10. GESTIONE DEL REAL TIME CLOCK

Un clock è un dispositivo hardware che emette impulsi ad alta precisione, solitamente onde quadre, ad intervalli regolari. Oltre al sistema centralizzato del clock che controlla la velocità con cui la CPU esegue le istruzioni, un computer può avere un real-time clock (clock a tempo reale) e un time-of-day clock (clock giornaliero), che sono attinenti tra loro, ma non uguali.

Analogamente ad un orologio da polso, un time-of-day clock è un cronometro. E' composto da un clock (orologio) accurato che manda un numero intero d'impulsi ogni secondo e da un contatore che registra gli impulsi. I programmi leggono il contatore per determinare l'ora e la data corrente, e i programmi privilegiati scrivono nel contatore per settare l'ora. Raramente è necessario resettarlo perché il time-of-days clocks continua correttamente il conteggio finché riceve corrente, indipendentemente dal fatto che la CPU sia carica di lavoro o sia ferma. (La storia è ricca d'avvenimenti sulla confusione introdotta da utenti che hanno settato il time-of-day clock dopo una mancanza di corrente.)

10.1 Il meccanismo del real-time clock

Diversamente dal time-of-day clock, un real-time clock non registra gli impulsi ne tiene traccia della data. Invece, pulsa regolarmente un numero di volte al secondo, avvertendo la CPU ogni volta che arriva un impulso, di eseguire un'interruzione. Così, una differenza tra i due clock è che uno controlla la CPU mentre l'altro è controllato dalla CPU:

> *La CPU quando vuole legge dal time-of-day clock per ottenere l'ora e la data corrente; il real-time clock forza la CPU ad eseguire un'interruzione ogni volta che si presenta un impulso.*

I due clock sono inoltre distinguibili da come contano gli impulsi. Il clock real-time non possiede un contatore, e non accumula le interruzioni. La responsabilità del conteggio delle interruzioni è del sistema:

> *Se la CPU impiega troppo tempo per servire un'interruzione del clock real-time, o se lavora con le interruzioni disabilitate per più di un ciclo di clock, essa perderà l'interruzione.*

Ovviamente, il sistema deve essere progettato per servire velocemente le interruzioni di clock. L'hardware aiuta dando la più alta priorità alle interruzioni del clock. Alcuni clock hardware memorizzano alcune interruzioni permettendo alla CPU di attendere più di un ciclo di clock. Così come, alcuni processori più lenti non possono permettersi di chiamare una procedura in C in ogni interruzione del clock, o dovrebbero perdere molto del loro tempo gestendo le interruzioni del clock.

10.2 Le interruzioni del real-time clock

Il tipico clock del processore PC opera a 4.77272MHz. I circuiti sulla scheda del processore dividono la frequenza in 4, producendo un segnale di 1.19318MHz per il timer chip 8253. Il timer chip, a sua volta, divide il segnale per 65536 per produrre una frequenza di 18.20648Hz, che è la

frequenza di un interruzione nel real time clock. Il clock fa avvenire un interruzione attraverso il vettore CLK_INT (tipo 08H) e spedite attraverso la tabella *intmap* descritta nel Capitolo 9.

Certi sistemi hanno un real-time clock più veloce – che ha una frequenza di interruzioni più elevata. Alcuni sistemi non possono permettersi di chiamare una procedura C in ogni interruzione del clock, o impiegherebbero molto del loro tempo trattando le interruzioni del clock. Come può evitare un sistema simile di spendere tutto il suo tempo processando interruzioni del real-time clock? La risposta è che la frequenza del clock deve essere tarata sul sistema. Rallentare il clock è un ottimizzazione significante perché permette al progettista la creazione di importanti funzioni sacrificando la precisione.

