

MISURA DEL TEMPO

Moltissime attività del computer, spesso in background, sono regolate dalla misura del tempo. Ad esempio, se il monitor si spegne automaticamente dopo aver terminato le attività alla console, ciò è dovuto a un timer che permette al kernel di sapere quanto tempo è passato dall'ultima pressione di un tasto o dal movimento del mouse. Se si riceve un avvertimento del sistema che chiede di cancellare un gruppo di file non più usati, si tratta del risultato di un programma che identifica tutti i file che non sono stati usati da lungo tempo. Per fare questo, i programmi devono poter leggere un timestamp (marca temporale: sequenza di caratteri che rappresenta una data e un orario – ndt) che identifica l'ultimo accesso ad ogni file. Questo timestamp deve essere registrato automaticamente dal kernel. Inoltre la misurazione del tempo regola le commutazioni di processo insieme ad eventi più visibili come gli eventi di time out.

Si possono distinguere due tipi principali di misura del tempo effettuate dal kernel:

- ora e data correnti, che possono essere ottenute dai programmi per mezzo delle API `time()`, `ftime()` e `gettimeofday()` e usate dal kernel come timestamp per i file e i pacchetti di rete;
- implementazione di timer – meccanismi in grado di avvisare il kernel o un programma utente che un certo intervallo di tempo è trascorso.

La misura del tempo si basa su circuiti hardware provvisti di oscillatori a frequenza fissa e contatori. La prima parte di questa esposizione descrive i componenti hardware alla base della misura del tempo; la seconda descrive le operazioni del kernel: implementazione del time-sharing (letteralmente “condivisione del tempo”) della CPU, aggiornamento della misura del tempo e delle statistiche di impiego delle risorse, controllo dei timer software. L'ultima parte si occupa delle chiamate di sistema legate al tempo e delle routine corrispondenti.

Orologio e circuiti di misura del tempo

Nei sistemi 80x86 il kernel deve interagire direttamente con vari diti di orologi e circuiti di misura del tempo. Gli *orologi* sono usati sia per tener traccia dell'orario corrente, sia per precise misure temporali. I *circuiti di misura del tempo* sono programmati dal kernel per inviare interrupt a intervalli definiti; questi ultimi sono fondamentali per implementare i timer software. Vengono ora brevemente descritti l'orologio e i circuiti presenti nei PC IBM compatibili.

Real Time Clock

Tutti i PC contengono un orologio chiamato *Real Time Clock* (RTC), indipendente dalla CPU e dagli altri circuiti. Il RTC continua a misurare il tempo anche quando il computer è spento, poiché è alimentato da una batteria. La RAM CMOS e il RTC sono integrati in un unico chip (Motorola 146818 o altro equivalente). Il RTC invia interrupt periodici sulla IRQ 8 a frequenze che vanno da 2 Hz a 8,192 Hz. Può essere programmato per attivare la linea 8 quando raggiunge un valore specifico, funzionando quindi da allarme.

Linux usa il RTC solo per determinare l'ora e la data; permette però ai processi di programmarlo agendo sul file di dispositivo `/dev/rtc`. Il kernel ha accesso al RTC attraverso le porte di I/O 0x70 e 0x71. L'amministratore di sistema può leggere e impostare l'orario sul RTC eseguendo il programma Unix *clock* che agisce direttamente su queste porte.

Time Stamp Counter

Tutti i processori 80x86 hanno un pin CLK di input, che riceve il segnale di clock da un oscillatore esterno. A partire dal Pentium, i processori hanno un contatore che viene incrementato ad ogni segnale di clock. Il contatore è accessibile attraverso il registro a 64 bit *Time Stamp Counter* (TSC) che viene letto per mezzo della istruzione assembly `rdtsc`. Quando usa il registro, il kernel deve tenere conto della frequenza del segnale di clock: se, per esempio, l'orologio ha una frequenza di 1 Ghz, il TSC viene incrementato ogni nanosecondo.

Linux sfrutta questo registro per misurare il tempo con maggior precisione rispetto a quella ottenibile con il Programmable Interval Timer (PIT). Per fare ciò, Linux deve determinare la frequenza del segnale di clock quando inizializza il sistema. Infatti, poiché la frequenza non è dichiarata all'atto della compilazione, la medesima immagine del kernel può girare su CPU i cui orologi hanno frequenze diverse. Il compito di determinare la frequenza della CPU è svolto durante il boot del sistema. La funzione `calibrate_tsc()` calcola la frequenza contando il numero di segnali di clock nell'intervallo di 5 millisecondi. Questo valore costante viene determinato impostando opportunamente uno dei canali del PIT¹.

Programmable Interval Timer

Oltre al RTC e al TSC, i PC hanno un altro dispositivo di misurazione del tempo chiamato Programmable Interval Timer (PIT). Il ruolo del PIT è simile a quello del timer del forno a microonde: informa l'utente che un certo intervallo di tempo è trascorso. Invece di far suonare un campanello, invia uno speciale interrupt chiamato *timer interrupt*. Inoltre, a differenza della sveglia, continua ad inviare interrupt alla frequenza fissa definita dal kernel. Ogni PC contiene almeno un PIT, implementato di solito per mezzo di un chip CMOS 8254 che usa le porte di I/O 0x40 – 0x43.

Linux programma il PIT perché invii interrupt sulla linea IRQ 0 a una frequenza di circa 1000 Hz, cioè uno ogni millisecondo (ms). Questo intervallo di tempo è definito `tick`, e la sua lunghezza in nanosecondi è memorizzata nella variabile `tick_nsec`. Su un PC `tick_nsec = 999,848` nanosecondi (che corrisponde ad una frequenza di 1000,15 Hz), ma il suo valore viene aggiornato dal kernel se il computer è sincronizzato su un clock esterno. I tick scandiscono il tempo per tutto il sistema: sono come i battiti del metronomo per un musicista che sta provando.

In generale, tick più brevi danno origine a timer con risoluzione più elevata, che si traducono in riproduzione multimediale più fluida e tempi di risposta inferiori nel multiplexing in I/O sincrono (chiamate di sistema `poll()` e `select()`). Questo però ha un costo; tick più brevi costringono la CPU a impiegare più tempo in modalità del kernel e quindi meno in modalità utente: di conseguenza, i programmi girano più lentamente. La frequenza degli interrupt del timer dipende dall'architettura hardware: le macchine più lente hanno un tick di circa 10 millisecondi (100 interrupt per secondo), mentre le più veloci hanno un tick di 1 ms (1000 – 1024 interrupt per secondo).

Alcune macro contengono costanti che determinano la frequenza degli interrupt del timer:

- `HZ` contiene il numero di timer interrupt per secondo (cioè la frequenza). Per i PC è 1000.
- `CLOCK_TICK_RATE` contiene il valore 1.193.182, cioè la frequenza di oscillazione interna del chip 8254.

¹ Per non perdere cifre significative nella divisione intera, `calibrate_tsc()` restituisce la durata in microsecondi di un tick di clock moltiplicato per 2 elevato alla 32.

- LATCH contiene il rapporto tra CLOCK_TICK_RATE e HZ, arrotondato all'intero più vicino. E' usato per programmare PIT.

Il PIT è inizializzato da setup_pit_timer():

```
void setup_pit_timer(void)
{
    extern spinlock_t i8253_lock;
    unsigned long flags;
    spin_lock_irqsave(&i8253_lock, flags);
    outb_p(0x34,PIT_MODE);          /* binary, mode 2, LSB/MSB, ch 0 */
    udelay(10);
    outb_p(LATCH & 0xff , PIT_CH0); /* Least Significant Byte */
    udelay(10);
    outb(LATCH >> 8 , PIT_CH0);    /* MSB */
    spin_unlock_irqrestore(&i8253_lock, flags);
}
```

La funzione outb() del C è equivalente all'istruzione assembly outb: copia il primo operando nella porta di I/O indicata dal secondo operando. La funzione outb_p() è simile alla precedente, ma introduce una pausa eseguendo un'istruzione nop per evitare confusione nell'hardware. La macro udelay() introduce anch'essa una pausa. La prima chiamata della funzione outb_p() è un'istruzione al PIT perché invii interrupt a una nuova frequenza. Le due chiamate successive forniscono la nuova frequenza al dispositivo. La costante LATCH a 16 bit è inviata alla porta di I/O 0x40 a 8 bit in due byte consecutivi. Il risultato è che il PIT invia interrupt alla frequenza di circa 1000 Hz.

