
| | 2,00 | Q_SCADE | t1 | true |

| RUNQUEUE - NRT | PER | RQL | CURR | VMIN |
|---|---|---|---|---|
| 3 | 6,00 | 3,00 | t2 | 100,50 |

| TASKS: | ID | LOAD | LC | Q | VRTC | SUM | VRT |
|---|---|---|---|---|---|---|---|
| CURRENT | t2 | 1.0 | 0,33 | 2,00 | 1,00 | 20,00 | 100,50 |
| RB TREE | t3 | 1.0 | 0,33 | 2,00 | 1,00 | 30,00 | 101,00 |
| | t1 | 1.0 | 0,33 | 2,00 | 1,00 | 12,00 | 102,00 |

2)

| EVENT ********** | TIME | TYPE | CONTEXT | RESCHEDULE |
|---|---|---|---|---|
| | 4,00 | Q_SCADE | t2 | true |

| RUNQUEUE - NRT | PER | RQL | CURR | VMIN |
|---|---|---|---|---|
| 3 | 6,00 | 3,00 | t3 | 101,00 |

| TASKS: | ID | LOAD | LC | Q | VRTC | SUM | VRT |
|---|---|---|---|---|---|---|---|
| CURRENT | t3 | 1.0 | 0,33 | 2,00 | 1,00 | 30,00 | 101,00 |
| RB TREE | t1 | 1.0 | 0,33 | 2,00 | 1,00 | 12,00 | 102,00 |
| | t2 | 1.0 | 0,33 | 2,00 | 1,00 | 22,00 | 102,50 |

3)

| EVENT ********** | TIME | TYPE | CONTEXT | RESCHEDULE |
|---|---|---|---|---|
| | 6,00 | Q_SCADE | t3 | true |

| RUNQUEUE - NRT | PER | RQL | CURR | VMIN |
|---|---|---|---|---|
| 3 | 6,00 | 3,00 | t1 | 102,00 |

| TASKS: | ID | LOAD | LC | Q | VRTC | SUM | VRT |
|---|---|---|---|---|---|---|---|
| CURRENT | t1 | 1.0 | 0,33 | 2,00 | 1,00 | 12,00 | 102,00 |
| RB TREE | t2 | 1.0 | 0,33 | 2,00 | 1,00 | 22,00 | 102,50 |
| | t3 | 1.0 | 0,33 | 2,00 | 1,00 | 32,00 | 103,00 |

4)

| EVENT ********** | TIME | TYPE | CONTEXT | RESCHEDULE |
|---|---|---|---|---|
| | 8,00 | Q_SCADE | t1 | true |

| RUNQUEUE - NRT | PER | RQL | CURR | VMIN |
|---|---|---|---|---|
| 3 | 6,00 | 3,00 | t2 | 102,50 |

| TASKS: | ID | LOAD | LC | Q | VRTC | SUM | VRT |
|---|---|---|---|---|---|---|---|
| CURRENT | t2 | 1.0 | 0,33 | 2,00 | 1,00 | 22,00 | 102,50 |
| RB TREE | t3 | 1.0 | 0,33 | 2,00 | 1,00 | 32,00 | 103,00 |
| | t1 | 1.0 | 0,33 | 2,00 | 1,00 | 14,00 | 104,00 |

# Esempio 2 – *Task* senza Attesa ma con Pesi Differenti

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

$RQL = 1.0 + 1.5 + 0.5 = 3{,}00$

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

i pesi assegnati squilibrano l'esecuzione dei *task*: il quanto di *t*2 è triplo rispetto a *t3*

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

però al *task* più leggero (con *LOAD* minimo) è 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)

| CONDIZIONI INIZIALI | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| RUNQUEUE - | NRT | PER | RQL | CURR | VMIN | | | |
| | 3 | 6,00 | 3,00 | t1 | 100,00 | | | |
| TASKS: | ID | LOAD | LC | Q | VRTC | SUM | VRT | |
| CURRENT | t1 | 1.0 | | | | 10,00 | 100,00 | |
| RB TREE | t2 | 1.5 | | | | 20,00 | 100,50 | |
| | t3 | 0.5 | | | | 30,00 | 101,00 | |

| CONDIZIONI INIZIALI | ********** | | | | | | |
|---|---|---|---|---|---|---|---|
| RUNQUEUE - | NRT | PER | RQL | CURR | VMIN | | |
| | 3 | 6,00 | 3,00 | t1 | 100,00 | | |
| TASKS: | ID | LOAD | LC | Q | VRTC | SUM | VRT |
| CURRENT | t1 | 1.0 | 0,33 | 2,00 | 1,00 | 10,00 | 100,00 |
| RB TREE | t2 | 1.5 | 0,50 | 3,00 | 0,67 | 20,00 | 100,50 |
| | t3 | 0.5 | 0,17 | 1,00 | 2,00 | 30,00 | 101,00 |

# Esempio 2 – *Task* senza Attesa ma con Pesi Differenti – Simulazione

| CONDIZIONI INIZIALI | ********** | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| RUNQUEUE - NRT | \|\| | PER | \|\| RQL | \|\| | CURR | \|\| VMIN | |
| 3 | \|\| | 6,00 | \|\| 3,00 | \|\| | t1 | \|\| 100,00 | |
| TASKS: | ID | \|\|LOAD | \|\| LC | \|\| | Q | \|\| VRTC | \|\| SUM | \|\| VRT |
| CURRENT | t1 | \|\| 1.0 | \|\| 0,33 | \|\| | 2,00 | \|\| 1,00 | \|\| 10,00 | \|\| 100,00 |
| RB TREE | t2 | \|\| 1.5 | \|\| 0,50 | \|\| | 3,00 | \|\| 0,67 | \|\| 20,00 | \|\| 100,50 |
| | t3 | \|\| 0.5 | \|\| 0,17 | \|\| | 1,00 | \|\| 2,00 | \|\| 30,00 | \|\| 101,00 |

