CAPITOLO 9 - ELABORAZIONE DEGLI INTERRUPT

L'interrupt hardware è un meccanismo potente per fornire il supporto di molti servizi del sistema operativo. Come descritto nel capitolo 2, una richiesta di interrupt da parte di una device avviene inviando un segnale alla CPU sulla sua linea di interrupt. Prima di eseguire una istruzione, la CPU controlla la linea di interrupt e chiama una procedura che manipoli l'interruzione se ne trova una pendente. Quando la routine chiamata "ritorna", il controllo viene ripassato al processo che stava eseguendo, come se niente fosse successo. Senza un meccanismo di interrupt il sistema operativo non potrebbe garantirsi la ripresa del controllo una volta partita l'esecuzione del processo utente.

Prima di osservare i dispositivi in dettaglio e vedere il motivo per cui effettuano interrupt e come il sistema gli risponde, esploreremo, in questo capitolo, le routine che PC-Xinu usa per prendere e rilasciare interrupt e il passaggio di controllo all'appropriata routine. I capitoli successivi affrontano più da vicino i dispositivi del clock e della tastiera, mostrando come le routine di elaborazione degli interrupt sono state progettate.

9.1 Inviare interrupt

Abbiamo detto che l'hardware chiama il gestore degli interrupt quando ne trova uno pendente. I termini *call* e *return* hanno un significato speciale quando sono applicate alle interruzioni. Primo, non ci sono chiamate a procedure nelle istruzioni dei programmi di interrupt - il processore simula una chiamata all'appropriata routine di interrupt nel mezzo dell'esecuzione di una normale istruzione del programma utente. Secondo, il processore salva automaticamente i registri FLAGS, CS e IP sullo stack (la routine di interrupt deve salvare e restituire ogni altro registro di cui necessita). Terzo, la routine di interrupt deve usare una speciale istruzione di ritorno che ristabilisca i vecchi valori dei registri FLAGS, CS e IP in un'unica istruzione atomica.

Poiché le routine di interrupt manipolano registri hardware e usano speciali sequenze di call/return, esse non possono essere scritte in linguaggi di alto livello come il C. La tentazione è di scrivere tutto il codice relativo all'elaborazione degli interrupt (una porzione significante del sistema operativo) in linguaggio assembler. Ma scrivere software con linguaggi di basso livello rende più difficoltosa la modifica e la comprensione. Così, per mantenere il codice di sistema più leggibile, il nostro progetto impiega una strategia a due livelli per tale elaborazione. Gli interrupt si ramificano in piccole routine di basso livello (*interrupt dispatch*) scritte in linguaggio assembler. Il dispatcher manipola lavori come il salvataggio e il ripristino di registri, l'identificazione delle device di interrupt, e il ritorno dall'interruzione dopo che essa è stata processata. Tuttavia fa anche qualcos'altro ossia chiama una routine ad alto livello che svolge la vera elaborazione passandogli le informazioni neccessarie.

PC-Xinu serve interrupt dal clock, dalla tastiera, e da altri pseudo-dispositivi come *Ctrl-Break* descritto nel capitolo 2. Esso gestisce anche interruzioni eccezionali. Questo capitolo si concentra sul dispatcher, rinviando la discussione della maggior parte dei servizi di interrupt ai capitoli seguenti.

9.2 L'interrupt dispatcher

I dispositivi si connettono col sistema attraverso meccanismi hardware chiamati controller. I controller, che possono essere semplici come un chip o complicati come un microprocessore, risiedono generalmente su schede connesse al bus di sistema. Essi possiedono hardware capace di

tradurre dati digitali in segnali che controllano e comunicano con dispositivi periferici quali tastiere e dischi come descritto nel capitolo 2.

Alcune device, come i terminali, permettono un trasferimento simultaneo dei dati in entrambe le direzioni e consiste di due dispositivi indipendenti, uno per l'input, e un altro per l'output. Spesso questi dispositivi impiegano un solo controller per l'input e l'output. Il controller può usare vettori separati per gli interrupt di input ed output o (come nel caso della comunicazione tra dispositivi nel PC) un vettore per entrambi i tipi. Nel secondo caso sarebbe necessario, per il software, distinguere gli interrupt di input da quelli output interrogando lo stesso controller, ma al livello più basso il sistema deve operare come se i dispositivi di input e di output fossero indipendenti.

