Volání funkcí

30. března 2022

Volání funkcí, nebo obecně podprogramů, obnáší celou řadu činností, jak na straně volajícího, tak i na straně volaného kódu, tj. kódu funkce. V tomto cvičení se zaměříme na obě strany volání. Nutno podotknout, že určitou představu o průběhu volání funkcí jsme si mohli udělat již z předchozích cvičení.

1 Volací konvence na platformě AMD64 a unixových operačních systémech

Volací konvence je postavena na několika málo jednoduchých pravidlech, která si pro naše potřeby můžeme shrnout následovně:

1. argumenty jsou předávány přes registry (`rdi`, `rsi`, `rdx`, `rcx`, `r8`, `r9`), zbývající argumenty přes zásobník (zprava doleva),
2. o odstranění hodnot ze zásobníku se stará volající funkce,
3. registr `al` obsahuje počet argumentů s plovoucí řádovou čárkou,
4. hodnoty na zásobníku jsou zarovnány na 8 B,
5. obsah registrů `rax`, `rdi`, `rsi`, `rdx`, `rcx`, `r8`, `r9`, `r10`, `r11` není při volání funkce zachován (caller-saved registry),
6. obsah registrů `rbx`, `rsp`, `rbp`, `r12`, `r13`, `r14`, `r15` musí být před a po zavolání funkce stejný (callee-saved registry).

Dříve než si ukážeme volání funkce, vytvoříme si pomocnou funkci, kterou budeme volat. Tato funkce má právě jeden argument typu celé číslo, které vypíše na standardní výstup.

```
void printi(int n){
    printf("%i\n", n);
}
```

1.1 Jednoduché volání funkce

Naše pomocná funkce má méně než sedm celočíselných argumentů, proto je její zavolání z assembleru přímočaré, jak lze vidět z následujícího kódu, který zavolá funkci `printi` s hodnotou 42.

```
global show_number
extern printi

section .text
;;
;; Funkce po svem zavolani vypise na standardni vystup hodnotu 42
;;
;; void show_number();
;;
show_number:
        mov edi, 42      ; hodnota predavana jako prvni argument funkci printi
        mov al, 0        ; pocet argumentu s plovouci radovou carkou
        call printi      ; zavolani funkce

        ret              ; navrat z funkce show_number
```

Z kódu je patrné, že stačí nastavit odpovídající registry a funkci zavolat instrukcí `call`. Abychom mohli volat funkci, která je definovaná v jiném zdrojovém kódu (nebo knihovně), musíme použít direktivu `extern symbol`, která udává, že se v daném kódu bude používat funkce (nebo hodnota) definovaná v jiné části programu.[1] Konkrétní adresa bude doplněna ve fázi linkování.

1.2 Mírně složitější volání funkce

Uvažujme funkci, která bude představovat odpočet hodnot, tj. bude na standardní výstup vypisovat hodnoty od $n, n-1, n-2, \ldots, 1, 0$. Její kód by v assembleru mohl vypadat následovně.

```
global final_countdown
extern printi

;;
;; Funkce vypisuje na standardni vystup hodnoty n, n - 1, ..., 0
;;
;; void final_countdown(int n);
;;
final_countdown:
        mov ecx, edi            ; registr ecx obsahuje aktualni vypisovanou hodnotu
countdown_loop:
        mov edi, ecx            ; predame argumenty funkci printi
        mov al, 0
        call printi             ; zavolame funkci printi

        sub ecx, 1              ; snizime hodnotu o 1
```

[1] Dalo by se říct, že `extern` je protipólem direktivy `global`.

```
        jns countdown_loop      ; pokud je vysledek nezaporny, opakujeme

        ret
```

Tato funkce si v registru `ecx` uchovává hodnotu, která se má vypsat, předá ji funkci `printi`, jak jsme vyděli v předchozí kapitole, a následně hodnotu sníží o jedna. Pokud je hodnota nezáporná, smyčka se opakuje. Všimněme si, že zde není použita instrukce `cmp` a využíváme toho, že instrukce `sub` nastavuje příznaky v příznakovém registru.

Nutno zdůraznit, že takto naprogramovaná funkce nejspíš nebude fungovat. Je to dáno tím, že sada registrů je sdílena napříč voláními jednotlivých funkcí. Použitá konvence určuje, že registr `ecx` patří mezi registry, o jejichž uložení se stará volající (caller-saved registr), to znamená, že po zavolání `call printi` se může v registru `ecx` nacházet libovolná hodnota.

Ukažme si tři možná řešení, jak se s tímto problémem vypořádat.

