INTERRUPT ed ECCEZIONI

Un interrupt viene definito di solito come un evento che altera la sequenza delle istruzioni eseguite da un processore. Gli eventi corrispondono a segnali elettrici generati da circuiti hardware sia all'interno che all'esterno del chip della CPU.

Gli interrupt si dividono in sincroni e asincroni:

- *interrupt sincroni* sono prodotti dall'unità di controllo della CPU mentre esegue istruzioni e sono detti sincroni perché generati solo dopo il termine dell'esecuzione di una istruzione
- *interrupt asincroni* sono generati da altri dispositivi hardware in qualsiasi momento rispetto ai segnali di clock della CPU.

I manuali Intel definiscono *eccezioni* gli interrupt sincroni ed *interrupt* gli interrupt asincroni. Qui si adotta questa classificazione; occasionalmente si usa il termine segnale di interrupt per indicare entrambi i tipi. Gli interrupt sono generati da timer o dispositivi di I/O; ad esempio la pressione di un tasto da parte dell'utente provoca un interrupt.

Le eccezioni invece sono causate o da errori di programmazione o da condizioni anomale gestite dal kernel. Nel primo caso il kernel invia al processo corrente uno dei segnali familiari ad ogni programmatore Unix. Nel secondo caso esegue i passi necessari a recuperare una condizione anomala, come una Page Fault o la richiesta di un servizio del kernel attraverso l'istruzione assembly int o sysenter.

Vengono dapprima descritte le motivazioni dell'introduzione degli interrupt. In seguito si descrive come un IRQ (Interrupt ReQuest) generato da un dispositivo di I/O origina un interrupt e come i processori 80x86 gestiscano interrupt ed eccezioni a livello hardware. Poi si descrive come Linux inizializza tutte le strutture dati necessarie e come gestisce gli interrupt a livello software. Non vengono trattati gli interrupt non standard di alcune architetture, ma solo gli interrupt "classici" comuni a tutti i PC.

Ruolo dei segnali di interrupt

Come suggerisce il nome, i segnali di interrupt forniscono un modo per deviare il processore su codice al di fuori del normale flusso di controllo. Quando giunge un segnale di interrupt la CPU deve sospendere ciò che sta facendo e passare ad una nuova attività; lo fa salvando il valore corrente del contatore di programma (registri eip e cs) nello stack in modalità del kernel e ponendo nel contatore stesso un indirizzo legato al tipo di interrupt.

Ci sono alcune similitudini con la commutazione di processo, ma esiste una fondamentale differenza: il codice eseguito dal gestore di un interrupt o di un'eccezione non è un processo. Piuttosto è una sequenza di istruzioni eseguita dal kernel per gestire l'interrupt (kernel control path, KCP) a spese del processo in esecuzione in quel momento. Al pari di un KCP, un gestore di interrupt è più leggero di un processo, ha un contesto più agile e richiede meno tempo ad essere impostato e poi rilasciato.

La gestione di interrupt è uno dei compiti più delicati del kernel, perché deve soddisfare le seguenti condizioni:

- Gli interrupt possono essere generati in ogni momento, mentre il kernel sta portando a termine qualche altro compito. L'obiettivo diventa dirottare l'interrupt e differirne la gestione il più possibile. Ad esempio, quando arriva un blocco di dati su una linea di rete e l'hardware invia un interrupt, il kernel può semplicemente segnalare la presenza di dati, riprendere l'esecuzione precedente il segnale

e svolgere in seguito il lavoro di spostare i dati in un buffer dove il processo in attesa può trovarli. Le azioni necessarie in risposta ad un interrupt si dividono in una parte urgente che il kernel esegue subito e in una parte che si può rinviare.

- Poiché gli interrupt possono capitare in ogni momento, può succedere che un interrupt sia ricevuto dal kernel mentre ne sta gestendo un altro. I gestori di interrupt devono essere strutturati in modo che i corrispondenti KCP possano essere nidificati. Quando termina l'ultimo KCP, il kernel deve riprendere l'esecuzione del processo interrotto o passare ad un altro se l'interrupt ha causato l'intervento dello scheduler.

- Anche se il kernel può accettare un interrupt mentre ne sta gestendo un altro, esistono alcune regioni critiche nel codice del kernel entro le quali gli interrupt devono essere disabilitati. Esse vanno limitate il più possibile e, per soddisfare la condizione precedente, il kernel deve essere eseguito per la maggior parte del tempo con gli interrupt abilitati.

Interrupt ed eccezioni

La documentazione di Intel riporta la seguente classificazione. **Interrupt**:

interrupt mascherabili: sono originati da tutte le IRQ dei dispositivi di I/O. Un interrupt mascherabile può trovarsi in due stati: mascherato o non mascherato; un interrupt mascherato viene ignorato dall'unità di controllo finché resta tale.

interrupt non mascherabili: sono originati solo da pochi eventi critici, tipo problemi hardware. Sono sempre riconosciuti dalla CPU.

Eccezioni: *Eccezioni rilevate dal processore*: hanno origine quando la CPU rileva una condizione anomala durante l'esecuzione di una istruzione. Sono divise in tre gruppi, a seconda del valore del registro eip salvato nello stack al rilevamento della eccezione stessa

- *fault*: in genere può essere corretta, e una volta fatto ciò, il programma può ripartire senza soluzione di continuità. Il valore di eip è l'indirizzo della istruzione che ha provocato la fault, ed è l'istruzione che può essere rieseguita quando il gestore di eccezione termina. Riprendere la stessa istruzione è necessario ogni volta che il gestore è capace di correggere la condizione anomala.

- *trap*: segnalata subito dopo l'esecuzione dell'istruzione che l'ha provocata; quando il kernel restituisce il controllo al programma, può riprendere senza soluzione di continuità. Il valore di eip è l'indirizzo dell'istruzione successiva a quella che ha provocato la trap. Una trap viene generata solo quando non è necessario eseguire di nuovo l'istruzione terminata. L'impiego prevalente è nel debugging. Il suo ruolo è quello di segnalare al debugger che una istruzione specifica è stata eseguita (ad esempio è stato raggiunto un breakpoint). Dopo che l'utente ha esaminato il breakpoint, può chiedere che venga ripresa l'esecuzione a partire dall'istruzione successiva.

- *abort*: si è in presenza di un errore grave; l'unità di controllo è in difficoltà e può non essere in grado di memorizzare nel registro eip la locazione precisa dell'istruzione che ha causato l'eccezione. E' usata per segnalare errori gravi, come guasti hardware e valori non validi o non consistenti nelle tabelle di sistema. Il segnale di interrupt inviato è un segnale di emergenza usato per passare il controllo al gestore, il quale non può fare altro che forzare l'arresto del processo.

Eccezioni programmate: avvengono su richiesta del programmatore per mezzo delle istruzioni int o int3; anche into (controllo dell'overflow) e bound (controlla che un indirizzo sia compreso tra due valori) danno origine a eccezioni programmate quando il controllo della condizione non risulta vero. Sono gestite dall'unità di controllo come trap e sono spesso chiamate *interrupt software*. Queste eccezioni hanno due usi

comuni: implementare chiamate di sistema e inviare notifiche al debugger.

Ogni interrupt o eccezione è identificata da un valore tra 0 e 255; Intel chiama questo numero senza segno a 8 bit *vettore*. I vettori di interrupt non mascherabili e delle eccezioni sono fissi, mentre quelli degli interrupt mascherabili possono essere manipolati programmando il controllore degli interrupt.

IRQ e interrupt

Ogni controller di dispositivo hardware in grado di generare richieste di interrupt ha normalmente una singola linea di output definita linea di Interrupt ReQuest (IRQ). Componenti più sofisticati come le schede PCI usano più linee, fino a 4. Tutte le linee di IRQ sono collegate ai pin di input di un circuito hardware chiamato Programmable Interrupt Controller (controllore di interruzioni programmabile – PIC), che svolge i seguenti compiti:

1 - Controlla le linee IRQ rilevando i segnali inviati. Se vengono attivate due o più linee, seleziona quella con il numero di pin inferiore.

2 - Se rileva un segnale:

a – converte il segnale nel vettore corrispondente;

b – pone il vettore in una porta di I/O, in modo che la CPU possa leggerlo attraverso il bus dati;

c – invia un segnale al pin INTR del processore;

d- attende fino a che la CPU accetta il segnale scrivendo in una delle porte di I/O del PIC; quando ciò avviene, azzera la linea INT;

3 - Torna al primo punto.

Le linee di IRQ sono numerate in sequenza a partire dallo 0, per cui la prima è IRQ0. Il vettore Intel di default associato a IRQn è n+32. La mappatura di IRQ nei vettori può essere modificata con opportune istruzioni di I/O alle porte del PIC.

Ogni linea può essere disabilitata selettivamente. Perciò il PIC può essere programmato per disabilitare IRQ, cioè gli si può ordinare di inoltrare o non inoltrare interrupt che si riferiscono ad una certa linea IRQ. Gli interrupt disabilitati non vanno persi; il PIC li invia alla CPU appena sono di nuovo abilitati. Questa tecnica è adottata da molti gestori di interrupt perché permette di processare IRQ dello stesso tipo in serie.

L'abilitazione/disabilitazione selettiva degli IRQ non è uguale al mascheramento/non mascheramento degli interrupt mascherabili. Quando il flag IF del registro eflags è azzerato, ogni interrupt mascherabile che proviene dal PIC viene temporaneamente ignorato dalla CPU. Le istruzioni assembly cli e sti rispettivamente azzerano e attivano il flag.

I PIC tradizionali sono implementati connettendo a cascata due chip di tipo 8259A. Ogni chip può gestire 8 diverse linee IRQ. Dato che la linea INT output dello slave è connessa al pin IRQ2 del PIC master, il numero di IRQ disponibili è 15.

Advanced Programmable Interrupt Controller (APIC)

La descrizione precedente si riferisce al PIC progettato per sistemi uniprocessore. In essi l'output del PIC master può essere connesso direttamente al pin INTR della CPU. Se il sistema ha due o più CPU, questo

schema non è applicabile e occorre un sistema più complesso.

Inoltrare gli interrupt ad ogni CPU è cruciale per sfruttare appieno il parallelismo nei sistemi SMP. Per questo motivo, a partire dal Pentium III Intel ha introdotto un nuovo componente chiamato *I/O Advanced Programmable Interrupt Controller (I/O APIC)*. Questo chip è la versione avanzata del vecchio 8259A; per supportare vecchi sistemi operativi, alcune schede madri li montano entrambi. In più, tutti i microprocessori attuali includono un APIC locale; ognuno di essi ha registri a 32 bit, un clock interno, un timer locale e due linee IRQ addizionali, LINT0 e LINT1 riservate agli interrupt dell'APIC locale. Tutti gli APIC locali sono collegati ad un I/O APIC esterno, creando così un sistema multi-APIC.

Lo schema di collegamento prevede un bus APIC che connette l'I/O APIC esterno con gli APIC locali di ogni CPU. Le linee di IRQ provenienti dai dispositivi sono connesse all'I/O APIC, che quindi si comporta come un router rispetto ai chip locali. Nelle schede madri del Pentium III il bus era seriale a tre linee; a partire dal Pentium 4 il bus è stato implementato per mezzo del bus di sistema. Per il resto, dato che il bus e i messaggi sono invisibili al software, non richiedono ulteriori spiegazioni.

L'I/O APIC comprende 24 linee IRQ, una Tabella di Ridirezione degli Interrupt (TRI) con 24 entry, registri programmabili e un'unità per i messaggi per comunicare sul bus APIC. A differenza del 8259A, la priorità degli interrupt non è legata al numero del pin: ogni entry della TRI può essere programmata individualmente per indicare il vettore e la priorità, il processore di destinazione e come esso viene scelto. L'informazione contenuta nella Tabella è usata per tradurre ogni segnale di IRQ esterno in un messaggio destinato ad uno o più APIC locale attraverso il bus.

Gli interrupt possono essere distribuiti fra le CPU in due modi:

distribuzione statica: il segnale viene inviato alla CPU indicata nella entry della TRI e può essere destinato a una specifica CPU, a un sottoinsieme delle CPU o a tutte contemporaneamente (broadcast);

distribuzione dinamica: il segnale viene inviato all'APIC locale del processore che esegue il processo con la priorità più bassa. Ogni APIC locale ha un task priority register (registro di priorità del task – TPR), usato per calcolare la priorità del processo corrente. Intel si aspetta che il registro venga modificato dal sistema operativo ad ogni commutazione di processo.

Se due o più CPU hanno la priorità più bassa, il carico viene ripartito usando la tecnica dell'arbitrato. Ad ogni CPU è assegnata una diversa priorità di arbitrato che va da 0 (inferiore) a 15 (superiore). Ogni volta che un interrupt viene inviato a una CPU, la sua priorità di arbitrato è posta a 0, mentre quella delle altre CPU è aumentata. Quando il registro di priorità contiene un valore superiore a 15, viene settato al valore di priorità della CPU vincente aumentato di 1. Di conseguenza gli interrupt sono distribuiti con un metodo simile al Round Robin tra le CPU con la stessa priorità.

Oltre a distribuire gli interrupt il sistema multi-APIC consente di generare *interrupt interprocessore*. Quando una CPU vuole inviare un interrupt a un'altra, memorizza il vettore di interrupt e l'identificatore dell'APIC locale bersaglio nell' Interrupt Command Register (ICR) del proprio APIC locale. Quindi un messaggio viene inviato sul bus all'APIC locale bersaglio, che di conseguenza inoltra l'interrupt Corrispondente alla propria CPU. Interrupt interprocessore sono componenti fondamentali dell'architettura SMP. Sono usati attivamente da Linux per scambiare messaggi tra CPU.

La maggior parte dei sistemi uniprocessore include un chip I/O APIC che può essere configurato in due modi:

- come un PIC 8259A esterno standard connesso alla CPU. L'APIC locale è disabilitato e le due linee LINT0 e LINT1 sono configurate come pin INTR e NMI rispettivamente
- come un I/O APIC esterno standard. L'APIC locale è abilitato e tutti gli interrupt esterni sono ricevuti

attraverso l'I/O APIC.

Eccezioni

I processori 80x86 ammettono una ventina di eccezioni (a seconda del modello). Il kernel deve fornire un gestore per ogni tipo. Per alcune di esse l'unità di controllo della CPU genera un codice di errore hardware e lo inserisce nello stack in modalità del kernel prima di avviare il gestore di eccezioni.

L'elenco seguente fornisce il vettore, il nome, il tipo e una breve descrizione delle eccezioni dei processori 80x86; ulteriori informazioni si trovano nella documentazione Intel.

0 - "Divide error" (fault): quando un programma tenta una divisione tra un intero e zero.

1 - "Debug" (trap o fault): quando il flag TF di eflags è settato per implementare l'esecuzione a passo singolo di un programma ad opera del debugger o quando l'indirizzo di una istruzione o di un operando ricade nell'intervallo di un registro di debug.

2 – non usata: riservata agli interrupt non mascherabili (quelli che usano il pin NMI)

3 - "Breakpoint" (trap): provocata da un'istruzione int3 (breakpoint).

4 - "Overflow" (trap): un'istruzione into (controllo dell'overflow) viene eseguita mentre il flag OF di eflags (oveflow) è attivato.

5 - "Bounds check" (fault): una istruzione bound (controlla che un indirizzo sia compreso tra due valori dati) viene eseguita su un operando che eccede i limiti dell'intervallo.

