# Operační systémy 1: Praktická cvičení

Petr Krajča

2024

# Prolog

Tento text představuje soubor praktických cvičení určených k pochopení a prohloubení znalostí o soudobých operačních systémech. Ke správnému pochopení operačních systémů, je potřeba porozumět tomu, jakým způsobem jsou programy vykonávany. Proto se text z podstatné části věnuje programování v jazyce symbolických adres, známém dnes spíše pod názvem assembler. Dále se text věnuje základním rozhraním operačních systémů Linux a WindowsNT pro práci s procesy a vlákny. Text je zakončen praktickou úkázkou, jak lze s dusud získanými znalostmi implementovat vlákna v uživatelském prostoru. Protože je programovací jazyk C lingua franca systémového programování, jednotlivá civčení se bez něj neobejdou. Proto do textu byly zařazeny dvě cvičení věnované jednak opakování základních znalostí z jazyka C a tak i procesu překladu tohoto jazyka.

# Obsah

# 1

# Jazyk C: Opakování

Cvičení z předmětů Operační systémy 1 a 2 jsou zaměřena na pochopení a procvičení základních služeb poskytovaných operačním systémem. Pro zvládnutí těchto cvičení je naprosto zásadní zvládnutí programování v jazyce C. Proto v úvodním cvičení připomeneme základní aspekty tohoto jazyka, se kterými budeme potřebovat pracovat.[1]

## 1.1 Datové typy

Jazyk C nabízí širokou škálu datových typů, ať už skalárních (obsahují jednu hodnotu) tak složených. Určitou nevýhodou je, že různé překladače jazyka C mohou nabízet datové typy odlišných vlastností.[2]

### 1.1.1 Skalární datové typy

Pro reprezentaci celých čísel máme pět znaménkových datových typů `char`, `short`, `int`, `long`, `long long`, které mají ještě svou neznaménkovou variantu uvozenou klíčovým slovem `unsigned`. Velikosti a rozsahy jednotlivých datových typů, jak je používají nejběžnější překladače (GCC, MSVC na platformě x86 a AMD64) ukazuje Tabulka 1.1.

Datové typy `int` a `long` mají v případě 32bitového překladače stejnou velikost, tj. 32 bitů. U 64bitových překladačů *v unixech* se používá `int` o velikosti 32 bitů a `long` o velikosti 64 bitů, kdežto na platformě Windows je velikost `int` a `long` stejná, tj. 32 bitů.

Název datového typu `char` naznačuje, že se jedná o typ sloužící k uložení znaků,[3] jedná se však o zcela obecný celočíselný typ, který může obsahovat nejen znaky ale obecně celá čísla. Následující deklarace a přiřazení jsou možné a ekvivalentí:

```
char foo = 'A';
char foo = 65;
```

[1] Tento text je pouze přehledový a nečiní si ambice popsat celý jazyk C.

[2] Díky tomu je možné v jazyce C programovat na různých obvyklých i méně obvyklých platformách.

[3] Nejčastěji z ASCII tabulky.

| typ | velikost (bit) | min. hodnota | max. hodnota |
|---|---|---|---|
| `char` | 8 | -128 | 127 |
| `unsigned char` | 8 | 0 | 255 |
| `short` | 16 | -32.768 | 32.767 |
| `unsigned short` | 16 | 0 | 65.535 |
| `int` | 32 | -2.147.483.648 | 2.147.483.647 |
| `unsigned int` | 32 | 0 | 4.294.967.295 |
| `long` | * | * | * |
| `unsigned long` | * | * | * |
| `long long` | 64 | -9.223.372.036.854.775.808 | 9.223.372.036.854.775.807 |
| `unsigned long long` | 64 | 0 | 18.446.744.073.709.551.616 |

Tabulka 1.1: Celočíselné datové typy a rozsahy

K reperezentaci čísel s plovoucí řádovou čárkou slouží datové typy `float`, `double` a `long double` o velikostech 32, 64 a 80 bitů. Čísla s plovoucí řádovou čárkou jsou obvykle zpracovávána jinak, proto se jim budeme věnovat v samostatném cvičení.

### 1.1.2 Ukazatele a pole

Specifickým případem skalárních datových typů jsou ukazatele. Hodnota ukazatele ukazuje na místo v paměti, kde je uložena hodnota daného typu. Následující příklad ilustruje jejich použití.

```
int a  = 42;        // proměnná typu int
int *p = &a;        // ukazatel na hodnotu typu int
                    // operátor & (reference) je použit k získání ukazatele (adresy) proměnné a
printf("%i\n", *p); // přečtení hodnoty dané ukazatelem p
                    // operátor * (dereference) získá hodnotu
*p = 123;           // zde je operátor dereference použit ke změně hodnoty dané ukazatelem p
```

Ukazatele mají několik důležitých rolí (i) umožňují předávat argumenty odkazem, (ii) umožňují práci s poli, (iii) umožňují práci s dynamicky alokovanou pamětí (objekty).

#### Předávání argumentů odkazem

Předávání argumentů odkazem je vhodné v situacích, kdy potřebujeme uložit výsledek do připravené paměti nebo chceme vrátit hodnotu přes argument (např. pokud máme více návratových hodnot), jak ukazuje následující příklad.

```
void add(int a, int b, int *x) {
  *x = a + b;
}

int z;
add(10, 20, &z);
```

#### Pole a ukazatele

Datový typ pole představuje celými čísly indexovanou kolekci hodnot stejného typu. Pole mohou být deklarována buď s pevnou, nebo nespecifikovanou velikostí.

```
int a[3]; // pole celých čísel o třech prvcích
int b[];  // pole celých čísel s nespecifikovanou velikostí
```

Pole jako hodnota si nenese informaci o své velikosti a jazyk nekontroluje, zda nepřistupujeme za hranici pole. Proto se následující kód sice provede, ale jeho provedení povede k nespecifikovanému chování, které se může projevit nežádoucím chováním ihned, později nebo vůbec.

```
int a[3];
int a[10] = 42;
```

Pole úzce souvisí s ukazateli. Pole můžeme chápat jako ukazatel na první prvek pole. Aritmetika s ukazateli nám pak umožňuje pole procházet, jak ilustruje následující příklad.

```
int a[3];
a[0]     = 1;         // přiřadí prvnímu prvku pole hodnotu 1
*a       = 1;         // to samé s využitím ukazatele
a[1]     = 2;         // přiřadí druhému prvku pole hodnotu 2
*(a + 1) = 2;         // to samé s využitím ukazatele
int *b   = a;         // přiřadí do b začátek pole a
b++;                  // pole b bude začínat na druhém prvku pole a
printf("%i\n", b[0]); // vypíše 2
```

#### Dynamická alokace paměti

K dynamické alokaci paměti slouží funkce `void *malloc(size_t size)`, která alokuje minimálně `size` bytů paměti a vrací na ni ukazatel, jak ukazují následující příklady.

```
int *p = (int *) malloc(sizeof(int));      // alokuje pamět pro jednu hodnotu typu int
int *a = (int *) malloc(sizeof(int) * 10); // alokuje pole typu int o velikosti deset prvků
```

Alokovanou paměť je nutné uvolnit pomocí funkce `free`.

Paměť vrácená funkcí `malloc` není nijak inicializovaná a může obsahovat libovolná data. Pokud potřebujeme paměť vynulovanou, můžeme použít funkci `calloc`. V případě, že potřebujeme změnit velikost alokované paměti, použijeme funkci `realloc`. Poznamenejme, že pokud chceme pomocí funkce realloc zvětšit množství alokované paměti, dojde k alokaci nového místa a data jsou do něj následně překopírována.

### 1.1.3 Řetězce

Specifickým případem ukazatelů jsou řetězce, což jsou ukazatele typu `char *`, které ukazují na první znak řetězce. Konec řetezce je indikován znakem `'\0'` (odpovídá hodnotě 0).

V případě řetězcových literálů je konec řetězce doplněn automaticky. U těchto literálů dejte pozor na to, že tyto řetězce mohou být, a často jsou, neměnné. Nelze tedy provést.

```
char *s = "abc";
s[0] = 'A'; // pravděpodobně selže
```

### 1.1.4 Pravdivostní hodnoty

Pro reprezentaci pravdivostních hodnot lze v jazyce C použít libovolný celočíselný typ včetně ukazatelů. Hodnota 0 nebo `NULL` odpovídá hodnotě *nepravda*, cokoliv jiného je chápano jako *pravda*. Následující kód ukazuje nejběžnější případy použití.

```
int a = 1;
int *p = NULL;

if (a == 2) { } // explicitní porovnání
if (a) { }      // test, jestli proměnná ,,a'' obsahuje nenulovou hodnotu
if (!p) { }     // test, zda je ukazatel roven NULL
```

### 1.1.5 Strukturované datové typy

Související hodnoty různých datových typů lze spojit do jednoho strukturovaného datového typu. Následující kód ukazuje vytvoření nového datového typu představujícího bod v rovině.

```
struct point {
    int x;
    int y;
};
```

Takto jsme vytvořili nový strukturovaný datový typ `struct point`, se kterým můžeme pracovat například následovně.

```
struct point a = { 2, 4 };
void point_print(struct point p) {
    printf("[%i, %i]\n", p.x, p.y);
}
```

Pokud chceme zavést jednodušší pojmenování, můžeme zavést alias pomocí klíčového slova `typedef`, jak ukazuje následující příklad.

```
typedef struct point {
    int x;
    int y;
} point;
```

Takto nám vznikne strukturovaný datový typ pojmenovaný jen `point`. Použití `typedef` pro deklaraci strukturovaného datavého typu není nezbytné.

Pozor, strukturované datové typy se předávájí hodnotou, tzn. předáme-li do funkce argument, kterým je hodnota strukturovaného datového typu, dojde k vytvoření kopie jeho hodnoty.

Vyzkoušejme si:

```
void point_move(struct point p, int dx, int dy) {
    p.x += dx;
    p.y += dy;
}


point_print(a);          // vypíše [2, 4]
point_move(a, 13, 37);
point_print(a);          // vypíše [2, 4]
```

Všimněme si, že v důsledku předávání argumentu `p` hodnotou došlo ke změně jednotlivých složek pouze v rámci funkce `point_move`. Pokud bychom chtěli, aby změna hodnot byla viditelná i mimo rozsah funkce `point_move`, musíme hodnotu `p` předat odkazem, a pak s ní i tak pracovat, jak ukazuje následující příklad.

```
void point_move(struct point *p, int dx, int dy) {
    (*p).x += dx;
    p->y += dy;
}


point_move(&a, 13, 37);
```

Obraty `(*p).x` a `p->x` mají shodný význam, avšak ten druhý je srozumitelnější a běžnější.

## 1.2 Operátory

Jazyk C nabízí širokou paletu operátorů, některé jsou intuitivní, např. aritmetické nebo relační operátory, těm se nebudeme podrobněji věnovat, jiné jsme již zmínili (reference a dereference), a teď zmíníme jen ty méně obvyklé nebo ty, které mají určité specifické chování. Chování základních operátorů, se kterými se běžně setkáte popisuje následující výčet.

- `a = b` (přiřazení) přiřadí do prvního operandu hodnotu druhého operandu a vyhodnotí se na jeho hodnotu
- `a++` (inkrementace) zvýší hodnotu operandu o 1 a vyhodnotí se na hodnotu před inkrementací
- `++a` (inkrementace) zvýší hodnotu operandu o 1 a vyhodnotí se na aktuální hodnotu
- `a--`, `--a` (dekrementace) analogicky inkrementaci
- `a ? b : c` (podmíněný výraz, ternární operátor) – pokud je první operand *pravda*, vyhodnotí se na druhý operand, jinak se vyhodnotí na třetí operand

- `a && b` (logický součin) – vyhodnotí se na *pravda*, pokud jsou oba operandy *pravdivé*, jinak se vyhodnotí na nepravda
- `a || b` (logický součet) – vyhodnotí se na *pravda*, pokud je alespoň jeden operand *pravda*, jinak se vyhodnotí na nepravda

U operátorů `&&` a `||` dochází ke zkrácenému vyhodnocení. Pokud je jasné, že hodnota výrazu bude pravda nebo nepravda již po vyhodnocení prvního operandu, druhý operand se již nevyhodnocuje, například následující kód je zcela validní.

```
int a = 1;
if (a || (1 / 0)) { ... }
```

### 1.2.1 Bitové operace

Při programování na té nejnižší úrovni často potřebujeme pracovat s jednotlivými bity, k čemuž slouží operátory `&` (bitový součin), `|` (bitový součet), `^` (bitová non-ekvivalence, výlučné nebo, XOR), `~` (negace, inverze bitů), << a >> (bitové posuny).

Jejich použití ukazují následující příklady:

```
  0101 0011          0101 0011          0101 0011
& 1001 0010        | 1001 0010        ^ 1001 0010
-----------        -----------        -----------
  0001 0010          1101 0011          1100 0001


   0101 0011          1001 0010          1001 0010
<<         1       >>         1       >>         1
------------       ------------       ------------
   1010 0110          1100 1001          0100 1001
                     znaménková        neznaménková
                      varianta           varianta
```

Pozor, u bitového posunu vpravo záleží na tom, jestli danou operaci provádíme s hodnotou znaménkového nebo neznaménkového typu. Pokud máme neznaménkovou hodnotu (např. `unsigned char`), doplní se vždy jako nejvyšší bit 0. U znaménkových typů (např. `int`) se doplní kopie nejvyššího bitu.

## 1.3 Úkoly k procvičení

1. Napište funkci `void int2bits(char *, int)`, která převede číslo na textový řetězec představující jeho zápis v binární podobě.
2. Napište funkci `int bits2int(char *)`, která převede textový řetězec představující zápis čísla v binární podobě (tj. `"010110010010..."`) na hodnotu typu `int`.

3. Implementujte funkci `void my_memcpy(void *dest, void *src, size_t size)`, která se chová jako funkce memcpy a přenese po jednolivých bytech obsah paměti z jednoho místa na druhé, předpokládejte, že úseky paměti se nepřekrývají.

4. Navrhněte vhodnou strukturu pro spojový seznam obsahující dvě hodnoty jméno (textový řetězec) a věk (celé číslo). Napište funkci, která bude přidávat prvky do seznamu a funkci, která vypíše obsah tohoto seznamu.

5. Napište funkci `short encode_date(char day, char month, short year)`, která zakóduje datum do 16bitového čísla následovně: `YYYY-YYYM-MMMD-DDDD`[4].

6. Napište funkci `void decode_date(short date, int *day, int *month, int *year)`, která dekóduje datum vytvořené předchozí funkcí a vrátí hodnoty pomocí předaných ukazatelů.

7. Napište funkci, která zjistí, v jakém pořadí jsou vyhodnocovány argumenty.

[4] Úlohu není možné vyřešit tak, aby měla univerzální řešení, a je nutné pracovat s nějakou kompromisní variantou.

# 2

# Jazyk C: Proces překladu

V tomto cvičení se zaměříme na praktické aspekty procesu překladu programů v jazyce C do spustitelné podoby. Jazyk C bereme pouze jako vzorový jazyk a zde nastíněné principy lze aplikovat i na další kompilované programovací jazyky.[1] Značně zjednodušeně řečeno, cílem je ukázat, co se skrývá pod zeleným trojúhelníčkem *Compile & Run* známým z vývojových prostředí. Pro demonstraci budeme používat překladač gcc a nástroje z operačního systému GNU/Linux, avšak podobné principy lze aplikovat i na další překladače, např. Clang/LLVM, MSVC.[2]

## 2.1 Překlad programu

Uvažujme jednoduchý příklad typu „Hello World".

```c
// hello.c
#include <stdio.h>

int main(int argc, char **argv)
{
    printf("Hello World!\n");
    return 0;
}
```

V nejjednodušší variantě program přeložíme a spustíme následovně:

```
gcc hello.c -o hello
./hello
```

V tomto případě název `hello.c` odpovídá zdrojovému kódu programu a `hello` výslednému binárnímu (spustitelnému) programu.

[1] Jednotlivé jazyky mohou mít své odlišnosti.
[2] Jednotlivé nástroje mohou mít své odlišnosti.

### 2.1.1 Fáze překladu

Ač to nemusí být na první pohled zjevné, zdrojový kód při svém překladu prochází několika postupnými transformacemi, které jsou skryty pod jedno spuštění překladače příkazem `gcc`.

1. Nejdříve je spouštěn preprocesor (příkaz `cpp`), který vloží hlavičkové soubory (direktiva `#include`), expanduje makra (direktiva `#define`), odstraní části kódu nesplňující danou podmínku (direktiva `#ifdef`), odstraní komentáře apod . Výstupem preprocesoru je kód programu, který obsahuje pouze kód v jazyce C.
2. V druhé fázi překladač přeloží zdrojový kód v jazyce C do jazyka symbolických adres, tj. vytvoří zápis programu v podobě jednotlivých instrukcí procesoru. Tyto instrukce jsou zapsány v textové podobě.
3. Kód v jazyce symbolických adres je nástrojem, který *assembler*[3] (příkaz `as`) přeložen do podoby objektového souboru.[4] Objektové soubory obsahují program přeložený do strojového kódu a další související informace jako jsou konstanty, informace o poskytovaných symbolech (funkcích, proměnných), ladící informace apod.
4. V poslední fázi překladu jsou objektové soubory sloučeny (příkaz `ld`) a spojeny s knihovnami[5] a je vytvořen výsledný binární soubor.

