Linux

Gabr1313

January 5, 2024

Contents

1 Introduzione **3**
- 1.1 Thread . . . 3
- 1.2 Sistema operativo . . . 3
- 1.3 Politica di scheduling . . . 4
- 1.4 Architettura . . . 4
- 1.5 Strutture dati per la gestione dei processi . . . 4
  - 1.5.1 Descrittore del processo . . . 4
  - 1.5.2 sStack (sPila) . . . 4

2 Meccanismi Hardware di supporto **5**
- 2.1 Considerazioni generali . . . 5
- 2.2 Modi di funzionamento . . . 5
- 2.3 Memoria virtuale . . . 5
- 2.4 Syscall . . . 5
- 2.5 Interrupt . . . 6
  - 2.5.1 Priorità e interrupt nidificati . . . 7
  - 2.5.2 ABI . . . 7

3 Gestione dello stato dei processi **8**
  - 3.0.1 SPila . . . 8
- 3.1 Context switch . . . 9
- 3.2 Preemption . . . 9
- 3.3 Gestione degli interrupt . . . 9
- 3.4 Attesa/Pronto . . . 9
- 3.5 Segnali . . . 10
- 3.6 Esempio: Driver di periferica . . . 11
- 3.7 Esempio: Mutex . . . 11
- 3.8 Attesa di exit e di un timeout . . . 11
- 3.9 Le funzioni dello scheduler . . . 12
- 3.10 Curiosità . . . 12

4 I servizi di sistema **13**
- 4.1 Bootstrap . . . 13
- 4.2 Idle . . . 13
- 4.3 System Call Interface . . . 13
- 4.4 Creazione dei processi (normali e leggeri) . . . 13
  - 4.4.1 sys_clone . . . 13
  - 4.4.2 Implementazione di fork() . . . 14
  - 4.4.3 Implementazione di clone() . . . 14
- 4.5 Eliminazione dei processi . . . 14

5 Lo scheduler **15**
5.1 Requisiti dei processi 15
5.2 Classi di Scheduling 15
5.3 Scheduling dei processi soft real-time 16
5.4 Scheduling dei processi normali 17
5.4.1 Virtual Time 17
5.5 Gestione di wait e wakeup 18
5.5.1 Necessità di rescheduling 18
5.5.2 Creazione e cancellazione di task 18
5.6 Assegnazione dei pesi ai processi 18

Chapter 1

Introduzione

Definizione 1 (Posix) *Standard per i sistemi operativi Unix (usato di seguito sempre).*

Definizione 2 (Processo) *Macchina virtuale che segue un programma*

Definizione 3 (System service) *Particolare funzioni di sistema a cui un processo può accedere*

- fork, exec, wait, exit
- comunicazione tra processi
- accesso ai file (periferiche)

1.1 Thread

I processi leggeri appartenenti allo stesso processo normale condividono la stessa memoria, a esclusione dello stack.

Sono identificati dalla coppia <**PID, TGID**>, dove PID = *process ID* e TGID = *Thread group ID*. Le funzioni associate sono `getid()` e `getpid()`.

1.2 Sistema operativo

Il sistema operativo è lo strato che separa l'hardware dall'utente. La usa funzione principale è permettere l'esecuzione di diversi processi in parallelo (con 1 sola cpu i processi vengono alternati).

Definizione 4 (Context switch) *Sostituzione processo in esecuzione*

L'esecuzione di un programma può essere sospesa per 2 motivi:

- Sospensione autonoma
- Sospensione forzata

Definizione 5 (Scheduler) *Componente del SO che decide quale processo mandare in esecuzione*

1.3 Politica di scheduling

- Processi più importanti vengono eseguiti prima
- A pari importanza i processi vengono eseguiti in maniera equa

Linux è un sistema time sharing

Definizione 6 (Preemption) *Pausa forzata dal SO di un processo allo scadere di un quanto di tempo*

Definizione 7 (Kernel non-preemptable) *La preemption è proibita quando un processo esegue servizi del sistema operativo*

Definizione 8 (Kernel module) *Moduli software inseriti nel sistema per permettere l'accesso alle periferiche senza la necessità di ricompilare il sistema. Sono inoltre caricati dinamicamente durante l'esecuzione quando necessario.*

1.4 Architettura

I file che contengono codice dipendente dall'architettura in `linux/arch` D'ora in poi si farà riferimento all'architettura **x86-64**.