9.2.1 La tabella intmap

PC-Xinu usa una tabella di interrupt, chiamata *intmap*, in cui ogni elemento è dedicato ai dispositivi richiedenti il servizio di interrupt del sistema operativo. Un elemento dell'*intmap* contiene informazioni sulla device e un'istruzione CALL all'interrupt dispatcher comune *intcom*. Il vettore di interrupt per una device punta all'istruzione CALL nel suo elemento dell'intmap; così quando si verifica una interruzione il processore effettua la chiamata che esegue immediatamente il codice nel dispatcher. La figura 9.1 mostra l'organizzazione logica.

Se tutti le interruzioni chiamano la stessa routine dispatch, come riesce il dispatcher a sapere quale routine di interrupt ad alto livello dovrà essere chiamata? Ricordiamo che l'istruzione CALL inserisce sullo stack l'indirizzo di ritorno prima di ramificarsi nel codice chiamato. Quando il processore esegue l'istruzione CALL nella tabella *intmap*, l'indirizzo del byte seguente l'istruzione viene lasciato sullo stack. Il codice di *intcom* può usare questo indirizzo come puntatore all'elemento *intmap* per la device, permettendo l'accesso alle informazioni tra cui l'indirizzo della routine di alto livello del dispositivo e altri specifici parametri.

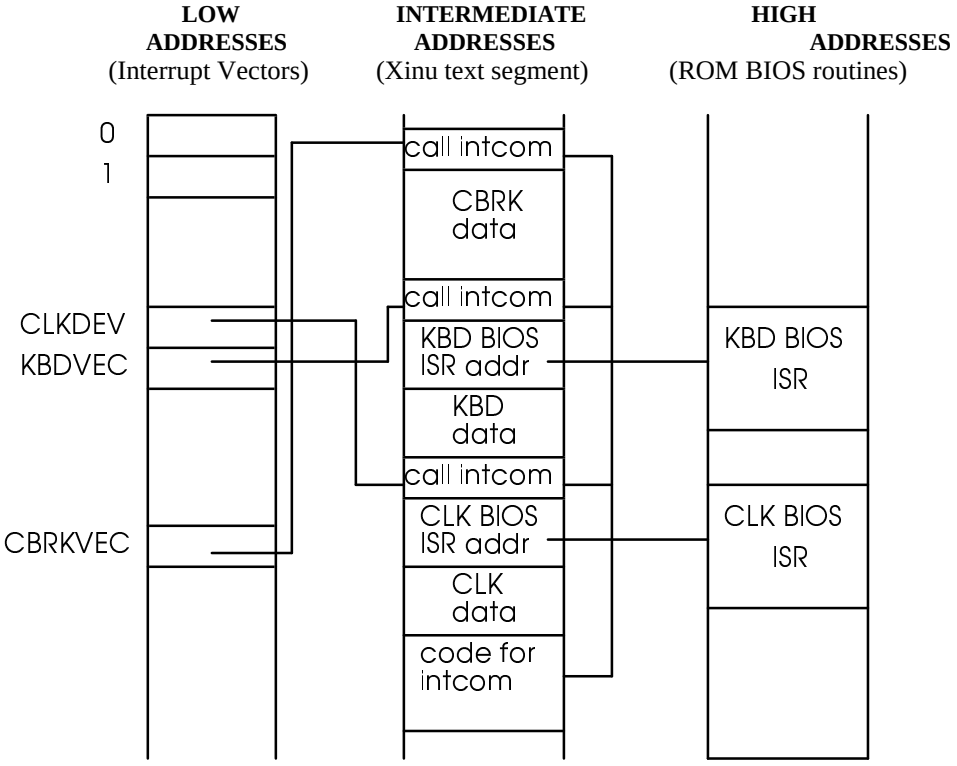

Figura 9.1 - Vettori di interrupt che puntano alla tabella intmap.

