



Vyhledávací formulář

Použití assembleru v Linuxu

Napsal Pavel Tišnovský

100		<i>;-----</i>
101		<i>; zstr_count:</i>
102		<i>; Counts a zero-terminated ASCII string to determine its size</i>
103		<i>; in: eax = start address of the zero terminated string</i>
104		<i>; out: ecx = count = the length of the string</i>
105		
106		<i>zstr_count: ; Entry point</i>
107	00000030 B9FFFFFF	<i> mov ecx, -1 ; Init the loop counter, pre-decrement</i>
108		<i> ; to compensate for the increment</i>
109		<i>.loop:</i>
110	00000035 41	<i> inc ecx ; Add 1 to the loop counter</i>
111	00000036 803C0800	<i> cmp byte [eax + ecx], 0 ; Compare the value at the string's</i>
112		<i> ; [starting memory address Plus the</i>
113		<i> ; loop offset], to zero</i>
114	0000003A 75F9	<i> jne .loop ; If the memory value is not zero,</i>
115		<i> ; then jump to the label called '.loop',</i>
116		<i> ; otherwise continue to the next line</i>
117		<i>.done:</i>
118		<i> ; We don't do a final increment,</i>
119		<i> ; because even though the count is base 1,</i>
120		<i> ; we do not include the zero terminator in the</i>

V dnešním článku se budeme zabývat v současnosti již možná poněkud okrajovým, ale stále zajímavým a mnohdy i užitečným tématem. Jedná se o tvorbu programů popř. jejich částí s využitím assembleru neboli jazyka symbolických adres (JSA). Zaměříme se na použití assembleru jak na platformě x86_64, tak i (a to možná především) na 32bitové platformě ARM.

Obsah

1. Použití assembleru v Linuxu
2. Vznik jazyka „assembly language“ a nástroje nazvaného assembler
3. Assemblery na domácích osmibitových mikropočítačích i na počítačích s procesory Motorola 68000
4. Assemblery v Linuxu
5. GNU Assembler
6. Netwide Assembler (NASM)
7. Volání funkcí kernelu – syscalls
8. Kostra jednoduché aplikace naprogramovaná v GNU Assembleru
9. Kostra jednoduché aplikace naprogramovaná v Netwide Assembleru
10. Varianta pro 32bitové ARMy s instrukční sadou Thumb
11. Repositář se zdrojovými kódy demonstračních příkladů
12. Odkazy na Internetu

1. Použití assembleru v Linuxu

V dnešním článku se seznámíme se základy práce s assemblerem v operačním systému Linux. Assembler neboli též *jazyk symbolických adres (JSA)* popř. alternativně *jazyk symbolických instrukcí (JSI)* je nízkoúrovňovým programovacím jazykem, který na hierarchii jazyků stojí nad strojovým kódem, ovšem hluboko pod vyššími kompilovanými programovacími jazyky typu C, D či C++. Typickou vlastností assembleru je jeho vazba na určitý typ procesoru popř. řadu procesorů (architekturu) – týká se to především sady dostupných instrukcí. Programy se ve většině typech assemblerů zapisují formou symbolických

instrukcí, přičemž každá instrukce je představována svou mnemotechnickou zkratkou a případnými operandy (konstantami, adresami, nepřímými adresami, jmény pracovních registrů procesoru atd.). Z několika assemblerů, které jsou pro Linux dostupné, se zaměříme na *GNU Assembler* a taktéž na novější *Netwide Assembler*, který však v současnosti nepodporuje všechny používané architektury.

```
7F80 20 00 44 55 4D 50 20 37 ..... DUMP 7
7F88 46 38 30 32 30 20 30 30 ..... F8020 00
7F90 20 34 44 20 34 35 20 34 ..... 4D 45 4
7F98 44 20 32 30 20 33 37 20 ..... D 20 37
7FA0 34 36 20 33 38 20 33 30 ..... 46 38 30
7FA8 20 32 30 20 33 30 20 33 ..... 20 30 3
7FB0 30 20 32 30 20 33 30 20 ..... 0 20 30
7FB8 33 30 0D 00 00 00 00 00 ..... 30.....
7FC0 00 00 00 00 00 00 00 00 .....
7FC8 00 00 00 00 00 00 00 00 .....
7FD0 00 00 00 00 00 00 00 00 .....
7FD8 00 00 00 00 00 00 00 00 .....
7FE0 00 00 00 80 86 C0 87 00 .....
7FE8 86 07 FB 0D FB 8B 7F 22 ..... 0.8.8.
7FF0 85 08 08 F2 7F 8B 7F 28 ..... 0.8.
7FF8 82 F8 7F 08 08 16 80 00 ..... I.....
8000 C3 03 80 3E 8A D3 F7 31 ..... >...1
8008 FF 7F CD 76 82 CD A3 82 ..... 0.8.
8010 2A 30 C0 2B 36 00 31 FF ..... *0.+6.1.
8018 7F CD EE 8B 2A 78 C0 22 ..... 0.8.
8020 72 C0 21 00 83 22 70 C0 ..... 0.8.
8028 21 16 80 E5 21 5F 80 22 ..... 0.8.
8030 74 C0 2A 70 C0 EB 2A 72 ..... 0.8.

** NO COMMAND **
```

Obrázek 1: Díky použití assembleru není nutné, aby programátoři pracovali ručně přímo se strojovým kódem (machine language). Pokud je to přeci jen z nějakého důvodu vyžadováno (mikrořadiče atd.), lze pro tento účel využít nástroje nazvané „monitory“.