1.5 Strutture dati per la gestione dei processi

1.5.1 Descrittore del processo

La funzione `get_current()` restituisce un puntatore al task corrente

```
struct task_struct {
    pid_t pid;
    pid_t tgid;
    volatile long state; // -1 unrunnable, 0 runnable, >0 stopped
    void *stack; //puntatore all' sStack del task
    ...
}

struct thread_struct {
    unsigned long sp0;
    unsigned long sp; // puntatore allo stack di modo S
    unsigned long usersp; // puntatore allo stack di modo U (idem)
    ...
}
```

1.5.2 sStack (sPila)

La sPila è diversa per ogni processo

Chapter 2

Meccanismi Hardware di supporto

2.1 Considerazioni generali

1. PC = program counter
2. SP = stack pointer
3. push o pop dalla pila sono un'unica istruzione
4. chiamate a funzioni salvano in automatico sulla pila il return address
5. PSR = Process state register: semplificazione di una struttura dati HW che contiene le informazioni riguardanti un processo
6. SSP = Supervisor stack pointer
7. USP = User stack pointer

2.2 Modi di funzionamento

- **Utente (U)**/Non privilegiato: non può eseguire istruzioni privilegiate e può accedere solo alle proprie zone di memoria
- **Supervisore (S)**/Kernel/Privilegiato: può eseguire anche le istruzioni privilegiate, a accedere alle zone di memoria di qualsiasi processo

2.3 Memoria virtuale

Lo spazio di indirizzamento è 2^{64} byte. Di questi sono utilizzabili solo i primi 2^{47} byte da U e gli ultimi 2^{47} byte da S. Gli altri indirizzi di memoria sono detti *non-canonical area*.

Una pagina di memoria è di $4Kb$. Ogni indirizzo virtuale prodotto dalla CPU viene mappato a un indirizzo fisico grazie alla **Tabella delle Pagine**.

2.4 Syscall

Definizione 9 (Syscall) *Particolare chiamata a funzione eseguita da un processo quando ha bisogno di un servizio dell'OS.*

Definizione 10 (Vettore di Syscall) *Struttura dati HW costituita da 2 celle, inizializzata dal SO durante l'avvio.*

- PC: indirizzo della funzione `system_call()`
- PSR

Una sycall quindi, differentemente da una chiamata a funzione, non indica l'indirizzo a cui deve saltare, ma questo è contenuto nel vettore di syscall.

Anche i valori di SSP e USP sono salvati in una struttura dati HW.

Una Syscall permette di passare dallo stato U allo stato S svolgendo le seguenti operazioni:

1. USP = SP
2. SP = SSP
3. push del PC(→UPila) sulla SPila
4. push del PSR sulla SPila
5. caricamento di PC(→SPila) e PSR contenuti nel vettore di syscall

Definizione 11 (Sysret) *Inverso della syscall.*

Una Sysret, eseguita all fine di una `system_call()` permette di passare dallo stato S allo stato U svolgendo le seguenti operazioni:

1. pop del PSR dalla SPila e ripristino registro
2. pop del PC(→UPila) dalla SPila e ripristino registro
3. SP = USP

2.5 Interrupt

A un insieme di eventi HW è associata un funzione, detta **Routine di Interrupt**. Il Meccanismo con cui funziona una Routine di Interrupt è lo stesso delle syscall, con però alcune differenze:

- Sono asincrone: attivate da eventi e non esplicitamente da un programma
- l'istruzione di ritorno è chiamata IRET
- il vettore di syscall è sostituito con la **Tabella degli Interrupt**

Se il processo in esecuzione è in modalità S, PC e PSR vengono comunque salvati sulla sPila.

Definizione 12 (Tabella degli Interrupt) *Struttura dati HW che contiene i* ***vettori di interrupt****, inizializzata dal SO durante l'avvio.*

Definizione 13 (Vettore di Interrupt) *Struttura dati HW costituita da 2 celle*

- PC: indirizzo della funzione `system_call()`
- PSR

Un meccanismo HW converte l'ID dell'interrupt nell'indirizzo del corrispondente vettore di interrupt.

