Interfacciamento di Assemby e linguaggio C

Uso di funzioni C in programmi Assembly

Quando si scrivono programmi in Assembly in GNU/Linux, è possibile utilizzare in modo abbastanza agevole funzioni scritte in linguaggio C al loro interno. Questa possibilità è particolarmente utile per tutte quelle funzioni che vanno a gestire input e output, sia da periferica che da file, evitando tutti i problemi di conversione dei valori, da stringhe a numerici e viceversa, che si dovrebbero affrontare usando le routine di I/O native di Assembly.

Il richiamo delle funzioni C dai sorgenti Assembly prevede le seguenti operazioni:

- caricamento sullo stack dei parametri della funzione, uno alla volta in ordine inverso rispetto a come appaiono nella sintassi della stessa in linguaggio C
- richiamo della funzione con l'istruzione: `call nome_funzione`
- ripristino del valore del registro ESP al valore precedente agli inserimenti nello stack

Esempio

Il seguente programma richiede l'inserimento di due valori (numerici) da tastiera, esegue un calcolo aritmetico (somma) e ne visualizza il risultato a monitor. Grazie all'uso delle funzioni C invece che delle routine native, non sono più necessarie le conversioni di tipo tra stringa e numero, ed il programma risulta più leggibile, oltre che più breve.

NOTA: si presti la massima attenzione al fatto che l'esecuzione delle funzioni `printf` e `scanf` avviene con l'uso, da parte del processore, dei registri accumulatori che vengono quindi modificati da tali esecuzioni e, nel caso contengano valori utili all'elaborazione, devono essere salvati in opportune etichette di appoggio.

```
######################
# filename: somma_c.s
######################

.section .bss
  val1: .long 0       # primo valore di input
  val2: .long 0       # secondo valore di input
  ris: .long 0        # risultato

.section .data
  invito: .string "Inserire un valore: "    # stringa terminata con \0
  formato: .string "%d"                     # formato input
  risultato: .string "Somma = %d\n"         # stringa per risultato
```

```
.section .text
  .global _start

_start:
  # Input primo valore
  pushl $invito
  call printf
  addl $4, %esp
  pushl $val1
  pushl $formato
  call scanf
  addl $8, %esp

  # Input secondo valore
  pushl $invito
  call printf
  addl $4, %esp
  pushl $val2
  pushl $formato
  call scanf
  addl $8, %esp

  # Calcolo val1 + val2
  movl (val2), %eax
  addl (val1), %eax
  movl %eax, (ris)

  # Stampa risultato
  pushl (ris)
  pushl $risult
  call printf
  addl $8, %esp

fine:
  movl $1, %eax
  int $0x80
```

Per generare l'eseguibile di nome `somma_c` è necessario aggiungere dei flag in fase di linking:

```
as -o somma_c.o somma_c.s
ld -dynamic-linker /lib/ld-linux.so.2 -lc -o somma_c
     somma_c.o
```

dove:

- `lc` significa che si devono collegare le librerie del linguaggio C
- `dynamic-linker /lib/ld-linux.so.2` serve per specificare il linker dinamico da usare per caricare dinamicamente le librerie del linguaggio C

Per ottenere più facilmente lo stesso risultato si può utilizzare lo script `gcc` per la traduzione del sorgente; in questo caso basta eseguire l'unico comando:

```
gcc -g -o somma_c somma_c.s
```

È necessario però sostituire la dichiarazione di etichetta globale nel sorgente. Questa esigenza è dovuta al fatto che il comando `gcc` si aspetta di trovare l'etichetta `main` nel file oggetto di cui effettuare il link.

as + ld	gcc

`.section .text`
`  .global _start`
 	
`_start:`
	`.section .text`
	`  .global main`
	
	`main:`

NOTA: l'opzione `-g` è analoga al flag `-gstabs` e serve solo se si ha intenzione di eseguire il programma con il debugger.

Uso di codice Assembly in programmi C

È possibile inserire in un programma scritto in linguaggio C delle porzioni di codice Assembly: si parla in tal caso di "*assembly inline*". Il motivo per cui questo può essere importante è che in questo modo è possibile scrivere delle funzioni estremamente ottimizzate, sfruttando appieno le caratteristiche dell'hardware, mentre possono rimanere in C tutte quelle operazioni, come l'input e l'output, che permettono di rendere il codice più leggibile, senza perdere in efficienza.