Programování v jazyku symbolických adres již v současnosti není nijak masivní záležitostí, a to především z toho důvodu, že tvorba aplikací ve vyšších programovacích jazycích je v porovnání s assemblerem mnohem rychlejší, aplikace jsou snáze přenositelné na jiné platformy a změna aplikací, tj. přidávání nových vlastností či refaktoring, je ve vyšších programovacích

jazycích jednodušší. Nesmíme taktéž zapomenout na to, že díky vývoji překladačů vyšších programovacích jazyků se běžně stává, že například algoritmus naprogramovaný v jazyku C může co do rychlosti snadno soutěžit s programem napsaným průměrným programátorem v assembleru. I přesto si však myslím, že assembler stále má své nezastupitelné místo, a to jak při zkoumání systémových volání v Linuxu a programování speciálního SW (části ovladačů, multimediální kodeky, některé kritické algoritmy typu FFT), tak i při práci na dnes velmi oblíbených osmibitových čipech tvořících například srdce Arduina a podobných jednodeskových mikropočítačů. Z tohoto důvodu se dnes seznámíme se způsobem tvorby jednoduchých aplikací v Linuxu, a to jak na platformě x86_64, tak i na platformě ARM. Podrobnějším popisem jednotlivých instrukcí se budeme zabývat až příště; dnes nás bude zajímat především toolchain a nástroje, které jsou v něm obsažené.

```
0310 CIOV = $E456 ;CIO ENTRY VECTOR
0320 RUNAD = $02E0 ;RUN ADDRESS
0330 EOL = $9B ;END OF LINE
0340 ;
0350 ; SETUP FOR CIO
0360 ; -----
0370 *= $0600
0380 START LDX #0 ;IOCB 0
0390 LDA #PUTREC ;WANT OUTPUT
0400 STA ICCOM,X ;ISSUE CMD
0410 LDA #MSG&255 ;LOW BYTE OF MSG
0420 STA ICBAL,X ; INTO ICBAL
0430 LDA #MSG/256 ;HIGH BYTE
0440 STA ICBAH,X ; INTO ICBAH
0450 LDA #0 ;LENGTH OF MSG
0460 STA ICBLH,X ; HIGH BYTE
0470 LDA #$FF ;255 CHAR LENGTH
0480 STA ICBLL,X ; LOW BYTE
```

Obrázek 2: Nestrukturovaný zdrojový kód psaný v assembleru.

2. Vznik jazyka „assemb ly language“ a nástroje nazvaného assembler

Assembly za sebou mají velmi dlouhý vývoj, protože první nástroje, které se začaly tímto názvem označovat, vznikly již v padesátých letech minulého století, a to na *mainframech* vyráběných společnostmi IBM i jejími konkurenty (UNIVAC, Burroughs, Honeywell, General Electric atd.). Před vznikem skutečných assemblerů byla situace poněkud složitá. První aplikace pro mainframy totiž byly programovány přímo ve strojovém kódu, který bylo možné přímo zadávat z takzvaného *řídícího panelu (control panel)* počítače či načítat z externích paměťových médií (děrných štítků, magnetických pásek atd.). Ovšem zapisovat programy přímo ve strojovém kódu je zdlouhavé, vedoucí k častým chybám a pro větší aplikace z mnoha důvodů nepraktické, o čemž se mohli relativně nedávno přesvědčit například i studenti programující na československém mikropočítači *PMI-80*. Z důvodu usnadnění práce programátorů tedy vznikly první utility, jejichž úkolem bylo transformovat programy zapsané s využitím symbolických jmen strojových instrukcí do (binárního) strojového kódu určeného pro konkrétní typ počítače a jeho procesoru.

```

ldx #$00
lda #0
ClearRAM
sta 0,x
inx
bne ClearRAM

; Initial map rendering
jsr update_map_screen
jsr blit_player

; Main game loop
main_loop:
; Debug, load location info into registers
lda player_map_x
asl
asl
asl
asl
ora player_screen_x
tax
lda player_map_y
asl
asl
asl
asl
ora player_screen_y
tay

; Wait for input
input_loop:
lda keypress
beq input_loop
ldx #0
stx keypress
; Act on input (key code in A)
cmp #97 ; Move left
bne test_input_1
jmp input_do_move_left
test_input_1:
cmp #104 ; Move left
bne test_input_2
jmp input_do_move_left
test_input_2:
cmp #100 ; Move right

```

Obrázek 3: Kód v assembleru je však možné psát i strukturovaně, používat subrutiny a funkce atd.