La gestione degli errori durante i programmi spesso viene trattata come un interrupt che termina forzatamente il programma eliminando il processo (abort).

2.5.1 Priorità e interrupt nidificati

Nel PSR, a ogni processo è associato un livello di priorità, e identicamente funziona per gli interrupt. Un interrupt viene accettato solo se il suo livello ha priorità superiore rispetto al processo in esecuzione, altrimenti è tenuto in sospeso. sospeso.

2.5.2 ABI

Gli eseguibili normalmente sono caricati dal SO, mentre gli eseguibili del sistema stesso devono essere in grado di partire autonomamente (bootstrap).

Definizione 14 (Application Binary Interface) *Regole che il compilatore utilizza durante la traduzione (indipendentemente dall'architettura HW), in modo che tutti i moduli siano coerenti.*

Funzioni della libreria glibc come `fork()` e `open()` sono dei wrapper attorno alla funzione

```
long syscall(long syscall_number, ...);
```

La chiamata assembly corrispondente è composta da varie istruzioni, l'ultima è la `SYSCALL`.

Chapter 3

Gestione dello stato dei processi

Ogni processo può trovarsi in uno dei 2 stati:

- **Attesa**: attende un evento
- **Pronto**: può essere eseguito se selezionato dallo scheduler. Il processo in esecuzione è chiamata **processo corrente**.

Un processo in esecuzione abbandona tale stato a causa di un evento:

- un servizio di sistema richiesto dal processo deve attendere un evento. Il processo passa allo stato di attesa.
- **Preemption**: il SO decide di sospendere l'esecuzione a favore di un altro processo. Il processo rimane nello stato di pronto.

Definizione 15 (Scheduler) *Componente del SO che esegue le seguenti funzioni:*

- Politica di scheduling: quale processo mettere in esecuzione e per quanto tempo
- Context switch: `schedule()`

Lo scheduler deve quindi gestire la struttura dati `runqueue`, la quale contiene 2 campi:

- **RB** (red-black tree): lista dei puntatori ai descrittori dei processi pronti
- **CURR**: puntatore al descrittore processo corrente

3.0.1 SPila

Linux su x86-64 assegna a ogni processo una sPila di 2 pagine (8Kb). Questa è utilizzata soltanto durante l'esecuzione di un servizio di sistema.

I valori di SSP e USP sono diversi per ogni processo: prima che avvenga un context switch vengono salvati nel descrittore del processo.

- **sp0**: indirizzo di base della SPila
- **sp**: valore di SP nel momento in cui il processo viene sospeso

3.1 Context switch

Quando P è in modalità U SSP(processo) = sp0(descrittore).
Quando P è in modalità S USP contiene il valore di ritorno sulla UPila.

Il context switch può avvenire solo in modalità S (vedi 3.2), quindi:

1. push di USP sulla sPila
2. sp(descrittore) = SP(processo)

Ovviamente quando P riprende l'esecuzione si eseguono le operazioni inverse:

1. SP(processo) = sp(descrittore)
2. pop USP dalla sPila e ripristino
3. SSP(processo) = sp0(descrittore)

3.2 Preemption

Essendo il Kernel non-preemptive (problemi durante il cambio di contesto mentre sono in esecuzione routine di interrupt annidate, ad esempio quando un processo viene iniziato da un processo, interrotto e rieseguito da un altro), se un task dovrebbe essere sospeso mentre si trova in modalità S, questo non avviene ma la flag `TIF_NEED_RESCHED` viene messa a 1. Al ritorno in modalità U avverrà il context switch.

Paradossalmente il SO può decidere di eseguire una preemption solo in modalità S (aka l'esecuzione può essere abbandonata solo dallo stato S). Si adotta quindi la seguente regola: se il processo è in modalità S il context switch avviene alla fine della routine la cui ultima istruzione fa tornare il processo in modalità U (SYSRET/IRET).

3.3 Gestione degli interrupt

La routine di interrupt è **trasparente** al processo: l'interrupt è eseguito nel contesto del processo di esecuzione. La routine di interrupt può risvegliare dei processi in stato di attesa.

