Traduzione

martedì 2 aprile 2024 17:59

Autore: Valerio Cannalire
Anno accademico delle lezioni: 2023/2024

Memoria di una funzione

Variabili locali di una funzione: dove vengono allocate in memoria?
In fortran un precompilacompilatore esamina l'intero codice, individua le funzioni e viene allocato dello spazio per ognuna di esse. Lo svantaggio più evidente è lo spreco di spazio: non è detto che le funzioni vengano chiamate contemporaneamente. Altro problema si riscontra nel caso delle funzioni ricorsive poiché solo a runtime si saprà quante volte queste verranno richiamate.

Si può assumere che se una funzione ne chiama un'altra allora terminerà dopo quella chiamata. Ciò permette di poter pensare a una soluzione basata sulla pila, permettendo sia di non dover necessariamente allocare spazio quando una funzione non è in esecuzione e soprattutto permettendo la ricorsione.

Prendendo il caso di una funzione che calcola il fattoriale sfruttando la ricorsione, si vorrebbe che il parametro *n* della chiamata della funzione in esecuzione fosse sempre quello puntato da %RSP.

Lo spazio allocato da ogni chiamata di funzione è detto record di attivazione: in esso vengono allocati i parametri e le variabili locali. Si potrebbe pensare di usare l'indirizzamento indiretto con offset con %RSP per accedere ai parametri e alle variabili locali memorizzate nel record di attivazione, ma siccome questo registro potrebbe essere soggetto a variazioni dovute ad altre operazioni sulla pila, si usa %RBP che viene inizializzato allo stesso punto del record di attivazione. Tra 2 diverse chiamate quindi cambia l'indirizzo contenuto in %RBP mentre gli offset dei contenuti del record di attivazione rimangono invariati.

I parametri, come le variabili locali, vengono allocati nel record di attivazione, solo che vengono inizializzati dal chiamante della funzione. La convenzione non specifica nulla di particolare: parametri e variabili locali possono essere liberamente mescolati all'interno del record di attivazione. Generalmente il compilatore non cambia l'ordine dei campi per cercare di occupare spazio nel modo più efficiente possibile. Il compito spetta al programmatore, perché il compilatore dovendo assicurare l'uguaglianza del programma non si prende la libertà di apportare queste modifiche. Solitamente si allocano prima i parametri e in seguito le variabili locali.

Affinché venga mantenuta la relazione tra chiamante e chiamato, prima dell'allocazione del record di attivazione è necessario implementare un prologo, e in seguito alla deallocazione è necessario un epilogo.

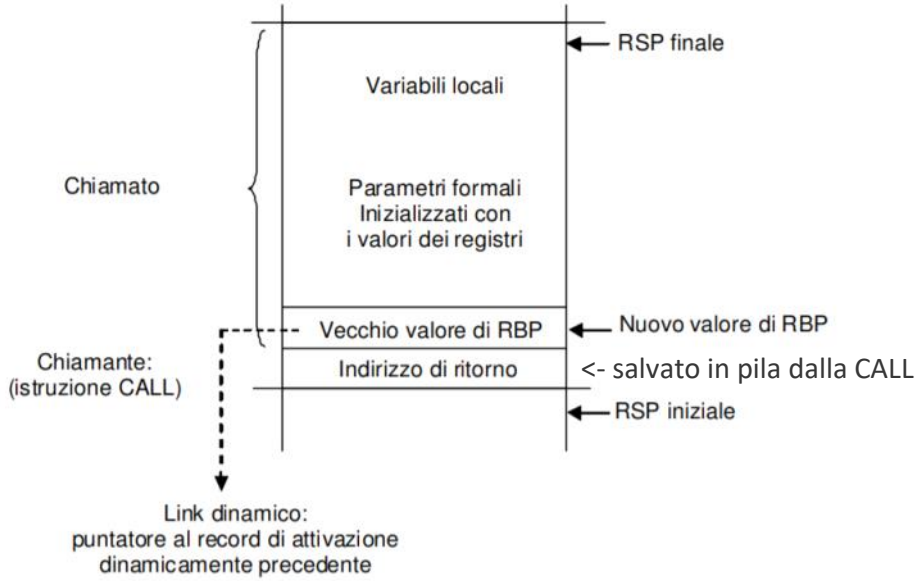
Prologo di una funzione:

```
push %RBP          ← Salvataggio in pila del precedente %RBP
mov %RSP, %RBP     ← Inizializzazione di %RBP
```

Epilogo di una funzione: Deallocamento della pila

```
mov %RBP, %RSP  }
pop %RBP        }  queste due istruzioni possono essere collassate nell'istruzione leave
ret
```

Record di attivazione:



Per visualizzare meglio come vengono allocati i dati nel record di attivazione conviene disegnare lo stack suddiviso in righe di 8 byte/celle. Per motivi di debugging (essendo la macchina little endian) conviene scegliere gli indirizzi crescenti verso sinistra. Infatti le celle vengono riempite a partire da destra scegliendo questo ordine di crescenza degli indirizzi. Quindi così si troveranno nella posizione corretta di lettura di un umano.