Těmto programům, jejichž možnosti se postupně vylepšovaly (například do nich přibyla podpora textových maker, řízení víceprůchodového překladu, vytváření výstupních sestav s překládanými symboly, později i skutečné linkování s knihovnami atd.), se začalo říkat *assemblery* a jazyku pro symbolický zápis programů pak *jazyk symbolických instrukcí* či *jazyk symbolických adres* – *assembly language* (někdy též zkráceně nazývaný *assembler*, takže toto slovo má vlastně dodnes oba dva významy). Jednalo se o svým způsobem převratnou myšlenku: sám počítač byl použit pro tvorbu programů, čímž odpadla namáhavá práce s tužkou a papírem. Posléze se zjistilo, že i programování přímo v assembleru je většinou pracné a zdoluhavé, takže se na mainframech začaly používat různé vyšší programovací jazyky, zejména FORTRAN a COBOL. Použití vyšších programovacích jazyků bylo umožněno relativně vysokým výpočetním výkonem mainframů i (opět relativně) velkou

kapacitou operační paměti; naopak se díky vyšším programovacím jazykům mohly aplikace přenášet na různé typy počítačů, což je nesporná výhoda.

```
z      = $f8
buffer = $0400
      *= $4000

setlfs = $ffba
setnam = $ffbd
open   = $ffc0
chkout = $ffc9
chrout = $ffd2
clrchn = $fcc6
chkin  = $ffc6
close  = $ffc3

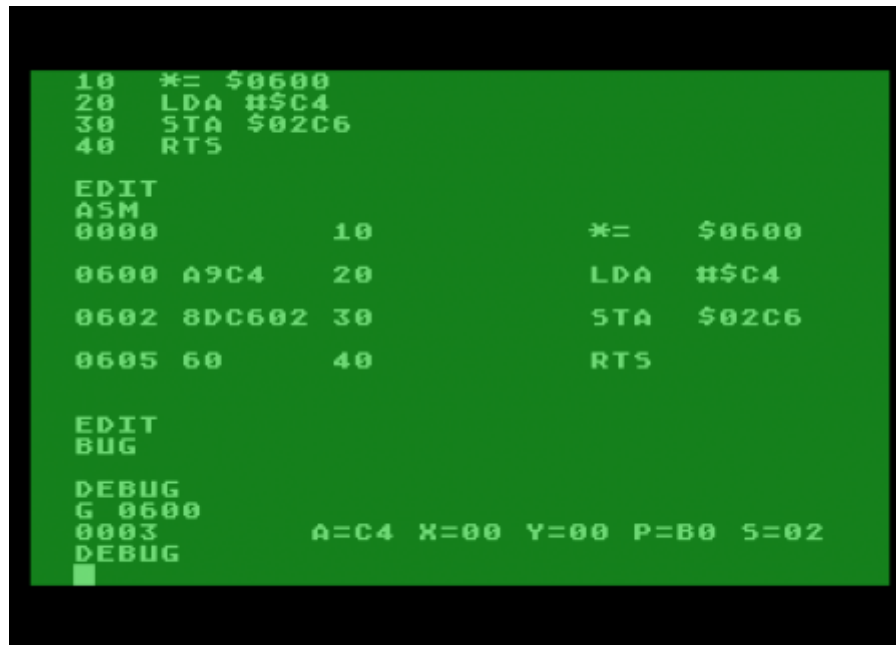
load   lda $d011
      pha
      lda #0
      sta $d011
      jsr openfiles
      lda #"i"
      sta com1+8
turbo-ass. v4.1+ - ttfl571/weird science
x:9 line:1 bot:8b49 insert: line
```

Obrázek 4: Assembler pro počítače Commodore C64.

3. Assemblery na domácích osmibitových mikropočítačích i na počítačích s procesory Motorola 68000

Oživení zájmu o programování v assembleru přinesl vznik minipočítačů (například známé řady *PDP*) a na konci sedmdesátých let minulého století pak zcela nového fenoménu, který nakonec přepsal celé dějiny výpočetní techniky – domácích osmibitových mikropočítačů. Na osmibitových domácích mikropočítačích se používaly dva typy *assemblerů*. Prvním typem byly assembly interaktivní, které uživateli nabízely poměrně komfortní vývojové prostředí, v němž bylo možné zapisovat jednotlivé instrukce v symbolické podobě, spouštět programy, krokovat je, vypisovat obsahy pracovních registrů mikroprocesoru atd. Výhodou byla nezávislost těchto assemblerů na rychlém externím paměťovém médiu

(například disketové jednotce), který mnoho uživatelů a programátorů ani nevlastnilo. Druhý typ assemblerů je používán dodnes – jedná se vlastně o běžné překladače, kterým se na vstupu předloží zdrojový kód (uložený na kazetě či disketě) a po překladu se výsledný nativní kód taktéž uloží na paměťové médium (odkud ho lze následně spustit). Tyto assembly byly mnohdy vybaveny více či méně dokonalým systémem maker (odtud ostatně pochází i označení *macroassembler*).



```
10  *= $0600
20  LDA #$C4
30  STA $02C6
40  RTS

EDIT
ASM
0000          10      *=      $0600

0600 A9C4      20      LDA    #$C4
0602 8DC602    30      STA    $02C6
0605 60        40      RTS

EDIT
BUG

DEBUG
G 0600
0003          A=C4 X=00 Y=00 P=B0 S=02
DEBUG
```

Obrázek 5: Atari Macro Assembler.