Idealmente, il clock hardware potrebbe essere rallentato, ma solitamente è sconveniente o impossibile. Invece, si può costruire un particolare gestore delle interruzioni del clock per simulare un clock più lento. La strada più semplice per simulare un clock più lento consiste nel dividere la frequenza di clock. Per esempio, il clock real-time nel Digital Equipment Corporation LSI 11/2 – il processore su cui inizialmente Xinu fu sviluppato – genera 60 impulsi al secondo, una frequenza che potrebbe richiedere molto tempo per la gestione delle interruzioni. Per simulare un clock più lento, il gestore del clock può essere programmato per ignorare cinque interruzioni di clock prima di processarne una, dividendo effettivamente la frequenza per 6. In pratica, il gestore del clock deve accettare tutte le interruzioni del clock, ma decrementa semplicemente un contatore e ritorna velocemente cinque volte su sei. Come risultato, il corpo principale del codice del gestore delle interruzioni del clock del LSI 11/2 è eseguito solo dieci volte al secondo.

La frequenza delle interruzioni del gestore degli interrupt del clock che sono effettivamente eseguite è comunemente conosciuto come *tick rate*, e noi sappiamo che un *tick* avviene ogni volta che il gestore delle interruzioni del clock principale è chiamato. Nel PC-Xinu, il codice di interruzione del clock principale è eseguito in ogni interruzione del clock, cosi il suo tick rate è lo stesso di quello delle interruzioni del real-time clock – approssimativamente 18.2Hz. Sebbene questa frequenza è abbastanza bassa da evitare un sovraccarico del gestore di interruzioni, la scelta della frequenza non è buona finché non è un numero intero di ticks al secondo. Questa frequenza crea difficoltà per le attività manuali che richiedono una risoluzione misurata in secondi o in multipli di secondi.

10.3 L'uso di un real-time clock

Il sistema operativo usa internamente un real-time clock per limitare la quantità di tempo di un processo in esecuzione, così come esternamente, fornisce i programmi utente con servizi come i timed delays (ritardi di tempo). Solitamente, il sistema mantiene una lista di "eventi", ordinati in base al tempo in cui sono arrivati. Ogni volta che arriva un interruzione del clock, esamina la lista degli eventi ed avvia ogni evento per cui l'attesa è finita.

Il nostro progetto permette che due tipi di evento vengano schedulati in futuro. Il primo è *preemption* (prerilascio): quando garantisce i servizi CPU al processo, il sistema schedula un evento preemption per impedire che il processo giri per sempre. Il secondo è un *timed delay* (ritardo temporizzato): quando un processo richiede un ritardo di tempo, leva i processi dallo stato corrente e schedula un evento di *wakeup* (risveglio) per farli partire al momento giusto.

Il sistema usa preemption per garantire che i processi di uguale priorità ricevano i servizi dal round-robin (spiegato nella politica di schedulazione nel capitolo 4). Ricorda che ogni volta che *resched* cambia di contesto resetta la variabile preempt a QUANTUM (QUANTUM è una costante simbolica definita nel file *kernel.h*). Il clock interrupt dispatcher decrementa preempt ogni tick di clock, chiamando *resched* quando raggiunge lo zero. Il processo attualmente in esecuzione è sempre il processo a più alta priorità che è nelle condizioni per accedere ai servizi CPU, ma gli altri di uguale priorità potrebbero aspettare nella lista ready. Se lo sono, *resched* colloca il processo corrente nella lista dei processi pronti dopo gli altri con la stessa priorità, e passa al primo processo

nella lista. Così, se k processi ad uguale priorità hanno bisogno della CPU, tutti i k processi si eseguono per, al più, QUANTUM clock ticks prima che ciascuno di essi riceva altri servizi.

Il valore della costante simbolica QUANTUM da la *granularità del preemption*; può essere cambiata prima che il sistema sia compilato. Settando un piccolo QUANTUM, come 2 o 3, si crea una granularità bassa, rischedulando ogni poche decine di secondo. Una bassa granularità tende a far lavorare tutti i processi ad eguale priorità precedendo approssimativamente allo stesso passo. Ma una bassa granularità introduce molto overhead perché forza la procedura di interruzione del clock a chiamare spesso *resched*. Settando QUANTUM elevato, come 100, si riduce l'overhead del cambio di contesto, ma si crea un'alta granularità di cambio di contesto. Il potenziale svantaggio di un alta granularità è che il processo può essere eseguito per molti secondi prima di passare ad un altro di uguale priorità.