Timer locale della CPU

L'APIC locale di cui sono dotati i microprocessori più recenti, fornisce un altro dispositivo di misura del tempo: il *timer locale della CPU*. E' simile al PIT in quanto può inviare un segnale singolo o interrupt periodici. Ci sono alcune differenze:

- il suo contatore è di 32 bit, mentre quello del PIT è di 16 bit; perciò può essere programmato per frequenze di interrupt molto basse (il contatore memorizza il numero di tick prima dell'invio dell'interrupt);
- invia interrupt solo al proprio processore, mentre il PIT invia un interrupt globale a tutte le CPU del sistema;
- il timer APIC si basa sul segnale di clock del bus (o sul segnale del bus dell'APIC nelle macchine più vecchie). Può essere programmato in modo da diminuire il contatore del timer ogni 1, 2, 4, 8, 16, 32, 64 o 128 segnali di clock del bus. Al contrario, il PIT, che fa uso del proprio segnale di clock, può essere programmato in modo più flessibile.

High Precision Event Timer

High Precision Event Timer (HPET) è un nuovo chip sviluppato da Intel e Microsoft. Anche se non è molto diffuso, Linux lo supporta. HPET fornisce un numero di interrupt che può venire sfruttato dal kernel. Il chip contiene fino a 8 contatori indipendenti a 32 o 64 bit. Ognuno è incrementato dal proprio segnale di clock, la cui frequenza deve essere di almeno 10 Mhz: perciò scatta almeno una volta ogni 100 nanosecondi. Ogni contatore è associato ad almeno 32 timer, ognuno dei quali è formato da un *comparatore* e un *match register* (registro di confronto). Il comparatore è un circuito che confronta il valore del contatore con quello del registro di confronto, e invia l'interrupt se verifica la corrispondenza. Alcuni dei timer possono essere abilitati ad inviare un interrupt periodico.

Il chip HPET può essere programmato per mezzo di registri mappati in memoria (come l'APIC). Il BIOS definisce la mappatura durante la fase di avvio e fornisce al kernel l'indirizzo iniziale. I registri di HPET permettono al kernel di leggere e scrivere i valori dei contatori e dei registri di confronto, di programmare interrupt singoli, abilitare o disabilitare interrupt periodici. Ci si aspetta che in futuro HPET rimpiazzi PIT sulle schede madri.

ACPI Power Management Timer

ACPI Power Management Timer (ACPI PMT) è un altro dispositivo incluso in quasi tutte le schede madri basate su ACPI. Il suo segnale di clock ha una frequenza fissa di circa 3,58 Mhz. Attualmente è un semplice contatore incrementato ad ogni tick di clock; per leggere il valore corrente del contatore, il kernel deve accedere ad una porta di I/O il cui indirizzo è definito dal BIOS in fase di inizializzazione.

ACPI PMT è preferibile a TSC se il sistema operativo o il BIOS possono regolare dinamicamente la frequenza o il voltaggio della CPU per risparmiare le batterie. Quando questo accade, la frequenza del TSC cambia – provocando distorsioni nella misura del tempo e altri effetti spiacevoli – mentre la frequenza di ACPI PMT non cambia. D'altra parte, la frequenza elevata di TSC è comoda per misurare piccoli intervalli di tempo. Tuttavia, se disponibile, è preferibile usare un HPET, a causa della sua architettura più articolata. A questo punto viene illustrato come Linux sfrutta i timer hardware.

Il sistema di misura del tempo in Linux

Linux deve eseguire diversi compiti correlati al tempo. Ad esempio, Linux periodicamente:

- aggiorna il tempo trascorso dall'accensione del sistema;
- aggiorna data e ora;
- determina, per ogni CPU, per quanto tempo è stato eseguito il processo corrente e lo sostituisce se ha superato il tempo che gli è stato assegnato;
- aggiorna le statistiche di uso delle risorse;
- controlla se l'intervallo di tempo associato ad ogni timer software è trascorso.

L'architettura di Linux per quanto riguarda la gestione del tempo è l'insieme delle strutture e delle funzioni correlate al flusso del tempo. Attualmente i sistemi multiprocessore hanno una gestione differente da quella dei sistemi uniprocessore:

- nei sistemi uniprocessore, le attività legate alla misura del tempo sono basate sugli interrupt inviati dal timer globale (o PIT o HPET);

- nei sistemi multiprocessore, le attività generali sono basate sugli interrupt del timer globale, mentre quelle specifiche per-CPU (riferite ad esempio al processo corrente) sono basate sugli interrupt del timer APIC locale.

Sfortunatamente, la distinzione tra i due casi è talvolta confusa. Ad esempio alcuni sistemi SMP non hanno APIC locali; oppure alcune schede madri hanno bug che rendono inutilizzabili gli interrupt timer locali. In questi casi il kernel si basa sul sistema di gestione tipo uniprocessore. Al contrario, alcuni sistemi uniprocessore recenti hanno un APIC locale. Questi casi ibridi non verranno trattati.

Il sistema di gestione del tempo in Linux dipende anche dalla disponibilità di TSC, di ACPI PMT e di HPET. Il kernel usa due funzioni di base: una per mantenere aggiornata l'ora corrente e una per contare il numero di nanosecondi trascorsi dall'ultimo secondo. Ci sono vari modi per ottenere quest'ultimo valore. Alcuni sono più precisi e sono applicabili se la CPU ha un TSC o HPET.

Strutture del sistema di misura del tempo

Vengono descritte le principali strutture dati dei sistemi 80x86.

L'oggetto timer

Per gestire i diversi dispositivi timer in modo uniforme, il kernel usa un "oggetto timer", un descrittore di tipo `timer_opts` che comprende il nome del timer e quattro metodi standard:

Nome del campo	descrizione
<code>name</code>	Stringa che identifica il nome del timer
<code>mark_offset</code>	Registra il momento esatto dell'ultimo tick; viene chiamata dal gestore di interrupt del timer
<code>get_offset</code>	Restituisce il tempo trascorso dall'ultimo tick
<code>monotonic_clock</code>	Restituisce il numero di nanosecondi trascorsi dall'inizializzazione del kernel
<code>delay</code>	Attende un numero dato di "cicli" (vedi "delay functions")

I metodi più importanti sono `mark_offset` e `get_offset`. Il primo viene chiamato dal gestore di interrupt timer e registra in una struttura dati il momento esatto dell'ultimo tick. Usando questo valore `get_offset` calcola il tempo trascorso dall'ultimo tick in microsecondi. Grazie a questi due metodi, il sistema di misura del tempo raggiunge una risoluzione inferiore al tick, cioè riesce a determinare il tempo con una precisione molto maggiore. L'operazione è chiamata interpolazione del tempo.

La variabile `cur_timer` contiene l'indirizzo dell'oggetto timer corrispondente al miglior dispositivo disponibile nel sistema. Inizialmente `cur_timer` punta a `timer_none`, un dispositivo fittizio usato in fase di inizializzazione. In questo momento la funzione `select_timer()` assegna a `cur_timer` l'indirizzo dell'oggetto timer adatto. La tabella mostra gli oggetti timer più usati in ordine di preferenza. La funzione sceglie HPET se disponibile; altrimenti sceglie ACPI PMT oppure TSCe, come ultima risorsa, l'onnipresente PIT. La colonna "interpolazione del tempo" elenca i dispositivi usati dai metodi `mark_offset` e `get_offset`; la colonna `delay` elenca quelli usati dal metodo omonimo.

Oggetto timer	Descrizione	Interpolazione	Delay
timer_hpet	High Precision Event Timer (HPET)	HPET	HPET
timer_pmtmr	ACPI Power Management Timer (ACPI PMT)	ACPI PMT	TSC
timer_tsc	Time Stamp Counter (TSC)	TSC	TSC
timer_pit	Programmable Interval Timer (PIT)	PIT	istruzione di ciclo
timer_none	Timer fittizio	nessuno	istruzione di ciclo

Da notare che il timer locale APIC non ha un oggetto timer corrispondente. Il motivo è che viene impiegato solo per generare interrupt periodici e non per ottenere una risoluzione superiore al tick.

La variabile jiffies

La variabile jiffies è un contatore che memorizza il numero di tick dall'accensione del sistema. E' incrementato di uno quando si verifica un interrupt timer, cioè ad ogni tick. E' una variabile a 32 bit, per cui arriva al suo limite in circa 50 giorni, un tempo relativamente breve per un server Linux. Il kernel gestisce facilmente l'overflow (traboccamento) di jiffies grazie alle macro time_after, time_after_eq, time_before e time_before_eq; esse forniscono il valore corretto anche quando si verifica un overflow.

Si potrebbe pensare che jiffies venga inizializzato a 0 all'avvio del sistema; invece viene inizializzato a 0xffffb6c20, che è il valore con segno di -300.000; perciò il contatore va in overflow 5 minuti dopo l'avvio. Questo è fatto di proposito, in modo che parti di codice del kernel con bug che non verificano il valore di jiffies si manifestano presto in fase di sviluppo e non passino inosservate nei kernel stabili.

