Code as Art

Blog about system programming and not only

Say hello to x64 Assembly [part 1]

Introduction

There are many developers between us. We write a tons of code every day. Sometime, it is even not a bad code :) Every of us can easily write the simplest co

```c
#include <stdio.h>

int main() {
  int x = 10;
  int y = 100;
  printf("x + y = %d", x + y);
  return 0;
}
```

gistfile1.c hosted with ❤ by GitHub

Every of us can understand what's this C code does. But... How this code works at low level? I think that not all of us can answer on this question, and me too that i can write code on high level programming languages like Haskell, Erlang, Go and etc..., but i absolutely don't know how it works at low level, after co I decided to take a few deep steps down, to assembly, and to describe my learning way about this. Hope it will be interesting, not only for me. Something a years ago I already used assembly for writing simple programs, it was in university and i used Turbo assembly and DOS operating system. Now I use Linux-x operating system. Yes, must be big difference between Linux 64 bit and DOS 16 bit. So let's start.

Preparation

Before we started, we must to prepare some things like As I wrote about, I use Ubuntu (Ubuntu 14.04.1 LTS 64 bit), thus my posts will be for this operating s architecture. Different CPU supports different set of instructions. I use *Intel Core i7 870* processor, and all code will be written processor. Also i will use nasr You can install it with:

```
sudo apt-get install nasm
```

It's version must be 2.0.0 or greater. I use *NASM version 2.10.09 compiled on Dec 29 2013* version. And the last part, you will need in text editor where you assembly code. I use Emacs with *nasm-mode.el* for this. It is not mandatory, of course you can use your favourite text editor. If you use Emacs as me you can nasm-mode.el and configure your Emacs like this:

```
(load "~/.emacs.d/lisp/nasm.el")
(require 'nasm-mode)
(add-to-list 'auto-mode-alist '("\\.\\(asm\\|s\\)$" . nasm-mode))
```

gistfile1.el hosted with ❤ by GitHub

That's all we need for this moment. Other tools will be describe in next posts.

x64 syntax

Here I will not describe full assembly syntax, we'll mention only those parts of the syntax, which we will use in this post. Usually NASM program divided into this post we'll meet 2 following sections:

- data section
- text section

The data section is used for declaring constants. This data does not change at runtime. You can declare various math or other constants and etc... The syntax data section is:

```
section .data
```

The text section is for code. This section must begin with the declaration *global _start*, which tells the kernel where the program execution begins.

```
section .text
global _start
_start:
```

Comments starts with *;* symbol. Every NASM source code line contains some combination of the following four fields:

[label:] instruction [operands] [; comment]

Fields which are in square brackets are optional. A basic NASM instruction consists from two parts. The first one is the name of the instruction which is to be
and the second are the operands of this command. For example:

MOV COUNT, 48 ; Put value 48 in the COUNT variable

Hello world

Let's write first program with NASM assembly. And of course it will be traditional Hello world program. Here is the code of it:

```
section .data
    msg db      "hello, world!"

section .text
    global _start
_start:
    mov     rax, 1
    mov     rdi, 1
    mov     rsi, msg
    mov     rdx, 13
    syscall
    mov    rax, 60
    mov    rdi, 0
    syscall
```

gistfile1.asm hosted with ❤ by GitHub

Yes, it doesn't look like *printf("Hello world").* Let's try to understand what is it and how it works. Take a look 1-2 lines. We defined *data* section and put there
constant with *Hello world* value. Now we can use this constant in our code. Next is declaration *text* section and entry point of program. Program will start to
7 line. Now starts the most interesting part. We already know what is it *mov* instruction, it gets 2 operands and put value of second to first. But what is it thes
etc... As we can read at wikipedia:

A central processing unit (CPU) is the hardware within a computer that carries out the instructions of a computer program by performing the basic arithr
logical, and input/output operations of the system.

Ok, CPU performs some operations, arithmetical and etc... But where can it get data for this operations? The first answer in memory. However, reading data fr
storing data into memory slows down the processor, as it involves complicated processes of sending the data request across the control bus. Thus CPU has
memory storage locations called **registers**:

64-bit register	Lower 32 bits	Lower 16 bits	Lower 8 bits
rax	eax	ax	al
rbx	ebx	bx	bl
rcx	ecx	cx	cl
rdx	edx	dx	dl
rsi	esi	si	sil
rdi	edi	di	dil
rbp	ebp	bp	bpl
rsp	esp	sp	spl
r8	r8d	r8w	r8b
r9	r9d	r9w	r9b
r10	r10d	r10w	r10b
r11	r11d	r11w	r11b
r12	r12d	r12w	r12b
r13	r13d	r13w	r13b
r14	r14d	r14w	r14b
r15	r15d	r15w	r15b

So when we write *mov rax, 1*, it means to put 1 to the *rax* register. Now we know what is it rax, rdi, rbx and etc... But need to know when to use rax but when

- rax - temporary register; when we call a syscal, rax must contain syscall number
- rdx - used to pass 3rd argument to functions
- rdi - used to pass 1st argument to functions
- rsi - pointer used to pass 2nd argument to functions

In another words we just make a call of *sys_write* syscall. Take a look on *sys_write*:

```
ssize_t sys_write(unsigned int fd, const char * buf, size_t count)
```

gistfile1.c hosted with ❤ by GitHub

It has 3 arguments:

- fd - file descriptor. Can be 0, 1 and 2 for standard input, standard output and standard error
- buf - points to a character array, which can be used to store content obtained from the file pointed to by fd.
- count - specifies the number of bytes to be written from the file into the character array

So we know that *sys_write* syscall takes three arguments and has number one in syscall table. Let's look again to our hello world implementation. We put 1 register, it means that we will use sys_write system call. In next line we put 1 to rdi register, it will be first argument of sys_write, 1 - standard output. Then v pointer to *msg* at rsi register, it will be second *buf* argument for sys_write. And then we pass the last (third) parameter (length of string) to rdx, it will be thirc sys_write. Now we have all arguments of sys_write and we can call it with *syscall* function at 11 line. Ok, we printed "Hello world" string, now need to do cor from program. We pass 60 to rax register, 60 is a number of exit syscall. And pass also 0 to rdi register, it will be error code, so with 0 our program must ex successfully. That's all for "Hello world". Quite simple :) Now let's build our program. For example we have this code in *hello.asm* file. Then we need to exec commands:

```
nasm -f elf64 -o hello.o hello.asm
ld -o hello hello.o
```

After it we will have executable *hello* file which we can run with ./hello and will see Hello world string in the terminal.

Conclusion

It was a first part with one simple-simple example. In next part we will see some arithmetic. If you will have any questions/suggestions write me a comment

All source code you can find - here.

Labels: asm, Linux, x64