Un interrupt può avvenire in 3 situazioni:

- P è in modalità U
- P è in modalità S
- P sta già eseguendo una routine di interrupt

3.4 Attesa/Pronto

Le attese possono essere di 3 tipi:

- I/O
- sblocco di un lock (mutex e semafori)
- scadere di un timer

Definizione 16 (Wait_queue) *Struttura dati che contiene i descrittore dei processi in attesa di un certo evento.*

Definizione 17 (Identificatore dell'evento) *Indirizzo della wait_queue.*

La macro `DECLARE_WAIT_QUEUE_HEAD` dichiara una wait_queue.

Ad ogni wait_queue è associata una flag che indica la tipologia di attesa:

Definizione 18 (Attesa esclusiva) *Un solo processo viene risvegliato all'avvenire dell'evento.*

Definizione 19 (Attesa non esclusiva) *Tutti i processi vengono risvegliati all'avvenire dell'evento.*

I processi in attesa esclusiva sono inserite alla fine della coda.

Le funzioni associate sono: `wait_event_interruptible_exclusive()` e `wait_event_interruptible()`

La funzione `wake_up()` risveglia i processi in attesa, mettendoli in stato di pronto.

1. Cambia lo stato da attesa a pronto
2. Elimina il/i processo/i dalla wait_queue e li inserisce nella runqueue

3.5 Segnali

Definizione 20 (Segnale) *Avviso asincrono inviato a un processo dal SO o da un altro processo.*

I segnali sono numerati da 1 a 31.

I segnali che non possono essere bloccati sono:

- SIGKILL
- SIGSTOP

Altri segnali sono:

- SIGINT: terminazione del processo (CTRL+C)
- SIGSTOP: sospensione del processo (CTRL+Z)

Quando il segnale viene inviato, il processo può essere in uno dei seguenti stati:

1. **Esecuzione U**: il segnale viene processato immediatamente
2. **Esecuzione S**: il segnale viene processato al ritorno in modalità U
3. **Pronto**: il segnale viene processato quando il processo ritorna in esecuzione
4. **Attesa**:
   - se l'attesa è **interrompibile** il processo viene risvegliato (pronto). Al risveglio il processo deve controllare se la condizione di attesa è diventata falsa, e in caso contrario tornare in attesa
     `wait_interruptible()`
     `wait_killable()`
   - se l'attesa è **non interrompibile** il segnale rimane pendente
     `wait_event()`

3.6 Esempio: Driver di periferica

1. P (mod U) richiede un servizio alla periferica PX (`write()`)
2. Syscall (mod S): il SO chiama la funzione del gestore X (`X.write()`) nel contesto di P(write): non avviene il context switch
3. X copia dal buffer dallo spazio del processo P al buffer di X C caratteri: `C = min(buffer.size(), char_to_be_written)`
4. `X.write()` invia il primo carattere
5. `X.write()` chiama `wait_event_interruptible()`
6. quando PX termina l'operazione genera un interrupt
7. Routine di interrupt di X: se nel buffer esistono altri caratteri, $\rightarrow$ 4
8. `wake_up()`: Se ci sono altri caratteri da scrivere $\rightarrow$ 3
9. `X.write()` chiama la Sysret (mod U)

3.7 Esempio: Mutex

Quando un mutex è libero, per evitare di fare una system call (e cioè passare alla modalità S e mettersi in attesa), Linux utilizza i **Futex** (Fast Userspace Mutex). Un futex possiede una **variabile intera** nello spazio U e una **waitqueue** nello spazio S.

1. Test e incremento della variabile intera in maniera atomica (in x86-64 esiste un'istruzione apposita)
2. Se la variabile è bloccata:
   - `sys_futex(FUTEX_WAIT,...)`
   - `wait_event_interruptible_exclusive(WQ, ...)`
3. Per sbloccare un mutex:
   - `sys_futex(FUTEX_WAKE,...)`
   - `wakeup(WQ)`

(`sys_futex` non non è una funzione, ma un numero sulla tabella delle routine delle syscall)

3.8 Attesa di exit e di un timeout

Nel caso dei futex e dei driver di periferica la funzione di wakeup conosce la coda dell'evento. Ci sono tuttavia 2 casi in cui ciò non avviene. In questi casi viene utilizzata la funzione `wake_up_process()`, la quale riceve in ingresso il puntatore al descrittore del processo da risvegliare.

