Protezione

G. Lettieri

21 Aprile 2018

Per capire perché abbiamo bisogno del meccanismo della protezione, partiamo da un esempio. Supponiamo di trovarci negli anni '50 o '60 del secolo scorso, quando un computer occupava l'area di una palestra ed una Università poteva averne uno o forse due. In questo periodo i computer venivano usati in modalità *batch*. Gli utenti—ricercatori o studenti—preparavano i programmi a casa, su fogli di carta, scrivendoli in linguaggio macchina o in FORTRAN. Portavano poi i loro fogli al centro di calcolo, dove alcuni impiegati potevano trascriverli su schede perforate. Ogni pacco di schede, contenente il programma di un utente, rappresentava un *job*. L'utente consegnava poi il suo job agli operatori del computer; in un secondo momento sarebbe dovuto ritornare a ritirare i risultati, tipicamente sotto forma di un tabulato stampato su carta.

Gli operatori, gli unici ad avere accesso alla sala del computer, aspettavano di avere un mazzo (*batch*) di job e poi lo caricavano sul lettore di schede del computer. Questo eseguiva i job uno alla volta, caricando automaticamente il prossimo job dopo aver terminato il precedente.

In questi sistemi si voleva massimizzare il numero di job completati ogni ora, sfruttando il costosissimo processore nel modo più efficiente possibile.

Supponiamo ora che il job dell'utente 1 debba caricare una lunga serie di dati da un nastro magnetico e decida di svolgere questa operazione a controllo di programma. Il costosissimo processore verrà così sprecato in un banale ciclo di istruzioni che legge ripetutamente i registri del controllore del nastro, per ricevere i dati e copiarli in memoria. La "vera" elaborazione del job 1, quella che ha davvero bisogno delle piene capacità della CPU, comincerà solo quando tutti i dati saranno stati trasferiti. Per gli operatori del computer sarebbe molto meglio se il controllore del nastro fosse programmato per una operazione in DMA. In questo modo, mentre i dati vengono trasferiti in memoria, si potrebbe utilizzare la CPU per cominciare ad eseguire il prossimo job del batch. Il controllore segnalerebbe poi il termine dell'operazione di trasferimento con una richiesta di interruzione. All'arrivo della richiesta si potrebbe ritornare al job 1.

Come realizziamo questo schema? Al di là di come si faccia a saltare da un job ad un altro, il problema è che non possiamo attenderci collaborazione da parte degli utenti stessi: l'utente del job 1 non ha interesse a cedere la CPU ad un altro job, e l'utente del job 2 non ha interesse a ridarla all'utente 1 quando arriva l'interruzione. Purtroppo, però, mentre è in esecuzione un certo programma, la CPU obbedisce a *quel* programma e dunque, nel nostro

caso, al volere degli utenti 1 e 2 e non al volere degli operatori. Gli operatori possono scrivere le necessarie routine: una per programmare il lettore del nastro in DMA e passare al prossimo job; un'altra per rispondere alla richiesta di interruzione ritornando al job precedente. Ma niente può costringere gli utenti ad usare queste routine. Il primo utente può non chiamare la prima routine degli operatori, scrivendo e leggendo direttamente dai registri del controllore. Il secondo utente può disabilitare le interruzioni, in modo che la seconda routine degli operatori non vada in esecuzione.

Abbiamo bisogno di un meccanismo aggiuntivo nel processore, in modo che questo non obbedisca sempre ciecamente alle istruzioni del programma corrente. Paragonando un programma ad un *testo* che il processore legge (ed esegue), aggiungiamo la possibilità di avere anche un *contesto*. Ad ogni istante ci sarà un contesto corrente, che determina come le istruzioni del programma corrente devono essere interpretate. Potremo avere un *contesto privilegiato*, nel quale verranno eseguiti i programmi degli operatori, e un contesto non privilegiato, o *utente*, in cui verranno eseguiti i programmi degli utenti. Quando il processore si trova nel contesto privilegiato obbedisce a tutte le istruzioni, come siamo abituati. Ma, quando si trova nel contesto non privilegiato, si rifiuta di eseguire le istruzioni che abilitano o disabilitano le interruzioni e tutte le istruzioni che leggono o scrivono nello spazio di I/O. Una volta aggiunto il meccanismo dei contesti, gli operatori devono fare in modo che il processore si trovi sempre nel contesto non privilegiato mentre sta eseguendo i job degli utenti. In questo modo gli utenti non possono più attuare le strategie di cui sopra.