Jak vypadá kód v jednotlivých fázích překladu, můžeme zjistit pomocí přepínačů příkazu `gcc`.

#### Preprocessor

Výsledek zpracování zdrojového souboru preprocessorem můžeme získat příkazem:

```
gcc -E hello.c
```

Na výstupu z preprocessoru si všimněme jednak absence komentářů a toho jaký kód byl vložen z hlavičkového souboru `#include <stdio.h>`. Když si například dohledáme funkci `printf`, vidíme, že je ve zdrojovém kódu přítomen jen její prototyp, nikoliv celá funkce.

#### Překlad do jazyka symbolických adres

Jak vypadá program přeložený do jazyka symbolických adres, zjistíme s pomocí přepínače `-S`.

```
gcc -S hello.c
```

V tomto případě překladač vygeneruje soubor `hello.s`, který obsahuje jednotlivé instrukce procesoru zapsané v jazyce symbolických adres a další dodatečné informace. Podrobněji se jazyku symbolických adres budeme věnovat v následujících cvičeních.

[3] Původně slovo *assembler* označovalo pouze nástroj, který vzal program v tzv. *jazyce symbolických adres* a přeložil jej do strojového kódu. Postupně se označení assembler přeneslo i na jazyk symbolických adres a dnes naprosto běžně pojem assembler označuje jak nástroj, tak i jazyk popisující program na úrovni jednotlivých instrukcí.

[4] Název objektový soubor nijak nesouvisí s objektově orientovaným programováním.

[5] Minimálně standardní knihovnou jazyka C.

### Překlad do objektového souboru

V předchozí fázi překladu jsme získali program, který již nabyl podoby jednotlivých instrukcí procesoru, ale je nutné jej přeložit do strojového kódu. To obstará assembler, který vygeneruje objektový kód. Objektový kód získáme přepínačem `-c`.

```
gcc -c hello.c
```

Překladač vygeneruje soubor `hello.o`, který již obsahuje přeložený strojový kód. Jelikož se jedná o binární formát dat, není možné[6] jej studovat prostým zobrazením. Můžeme to posoudit zobrazením pomocí nástroje `hexdump`.

```
hexdump -C hello.o
```

Chceme-li prozkoumat obsah objektového souboru, musíme použít vhodný nástroj, kterým je například `objdump`. Nástroj `objdump` umožňuje zobrazit jednotlivé logické části objektového souboru, tzv. sekce.

```
objdump -s hello.o
```

Ve výpisu bychom měli minimálně vidět sekci `.text`, která obsahuje program přeložený do strojového kódu, a sekci `.rodata`, která obsahuje data programu, která jsou jen pro čtení, v našem případě řetězec `Hello World!`.

Další užitečnou funkcí, kterou nástroj `objdump` nabízí, je tzv. disassembling, opačný proces k assembleru, kdy je strojový kód přeložen zpět na svou textovou reprezentaci ve formě jazyka symbolických adres. Tato funkcionalita se skrývá pod přepínačem `-d`, případně ve spojení s přepínačem `-M intel`, který zajistí zobrazení kódu v syntaxi, jak byla použita v průběhu přednášky.

Pokud provedeme následující příkaz:

```
objdump -d -M intel hello.o
```

Uvidíme, že objektový soubor opravdu obsahuje jen námi vytvořenou funkci a neobsahuje kód funkce `printf`.

### Vytvoření spustitelného souboru

Při vytvoření spustitelného souboru dochází k tomu, že jednotlivé objektové soubory, ze kterých se program skládá,[7] jsou sloučeny společně s knihovnami. K vytvoření výsledného souboru můžeme použít buď příkaz `gcc` nebo zavolat tzv. *linker* (příkaz `ld`).

```
gcc -o hello hello.o
```

Což odpovídá přibližně:

[6] Nebo minimálně pohodlné.

[7] V ukázkovém příkladu se program skládá z právě jednoho objektového souboru.

```
ld -o hello hello.o /lib64/crt1.o --dynamic-linker /lib64/ld-linux-x86-64.so.2 -lc
```

V tomto případě linker spojí vstupní objektové soubor se souborem `crt1.o` (C runtime), který obsahuje funkce nutné pro běh programu v C, společně se standardní knihovnou jazyka C (přepínač `-lc`). Přepínač `--dynamic-linker` zajišťuje, že knihovna jazyka C bude připojena dynamicky. Podrobněji to bude vysvětleno na přednášce.

Ze zápisu obou variant je zjevné, že první varianta vytvoření binárního souboru je komfortnější.

### 2.1.2 Nástroje pro překlad programu

Rozdělení překladu do jednotlivých fází, zejména generování objektových souborů, umožňuje rozdělit překlad do logických celků, tj. mít související funkce v oddělených souborech a v případě změny překládat jen změněné soubory. Aby se daly větší programy pohodlně překládat, vznikl nástroj `make`, který sestaví program dle zadaných pravidel. Pravidla pro překlad se zapisují do souboru, který se jmenuje `Makefile`, a jednotlivá pravidla mají následující tvar:

```
cil: zavislosti
<TAB!>prikaz pro sestaveni cile
<TAB!>prikaz pro sestaveni cile
```

Pro náš ukázkový příklad by `Makefile` mohl vypadat následovně.

```
hello: hello.o
	gcc -o hello hello.o

hello.o: hello.c
	gcc -c hello.c
```

První pravidlo udává, že pro sestavení programu `hello` potřebujeme mít soubor `hello.o`, a pokud jej máme, program se přeloží pomocí `gcc -o hello hello.o`. Druhé pravidlo říká, že pro vytvoření programu `hello.o` potřebujeme `hello.c` a výsledný soubor získáme příkazem `gcc -c hello.c`.

Program přeložíme pomocí příkazu `make`. Je důležité, že nástroj `make` automaticky vyřeší všechny závislosti a spustí příkazy ve správném pořadí.[8] Současně nástroj `make` zajistí, že se překládají jen změněné části programu a ty, které na nich závisí.

#### Sestavení většího programu

V tomto jednoduchém příkladu je použití nástroje `make` příslovečný kanón na vrabce, ale již u mírně větších projektů se ukazuje jeho užitečnost.

Předpokládejme, že budeme chtít mít nějaké funkce[9] v odděleném souboru se zdrojovými kódy a ty volat z programu `hello.c`.

V takovém případě budeme postupovat následovně.

[8] S pomocí přepínače `-j` je možné překlad spustit i paralelně.

[9] Pro jednoduchost budeme uvažovat jen jednu funkci pro výpočet faktoriálu

**Hlavičkový soubor**

Nejdříve vytvoříme hlavičkový soubor, nazvěme jej `myfuncs.h`. Tento hlavičkový soubor by měl obsahovat deklaraci prototypu funkce.

```
// myfuncs.h
#ifndef MYFUNCS_H
#define MYFUNCS_H

/* funkce pro vypocet faktorialu */
unsigned int fact(unsigned int n);

#endif
```

Všimněme si, že v hlavičkovém souboru deklarujeme jen název funkce, typy argumentů a typ návratové hodnoty. V hlavičkovém souboru se nenachází tělo funkce. Dále stojí za povšimnutí direktivy preprocesu na začátku souboru. Ty zajišťují, že hlavičkový soubor je do kódu vložen nanejvýš jednou.

Kód funkce je pak uveden v samostatném souboru `myfuncs.c`.

```
// myfuncs.c
#include "myfuncs.h"

unsigned int fact(unsigned int n)
{
        if (n == 0) return 1;
        return n * fact(n - 1);
}
```

Použití funkce pro výpočet faktoriálu se neliší od použití funkcí, např. ze standardní knihovny, tj. vložíme hlavičkový soubor a voláme funkci běžným způsobem.

```
// hello.c
#include <stdio.h>
#include "myfuncs.h"

int main(int argc, char **argv)
{
        printf("5! = %i\n", fact(5));
        return 0;
}
```

Při překladu doplníme do `Makefile` pravidlo pro překlad `myfuncs.o` a doplníme jej jako závislost pro sestavení programu `hello`.

```
hello: hello.o myfuncs.o
	gcc -o hello hello.o myfuncs.o

hello.o: hello.c
	gcc -c hello.c

myfuncs.o: myfuncs.c myfuncs.h
	gcc -c myfuncs.c
```

Při sestavování programu bývá zvykem nastavit přepínače udávající chování překladače, např. míru optimalizace (přepínač `-O1` až `-O3`), standard jazyka (např. `-std=c99`), zobrazení upozornění (přepínač `-W`) nebo generování ladicích informací (přepínač `-g`). Abychom tyto volby mohli nastavit jednotně, podporuje nástroj `make` proměnné podobně jako například unixový shell. Ukázkový `Makefile` bychom mohli vylepšit následovně.

```
CFLAGS=-O1 -Wall -std=c99 -g

hello: hello.o myfuncs.o
	gcc $(CFLAGS) -o hello hello.o myfuncs.o

hello.o: hello.c
	gcc $(CFLAGS) -c hello.c

myfuncs.o: myfuncs.c myfuncs.h
	gcc $(CFLAGS) -c myfuncs.c
```

## 2.2 Programování v assembleru

Způsob překladu popsaný v předchozí kapitole má ještě jeden zásadní důsledek a tím je možnost vytvářet programy, jejichž jednotlivé části jsou napsány v různých jazycích. Jelikož objektové soubory obsahují přeložený strojový kód ve standardizovaném formátu, je možné je vytvářet v různých jazycích a následně je linkerem nechat spojit do spustitelného formátu. Pro nás je to zajímavé z toho důvodu, že jedním z těch jazyků může být jazyk symbolickcýh adres.

### 2.2.1 Program v jazyce symbolických adres a jeho překlad

Pro překlad programu v jazyce symbolických adres budeme používat assembler `nasm`, který je mírně přívětivější než nástroj `as`.[10] Ukážeme si to na jednoduchém příkladu funkce vracející hodnotu `42`.

```
; soubor demo.asm
global foo
```

[10] Tento nástroj vzniknul primárně pro potřeby překladačů a nemusí být úplně uživatelsky přívětivý.

```
section .text
foo:
        mov eax, 42
        ret
```

Ukázkový soubor obsahuje několik základních částí.

Na začátku máme komentář vyznačený znakem středník. Následuje direktiva `global`, která označuje jaké symboly (v terminologii C: funkce a proměnné) daný zdrojový soubor poskytuje, v našem případě to bude funkce `foo`. Následuje vyznačení sekce `.text`, která obsahuje kód programu zapsaný v jazyce symbolických adres. Naše funkce je vyznačena pomocí návěstí `foo:` (identifikátor + dvojtečka) a následuje kód samotné funkce.

Funkce se skládá ze dvou instrukcí. První instrukce uloží do registru `eax` návratovou hodnotu. Jedná se o konvenci, kdy celočíselné návratové hodnoty jsou předávány registrem `rax`, resp. `eax`, dle velikosti typu návratové hodnoty. Návrat z funkce je realizován instrukcí `ret`. Podrobněji se volání funkcí budeme věnovat na přednášce a v následujících cvičeních.

Překlad program v jazyce symbolických adres zajistíme příkazem `nasm`.

```
nasm -f elf64 demo.asm
```

Překladač v tomto případě vygeneruje soubor `demo.o`, který je ve formátu `elf64`, což je formát, který implicitně používá i překladač `gcc`.

### 2.2.2 Spojení s programem v jazyce C

Na straně jazyka C použijeme prostředky, které již známe. Nejdříve deklarujeme protyp funkce[11] a tuto funkci zavoláme.

```
#include <stdio.h>

/* prototyp funkce napsane v assembleru */
int foo();

int main()
{
        printf("Answer to life, etc.: %i\n", foo());
        return 0;
}
```

Samotné provázání funkcí je realizováno v linkovací fázi a pro pohodlné sestavení můžeme použít nástroj `make`, kdy do `Makefile` vložíme pravidlo pro překlad souboru v jazyce symbolických adres:

```
hello: hello.o demo.o
        gcc -o hello hello.o demo.o
```

[11] Mohli bychom jej umístit do samostatného hlavičkového souboru.

```
hello.o: hello.c
        gcc -c hello.c

demo.o: demo.asm
        nasm -f elf64 demo.asm
```

## 2.3 Úkoly k procvičení

1. Vezměte funkce `int2bits` a `bits2int` a umístěte je do samostného souboru `bits.c` a vytvořte odpovídající hlavičkový soubor `bits.h`.
2. To samé proveďte pro funkce `encode_date` a `decode_date` a ty umístěte do souborů `dates.[ch]`.
3. Vytvořte program, který výše popsané funkce bude používat.
4. Pro překlad programu vytvořte vhodný *makefile* a program přeložte.
5. S pomocí nástrojů a přepínačů překladače popsaných v přiloženém textu se podívejte na jednotlivé fáze překladu.
6. Na základě vzoru v přiloženém textu napište v jazyce symbolických adres funkci, která do registru `edi` uloží hodnotu 10 představující jednu stranu obdélníka, do registru `esi` uloží hodnotu 17 představující druhou stranu obdélníka a vrátí obvod obdélníka s těmito rozměry.
7. Výše popsanou funkci vyzkoušejte.

# 3

# Externí assembler, registry a základní aritmetické operace

Pomocí assembleru jsme schopni vytvářet programy na té nejnižší možné úrovni, tj. na úrovni jednotlivých instrukcí procesoru. V minulosti nebylo neobvyklé, že programátoři přepisovali kritické rutiny svých programů do assembleru, aby dosáhli maximálního výkonu. Díky masivnímu pokroku v oblasti překladačů tato doba dávno minula a moderní překladače jsou schopny vytvořit efektivnější kód než programátor. Jsou však oblasti, kde použití assembleru má svůj nezastupitelný význam a to je systémové programování (operační systémy, překladače), systémy s omezenými prostředky (vestavné systémy, spotřební elektronika, řídící systémy) a aplikace využívající specializované instrukce procesoru (zpracování obrazu, kryptografie). V tomto a následujících cvičeních si na instrukční sadě procesorů rodiny Intel x86 (resp. AMD64) ukážeme činnost procesoru, a jak je běh programu realizován pomocí jednotlivých instrukcí.

## 3.1 Assembler

Vytvářet program nebo jeho části pomocí assembleru můžeme dvěma způsoby. (i) Buď můžeme všechen kód napsat v assembleru a přeložit jej pomocí assembleru[1] do strojového kódu, který lze spustit jako samostatný program, případně volat z vyššího programovacího jazyka. (ii) Můžeme také kombinovat kód ve vyšším programovacím jazyce (např. C) s kódem v assembleru pomocí tzv. *inline assembleru*, jak ukazuje následující příklad pro překladač MSVC (Visual Studio):

```
int inc(int n) {
   _asm {
        mov eax, n  // do registru eax nacteme hodnotu argumentu
        add eax, 1  // k hodnote pricteme jednicku
```

[1] Původně slovo *assembler* označovalo nástroj, který vzal program v tzv. *jazyce symbolických adres* a přeložil jej do strojového kódu. Postupně se označení assembler přeneslo i na jazyk symbolických adres a dnes naprosto běžně pojem assembler označuje jak nástroj, tak i jazyk popisující program na úrovni jednotlivých instrukcí.

```
        mov n, eax  // hodnotu vratime do promenne n
    }
    return n;
}
```

Protože překladač MSVC nepodporuje v inline assembleru jinou architektu než i386 a inline assembler v překladači GCC není úplně intuitivní, budeme v tomto a několika následujících cvičeních používat externí assembler *nasm* způsobem, jak jsme si ukázali v předchozím cvičení.

Připomeňme klíčovou část, tj. příklad funkce vracející hodnotu 42.

```
; soubor demo.asm
global foo
section .text
foo:
        mov eax, 42
        ret
```

Na začátku máme komentář vyznačený znakem středník. Následuje direktiva `global`, která označuje jaké symboly (v terminologii C: funkce a proměnné) daný zdrojový soubor poskytuje. V našem případě to bude funkce `foo`. Následuje vyznačení sekce `.text`, která obsahuje kód programu zapsaný v jazyce symbolických adres. Naše funkce je vyznačena pomocí návěští `foo:` (identifikátor + dvojtečka) a následuje kód samotné funkce.

Funkce se skládá ze dvou instrukcí. První instrukce uloží do registru `eax` návratovou hodnotu. Jedná se o konvenci, kdy celočíselné návratové hodnoty jsou předávány registrem `rax`, resp. `eax` (podle velikosti návratové hodnoty). Návrat z funkce je realizován instrukcí `ret`. Podrobněji se volání funkcí budeme věnovat na přednášce a v následujících cvičeních.

Překlad program v jazyce symbolických adres zajistíme příkazem `nasm`.

```
nasm -f elf64 demo.asm
```

## 3.2 Instrukční sada x86/AMD64

Instrukční sada procesorů x86/AMD64 je velmi košatá a není možné ji v rámci jednotlivých cvičení popsat celou. Proto u jednotlivých cvičení budou popsány jen určité tématické části a pro další informace odkazujeme laskového čtenáře k dalším materiálům:

- Brandejs M. Mikroprocesory Intel Pentium. Brno: Fakulta informatiky, Masarykova univerzita, 2010.[2]
- Přehled všech operací procesorů rodiny x86[3]. (Není potřeba znát.)

