1 Lezione del 17-03-25

1.1 Protezione

Tutti i programmi che abbiamo visto finora hanno il pieno controllo su la macchina su cui sono in esecuzione. Questo significa che possono impattare qualsiasi regione di memoria, incluso il loro stesso codice macchina, o i frame di stack di programmi lanciati prima di loro.

Un approccio di questo tipo non è ideale quando più programmi, magari di utenti diversi, vengono lanciati ed eseguiti *quasi* in contemporanea (*time-sharing*) sulla stessa macchina.

Un esempio di questa situazione può verificarsi nel caso di esecuziono *batch*, cioè di esecuzione successiva di più programmi, magari scritti da più utenti. Vorremmo massimizzare l'uso della CPU sospendendo un programma e iniziandone un altro nel caso il primo fra questi inizi un'operazione che richiede una quantità significativa di tempo (ad esempio un accesso a un dispositivo di I/O). In questo caso, visto che non possiamo fidarci della benevolenza degli utenti nell'inserire istruzioni esplicite per il cambio da un programma all'altro, vorremo agire sull'hardware per, ad esempio, vietare all'utente l'uso di certe istruzioni (qui `IN` e `OUT`) e costringerlo ad usare primitive messe a disposizione dal sistema.

Chiaramente, però, le primitive dovranno poter usare `IN` e `OUT` per fare l'I/O vero e proprio con i dispositivi. Per permettere questo doppio comportamento introduciamo l'idea di **protezione**.

1.1.1 Contesti di esecuzione

Il programma nella memoria potrà essere in esecuzione, in un momento qualsiasi, in uno di due **contesti**, o *modi* (vedremo nell'architettura x86 corrente, si parla di protezione a *ring*): il contesto **sistema** e il contesto **utente**. Le istruzioni di cui permetterà l'esecuzione saranno quindi determinate dal contesto corrente.

Forniamo allora il processore di un apposito registro, il **CS** (*Code Segment*), a 2 bit. I 2 bit sono necessari in quanto storicamente (il meccanismo descritto viene introdotto nell'architettura x86 a partire dal 286) si definivano quattro contesti, o **ring**:

CS	Ring	Tipo
00	Ring 0	Kernel (sistema)
01	Ring 1	Driver
10	Ring 2	/ /
11	Ring 3	Utente

Il nome CS deriva dal fatto che questo registro era pensato per gestire la *segmentazione* della memoria. Sia questo meccanismo, che i due ring interni (l'1 e il 2) sono pressoché inutilizzati nell'architettura x86-64 moderna, e quindi li ignoreremo, portandoci effettivamente alla situazione dove CS rappresenta un flag che distingue fra contesto *sistema* e contesto *utente*, come avevamo ipotizzato.

1.1.2 Transizioni fra contesti

Ipotizziamo quindi che all'avvio si parta in contesto sistema, e che si passi al contesto utente quando si esegue un programma utente, Per permettere all'utente di "accedere"

alle istruzioni privilegiate, vogliamo che questo disponga di un modo di tornare al contesto sistema, ma lasciando il controllo al sistema operativo (altrimenti sarebbe inutile introdurre l'idea di un contesto utente in primo luogo). Di contro, vogliamo un modo per il sistema operativo di restituire in sicurezza il controllo al programma, previa transizione del processore in contesto utente.

Vediamo come il meccanismo dell'interruzione fornisce un metodo per gestire questa situazione.

Introdurremo un tipo di interruzione apposito, che restituisce il controllo al sistema operativo (semplicemente passando ad un gestore di interruzione definito dal sistema operativo) passando a contesto sistema. Il tipo di operazione che stiamo richedendo al sistema operativo potrà essere passato in qualche registro specifico, solitamente `%EAX`. Il problema potrebbe essere chiaramente che l'utente ha la possibilità di modificare tutta la memoria, e quindi la stessa IDT e il gestore impostato.

1.1.3 Protezione di memoria

Si rende quindi necessario un meccanismo di gestione degli accessi in memoria. In contesto utente, quindi, oltre a permettere l'utilizzo di solo alcune istruzioni *non privilegiate*, il processore dovrà permettere l'accesso solo a determinate regioni di memoria. Visto che non abbiamo ancora introdotto l'idea di *memoria virtuale*, modellizziamo temporaneamente questa configurazione con un apposito registro a controllo sistema che decide quali regioni di memoria sono o non sono accessibili.

Abbiamo quindi l'immagine completa del meccanismo della protezione, che avevamo introdotto per privilegiare le sole istruzioni, ma ci rendiamo adesso conto deve consistere in:

- Protezione delle **istruzioni** attraverso il loro privilegiamento al contesto sistema, come avevamo visto;
- Protezione della **memoria** definendo regioni accessibili in sola modalità sistema.

1.1.4 Transizione da contesto utente a contesto sistema

Vediamo nel dettaglio come si passa dal contesto utente al contesto sistema. Per questo sfrutteremo l'istruzione x86 `INT`, che permette di generare un interruzione software sulla base del tipo fornito come operando Si potrà quindi avere una chiamata a sistema del tipo:

```
1 mov $0x00, %eax # tipo chiamata
2 int $0x80       # chiamata sistema (per x86, in x96-64 esiste syscall)
```

Questo si tradurrà a livello processore nel salvataggio dello stato corrente di esecuzione, la transizione al contesto sistema e lo spostamento in IP della prima istruzione di un apposito sottoprogramma di servizio atto a gestire l'eccezione (e quindi soddisfare, se possibile, la richiesta del programma per cui questo ha sollevato in primo luogo l'interruzione).