Assembly byly mezi programátory poměrně populární i na počítačích *Amiga* a *Atari ST*, a to i díky tomu, že instrukční kód mikroprocesorů *Motorola 68000* byl do značné míry ortogonální, obsahoval relativně velké množství registrů (univerzální datové registry D0 až D7 a adresové registry A0 až A7) a navíc bylo možné používat i takové adresovací režimy, které korespondovaly s konstrukcemi používanými ve vyšších programovacích jazycích (přístupy k prvkům polí, přístup k lokálním proměnným umístěným v zásobníkovém rámci, autoinkrementace adresy atd.). Podívejme se na jednoduchý příklad rutiny (originál najdete [zde](#)), která sečte všechny prvky (16bitové integery – načítá se vždy jen 16bitové slovo) v poli. V tomto příkladu se používá autoinkrementace adresy při adresování prvků polí a taktéž instrukce **DBRA** provádí dvě činnosti – snížení hodnoty registru o jedničku a skok v případě, že je výsledek nenulový:

```

    moveq #0, d0      ; potřebujeme vynulovat horních 16 bitů d0
    moveq #0, d1      ; mezivýsledek
loop:
    move.w (a0)+, d0   ; horních 16 bitů d0 je pořád nastaveno na 0
    add.l d0, d1
    dbra d2, loop      ; d2 je použit jako počítadlo

```

4. Assemblery v Linuxu

V této kapitole budeme pod termínem „assembler“ chápat programový nástroj určený pro transformaci zdrojového kódu naprogramovaného v jazyku symbolických adres do strojového kódu. Pro Linux vzniklo hned několik takových nástrojů, přičemž některé nástroje jsou komerční a jiné patří mezi open source. Z nekomerčních nástrojů, které nás samozřejmě na serveru mojefedora.cz zajímají především, se jedná o známý *GNU Assembler*, dále pak o nástroj nazvaný *Netwide assembler* (*NASM*), nástroj *Yasm Modular Assembler* či až překvapivě výkonný *vasm*. *NASM* a *Yasm* jsou pro první krůčky v assembleru velmi dobře použitelné, neboť mají dobře zpracovaný mechanismus reakce na chyby, dají se v nich psát čitelné programy atd. Určitý problém nastává v případě, kdy je nutné vyvíjet aplikace určené pro jinou architekturu, než je i386 či x86_64, a to z toho důvodu, že ani *Netwide assembler* ani *Yasm* nedokážou pracovat s odlišnou instrukční sadou. Naproti tomu *GNU Assembler* tímto problémem ani zdaleka netrpí, takže se v následujících kapitolách budeme zabývat jak nástrojem *NASM*, tak i *GNU Assemblerem*.

5. GNU Assembler

GNU Assembler (*gas*) je součástí skupiny nástrojů nazvaných *GNU Binutils*. Jedná se o nástroje určené pro vytváření a správu binárních souborů obsahujících takzvaný „objektový kód“, dále nástrojů určených pro práci s knihovnami strojových funkcí i pro profilování. Mezi *GNU Binutils* patří vedle *GNU Assembleru* i linker **ld**, profiler **gprof**, správce archivů strojových funkcí **ar**, nástroj pro odstranění symbolů z objektových a spustitelných souborů **strip** a několik pomocných utilit typu **nm**,

objdump, **size** a **strings**. *GNU Assembler* je možné použít buď pro překlad uživatelem vytvořených zdrojových kódů nebo pro zpracování kódů vygenerovaných překladači vyšších programovacích jazyků (**GCC** atd.). Zajímavé je, že všechny moderní verze *GNU Assembleru* podporují jak původní AT&T syntaxi, tak i (podle mě čitelnější) syntaxi používanou společností Intel.

6. Netwide Assembler (NASM)

Netwide Assembler (NASM) vznikl v době, kdy začali na operační systém Linux přecházet programátoři znající operační systémy DOS a (16/32bit) Windows. Tito programátoři byli většinou dobře seznámeni s možnostmi assemblerů, které se na těchto platformách používaly nejčastěji – *Turbo Assembleru (TASM)* společnosti Borland i *Microsoft Macro Assembleru (MASM)* a tak jim možnosti *GNU Assembleru* (který má své kořeny na odlišných architekturách) příliš nevyhovovaly. Výsledkem snah o vytvoření nástroje podobnému *TASMu* či *MASMu* byl právě *NASM*, který podporuje stejný způsob zápisu operandů instrukcí a navíc ještě zjednodušuje zápis těch instrukcí, u nichž je jeden operand tvořen nepřímou adresou. *NASM* byl následován projektem *Yasm* (fork+přepis), ovšem základní vlastnosti a především pak vazba na platformu i386 a x86_64 zůstaly zachovány (to mj. znamená, že například na *Raspberry Pi* možnosti těchto dvou nástrojů plně nevyužijeme, což je určitě škoda).

7. Volání funkcí kernelu – syscalls

Vzhledem k tomu, že i ta nejjednodušší aplikace naprogramovaná v assembleru musí nějakým způsobem ukončit svou činnost, je nutné buď zavolat vhodnou knihovní funkci (z **libc**) popř. použít takzvaný „syscall“. V kontextu Linuxu se pod tímto termínem skrývá volání nějaké funkce umístěné přímo v jádru operačního systému. V praxi to funguje následovně: podle požadavků konkrétní funkce se naplní pracovní registry popř. datové struktury uložené v paměti, následně se číslo služby uloží do pracovního registru **eax** (i386/x86_64) nebo do pracovního registru **r7** (32bitový ARM s použitím EABI) popř. **x8** (ARM64) a následně se zavolá nějaká instrukce, která přepne kontext procesoru do privilegovaného režimu „jádra“ (vyvolá výjimku atd.). Na procesorech s architekturou i386 či x86_64 je touto instrukcí **INT 80h**, u 32bitových ARMů s EABI je to instrukce **SWI 0h** a u ARM64 instrukce **SVC #0**:

Architektura	Číslo služby v	Instrukce pro syscall	Návratová hodnota v
i386	eax	INT 80h	eax
x86_64	rax	SYSCALL	rax
ARM 32 s EABI	r7	SWI 0h	r0
ARM 64	x8	SVC #0	x0
Motorola 68k	d0	TRAP #0	d0