Non dobbiamo però dimenticare che il computer è stato acquistato per servire gli utenti, non gli operatori: dobbiamo fornire all'utente 1 un modo per leggere i suoi dati dal nastro. L'utente 1 può farlo, ma non ha altra scelta se non chiamare la routine fornitagli dagli operatori. Dovrà però farlo utilizzando una istruzione speciale che, oltre a saltare alla routine, porti anche il processore nel contesto priviligiato, altrimenti nemmeno la routine degli operatori potrebbe interagire con il controllore del nastro. Oltre al meccanismo dei contesti, quindi, dobbiamo aggiungere al processore un meccanismo per passare da un contesto ad un altro. Questo meccanismo dovrà essere a sua volta protetto, in modo che gli utenti non possano indurre la CPU a portarsi nel contesto privilegiato e poi eseguire codice scritto da loro stessi invece che dagli operatori.

C'è un'altra cosa a cui stare attenti. Ora in memoria possiamo avere più programmi: quello scritto dagli operatori (contenente almeno le routine per i trasferimenti dal nastro) e quelli scritti dagli utenti. Nella CPU che abbiamo visto fino ad ora un programma può accedere liberamente a tutte le locazioni di memoria. Se però un utente potesse modificare le routine degli operatori, ovviamente tutto lo schema di protezione non avrebbe alcun effetto. Dobbiamo quindi anche fare in modo che, mentre è attivo il contesto non privilegiato, non sia possibile accedere alle locazioni di memoria che contengono le routine e le strutture dati degli operatori. Inoltre, ricordandoci che il computer serve agli utenti e non agli operatori, è ancora più importante garantire che il programma di un utente non possa modificare il programma di un altro utente. Possiamo realizzare questo ulteriore requisito prevedendo di avere tanti diversi contesti

non privilegiati—per esempio, uno per ogni job. Il contesto di ogni job contiene solo il codice e i dati di quel job, ma non quelli degli altri job. Quando si passa da un job ad un altro si deve quindi cambiare anche il contesto corrente e fare in modo che, mentre è attivo un dato contesto, non sia possibile accedere alla memoria che contiene gli altri job.

Si noti che l'esempio dei sistemi batch è stato scelto perché particolarmente semplice. La necessità di poter portare avanti più programmi contemporaneamente nasce però in diversi ambiti. Ormai è data per scontata in tutti i sistemi che vanno dai supercalcolatori, ai server, ai personal computer, agli smartphone/tablet fino anche a molti sistemi embedded, con la sola eccezione dei più semplici tra questi ultimi. Il termine *job* era in uso nei sistemi batch, mentre in seguito useremo il concetto più generale di *processo*: un processo è un programma in esecuzione. Il fatto che i processi debbano appartenere a diversi utenti non è fondamentale: in molte applicazioni uno stesso utente può voler eseguire più di un processo contemporaneamente e, anche in quel caso, vogliamo che ogni processo abbia un suo contesto indipendente (per fare in modo, per esempio, che i bug di un processo non si propaghino in altri processi). Se però uno stesso utente esegue più processi, può aver senso che questi lavorino su qualche struttura dati in comune, quindi i loro contesti potrebbero anche condividere in tutto o in parte la memoria.

Riassumendo, abbiamo bisogno di:

1. un modo per definire dei contesti, privilegiati e non;
2. un modo per stabilire quale contesto è quello corrente;
3. delle regole che dicano cosa si può fare o non si può fare in ogni contesto;
4. un modo per passare da un contesto ad un altro (cambiare contesto).

Cambiare contesto per innalzare o abbassare il livello di privilegio è anche detto "cambio di livello". Cambiare contesto per passare da un processo ad un altro è anche detto "cambio di processo", ed è in genere una operazione molto più costosa del semplice cambio di livello.

1 La protezione nei processori Intel/AMD a 64 bit

Il meccanismo della protezione è stato aggiunto dall'Intel nel processore 80286 (1982), che era a 16 bit. Il meccanismo era strettamente intrecciato con un altro meccanismo, la segmentazione, che però non trovò molti utilizzi. Il meccanismo di protezione nell'80286 era molto sofisticato: i "livelli di privilegio" non erano soltanto due (privilegiato e non privilegiato), ma quattro; inoltre, il processore poteva eseguire interamente in hardware non solo il cambio di livello, ma anche il cambio di processo.