Quando si fermano, i processi raramente usano la CPU abbastanza a lungo da giustificare preemption. Un processo chiama *resched* volontariamente eseguendo una routine di sistema come *wait* o facendo operazioni di input o di output (I/O).Infatti le operazioni di input/output sono lente paragonate ai calcoli, perché i processi impiegano molto del loro tempo aspettando i dispositivi. Comunque, senza la caratteristica di essere preemptive, il sistema potrebbe non ottenere più il controllo da un processo che esegue un loop infinito, è importante così includerlo in ogni sistema che supporta la multiprogrammazione.

Il sistema inoltre usa il real-time clock per servire le richieste per i timed delay. Per esempio, quando il processo in esecuzione richiede un ritardo, PC-Xinu sposta il processo in una lista di processi "sleeping", ordinati per essere risvegliati dopo il numero appropriato di tick del clock. In ogni tick del clock, la routine di interruzione del clock esamina i processi sleeping e sposta quelli che hanno aspettato il tempo specificato nella lista dei processi ready. La routine deve gestire quale ritardo sarà considerato il prossimo.

10.4 Delta list processing

Poiché non è possibile permettersi di cercare arbitrariamente attraverso una lunga lista di processi sleeping per trovare quello che deve essere svegliato ogni tick del clock, il sistema tiene i processi sospesi in una struttura datti chiamata *delta list*. Come le altre liste di processi, i processi sospesi della delta list risiedono in delle q strutture. La variabile *clockq* contiene l'indice della testa di q. Ad ogni tick del clock, il clock interrupt dispatcher esamina il primo processo in *clockq* e chiama la routine ad alto livello *wakeup* per svegliare i processi per cui è finito il loro tempo di attesa.

A differenza delle altre lista di struttura q, la *delta list* non è né ordinata incrementando le chiavi (keys), né in modo FIFO. Invece, le chiavi memorizzano le successive delta (differenze) di ritardo:

> *I processi nel clockq sono ordinati in base al tempo con cui saranno risvegliati; ogni chiave dice il numero di tick del clock che il processo deve aspettare in più rispetto al precedente nella lista.*

Il primo processo nella lista è quello che deve attendere di meno, e la sua chiave da il rimanente ritardo in tick del clock affinché sia svegliato. L'organizzazione delta permette alla routine di interruzione del clock di decrementare la prima chiave ogni tick del clock senza scorrere la lista perché i rimanenti ritardi sono relativi ad esso. Per esempio, se quattro processi hanno bisogno di aspettare 17, 27, 28 e 32 tick, allora le loro chiavi nella delta list contengono 17, 10, 1 e 4. Dando solo la delta list, la somma parziale delle chiavi da l'attesa totale prima del risveglio dei processi. L'attesa totale prima del risveglio del primo processo è 17, il totale per il secondo è 17+10, il totale per il terzo è 17+10+1, ed il totale per l'ultimo è 17+10+1+4.

Le procedure che modificano la delta list sono facili da progettare, ma i dettagli possono essere complessi; affrontare uno studio specifico del codice è meritevole. La procedura *insertd*, mostrata sotto, inserisce un processo *pid* in *clockq*, mettendo il suo ritardo nel parametro *key*. Come con le code di priorità, il campo *qkey* di ogni nodo nella lista memorizza il valore della chiave per quel nodo. Nel codice, la variabile *next* scorre la lista cercando il posto per inserire il nuovo processo.