In alcuni casi, però, il kernel ha bisogno del vero valore di tick a partire dall'avvio del sistema. Nell'architettura 80x86 il valore di jiffies è equiparato dal linker ai 32 bit meno significativi di un contatore a 64 bit chiamato jiffies_64. Con un tick ogni millisecondo, la variabile jiffies_64 va in overflow dopo alcune centinaia di milioni di anni, cioè non va mai in overflow.

Ci si può chiedere perché la variabile jiffies non sia stata dichiarata fin da subito come unsigned long long a 64 bit. Il problema è che l'accesso a variabili a 64 bit non può essere atomico, per cui ogni lettura sui 64 bit richiede una tecnica di sincronizzazione per evitare che il contatore sia aggiornato durante la lettura. Di conseguenza, un'operazione su una variabile a 64 bit è molto più lenta.

La funzione get_jiffies_64() legge e restituisce il valore di jiffies_64:

```
u64 get_jiffies_64(void)
{
    unsigned long seq;
    u64 ret;
    do {
        seq = read_seqbegin(&xtime_lock);
        ret = jiffies_64;
    } while (read_seqretry(&xtime_lock, seq));
    return ret;
}
```

L'operazione di lettura è protetta dal seqlock xtime_lock; la funzione inizia la lettura quando è sicura che la variabile non è stata aggiornata da un altro KCP.

L'aggiornamento del valore della variabile deve essere protetto con `write_seqlock(&xtime_lock)` e `write_sequnlock(&xtime_lock)`. Da notare che l'istruzione `++jiffies_64` incrementa anche la variabile a 32 bit `jiffies`, perché corrisponde alla metà inferiore di `jiffies_64`.

La variabile `xtime`

La variabile `xtime` contiene l'ora e la data correnti; è una struttura di tipo `timespec` con due campi:

`tv_sec`: contiene il numero di secondi trascorsi dalla mezzanotte del 1° gennaio 1970 (UTC);

`tv_nsec`: contiene il numero di nanosecondi trascorsi dall'ultimo secondo (va da 0 a 999.999.999).

La variabile viene aggiornata di solito una volta ogni tick, quindi 1000 volte al secondo. I programmi utente ottengono ora e data corrente dalla variabile `xtime`. Il kernel si riferisce spesso alla variabile, ad esempio quando aggiorna i timestamp degli inode.

Il `seqlock` `xtime_lock` evita le race conditions che possono capitare per accessi concorrenti alla variabile `xtime`; protegge anche la variabile `jiffies_64` e, in generale, varie regioni critiche del sistema di misura del tempo.

Sistema di misura del tempo nei sistemi uniprocessore

Tutte le attività legate al tempo in questi sistemi sono regolate dagli interrupt generati dal PIT sulla linea IRQ 0. Come al solito, alcune attività sono eseguite appena possibile dopo l'interrupt, mentre altre sono demandate a funzioni differibili.

Fase di inizializzazione

Durante l'inizializzazione del kernel, la funzione `time_init()` imposta il sistema di misura del tempo. Di solito² esegue le azioni seguenti:

1 – Inizializza la variabile `xtime`. Il numero di secondi trascorsi dal 1° gennaio 1970 viene letto dal Real Time Clock per mezzo della funzione `get_cmos_time()`. Il campo `tv_nsec` è impostato in modo che l'overflow della variabile `jiffies` coincida con un incremento del campo `tv_sec`, cioè cada allo scadere di un secondo.

2 – Inizializza la variabile `wall_to_monotonic`. Questa variabile è dello stesso tipo `timespec` di `xtime` e sostanzialmente contiene il numero di secondi e nanosecondi da aggiungere a `xtime` per ottenere un flusso monotonic (cioè sempre crescente) di tempo. Infatti, sia lo scatto dei secondi che la sincronizzazione con orologi esterni potrebbe alterare i campi di `xtime` in modo da non avere un incremento monotonic. Talvolta il kernel ha necessità di avere una fonte realmente monotonic.

3 – Se il kernel supporta HPET, chiama `hpet_enable()` per determinare se il firmware ACPI ha testato il chip e mappato i suoi registri in memoria. Se sì, `hpet_enable()` programma il primo timer del chip in modo che

² `time_init()` è eseguita prima di `mem_init()`, che inizializza le strutture della memoria. Sfortunatamente i registri di HPET sono mappati in memoria, per cui l'inizializzazione di HPET va fatta dopo l'esecuzione di `mem_init()`. Linux adotta questa soluzione: se il kernel supporta HPET, `time_init()` si limita ad avviare la funzione `hpet_time_init()`. Quest'ultima viene eseguita dopo `mem_init()` ed esegue le operazioni descritte sopra.

generi l'interrupt sulla linea IRQ 0 1000 volte al secondo. Altrimenti, se HPET non è disponibile, il kernel usa PIT; il chip è già stato programmato dalla funzione `init_IRQ()` a generare 1000 interrupt al secondo.

4 – Chiama `select_timer()` per scegliere il miglior dispositivo disponibile nel sistema, e impostare la variabile `cur_timer` con l'indirizzo dell'oggetto timer corrispondente.

5 – Chiama `setup_irq(0,&irq0)` per impostare l'interrupt gate corrispondente a IRQ0, la linea associata alla sorgente di interrupt timer del sistema (PIT o HPET). La variabile `irq0` è definita staticamente come:

```
struct irqaction irq0 = {timer_interrupt, SA_INTERRUPT, 0, "timer", NULL, NULL};
```

Da ora in poi la funzione `timer_interrupt()` sarà chiamata ad ogni tick con gli interrupt disabilitati, perché il campo `status` del descrittore di IRQ 0 ha il flag `SA_INTERRUPT` attivato.

Il gestore di interrupt timer

La funzione `timer_interrupt()` è la routine di servizio del PIT o di HPET; compie le seguenti azioni:

1 – Protegge le variabili correlate al tempo chiamando `write_seqlock()` su `xtime_lock`.

2 – Esegue il metodo `mark_offset` dell'oggetto `cur_timer`. Come spiegato in precedenza, ci sono quattro possibili casi:

- `cur_timer` punta all'oggetto `timer_hpet`: in questo caso la fonte degli interrupt è HPET. Il metodo `mark_offset` controlla che non sia stato perso nessun interrupt dall'ultimo tick; in questo caso, aggiorna `jiffies_64`. Poi il metodo registra il valore corrente del contatore periodico HPET.
- `cur_timer` punta all'oggetto `timer_pmtmr`: in questo caso il chip PIT è la fonte degli interrupt, ma il kernel usa APIC PMT per misurare il tempo con risoluzione superiore. Il metodo `mark_offset` controlla che non sia stato perso nessun interrupt dall'ultimo tick; in questo caso, aggiorna `jiffies_64`. Poi registra il valore corrente del contatore APIC PMT.
- `cur_timer` punta all'oggetto `timer_tsc`: in questo caso il chip PIT è la fonte degli interrupt, ma il kernel usa TSC per misurare il tempo con risoluzione superiore. Il metodo `mark_offset` controlla che non sia stato perso nessun interrupt dall'ultimo tick; in questo caso, aggiorna `jiffies_64`. Poi registra il valore corrente del contatore TSC.
- `cur_timer` punta all'oggetto `timer_pit`: in questo caso il chip PIT è la fonte degli interrupt, e non c'è altra sorgente di interrupt. Il metodo `mark_offset` non fa nulla.

3 – Chiama `do_timer_interrupt()` che esegue le seguenti azioni:

- incrementa di uno il valore di `jiffies_64`. L'operazione può essere eseguita in modo sicuro perché il KCP ha ancora il `seqlock xtime_lock` in scrittura;
- chiama `update_times()` per aggiornare data e ora del sistema e calcolare il carico corrente del sistema; queste operazioni verranno discusse in seguito.
- chiama `update_process_times()` per eseguire varie operazioni di contabilità per la CPU locale.
- chiama `profile_tick()`.

- se l'orologio di sistema è sincronizzato con un orologio esterno (`adjtimex()` è stata chiamata in precedenza) chiama `set_rtc_mmss()` una volta ogni 660 secondi (ogni 11 minuti) per regolare il Real Time Clock. Questa tecnica aiuta i sistemi in rete a sincronizzare gli orologi.

4 – Rilascia il seqlock `xtime_lock` chiamando `write_sequnlock()`.

5 – Restituisce il valore 1 per indicare che l'interrupt è stato gestito.

Sistema di misura del tempo nei sistemi multiprocessore

Questi sistemi si basano su due fonti diverse di interrupt timer; quelli generati da PIT oppure HPET e quelli generati dai timer locali delle CPU.

