Code as Art

Blog about system programming and not only

Say hello to x64 Assembly [part 3]

Stack

Some time ago i started to write a series of posts about assembly x64 programming. It is third part and it will be about stack. The stack is special region in (built into the CPU), which operates on the principle lifo (Last Input, First Output).

We have 16 general-purpose registers for temporary data storage. They are RAX, RBX, RCX, RDX, RDI, RSI, RBP, RSP and R8-R15. It's too few for serious app we can store data in the stack. Yet another usage of stack is following: When we call a function, return address copied in stack. After end of function executic copied in commands counter (RIP) and application continue to executes from next place after function.

For example:

```
1  global _start
2  
3  section .text
4  
5  _start:
6                  mov rax, 1
7                  call incRax
8                  cmp rax, 2
9                  jne exit
10                 ;;
11                 ;; Do something
12                 ;;
13 
14 incRax:
15                 inc rax
16                 ret
```

gistfile1.asm hosted with ❤ by GitHub

Here we can see that after application runnning, *rax* is equal to 1. Then we call a function *incRax*, which increases *rax* value to 1, and now *rax* value must be execution continues from 8 line, where we compare *rax* value with 2. Also as we can read in System V AMD64 ABI, the first six function arguments passed in They are:

- rdi - first argument
- rsi - second argument
- rdx - third argument
- rcx - fourth argument
- r8 - fifth argument
- r9 - sixth

Next arguments will be passed in stack. So if we have function like this:

```
int foo(int a1, int a2, int a3, int a4, int a5, int a6, int a7)
{
    return (a1 + a2 - a3 - a4 + a5 - a6) * a7;
}
```

gistfile1.c hosted with ❤ by GitHub

Then first six arguments will be passed in registers, but 7 argument will be passed in stack.

Stack pointer

As i wroute about we have 16 general-purpose registers, and there are two interesting registers - *RSP* and *RBP*. *RBP* is the base pointer register. It points to t the current stack frame. *RSP* is the stack pointer, which points to the top of current stack frame.

Commands

We have two commands for work with stack:

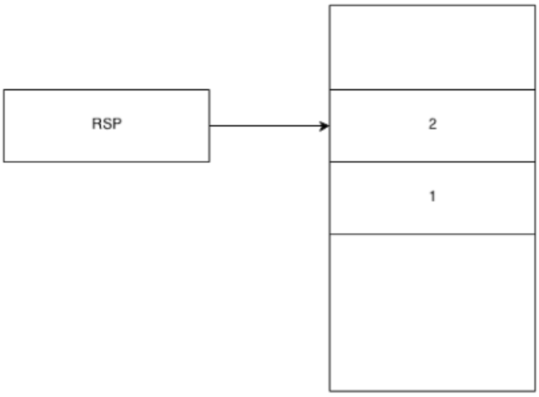
- push argument - increments stack pointer (RSP) and stores argument in location pointed by stack pointer
- pop argument - copied data to argument from location pointed by stack pointer

Let's look on one simple example:

```
global _start

section .text

_start:
                mov rax, 1
                mov rdx, 2
                push rax
                push rdx

                mov rax, [rsp + 8]

                ;;
                ;; Do something
                ;;
```

gistfile1.asm hosted with ❤ by **GitHub**

Here we can see that we put 1 to *rax* register and 2 to *rdx* register. After it we push to stack values of these registers. Stack works as LIFO (Last In First Out). S
stack or our application will have following structure:



Then we copy value from stack which has address *rsp + 8*. It means we get address of top of stack, add 8 to it and copy data by this address to *rax*. After it *r*
be 1.

Example

Let's see one example. We will write simple program, which will get two command line arguments. Will get sum of this arguments and print result.

```
section .data
                SYS_WRITE equ 1
                STD_IN    equ 1
                SYS_EXIT  equ 60
                EXIT_CODE equ 0

                NEW_LINE   db 0xa
                WRONG_ARGC db "Must be two command line argument", 0xa
```

gistfile1.asm hosted with ❤ by **GitHub**

First of all we define *.data* section with some values. Here we have four constants for linux syscalls, for sys_write, sys_exit and etc... And also we have two str
just new line symbol and second is error message.

Let's look at .text section, which consists from code of program:

```
section .text

        global _start

_start:
                pop rcx
                cmp rcx, 3
                jne argcError

                add rsp, 8
```

```
11                  pop rsi
12                  call str_to_int
13
14                  mov r10, rax
15                  pop rsi
16                  call str_to_int
17                  mov r11, rax
18
19                  add r10, r11
```

gistfile1.asm hosted with ❤ by GitHub

Let's try to understand, what is happening here: After *_start* label first instruction get first value from stack and puts it to *rcx* register. If we run application wi line arguments, all of their will be in stack after running in following order:

- [rsp] - top of stack will contain arguments count.
- [rsp + 8] - will contain argv[0]
- [rsp + 16] - will contain argv[1]
- and so on...

