Eccezioni

G. Lettieri

20 Aprile 2018

Le prime 32 entrate della Interrupt Descriptor Table sono riservate per le *eccezioni*. Con questo termine ci si riferisce a condizioni di errore o speciali che il processore rileva mentre sta eseguendo le normali istruzioni. Per esempio, una operazione di divisione (`div` o `idiv`) in cui il divisore è zero causa una eccezione di "divisione per zero". Il processore tratta le eccezioni in modo molto simile alle interruzioni esterne: vi associa un tipo numerico (compreso tra 0 e 31) e lo usa per accedere ad un gate della IDT, dove trova l'indirizzo di una routine a cui saltare. I tipi delle eccezioni sono fissi e consultabili sul manuale del processore. Per esempio, l'eccezione di divisione per zero ha tipo 0. Il meccanismo di salto è del tutto simile a quello delle interruzioni esterne, con il salvataggio in pila di informazioni analoghe. In particolare, anche le routine di gestione delle eccezioni, se vogliono tornare al programma principale, devono farlo con una istruzione **`iretq`** (e non una semplice **`ret`**).

Le eccezioni sono classificabili ulteriormente come segue:

trap vengono sollevate solo tra l'esecuzione di una istruzione e la successiva;

fault vengono sollevate *durante* l'esecuzione di una istruzione;

abort possono verificarsi in qualsiasi momento e indicano errori particolarmente gravi.

Per le eccezioni di tipo fault il processore salva in pila l'indirizzo dell'istruzione che stava eseguendo mentre ha rilevato il fault, e non l'indirizzo dell'istruzione successiva come per i trap e le interruzioni esterne. L'idea è che a volte la routine di gestione dell'eccezione può correggere la condizione che ha causato il fault e poi ritornare (con la **`iretq`** finale) a *rieseguire* l'istruzione, che questa volta non dovrebbe più generare il fault o al massimo dovrebbe generarne uno diverso, per qualche altra condizione di errore.

Si noti che, per rendere possibile la riesecuzione di una istruzione che era stata precedentemente eseguita fino ad un certo punto (e dunque poteva aver già modificato qualche registro), il processore deve essere in grado di ritornare allo stato precedente l'inizio della prima esecuzione. Per quanto riguarda lo stato dei registri, il processore può apportare tutte le modifiche in delle copie di lavoro, trasferendo il contenuto dalle copie ai veri registri solo alla fine dell'esecuzione di ogni istruzione. Se si verifica un fault durate l'esecuzione, è sufficente ignorare

```
1 int main()
2 {
3         int x = 3, y = 0;
4         return x / y;
5 }
```

Figura 1: Un programma che causa una eccezione di "divisione per zero".

le copie di lavoro per ritornare allo stato precedente. Il caso delle istruzioni che scrivono in memoria va invece esaminato istruzione per istruzione, ma in generale non ci sono problemi se si esegue una sola scrittura alla fine dell'esecuzione (che è il caso più comune).

La libreria `libce`, che abbiamo usato fino ad ora per tutti gli esempi, inizializza la IDT in modo che ogni eccezione causi un salto ad una funzione (della libreria stessa) che stampa un messaggio di errore e blocca il processore eseguendo l'istruzione `hlt` (si veda la funzione `init_idt` nel file `64/init_idt.s`, seguendo la catena di chiamate fino a `gestore_eccezioni()`).

In Figura 1 troviamo un semplice programma che causa una eccezione. Si noti che il programma deve essere compilato senza ottimizzazioni, altrimenti il compilatore si accorge facilmente del tentativo di dividere per 0 e, dal momento che lo standard dichiara tale operazione come "indefinita", può tradurre l'operazione in qualunque modo. In questo caso, molto probabilmente, si limiterebbe ad eliminare l'operazione di divisione. Scriviamo il codice di Figura 1 in un file di nome `prova.cpp` e compiliamolo con lo script `compile`[1]. Possiamo controllare che il compilatore abbia effettivamente generato l'istruzione di divisione osservando l'output del comando

```
objdump -d | grep idivl
```

Dovremmo ottenere qualcosa di simile:

```
20015d: f7 7d f8                idivl  -0x8(%rbp)
```

Il primo numero è l'indirizzo a cui si troverà l'istruzione quando il programma verrà caricato.

