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 introduciamo l'idea di **protezione**. Il programma nella memoria potrà essere in esecuzione, in un momento qualsiasi, in uno di due contesti: il contesto **sistema** e il contesto **utente**. Le istruzioni di cui permetterà l'esecuzione saranno quindi determinate dal contesto corrente.

Ipotizziamo quindi che all'avvio si parta in contesto sistema, e che con una determinata istruzione si passi al contesto 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).

Effettuiamo ciò introducendo 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.

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.

Notiamo poi che effettivamente abbiamo sottratto all'utente l'accesso alle interruzioni, in quanto gli neghiamo l'accesso alla IDT, e in caso di chiamata di un interruzione si porta direttamente il processore al contesto sistema.

Dal punto di vista dell'architettura che stiamo studiando, la chiamata di un interruzione può essere effettuata usando l'istruzione `INT`, che genra un interruzione software sulla base del tipo fornito come operando. Si potrà quindi avere una chiamata a sistema del tipo:

```
mov $0x00, %eax # tipo chiamata
int $0x80       # chiamata sistema
```

1.1.1 Struttura della IDT

Per comprendere il funzionamento del meccanismo di chiamata di interruzione, in particolare per le chiamate software che ci permettono di implementare la protezione come visto sopra, vediamo nel dettaglio la struttura di un entrata della IDT.

tabella idt (su osdev c'è tutto tranne L, dovrebbe essere il livello d'arrivo)

discorso pila utente pila sistema

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 contesto corrente è maggiore del DPL, viene lanciato un errore. La motivazione è principalmente una questione di simmetria nel meccanismo di chiamata delle interruzioni, piuttosto che una ragione di sicurezza: capito poco

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 destinazinoe, che permette di chiamare interruzioni in contesto utente (e di cui non hai ancora parlato), 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.

appunti così introdotto nel 286 TSS: task segment (per switching di task lv. hardware) SS: segment (non vengono più usati nell'architettura AMD, usa la paginazione)

DPL: livello richiesto per chiamare l'interrupt L: livello di destinazione dell'interrupt P: indica se quell interrupt c'è o non ce

poi il tipo di gate (non bastava la posizione nella tabella ? chiarisci)