

# Realizzazione delle primitive

G. Lettieri

12 Aprile 2023

Vediamo ora come le primitive sono realizzate nel sistema didattico.

## 1 Atomicità

Le primitive del nostro sistema devono lavorare su un insieme di strutture dati globali, come i descrittori di processo e le code dei processi. Che succederebbe se, mentre una primitiva sta lavorando su una di queste strutture, sia  $S$ , una interruzione (sia essa una interruzione esterna, una eccezione o una **int**) causasse un salto ad un'altra primitiva, la quale cercasse di accedere alla stessa struttura  $S$ ? Pensiamo, per esempio, ad una primitiva che sta cercando di inserire un nuovo `des_proc*`, sia  $d1$ , in lista `pronti`, supponiamo in testa. Per farlo deve modificare due puntatori: copiare `pronti` in `d1->puntatore` e scrivere  $d1$  in `pronti`. Supponiamo che, tra la prima e la seconda scrittura, il processore salti, per effetto di una interruzione, ad un'altra routine di sistema, e che questa cerchi di inserire un altro `des_proc*`, sia  $d2$ , in testa alla coda `pronti`. Questa seconda primitiva copierà `pronti` in `d2->puntatore` e scriverà  $d2$  in `pronti`. Al termine della seconda primitiva si ritornerà alla prima, che proseguirà dal punto in cui si era interrotta, e in particolare scriverà  $d1$  in `pronti`, cancellando quanto vi aveva scritto la seconda. L'effetto è che il `des_proc* d2` non è più puntato da niente.

Chiaramente non vogliamo che quanto appena descritto possa accadere, ma questo è solo uno degli infiniti problemi che si potrebbero presentare (si pensi, per esempio, ad una `salva_stato` interrotta da un'altra `salva_stato`). Più in generale, quello che abbiamo descritto è un problema di “interferenza” tra due flussi di esecuzione che lavorano su una stessa struttura dati. In generale, noi vogliamo che ogni struttura dati si trovi in uno stato “consistente”. Per esempio, una lista è in uno stato consistente se tutti e soli i suoi elementi sono raggiungibili dalla testa. In un sistema che non prevede interruzioni, le operazioni che manipolano una struttura dati vengono scritte assumendo che la struttura dati si trovi sempre in uno stato consistente quando l'operazione inizia, e assicurandosi di portarla in un nuovo stato consistente *alla fine* dell'operazione. Nel mezzo dell'operazione, però, la struttura dati può passare temporaneamente attraverso stati non consistenti. Ripensiamo all'operazione di inserimento in testa alla coda `pronti` eseguita dalla prima primitiva: subi-

to dopo la copia di `pronti` in `d1->puntatore`, la lista non è in uno stato consistente, in quanto `d1` ne fa concettualmente parte, ma non è ancora puntato da `pronti`. Questo stato inconsistente non è un problema in un sistema senza interruzioni, in quanto non è osservabile da nessun'altra operazione sulla coda: la routine di inserimento proseguirà scrivendo `d1` in `pronti`, riportando così la lista in uno stato consistente. In presenza di interruzioni, però, lo stato inconsistente diventa improvvisamente visibile da un'altra operazione, che era stata scritta assumendo che ciò non potesse mai accadere, e che dunque non è preparata per affrontare la situazione.

Abbiamo due modi principali per evitare i malfunzionamenti causati dall'interferenza:

- scrivere tutte le routine in modo da tener conto di tutti i modi in cui queste si possono mescolare, in modo che funzionino in ogni caso (possibile e anzi desiderabile, ma molto complesso);
- prevenire *a priori* l'interferenza, eliminando tutte le sorgenti di interruzione durante l'esecuzione delle primitive (o almeno delle parti critiche di esse).

Noi adotteremo la seconda soluzione per tutte le primitive del modulo `sistema`, e per la loro intera durata. In particolare, i gate che portano a tali primitive saranno di tipo "interrupt", con disabilitazione automatica delle interruzioni esterne mascherabili. Durante la scrittura delle primitive, poi, staremo attenti a non causare eccezioni e a non chiamare altre primitive tramite `int`. Con questi accorgimenti, le nostre primitive gireranno in un contesto "atomico": una volta iniziate saranno portate a compimento, senza che niente le possa interrompere<sup>1</sup>. Diventeranno in questo modo molto simili alle singole istruzioni di linguaggio macchina, la cui atomicità è garantita dal processore (che gestisce interruzioni esterne e eccezioni solo tra una istruzione e la successiva).

