Linux Assembly

Juhász Péter Károly "Stone"
<stone@elte.hu>

2003. december 21.

1. Bevezető

A következőkben bemutatom, hogyan lehet, *GNU/Linux* alatt az *Assembly* nyelvet használni. Ehez a *NASM* 0.98.34-es verziója és a *GNU ld* 2.13.90.0.2-es verziója lesz segítségemre, az elöbbi letölthető a http://nasm.sourceforge.net cimről az utóbbi minden tisztességes Linux disztribúcióban megtalálható.

Az asszembly rejtelmeibe a *Helló világ!* különféle változatain keresztül fogunk egyre beljebb ásni. A leg alapabtól egészen az olyanokig, amit *makró*kat és *eljárás*okat, fognak használni.

2. Helló világ! első változat

Egyenlőre semmi extra nincsen benne csak kiírja az üzenetet a képernyőre és kilép.

```
segment .text

        global _start

_start:
        mov     eax,4
        mov     ebx,1
        mov     ecx,msg
        mov     edx,len
        int     0x80

        mov     eax,1
        mov     ebx,0
        int     0x80

msg     db      "Hello vilag!",0x0a
len     equ     $ - msg
```

A program a következő parancskombinációval fordítható le:

```
nasm -f aout hello.asm
ld -s -o hello hello.o
```

Most nézzük meg egy kicsit közelebbről a programot! A program egy szegmensdefinícióval indul, ezt `.text`-nek nevezzük el. Minden programba kell legalább egy ilyen nevű szegmens. Ebbe kell egy `_start:` cimke amit a `global _start` sorral tehetünk láthatóvá a linker számára erre is szükségünk van minden programba, hogy a linker aztán tudja, hogy hol van a futtatandó program eleje. Majd a program szövege.

Most nézzük a tényleges programot! Két részből áll a kiírásból és a kilépésből. Lássuk először a kiírást:

```
        mov     eax,4
        mov     ebx,1
        mov     ecx,msg
        mov     edx,len
        int     0x80
```

Itt a Linux *write* rendszerhívását használjuk aminek ugynazok a paraméterei mint a megegyező nevű C függvénynek: a file leíró, az üzenet címe és az üzenet hossza. Ezeket sorban az `ebx`, `ecx` és `edx` regiszterbe várja a program, az `eax`-be a write rendszerhívás száma kerül, ami 4. És mindezt a 0x80-as interrupt meghívásval hajtathatjuk végre.

Jelen esetünkben a file leíró a sztenderd kimenet, aminek a kódja 1, az üzenet címére az `msg:` cimke mutat az üzenet hosszát meg a `len:` cimke által mutatott memóriaterületre számolja ki a fordító az üzenet eleji és az üzenet végi memóriacímeg külöbnségéből, az `equ` direktíva segítségével.

Most nézzük a kilépést:

```
        mov     eax,1
        mov     ebx,0
        int     0x80
```

A kilépést a *quit* rendszerhívás segítségével oldjuk meg, aminek a száma 1, amit megint az `eax`-be teszünk be. Az `ebx`-be megy az egyetlen paraméter a visszatéreśi érték, amit sikeres program végrehajtás esetén nullára szokás állítani. Az egészet megint a 0x80-as interrupt meghívásával hajthatjuk végre.

Most tergyünk egy kis kitérőt, a rendszerhívások irányába! Mik is azok és hogy kell őket használni? Mint *DOS*-ban már láttuk, van egy `int 21h` nevü valami, amit ha meghívunk akkor a regiszterek értékétől függően különféle dolgok történnek. Na a Linux `int 0x80` rendszerhívása nagyon hasonló ehez. A rendszerhívások listáját megtalálhatjuk a következő címen:

`http://www.lxhp.in-berlin.de/lhpsysc0a.html`

Valamint ha tudjuk a rendszerhívás számát, csak a paraméterekre nem emlékszünk hirtelen használhatjuk a `man 2 valami` parancsot is mivel ugyanazok lesznek a paraméterek. Hála az isteneknek a Linux betartja a szabványokat.

3. Helló világ! második változat

Most módosítsuk egy kicsit az előző programot úgy, hogy a szöveget ne a képernyőre, hanem egy fileba írja ki. Ezzel megismerhetünk még két rendszerhívást valamint egy kicsit beljebb ásunk a szegmensek témakörébe.

```
section .data

msg     db      "Hello vilag!",0x0a
len     equ     $ - msg
file    db      "hello.txt",0
fd      dw      0

section .text
        global _start

_start:
        mov     eax,5
        mov     ebx,file
        mov     ecx,(1q|100q)
        mov     edx,400q
```

```
int     0x80
mov     [fd],eax

mov     eax,4
mov     ebx,[fd]
mov     ecx,msg
mov     edx,len
int     0x80

mov     eax,6
mov     ebx,[fd]
int     0x80

mov     eax,1
mov     ebx,0
int     0x80
```

Nézzük meg közelebbről az ujdonságokat! Feltünt egy új dolog a `.data` szegmens. Ez egy írható és olvasható terület a programban, mivel a `.text` szegmens nem írható, csak olvasható. Itt azért van szükségünk arra, hogy az adataink írható területen legyenek, mert majd az *open* rendszerhívás által visszaadott file leírót elfogjuk tárolni, a későbbi felhasználás végett. Ezen felül még lehet, egy `.bss` nevü szegmens, amit a gép a még nem inicializált változóknak tart fent. Erre nem lesz szükségünk a továbbiakban.