I 2 casi sono:

- **Attesa di exit**: processo padre chiama `wait()`. Il figlio chiama `exit()`. Il figlio nel descrittore possiede il puntatore al processo padre da risvegliare.
- **Attesa di un timeout**: Ogni elemento della lista dei timeout possiede il puntatore al processo da risvegliare. Una particolare funzione (ogni tanto) controlla lo scadere dei timer e risveglia i processi.

3.9 Le funzioni dello scheduler

- `schedule()`: se in attesa chiama `deque_task()`. Successivamente esegue il context switch.
- `check_preempt_curr()`: se il task deve essere preempted `TIF_NEED_RESCHED = 1`.
- `enqueue_task()`: inserisce il task nella runqueue
- `dequeue_task()`: rimuove il task dalla runqueue
- `resched()`: `TIF_NEED_RESCHED = 1`
- `task_tick()`: scheduler periodico invocato dall'interrupt del clock. Aggiorna vari contatori e determina se il task debba essere preempted.

3.10 Curiosità

- In sistemi multiprocessore il problema di esecuzioni concorrenti è risolto usando meccanismi di sincronizzazione (lock). Il kernel, per semplicità, rimane non preemptive.
- In sistemi multiprocessore per attese brevi invece che utilizzare i Mutex (i quali richiedo un context switch) utilizza gli **Spinlock**: il task entra in un loop finché non ottiene il lock (ha senso solo in sistemi multiprocessore).
- Esistono altri stati di un processo: `TASK_STOPPED`, `EXIT_ZOMBIE` e `EXIT_DEAD`. Questi processi non appartengono a nessuna `runqueue` o `waitqueue` e sono acceduti tramite `PID`.

Chapter 4

I servizi di sistema

4.1 Bootstrap

Le operazioni avviamento successive alla creazione del primo processo, sono svolte dal programma `init`, un programma user che crea un processo per ogni terminale sul quale potrebbe essere eseguito un login. Se il login va a buon fine viene lanciato il programma `shell` (interprete comandi).

4.2 Idle

`Idle` è il processo 1, che non svolge alcuna funzione utile dopo aver concluso l'avvio del sistema. Infatti dopo aver concluso le operazioni di avviamento:

- i diritti di esecuzione di `Idle` sono inferiori a quelli di qualsiasi altro processo
- non è mai in stato di attesa

4.3 System Call Interface

La funzione `system_call()` controlla che il numero nel registro `rax` sia valido, e lo utilizza come offset su una tabella che contiene l'indirizzo della routine da eseguire (come una `switch`).

4.4 Creazione dei processi (normali e leggeri)

La libreria più utilizzata è la **Native Posix Thread Library**. I processi normali sono creati da una `fork()`, quelli leggeri da una `thread_create()`.

La funzione `clone()` permette d creare un processo con caratteristiche di condivisione.

4.4.1 sys_clone

`sys_clone` non è una funzione, ma un numero sulla tabella delle routine delle syscall.

Il servizio di sistema chiamata da `clone()` è `sys_clone()`

```
long sys_clone(unsigned long flags, void *child_stack, ...);
```

La uPila del figlio è un clone di quella del padre. In particolare se `child_stack = NULL` l'indirizzo virtuale di USP è lo stesso di quello del padre, altrimenti è quello specificato.

4.4.2 Implementazione di fork()

```
pid_t fork() {
    ...
    syscall(sys_clone, ...); /* no falgs */
    ...
}
```

4.4.3 Implementazione di clone()

```
clone(void (*fn)(void *), void *child_stack, int flags, void *arg, ... ) {
    //push arg e indirizzo di fn utilizzando child_stack
    syscall(sys_clone, ...);
    if (child) {
        //pop arg e fn dalla pila
        fn(arg);
        syscall(sys_exit, ... );
    }
    else return;
}
```

4.5 Eliminazione dei processi

- `sys_exit`
- `sys_exit_group`: invia il `signal` di terminazione ai processi del gruppo ed esegue poi la `sys_exit`