Uno sguardo al codice chiarificherà i dettagli. La tabella di invio interrupt *intmap* è definita nel file *io.h*.

```
/* io.h -  fgetc, fputc, getchar, isbaddev, putchar */

#include <mem.h>

#define     INTVECI     inint       /* routine interrupt dispatch di input */
#define     INTVECO     outint      /* routine interrupt dispatch di output */
extern      int   INTVECI();
extern      int   INTVECO();

#define NMAPS     0x20        /* numero di elementi dell'intmap */
struct      intmap_t {        /* device-to-interrupt routine mapping   */
      char  ivec;             /* numero di interrupt */
      char  callinst;         /* l'istruzione call */
      word  intcom;           /* codice di interrupt comune */
      word  oldisr_off;       /* vecchio offset della int. serv. routine */
      word  oldisr_seg;       /* vecchio segmento della int. serv. routine */
      int   (*newisr)();      /* puntatore alla nuova ser. routine */
      word  mdevno;           /* minor device number             */
      word  iflag;            /* se diverso da zero, chiama vecchia isr */
      };

/*
 * NOTA: La struttura intmap occupa un totale di 8 word o 16 byte
 * per record.
 */

extern struct intmap_t *sys_imp;    /* puntatore alla tabella intmap */
extern      int   nmaps;            /* numero di elementi dell'intmap attivi */


#define BADDEV          (-1)

#define isbaddev(f)     ( (f)<0 || (f)>=NDEVS )
#define fopen(n,m)      open(STDIO, (n), (m))
#define fclose(d)       close(d)

/* Procedura di I/O in linea */

#define     getchar()         getc(STDIN)
#define     putchar(ch)       putc(STDOUT,(ch))
#define     fgetc(unit)       getc((unit))
#define     fputc(ch,unit)    putc((unit),(ch))

extern int  _doprnt();        /* output formattato */
```

Ogni elemento in intmap corrisponde ad una device. Il tipo di interrupt (da dove viene l'interrupt) è memorizzato, durante l'inizializzazione del sistema, nel campo *ivec*. Di seguito abbiamo tre byte per l'istruzione CALL. L'indirizzo (segmento:offset) della routine di servizio precedentemente installato nel vettore, si trova nel campo *oldisr*. Questo indirizzo e il tipo di interrupt in *ivec* sono usati per ripristinare il vettore di interrupt al valore predente dopo la conclusione di PC-Xinu.

L'elemento *newisr* è un puntatore al codice della routine di servizio interrupt di PC-Xinu e viene chiamato da *intcom* quando avviene l'interrupt di quel dispositivo. Dopo esser stata chiamata *newisr*, verrà passato il *mdevno* dalla tabella *intmap*.