6 - "Invalid opcode" (fault): la CPU trova un operando (la parte dell'istruzione in codice macchina che indica l'operazione da eseguire) non valido.

7 - "Device not available" (fault): un'istruzione ESCAPE, MMX o SSE/SSE2 viene eseguita con il flag TS di cr0 attivato.

8 - "Double fault" (abort): normalmente quando la CPU incontra una eccezione mentre sta chiamando il gestore per una eccezione precedente, esse possono essere gestite in serie. Se in qualche caso non lo può fare, genera questa eccezione.

9 - "Coprocessor segment overrun" (abort): problemi col coprocessore matematico (solo nei vecchi 386).

10 - "Invalid TSS" (fault): la CPU ha tentato una commutazione di contesto con un processo che ha un TSS non valido.

11 - "Segment not present" (fault): viene referenziato un segmento non presente in memoria (flag Segment-Present azzerato).

12 - "Stack segment fault" (fault): l'istruzione tenta di superare il limite del segmento dello stack, o il segmento indicato da ss non è presente in memoria.

13 - "General protection" (fault): una delle regole della modalità protetta del 80x86 è stata violata.

14 - "Page Fault" (fault): la pagina indirizzata non è in memoria, l'entry della Tabella di Pagina è nulla oppure è stato violato uno dei meccanismi di protezione della paginazione.

15 – riservato da Intel

16 - “Floating-point error” (fault): l'unità in virgola mobile della CPU segnala un errore, ad esempio un overflow o una divisione per 0.

17 - “Alignment check” (fault): l'indirizzo di un operando non è allineato correttamente (ad es. non è multiplo di 4).

18 - “Machine check” (abort): un meccanismo di controllo ha rilevato un errore nella CPU o nel bus.

19 - “SIMD floating point exception” (fault): l'unità SSE o SSE2 integrata nella CPU segnala un errore in una operazione in virgola mobile.

I valori da 20 a 31 sono riservati da Intel per usi futuri. Ogni eccezione è gestita da un gestore specifico che di solito invia un segnale al processo che ha provocato l'eccezione.

eccezione	gestore	segnale
0	divide_error()	SIGFPE
1	debug()	SIGTRAP
2	nmi()	nessuno
3	int3()	SIGTRAP
4	overflow()	SIGSEGV
5	bounds()	SIGSEGV
6	invalid_op()	SIGILL
7	device_not_available()	nessuno
8	doublefault_fn()	nessuno
9	coprocessor_segment_overrun()	SIGFPE
10	invalid_TSS()	SIGSEGV
11	segment_not_present()	SIGBUS
12	stack_segment()	SIGBUS
13	general_protection()	SIGSEGV
14	page_fault	SIGSEGV
15	nessuno	nessuno
16	coprocessor_error()	SIGFPE
17	alignment_check()	SIGBUS
18	machine_check()	nessuno
19	simd_coprocessor_error()	SIGFPE

Interrupt Descriptor Table

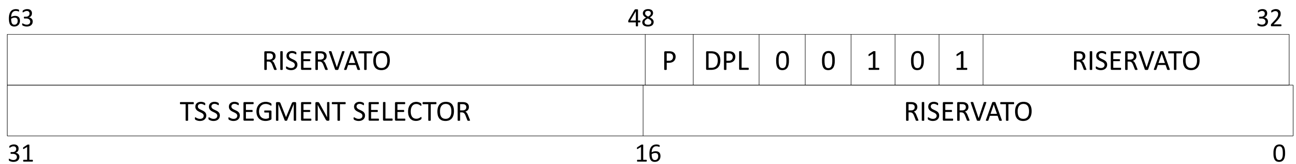
Una tabella chiamata *Tabella dei Descrittori degli Interrupt* (IDT) associa ogni vettore di interrupt o di eccezione all'indirizzo del corrispondente gestore. La IDT deve essere opportunamente inizializzata dal kernel prima di abilitare gli interrupt. Il formato è simile a quello della GDT o LDT. Ogni entry corrisponde ad un vettore di interrupt o di eccezione e consiste di un descrittore di 8 byte; perciò sono richiesti al massimo

8 x 256 = 2048 byte per memorizzare la IDT.

Il registro idtr della CPU consente alla IDT di essere allocata ovunque in memoria; esso specifica sia l'indirizzo base che il limite (cioè la massima lunghezza) della IDT. Deve essere inizializzato prima dell'abilitazione degli interrupt per mezzo dell'istruzione assembly lidt.

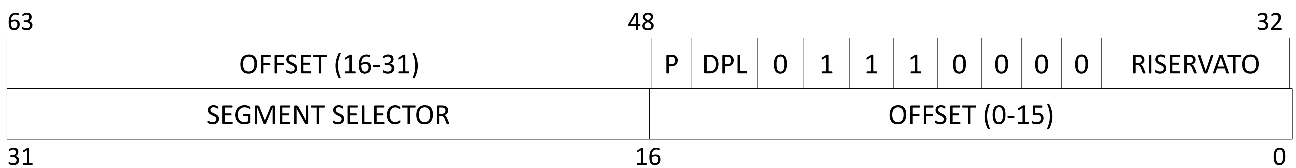
Una IDT può contenere tre tipi di descrittori; in particolare il campo Type codificato dai bit 40-43 identifica il tipo di descrittore.

Task Gate Descriptor



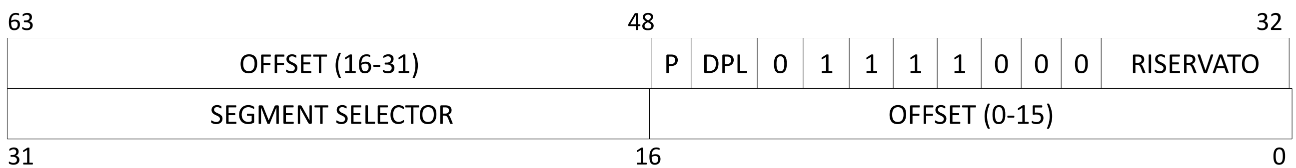
63	48							32
RISERVATO	P	DPL	0	0	1	0	1	RISERVATO
TSS SEGMENT SELECTOR	RISERVATO							
31	16							0

Interrupt Gate Descriptor



63	48										32
OFFSET (16-31)	P	DPL	0	1	1	1	0	0	0	0	RISERVATO
SEGMENT SELECTOR	OFFSET (0-15)										
31	16										0

Trap Gate Descriptor



63	48										32
OFFSET (16-31)	P	DPL	0	1	1	1	1	0	0	0	RISERVATO
SEGMENT SELECTOR	OFFSET (0-15)										
31	16										0

I descrittori sono:

Task gate: include il TSS del processo che deve sostituire quello corrente al ricevimento del segnale di interrupt.

Interrupt gate: include il Selettore di Segmento e l'offset di un gestore di interrupt o eccezione. Quando trasferisce il controllo al segmento appropriato, il processore azzera il flag IF, disabilitando gli interrupt mascherabili.

Trap gate: simile al precedente, ma non viene modificato IF.

Linux usa gli interrupt gate per gestire gli interrupt e i trap gate per gestire le eccezioni. “Double fault” è l'unica eccezione che viene gestita per mezzo di un task gate.

Gestione hardware di interrupt ed eccezioni

In questa fase, il kernel è già inizializzato e la CPU è in modalità protetta. Dopo aver eseguito una istruzione, i registri cs ed eip contengono l'indirizzo logico della istruzione successiva. Prima di procedere, l'unità di controllo verifica se siano stati generati un interrupt o una eccezione: se si, la CPU compie le azioni seguenti:

1 – Determina il vettore i (0 ≤ i ≤ 255) associato all'interrupt o all'eccezione.

2 – Legge la i-esima entry della IDT indicata dal registro idtr.

3 – Ottiene l'indirizzo base della GDT dal registro gdtr e consulta la GDT per estrarre il Descrittore di Segmento identificato dal selettore nella entry della IDT. Questo descrittore contiene l'indirizzo base del segmento che include il gestore di I o di eccezioni.

4 – Controlla che l'interrupt provenga da una origine autorizzata. Prima di tutto, confronta il Current Privilege Level (CPL) memorizzato nei due bit meno significativi del registro cs con il Descriptor Privilege Level (DPL) del Descrittore di Segmento incluso nella GDT. Se CPL è inferiore a DPL genera una eccezione di “General protection”, perché il gestore di interrupt non può avere un privilegio inferiore a quello del programma che ha generato l'interrupt In caso di eccezione programmata, esegue un altro controllo: confronta il CPL con il DPL del gate descriptor incluso nella IDT e genera un eccezione di “General protection” se DPL è inferiore a CPL. Quest'ultimo controllo previene gli accessi delle applicazioni utente a specifici trap o interrupt gate.

5 – Controlla se è in corso un cambio di livello di privilegio, cioè se CPL è diverso dal DPL del Descrittore di Segmento. Se così è, la CPU deve usare lo stack associato al nuovo livello di privilegio, e perciò:

a – legge il registro tr per accedere al TSS del processo corrente;

b – carica nei registri ss e esp il valore adatto di segmento e di puntatore di stack associati al nuovo livello di privilegio; tali valori si trovano nel TSS;

c – salva nel nuovo stack i valori precedenti di ss e esp.

6 – Se si è verificata una fault, carica in cs ed eip l'indirizzo logico della istruzione che ha provocato l'eccezione, in modo da poterla eseguire di nuovo.

7 – Salva il contenuto di eflags, cs ed eip nello stack.

8 – Se l'eccezione è accompagnata da un codice di errore, lo salva nello stack.

9 – carica in cs ed eip il Selettore di Segmento e l'Offset del Gate Descriptor contenuto nella i-esima entry della IDT. Questi valori rappresentano l'indirizzo logico della prima istruzione del gestore di interrupt o di eccezione.

Il punto 9 equivale ad un salto al gestore; l'istruzione processata dall'unità di controllo dopo essersi occupata del segnale di I è la prima istruzione del gestore selezionato.

Dopo che il gestore ha svolto il suo compito, deve restituire il controllo al processo interrotto eseguendo l'istruzione iret, che fa sì che l'unità di controllo:

1 - carichi nei registri cs, eip ed eflags i valori salvati nello stack. Se il codice di errore è stato inserito in cima allo stack, esso deve essere estratto prima di eseguire iret;

2 – controlli se il CPL del gestore è uguale al valore contenuto nei due bit meno significativi si cs (significa che il processo interrotto era eseguito con lo stesso livello di privilegio del gestore)). Se è uguale, iret termina; diversamente, la CPU passa al punto successivo;

3 – carichi in ss ed esp i valori salvati nello stack e ritorni allo stack associato al precedente livello di privilegio;

4 – esamini il contenuto dei registri di segmento ds, es, fs e gs; se uno di loro contiene un selettore che si riferisce a un Descrittore di Segmento con DPL inferiore a CPL, lo azzera. Questo evita che programmi in

modalità utente con CPL = 3 usino segmenti usati in precedenza da routine del kernel (con DPL = 0). Se i registri non venissero azzerati, si potrebbe tentare un exploit per accedere allo spazio di indirizzi del kernel.

Esecuzione nidificata di gestori di interrupt ed eccezioni

Ogni interrupt o eccezione dà origine ad un KCP o ad una sequenza di istruzioni che opera in modalità del kernel per conto del processo corrente. Ad esempio, quando un dispositivo di I/O genera in interrupt, la prima azione del KCP consiste nel salvare i registri nello stack in modalità del kernel, e l'ultima nel ripristinarli.

I KCP possono essere nidificati senza limiti; un gestore può essere interrotto da un altro gestore. Di conseguenza, non è detto che l'ultima istruzione di un KCP che gestisce un interrupt riporti il il processo corrente in modalità utente: può accadere che riporti in esecuzione il KCP interrotto e che la CPU continui in modalità del kernel.

Il prezzo da pagare è che un gestore di interrupt non può mai essere bloccante, cioè durante la sua esecuzione non può avvenire una commutazione di processo. Infatti, i valori necessari a ripristinare l'esecuzione di un KCP sono salvati nello stack in modalità del kernel collegato al processo corrente.

Ipotizzando che il kernel non abbia bug, le eccezioni dovrebbero verificarsi quasi esclusivamente in modalità utente, dato che sono generate da errori di programmazione o dai debugger. Una “Page Fault”, invece, ha luogo anche in modalità del kernel; avviene quando un processo tenta di indirizzare una pagina del suo spazio di indirizzamento non presente in RAM. Nel gestire questa eccezione, il Kernel può sospendere il processo corrente e rimpiazzarlo con un altro fino a che la pagina non sia disponibile. Il KCP che gestisce l'eccezione riprende l'esecuzione appena il processo ottiene di nuovo la CPU.

Poiché il gestore di “Page Fault” non dà origine ad altre eccezioni, al massimo due KCP associati all'eccezione (il primo derivato da una chiamata di sistema, il secondo dalla “Page Fault”) devono essere impilati sullo stack, uno di seguito all'altro.

A differenza delle eccezioni, gli interrupt non usano strutture specifiche del processo corrente, anche se il KCP che li gestisce opera nel suo ambiente di esecuzione. Come conseguenza, non è possibile prevedere quale processo sarà in esecuzione quando giunge un interrupt.

Un gestore di interrupt può prendere il posto di un altro gestore di interrupt o di eccezione prima del suo termine (preemption o pre-rilascio), mentre un gestore di eccezione non può mai prendere il posto di un gestore di interrupt. L'unica eccezione che può sorgere in modalità del kernel è la “Page Fault”; i gestori di interrupt però non compiono mai operazioni che possano condurre ad una Page Fault e quindi ad una commutazione di contesto.

Linux adotta la preemption per i KCP per due ragioni:

- per migliorare le prestazioni dei controller programmabili di interrupt e di dispositivo. Se ad esempio il controller di un dispositivo invia un segnale di IRQ, il PIC lo trasforma in un interrupt esterno e sia il PIC che il controller rimangono bloccati finché il PIC non riceve la conferma dalla CPU. Grazie al pre-rilascio dei KCP, il kernel è in grado di inviare la conferma anche quando sta gestendo un interrupt precedente;

- per implementare un modello di interrupt senza livelli di priorità. Poiché l'esecuzione di ogni gestore può essere differita da un altro gestore, non c'è necessità di stabilire priorità predefinite fra dispositivi hardware. Ciò semplifica il codice del kernel e ne migliora la portabilità.

In sistemi multiprocessore, possono essere in esecuzione concorrente molti KCP; un KCP associato ad una

eccezione può iniziare l'esecuzione su una CPU e spostarsi su di un'altra a causa di una commutazione di processo.

Inizializzazione della Interrupt Descriptor Table

Dopo la descrizione del livello hardware, si passa all'inizializzazione della IDT. Prima di abilitare gli interrupt il kernel deve caricare l'indirizzo iniziale della IDT nel registro idtr e inizializzare le sue entry. Queste operazioni sono svolte nella fase di inizializzazione del sistema.

L'istruzione int permette ad un processo in modalità utente di inviare un segnale di interrupt con vettore compreso tra 0 e 255. La inizializzazione della IDT va fatta con cautela per bloccare interrupt illegali ed eccezioni simulate da processi attraverso int. Ciò può avvenire impostando a 0 il DPL di particolari descrittori; quando un processo invia uno di questi segnali, viene generata un'eccezione di “General protection”.