1)

| EVENT | ********** | TIME | \|\| | TYPE | \|\| CONTEXT | \|\| RESCHEDULE |
|---|---|---|---|---|---|---|
| | | 2,00 | \|\| | Q_SCADE | \|\| t1 | \|\| true |
| RUNQUEUE - NRT | \|\| | PER | \|\| RQL | \|\| | CURR | \|\| VMIN | |
| 3 | \|\| | 6,00 | \|\| 3,00 | \|\| | t2 | \|\| 100,50 | |
| TASKS: | ID | \|\|LOAD | \|\| LC | \|\| | Q | \|\| VRTC | \|\| SUM | \|\| VRT |
| CURRENT | t2 | \|\| 1.5 | \|\| 0,50 | \|\| | 3,00 | \|\| 0,67 | \|\| 20,00 | \|\| 100,50 |
| RB TREE | t3 | \|\| 0.5 | \|\| 0,17 | \|\| | 1,00 | \|\| 2,00 | \|\| 30,00 | \|\| 101,00 |
| | t1 | \|\| 1.0 | \|\| 0,33 | \|\| | 2,00 | \|\| 1,00 | \|\| 12,00 | \|\| 102,00 |

2)

| EVENT | ********** | TIME | \|\| | TYPE | \|\| CONTEXT | \|\| RESCHEDULE |
|---|---|---|---|---|---|---|
| | | 5,00 | \|\| | Q_SCADE | \|\| t2 | \|\| true |
| RUNQUEUE - NRT | \|\| | PER | \|\| RQL | \|\| | CURR | \|\| VMIN | |
| 3 | \|\| | 6,00 | \|\| 3,00 | \|\| | t3 | \|\| 101,00 | |
| TASKS: | ID | \|\|LOAD | \|\| LC | \|\| | Q | \|\| VRTC | \|\| SUM | \|\| VRT |
| CURRENT | t3 | \|\| 0.5 | \|\| 0,17 | \|\| | 1,00 | \|\| 2,00 | \|\| 30,00 | \|\| 101,00 |
| RB TREE | t1 | \|\| 1.0 | \|\| 0,33 | \|\| | 2,00 | \|\| 1,00 | \|\| 12,00 | \|\| 102,00 |
| | t2 | \|\| 1.5 | \|\| 0,50 | \|\| | 3,00 | \|\| 0,67 | \|\| 23,00 | \|\| 102,50 |

3)

| EVENT | ********** | TIME | \|\| | TYPE | \|\| CONTEXT | \|\| RESCHEDULE |
|---|---|---|---|---|---|---|
| | | 6,00 | \|\| | Q_SCADE | \|\| t3 | \|\| true |
| RUNQUEUE - NRT | \|\| | PER | \|\| RQL | \|\| | CURR | \|\| VMIN | |
| 3 | \|\| | 6,00 | \|\| 3,00 | \|\| | t1 | \|\| 102,00 | |
| TASKS: | ID | \|\|LOAD | \|\| LC | \|\| | Q | \|\| VRTC | \|\| SUM | \|\| VRT |
| CURRENT | t1 | \|\| 1.0 | \|\| 0,33 | \|\| | 2,00 | \|\| 1,00 | \|\| 12,00 | \|\| 102,00 |
| RB TREE | t2 | \|\| 1.5 | \|\| 0,50 | \|\| | 3,00 | \|\| 0,67 | \|\| 23,00 | \|\| 102,50 |
| | t3 | \|\| 0.5 | \|\| 0,17 | \|\| | 1,00 | \|\| 2,00 | \|\| 31,00 | \|\| 103,00 |

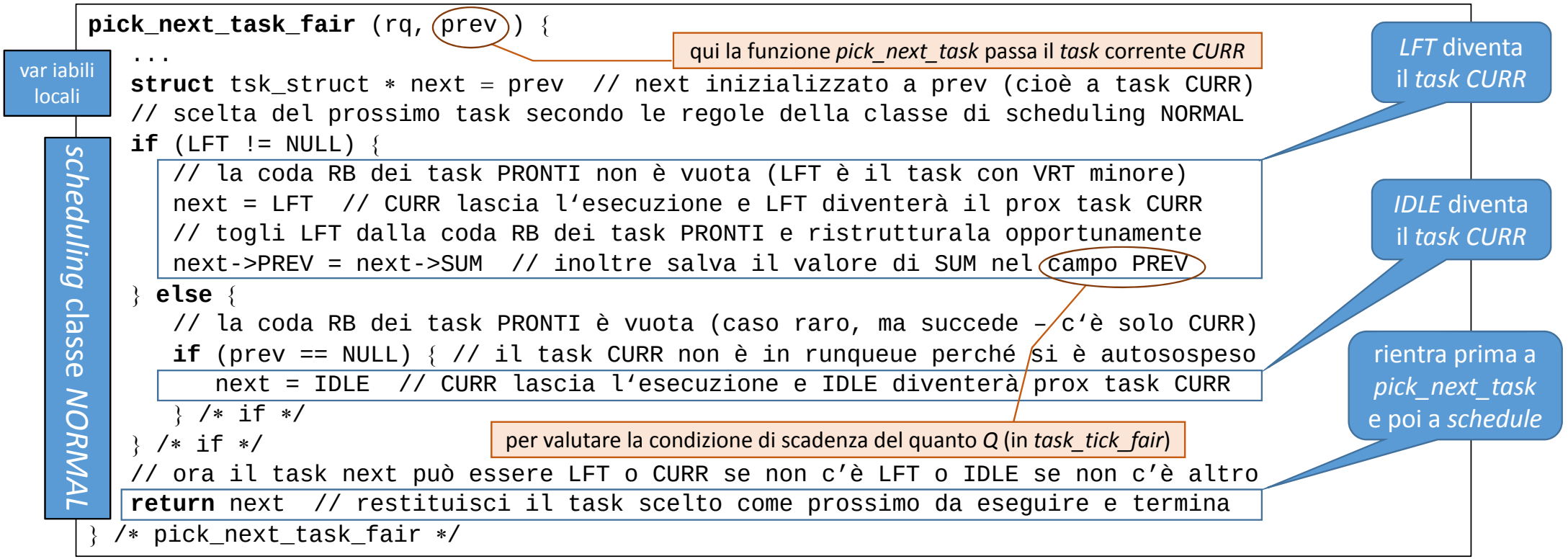
4)