So we get command line arguments count and put it to *rcx*. After it we compare *rcx* with 3. And if they are not equal we jump to *argcError* label which just pr message:

```
argcError:
        ;; sys_write syscall
        mov     rax, 1
        ;; file descritor, standard output
        mov     rdi, 1
        ;; message address
        mov     rsi, WRONG_ARGC
        ;; length of message
        mov     rdx, 34
        ;; call write syscall
        syscall
        ;; exit from program
        jmp exit
```

gistfile1.asm hosted with ❤ by GitHub

Why we compare with 3 when we have two arguments. It's simple. First argument is a program name, and all after it are command line arguments which we program. Ok, if we passed two command line arguments we go next to 10 line. Here we shift *rsp* to 8 and thereby missing the first argument - the name of t Now *rsp* points to first command line argument which we passed. We get it with *pop* command and put it to *rsi* register and call function for converting it to we read about *str_to_int* implementation. After our function ends to work we have integer value in *rax* register and we save it in *r10* register. After this we do operation but with *r11*. In the end we have two integer values in *r10* and *r11* registers, now we can get sum of it with *add* command. Now we must convert re and print it. Let's see how to do it:

```
mov rax, r10
;; number counter
xor r12, r12
;; convert to string
jmp int_to_str
```

gistfile1.asm hosted with ❤ by GitHub

Here we put sum of command line arguments to rax register, set *r12* to zero and jump to *int_to_str*. Ok now we have base of our program. We already know string and we have what to print. Let's see at *str_to_int* and *int_to_str* implementation.

```
str_to_int:
            xor rax, rax
            mov rcx,  10
next:
            cmp [rsi], byte 0
            je return_str
            mov bl, [rsi]
            sub bl, 48
            mul rcx
            add rax, rbx
            inc rsi
            jmp next

return_str:
            ret
```

gistfile1.asm hosted with ❤ by GitHub

At the start of *str_to_int*, we set up *rax* to 0 and *rcx* to 10. Then we go to *next* label. As you can see in above example (first line before first call of str_to_int) *argv[1]* in *rsi* from stack. Now we compare first byte of *rsi* with 0, because every string ends with NULL symbol and if it is we return. If it is not 0 we copy it's v byte *bl* register and substract 48 from it. Why 48? All numbers from 0 to 9 have 48 to 57 codes in asci table. So if we substract from number symbol 48 (for from 57) we get number. Then we multiply *rax* on *rcx* (which has value - 10). After this we increment *rsi* for getting next byte and loop again. Algorthm is sim example if *rsi* points to *'5' '7' '6' '\000'* sequence, then will be following steps:

- rax = 0
- get first byte - 5 and put it to rbx
- rax * 10 --> rax = 0 * 10
- rax = rax + rbx = 0 + 5
- Get second byte - 7 and put it to rbx
- rax * 10 --> rax = 5 * 10 = 50
- rax = rax + rbx = 50 + 7 = 57
- and loop it while rsi is not \000

After *str_to_int* we will have number in *rax*. Now let's look at *int_to_str*:

```
int_to_str:
                mov rdx, 0
                mov rbx, 10
                div rbx
                add rdx, 48
                add rdx, 0x0
                push rdx
                inc r12
                cmp rax, 0x0
                jne int_to_str
                jmp print
```


Here we put 0 to *rdx* and 10 to *rbx*. Than we exeute *div rbx*. If we look above at code before str_to_int call. We will see that *rax* contains integer number - su command line arguments. With this instruction we devide *rax* value on *rbx* value and get reminder in *rdx* and whole part in *rax*. Next we add to *rdx* 48 and *0x* adding 48 we'll get asci symbol of this number and all strings much be ended with 0x0. After this we save symbol to stack, increment r12 (it's 0 at first iterat to 0 at the _start) and compare rax with 0, if it is 0 it means that we ended to convert integer to string. Algorithm step by step is following: For example we h 23

- 123 / 10. rax = 12; rdx = 3
- rdx + 48 = "3"
- push "3" to stack
- compare rax with 0 if no go again
- 12 / 10. rax = 1; rdx = 2
- rdx + 48 = "2"
- push "2" to stack
- compare rax with 0, if yes we can finish function execution and we will have "2" "3" ... in stack

We implemented to useful function *int_to_str* and *str_to_int* for converting integer number to string and vice versa. Now we have sum of two integers which v into string and saved in the stack. We can print result:

```
print:
        ;;;; calculate number length
        mov rax, 1
        mul r12
        mov r12, 8
        mul r12
        mov rdx, rax
        ;;;;

        ;;;; print sum
        mov rax, SYS_WRITE
        mov rdi, STD_IN
        mov rsi, rsp
        ;; call sys_write
        syscall
        ;;;;

        jmp exit
```