Operační systémy 1

6. Cvičení

Radek Janoštík

KATEDRA
INFORMATIKY
UNIVERZITA PALACKÉHO V OLOMOUCI

Úkoly z minula – jak to šlo?

- Kolik se vám toho podařilo udělat?

```
1 void swap(int *a, int *b)
2 void division(unsigned int x, unsigned int y, unsigned int
  *result, unsigned int *remainder)
3 void countdown(int *values)
4 void nasobky(short *multiples, short n)
5 int minimum(int count, int *values)
6 unsigned int my_strlen(char *s)
7 void my_strcat(char *dest, char *src)
```

- Dalo něco zabrat?
- Nějaké dotazy?

Úkoly z minula – jak to šlo?

- Kolik se vám toho podařilo udělat?

```
1 void swap(int *a, int *b)
2 void division(unsigned int x, unsigned int y, unsigned int
  *result, unsigned int *remainder)
3 void countdown(int *values)
4 void nasobky(short *multiples, short n)
5 int minimum(int count, int *values)
6 unsigned int my_strlen(char *s)
7 void my_strcat(char *dest, char *src)
```

- Dalo něco zabrat?
- Nějaké dotazy?
- Prezenčka (až přijdou všichni :))

Volání funkcí

- Funkce/podprogramy pomáhají se znovupoužitelností kódu
- Umožňují také abstrakce problémů (obecnější řešení použité v konkrétním případě)
- Zatím „jsme byli“ v roli volaného kódu
  - Řešili jsme, které registry *můžeme použít*
  - Některé registry jsme museli *vracet do původního stavu*
- Při voláni funkcí naopak může registry *někdo změnit nám*

Volání funkcí

- Funkce/podprogramy pomáhají se znovupoužitelností kódu
- Umožňují také abstrakce problémů (obecnější řešení použité v konkrétním případě)
- Zatím „jsme byli“ v roli volaného kódu
  - Řešili jsme, které registry *můžeme použít*
  - Některé registry jsme museli *vracet do původního stavu*
- Při voláni funkcí naopak může registry *někdo změnit nám*
- $\Rightarrow$ Potřeba striktně dodržovat **volací konvence**
  - Nedodržení povede k chybám

Volací konvence platformy AMD64 a unixových OS

1. Argumenty jsou předávány přes registry (`rdi, rsi, rdx, rcx, r8, r9`), zbývající argumenty přes zásobník (zprava doleva).

Volací konvence platformy AMD64 a unixových OS

1. Argumenty jsou předávány přes registry (`rdi, rsi, rdx, rcx, r8, r9`), zbývající argumenty přes zásobník (zprava doleva).
2. O odstranění argumentů ze zásobníku se stará volající funkce.

Volací konvence platformy AMD64 a unixových OS

1. Argumenty jsou předávány přes registry (`rdi, rsi, rdx, rcx, r8, r9`), zbývající argumenty přes zásobník (zprava doleva).
2. O odstranění argumentů ze zásobníku se stará volající funkce.
3. Registr `al` obsahuje počet argumentů s plovoucí řádovou čárkou.

Volací konvence platformy AMD64 a unixových OS

1. Argumenty jsou předávány přes registry (`rdi`, `rsi`, `rdx`, `rcx`, `r8`, `r9`), zbývající argumenty přes zásobník (zprava doleva).
2. O odstranění argumentů 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.

Volací konvence platformy AMD64 a unixových OS

1. Argumenty jsou předávány přes registry (`rdi, rsi, rdx, rcx, r8, r9`), zbývající argumenty přes zásobník (zprava doleva).
2. O odstranění argumentů 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).

Volací konvence platformy AMD64 a unixových OS

1. Argumenty jsou předávány přes registry (`rdi, rsi, rdx, rcx, r8, r9`), zbývající argumenty přes zásobník (zprava doleva).
2. O odstranění argumentů 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).

