Přístup do paměti

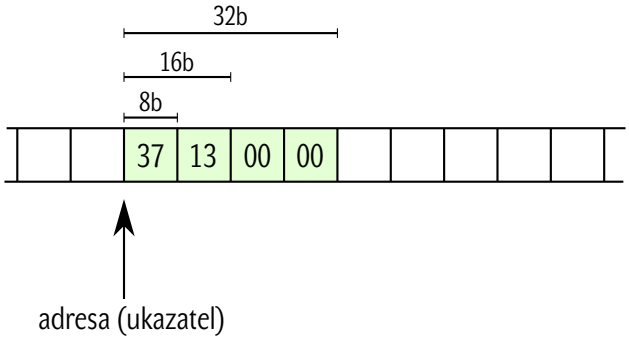
5

Přístup do paměti, ať už se jedná o čtení či zápis dat, patří k nejzásadnějším operacím, které procesor nabízí. V předchozích cvičeních jsme však tyto operace upozadili, abychom snížili učící křivku. To však je aplikovatelné pouze v omezeném množství situací a porozumění práci s pamětí je důležité nejen pro porozumění činnosti procesoru ale i vyšším programovacím jazykům.

1 Paměťový model

Pro jednoduchost budeme předpokládat, že paměť je jednotný spojitý prostor,[1] který se na platformě AMD64 skládá ze 2^{64} paměťových buněk o velikosti jeden byte. To znamená, že ke každé paměťové buňce můžeme přiřadit číslo, tzv. adresu, z rozsahu 0 až $2^{64} - 1$ a současně platí, že máme-li paměťovou buňku na adrese a, následující (sousední) buňka se nachází na adrese $a + 1$.

Hodnoty v paměti počítače jsou uloženy v po sobě následujících buňkách, např. 32bitová celá čísla (typu `int`) obsadí čtyři po sobě jdoucí buňky, 16bitové hodnoty dvě buňky atd. Jelikož jsou hodnoty uloženy v konkrétních paměťových buňkách, můžeme určit adresu, kde se hodnota nachází. Ta odpovídá adrese první paměťové buňky, jež obsahuje danou hodnotu. A samozřejmě v opačném směru, máme-li adresu v paměti, můžeme do ní uložit hodnotu. Toto pojetí ilustruje Obrázek 1.



Obrázek 1: Ilustrace uložení hodnoty v paměti

[1] Ve skutečnosti není tento prostor využitelný celý a jsou v něm oblasti, které mají speciální význam, např. jsou určeny pro přístup k hardware, obsahují jádro operačního systému nebo jsou z jiných technických důvodů prakticky nevyužitelné. Pro potřeby cvičení zaměřeného na adresování paměti to můžeme zanedbat.

2 Adresace paměti

Instrukční sada procosorů AMD64 má docela široké možnosti práce s pamětí. Většina instrukcí umožňuje, aby jeden z operandů, ať už zdrojový nebo cílový, odkazoval na místo v paměti.[2] Přístup k paměti se zapisuje ve tvaru *velikost* [*adresa*], kde velikost je klíčové slovo `byte`, `word`, `dword`, `qword` označující hodnotu o velikosti 1, 2, 4 a 8 B. Adresy můžeme chápat jako běžná celá čísla. Proto se na platfromě x86 pro uložení adres používají běžné registry `rax`, ..., `rdx`, `rsi`, `rdi`, `rbp`, `rsp`, `r8`, ..., `r15`.

Například:

```
mov eax, dword [0x12345678]  ; nacte do registru eax hodnotu z adresy 0x12345678
mov ax, word [rbx]           ; nacte do registru ax hodnotu z adresy, ktera je v rbx
add al, byte [rbx]           ; pricte k al hodnotu bytu na adrese rbx
mov byte [rbx], 0            ; nastavi byte na adrese dane registrem rbx na 0
add dword [rbx], 2           ; pricte k hodnote na adrese rbx hodnotu 2
```

Pokud je z velikosti registrů jasné, s jak velkou hodnotou pracujeme, můžeme *velikost* vypustit. To platí pro první tři příklady, nikoliv však pro poslední dva.