`sys_exit` e `sys_exit_group` non sono una funzione, ma un numero sulla tabella delle routine delle syscall.

```
sys_exit(code) {
    struct task_struct *tsk = current( ) // il processo che esegue exit
    exit_mm(tsk);       // rilascia la memoria del processo
    exit_sem(tsk)       // rimuovi il processo dalle code per semafori
                        // o mutex (post su semafori, unlock su mutex)
    exit_files(tsk)     // rilascia i files

    //notifica il codice di uscita al processo padre

    wakeup_up_parent(tsk->p_pptr)  //invoca wake_up del padre
    schedule( );        // esegui un nuovo processo
}
```

La terminazione del singolo thread termina con `sys_exit` (vedi implementazione `clone()`.

La funzione `exit()`, usata per terminare un processo con i suoi thread, invece, è implementata invocando `sys_exit_group()`, eliminando i processi con lo stesso TGID.

Chapter 5

Lo scheduler

Il comportamento dello scheduler è orientato a garantire le seguenti condizioni:

- Processi più importanti vengono eseguiti prima
- A pari importanza i processi vengono eseguiti in maniera equa

Definizione 21 (Politica Round Robin) *Assegnare a ogni processo uno stesso quanto di tempo in ordine circolare.*

Definizione 22 (Priorità) *Diritto di esecuzione*

I momenti in cui lo scheduler viene invocato sono:

- un processo si auto-sospende
- un processo viene risvegliato
- scadere del quanto di tempo

5.1 Requisiti dei processi

1. **Real-time**: garanzia che non vengano eseguiti con un ritardo superiore a un certo limite
2. **semi-Real-time**: rapidità di risposto senza vincoli stringenti sul ritardo
3. **normali**:
    - **I/O bound**
    - **CPU bound**

5.2 Classi di Scheduling

Diversi processi possono avere diverse politiche di scheduling. Queste sono realizzate da una **Scheduler Class**, il cui puntatore è salvato nel descrittore del processo. Ad esempio la classe `fair_sched_class`:

```
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,
};
```

Quando ad esempio un processo in mod S chiama `enqueue_task()`, questa a sua volta chiama `p->sched_class->enqueue_task()`.

Le 3 classi più importanti sono, in ordine di diritto di esecuzione:

- `SCHED_FIFO`
- `SCHED_RR`
- `SCHED_NORMAL`

La funzione `schedule()` invoca `pick_next_task()`, la quale invoca l'istruzione specifica della singola classe. Poi procede al context switch.

```
schedule() {
struct tsk_struct *prev = CURR;
if (prev->stato == ATTESA) dequeue(prev);
// if (prev terminato) dequeue(prev);
next = pick_next_task(rq, prev);
if (next != prev) {
context_switch(prev, next);
// CURR->START = NOW;
}
TIF_NEED_RESCHED = 0;
}

pick_next_task(rq, prev) {
for (/* class in ordine di importanza */) {
next = class->pick_next_task(rq, prev);
if (next != NULL) return next;
}
}
```

5.3 Scheduling dei processi soft real-time

Per supportare i processi RT, Linux utilizza `SCHED_FIFO` e `SCHED_RR`. A ogni processo è attribuita una **priorità statica** (da 1 a 99), la quale è memorizzata in `task_struct.static_prio`. In entrambi i casi processi con maggiore priorità hanno diritto di esecuzione. Solo nel caso di processi con uguale priorità le politiche differiscono.

5.4 Scheduling dei processi normali

Lo scheduler dei processi normali è chiama **Completely Fair Scheduler (CFS)**. IL CFS deve determinare ragionevolmente la durata dei quanti, assegnare più tempo ai processi più importanti e permettere a processi in attesa di tornare rapidamente in esecuzione.

Supponendo che tutti i task abbiamo `t.LOAD = 1` e che questi non si autosospendano. Il CFS determina un periodo **PER** all'interno del quale ognuno degli **NRT** processi riceve un quanto di tempo **Q**. I task sono ordinati in un **red-black tree**.

$$\texttt{PER = max(LT, NRT * GR)}$$

dove **LT** è la latenza (default 6ms) e **GR** è il granularità (default 0.75ms).