[2] `http://www.fi.muni.cz/usr/brandejs/Brandejs_Mikroprocesory_Intel_Pentium_2010.pdf`
[3] `http://ref.x86asm.net/coder64.html`

### 3.2.1 Registry

Procesory rodiny AMD64 nabízí následující registry:

- 64bitové: `rax`, `rbx`, `rcx`, `rdx`, `rsi`, `rdi`, `r8` až `r15`,
- 32bitové: `eax`, `ebx`, `ecx`, `edx`, `esi`, `edi`, `r8d` až `r15d`,
- 16bitové: `ax`, `bx`, `cx`, `dx`, `si`, `di`, `r8w` až `r15w`,
- 8bitové: `ah`, `al`, `bh`, `bl` `ch`, `cl`, `dh`, `dl`, `sil`, `dil`, `r8b` až `r15b`.

Přičemž vzájemně si odpovídající skupiny registrů (např. `rax`, `eax`, `ah`, `al`) sdílí stejnou paměť.

Vedle těchto registrů existují a jsou přístupné ještě registry `rsp` a `rbp`, které ale mají specifickou funkci a nelze je použít libovolně. Další registry jako jsou `rip` a `rf(lags)` mohou být měněny jen vybranými instrukcemi.

### 3.2.2 Přehled základních aritmetických instrukcí

Instrukce mají obvykle tvar:

$$\langle \textit{ název instrukce } \rangle \ \langle \textit{ cílový operand } \rangle \ [\ , \langle \textit{ další operand } \rangle, \ldots\ ]$$

Například instrukce ščítání `add` má právě dva operandy, kdy k prvnímu operandu je přičtena hodnota operandu druhého, tzn. máme-li instrukci `add eax, ebx`, znamená to, že k hodnotě v registru `eax` je přičtena hodnota `ebx`. To odpovídá výrazu `eax += ebx`, jak jej známe z vyšších programovacích jazyků.

Operandy instrukcí mohou být

- `r` – registry,
- `m` – adresa místa v paměti,
- `i` – přímé hodnoty (konstanty).

Každá instrukce připouští jen určité kombinace operandů.[4] Význam jednotlivých základních aritmetických instrukcí a jaké jsou přípustné operandy, ukazuje následující výčet.

```
mov r/m, r/m/i        ; op1 := op2
add r/m, r/m/i        ; op1 := op1 + op2
sub r/m, r/m/i        ; op1 := op1 - op2
neg r/m               ; op1 := - op1

inc r/m               ; op1 := op1 + 1
dec r/m               ; op1 := op1 - 1

mul r/m               ; edx:eax := eax * op1
```

[4] Navíc paměť lze v jedné instrukci adresovat pouze jednou.

```
mul r/m              ; rdx:rax := rax * op1
imul r, r/m          ; op1 := op1 * op2
imul r, r/m, i       ; op1 := op2 * op3

div r/m              ; eax := edx:eax / op1; edx := edx:eax % op1  (neznaménkové dělení)
div r/m              ; rax := rdx:rax / op1; rdx := rdx:rax % op1  (neznaménkové dělení)
idiv r/m             ; eax := edx:eax / op1; edx := edx:eax % op1  (znaménkové dělení)
idiv r/m             ; rax := rdx:rax / op1; rdx := rdx:rax % op1  (znaménkové dělení)
```

Instrukce pro násobení a dělení jsou atypické v tom, že mají určené registry, se kterými pracují, a ty jsou ještě ovlivněny velikostí operandu dané instrukce. To znamená, že pokud máme operand instrukce `mul` 32bitový, bude výsledek násobení uložen do dvojice registrů `edx` a `eax`, kde registr `eax` obsahuje spodních 32 bitů výsledku a registr `edx` horních 32 bitů. Máme-li operand 64bitový, bude výsledek uložen do dvojice registrů `rax`, `rdx`.

V případě delění je situace ošemetnější. Pokud je operand (tj. dělitel) 32bitový, dělí se obsah dvojice registrů `edx` a `eax`, podíl je uložen do registru `eax` a zbytek po dělení do registru `edx`. V případě, že je dělitel 64bitový, je postup analogický, jen s tím rozdílem, že jsou použity registry `rax` a `rdx`.

Z toho plyne, že při dělení nestačí nastavit jen obsah registru `eax` (resp. `rax`), vždy musíme korektně nastavit i hodnotu v registru `edx` (resp. `rdx`).[5]

## 3.3 Praktická práce s assemblerem

Programování na úrovni assembleru si můžeme vyzkoušet implementací jednoduchých funkcí, jak jsme si uvedli v části 3.1 a v předchozím cvičení. Princip implementace zůstává stejný, tj.

- Pomocí direktivy `global` určíme, jaké funkce jsou implementovány v assembleru.
- V sekci `.text` implementujeme jednotlivé funkce, které vyznačíme návěštím. Funkcí může být více.
- Funkce vrací výsledek v registru `eax` (resp. `rax`) a je ukončena instrukcí `ret`.
- V jazyce C definujeme prototypy daných funkcí a voláme je standardním způsobem.
- Při sestavování programu sloučíme kód v C a v assembleru.

Aby funkce mohly dělat něco smysluplného, je potřeba jim předat argumenty. Pokud budeme uvažovat unixový operační systém a funkce mající jen celočíselné argumenty, kterých není více než šest, můžeme předpokládat, že argumenty jsou uloženy v následujících registrech v tomto pořadí: `rdi`, `rsi`, `rdx`, `rcx`, `r8`, `r9`.

Dále platí, že obsah registrů: `rbx`, `rsp`, `rbp`, `r12`, `r13`, `r14`, `r15`, musí být na konci funkce stejný, jako na jejím začátku.

Dále platí, že obsah registrů: `rbx`, `rsp`, `rbp`, `r12`, `r13`, `r14`, `r15`, musí být na konci funkce stejný, jako na jejím začátku.

[5] Pokud pracujeme s kladnými čísly, stačí zajistit, že obsah `edx` (resp. `rdx`) bude 0.

Tato informace je natolik zásadní, že je zde uvedena pro jistotu dvakrát!!! (A se třemi vykřičníky!!!) Nedodržení tohoto pravidla může vést (a často vede) k tomu, že se v programu objeví chyby, často na na první pohled nesouvisejících místech.

### 3.3.1 Ukázka použití

Nyní si ukážeme implementaci dvou funkcí, jedna bude inkrementovat hodnotu svého argumentu o jedna, druhá spočítá obvod obdélníka.

#### Část v assembleru

Soubor: `tutorial03.asm`

```
global inc1
global rectangle_circumference

section .text

;;
;; funkce majici jeden argument, vracejici hodnotu o jedna vyssi
;;
inc1:
    mov eax, edi    ; presune prvni argument do registru eax
    add eax, 1      ; zvysi hodnotu o jedna
    ret             ; navrat z funkce

;;
;; vypocet obvodu obdelnika
;; funkce ma dva argumenty (velikost strany obdelnika)
;;
rectangle_circumference:
    mov eax, edi    ; ulozi do eax jednu stranu obdelnika
    add eax, esi    ; pricte druhou stranu
    add eax, eax    ; vynasobi dvema
    ret
```

#### Část v C

Soubor: `tutorial03-test.c`

```
#include <stdio.h>

/* prototypy funkci napsanych v assembleru */
int inc1(int arg);
```

```
int rectangle_circumference(int a, int b);

int main()
{
    printf("5 + 1: %i\n", inc1(5));
    printf("obvod obdelnika o stranach 4x5: %i\n", rectangle_circumference(4, 5));
    return 0;
}
```

**Makefile**

```
tutorial03-test: tutorial03-test.o tutorial03.o
    gcc -o tutorial03-test tutorial03-test.o tutorial03.o

tutorial03-test.o: tutorial03-test.c
    gcc -c tutorial03-test.c

tutorial03.o: tutorial03.asm
    nasm -f elf64 tutorial03.asm
```

## 3.4 Úkoly k procvičení

1. Napište v assembleru funkci `int obsah_obdelnika(int a, int b)`, která spočítá obsah obdélníka.
2. Napište v assembleru funkci `int obvod_ctverce(int a)`, která spočítá obvod čtverce.
3. Napište v assembleru funkci `int obsah_ctverce(int a)`, která spočítá obsah čtverce.
4. Napište v assembleru funkci `int obvod_trojuhelnika(int a, int b, int c)`, která spočítá obvod trojúhelníka.
5. Napište v assembleru funkci `int obvod_trojuhelnika2(int a)`, která spočítá obvod rovnostranného trojúhelníka.
6. Napište v assembleru funkci `int obsah_trojuhelnika2(int a, int b)`, která spočítá obsah pravoúhlého trojúhelníka.
7. Napište v assembleru funkci `int obsah_trojuhelnika3(int a, int va)`, která
8. spočítá obsah trojúhelníka z velikosti strany a příslušné výšky.
9. Napište v assembleru funkci `int objem_krychle(int a)`, která spočítá objem krychle.
10. Napište funkci unsigned `int avg(unsigned int a, unsigned int b, unsigned int c)` pro výpočet aritmetického průměru tří čísel typu unsigned int.

# 4

# Řízení výpočtu

V tomto cvičení si ukážeme, jak jsou na úrovni procesoru řešeny konstrukce pro řízení výpočtu, které můžeme znát z vyšších programovacích jazyků. Zejména si ukážeme, jak lze realizovat podmínky a cykly.

## 4.1 Skoky

Společným nástrojem pro implementaci řídících struktur jsou tzv. *skoky*, které mohou přenést provádění výpočtu na zadanou adresu. Připomeňme, že prováděný program je uložen v paměti jako data a registr `rip` ukazuje na adresu instrukce, která má být provedena jako další. Standardně jsou instrukce prováděny v řadě za sebou, avšak změnou hodnoty v registru `rip` můžeme určit, jaký kód se má provádět jako další. Ze své povahy registr `rip` patří mezi řídící registry a není možné jej měnit přímo, např. instrukcemi typu `mov`. Je proto potřeba použít speciálních instrukcí, které se označují jako skoky a které, obrazně řečeno, provedou „skok" na zadanou adresu v programu, odkud se bude další kód provádět. Skoky se dělí na dva základní typy: (i) *nepodmíněné* a (ii) *podmíněné*.

### 4.1.1 Nepodmíněné skoky

Nepodmíněné skoky vždy převedou řízení výpočtu na zadanou adresu. Na platformě x86 se pro nepodmíněné skoky používá instrukce `jmp r/m/i`, která má právě jeden operand, kterým je adresa, na kterou má být proveden skok.

Nejtypičtěji se instrukce `jmp` používá s operandem, kterým je konstanta udávající konkrétní adresu cíle skoku. Avšak při zápisu kódu v assembleru nevíme, na jakých adresách jsou (resp. budou) jednotlivé instrukce uloženy. Proto se namísto konkrétní adresy uvádí tzv. *návěští* (anglicky *label*), které představuje symbolické pojmenování adresy, kam může být provedený nějaký skok, a ve fázi překladu se assembler postará o korektní doplnění adresy.[1] Ukažme si to na příkladu.

```
1     mov eax, 0x42
2 foo:
```

[1] S návěštími jsme se již setkali při vytváření funkcí, kdy návěští identifikují začátek (vstupní bod) funkce. V obou případech (ať už se jedná o skoky, či volání funkcí) návěští zastupují adresu v kódu programu.

```
3       inc eax
4       jmp foo
```

Na prvním řádku jsme do registru `eax` uložili hodnotu, ta pro tento příklad není důležitá. Na druhém řádku máme deklarováno návěští `foo`. Deklarace je ve tvaru *jméno následované dvojtečkou*.[2] Samotná deklarace návěští nemá žádný konkrétní vliv na vygenerovaný kód, je to jen pojmenování místa v kódu. Ve skutečném programu by mělo být pojmenování návěští voleno tak, aby bylo jasné, jaký význam má daný úsek kódu.[3]

Na řádku číslo 3 máme instrukci, která zvýší hodnotu registru `eax` o 1, opět to v našem ukázkovém příkladu nehraje zásadní roli. Na čtvrtém řádku máme instrukci `jmp`, která provede skok na adresu, která je dána návěštím `foo`.[4] Tj. znovu se provede instrukce `inc` a skok `jmp`. To se bude opakovat do doby, než program násilně ukončíme. Jak ukončit smyčku si ukážeme v následující kapitole.

Jak jsme již zmínili, nejčastěji se používá skok na konkrétní adresu, tj. kdy se instrukce `jmp foo` převede na tvar `jmp 0x12`. V případě architektury AMD64 se používají tzv. relativní skoky, kdy cílová adresa je vztažena k aktuální adrese, tj. k registru `rip`. Máme-li například instrukci `jmp 0x12`, bude skok provedený na adresu `rip + 0x12`.

Je možné použít variantu instrukce `jmp qword [rbx]`, kdy nejdříve ze zadané adresy (uložené v registru `rbx`) přečteme adresu, kam se má provést skok, a až na takto získanou adresu skok provedeme. Tato varianta umožňuje efektivně implementovat konstrukce typu `switch-case` nebo virtuální metody v objektově orientovaných jazycích.

Všimněme si, že instrukce nepodmíněného skoku umožňuje provést skok na libovolné místo v programu. To dává prográmatorovi velice široké možnosti pro jeho použití. Avšak při použití skoků je dobré se držet logického členění programu, které rámcově kopíruje bloky kódu, jak je známe z vyšších programovacích jazyků. V opačném případě hrozí, že program bude buď špatně srozumitelný nebo bude obsahovat chyby.

Z pohledu programování mají takto široce pojaté skoky koncepční nedostatky, viz legendární článek *E. Dijkstra: Go To Statement Considered Harmful*,[5] a proto dnešní programavocí jazyky skoky téměř nepoužívají nebo používají skoky jen s omezenými možnostmi. Na úrovni jednotlivých instrukcí procesoru se však skokům vyhnout nedá.

### 4.1.2 Podmíněné skoky

Druhou variantou, jak realizovat řízení výpočtu, jsou podmíněné skoky. To jsou skoky, k jejichž provedení dojde pouze v případě, že je splněna zadaná podmínka, jinak program pokračuje následující instrukcí. Podmínky, jež se využívají u podmíněných skoků, jsou určeny registrem `rf`, kde jsou mimo jiné čtyři jednobitové příznaky ZF, CF, SF, OF, které jsou nastaveny po provedení aritmetických operací.[6] Tyto příznaky mají následující význam:

- ZF (zero flag) – výsledek byl nula,

[2] Pro lepší čitelnost se návěští píšou jako předsazená jednotlivým instrukcím

[3] Všimněme si analogie s pojmenováváním proměnných ve vyšších programovacích jazycích.

[4] Při překladu assembler doplní konkrétní adresu.

[5] `https://homepages.cwi.nl/~storm/teaching/reader/Dijkstra68.pdf`

[6] Pozor, ne všechny instrukce nastavují všechny příznaky. Informace o tom, jaké příznaky jsou nastavovány, je potřeba čerpat z dokumentace procesoru.

- SF (sign flag) – výsledek je nezáporný (0) nebo záporný (1),[7]
- CF (carry flag) – výsledek je větší nebo menší než největší/nejmenší možné číslo,
- OF (overflow flag) – příznak přetečení znaménkové hodnoty mimo daný rozsah.

Pro každý z těchto čtyř příznaků existují dvě instrukce, které provedou skok, pokud je příznak nastaven, nebo nenastaven. Ukažme si to na příznaku ZF. Pro tento příznak existuje instrukce `jz`[8], která provede skok, pokud je příznak ZF nastaven na 1 (tj. předchozí operace skončila nulou), a dále existuje instrukce `jnz`[9], která skok provede, pokud je příznak ZF nastaven na 0 (tj. předchozí operace neskončila nulou). Použití ukazuje následující kód.

```
1       sub eax, 1
2       jz foo
3       inc ebx
4   foo:
```

Na prvním řádku jsme provedli aritmetickou operaci (odečetli jsme jedničku od registru `eax`). Tato operace nastavila příznaky v registru `rf` a pomocí instrukce `jz` můžeme otestovat, zda byl výsledek nula. Pokud ano, instrukce provede skok na místo v programu dané návěštím `foo`. V opačném případě se pokračuje následující instrukcí, což je v tomto případě instrukce na třetím řádku, která zvýší hodnotu v registru `ebx` o jedna.

Podobně existují podmíněné skoky pro příznaky SF (`js`, `jns`), CF (`jc`, `jnc`), OF (`jo`, `jno`).

Práce na úrovni jednotlivých příznaků a aritmetických operací není úplně komfortní a není s to podchytit řadu běžných situací, které v programech nastávají. Proto instrukční sada procesorů x86 obsahuje instrukci `cmp r/m, r/m/i`, která porovná dvě hodnoty tak, že je od sebe odečte a nastaví příznaky v registru `rf`.[10]

Vedle této instrukce existují dvě sady instrukcí podmíněných skoků, které slouží k porovnání celočíselných hodnot tak, jak se běžně používá ve vyšších programovacích jazycích. Dvě sady potřebujeme, protože musíme rozlišovat mezi tím, zda porovnáváme znaménkové nebo neznaménkové hodnoty.

