Le primitive

G. Lettieri

3 Giugno 2017

Il nucleo mette a disposizione dei programmi utente una serie di "primitive", funzioni che i programmi utente possono chiamare liberamente per svolgere (in maniera controllata) alcune operazioni che altrimenti gli sarebbero vietate.

L'idea è che i programmi utente devono essere considerati *non fidati*, e dunque il funzionamento del sistema non deve dipendere dal fatto che i programmi utente si comportino correttamente. Si pensi, per esempio, alle code e descrittori dei processi: dalla corretta gestione di queste strutture dati dipende tutta la multiprogrammazione del sistema, dunque i programmi utente non devono potervi accedere liberamente. D'altro canto i programmi utente devono poter creare nuovi processi, e questa operazione comporta (come abbiamo visto), la creazione e inizializzazione di un `des_proc` e di un `proc_elem`, l'inserimento di quest'ultimo in coda pronti, etc.

Per conciliare queste due esigenze constrastanti, si opera in questo modo:

- i programmi utente vengono eseguiti con il processore a livello utente;
- le stutture dati critiche vengono rese inaccessibili da livello utente;
- il programmatore di sistema scrive delle funzioni che svolgono le operazioni per conto dell'utente (per es., creare un processo), assicurandosi di manipolare correttamente le strutture dati;
- il programmatore di sistema permette agli utenti di invocare la sue funzioni esclusivamente tramite *gate* della IDT (dunque tramite l'istruzione **`int`**) che innalzino il livello del processore (portandolo a sistema).

Mentre sono in esecuzione le funzioni scritte dal programmatore di sistema, e solo allora, il processore si trova a livello sistema e può manipolare le strutture dati critiche, altrimenti queste sono inaccessibili.

1 Meccanismo di chiamata

A livello Assembler, invocare una primitiva non è come invocare una semplice funzione in quanto, come abbiamo detto, è necessario passare attraverso un gate della IDT con una istruzione **`int`**.

Al livello del C++, però, possiamo fare in modo che la primitiva si usi come una qualunque funzione, per maggior comodità dell'utente.