Si noti che in molti sistemi reali l'atomicità viene considerata un prezzo troppo alto da pagare e viene rilasciata in vari modi. Noi la rilasceremo solo nel modulo `io`, ma alcuni testi d'esame considerano il caso di rilassamento anche nel modulo `sistema`.

## 2 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. Il meccanismo che usiamo è illustrato in Figura 3. L'utente, nel file `utente.cpp`, dichiara e chiama la funzione `primitiva_i()`, con l'obiettivo di eseguire la funzione

---

<sup>1</sup>Salvo bug e interruzioni esterne non mascherabili, che comunque causeranno il blocco dell'intero sistema.

`c_primitiva_i()` che è contenuta nel modulo `sistema`. La `primitiva_i()` è in realtà solo un piccolo programma di interfaccia, scritto in assembler e contenuto nel file `utente.s`, che si limita ad eseguire 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 in 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_i`, 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 (indirizzo 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 deve trovare a livello sistema;
- per i motivi che abbiamo visto nella sezione precedente, il campo I/T indica che il gate è di tipo "interrupt", in modo che le interruzioni esterne mascherabili siano 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 funzione `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 3 si vede che, nel tradurre la chiamata a `primitiva_i()`, il compilatore C++ copierà i parametri 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`).

## 2.1 Primitive che restituiscono un risultato

Si noti che la primitiva in Figura 3 è di tipo **void**. La presenza della chiamata di `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 1. La funzione `primitiva_j()`, che è quella direttamente invocata dall'utente, è dichiarata di tipo *tipo<sub>r</sub>*, 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()`.

## 2.2 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 2. 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 2 vediamo che `c_primitiva_k()` è ora dichiarata di tipo *tipo<sub>r</sub>*, come la corrispondente `primitiva_k()` e il risultato *ris* può essere restituito con una normale **return**.

Si noti però che, con questo schema, eventuali registri *scratch* sporcati dalla primitiva non verrebbero ripristinati al ritorno a livello utente. Questa può essere considerata una violazione della riservatezza dei dati del sistema.

## 3 Scrivere nuove primitive

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

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

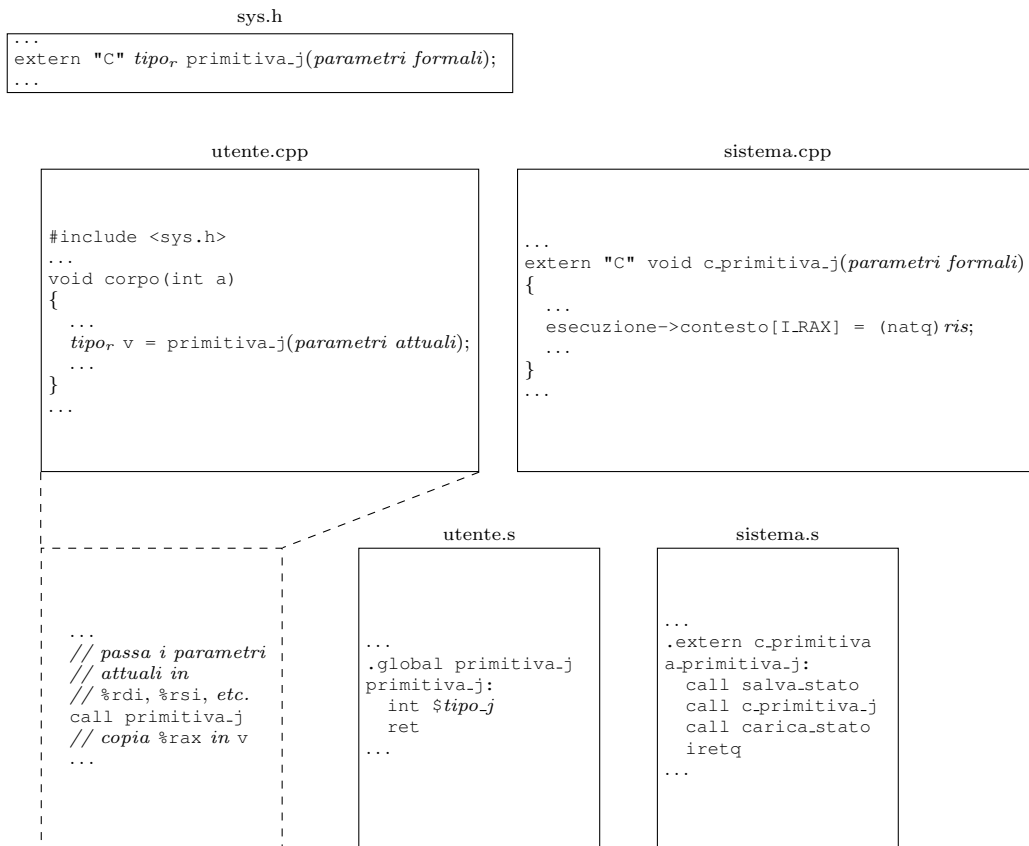


Figura 1: Primitive che devono restituire un risultato.