La tabella *intmap* è definita nel file *intmap.asm*:

```
; intmap.asm -
; low-level interrupt transfer table and dispatcher

      include dos.asm

tblsize equ 20h         ; definisce la massima dimensione della tabella intmap
stksize equ 1000h       ; massima dimensione dello stack di sistema

      dseg

      public _sys_imp

intstack    db    stksize dup (?)   ; stack di interrupt
topstack    label byte
_sys_imp    dd                      ; puntatore far a intmap

      endds

      pseg

      public      pcxflag, ssave, spsave

pcxflag     dw    1           ; zero se schedulazione disabilitata
spsave      dw    ?           ; salva il registro puntatore allo stack
ssave       dw    ?           ; salva il registro di segmento dello stack

;---------------------------------------------------------------------------
; intmap  --  tabella interrupt dispatch
;---------------------------------------------------------------------------
intmap      label byte

      rept  tblsize

      db          ?               ; ivec   - numero di vettore interrupt
      call        intcom
      dd          ?               ; oldisr - vecchia isr del bios (seg:off)
      dw          -1              ; newisr - nuovo indirizzo di codice a isr
      dw          ?               ; mdevno - minor device number
      dw          ?               ; iflag  - interrupt flag

      endm

      ASSUME DS:NOTHING

;---------------------------------------------------------------------------
; intcom  --  interrupt dispatch comune
;---------------------------------------------------------------------------
; Questa procedura manipola codice comune a tutte le procedure di servizio di
; interrupt.
intcom  proc    near
      push  bp
      mov   bp,sp
      push  ax                          ; mette nello stack i registri
      push  bx
      mov   bx,[bp+2]                   ; ottiene puntatore ai dati intmap
      mov   ax,cs:[bx+8]                ; ottiene il flag di interrupt
      or    al,al                       ; è zero?
      je    short nobios                ; si, abilita la chiamata al BIOS
      pushf                             ; push dei flag per simulare interrupt
```

```
      call    cs:dword ptr[bx]           ; chiama le ISR BIOS
      cli                                ; recede gli interrupt
nobios:
      push  cx                           ; salva il resto dei registri
      push  dx
      push  si
      push  di
      push  ds
      push  es
      mov   cs:sssave,ss                 ; salva l'ambiente stack
      mov   cs:spsave,sp
      mov   cx,cs                        ; ottiene il segmento codice
; bp+6 punta al segmento codice dove accadono gli interrupt
      cmp   cx,[bp+6]                    ; controlla se abbiamo interrupt
      jnz   short newstack
; i segmenti stack e data sono OK; possiamo effettuare la nostra ISR
      push  cs:word ptr[bx+6]            ; passa il minor dev.no
      call  cs:dword ptr[bx+4]           ; chiama la ISR in C (salva si, di)
      add   sp,2                         ; dealloca parametro (convenzione C)
      jmp   short popregs
newstack:
; adesso prepariamo uno stack temporaneo, disabilitiamo la schedulazione ed
effettuiamo la nostra ISR
      mov   ax,DGROUP
      mov   ds,ax                        ; setta ds a DGROUP
      ASSUME      DS:DGROUP
      mov   ss,ax                        ; prepara stack temporaneo in DGROUP
      mov   sp,offset topstack
      xor   ax,ax                  ; pulisce il pcxflag per prevenire resched
      xchg  ax,cs:pcxflag
      push  ax                           ; lo salva per successivo repristino
      push  cs:word ptr[bx+6]            ; passa il minor dev. no.
      call  cs:word ptr[bx+4]      ; chiama la nostra routine (salva si, di)
      add   sp,2                         ; dealloca il parametro (convenzione C)
      pop   cs:pcxflag                   ; ripristina il pcxflag
      mov   ss,cs:ssave                  ; ripristina il vecchio stack
      mov   sp,cs:spsave
      ASSUME      DS:NOTHING
popregs:
      pop   es                           ; ripristina tutti i registri
      pop   ds
      pop   di
      pop   si
      pop   dx
      pop   cx
      pop   bx
      pop   ax
      pop   bp
      add   sp,2                         ; rimuove il puntatore all'area intmap
      iret
intcom      endp

      endps

      end
```

La direttiva REPT nella definizione di *intmap* è un modo conveniente per allocare un numero identico di elementi con codice unico. In questo caso *tblsize* è posto a 20H, e significa che ci saranno 20H elementi nella tabella intmap.

La tabella *intmap* è nel segmento codice, i quali dati sono generalmente inaccessibili ai programmi C. Poiché gli elementi di *intmap* sono generati da una routine in C, *intmap.asm* definisce una variabile di segmento dati pubblico *sys_imp* contenente il segmento:offset che punta a *intmap*. Questa variabile è disponibile ai programmi C come puntatore far ai dati di *intmap*. Ricordiamo che l'underscore prima del nome sys_imp permette l'accesso alle variabili ai programmi C.

9.2.2 Implementazione dell'Interrupt Dispatcher

Intcom comincia con l'inserimento nello stack dei registri BP, AX e BX. Poiché il dispatcher è una routine di servizio di interrupt e questi registri sono usati per le attività di dispatch interrupt, è necessario che essi siano salvati prima del loro uso. Essi saranno prelevati dallo stack al ritorno del processo interrotto.

Ricordiamo che sull'entrata all'interrupt dispatcher, l'offset CS dell'elemento di *intmap* della device (attualmente, all'indirizzo *oldisr*) è nello stack come risultato della istruzione CALL. Il registro BP che riferisce lo stack, è usato per riottenere questo puntatore (alla CALL) per salvarlo nel registro BX. Il puntatore sarà usato per accedere a tutte le componenti dell'elemento di intmap. Prima di chiamare la routine di servizio in C i rimanenti registri DX, SI, DI, DS e ES vengono inseriti sullo stack e tale ambiente di stack SS:SP viene salvato nel segmento:offset *sssave:spsave*.