Následující tabulka ukazuje instrukce pro porovnání znaménkových hodnot.

| instrukce | alt. jméno | příznaky | podmínka |
|---|---|---|---|
| `jg` | `jnle` | (SF = OF) & ZF = 0 | $A > B$ |
| `jge` | `jnl` | (SF = OF) | $A \geq B$ |
| `jl` | `jnge` | (SF $\neq$ OF) | $A < B$ |
| `jle` | `jng` | (SF $\neq$ OF) nebo ZF = 1 | $A \leq B$ |

Písmeno `g` v názvu instrukce značí *greater* (větší) a `l` značí *lesser* (menší), tj. `jg` odpovídá jump-if-greater apod.

Podobnou sadu podmíněných skoků pro porovnání neznaménkových hodnot popisuje následující tabulka.

[7] Odpovídá kopii nejvyššího bitu výsledku.
[8] jump-if-zero
[9] jump-if-not-zero
[10] Chová se podobně jako instrukce `sub`, avšak nemění žádný ze svých operandů.

| instrukce | alt. jméno | příznaky | podmínka |
|---|---|---|---|
| `ja` | `jnbe` | (CF or ZF) = 0 | $A > B$ |
| `jae` | `jnb` | CF = 0 | $A \geq B$ |
| `jb` | `jnae` | CF = 1 | $A < B$ |
| `jbe` | `jna` | (CF or ZF) = 1 | $A \leq B$ |

Písmeno `a` v názvu instrukce značí *above* (větší) a `b` značí *below* (menší), tj. `ja` odpovídá jump-if-above apod.

Následující příklad ukazuje použití podmíněných skoků pro výpočet absolutní hodnoty.

```
1  ;;
2  ;; funkce pro vypocet absolutni hodnoty
3  ;; nazev `absi' zvolen z duvodu kolize s klicovym slovem `abs'
4  ;;
5  ;; int absi(int n);
6  ;;
7  absi:
8          mov eax, edi   ; precteni argumentu n
9          cmp eax, 0     ; porovnani s 0
10         jge konec      ; skok na konec funkce
11         neg eax        ; otoceni znameka
12 konec:
13         ret            ; navrat z funkce
```

Nejdříve si do registru `eax` uložíme argument funkce, číslo `n`, to je uloženo v registru `edi`. Následně hodnotu tohoto argumentu porovnáme s 0. Pokud je hodnota větší nebo rovna 0, provedeme skok na návěští `konec`. Připomeňme, že pracujeme se znaménkovou hodnotou, a proto jsme použili instrukci `jge`. Pokud je hodnota menší než 0, pokračujeme další instrukcí a provedeme negaci (řádek s číslem 11). Na konci se nám obě větve výpočtu opět spojí a provedeme návrat z funkce instrukcí `ret`.

V předchozí kapitole jsme viděli, jak lze pomocí instrukce nepodmíněného skoku implementovat nekonečnou smyčku. Pomocí podmíněného skoku však můžeme do takto sestrojené smyčky vložit koncovou podmínku a cyklus ukončit.[11] Jak může vypadat spojení podmíněného a nepodmíněného skoku ukazuje následující příklad funkce počítající faktoriál čísla `n`.

```
;;
;; funkce pro vypocet faktorialu
;; int fact(int n);
;;
fact:
     mov ecx, edi    ; ecx -- vstupni argument (n)
     mov eax, 0x1    ; eax -- strada vyslednou hodnotu tj. n * (n - 1) * (n - 2) * ... * 1
fact_loop:
```

[11] Někdy se též říká „vyskočit z cyklu".

```
    cmp ecx, 0x0
    jle konec       ; narazili jsme na konec, ukoncime funkci
    imul ecx        ; vynasob eax hodnotou n
    sub ecx, 0x1    ; opakuj pro n - 1
    jmp fact_loop
konec:
    ret
```

### Praktická poznámka závěrem

Instrukce (podmíněných) skoků zásadním způsobem ovlivňují naplnění instrukční pipeline a tím rychlost zpracování programů, proto pokud můžeme instrukci skoku ušetřit, je to vždy vítané.

U začínajících programátorů[12] je možné najít konstrukce typu:

```
    cmp eax, 0
    jge foo
    jmp bar
foo:
    ;; nejaky kod
bar:
```

Takto napsaný kód sice nejspíš bude provádět, co jeho autor zamýšlel, ale není úplně dobrý, protože jednak bude tento kód špatně čitelný a také obsahuje instrukci navíc. Můžeme jej totiž přepsat do podoby.

```
    cmp eax, 0
    jl bar
foo:
    ;; nejaky kod
bar:
```

## 4.2 Úkoly k procvičení

1. Napište funkci `int sgn(int i)`, která vrací hodnoty -1, 0, 1 v závislosti na tom, zda-li je hodnota i záporná, nulová nebo kladná.
2. Napište funkci `char max2c(char a, char b)`, která vrací největší hodnotu. Vyzkoušejte, že funkce funguje správně pro kladné i záporné argumenty, i jejich kombinaci.
3. Napište funkci `unsigned short min3us(unsigned short a, unsigned short b, unsigned short c)`, která vrací nejmenší hodnotu ze zadaných parametrů. Vyzkoušejte, že funkce funguje správně i pro hodnoty větší než 32768.
4. Napište funkci `int kladne(int a, int b, int c)`, která vrací 1, pokud jsou všechny argumenty kladné, jinak 0.

[12] Resp. těch, kteří teprvé získávají zkušenost s programovaním v assembleru.

5. Napište funkci `int mocnina(int n, unsigned int m)` vracející mocninu $n^m$.

6. Do registrů `al`, `bl` vložte vhodné hodnoty, proveďte s nimi operace add a sub a pomocí instrukcí `jz`, `js`, `jc` a `jo` ověřte, zda byl nastavený příznak, nebo ne.

# 5

# Přístup do paměti

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.

## 5.1 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 $a + 1$.

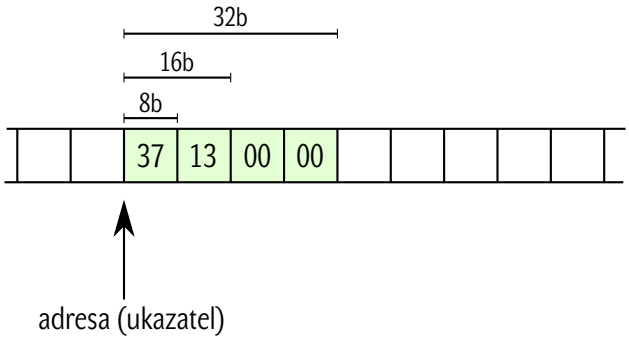
Hodnoty v paměti počítače jsou uloženy v po sobě následujících buňkách, např. 32bitová celá čísla (typu `int`) obsadí čtyři po sobě jdoucí buňky, 16bitové hodnoty dvě buňky atd. Jelikož jsou hodnoty uloženy v konkrétních paměťových buňkách, můžeme určit adresu, kde se hodnota nachází. Ta odpovídá adrese 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 5.1.

## 5.2 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`.

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




Obrázek 5.1: Ilustrace uložení hodnoty v paměti

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.

## 5.3 Použití společně s datovými typy

### 5.3.1 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
```

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);
```

### 5.3.2 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 prvku
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] Jako všechny ukazatele na platformě AMD64.

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

### 5.3.3 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.

```
;;
;; 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);
```

### 5.3.4 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`.

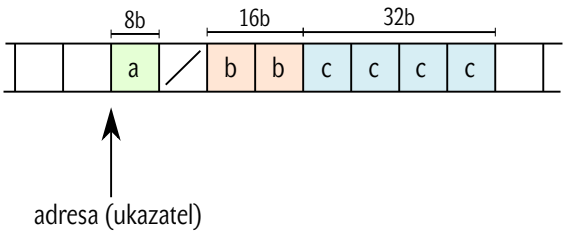
Při práci se strukturami je potřeba brát v potaz, že pro efektivnější práci se strukturami dochází k tzv. zarovnání velikosti struktury (padding) na vhodný násobek 4 nebo 8 B. To znamená, že i když výše uvedená struktura potřebuje k uchování dat pouze 6 bytů, ve skutečnosti zabere v paměti 8 bytů. Ověřte 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 5.2. 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 5.2: 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
```

## 5.4 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.

# 6

# 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.

## 6.1 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 `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).

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);
}
```

### 6.1.1 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 prvni 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.[1] Konkrétní adresa bude doplněna ve fázi linkování.

### 6.1.2 Mírně složitější volání funkce

Uvažujme funkci, která bude představovat odpočet hodnot, tj. bude na standardní výstup vypisovat hodnoty od $n, n-1, n-2, \ldots, 1, 0$. 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
```

[1] Dalo by se říct, že `extern` je protipólem direktivy `global`.

```
        mov al, 0
        call printi             ; zavolame funkci printi

        sub ecx, 1              ; snizime hodnotu o 1
        jns countdown_loop      ; pokud je vysledek nezaporny, opakujeme

        ret
```

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.

#### 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.

### 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                ; ulozime obsah rbx
3          mov ebx, edi            ; registr 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
```

### 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 hodnot, se kterými 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. epilog.

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

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`.

## 6.2 Poznámky

### 6.2.1 Zarovnání zásobníku

Specifikace Linuxového ABI[2] 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.

### 6.2.2 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`.

## 6.3 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] `https://refspecs.linuxbase.org/elf/x86_64-abi-0.99.pdf`

2. Napište funkci `void print_rect(int rows, int cols)`, která s pomocí volání funkce `print_row` vykreslí na standardní výstup vyplněný obdélník skladající 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 řetezce `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`.

# 7

# Systémová volání

V předchozích cvičeních jsme při programování velkou měrou využívali propojení mezi jazykem symbolických adres (assemblerem) a jazykem C. Jazyk C a jeho standardní knihovna nás odstiňoval od přímé komunikace s operačním systémem, resp. jeho jádrem. Například, pokud jsme chtěli vypsat něco na standardní výstup, bylo to realizováno funkcí `printf`. Ta se postarala o (i) sestavení vypisovaného řetězce a (ii) jeho zápis na standardní výstup. První část, sestavení řetezce, je obvykle vyřešena v rámci standardního kódu v jazyce C, druhá část, zápis na standardní výstup, je již řešena jádrem operačního systému. V tomto cvičení si ukážeme, jakým způsobem jsou na platformě Linux (AMD64) řešena systémová volání, tj. volání jádra OS, a jak sestavit plnohodnotnou aplikaci jen s využitím kódu v assembleru.

## 7.1 Minimální program

Každý operační systém poskytuje uživatelským procesům sadu služeb, jako je vytvoření souboru, zápis/čtení souboru, spuštění nového procesu apod. Tyto funkce jsou obvykle poskytovány jádrem operačního systému pomocí jednoznačně definovaného rozhraní.

### 7.1.1 Systémové volání

V případě operačního systému Linux (na platformě AMD64) je toto rozhraní realizováno následovně:

1. každá služba OS (např. otevření souboru, změna adresáře) je identifikována číslem, které je uloženo v registru `rax`,
2. argumenty předávané jádru (např. název souboru, příznaky) jsou uloženy v registrech `rdi`, `rsi`, `rdx`, `r10`, `r8`, `r9` (v tomto pořadí),
3. služba OS je zavolána instrukcí `syscall`,
4. návratová hodnota je uložena v registru `rax` (záporné hodnoty indikují chybu), obsah registrů `rcx` a `r11` může být změněn, obsah dalších registrů je zachován.

### 7.1.2 Implementace minimálního programu

Každý program by měl obsahovat minimálně jedno systémové volání. Jedná se o volání `exit`,[1] které se postará o to, že aktuálně běžící proces je ukončen. Systémové volání `exit` má jeden parametr, který udává kód, s jakým byl proces ukončen.[2] Kompletní výčet služeb a jejich čísel, který lze snadno procházet, včetně odkazů na související dokumentaci, najdete například na stránkách Chromium OS,[3] který je postavený na Linuxu.

Systémové volání `exit` má v Linuxu na platformě AMD64 přiřazený kód 60 (dekadicky).[4]

#### Program

Nyní máme všechny potřebné informace k vytvoření nejmenšího možného programu. Ten by mohl vypadat následovně.

```
1 global _start
2
3 SYS_EXIT        equ 60
4
5 section .text
6 _start:
7         mov rax, SYS_EXIT
8         mov rdi, 42
9         syscall
```

Samotné systémové volání je realizováno na řádcích 7 až 9, kdy do registru `rax` je přiřazeno číslo služby, do registru `rdi` návratový kód programu a instrukce `syscall` provede samotné systémové volání, v jehož důsledku dojde k ukončení aktuálně běžícího programu.

V tomto příkladu máme několik věcí, které s vykonáváním kódu přímo nesouvisí, ale jsou pro něj zásadní. Jednak je to direktiva `equ`, která slouží k definici konstant. Na levé straně této direktivy je symbolické pojmenování (např. `SYS_EXIT`) a na pravé hodnota (např. 60), kterou bude každý výskyt tohoto symbolického pojmenování nahrazen. V našem případě bude na řádku 7 do registru `rax` přiřazena hodnota 60. Význam těchto konstant je dvojí, jednak nám umožňuje zlepšit čitelnost kódu (místo čísla známe z pojmenování jeho význam) a v případě potřeby můžeme snadno změnit hodnotu na všech místech, kde se tato konstanta používá.

Další věcí, kterou je nutné u tohoto příkladu zmínit, je návěští `_start`, které představuje vstupní bod programu, jinými slovy adresu, odkud se začne program vykonávat. Toto návěští musí být deklarované jako `global`, aby linker byl schopen identifikovat danou adresu v programu.[5]

[1] Někdy též označované jako `sys_exit`, aby bylo zřejmé, že se jedná o systémové volání.

[2] Hodnota 0 obvykle indikuje korektní ukončení, jiná hodnota chybu.

[3] `https://www.chromium.org/chromium-os/developer-library/reference/linux-constants/syscalls/`

[4] Na jiných platformách se mohou čísla služeb lišit.

[5] Máme-li program v jazyce C, i ten je spouštěn od adresy dané symbolem `_start`. Na této adrese se obvykle nachází kód, který se postará o zpracování argumentů a zavolá funkci `main`, ta vykoná program, a její návratová hodnota je pak předána operačnímu systému pomocí volání `exit`.

#### Překlad

Abychom program mohli spustit, musíme jej vhodným způsobem přeložit. Nejdříve sestavíme objektový soubor. K tomu použijeme `nasm` způsobem, jako jsme používali již dříve. Rozdíl je ve vytvoření spustitelného binárního souboru, kdy k linkování nepoužijeme překladač `gcc` jako dříve, ale použijeme přímo `ld`. Jelikož máme kód navržený tak, aby co nejvíc vyhovoval potřebám linkování a nepřipojujeme žádné knihovny, použijeme jen přepínač `-o`, který udává název vygenerovaného binárního souboru. Odpovídající `Makefile` vypadá následovně.

```
tutorial07: tutorial07.o
        ld -o tutorial07 tutorial07.o

tutorial07.o: tutorial07.asm
        nasm -f elf64 tutorial07.asm
```

#### Spuštění

Program po svém spuštění (`./tutorial07`) neudělá nic a ihned se ukončí. Abychom ověřili, že program pracuje správně, použijeme proměnnou `$?` shellu, která obsahuje návratový kód naposledy spuštěného programu. Měli bychom tedy dostat:

```
$ ./tutorial07
$ echo $?
42
```

## 7.2 Hello World

Nyní si ukážeme složitější příklad, který na standardní výstup vypíše řetězec `Hello World!`.

```
global _start

;
; deklarace konstant
;
SYS_WRITE       equ 1   ; systemove volani pro zapis do souboru
SYS_EXIT        equ 60  ; systemove volani pro ukonceni programu
STDOUT          equ 1   ; deskriptor souboru standardniho vystupu
STR_HELLO_LEN   equ 13  ; delka vypsaneho retezce

;
; spustitelny kod
;
section .text
_start:
```

```
        mov rax, SYS_WRITE      ; vypsani retezce Hello World
        mov rdi, STDOUT
        mov rsi, str_hello
        mov rdx, STR_HELLO_LEN
        syscall

        mov rax, SYS_EXIT       ; ukonceni programu
        mov rdi, 42
        syscall

;
; (inicializovana) data programu
;
section .data
str_hello:
        db "Hello World!", 10
```

V tomto příkladu využíváme systémové volání `write`, které má tři parametry: (i) deskriptor souboru, do kterého se bude zapisovat, (ii) řetezec, který se má zapsat do souboru, (iii) délka řetězce. Protože, chceme zapisovat na standardní výstup a ten je reprezentován souborem s deskriptorem 1, přiřadíme tuto hodnotu do registru `rdi`, délku řetezce (tj. 13) přiřadíme do registru `rdx` a zbývá se vypořádat s adresou resp. uložením vypisovaného řetezce.