A NASM nem foglalkozik a szegmens regiszterekkel, nincs *ASSUME* mint például MASM használatakor. Olyan értékeket állítunk be a szegmens regiszterekbe amit akarunk. Valamint a fordító nem generál *szegmens overrride prefixet* sem.

Most lássuk a program többi részét! Először is a file megnyitást:

```
mov eax,5
mov ebx,file
mov ecx,(1q|100q)
mov edx,4001
int 0x80
mov [fd],eax
```

Az *open* rendszerhívás kódja 5, ez megy az `eax` regiszterbe, az `ebx`-be megy a file név, az `ecx`-be a flagek, itt konkrétan a `1q` (ez egy oktális szám, ezt jelzi a *q* e végén) ami a csak írásra való megnyitást jelenti valamint a `100q` ami a létrehzoást jelenti, végezetül az `edx`-be jönnek a jogok, itt `400q` ami a tulajdonos által olvashatóságot jelenti. A többit flag értéket, valamint a jogokat mindenki megnézheti a fentebb említett weboldalon. A 0x80 az interrupt meghívása után az open a file leírót az `eax` regiszterben adja vissza, amit mi elmentünk az `fd` által hivatkozott és a `data` szegmensben létrehozott két bytenyi memóriaterületre.

A kiírásról annyit, hogy csak abban módusul, hogy az `ebx` regiszterbe nem 1-et, hanem az fd-ben tárolt fileleírót tesszük.

Végezetül jöjjön a file lezárása, amit a *close* rendszerhívással végzünk:

```
mov eax,6
mov ebx,[fd]
int 0x80
```

A hívás kódja 6, az egyetlen paramétere a file leíró, amit értelemszerűen az `ebx`-be vár.

Ezekután megint a már megtanult quit rendszerhívás maradt már csak hátra. Ami most változatlan, az előző változathoz képest.

4. Helló világ! harmadik változat

Most térjünk vissza az első helló világ programhoz és megint csináljunk rajta egykét módosítást, nevesen a *makrók* használatával!

```
%macro  exit    1
        mov     eax,1
        mov     ebx,%1
        int     0x80
%endmacro

%macro  print  1+
        jmp     %%end

%%msg   db      %1
%%len   equ     $ - %%msg

%%end   mov     eax,4
        mov     ebx,1
        mov     ecx,%%msg
        mov     edx,%%len
        int     0x80
%endmacro

section .text
        global  _start

_start:
        print   "Hello vilag!",0x0a
        exit    0
```

Nézzük meg egy kicsit közelebbről! Először is teszünk egy kis kitérőt a makrók irányába, mik is ezek? A makró az egy programrész, aminek van neve, és maraméterei, elég egyszer megírni és a programban később a nevével hivatkozhatunk rá. Hogyan kell makrókat megadni? Minden makrót a következő módon kell definiálni:

```
%macro  makro_nev  parameterek_szama

...

%endmacro
```

Ha paraméterszámnak olyat adunk meg, hogy például *2+*, akkor azt úgy veszi, hogy két paraméter lessz, de ha többet is megadunk akkor az utolsó paraméterhez csapja hozzá a felesleget. A mi programunkban a print-et `1+`-al definiáltuk így azt mondtuk meg neki, hogy bármennyi parmétert is adunk meg azok mind közösen az első paramétert alkotják. A paraméterekre a `%1`, `%2`, ...-al hivatkozhatunk. Használhatunk lokális cimkéket is a makróban amiket a `%%`-al kell kezdeni, pl.: `%%msg`. A paramétereket vesszővel elválasztva adjhatjuk át a makróra való hivatkozáskor.

Most lássuk a makróinkat! Itt van mindjárt az első:

```
%macro  exit    1
        mov     eax,1
        mov     ebx,%1
        int     0x80
%endmacro
```

Itt egy exit nevü makrót definiálunk, aminek egy paramétere van, nevesen a kilépési státusz.

Most nézzük az érdekesebb print makrót!

```
%macro  print  1+
        jmp     %%end

%%msg   db      %1
%%len   equ     $ - %%msg

%%end   mov     eax,4
        mov     ebx,1
        mov     ecx,%%msg
        mov     edx,%%len
        int     0x80
%endmacro
```

Ennek is egy paramétere van, de ez már tartalmazhat vesszőket, ezért használtuk a *1+* paraméterszámot. A makró első utasításával elugrunk egy lokális címke által mutatott utasításra, erre azért van szükség, mert tettünk a makróba adatokat is amiket nem szeretnénk végrehajtani, ez `%%end` cimke után meg már a megismert kiírás művelet található.