In alcuni casi, però, un processo in modalità utente deve poter generare un'eccezione programmata. Per consentire ciò, è sufficiente impostare a 3 il DPL del descrittore corrispondente.

Interrupt, Trap e System Gates

Intel fornisce tre tipi di descrittori di interrupt: Task, Interrupt e Trap Gate Descriptor. Linux usa una terminologia diversa per i descrittori della IDT:

Interrupt gate: è un interrupt gate Intel con DPL = 0, non accessibile da processi in modalità utente. Tutti i gestori di interrupt sono definiti in questo modo.

System gate: un trap gate Intel con DPL = 3, accessibile in modalità utente. Usato per i gestori delle eccezioni 4, 5 e 128; di conseguenza, le tre istruzioni into, bound e int $0x80 possono essere impiegate in modalità utente.

System interrupt gate: un interrupt gate Intel con DPL = 3; il gestore associato all'eccezione 3 è definito in questo modo; di conseguenza l'istruzione int3 può essere impiegata in modalità utente.

Trap gate: un trap gate Intel con DPL = 0; la maggior parte dei gestori delle eccezioni è definita così.

Task gate: un task gate Intel con DPL = 0; il gestore dell'eccezione “Double fault” è definito in questo modo.

Per inserire gate nella IDT vengono usate queste funzioni dipendenti dall'architettura:

set_intr_gate(n, addr): inserisce un interrupt gate nella n-esima entry della IDT. Il Selettore di Segmento è impostato al Selettore di Segmento del codice del kernel. Il campo Offset è impostato ad addr, cioè all'indirizzo del gestore di interrupt; DPL = 0.

set_system_gate(n, addr): inserisce un trap gate nella n-esima entry della IDT. Il Selettore di Segmento è impostato al Selettore di Segmento del codice del kernel. Il campo Offset è impostato ad addr, cioè all'indirizzo del gestore di eccezioni; DPL = 3.

set_system_intr_gate(n, addr): inserisce un interrupt gate nella n-esima entry della IDT. Il Selettore di Segmento è impostato al Selettore di Segmento del codice del kernel. Il campo Offset è impostato ad addr, cioè all'indirizzo del gestore di eccezioni; DPL = 3.

set_trap_gate(n, addr): simile alla precedente ma con DPL = 0.

set_task_gate(n, addr): inserisce un task gate nella n-esima entry della IDT. Il Selettore di Segmento contiene l'indice nella GDT del TSS che punta alla funzione da attivare. Il campo Offset vale 0; DPL = 3.

Inizializzazione preliminare della IDT

La IDT viene inizializzata ed usata dalle routine del BIOS quando il computer è ancora in modalità reale. Quando Linux assume il controllo, la IDT viene spostata in un'altra zona di RAM e reinizializzata poiché Linux non usa le routine del BIOS.

La IDT è contenuta nella tabella idt_table, che ha 256 entry. La variabile di 6 byte idt_descr contiene sia la dimensione che l'indirizzo della IDT, ed è usata nella fase di inizializzazione del sistema quando il kernel imposta il registro idtr con l'istruzione assembly lidt.

Durante l'inizializzazione del kernel la funzione assembly setup_idt() riempie le 256 entry di idt_table con lo stesso interrupt gate che punta al gestore di interrupt ignore_int():

```
setup_idt:
        lea ignore_int, %edx
        movl $(__KERNEL_CS << 16), %eax
        movw %dx, %ax          /* selector = 0x0010  = cs */
        movw $0x8e00, %dx      /* interrupt gate, dp1 = 0, present */
        lea idt_table, %edi
        mov $256, %ecx
rp_sidt:
        movl %eax, (%edi)
        movl %edx, 4(%edi)
        addl $8, %edi
        dec %ecx
        jne rp_sidt
        ret
```

Il gestore ignore_int(), scritto in assembly, è un gestore nullo che esegue le seguenti azioni:

1 – salva il contenuto di alcuni registri nello stack;

2 – chiama la funzione printk() per stampare il messaggio “Unknown interrupt”;

3 – ripristina il contenuto dei registri;

4 – esegue una istruzione iret per riavviare il programma interrotto.

In condizioni normali non dovrebbe venire mai eseguito. La comparsa del messaggio “Unknown interrupt” sulla console o nei file di log evidenzia o un problema di hardware (un dispositivo invia segnali di interrupt non previsti) o un problema del kernel (un interrupt o un'eccezione vengono gestiti in modo non appropriato).

Dopo questa prima fase, il kernel rimpiazza nella IDT alcune delle entry nulle con valori significativi di gestori di trap e interrupt. Alla fine la IDT possiede un gestore specifico per ogni eccezione ed interrupt; di seguito viene esposto come ciò avviene.

Gestione delle eccezioni

La maggior parte delle eccezioni sono interpretate da Linux come condizioni di errore, e il kernel invia al processo che ha provocato l'eccezione un segnale per notificare la condizione anomala. Se un processo tenta una divisione per 0, la CPU genera una eccezione “Divide error” ed il gestore invia un segnale SIGFPE al processo , che tenta di rimediare oppure, se non ha un gestore per il segnale, termina.

Ci sono un paio di casi in cui Linux sfrutta le eccezioni per gestire con maggiore efficienza l'hardware. Un primo caso è l'uso di “Device not available” assieme al flag TS del registro cr0 per forzare il caricamento dei registri in virgola mobile con nuovi valori. Il secondo riguarda l'uso di “Page Fault” per rinviare l'allocazione di frame di pagina il più possibile. Il gestore corrispondente è complesso perché l'eccezione può a volte indicare una condizione di errore, mentre altre volte no.

I gestori di eccezioni hanno una struttura standard:

1 – Salvano il contenuto di vari registri nello stack in modalità del kernel (questa parte di codice è scritta in assembly).

2 – Gestiscono l'eccezione per mezzo di funzioni scritte in C.

3 – Escono per mezzo della funzione ret_from_exception().

L'IDT deve essere inizializzata con un gestore per ogni eccezione riconosciuta; questo è il compito della funzione trap_init(), che inserisce i valori definitivi nelle entry che si riferiscono alle eccezioni e agli interrupt non mascherabili. Lo fa servendosi delle funzioni set_trap_gate(), set_intr_gate(), set_system_gate(), set_system_intr_gate() e set_task_gate():

```
        set_trap_gate(0,&divide_error);
        set_intr_gate(1,&debug);
        set_intr_gate(2,&nmi);
        set_system_intr_gate(3, &int3); /* int3-5 can be called from all */
        set_system_gate(4,&overflow);
        set_system_gate(5,&bounds);
        set_trap_gate(6,&invalid_op);
        set_trap_gate(7,&device_not_available);
        set_task_gate(8,GDT_ENTRY_DOUBLEFAULT_TSS);
        set_trap_gate(9,&coprocessor_segment_overrun);
        set_trap_gate(10,&invalid_TSS);
        set_trap_gate(11,&segment_not_present);
        set_trap_gate(12,&stack_segment);
        set_trap_gate(13,&general_protection);
        set_intr_gate(14,&page_fault);
        set_trap_gate(15,&spurious_interrupt_bug);
        set_trap_gate(16,&coprocessor_error);
        set_trap_gate(17,&alignment_check);
#ifdef CONFIG_X86_MCE
        set_trap_gate(18,&machine_check);
#endif
        set_trap_gate(19,&simd_coprocessor_error);
        set_system_gate(SYSCALL_VECTOR,&system_call);
```

La “Double fault” è gestita per mezzo di un task gate perché indica un serio problema del kernel. Perciò il

gestore che tenta di stampare il contenuto dei registri non si fida del valore corrente di esp. In questo caso la CPU estrae il Task Gate Descriptor contenuto nella entry 8. Esso punta allo speciale descrittore di segmento TSS memorizzato nella 32° entry della GDT. Quindi la CPU carica i registri eip ed esp con i valori contenuti nel TSS. Il risultato è che il processore esegue la funzione doublefault_fn() sul suo stack privato.

Nel trattare le azioni svolte da un tipico gestore di eccezioni, non verranno esaminati i codici dei segnali inviati ai processi, le eccezioni che capitano mentre il kernel è in modalità emulazione MS-DOS e le eccezioni di tipo “Debug”.

Salvataggio dei registri

Handler_name indica un generico gestore di eccezioni; esso inizia con le seguenti istruzioni assembly:

```
handler_name:
        pushl $0          /* only for some exceptions */
        pushl $do_handler_name
        jmp error_code
```

Se non è previsto che l'unità di controllo inserisca nello stack un codice di errore hardware, la funzione inizia con pushl $0 per creare uno spaziatore nello stack con valore nullo. Poi inserisce l'indirizzo della funzione C, il cui nome è formato dal nome del gestore con prefisso do_. Il frammento di codice etichettato error_code è uguale per tutti tranne che per “Device not available”, e compie le seguenti azioni:

1 - Salva nello stack i registri che possono essere usati dalle funzioni C.

2 - Esegue un'istruzione cld per azzerare il flag di direzione di eflags DF, assicurando così che l'auto-incremento dei registri esi ed edi sia impiegato con istruzioni string[1].

3 – Copia in edx il codice di errore hardware salvato sullo stack nella locazione esp + 36 e inserisce sullo stack al suo posto il valore -1. Questo valore viene usato per separare le eccezioni 0x80 dalle altre.

4 – Carica in edi l'indirizzo di do_handler_name() salvata sullo stack nella posizione esp + 32; scrive al suo posto il contenuto di es.

5 – Copia in eax la locazione corrente in cima allo stack in modalità del kernel; questo indirizzo identifica la cella di memoria che contiene il valore dell'ultimo registro salvato al punto 1.

6 – Carica lo user data Segment Selector nei registri ds ed es.

7 – Chiama la funzione C il cui indirizzo è ora memorizzato in edi. Questa funzione riceve i propri argomenti dai registri eax ed edx invece che dallo stack.

Inizio e fine del gestore di eccezioni

Le funzioni che implementano il gestore hanno nome do_ + nome del gestore. La maggior parte di esse chiama do_trap() per memorizzare il codice di errore e il vettore di eccezione nel descrittore di processo di current, poi invia a current un segnale:

```
current->thread.error_code = error_code;
current->thread.trap_no = vector;
```

1 una singola istruzione assembly, come rep; movsb che è capace di agire su un intero blocco di dati.

force_sig(sig_number, current);

Il processo si occupa del segnale appena termina l'esecuzione del gestore; tale operazione avviene o in modalità utente ad opera del gestore di segnali del processo, se esiste, o in modalità del kernel. In quest'ultimo caso normalmente il kernel uccide il processo. I segnali inviati sono stati elencati in precedenza.

Il gestore di eccezioni controlla sempre se l'eccezione è avvenuta in modalità del kernel o in modalità utente e, nel primo caso, se è dovuta ad un argomento non valido passato ad una chiamata di sistema. Se non lo è, è dovuta ad un bug del kernel. Per evitare che il comportamento anomalo del kernel possa corrompere i dati sull'hard disk, il gestore chiama die() che stampa a video il contenuto dei registri della CPU (questo dump è chiamato kernel oops) e arresta il processo corrente chiamando do_exit().

Quando il gestore termina, il codice esegue un jmp a ret_from_exception(); questa funzione viene descritta più avanti.

Gestione degli interrupt

Dato che la maggior parte delle eccezioni provoca l'invio di un segnale al processo corrente, la cui gestione è differita al momento in cui il processo lo riceve, la gestione delle eccezioni viene portata a termine in breve tempo.

Questo non vale per gli interrupt, in quanto spesso giungono molto dopo che il processo a cui sono destinati è stato sospeso (è il caso di una richiesta di trasferimento dati), e il processo corrente non ha alcuna relazione con essi. Per cui inviare un segnale al processo corrente non avrebbe alcun senso.

La gestione degli interrupt dipende dal tipo di interrupt. Si distinguono tre tipi principali:

I/O interrupt: un dispositivo di I/O richiede attenzione. Il gestore deve interrogare il dispositivo per decidere l'azione da compiere.

Timer interrupt: generati da un timer, o APIC locale o esterno. Il segnale informa il kernel che un certo intervallo di tempo è trascorso. Vengono gestiti per lo più come I/O interrupt.

Interprocessor interrupt: una CPU invia un segnale ad un'altra.

Gestione degli interrupt I/O

Un gestore di questo tipo deve essere abbastanza flessibile da servire diversi dispositivi. Ad esempio, nel bus PCI vari dispositivo condividono la stessa linea di IRQ; ciò significa che il valore del vettore non contiene tutte le informazioni necessarie. In alcune vecchie architetture, alcuni dispositivi (es. schede ISA) non potevano condividere le linee di IRQ.

La flessibilità dei gestori è ottenuta in due modi:

- *Condivisione di IRQ*: il gestore esegue varie *interrupt service routines* (ISR). Ogni ISR è una funzione correlata ad un dispositivo tra quelli che condividono la linea di IRQ. Poiché il dispositivo non è determinabile a priori, è compito della routine identificare chi richiede l'attenzione: fatto ciò, la ISR svolge le operazioni richieste dal particolare dispositivo.

- *Allocazione dinamica di IRQ*: una linea di IRQ è associata al driver di dispositivo all'ultimo momento possibile: ad esempio, la linea viene associata ad un floppy disk solo quando l'utente ne richiede

l'uso. In questo modo un vettore può essere usato per dispositivi diversi anche se essi non condividono la stessa linea. Chiaramente i dispositivi non possono essere impiegati contemporaneamente.

Non tutte le azioni da intraprendere in occasione di un interrupt hanno la stessa urgenza. Il gestore di per sé non è il luogo migliore per tutte le azioni. Lunghe operazioni non essenziali dovrebbero essere differite perché, mentre un gestore è in esecuzione, i segnali sulla linea IRQ corrispondente sono temporaneamente ignorati.

Inoltre, il processo sul quale si inserisce l'esecuzione del gestore deve rimanere in TASK_RUNNING per evitare uno stallo del sistema. Di conseguenza, i gestori di interrupt non possono compiere azioni bloccanti come operazioni di I/O su dico. Linux divide le azioni da compiere in seguito ad un interrupt in tre classi:

- *Critiche*: azioni come confermare al PIC l'interrupt, riprogrammare il PIC o il controllore del dispositivo oppure aggiornare le strutture a cui accedono sia il dispositivo che il processore, vanno eseguite rapidamente. Le azioni critiche sono eseguite dal gestore immediatamente, con gli interrupt mascherabili disabilitati.

- *Non critiche*: azioni come aggiornare le strutture usate solo dal processore (ad esempio leggere il codice dopo la pressione di un tasto) possono terminare in fretta e sono eseguite dal gestore immediatamente, con gli interrupt abilitati.

- *Non critiche differibili*: azioni come copiare il buffer nello spazio di memoria di un processo (ad esempio inviare il buffer di linea della tastiera al processo di destinazione) si possono rimandare per lungo tempo senza influenzare le operazioni del kernel; il processo interessato deve solo aspettare i dati. Sono eseguite da funzioni separate.

Riguardo al tipo di circuito che ha provocato l'interrupt, tutti i gestori compiono le seguenti operazioni fondamentali:

1 – Salvare il valore di IRQ e il contenuto dei registri nello stack in modalità del kernel.

2 – Inviare la conferma al PIC che sta servendo la linea IRQ, permettendogli di ricevere altri interrupt.