Pro data jsou v kódu, ať už assembleru nebo výsledném binárním souboru, vyčleněny samostatné sekce:

- `.data` (obecná data),
- `.rodata` (data jen pro čtení),
- `.bss` (neinicializovaná data).

V našem příkladu jsme použili sekci `.data` a umístili do ní textový řetězec pomocí pseudo-instrukce `db`. Pseudo-instrukce `db` umožňuje definovat a alokovat místo pro jednobytové hodnoty, případně řetězce, jak lze vidět v našem příkladu. Alternativně lze pomocí pseudo-instrukcí `dw`, `dd` a `dq` vytvořit místo pro hodnoty o velikostech 2, 4 a 8 bytů. Odkaz na dané místo v paměti je v assembleru řešen standardním návěštím jako při skocích nebo při volání podprogramů.

Při spuštění jsou hodnoty ze sekcí `.data` a `.rodata` načtena ze souboru do paměti a program k nim může přistupovat pomocí instrukcí pro práci s pamětí.

## 7.3 Čtení dat ze standardního vstupu

V dalším příkladu si ukážeme čtení dat ze standardního vstupu a jejich opětovný výpis na standardní výstup. Tento příklad se bude lišit v tom, že bude používat další systémové volání a bude používat oblast neinicializovaných dat pro uložení načtených a vypisovaných hodnot.

```
global _start

;
; deklarace konstant
;
SYS_READ        equ 0   ; systemove volani pro cteni ze souboru
SYS_WRITE       equ 1   ; systemove volani pro zapis do souboru
SYS_EXIT        equ 60  ; systemove volani pro ukonceni programu
STDIN           equ 0   ; deskriptor souboru standardniho vstupu
STDOUT          equ 1   ; deskriptor souboru standardniho vystupu
BUFFER_SIZE     equ 64  ; velikost bufferu
EOK             equ 0   ; konstanta signalizujici, ze program skoncil v poradku
EINPUT          equ 1   ; konstanta signalizujici, ze program skoncil chybou

;
; spustitelny kod
;
section .text
_start:
        mov rax, SYS_READ       ; nacte data ze standardniho vstupu
        mov rdi, STDIN
        mov rsi, input_buffer
        mov rdx, BUFFER_SIZE
        syscall

        cmp rax, 0
        jl fail                 ; pokud je vysledek zaporny => chyba

        mov rdx, rax            ; rax obsahuje pocet nactenych bytu (predavame jako 3. argument)
        mov rax, SYS_WRITE
        mov rdi, STDOUT         ; vypisujeme na standardni vystup
        mov rsi, input_buffer
        syscall                 ; vypsani obsahu bufferu

        jmp success             ; korektni ukonceni programu

fail:                           ; chyba pri cteni dat
        mov rdi, EINPUT
        jmp exit

success:                        ; uspesne ukonceni programu
        mov rdi, EOK
```

```
exit:                           ; predpoklada, ze v rdi je navratovy kod, a ukonci program
        mov rax, SYS_EXIT
        syscall

section .bss
input_buffer:
        resb BUFFER_SIZE
```

Tento ukázkový příklad nejdříve načte data ze standardního vstupu, k tomu slouží volání `read`, kde jako deskriptor souboru uvedeme číslo 0 (tj. standardní vstup). Data jsou načtena do bufferu o velikosti `BUFFER_SIZE`. Tento buffer je umístěn v sekci neinicilizovaných dat (`.bss`) a je určen návěštím `input_buffer`. K alokaci místa je použita pseudo-instrukce `resb n`, která vyhradí úsek paměti o velikosti `n` bytů. Analogicky máme pseudo-instrukce `resw`, `resd`, `resq`, které vyhradí místo o `n` 16bitových, 32bitových a 64bitových slovech. Protože hodnoty v sekci `.bss` jsou neinicializované, nezabírají žádné místo v binárním souboru, tím se tato sekce liší od `.data` nebo `.rodata`.

Po provedení operace čtení se ověří, zda nedošlo k chybě. Systémové volání `read` v takovém případě vrací zápornou hodnotu, jinak vrací počet bytů, které se úspěšně podařilo načíst. Pokud došlo k chybě, je to signalizováno návratovým kódem programu. Pokud data byla úspěšně načtena, jsou obratem vypsána pomocí systémového volání `write` a program je ukončen s návratovým kódem 0.

To, že program funguje správně, můžeme ověřit například s pomocí příkazu `echo`.

```
$ echo "abc" | ./tutorial07
abc
```

Ladit program, který přistupuje přímo ke službám jádra operačního systému nemusí být úplně pohodlné. Užitečným pomocníkem je nástroj `strace`, který pro spuštěný program ukazuje, jaká systémová volání byla zavolána, s jakými parametry a jaké byly návratové hodnoty.

V našem případě by spuštění a výstup programu měl vypadat následovně:

```
$ echo "abc" | strace ./tutorial07
execve("./tutorial07", ["./tutorial07"], 0x7ffc3a341dc0 /* 100 vars */) = 0
read(0, "abc\n", 64)                    = 4
write(1, "abc\n", 4abc
)                       = 4
exit(0)                                 = ?
+++ exited with 0 +++
```

Ve výpisu vidíme spuštění programu, volání `read`, `write` i `exit`.

## 7.4 Úkoly k procvičení

Všechny následující programy vytvořte v assembleru bez použití kódu v jazyce C.

1. Vytvořte program `rect`, který na terminál vypíše obdélník složený ze znaků `'*'` o stranách $20 \times 5$.

2. Vytvořte program `mypwd`, který se bude chovat podobně jako standardní unixový příkaz pwd a vypíše na standardní výstup plnou cestu k aktuálnímu adresáři. Jaký je aktuální adresář zjistíte pomocí systémového volání `getcwd`.

3. Vytvořte program `mypwd2`, který vypíše jméno aktuálního adresáře, tj. jméno za posledním znakem `'/'`.

4. Upravte poslední ukázkový příklad tak, aby vracel počet řádků přečtených ze standardního vstupu. Tyto úpravy provádějte postupně.

   - Spočítejte řádky na vstupu a výsledek vraťte v návratovém kódu.
   - Spočítejte řádky a jejich počet vypište na standardní výstup.
   - Upravte program, aby pracoval s libovolně velkým vstupem, tj. zpracovával vstup, dokud systémové volání `read` nevrátí 0 nebo zápornou hodnotu.

# 8

# Windows API a základní práce s procesy

Operační systém Microsoft Windows poskytuje velmi širokou škálu funkcí, které mohou programy pro svůj běh využívat, počínaje základní prací se systémem, jako je práce s procesy a soubory, přes komunikaci po síti, a tvorbou grafických aplikací konče. Tato funkcionalita se souhrně označuje jako Windows API (nebo zkráceně WinAPI) a je poskytována jako sada funkcí jazyka C. Práce s tímto rozhraním má svá specifika, která si v tomto cvičení ukážeme.

## 8.1 Typový systém WinAPI

Rozhraní WinAPI využívá vlastní sadu datových typů, které narozdíl od jazyka C mají jednoznačně definované velikosti a rozsahy hodnot.

Výčet nejčastěji používaných typů ukazuje Tabulka 8.1.

| typ | význam |
|---|---|
| `DWORD` | 32bitové neznaménkové celé číslo |
| `WORD` | 16bitové neznaménkové celé číslo |
| `BYTE` | 8bitové neznaménkové celé číslo |
| `BOOL` | pravdivostní hodnota |
| `VOID` | neplatná hodnota |
| `LPDWORD` | ukazatel na hodnotu typu `DWORD` |
| `LPWORD` | ukazatel na hodnotu typu `WORD` |
| `LPBYTE` | ukazatel na hodnotu typu `BYTE` |
| `LPBOOL` | ukazatel na hodnotu typu `BOOL` |
| `LPVOID` | ukazatel na hodnotu typu `VOID` |
| `LPSTR` | ukazatel na hodnotu typu `char`, řetezec obsahující jednobytové znaky (ANSI) |
| `LPWSTR` | ukazatel na hodnotu typu `wchar_t`, řetezec obsahující jednobytové znaky (UNICODE) |
| `HANDLE` | obecný identifikátor objektu (technicky `void *`) |

Tabulka 8.1: Přehled nejčastěji používaných typů ve WinAPI.

Tento výčet není ani v nejmenším úplný, další typy lze nalézt v dokumentaci.[1] Může se stát, že datové typy, které používá WinAPI, nebudou zcela kompatibilní s typovým systémem jazyka C/C++. Zejména se to týká novějších verzí překladačů, které jsou při práci s datovými typy striktnější. V takovém případě je nutné upravit nastavení projektu a povolit benevolentnější práci s typy a to následovně: *Project* → *Properties* → *C/C++* → *Language* → *Conformance Mode* → ***Default***.

## 8.2 Práce s řetězci

Další úskalí, které WinAPI přináší, spočívá v práci s řetězci. WinAPI umožňuje pracovat s dvěma typy řetězců, které jsou označovány jako *ANSI* a *Unicode*. První typ řetězců používá pro reprezentaci znaků jednobytové hodnoty (typ `char`) a druhý typ používá „široké znaky" o velikost dva byty (typ `wchar_t`).

Pracuje-li některá z funkcí WinAPI s řetezci, je obvykle nabízena ve dvou variantách. Buď s s příponou A, pokud pracuje s ANSI řetezci, nebo s příponou W pracuje-li s „Unicode" řetezci, např. `CreateFileA` (ANSI) nebo `CreateFileW` (Unicode). Při programování se reálně používá makro `CreateFile`, které se podle nastavení projektu expanduje na `CreateFileA` nebo `CreateFileW`.

Standardně jsou k dispozici funkce pro práci s oběma typy řetězců a programátor má na výběr ze dvou možností: (i) Buď si zvolit jeden typ řetězců, nastavit jej v projektu, a ten konzistentně v rámci jedné aplikace používat nebo (ii) použít pomocné sady maker v hlavičkovém souboru `tchar.h`, které se expandují na daný typ funkcí pro práci s řetězci podle nastavení projektu. Toto řešení je univerzálnější, ale zápis kódu se bude lišit od tradičního kódu v jazyce C.

Rozdíly při práci s jednotlivými typy znaků nastiňuje Tabulka 8.2.

| | ANSI | Unicode | `tchar.h` |
|---|---|---|---|
| typ znaku | `char` | `wchar_t` | `_TCHAR` |
| řetězcový literál | `"abc"` | `L"abc"` | `_T("abc")` |
| hlavní funkce | `main` | `wmain` | `_tmain` |
| délka řetězce | `strlen` | `wcslen` | `_tcslen` |
| kopie řetezce | `strcpy` | `wcscpy` | `_tcscpy` |
| spojení řetezců | `strcat` | `wcscat` | `_tcscat` |
| porovnání řetezců | `strcmp` | `wcscmp` | `_tcscmp` |
| formátovaný výstup | `printf` | `wprintf` | `_tprintf` |

Tabulka 8.2: Práce s řetezci různých typů

Jaký typ znaků bude použit lze nastavit ve vlastnostech projektu: *Project* → *Properties* → *Advanced* → *Character Set*.

## 8.3 Rozhraní pro práci se soubory

Základní principy práce s WinAPI a to, jak pracovat s řetezci univerzálním způsobem, si ukážema na práci se soubory. Pro práci se soubory má Windows vlastní sadu funkcí.[2] Klíčovou funkcí je `CreateFile`,

[1] `https://docs.microsoft.com/en-us/windows/win32/winprog/windows-data-types`

[2] `https://docs.microsoft.com/en-us/windows/win32/api/fileapi/`

která slouží k vytvoření nebo otevření souboru.[3]

Funkce `CreateFile`,[4] jako svůj argument bere cestu k souboru a další příznaky, jak se souborem pracovat. Jedná se jednak o režim přístupu k souboru (zda jej chcem číst nebo do něj zapisovat), režim souběžného přístupu, zda může být otevřený soubor otevřen ještě jednou (např. pro čtení), jak se zachovat, pokud soubor existuje nebo neexistuje. Další příznaky, jako bezpečností atributy, atributy souboru nebo šablonu souboru nebudeme využívat. Otevření nebo vytvoření souboru pro zápis i čtení ilustruje následující kód.

```
#include <windows.h>
#include <tchar.h>
#include <stdio.h>
int _tmain(int argc, TCHAR* argv[])
{
    HANDLE hFile = CreateFile(
        _T("foo.txt"),                  // cesta k souboru
        GENERIC_READ | GENERIC_WRITE,   // rezim prace se souborem
        FILE_SHARE_READ,                // rezim sdileneho pristupu k souboru (pro cteni)
        NULL,                           // bezpecnostni atributy
        OPEN_ALWAYS,                    // jak nalozit s (ne)existujicimi soubory
        0,                              // priznaky souboru
        NULL);                          // odkaz na sablonu

    if (hFile == INVALID_HANDLE_VALUE) {
        _tprintf(_T("error opening file\n"));
        return -1;
    }

    _TCHAR* str = _T("Hello world!");
    if (WriteFile(hFile, str, sizeof(_TCHAR) * _tcslen(str), NULL, NULL))
        _tprintf(_T("ok"));

    CloseHandle(hFile);
    return 0;
}
```

Všimněme si, že návratovou hodnotou funkce `CreateFile` je hodnota typu `HANDLE`. Jedná se o unikátní identifikátor (v rámci spuštěného procesu) pro jednotlivé objekty poskytované jádrem operačního systému. Datový typ `HANDLE` se používá i pro práci s dalšími objekty, např. procesy, vlákny, synchronizačními objekty atd. Technicky se jedná o typ `void *`, ale správně bychom tuto vlastnost neměli předjímat.

Pokud operace `CreateFile` z nějakého důvodu selže, je jako návratová hodnota vrácena konstanta `INVALID_HANDLE_VALUE`. Podrobnosti o chybě můžeme zjistit pomocí funkce `GetLastError()`.

[3] Název `CreateFile` přirozeně asociuje vytvoření souboru, proto může být matoucí, že funkce slouží i k otevření existujícího souboru. Ve skutečnosti se slovo *Create* vztahuje k vytvoření objektu, který reprezentuje soubor, a v tomto smyslu je slovo *Create* používáno konzistentně i v dalších částech WinAPI.

[4] `https://docs.microsoft.com/en-us/windows/win32/api/fileapi/nf-fileapi-createfilew`

V ukázkovém kódu je na řádku 21 ukázán zápis do souboru pomocí funkce `WriteFile`.[5] Svým chováním se funkce příliš neliší od funkce `fwrite` ze standardní knihovny jazyka C, má navíc dva argumenty. Jedním je možné získat počet zapsaných bytů (předáváme ukazatel na hodnotu, kam se výsledek zapíše), druhým je struktura umožňující asynchronní práci se souborem, což nebudeme využívat.

Práci se souborem je nutné ukončit jeho uzavřením, k tomu slouží funkce `CloseHandle`. Tato funkce je obecná a slouží k práci i s dalšími objekty.

**Úkoly**

1. Spusťte program a podívejte se do vytvořeného souboru, ideálně s textovým editorem nebo prohlížečem, který umí soubor zobrazit v hexadecimální podobě (např. Total Commander).
2. Změnte nastavení znaků v projektu a opakujte krok 1.
3. Upravte program tak, aby přečetl obsah vámi zvoleného souboru a vypsal jej na terminál. Použijte funkci `ReadFile`.[6]

## 8.4 Vytvoření nového procesu

Vytvoření procesu je ve WinAPI přímočaré. Funkci `CreateProcess`[7] předáme název (resp. cestu) ke spouštěnému programu, argumenty, a další příznaky či odkazy na struktury, kterými se vytvoření souboru má řídit. Důležité jsou dvě struktury `STARTUPINFO` a `PROCESS_INFORMATION`. První struktura slouží k předání doplňujících informací pro spouštěný proces (např. pozice okna). Druhá struktura slouží k předání informací o vytvořeném procesu, např. handle na vytvořený proces. Při inicializaci struktury `STARTUPINFO` je nutné do atributu `cb` zadat velikost struktury v bytech.

```
#include <windows.h>
#include <tchar.h>

int _tmain(int argc, TCHAR* argv[])
{
    STARTUPINFO si;
    PROCESS_INFORMATION pi;

    ZeroMemory(&si, sizeof(si));
    si.cb = sizeof(si);
    ZeroMemory(&pi, sizeof(pi));

    // vytvori novy proces
    if (!CreateProcess(
        _T("C:\\WINDOWS\\system32\\notepad.exe"),          // cesta k programu
        _T("C:\\WINDOWS\\system32\\notepad.exe foo.txt"),  // uplny seznam argumentu
```

[5] https://docs.microsoft.com/en-us/windows/win32/api/fileapi/nf-fileapi-writefile
[6] https://docs.microsoft.com/en-us/windows/win32/api/fileapi/nf-fileapi-readfile
[7] https://docs.microsoft.com/en-us/windows/win32/api/processthreadsapi/nf-processthreadsapi-createprocessa

```
        NULL,       // nastaveni bezpecnostnich atributu
        NULL,       // nastaveni bezpecnostnich atributu
        FALSE,      // zakaz dedeni handlu mezi procesy
        0,          // priznaky pro vytvoreni procesu
        NULL,       // definice prostredi (pouzije se prostredi aktualniho procesu)
        NULL,       // pracovni adresar (pouzije se adresar aktualniho procesu)
        &si,        // ukazatel na strukturu s informacemi ke spoustenumu procesu
        &pi)        // ukazatel na strukturu, kterou jsou predany informace o vzniklem procesu
    ) {
        _tprintf(_T("CreateProcess failed (%d).\n"), GetLastError());
        return 1;
    }

    // ceka na ukonceni procesu
    WaitForSingleObject(pi.hProcess, INFINITE);

    // uzavre proces
    CloseHandle(pi.hProcess);
    CloseHandle(pi.hThread);
    return 0;
}
```

**Úkol:**

4. Rozšiřte ukázkový příklad o volání funkce `GetExitCodeProcess`,[8] která vrací návratový kód ukončeného procesu, a tento kód vypište.