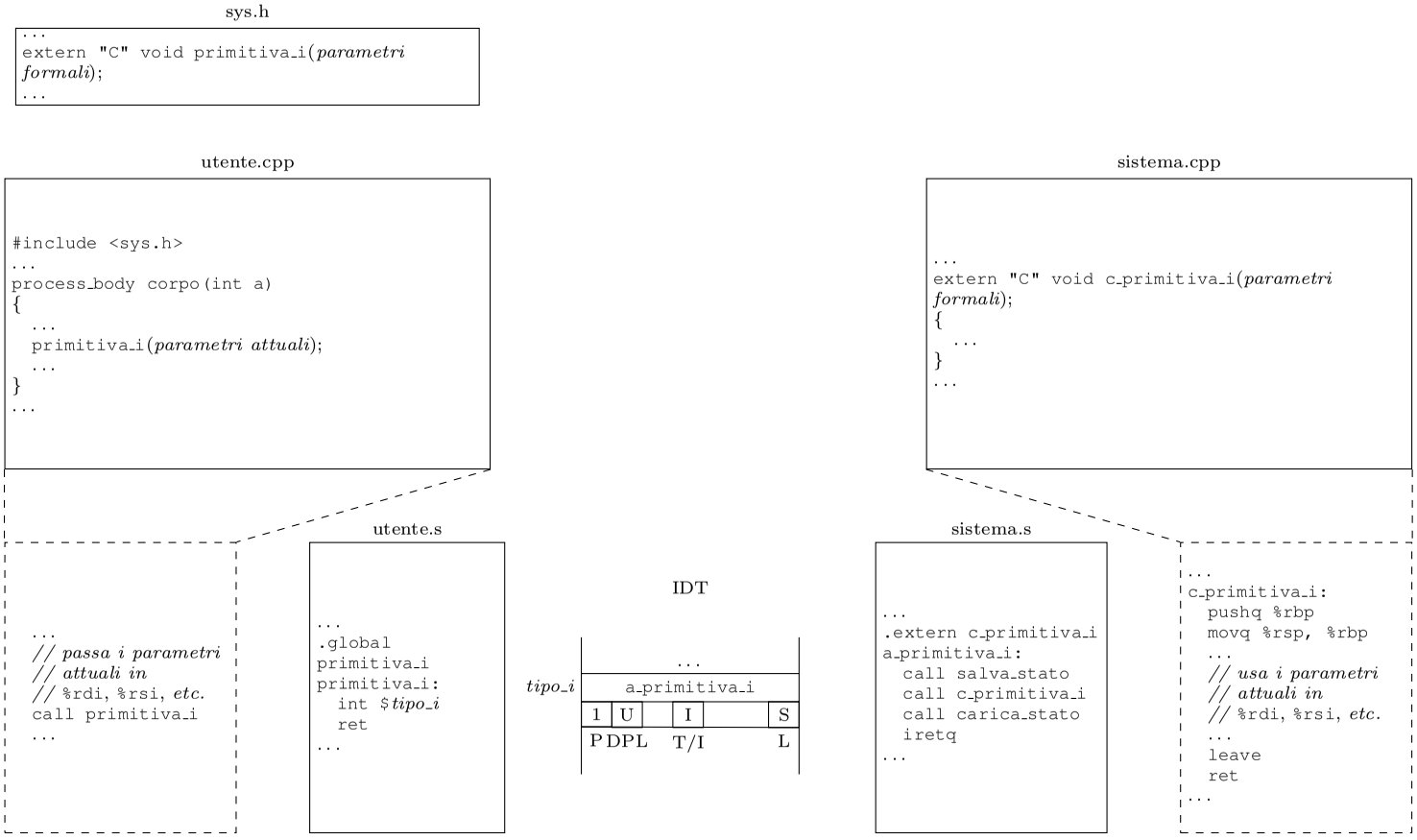

Figura 1: Primitive che possono causare un cambio di processo.

Il meccanismo è illustrato in Figura 1. L'utente, nel file `utente.cpp`, dichiara e chiama la funzione `primitiva_i()`, con l'obiettivo di eseguire la funzione `c_primitiva_i()`, contenuta nel modulo sistema. La funzione `primitiva_i()` è in realtà solo un piccolo programma di interfaccia, scritto in assembler e contenuto nel file `utente.s`, che si limita ad esegure l'istruzione **`int`** con l'indice del gate della primitiva nella IDT (*tipo_i*).

L'istruzione **`int`** si preoccuperà di innalzare il livello di privilegio (con le varie operazioni connesse, tra cui il cambio di pila) e saltare al codice della primitiva nel modulo sistema, salvando il pila l'indirizzo dell'istruzione successiva (una **`ret`**, in questo caso). Se non ci sarà un cambio di processo, questa è l'istruzione a cui il processore salterà al termine dell'esecuzione della primitiva.

Si noti che, a livello Assembler, anche *ritornare* da una primitiva non è come ritornare da una normale funzione, in quanto è necessario eseguire l'istruzione **`iretq`**, che è l'unica che permette di riportare il processore a livello utente. Anche nel modulo sistema dovremo quindi aggiungere una funzione di interfaccia (`a_primitiva_i` in Figura), che chiami la `c_primitiva_i()` e poi esegua la **`iretq`**. Con questo accorgimento, la `c_primitiva_i()` può essere scritta in C++ e compilata normalmente.

Affinchè l'istruzione "`int` *$tipo_i*" salti alla funzione `a_primitiva_it`, con innalzamento di privilegio, il programmatore di sistema deve predisporre l'entrata della IDT di offset *tipo_i* come illustrato in figura:

- il campo IND (indirizzzo a cui saltare) contiene `a_primitiva_i`;
- il campo P (gate Present) contiene 1, ad indicare che il gate è implementato;
- il campo DPL (Descriptor Privilege Level) indica che il gate può essere utilizzato da livello utente tramite una istruzione **`int`**;
- il campo L (Level) indica che, dopo il salto, il processore si deve trovare a livello sistema;
- per motivi che vedremo in seguito, il campo I/T indica che il gate è di tipo "interrupt", in modo che le interruzioni esterne mascherabili vengano disabilitate.

La `a_primitiva_i` dovrà anche chiamare `salva_stato` e `carica_stato`, per realizzare il meccanismo del cambio di processo. In questo modo la `c_primitiva_i()` può sospendere il processo corrente e schedularne un altro, semplicemente cambiando il valore della variabile `esecuzione`.