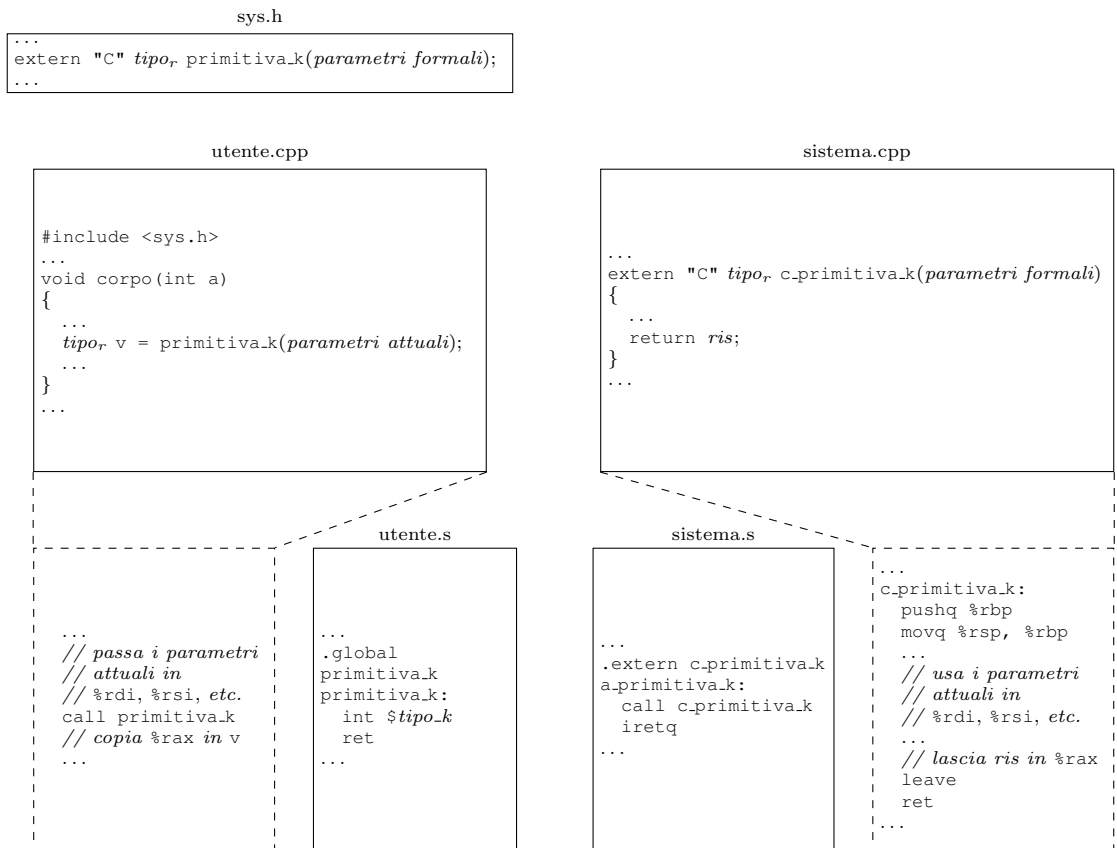


Figura 2: Primitive che non causano mai un cambio di processo.

Per prima cosa occorre assegnare un tipo di interruzione alla primitiva. I tipi di tutte le primitive già realizzate sono definiti nel file `constanti.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, `0x2a`) e definiamo una nuova macro:

```
#define TIPO_GETID    0x2a
```

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`.

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

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

```
extern "C" void c_getid()
{
    esecuzione->contesto[I_RAX] = 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" natw 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 `utente.cpp` nella directory `utente`:

```
#include <all.h>

void corpo(int a)
{
    natw id;

    id = getid();
    printf("Il mio id: %hu", id);
    terminate_p();
}

int main()
{
    activate_p(corpo, 0, 20, LIV_UTENTE);
    terminate_p();
}
```

Per provare il programma utente lanciamo il comando “make” (correggendo eventuali errori di sintassi) e poi avviamo il sistema con “boot”.

### 3.1 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` (`nullptr` se tale processo non esiste).

`void schedulatore()`

Sceglie il prossimo processo da mettere in esecuzione (cambia il valore della variabile esecuzione).