V rámci operačního systému má každý proces přiřazené číslo (*process id* nebo zkráceně *pid*). Pomocí funkce `OpenProcess`[9] můžeme získat handle na existující proces na základě tohoto identifikátoru a to například následovně:

```
HANDLE hProcess = OpenProcess(PROCESS_ALL_ACCESS, FALSE, pid);
```

Pokud máme dostatečná oprávnění, můžeme s procesem dále pracovat, například zjistit o něm informace nebo jej ukončit.

**Úkol:**

5. Spusťte vhodnou aplikaci (notepad, kalkulačku), v TaskManageru zjistěte jeho pid.
6. Pomocí funkce `GetProcessImageFileName`[10] zjistěte cestu ke spuštěnému souboru.
7. Pomocí funkce `TerminateProcess`[11] jej ukončete.

[8] https://docs.microsoft.com/en-us/windows/win32/api/processthreadsapi/nf-processthreadsapi-getexitcodeprocess
[9] https://docs.microsoft.com/en-us/windows/win32/api/processthreadsapi/nf-processthreadsapi-openprocess
[10] https://docs.microsoft.com/en-us/windows/win32/api/psapi/nf-psapi-getprocessimagefilenamea
[11] https://docs.microsoft.com/en-us/windows/win32/api/processthreadsapi/nf-processthreadsapi-terminateprocess

# 9

# Základní práce s procesy v unixech

Vznik operačního systému Unix se datuje do přelomu šedesátých a sedmdesátých let minulého století. Omezený výkon počítačů té doby se propsal i do architektury tohoto operačního systému[1] a mimo jiné i do návrhu jeho rozhraní pro práci s procesy, které je ve srovnání s Windows výrazně jednodušší, přičemž pro drtivou většinu situací zcela dostačující.

## 9.1 Procesy v unixech

Historicky je v unixech základní entitou vykonávající program proces. Jinými slovy na proces můžeme nahlížet jako na instanci běžícího programu. K tradici unixových operačních systémů patří, že jednotlivé programy jsou malé, plnící jeden účel, a je možné je skládat do větších celků pomocí skriptů v unixovém *shell*u. Shell slouží jednak jako rozhraní pro interaktivní práci uživatelů formou příkazového řádku, ale protože se velice často jedná o plnohodnotný programovací jazyk, je možné v shellu sestavit program, který spouští a propojuje menší programy do větších programů.

**Poznámka:** V unixových operačních systémech se dá setkat s podrobnou dokumentací k jednotlivým programům/nástrojům. Ta je k dispozici jako program `man`, který (podle toho, co je nainstalováno) obsahuje jednak popis nástrojů (např. `man ls` vypíše dokumentaci k příkazu `ls`) nebo funkcí standardní knihovny, např. `man fork` zobrazí popis funkce `fork()`.

### 9.1.1 Identifikace a organizace procesů

V unixových operačních systémech tvoří procesy stromovou hierarchii, přičemž v kořeni tohoto stromu je proces označovaný jako `init`, který je spuštěn jako první proces po spuštění operačního systému. Podobně jako na platformě Microsoft Windows je každý proces identifikován svým číslem, které se označuje jako *proces id* (zkráceně pid).

Aktuální pid procesu lze získat pomocí funkce `pid_t getpid()`, jejíž prototyp je v hlavičkovém souboru `unistd.h`.

[1] A do operačních systémů z něj odvozených jako je např. GNU/Linux.

Pro získání přehledu o běžících procesech se v unixech používá příkaz `ps`. Ten bez parametrů vypíše seznam procesů spojených s aktuálním terminálem. Pomocí dalších přepínačů lze tento výpis rozšířit, např. `ps -u` vypíše všechny procesy daného uživatele, `ps aux` vypíše podrobné informace o všech procesech. Více viz dokumentace programu `ps`.

Pokud by nás zajímala hierarchie procesů, můžeme použít nástroj `pstree`, který jednotlivé procesy zobrazí jako strom.

**Úkoly:**

1. Napište program, který po svém spuštění vypíše své pid a bude provádět nějakou činnost, nedojde k jeho ukončení.
2. Identifikujte program pomocí nástroje `ps`.
3. Identifikujte program pomocí nástroje `pstree`.
4. Program ukončete pomcí kombinace kláves `ctrl+c`.

### 9.1.2 Vytvoření nového procesu

V unixových operačních systémech k vytvoření nového procesu slouží systémové volání `fork`, které je programům v jazyce C k dispozici jako funkce `pid_t fork()`, jejíž prototyp se nachází v hlavičkovém souboru `unistd.h`. Toto systémové volání vytvoří nový proces, který je potomkem procesu, jenž zavolal `fork`[2] a je to klon rodičovského procesu. To znamená, že rodič i potomek vykonávají stejný kód a každý má vlastní kopii dat. Rozdíl je v tom, že potomek má přiřazené nové unikátní id a rodič i potomek mají svůj oddělený paměťový prostor. Protože po zavolání `fork()` existují v paměti dva procesy vykonávající stejný kód, je potřeba je nějakým způsobem rozlišit. K tomu slouží návratová hodnota této funkce. Pokud se vykonávaný kód nachází v rodiči, funkce `fork` vrací pid potomka, pokud se vykonávaný kód nachází v potomkovi, vrací funkce `fork()` hodnotu 0. Pokud `fork()` vrátí zápornou hodnotu, došlo k chybě.

**Úkol:** Vytvořte program, který zavolá `fork()` a ověřte výše popsané chování.

### 9.1.3 Spuštění jiného programu

Vytvoření identické kopie procesu má pouze omezené použití a v praxi často potřebujeme spustit jiný program. K tomu slouží systémové volání `exec`, které do paměti nahraje kód programu a začne jej vykonávat. Toto systémové volání je k dispozici jako sada funkcí s prototypy v hlavičkovém souboru `unistd.h`.

```
int execl(const char *path, const char *arg, ...);
int execlp(const char *file, const char *arg, ...);
int execle(const char *path, const char *arg , ..., char * const envp[]);
int execv(const char *path, char *const argv[]);
int execvp(const char *file, char *const argv[]);
int execve(const char *filename, char *const argv [], char *const envp[]);
```

[2] Tj. rodičovského procesu.

Funkce `execl` a `execv` se liší ve formě argumentů, kdy první tři jsou funkce s proměnlivým počtem argumentů a poslední tři funkce své argumenty přebírají formou ukazatele na pole hodnot. V obou případech je konec pole signalizován hodnotu `NULL`. S výjimkou funkcí `execlp` a `execvp` se uvádí plná cesta k souboru s programem. Funkce `execlp` a `execvp` umí pracovat jen s jménem programu a vyhledávají daný program podle nastavení proměnné prostředí `PATH`. Funkce `execle` a `execve` předávají spouštěnému programu proměnné prostředí ve tvaru `promenna=hodnota`.

Ve všech případech, pokud došlo k selhání funkce, je vrácena záporná hodnota.

Pozor, jako první prvek pole argumentů, je předávaný název programu.

**Úkoly:**

5. S použitím `exec` spusťte program `uname --all`.
6. Ověřte, že pokud volání `exec` neselže, je nahrazen aktuální kód programu.
7. Spojte volání `fork` a `exec` tak, aby rodič vykonával nějakou činnost, zatímco potomek zavolá `uname --all` a ukončí se.

### 9.1.4 Ukončení procesu

Aktuálně běžící proces je možné ukončit systémovým voláním `exit`, které je realizováno jako funkce `void exit(int)`, kde argument odpovídá návratové hodnotě, která je předána rodiči. Používat by se měly hodnoty jen z rozsahu 0 až 127, další hodnoty mají svůj vyčleněný význam. Tuto funkci najdeme deklarovanou v hlavičkovém souboru `stdlib.h`, kde je k dispozici ještě prototyp funkce `void abort()`, která ukončí aktuálně běžící program a uloží na disk obraz paměti (tzv. core dump), který je možné použít další k analýze programu a identifikaci chyb.

K ukončení běžícího programu jiným procesem slouží nástroj `kill`, který zašle signál procesu, aby se ukončil.[3] Používá se ve tvaru `kill <pid>`.

**Úkoly:**

8. Upravte program(y) z předchozích úkolů tak, aby na neúspěšné volání `fork` nebo `exec` zareagovaly voláním `exit` nebo `abort`.
9. Ukončete nějaký běžící (ideálně nějaký méně důležitý) process s pomocí nástroje `kill`.

### 9.1.5 Čekání

Při souběžné práci s procesy nemáme garantované, že budou prováděny v určitém pořadí, ani jak bude jejich vzájemný souběh řešen. Při testování může být výhodné proces na nějakou dobu uspat. K tomu se dá použít funkce `unsigned sleep(unsigned seconds)`, která uspí aktuální proces na daný počet sekund.

#### Čekání na potomka

Často potřebujeme provést kód v potomkovi a v rodiči počkat až tento kód doběhne. K tomu máme dispozici funkce `pid_t wait(int *status)` a `pid_t waitpid(pid_t pid, int *status, int options)`

[3] Toto je výchozí chování, nástroj umožňuje zasílat i další signály.

v hlavičkovém souboru `sys/wait.h`. Obě tyto funkce zastaví běh programu, dokud neskončí některý potomek (funkce `wait`) nebo potomek s daným pid (funkce `waitpid`).[4] Obě funkce vrací pid potomka, a pokud ukazatel `status` není NULL, jsou vráceny informace o ukončení potomka. S těmi se pracuje pomocí sady maker, např.

- `WIFEXITED(status)` vrací nenulové číslo, pokud potomek skončil normálně,
- `WEXITSTATUS(status)` vrací návratový kód potomka, smí se použít, pokud je `WIFEXITED(status)` nenulová hodnota,
- `WIFSIGNALED(status)` vrací nenulové číslo, pokud byl potomek ukončen signálem,
- `WTERMSIG(status)` vrací číslo signálu, který proces ukončil, smí se použít, pokud je `WIFSIGNALED(status)` nenulová hodnota.

### Zombie procesy

Pokud potomek skončí a rodič na něj nečeká voláním `wait`, vznikne tzv. *zombie proces*, tj. proces, který skončil, ale ještě v systému existuje, dokud si rodičovský proces nevyzvedne informace o jeho ukončení pomocí `wait`. Pokud si rodič tyto informace nevyzvedne nikdy (ani po svém ukončení), zombie proces je adoptován procesem init, který jej odstraní ze systému.

**Úkol:**

10. Upravte předchozí úkol(y) tak, aby čekaly na dokončení potomka.

[4] Může se stát, že potomek dokončí svou činnost dřív, než je zavolána funkce `wait`, v takovém případě není běh programu pozastaven a funkce vrací výsledek okamžitě.

# 10

# Základní práce s vlákny ve Windows

V operačním systému Microsoft Windows základní jednotkou vykonávající program[1] je *vlákno*. V každém procesu může běžet jedno nebo více vláken, kdy vlákno může vykonávat libovolnou část kódu programu, včetně částí, které vykonávají jiná vlákna. Při spuštění procesu je spuštěno (primární) vlákno, které vykonává funkci `main`. Každé vlákno má vlastní kontext, tj. má přidělené registry a zásobník. Další prostředky zejména paměť[2] a systémové prostředky[3] jsou sdíleny mezi všemi vlákny, což umožňuje pohodlnou spolupráci více vláken při řešení úloh. Avšak je potřeba mít na paměti, že běh jednotlivých vláken je plánován operačním systémem a činnosti, jež vlákna provádí, se mohou téměř libovolně prolínat. Je proto nutné zajistit, aby v jeden okamžik s jedněmi daty nepracovalo více vláken současně. Toho lze dosáhnout buď pomocí synchronizace vláken nebo vhodně navrženým kódem, který zajistí, že nebudou měněna data, se kterými pracuje více vláken současně.

## 10.1 Vytvoření vlákna

K vytvoření vlákna slouží funkce, `CreateThread`[4] které vedle atributů definujících vlastnosti vytvořeného vlákna předáváme především odkaz na funkci, která představuje kód vykonávaný vláknem a odkaz na data, se kterými vlákno pracuje. Vytvoření vlákna ilustruje následující kód.

```
1 #include <windows.h>
2 #include <tchar.h>
3 #include <stdio.h>
4
5 #define COUNT   (20)
6
7 DWORD WINAPI ThreadFunc(LPVOID lpParam)
8 {
```

[1] instrukce programu
[2] včetně prováděného kódu
[3] objekty poskytované operačním systémem
[4] `https://docs.microsoft.com/en-us/windows/win32/api/processthreadsapi/nf-processthreadsapi-createthread`

```
9           _tprintf(_T("Spusteno vlakno s id: %i\n"), GetCurrentThreadId());
10          DWORD x = *(DWORD *)lpParam;
11          for (int i = 0; i < COUNT; i++) {
12                  _tprintf(_T("thr #%i: %i\n"), x, i);
13                  Sleep(1);
14          }
15
16          return 0;
17  }
18
19  int _tmain()
20  {
21          DWORD dwThreadId;
22          DWORD dwThrdParam = 1;
23
24          HANDLE hThread = CreateThread(
25                  NULL,                           // bezpecnostni atributy
26                  0,                              // velikost zasobniku (0 -> implicitni hodnota)
27                  ThreadFunc,                     // funkce provadena vlaknem
28                  &dwThrdParam,                   // argument predany vlaknu
29                  0,                              // priznaky pro vytvorene vlakno
30                  &dwThreadId);                   // vraci id vlakna
31
32          if (hThread == NULL) {
33                  _tprintf(_T("Vytvoreni vlakna selhalo\n"));
34                  ExitProcess(0);
35          } else {
36                  _tprintf(_T("Vytvoreno vlakno s id: %i\n"), dwThreadId);
37          }
38
39          for (int i = 0; i < COUNT; i++) {
40                  _tprintf(_T("thr #0: %i\n"), i);
41                  Sleep(1);
42          }
43
44          WaitForSingleObject(hThread, INFINITE);  // ceka na skonceni vlakna
45          CloseHandle(hThread);                    // ukonci praci s vlaknem
46  }
```

Při popisu tohoto kódu začneme uprostřed, na řádcích 24 až 30, kde dochází k vytvoření vlákna. Důležitý je třetí argument na řádku 27 udávající funkci, kterou bude vlákno vykonávat. Velmi často používáný je i čtvrtý argument na řádku 28, kterým formou ukazatele můžeme do vlákna předat data, se kterými

se bude pracovat.[5] Funkce `CreateThread` vrací handle na dané vlákno, se kterým můžeme následně pracovat. Zejména bychom měli (i) otestovat, zda došlo k vytvoření vlákna (řádky 32 až 35), (ii) počkat na dokončení vlákna (řádek 44) a ukončit práci s daným vláknem (řádek 45).

Součástí ukázkového kódu je i zjištění identifikátoru vlákna, viz šestý argument funkce `CreateThread` a řádek 36. Abychom vyvíjeli nějakou činnost v primárním vláknu, obsahuje funkce `main` na řádcích smyčku, která vypisuje čísla od 0 do `COUNT` a po každém výpisu se na 1 ms uspí pomocí volání funkce `Sleep`.[6]

Funkce vykonávající vlákno je typu `DWORD WINAPI ThreadFunc(LPVOID lpParam)`, kde argument `lpParam` je ukazatel předaný funkci `CreateThread`. V těle funkce můžeme vykonávat libovolný kód, jak ukazuje náš příklad na řádcích 7 až 17. V této funkci nejdříve vypíšeme id vlákna. Při spuštění programu si všimněme, že moment vytvoření vlákna a jeho spuštění nemusí být stejný. Ve výpisu programu se to projeví tak, že primární vlákno vstoupí do smyčky na řádcích 39 až 42, a až následně dojde ke spuštění vlákna.

Dále vlákno převezme svůj argument a začne ve smyčce (řádek 11 až 14) vypisovat text. Funkce vrací hodnotu typu `DWORD`, kterou můžeme předat informaci související s ukončením vlákna. Tuto hodnotu můžeme vyzvednout pomocí funkce `GetExitCodeThread`.[7]

**Úkoly:**

1. Odstraňte z kódu volání `Sleep`, případně změnte jeho argumenty, a sledujte, jak se změní průběh programu.

2. Rozšiřte ukázkový program tak, aby vyzvedl a vypsal návratovou hodnotu spuštěného vlákna. Zamyslete se nad tím, kam volání funkce `GetExitCodeThread` umístit.

