Protezione

G. Lettieri

24 Marzo 2024

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[1]. 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 nastro o 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 serie di dati da un nastro magnetico e decida di svolgere questa operazione a controllo di programma. Il nastro va però riavvolto, operazione che richiede diversi secondi. Il costosissimo processore verrà così sprecato in un banale ciclo di istruzioni che legge ripetutamente i registri del controllore del nastro, per attendere che il nastro raggiunga la posizione desiderata. La "vera" elaborazione del job 1, quella che ha davvero bisogno delle piene capacità della CPU, comincerà solo quando tutti i dati saranno stati letti. Per gli operatori del computer sarebbe molto meglio se il controllore del nastro fosse programmato per inviare una richiesta di interruzione dopo il riavvolgimento. In questo modo, durante il tempo di riavvolgimento, si potrebbe utilizzare la CPU per cominciare ad eseguire il prossimo job del batch. All'arrivo della richiesta di interruzione si potrebbe poi ritornare al job 1.

[1] L'Università di Pisa era una delle poche ad averne due: la CEP (Calcolatrice Elettronica Pisana), il primo calcolatore scientifico interamente progettato e costruito in Italia, ora custodita al Museo degli Strumenti per il Calcolo, e l'IBM 7090, frutto di una donazione dell'IBM stessa.

Si noti come l'esempio sia per certi versi simile a quello che abbiamo usato per introdurre il meccanismo delle interruzioni, in quanto in entrambi i casi vorremmo usare l'unica CPU per portare avanti concorrentemente due compiti: nell'esempio di Dijkstra i due compiti erano la stampa e il calcolo, in questo esempio sono i job 1 e 2. C'è però una differenza cruciale: questa volta i due compiti arrivano da due utenti diversi, che non si conoscono fra loro. Questo vuol dire che non possiamo attenderci collaborazione: 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.

Per focalizzare le idee, supponiamo che gli operatori abbiamo scritto due routine e le abbiano caricate in memoria, insieme ai job degli utenti:

1. Routine 1: avvia il riavvolgimento del nastro e cede il controllo ad un altro job;

2. Routine 2: (da associare alle richieste di interruzione provenienti dal controllore del nastro) restituisce il controllo al job che aveva invocato la Routine 1.

Il problema che gli operatori hanno è: come costringere il primo utente a chiamare la Routine 1, invece di dialogare direttamente con il controllore del nastro; come impedire al secondo utente di disabilitare le richieste di interruzione. Ma non basta: ora in memoria possiamo avere più programmi: quello scritto dagli operatori (contenente almeno le Routine 1 e 2 descritte qui sopra) e quello scritto dall'utente il cui job è attualmente in esecuzione. Nella CPU che abbiamo visto fino ad ora, quando un programma è in esecuzione, può accedere liberamente a tutte le locazioni di memoria. Cosa impedisce al programma dell'utente di sovrascrivere le Routine 1 e 2, o di modificare il contenuto della tabella IDT per impedire che la Routine 2 vada in esecuzione?

