KATEDRA
INFORMATIKY
UNIVERZITA PALACKÉHO V OLOMOUCI

Přístup do paměti

KMI/OS1 Operační systémy I

Mgr. Markéta Trnečková, Ph.D.
www.marketa-trneckova.cz

Příklad z minula

Příklad

Napište funkci `void division(unsigned int x, unsigned int y, unsigned int *result, unsigned int *remainder)`, která celočíselně vydělí hodnotu `x` hodnotou `y` a výsledek uloží na místo v paměti dané ukazatelem `result` a zbytek po dělení uloží do paměti dané ukazatelem `remainder`.

Pozor!

- Argumenty funkci jsou předávány registry: `edi` (`x`), `esi` (`y`), `rdx` (ukazatel `result`) a `rcx` (ukazatel `remainer`)
- Při dělení ale používáme registr `rdx`, ten je potřeba vynulovat
- Před tím, ale musíme hodnotu někam uložit.
- Po dělení `edx` obsahuje zbytek po dělení, `eax` obsahuje podíl. Tyto hodnoty chceme uložit na místa daná ukazatelі`result` a `remainder`.

Příklad z minula

Řešení

Příklad

```
; funkce division
division:
    mov eax, edi
    mov r8, rdx ; rdx obsahuje 3. argument
    mov rdx, 0
    idiv esi
    mov dword [r8], eax
    mov dword [rcx], edx
    ret
```

Volání funkcí

Učební text ke cvičení:
`https://phoenix.inf.upol.cz/~krajcap/courses/2025LS/OS1/tutorial06.pdf`

Volání funkcí

- Volání funkcí obnáší celou řadu činností, jak na straně volajícího kódu, tak i na straně volaného kódu
- Zaměříme se na volání na platformě AMD64 a unixových operačních systémech

Volání funkcí

Důležité body

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 8B
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**).

Volání funkcí

Příklad

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

- funkce má méně než 7 celočíselných argumentů, její volání je tedy snadné
- zavoláme ji s hodnotou 42

Příklad

```
global show_number
extern printi
section .text
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
```

Volání funkcí

Příklad

```
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
```

- musíme nastavit příslušné registry (zde jen `edi` pro první argument a `al` s počtem argumentů s plovoucí čárkou)
- funkci voláme `call`
- to, že je funkce definovaná v jiném souboru dáme kódu vědět direktivou `extern`

Volání funkcí

- V asm definujeme funkci, která bude do na výstup vypisovat hodnoty `n`, `n-1`, ..., `0`

Příklad

```
global final_countdown
extern printi
;; 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
```

Příklad

Jak bude vypadat část kódu v .c?

Volání funkcí

Příklad

```
global final_countdown
extern printi
;; 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
```

- **Funguje kód?** Proč ne?
- registry jsou sdíleny napříč voláními jednotlivých funkcí
- 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

Volání funkcí

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.

Příklad

```
final_countdown:
    mov ecx, edi ; registr ecx obsahuje aktualni vypisovanou hodnotu
countdown_loop:
    push rcx     ; ulozime obsah registru ecx
    mov edi, ecx ; predame argumenty funkci printi
    mov al, 0
    call printi ; zavolame funkci
    pop rcx      ; obnovime obsah registru rcx
    sub ecx, 1 ; snizime hodnotu o 1
    jns countdown_loop ; pokud je vysledek nezaporny, opakujeme
    ret
```

Na zásobník není ukládána 32bitová hodnota (se kterou pracujeme), ale celý 64bitový registr `rcx`, tím je zajištěno, že zásobník je zarovnán na 8 bytů

Volání funkcí

Použití callee-saved registru

- Použijeme jiný registr, u nějž bude zajištěno, že jeho obsah bude zachován i po volání funkce. 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.

Příklad

```
final_countdown:
    push rbx ; ulozime obsah rbx
    mov ebx, edi ; registr ebx obsahuje aktualni vypisovanou hodnotu
countdown_loop:
    mov edi, ebx ; predame argumenty funkci printi
    mov al, 0
    call printi ; zavolame funkci
    sub ebx, 1 ; snizime hodnotu o 1
    jns countdown_loop ; pokud je vysledek nezaporny, opakujeme
    pop rbx ; obnovime obsah registru rbx
    ret
```

Volání funkcí

Použití lokální proměnné

- Předpokládali jsme, že jsou hodnoty uloženy v registrech
- Obsah registrů jsme případně ukládali dočasně na zásobník
- Pokud, ale potřebujeme předat více hodnot (ukazatel na ně), musíme použít lokální proměnné
- Toto řešení je obecnější, ale složitější
- Vyžaduje inicializaci funkce – **prolog**
- Vyžaduje provedení několika operací na konci – **epilog**

Volání funkcí

Použití lokální proměnné

Příklad

```
final_countdown:
    push rbp ; ulozime obsah rbp
    mov rbp, rsp ; rbp obsahuje adresu ramce na zasobniku
    sub rsp, 8 ; vytvorime prostor pro jednu lokalni promennou
    mov dword [rbp-8],edi ; [rbp-8] obsahuje aktualni vypisovanou hodnotu
countdown_loop:
    mov edi, dword [rbp-8] ; predame argumenty funkci printi
    mov al, 0
    call printi ; zavolame funkci
    sub dword [rbp-8],1 ; snizime hodnotu o 1
    jns countdown_loop ; pokud je vysledek nezaporny, opakujeme
    mov rsp, rbp ; odstranime lokalni promenne
    pop rbp ; obnovime hodnotu rbp
    ret
```

Volání funkcí

Použití lokální proměnné

Prolog:

- Uložíme na zásobník hodnotu registru `rbx`
- Budeme ho používat jako ukazatel na místo na zásobníku
- Proto do něj uložíme hodnotu registru `rsp`
- Posuneme vrchol zásobníku o 8 bytů, čímž si vytvoříme místo pro 1 (lokální) proměnnou
- Adresa této proměnné je `rbp - 8`
- Pokud s touto proměnnou chceme pracovat, místo registru používáme ukazatel na tuto paměť

Epilog:

- Odstraníme hodnoty na zásobníku tím, že posuneme vrchol zásobníku na místo, kam ukazoval před vytvořením prostoru pro lokální proměnnou
- obnovíme hodnotu registru `rbp`

Volání funkcí
Závěrečné poznámky

- Specifikace Linuxového ABI2 vyžaduje, aby oblast zásobníku, kde jsou uloženy argumenty byla zarovnána na 16 bytů
- Tj. hodnota v registru `rsp` můsí být před provedením instrukce `call` násobkem 16
- Proto do něj uložíme hodnotu registru `rsp`
- Důsledky:
  - Na začátku provádění funkce není hodnota v registru `rsp` zarovnána na násobek 16, protože je na zásobník uložena návratová adresa (8 bytů).
  - 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.

Volání funkcí

Závěrečné poznámky

- 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`

```
gcc -no-pie -o foo foo.o bar.o
```

Volání funkcí

Úkoly k procvičení

Všechny následující funkce naprogramujte v assembleru a voláním z jazyka C ověřte, že fungují dle očekávání.

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 `unsinged 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`.

Přístup do paměti

Úkoly k procvičení

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`.