3. Rozšiřte ukázkový program, aby pracoval obecně s `N` vlákny, kde `N` je konstanta zadaná v kódu programu.

## 10.2 Manipulace s vlákny

Operační systém Microsoft Windows nabízí širokou škálu funkcí pro práci s vlákny, některé již byly zmíněny v předchozí kapitole. Zmiňme si dále funkce:

- `GetCurrentThread`[8] vracející pseudo-handle sloužící k manipulaci s aktuálním vláknem. Pseudo-handle zde funguje jako zvláštní konstanta odkazující na aktuální vlákno a má platnost pouze v rámci daného vlákna. Pokud bychom chtěli získat plnohodnotný handle, musíme pseudo-handle převést pomocí funkce `DuplicateHandle`[9].

[5] Pokud předáváme odkaz na lokální proměnné, tj. data, která jsou alokována na zásobníku, je nutné zajistit, aby vlákno jež takto data předává, neskončilo dřív než vlákno jím spuštěné. Jinak by společně se skončeným vláknem zanikla i data umístěná na jeho zásobníku. Alternativně můžeme předávat dynamicky alokovaná data.

[6] `https://docs.microsoft.com/en-us/windows/win32/api/synchapi/nf-synchapi-sleep`

[7] `https://docs.microsoft.com/en-us/windows/win32/api/processthreadsapi/nf-processthreadsapi-getexitcodethread`

[8] `https://docs.microsoft.com/en-us/windows/win32/api/processthreadsapi/nf-processthreadsapi-getcurrentthread`

[9] `https://docs.microsoft.com/en-us/windows/win32/api/handleapi/nf-handleapi-duplicatehandle`

- `SuspendThread`[10] a `ResumeThread`[11] sloužící k uspání a probuzení vlákna. Funkce `SuspendThread` může být opakovaně zavolána i na již uspané vlákno. Každé vlákno má přiřazeno počítadlo, které indikuje počet uspání daného vlákna. Funkce `ResumeThread` toto počítadlo snižuje, a pokud toto počítadlo klesne na 0, je vlákno probuzeno.
- `ExitThread`[12] ukončí právě prováděné vlákno.
- `TerminateThread`[13] ukončí jiné vlákno na základě předaného handlu. Ukončení vlákna touto funkcí není korektní, může k němu dojít v libovolném bodě daného vlákna, a tím pádem se může program dostat do nekonzistentního stavu.

**Úkoly:**

4. Ukázkový příklad upravte tak, že po provedení `(COUNT / 2)` cyklů se vlákno uspí a je probuzeno v momentě, kdy primární vlákno zpracuje celou smyčku.
5. Naprogramujte funkci `int parfib(int)`, která spočítá Fibonacci číslo rekurzivní způsobem s využitím právě dvou vláken. Dvě počáteční větve výpočtu spusťte v samostatných vláknech.
6. Naprogramujte funkci `int pmin(int *numbers, unsigned int count, unsigned int threads)`, která použije `threads` vláken k tomu, aby v poli `numbers`, které obsahuje `count` hodnot, našlo nejmenší hodnotu.

[10] `https://docs.microsoft.com/en-us/windows/win32/api/processthreadsapi/nf-processthreadsapi-suspendthread`
[11] `https://docs.microsoft.com/en-us/windows/win32/api/processthreadsapi/nf-processthreadsapi-resumethread`
[12] `https://docs.microsoft.com/en-us/windows/win32/api/processthreadsapi/nf-processthreadsapi-exitthread`
[13] `https://docs.microsoft.com/en-us/windows/win32/api/processthreadsapi/nf-processthreadsapi-terminatethread`

# 11

# Základní práce s vlákny v Linuxu

V unixových operačních systémech byl z historických důvodů základní jednotkou vykonávající program[1] proces. Jelikož se ale koncept vláken osvědčil, byla později vlákna do unixových operačních systémů doplněna a práce s nimi je velmi podobná tomu, co jsme viděli v případě operačního systému MS Windows.[2] V případě operačního systému Linux se s vlákny interně pracuje podobně jako s procesy, každé vlákno má svou sadu registrů, zásobník atd. a paměťový prostor a systémové prostředky jsou sdíleny v rámci jednoho procesu.

## 11.1 Vytvoření vlákna

K vytvoření vlákna slouží funkce:

```
int pthread_create(
        pthread_t *thread,
        pthread_attr_t *attr,
        void *(*start_routine)(void*),
        void *arg)
```

Tato funkce má právě čtyři argumenty, prvním je ukazatel na hodnotu typu `pthread_t`, která obsahuje identifikátor vytvořeného vlákna a umožňuje s tímto vláknem pracovat. Druhý argument představuje atributy, které má vytvořené vlákno mít. Pokud použijeme hodnotu `NULL`, použije se výchozí nastavení. Třetí argument udává funkci, která představuje kód, který se má vláknem vykonávat. Tato funkce vrací hodnotu typu `void *` a akceptuje jeden argument typu `void *`, který je vláknu předán při jeho vytvoření. K tomu slouží čtvrtý argument funkce `pthread_create`.

Následující kód demonstruje vytvoření vlákna:

```
1 #include <stdio.h>
```

[1] instrukce programu

[2] Tím, že vlákna byla doplněna až později, vznikají ostré hrany a koncepční problémy, např. jak by se měl OS zachovat, pokud jedno z vláken zavolá `fork()`?

```c
#include <pthread.h>
#include <time.h>

#define COUNT   (20)

static void msleep(int ms)
{
        struct timespec t;
        t.tv_sec  = ms / 1000;
        t.tv_nsec = (ms % 1000) * 1000000;
        nanosleep(&t, NULL);
}

static void *thread_func(void *arg)
{
        int id = *((int *) arg);
        printf("Spusteno vlakno: %i\n", id);

        for (int i = 0; i < COUNT; i++) {
                printf("thr #%i: %i\n", id, i);
                msleep(5);
        }
        return (void *) 42;
}

int main()
{
        int id = 1;
        long result;
        pthread_t thread;
        if (pthread_create(&thread, NULL, thread_func, &id)) {
                fprintf(stderr, "Vytvoreni vlakna selhalo\n");
                return 1;
        }

        for (int i = 0; i < COUNT; i++) {
                printf("thr #main: %i\n", i);
                msleep(5);
        }
        pthread_join(thread, (void **) &result);
        printf("Result: %li\n", result);
        return 0;
}
```

Při popisu tohoto kódu začneme spíše od konce, od řádků 31 až 35, kde dochází k vytvoření vlákna, které je dané funkcí `thread_func` a kterému je předán ukazatel `&id`.[3] Pokud vytvoření vlákna z nějakého důvodu selže, je vrácena nenulová hodnota.

S vláknem můžeme pracovat pomocí hodnoty, která je určena prvním argumentem funkce `pthread_create`. Zejména bychom měli v některém z bodů programu počkat na dokončení daného vlákna. K tomu slouží funkce `pthread_join`, která jednak čeká na doběhnutí daného vlákna, a také umožňuje vyzvednout hodnotu vrácenou funkcí vláknem.[4]

Struktura programu se neliší od toho, co jsme viděli v předchozím cvičení. Narozdíl od předchozího cvičení ukázkový kód obsahuje pomocnou funkci `msleep`, která uspí aktuální vlákno na zadaný počet milisekund.[5]

Protože tento způsob práce s vlákny stojí mimo standardní knihovnu jazyka C, je potřeba při překladu buď použít přepínač `-pthread`[6] nebo připojit knihovnu `libpthread`, tj. použít přepínač `-lphtread`.[7]

### 11.1.1 Ukončení vlákna a uvolnění prostředků

Všimněme si, že při skončení práce s vláknem nikde explicitně neuvolňujeme prostředky s vláknem spojené. Jinými slovy chybí ekvivalent volání `CloseHandle`, jak jsme jej viděli ve Windows. Je to dáno tím, že o uvolnění těchto prostředků se stará funkce `pthread_join`.

Pokud bychom chtěli vlákno, které jen spustíme a necháme jej běžet s tím, že nás výsledek nezajímá, nebudeme mít v kódu vhodné místo pro volání `pthread_join`. V takovém případě musíme vlákno vytvořit s atributem `PTHREAD_CREATE_DETACHED`[8] nebo tento atribut nastavit za běhu funkcí `pthread_detach`. Pokud toto chceme nastavit u právě běžícího vlákna, můžeme použít funkci `pthread_self`, která vrací identifikátor aktuálně běžícího vlákna.

**Úkoly:**

1. Odstraňte z kódu volání `msleep`, případně změnte jeho argumenty, a sledujte, jak se změní průběh programu.

2. Rozšiřte ukázkový program, aby pracoval obecně s `N` vlákny, kde `N` je konstanta zadaná v kódu programu.

3. Naprogramujte funkci `int parfib(int)`, která spočítá Fibonacci číslo rekurzivní způsobem s využitím právě dvou vláken. Dvě počáteční větve výpočtu spusťte v samostatných vláknech.

[3] Pokud předáváme odkaz na lokální proměnné, tj. data, která jsou alokována na zásobníku, je nutné zajistit, aby funkce, jež takto data předává, neskončila dřív než vlákno, které vytvořila. Jinak by společně s ukončenou funkcí zanikla i data umístěná zásobníku. Alternativně můžeme předávat dynamicky alokovaná data.

[4] V ukázkovém případě se využívá přetypování mezi číslem a typem ukazatel. Toto je relativně běžný postup, jak předávat do (resp. z) vlákna celočíselné hodnoty.

[5] Standardně je k dispozici funkce `sleep`, která pracuje s rozlišením na sekundy a `nanosleep`, která pracuje s rozlišením na nanosekundy.

[6] Novější verze GNU/Linux.

[7] Starší verze GNU/Linux.

[8] Viz funkce `pthread_attr_init`.

4. Naprogramujte funkci `int pmin(int *numbers, unsigned int count, unsigned int threads)`, která použije `threads` vláken k tomu, aby v poli `numbers`, které obsahuje `count` hodnot, našlo nejmenší hodnotu.

# 12

# Vlákna v uživatelském prostoru

Při studiu operačních systémů jsme velice často limitováni jejich složitostí, která z velké míry promení ze složitosti soudobého hardwaru. Máme proto omezené možnosti, jak se prakticky podívat na to, jak jsou jednotlivé částí OS implementovány. V tomto cvičení si ukážeme, jak je možné implementovat vlákna v uživatelském prostoru. Díky tomu můžeme nahlédnout, jak je řešena správa procesoru a multiprogramování, aniž bychom potřebovali podrobné znalosti implementace OS. Implementace vláken v uživatelském prostoru totiž používá stejné principy jako jádro OS, avšak využívá mnohem jednodušší prostředky.

## 12.1 Veřejné rozhraní

Začneme popisem rozhraní, které pro práci s vlákny budeme používat. Naše implementace vláken v uživatelském prostředí bude s menšími odlišnostmi kopírovat rozhraní knihovny *pthreads*.

### 12.1.1 Funkce a datové typy

Pro práci s vlákny budeme mít datový typ `uthread_t` reprezentující vlákno a sadu čtyř funkcí.

```
/** vytvori nove vlakno, rozhrani kopiruje pthreads */
uthread_t uthread_create(void * (*thr_proc)(void *), int attributes, void *arg);
/** prepne aktualni vlakno a zacne provadet jine */
void uthread_yield();
/** pocka na dobehnuti vlakna thread a pres argument result vrati navratovou hodnotu */
void uthread_join(uthread_t thread, void **result);
/** spusti planovac vlaken */
void uthread_start_scheduler();
```

Funkce `uthread_create` vytvoří nové vlákno, které je specifikované ukazatelem na funkci typu `void * (*f)(void *)`, stejně jako je tomu v případě knihovny pthreads. Při spuštění vlákna je této funkci předán argument `arg`. U vlákna je možné nastavit jeden ze dvou atributů:

- `UTHREAD_JOINABLE`, který značí, že jiné vlákno bude čekat na jeho dokončení, viz funkce `uthread_join`. Prostředky, které jsou pro toto vlákno alokovány, budou uvolněny společně s ukončením funkce `uthread_join`.
- `UTHREAD_DETACH`, který značí, že prostředky vlákna jsou uvolněny ihned po jeho skončení a žádné jiné vlákno nebude čekat na jeho dokončení.

Funkce `uthread_yield` se postará o přepnutí aktuálního vlákna na jiné. Narozdíl od knihovny pthreads, tato funkce je naprosto zásadní, protože naše implementace vláken používá kooperativní multitasking a bez ní by nedocházelo k přepínání vláken.

Funkce `uthread_join` zajistí, že aktuální vlákno čeká na dokončení zadaného vlákna, a pokud je jí předán ukazatel na místo v paměti, převezme návratovou hodnotu vlákna a uloží ji na místo dané tímto ukazatelem.

Poslední funkcí, kterou budeme potřebovat pro práci s vlákny, je funkce `uthread_start_scheduler`, která aktivuje plánovač a s ním přepínání námi vytvořených vláken. Před zavoláním této funkce je nutné vytvořit alespoň jedno vlákno, které by mohlo běžet. Funkce `uthread_start_scheduler` skončí (a pokračuje v provádění kódu) v momentě, kdy všechna námi vytvořená vlákna doběhnou do konce.

### 12.1.2 Příklad použití

Následující kód demonstruje vytvoření vláken a spuštění plánovače voláním funkce `uthread_start_scheduler()`.

```
uthread_t thr1 = uthread_create(&foo_thread, UTHREAD_DETACHED, INT_TO_PTR(1));
uthread_t thr2 = uthread_create(&foo_thread, UTHREAD_DETACHED, INT_TO_PTR(42));

printf("spustim vlakna ...\n");
uthread_start_scheduler();
printf("pokracuje se jiz bez vlaken\n");
```

Kód funkce s vláknem se přilíš neliší od kódu, který bychom použili s knihovnou pthreads.

```
void *foo_thread(void *arg) {
        printf("Spusteno vlakno A\n");
        int step = (long) arg;
        for (int i = 0; i < 10; i++) {
                printf("A:%i\n", i * step);
                uthread_yield();
        }
        return 0;
}
```

Avšak je nutné na vhodných místech umožnit běh jiných vláken pomocí volání funkce `uthread_yield()`.

## 12.2 Implementace

Při implementaci vláken v uživatelském prostoru si musíme uvědomit jednu důležitou věc. Kdykoliv dojde k zavolání funkce `uthread_yield()`, musíme si pro aktuálně běžící vlákno uložit adresu, kde se právě v kódu nachází (registr `rip`) a obsah registrů, které kód vlákna používá. To je nutné proto, abychom se mohli přepnout do jiného vlákna, a po čase se vrátit do vlákna a pokračovat kódem za voláním funkce `uthread_yield()`. Tím, že implementujeme vlákna v uživatelském prostoru, se nám celý problém výrazným způsobem zjednodušuje. Jednak při volání funkce `uthread_yield` dojde k provedení instrukce `call uthread_yield`, která uloží obsah registru `rip` na zásobník, a tím se nám uložení aktuálního místa v programu vyřeší z části zcela přirozeně. A dále nemusíme ukládat obsah všech registrů, protože konvence vyžaduje, aby po návratu z funkce `uthread_yield` zůstal zachován jen obsah callee-saved registrů, tj. `rsp`, `rbp`, `rbx`, `r12`, ..., `15`.[1]

Aby přepínání mohlo fungovat, je nutné ještě zajistit, že každé vlákno má svůj vlastní zásobník. Nastavení zásobníku a uložení, resp. načtení, obsahu registrů, nelze vyřešit přímo z jazyka C, ale budeme si muset pomoci krátkým kódem v assembleru.

### 12.2.1 Datové struktury

Nyní si představíme hlavní datové struktury, se kterými budeme pracovat.[2]

#### Thread control block (TCB)

Nejdůležitější datovou strukturou, se kterou budeme pracovat je `struct uthread_tcb`,[3] která obsahuje všechny informace o vláknu, tzv. *thread control block (TCB)*:

```
struct uthread_tcb {
        // kontext vlakna
        uint64_t rip;
        uint64_t rsp;

        // callee-saved registry
        uint64_t rbp;
        uint64_t rbx;
        uint64_t r12;
        uint64_t r13;
        uint64_t r14;
        uint64_t r15;

        // servisni informace
```

[1] Pokud by se jednalo o preemptivní přepínání, adresa aktuálního místa v programu by byla uložena při přerušení a bylo by nutné uložit obsah všech registrů.

[2] Vedle nich budou v kódu použity i další a jednodušší datové typy, jejichž význam by měl být zřejmý přímo z kódu, a proto se nebudeme věnovat jejich popisu.