L'array *intstack* in *intmap.asm* merita un commento speciale. Le interruzioni che avvengono durante una chiamata BIOS possono usare segmenti dati e stack che differiscono da quelli usati da PC-Xinu. Le routine di interrupt in C devono essere chiamate con segmenti di stack e dati in accordo con la struttura della memoria data nei capitoli 2 e 8. Se l'interrupt avviene fuori dall'ambiente PC-Xinu, *intcom* deve mettere sù uno stack locale prima di chiamare la routine di servizio in C. Dopo che lo stack locale è stato usato, *intcom* ripristina l'ambiente di stack SS:SP grazie alle variabili salvate *sssave:spsave*.

Il codice illustra un'altra caratteristica del linguaggio assembler 8088. Il riferimento ai dati usa per default il registro di segmento DS, ma alcuni dati di *intmap* risiedono nel segmento codice. Per riferirsi alle variabili *pcxflag*, *sssave* e *spsave* del segmento codice viene usato un *segment override*. In linguaggio assembler viene espresso usando il prefisso "CS:" sull'istruzione dati. Poiché i programmi C riferiscono soltanto le variabili nel segmento dati per default, le variabili nel codice di segmento possono essere accessibili soltanto attraverso il linguaggio assembler.

9.2.3 Schedulazione rimandata

Una routine si dice *reentrant* se simultanee chiamate di più processi possono eseguire il suo codice nello stesso momento. Il codice *reentrant* abbonda nei sistemi operativi che supportano processi multipli (*resched* è uno dei molti esempi di PC-Xinu). Le routine *reentrant* usano normalmente variabili locali appartenenti allo stack space del processo utente; il riferimento ad ogni variabile globale è generalmente controllato da un'accesso delimitato con chiamate a *disable* ed *enable* interrupt.

I servizi BIOS non sono *reentrant* e poichè tipicamente abilitano gli interrupt non possono proteggere le loro variabili globali dall'accesso di multipli processi all'occorrenza di una interruzione. PC-Xinu usa una singola routine, *resched*, per alternare i processi e l'unico modo per essere chiamata durante i servizi BIOS è tramite un interrupt (per esempio del clock o della tastiera). Quando una routine BIOS è in esecuzione organizza il suo segmento dati per accedere alle variabili globali contenute. Come descritto nella precedente sezione, il codice di *intcom* mette sù

uno stack locale, nell'ambiente PC-Xinu, per manipolare gli interrupt che avvengono durante i servizi BIOS. Questi interrupt non devono effettuare schedulazione, poichè potrebbe verificarsi in un'altra chiamata alla routine BIOS.

La variabile *pcxflag* in *intmap.asm* serve da tramite per rimandare la schedulazione. Se è messa a zero la schedulazione è disabilitata, perciò non avverranno cambi di contesto e sarà evitato l'accesso a più processi. Se è posta a 1 la schedulazione di PC-Xinu è abilitata. Il codice dell'interrupt dispatcher mostra che *pcxflag* contiene il valore 0 quando lo stack locale è usato per servire gli interrupt BIOS. Questa sicurezza contro la schedulazione da interrupt durante le attività BIOS evita la distruzione in routine *non-reentrant*.

9.2.4 Al BIOS oppure no

Un elemento della tabella *intmap* per un dispositivo contiene l'indirizzo segmento:offset della vecchia routine di servizio interrupt che era nel vettore prima dell'inizializzazione di PC-Xinu. In molti casi questa routine è chiamata dall'interrupt dispatcher prima di chiamare la routine di servizio di PC-Xinu. Per esempio, gli interrupt della tastiera devono essere processati dal BIOS prima di essere serviti da PC-Xinu. Ci sono interrupt, tuttavia, che non dovrebbero essere manipolati dalle vecchie ISR. Un esempio è *Ctrl-Break*, il tipo interrupt 1BH, invocato sulla recezione della sequenza tasti Ctrl-Break. Se la vecchia ISR fosse eseguita prima, PC-Xinu potrebbe terminare e ritornare il controllo a MS-DOS, privando una possibilità a Xinu di ristabilire il vettore di interrupt dalla tabella *intmap*, un'importante attività conclusiva di programma.