Con il processore 80386 (1985) l'Intel passò a 32 bit e supportò anche la paginazione, ma sempre in aggiunta alla segmentazione, estesa a 32 bit. Il meccanismo di paginazione dell'80386, però, supportava (e continua a supportare nei processori moderni) solo due livelli di privilegio: *sistema* (privilegiato) e *utente* (non privilegiato). I processori successivi (80486, Pentium, Pentium Pro, ...) non hanno cambiato questa architettura di base, ma né la segmentazione, né il meccanismo di cambio di processo via hardware sono mai stati usati in modo significativo.

Quando l'AMD ha introdotto l'architettura a 64 bit ha rimosso il cambio di processo in hardware (che possiamo dunque ignorare) e *quasi* completamente disattivato il supporto alla segmentazione. Purtroppo, sempre per motivi di compatibilità, le cose sono rimaste molto più complicate di come avrebbero potuto essere. Nel seguito cercheremo di semplificare la descrizione nascondendo il più possibile i riferimenti alla segmentazione; i dettagli si potranno trovare nel codice eseguibile del sistema che realizzeremo.

1.1 Livelli di privilegio

Il processore si trova ad ogni istante o a livello (di privilegio) sistema, o a livello utente. Il livello corrente è deciso da un campo nel registro `CPL`.

Per quanto riguarda la protezione della memoria, facciamo subito una semplificazione (che poi rimuoveremo quando parleremo della paginazione). Immaginiamo che il processore abbia un registro, in cui si può scrivere solo da livello sistema, che contenga un indirizzo *limite* che faccia da spartiacque tra la memoria usata dal sistema e quella utilizzabile dagli utenti. Chiamiamo M1 la parte di memoria ad indirizzi inferiori al limite e M2 la rimanente. Quando il processore si trova a livello utente sono vietati tutti gli accessi alle locazioni di M1, mentre a livello sistema il limite viene ignorato (tutti gli indirizzi sono permessi). All'accensione il processore si trova a livello sistema e carica ed esegue il boostrap loader (da una ROM). Il boostrap loader carica le routine e le strutture dati del sistema all'inizio della memoria, quindi scrive nel registro limite l'indirizzo della prima locazione non utilizzata, definendo così la separazione tra M1 e M2.

Il cambio di livello è strettamente connesso con il meccanismo delle interruzioni. Infatti, il livello di privilegio (e dunque il contenuto di `CPL`) può essere cambiato solo in due modi:

- innalzato (o lasciato inalterato) passando attraverso un gate della IDT;
- abbassato (o lasciato inalterato) tramite una istruzione `iretq`.

Il termine *gate* (cancello) ci deve proprio far pensare ad un cancello che permette di entrare nel territorio privilegiato del sistema. Ci sono tre modi per attraversare un cancello. Due li conosciamo già: interruzioni esterne ed eccezioni. Il terzo lo introdurremo a breve.

Se torniamo con la mente all'esempio di partenza, vediamo che ha senso che la ricezione di una interruzione o il sollevarsi di una eccezione causino un salto ad una routine di sistema. Nell'esempio, la conclusione del trasferimento

dati era segnalata da una interruzione. In risposta a questa vogliamo che vada in esecuzione una routine di sistema che faccia ripartire il job 1. Anche per le eccezioni dovrebbe essere chiaro cosa vogliamo: se un utente prova a disabilitare le interruzioni, a leggere o scrivere nei registri del controllore del nastro, o a leggere o scrivere nelle locazioni di M1, vogliamo che venga fermato. Un modo per fermarlo è che il processore, sapendo che queste operazioni sono vietate a livello utente, sollevi una "eccezione di protezione". In risposta all'eccezione vogliamo che vada in esecuzione un'altra routine del sistema, che termini il job corrente e ne metta in esecuzione un altro. Si capisce anche perché deve essere la `iretq`, che si trova in fondo a tutte le routine di risposta alle interruzioni ed eccezioni, l'istruzione che si preoccupa di riportare il processore a livello utente.

Affinché questo meccanismo sia efficace, gli utenti non devono essere in grado di modificare il contenuto della IDT. Questo si ottiene rendendo privilegiata l'istruzione `LIDT` (che carica l'indirizzo della IDT nel registro `IDTR` che il processore usa per accedere alla tabella) e allocando la IDT nella memoria M1. Questo può essere fatto in fase di inizializzazione del sistema: la IDT è semplicemente una delle strutture dati del sistema, caricate in M1 dal bootstrap loader. Dopo il caricamento, il bootstrap loader salta ad una ben precisa routine del sistema appena caricato, la quale si preoccupa di inizializzare tutte le strutture dati di sistema (compresa la IDT) e il processore, in particolare caricando il registro `IDTR` con l'indirizzo della IDT.