Aby bylo možné snadno podchytit různé módy pro přístup do paměti (ukazatele, přístup k jednotlivým prvkům pole, k strukturovaným datovým typům) je možné adresu zadat ve tvaru:

$$adresa = posunuti + baze + index \times factor$$

Kde *posunutí* je konstanta, *báze* a *index* jsou registry a *factor* je číslo 1, 2, 4 nebo 8, přičemž libovolnou část lze vypustit, ale není možné nic dalšího doplnit.

3 Použití společně s datovými typy

3.1 Ukazatele

Nejjednodušší je práce s ukazateli. Ukazatel odpovídá adrese v paměti (tj. celému číslu), použití se neodchyluje od toho, co jsme již viděli. Ukažme si to na následujícím příkladu.

```
;;
;; funkce zvysi hodnotu danou ukazatel o 1
;;
;; void incref(int *n);
;;
incref:
        mov dword edx, [rdi]    ; precteme hodnotu do registru edx (1. operand obsahuje ukazatel)
        add edx, 1              ; zvysime hodnotu o 1
        mov dword [rdi], edx    ; ulozime hodnotu zpet
        ret
```

[2] V rámci jedné instrukce je však možné adresovat paměť nejvýše jednou.

V tomto příkladu máme funkci, které předáváme ukazatel na 32bitovou hodnotu (typu `int`). I když se jedná o ukazatel na 32bitovou hodnotu, má tento ukazetel 64bitů.[3] Tento ukazatel je předán jako první argument funkce, je proto v registru `rdi`. Na prvním řádku načteme hodnotu danou ukazatelem do registru `edx`, následně hodnotu zvýšíme o jedna (druhý řádek) a uložíme zpět na adresu danou ukazatelem v registru `rdi` (třetí řádek).[4] Funkce nevrací žádnou hodnotu, nemusíme proto nastavovat žádnou hodnotu do registru `rax`.

Že námi vytvořená funkce pracuje dle předpokladu, můžeme ověřit následně v jazyce C.

```
int a  = 42;
int *p = &a;
incref(p);
printf("%i\n", a);
```

3.2 Pole

Při práci s poli je klíčové získat adresu prvního prvku pole, další prvky jsou umístěny v následujících paměťových buňkách a přístup k poli se pak shoduje s prací s ukazateli. Pokud funkci předáváme pole jako jeden z argumentů, získáváme přímo daný ukazatel.

```
;;
;; Funkce secte count prvku v poli array.
;;
;; int sum(int count, int *array);
;;
sum:
        mov eax, 0              ; prubezny soucet
        mov ecx, 0              ; index aktualniho pruku
sum_loop:
        cmp edi, ecx            ; testujeme, zda jsme na konci pole
        je sum_done             ; ukonceni cyklu
        add eax, [rsi + rcx * 4]  ; pricteni hodnoty do prubezneho souctu
        add ecx, 1              ; prechod na dalsi prvek
        jmp sum_loop
sum_done:
        ret                     ; vraceni vysledku (v eax)
```

3.3 Řetězce

Práce s řetězci je opět variací na práci s ukazateli nebo poli. Důležitým rysem řetězců v jazyce C je to, že konec řetezce je určen znakem `'\0'`, tj. hodnotou 0.

[3] Jako všechny ukazatele na platformě AMD64.

[4] Na architektuře AMD64 je možné všechny tři řádky redukovat na jednu operaci `add dword [rdi], 1`.

```
;;
;; Replika funkce strcpy ze standardni knihovny.
;; Funkce prekopiruje retezec src do pameti danou ukazatelem dst.
;;
;; void my_strcpy(char *dst, char *src);
;;

my_strcpy:
        mov al, byte [rsi]    ; precteme jeden (prvni znak) ze zdrojoveho retezce
        mov byte [rdi], al    ; ulozime tento znak do ciloveho retezce
        cmp al, 0             ; pokud je to znak \0, koncime
        je done
        add rdi, 1            ; posuneme se k dalsimu znaku
        add rsi, 1
        jmp my_strcpy         ; skok na zacatek cyklu
done:
        ret                   ; konec funkce
```

Tento kód můžeme v jazyce C vyzkoušet například následovně.

```
char *duplicate = malloc(1024);
my_strcpy(duplicate, "hello world");
printf("%s\n", duplicate);
free(duplicate);
```

3.4 Strukturované datové typy

Práce se strukturovanými datovými typy na úrovni procesoru se nijak neodlišuje od toho, co jsme již viděli.