In Linux 2.6 gli interrupt del timer globale – sia PIT o HPET – scandiscono attività non correlate ad una specifica CPU, come gestire i timer software e mantenere aggiornato il sistema. Invece un interrupt della CPU locale scandisce attività correlate con la CPU locale, come monitorare per quanto tempo è stato eseguito il processo corrente e aggiornare le statistiche sull'uso delle risorse.

Fase di inizializzazione

Il gestore degli interrupt timer globali è inizializzato dalla funzione `time_init()` già descritta in precedenza. Il kernel riserva il vettore di interrupt 239 (0xef) per gli interrupt del timer locale. Durante l'inizializzazione del kernel, la funzione `apic_intr_init()` imposta l'interrupt gate della IDT corrispondente al vettore 239 con l'indirizzo del gestore di interrupt a basso livello `apic_timer_interrupt()`. In più ogni APIC deve essere istruito su quanto spesso generare un interrupt locale. La funzione `calibrate_APIC_clock()` calcola quanti segnali del clock del bus sono ricevuti dall'APIC locale della CPU che ha eseguito il boot durante un tick (1 ms). Questo valore esatto viene poi usato per programmare gli APIC locali in modo da generare un interrupt locale ad ogni tick. Questo compito è svolto dalla funzione `setup_APIC_timer()`, eseguita una volta per ogni CPU del sistema.

Tutti gli APIC locali sono sincronizzati perché sono basati sul segnale di clock del bus comune. Questo significa che il valore calcolato da `calibrate_APIC_clock()` per la CPU di boot va bene anche per le altre CPU.

Il gestore degli interrupt del timer globale

La versione SMP del gestore `timer_interrupt()` differisce da quella uniprocessore in pochi aspetti:

- la funzione `do_timer_interrupt()`, chiamata da `timer_interrupt()`, scrive nella porta di I/O del chip APIC per riconoscere l'IRQ del timer;
- la funzione `update_process_times()` non viene chiamata, perché esegue azioni per una specifica CPU;
- la funzione `profile_tick()` non viene chiamata, perché anch'essa svolge azioni per una specifica CPU.

Il gestore di interrupt del timer locale

Questo gestore si occupa delle attività collegate al tempo in una specifica CPU del sistema, soprattutto della profilatura del codice del kernel e del controllo del tempo di esecuzione del processo corrente. La funzione assembly `apic_timer_interrupt()` è equivalente a:

```
apic_timer_interrupt:
    pushl $(239-256)
    SAVE_ALL
    movl %esp, %eax
    call smp_apic_timer_interrupt
    jmp ret_from_intr
```

Il gestore a basso livello è molto simile ai gestori di altri interrupt. Il gestore di alto livello `smp_apic_timer_interrupt()` esegue le seguenti azioni:

- 1 – Ricava il numero logico della CPU (ad es. n).
- 2 – Incrementa il campo `apic_timer_irqs` dell'entry numero n dell'array `irq_stat`.
- 3 – Conferma l'interrupt all'APIC locale.
- 4 – Chiama `irq_enter()`.
- 5 – Chiama la funzione `smp_local_timer_interrupt()`.
- 6 – Chiama `irq_exit()`.

La funzione `smp_local_timer_interrupt` esegue le operazioni specifiche per CPU:

- 1 – Chiama la funzione `profile_tick()`.
- 2 – Chiama `update_process_times()` per controllare quanto a lungo è stato eseguito il processo corrente e aggiornare alcune statistiche della CPU locale.

L'amministratore di sistema può cambiare la frequenza di campionamento del profilatore del codice del kernel scrivendo in `/proc/profile`. Per applicare la modifica, il kernel cambia la frequenza di invio degli interrupt del timer locale. Comunque la funzione `smp_local_timer_interrupt()` continua a chiamare la `update_process_times()` ad ogni tick.

Aggiornamento di data ed ora

I programmi utente ricavano data ed ora dalla variabile `xtime`. Il kernel deve aggiornare periodicamente questa variabile in modo che il suo valore sia ragionevolmente preciso. La funzione `update_times()`, chiamata dal gestore di interrupt del timer globale, agisce in questo modo:

```
void update_times() {
    unsigned long ticks;
    ticks = jiffies - wall_jiffies;
    if(ticks) {
        wall_jiffies += ticks;
        update_wall_time();
    }
    calc_load(ticks);
}
```

}

Da ricordare che al momento dell'esecuzione di questa funzione, il seqlock `xtime_lock` è già stato acquisito in lettura. La variabile `wall_jiffies` contiene l'orario dell'ultimo aggiornamento di `xtime`. Il suo valore può essere inferiore a `jiffies - 1` perché alcuni interrupt del timer possono andare persi se gli interrupt rimangono disabilitati a lungo; in pratica, il kernel non aggiorna `xtime` a ogni tick. Comunque nessun tick va perso e nel lungo periodo, `xtime` contiene l'orario preciso. Il controllo degli interrupt perduti è svolto dal metodo `mark_offset` di `cur_timer`.

La funzione `update_wall_time()` chiama la `update_wall_time_one_tick()` per `ticks` volte consecutive; normalmente ad ogni chiamata viene aggiunto `1.000.000` a `xtime.tv_nsec`. Se il valore supera `999.999.999`, la funzione aggiorna anche il valore di `tv_sec`. Se è stata fatta una chiamata di sistema `adjtimex()`, per ragioni esposte in seguito, la funzione deve aggiustare leggermente il valore `1.000.000`, in modo che l'orologio rallenti o acceleri un po'. La funzione `calc_load()` viene descritta in seguito.

Aggiornamento delle statistiche del sistema

Il kernel deve periodicamente raccogliere una varietà di dati per:

- controllare il limite della risorsa di CPU del processo corrente;
- aggiornare le statistiche sul carico di lavoro della CPU locale;
- calcolare il carico dell'intero sistema;
- profilare il codice del kernel.

Aggiornamento delle statistiche della CPU locale

La funzione `update_process_times()` viene chiamata o dal gestore degli interrupt del timer globale nei sistemi uniprocessore o da quello locale nei sistemi multiprocessore, per aggiornare alcune statistiche del kernel:

1 – La funzione controlla per quanto tempo è stato eseguito il processo corrente. A seconda se il processo è in modalità utente o del kernel al momento dell'interrupt, chiama la funzione `account_user_time()` o `account_system_time()`. Ognuna delle funzioni esegue le azioni seguenti:

- aggiorna il campo `utime` (tick trascorsi in modalità utente) o `stime` (tick in modalità del kernel) del descrittore del processo corrente. In esso si trovano altri due campi chiamati `cutime` e `cstime` per contare il numero di tick trascorsi nelle due modalità di esecuzione dal processo figlio. Per ragioni di efficienza, questi campi non vengono aggiornati da `update_process_times()`, ma solo quando il genitore richiede informazioni sullo stato di uno dei suoi figli;
- controlla se è stato raggiunto il limite totale di tempo della CPU; se è così, invia i segnali `SIGXCPU` e `SIGKILL` a `current`;
- chiama `account_it_virt()` e `account_it_prof()` per controllare i timer del processo;
- aggiorna alcune statistiche contenute nella variabile per-CPU `kstat`.

2 – Chiama `raise_softirq()` per attivare il tasklet `TIMER_SOFTIRQ` sulla CPU locale.

3 – Se deve essere recuperata una vecchia versione di una struttura protetta da RCU, controlla se la CPU locale è passata attraverso uno stato quiescente e chiama `tasklet_schedule()` per attivare `rcu_tasklet`.

4 – Chiama la funzione `scheduler_tick()` che decrementa il contatore di “time slice” (il quanto di tempo di esecuzione riservato ad ogni processo) del processo corrente e controlla che non sia esaurito.

Controllo del carico di sistema

Ogni kernel Unix registra il carico di lavoro della CPU nel sistema. Queste statistiche sono usate da diverse utility di amministrazione come `top`. Un utente che lancia il comando `uptime` vede statistiche come il carico di lavoro relativo all'ultimo minuto, agli ultimi 5 e 15 minuti. In un sistema uniprocessore, il valore 0 indica che non vi sono processi attivi (oltre al processo 0 o `swapper`), il valore 1 indica che la CPU è impegnata al 100% con un solo processo e valori maggiori di uno indicano che la CPU è condivisa da più di un processo attivo³.

Ad ogni tick, `update_times` chiama `calc_load` che conta il numero di processi nello stato `TASK_RUNNING` e `TASK_UNINTERRUPTIBLE`, e usa questo numero per calcolare il carico di lavoro.