Ora, se torniamo ancora una volta all'esempio, ci rendiamo conto che abbiamo bisogno di un terzo modo per attraversare i cancelli: l'utente 1 deve poter invocare una routine di sistema quando ha bisogno di caricare i dati dal nastro. Per far questo, il processore prevede una nuova istruzione:

`INT` $*tipo*.

Questa istruzione ha un unico parametro immediato, *tipo*, che deve essere un numero tra 0 e 255. Il tipo ha lo stesso significato del tipo delle interruzioni esterne e delle eccezioni: è utilizzato per accedere ad un gate della IDT, dove il processore trova l'indirizzo della routine a cui saltare e l'informazione che gli dice se il livello di privilegio deve essere innalzato. Si dice che l'istruzione genera una "interruzione software". Le interruzioni software sono del tutto analoghe a quelle esterne ed alle eccezioni di tipo trap, con salvataggio in pila di informazioni analoghe. In particolare, l'instruction pointer salvato in pila è quello dell'istruzione successiva alla `INT`. Anche le routine di risposta alle interruzioni software devono terminare con `iretq` e non `ret`.

Le routine che vanno in esecuzione tramite una `INT` prendono comunemente il nome di *primitive di sistema*. Chi scrive il codice di sistema deve fornire ai suoi utenti la necessaria documentazione su: quali sono sono le primitive disponibili; quale tipo è necessario usare per invocare ciascuna primitiva; quali parametri ciascuna primitiva si aspetta e come le vanno passati (tipicamente tramite i registri). L'istruzione `INT` svolge un compito simile a quello di una `CALL`. Ci si può chiedere come mai serva una nuova istruzione per invocare una primitiva, invece di aggiungere la possibilità di innalzare il livello di privilegio

alla normale `CALL`. Ma l'istruzione `CALL` specifica direttamente l'indirizzo a cui saltare, e questo causa due problemi:

1. l'utente potrebbe saltare in mezzo al codice delle primitive (per esempio, per scavalcare dei controlli);
2. se gli indirizzi vengono specificati in forma simbolica, i programmi utente dovrebbero essere collegati con il codice di sistema per poter risolvere i simboli.

Il primo problema è di gran lunga più grave, mentre il secondo è solo una scomodità. L'istruzione `INT` non ha questi problemi, in quanto l'utente specifica solo il tipo della primitiva che intende invocare, mentre l'indirizzo a cui saltare è scritto nella IDT, a cui lui non ha accesso. C'è un ultimo vantaggio, che per il momento non possiamo apprezzare pienamente: è molto comodo, per chi scrive il sistema, che i modi con cui si può accedere al sistema siano il più possibile uniformi.

1.2 Interruzioni e protezione

Vediamo più in dettaglio come funziona il meccanismo delle interruzioni. Ogni gate della IDT occupa 16 byte e contiene le seguenti informazioni:

- un bit P (Presenza), che indica se il gate contiene informazioni significative (il sistema non deve necessariamente utilizzare tutti i gate);
- il puntatore alla routine a cui saltare (8 byte);
- un bit I/T che indica se il gate è di Interrupt o Trap;
- un campo L che indica il livello di privilegio a cui il processore si deve portare dopo aver attraversato il gate;
- un campo DPL (Descriptor Privilege Level, chiamato anche GL per Gate Level) che indica il livello di privilegio minimo che il processore deve avere per poter attraversare il gate nel caso di interruzione software.

Quando il processore accetta una interruzione esterna, genera una eccezione o esegue una interruzione software,

1. si procura il *tipo* dell'interruzione:
   - in caso di eccezione, il tipo è implicito;
   - in caso di interruzione esterna, riceve il tipo dall'APIC;
   - in caso di interruzione software, il tipo è il parametro dell'istruzione `INT`.

   Usa quindi il tipo come indice nella tabella IDT, per accedere al corrispondente gate.

2. Se il bit P del gate è zero, il processore smette di gestire l'interruzione e genera una eccezione per "gate non presente" (tipo 11).