Per capire nel dettaglio cosa accade nel processore è necessario:

- Capire come è strutturata la Interrupt Descriptor Table (IDT) all'interno della memoria del sistema, che supponiamo essere privilegiata (altrimenti l'utente potrebbe manometterla);

- Capire come viene gestita un interruzione software, cioè come si conserva lo stato al momento dell'interruzione, e come si inizia l'esecuzione del gestore in contesto sistema.

Vediamo questi dettagli in ordine.

1.1.5 Struttura della IDT

Vediamo quindi nel dettaglio la struttura di un entrata della IDT. Questa viene a trovarsi nella memoria privilegiata a partire da un indirizzo, come avevamo detto, contenuto nel registro IDTR. L'impostazione di questo registro si fa attraverso apposite istruzioni, sempre ad accesso privilegiato.

Le entrate dell'IDT si chiamano **gate IDT**, che si distinguono in 3 tipi, *Task Gate*, *Interrupt Gate* e *Trap Gate*, che al momento non vediamo. La struttura a livello di memoria contiene le seguenti informazioni:

- L'offset della routine di gestione dell'interruzione, in alcune modalità comprendente dell'indice di segmento, ecc...;
- **P:** un flag di **presenza**, indica se il descrittore è effettivamente abilitato;
- **L:** il livello di protezione (contesto sistema o utente) a cui deve essere eseguito il gestore;
- **I/T:** il tipo di interruzione fra quelli sopra definiti.

1.1.6 Gestione dell'interruzione software

A questo punto la chiamata di interrupt sta effettivamente nella transizione fra due **pile**: la separazione fra contesto utente e contesto sistema viene infatti resa possibile anche dalla presenza di due pile separate, di cui l'ultima chiaramente sta in memoria protetta. Il programma è normalmente in esecuzione nella pila utente: al momento del sollevamento di un interruzione software, si passa all'esecuzione (se alcune condizioni che vedremo fra poco sono rispettate) della routine di gestione definita dal sistema operativo. Questa imposta un nuovo frame sulla pila con i seguenti dati:

- L'instruction pointer **RIP**, da dove si vorrà ripartire nell'esecuzione una volta gestita l'interruzione. Notiamo che in verità questo indirizzo, che è fra l'altro in memoria virtuale, è corredato a seconda del tipo di gate dall'**SS** (*Stack Segment*) o dal **TSS** (*Task State Segment*), utili alla memoria segmentata che come abbiamo visto non ci è di interesse. La caratteristica importante è che si conserva un riferimento a dove ripartire, in memoria, nell'esecuzione una volta gestita l'interruzione;
- Il contenuto attuale di **CS**, cioè il contesto al momento della chiamata, che chiaramente vorremo ristabilire in seguito;
- Come abbiamo visto, anche **EFLAGS** viene memorizzato, in quanto gli interrupt mascherabili vengono mascherati in fase di gestione di un interrupt sistema (attraverso il flag **IF**), e vogliamo resettare questo comportamente al termine della gestione.

Un caso particolare ma permesso è rappresentato dalla situazione dove **L**, il livello di destinazione, corrisponde allo stato attuale (ad esempio, sono permesse chiamate di

interruzioni da contesto utente a contesto utente, o da contesto sistema a contesto sistema). In questo caso, chiaramente, tutta questa operazione verrà svolta su un unica pila (sia questa la pila utente o la pila sistema). Noteremo fra poco come questa possibilità rivela delle falle di sicurezza che vanno gestite.

1.1.7 Transizione da contesto sistema a contesto utente

La transizione inversa a quella vista adesso viene fatta semplicemente ritornando dall'interruzione attraverso la `IRETQ`. In questo caso si preleva dalla pila sistema (utente se eravamo in un interruzione a gestione livello utente) le informazioni che vi avevamo inserito al momento della chiamata dell'interruzione (RIP, CS ed EFLAGS) e si ristabilisce lo stato precedente al sollevamento dell'istruzione. Anche qui vi sono delle particolarità, che verranno spiegato, assieme a quelle annunciate in precedenza, nel paragrafo seguente.

1.1.8 Particolarità della gestione delle interruzioni software

Notiamo una particolarità riguardo alla transizione di contesto in fase di chiamata dell'interruzione (nota osservando il contesto attuale e il DPL dell'interruzione lanciata), e riguardo alla transizione di contesto in fase di ritorno dall'interruzione (nota osservando il contesto attuale e il contesto salvato in pila).

Infatti, in fase di chiamata (quando si usa la `INT`), se il DPL è minore del contesto corrente, viene lanciato un errore. La motivazione è principalmente una questione di simmetria nel meccanismo di chiamata delle interruzioni, piuttosto che una ragione di sicurezza: si vuole che le interruzioni ci portino in contesti maggiori o uguali del livello presente in CS.

Viceversa, se si prova a passare ad un livello superiore in fase di ritorno dall'interuzione (cioè quando si usa la `IRETQ`), viene lanciato un altro errore. La motivazione è che, visto che prevediamo nell'IDT il flag **L**, livello di destinazione, che permette di chiamare interruzioni in contesto utente, l'utente potrebbe impostare un frame di pila dove si richiede effettivamente l'accesso ad un livello di protezione superiore, e poi usare `IRETQ` per ritornare da tale frame di pila e passare quindi a tale livello di accesso.