Samotná čísla jednotlivých funkcí kernelu naleznete například na adrese

http://docs.cs.up.ac.za/programming/asm/derick_tut/syscalls.html. Nás bude zajímat hned první řádek této tabulky, který říká:

%eax	Name	Source	%ebx	%ecx	%edx	%esx	%edi
1	sys_exit	kernel/exit.c	int	-	-	-	-

Co to znamená? Jedná se o funkci určenou pro ukončení aplikace, přičemž číslo syscallu je rovno jedné a zapisuje se do registru **eax**. Jediným parametrem je návratová hodnota (typu int), která se zapisuje do registru **ebx** (na ARMu je to **r0**, což se však zde nedozvíme). Podobně lze používat další funkce, jak si ostatně ukážeme příště.

8. Kostra jednoduché aplikace naprogramovaná v GNU Assembleru

Podívejme se nyní na to, jak může vypadat kostra velmi jednoduché aplikace naprogramované v GNU Assembleru pro procesory řady i386 či x86_64. Celý zdrojový kód je rozdělen na řádky, přičemž na jednotlivých řádcích mohou být

komentáře, deklarace různých konstant a symbolů (**sys_exit=1**), speciální direktivy (**.section**), návěští/labels (**_start**) a samozřejmě i samotný kód reprezentovaný mnemotechnickými názvy instrukcí a jejich operandů. Důležitý je symbol **_start**, protože ten je používán i linkerem a specifikuje vstupní bod do programu:

```
# asmsyntax=as

# Sablona pro zdrojovy kod Linuxoveho programu naprogramovaneho
# v assembleru GNU AS.
#
# Autor: Pavel Tisnovsky

# Linux kernel system call table
sys_exit=1

#-----
.section .data

#-----
.section .bss

#-----
.section .text
        .global _start          # tento symbol ma byt dostupny i linkeru

_start:
        movl  $sys_exit,%eax    # cislo sycallu pro funkci "exit"
        movl  $0,%ebx           # exit code = 0
        int   $0x80             # volani Linuxoveho kernelu
```

Povšimněte si rozdělení do sekcí – sekce pojmenované **.data** a **.bss** jsou prázdné, samotný kód je umístěn do sekce pojmenované **.text**, což může být matoucí, protože ve výsledném binárním souboru tato sekce taktéž obsahuje binární data (instrukce). Instrukce jsou v programu pouze tři a slouží pro naplnění pracovních registrů **eax** a **ebx** (funkce číslo 1, návratová hodnota 0) a zavolání syscallu. Používáme zde původní AT&T syntaxi GNU Assembleru, proto se do instrukce **movl** operandy zapisují v pořadí zdroj,cíl.

Překlad (assemblerem) a následné slinkování do spustitelného souboru se provede následovně:

```
as template.s -o template.o
ld -s template.o
```

Výsledný soubor má velikost 344 bajtů:

```
00000000: 7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00 .ELF.....
00000010: 02 00 3e 00 01 00 00 00 78 00 40 00 00 00 00 00 ..>.....x.@.....
00000020: 40 00 00 00 00 00 00 00 98 00 00 00 00 00 00 00 @.....
00000030: 00 00 00 00 40 00 38 00 01 00 40 00 03 00 02 00 ....@.8...@.....
00000040: 01 00 00 00 05 00 00 00 00 00 00 00 00 00 00 00 .....
00000050: 00 00 40 00 00 00 00 00 00 00 40 00 00 00 00 00 ..@.....@.....
00000060: 84 00 00 00 00 00 00 00 84 00 00 00 00 00 00 00 .....
00000070: 00 00 20 00 00 00 00 00 b8 01 00 00 00 bb 00 00 .. .....
00000080: 00 00 cd 80 00 2e 73 68 73 74 72 74 61 62 00 2e .....shstrtab..
00000090: 74 65 78 74 00 00 00 00 00 00 00 00 00 00 00 00 text.....
000000a0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
000000b0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
000000c0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
000000d0: 00 00 00 00 00 00 00 00 0b 00 00 00 01 00 00 00 .....
000000e0: 06 00 00 00 00 00 00 00 78 00 40 00 00 00 00 00 .....x.@.....
000000f0: 78 00 00 00 00 00 00 00 0c 00 00 00 00 00 00 00 x.....
00001000: 00 00 00 00 00 00 00 00 01 00 00 00 00 00 00 00 .....
00001100: 00 00 00 00 00 00 00 00 01 00 00 00 03 00 00 00 .....
00001200: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
```

```

0000130: 84 00 00 00 00 00 00 00 11 00 00 00 00 00 00 00 .....
0000140: 00 00 00 00 00 00 00 00 01 00 00 00 00 00 00 00 .....
0000150: 00 00 00 00 00 00 00 00 .....

