x86 assembly – skoky, funkce a syscally

Milan Radojčić

SSPŠaG – KBB

5. května 2026

Hlavní body

Skoky

Funkce

Systémová volání

Hlavní body

Instrukce `jmp`

- Nepodmíněný skok – vždy skočí na daný label
- Programu, který nikdy neskončí:

```
    segment .text
    global main
main:
    jmp main
```

- Assembler z této instrukce vygeneruje relative jump instrukci, ve které je offset o kolik se má skočit.

Podmíněné skoky

- Skoky které se vykonají jen pokud je nastavená příslušná flaga v registru `rflags`.
- Velice užitečná v této sekci bude instrukce `cmp`.
  - Odečte první argument od druhého a nastaví správné flagy (overflow, carry, zero, sign).
  - Výsledek nikam neukládá. Hodnota nás totiž nezajímá, chceme jen nastavit správný flag.

Typy podmíněných skoků

Instrukce	Význam	Aliasy	Flagy
jz	jump if zero	je	ZF = 1
jnz	jump if not zero	jne	ZF = 0
jg	jump if > zero	jnle	ZF = 0, SF = 0
jge	jump if ≥ zero	jnl	SF = 0
jl	jump if < zero	jnge js	SF = 1
jle	jump if ≤ zero	jng	ZF = 1 nebo SF = 1
jc	jump if carry	jb jnae	CF = 1
jnc	jump if not carry	jae jnb	CF = 0

ZF = zero flag, SF = sign flag, CF = carry flag

Jednoduchý `if` v assembly

Kód v C:

```c
if (a < b) {
    temp = a;
    a = b;
    b = temp;
}
```

Kód v assembly:

```
    mov rax, [a]
    mov rbx, [b]
    cmp rax, rbx
    jge in_order
    mov [temp], rax
    mov [a], rbx
    mov [b], rax
in_order:
```

if/else

Kód v C:

```
if (a < b) {
    max = b;
} else {
    max = a;
}
```

Kód v assembly:

```
    mov rax, [a]
    mov rbx, [b]
    cmp rax, rbx
    jnl else
    mov [max], rbx
    jmp endif
else:
    mov [max], rax
endif:
```

if/if-else/else

```
if (a < b) {
    result = 1;
} else if (a > c) {
    result = 2;
} else {
    result = 3;
}
```

```
    mov rax, [a]
    mov rbx, [b]
    cmp rax, rbx
    jnl else_if
    mov qword [result], 1
    jmp endif
else_if:
    mov rcx, [c]
    cmp rax, rcx
    jng else
    mov qword [result], 2
    jmp endif
else:
    mov qword [result], 3
endif:
```

while loop

Kód v C:

```
while (x < 10) {
    x = x + 1;
}
```

Kód v assembly:

```
    mov rcx, [x]
loop:
    cmp rcx, 10
    jge end
    add rcx, 1
    jmp loop
end:
    mov [x], rcx
```

for loop

Kód v C:

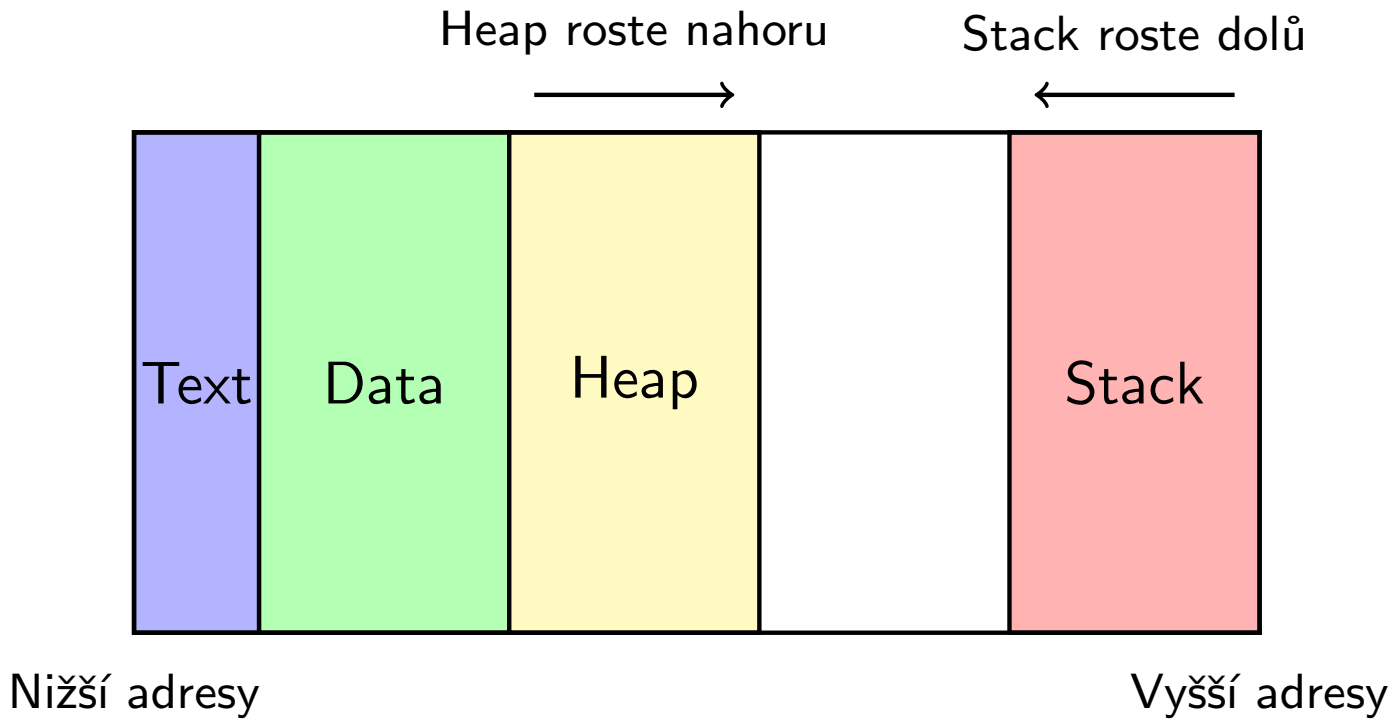
```
for (int i = 0; i < 10;
 ↪  i++) {
    a[i] = 0;
}
```

Kód v assembly:

```
    mov rcx, 0
loop:
    cmp rcx, 10
    jge end
    mov [a + rcx], 0
    add rcx, 1
    jmp loop
end:
```

Hlavní body

Rozdělení paměti



Pomocí příkazu `cat /proc/$$/maps` se můžeme podívat, jak jsou namapované různé sekce paměti spuštěného shellu.

Stack

- Stack je li-fo (last in – first out) struktura, do které můžeme ukládat hodnoty.
- Máme dva speciální registry na správu stacku:
  - `rbp` – stack base pointer – ukazuje na začátek stacku (nejvyšší adresa která patří na stack)
  - `rsp` – stack pointer – ukazuje na hořejšek stacku

Instrukce push a pop

- Instrukce `push` odečte 8 od `rsp` a uloží danou hodnotu na adresu, kam teď ukazuje `rsp`.

```
push 7 ; uloží hodnotu 7 na stack
```

- Instrukce `pop` funguje opačně: hodnotu, co je uložena v `rsp`, uloží do zadaného místa a sníží `rsp` o 8.

```
pop rax ; uloží hodnotu ze stacku do rax
```

Instrukce `call` a `ret`

- `call` pushne na stack adresu příští instrukce a předá kontrolu kódu podle předaného labelu.

```
call my_function ; zavolá funkci my_function
```

- Předchozí kód je stejný jako tento kód:

```
    push next_instruction
    jmp my_function
next_instruction:
```

- Instrukce `ret` popne hodnotu ze stacku a vrátí se na adresu, která je tam uložená.

Předávání parametrů

- Funkcím chceme většinou předávat nějaké parametry a chceme, aby nám také mohly nějaké data vracet.
- Na předávání parametrů existují různé konvence:
  - Linux – System V Application Binary Interface
  - Windows – Microsoft x64 Calling Convention
- Linux i Windows – Funkce vracejí hodnoty v registru `rax` (`xmm0` pro floating-point čísla).
- Linux
  - Prvních 6 číselných parametrů je předáno v registrech: `rdi`, `rsi`, `rdx`, `rcx`, `r8` a `r9`.
  - Ostatní parametry jsou pushnuty na stack.