1.2.1 Použití caller-saved registru

Chceme-li zachovat hodnotu v registru `ecx`, musíme ji před zavoláním funkce `printi` někam uložit a po návratu z funkce ji obnovit. Pro tyto účely se nabízí použít zásobník. Ten se pro tyto účely také běžně používá.

```
1  final_countdown:
2          mov ecx, edi             ; registr ecx obsahuje aktualni vypisovanou hodnotu
3  countdown_loop:
4          push rcx                 ; ulozime obsah registru ecx
5
6          mov edi, ecx             ; predame argumenty funkci printi
7          mov al, 0
8          call printi              ; zavolame funkci
9
10         pop rcx                  ; obnovime obsah registru rcx
11
12         sub ecx, 1               ; snizime hodnotu o 1
13         jns countdown_loop       ; pokud je vysledek nezaporny, opakujeme
14
15         ret
```

Tento kód, již zcela funkční, se liší ve dvou řádcích (4 a 10), které se postarají o uložení hodnoty na zásobník a její vrácení do registru `ecx`. Všimněme si, že na zásobník není ukládána 32bitová hodnota (se kterou pracujeme), ale celý 64bitový registr `rcx`, tím je zajištěno, že zásobník je zarovnán na 8 bytů.

1.2.2 Použití callee-saved registru

Další řešení je rezignovat na použití registru `ecx` a použít jiný registr, u nějž bude zajištěno, že jeho obsah bude zachován i po volání funkce. Mezi takové registry patří např. `ebx`. Pozor, pokud takový registr

chceme použít, musíme zajistit, aby i po provedení naší funkce v registru byla stejná hodnota jako před jejím zavoláním. K tomu se obvykle používá zásobník a hodnota tohoto registru je uložena na začátku volání funkce a obnovena na konci, viz následující kód (řádky 2 a 13).

```
1  final_countdown:
2          push rbx                ; ulozime obsah rbx
3          mov ebx, edi            ; registr ebx obsahuje aktualni vypisovanou hodnotu
4
5  countdown_loop:
6          mov edi, ebx            ; predame argumenty funkci printi
7          mov al, 0
8          call printi             ; zavolame funkci
9
10         sub ebx, 1              ; snizime hodnotu o 1
11         jns countdown_loop      ; pokud je vysledek nezaporny, opakujeme
12
13         pop rbx                 ; obnovime obsah registru rbx
14         ret
```

1.2.3 Použití lokální proměnné

Obě dvě dosud zmíněné varianty předpokládaly, že hodnoty, se kterými pracujeme, jsou uloženy v registrech a v případě nutnosti můžeme jejich obsah dočasně uložit na zásobník a následně obnovit. Pokud je hodnot, se kterými pracujeme více, případně potřebujeme předat na ně ukazatel, je nutné použít lokální proměnné. Takové řešení je sice obecnější než ad hoc použití registrů, ale vyžaduje složitější inicializaci funkce, tzv. prolog a odpovídající operace na konci volání funkce, tzv. epilog.

```
1  final_countdown:
2          push rbp                ; ulozime obsah rbp
3          mov rbp, rsp            ; rbp obsahuje adresu ramce na zasobniku
4          sub rsp, 8              ; vytvorime prostor pro jednu lokalni promennou
5
6          mov [rbp - 8], edi      ; [rbp - 8] obsahuje aktualni vypisovanou hodnotu
7
8  countdown_loop:
9          mov edi, [rbp - 8]      ; predame argumenty funkci printi
10         mov al, 0
11         call printi             ; zavolame funkci
12
13         sub dword [rbp - 8], 1  ; snizime hodnotu o 1
14         jns countdown_loop      ; pokud je vysledek nezaporny, opakujeme
15
16         mov rsp, rbp            ; odstranime lokalni promenne
```

```
17          pop rbp                 ; obnovime hodnotu rbp
18          ret
```

V prologu funkce nejdříve uložíme na zásobník hodnotu registru `rbp` (řádek 2), ten budeme používat jako ukazatel na místo v zásobníku, kde jsou lokální proměnné. Proto do tohoto registru uložíme hodnotu `rsp` (řádek 3) a následně posuneme vrchol zásobníku o 8 bytů níž (řádek 4), čímž vytvoříme místo na zásobníku. Nyní platí, že máme na zásobníku místo pro jednu proměnnou o velikosti až 8 bytů a její adresa je dána jako `rbp - 8`. Chceme-li pak s touto hodnotou pracovat, místo registru uvedeme odkaz na místo v paměti.

V závěru funkce, tzv. epilogu, provedeme odstranění hodnot ze zásobníku (řádek 16) tím, že posuneme vrchol zásobníku na místo, kam ukazoval před vytvoření prostoru pro lokální proměnné a obnovíme hodnotu v registru `rbp`.