3 – Eseguire la ISR associata ai dispositivi.

4 – Terminare con un salto all'indirizzo di ret_from_intr().

Per descrivere lo stato della linea di IRQ e le funzioni da eseguire in caso di interrupt sono necessarie varie strutture. Di seguito vengono descritte le funzioni coinvolte.

Vettori di interrupt in Linux

intervallo di valori	uso
0 – 19 (0x0 – 0x13)	Interrupt non mascherabili ed eccezioni
20 – 31 (0x14 – 0x1f)	Riservato Intel
32 – 127 (0x20 – 0x7f)	Interrupt esterni (IRQ)
128 (0x80)	Eccezioni programmate per chiamate di sistema
129 – 238 (0x81 – 0xee)	Interrupt esterni (IRQ)

intervallo di valori	uso
239 (0xef)	Interrupt del timer locale APIC
240 (0xf0)	Interrupt del sensore termico locale APIC (introdotto nel Pentium 4)
241 – 250 (0xf1 – 0xfa)	Riservato per usi futuri da Linux
251 – 253 (0xfb – 0xfd)	Interrupt interprocessore
254 (0xfe)	Interrupt di errore APIC locale (quando APIC rileva un errore)
255 (0xff)	interrupt spurio di APIC locale (se la CPU maschera un interrupt)

Come descritto in tabella, agli IRQ fisici può essere assegnato un vettore nell'intervallo 32 – 238. Linux usa il vettore 128 per implementare le chiamate di sistema.

L'architettura PC IBM compatibile richiede che particolari dispositivi vengano associati staticamente a specifiche linee IRQ:

- Il dispositivo timer deve essere connesso alla linea 0.
- Il PIC 8259A slave deve essere connesso alla linea 2 (tuttora supportato da Linux).
- Il coprocessore matematico esterno deve essere connesso alla linea 13 (tuttora supportato da Linux).
- In generale un dispositivo di I/O può essere connesso a un limitato numero di linee. Come conseguenza, se si ha a che fare con un vecchio PC che non supporta la condivisione di IRQ, l'installazione di una scheda può non andare a buon fine se entra in conflitto con altri dispositivi per la linea di IRQ.

Ci sono tre modi di scegliere una linea per un dispositivo con IRQ configurabile:

- tramite ponticelli (jumpers), solo per le schede più vecchie;
- per mezzo di un programma di servizio fornito con il dispositivo, all'atto della sua installazione. Il programma può chiedere all'utente di indicare un numero di IRQ disponibile o interrogare il sistema per determinarlo da sé;
- con un protocollo hardware eseguito in fase di avvio. Le periferiche dichiarano quale linea sono pronti ad usare; il valore finale è negoziato per ridurre il più possibile i conflitti. Una volta fatto ciò, ogni gestore di interrupt può leggere l'IRQ assegnato usando una funzione che accede alle porte di I/O del dispositivo. Ad esempio i driver per i dispositivi PCI standard usano un gruppo di funzioni del tipo pci_read_config_byte() per accedere allo spazio di configurazione dei dispositivi.

Esempio di attribuzione di IRQ per un PC.

IRQ	INT	dispositivo
0	32	Timer
1	33	Tastiera
2	34	PIC a cascata

IRQ	INT	dispositivo
3	35	Seconda porta seriale
4	36	Prima porta seriale
6	38	Floppy disk
8	40	Orologio di sistema
10	42	Interfaccia di rete
11	43	Porta USB, scheda audio
12	44	Mouse PS/2
13	45	Coprocessore matematico
14	46	Canale IDE primario
15	47	Canale IDE secondario

Il kernel deve scoprire a quale dispositivo di I/O corrisponde il numero di IRQ prima di abilitare gli interrupt, in modo da essere in grado di gestirli. La corrispondenza è stabilita durante l'inizializzazione di ogni dispositivo.

Strutture dati degli IRQ

Quando vengono descritte operazioni complicate che comportano transizioni di stato, è di aiuto capire prima dove sono memorizzati i dati fondamentali. Vengono descritte le strutture alla base della gestione degli interrupt e come appaiono in vari descrittori.

Ogni vettore di interrupt ha il proprio descrittore irq_desc_t, i cui campi sono elencati in tabella. Tutti i descrittori di questo tipo sono raggruppati nell'array irq_desc.

Descrittore irq_desc_t:

Campo	Descrizione
handler	Punta all'oggetto PIC (descrittore hw_irq_controller) che serve la linea IRQ
handler_data	Puntatore ai dati usati dai metodi PIC
action	Identifica le ISR da chiamare quando arriva un IRQ. Il campo punta al primo elemento della lista di descrittori irqaction associati all'IRQ
status	Set di flag che descrivono lo stato della linea IRQ
depth	contiene 0 se la linea è abilitata e un valore positivo se è stata disabilitata almeno una volta
irq_count	Contatore del numero di interrupt generati (a fini diagnostici)
irqs_unhandled	Contatore del numero di interrupt non gestiti (a fini diagnostici)
lock	Spinlock usato per regolare gli accessi al descrittore di IRQ e al PIC

Un interrupt è inaspettato se non è gestito dal kernel, o perché non c'è una ISR associata alla linea di IRQ, o perché la ISR associata alla linea non riconosce l'interrupt come proveniente dal proprio dispositivo hardware. Di solito il kernel controlla il numero di interrupt inaspettati su ogni linea, in modo da disabilitare la linea in caso di malfunzionamento del dispositivo che ripete continuamente un interrupt. Poiché la linea

può essere condivisa da più dispositivi, il kernel non la disabilita al primo interrupt non gestito. Piuttosto memorizza in irq_count e irqs_unhandled il numero totale e il numero di interrupt inaspettati; quando viene raggiunto il valore 100.000 di irq_count, il kernel disabilita la linea se il numero di interrupt non gestiti è superiore a 99.900 (quindi se meno di 101 interrupt sugli ultimi 100.000 sono attesi).

I flag che descrivono lo stato di una linea di IRQ sono elencati in tabella.

Nome	Descrizione
IRQ_INPROGRESS	Un gestore per l'IRQ è stato eseguito
IRQ_DISABLED	La linea di IRQ è stata disabilitata volontariamente da un dispositivo
IRQ_PENDING	E' stata inviata una IRQ; confermata al PIC, ma l'interrupt non è ancora stato gestito dal kernel
IRQ_REPLAY	La linea IRQ è stata disabilitata ma l'IRQ precedente non è stata confermata al PIC
IRQ_AUTODETECT	Il kernel sta usando la linea di IRQ per testare il dispositivo
IRQ_WAITING	Il kernel usa la linea di IRQ per testare il dispositivo; l'interrupt corrispondente non è stato ricevuto
IRQ_LEVEL	Non usato nell'architettura 80x86
IRQ_MASKED	Non usato
IRQ_PER_CPU	Non usato nell'architettura 80x86

Il campo depth e il flag IRQ_DISABLED del descrittore irq_desc_t indicano se la linea è abilitata o no. Ogni volta che viene chiamata disable_irq() o disable_irq_nosync(), il campo depth è incrementato; se vale 0, la funzione disabilita la linea IRQ e attiva il flag IRQ_DISABLED (la differenza tra le due funzioni è che disable_irq() attende che tutti i gestori di interrupt attivi per quella linea su altre CPU abbiano terminato prima di ritornare). Al contrario, ogni chiamata di enable_irq() decrementa il campo; se depth diventa 0, la funzione abilita la linea di IRQ e azzera il flag IRQ_DISABLED.

Durante l'inizializzazione del sistema, la funzione init_IRQ() imposta il campo status di ogni descrittore a IRQ_DISABLED. Inoltre aggiorna l'IDT rimpiazzando gli interrupt gates impostati da setup_idt() con le istruzioni seguenti:

```
for(i = 0; i < NR_IRQS; i++)
        if(i + 32 != 128)
                set_intr_gate(i + 32, interrupt[i]);
```

Questo codice cerca nell'array interrupt l'indirizzo del gestore da usare per impostare l'interrupt gate. Ogni elemento n dell'array interrupt contiene l'indirizzo del gestore di interrupt per IRQn. Da notare che il gate corrispondente al vettore 128 non viene modificato, perché è utilizzato per le eccezioni programmate delle chiamate di sistema.

In aggiunta al chip 8259A, Linux supporta vari altri circuiti PIC come SMP IO-APIC, Intel PIIX4's internal 8259 PIC e SGI's Visual Workstation Cobalt (IO-)APIC. Per gestirli tutti in modo uniforme, Linux usa un *oggetto* PIC formato dal nome PIC e da sette metodi standard. Il vantaggio di questo approccio ad oggetti è che i driver non devono preoccuparsi del tipo di PIC installato sul sistema. Ogni sorgente di interrupt visibile ai driver è collegata in modo trasparente al controller appropriato. La struttura che definisce l'oggetto PIC è chiamata hw_interrupt_type (o anche hw_irq_controller).

Si consideri un computer uniprocessore con due PIC 8259A che forniscono 16 IRQ standard. Il campo handler di ognuno dei 16 descrittori irq_desc_t punta alla variabile i8259A_irq_type che descrive il chip PIC

8259A. Questa variabile è inizializzata come segue:

```
struct hw_interrupt_type  i8259A_irq_type  = {
        .typename       = "XT_PIC",
        .startup        = startup_8259A_irq,
        .shutdown       = shutdown_8259A_irq,
        .enable         = enable_8259A_irq,
        .disable        = disable_8259A_irq,
        .ack            = mask_and_ack_8259A,
        .end            = end_8259A_irq,
        .set_affinity   = NULL
};
```

Il primo campo è il nome del PIC. Seguono i puntatori a sei diverse funzioni usate per programmare il PIC. Le prime due avviano e chiudono una linea IRQ, ma nel caso del chip 8259A queste funzioni coincidono con la terza e la quarta che abilitano e disabilitano la linea. La mask_and_ack_8259A conferma il segnale ricevuto inviando i byte appropriati alla porta di I/O del chip. L'ultima funzione viene chiamata quando il gestore dell'interrupt per la linea termina. L'ultimo metodo è impostato a NULL: viene usato nei sistemi multiprocessore per dichiarare "l'affinità" della CPU per una specifica IRQ, cioè quali CPU sono abilitate a gestire specifiche IRQ.

Diversi dispositivi possono condividere una singola IRQ; perciò il kernel mantiene descrittori irqaction, ognuno dei quali si riferisce a un dispositivo hardware specifico e ad uno specifico interrupt. I campi di questo descrittore e i flag sono mostrati nelle tabelle seguenti.

Descrittore irqaction

Campo	Descrizione
handler	Punta alla ISR per un dispositivo di I/O. E' il campo chiave per la condivisione delle IRQ
flags	Descrivono le relazioni tra linea IRQ e il dispositivo di I/O
mask	Non usato
name	Nome del dispositivo di I/O (mostrato quando si elencano le IRQ dal file /proc/interrupt)
dev_id	Campo privato per il dispositivo; identifica il dispositivo stesso (può essere uguale ai numeri minor e major) oppure punta a dati del driver del dispositivo
next	Punta all'elemento successivo della lista dei descrittori irqaction. Gli elementi della lista si riferiscono a dispositivi che condividono la stessa IRQ
irq	Linea di IRQ
dir	Punta al descrittore della directory /proc/irq/n associata alla IRQn

Flag del descrittore irqaction

Flag	Descrizione
SA_INTERRUPT	Il gestore va eseguito con gli interrupt disabilitati
SA_SHIRQ	Il dispositivo consente la condivisione della linea IRQ

Flag	Descrizione
SA_SAMPLE_RANDOM	Il dispositivo può essere considerato una fonte di eventi casuali; può venire impiegato dal kernel come generatore di numeri casuali. Gli utenti usano questa fonte attraverso i file di dispositivo /dev/random e /dev/urandom

Infine l'array irq_stat include NR_CPUS entry, una per ogni CPU del sistema. Ogni entry, di tipo irq_cpustat_t comprende alcuni puntatori e flag usati dal kernel per tenere traccia di quanto sta facendo ogni CPU.

Campi della struttura irq_cpustat_t.

Campo	Descrizione
__softirq_pending	Insieme di flag che identificano softirq pendenti
idle_timestramp	Tempo nel quale la CPU diviene inattiva (ha significato solo se la CPU è inattiva)
__nmi_count	Numero di occorrenze di interrupt NMI
apic_timer_irqs	Numero di occorrenze di interrupt di timer APIC

Distribuzione degli IRQ nei sistemi multiprocessori

Linux supporta il modello multiprocessore simmetrico (SMP); ciò significa che non deve privilegiare una CPU rispetto alle altre. Di conseguenza il kernel tenta di distribuire i segnali di IRQ che provengono dai dispositivi hardware in round-robin, e le varie CPU dovrebbero trascorrere lo stesso tempo nel gestire gli interrupt di I/O. Si è già detto che il sistema multi-APIC ha meccanismi sofisticati per distribuire dinamicamente le IRQ tra le CPU.

Durante l'avvio, la CPU che fa il boot, esegue la funzione setup_IO_APIC_irqs() per inizializzare il chip I/O APIC. Le 24 entry della Tabella di Ridirezione degli Interrupt del chip sono riempite, in modo che i segnali di IRQ vengono distribuite tra le CPU in base allo schema a “priorità inferiore”. In più, durante l'avvio, tutte le CPU eseguono la funzione setup_local_APIC() che si incarica di inizializzare gli APIC locali. In particolare il Task Priority Register (TPR) di ogni chip è impostato ad un valore fisso, a significare che ogni CPU ha intenzione di gestire ogni tipo di segnale, a prescindere dalla sua priorità. Il kernel non modifica più questo valore.

Se i TPR contengono lo stesso valore, tutte le CPU hanno la stessa priorità. Per operare una scelta, il sistema multi-APIC usa i valori nei registri di priorità dell'arbitraggio degli APIC locali, come spiegato in precedenza. Poiché tali valori sono cambiati automaticamente dopo ogni interrupt, i segnali IRQ sono distribuiti in maniera uniforme tra le CPU.

In breve, quando un dispositivo invia un segnale IRQ, il sistema multi-APIC sceglie una delle CPU ed invia il segnale al suo APIC locale, che a sua volta lo invia alla propria CPU. Nessun altra CPU è coinvolta.

Tutto il lavoro è svolto dall'hardware, senza interessare il kernel. Sfortunatamente a volte l'hardware non riesce a distribuire i segnali in modo equilibrato. Linux 2.6 fa uso di uno speciale thread, chiamato kirqd per correggere il sistema in caso di bisogno.

Il thread sfrutta una funzione dei sistemi multi-APIC chiamata affinità della CPU: modificando le entry della Tabella di Ridirezione degli interrupt è possibile instradare un segnale verso una particolare CPU. Lo si può

ottenere chiamando la funzione set_ioapic_affinity_irq(), che opera su due parametri: il vettore IRQ da ridirigere e una maschera a 32 bit che indica la CPU che può ricevere il segnale. L'affinità di un IRQ può essere cambiata anche dall'amministratore di sistema impostando una nuova maschera bitmap delle CPU nel file /proc/irq/n/smp_affinity, dove n è il vettore di interrupt.

kirqd esegue periodicamente la funzione do_irq_balance(), che tiene traccia del numero di IRQ ricevuti dalle singole CPU in un intervallo di tempo. Se trova uno squilibrio significativo tra la CPU con carico maggiore e quella con carico minore, seleziona un IRQ da spostare da una CPU all'altra, oppure ruota tutti gli IRQ tra le CPU.