| EVENT | ********** | TIME | \|\| | TYPE | \|\| CONTEXT | \|\| RESCHEDULE |
|---|---|---|---|---|---|---|
| | | 8,00 | \|\| | Q_SCADE | \|\| t1 | \|\| true |
| RUNQUEUE - NRT | \|\| | PER | \|\| RQL | \|\| | CURR | \|\| VMIN | |
| 3 | \|\| | 6,00 | \|\| 3,00 | \|\| | t2 | \|\| 102,50 | |
| TASKS: | ID | \|\|LOAD | \|\| LC | \|\| | Q | \|\| VRTC | \|\| SUM | \|\| VRT |
| CURRENT | t2 | \|\| 1.5 | \|\| 0,50 | \|\| | 3,00 | \|\| 0,67 | \|\| 23,00 | \|\| 102,50 |
| RB TREE | t3 | \|\| 0.5 | \|\| 0,17 | \|\| | 1,00 | \|\| 2,00 | \|\| 31,00 | \|\| 103,00 |
| | t1 | \|\| 1.0 | \|\| 0,33 | \|\| | 2,00 | \|\| 1,00 | \|\| 14,00 | \|\| 104,00 |

# Pseudo-codice di *pick_next_task_fair* (classe *NORMAL*)

seleziona il prossimo *task* da eseguire secondo le regole di *CFS*, basate sull'uso di *VRT*

prende il *task* con *VRT minore*: *LFT,* o *CURR* se non c'è *LFT,* o *IDLE* se non c'è altro



# *Virtual RunTime* – Attesa e Risveglio di un *Task* – I

- quando la funzione *wake_up* risveglia un *task tw*, deve ricalcolare il *VRT* di *tw*

  *wake_up* viene eseguita da un *task* DIVERSO dal *task tw* risvegliato !

  *NRT* cambia ! ricalcola subito *PER*, *RQL*, *LC*, *Q* e *VMIN*

- si dovrebbe prendere il *min VRT* tra quello di *tw* e quello del *task LFT* in testa a *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 *task_tick*

formula provvisoria

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

- così il *task tw* risvegliato parte con un valore di *VRT* che lo candida all'esecuzione nel prossimo futuro, ma senza dargli credito eccessivo rispetto a tutti gli altri *task*
- 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 *WGR* (*wake_up granularity*), cosicché 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*

*condizione* (1)

```
if (tw->schedule_class == classe con diritto di esec. maggiore di SCHED_NORMAL) ||
```

*condizione* (2)

```
   (tw->vrt + WGR * tw->load_coeff) < CURR->vrt {
```

*significativamente inferiore*

or

```
   resched ( )  // poni TIF_NEED_RESCHED a 1
} /* if */
```

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

granularità di *wake_up*, rappresenta una sorta di *quanto minimo* per la *preemption* (per default *WGR* vale 1 *ms*)

**NOTA BENE**: anche quando la condizione di *preemption* è vera, non è detto che il *task tw* risvegliato sarà il prossimo *task CURR*, poiché *tw* potrebbe non diventare subito *task LFT* in coda *RB*.

# Esempio 3 – *Task* con Attesa e Risveglio

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

| CONDIZIONI INIZIALI | | | | | | | |
|---|---|---|---|---|---|---|---|
| RUNQUEUE - NRT | PER | RQL | CURR | VMIN | | | |
| 3 | 6,00 | 3,00 | t1 | 100,00 | | | |
| TASKS: ID | LOAD | LC | Q | VRTC | SUM | VRT | |
| CURRENT t1 | 1 | | | | 10,00 | 100,00 | |
| RB TREE t2 | 1 | | | | 20,00 | 100,50 | |
| t3 | 1 | | | | 30,00 | 101,00 | |
| Events of task t1: WAIT at 1.0 WAKEUP after 5.0 (wakeup is 5.0 ms after wait) | | | | | | | |
| Events of task t3: WAIT at 3.0 WAKEUP after 1.0 (wakeup is 1.0 ms after wait) | | | | | | | |

| CONDIZIONI INIZIALI ********** | | | | | | |
|---|---|---|---|---|---|---|
| RUNQUEUE - NRT | PER | RQL | CURR | VMIN | | |
| 3 | 6,00 | 3,00 | t1 | 100,00 | | |
| TASKS: ID | LOAD | LC | Q | VRTC | SUM | VRT |
| CURRENT t1 | 1.0 | 0,33 | 2,00 | 1,00 | 10,00 | 100,00 |
| RB TREE t2 | 1.0 | 0,33 | 2,00 | 1,00 | 20,00 | 100,50 |
| t3 | 1.0 | 0,33 | 2,00 | 1,00 | 30,00 | 101,00 |
| Events of task t1: WAIT at 1.0; WAKEUP after 5.0; | | | | | | |
| Events of task t3: WAIT at 3.0; WAKEUP after 1.0; | | | | | | |