Se lanciamo ora l'eseguible con lo script `boot` dovremmo vedere che la macchina virtuale parte e si ferma, scrivendo sulla console un messaggio di errore. Il messaggio mostra il tipo dell'eccezione (0) e l'instruction pointer salvato in pila. In questo caso dovrebbe coincidere con 0x20015d, perché l'eccezione 0 è di tipo fault.

Per vedere la differenza con le eccezioni di tipo trap proviamo a generare l'eccezione di *break-point*, che ha tipo 3. Questa eccezione è sollevata dall'istruzione

[1] Come per tutti gli esempi che usano gli script `compile` e `boot`, dobbiamo trovarci in una directory che contiene inizialmente solo i file `*.cpp` e `*.s` dell'esempio.

```
1  int main()
2  {
3          int x = 0;
4          x++;
5          asm("int3");
6          x++;
7          return x;
8  }
```

Figura 2: Un programma che causa una eccezione di "break-point".

int3 alla fine della sua esecuzione. Essendo di tipo trap, il processore deve salvare in pila l'indirizzo dell'istruzione successiva. Dopo aver copiato il codice di Figura 2 in un file `prova.cpp` e averlo compilato con lo script `compile`, prendiamo nota dell'indirizzo dell'istruzione **int3** e delle istruzioni che le stanno attorno, con il comando:

```
objdump -d | grep -C 1 int3
```

Dovremmo ottenere un output del genere:

```
200152: 83 45 fc 01           addl   $0x1,-0x4(%rbp)
200156: cc                    int3
200157: 83 45 fc 01           addl   $0x1,-0x4(%rbp)
```

Se ora lanciamo il programma sulla macchina virtuale questa si ferma nuovamente con un messaggio di errore, questa volta per l'eccezione di tipo 3. L'instruction pointer salvato deve essere quello dell'istruzione *successiva* alla **int3** (0x200157 nel nostro caso).

Per eseguire una routine di nostra scelta ogni volta che si verifica una eccezione, operiamo in modo del tutto analogo a come abbiamo fatto per le interruzioni esterne: prepariamo un gate della IDT in modo che punti alla nostra funzione. Questa volta, però, non possiamo scegliere il gate che preferiamo, ma dobbiamo usare quello relativo all'eccezione che vogliamo intercettare. Per esempio, se vogliamo intercettare le eccezioni di divisione per zero dobbiamo usare il gate 0, e se vogliamo intercettare i break-point dobbiamo usare il gate 3.

Come esempio intercettiamo le eccezioni di tipo 1. Queste sono eccezioni di *debug* generate, tra le altre cose, dal meccanismo del *Single Step* (esecuzione passo passo). Tale meccanismo viene attivato settando il flag TF del registro dei flag. Quando tale flag è attivo il processore genera una eccezione di debug dopo aver eseguito *ogni istruzione*. Si tratta di una eccezione di trap, quindi l'instruction pointer salvato è quello dell'istruzione successiva (ancora da eseguire).

Il programma in Figura 3 associa la funzione `a_debug` al gate numero 1 (linea 14). La funzione `a_debug` è definita in Figura 4 e si limita a chiamare

```
1  #include <libce.h>
2
3  extern "C" void enable_single_step();
4  extern "C" void disable_single_step();
5  extern "C" void a_debug();
6  extern "C" void c_debug(void *rip)
7  {
8          flog(LOG_DEBUG, "rip=%p", rip);
9  }
10
11 int main()
12 {
13         int x = 0;
14         gate_init(1, a_debug);
15         enable_single_step();
16         x++;
17         x++;
18         x++;
19         x++;
20         disable_single_step();
21         pause();
22         return x;
23 }
```

Figura 3: Un programma che intercetta l'eccezione di debug, parte C++.

```
1  #include "libce.s"
2  .global enable_single_step
3  enable_single_step:
4          pushf
5          orw $0x0100, (%rsp)
6          popf
7          ret
8  .global disable_single_step
9  disable_single_step:
10         pushf
11         andw $0xFEFF, (%rsp)
12         popf
13         ret
14 .global a_debug
15 a_debug:
16         salva_registri
17         movq 120(%rsp), %rdi
18         call c_debug
19         carica_registri
20         iretq
```

Figura 4: Un programma che intercetta l'eccezione di debug, parte Assembler.

la funzione `c_debug` passandole il valore dell'instruction pointer salvato in pila dal processore (linea 17). Si noti che la funzione salva e ripristina tutti i registri, perché verrà chiamata dopo ogni istruzione del programma principale (se TF è settato). La macro `salva_registri` è definita nel file `libce.s` (nella libreria) e salva in pila tutti i registri generali tranne **`rsp`** (perché questo verrà ripristinato dal normale utilizzo della pila). Dunque, al momento di eseguire l'istruzione alla linea 17, l'instruction pointer si troverà 120 byte più in basso rispetto alla cima corrente della pila ($120 = 8 \times 15$).