Caller-saved a callee-saved registry

- Když zavoláme funkci tak stav všech registrů zůstává stejný.
- Nastává problém: co když zavolaná funkce chce použít nějaký registr, ale neví, jestli v něm nemá volající funkce uloženou hodnotu, kterou potřebuje.
- Řešení: registry jsou rozděleny na caller-saved a callee-saved:
  - **caller-saved** – volající kód má zodpovědnost si uložit hodnotu v registru někam stranou, pokud ji potřebuje
  - **callee-saved** – kód, který je zavolán má zodpovědnost v registru hodnotu uchovat
  - registry `rbx`, `rsp`, `rbp` a `r12`-`r15` jsou callee-saved, ostatní jsou caller saved

Caller-saved

```
mov rax, 1
mov rbx, 2
add rax, rbx
push rax
call my_function
pop rax
add rax, 3
```

```
my_function:
    mov rax, 6
    ret
```

Callee-saved

```
mov rdi, 1
mov rbx, 2
add rdi, rbx
call my_function
add rdi, 3
```

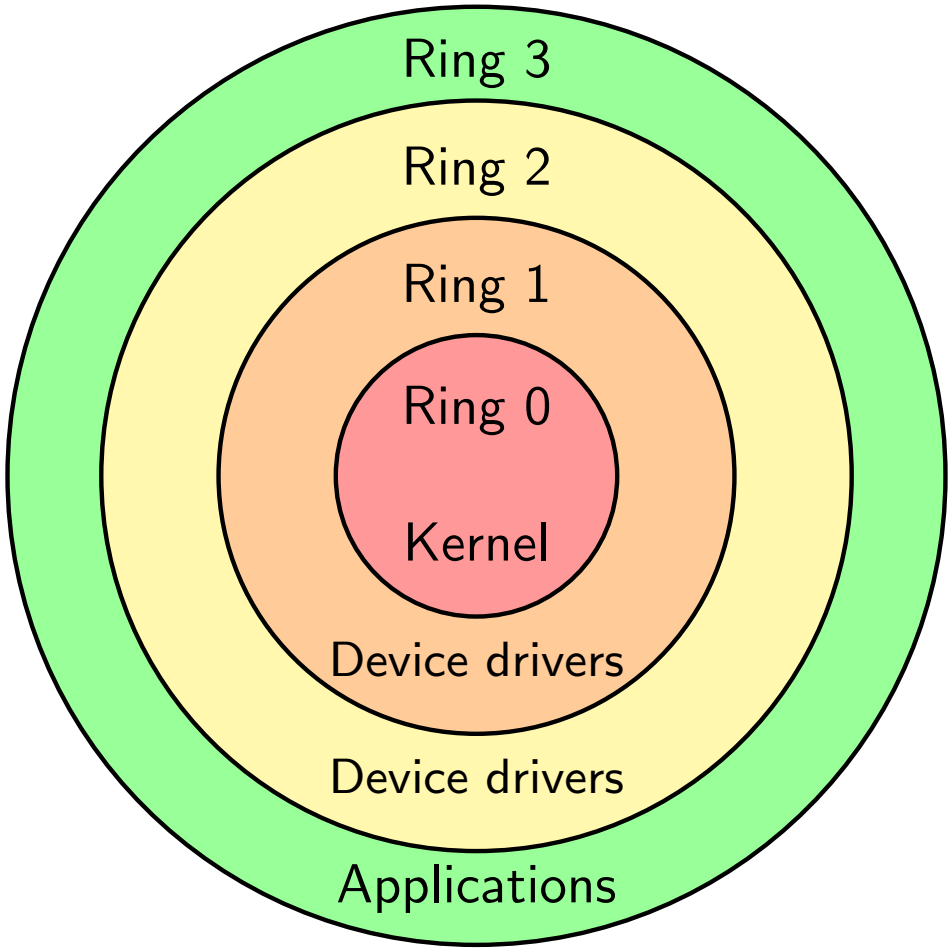
```
my_function:
    push rdi
    mov rdi, 1
    mov rax, 3
    add rax, rdi
    pop rdi
    ret
```