Abbiamo bisogno di un meccanismo aggiuntivo nel processore, in modo che questo non obbedisca *sempre* ciecamente alle istruzioni del programma corrente: deve farlo se il programma che sta eseguendo è stato scritto dagli operatori, ma se il programma è stato scritto dagli utenti deve quantomeno "rifiutarsi" di eseguire le istruzioni di I/O e le istruzioni che abilitano o disabilitano le interruzioni, o accedono agli indirizzi contenenenti codice o dati degli operatori. Le istruzioni in sè, però, non hanno "colore": come può il processore distinguere, per esempio, l'istruzione **`in`** di un operatore dall'identica istruzione **`in`** di un utente? Se paragoniamo un programma ad un *testo* che il processore legge (ed esegue), il problema che abbiamo è quello di stabilire quale sia il *contesto*. Con questa parola ci si riferisce, nel linguaggio comune come in quello tecnico che ci riguarda, a tutto ciò che è necessario sapere per interpretatare correttamente un dato testo, ma che non è scritto esplicitamente nel testo stesso. Nel nostro caso il testo è l'istruzione che il processore sta eseguendo. Ad ogni istante ci deve essere anche un *contesto corrente*, che determini come l'istruzione debba essere interpretata. 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. Il processore deve solo sapere qual è il contesto corrente: se è quello privilegiato deve obbedire a tutte le istruzioni, come siamo abituati, ma se il contesto corrente è quello non privilegiato, si deve rifiutare di eseguire le istruzioni che abilitano o disabilitano le interruzioni e tutte le istruzioni che leggono o scrivono nello spazio di I/O. Il vincolo sulle istruzioni che accedono alle routine o alle strutture dati degli operatori è più complicato perché si applica a qualunque istruzione che abbia operandi in memoria, ma può essere semplificato nel modo seguente: si stabilisce *a-priori* che certi indirizzi, per esempio, tutti quelli inferiori ad un valore fissato, siano riservati agli operatori. Nel contesto non privilegiato il processore confronterà l'indirizzo di ogni operando con il valore fissato, riutandosi di eseguire l'istruzione se lo trova inferiore.

Per fare in modo che il processore sappia quale è il contesto corrente è sufficiente che abbia un nuovo registro in cui questa informazione è memorizzata (nel nostro caso basta un bit). Oltre ad aggiungere questo registro e i controlli durante la decodifica ed esecuzione di ogni istruzione, modifichiamo il processore in modo che all'avvio si trovi nel contesto privilegiato, e prevediamo una istruzione che lo porti nel contesto non privilegiato. Gli operatori sono gli unici ad essere presenti, la mattina, quando l'elaboratore viene avviato, e dunque possono fare in modo che il loro programma venga caricato agli indirizzi riservati e sia il primo a partire. Questo programma inizializzerà tutto il necessario, caricherà il primo job utente, eseguirà l'istruzione che porta il processore nel contesto non privilegiato e poi cederà il controllo al job utente. Da questo momento in poi l'esecuzione del programma utente sarà soggetta a tutti i vincoli che abbiamo visto.

Ora, però, che ci serve anche un meccanismo con cui sia possibile riportare occasionalmente il processore nel contesto privilegiato. Per capire perché, ricordiamoci che il computer è stato acquistato per servire gli utenti, non gli operatori: gli operatori devono fornire all'utente 1 un modo per riavvolgere il nastro e leggere i suoi dati. L'idea è che l'utente 1 debba farlo esclusivamente invocando la Routine 1 scritta dagli operatori. Ma, ricordiamo, le istruzioni (e dunque le routine) non hanno colore. Dobbiamo aggiungere una istruzione speciale che, oltre a saltare alla Routine 1, porti anche il processore nel contesto privilegiato, altrimenti nemmeno la Routine 1 potrebbe interagire con il controllore del nastro. 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: l'innalzamento di privilegio deve essere associato ad un trasferimento di controllo ad un indirizzo scelto dagli operatori e non dagli utenti. Come fare? Nel nostro esempio, questo ritorno al contesto privilegiato deve avvenire in almeno due occasioni:

1. Nel job 1, quando l'utente 1 invoca la Routine 1;
2. Nel job 2, quando il nastro ha finito di riavvolgersi e invia la richiesta di interruzione che manda in esecuzione la Routine 2.

Il secondo caso ci fornisce l'idea decisiva: legare strettamente l'innalzamento del livello di privilegio con la risposta alle richieste di interruzione e alle eccezioni. Se gli operatori caricano la tabella delle interruzioni nella parte di memoria inaccessibile agli utenti, le destinazioni dei salti in risposta alle richieste di interruzione e alle eccezioni non è sotto il controllo degli utenti. Il secondo caso è così automaticamente risolto. Per il primo caso, possiamo ricorrere al seguente trucco: gli operatori associano la Routine 1 ad una entrata della tabella delle interruzioni e dicono all'utente 1 che, per invocarla, deve volontariamente causare la corrispondente eccezione. Infine, abbiamo una risposta naturale a cosa dovrebbe fare il processore quando si "rifiuta" di eseguire una istruzione: solleva una eccezione. Gli operatori assoceranno a questa eccezione un programma che stampa un messaggio di errore e carica un altro job dal batch.

