Linux Assembly Tutorial

Quickstart

Written by: Derick Swanepoel (derick@maple.up.ac.za)
Version 1.0 - 2002-04-19, 01:50am

Download as zipfile

`JMP` Step-by-Step Guide

Contents

---

1. Intro

This Quickstart aims to show you the ropes on Linux assembly as quickly as possible. Basically, it just points out the differences between a Linux and DOS assembly program with just enough explanation not to confuse you. For more detail and why things are the way they are, see the Step-by-Step Guide.

2. Comparison of a Linux assembly program and a DOS assembly program

Linux

```
SECTION .DATA
        hello:     db 'Hello world!',10
        helloLen:  equ $-hello

SECTION .TEXT
        GLOBAL _START

_START:

        ; Write 'Hello world!' to the screen
        mov eax,4            ; 'write' system call
        mov ebx,1            ; file descriptor 1 = screen
        mov ecx,hello        ; string to write
        mov edx,helloLen     ; length of string to write
        int 80h              ; call the kernel

        ; Terminate program
        mov eax,1            ; 'exit' system call
        mov ebx,0            ; exit with error code 0
        int 80h              ; call the kernel
```

Compiling: `nasm -f elf hello.asm`
Linking: `ld -s -o hello hello.o`

DOS

```
DOSSEG
.MODEL LARGE
.STACK 200h

.DATA
        hello      db 'Hello world!',10,13,'$'
        helloLen   db 14

.CODE
        ASSUME CS:@CODE, DS:@DATA

START:
        mov ax,@data
        mov ds,ax

        ; Write 'Hello world!' to the screen
        mov ah,09h             ; 'print' DOS ser
        mov dx,offset hello    ; string to write
        int 21h                ; call DOS servic

        ; Terminate program
        mov ah,4Ch             ; 'exit' DOS serv
        mov ax,0               ; exit with error
        int 21h                ; call DOS servic
END START
```

Compiling: `tasm hello.asm`
Linking: `tlink hello.obj`

Lets compare each part in the two programs:

- The first three lines of the DOS program doesn't exist in the Linux program. Linux is a 32-bit protected mode operating system, and in 32-bit assembly there are no memory models. Also, all segment registers and paging have already been set up to give you the same

32-bit 4Gb address space, so you can ignore all segment registers. It is also not necessary to specify the stack size.

- Some differences between the Linux NASM structure and DOS TASM/MASM structure:
  - The data / code sections are defined by writing `SECTION .DATA` instead of just `.DATA`
  - Linux NASM allows us to declare constants with the `EQU` instruction, for example:
    `bufferlen: equu 400`
    So whenever it sees `bufferlen` in your program, it will substitute the value '400'. That means you don't have to put square brackets around `bufferlen` to get its actual value. (Note: this only works for constants. The values of all other variables are still obtained using `[varname]`).
  - Another neat NASM feature is the '$' token: when NASM sees an '$' it substitutes it with the assembly position at the beginning of that line. So what does this mean? This gives us an easy way to define the length of a string we've just declared. After declaring a variable containg a string