Jednoduché volání funkce

- Mějme funkci

```
1      void printi(int n){
2        printf("%i\n", n);
3      }
```

- Základní volání

```
1  global show_number
2  extern printi
3
4  section .text
5  ;; Funkce po svem zavolani vypise na standardni vystup hodnotu 42
6  ;;
7  ;; void show_number();
8  show_number:
9    mov edi, 42 ; hodnota prvniho argumentu
10   mov al, 0   ; pocet argumentu s plovouci radovou carkou
11   call printi ; zavolani funkce
12   ret         ; navrat z funkce show_number
```

Problém změny hodnoty v registrech

- Mějme (nejspíš nefunkční) příklad:

```
1  global final_countdown
2  extern printi
3  ;;
4  ;; Funkce vypisuje na standardni vystup hodnoty n, n - 1, ..., 0
5  ;;
6  ;; void final_countdown(int n);
7  ;;
8  final_countdown:
9      mov ecx, edi ; ecx obsahuje aktualni vypisovanou hodnotu
10 countdown_loop:
11     mov edi, ecx ; predame argumenty funkci printi
12     mov al, 0
13     call printi  ; zavolame funkci printi
14     sub ecx, 1   ; snizime hodnotu o 1
15     jns countdown_loop ; pokud je vysledek nezaporny, opakujeme
16     ret
```

Uložení hodnoty caller-saved registru

- Hodnotu v `ecx` uložíme na zásobník a po volání obnovíme

```
1  final_countdown:
2      mov ecx, edi ; 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     ret
```

- Na zásobník se vždy ukládá celý registr – 64 bitů

Použití callee-saved registru

- Místo `ecx` použijeme `ebx` ⇒ musíme zachovat jeho původní hodnotu

```
1  final_countdown:
2      push rbx       ; ulozime obsah rbx
3      mov ebx, edi   ; 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
```

Použitích lokálních proměnných

- Hodnoty byly v registrech $\Rightarrow$ dodrželi jsme konvence volání
- Při více hodnotách zdlouhavé, snadno se udělá chyba
  - Nemožnost předat ukazatel na hodnotu
- $\Rightarrow$ použití lokálních proměnných
  - Obecnější řešení
  - Přesná pravidla, složitější inicializace funkce – *prolog*
  - Uvedení všeho do původního stavu na konci funkce – *epilog*

Prolog

- Uložíme původní hodnotu registru `rbp` (register base pointer), fixní hodnota, neposunuje jej push a pop
- Do `rbp` uložíme současnou hodnotu `rsp` (register stack pointer)
- A vytvoříme prostor pro lokální proměnné (velikost vždy 8 B)

```
1 final_countdown:
2     push rbp      ; ulozime obsah rbp
3     mov rbp, rsp ; rbp obsahuje adresu ramce na zasobniku
4     sub rsp, 8    ; prostor pro jednu lokalni promennou
5
6     mov dword [rbp - 8], edi ; [rbp - 8] obsahuje aktualni
          vypisovanou hodnotu
```

Epilog

- Na konci funkce „odstraníme lokální proměnné“ (pouze posuneme ukazatel)
- Obnovíme původní hodnotu `rbp`

```
1 mov rsp, rbp ; odstranime lokalni promenne
2 pop rbp      ; obnovime hodnotu rbp
3 ret          ; návrat z funkce
```

Celá funkce final_countdown

```
1  final_countdown:
2      push rbp      ; ulozime obsah (puvodni) rbp
3      mov rbp, rsp  ; rbp obsahuje adresu ramce na zasobniku
4      sub rsp, 8    ; vytvorime prostor pro jednu lokalni promennou
5
6      mov dword [rbp - 8], edi ; [rbp - 8] obsahuje aktualni
           vypisovanou hodnotu
7
8  countdown_loop:
9      mov edi, dword [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
```

Poznámky

- Specifikace linuxového ABI vyžaduje zarovnání argumentů funkce na 16 bajtů
  - $\Rightarrow$ hodnota `rsp` před voláním `call` násobkem 16.
- Důsledky:
  - Na začátku funkce **není** `rsp` násobkem 16 (uložena návratová adresa)
  - Před zavoláním bychom měli upravit vrchol zásobníku aby byl `rsp` násobkem 16.
- Pro využívání volání funkcí stylem `call foo` je potřeba vypnout *position independent executable (PIE)*
  - `gcc -no-pie -o foo foo.o bar.o`.

Úkol (1/2)

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.

Úkol (2/2)

4. Napište funkci `char *my_strdup(char *s)`, která vytvoří kopii řetězce s. Použijte volání funkcí malloc a strlen.

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 printi a factorial.