Profilatura del sistema

Linux include un profilatore di codice minimo chiamato `readprofile`, usato dagli sviluppatori per scoprire quanto tempo il kernel trascorre in modalità del kernel. Il profilatore identifica gli “hot spot” - cioè le parti del codice eseguite più di frequente. Questo controllo è molto importante per scoprire funzioni che richiedono ottimizzazione.

Il profilatore si basa su un semplice algoritmo Monte Carlo: ad ogni interrupt, il kernel determina se sta operando in modalità del kernel; se sì, ricava dallo stack il valore del registro `eip` e lo usa per scoprire che attività stava svolgendo prima dell'interrupt. Con l'andar del tempo, i campioni si accumulano dando origine agli hot spot.

La funzione `profile_tick()` raccoglie i dati per il profilatore. È chiamata da `do_timer_interrupt()` nei sistemi uniprocessore e da `smp_local_timer_interrupt()` nei sistemi multiprocessore. Per abilitare il profilatore, al kernel va passato in fase di boot il parametro `profile=N` dove 2^N indica le dimensioni dei frammenti di codice da profilare. I dati raccolti si trovano in `/proc/profile`. I contatori sono azzerati con una scrittura nello stesso file; nei sistemi multiprocessore, la scrittura ha anche l'effetto di variare la frequenza di campionamento. Gli sviluppatori del kernel non accedono al file direttamente; lo fanno per mezzo del comando `readprofile`.

Linux include anche un altro profilatore chiamato `oprofile`. Oltre ad essere più flessibile e configurabile del precedente, può essere usato per scoprire hot spot nel codice del kernel, delle applicazioni e nelle librerie di sistema. Quando viene usato `oprofile`, `profile_tick()` chiama `timer_notify()` per raccogliere i dati.

Controllo di NMI Watchdog

³ Linux include nel calcolo del carico tutti i processi in `TASK_RUNNING` e `TASK_UNINTERRUPTIBLE`; questi ultimi di solito sono molto pochi, per cui un carico elevato significa che la CPU è realmente occupata.

Nei sistemi multiprocessore, Linux offre un'altra funzione agli sviluppatori: un *sistema watchdog*⁴, che può servire a scovare i bug del kernel che provocano stalli nel sistema (system freeze). Per attivare il watchdog il kernel deve essere avviato con l'opzione `nmi_watchdog`.

Il sistema si basa su una intelligente caratteristica hardware degli APIC locali e di I/O: essi possono generare periodici interrupt NMI (non mascherabili) per ogni CPU. Poiché non sono mascherabili dalla istruzione `cli`, il sistema può registrare deadlock anche con gli interrupt disabilitati.

Di conseguenza, la CPU, indipendentemente dalle operazioni in corso, ad ogni tick esegue il gestore di interrupt NMI, che a sua volta chiama `do_nmi()`. Questa funzione ottiene il numero logico `n` della CPU, poi controlla il campo `apic_timer_irqs` della entry numero `n` di `irq_stat`. Se la CPU funziona normalmente, il valore del campo deve essere diverso da quello rilevato nel precedente interrupt NMI. Quando la CPU funziona normalmente, il valore della entry `n` è incrementato dal gestore di interrupt; se il contatore non viene incrementato, vuol dire che il gestore non è stato eseguito in occasione dell'ultimo tick. Questo non è un buon segno.

Quando il gestore di interrupt NMI registra uno stallo della CPU, fa suonare tutti gli allarmi: registra il fatto nei log di sistema, fa il dump dei registri della CPU e dello stack del kernel (kernel oops), poi uccide il processo corrente. Questo dà modo agli sviluppatori di capire cosa è andato storto.

Timer software e Delay Functions

Un *timer* è un strumento software che permette alle funzioni di essere chiamate in un momento futuro, dopo che è trascorso un certo intervallo di tempo; un *time-out* indica il momento in cui è scaduto l'intervallo di tempo associato al timer.

I timer sono ampiamente usati sia dal kernel che dai processi. La maggior parte dei driver di dispositivo usano i timer per rilevare anomalie - ad esempio i floppy usano i timer per spegnere il motore dopo che il floppy è rimasto inattivo per un certo periodo e le stampanti parallele li usano per rilevare condizioni erranee di stampa. Sono usati anche dai programmatori per avviare l'esecuzione di specifiche funzioni in un dato momento.

Implementare un timer è semplice: ognuno di essi contiene un campo che indica il momento in cui il timer scadrà. Questo valore è calcolato aggiungendo un adeguato numero di tick al valore corrente di jiffies. Il campo non cambia. Ogni volta che il kernel controlla il timer, lo confronta con il valore corrente di jiffies, e il timer scade quando il valore di jiffies è maggiore o uguale al valore memorizzato nel timer.

Linux considera due tipi di timer: dinamici e di intervallo. Il primo tipo è usato dal kernel, il secondo può essere impiegato da processi in modalità utente. Una particolarità per i timer in Linux: dato che sono gestiti da funzioni differibili che vengono eseguite anche molto tempo dopo la loro attivazione, il kernel non può garantire che la funzione legata al timer venga eseguita subito dopo il time-out. Può solo garantire che la funzione verrà eseguita o allo scadere o con un ritardo di alcune centinaia di millisecondi al massimo. Per questo motivo non sono adatti alle applicazioni in tempo reale, nelle quali i tempi di scadenza vanno tassativamente rispettati. Oltre ai timer, il kernel usa anche delay function, che eseguono un ciclo in attesa che trascorra un certo intervallo.

Timer dinamici

⁴ Letteralmente "cane da guardia", è un sistema di temporizzazione hardware che consente alla CPU di scoprire loop infiniti o deadlock - ndt

Questi timer possono essere creati e distrutti dinamicamente. Non esiste limite al numero di timer attivi. Un timer dinamico è contenuto nella struttura `timer_list`:

```
struct timer_list {
    struct list_head entry;
    unsigned long expires;
    spinlock_t lock;
    unsigned long magic;
    void (*function)(unsigned long);
    unsigned long data;
    tvec_base_t *base;
};
```

Il campo `function` contiene l'indirizzo della funzione da eseguire quando il timer scade. Il campo `data` contiene un parametro da passare alla funzione. Grazie al campo `data` è possibile definire una funzione di uso generale che gestisce i time-out di diversi driver di dispositivo; il campo `data` può contenere l'ID del dispositivo o altri dati significativi che possono essere usati dalla funzione per distinguere il dispositivo.

Il campo `expires` indica la scadenza del timer; il tempo è indicato in numero di tick trascorsi dall'avvio del sistema. I timer che hanno un valore inferiore o uguale al valore di `jiffies` sono considerati scaduti. Il campo `entry` è usato per inserire il timer in una delle liste circolari a doppio collegamento che raggruppano i timer in base al valore di `expires`. L'algoritmo usato da queste liste è descritto più avanti.

Per creare e attivare un timer dinamico, il kernel deve:

1 – Creare, se necessario, un oggetto `timer_list t`; questo può avvenire in vari modi:

- definendo una variabile globale statica nel codice;
- definendo una variabile locale entro una funzione; in questo caso l'oggetto sta nello stack in modalità del kernel;
- includendo l'oggetto in un descrittore allocato dinamicamente.

2 – Inizializzare l'oggetto tramite la funzione `init_timer(&t)`. Essa imposta il campo `t.base` a `NULL` e lo spin lock `t.lock` a "sbloccato".

3 – Inserire nel campo `function` l'indirizzo della funzione da avviare alla scadenza del timer e, se necessario, inserire nel campo `data` il parametro da passare alla funzione.

4 – Se il timer non è inserito in nessuna lista, assegnare un valore opportuno al campo `expires` e chiamare `add_timer(&t)` per inserirlo nella lista corrispondente.

5 – Se il timer è già inserito in una lista, aggiornare `expires` chiamando `mod_timer()`, che si incarica anche di spostare il timer nella lista opportuna.

Una volta che il timer è scaduto, il kernel rimuove l'elemento `t` dalla lista. Spesso comunque un processo deve rimuovere direttamente il timer per mezzo delle funzioni `del_timer()`, `del_timer_sync()` oppure `del_singleshot_timer_sync()`. Invece un processo quiescente deve essere risvegliato prima del time-out; in questo caso il processo può scegliere di distruggere il timer. Chiamare `del_timer()` su un timer già rimosso da una lista non crea problemi, per cui rimuovere il timer tramite la funzione `del_timer` stesso è considerata una buona pratica.

In Linux 2.6 un timer dinamico è legato alla CPU che l'ha attivato, quindi la funzione legata al timer viene eseguita sulla stessa CPU che ha eseguito `add_timer()` o che eseguirà `mod_timer()`. La funzione `del_timer()`, comunque, può disattivare qualunque timer dinamico, anche non collegato alla CPU locale.