La funzione `c_debug` (Figura 3) stampa il valore dell'instruction pointer utilizzando la funzione di libreria `flog()`, che invia il messaggio sulla porta seriale. Nel nostro caso abbiamo impostato la macchina virtuale affinché tutto ciò che viene inviato alla porta seriale venga mostrato sul terminale da cui abbiamo eseguito `boot`, quindi è lì che vedremo questi messaggi. La funzione `flog()` accetta un primo valore che indica il tipo di messaggio che si sta inviando (i possibili valori sono `LOG_DEBUG`, `LOG_INFO`, `LOG_WARN` e `LOG_ERR`). L'effetto è solo quello di cambiare le prime tre lettere della linea che viene inviata alla porta seriale, ma l'informazione potrebbe essere usata per filtrare i tipi di messaggi che si vuole o non si vuole vedere. Il secondo argomento è una stringa che verrà interpretata in modo simile a come opera la funzione `printf()` della libreria standard del C (la funzione si può usare anche in C++, includendo `<cstdio>`). La stringa rappresenta un modello per il messaggio che deve essere generato. Tutti i caratteri diversi da "`%`" verranno riprodotti così come sono, mentre i caratteri "`%`" rappresentano dei segnaposto per dei valori che devono essere inseriti in quel punto. Per ogni segnaposto deve essere passato un ulteriore parametro (in ordine, dopo la stringa) e deve essere specificato, dopo il carattere "`%`", in che modo il valore del parametro deve essere mostrato. Nel nostro caso usiamo il carattere "`p`" per specificare che vogliamo mostrare un puntatore (verrà stampato in esadecimale) e passiamo `rip` come parametro corrispondente. Altri caratteri possibili sono "`d`" per numeri da mostrare in base 10, "`x`" per numeri da mostrare in esadecimale, e "`s`" per stringhe di caratteri. I caratteri "`d`" e "`x`" richiedono parametri di tipo **`int`**, ma possono essere preceduti dal carattere "`l`" per usare parametri di tipo **`long`**.

La funzione `main()` abilita il Single Step (linea 15), esegue un po' di istruzioni, e poi lo disabilita (linea 20). Le funzioni di abilitazione e disabilitazione sono scritte in Assembler (Figura 4). Si noti che non ci sono istruzioni apposite per modificare il flag TF, quindi utilizziamo le istruzioni **`pushf`** per salvare il contenuto del registro dei flag in pila, manipoliamo il valore in cima alla pila, e infine lo ricarichiamo nel registro dei flag con l'istruzione **`popf`**.

Compilando e avviando il programma dovremmo vedere un output del genere sulla console (non sul monitor della macchina emulata):

```
DBG 0 rip=0000000000200190
DBG 0 rip=0000000000200194
DBG 0 rip=0000000000200198
DBG 0 rip=000000000020019c
DBG 0 rip=00000000002001a0
```

```
DBG 0 rip=00000000002001b8
DBG 0 rip=00000000002001b9
DBG 0 rip=00000000002001bf
DBG 0 rip=00000000002001c0
```

Se si va a guardare a cosa corrispondono i vari indirizzi (`objdump -dS`) si noteranno alcune cose.

- Il processore genera (o non genera) l'eccezione di debug alla fine di una istruzione, ma lo fa in base al valore che TF aveva subito prima di eseguirla. L'istruzione alla linea 6 di Figura 4 porta TF a 1, ma prima che iniziasse TF valeva 0. Quindi il processore non genera l'eccezione alla sua fine, ma solo alla fine dell'istruzione successiva, che parte quando TF è già 1. L'eccezione è di tipo trap, quindi l'instruction pointer salvato è quello dell'istruzione ancora successiva. Il primo **rip** stampato, dunque, dovrebbe essere quello della prima istruzione `x++` di `main()`. L'ultimo **rip** stampato dovrebbe essere invece quello dell'istruzione **ret** alla linea 13 di Figura 4: l'istruzione **popf** porta TF a 0, ma quando era partita TF era ancora 1, e dunque verrà generata ancora una eccezione di debug alla sua fine, salvando l'instruction pointer dell'istruzione successiva.