La sintassi da utilizzare è la seguente:

```
__asm__ (
        istruzioni assembly
        : operandi di output (opzionali)
        : operandi di input (opzionali)
        : lista dei registri modificati (opzionale)
);
```

È necessario dichiarare che un blocco di codice è scritto in linguaggio Assembly mediante la keyword `asm` preceduta e seguita da due '_' (underscore).
All'interno del blocco di codice si troveranno prima le istruzioni Assembly da eseguire, ogni riga racchiusa da virgolette ("") e termina; il secondo parametro è costituito dagli operandi che ospiteranno i relativi risultati (opzionali perché la routine potrebbe anche non prevedere valori di output); il terzo parametro sono i valori di input (opzionali in quanto possono essere assenti); l'ultimo parametro è la lista dei registri modificati (*clobbered*) nell'ambito delle istruzioni Assembly (opzionale perché potremmo non modificarne alcuno o gestire il loro salvataggio e ripristino con lo stack).

Esempio

```
// *********************
// filename: inline.c
// *********************

#include <stdio.h>

int main() {
  int val1, val2;
  printf("Inserire val1: ");
  scanf("%d",&val1);
  printf("Inserire val2: ");
  scanf("%d",&val2);
  __asm__("movl %0, %%eax; movl %1, %%ebx;"
          "xchgl %%eax, %%ebx;"
          : "=r" (val1), "=r" (val2)
          : "r" (val1), "r" (val2)
          : "%eax", "%ebx"
  );
  printf ("Valori scambiati val1=%d val2=%d\n",val1,val2);
  return 0;
}
```

Per la compilazione di questo esempio si può usare direttamente il comando gcc con l'opzione per la compilazione a 32 bit:

```
gcc -m32 -o inline inline.c
```

NOTA: per fare uso dei registri occorre raddoppiare la quantità di % nel prefisso in quanto un solo % viene usato per gli alias. In questo esempio gli alias sono %0 e %1, associati ai primi (e unici) due operandi di input.

Nella fase di assegnazione degli operandi si possono (ed è opportuno) specificare con esattezza i registri o le etichette di memoria da associare (invece di usare «r») indicandoli per esteso o con le seguenti abbreviazioni:

- 'a' per EAX
- 'b' per EBX
- 'c' per ECX
- 'd' per EDX
- 'D' per EDI
- 'S' per ESI
- 'm' per una etichetta di memoria

Esempio

```
// *********************
// filename: inline2.c
// *********************

#include <stdio.h>

int main() {
  int val1, val2, somma;
  printf("Inserire val1: ");
  scanf("%d",&val1);
  printf("Inserire val2: ");
  scanf("%d",&val2);
  __asm__("addl %%ebx, %%eax;"
          : "=a" (somma)
          : "a" (val1), "b" (val2)
  );
  printf("Somma = %d\n",somma);
  return 0;
}
```

Un'altra alternativa consiste nel richiamare da un programma C dei moduli assembly esterni. Questo è possibile alle seguenti condizioni:

- in C i moduli assembly devono essere dichiarati come 'esterni'

- nel modulo assembly la procedura deve essere dichiarata 'globale' tramite la direttiva '.global' che la rende richiamabile in altri moduli
- si devono compilare separatamente i due moduli per ottenere i rispettivi file 'oggetto' e poi collegarli in un unico eseguibile.

NOTA: i parametri eventualmente passati alla funzione Assembly si trovano nello stack a partire dall'ultimo a destra, mentre l'eventuale valore di ritorno va inserito in AL, AX, EAX, o EDX:EAX, coerentemente con la sua dimensione in bit.

Esempio

Questo esempio consiste in due sorgenti: uno per il programma principale in C e uno per la funzione Assembly. Nel programma principale si predispone l'input di due valori interi, il richiamo della funzione Assembly per il calcolo della potenza con base ed esponente i due valori, e la stampa del risultato.

```
extern int pot_asm(int b, int e);

#include <stdio.h>

int main() {

     int base, esp, potenza;
     printf("Inserire base: ");
     scanf("%d",&base);
     printf("Inserire esponente: ");
     scanf("%d",&esp);
     potenza=pot_asm(base,esp);
     printf ("Valore della potenza = %d\n",potenza);
     return 0;
}
```

Nella funzione Assembly invece si effettua il calcolo della potenza come successione di prodotti.

```
.data
.text
     .global pot_asm          # funzione deve essere globale

pot_asm:
     pushl %ebp               # event. chiamate nidificate
     movl %esp, %ebp          # imposto %ebp
     movl 8(%ebp), %ebx       # leggo la base
     movl 12(%ebp), %ecx      # leggo l'esponente
     xorl %edx, %edx          # azzero %edx
     movl $1, %eax            # imposto accumulatore
```

```
ciclo:
     mull %ebx                  # %eax = %eax*%ebx
     loopl ciclo                # ciclo %ecx volte

# al termine il ris si trova in %eax o in

     popl %ebp                  # ripristino %ebp precedente

ret
```

I sorgenti vanno quindi compilati separatamente per generare i file oggetto e poi collegati assieme per ottenere l'eseguibile:

```
gcc -c -o c-asm.o c-asm.c
gcc -c -o c-asm-funz.o c-asm-funz.s
gcc -o c-asm c-asm.o c-asm-funz.o
```