Per questa ragione gli elementi di *intmap* hanno un campo *ifield* che, se diverso da zero, suggerisce all'interrupt dispatcher di chiamare la vecchia ISR prima di procedere alla manipolazione di PC-Xinu. Il valore di questo flag per ogni dispositivo è determinato dalle configurazioni di PC-Xinu, discusse nel capitolo 11.

9.3 Controllo del rinvio della schedulazione da parte del processo

Pcxflag è resettato dall'interrupt dispatcher per differire la schedulazione quando viene usato uno stack locale per l'elaborazione dell'interrupt. Ma ci sono occasioni in cui un processo può desiderare il rinvio della schedulazione mentre lascia gli interrupt abilitati. Ad esempio, posizionare il cursore sullo schermo e visualizzare un carattere nella posizione corrente sono chiamate al BIOS separate: se avviene un cambio di contesto tra queste chiamate un altro processo protrebbe muovere il cursore prima che il processo corrente visualizzi il carattere. Per effettuare queste attività in modo indivisibile un processo chiama *xdisable* per rimandare la schedulazione. La dilazione si protrarrà fin quando il processo non chiama *xrestore*.

Le macro *xdisable* e *xrestore* definite in *kernel.h* sono analoghe a *disable* e *restore*, tranne per il fatto che le prime salvano e ripristinano il valore di *pcxflag*. Esse si espandono nella chiamata a due routine di servizio in assembler *sys_xdisabl* e *sys_xrestor*, simili a *sys_disabl* e *sys_restore* discusse nel capitolo 5.

```
; xeidi.asm - _sys_xdisabl, _sys_xrestor, _sys_pcxget, _sys_getstk

      include dos.asm         ; macro di segmento

      dseg
; segmento dati nullo
      endds

      pseg
```

```
      public      _sys_xdisabl, _sys_xrestor, _sys_pcxget, _sys_getstk
      extrn pcxflag:word
      extrn ssave:word
      extrn spsave:word

;-------------------------------------------------------------------------------
; _sys_xdisabl  --  ritorna pcxflag e disabilita il cambio di contesto
;-------------------------------------------------------------------------------
; int sys_xdisabl()
_sys_xdisabl      proc  near
      pushf
      cli                       ; disabilita interrupt
      xor   ax,ax
      xchg  ax,cs:pcxflag
      popf
      ret
_sys_xdisabl      endp

;-------------------------------------------------------------------------------
; _sys_xrestor  --  ripristina pcxflag
;-------------------------------------------------------------------------------
; sys_xrestor(ps)
; int ps;
_sys_xrestor      proc  near
      push  bp
      mov   bp,sp
      pushf
      cli                       ; disabilita interrupt
      mov   ax,[bp+6]           ; ottiene la word flag passata
      mov   cs:pcxflag,ax       ; resetta pcxflag al valore passato
      popf
      pop   bp
      ret
_sys_xrestor      endp

;-------------------------------------------------------------------------------
; _sys_pcxget  --  ottiene il valore corrente di pcxflag
;-------------------------------------------------------------------------------
; int sys_pcxget()
_sys_pcxget proc  near
      pushf
      cli                       ; disabilita interrupt
      mov   ax,cs:pcxflag
      popf
      ret
_sys_pcxget endp

;-------------------------------------------------------------------------------
; sys_getstk  --  ottiene i parametri stack per la visualizzazione panic
;-------------------------------------------------------------------------------
; sys_getstk(sssp,spsp)
; int *sssp, *spsp
_sys_getstk proc  near
      ASSUME DS:DGROUP
      push  bp
      mov   bp,sp
      mov   bx,[bp+4]
      mov   ax,cs:sssave
      mov   [bx],ax
      mov   bx,[bp+6]
      mov   ax,cs:spsave
      mov   [bx],ax
```

```
      pop   bp
      ret
      mov   ax,cs:spsave
      mov   dx,cs:sssave
      ret
_sys_getstk endp

      endps

      end
```

Se la schedulazione è rinviata e il processo corrente chiama *resched* (indirettamente attraverso *send* o *signal*, per esempio), *resched* ritorna innocuamente. Tuttavia, se un processo cambia il suo stato e chiama *resched* (indirettamente attraverso *receive* o *wait*, per esempio) con la schedulazione rimandata, il sistema sarà lasciato in uno stato impossibile. Conseguentemente, il rinvio della schedulazione dovrebbe seguire la semplice regola:

> *Un processo in esecuzione con la schedulazione rimandata può chiamare solo procedure che lasciano il processo nello stato corrente.*

Le chiamate BIOS non possono cambiare lo stato di un processo, perciò sono sicure da chiamare con la schedulazione rinviata.