Le chiavi nella delta list specificando il ritardo relativo al loro predecessore, non possono essere confrontate direttamente col valore iniziale di *key*, che specifica il ritardo relativo al tempo attuale. Per rendere il ritardo paragonabile, *insertd* sottrae il ritardo relativo da *key* man mano che procede la ricerca, mantenendo la seguente invariante:

> *In ogni istante durante la ricerca, sia key sia q[next].qkey specificano il ritardo relativo al tempo in cui il predecessore di "next" si sveglia.*

Insertd inserisce il nuovo processo nel punto dove il suo ritardo relativo è minore al ritardo relativo di quelli a sinistra nella lista. Da notare che *insertd* non ha esplicitamente il controllo per la fine della lista, perché il valore di key nella coda forza l'inserzione. Dopo che il processo *pid* è collegato nella lista, *insertd* sottrae il ritardo extra che è stato introdotto, dal ritardo del processo successivo.

```
/* insertd.c - insertd */

#include <conf.h>
#include <kernel.h>
#include <q.h>

/*------------------------------------------------------------------------
 * insertd -- inserisce il processo pid nella delta list
 *            passando la sua chiave
 *------------------------------------------------------------------------
 */

INTPROC  insertd(pid, head, key)
         int  pid;
         int  head;
         int  key;
{
         int  next;  /* scorre la lista                          */
         int  prev;  /*predecessore di next                */

         for (prev=head, next=q[head].qnext ;
             q[next].qkey < key ; prev=next, next=q[next].qnext)
                  key -= q[next].qkey;
         q[pid].qnext = next;
         q[pid].qprev = prev;
         q[pid].qkey = key;
         q[prev].qnext = pid;
         q[next].prev = pid;
         if (next < NPROC)
              q[next].qkey -= key;
         return(OK);
}
```

10.5 Mettere un processo nello stato sleep

I programmi utente solitamente non accedono alla coda del real-time clock direttamente; essi chiamano una procedura di sistema che provvede ad una ritardo. Il sistema chiama *sleept(n)* che fa attendere il processo chiamante n ticks. Così facendo si inserisce il processo nella delta list dei processi sleeping.

Quando un processo è spostato nella lista dei processi sleeping, per lungo tempo non sarà ne *ready* ne *current*. In che stato deve essere posto? I processi sleeping differiscono dai processi che sono suspended, da quelli che aspettano di ricevere messaggi o da quelli waiting per i semafori, così nessuno di questi stati va bene. E' il momento quindi di aggiungere un nuovo stato di processo; sarà chiamato *sleeping* e denotato con la costante simbolica *PRSLEEP* . Il nuovo diagramma dei processi delle transizioni tra stati è mostrato nella figura 10.1.