Per calcolare Q, invece, si utilizza una media pesata, dove `RQL` è run queue load e `LC` è load coefficient:

$$\texttt{RQL} = \sum \texttt{LOAD}$$

$$\texttt{LC = t.LOAD / RQL}$$

$$\texttt{t.Q = PER * t.LC = PER * t.LOAD / RQL}$$

5.4.1 Virtual Time

Definizione 23 (Virtual Runtime (VRT)) *Misura del tempo di esecuzione di un processo modificato in base a opportuni coefficienti*

Sia `DELTA` il tempo che un task viene eseguito in *ns*. `SUM` rapprensenta il tempo effettivamente trascorso in esecuzione per ogni processo.

$$\texttt{t.VRTC = 1 / t.LOAD}$$

$$\texttt{t.VRT = t.VRT + DELTA * t.VRTC = t.VRT + DELTA / t.LOAD}$$

$$\texttt{SUM = SUM + DELTA}$$

Si ottiene così l'effetto che VRT aumenti più velocemente nei processi più leggeri (i processi più pesanti utilizzano la CPU per più tempo a ogni ciclo).

Inoltre, la runqueue aggiorna continuamente la variabile `VMIN` (LFT = left of RB) (il perchè del `max(...)` è spiegato in seguito):

$$\texttt{VMIN = max(VMIN, min(CURR.VRT, LFT.VRT))}$$

La funzione `tick()` risulta essere quindi

```
tick() {
    DELTA = NOW - CURR->START;
    CURR->SUM = CURR->SUM + DELTA; // current time
    CURR->VRT = CURR->VRT + DELTA * CURR.VRTC;
    CURR->START = NOW;
    VMIN = max(VMIN, min(CURR.VRT, LFT.VRT)); // FALSO TODO
    if (CURR->SUM - CURR->PREV >= Q) resched(); // è scaduto il quanto di tempo
}
```

dopodichè `schedule()` chiama `pick_next_task()` che eventualmente chiama `pick_next_task_fair()`

```
pick_next_task_fair(rq, previous_task) {
    if (LFT != NULL) {
        CURR = LFT;
        // elimina LFT da RB ristrutturando la coda opportunamente
        CURR->PREV = CURR->SUM;
    } else if (CURR == NULL) { // CURR è stato eliminato perchè in ATTESA
        CURR = IDLE;
    }
    return CURR;
}
```

5.5 Gestione di wait e wakeup

Un processo risvegliato dall'attesa è plausibile abbia un `VRT` molto basso: potrebbe perciò essere modificato. Inoltre modifica i valori di `NRT` e `RQL`, e quindi `LC` e `Q` di ogni processo.

```
tw.VRT = max(tw.VRT, (VMIN - LT/2))
```

Dato che è opportuno che la crescita di `VMIN` sia monotona:

```
VMIN = max(VMIN, min(CURR.VRT, LFT.VRT))
```

5.5.1 Necessità di rescheduling

Gli eventi WAIT implicano sempre un rescheduling, mentre gli eventi WAKEUP eseguono un rescheduling se:

- la classe del processo risvegliato ha priorità maggiore, oppure
- `tw->vrt + WGR * tw->load_coeff < CURR->vrt` dove `WRG` (wakeup granularity) è solitamente impostato a 1ms.

5.5.2 Creazione e cancellazione di task

`exit()` e `clone()` implicano un ricalcolo dei parametri della runqueue. In particolare durante una clone

- `tnew` viene creato
- i paramtetri della runqueue vengono ricalcolati
- `tnew.VRT = VIMN + tnew.Q * tnew.VRTC`: motivo per cui i valori di `VRT` possono differire molto dai valori di `SUM`

Il nuovo processo non viene inserito in prima posizione nel RB.

Il rescheduling viene valutato come se avvenisse una WAKEUP.

5.6 Assegnazione dei pesi ai processi

Definizione 24 (Nice Value) *Valore assegnabile dall'utente: -20 è massima priorità, 19 è minima.*

$$\texttt{t.LOAD} = 1024 \,/\, (1.25^{\,\texttt{t.NICE}})$$