9.4 Le regole per l'elaborazione dell'interrupt

Poiché le routine di interrupt esaminano e modificano strutture dati globali come buffer di I/O, queste routine devono essere progettate per prevenire l'interferimento di altri processi. Generalmente ciò viene garantito rendendo le routine di interrupt non interrompibili. Per esempio, PC-Xinu disabilita gli interrupt chiamando una routine assembler per pulire il registro FLAGS. Gli interrupt rimangono disabilitati anche se queste chiamano altre procedure. Quando la procedura di alto livello finisce, il controlla ripassa all'interrupt dispatcher, il quale ripristina il registro FLAGS e ritorna al punto in cui il processo era stato originalmente interrotto. Solo dopo il ritorno del dispatcher gli interrupt saranno nuovamente abilitati.

Le routine di interrupt possono anche abilitare le interruzioni chiamando *resched*, se esso alterna a un processo che ha gli interrupt abilitati. L'interrupt del clock, ad esempio, chiama *resched* direttamente, mentre quello della tastiera chiama *send* per indicare che un carattere è disponibile, chiamando indirettamente *resched*. In ogni caso, quando la chiamata raggiunge *resched*, esso dovrebbe permettere l'esecuzione di un processo che aveva gli interrupt abilitati. Così, strutture dati condivise dovrebbero essere lasciate in uno stato valido prima di chiamare qualsiasi routine che cambi contesto. Ricapitoliamo:

> *Gli interrupt saranno disabilitati quando il dispatcher chiama una routine di interrupt di alto livello; questa deve essere strutturata per mantenere ulteriormente gli interrupt disabilitati fino il completamento della modifica delle strutture dati globali.*

Ci sono parecchie caratteristiche da considerare quando si costruiscono routine di interrupt. Per prima cosa, esse non devono mantenere gli interrupt disabilitati troppo a lungo. In caso contrario, i dispositivi falliranno la corretta esecuzione. Per esempio, se il processore non accetta un carattere da una porta seriale prima che un'altro arrivi, quest'ultimo dato sarà perso. Perciò gli interrupt devono essere progettati per essere il più veloce possibile.

Un'altra costrizione sorge perché il codice di interruzione è eseguito da qualsiasi processo che capita in esecuzione quando avvengono gli interrupt. In particolare le routine di interrupt dovrebbero funzionare correttamente anche se eseguiti dal processo *null*. Ricordiamo che *resched* assume ciecamente la presenza di almeno un processo pronto, perciò il processo *null* deve essere sempre *current* o *ready*. La più importante conseguenza è:

> *Le routine di interrupt possono chiamare solo procedure che lasciano il processo in esecuzione nello stato corrente o ready.*

Quindi possono usare routine che sfruttano primitive come *send* o *signal*, ma non quelle che fanno riferimento a primitive come *wait*.

9.5 Schedulazione durante l'elaborazione di un interrupt

Le routine di interrupt dovrebbero permettere la schedulazione? Abbiamo già visto che esse non dovrebbero esplicitamente abilitare ulteriori interrupt durante lo svolgimento di uno di questi e inoltre devono lasciare le strutture dati globali in uno stato valido prima di rischedulare. Ciò potrebbe sembrere non soddisfacibile perché il cambio di contesto verso un processo avente gli interrupt abilitati inizierebbe una sequenza che si ammucchierebbe sino all'overflow. La schedulazione tuttavia è importante perché è l'unica via di influenza delle interruzioni sui processi in esecuzione. Dobbiamo convincerci che la schedulazione da un interrupt è sicura fin quando le strutture dati sono valide.