```

void inserimento_lista(des_proc*& p_lista, des_proc* 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.

des_proc* rimozione_lista(des_proc*& p_lista)
    Estrae l'elemento in testa alla p_lista e ne restituisce un puntatore
    (nullptr se la lista è vuota).

void inspronti()
    Inserisce il des_proc puntato da esecuzione in testa alla coda pronti
    .

void c_abort_p()
    Distrugge il processo puntato da esecuzione e chiama schedulatore
    ().

```

### 3.2 Chi esegue le primitive atomiche

È naturale pensare che le azioni svolte da una primitiva siano eseguite dal processo che l'ha invocata. Ci sono molte cose che ci inducono a concettualizzare l'esecuzione in questo modo: principalmente il fatto che il flusso di controllo è sostanzialmente continuo, almeno fino a quando non si “salta” ad un altro processo, ma anche il fatto che il contesto corrente è in buona parte quello del processo invocante: lo stato iniziale dei registri è quello lasciato dal processo, la primitiva usa la pila sistema di quel processo e, come vedremo, anche la sua memoria (virtuale). A ciò si aggiunga che, per alcuni tipi di primitive, questo è effettivamente il modo migliore di ragionare (nel nostro caso è così per le primitive del modulo di I/O, che non abbiamo ancora trattato).

Nel caso delle primitive atomiche, però, questo modo di pensare può portare a diverse confusioni. Ripensiamo alla definizione di processo come la sequenza di stati attraverso cui processore e memoria (privata del processo) passano durante l'esecuzione di un programma su dei dati; ricordiamo che possiamo visualizzare la sequenza di stati come il filmato della storia del processo, in cui ogni fotogramma ritrae il processo mentre ha appena finito di eseguire una istruzione e sta per eseguire la prossima. Tutta l'esecuzione di una primitiva atomica si svolge tra *due* di questi fotogrammi: uno in cui il processo è ritratto mentre sta per eseguire l'istruzione **int** e uno che lo ritrae subito dopo. Tutte le azioni svolte dalla primitiva non aggiungono nuovi fotogrammi tra questi due, e dunque non fanno parte della storia del processo. Tra questi due fotogrammi possono anche andare in esecuzione altri processi, che possono invocare altre primitive: tutto ciò è invisibile nella storia del processo.

Per rendere le cose meno astratte, pensiamo a cosa accade all'ingresso di una nostra primitiva: il meccanismo di interruzione e la funzione `salva_stato` scattano una foto dello stato del processo invocante. La foto coincide sostanzialmente con il primo fotogramma di cui abbiamo parlato, contenente lo stato