Most a programunk már csak két sor, ami erre a két makróra hivatkozik.

```
        print   "Hello vilag!",0x0a
        exit    0
```

Látható, hogy miért használtuk a 1+ paraméterszámot a print makrónál, így át tudjuk neki adni a sorvége 0x0a-val ellátott sztringet paraméterként.

5. Helló világ! negyedik változat

Most nézzük meg, hogyan tudunk eljárásokat használni assembly nyelven. Itt is van a következő program ami pont ezt mutatja be.

```
section .text

        global  _start

print:
        mov     eax, 4
        mov     ebx, 1
        mov     edx, [esp + 4]
        mov     ecx, [esp + 8]
        int     0x80
        ret

_start:
        mov     eax, msg
        push    eax
        mov     eax, len
        push    eax
        call    print
        add     esp, 8

        mov     eax, 1
        mov     ebx, 0
```

```
        int     0x80

msg     db      "Hello vilag!", 0x0a
len     equ     $ - msg
```

Nézzük, meg egy kicsit közelebbbről! A programban definiálunk egy print nevü eljárást, amit úgy írtunk meg, hogy két paramétere van az első a kiirandó szöveg címe, a második a kiirandó szöveg hossza. Ezeket, a vermen keresztül adjuk át neki, ezt írj le a következő négy sor:

```
        mov     eax, msg
        push    eax
        mov     eax, len
        push    eax
```

majd, a `call print` hívással adjuk át neki a vezérlést.

Maga a print eljárásunk a már megismert *write* rendszerhívás segítségevél íja ki a szöveget. Ehhez a veremből kell kinyernie a két paramétert.

A verem tetejére a `call` hívás ráteszi a visszatéreśi címet, így a paraméterk az alatt lesznek, aminek a címei sorban az `[esp + 4]`, `[esp + 8]`, `[esp + 12]`, ... lesznek.

Majd mint minden eljárásbÍ ebből is valahogy visszakell térni a főprogramhoz, ezt a `ret` utasítással tehetjük (és teszzük is) meg.

Az eljárás után van egy olyan sor, hogy `add esp, 8` ez azért kell, mert a veremben még mindig benne vannak a függvényünk paraméterei és ezeket kikell szedni, és mivel a verem teteje fele vannak a kisebb címek és ha hozzádaunk 8-at azzal a legutolsó két `push` eredményét töröljük le.

A program többi részében nincsen semmi új, azok ugyanazok mint eddig.

6. Helló világ! ötödik változat

Most nézzük meg, hogy hogyan használhatjuk fel az assembly tudásunkat, ha például C-ben programozunk. A programunk még mindig a Helló világ legyen, de most írjuk meg C-ben de használjunk hozzá egy kis assemblyt!

A következő file-t nevezzük main.c-nek:

```
#include <stdio.h>

extern int osszead(int,int);

int main() {
  printf("Hello vilag! %d\n", osszead(2,3));
  return 0;
}
```

és ezt meg osszead.asm-nek:

```
bits    32

section .text

global  osszead

osszead:
        push    ebp
        push    esi
push edi
push ebx
```

```
        mov     eax,[esp + 20]
        mov     ebx,[esp + 24]
        add     eax,ebx

pop ebx
pop esi
pop edi
pop ebx

        ret
```

és fordítsuk le őket a következő módon:

```
nasm -f elf osszead.asm
gcc main.c osszead.o
```

és ha ezekután kipróbáljuk a következő kimenetet kapjuk:

```
Hello vilag! 5
```

Lássuk egy kicsit közelebbről, a C programban van egy külső függvény hívás amit az ismert `extern` kulcsszóval adtunk meg. Az assembly programunkban meg van egy ugyanolyan nevü publikus függvény (függvény, mert `ret` van a végén) ami összeadja a két paraméterként kapott számot.

Hogyan is müködik ez? Először is az elején van egy olyan, hogy `bits 32` ez azért kell, hogy minden utasítást 32 bitesen fordítson, ez ahhoz kell, hogy a C programunkal kompatibilis legyen. Az asszemblyben megírt függvény a C programtól a paramétereit, fordított sorrendben a veremben kapja meg. A verem tetejére még egy visszatérési érték is bekerül, így, a paraméterek címei sorban az `[esp + 4]`, `[esp + 8]`, `[esp + 12]`, ... lesznek, jelen esetben csak kettő. Mivel a programban még egykét tegiszter értékét elmentjük a verembe ezek az értékek változnak jelen esetben 16-al nőnek. Ha ezeket nem mentenénk el akkor nḿelyik fordítóval Segmentation fault-os programot gyártanánk. Pontosan úgy, ahogy az előző programnál láttuk. A promgram beilvassa őket az `eax` és `ebx` regiszterbe, majd összeadja őket az `eax`-be. A fügvények visszatéreési értékét az `eax` regiszterbe kell tenni. Végül a `ret` utasítással visszatér a főprogramhoz.