Timer dinamici e race condition

Essendo asincroni, i timer sono soggetti a race condition. Ad esempio si consideri il caso di un timer la cui funzione agisce su una risorsa che può essere rilasciata (ad es. un modulo del kernel o una struttura di un file). Rilasciare la risorsa senza arrestare il timer può portare a corruzione dei dati se la funzione viene attivata quando la risorsa non esiste più. Perciò è buona norma arrestare il timer prima di rilasciare la risorsa:

```
del_timer(&t);
x_release_resources()
```

In un sistema multiprocessore questo codice non è sicuro perché la funzione del timer potrebbe essere in esecuzione su un'altra CPU quando viene chiamata `del_timer()`; di conseguenza le risorse potrebbero essere rilasciate prima dell'arresto del timer. Per evitare questa eventualità il kernel offre la funzione `del_timer_sync()`. Essa rimuove il timer dalla lista, poi controlla se la funzione relativa è in esecuzione su un'altra CPU; in questo caso `del_timer_sync()` attende fino a che la funzione ha terminato.

Questa funzione è piuttosto complessa e lenta, perché deve tenere conto del caso in cui la funzione del timer riattiva sé stessa. Se lo sviluppatore sa che la funzione del timer non riattiva mai il timer, può usare la funzione `del_singleshot_timer_sync()`, più semplice e veloce, per disattivare un timer e attendere il termine della funzione.

Esistono anche altri tipi di race condition: ad esempio, il modo corretto di modificare il campo `expires` di un timer già attivato consiste nell'utilizzare `mod_timer()` piuttosto che distruggerlo e crearne un altro. In quest'ultimo modo, due KCP che vogliono modificare `expires` nello stesso timer potrebbero mescolarsi con conseguenze negative. L'implementazione delle funzioni del timer è sicura per i multiprocessori per l'uso dello spin lock incluso in ogni oggetto `timer_list`; ogni volta che il kernel deve avere accesso a un timer dinamico, disabilita gli interrupt e acquisisce lo spin lock.

Strutture per i timer dinamici

Non è facile scegliere le strutture più adatte per implementare timer dinamici. Collegare tutti i timer in un'unica lista potrebbe degradare le performance del sistema, perché scandire una lunga lista ad ogni tick è oneroso. D'altra parte mantenere una lista ordinata potrebbe non essere più efficiente, perché le operazioni di inserimento ed eliminazione sono onerose anch'esse.

La soluzione adottata si basa su una struttura intelligente che divide i valori di `expires` in blocchi di tick e consente ai timer dinamici di filtrare in modo efficiente da liste con intervalli più ampi a liste con intervalli più ristretti. In più, nei sistemi multiprocessore l'insieme dei timer attivi è diviso tra le varie CPU.

La struttura principale per i timer dinamici è una variabile per-CPU chiamata `tvsec_bases`: comprende `NR_CPUS` elementi, una per ogni CPU. Ogni elemento è una struttura `tvec_base_t`, che include tutti i dati necessari a gestire i timer legati alla CPU:

```
typedef struct tvec_t_base_s {
    spinlock_t lock;
```

```

    unsigned long timer_jiffies;
    struct timer_list *running_timer;
    tvec_root_t tv1;
    tvec_t tv2;
    tvec_t tv3;
    tvec_t tv4;
    tvec_t tv5;
} tvec_base_t;

```

Il campo tv1 è una struttura di tipo tvec_root_t, che include un array vec di 256 list_head elementi, cioè liste di timer dinamici. Contiene tutti i timer che scadono nei successivi 255 tick.

I campi tv2, tv3, tv4 sono strutture di tipo tvec_t che consistono di un array vec di 64 elementi list_head. Queste liste contengono tutti i timer che scadono nei successivi $2^{14}-1$, $2^{20}-1$, $2^{26}-1$ tick rispettivamente. Tv5 è identico ai precedenti, tranne che l'ultimo elemento del vettore è una lista che include tutti i timer con tempi di scadenza molto lunghi, e non richiede di essere riempita da elementi di un'altra lista.

Il campo timer_jiffies rappresenta il tempo di scadenza più breve del timer ancora da controllare: se coincide con il valore di jiffies, non esiste lavoro arretrato per la funzione differibile; se è inferiore a jiffies, bisogna occuparsi di liste di timers che si riferiscono a tick precedenti. Il campo è impostato a jiffies all'avvio del sistema e incrementato solo dalla funzione run_timer_softirq() descritta in seguito. Da notare che timer_jiffies potrebbe perdere molto terreno rispetto a jiffies se le funzioni differibili non vengono eseguite per lungo tempo – ad esempio se esse sono disabilitate a lungo perché è stato ricevuto un gran numero di interrupt. Nei sistemi multiprocessore il campo running_timer punta alla struttura timer_list dei timer dinamici gestiti attualmente dalla CPU locale.

Gestione dei timer dinamici

A dispetto della struttura ben congegnata, la gestione dei timer è onerosa in termini di tempo e non può essere eseguita dal gestore di interrupt. In Linux 2.6 viene eseguita da una funzione differibile, in particolare dal softirq TIMER_SOFTIRQ. La funzione run_timer_softirq() è la funzione differibile associata a questo softirq. Essa esegue le seguenti azioni:

1 – Memorizza nella variabile locale *base* l'indirizzo della struttura tvec_base_t associata alla CPU locale.

2 – Acquisisce lo spin lock base->lock e disabilita gli interrupt locali.

3 – Inizia un ciclo while che termina quando base->timer_jiffies diviene superiore a jiffies. In ogni esecuzione del ciclo esegue le seguenti azioni:

- calcola l'indice della lista in base->tv1 che contiene il prossimo timer da gestire:

```
index = base->timer_jiffies & 255;
```

- se l'indice è zero, tutte le liste in base->tv1 sono state controllate, per cui sono vuote: perciò la funzione fa filtrare i timer chiamando cascade():

```

if(!index && (!cascade(base, &base->tv2, (base->timer_jiffies>> 8)&63)) &&
    (!cascade(base, &base->tv3, (base->timer_jiffies>> 14)&63)) &&
    (!cascade(base, &base->tv4, (base->timer_jiffies>> 20)&63)))
    cascade(base, &base->tv5, (base->timer_jiffies>> 26)&63);

```

La prima chiamata di `cascade` riceve come argomento l'indirizzo di `base` e di `base->tv2`, e l'indice della lista in `base->tv2` che include i timer che scadranno nei prossimi 256 tick. Questo indice è determinato esaminando i bit appropriati del valore di `base->timer_jiffies`. `Cascade()` sposta tutti i timer nella lista `base->tv2` nella lista di `base->tv1`; poi, restituisce un valore positivo, a meno che tutte le liste `base->tv2` siano ora vuote. Se è così, `cascade()` viene chiamata ancora per riempire `base->tv2` con i timers della lista `base->tv3`, e così via;

- incrementa di 1 `base->timer_jiffies`;
- per ogni timer nella lista `base->tv1.vec[index]` esegue la corrispondente funzione. In particolare per ogni elemento `timer_list` nella lista esegue le azioni seguenti:
 - rimuove `t` dalla lista `base->tv1`;
 - nei sistemi multiprocessore, imposta `base->running_timer` a `&t`;
 - imposta `t.base` a `NULL`;
 - rilascia lo spin lock `base->lock` e abilita gli interrupt;
 - esegue la funzione del timer `t.function` passandole l'argomento `data`;
 - acquista lo spin lock `base->lock` e disabilita gli interrupt;
 - continua con il timer successivo nella lista, se esiste.
- Tutti i timer della lista sono stati gestiti. Continua con la prossima iterazione del ciclo `while`.

4 – Il ciclo `while` è terminato; significa che tutti i timer scaduti sono stati gestiti. Nei sistemi multiprocessore imposta `base->running_timer` a `NULL`.

5 – Rilascia lo spin lock `base->lock` e abilita gli interrupt locali.

Dato che il valore di `jiffies` e `timer_jiffies` di solito coincide, il ciclo `while` viene spesso eseguito una sola volta. In generale viene eseguito `jiffies – base->timer_jiffies + 1` volte consecutive. In più, se viene ricevuto un interrupt timer durante l'esecuzione di `run_timer_softirq()`, i timer che scadono in occasione del tick vengono tenuti in considerazione, perché la variabile `jiffies` è incrementata in modo asincrono dal gestore di interrupt del timer globale.

Da notare che `run_timer_softirq()` disabilita gli interrupt e acquisisce lo spin lock `base->lock` prima di entrare nel ciclo esterno; gli interrupt sono abilitati e lo spin lock rilasciato subito prima di chiamare ogni funzione del timer, fino al termine. Questo assicura che le strutture dei timer dinamici non vengano corrotte da kernel control path intercalati tra loro.