di processore e memoria privata subito prima dell'esecuzione della `int`, con l'eccezione di `rip` che, per come funziona il meccanismo, è già quello dell'istruzione successiva e dunque fa già parte, a rigore, del secondo fotogramma. Tutto ciò che accade fino a quando il processo non ritorna in esecuzione può servire a definire il secondo fotogramma, ma per tutto questo tempo dobbiamo immaginare che il processo sia fermo al fotogramma precedente e non stia compiendo azioni. Per esempio, a meno che il codice di sistema non abbia volutamente scritto nel campo `contesto` del descrittore del processo, i registri generali del secondo fotogramma conterranno gli stessi valori che avevano nel primo, anche se tra i due fotogrammi la CPU ha eseguito molteplici processi e primitive.

Queste considerazioni valgono in particolare per tutto il tempo di esecuzione della primitiva che il processo aveva invocato: mentre il processore sta eseguendo il codice della primitiva, il processo invocante va pensato "congelato", in uno stato temporaneamente simile a quello di tutti gli altri processi che non erano in esecuzione. Questo è vero in generale per tutto il codice del modulo sistema eseguito atomicamente (che comprende anche le routine di risposta alle eccezioni e alle interruzioni esterne). Dobbiamo pensare quindi che la primitiva è eseguita da un'entità diversa dai processi e che, anche se usa le risorse del processo che l'ha invocata, le abbia soltanto prese "in prestito".

Passiamo ora ad una delle cose che causano maggior confusione nella scrittura di primitive atomiche. La variabile `esecuzione` punta inizialmente al processo che *era* in esecuzione quando la primitiva è stata invocata. Quando cambiamo il valore della variabile `esecuzione` o invociamo `schedulatore()` (che non fa altro che scrivere un nuovo valore in `esecuzione`) non significa che stiamo cedendo il controllo ad un altro processo *in quel momento*. Ci sono diversi campanelli di allarme che dovrebbero suonare e impedire di cadere in questo errore:

- come può la semplice scrittura in una variabile in memoria deviare immediatamente il flusso di controllo?
- le primitive atomiche non possono sospendersi, per definizione; come è possibile che `schedulatore()` contravvenga a questa regola?

Se questi non dovessero bastare, aggiungiamo la conclusione del ragionamento che abbiamo fatto fino ad ora: le primitive non sono eseguite da un processo, quindi non ha senso pensare che `schedulatore()` causi un cambio del processo corrente, perché *non c'è alcun processo corrente*. Il nuovo processo andrà in esecuzione alla prossima invocazione della coppia `call carica_stato; iretq`.

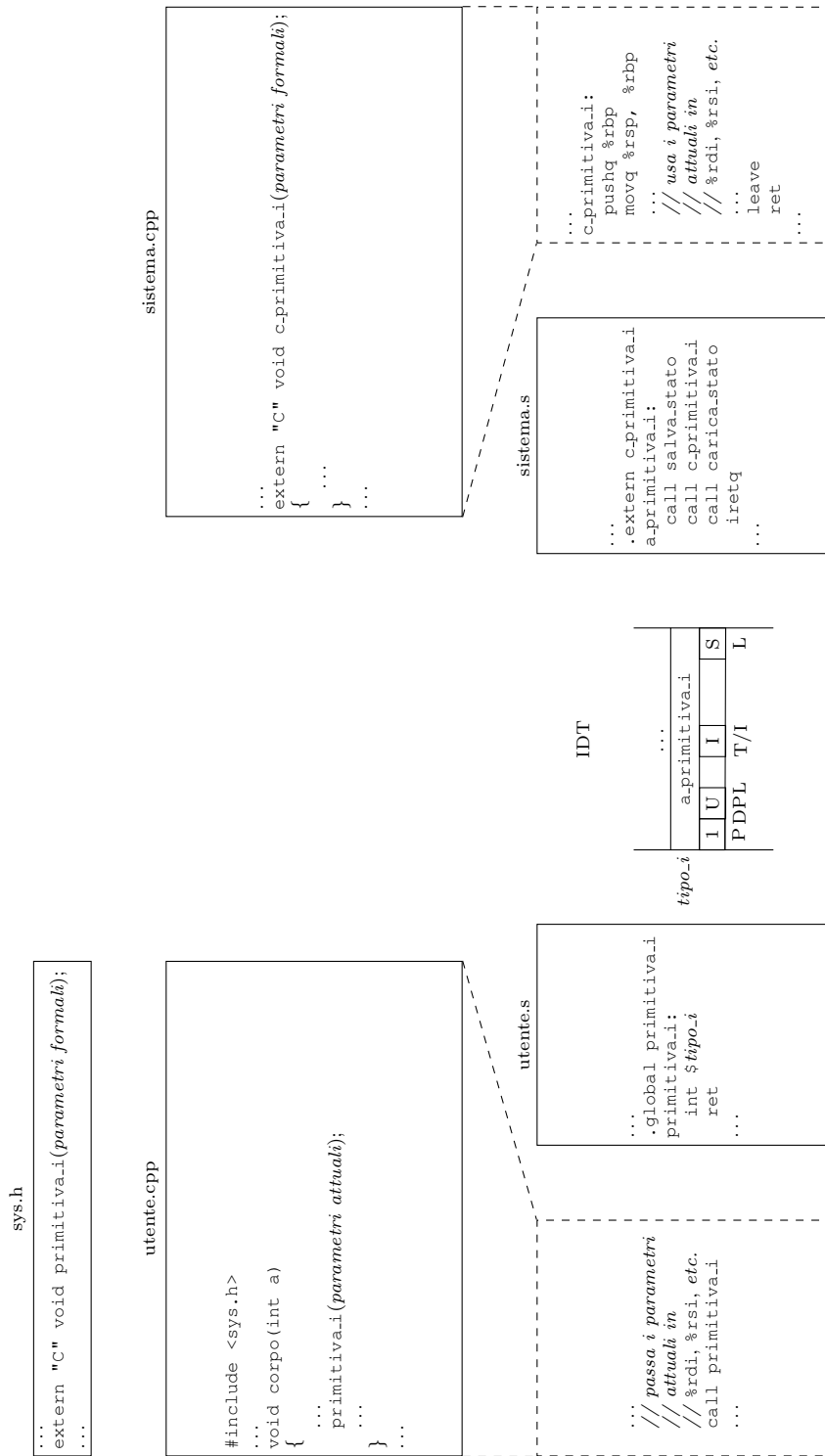


Figura 3: Schema di chiamata di una primitiva.