Per le dimensioni e gli allineamenti dei tipi standard e di quelli derivati si fa riferimento alle tabelle viste nella lezione 4 "la memoria centrale":

Tipi standard	**sizeof (in byte)**	**alignof**
char	1	1
short	2	2
int	4	4
long	8	8

Tipi derivati	**sizeof (in byte)**	**alignof**
int v[5];	sizeof(int)*5	alignof(int)
struct s{tipo1 a, … tipon b}	multiplo dell'allineamento	Max degli allineamenti dei tipi
Puntatori	8	8
Riferimenti	8	8

Lo spazio per le variabili/parametri viene allocato decrementando %RSP. Siccome non è noto cosa sarà salvato successivamente in pila (potrebbe essere un long), per mantenere l'allineamento di questa %RSP viene incrementato/decrementato di 8 indirizzi per volta. In realtà l'allineamento richiesto dal processore Intel deve essere fatto a multipli di 16 per alcune particolari funzioni che non si vedono durante il corso.

Alla fine della vita della funzione %RSP deve essere nuovamente incrementato dello spazio che precedentemente era stato allocato per variabili e parametri? No, perché l'istruzione *mov %RBP, %RSP* nell'epilogo annulla tutto.

Se una funzione stava operando su dei registri? Devono essere salvati anche loro in pila? Sarebbe ottimo salvare e ripristinare solo i registri che si vogliono usare nella funzione che si chiama e che sono stati utilizzati da una delle funzioni a monte di quelle chiamate. Tuttavia solo il chiamante sa quali registri userà, ignorando ciò che è stato fatto dalle precedenti chiamate e ciò che potrebbe essere fatto dalle successive.

Il compromesso si raggiunge definendo alcuni registri scratch: questi possono essere sovrascritti da ogni chiamata non curandosi delle conseguenze. I restanti registri vengono salvati in pila e poi ripristinati quando il controllo ripassa alla rispettiva chiamata.

Registri scratch: %RAX, %RCX, %RSI, %RDI, %R8, %R9, %R10, %R11

Se devono essere usati registri non scratch all'interno della funzione questi allora vengono salvati in pila in seguito all'allocazione dello spazio per le variabili e i parametri, per poi essere ripristinati prima dell'epilogo.

Come vengono passati i parametri? Nell'architettura a 32 bit si sfruttava la pila per far ciò, mentre invece da quando si è passati ai 64 bit è stata istituita una convenzione che sfrutta i registri per farlo. La convenzione deve essere rispettata da chiamante e chiamato.
L'ordine in cui i parametri vengono passati nei registri è: %RDI, %RSI, %RDX, %RCX, %R8, %R9.
Non è un ordine casuale ma frutto di esperimenti sull'efficienza dei programmi. Indipendentemente dalla dimensione di un parametro è dedicato un intero registro per ognuno di essi. Se i parametri fossero più di 6 allora dovrebbero essere passati attraverso la pila. Nel nostro corso non si vede. I parametri vengono copiati in pila dai registri dopo il salvataggio dei registri non scatch. I parametri vengono passati per valore.

Il valore di ritorno viene passato nel registro %RAX qualora sia 8 byte, mentre viene spezzato in parte alta su %RDX e parte bassa su %RAX qualora sia su 16 byte. Ciò spiega perché il return 0 venga tradotto in XOR %RAX, %RAX. Nel caso il valore di ritorno occupi più di 16 byte allora questo dovrebbe essere passato in pila (non si vede nel nostro corso).

Mangling

L'overloading delle funzioni in C++ è realizzato sfruttando il collegatore. Questo non riconoscerebbe le differenze tra 2 funzioni diverse che hanno lo stesso nome. Quando viene assemblato il programma, le funzioni devono quindi essere identificate da stringhe all'interno delle quali oltre al nome della funzione stessa sono presenti anche i parametri così da indentificare univocamente la coppia funzione-parametri di ingresso. Siccome il tipo di ritorno non differenzia per l'overloading, si ignora. Questo processo di ridenominazione è detto mangling, e le regole che si seguono sono quelle valide per GCC.

La stringa di una funzione è composta da:

"_Z" + numero di caratteri del nome della funzione + nome della funzione + argomenti

Tipi standard

Per ogni argomento di tipo standard è sufficiente specificare una singola lettera:

v ⟹ void, nel caso la funzione non abbia argomenti
i ⟹ int
l ⟹ long
c ⟹ char
s ⟹ short
b ⟹ bool
f ⟹ float
d ⟹ double

