Azzolini Riccardo 2019-10-03

# Memory layout dei programmi

## 1 Aree di memoria

L'esecuzione di un programma necessita di tre aree di memoria (che in UNIX sono chiamate *regioni*):

**Area di testo**: contiene il *testo* del programma, cioè le istruzioni in linguaggio macchina. Non è modificabile dal programma stesso.

**Area dati**: contiene le *variabili globali* (cioè quelle condivise da tutte le procedure) e le strutture dati dinamiche. Queste ultime si trovano in una parte di quest'area chiamata **area heap**, alla quale puntano quindi le variabili che fanno riferimento alle strutture dinamiche.

L'area dati ha:

- contenuto variabile (cambia quando si eseguono assegnamenti alle variabili);
- dimensione variabile (per la presenza di strutture dati dinamiche).

**Area di stack**: contiene i **record di attivazione** (**frame**) delle procedure già chiamate ma non ancora terminate. L'indirizzo della cima (top) dello stack, dove è situato il record di attivazione della procedura corrente, è contenuto nel registro di controllo SP (Stack Pointer). Ciascun record di attivazione contiene varie informazioni, tra cui:

- i *parametri attuali*;
- le *variabili locali*;
- il **return address** (RA), cioè l'indirizzo della prossima istruzione da eseguire al termine della procedura (ottenuto salvando il valore del PC al momento della chiamata);
- un puntatore alla cima del record di attivazione della procedura chiamante.

L'area di stack ha:

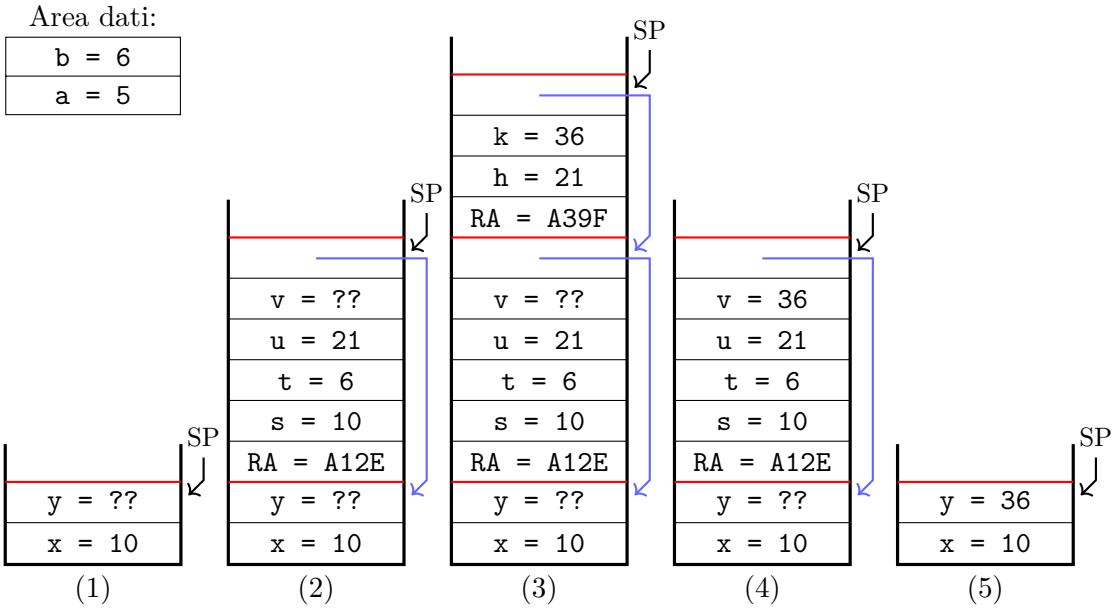
- contenuto variabile (per gli assegnamenti alle variabili locali);
- dimensione variabile (quando vengono chiamate o terminano le procedure).

## 2 Esempio: esecuzione di un programma in C

```c
#include <stdio.h>

// Variabili globali -> area dati
int a = 5;
int b = 6;

int f2(int h) {
    int k = h + 15; // Variabile locale
    // (punto 3)
    return k;
}

int f1(int s, int t) {
    int u = a + s + t; // Variabile locale
    // (punto 2)
    int v = f2(u); // Variabile locale
    // (punto 4)
    return v;
}

int main(void) {
    int x = 10; // Variabile locale
    // (punto 1)
    int y = f1(x, b); // Variabile locale
    // (punto 5)
    printf("ecco il valore di y: %d\n", y);
}
```

Per questo esempio, si assume che:

- gli indirizzi sono a 16 bit, e quindi rappresentabili con 4 cifre esadecimali;
- nel programma compilato:
  - l'istruzione macchina di `main` che segue la chiamata a `f1` si trova all'indirizzo `A12E`;
  - l'istruzione macchina di `f1` che segue la chiamata a `f2` si trova all'indirizzo `A39F`.