    ```
            hello:     db 'Hello world!',10
    ```

    we can put on the next line

    ```
            helloLen:  equ $-hello
    ```

    This will make `helloLen` equal to (position at beginning of line) - (position of `hello`). If you look at those two lines in the program, you can see this will give us the length of `'Hello world!',10`, which is 13 (12 characters plus the linefeed character).
    If this doesn't make sense, don't worry, just know that this is an easy way to define the length of a string.
  - Strings you want to print out in Linux don't need to be $-terminated like in DOS. Instead, you supply the length of the string as one of the parameters. (This is much more flexible, because you can now print out only a part of the string, and your computer won't blow up when you forget the $ like in DOS.)
  - To print out a string with a newline at the end, you only need to need to add a linefeed character (10) to the end of the string in Linux. In DOS, you need both a linefeed and a carriage return (13).
  - The `.CODE` section is called `.TEXT` in Linux
  - Right at the beginning of the `.TEXT` section, there must be a declaration specifying the entry point of the program: `GLOBAL _START`
  - There is no `ASSUME` directive like in the DOS program, because we don't need to worry about segments.
  - The program's entry point is called whatever you declared it to be at the beginning of the `.TEXT` section (in our case it's `_START`). Also, the Linux program doesn't end with `END _START` like the DOS program.
- The first two lines after the `START:` label in the DOS program make sure that the DS register points to the data segment. Once again, Linux doesn't need this because the segments are taken care of for us.
- In 16-bit DOS assembly, we use the normal 16-bit registers AX, BX, CX, DX etc. In 32-bit Linux assembly we use the 32-bit extended registers EAX, EBX, ECX, EDX etc. (Note that there is no such thing as EAL. AX is the low 16 bits of EAX, while AH is the high 8 bits of AX and AL is the low 8 bits of AX.)
- In DOS, when we want the memory address of a variable to be put in a register, we must use `offset` to point to the offset of the variable in the correct segment (`mov dx,offset hello`). In Linux, we don't need `offset` because it's implied - we just write `mov ecx,hello`
- In DOS, we call `int 21h` to use a DOS service like printing out a string. In Linux, you use system calls, which are accessed by calling `int 80h` (the kernel interrupt). In DOS the function number (eg. 9 to print a string) always goes in AX; in Linux it always goes in EAX. As in the example, if we want to print out a string we use the "write" syscall, which is function number 4. We put '4' in EAX, the number of the file descriptor to write to in EBX (in this case '1', the screen), the location of the string to print in ECX (`mov ecx,hello`), and the length of the string in EDX (`mov ecx,helloLen`). Then we call the kernel interrupt (`int 80h`), and voila!
- To exit a Linux program, we use the exit syscall (function 1, so we write `mov eax,1`). We want to exit with an exit code of 0 (no error), so we put 0 in EBX. Then we call the kernel with `int 80h` again, and we're done.

In this case it looks like it's more work than in DOS, but when it comes to creating, reading and writing files, Linux syscalls are much easier to use than their DOS counterparts.

3. More About Linux System Calls

There are six registers that are used for the arguments that a system call takes. The first argument goes in EBX, the second in ECX, then EDX, ESI, EDI, and finally EBP, if there are so many. If there are more than six arguments, EBX must contain the memory location where the list of arguments is stored.

All the syscalls are listed in `/usr/include/asm/unistd.h`, together with their numbers. However, for your convenience you can simply find them in this Linux System Call Table, together with some other useful information (eg. what arguments they take). The syscalls are fully documented in section 2 of the manual pages, so you can just go `man 2 write` to find out what the write syscall does, what arguments it takes, etc.

4. Command Line Arguments and Writing to a File

Linux

```
section .data
        hello     db    'Hello, world!',10      ; Our dear string
        helloLen  equ   $ - hello               ; Length of our dear string

section .text
    global _start

_start:
        pop     ebx             ; argc (argument count)
        pop     ebx             ; argv[0] (argument 0, the program name)
        pop     ebx             ; The first real arg, a filename

        mov     eax,8           ; The syscall number for creat() (we already have the filename in ebx)
        mov     ecx,00644Q      ; Read/write permissions in octal (rw_rw_rw_)
        int     80h             ; Call the kernel
                                ; Now we have a file descriptor in eax

        test    eax,eax         ; Lets make sure the file descriptor is valid
        js      skipWrite       ; If the file descriptor has the sign flag
                                ;  (which means it's less than 0) there was an oops,
                                ;  so skip the writing. Otherwise call the filewrite "procedure"
        call    fileWrite

skipWrite:
        mov     ebx,eax         ; If there was an error, save the errno in ebx
        mov     eax,1           ; Put the exit syscall number in eax
        int     80h             ; Bail out

; proc fileWrite - write a string to a file
fileWrite:
        mov     ebx,eax         ; sys_creat returned file descriptor into eax, now move into ebx
        mov     eax,4           ; sys_write
                                ; ebx is already set up
        mov     ecx,hello       ; We are putting the ADDRESS of hello in ecx
        mov     edx,helloLen    ; This is the VALUE of helloLen because it's a constant (defined with equ)
        int     80h

        mov     eax,6           ; sys_close (ebx already contains file descriptor)
        int     80h
        ret
; endp fileWrite
```