# Esempio 3 – *Task* con Attesa e Risveglio – Simulazione

| CONDIZIONI INIZIALI | ********** | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| RUNQUEUE - NRT | PER | RQL | CURR | VMIN | | | | |
| 3 | 6,00 | 3,00 | t1 | 100,00 | | | | |
| TASKS: | ID | LOAD | LC | Q | VRTC | SUM | VRT | |
| CURRENT | t1 | 1.0 | 0,33 | 2,00 | 1,00 | 10,00 | 100,00 | |
| RB TREE | t2 | 1.0 | 0,33 | 2,00 | 1,00 | 20,00 | 100,50 | |
| | t3 | 1.0 | 0,33 | 2,00 | 1,00 | 30,00 | 101,00 | |
| Events of task t1: | WAIT at 1.0; | WAKEUP after 5.0; | | | | | | |
| Events of task t3: | WAIT at 3.0; | WAKEUP after 1.0; | | | | | | |

1)

| EVENT | ********** | TIME | TYPE | CONTEXT | RESCHEDULE | | | |
|---|---|---|---|---|---|---|---|---|
| | | 1,00 | WAIT | t1 | true | | | |
| RUNQUEUE - NRT | PER | RQL | CURR | VMIN | | | | |
| 2 | 6,00 | 2,00 | t2 | 100,50 | | | | |
| TASKS: | ID | LOAD | LC | Q | VRTC | SUM | VRT | |
| CURRENT | t2 | 1.0 | 0,50 | 3,00 | 1,00 | 20,00 | 100,50 | |
| RB TREE | t3 | 1.0 | 0,50 | 3,00 | 1,00 | 30,00 | 101,00 | |
| WAITING | t1 | 1.0 | 0,33 | 2,00 | 1,00 | 11,00 | 101,00 | |

2)

| EVENT | ********** | TIME | TYPE | CONTEXT | RESCHEDULE | | | |
|---|---|---|---|---|---|---|---|---|
| | | 4,00 | Q_SCADE | t2 | true | | | |
| RUNQUEUE - NRT | PER | RQL | CURR | VMIN | | | | |
| 2 | 6,00 | 2,00 | t3 | 101,00 | | | | |
| TASKS: | ID | LOAD | LC | Q | VRTC | SUM | VRT | |
| CURRENT | t3 | 1.0 | 0,50 | 3,00 | 1,00 | 30,00 | 101,00 | |
| RB TREE | t2 | 1.0 | 0,50 | 3,00 | 1,00 | 23,00 | 103,50 | |
| WAITING | t1 | 1.0 | 0,33 | 2,00 | 1,00 | 11,00 | 101,00 | |

3)

| EVENT | ********** | TIME | TYPE | CONTEXT | RESCHEDULE | | | |
|---|---|---|---|---|---|---|---|---|
| | | 6,00 | WAKEUP | t3 | true | | | |
| tw.vrt+WGR*tw.LC=101,00+1,00*0,33=101,33 < curr.vrt=103,00 | | | | | | | | |
| RUNQUEUE - NRT | PER | RQL | CURR | VMIN | | | | |
| 3 | 6,00 | 3,00 | t1 | 103,00 | | | | |
| TASKS: | ID | LOAD | LC | Q | VRTC | SUM | VRT | |
| CURRENT | t1 | 1.0 | 0,33 | 2,00 | 1,00 | 11,00 | 101,00 | |
| RB TREE | t3 | 1.0 | 0,33 | 2,00 | 1,00 | 32,00 | 103,00 | |
| | t2 | 1.0 | 0,33 | 2,00 | 1,00 | 23,00 | 103,50 | |

da (iniz) a (1) *NRT* cambiato: $3 \rightarrow 2$
ricalcolati subito *PER* (invariato), *RQL*, *LC*, *Q* e *VMIN*

da (2) a (3) *NRT* cambiato: $2 \rightarrow 3$
ricalcolati subito *PER* (invariato), *RQL*, *LC*, *Q* e *VMIN*

# Esempio 3 – *Task* con Attesa e Risveglio – Simulazione (cont.)

4)

| EVENT ********** | TIME | TYPE | CONTEXT | RESCHEDULE |
|---|---|---|---|---|
| | 8,00 | Q_SCADE | t1 | true |
| RUNQUEUE - NRT | PER | RQL | CURR | VMIN |
| 3 | 6,00 | 3,00 | t3 | 103,00 |
| TASKS: ID | LOAD | LC | Q | VRTC | SUM | VRT |
| CURRENT t3 | 1.0 | 0,33 | 2,00 | 1,00 | 32,00 | 103,00 |
| RB TREE t1 | 1.0 | 0,33 | 2,00 | 1,00 | 13,00 | 103,00 |
| t2 | 1.0 | 0,33 | 2,00 | 1,00 | 23,00 | 103,50 |

5)

| EVENT ********** | TIME | TYPE | CONTEXT | RESCHEDULE |
|---|---|---|---|---|
| | 9,00 | WAIT | t3 | true |
| RUNQUEUE - NRT | PER | RQL | CURR | VMIN |
| 2 | 6,00 | 2,00 | t1 | 103,00 |
| TASKS: ID | LOAD | LC | Q | VRTC | SUM | VRT |
| CURRENT t1 | 1.0 | 0,50 | 3,00 | 1,00 | 13,00 | 103,00 |
| RB TREE t2 | 1.0 | 0,50 | 3,00 | 1,00 | 23,00 | 103,50 |
| WAITING t3 | 1.0 | 0,33 | 2,00 | 1,00 | 33,00 | 104,00 |

da (4) a (5) *NRT* cambiato: $3 \rightarrow 2$
ricalcolati subito *PER* (invariato), *RQL*, *LC*, *Q* e *VMIN* (invariato)

