Azzolini Riccardo 2019-10-02

Program interrupt

1 Program interrupt

Oltre agli hardware interrupt, esistono due tipi di **program interrupt**:

- *eccezioni*;
- *software interrupt*.

2 Eccezioni

Le **eccezioni**, dette anche **interrupt interni**, si suddividono in:

- *eccezioni aritmetiche* (divisione per zero, overflow);
- *eccezioni di indirizzamento* (page fault);
- *violazione delle protezioni di memoria* (violation fault).

A differenza degli hardware interrupt, questi non sono eventi asincroni rispetto all'esecuzione del programma, perché sono causati da *situazioni anomale*[1] interne al programma stesso. Tali situazioni richiedono l'intervento del SO, che spesso impone la terminazione del programma.

3 Software interrupt

I programmi user devono avere il modo di richiedere l'intervento del SO, per usufruire dei servizi che questo offre (I/O, ecc.). Ciò non può avvenire mediante una normale chiamata di procedura, perché sarebbe necessario passare dalla modalità user alla modalità kernel prima di effettuare la chiamata (dato che il codice del SO si trova nella kernel area della RAM), ma il programma user non può farlo.

L'unico modo per richiedere l'intervento del SO sono i **software interrupt**, detti anche **trap**: essi sono causati da un'*apposita istruzione*, chiamata **Software Interrupt Instruction** o, appunto, `TRAP`.

[1] Come caso particolare, i page fault non sono causati da situazioni anomale, ma sono invece fondamentali per il funzionamento dei programmi: essi permettono di mantenere i programmi in parte in RAM e in parte sul disco.

Di solito, il linguaggio macchina prevede una singola istruzione `TRAP`, che si usa per chiedere tutti i possibili interventi del SO. Essa ha quindi un *parametro*, che specifica il tipo di intervento richiesto e che, a seconda dell'instruction set, può essere passato in due modi:

- come operando della `TRAP`;
- ponendolo sullo stack.

Il valore del parametro viene poi assegnato ai bit *Interrupt Code* (IC) della PSW, che l'handler legge per determinare cosa fare.

Essendo causati da un'istruzione scritta esplicitamente nel programma, anche i software interrupt (come le eccezioni) non sono eventi asincroni.

3.1 Funzionamento

Quando un programma esegue l'istruzione `TRAP`:

1. I valori dei registri di controllo vengono salvati nella SRIA.
2. I registri di controllo vengono impostati ai valori specificati nell'interrupt vector.
3. L'interrupt handler:
   a) salva il contenuto dei registri generali;
   b) controlla l'IC per individuare il parametro della `TRAP`;
   c) esegue le sue funzioni, in base al parametro;
   d) invoca lo scheduler.
4. Se seleziona il programma interrotto, lo scheduler:
   a) ripristina i registri generali;
   b) ripristina i registri di controllo con i valori salvati nella SRIA (mediante l'istruzione `IRET`), facendo così ripartire il programma.

Solitamente, il SO deve restituire un valore al programma che ne ha richiesto l'intervento (ad esempio, un codice che indica l'esito dell'operazione). Questo valore deve essere posto in uno dei registri generali (tipicamente, il registro numero 0).

Siccome, quando il programma riparte, i contenuti dei registri vengono ripristinati a quelli salvati in memoria al momento dell'interrupt, il SO non può semplicemente assegnare il valore da restituire al registro, perché questo verrebbe sovrascritto: invece, modifica direttamente il valore salvato in memoria, che verrà caricato nel registro quando il programma ripartirà.

4 Priorità dei program interrupt

I program interrupt vengono trattati come gli hardware interrupt, e si collocano quindi nella gerarchia delle priorità, tipicamente con priorità minore a tutti gli hardware interrupt:

1. machine error;
2. clock interrupt;
3. disk interrupt;
4. fast device interrupt;
5. slow device interrupt;
6. exception;
7. trap.

5 System call

Una **system call** è una *richiesta al SO* effettuata da un programma.