I parametri formali della `c_primitiva_i()` sono gli stessi della corrispondente `primitiva_i()`. In Figura 1 si vede che, nel tradurre la chiamata a `primitiva_i()`, il compilatore C++ copierà i paramtri attuali nei registri **`rdi`**, **`rsi`**, etc. Questi registri non vengono modificati mentre si passa da `primitiva_i` e `a_primitiva_i`, quindi la funzione `c_primitiva_i()` li troverà ancora lì, dove il compilatore C++ se li aspetta.

Sia `primitiva_i()` che `c_primitiva_i()` sono dichiarate **`extern "C"`**. In questo modo il compilatore C++ assume che le due funzioni seguano lo standard di aggancio del linguaggio C, che non prevede l'overloading delle funzioni e dunque non richiede che i nomi delle funzioni vengano trasformati come abbiamo visto nel caso del C++. Facciamo questo per comodità, dal momento che non prevediamo di sfruttare l'overloading per le primitive.

Normalmente il programmatore di sistema fornisce all'utente anche il file `utente.s` e un header file che contenga le dichiarazioni delle primitive (`primitiva_i()`, nell'esempio). L'utente non deve fare altro che includere tale file, con la direttiva **`#include`**, nel suo `utente.cpp`. Nel nostro caso le dichiarazioni si trovano nel file `sys.h` (nella directory `utente/include`).

1.1 Primitive che restituiscono un risultato

Si noti che la primitiva in Figura 1 è di tipo **`void`**. La presenza della `carica_stato` rende un po' complicato restituire un valore dalla `c_primitiva_i()` alla `primitiva_i()` tramite il registro **`rax`**. Una istruzione di "`return` *ris*;" nella `c_primitiva_i()` verrebbe tradotta dal compilatore C++ lasciando il valore *ris* nel registro **`rax`**, ma la `carica_stato` sovrascriverebbe tale valore prima che la `primitiva_i()` possa vederlo.

Se una primitiva deve restituire un valore, occorre operare come in Figura 2. La funzione `primitiva_j()`, che è quella direttamente invocata dall'utente, è dichiarata di tipo $tipo_r$, ma la corrispondente `c_primitiva_j()` è **`void`**. Il valore *ris* deve essere restituito modificando il campo `contesto[I_RAX]` del descrittore del processo, in modo che la successiva `carica_stato` ricopi *ris* nel registro **`rax`**, dove se lo aspetta il compilatore C++ dopo la chiamata a `primitiva_j()`.

1.2 Primitive che non causano cambi di processo

Se una primitiva non causa mai un cambio di processo, alcune cose possono essere semplificate come in Figura 3. Dal punto di vista dell'utente tutto resta invariato, ma il programmatore di sistema può eliminare le chiamate a `salva_stato` e `carica_stato` in `c_primitiva_k`. Inoltre, se la primitiva deve restituire un valore, può tranquillamente lasciarlo in **`rax`**, in quanto ora non verrà sovrascritto. In Figura 3 vediamo che `c_primitiva_k()` è ora dichiarata di tipo $tipo_r$ come la corrispondente `primitiva_k()` e il risultato *ris* può essere restituito con una normale **`return`**.

1.3 Controllo dei parametri

Dal momento che il programmatore utente è non fidato, le primitive di sistema devono sempre controllare i parametri che ricevono. Nel caso di parametri di tipo puntatore o riferimento, le primitive devono controllare che gli indirizzi che l'utente sta passando facciano effettivamente parte dello spazio utente, e non dello spazio sistema (problema del Cavallo di Troia). In `sistema.s` sono