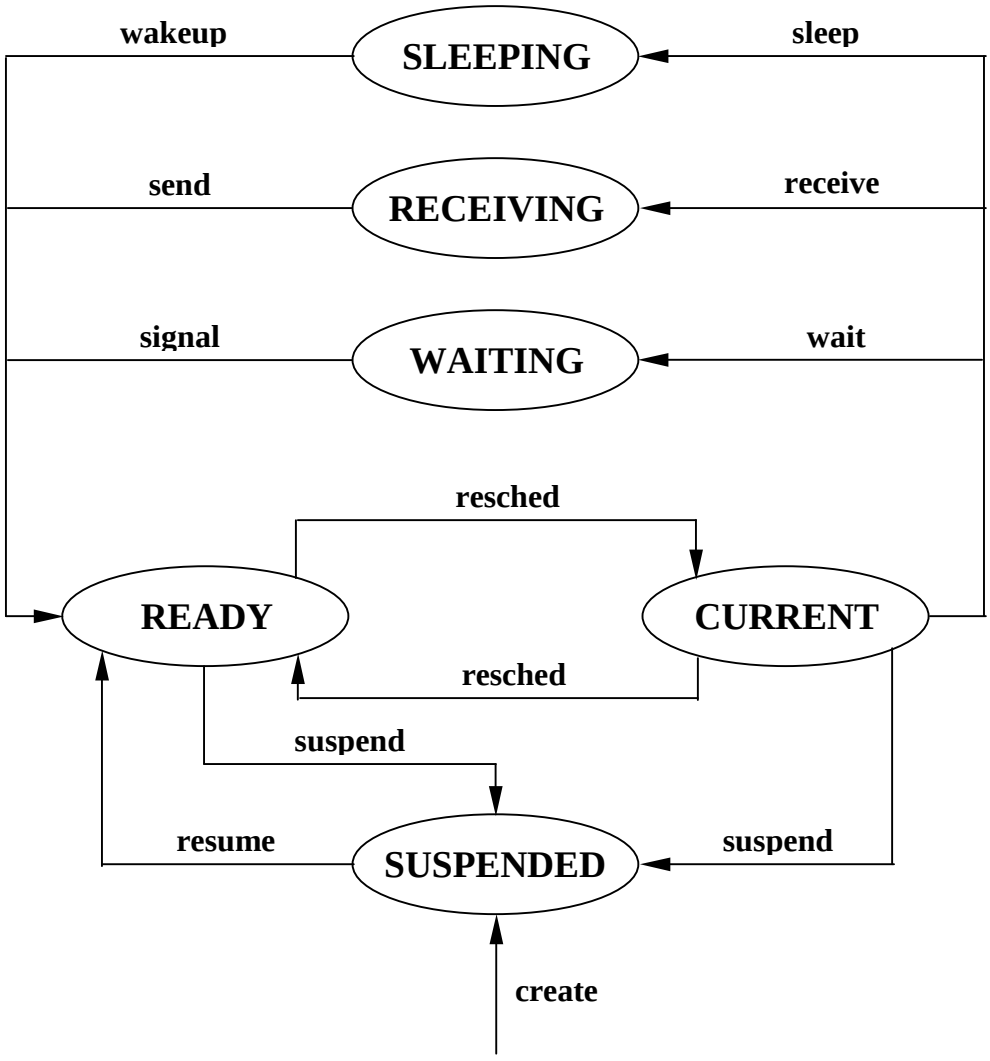

Figura 10.1 Il diagramma di transizione tra gli stati dei processi per lo stato 'sleep'

L’implementazione di *sleept* è semplice. Come mostrato sotto, è usato *insertd* per muovere il processo corrente nella lista dei processi sleeping, cambiando il suo stato in *sleeping*, e chiama *resched* per allocare altri processi per l’esecuzione.

```
/* sleept.c - sleept */

#include <conf.h>
#include <kernel.h>
#include <proc.h>
#include <q.h>
#include <sleep.h>

/*------------------------------------------------------------------------
 * sleept - aspetta la chiamata per un tempo specificato nel
 * system ticks
 *------------------------------------------------------------------------
 */
SYSCALL sleept(n)
int n;
{
          int  ps;

          if ( n<0 )
                    return (SYSERR);
          if ( n==0 )
                    return (OK);
          disable(ps);
          insertd(currpid, clockq, n);
          slnempty = TRUE;
          sltop = & q[q[clockq].qnext].qkey;
          proctab[currpid].pstate = PRSLEEP;
          resched();
          restore(ps);
          return(OK);
}
```

Sleept richiama tre variabili esterne definite nel file *sleep.h* (sotto): *clockq*, *sltop*, e *slnempty*. *Clockq* contiene i *q* indici della lista dei processi sleeping.

```
/* sleep.h */

extern int clockq;     /*indice della lista dei proc. sleeping */
extern int *sltop;     /*indirizzo della prima chiave in clockq*/
extern int slnempty;  /*1 sse clockq è non vuoto                */
extern long tod;       /*time of day ticks dalla partenza       */
extern int defclk;    /*>0 sse le interruzioni del clock sono
                        deferred (rinviate)                     */
extern int clkdiff;   /*numero di ticks deferred (rinviati)*/

#define TICSN 19663   /*numero di ticks per 1080 secondi      */
#define TICSD 1080
```