3. Altrimenti, se sta gestendo una interruzione software (o eseguendo una `int3`), confronta il livello corrente (registro `CPL`) con il campo DPL del gate. Se il livello corrente è meno privilegiato di DPL, genera una eccezione di protezione (tipo 13).

4. Altrimenti, confronta il CPL con il campo L. Se L è inferiore, genera una eccezione di protezione (tipo 13).

5. Altrimenti, salva in un registro di appoggio (chiamiamolo `sRSP`) il contenuto corrente di `RSP`.

6. Se CPL è diverso da L, esegue un *cambio di pila*, caricando un nuovo valore in `RSP` (si veda la Sezione 1.3 più avanti per sapere da dove viene prelevato il nuovo valore).

7. Salva in pila (la nuova o la vecchia, a seconda che al punto precedente abbia cambiato pila o meno) 5 parole lunghe (8 byte ciascuna):

   - una parola lunga non significativa (segmentazione ...);
   - il contenuto di `sRSP` (questo punta alla vecchia pila, nel caso di cambio pila);
   - il contenuto di `RFLAGS`;
   - il contenuto di `CPL`;
   - il contenuto di `RIP`.

8. Azzera in ogni caso TF (disabilita una eventuale modalità Single Step) e azzera IF (Interrupt Flag) solo se il gate è di tipo Interrupt (campo I/T del gate).

9. Salta infine all'indirizzo della routine puntata dal gate.

Il controllo eseguito al punto 3 può essere usato per imporre che l'utente usi l'istruzione `INT` solo con i tipi associati a primitive di sistema, e non con i tipi associati alle interruzioni esterne o alle eccezioni. I programmatori di sistema possono impostare DPL=sistema in tutti i gate delle interruzioni esterne e delle eccezioni, e DPL=utente in quelli che puntano alle primitive. In questo modo le interruzioni esterne e le eccezioni vengono gestite normalmente (perché DPL non viene controllato in questi casi), mentre ogni tentativo dell'utente di usare un gate che non è assegnato ad una primitiva genera una eccezione di protezione.

Il controllo effettuato al punto 4 corrisponde a quanto si era detto: le interruzioni devono solo poter innalzare il livello di privilegio (o lasciarlo inalterato) e mai abbassarlo. Il punto è che una richiesta di interruzione manda in esecuzione una routine e, normalmente, ci aspettiamo che questa routine faccia ciò che deve fare e poi ritorni al programma principale. Ma se la routine è di livello utente non possiamo fare nessuna assunzione, nemmeno sul fatto che

prima o poi termini. Quindi, se il processore si trova a livello sistema (e dunque sta svolgendo operazioni importanti), è pericoloso che venga interrotto da una routine di livello utente. È comunque abbastanza insolito che un gate di interruzione contenga L=utente e noi assumeremo che non avvenga, rendendo dunque ridondante questo controllo.

Si noti che il meccanismo contempla anche i casi CPL=utente/L=utente e CPL=sistema/L=sistema. Possiamo ignorare quello con L=utente, per quando appena detto. Il caso CPL=sistema/L=sistema si può invece facilmente verificare: se il sistema invoca esso stesso una primitiva, o se una routine di sistema di tipo trap viene interrotta da una eccezione o da una interruzione esterna. In questi casi non ci sarà cambio pila.

Il cambio pila eseguito al punto 6 in caso di innalzamento del livello di privilegio (da utente a sistema) ha due motivazioni:

- il processore deve garantire di poter scrivere le 5 parole lunghe senza sovrascrivere altre cose, e non può dunque fidarsi del contenuto di `RSP` controllato dall'utente;
- è bene che queste informazioni siano salvate nella memoria di sistema (M1), in modo che l'utente non le possa corrompere. In particolare, è bene che l'utente non possa modificare il valore salvato di `CPL`, che dice a che livello si trovava il processore prima dell'interruzione.

L'istruzione `iretq` svolge le operazioni opposte a quelle del meccanismo di interruzione.

a. Confronta il valore corrente di `CPL` con quello salvato in pila; se quello salvato è più privilegiato di quello corrente genera una eccezione di protezione (tipo 13).

b. Ripristina i valori di `RIP`, `CPL`, `RFLAGS` e `RSP` leggendo i corrispondenti valori dalla pila.

Si noti che con il ripristino di `RSP` processore ritorna ad usare la pila originaria, nel caso questa fosse stata cambiata all'avvio dell'interruzione. Il ripristino di `RFLAGS` ripristina in particolare i vecchi valori di TF e IF, eventualmente riabilitando il Single Step e le interruzioni esterne. Il ripristino di `CPL` riporta il processore al livello di privilegio precedente.