Stack multipli in modalità del kernel

Il descrittore thread_info di ogni processo è abbinato allo stack in modalità del kernel nella struttura thread_union, formata da una o due frame di pagina, in base ad una opzione impostata in fase di compilazione del kernel. Se la dimensione è di 8 KB, lo stack è usato per ogni tipo di KCP: eccezioni, interrupt e funzioni differibili. Se invece la dimensione è di 4 KB, il kernel usa tre tipi di stack in modalità del kernel:

- *exception stack*, usato per gestire le eccezioni, incluse le chiamate di sistema. E' lo stack contenuto in thread_union, per cui il kernel usa un diverso exception stack per ogni processo del sistema;
- *hard IRQ stack*, usato per gestire gli interrupt. Ne esiste uno solo per ogni CPU del sistema, ed è contenuto in un unico frame di pagina;
- *soft IRQ stack*, usato per le funzioni differibili (softirq e tasklet). Ne esiste uno solo per ogni CPU del sistema, ed è contenuto in un unico frame di pagina.

Tutti gli hard IRQ stack sono contenuti nell'array hardirq_stack, mentre tutti i soft IRQ stack sono contenuti nell'array softirq_stack. Ogni elemento di questi array è una union di tipo irq_ctx che occupa una singola pagina. In fondo alla pagina è contenuta una struttura thread_info, mentre le locazioni di memoria libere sono usate per lo stack, ricordando che lo stack cresce verso indirizzi inferiori. Questi stack sono molto simili a quello per le eccezioni: la differenza è che la struttura thread_info che contengono non è abbinata ad un processo ma ad una CPU.

Gli array hardirq_ctx e softirq_ctx aiutano il kernel a trovare il relativo stack per ogni CPU: essi contengono puntatori agli elementi irq_ctx corrispondenti.

Salvataggio dei registri per il gestore di interrupt

Quando una CPU riceve un interrupt inizia ad eseguire il codice all'indirizzo trovato nel gate della IDT. Come per altre commutazioni di contesto, la necessità di salvare i registri lascia il programmatore con un compito piuttosto confuso, in quanto i registri vanno manipolati con codice assembly. Comunque, all'interno di queste operazioni, il processore deve chiamare ed uscire da una funzione C.

Il salvataggio dei registri è la prima operazione del gestore di interrupt L'indirizzo del gestore per la IRQ numero n è contenuto inizialmente nella entry interrupt[n], poi viene copiato nel gate della IDT.

L'array interrupt è assemblato con poche istruzioni assembly nel file *arch/i386/kernel/entry.S*. L'array comprende NR_IRQS elementi, dove la macro NR_IRQS vale 224 se il kernel supporta un chip I/O APIC

recente, oppure 16 se usa il vecchio chip 8259A. L'elemento a indice n contiene l'indirizzo delle due istruzioni assembly:

```
pushl $n-256
jmp common_interrupt
```

Viene salvato nello stack il numero di IRQ associato all'interrupt meno 256. Il kernel rappresenta gli IRQ con numeri negativi, perché riserva valori positivi alle chiamate di sistema. Si può eseguire lo stesso codice per tutti i gestori facendo riferimento a questo numero. Il codice comune inizia all'etichetta common_interrupt:

```
common_interrupt:
        SAVE_ALL
        movl %esp, %eax
        call do_IRQ
        jmp ret_from_intr
```

La macro SAVE_ALL si espande nel codice seguente:

```
cld
push %es
push %ds
pushl %eax
pushl %ebp
pushl %edi
pushl %esi
pushl %edx
pushl %ecx
pushl %ebx
movl $__USER_DS, %edx
movl %edx, %ds
movl %edx, %es
```

SAVE_ALL salva tutti i registri che possono essere usati dal gestore, eccetto eflags, cs, eip, ss e esp, che vengono salvati automaticamente dall'unità di controllo. Infine carica il selettore di segmento __USER_DS nei registri es e ds.

Dopo aver salvato i registri, l'indirizzo della cima dello stack viene copiato in eax; poi il gestore di interrupt chiama do_IRQ(). Quando viene eseguita l'istruzione ret della funzione, il controllo passa a ret_from_intr().

La funzione do_IRQ()

La funzione viene chiamata per eseguire le ISR associate ad un interrupt .Viene dichiarata in questo modo:

```
__attribute__((regparm(3))) unsigned int do_IRQ(struct pt_regs *regs)
```

La parola chiave regparm indica alla funzione di cercare nel registro eax il valore dell'argomento regs; come visto in precedenza, eax punta alla locazione dello stack che contiene l'ultimo valore di registro salvato da SAVE_ALL. La do_IRQ esegue le seguenti operazioni:

1 – Esegue la macro irq_enter() che incrementa il contatore del numero di gestori di interrupt nidificati. Il contatore è contenuto nel campo preempt_count della struttura thread_info del processo corrente.

2 – Se la dimensione di thread_union è 4 KB, passa allo hard IRQ stack. In particolare:

a – Esegue la funzione current_thread_info() per ottenere l'indirizzo del descrittore thread_info associato allo stack in modalità del kernel indirizzato da esp

b – Confronta l'indirizzo così ottenuto con quello in hardirq_ctx[smp_processor_id()], cioè l'indirizzo della struttura thread_info associata alla CPU locale. Se sono uguali, cioè il kernel sta già usando lo stack giusto, passa al punto 3. Questo accade quando un IRQ viene ricevuto mentre il kernel sta gestendo un altro interrupt.

c – Si deve procedere al cambio dello stack. Salva il puntatore al PD corrente nel campo task del descrittore thread_info nella union irq_ctx della CPU locale. Questo permette alla macro current di lavorare correttamente mentre il kernel usa l'hard IRQ stack.

d – Memorizza il valore corrente di esp nel campo previous_esp del descrittore thread_info nella union irq_ctx (questo campo è usato solo per preparare la chiamata di sistema trace per un kernel oops).

e – Carica in esp la locazione della cima dell'hard IRQ stack (valore hardirq_ctx[smp_processor_id()] + 4096; il valore precedente di esp viene salvato in ebx)

3 – Chiama la __do_IRQ() passandole il puntatore regs e il numero di IRQ ottenuto da regs->orig_eax.

4 - Se è avvenuta la commutazione di stack del punto 2a, la funzione copia il puntatore allo stack originale dal registro ebx a esp, ritornando così allo stack utilizzato all'inizio.

5 – Esegue la macro irq_exit(), che decrementa il contatore degli interrupt e controlla se funzioni differibili del kernel sono in attesa di essere eseguite.

6 – Termina: il controllo viene trasferito alla funzione ret_from_intr().

La funzione __do_IRQ()

La __do_IRQ() riceve come parametri un numero di IRQ (tramite il registro eax) e un puntatore alla struttura pt_regs dove sono stati salvati i valori dei registri in modalità utente (tramite il registro edx).

```
irq_desc_t *desc = irq_desc + irq;
struct irqaction * action;
unsigned int status;
. . .
spin_lock(&desc->lock);
desc->handler->ack(irq);
status = desc->status & ~(IRQ_REPLAY | IRQ_WAITING);
status |= IRQ_PENDING;
action = NULL;
if (likely(!(status & (IRQ_DISABLED | IRQ_INPROGRESS)))) {
        action = desc->action;
        status &= ~IRQ_PENDING;
        status |= IRQ_INPROGRESS;
}
desc->status = status;
if (unlikely(!action))
        goto out;
```

```
	for (;;) {
		irqreturn_t action_ret;
		spin_unlock(&desc->lock);
		action_ret = handle_IRQ_event(irq, regs, action);
		spin_lock(&desc->lock);
		if (!noirqdebug)
			note_interrupt(irq, desc, action_ret);
		if (likely(!(desc->status & IRQ_PENDING)))
			break;
		desc->status &= ~IRQ_PENDING;
	}
	desc->status &= ~IRQ_INPROGRESS;
out:
	desc->handler->end(irq);
	spin_unlock(&desc->lock);
	return 1;
```

Prima di accedere al descrittore di IRQ, il kernel acquisisce lo spinlock corrispondente, per impedire accessi concorrenti di altre CPU. Questo meccanismo è necessario in sistemi multiprocessore perché altri interrupt dello stesso tipo potrebbero essere inviati e altre CPU potrebbero occuparsene. Senza spinlock il descrittore potrebbe essere utilizzato da diverse CPU in modo concorrente, situazione che va assolutamente evitata.

Poi la funzione chiama il metodo ack del descrittore. Quando è in uso il vecchio PIC 8259A, la funzione corrispondente mask_and_ack_8259A() conferma l'interrupt al PIC e disabilita la linea IRQ. Questo assicura che la CPU non accetti ulteriori interrupt di questo tipo fino al termine dell'attività del gestore. La __do_IRQ() viene eseguita con gli interrupt locali disabilitati; infatti l'unità di controllo della CPU azzera il flag IF di eflags poiché il gestore è chiamato per mezzo di un interrupt gate della IDT. Tuttavia il kernel deve abilitare di nuovo gli interrupt locali prima di eseguire la ISR dell'interrupt corrente.

Quando si usa un I/O APIC, le cose sono molto più complesse. A seconda del tipo, l'accettazione dell' interrupt può essere fatta col metodo ack o può venir rinviata al termine dell'attività del gestore (ossia col metodo end). In ogni caso si può dare per certo che l'APIC locale non accetti interrupt dello stesso tipo, anche se altre CPU possono farlo.

La __do_IRQ() inizializza alcuni flag del descrittore. Attiva IRQ_PENDING perché l'interrupt è stato confermato ma non ancora gestito; azzera IRQ_WAITING e IRQ_REPLAY.

Ora la funzione controlla se può veramente gestire l'interrupt Ci sono tre casi in cui non è necessario fare nulla:

IRQ_DISABLED è attivato: una CPU potrebbe eseguire __do_IRQ() anche se la linea IRQ corrispondente fosse disabilitata; la spiegazione di questo caso non intuitivo si trova più avanti. In più, alcune schede madri con bug possono generare interrupt spuri anche con la linea IRQ disabilitata.

IRQ_INPROGRESS è attivato: in un sistema multiprocessore, un'altra CPU può stare gestendo un interrupt precedente dello stesso tipo. Linux rimanda la gestione di questo interrupt a quella CPU. Questo porta ad una semplificazione dell'architettura, perché le ISR dei driver di dispositivo non devono essere rientranti (la loro esecuzione è in successione). Inoltre la CPU liberata dalla gestione può tornare a l proprio lavoro senza sporcare la cache hardware: ciò è positivo per le prestazioni del sistema. Il flag IRQ_INPROGRESS è attivato ogni volta che alla CPU è affidata l'esecuzione di una ISR; perciò la __do_IRQ() controlla il flag prima di iniziare.

irq_desc[irq].action = NULL: non esiste una ISR associata all'interrupt. Di solito è il caso in cui il kernel sta

testando un componente hardware.

Se nessuno di questi tre casi si verifica, l'interrupt va gestito. La funzione attiva IRQ_INPROGRESS e inizia un ciclo. Ad ogni iterazione, azzera IRQ_PENDING, rilascia lo spinlock ed esegue la ISR chiamando handle_IRQ_event(). Quando questa termina, acquisisce di nuovo lo spinlock e controlla il valore di IRQ_PENDING. Se è zero, nessun altro interrupt dello stesso tipo è stato inviato a un'altra CPU, e il ciclo termina. Se il flag è attivato, un'altra CPU ha eseguito la do_IRQ() per lo stesso tipo di interrupt mentre veniva eseguita handle_IRQ_event(). Perciò do_IRQ() compie un'altra iterazione del ciclo, gestendo il nuovo interrupt (dato che IRQ_PENDING è un flag e non un contatore, tiene conto solo della seconda riproposizione dell'interrupt. Se l'interrupt si ripete una terza volta, va semplicemente perso).

__do_IRQ() sta terminando, sia che abbia eseguito l'ISR, sia che non abbia fatto nulla. Chiama il metodo end del descrittore di IRQ. Se è in uso il vecchio PIC 8259A, la funzione corrispondente end_8259A_irq() abilita la linea IRQ (a meno che non si trattasse di interrupt spurio). Se è in uso un I/O APIC, il metodo end conferma l'interrupt (se non lo ha fatto il metodo ack).

Infine __do_IRQ() rilascia lo spinlock.

Recupero di un interrupt perso

La funzione __do_IRQ()è breve e semplice ed opera efficacemente nella maggior parte dei casi. I flag IRQ_PENDING, IRQ_INPROGRESS e IRQ_DISABLED garantiscono che gli interrupt vengano gestiti correttamente anche se l'hardware presenta malfunzionamenti. Però nei sistemi multiprocessore le cose non sempre sono così semplici.

Se la CPUx ha una linea IRQ abilitata, un dispositivo hardware può attivare la linea e il sistema multi-APIC può scegliere la CPUx per gestire l'interrupt. Prima che questa confermi l'interrupt, la linea viene mascherata da un'altra CPU: di conseguenza il flag IRQ_DISABLED è attivato. Subito dopo la CPUx inizia la gestione dell'interrupt in sospeso; perciò la do_IRQ() conferma l'interrupt e ritorna senza aver eseguito la ISR perché trova il flag IRQ_DISABLED attivato. Perciò, anche se l'interrupt è avvenuto prima che la linea fosse disabilitata, va perduto.

Per fronteggiare questi casi, la funzione enable_irq(), usata dal kernel per abilitare la linea, controlla prima se un interrupt è andato perso. Se è così, costringe l'hardware a ripetere l'interrupt perduto:

```
irq_desc_t *desc = irq_desc + irq;
unsigned long flags;
spin_lock_irqsave(&desc->lock, flags);
switch (desc->depth) {
case 0:
        WARN_ON(1);
        break;
case 1: {
        unsigned int status = desc->status & ~IRQ_DISABLED;
        desc->status = status;
        if ((status & (IRQ_PENDING | IRQ_REPLAY)) == IRQ_PENDING) {
                desc->status = status | IRQ_REPLAY;
                hw_resend_irq(desc->handler,irq);
        }
        desc->handler->enable(irq);
        /* fall-through */
}
```

```
default:
        desc->depth--;
}
spin_unlock_irqrestore(&desc->lock, flags);
```

La funzione riconosce che un interrupt è andato perduto controllando il flag IRQ_PENDING. Il flag è sempre azzerato quando il gestore di interrupt termina; perciò se la linea è disabilitata e il flag è attivato, un interrupt è stato confermato ma non gestito. In questo caso hw_resend_irq() genera un nuovo interrupt forzando l'APIC locale a produrre un self-interrupt. Il ruolo di IRQ_REPLAY è di garantire che venga prodotto un solo self-interrupt. Da ricordare che __do_IRQ() azzera questo flag quando inizia a gestire l'interrupt.

Interrupt Service Routine

Una ISR gestisce un interrupt eseguendo una operazione specifica per un tipo di dispositivo. Per eseguire una ISR il gestore chiama la funzione handle_IRQ_event() che esegue le seguenti azioni:

1 – Abilita gli interrupt locali con l'istruzione assembly sti se il flag SA_INTERRUPT è azzerato.