Sebbene il tick rate per il PC-Xinu è relativamente basso, il processo di interruzione del clock è costoso. Le variabili *sltop* e *slnempty* aiutano all'ottimizzazione del processo di interruzione rendendo facile determinare quale processo sleeping deve essere svegliato (awakened). *Slnempty* serve per indicare se la lista *clockq* è correntemente nonempty (non vuota). Se qualche processo è in *clockq, sltop* da l'indirizzo della chiave del primo processo. La routine di interruzione salta completamente la coda dei processi sleeping quando *slnempty* è zero, ed usa *sltop* per localizzare velocemente la delta-key del primo processo sleeping quando *slnempty* è diverso da zero.

10.6 Ritardo misurato in secondi

La dimensione di un intero, 16 bits, limita il ritardo allocato da *sleept* a 2^{15}-1 ticks, che sono circa 1800 secondi, o 30 minuti. La chiamata di sistema *sleep* fornisce una soluzione per i processi che devono aspettare fino a 9 ore, infatti il suo argomento specifica un ritardo misurato in secondi piuttosto che in ticks. *Sleep* usa *sleept* ripetutamente per schedulare ritardi brevi finché il ritardo totale è trascorso.

```
/* sleep.c - sleep */

#include <conf.h>
#include <kernel.h>
#include <sleep.h>

/*------------------------------------------------------------------------
 * sleep -- sospende il processo chiamante per n secondi
 *------------------------------------------------------------------------
 */
SYSCALL sleep(n)
int n;
{
          int ps;

          if ( n<0 )
                    return(SYSERR);
          if ( n==0 )  {
                    disable(ps);
                    resched();
                    restore(ps);
                    return(OK);
          }
          while ( n>=TICSD )  {
                    sleept(TICSN);
                    n -= TICSD;
          }
          if (n>0)
               sleept((int)((long)n*(long)TICSN/(long)TICSD));
          return(OK);
}
```

I numeri TICSN e TICSD (hanno valutazione rispettivamente 19663 e 1080) definiti in sleep.h meritano un commento speciale. Il quoziente TICSN/TICSD produce un valore approssimativamente molto vicino al tick rate, con un errore di circa 25 parti per miliardo. Esprimendosi in un altro modo, TICSN ticks equivale a TICSD secondi. Quando si schedula un ritardo misurato in secondi, il ritardo che supera TICSD può essere schedulato come (possibilmente

ripetuto) ritardo di TICSN ticks usando *sleept*. Quando il ritardo rimanente in secondi è minore di TICSD, esso è usato per determinare quanti ticks dovrebbero essere schedulati come una frazione di TICSN per ottenere l'attesa richiesta in secondi. Per esempio, se *n* è il numero di secondi da attendere, il quoziente

*n**TICSN/TICSD

sarà il numero di ticks da schedulare per *sleept*. Da osservare che la moltiplicazione *n**TICSN dovrebbe essere fatta prima di dividere per TICSD per evitare un errore di arrotondamento (round-off) intero. In vista della grandezza di questo numero, questo prodotto deve essere riportato usando un intero long.

Da notare che una richiesta per un ritardo di un secondo significherà schedulare un ritardo di esattamente 18 tick e si concluderà con un attesa di approssimativamente 0.988659 secondi, dando un errore di circa 1.1341 percento. Questo errore può accumularsi quando tanti ritardi sono schedulati ripetutamente per un lungo periodo di tempo. E' da osservare anche che *sleep* è progettato per causare un immediata rischedulazione se il tempo di sleep è zero. Poiché *resched* non può essere chiamato con le interruzioni abilitate, questa caratteristica realizza una soluzione semplice per un processo che rischedula se stesso.