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.

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. Oltre ad essere legato al meccanismo delle interruzioni, 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.

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 *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 **`cs`** (Code Selector).

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 **`cs`**) 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.

Come abbiamo visto nell'esemoio, 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 riavvolgimento del nastro 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 **`lidtr`** (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 (`start`) 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.

Sempre nell'esempio ci siamo resi conto che l'utente 1 deve avere un modo per invocare la routine che avvia l'operazione sul nastro. Riusare il meccanismo delle eccezioni, come suggerito nell'esempio, è possibile(ed è anche la soluzione usata in qualche vecchio processore, ma il processore Intel offre un modo più esplicito, sotto forma di 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"—il terzo, e ultimo, modo in cui si può attraversare un gate della IDT. 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 **`cs`**) 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 **`cs`** 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 **`cs`** 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 quadruple (8 byte ciascuna), in ordine:
   - 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 **cs**;
   - il contenuto di **rip**.

   (in cima alla pila c'è dunque **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 **cs**=utente/L=utente e **cs**=sistema/L=sistema. Possiamo ignorare quello con L=utente, per quando appena detto. Il caso **cs**=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 **cs**, che dice a che livello si trovava il processore prima dell'interruzione.

Si noti però che la seconda motivazione è importante solo quando il sistema dispone di più di un processore (cosa che noi non prenderemo in considerazione). In presenza di un unico processore, infatti, il codice di sistema stesso potrebbe copiare le informazioni dalla pila in memoria M1, senza il timore che il codice utente possa interferire nella copia. La prima motivazione, invece, resta vera in ogni caso.

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

(a) Confronta il valore corrente di **cs** 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**, **cs**, **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 **cs** 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 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

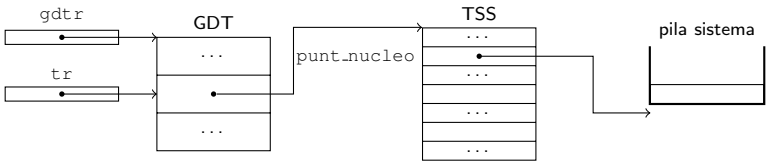

Figura 1: Strutture dati del sistema associate ad ogni processo.

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. È necessario, però, continuare a definirne almeno uno, per compatibilità con il fondamentale meccanismo delle interruzioni. Chiameremo `punt_nucleo` il campo del TSS che punta alla pila di livello sistema.

Nell'idea originaria ci doveva essere un TSS per ogni processo ma, ad ogni istante, uno solo è quello "attivo". Il registro **`tr`** (Task Register) serve a identificare il TSS attivo e può essere caricato usando l'istruzione **`ltr`**. Tutti i TSS esistenti sono descritti da una tabella GDT (Global Descriptor Table), puntata dal registro `gtdr` che può essere caricato con l'istruzione **`lgdtr`**. La GDT 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.

Ovviamente, le istruzioni **`lgdtr`** e **`ltr`** sono privilegiate e la GDT e tutti i TSS devono trovarsi in M1.

Nell'idea originaria, il software di sistema deve aggiornare **`tr`** ogni volta che cambia processo. Noi useremo un unico TSS, caricando **`tr`** una sola volta in fase di inizializzazione. Avremo però una pila sistema diversa per ogni processo e, ad ogni cambio di processo, dovremo aggiornare il campo `punt_nucleo` di quell'unico TSS.

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 **`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. Lo stesso accade se si tenta di modificare il flag IF tramite **`popf`**.