Questo algoritmo complesso garantisce ottime prestazioni. Si consideri che il `softirq TIMER_SOFTIRQ` venga eseguito subito dopo l'interrupt corrispondente. Poi, in 255 interrupt su 256 (99,6% dei casi), la funzione `run_timer_softirq()` esegue solo le funzioni dei timer scaduti, se ce ne sono. Per rifornire periodicamente `base->tv1.vec` sono sufficienti 63 time out su 64 per ripartire una lista `base->tv2` nelle 256 liste di `base->tv1`. L'array `base->tv2.vec` deve essere rifornito nello 0,006% dei casi (cioè ogni 16,4 secondi). `base->tv3.vec` viene rifornito ogni 17 minuti e 28 secondi; `base->tv4.vec` ogni 18 ore e 38 minuti; infine `base->tv5.vec` non viene mai rifornito.

Un'applicazione di timer dinamici: la chiamata di sistema nanosleep()

Per mostrare come i risultati delle attività fin qui illustrate sono utilizzati nel kernel, viene mostrato un esempio della creazione e dell'uso di un time out di un processo.

Viene considerata la routine di servizio della chiamata di sistema nanosleep(), cioè sys_nanosleep(), che riceve come parametro un puntatore ad una struttura timespec, e sospende il processo che la chiama fino a che l'intervallo di tempo specificato non scade. La routine chiama dapprima copy_from_user() per copiare il valore contenuto nella struttura timespec nella variabile locale t. Se timespec indica ritardo non nullo, la funzione esegue il codice seguente:

```
current->state = TASK_INTERRUPTIBLE;
remaining = schedule_timeout(timespec_to_jiffies(&t) + 1);
```

La funzione timespec_to_jiffies converte in tick l'intervallo contenuto in timespec. Per sicurezza, sys_nanosleep() aggiunge un tick a questo valore. Il kernel implementa i time out dei processi usando timer dinamici. Essi compaiono in schedule_timeout(), che esegue le istruzioni seguenti:

```
fastcall signed long __sched schedule_timeout(signed long timeout)
{
    struct timer_list timer;
    unsigned long expire;
    switch (timeout)
    {
    case MAX_SCHEDULE_TIMEOUT:
        /*
         * These two special cases are useful to be comfortable
         * in the caller. Nothing more. We could take
         * MAX_SCHEDULE_TIMEOUT from one of the negative value
         * but I'd like to return a valid offset (>=0) to allow
         * the caller to do everything it want with the retval.
         */
        schedule();
        goto out;
    default:
        /*
         * Another bit of PARANOID. Note that the retval will be
         * 0 since no piece of kernel is supposed to do a check
         * for a negative retval of schedule_timeout() (since it
         * should never happens anyway). You just have the printk()
         * that will tell you if something is gone wrong and where.
         */
        if (timeout < 0)
        {
            printk(KERN_ERR "schedule_timeout: wrong timeout "
                "value %lx from %p\n", timeout,
                __builtin_return_address(0));
            current->state = TASK_RUNNING;
        }
    }
}
```

```

        goto out;
    }
}
expire = timeout + jiffies;
init_timer(&timer);
timer.expires = expire;
timer.data = (unsigned long) current;
timer.function = process_timeout;
add_timer(&timer);
schedule();
del_singleshot_timer_sync(&timer);
timeout = expire - jiffies;
out:
    return timeout < 0 ? 0 : timeout;
}

```

Quando viene chiamata `schedule()`, un altro processo viene selezionato per l'esecuzione; quando il primo riprende, la funzione rimuove il timer dinamico. Nell'ultima istruzione, la funzione restituisce 0 se il time-out è scaduto, oppure il numero di tick rimasti prima dello scadere se il processo è stato risvegliato per qualche altra ragione.

Quando il time-out scade, viene eseguita la funzione del timer:

```

void process_timeout(unsigned long __data) {
    wake_up_process((task_t *)__data);
}

```

`process_timeout()` riceve come parametro il puntatore al descrittore del processo contenuto nel campo `data` dell'oggetto `timer`. Come risultato, il processo viene risvegliato. Una volta risvegliato, il processo continua l'esecuzione di `sys_nanosleep()`. Se il valore restituito da `schedule_timeout()` indica che il time-out è scaduto (valore 0), la chiamata di sistema termina. Altrimenti essa viene automaticamente riavviata.

Delay functions

I timer software sono poco utili quando il kernel deve attendere per breve tempo, cioè meno di pochi millisecondi. È il caso di alcuni driver di dispositivo che devono attendere per un numero predefinito di millisecondi che l'hardware completi un'operazione. Poiché un timer software ha un costo non indifferente di impostazione e un tempo di attesa minimo di un millisecondo, i driver di dispositivo non riescono a sfruttarli convenientemente. In questi casi il kernel usa le funzioni `udelay()` e `ndelay()`: la prima riceve come parametro un tempo in microsecondi e ritorna allo scadere dell'intervallo; la seconda è simile, ma riceve il parametro in nanosecondi.

Essenzialmente le due funzioni sono così definite:

```

void udelay(unsigned long usecs) {
    unsigned long loops;
    loops = (usecs*HZ*current_cpu_data.loops_per_jiffy)/1000000;
    cur_timer->delay(loops);
}
void ndelay(unsigned long nsecs) {
    unsigned long loops;
    loops = (nsecs*HZ*current_cpu_data.loops_per_jiffy)/1000000000;
    cur_timer->delay(loops);
}

```

Entrambe le funzioni si basano sul metodo delay dell'oggetto timer cur_timer, che riceve come parametro un intervallo di tempo in "loop". La durata di un loop, dipende dall'oggetto timer referenziato da cur_timer:

- se punta a timer_hpet, timer_pmtmr, e timer_tsc, un loop corrisponde a un ciclo di CPU, cioè l'intervallo tra due segnali di clock della CPU;
- se punta a timer_none o timer_pit, corrisponde alla durata di una singola iterazione di una istruzione di ciclo.

Durante l'inizializzazione, dopo che cur_time è stato impostato da select_timer(), il kernel esegue calibrate_delay(), che determina quanti loop sono contenuti in un tick. Questo valore viene salvato nella variabile current_cpu_data.loops_per_jiffy, in modo da poter essere usato da udelay() e ndelay() per convertire micro e nanosecondi in loop. Ovviamente il metodo cur_timer->delay() usa i circuiti HPET o TSC, se disponibili, per una accurata misurazione del tempo. Altrimenti il metodo esegue loop iterazioni di una istruzione di ciclo.

Chiamate di sistema correlate alla misura del tempo

Varie chiamate di sistema permettono ai processi in modalità utente di leggere e modificare data e ora e di creare timer.

time() e gettimeofday()

I processi in modalità utente possono leggere data e ora per mezzo di:

time() - restituisce il numero di secondi trascorsi dalla mezzanotte del 1° gennaio 1970 (UTC);

gettimeofday() - restituisce in una struttura timeval il numero di secondi dalla mezzanotte del 1° gennaio 1970 e il numero di microsecondi trascorsi dall'ultimo secondo (un'altra struttura chiamata timezone non è impiegata). Sostituisce time() che è ancora inserita in Linux per compatibilità. Un'altra funzione impiegata è ftimer(), non più implementata come chiamata di sistema, che restituisce il numero di secondi trascorsi dalla mezzanotte del 1° gennaio 1970 e il numero di millisecondi nell'ultimo secondo.

La chiamata di sistema `gettimeofday()` è implementata per mezzo della funzione `sys_gettimeofday()`. Per calcolare data e ora correnti, questa funzione chiama `do_gettimeofday()`, che esegue le seguenti azioni:

1 – Acquisisce il seqlock `xtime_lock`.

2 – Determina il numero di microsecondi trascorsi dall'ultimo interrupt timer chiamando il metodo `get_offset` dell'oggetto `cur_timer`:

```
usec = cur_timer->getoffset();
```

Ci sono quattro possibili casi:

- se `cur_timer` punta all'oggetto `timer_hpet`, il metodo confronta il valore corrente del contatore di HPET con il valore dello stesso contatore salvato nell'ultima esecuzione del gestore di interrupt;
- se punta all'oggetto `timer_pmtmr`, confronta il valore del contatore di ACPI PMT con il valore dello stesso contatore salvato nell'ultima esecuzione del gestore di interrupt;
- se punta all'oggetto `timer_tsc`, confronta il valore del contatore di TSC con il valore dello stesso contatore salvato nell'ultima esecuzione del gestore di interrupt;
- se punta all'oggetto `timer_pit`, legge il valore corrente del contatore di PIT per calcolare il numero di microsecondi trascorsi dall'ultimo interrupt del PIT.