w ⟹ wchar_t
a ⟹ signed char
h⟹ unsigned char
t ⟹ unsigned short

j ⟹ unsigned int
m ⟹ unsigned long

Es:
somma(int a, int b) ⟹ _Z5sommaii

Tipi derivati

- Puntatori: "P" deve precedere la lettera che specifica il tipo del puntatore.
  Es:
  foo(int a, int* b) ⟹ _Z3fooiPi
  foo(int** a) ⟹ _Z3fooPPi
- Riferimento: "R" deve precedere la lettera che specifica il tipo del riferimento. Nel caso di riferimenti di puntatori R precede P
  Es:
  foo(int& a) ⟹ _Z3fooRi
  foo(int*& a) ⟹ _Z3fooRPi
- Const (non funzione ma parametro): Ricordarsi che *const* fa riferimento a ciò che viene prima nella firma della funzione tranne quando si trova in cima, caso in cui si riferisce a ciò che immediatamente gli succede. Nell'etichetta "K" deve precedere la lettera che specifica il tipo del parametro. Nel caso dei tipi predefiniti costanti si può omettere K perché tanto siccome i parametri vengono passati per riferimento allora non possono essere modificati. Anche nel caso dei puntatori costanti (non dei puntatori a costante) si può omettere K. K precede ciò che è costante, che sia un tipo semplice o derivato.
  Es:
  foo(const int* a) ⟹ _Z3fooPKi
  foo(int const* a) ⟹ _Z3fooPKi
  foo(int* const a) ⟹ _Z3fooPi
  foo(const int* const a) ⟹ _Z3fooPKi
- Array: vengono implicitamente convertiti in puntatori. Per questo motivo l'eventuale indice passato insieme all'array come argomento non avrebbe alcun effetto.
- Strutture, enumerati: numero di caratteri del nome + nome. Nel caso di questi tipi semplici utente (no classi, puntatori e riferimenti) K può essere omesso.
  Es:
  somma(int a, elem b) ⟹ _Z5sommai4elem
  somma(int a, const elem b) ⟹ _Z5sommai4elem
  somma(int a, const elem* b) ⟹ _Z5sommaiPK4elem

Tipi derivati ripetuti

- S_ => per riferisi al primo tipo utente incontrato nella stringa
- S0_ => per riferisi al secondo
- S1_ => al terzo e così via
  Es:
  foo(elem a, elem b) ⟹ _Z3foo4elemS_
  foo(elem a, miotipo c, elem b, miotipo d) ⟹ _Z3foo4elem7miotipoS_S0_
  foo(elem a, elem* b) ⟹ _Z3foo4elemPS_

Funzioni membro di classi

La stringa di una funzione membro di una classe è composta da:
"_ZN" + numero di caratteri del nome della classe + nome della classe +
numero di caratteri del nome della funzione + nome della funzione + E + argomenti (come sopra)
E rappresenta il puntatore implicito a this.
Nel caso di funzioni membro S_ si riferisce alla classe a cui appartengono.

Le stringhe dei costruttori prevedono "C1" al posto del numero dei carattere e del nome del metodo.
Le stringhe dei distruttori prevedono "D1" al posto del numero dei carattere e del nome del metodo.

Ricordarsi che i costruttori non prevedono alcun ritorno, per cui non sarà necessario passare nulla in %RAX. I costruttori scrivono direttamente in memoria all'indirizzo puntato da this.

Esempio di traduzione (prova scritta del 2024-06-07)

```
////////   cc.h
#include <iostream>
using namespace std;
struct st1 { char vc[4]; };
class cl {
    st1 s; long v[4];
public:
    cl(char c, st1& s2);
    void elab1(st1 s1);
    void stampa() {
        for (int i = 0; i < 4 ;i++) cout << s.vc[i] << ' '; cout << endl;
        for (int i = 0; i < 4; i++) cout << v[i] << ' '; cout << endl << endl; }
};
```

```
////////    es1.cpp
#include "cc.h"
void cl::elab1(st1 s1)
{
    cl cla('k', s1);
    for (int i = 0; i < 4; i++) {
        if (s.vc[i] <= s1.vc[i]) {
            s.vc[i] = cla.s.vc[i]; v[i] = i - cla.v[i];
        }
    }
}
```

```
////////     es1.s
.global _ZN2cl5elab1E3st1

.set this, -8
.set s1, -12
.set cla, -56
.set i, -60

.set s, 0
.set v, 8

_ZN2cl5elab1E3st1:

push %rbp
mov %rsp, %rbp

sub $64, %rsp
mov %rdi, this(%rbp)
mov %esi, s1(%rbp)

lea cla(%rbp), %rdi
mov $'k', %rsi
lea s1(%rbp), %rdx
call _ZN2clC1EcR3st1
```

```
movl $0, i(%rbp)
mov this(%rbp), %rdi
lea cla(%rbp), %rsi

inizio_ciclo:
cmpl $4, i(%rbp)
jge fine_ciclo

movslq i(%rbp), %rcx
mov s1(%rbp, %rcx, 1), %al
cmpb s(%rdi, %rcx, 1), %al
jl fine_if

mov s(%rsi, %rcx, 1), %al
mov %al, s(%rdi, %rcx, 1)

movslq i(%rbp), %rax
sub v(%rsi, %rcx, 8), %rax
mov %rax, v(%rdi, %rcx, 8)

fine_if:
incl i(%rbp)
jmp inizio_ciclo

fine_ciclo:
leave
ret
```