```

Na interní obsah souboru se můžeme podívat utilitou **objdump**, a to následujícím způsobem:

```
objdump -f -d -t -h a.out
```

```

a.out:      file format elf32-i386
architecture: i386, flags 0x00000102:
EXEC_P, D_PAGED
start address 0x08048054

```

Sections:

Idx	Name	Size	VMA	LMA	File off	Algn
0	.text	0000000c	08048054	08048054	00000054	2**2

CONTENTS, ALLOC, LOAD, READONLY, CODE

SYMBOL TABLE:

no symbols

Disassembly of section .text:

08048054 <.text>:

```

8048054:      b8 01 00 00 00      mov     $0x1,%eax
8048059:      bb 00 00 00 00      mov     $0x0,%ebx
804805e:      cd 80               int     $0x80

```

Vidíme, že uvnitř spustitelného souboru se skutečně nachází sekce nazvaná **.text**. Tato sekce je neměnitelná a obsahuje kód; její zarovnání je na celá slova. Obsahem je dvanáct bajtů obsahujících trojici instrukcí (ty jsou zde vypsány tak, jak je disassembler získal ze souboru, tj. již bez symbolických konstant atd.).

V 64bitové variantě je soubor nepatrně odlišný, ale ne příliš:

```
a.out:      file format elf64-x86-64
architecture: i386:x86-64, flags 0x00000102:
EXEC_P, D_PAGED
start address 0x0000000000400078
```

Sections:

Idx	Name	Size	VMA	LMA	File off	Algn
0	.text	0000000c	0000000000400078	0000000000400078	00000078	2**0
CONTENTS, ALLOC, LOAD, READONLY, CODE						

SYMBOL TABLE:

no symbols

Disassembly of section .text:

0000000000400078 <.text>:

400078:	b8 01 00 00 00	mov	\$0x1,%eax
40007d:	bb 00 00 00 00	mov	\$0x0,%ebx
400082:	cd 80	int	\$0x80

Poznámka: význam jednotlivých instrukcí si podrobněji popíšeme příště, dnes se seznamujeme především s použitím toolchainu.

9. Kostra jednoduché aplikace naprogramovaná v Netwide Assembleru

V *Netwide Assembleru* se stejná aplikace naprogramuje nepatrně odlišným způsobem, a to kvůli rozdílné syntaxi a sémantice. Důležité a na první pohled viditelné je otočení operandů u instrukcí (cíl, zdroj) a taktéž to, že u instrukce **mov** se

nemusí nijak specifikovat typ operandů – to je zajištěno assemblerem automaticky. Dále se odlišně zapisují symbolické konstanty s využitím direktivy **equ**:

```
; asmsyntax=nasm

; Sablona pro zdrojovy kod Linuxoveho programu naprogramovaneho
; v assembleru NASM.
;
; Autor: Pavel Tisnovsky


; Linux kernel system call table
sys_exit equ 1


;-----
section .data


;-----
section .bss


;-----
section .text
    global _start          ; tento symbol ma byt dostupny i linkeru

_start:
    mov     eax,sys_exit    ; cislo sycallu pro funkci "exit"
    mov     ebx,0           ; exit code = 0
    int     80h            ; volani Linuxoveho kernelu
```

Překlad se provede příkazem:

```
nasm -felf32 template.asm  
ld -s template.o
```

popř. pro 64bitový systém příkazem:

```
nasm -felf64 template.asm  
ld -s template.o
```

10. Varianta pro 32bitové ARMy s instrukční sadou Thumb

Stejná aplikace, ale určená pro 32bitové mikroprocesory ARM (například pro Raspberry Pi), musí být v GNU Assembleru vytvořena nepatrně odlišně, což je ostatně patrné z následujícího kódu. Povšimněte si především toho, že komentáře u instrukcí musí začínat znakem @ a samozřejmě i instrukční soubor je jiný:

```
# asmsyntax=as  
  
# Sablona pro zdrojovy kod Linuxoveho programu naprogramovaneho  
# v assembleru GNU AS.  
#  
# Autor: Pavel Tisnovsky  
  
# Linux kernel system call table  
sys_exit=1
```

```

#-----
.section .data

#-----

.section .bss

#-----

.section .text
        .global _start          @ tento symbol ma byt dostupny i z linkeru

_start:
        mov     r7,$sys_exit     @ cislo sycallu pro funkci "exit"
        mov     r0,#0            @ exit code = 0
        svc     0                @ volani Linuxoveho kernelu

```

Překlad a slinkování proved'te těmito dvěma příkazy:

```

as arm_thumb.s -o arm_thumb.o
ld -s arm_thumb.o

```

Výsledkem by měl být binární soubor o délce pouhých 311 bajtů:

```

00000000: 7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00 .ELF.....
00000010: 02 00 28 00 01 00 00 00 54 80 00 00 34 00 00 00 ..(.....T...4...
00000020: 98 00 00 00 00 02 00 05 34 00 20 00 01 00 28 00 .....4. ...(.
00000030: 04 00 03 00 01 00 00 00 00 00 00 00 00 80 00 00 .....
00000040: 00 80 00 00 60 00 00 00 60 00 00 00 05 00 00 00 ....`....`.....
00000050: 00 80 00 00 01 70 a0 e3 00 00 a0 e3 00 00 00 ef .....p.....
00000060: 41 13 00 00 00 61 65 61 62 69 00 01 09 00 00 00 A....aeabi.....
00000070: 06 01 08 01 00 2e 73 68 73 74 72 74 61 62 00 2e .....shstrtab..