DOS

```
DOSSEG
.MODEL LARGE
.STACK 200h

.DATA
filename        db 14 dup (0)
filehandle      dw
hello           db 'Hello World!',10,13,'$'
helloLen        db 12

.CODE
ASSUME CS:@CODE, DS:@DATA

START:
        mov     AX,@DATA
        mov     ES,AX                  ; Point ES to the data segment for now

        mov     ah,62h
        int     21h                    ; Get the PSP
        mov     ds,bx
        mov     bx,81h                 ; Starting at the first printable character
        add     bl, byte ptr [ds:80h]  ; Get address of last character
        mov     cl, byte ptr [ds:80h]  ; Also put it in CL
        inc     cl
        mov     [ds:bx], word ptr 0    ; Null terminate the argument

        mov     si,81h
        mov     di,0                   ; Copy the first argument into the data segment
        rep     movsb                  ;  into the filename variable

        mov     AX,@DATA
```

```
        mov     DS,AX                   ; Point DS to the data segment, like normal

        call    fileCreate
        call    fileWrite
        call    fileClose

        mov     AX,4C00h
        int     21h                     ; Bye-bye!
END START

proc fileCreate
        mov     ah,3Ch                  ; Creat DOS service (yes, it is called 'creat')
        mov     cx,0                    ; File attributes
        mov     dx,offset filename      ; Put ADDRESS of filename in DX
        int     21h

        mov     [filehandle],ax         ; File handle is returned in AX, put in a variable
        ret
endp fileCreate

proc fileClose
        mov     ah,3Eh
        mov     bx,[filehandle]
        int     21h
        ret
endp fileClose

proc fileWrite
        mov     ah, 40h
        mov     bx, [filehandle]
        mov     dx, offset hello        ; ADDRESS of string to be written
        xor     cx, cx                  ; If I don't do this, things blow up in my face
        mov     cl, [helloLen]          ; VALUE of length of string to be written
        int     21h
        ret
endp fileWrite
```

As you can see, the Linux program is much simpler than the DOS one (40 lines in Linux, with liberal commenting, vs. 66 for DOS). Everything makes sense in the Linux program, whereas a lot of the stuff in the DOS one still makes me go "Huh?" Lets check out the differences:

1. Firstly, getting the command line arguments of the Linux program is *way* easier than the DOS one. All the arguments are sitting on the stack when the program starts, so all we need to do is `pop` them off. The first value popped off is the number of arguments (called argc in C/C++), the second is the name of the program, and finally we get the actual command line arguments. Coolest of all, when we pop the command line argument off the stack, it actually puts the address of that string in EBX, so once again no segment/offset missions.
   This just took us an entire 3 instructions - compared to the 14 insane ones for the DOS program! No messing around with PSPs and stuff - simple, isn't it?
2. **NB:** NASM doesn't have procedures like you may have used in TASM. That's because procedures don't really exist in assembly: everything is a label. So if you want to write a "procedure" in NASM, you don't use `proc` and `endp`, but instead just put a label (eg. `filewrite:`) at the beginning of the "procedure's" code. If you want to, you can put comments at the start and end of the code just to make it look a bit more like a procedure (like I did in the example).
3. **NB^2:** When you jump to a label with `JMP` or any of the jump instructions, you *don't* `RET` from it. Never! If you're lucky it won't explode on you, but it's definitely not right. The only time you `RET` is when you've called the "procedure" with `CALL`. Otherwise you're just going to have to jump around like a kangaroo weaving a spaghetti code masterpiece. (Note that this is applicable to any assembly, not just Linux or NASM).
4. Next we create the file: notice the file permissions in Linux (you can find out more about them by reading the creat syscall's manpage – yes, it is spelled "creat"). Since we want to be smart with Linux, why not also include some error checking while we're at it? We can easily check if the creat syscall failed by checking the value it returned: if it's less than 0 then something broke, so skip the writing part and exit with the error code.
5. Now we write 'Hello world!' to the file using the file descriptor (called file handle in DOS) returned by the creat syscall. Then we close it, and exit.

Not so hectic at all.

<u>On the side:</u> If you look at the DOS service functions (int 21h), you may notice that there are quite a few that have exactly the same names as their Unix/Linux syscall counterparts – even though DOS is quite unlike Unix and very much incompatible with it. For example: DOS 3Ch = CREAT, Unix 08h = creat and DOS 43h = CHMOD, Unix 0Fh = chmod. Mmm... so where did these DOS functions get their names? From Unix of course! What is really amusing is that Microsoft never bothered to spell "CREAT" right – they kept it exactly like Unix's "creat".

5. Compiling and Linking

To compile a program with NASM:

```
nasm -f elf program.asm
```

To link the object file produced by NASM into an executable:

```
ld -s -o program program.o
```

The `-f elf` option tells NASM to compile this in Linux ELF format. This option is necessary because NASM can compile many different formats (even DOS .COM files if you're so inclined).
The `-s` option for Ld tells it to strip all symbol information (which you don't need) from the output file. `-o` *`program`* specifies the name of the output executable file. If you leave it out it will always be `a.out`

Appendix A. References

[Writing a useful program with NASM]
[The NASM documentation]
[Introduction to UNIX assembly programming]
[Linux Assembler Tutorial by Robin Miyagi]
Section 2 of the manpages