2 – Esegue ogni ISR dell'interrupt con il codice:

```
int ret, retval = 0, status = 0;
if (!(action->flags & SA_INTERRUPT))
        local_irq_enable();
do {
        ret = action->handler(irq, action->dev_id, regs);
        if (ret == IRQ_HANDLED)
                status |= action->flags;
        retval |= ret;
        action = action->next;
} while (action);
if (status & SA_SAMPLE_RANDOM)
        add_interrupt_randomness(irq);
local_irq_disable();
return retval;
```

All'inizio del ciclo, action punta all'inizio di una lista di strutture irqaction che indicano le azioni da compiere alla ricezione dell'interrupt.

3 – Disabilita gli interrupt locali con l'istruzione assembly cli.

4 – Termina restituendo come valore di retval 0, se nessuna ISR ha riconosciuto l'interrupt, 1 nel caso contrario.

Tutte le ISR ricevono gli stessi parametri, passati ancora una volta per mezzo dei registri eax, edx, ecx rispettivamente:

irq: numero di IRQ

dev_id: identificatore del dispositivo

regs: puntatore alla struttura pt_regs nello stack in modalità del kernel che contiene i registri salvati subito dopo l'arrivo dell'interrupt; pt_regs è formata da 15 campi:

- i primi 9 sono i registri inseriti nello stack da SAVE_ALL
- il decimo, referenziato da un campo chiamato orig_eax, codifica il numero di IRQ
- i rimanenti corrispondono ai registri salvati in automatico dall'unità di controllo.

Il primo parametro permette ad una sola ISR di gestire diverse linee IRQ, il secondo permette a una sola ISR di occuparsi di diversi dispositivi dello stesso tipo, e l'ultimo permette alla ISR di accedere al contesto di esecuzione del KCP interrotto. In pratica, la maggior parte delle ISR non usano questi parametri.

Ogni ISR restituisce il valore 1 se l'interrupt è stato effettivamente gestito, cioè se il segnale è stato inviato dal dispositivo gestito dalla ISR; altrimenti restituisce 0. Questo valore di ritorno consente al kernel di aggiornare il contatore degli interrupt inaspettati.

Il flag SA_INTERRUPT del descrittore IRQ determina se gli interrupt devono essere abilitati o no quando do_IRQ() chiama una ISR. Una ISR chiamata con gli interrupt in uno stato, può cambiarli nello stato opposto. In un sistema uniprocessore può essere fatto con le istruzioni cli (disabilita) e sti (abilita).

La struttura di una ISR dipende dalle caratteristiche del dispositivo.

Allocazione dinamica delle linee IRQ

Alcuni vettori sono riservati a specifici dispositivi, mentre i rimanenti sono gestiti dinamicamente. C'è tuttavia un modo in cui la stessa linea IRQ può essere usata da diversi dispositivi hardware anche se non ammettono la condivisione delle IRQ.

Prima di attivare un dispositivo che deve usare una linea di IRQ, il suo driver chiama request_irq(). Questa funzione crea un nuovo descrittore irqaction e lo inizializza con i valori dei parametri; poi chiama setup_irq() per inserire il descrittore nella lista IRQ appropriata. Il driver interrompe l'operazione se setup_irq() restituisce un codice di errore, che di solito significa che la linea IRQ è già usata da un altro dispositivo che non ammette la condivisione degli interrupt. Quando l'operazione è conclusa, il driver chiama free_irq() per rimuovere il descrittore dalla lista e rilasciare l'area di memoria.

Per vedere come funziona il sistema, si consideri l'esempio di un programma che vuole indirizzare il file di dispositivo /dev/fd0, che corrisponde al primo floppy disk; i floppy sono vecchi dispositivi che non ammettono la condivisione degli IRQ.

Il programma può farlo sia accedendo direttamente a /dev/fd0 oppure montando un filesystem su di esso. Ai controller dei floppy viene assegnato l'IRQ 6; il driver può fare la richiesta seguente:

```
request_irq(6, floppy_interrupt, SA_INTERRUPT | SA_SAMPLE_RANDOM,
                "floppy", NULL);
```

Come si può osservare, l'ISR floppy_interrupt() deve essere eseguita con gli interrupt disabilitati (SA_INTERRUPT attivato) e senza condivisione di IRQ (SA_SHIRQ mancante). Il flag SA_SAMPLE_RANDOM impostato indica che il floppy è una buona fonte di eventi casuali da usare per il generatore di numeri casuali del kernel. Quando l'operazione sul floppy è conclusa (o l'operazione di I/O su /dev/fd0 termina, o il filesystem è smontato), il driver rilascia l'IRQ 6:

```
free_irq(6, NULL);
```

Per inserire un descrittore irqaction nella lista appropriata, il kernel chiama la funzione setup_irq(), passandole il parametro irq_nr, il numero di IRQ, e new (l'indirizzo di un descrittore irqaction allocato in

precedenza). La funzione:

1 – Controlla se un altro dispositivo sta già usando l'IRQ irq_nr e, se è così, se il flag SA_SHIRQ nel descrittore irqaction di entrambi i dispositivi specifica se la linea IRQ può essere condivisa. Restituisce un codice di errore se la linea IRQ non può essere usata.

2 – Aggiunge *new (il nuovo descrittore irqaction puntato da new) alla fine della lista a cui punta irq_desc[irq_nr]->action.

3 – Se nessun altro dispositivo condivide la stessa IRQ, la funzione azzera IRQ_DISABLED, IRQ_AUTODETECT, IRQ_WAITING e IRQ_INPROGRESS nel campo flags di *new e chiama il metodo startup dell'oggetto PIC irq_desc[irq_nr]->handler per assicurarsi che i segnali IRQ siano abilitati.

Segue un esempio di come viene usata setup_irq(), fin dall'inizializzazione del sistema. Il kernel inizializza il descrittore irq0 del timer di sistema eseguendo le seguenti istruzioni della funzione time_init():

```
struct irqaction irq0 = {timer_interrupt, SA_INTERRUPT, 0, “timer”, NULL, NULL};
setup_irq(0, &irq0);
```

Viene inizializzata la variabile irq0 di tipo irqaction: al campo handler viene assegnato l'indirizzo della funzione timer_interrupt(), il campo flags è impostato a SA_INTERRUPT, il nome è impostato a “timer” e il quinto campo a NULL, per indicare che non viene usato nessun dev_id. Poi il kernel chiama setup_irq() per inserire irq0 nella lista dei descrittori irqaction associati a IRQ0.

Gestione degli interrupt interprocessore

Questi segnali permettono ad una CPU di inviare interrupt ad ogni altra CPU del sistema. Un interrupt interprocessore non è inviato attraverso una linea IRQ ma direttamente come messaggio sul bus che connette gli APIC locali di tutte le CPU (un bus dedicato nei modelli più vecchi, il bus di sistema dal Pentium 4 in avanti).

Linux usa tre tipi di questi interrupt:

CALL_FUNCTION_VECTOR (vettore 0xfb): inviato a tutte tranne che alla CPU mittente, le forza ad eseguire una funzione passata da quest'ultima. Il gestore corrispondente è chiamato call_function_interrupt(). La funzione, il cui indirizzo è indicato nella variabile globale call_data, può, ad esempio, costringere le altre CPU ad arrestarsi oppure ad impostare il contenuto dei Memory Type Range Registers (registri addizionali destinati a facilitare le operazioni sulla cache). Di solito questo interrupt è inviato a tutte le CPU tranne a quella che sta eseguendo la funzione chiamante per mezzo della smp_call_function().

RESCHEDULE_VECTOR (vettore 0xfc): il gestore di questo interrupt, chiamato reschedule_interrupt(), si limita a confermare il segnale. La schedulazione avviene automaticamente al ritorno dall'interrupt.

INVALIDATE_TLB_VECTOR (vettore 0xfd): forza le CPU, eccetto quella mittente, a invalidare i propri TLB. Il gestore corrispondente è chiamato invalidate_interrupt().

Il codice assembly dei gestori di interrupt interprocessore è generato dalla macro BUILD_INTERRUPT: salva i registri, inserisce nello stack il numero del vettore meno 256 e chiama una funzione C che ha il nome del gestore di basso livello preceduto da smp_ (nel caso di CALL_FUNCTION_VECTOR, il gestore a basso livello call_function_interrupt() chiama la funzione smp_call_function_interrupt()). Ogni gestore di alto livello conferma l'interrupt all'APIC locale e poi svolge il proprio compito specifico.

Le funzioni elencate facilitano l'invio di interrupt interprocessore:

send_IPI_all(): invia il messaggio a tutte le CPU, anche a quella mittente

send_IPI_allbutself(): non lo invia al mittente

send_IPI_self(): invia un messaggio solo al mittente

send_IPI_mask(): invia il messaggio ad un gruppo di CPU indicate da una maschera di bit.

Softirq e tasklet

Alcune operazioni eseguite dal kernel non sono critiche: possono essere rimandate per un lungo periodo, se necessario. Da ricordare che le ISR legate ad un interrupt sono eseguite in serie e spesso non interviene un altro interrupt prima che il gestore abbia terminato il suo lavoro. Al contrario, le operazioni differibili si possono eseguire con gli interrupt abilitati; svolgerle al di fuori del gestore di interrupt rende più rapida la risposta del kernel. Questa è una caratteristica importante per molte applicazioni che si aspettano una risposta agli interrupt in pochi millisecondi.

Linux usa due tipi di funzioni del kernel non urgenti e soggette ad interruzione: le cosiddette *funzioni differibili* (*softirq e tasklet*) e quelle eseguite per mezzo di *work queue*.

Softirq e tasklet sono strettamente correlati, poiché i tasklet sono implementati sopra ai softirq. Difatti il termine "softirq" che compare nel codice sorgente del kernel, spesso indica entrambi i tipi di funzione. Un altro termine usato è *interrupt context*: indica che il kernel sta eseguendo un gestore di interrupt o una funzione differibile.

I softirq sono allocati staticamente, cioè definiti al momento della compilazione, mentre i tasklet possono essere allocati e inizializzati a runtime, ad esempio quando viene caricato un modulo del kernel. I softirq possono essere eseguiti in modo concorrente su diverse CPU, anche se sono dello stesso tipo. Perciò i softirq sono funzioni rientranti [2] e devono proteggere le proprie strutture con spinlock. I tasklet non devono preoccuparsi di questo, perché la loro esecuzione è controllata attentamente dal kernel. I tasklet dello stesso tipo sono sempre eseguiti in serie: in altre parole, lo stesso tipo di tasklet non può essere eseguito da due CPU nello stesso tempo. Comunque tasklet di tipo diverso possono essere eseguiti in modo concorrente da varie CPU. L'esecuzione in serie semplifica la vita di chi scrive driver di dispositivo, perché i tasklet non devono essere rientranti.

Sulle funzioni differibili possono essere eseguite 4 azioni:

Inizializzazione: definisce una nuova funzione differibile; di solito avviene quando il kernel inizializza sé stesso o carica un modulo.

Attivazione: identifica un funzione differibile come "in attesa" di essere eseguita quando il kernel schedulerà una nuova esecuzione di funzioni differibili. Può avvenire in ogni momento, anche durante un interrupt.

Mascheramento: disabilita selettivamente una funzione differibile in modo che non venga eseguita dal kernel anche se attivata. Talvolta disabilitare le funzioni è essenziale.

---

2 - una funzione è rientrante se una singola copia del codice in memoria può essere condivisa ed eseguita da utenti multipli e processi separati. Per esserlo, deve rispettare 4 condizioni:
a – nessuna porzione di codice può essere alterata durante l'esecuzione;
b – il codice non deve richiamare nessuna routine che non sia rientrante;
c – il codice deve usare solo variabili temporanee allocate sullo stack;
d – il codice non deve alterare variabili globali o aree di memoria condivisa né usare variabili locali statiche. ndt

Esecuzione: esegue una funzione differibile “in attesa” insieme con altre dello stesso tipo nello stesso stato; l'esecuzione avviene in un istante specifico.

Attivazione ed esecuzione sono collegate: una funzione attivata da una CPU deve essere eseguita da questa. Non c'è una ragione di per sé evidente che questa regola porti beneficio alle prestazioni del sistema. Legare la funzione alla CPU che l'ha attivata potrebbe in teoria portare ad un miglior uso della cache hardware. Dopo tutto è intuitivo che il thread del kernel che realizza l'attivazione faccia uso di strutture usate anche dalla funzione differibile. D'altra parte le linee di cache potrebbero non essere più presenti, dato che la sua esecuzione può essere ritardata per lungo tempo. Addirittura il legame con la CPU potrebbe essere dannoso, perché una CPU verrebbe sovraccaricata, mentre le altre potrebbero essere sottoutilizzate.

Softirq

Linux usa un numero limitato di softirq. Per alcuni scopi i tasklet sono altrettanto validi e più facili da scrivere perché non rientranti.

Per questo sono definiti solo 6 tipi di softirq:

Softirq	Indice (priorità)	Descrizione
HI_SOFTIRQ	0	Gestisce tasklet ad alta priorità
TIMER_SOFTIRQ	1	Tasklet collegati agli interrupt del timer
NET_TX_SOFTIRQ	2	Trasmette pacchetti alla scheda di rete
NET_RX_SOFTIRQ	3	Riceve pacchetti dalla scheda di rete
SCSI_SOFTIRQ	4	Elaborazione di comandi SCSI post-interrupt
TASKLET_SOFTIRQ	5	Gestisce tasklet regolari

L'indice del softirq. definisce la sua priorità: un indice inferiore significa priorità superiore, perché i softirq vengono eseguiti a partire dall'indice 0.

Strutture dati usate per i softirq

La struttura principale è il vettore softirq_vec, che include 32 elementi di tipo softirq_action. La priorità di un softirq è l'indice del corrispondente elemento softirq_action nell'array; solo i primi 6 elementi dell'array sono effettivamente usati. La struttura softirq_action è formata da due campi: un puntatore alla funzione softirq chiamato action e un puntatore a una struttura dati generica che può essere usata dalla funzione, chiamato data.

Un altro campo fondamentale, usato per tenere traccia della preemption del kernel e dell'annidamento dei KCP è il campo di 32 bit preempt_count, contenuto nel campo thread_info di ogni PD. Questo campo contiene tre distinti contatori e un flag:

Bit	Descrizione
0 – 7	Contatore di preemption (valore massimo = 255)
8 – 15	Contatore di softirq (valore massimo = 255)

Bit	Descrizione
16 – 27	Contatore di hardirq (valore massimo = 4096)
28	flag PREEMPT_ACTIVE

Il primo contatore tiene traccia di quante volte la preemption del kernel è stata disabilitata esplicitamente sulla CPU locale: il valore 0 significa che la preemption non è mai stata disabilitata. Il secondo contatore indica quanti livelli di profondità ha raggiunto la disabilitazione delle funzioni differibili (0 significa che le funzioni sono abilitate). Il terzo specifica il numero di gestori di interrupt annidati nella CPU locale (il valore è aumentato da irq_enter() e diminuito da irq_exit().

C'è una buona ragione che giustifica il nome del campo preempt_count: la preemption del kernel deve essere disabilitata sia quando questo viene richiesto esplicitamente dal codice del kernel (contatore di preemption diverso da 0) sia quando il kernel è in fase di elaborazione in un contesto di interrupt. Perciò, per determinare se il processo corrente può essere “preempted” (rilasciato in anticipo), il kernel controlla se il valore di preempt_count è zero.