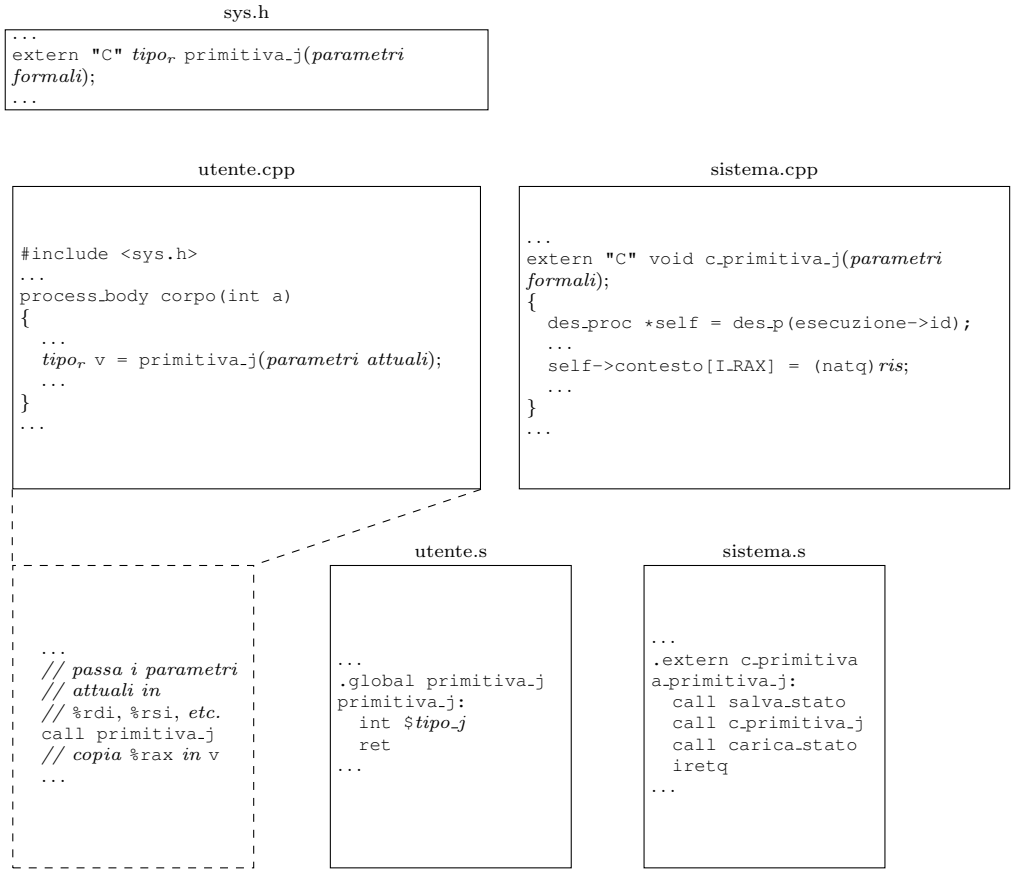

Figura 2: Primitive che possono causare un cambio di processo e devono restituire un valore.