10.7 Procedura di interruzione del clock

L'*intcom* interrupt dispatcher chiama la procedura di servizio di interruzione del clock *clkint* ogni clock tick. Poiché è stato chiamato da un interrupt dispatcher, *clkint* assume che le interruzioni sono state disabilitate subito dopo essere entrati; la keyword INTPROC ricorda il lettore di questa assunzione. Il parametro di *ckint* è un parametro finto passato automaticamente da *intcom*. *Clkint* prima incrementa il valore globale *tod* time-of-day, il quale conta i numeri di tick finché il sistema parte. Se deferred clock processing è in esecuzione, *clkdiff* è incrementato e *clkint* ritorna immediatamente.

Se la lista dei processi sleeping non è vuota, *clkint* decrementa la prima chiave in *clockq*,e se il ritardo ha raggiunto lo zero, *wakeup* è chiamato per svegliare i processi di cui il tempo di attesa è stato raggiunto. Nota ora che *slnempty* e *sltop* eliminano la computazione subscript al momento dell'interruzione. Finalmente, il contatore preemption è decrementato e si rischedulerà il processo corrente se la sua fetta di tempo è trascorsa.

```
/* clkint.c - clkint */


#include <conf.h>
#include <kernel.h>
#include <sleep.h>
#include <io.h>


/*------------------------------------------------------------------
 * clkint -- routine di servizio del clock
 * chiamata in ogni clock tick e quando parte il deferred clock
 *------------------------------------------------------------------
 */


INTPROC clkint(mdevno)
int mdevno;                         /* minor device number       */
```

```
{
        int i;

        tod++;
        if (defclk) {
                clkdiff++;
                return;
        }
        if (slnempty)
                if ( (--*sltop) <= 0)
                        wakeup();
        if ( (--preempt) <= 0)
                resched();
}
```

10.8 Risveglio dei processi sleeping

Wakeup rende ready tutti i processi il cui ritardo di tempo è trascorso. Il valore della prima chiave di *clockq* tiene in considerazione ogni tick che è stato accumulato dall'ultima chiamata di *wakeup*. *Wakeup* rimuove e rende ready il primo processo di *clockq* il cui ritardo di tempo è trascorso, e propaga ogni insufficienza di tick accumulati al prossimo oggetto nella lista waiting. Infine *wakeup* resetta *sltop* e *slnempty* per riflettere lo stato della nuova coda prima di chiamare *resched*, perché la rischedulazione potrebbe scegliere un altro processo da eseguire (ed il clock da interrompere).

```
/* wakeup.c - wakeup */

#include <conf.h>
#include <krnel.h>
#include <proc.h>
#include <q.h>
#include <sleep.h>

/*------------------------------------------------------------------
 * wakeup -- chiamata dal clock interrupt dispatcher per
 * risvegliare un processo
 *------------------------------------------------------------------
 */
wakeup()
{
        register int makeup;     /* makeup per l'ultimo valore*/
        register int k;          /* valore della chiave       */

        makeup=0;
        while (nonempty(clockq)&&(k=firstkey(clockq))<=makeup){
                makeup -= k;
                ready(getfirst(clockq));
        }
        if (slnempty = nonempty(clockq)) {
                sltop = &firstkey(clockq);
                *sltop -=makeup;
        }
        resched();
}
```

10.9 Deferred clock processing

Il clock dispatcher include un'altra caratteristica che lo complica: il *deferred clock processing*. In sostanza, il modo deferred clock permette al sistema di accumulare clock ticks senza avviare gli eventi. La differenza tra l'ignorazione delle interruzione del clock ed il rinvio è che il gestore del clock può schedulare eventi che "potrebbero essere accaduti" quando lascia il modo deferred e ritorna al normal mode. Se il clock rimane deferred solo per pochi tick alla volta, il sistema apparirà lavorare normalmente.

Il motivo del deferred clock processing è che il sistema operativo mantiene disabilitate le interruzioni mentre cambia contesto. Disabilitare le interruzioni non crea nessun problema quando l'input arriva dalla tastiera perché l'utente scrive lentamente rispetto alla velocità con cui un computer elabora le informazioni. Ma per alcune comunicazioni tra computer e computer, i dati sono mandati e ricevuti ad un alta velocità. Se si ha un cambio di contesto quando si sta ricevendo un blocco di caratteri, alcuni di questi caratteri potrebbero essere persi.