6)

| EVENT ********** | TIME | TYPE | CONTEXT | RESCHEDULE |
|---|---|---|---|---|
| | 10,00 | WAKEUP | t1 | false |
| tw.vrt+WGR*tw.LC = 104,00+1,00*0,33=104,33 < curr.vrt=104,00 | | | | |
| RUNQUEUE - NRT | PER | RQL | CURR | VMIN |
| 3 | 6,00 | 3,00 | t1 | 103,50 |
| TASKS: ID | LOAD | LC | Q | VRTC | SUM | VRT |
| CURRENT t1 | 1.0 | 0,33 | 2,00 | 1,00 | 14,00 | 104,00 |
| RB TREE t2 | 1.0 | 0,33 | 2,00 | 1,00 | 23,00 | 103,50 |
| t3 | 1.0 | 0,33 | 2,00 | 1,00 | 33,00 | 104,00 |

da (5) a (6) *NRT* cambiato: $2 \rightarrow 3$
ricalcolati subito *PER* (invariato), *RQL*, *LC*, *Q* e *VMIN*

7)

| EVENT ********** | TIME | TYPE | CONTEXT | RESCHEDULE |
|---|---|---|---|---|
| | 11,00 | Q_SCADE | t1 | true |
| RUNQUEUE - NRT | PER | RQL | CURR | VMIN |
| 3 | 6,00 | 3,00 | t2 | 103,50 |
| TASKS: ID | LOAD | LC | Q | VRTC | SUM | VRT |
| CURRENT t2 | 1.0 | 0,33 | 2,00 | 1,00 | 23,00 | 103,50 |
| RB TREE t3 | 1.0 | 0,33 | 2,00 | 1,00 | 33,00 | 104,00 |
| t1 | 1.0 | 0,33 | 2,00 | 1,00 | 15,00 | 105,00 |

# Esempio 4 – *Task* con Attesa e Risveglio – Simulazione

variante dell'esempio 3
con tempi un po' diversi

| CONDIZIONI INIZIALI | ********** | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| RUNQUEUE - NRT | \|\| | PER | \|\| | RQL | \|\| | CURR | \|\| | VMIN |
| | | 3 | \|\| | 6,00 | \|\| | 3,00 | \|\| | t1 | \|\| | 100,00 |
| TASKS: | ID | \|\| | LOAD | \|\| | LC | \|\| | Q | \|\| | VRTC | \|\| | SUM | \|\| | VRT |
| CURRENT | t1 | \|\| | 1.0 | \|\| | 0,33 | \|\| | 2,00 | \|\| | 1,00 | \|\| | 10,00 | \|\| | 100,00 |
| RB TREE | t2 | \|\| | 1.0 | \|\| | 0,33 | \|\| | 2,00 | \|\| | 1,00 | \|\| | 20,00 | \|\| | 100,50 |
| | t3 | \|\| | 1.0 | \|\| | 0,33 | \|\| | 2,00 | \|\| | 1,00 | \|\| | 30,00 | \|\| | 101,00 |
| Events of task t1: WAIT at 1.0; WAKEUP after 0.6; | | | | | | | | | | | | | |
| Events of task t3: WAIT at 3.0; WAKEUP after 1.0; | | | | | | | | | | | | | |

1)

| EVENT | ********** | TIME | \|\| | TYPE | \|\| | CONTEXT | \|\| | RESCHEDULE |
|---|---|---|---|---|---|---|---|---|
| | | 1,00 | \|\| | WAIT | \|\| | t1 | \|\| | true |
| RUNQUEUE - NRT | \|\| | PER | \|\| | RQL | \|\| | CURR | \|\| | VMIN |
| | 2 | \|\| | 6,00 | \|\| | 2,00 | \|\| | t2 | \|\| | 100,50 |
| TASKS: | ID | \|\| | LOAD | \|\| | LC | \|\| | Q | \|\| | VRTC | \|\| | SUM | \|\| | VRT |
| CURRENT | t2 | \|\| | 1.0 | \|\| | 0,50 | \|\| | 3,00 | \|\| | 1,00 | \|\| | 20,00 | \|\| | 100,50 |
| RB TREE | t3 | \|\| | 1.0 | \|\| | 0,50 | \|\| | 3,00 | \|\| | 1,00 | \|\| | 30,00 | \|\| | 101,00 |
| | | | | | | | | | | | | | |
| WAITING | t1 | \|\| | 1.0 | \|\| | 0,33 | \|\| | 2,00 | \|\| | 1,00 | \|\| | 11,00 | \|\| | 101,00 |

*NRT* cambiato: 3 → 2
ricalcolati subito ...

2)

| EVENT | ********** | TIME | \|\| | TYPE | \|\| | CONTEXT | \|\| | RESCHEDULE |
|---|---|---|---|---|---|---|---|---|
| | | 1,60 | \|\| | WAKEUP | \|\| | t2 | \|\| | false |
| tw.vrt+WGR*tw.LC=101,00+1,00*0,33=101,33 < curr.vrt=101,10 | | | | | | | | |
| RUNQUEUE - NRT | \|\| | PER | \|\| | RQL | \|\| | CURR | \|\| | VMIN |
| | 3 | \|\| | 6,00 | \|\| | 3,00 | \|\| | t2 | \|\| | 101,00 |
| TASKS: | ID | \|\| | LOAD | \|\| | LC | \|\| | Q | \|\| | VRTC | \|\| | SUM | \|\| | VRT |
| CURRENT | t2 | \|\| | 1.0 | \|\| | 0,33 | \|\| | 2,00 | \|\| | 1,00 | \|\| | 20,60 | \|\| | 101,10 |
| RB TREE | t3 | \|\| | 1.0 | \|\| | 0,33 | \|\| | 2,00 | \|\| | 1,00 | \|\| | 30,00 | \|\| | 101,00 |
| | t1 | \|\| | 1.0 | \|\| | 0,33 | \|\| | 2,00 | \|\| | 1,00 | \|\| | 11,00 | \|\| | 101,00 |

*NRT* cambiato: 2 → 3
ricalcolati subito ...