```

```

0000080: 74 65 78 74 00 2e 41 52 4d 2e 61 74 74 72 69 62 text..ARM.attrib
0000090: 75 74 65 73 00 00 00 00 00 00 00 00 00 00 00 00 utes.....
00000a0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
00000b0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
00000c0: 0b 00 00 00 01 00 00 00 06 00 00 00 54 80 00 00 .....T...
00000d0: 54 00 00 00 0c 00 00 00 00 00 00 00 00 00 00 00 T.....
00000e0: 04 00 00 00 00 00 00 00 11 00 00 00 03 00 00 70 .....p
00000f0: 00 00 00 00 00 00 00 00 60 00 00 00 14 00 00 00 .....`.....
0000100: 00 00 00 00 00 00 00 00 01 00 00 00 00 00 00 00 .....
0000110: 01 00 00 00 03 00 00 00 00 00 00 00 00 00 00 00 .....
0000120: 74 00 00 00 21 00 00 00 00 00 00 00 00 00 00 00 t...!.....
0000130: 01 00 00 00 00 00 00 00 .....

```

Pokud vás zajímá interní struktura tohoto souboru, opět pomůže nástroj **objdump**:

```
objdump -f -d -t -h a.out
```

```

a.out:      file format elf32-littlearm
architecture: armv4, flags 0x00000102:
EXEC_P, D_PAGED
start address 0x00008054

```

Sections:

Idx	Name	Size	VMA	LMA	File off	Algn
0	.text	0000000c	00008054	00008054	00000054	2**2
			CONTENTS, ALLOC, LOAD, READONLY, CODE			
1	.ARM.attributes	00000014	00000000	00000000	00000060	2**0
			CONTENTS, READONLY			

SYMBOL TABLE:

no symbols

Disassembly of section .text:

```

00008054 <.text>:
      8054:      e3a07001      mov     r7, #1
      8058:      e3a00000      mov     r0, #0
      805c:      ef000000      svc     0x00000000

```

V následující části si řekneme, jak vytvořit složitější kód obsahující podmínky, smyčky atd., a to opět v assembleru procesorů i386/x86_64 i ARM (s instrukční sadou Thumb).

11. Repositář se zdrojovými kódy demonstračních příkladů

Oba dva demonstrační příklady byly společně s podpůrnými skripty uloženy do GIT repositáře dostupného na adrese <https://github.com/tisnik/presentations/>:

#	Soubor	Popis	Adresa v repositáři
1	template.s	kód pro i386/x86_64	https://github.com/tisnik/presentations/blob/master/assembler/01_gas_template/template.s
2	arm_thumb.s	kód pro ARM s Thumb	https://github.com/tisnik/presentations/blob/master/assembler/01_gas_template/arm_thumb.s
3	assemble	skript pro překlad na i386/x86_64	https://github.com/tisnik/presentations/blob/master/assembler/01_gas_template/assemble
4	as_arm	skript pro překlad na ARMu	https://github.com/tisnik/presentations/blob/master/assembler/01_gas_template/as_arm

5	disassemble	skript pro zpětný překlad	https://github.com/tisnik/presentations/blob/master/assembler/01_gas_template/disassemble
6	clean	vyčištění adresáře	https://github.com/tisnik/presentations/blob/master/assembler/01_gas_template/clean
7	template.asm	kód pro i386/x86_64	https://github.com/tisnik/presentations/blob/master/assembler/02_nasm_template/template.asm
8	assemble_i386	skript pro překlad na i386	https://github.com/tisnik/presentations/blob/master/assembler/02_nasm_template/assemble_i386
9	assemble_x64_64	skript pro překlad na x86_64	https://github.com/tisnik/presentations/blob/master/assembler/02_nasm_template/assemble_x64_64
10	disassemble	skript pro zpětný překlad	https://github.com/tisnik/presentations/blob/master/assembler/02_nasm_template/disassemble
11	clean	vyčištění adresáře	https://github.com/tisnik/presentations/blob/master/assembler/02_nasm_template/clean

12. Odkazy na Internetu

1. Programování v assembleru na OS Linux

<http://www.cs.vsb.cz/grygarek/asm/asmlinux.html>

2. Is it worthwhile to learn x86 assembly language today?
<https://www.quora.com/Is-it-worthwhile-to-learn-x86-assembly-language-today?share=1>
3. Why Learn Assembly Language?
<http://www.codeproject.com/Articles/89460/Why-Learn-Assembly-Language>
4. Is Assembly still relevant?
<http://programmers.stackexchange.com/questions/95836/is-assembly-still-relevant>
5. Why Learning Assembly Language Is Still a Good Idea
<http://www.onlamp.com/pub/a/onlamp/2004/05/06/writegreatcode.html>
6. Assembly language today
<http://beust.com/weblog/2004/06/23/assembly-language-today/>
7. Assembler: Význam assembleru dnes
<http://www.builder.cz/rubriky/assembler/vyznam-assembleru-dnes-155960cz>
8. Assembler pod Linuxem
<http://phoenix.inf.upol.cz/linux/prog/asm.html>
9. AT&T Syntax versus Intel Syntax
<https://www.sourceware.org/binutils/docs-2.12/as.info/i386-Syntax.html>
10. Linux Assembly website
<http://asm.sourceforge.net/>
11. Using Assembly Language in Linux
<http://asm.sourceforge.net/articles/linasm.html>
12. vasm
<http://sun.hasenbraten.de/vasm/>
13. vasm – dokumentace
<http://sun.hasenbraten.de/vasm/release/vasm.html>
14. The Yasm Modular Assembler Project
<http://yasm.tortall.net/>