Per risolvere il problema, il sistema di I/O ha bisogno di proibire il cambio di contesto per brevi periodi di tempo anche se tuttavia le interruzioni rimangono abilitate. Idealmente, il sistema dovrebbe essere capace di "compensare per il tempo perduto" quando il cambio di contesto è riabilitato di nuovo. Anche se tuttavia è impossibile da prevenire il cambio di contesto senza modificare il comportamento del sistema, l'idea è di trovare una soluzione che ha il minimo impatto senza cambiare il progetto di partenza. Deferred clock processing consiste nel posporre, ma non ignorare, il cambio di contesto. Durante un periodo di deferred clock, il cambio di contesto è sospeso dalle interruzioni del deferring clock. Quando il deferral (rinvio) termina, si riprende la normale procedura.

10.9.1 Procedure per il passaggio da e per il Deferred Mode

Un processo può porre il clock nel *deferred mode* chiamando *stopclk* e ritornare al clock nel modo *real-time* chiamando *strtclk*. Più processi possono richiedere che il clock sia deferred - esso rimarrà deferred finche tutti i processi avranno chiamato *strtclk*. *Stopclk* conta le richieste di deferral incrementando *defclk*, e *strtclk* conta le richieste di ritorno decrementandolo. Finché *defclk* rimane positivo, il gestore delle interruzioni conta i clock ticks in *clkdiff* senza processarli.

Strtclk "compensa per il tempo perduto" quando *defclk* raggiunge di nuovo lo zero, stabilendo tutti gli eventi che sarebbero dovuti accadere mentre il clock era nello stato deferred. Per fare ciò, *strtclk* aggiorna il contatore preemption e sottrae i ticks accumulati dal ritardo dei processi sleeping, svegliando i processi il cui tempo di attesa è stato raggiunto.

Il codice sia per *strtclk* che per *stopclk* è contenuto nel file *ssclock.c*, mostrato sotto.

```
/* ssclock.c - stopclk, strtclk */

#include <conf.h>
#include <kernel.h>
#include <q.h>
#include <sleep.h>

/*------------------------------------------------------------------------
 * stopclk -- passa al clock nel defer mode
 *------------------------------------------------------------------------
 */
stopclk()
{
        int  ps;
```

```
        disable(ps);
        defclk++;
        restore(ps);
}

/*------------------------------------------------------------
 * strtclk -- porta il clock fuori dal defer mode
 *------------------------------------------------------------
 */
strtclk()
{
        int  ps;
        int  makeup;
        int  next;

        disable(ps);
        if ( defclk<=0 || --defclk>0) {
                restore(ps);
                return;
        }
        makeup = clkdiff;
        clkdiff = 0;
        if (slnempty)
                if ( (*sltop -= makeup) <= 0 )
                        wakeup();
        if ( (preemt -= makeup) <= 0 )
                resched();
        restore(ps);
}
```

10.10 Inizializzazione del clock

La procedura *clkinit*, mostrata sotto, esegue l'inizializzazione necessaria.

```
/* clkinit.c - clkinit */

#include <conf.h>
#include <kernel.h>
#include <q.h>
#include <sleep.h>
/*------------------------------------------------------------------
 * clkinit -- inizializza il clock e la coda di sleep (chiamata
 * all'avvio)
 *------------------------------------------------------------------
 */
clkinit()
{
        tod = 0L;               /*tempo iniziale del giorno  */
        preempt = QUANTUM;      /*tempo iniziale quantum */
        slnempty = FALSE;       /*inizialmente, nessun processo
                                  addormentato              */
        clkdiff = 0;            /*zero deferred ticks       */
        defclk = 0;             /*clock non è deferred      */
        clockq = newqueue();    /*alloca la coda per il clock*/
}
```