per il seguito
vedi dispensa online

*APPROFONDIMENTO*

# *Virtual RunTime* – Creazione e Terminazione di un *Task*

*NRT* cambia !
ricalcola subito
*PER, RQL, LC, Q*
e *VMIN*

- se un *task* termina (*sys_exit*), occorre rifare immediatamente lo *scheduling*
- se un *task tnew* viene creato (*sys_clone*), se ne inizializzano *SUM* e *VRT* così

$tnew.SUM = 0$ e $tnew.VRT = VMIN + tnew.Q \times tnew.VRTC$ // in ***sys_clone***

- il nuovo *task tnew* parte con valori di $SUM = 0$ e di *VRT* circa allineato agli altri *task*
- la condizione completa di *preemption* è la stessa da valutare per risvegliare un *task*

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

è $\Delta$ *VRT* nell'ipotesi $DELTA = Q$
*al clone si addebita un quanto a saldo della procedura*

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

- a differenza di quanto può succedere risvegliando un *task*, il *VRT* iniziale del nuovo *task* non è tale da posizionare il *task* in testa alla coda *RB* (cioè come *LFT*)
- 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

# Esempio 6 – Creazione e Terminazione di *Task*– Simulazione

per l'esempio 5
con il task *IDLE*
vedi dispensa online

| CONDIZIONI INIZIALI ********** | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| RUNQUEUE - NRT | PER | RQL | CURR | VMIN | | | | |
| 3 | 6,00 | 3,00 | t1 | 100,00 | | | | |
| TASKS: | ID | LOAD | LC | Q | VRTC | SUM | VRT | |
| CURRENT | t1 | 1.0 | 0,33 | 2,00 | 1,00 | 10,00 | 100,00 | |
| RB TREE | t2 | 1.0 | 0,33 | 2,00 | 1,00 | 20,00 | 100,50 | |
| | t3 | 1.0 | 0,33 | 2,00 | 1,00 | 30,00 | 101,00 | |
| Events of task t1: EXIT at 1.0; | | | | | | | | |
| Events of task t3: CLONE at 1.0; | | | | | | | | |

*NRT* cambiato: 3 → 2
ricalcolati subito
*PER* (invariato), *RQL*, *LC*, *Q* e *VMIN*

1)

| EVENT ********** | TIME | TYPE | CONTEXT | RESCHEDULE |
|---|---|---|---|---|
| | 1,00 | EXIT | t1 | true |
| RUNQUEUE - NRT | PER | RQL | CURR | VMIN |
| 2 | 6,00 | 2,00 | t2 | 100,50 |
| TASKS: | ID | LOAD | LC | Q | VRTC | SUM | VRT |
| CURRENT | t2 | 1.0 | 0,50 | 3,00 | 1,00 | 20,00 | 100,50 |
| RB TREE | t3 | 1.0 | 0,50 | 3,00 | 1,00 | 30,00 | 101,00 |

2)

| EVENT ********** | TIME | TYPE | CONTEXT | RESCHEDULE |
|---|---|---|---|---|
| | 4,00 | Q_SCADE | t2 | true |
| RUNQUEUE - NRT | PER | RQL | CURR | VMIN |
| 2 | 6,00 | 2,00 | t3 | 101,00 |
| TASKS: | ID | LOAD | LC | Q | VRTC | SUM | VRT |
| CURRENT | t3 | 1.0 | 0,50 | 3,00 | 1,00 | 30,00 | 101,00 |
| RB TREE | t2 | 1.0 | 0,50 | 3,00 | 1,00 | 23,00 | 103,50 |

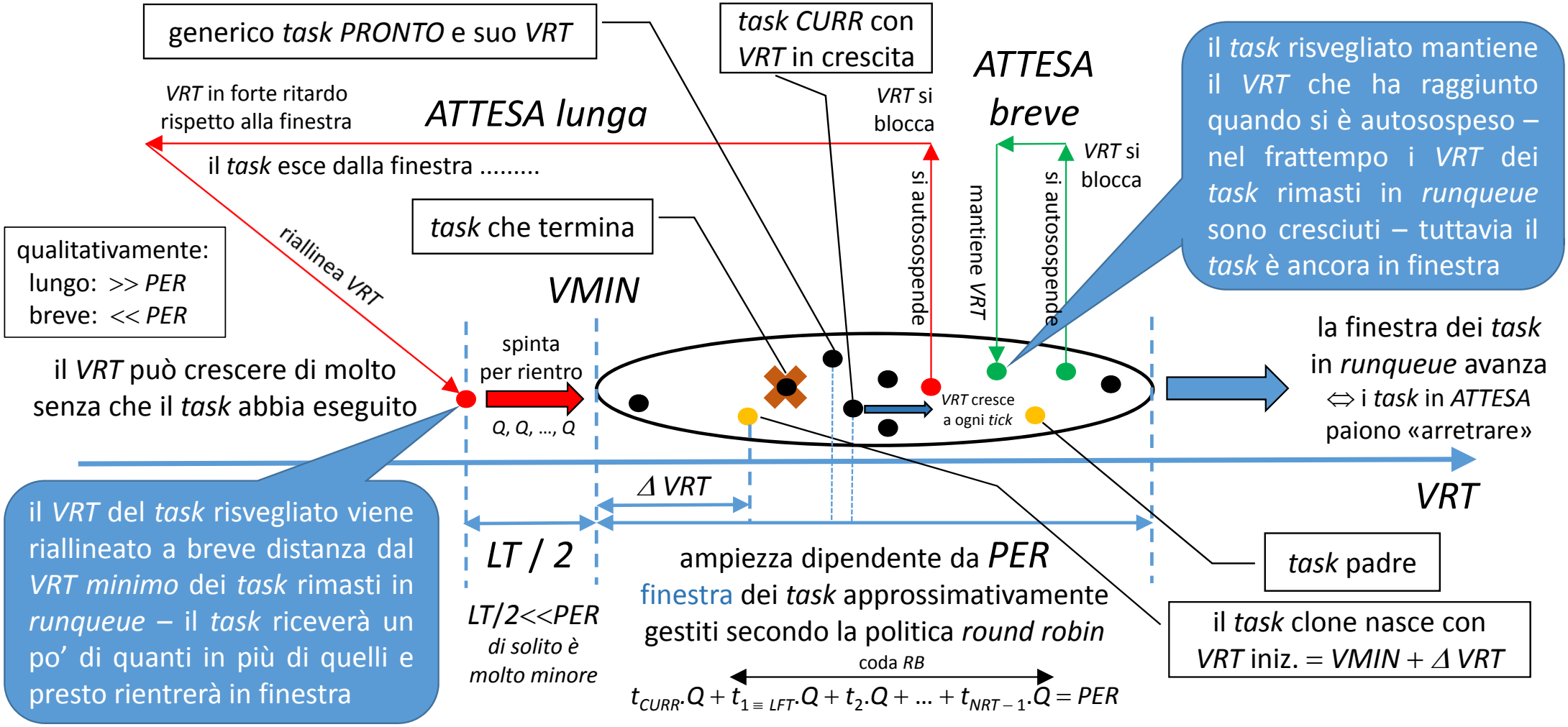
*NRT* cambiato: 2 → 3
ricalcolati subito
*PER* (invariato), *RQL*, *LC*, *Q* e *VMIN*