- Il processore non genera ulteriori eccezioni di Single Step mentre sono in esecuzione le funzioni `a_debug` e `c_debug`. Questo perché il flag TF viene automaticamente resettato ogni volta che si attraversa un gate della IDT. L'idea è che vogliamo eseguire passo passo un certo programma da debuggare, ma non vogliamo eseguire passo passo anche il debugger stesso. Il vecchio valore di TF viene salvato in pila, insieme a tutti gli altri flag, e ripristinato dalla **iretq** che termina la routine di eccezione. Si noti come, in base alla regola precedente, non ci sarà una nuova eccezione subito dopo la **iretq**, ma solo dopo l'istruzione successiva, che è quello che vogliamo.

Esercizio 1

L'istruzione **int3** può essere usata da un debugger per inserire un break-point in un punto del programma da debuggare. La codifica dell'istruzione **int3** in linguaggio macchina è 0xCC, su un solo byte. Il debugger può dunque sostituire il primo byte dell'istruzione a cui ci si vuole fermare con 0xCC, e poi restituire il controllo al programma. Il programma ora esegue liberamente, ma il processore genererà una eccezione di break-point se l'esecuzione arriva alla **int3** così inserita. L'eccezione sarà intercetta dal debugger, che così riacquisirà il controllo del processore e potrà chiedere ulteriori istruzioni all'utente.

Vogliamo modificare il `main` di Figura 5 in modo da inserire un break-point all'inizio della funzione `foo()` che faccia saltare ad una funzione che invii il messsagio "`breakpoint all'indirizzo` x" (dove x è l'indirizzo della prima istruzione di `foo`) su log. Dopo l'invio del messaggio l'esecuzione del programma deve riprendere normalmente.

```
1  #include <libce.h>
2  void foo() {
3          printf("foo()\n");
4  }
5
6  int main()
7  {
8          foo();
9          pause();
10         return 0;
11 }
```

Figura 5: Esercizio 1

Soluzione

La soluzione è in Figura 6. Alla riga 18 del file C++ associamo la funzione `a_debug` all'eccezione 3 (breakpoint), quindi sovrascriviamo il primo byte di `foo` con l'istruzione **`int3`** (riga 21). Per poter poi eseguire correttamente il programma, ci salviamo il byte che stiamo sostituendo (riga 20).

La chiamata a `foo()` (riga 23) causerà ora un salto alla `a_debug` e poi alla `c_debug()`. Questa invia il messaggio al log (riga 12) e poi ripristina il primo byte della `foo()` (riga 13). Si noti che **`int3`** è di tipo trap e dunque il processore salva in pila l'indirizzo dell'istruzione successiva. Per questo motivo dobbiamo sottrarre 1 a `rip` prima di usarlo (riga 11).

Sempre per questo motivo, la `a_debug` deve decrementare di 1 l'indirizzo in cima alla pila prima di eseguire la **`iretq`** che ritornerà al programma interrotto (riga 7 del file assembler). In questo modo possiamo eseguire l'istruzione che avevamo sostituito con **`int3`**.

Esercizio 2

Quando gdb raggiunge un breakpoint ridà il controllo all'utente. Se questo decide di continuare l'esecuzione (istruzione `c`), come fa gdb garantire che l'istruzione su cui era stato inserito il break-point venga ora eseguita, e allo stesso tempo che ci sia una nuova eccezione di break-point se il programma dovesse ripassare in seguito da quella stessa istruzione?

```
#include <libce.h>
void foo() {
        printf("foo()\n");
}

natb old;
extern "C" void a_debug();
extern "C" void c_debug(void *rip)
{
        natb *p = reinterpret_cast<natb*>(rip);
        p--;
        flog(LOG_DEBUG, "breakpoint all'indirizzo %p", p);
        *p = old;
}

int main()
{
        gate_init(3, a_debug);
        natb *p = reinterpret_cast<natb*>(foo);
        old = *p;
        *p = 0xCC;

        foo();
        pause();
        return 0;
}
```

```
#include "libce.s"
a_debug:
        salva_registri
        movq 120(%rsp), %rdi
        call c_debug
        carica_registri
        decq (%rsp)
        iretq
```

Figura 6: Soluzione esercizio 1