Il controllo eseguito al punto a corrisponde a quanto detto in precedenza: l'istruzione `iretq` può solo abbassare (o lasciare inalterato) il livello di privilegio, mai innalzarlo. Diversamente dal complementare controllo nel caso delle interruzioni (punto 4), questo controllo è molto importante. Se non venisse effettuato, sarebbe facile per l'utente eseguire codice di suo piacimento a livello sistema (come?).

Si noti che la routine di inizializzazione del sistema deve avere un modo per poter passare a livello utente una volta che tutte le strutture dati sono state inizializzate. Per farlo è sufficiente che prepari una pila nello stesso stato in cui l'avrebbe lasciata una interruzione proveniente dal livello utente, ed esegua

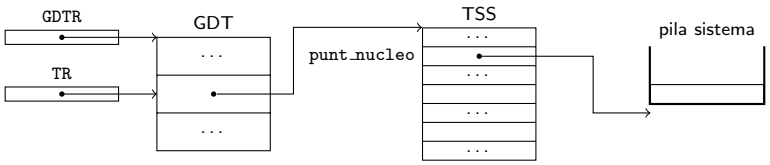

Figura 1: Strutture dati del sistema associate ad ogni processo.

una `iretq`. Da quel momento in poi il controllo passa agli utenti. Il sistema interviene solo in risposta alle richieste di interruzione esterne, le eccezioni, e le interruzioni software.

1.3 Il registro `TR` e i Task State Segment

Al punto 6 della Sezione precedente, abbiamo visto che il processore cambia pila quando deve innalzare il livello di privilegio. In puntatore alla nuova pila è prelevato dal *Task State Segment* (TSS) corrente. I TSS sono delle strutture in memoria contenenti spazio per diversi campi, tra cui il puntatore alla pila che ci interessa. Erano usati dal meccanismo di cambio di processo in hardware, che, come abbiamo visto, non è più disponibile. Devono però essere mantenuti, per compatibilità con il fondamentale meccanismo delle interruzioni. Chiameremo `punt_nucleo` il campo che punta alla pila di livello sistema.

Nel sistema ci possono essere più TSS (l'idea è di averne uno per ogni processo) ma, ad ogni istante, uno solo è quello corrente, individuato dal registro `TR` (Task Register). Tutti i TSS esistenti sono descritti da una tabella GDT (Global Descriptor Table), puntata dal registro `GTDR` che può essere caricato con l'istruzione `LGDT`. La GTD contiene alcune entrate che non ci interessano (segmentazione...) e una entrata per ogni TSS. Le entrate relative ai TSS contengono l'indirizzo di partenza (base) del TSS e la sua grandezza in byte. Il registro `TR` contiene l'offset (rispetto all'inizio della GDT) dell'entrata della GDT che punta al TSS corrente. Si veda la Figura 1.

Si noti che l'idea originaria era di avere dunque una diversa pila sistema per ogni processo. Anche noi faremo così e, dunque, dovremo prevedere un diverso TSS per ogni processo e dovremo aggiornare `TR` ad ogni cambio di processo, tramite l'istruzione `LTR`.

Ovviamente, le istruzioni `LGDT` e `LTR` sono privilegiate e la GDT e tutti i TSS devono trovarsi in M1.

1.4 Livelli di privilegio e operazioni di I/O

Siamo partiti da un esempio in cui volevamo vietare agli utenti di abilitare/disabilitare le interruzioni e accedere allo spazio di I/O. Nell'architettura Intel queste operazioni non sono automaticamente vietate quando il processore si trova a livello utente. La possibilità o meno di eseguire queste operazioni è determinata, invece, dal campo IOPL (I/O Privilege Level) che si trova nel

registro dei flag. Questo campo specifica il livello di privilegio minimo che il processore deve avere per poter abilitare o disabilitare le interruzioni ed eseguire le istruzioni `IN` e `OUT` (in tutte le loro varianti). Il campo IOPL può essere modificato solo a livello sistema. Il tentativo di modificarli da livello utente (per esempio tramite `pushf`/`popf`) non genera, come ci potremmo aspettare, una eccezione di protezione, ma viene semplicemente ignorato: IOPL continua a mantenere il suo valore senza che venga segnalato alcun errore.