Hlavní body

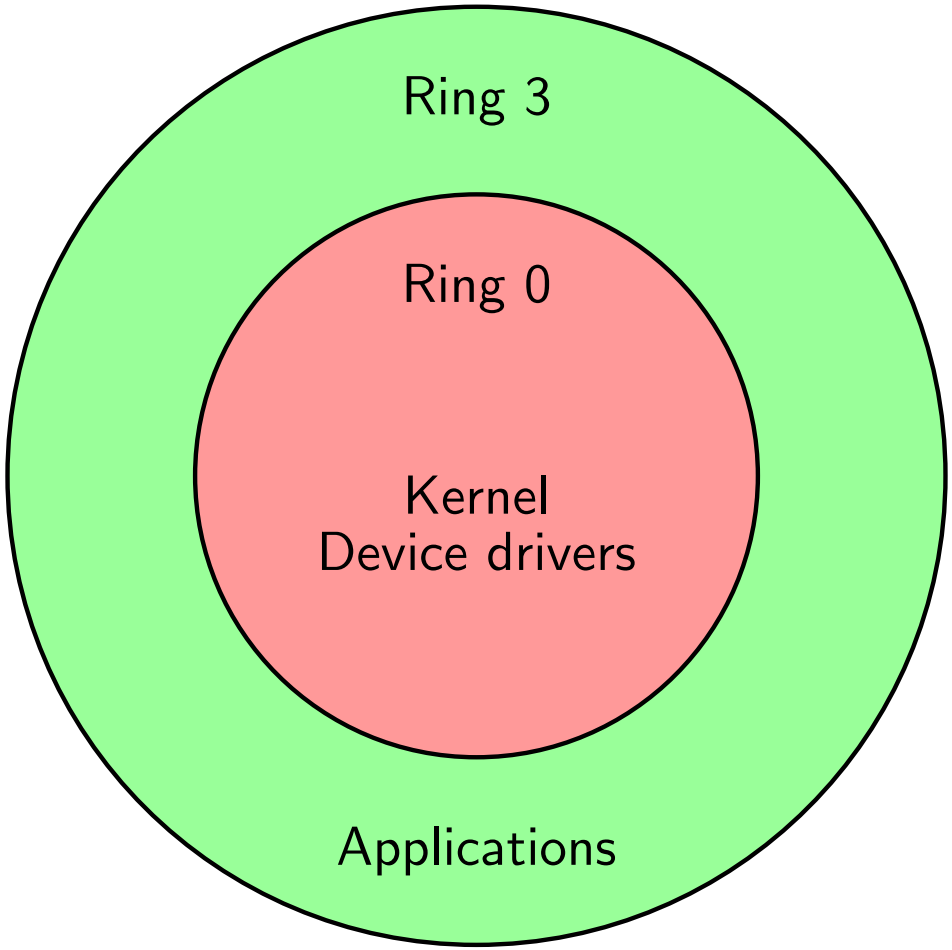
Protection rings

- Většina operačních systémů odděluje kód, který běži jako součást kernelu (kernel mode) a jako uživatelská aplikace (user mode).
- Kód běžící v user modu, nemá stejná práva a nemůže provádět stejné věci jako v kernel modu.
- Tato funkcionalita musí být podporována hardwarem, z tohoto důvodu jsou součástí architektury x86 **protection rings**.

Protection rings



Protection rings – nejčastější implementace v OS



Protection rings

- Operace které nemůžeme provádět v ringu 3 jsou například:
  - přístup k IO zařízením (včetně přímého zápisu na disk atd.)
  - nastavit (přímo) současný ring
  - přistupovat k fyzické paměti

Systémová volání

- Pomocí systémových volání provádíme skrz OS operace, na které nemáme v user modu právo.
- Často to jsou věci, u kterých musí OS ověřit nějaké práva, aby operaci povolil.
  - Např. přístupová práva u souboru.
- Systémová volání se provádí pomocí instrukce `syscall`.
- Do registru `rax` se uloží číslo syscallu a do registrů `rdi`, `rsi`, `rdx`, `r10`, `r8` a `r9` jsou umístěny parametry.
- Návratová hodnota je poté uložena v registru `rax`.

Hello World

```
    segment .data
hello:
    db "Hello World!", 0x0a ; 0x0a je newline
    segment .text
    global _start
_start:
    mov eax, 1 ; 1 je syscall write
    mov edi, 1 ; file descriptor (stdout)
    lea rsi, [hello] ; co zapsat
    mov edx, 13 ; kolik bytu zapsat
    syscall ; write(1, hello, 13)
    mov eax, 60 ; syscall 60 je exit
    xor edi, edi ; uloží 0 do edi
    syscall ; exit(0)
```

Hello World – stejný kód v C

```c
#include <unistd.h>

int main() {
    write(1, "Hello World!\n", 13);
    _exit(0);
}
```