3)

| EVENT ********** | TIME | TYPE | CONTEXT | RESCHEDULE |
|---|---|---|---|---|
| | 5,00 | CLONE | t3 | false |
| tnew.vrt+WGR*tnew.LC=104.00+1.00*0.33=104.33 < curr.vrt=102.00 | | | | |
| RUNQUEUE - NRT | PER | RQL | CURR | VMIN |
| 3 | 6,00 | 3,00 | t3 | 102,00 |
| TASKS: | ID | LOAD | LC | Q | VRTC | SUM | VRT |
| CURRENT | t3 | 1.0 | 0,33 | 2,00 | 1,00 | 31,00 | 102,00 |
| RB TREE | t2 | 1.0 | 0,33 | 2,00 | 1,00 | 23,00 | 103,50 |
| | t4 | 1.0 | 0,33 | 2,00 | 1,00 | 0,00 | 104,00 |

Il task creato (*tnew*) viene inserito (in ordine di *VRT*) nella coda *RB* con *ID* progressivo *t4*, *SUM* = 0 e *VRT* calcolato secondo la regola vista prima

# *CFS* – *VRT* dall'Attesa al Risveglio e alla Clonazione o Fine



# Assegnamento del Peso a un *Task*

- l'assegnamento di un peso a un *task* si basa sul parametro *nice_value* del *task*
- a pari priorità statica (*static_prio*) e politica di *scheduling*, un *task* con *nice_value* maggiore ottiene meno tempo di *CPU* rispetto a uno con *nice_value* minore
- *nice_value* va da –20 (task poco gentile, pretende molto tempo) a +19 (task molto gentile, ne chiede poco), con valore di default pari a 0 (valori negativi assegnabili solo da admin)
- l'utente può assegnare un *nice_value* diverso a ciascun suo *task* (comando *nice* di *shell*)
- admin può cambiare dinamicamente il *nice_value* di ogni task (comando *renice* di *shell*)
- *CFS* converte il *nice_value* assegnato al *task t* nel peso del *task t*, cioè in *t.LOAD*
- la regola di conversione è approssimata dalla formula esponenziale seguente:

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

- ma la regola effettiva usata da *CFS* è codificata in forma tabulare, per efficienza

# *CFS* – Invarianti (valgono quando la *CPU* è in modo *U*)

(1) la somma dei *coeff. di peso* (*load_coeff*) dei *task* in *runqueue* è *sempre* uguale a uno

$$t_{CURR}.LC + t_{1 \equiv LFT}.LC + t_2.LC + \ldots + t_{NRT-1}.LC = 1$$

coda *RB* (può essere vuota)

(2) la somma dei *quanti* dei *task* in *runqueue* è *sempre* uguale al periodo di *scheduling*

$$t_{CURR}.Q + t_{1 \equiv LFT}.Q + t_2.Q + \ldots + t_{NRT-1}.Q = PER$$

coda *RB* (può essere vuota)

il sistema funziona in *round robin* puro

(3) se il numero $NRT \geq 1$ di *task* in *runqueue* è *costante* e *nessun task si autosospende*, cioè ogni *task* usa l'intero quanto, l'incremento $\Delta$ di *VRT* per ogni turno non dipende dal *task*

$$\Delta\, VRT = PER \,/\, RQL$$

(4) se, oltre a valere (3), tutti i task hanno *lo stesso peso* $LOAD = 1$ *e dunque* $VRTC = 1$, allora

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

ossia l'incremento $\Delta$ *VRT* coincide con l'incremento *Q* del tempo reale di esecuzione

# Relazione tra funzioni di *scheduler* e nucleo




*RIASSUNTO*

# CFS – Scheduling RR puro (una CPU)

∀ *tick* verifica se *Q* scade, se sì *resched*

*CURR.SUM* – *CURR.PREV* >= *CURR.Q*

aggiornato da *pick_next_task_fair*

*task_tick_fair* chiamata da *interrupt routine RT clock*

*tick*

tempo REALE dal *tick* precedente

$CURR.SUM = CURR.SUM + DELTA$
$CURR.VRT = CURR.VRT + DELTA \times CURR.VRTC$
$VMIN = min\ (CURR.VRT,\ LFT.VRT)$

∀ *tick* aggiorna i contatori *SUM* e *VRT* di *CURR* e il *VRT minimo* di tutti i *task* in *runqueue*