1. In questo punto del codice, lo stack contiene solo il record di attivazione di `main`, con la variabile locale `x` inizializzata (al valore `10`), mentre `y` non è ancora stata inizializzata[1] (qui indicato con `??`). Invece, le variabili globali `a` e `b`, memorizzate nell'area dati, sono state inizializzate ancora prima dell'esecuzione di `main`.
2. A questo punto, sopra al frame di `main` è stato aggiunto quello di `f1`, nel quale:
   - i valori dei parametri attuali (`x` e `b`) sono stati assegnati ai parametri formali `s` e `t` (copiandoli tali valori dal frame di `main` a quello di `f1`);
   - la variabile `u` è già stata inizializzata, mentre `v` non ancora;
   - il valore del return address è l'indirizzo dell'istruzione macchina di `main` successiva alla chiamata di `f1`;
   - il valore in cima al frame, e quindi alla pila, è l'indirizzo del valore in cima al record di attivazione di `main`.
3. Con la chiamata di `f2`, è stato aggiunto un ulteriore record di attivazione in cima alla pila:
   - al parametro formale `h` è stato assegnato il valore del parametro attuale (`u`);
   - la variabile locale `k` è stata inizializzata;
   - il RA è l'indirizzo dell'istruzione macchina di `f1` successiva alla chiamata di `f2`.

[1] A seconda del linguaggio di programmazione, una variabile non inizializzata può contenere un valore indeterminato, oppure un valore di default.

4. Quando `f2` è terminata:
    - il valore di `k` è stato copiato in `v` (nel frame di `f1`);
    - il record di attivazione di `f2` è stato rimosso logicamente[2] dallo stack, assegnando al registro SP l'indirizzo in cima alla pila (che è l'indirizzo del frame di `f1`);
    - il return address è stato assegnato al PC, facendo riprendere l'esecuzione di `f1` dal punto successivo alla chiamata appena terminata.
5. Essendo terminata anche `f1`, il suo record di attivazione viene rimosso dallo stack, lasciando solo quello di `main`, e il valore restituito è stato assegnato a `y`. L'esecuzione di `main` continuerà dall'istruzione macchina successiva alla chiamata di `f1`.

# 3 Indirizzi di memoria

Il codice di memoria contiene indirizzi di memoria, che si riferiscono:

- all'area di testo (es. istruzioni di salto);
- all'area dati (es. variabili globali);
- all'area di stack (es. variabili locali e parametri delle procedure).

Questi indirizzi non sono però indirizzi reali, bensì **indirizzi virtuali**. Infatti, quando un programma viene scritto/tradotto in linguaggio macchina (dal programmatore o dal compilatore), non si può sapere quali parti della RAM verranno effettivamente assegnate al programma per l'esecuzione. Perciò, il codice viene scritto usando indirizzi virtuali che partono da 0: in *fase di esecuzione*, questi vengono poi tradotti in indirizzi reali dalla MMU. In questo modo, ciascuna delle tre aree di memoria del programma (testo, dati e stack) può essere caricata in qualsiasi luogo della RAM, anche in modo non contiguo.

Se, invece, il codice contenesse indirizzi reali, non servirebbe la MMU, ma in compenso:

- non si potrebbero caricare in memoria due programmi con overlapping di memoria (cioè che hanno alcuni indirizzi in comune), e in particolare più istanze del medesimo programma;
- i compilatori dovrebbero generare indirizzi evitando gli overlapping, il che è praticamente impossibile su computer general purpose, che possono eseguire numerosi programmi provenienti da fonti diverse;
- i programmi non sarebbero portabili.

[2] I contenuti del record di attivazione rimosso rimangono in RAM, ma verranno poi sovrascritti da eventuali chiamate successive (alla stessa o ad altre procedure).

## 3.1 Traduzione degli indirizzi

Supponendo che a ciascun programma venga assegnata un'area contigua di memoria (diversamente da ciò che avviene nei sistemi moderni), un semplice meccanismo per la traduzione degli indirizzi da virtuali a reali è l'uso di un **Relocation Register** (**RR**).

Se al programma in esecuzione è assegnata un'area di memoria che inizia all'indirizzo $a$, il RR assume valore $a$. Tale valore viene poi sommato, a livello hardware, a tutti gli indirizzi ai quali il programma accede: in questo modo, il primo indirizzo virtuale (0) viene tradotto nel primo indirizzo dell'area di memoria reale ($a + 0 = a$), e così via.

In questo caso, la MMU è composta semplicemente dal RR e da un sommatore. Se si implementa anche la protezione della memoria con i registri LBR e UBR, si può usare direttamente LBR come Relocation Register.