La macro in_interrupt() controlla i contatori hardirq e softirq; se uno di essi è positivo, la macro restituisce un valore diverso da zero, altrimenti restituisce zero. Se il kernel non fa uso di diversi stack in modalità del kernel, la macro controlla il campo preempt_field del processo corrente; se invece fa uso di stack multipli, la macro controlla il campo del descrittore thread_info contenuto nella union irq_ctx associata alla CPU corrente. In questo caso la macro restituisce sempre un valore diverso da zero perché il campo è sempre impostato ad un valore positivo.

L'ultima struttura fondamentale è una maschera di 32 bit specifica per ogni CPU che descrive i softirq “in attesa”; è contenuta nel campo __softirq_pending della struttura irq_cpustat_t (una per ogni CPU del sistema). Per leggere e impostare il valore della maschera, il kernel usa la macro local_softirq_pending().

Gestione dei softirq

La funzione open_softirq() si incarica dell'inizializzazione. Usa tre parametri: l'indice del softirq, un puntatore alla funzione softirq da eseguire e un altro puntatore ad una struttura che può essere richiesta dalla funzione softirq. open_softirq() si limita a a inizializzare la entry appropriata dell'array softirq_vec.

I softirq sono attivati da raise_softirq(). Questa funzione riceve come parametro l'indice nr e compie le seguenti azioni:

1 – Esegue la macro local_irq_save per salvare lo stato del flag IF del registro eflags e disabilitare gli interrupt sulla CPU locale.

2 – Identifica il softirq come “in attesa” impostando il bit corrispondente all'indice nr nella maschera di bit softirq della CPU locale.

3 – Se in_interrupt() restituisce il valore 1, passa al punto 5. Questo indica che o la raise_softirq() è stata chiamata in un contesto di interrupt o che i softirq sono attualmente disabilitati.

4 – In caso contrario chiama wakeup_softirqd() per riattivare, se necessario, il thread del kernel *ksoftirqd* della CPU locale.

5 – Esegue la macro local_irq_restore per ripristinare il valore di IF salvato al punto 1.

I controlli per la presenza di softirq attivi vanno ripetuti periodicamente, ma senza produrre eccessivo

sovraccarico. Sono effettuati in pochi punti del codice del kernel. Segue una lista dei punti più significativi, tenendo conto che la posizione dei controlli cambia a seconda della versione del kernel e dell'architettura:

- quando il kernel chiama local_bh_enable()[3] per abilitare i softirq sulla CPU locale;
- quando la do_IRQ()termina di gestire un interrupt di I/O e chiama la macro irq_exit();
- se il sistema usa un I/O APIC, quando smp_apic_timer_interrupt() termina di gestire un interrupt di timer locale;
- in un sistema multiprocessore, quando una CPU termina l'esecuzione di una funzione chiamata da un interrupt interprocessore di tipo CALL_FUNCTION_VECTOR;
- quando viene risvegliato uno dei thread del kernel *ksoftirqd*.

La funzione do_softirq()

Se vengono rilevati softirq in attesa in uno dei punti di controllo citati prima (local_softirq_pending() è diverso da zero), il kernel chiama do_softirq() per occuparsi di essi. La funzione compie le seguenti azioni:

1 – Se in_interrupt() restituisce il valore 1, la funzione ritorna. Ciò indica o che do_softirq() è stata chiamata in un contesto di interrupt o che i softirq sono attualmente disabilitati.

2 – Esegue local_irq_save per salvare lo stato del flag IF e disabilitare gli interrupt sulla CPU locale.

3 – Se la dimensione della struttura thread_union è 4 KB, passa allo stack soft IRQ. Questo punto è simile al punto 2 di do_IRQ(); chiaramente viene usato l'array softirq_ctx al posto di hardirq_ctx.

4 – Chiama __do_softirq().

5 – Se avvenuto il passaggio allo stack soft IRQ, ripristina lo stack pointer originale nel registro esp , ritornando allo stack delle eccezioni precedentemente in uso.

6 – Esegue local_irq_restore per ripristinare lo stato del flag IF salvato al punto 2 e ritorna.

La funzione __do_softirq()

La funzione __do_softirq() legge la maschera di bit della CPU ed esegue le funzioni differibili che corrispondono ad ogni bit attivo. Mentre è in corso l'esecuzione, altri softirq in attesa possono apparire; per assicurare un ritardo contenuto per le funzioni differibili, __do_softirq() rimane in esecuzione fino a che tutti i softirq in attesa sono stati eseguiti. Questo meccanismo a volte obbliga la funzione a rimanere attiva per lunghi periodi, ritardando considerevolmente i processi in modalità utente. Per questo la funzione esegue un numero fisso di iterazioni, poi ritorna. I rimanenti softirq in attesa, verranno gestiti dal thread *ksoftirqd.* Seguono le principali azioni della funzione __do_softirq():

1 – Inizializza il contatore di iterazioni a 10.

2 – Copia la maschera di bit di softirq della CPU locale (selezionata da local_softirq_pending()) nella variabile locale pending.

3 – Chiama local_bh_disable() per incrementare il contatore softirq. E' in qualche modo contrario alla logica

3 Il nome si riferisce ad una funzione differibile chiamata “bottom half” che non esiste più in Linux 2.6

che le funzioni differibili vengano disabilitate prima di essere eseguite, ma questa procedura ha un significato reale. Poiché le funzioni differibili vengono eseguite con gli interrupt abilitati, un interrupt può capitare nel mezzo dell'esecuzione di __do_softirq(). Quando do_IRQ() esegue la macro irq_exit(), potrebbe essere avviata un'altra istanza della __do_softirq(). Questo va evitato, perché le funzioni differibili devono essere eseguite in modo seriale. Perciò la prima istanza di __do_softirq() disabilita le funzioni differibili, così che ogni sua eventuale nuova istanza venga arrestata al punto 1.

4 – Azzera la bitmap dei softirq della CPU locale, in modo che nuovi softirq possano essere attivati (il valore della maschera di bit è stato già salvato nella variabile locale pending).

5 – Esegue local_irq_enable() per abilitare gli interrupt locali.

6 – Per ogni bit attivato nella variabile pending, esegue la corrispondente funzione softirq; da ricordare che l'indirizzo della funzione per il softirq di indice n è conservato in softirq_vec[n]->action.

7 – Esegue local_irq_disable() per disabilitare gli interrupt.

8 – Copia la maschera di bit dei softirq per la CPU locale nella variabile locale pending e decrementa il contatore di iterazioni .

9 – Se pending non è zero, cioè almeno un softirq è stato attivato a partire dall'inizio dell'ultima iterazione, e il contatore di iterazione è ancora positivo, torna al punto 4.

10 – Se ci sono più softirq in attesa, chiama wakeup_softirqd() per risvegliare il thread che si occupa del softirq per la CPU locale.

11 – Sottrae 1 dal contatore di softirq, abilitando di nuovo le funzioni differibili.

```
#define MAX_SOFTIRQ_RESTART 10

asmlinkage void __do_softirq(void)
{
	struct softirq_action *h;
	__u32 pending;
	int max_restart = MAX_SOFTIRQ_RESTART;
	int cpu;
	pending = local_softirq_pending();
	local_bh_disable();
	cpu = smp_processor_id();
restart:
	/* Reset the pending bitmask before enabling irqs */
	local_softirq_pending() = 0;
	local_irq_enable();
	h = softirq_vec;
	do {
		if (pending & 1) {
			h->action(h);
			rcu_bh_qsctr_inc(cpu);
		}
		h++;
		pending >>= 1;
	} while (pending);
```

```
	local_irq_disable();
	pending = local_softirq_pending();
	if (pending && --max_restart)
		goto restart;
	if (pending)
		wakeup_softirqd();
	__local_bh_enable();
}
```

Il thread ksoftirqd

Nei kernel più recenti, ogni CPU ha il proprio thread *ksoftirqd/n*, dove n è il numero logico della CPU. Ogni thread esegue la funzione ksoftirqd() che esegue il codice seguente:

```
	set_user_nice(current, 19);
	current->flags |= PF_NOFREEZE;

	set_current_state(TASK_INTERRUPTIBLE);

	while (!kthread_should_stop()) {
		if (!local_softirq_pending())
			schedule();

		__set_current_state(TASK_RUNNING);

		while (local_softirq_pending()) {
			/* Preempt disable stops cpu going offline.
			   If already offline, we'll be on wrong CPU:
			   don't process */
			preempt_disable();
			if (cpu_is_offline((long)__bind_cpu))
				goto wait_to_die;
			do_softirq();
			preempt_enable();
			cond_resched();
		}

		set_current_state(TASK_INTERRUPTIBLE);
	}
	__set_current_state(TASK_RUNNING);
	return 0;
```

Il thread controlla la bit mask local_softirq_pending() e chiama se necessario do_softirq(). Se non ci sono softirq in attesa, pone il processo corrente nello stato TASK_INTERRUPTIBLE e chiama cond_resched() per operare un cambio di contesto se richiesto dal processo corrente (flag TIF_NED_RESCHED di thread_info attivato).

ksoftirqd è la soluzione per un problema di trade off. Le funzioni softirq possono riattivare sé stesse; lo fanno sia i softirq di rete sia i softirq tasklet. In più, eventi esterni come un flood di pacchetti su una scheda

di rete, possono attivare i softirq con frequenza elevata. Il problema di un elevato volume di softirq è risolto dal thread del kernel. Senza di esso, gli sviluppatori del kernel sarebbero di fronte a due strategie alternative.

La prima consiste nell'ignorare nuovi softirq mentre do_softirq() è in esecuzione. In altre parole, la funzione potrebbe determinare quali softirq sono in attesa al suo avvio ed eseguirli. Poi potrebbe terminare senza ripetere il controllo. La soluzione non è soddisfacente. Si pensi ad un softirq che viene riattivato mentre do_softirq() è in esecuzione; nel caso peggiore il softirq non è eseguito di nuovo fino all'interrupt successivo, anche se la macchina è inattiva. Il risultato è una latenza inaccettabile per i softirq.

La seconda strategia consiste nel controllare di continuo i softirq in attesa. La funzione potrebbe iniziare il controllo e terminare solo se non ci sono più softirq in attesa. Questa soluzione potrebbe essere soddisfacente per il networking, ma non per i normali utenti del sistema: se venisse ricevuto un intenso flusso di pacchetti, la do_softirq() non ritornerebbe mai e i programmi utente sarebbero praticamente arrestati.

Il thread *ksoftirqd* tenta di risolvere questo problema. La do_softirq() determina quali softirq sono in attesa e li esegue. Dopo alcune iterazioni, se il flusso di softirq non si arresta, la funzione risveglia il thread e termina. Quest'ultimo ha bassa priorità, per cui i programmi utente hanno la possibilità di essere eseguiti: se però la macchina è inattiva, i softirq in attesa vengono eseguiti rapidamente.

Tasklet

I tasklet sono il modo preferito per implementare funzioni differibili per driver di I/O. Sono implementati al di sopra dei due tipi di softirq chiamati HI_SOFTIRQ e TASKLET_SOFTIRQ. Vari tasklet, ognuno dei quali esegue la propria funzione, possono essere associati ad uno stesso softirq. Non c' è una vera differenza tra i due tipi di softirq, tranne il fatto che do_softirq() esegue prima i tasklet di HI_SOFTIRQ rispetto agli altri.

I tasklet normali e quelli a priorità elevata sono memorizzati rispettivamente negli array tasklet_vec e tasklet_hi_vec. Entrambi includono NR_CPUS elementi di tipo tasklet_head, che sono puntatori ad una lista di *descrittori di tasklet*. Il descrittore è una struttura di tipo tasklet_struct che ha i campi seguenti:

Nome	Descrizione
next	Puntatore al descrittore successivo della lista
state	Stato del tasklet
count	Contatore di lock
func	Puntatore alla funzione del tasklet
data	Un intero long senza segno che può essere usato da func

Il campo state include due flag:

TASKLET_STATE_SCHED: se attivato, indica che il tasklet è in attesa (è stato schedulato per l'esecuzione); significa anche che il descrittore del tasklet è inserito in una delle liste degli array tasklet_vec e tasklet_hi_vec.

TASKLET_STATE_RUN: se attivato, indica che il tasklet è in fase di esecuzione; in sistemi uniprocessore non è usato perché non c'è necessità di eseguire questo controllo.

Se uno sviluppatore sta scrivendo un driver di dispositivo e vuole usare un tasklet, prima di tutto deve allocare una nuova struttura tasklet_struct ed inizializzarla con tasklet_init(); questa funzione riceve come parametri l'indirizzo del descrittore del tasklet, l'indirizzo della funzione e il suo argomento intero opzionale.

Il tasklet può essere disabilitato selettivamente chiamando o tasklet_disable_nosync() o tasklet_disable(). Entrambe incrementano il campo cont, ma l'ultima non ritorna fino a che un'istanza del tasklet già in esecuzione non ha terminato. Per abilitare i tasklet si usa tasklet_enable().

Per attivare il tasklet si usa o tasklet_schedule() o tasklet_hi_schedule(), a seconda della priorità. Le funzioni sono molto simili ed eseguono le azioni seguenti:

1 – Controllano il flag TASKLET_STATE_SCHED; se è attivato, ritornano, poiché il tasklet è già stato schedulato.

2 – Chiamano local_irq_save per salvare lo stato del flag IF e disabilitare gli interrupt locali.

3 – Aggiungono il descrittore del tasklet all'inizio della lista puntata da tasklet_vec[n], dove n è il numero logico della CPU.

4 – Chiamano raise_softirq_irqoff() per attivare o TASKLET_SOFTIRQ o HI_SOFTIRQ; la funzione è simile a raise_softirq(), eccetto che presuppone che gli interrupt locali siano ancora disabilitati.

5 – Chiamano local_irq_restore per ripristinare lo stato del flag IF.

Infine occorre considerare come vengono eseguiti i tasklet. La funzione softirq associata al tipo HI_SOFTIRQ è chiamata tasklet_hi_action(), mentre quella associata a TASKLET_SOFTIRQ è chiamata tasklet_action(). Ancora una volta le funzioni sono molto simili e ognuna di esse:

1 – Disabilita gli interrupt locali.

2 – Ricava il numero logico n della CPU.

3 – Copia l'indirizzo della lista puntata da tasklet_vec[n] o tasklet_hi_vec[n] nella variabile locale list.

4 – Memorizza un indirizzo NULL nella lista, vuotando così la lista dei descrittori di tasklet schedulati.

5 – Abilita gli interrupt locali.

6 – Per ogni descrittore di tasklet puntato da list:

a – in un sistema multiprocessore controlla il flag TASKLET_STATE_RUN del tasklet.

- se è attivato, un tasklet dello stesso tipo è in esecuzione su un'altra CPU; la funzione reinserisce il descrittore nella lista e attiva di nuovo TASKLET_SOFTIRQ o HI_SOFTIRQ. Così l'esecuzione è rimandata fino a che nessun altro tasklet dello stesso tipo è in esecuzione su una CPU.
- Se è azzerato, nessun altro tasklet è in esecuzione; attiva il flag in modo che la funzione non venga eseguita su un'altra CPU

b – Controlla se il tasklet è disabilitato controllando il campo count del descrittore. Se è disabilitato, azzera il flag TASKLET_STATE_RUN e reinserisce il descrittore di tasklet nella lista; poi attiva di nuovo TASKLET_SOFTIRQ o HI_SOFTIRQ

c – Se il tasklet è abilitato, azzera TASKLET_STATE_SCHED ed esegue la funzione del tasklet.

