

# Volání funkcí

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 v následujících bodech:

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 a1 obsahuje počet argumentů s plovoucí řádovou čárkou,
4. hodnoty na zásobníku jsou zarovnané 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 naše pomocná funkce 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 prni 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 poté 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.<sup>1</sup> 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  $n, n - 1, n - 2, \dots, 1, 0$ . 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
    jns countdown_loop ; pokud je vysledek nezaporny, opakujeme

    ret

```

---

<sup>1</sup>Dalo by se říct, že `extern` je protipólem direktivy `global`.

Tato funkce si v registru `ecx` uchovává hodnotu, která se má vypsat, předá ji funkci `printi`, jak jsme viděli 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                ; uložíme obsah rbx
3     mov ebx, edi           ; registr ebx obsahuje aktuální vypisovanou hodnotu
4
5     countdown_loop:
6     mov edi, ebx           ; předáme argumenty funkci printi
7     mov al, 0
8     call printi           ; zavoláme funkci
9
10    sub ebx, 1              ; snížíme hodnotu o 1
11    jns countdown_loop    ; pokud je výsledek nezáporný, opakujeme
12
13    pop rbx                ; obnovíme 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 hodnota, se kterou 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. epilóg.

```

1 final_countdown:
2     push rbp                ; uložíme obsah rbp
3     mov rbp, rsp           ; rbp obsahuje adresu ramce na zásobníku
4     sub rsp, 8             ; vytvoříme prostor pro jednu lokální proměnnou
5
6     mov [rbp - 8], edi     ; [rbp - 8] obsahuje aktuální vypisovanou hodnotu
7
8     countdown_loop:
9     mov edi, [rbp - 8]    ; předáme argumenty funkci printi
10    mov al, 0
11    call printi           ; zavoláme funkci
12
13    sub dword [rbp - 8], 1 ; snížíme hodnotu o 1
14    jns countdown_loop    ; pokud je výsledek nezáporný, opakujeme
15
16    mov rsp, rbp          ; odstraníme lokální proměnné
17    pop rbp              ; obnovíme 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 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`.

## 2 Poznámky

### 2.1 Zarovnání zásobníku

Specifikace Linuxového ABI<sup>2</sup> vyžaduje, aby oblast zásobníku, kde jsou uloženy argumenty byla zarovnána na 16 bytů, tj. aby hodnota v registru `rsp` byla před provedením instrukce `call` násobkem 16. To má dva důsledky. (i) Na začátku provádění funkce není hodnota v registru `rsp` zarovnána na násobek 16, protože je na zásobník uložena návratová adresa. (ii) Před zavoláním funkce, bychom měli upravit vrchol zásobníku tak, aby hodnota v registru `rsp` byla násobkem 16. V případě některých Linuxových distribucí toto zarovnání není explicitně vynucováno.

### 2.2 Překlad

Některé Linuxové distribuce překládají programy tak, aby mohly být umístěny na libovolné místo v paměti, jako tzv. *position independent executable (PIE)*, to částečně ovlivňuje, jak jsou volány funkce. Abychom mohli volat funkce stylem `call foo`, je v takovém případě nutné použít při překladu přepínač `-no-pie`, tj. `gcc -no-pie -o foo foo.o bar.o`.

## 3 Úkoly k procvičení

Napište v assembleru následující funkce:

1. Napište funkci `void print_row(int n, char c)`, která s pomocí volání funkce `putchar` vypíše na standardní výstup řádek skládající se z `n` opakování znaku `c`. Výpis by měl být ukončen znakem `\n`.
2. Napište funkci `void print_rect(int rows, int cols)`, která s pomocí volání funkce `print_row` vykreslí na standardní výstup vyplněný obdélník skládající se ze znaků `*` mající `rows` řádků a `cols` sloupců.
3. Napište funkci `unsigned int factorial(unsigned int n)`, která rekurzivním způsobem spočítá hodnotu faktoriálu.
4. Napište funkci `char *my_strdup(char *s)`, která vytvoří kopii řetězce `s`. Použijte volání funkcí `malloc` a `strlen`.

---

<sup>2</sup>[https://refspecs.linuxbase.org/elf/x86\\_64-abi-0.99.pdf](https://refspecs.linuxbase.org/elf/x86_64-abi-0.99.pdf)

5. Napište funkci `unsigned int fib(unsigned short n)`, která rekurzivně vypočítá hodnotu n-tého fibonacciho čísla.
6. Napište funkci `void print_facts(unsigned char n)`, která vypíše prvních n hodnot faktoriálu s pomocí volání `printf` a `factorial`.