15. 680x0:AsmOne
<http://www.amigacoding.com/index.php/680x0:AsmOne>
16. ASM-One Macro Assembler
http://en.wikipedia.org/wiki/ASM-One_Macro_Assembler
17. ASM-One pages
<http://www.theflamearrows.info/documents/asmone.html>
18. Základní informace o ASM-One
<http://www.theflamearrows.info/documents/asminfo.html>
19. Linux Syscall Reference
<http://syscalls.kernelgrok.com/>
20. Programming from the Ground Up Book – Summary
<http://savannah.nongnu.org/projects/pgubook/>
21. IBM System 360/370 Compiler and Historical Documentation
<http://www.edelweb.fr/Simula/>
22. IBM 700/7000 series
http://en.wikipedia.org/wiki/IBM_700/7000_series
23. IBM System/360
http://en.wikipedia.org/wiki/IBM_System/360
24. IBM System/370
http://en.wikipedia.org/wiki/IBM_System/370
25. Mainframe family tree and chronology
http://www-03.ibm.com/ibm/history/exhibits/mainframe/mainframe_FT1.html
26. 704 Data Processing System
http://www-03.ibm.com/ibm/history/exhibits/mainframe/mainframe_PP704.html
27. 705 Data Processing System
http://www-03.ibm.com/ibm/history/exhibits/mainframe/mainframe_PP705.html

28. The IBM 704
<http://www.columbia.edu/acis/history/704.html>
29. IBM Mainframe album
http://www-03.ibm.com/ibm/history/exhibits/mainframe/mainframe_album.html
30. Osmibitové muzeum
<http://osmi.tarbik.com/>
31. Tesla PMI-80
<http://osmi.tarbik.com/cssr/pmi80.html>
32. PMI-80
<http://en.wikipedia.org/wiki/PMI-80>
33. PMI-80
<http://www.old-computers.com/museum/computer.asp?st=1&c=1016>

Pavel Tišnovský



25. 5. 2016

Články

assembler, gas, gdb, nativní kód, programování,
strojový kód

Předchozí příspěvek

Další příspěvek

7 Komentáře

PŘIDEJTE SVŮJ



lzap

25. 5. 2016 at 19:56

Nikdy jsem AT&T syntaxi nep říšel na chuť, překvapením pro mě je, že gasm umí Intel syntaxi ...



atarist

25. 5. 2016 at 20:47

Tak tak, stará dobrá Inteli syntaxe, na te jsem vyrůstal (i když kupodivu ne na Intelim cipu)



marekd

28. 5. 2016 at 20:45

Taky mám raději intel syntaxi 😊 Nechapu proč kolem „linuxu“ je všude jenom at&t syntaxe.



Dan Horák

27. 5. 2016 at 13:54

Asi bych měl udělat pull request, když už jsem ty příklady převedl i na mainframe 😊



Pavel Tisnovsky

27. 5. 2016 at 14:08

jj urcite, to by bylo fajn 😊



Martin

28. 5. 2016 at 09:33

Zavzpomínal jsem si na star é dobré časy s assembler em na ZX Spectrum. 😊



SW

28. 5. 2016 at 10:37

A skvely editor Prometheus. Pr vni legalni soft na ZX Spectrum+, kter y jsem si poridil 😊

Napsat komentář

Vaše e-mailová adresa nebude zveřejněna.

Jméno

Email

Webová stránka

Nejsem robot

reCAPTCHA

[Ochrana soukromí](#) - [Smluvní podmínky](#)

Odeslat komentář

ASSEMBLER

Toto je 1. díl z 24 dílného seriálu na
mojefedora.cz

1

Použití assembleru v Linuxu

2

Použití assembleru v Linuxu:
volání služeb nabízených
jádrom

3

Použití assembleru v Linuxu:
problematika systémové
funkce sys_read

4

Použití assembleru v Linuxu:
podmínky, rozvětvení a
programové smyčky



5

Použití assembleru v Linuxu:
podmínky, rozvětvení a
programové smyčky na
procesorech ARM

6

Použití assembleru v Linuxu:
volání podprogramů a použití
zásobníku

7

Použití assembleru v Linuxu:
zásobníkové rámce na
architektuře Intel, volání
podprogramů na architektuře
ARM

8

Použití assembleru v Linuxu:
makra v GNU Assembleru

9

Použití assembleru v Linuxu:
makra v GNU Assembleru
(dokončení)

10

Použití assembleru v Linuxu:
zpracování celých čísel se
znaménkem

11

Logické a bitové operace na
mikroprocesorech řady x86

12

Použití assembleru v Linuxu:
aritmetické a logické
instrukce i bitové posuny v
praxi

13

Použití assembleru v Linuxu:
operace s jednotlivými bity,
koncept Booleovského
procesoru

14

Použití assembleru v Linuxu:
práce s matematickým

15

Použití assembleru v Linuxu:
práce s matematickým
koprocesorem (pokračování)

16

Použití assembleru v Linuxu:
volání funkcí ze standardní
knihovny jazyka C

17

Použití assembleru v Linuxu:
konvence při volání
knihovních funkcí na
mikroprocesorech ARM

18

Použití assembleru v Linuxu:
volání knihovní funkce printf
s proměnným počtem
parametrů

19

Použití assembleru v Linuxu:
RISCová architektura
AArch64

20

Použití assembleru v Linuxu:
RISCová architektura
AArch64 (programové
smyčky)

21

Použití assembleru v Linuxu:
podmínky při zpracování dat
na architektuře AArch64

22

Použití assembleru v Linuxu:
assembler a jazyk C

23

Kombinace assembleru a
programovacího jazyka C na
procesorech ARM

24