[3] Datový typ `uthread_t` je aliasem této struktury.

```
        void *(*thread_proc)(void *arg); // funkce vykonavana vlaknem
        void *stack;                     // zacatek zasobniku
        void *arg;                       // predany argument
        void *result;                    // vysledna hodnota
        uint32_t id;                     // identifikator vlakna
        uint8_t attrs;                   // vlastnosti vlakna
        enum uthread_status status;      // stav vlakna

        // slouzi k umisteni vlakna do oboustranneho spojoveho seznamu (napr. fronty)
        struct uthread_tcb *prev;
        struct uthread_tcb *next;

        // ukazatel na vlakno, ktere ceka na dokonceni tohoto vlakna
        struct uthread_tcb *blocked_thread;

        // informace pro planovac
        uint64_t runtime;                // cas, po ktery vlakno bezelo
};
```

Informace v této struktuře se dají rozdělit do několika skupin:

1. obsah registrů (kontext vlákna),
2. servisní informace obsahující informace o vlastnostech vlákna (id, status, atributy, funkce vlákna, argument, návratová hodnota),
3. vlákno čekající ve funkci `uthread_join`,
4. informace pro plánovač,
5. pomocné atributy (`prev` a `next`) sloužící k uložení vlákna do fronty, která je realizována jako oboustranný spojový seznam.

S datovým typem `struct uthread_tcb` se setkáme hned na několika místech. Jednak slouží k uložení informací o jednotlivých vláknech. Dále v globální proměnné `uthread_active_tcb` si budeme držet ukazatel na TCB aktuálně běžícího vlákna a vedle toho nám tato struktura budou sloužit k vytvoření front čekajících vláken, ať už připravených k běhu, tak čekajících na synchronizaci s jinými vlákny.

#### Fronty vláken

Pro realizaci front čekajících vláken budeme používat oboustranný spojový seznam reprezentovaný typem:

```
struct uthread_queue {
        struct uthread_tcb *head;
        struct uthread_tcb *tail;
};
```

Všimněme si, že tato struktura obsahuje jen ukazatel na začátek a konec fronty. Seřazení vláken se děje přímo v typu `struct uthread_tcb` pomocí atributů (`prev` a `next`). Toto řešení není omezující, protože předpokládáme, že každé vlákno se může nacházat nanejvýš v jedné frontě. Funkce, pro práci s frontou jsou definovány v souborech `uthreads-util.[ch]`, a protože jejich význam by měl zřejmý, nebudeme je podrobně popisovat.

### 12.2.2 Plánovač

Nad frontou vláken si postavíme jednoduchý plánovač typu round-robin.[4] Tento plánovač bude používat globální proměnnou `struct uthread_queue queue` a budou v něm uložena vlákna připravená k běhu. S plánovačem se pracuje pomocí tří obligátních funkcí: (i) pro inicializaci, (ii) pro zařazení vlákna do fronty a (iii) pro vybrání vlákna z fronty.

```
void uthread_scheduler_init();
void uthread_scheduler_enqueue(struct uthread_tcb *thread);
struct uthread_tcb *uthread_scheduler_dequeue();
```

### 12.2.3 Uložení a načtení kontextu

Při přepínání vláken je klíčové uložit a obnovit kontext prováděného vlákna, tj. obsah registrů. V tomto místě si pomůžeme dvěma funkcemi v assembleru. Mírně jednodušší je obnovit kontext z TCB a začít provádět program od zadaného místa, k tomu slouží funkce `void uthread_run()`:

```
thread_run:
    mov rdx, [uthread_active_tcb] ; ziska adresu TCB
    mov rsp, [rdx + 8]            ; obnovi obsah registru
    mov rbp, [rdx + 16]
    mov rbx, [rdx + 24]
    mov r12, [rdx + 32]
    mov r13, [rdx + 40]
    mov r14, [rdx + 48]
    mov r15, [rdx + 56]
    jmp [rdx]                     ; skoci na adresu uthread_active_tcb->rip
```

Uložení kontextu provedeme ve velice podobném duchu (funkcí `uthread_internal_yield`), avšak nejdříve odeberem hodnotu ze zásobníku (ta obsahuje návratovou adresu z funkce `uthread_yield`) a uložíme ji do `uthread_active_tcb->rip`. Dále uložíme obsah všech registrů a provedeme skok do funkce, která se postará o přepnutí do jiného vlákna. Tou je funkce `uthread_switch`.

```
uthread_internal_yield:
    pop rax ; nacte navratovou adresu z/do vlakna
    mov rdx, [uthread_active_tcb]
    mov [rdx], rax
```

[4] V principu je možné vytvořit si libovolný plánovač, který bude mít stejné rozhraní jako náš ukázkový plánovač.

```
    mov [rdx + 8], rsp
    mov [rdx + 16], rbp
    mov [rdx + 24], rbx
    mov [rdx + 32], r12
    mov [rdx + 40], r13
    mov [rdx + 48], r14
    mov [rdx + 56], r15
    jmp uthread_switch
```

Funkce `uthread_internal_yield` a `uthread_switch` mají jeden argument, který indikuje, zda se má vlákno po přepnutí zařadit mezi ostatní vlákna, nebo jestli je blokováno a čeká na nějakou událost. Protože tento argument je předán funkci `uthread_internal_yield` v registru `rdi`, je automaticky předán i funkci `uthread_switch` při provedení instrukce `jmp`.

### 12.2.4 Přepnutí vláken

Mechanismus přepínání vláken je řešen funkcí `void uthread_switch(int block_current_thread)`. Pokud existuje vlákno, které je právě přepínáno,[5] tak na základě `block_current_thread` zařazeno plánovačem mezi vlákna čekající na provedení, nebo je zablokováno. Následně je plánovačem vybráno další vlákno, to je nastaveno do `uthread_active_tcb` a jeho provední je aktivováno zavoláním `uthread_run()`. Zjednodušenou variantu této funkce ukazuje následující kód.[6]

```
void uthread_switch(int block_current_thread) {
        // pokud existuje aktivni vlakno, zaradime jej do fronty
        if (uthread_active_tcb) {
                // zmena stavu a zarazeni do fronty
                if (!block_current_thread) {
                        uthread_active_tcb->status = UT_READY;
                        uthread_scheduler_enqueue(uthread_active_tcb);
                } else {
                        uthread_active_tcb->status = UT_BLOCKED;
                        blocked++;
                }
        }
        // volba a aktivace noveho vlakna
        uthread_active_tcb = uthread_scheduler_dequeue();
        uthread_run();
}
```

Nad těmito funkcemi si vytvoříme jednoduché rozhraní, které se stará o přepínání vláken mezi stavy *ready* a *blocked*.

[5] V případě, že nějaké vlákno dokončení svou činnost, tak je hodnota `uthread_active_tcb` nastavena na `NULL`.
[6] Reálný kód navíc řeší počítání procesorového času nebo ukončení plánovače.

```
/** prepne aktualni vlakno (prepnuti READY -> READY) */
void uthread_yield() {
        uthread_internal_yield(0);
}
/** zablokuje aktualni vlakno a da provadet dalsi (prepnuti READY -> BLOCKED) */
static inline void uthread_block() {
        uthread_internal_yield(1);
}
/** prepne vlakno ze stavu BLOCKED do stavu READY */
static inline void uthread_wakeup(struct uthread_tcb *thread) {
        thread->status = UT_READY;
        blocked--;
        uthread_scheduler_enqueue(thread);
}
```

### 12.2.5 Vytvoření a zrušení vlákna

Při vytvoření vlákna je nutné primárně zajistit inicializaci struktury `uthread_tcb` a alokaci zásobníku pro vlákno. Protože je zásobník úsek paměti jako každý jiný, můžeme tak učinit funkcí `malloc`. Následující kód nastiňuje, jak je vlákno vytvořeno.

```
uthread_t uthread_create(void * (*thr_proc)(void *), int attributes, void *arg) {
    struct uthread_tcb *tcb   = malloc(sizeof(struct uthread_tcb));
    // vytvori zasobnik pro nove vytvorene vlakno
    unsigned char      *stack = malloc(UTHREAD_STACK_SIZE);
    // nevolame primo funkci, ale obalovou funkci, ktera resi uvolneni
    // prostredku a synchronizaci s ostatnimi vlakny
    tcb->rip            = (uint64_t) uthread_wrapper;
    tcb->rsp            = (uint64_t) (stack + UTHREAD_STACK_SIZE);
                                    // zasobnik roste od vyssich adres
    tcb->stack          = stack;
    tcb->id             = threads_total++;
    tcb->arg            = arg;
    tcb->thread_proc    = thr_proc;
    tcb->attrs          = attributes;
    tcb->blocked_thread = NULL;
    tcb->runtime        = 0;
    tcb->status         = UT_NEW;
    uthread_scheduler_enqueue(tcb);
    return tcb;
}
```

Nejdříve jsou nastaveny vlastnosti vlákna a následně je vlákno zařazeno mezi procesy připravené k běhu. Ale protože při skončení vlákna musíme buď uvolnit prostředky s ním spojené (např. zásobník) nebo

zajistit synchronizaci s jiným vláknem pomocí funkce `uthread_join`, nemůžeme nastavit do registru `rip` přímo adresu funkce vlákna, ale musíme zavolat obalovou funkci, která tyto úkony obstará za nás.

Tato funkce vypadá následovně:

```
static void uthread_wrapper() {
        // spusti kod vlakna se zadanym argumentem
        void *result = uthread_active_tcb->thread_proc(uthread_active_tcb->arg);

        // resi uvolneni prostredku vlakna
        if (uthread_active_tcb->attrs & UTHREAD_DETACHED) {
                // uvolnime prostredky okamzite, jine vlakno neceka
                uthread_dispose(uthread_active_tcb);
        } else {
                // ulozime vysledek, a pokud existuje vlakno, ktere ceka na dokonceni
                // tohoto vlakna, probudime jej
                uthread_active_tcb->result = result;
                uthread_active_tcb->status = UT_TERMINATED;
                if (uthread_active_tcb->blocked_thread) {
                        uthread_wakeup(uthread_active_tcb->blocked_thread);
                }
        }
        uthread_active_tcb = NULL;
        uthread_switch(0);
}
```

Nejdříve spustíme kód vlákna zavoláním funkce s tím, že do funkce předáme argument, se kterým má být vlákno spuštěno. Adresa volané funkce i argument jsou uloženy v TCB.

Po skončení funkce, která reprezentuje vlákno, získáme návratovou hodnotu. A následně, podle nastaveného atributu, naložíme s vláknem.

Pokud vlákno bylo vytvořeno jako `UTHREAD_DETACHED`, uvolníme všechny jeho prostředky okamžitě. Funkce `uthread_dispose` uvolní veškerou paměť, která byla pro vlákno alokována, např. zásobník.

V případě, že vlákno bylo vytvořeno s atributem `UTHREAD_JOINABLE`, uložíme návratovou hodnotu do TCB a vláknu nastavíme stav *terminated*. Pokud nějaké vlákno zavolalo funkci `uthread_join` a čeká na právě skončené vlákno (víme to z atributu `uthread_active_tcb->blocked_thread`), tak jej probudíme, přesenume jej ze stavu blocked do stavu ready. V každém případě voláme funkci `uthread_switch`, která přepne na další vlákno.

U vláken, které jsou typu `UTHREAD_JOINABLE`, se mimo jiné o uvolnění prostředků stará funkce `uthread_join`. U této funkce mohou nastat dva stavy. Buď vlákno, na které má počkat ještě nedoběhlo. V takovém případě dojde k uspání aktuálního vlákna, a bychom po skončení vlákna věděli, které vlákno se má probudit, uložíme si tuto informace do TCB jako atribut `blocked_thread`. Pokud vlákno, na které se má počkat skončilo, uložíme návratovou hodnotu a uvolníme přidělené prostředky, jak ukazuje následující kód.

```
void uthread_join(uthread_t thread, void **result) {
```

```
        // pokud vlakno jeste nedobehlo, uspime aktualni vlakno
        if (thread->status != UT_TERMINATED) {
                thread->blocked_thread = uthread_active_tcb;
                uthread_block();
        }
        // vratime hodnotu a uvolnime prostredky
        if (result) {
                *result = thread->result;
        }
        uthread_dispose(thread);
}
```

### 12.2.6 Aktivace a deaktivace prostředí

Poslední záležitost, kterou musíme ošetřit, je spuštění a ukončení plánovače, přičemž chceme, aby po skončení běhu všech vláken program pokračoval standardním způsobem. Abychom toho mohli docílit, potřebujeme si uložit stav registrů před spuštěním plánovače. Ten pak obnovíme po skončení všech vláken. Zde si opět pomůžeme krátkým kódem v assembleru.

```
global uthread_start_scheduler
global uthread_mainthread_context
uthread_start_scheduler:
    ; ulozi kontext volajici funkce
    mov rdx, uthread_mainthread_context
    mov [rdx + 8], rsp
    mov [rdx + 16], rbp
    mov [rdx + 24], rbx
    mov [rdx + 32], r12
    mov [rdx + 40], r13
    mov [rdx + 48], r14
    mov [rdx + 56], r15

    ; do uthread_mainthread_context->rip  ulozi adresu kodu, ktery funkci ukonci
    mov qword [rdx], uthreads_complete

    mov qword [uthread_active_tcb], 0 ; na zacatku neni aktivni zadne vlakno
    mov rdi, 0                        ; neblokujeme vlakno
    jmp uthread_switch                ; spustime planovac

uthreads_complete:
    ret

section .bss
```

```
uthread_mainthread_context:
   resq 8 ; pamet nutna pro ulozeni kontextu, s ostatnimi hodnotami nepracujeme
```

Vedle samotné funkce pro spuštění plánovače si vytvoříme globální proměnnou `uthread_mainthread_context`, která je typu `struct uthread_tcb`. Do ní uložíme kontext vlákna, které zavolalo funkci `uthread_start_scheduler`. V momentě, kdy všechna námi vytvořená vlákna skončí, chceme, aby se provedl návrat z funkce `uthread_start_scheduler()`, proto do atributu `rip` uložíme adresu instrukce `ret`.

Dále aktivujeme plánovač, a to tak, že nastavíme, že není aktivní žádné vlákno, a provedeme skok do funkce `uthread_switch`, která vybere první vlákno určené k běhu. Následně se mezi sebou jednotlivá vlákna střídají ve využívání procesoru.

Pokud není k dispozici vlákno, které by mohlo být vykonávano, nastaví se jako aktivní vlákno ukazatel na `uthread_mainthread_context` a je zavolána funkce `uthread_run()`. Díky ní je obnoven obsah registrů funkce, která volala `uthread_start_scheduler`, skočí se na instrukci `ret`, a program může dál pokračovat.

## 12.3 Synchronizace

V případě kooperativního multitaskingu se můžeme chybám souběhu (race condition) snadno vyvarovat, protože můžeme vložit přepnutí na vhodná místa a nemusíme uvažovat, že může dojít k přepnutí vláken v libovolném bodě. Přesto synchronizační prostředky dávají i v tomto prostředí smysl. Ukážeme si proto, jakým způsobem se dají implementovat semafory s pasivním čekáním. Tato implementace je přímočará a vychází z funkcí, které jsme si představili v předchozí kapitole.

Základem je struktura, která obsahuje hodnotu semaforu a seznam vláken, které na tomto semaforu čekají:

```
struct uthread_sem {
        // hodnota semaforu, pokud je hodnota < 0, znamena to pocet cekajicich vlaken
        int value;
        // fronta vlaken cekajicich na semaforu
        struct uthread_queue blocked_threads;
};
```

V následujícím kódu můžeme vidět funkce pro inicializaci, snížení a zvýšení hodnoty semaforu. Naše řešení umožňuje, aby hodnoty šly do záporných čísel. To nám signalizuje, kolik vláken na daném semaforu čeká.

```
/** inicializuje semafor */
void uthread_sem_init(uthread_sem_t *sem, int value) {
        sem->value = value;
        uthread_queue_init(&sem->blocked_threads);
}
/** snizi hodnotu semaforu o jedna, a pokud je uz na nule, tak ceka */
void uthread_sem_wait(uthread_sem_t *sem) {
```

```
        sem->value--;
        if (sem->value < 0) {
                uthread_queue_put(&sem->blocked_threads, uthread_active_tcb);
                uthread_block();
        }
}
/** zvysi hodnotu semaforu o jedna, a pokud nejake vlakno na nej ceka, probudi jej  */
void uthread_sem_post(uthread_sem_t *sem) {
        if (sem->value < 0) {
                struct uthread_tcb *blocked = uthread_queue_poll(&sem->blocked_threads);
                uthread_wakeup(blocked);
        }
        sem->value++;
}
```

Aby semafory mohly fungovat jako synchronizační nástroj, musí být jeho operace prováděny atomicky. To je v případě kooperativního multitaskingu přirozeně splněno, pokud bychom měli preemptivní multitasking, museli bychom operace semaforu ještě obalit zámky, nejspíše spinlocky.

## Úkoly

1. Doplňte funkci `void uthread_set_priority(int)`, která aktuálnímu vláknu nastaví zadanou prioritu.
2. Upravte plánovač tak, aby fungoval jako Completely Fair Scheduler z Linuxového jádra. Uspořádejte vlákna podle toho, kolik dostaly procesorového času a vždy vyberte to, co dostalo nejméně. Aby bylo implementace jednodušší, nemusíte používat červeno-černý strom pro evidenci vláken. Postačí libovolná datavá strukturu (např. pole), kde budou vlákna seřazena. Do řešení zahrňte prioritu vlákna.
3. (Volitelně) Implementujte *lottery scheduler*, který je postavený na tom, že každé vlákno bude mít přiděleno určité množství losů, ze kterých bude plánovač vybírat.
4. Vyzkoušejte si prakticky, jak jsou vlákna přidělována různými plánovači.