Proměnné typu struktura (`struct`) jsou v jazyce C v paměti uloženy jednoduše jako její členy za sebou, tzn. adresa struktury je stejná jako adresa jejího prvního prvku. Např. proměnná `qux`, která je typu `struct foo`:

```
struct foo {
   int bar;
   short baz;
}
struct foo qux;
```

je uložena jako 6 bytů – 4 byty pro `bar`, 2 byty pro `baz`, přes proměnnou `foo` se v assembleru dostaneme k prvnímu prvku `foo.bar`.

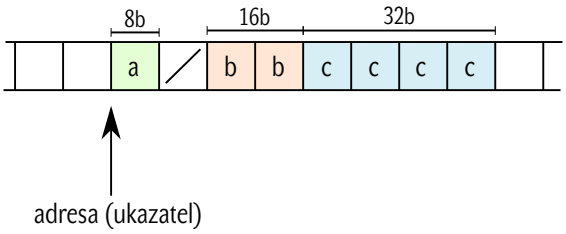
Při práci se strukturami je potřeba brát v potaz, že pro efektivnější práci se strukturami dochází k tzv. zarovnání velikosti struktury (padding) na vhodný násobek 4 nebo 8 B. To znamená, že i když výše uvedená struktura potřebuje k uchování dat pouze 6 bytů, ve skutečnosti zabere v paměti 8 bytů. Ověřte si to vhodným použitím operátoru `sizeof`.

Vedle zarovnání velikosti strukturovaných datových typů je potřeba dbát i na zarovnání jednotlivých členů struktury. Hodnoty velikosti jeden byte jsou zarovnávany na 1 B, dvoubytové hodnoty na 2 B, čtyřbytové hodnoty na 4 B, atd. V praxi to znamená, že jednotlivé členy daného typu začínají vždy na násobku svého zarovnání, např. hodnoty typu `int` jsou ve struktuře uložené vždy na pozici, která je násobkem čtyř, apod. Toto chování je nutné mít na paměti při návrhu datových struktur, protože špatně zvolené pořadí jednotlivých členů může vést k nevhodné spotřebě paměti.

Uvažujme následující strukturu:

```
struct foo {
        char a;
        short b;
        int c;
};
```

Její rozložení v paměti ukazuje Obrázek 2. Všimněme si volného bytu mezi členy `a` a `b`, jenž je vložen proto, aby hodnota typu `short` byla zarovnána na 2 B. Pozici, na které je daný člen uložen, je možné v jazyce C zjistit pomocí makra `offsetof(struktura, clen)` z hlavičkového souboru `stddef.h`.



Obrázek 2: Rozložení paměti ukázkové datové struktury

Práci s touto strukturou z pohledu assembleru ilustruje následující příklad.

```
;;
;; void do_foo(struct foo *x)
;;

do_foo:
        mov al, [rdi]       ; precte hodnotu clenu a do registru al
        mov cx, [rdi + 2]   ; precte hodnotu clenu b do registru cx
        mov [rdi + 4], eax  ; zapise hodnotu v registru eax do clenu c
        ret
```

4 Úkoly k procvičení

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

1. Napište funkci `void swap(int *a, int *b)`, která prohodí hodnoty, které jsou dány ukazateli `a` a `b`.

2. 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`.

3. Napište funkci `void countdown(int *values)`, která do pole `values` uloží posloupnost $10, 9, 8, \ldots, 1$ (v tomto pořadí).

4. Napište funkci `void nasobky(short *multiples, short n)`, která do pole `multiples` uloží prvních deset násobků čísla `n`.

5. Napište funkci int minimum(int count, int *values), která vrací nejmenší prvek pole values obsahující count hodnot. Vyzkoušejte, že funkce funguje správně pro kladná i záporná čísla.

6. Napište funkci `unsigned int my_strlen(char *s)`, která se bude chovat jako funkce `strlen` ze standardní knihovny jazyka C.

7. Napište funkci `void my_strcat(char *dest, char *src)`, která se bude chovat jako funkce strcat ze standardní knihovny jazyka C.