Da notare che finché la funzione non riattiva sé stessa, ogni attivazione di tasklet provoca una sola esecuzione della funzione del tasklet.

Work queue

Le *work queue* (code di lavoro, WQ) sono state introdotte in Linux 2.6 e rimpiazzano un costrutto simile chiamato “task queue”. Esse consentono alle funzioni del kernel di essere attivate, in modo simile alle funzioni differibili, e poi eseguite da speciali thread del kernel chiamati *worker thread*.

Funzioni differibili e WQ sono differenti: le prime sono eseguite nel contesto di un interrupt, mentre le funzioni delle WQ sono eseguite nel contesto di un processo. Questo permette di eseguire funzioni bloccanti (ad esempio funzioni che richiedono accessi al disco), mentre nel contesto di un interrupt non può avvenire nessuna commutazione di contesto. Né funzioni differibili né funzioni in WQ possono accedere allo spazio di indirizzamento in modalità utente di un processo. Infatti una funzione differibile non può fare ipotesi sul processo in corso di esecuzione. D'altra parte una funzione in una WQ è eseguita da un thread del kernel, per cui non esiste uno spazio di indirizzamento in modalità utente.

Strutture dati delle work queue

La struttura principale è un descrittore chiamato workqueue_struct che contiene, tra le altre cose, un array di NR_CPUS elementi, dove NR_CPUS è il numero massimo di CPU del sistema[4]. Ogni elemento è un descrittore di tipo cpu_workqueue_struct, i cui campi sono i seguenti:

Campo	Descrizione
lock	Spinlock per proteggere la struttura
remove_sequence	Numero di sequenza usato da flush_workqueue()
insert_sequence	Numero di sequenza usato da flush_workqueue()
worklist	Inizio della lista di funzioni in attesa
more_work	Coda in cui il thread è in attesa di altro lavoro da compiere
work_done	Coda in cui il processo è in attesa del flush della WQ
wq	Puntatore alla struttura workqueue_struct che contiene il descrittore
thread	Puntatore al descrittore di processo del thread della struttura
run_depth	Profondità di esecuzione di run_workqueue (diventa >1 se una funzione nella WQ si blocca)

Il campo worklist è l'inizio di una lista a collegamento doppio che raggruppa le funzioni in attesa della WQ. Ogni funzione in attesa è rappresentata da una struttura work_struct, i cui campi sono i seguenti:

Campo	Descrizione
pending	Vale 1 se la funzione è già in una WQ, 0 se non lo è

4 La ragione della duplicazione delle strutture è che una struttura per ogni CPU porta a codice molto più efficiente

Campo	Descrizione
entry	Puntatore agli elementi precedente e successivo della lista
func	Indirizzo della funzione in attesa
data	Puntatore passato come parametro della funzione
wq_data	Di solito punta al descrittore del genitore di cpu_workqueue_struct
timer	Timer software usato per ritardare l'esecuzione della funzione.

Funzioni delle work queue

La funzione create_workqueue(foo) riceve come parametro una stringa di caratteri e restituisce l'indirizzo del descrittore di una workqueue_struct per la WQ appena creata. Crea anche n worker thread (dove n è il numero delle CPU del sistema) chiamati foo/0, foo/1, …, foo/n.

La funzione create_singlethread_workqueue() è simile, ma crea un solo thread, indipendentemente dal numero di CPU. Per eliminare una WQ il kernel chiama destroy_workqueue(), che riceve come parametro un puntatore all'array workqueue_struct.

queue_work() inserisce una funzione (già collegata al descrittore work_struct) in una WQ; riceve un puntatore wq al descrittore workqueue_struct e un puntatore work al descrittore work_struct; esegue le azioni seguenti:

1 – Controlla se la funzione da inserire è già presente nella WQ (work->pending = 1); se è così, termina.

2 – Aggiunge il descrittore work_struct alla lista WQ e imposta work->pending a 1.

3 – Se un worker thread è quiescente nella coda di attesa more_work del descrittore cpu_workqueue_struct della CPU locale, la funzione lo risveglia.

La funzione queue_delayed_work() è quasi identica alla precedente, tranne che riceve un terzo parametro che rappresenta un intervallo di ritardo in tick di sistema. Viene usata per assicurare un minimo ritardo prima dell'esecuzione delle funzioni in attesa. In pratica questa funzione si basa sul timer software nel campo timer del descrittore work_struct per ritardare l'inserimento dela descrittore work_struct nella lista della WQ. cancel_delayed_work() cancella una funzione WQ schedulata in precedenza, purché il descrittore work_struct corrispondente non sia già stato inserito nella lista.

Ogni worker thread esegue di continuo un ciclo entro la funzione worker_thread(); per la maggior parte del tempo rimane quiescente in attesa di un lavoro da mettere in coda. Una volta risvegliato, chiama la funzione run_workqueue() che rimuove tutti i descrittori work_struct dalla lista della work queue del thread ed esegue la corrispondente funzione in attesa. Poiché le funzioni possono bloccarsi, il worker thread può essere messo a riposo ed anche migrare su un'altra CPU[5].

Talvolta il kernel deve attendere fino a che tutte le funzioni in una WQ vengano eseguite. flush_workqueue() riceve l'indirizzo di un descrittore workqueue_struct e blocca il processo chiamante fino a che tutte le funzioni in attesa terminano. La funzione, comunque, non attende ogni funzione aggiunta alla WQ dopo la chiamata di flush_workqueue(); i campi remove_sequence e insert_sequence dei descrittori cpu_workqueue_struct sono usati per riconoscere la funzioni aggiunte di recente.

5 Un worker thread può essere eseguito da ogni CPU, non solo quella che corrisponde al descrittore cpu_workqueue_struct. Quindi queue_work() inserisce una funzione nella coda della CPU locale, ma la funzione può essere eseguita da ogni CPU.

La work queue predefinita

Nella maggior parte dei casi, creare un intero set di worker thread per eseguire una funzione è uno spreco. Perciò il kernel offre una WQ predefinita chiamata *event,* che può essere liberamente usata da ogni sviluppatore del kernel. Non è altro che una WQ standard che può includere funzioni di differenti strati del kernel e di driver di I/O; il suo descrittore workqueue_struct è memorizzato nell'array keventd_wq. Per usare questa WQ predefinita, il kernel offre le funzioni elencate in tabella:

Funzioni di WQ predefinite	Funzioni equivalenti di WQ standard
schedule_work(w)	queue_work(keventd_wq, w)
schedule_delayed_work(w,d)	queue_delayed_work(keventd_wq, w,d) (su ogni CPU)
schedule_delayed_work_on(cpu,w,d)	queue_delayed_work(keventd_wq, w,d) (su una CPU definita)
flush_schedule_work()	flush_workqueue(keventd_wq)

La WQ predefinita preserva risorse significative di sistema quando la funzione è invocata raramente. D'altra parte le funzioni eseguite nella WQ predefinita non dovrebbero bloccarsi per lungo tempo: poiché l'esecuzione delle funzioni in attesa avviene in serie su ogni CPU, un lungo ritardo influisce negativamente sugli altri utenti della WQ predefinita.

In aggiunta alla coda events, in Linux 2.6 si trovano alcune WQ specializzate . La più significativa è kblockd usata dallo strato dei dispositivi a blocchi.

Ritorno da interrupt ed eccezioni

Anche se l'obiettivo del ritorno da interrupt ed eccezioni è chiaro – riprendere l'esecuzione di qualche programma – prima di realizzarlo bisogna considerare vari aspetti:

numero di KCP eseguiti in modo concorrente: se ne esiste solo uno, la CPU deve ritornare in modalità utente;

richieste di commutazione di processi in attesa: se c'è qualche richiesta, il kernel deve eseguire lo scheduling; altrimenti il controllo ritorna al processo corrente;

segnali in attesa: se un segnale è inviato al processo corrente, deve essere gestito;

modalità a passo-singolo: se un debugger sta tracciando il processo corrente, la modalità a passo singolo deve essere ripristinata prima di tornare alla modalità utente;

modalità virtuale 8086: se la CPU è in modalità virtuale, il processo corrente sta eseguendo un vecchio programma in modalità reale, così deve essere gestito in modo speciale.

Alcuni flag sono usati per tenere traccia delle richieste di commutazione di processo e di segnali in sospeso, nonché di esecuzione a passo singolo; sono memorizzati nel campo flags del descrittore thread_info. Il campo contiene altri flag non correlati al ritorno da interrupt ed eccezioni.

Il codice del kernel in assembly che realizza queste operazioni non è tecnicamente una funzione, perché il controllo non è mai restituito alla funzione chiamante. E' una porzione di codice con due punti di ingresso: ret_from_intr() e ret_from_exception(). Come suggerito dai nomi, il kernel usa il primo quando ritorna da un interrupt e il secondo quando ritorna da un'eccezione. Il riferimento ad essi come funzioni facilita la

descrizione.

La differenza tra i due punti di ingresso esiste solo se il kernel è stato compilato senza supporto al pre-rilascio; in questo caso gli interrupt locali sono immediatamente disabilitati al ritorno da una eccezione.

Flag	Descrizione
TIF_SYSCALL_TRACE	Le chiamate di sistema vengono tracciate
TIF_NOTIFY_RESUME	Non usata nella piattaforma 80x86
TIF_SIGPENDING	Il processo ha segnali in sospeso
TIF_NEED_RESCHED	Deve essere effettuato lo scheduling
TIF_SINGLESTEP	Ripristina l'esecuzione a passo singolo al ritorno in modalità utente
TIF_IRET	Forza il ritorno da una chiamata di sistema con iret piuttosto che sysexit
TIF_SYSCALL_AUDIT	Le chiamate di sistema devono essere controllate
TIF_POLLING_NRFLAG	Il processo idle sta interrogando ciclicamente il flag TIF_NEED_RESCHED
TIF_MEMDIE	Il processo deve essere distrutto per recuperare memoria

Punti di ingresso

Sono equivalenti in sostanza a questo codice assembly:

```
ret_from_exception:
        cli                     ; assente se ipreemption non è supportata
ret_from_interrupt:
        movl $-8192, %ebp       ; -4096 se vengono usati gli stack multipli
        andl %esp, %ebp
        movl 0x30(%esp), %eax
        movb 0x2c(%esp), %al
        testl $0x00020003, %eax
        jnz resume_userspace
        jmp resume_kernel
```

Quando si ritorna da un interrupt, gli interrupt locali sono disabilitati; perciò l'istruzione cli è eseguita solo al ritorno da un'eccezione.

Il kernel carica l'indirizzo del descrittore thread_info di current nel registro ebp. Poi i valori di cs ed eflags, salvati nello stack al momento della gestione dell'interrupt o dell'eccezione, sono adoperati per verificare se il programma interrotto era eseguito in modalità utente, o se il flag VM era attivato (in questo caso i programmi sono eseguiti in modalità virtuale 8086). In entrambi questi ultimi i casi, viene eseguito un salto a resume_userspace; altrimenti si passa a resume_kernel.

Ripresa dell'esecuzione di un kernel control path

```
resume_kernel:
```

```
        cli                  ; le tre istruzioni mancano se la preemption non è supportata
        cmpl $0, 0x14(%ebp)
        jz need_resched
    restore all:
        popl %ebx
        popl %ecx
        popl %edx
        popl %esi
        popl %edi
        popl %ebp
        popl %eax
        popl %ds
        popl %es
        addl $4, %esp
        iret
```

Se il campo preempt_count di thread_info è zero (pre-rilascio abilitato), il kernel salta a need_resched, in caso contrario occorre far ripartire il programma interrotto. Il codice ripristina i registri salvati al momento dell'interrupt o dell'eccezione e passa il controllo eseguendo iret.

Controllo della preemption del kernel

Quando viene eseguito questo codice, nessuno dei KCP non terminati è un gestore di interrupt, altrimenti il campo preempt-count sarebbe maggiore di zero. Però ci potrebbero essere fino a due KCP associati ad eccezioni, accanto a quello che sta terminando.

```
    need_resched:
        movl 0x8(%ebp), %ecx
        testb $(1<<TIF_NEED_RESCHED), %cl
        jz restore_all
        call preempt_schedule_irq
        jmp need_resched
```

Se il flag TIF_NEED_RESCHED nel campo flags di current->thread_info è zero non è richiesta nessuna commutazione di processo, per cui si salta all'etichetta restore_all. Lo stesso salto è eseguito se il KCP che deve essere ripreso era eseguito con gli interrupt locali disabilitati. In questo caso una commutazione di processo potrebbe corrompere le strutture dati del kernel.

Se è richiesta una commutazione di processo, viene chiamata la preempt_schedule_irq(): attiva il flag PREEMPT_ACTIVE nel campo preempt_count, imposta temporaneamente a -1 il contatore del “big kernel lock”, abilita gli interrupt locali e chiama schedule() per selezionare un altro processo per l'esecuzione. Quando il processo precedente riprenderà l'esecuzione, preempt_schedule_irq() ripristina il valore del contatore del “big kernel lock”, azzera il flag PREEMPT_ACTIVE e disabilita gli interrupt locali. La funzione schedule()continuerà ad essere chiamata finché TIF_NEED_RESCHED del processo corrente sarà attivo.

Ripresa dell'esecuzione di un programma in modalità utente

In questo caso viene eseguito un salto all'etichetta resume_userspace:

```
    resume_userspace:
```

```
cli
movl 0x8(%ebp), %ecx
andl $0x0000ff6e, %ecx
je restore_all
jmp work_pending
```

Dopo aver disabilitato gli interrupt locali, viene fatto un controllo sul valore del campo flags di current->thread_info. Se non è attivato nessun flag tranne TIF_SYSCALL_TRACE, TIF_SYSCALL_AUDIT o TIF_SINGLESTEP, non rimane nulla da fare: viene fatto un salto a restore_all, riprendendo l'esecuzione del programma in modalità utente.

Controllo per il rescheduling

I flag nel descrittore thread_info stabiliscono se è richiesto un lavoro aggiuntivo prima di riprendere il programma interrotto.

```
work_pending:
        testb $(1<<TIF_NEED_RESCHED), %cl
        jz work_notifysig
work_resched:
        call schedule
        cli
        jmp resume_userspace
```

Se la richiesta di una commutazione di contesto è in attesa, viene chiamata schedule() per selezionare un altro processo. Quando verrà ripreso il processo precedente, verrà eseguito un jmp a resume_userspace.

Gestione dei segnali in sospeso, modalità virtuale 8086 ed esecuzione a passo singolo

Ci sono altre attività da svolgere prima della commutazione di contesto:

```
work_notifysig:
        movl %esp, %eax
        testl $0x00020000, 0x30(%esp)
        je 1f
work_notifysig_v86:
        pushl %ecx
        call save_v86_state
        popl %ecx
        movl %eax, %esp
1:
        xorl %edx, %edx
        call do_notify_resume
        jmp restore_all
```

Se il flag VM di eflags del programma in modalità utente è attivato, viene chiamata save_v86_state() per costruire le strutture della modalità virtuale nello spazio di indirizzamento in modalità utente. Poi viene chiamata do_notify_resume() per occuparsi dei segnali in attesa e della esecuzione a passo singolo. Finalmente viene fatto il salto all'etichetta restore_all per riprendere il programma interrotto.