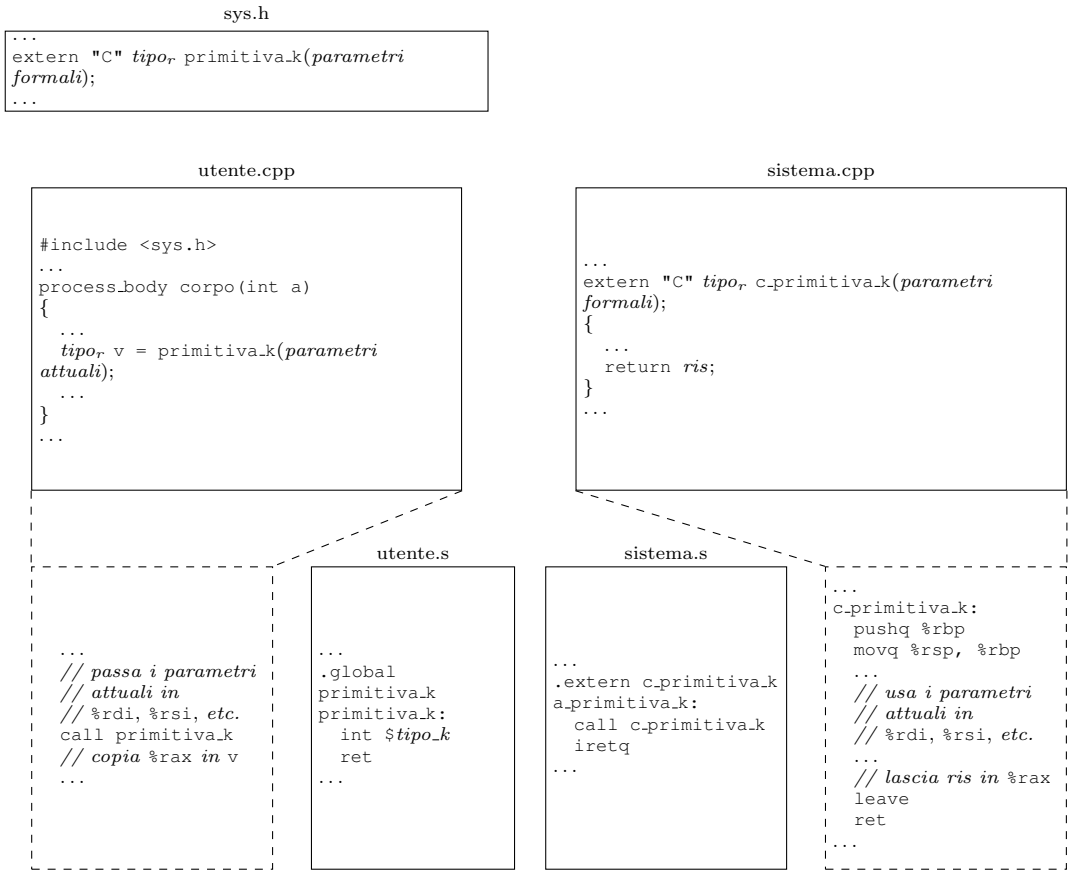

Figura 3: Primitive che non causano mai un cambio di processo.

definite un paio di macro che possono essere usate per controllare questo tipo di parametri e abortire il processo in caso di errore. L'idea è che i puntatori in genere puntano ad aree di memoria più grandi di un byte, e la primitiva deve controllare che tutti i byte dell'area siano accessibili dall'utente.

`cavallo_di_troia`
: richiede un parametro, che deve essere un operando assembler (per es., un registro); controlla che l'operando contenga un indirizzo a cui l'utente può accedere;

`cavallo_di_troia2`
: richiede due parametri, ciascuno dei quali deve essere un operando assembler; il primo parametro ha lo stesso significato che per `cavallo_di_troia`, mentre il secondo deve specificare la lunghezza della zona di memoria puntata.

Tali macro devono essere usate nella parte Assembler della primitiva, prima di chiamare la parte C++. Se la parte assembler chiama `salva_stato`, queste macro devono essere chiamate dopo (in quanto sporcano alcuni registri).

Per esempio, consideriamo una primitiva `p(char *buf, natl dim)` che riceve un puntatore ad un buffer `buf`, di dimensione `dim`. Per controllare il problema del Cavallo di Troia, la parte assembler della primitiva dovà chiamare

```
cavallo_di_troia %rdi
cavallo_di_troia2 %rdi %rsi
```

in quanto il parametro `buf` verrà passato in `rdi` e il parametro `dim` in `rsi`.

2 Scrivere nuove primitive

Per aggiungere una nuova primitiva si deve seguire lo schema appropriato di Figura 1, 2 o 3 e predisporre una entrata della IDT.

Supponiamo di voler aggiungere una primitiva `getid()`, senza parametri, che restituisce l'identificatore del processo che la invoca.

Per prima cosa occorre assegnare un tipo di interruzione alla primitiva. I tipi di tutte le primitive già realizzate sono definiti nel file `costanti.h` nella cartella `include`, con nomi che iniziano con `TIPO_`. Si noti che i tipi sono definiti come macro, in modo che il file possa essere utilizzato sia dal C++ che dall'assembler. Scegliamo un tipo non utilizzato (per esempio, 0x59) e definiamo una nuova macro:

```
#define TIPO_GETID   0x59
```

Per caricare il corrispondente gate della IDT possiamo aggiungere una riga alla funzione `init_idt` che si trova nel file `sistema.s`. Tale funzione è chiamata all'avvio del sistema e si occupa di inizializzare la tabella IDT. Possiamo usare la macro `carica_gate` che richiede tre parametri:

- il tipo della primitiva (da cui si deduce il gate della IDT da inizializzare);
- l'indirizzo a cui saltare quando qualcuno usa il gate;
- il livello di privilegio minimo richiesto per utilizzare il gate tramite una **int** (campo DPL del gate).

La macro inizializza sempre il campo P con 1 (gate implementato), il campo I/T con I (gate di tipo interrupt) e il campo L con S (la primitiva andrà in esecuzione a livello sistema). Nel nostro caso aggiungeremo la riga

```
carica_gate  TIPO_GETID  a_getid  LIV_UTENTE
```

dove `LIV_UTENTE` è una costante che specifica il livello utente (esiste anche la costante `LIV_SISTEMA` per indicare che il gate può essere usato solo da livello sistema).

