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 řetězce, 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.

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.

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.

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

[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.

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]

1.2.1 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 bez jasného významu máme v kódu pojmenovanou hodnotu, u které jsme schopni určit na základě jména její 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]

1.2.2 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

[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`.

`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
```

1.2.3 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
```

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) řetězec, 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 řetězce (tj. 13) přiřadíme do registru `rdx` a zbývá se vypořádat s adresou resp. uložením vypisovaného řetězce.

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 pseudoinstrukce `db`. Pseudoinstrukce `db` umožňuje definovat a alokovat místo pro jednobytové hodnoty, případně řetězce, jak lze vidět v našem příkladu. Alternativně lze pomocí pseudoinstrukcí `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.

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 pseudoinstrukce `resb n`, která vyhradí úsek paměti o velikosti `n` bytů. Analogicky máme pseudoinstrukce `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`.

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 20×5.
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.