*tick* ≈ 0,01 *ms*

**ROUND ROBIN**

il numero di *task* è costante
(non ci sono né *clone* né *exit*)

nessun *task* si autosospende
(ogni *task* consuma tutto il quanto)

costanti / parametro base / parametri derivati

| | | |
|---|---|---|
| *LT* | latenza | 6 *ms* |
| *GT* | granularità | 0,75 *ms* |
| *NRT* | num. totale di *task* | ≥ 1 |
| *t.LOAD* | = peso del *task t* | *(re)nice* |
| *PER* | = *max* (*LT*, *NRT* × *GR* ) | ≥ 6 *ms* |
| *RQL* | = Σ pesi di tutti i *task* | ≠ 0 |
| *t.LC* | = *t.LOAD* / *RQL* | ≤ 1 |
| *t.Q* | = *PER* × *t.LC* | ≤ *PER* |
| *t.VRTC* | = 1 / *t.LOAD* | ≠ 0 |

aggiorna ogniqualvolta *t.LOAD* varia

*task* (re)inserito in *RB* e (ri)ordinato per *VRT*

continua ad aggiornare *SUM* e *VRT* di *CURR*

smetti di aggiornare *SUM* e *VRT* del vecchio *CURR*

*CURR* esecuzione

*schedule*
(chiama *pick_next_task_fair*)
(effettua *context_switch*)

*PRONTO*

riprendi ad aggiornare *SUM* e *VRT* del nuovo *CURR*

*puntatore a CURR*

*VRT* crescenti

*LFT*

coda *RB*

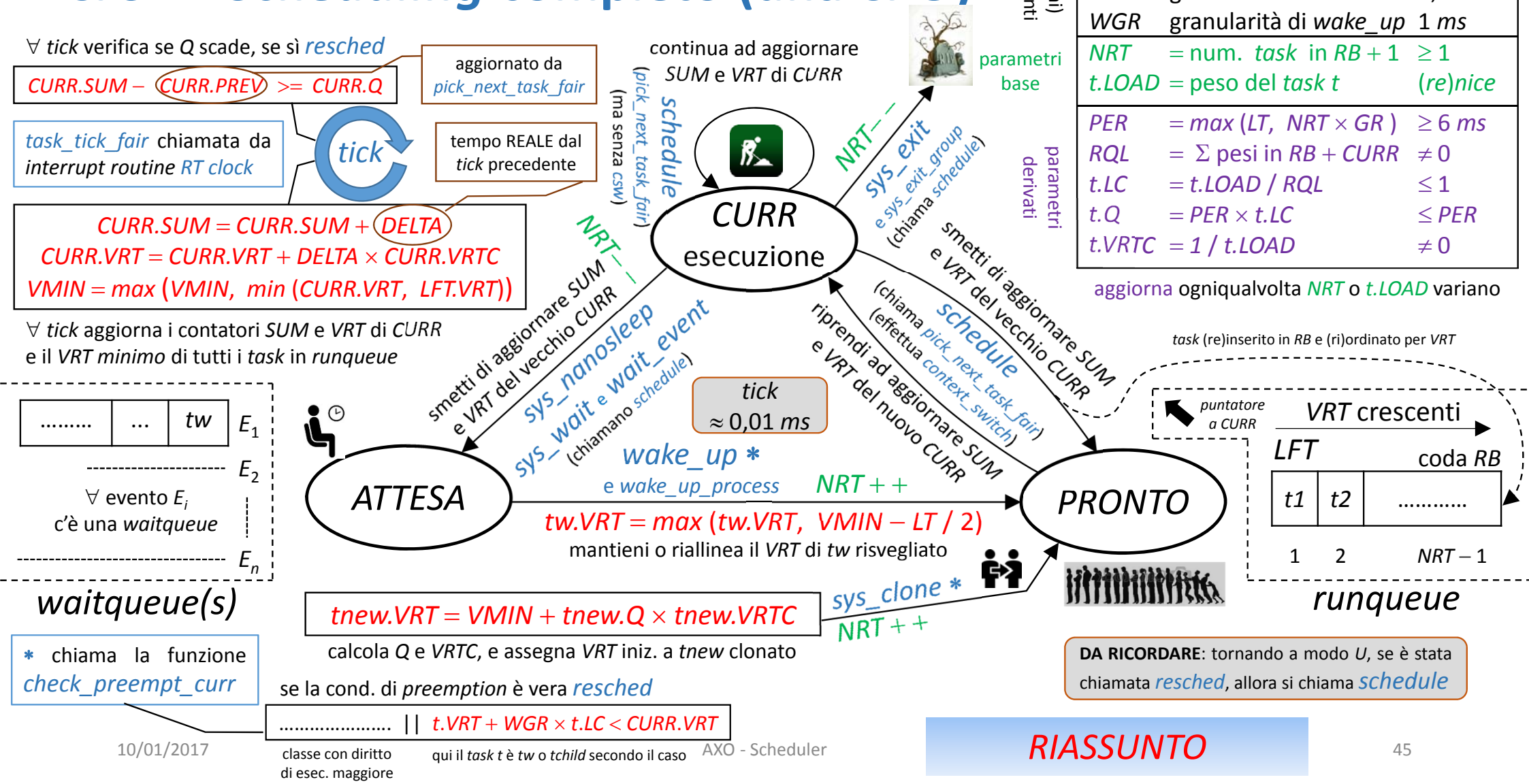
| *t1* | *t2* | ..... |
|---|---|---|

1 2 *NRT* – 1

*runqueue*

**DA RICORDARE**: tornando a modo *U*, se è stata chiamata *resched*, allora si chiama *schedule*

# CFS – *Scheduling* completo (una *CPU*)




## RIASSUNTO