Sempre nel file `sistema.s` dobbiamo scrivere la funzione `a_getid`. In questo caso possiamo adottare lo schema di Figura 3 (questa primitiva non ha sicuramente bisogno di bloccare il processo che la chiama):

```
.extern c_getid
a_getid:
  call c_getid
  iretq
```

Infine, scriviamo la primitiva vera e propria (in `sistema.cpp`):

```
extern "C" natl c_getid()
{
  return esecuzione->id;
}
```

Dal punto di vista del modulo sistema non dobbiamo fare altro. Possiamo ricompilare il modulo sistema con il comando "`make`" e correggere eventuali errori di sintassi.

Il programmatore di sistema, come detto, dovrebbe anche fornire la dichiarazione e il programma assembler di interfaccia per l'utente. In `sys.h` aggiungiamo

```
extern "C" natl getid();
```

e nel file `utente.s`

```
.global getid
getid:
     int $TIPO_GETID
     ret
```

Il compito del programmatore di sistema finisce qui. A questo punto gli utenti possono scrivere i loro programmi e usare la nuova primitiva. Per esempio, scriviamo il seguente programma utente nel file `provagetid.in` nella directory `utente/prog`:

```
#include <sys.h>
#include <lib.h>

process prova body corpo(0), 20, LIV_UTENTE;
process_body corpo(int a)
{
  char buf[10];
  natl id;

  writeconsole("Il mio id: ");
  id = getid();
  int_conv(id, buf);
  writeconsole(buf);
}
```

Per provare il programma utente lanciamo il comando "`make swap`" (correggendo eventuali errori di sintassi) e poi avviamo il sistema con "`./run`".

2.1 Funzioni di supporto

Le seguenti funzioni sono già definite in `sistema.cpp` e possono essere utilizzate nel definire nuove primitive.

`des_proc *des_p(natl id)`
: restituisce un puntatore al descrittore del processo di identificatore `id` (0 se tale processo non esiste).

`void schedulatore()`
: sceglie il prossimo processo da mettere in esecuzione (cambia il valore della variabile `esecuzione`).

`void inserimento_lista(proc_elem *&p_lista, proc_elem *p_elem)`
: inserisce `p_elem` nella lista `p_lista`, mantenendo l'ordinamento basato sul campo `precedenza`. Se la lista contiene altri elementi che hanno la stessa precedenza del nuovo, il nuovo viene inserito come ultimo tra questi.

`void rimozione_lista(proc_elem *&p_lista, proc_elem *&p_elem)`
: estrae l'elemento in testa alla `p_lista` e ne restituisce un puntatore in `p_elem`.

`inspronti()`
: inserisce il `proc_elem` puntato da `esecuzione` in testa alla coda `pronti`.

`abort_p()`
: distrugge il processo corrente e salta ad un altro. Attenzione: la funzione non ritorna al chiamante.

2.2 Considerazioni generali

Discutiamo qui alcune cose a cui prestare attenzione nella scrittura delle primitive.

- Le primitive girano a interruzioni disabilitate e eventuali eccezioni (compreso il page faul) si verificano solo in caso di errore di programmazione. Quindi, durante l'esecuzione una primitiva, il processore non fa altro che eseguire il codice della primitiva stessa: niente altro avviene concorrentemente nella CPU. In particolare, se eseguiamo un ciclo infinito in una primitiva, tutto il sistema si ferma. È un grave errore scrivere una cosa del genere nel codice di una primitiva:

  ```
  int i = 0;
  while (i == 0) {
     // altre cose che non modificano i
  }
  ```

  Nessuno può modificare il valore della variabile `i`, quindi questo è un ciclo infinito.

- Cambiare il valore della variabile `esecuzione` non cambia magicamente il processo corrente. Il salto al nuovo processo avverrà solo quando verrà chiamata la funzione `carica_stato`, seguita dall'istruzione **`iretq`**.

- La funzione `schedulatore()` non fa altro che cambiare il valore della variabile `esecuzione` (quindi non salta lei stessa ad un nuovo processo).

- Se un processo viene sospeso mentre esegue una primitiva (mettendo qualche altro processo in `esecuzione` al suo posto), al suo suo risveglio riparte dall'istruzione successiva alla **`int`** con cui aveva chiamato la primitiva. In generale riparte dall'ultimo stato salvato e, nel caso di sospensione durante l'esecuzione di una primitiva, l'ultimo stato è quello salvato dalla `salva_stato` all'inizio della parte assembler della primitiva.