Per capire questo stato di sicurezza, considera la serie di eventi nascenti da una chiamata di *resched* da una manipolazione delle interruzioni. Supponiamo che un processo P sia in esecuzione con gli interrupt abilitati. Se ne avviene uno l'hardware usa lo stack di P per salvare i registri e continua fin quando viene caricato l'interrupt dispatcher. Poiché tale meccanismo disabilita gli interrupt P esegue queste routine con gli interrupt disabilitati. Rimangono disabilitati anche quando viene chiamata la routine di alto livello. Supponiamo adesso che la routine ad alto livello chiami *resched* il quale passa il controllo ad un altro processo detto Q. Se a Q succede di riabilitare gli interrupt (per esempio dal ritorno di una system call) potrebbe avvenire un altro interrupt. Cosa impedisce un cliclo infinito dove infiniti interrupt si ammucchiano fino all'overflow dello stack? Richiamiamo il fatto che ogni processo ha il proprio stack. Il processo P aveva un interrupt sul suo stack quando è stato fermato da un cambio di contesto. La nuova interruzione avviene quando il processore sta usando lo stack di Q. Prima che un altro interrupt possa accavallarsi sullo stack di P, esso dovrebbe riottenere il controllo della CPU e riabilitare le interruzioni. Ma P stava eseguendo con gli interrupt disabilitati quando ha chiamato lo scheduler. Il cambio di contesto ha salvato P con le interruzioni disabilitate, così, quando P sarà nuovamente schedulato, sarà ripristinato il registro FLAGS e P continuerà con gli interrupt disabilitati.

Essi rimangono disabilitati quando *resched* ritorna alla interrupt routine ad alto livello e quando questa ritorna all'interrupt dispatcher. L'abilitazione avverrà dopo il punto di chiamata del dispatcher. Perciò ulteriori interruzioni non possono verificarsi quando il processo P sta eseguendo il codice di interrupt (sebbene possono avvenire a un altro processo mentre P non è in esecuzione). Solo un numero finito di processi esistono ed ognuno di essi può eseguire, a turno, al massimo un interrupt. Guardando questo differente modo, possiamo dire che:

> *La schedulazione durante l'elaborazione di un interrupt assicura che: (1) le routine di interrupt lasciano i dati globali in uno stato valido prima della*

schedulazione, e (2) nessuna procedura abilita le interruzioni se prima non li disabilita.

Se ricordi, vedrai che abbiamo usato queste regole in tutte le procedure costruite fin qui: una procedura che disabilita gli interrupt in entrata li ripristina sempre prima di ritornare al suo chiamante; nessuna routine li abilita esplicitamente. Poiché le interruzioni sono disabilitate all'entrata dell'interrupt dispatcher, essi sono riprese al suo ritorno. L'unica eccezione alla regola della disabilitazione e ripristino si trova nell'inizializzazione delle procedure che abilitano gli interrupt all'avvio del sistema.

9.6 Vettori di interrupt

I vettori di interrupt usati da PC-Xinu possono essere classificati in due categorie: vettori eccezionali e vettori dei dispositivi di interrupt. I primi saranno discussi con maggiori dettagli nel capitolo 19. Per il momento pensiamoli come vettori corrispondenti a interrupt generati dal processore piuttosto che da una device di input/output. Entrambi gli indirizzi dei vettori sono collezionati nel file *bios.h*.

```
/* bios.h */

/*------------------------------------------------------------------------
 * Informazioni per il PC dell'interfaccia ROM BIOS
 *------------------------------------------------------------------------
 */

/* vettori di eccezione */
#define DB0VEC    0x00            /* divisione per zero */
#define SSTEPVEC  0x01            /* passo singolo */
#define BKPTVEC   0x03            /* breakpoint */
#define OFLOWVEC  0x04            /* overflow */

/* vettori di interrupt dei dispositivi */
#define CLKVEC    0x08            /* clock */
#define KBDVEC    0x09            /* tastiera */
#define COM1VEC   0x0b            /* COM1 */
#define COM2VEC   0x0c            /* COM2 */
#define FLOPVEC   Ox0e            /* floppy */
#define PRLLVEC   0x0f            /* porta parallela */
#define CBRKVEC   0x1b            /* Ctrl-Break */

#define BIOSFLG   0x100           /* flag BIOS per intmap */

extern int _panic();              /* handler d'eccezione */
extern int cbrkint();             /* handler ctrl-break */
extern int clkint();              /* handler dell'interrupt clock */
```

Delle sette interruzioni dei dispositivi solo CLKVEC, KBDVEC e CBRKVEC sono usate da PC-Xinu.