- Le system call sono lo strumento messo a disposizione dei programmi per avvalersi dei servizi offerti dal SO: esse costituiscono quindi l'interfaccia tra i programmi utente e il SO.
- Le system call sono realizzate con la `TRAP`, e ne esiste una per ogni valore consentito del parametro della `TRAP`. Se l'IC è costituito da n bit, possono esistere al massimo 2^n system call. Tipicamente, si ha $n \geq 8$. Comunque, quali e quante sono esattamente le system call dipende dal SO.
- Il programma che invoca la system call ottiene un risultato (il valore del dato richiesto, quale ad esempio il clock di sistema, oppure un codice che indica se l'operazione è riuscita, segnalando eventuali errori, ecc.), solitamente memorizzato in un registro.

5.1 Invocazione

- In assembly, per invocare le system call si usa l'istruzione `TRAP`. Oltre al parametro della `TRAP`, che specifica quale system call eseguire, ciascuna system call richiede dei propri parametri, che devono essere passati opportunamente. Il programmatore deve quindi sapere quali sono questi parametri e dove l'interrupt handler si aspetta di trovarli.

- In C, è disponibile una *libreria di funzioni associate alle system call*. Esse sono scritte in assembly, perché al loro interno devono eseguire l'istruzione `TRAP`, ma sono invocabili dal programma in C. I parametri e il valore restituito sono gestiti come per le normali chiamate di funzione.

5.2 Esempio: system call UNIX/Linux

In UNIX/Linux si hanno circa 100 system call. Di queste, alcune servono per interagire con il file subsystem (che gestisce sia i file che i dispositivi, dato che in UNIX questi due hanno la medesima interfaccia), mentre altre riguardano il process control subsystem, ecc.

Alcuni esempi di system call per la gestione dei file/device sono:

- *create*, per creare un file;
- *open*, per creare un accesso a un file/device (il che è necessario prima di poter effettuare letture e/o scritture su di esso);
- *read*, per leggere da un file/device il numero di byte specificato come parametro;
- *write*, per scrivere in un file/device il numero di byte specificato come parametro;
- *close*, per invalidare un accesso a un file/device quando si ha finito di operare su di esso.

Un esempio di programma in C che usa queste system call mediante le funzioni di libreria corrispondenti è:

```c
#include <fcntl.h>
#include <unistd.h>

int main(void) {
    int fd = open("fileDiProva", O_RDONLY);
    char buf1[20];
    read(fd, buf1, 20);
    char buf2[30];
    read(fd, buf2, 30);
    close(fd);
}
```

- La system call *open* ha 2 parametri, il nome del file da aprire e il tipo di apertura, e restituisce un "file descriptor", cioè un numero intero che consente di accedere al file. In questo programma, viene invocata la funzione di libreria corrispondente, `open`, per aprire il file chiamato `fileDiProva` in sola lettura (`O_RDONLY`).

- La system call *read* ha 3 parametri: il file descriptor del file a cui accedere, il buffer di memoria su cui scrivere, e il numero di byte da leggere. Questo programma la invoca mediante la funzione di libreria `read`, leggendo i primi 20 byte di `fileDiProva`, che vengono messi nel buffer `buf1`, e poi i 30 byte successivi, che vengono memorizzati in `buf2`.
- La system call *close*, e quindi la funzione di libreria `close`, ha un singolo parametro, il file descriptor (accesso al file) da invalidare. Invocandola, questo programma segnala al SO che non ha più bisogno di accedere al file.

5.3 Funzioni di libreria in Windows

In UNIX/Linux si ha, di norma, una funzione di libreria per ogni system call.

Invece, Windows definisce alcune migliaia di procedure messe a disposizione dei programmi per invocare i servizi del SO, ma le system call sono molte meno delle procedure:

- procedure diverse invocano le stesse system call;
- alcune procedure non ne invocano alcuna, e vengono eseguite interamente in modalità user.