3 – Se alcuni interrupt sono stati persi, la funzione aggiunge a `usec` il ritardo corrispondente:

```
usec += (jiffies - wall_jiffies) * 1000;
```

4 – Aggiunge a `usec` i microsecondi trascorsi nell'ultimo secondo:

```
usec += (xtime.tv_nsec / 1000);
```

5 – Copia il contenuto di `xtime` nel buffer dello spazio utente specificato dal parametro della chiamata di sistema `tv`, aggiungendo il valore di `usec` al campo dei microsecondi:

```
tv->tv_sec = xtime->tv_sec;  
tv->tv_usec = xtime->tv_usec + usec;
```

6 – Chiama `read_seqretry()` sul seqlock `xtime_lock` e ritorna al punto 1 se un altro kernel control path ha acquisito contemporaneamente il lock in scrittura.

7 – Controlla l'overflow nel campo dei microsecondi, correggendo anche il campo dei secondi se necessario:

```
while (tv->tv_usec >= 1000000) {  
    tv->tv_usec -= 1000000;  
    tv->tv_sec++;  
}
```

I processi in modalità utente con privilegi di root possono modificare data e ora correnti o con l'obsoleta `stime()` o con `gettimeofday()`. La `sys_settimeofday()` chiama `do_settimeofday()`, che esegue operazioni complementari a quelle della precedente.

Entrambe le chiamate di sistema modificano il valore di `xtime` lasciando il registro RTC invariato. Il nuovo valore di orario viene perso allo spegnimento del sistema, a meno che l'utente non esegua il programma `clock` per modificare il valore di RTC.

adjtimex()

Anche se a causa della deriva degli orologi tutti i sistemi si discostano dall'ora esatta, cambiare bruscamente l'ora è una seccatura amministrativa e un comportamento rischioso. Si consideri un gruppo di programmatori impegnati su un programma di grandi dimensioni; la ricompilazione dei file oggetto non aggiornati dipende dai timestamp. Un cambiamento dell'orario del sistema può confondere il programma `make` e portare ad una compilazione non corretta. Mantenere gli orologi sincronizzati è importante anche nell'implementazione di un filesystem distribuito di rete. In questo caso è saggio sincronizzare gli orologi dei PC in rete, per far sì che i timestamp associati agli inode siano coerenti. Perciò i sistemi sono spesso configurati per usare un protocollo di sincronizzazione come Network Time Protocol (NTP) per variare gradualmente l'orario ad ogni tick. Questa utility dipende dalla chiamata di sistema `adjtimex()`.

La chiamata di sistema è presente in diverse varianti di Unix, anche se non andrebbe impiegata in programmi portabili. Riceve come parametro un puntatore ad una struttura `timex`, aggiorna i parametri del kernel a partire dai valori del campo `timex` e restituisce la stessa struttura con i valori correnti del kernel. Questi valori sono usati da `update_wall_time_one_tick()` per variare leggermente il numero di microsecondi aggiunti a `xtime.tv_usec` a ogni tick.

setitimer() e alarm()

Linux consente ai processi in modalità utente di attivare timer speciali chiamati *interval timer* (che non hanno nulla a che vedere con il chip PIT). Questi timer software inviano periodicamente segnali Unix ai processi. E' possibile anche attivare un timer perché invii un solo segnale dopo un certo intervallo. Ogni interval timer è caratterizzato da:

- la frequenza con cui sono inviati i segnali; un valore nullo indica che deve essere inviato un solo segnale;
- il tempo che resta all'invio del segnale

A questi timer si applicano le avvertenze riportate in precedenza: essi garantiscono l'invio del segnale dopo la scadenza del timer, ma senza poter prevedere il momento esatto. Gli interval timer sono attivati attraverso la chiamata di sistema POSIX `setitimer()`. Il primo parametro specifica quale delle politiche seguenti deve essere adottata:

`ITIMER_REAL`: tempo effettivo trascorso; il processo riceve un segnale `SIGALRM`.

ITIMER_VIRTUAL: tempo trascorso dal processo in modalità utente; il processo riceve un segnale SIGVTALRM.

ITIMER_PROF: tempo trascorso dal processo in modalità utente e del kernel; il processo riceve un segnale SIGPROF.

Gli interval timer possono essere a segnale singolo o periodico. Il secondo parametro di `setitimer()` punta ad una struttura di tipo `itimerval` che specifica la durata iniziale del timer in secondi e nanosecondi e la durata da usare quando il timer è automaticamente riavviato (oppure zero se il timer è a segnale singolo). Il terzo parametro è un puntatore opzionale a una struttura `itimerval` che contiene i parametri precedenti.

Per implementare un timer per ognuna delle politiche citate, il descrittore di processo contiene tre paia di campi:

- `it_real_incr` e `it_real_value`
- `it_virt_incr` e `it_virt_value`
- `it_prof_incr` e `it_prof_value`

Il primo campo di ogni coppia contiene l'intervallo in tick tra due segnali; l'altro, il valore corrente del timer.

Il timer di tipo `ITIMER_REAL` è implementato usando timer dinamici, perché il kernel deve inviare segnali al processo anche se esso non è in esecuzione. Perciò ogni descrittore di processo contiene un oggetto timer dinamico chiamato `real_timer`. La chiamata di sistema `setitimer()` inizializza il campo `real_timer`, poi chiama `add_timer()` per inserirlo nella lista appropriata. Quando il timer scade, il kernel esegue `it_real_fn()`, la quale invia un segnale `SIGALRM` al processo; poi, se `it_real_incr` è diverso da zero, imposta di nuovo il campo `expires`, riattivando il timer.

I timer di tipo `ITIMER_VIRTUAL` e `ITIMER_PROF` non richiedono timer dinamici, perché possono essere aggiornati mentre il processo è in esecuzione. Le funzioni `account_it_virt()` e `account_it_prof()` sono chiamate da `update_process_times()`, che viene chiamata o dal gestore dell'interrupt timer PIT o dal gestore degli interrupt locali (SMP). Perciò i due timer sono aggiornati ad ogni tick e, alla scadenza, il segnale viene inviato al processo.

La chiamata di sistema `alarm()` invia un segnale `SIGALRM` al processo chiamante quando scade un dato intervallo di tempo. E' molto simile a `setitimer()` chiamata col parametro `ITIMER_REAL`, perché usa timer dinamici di tipo `real_timer` inclusi nel descrittore di processo. Perciò le due funzioni non possono essere usate contemporaneamente.

Chiamate di sistema per timer POSIX

Lo standard POSIX 1003.1b ha introdotto nuovi tipi di timer software per applicazioni in modalità utente, in particolare multithread e real-time. Sono spesso indicati come timer POSIX. Ogni implementazione di timer POSIX deve offrire ai programmi alcuni orologi POSIX, cioè misuratori di tempo virtuali con risoluzione e proprietà predefinite. Quando un'applicazione vuole usare un timer di questo tipo, crea una nuova risorsa timer specificando uno degli orologi definiti dallo standard come base.

Il kernel 2.6 offre due tipi di orologi POSIX:

`CLOCK_REALTIME`: orologio virtuale che rappresenta l'orologio del tempo reale del sistema, cioè il valore di `xtime`. La risoluzione restituita dalla chiamata di sistema `clock_getres()` è 999.848 nanosecondi, che corrisponde a circa 1000 aggiornamenti di `xtime` al secondo.

`CLOCK_MONOTONIC`: rappresenta l'orologio del tempo reale del sistema depurato da ogni distorsione indotta dalla sincronizzazione con altre sorgenti esterne. E' dato dalla somma delle due variabili `xtime` e `wall_to_monotonic`. La risoluzione è identica al precedente.

Linux implementa questi orologi per mezzo di timer dinamici, simili a quelli di tipo `ITIMER_REAL`. I timer POSIX però sono molto più flessibili e affidabili di quelli standard. Un paio di differenze sono le seguenti:

- quando un timer tradizionale scade, il kernel invia un segnale `SIGALRM` al processo che lo ha attivato; nel caso di un timer POSIX, può essere inviato un qualunque segnale ad un singolo thread oppure all'intera applicazione multithread. Il kernel può anche eseguire una funzione di notifica, oppure non fare nulla (la gestione dell'evento dipende da una libreria in modalità utente);
- quando un timer tradizionale scade più volte ma il processo non può ricevere il segnale (ad es. perché non è in esecuzione o il segnale è bloccato), solo il primo `SIGALRM` viene ricevuto: i successivi vanno perduti. Nel caso del timer POSIX accade la stessa cosa, ma il processo può chiamare `timer_getoverrun()` per ottenere il numero di volte che il timer è scaduto dopo l'invio del primo segnale.