We are going to put our **executable code** in the .text section and our **variables** in the .data section:

Firstly, this file, /usr/include/x86\_64-linux-gnu/asm/unistd\_64.h, contains all the syscalls for Linux x64. Let's search for the exit syscall:

Now, let's see how we are going to use registers to invoke Linux x64 syscalls:



**Data types**

**Name Directive Bytes Bits**

Byte db 1 8

Word dw 2 16

Doubleword dd 4 32

Quadword dq 8 64

**Hello world**

**$ cat /usr/include/x86\_64-linux-gnu/asm/unistd\_64.h | grep write**

**#define \_\_NR\_write 1**

**#define \_\_NR\_pwrite64 18**

**#define \_\_NR\_writev 20**

**#define \_\_NR\_pwritev 296**

**#define \_\_NR\_process\_vm\_writev 311**

**#define \_\_NR\_pwritev2 328**

We can see that the write syscall is number 1; now let's look at its arguments:

**$ man 2 write**

The write syscall has three arguments; the first one is the file descriptor:

**ssize\_t write(int fd, const void \*buf, size\_t count);**

The file descriptor has three modes:

**Integer value Name Alias for** stdio.h

0 Standard input stdin

1 Standard output stdout

2 Standard erro stderr

As we are going to print hello world on the screen, we are going to choose standard output 1, the second argument, which is a pointer to the string we want to print; the third argument is the count of the string, including spaces.



And now, let's jump to the full code:

global \_start

section .text

\_start:

mov rax, 1

mov rdi, 1

mov rsi, hello\_world

mov rdx, length

syscall

section .data

hello\_world: db 'hello world',

In the .data section, which contains all the variables, the first variable in the code is

the hello\_world variable with data type byte (db), and it contains a hello world string along with 0xa, which means a new line, like in \n in C. The second variable is length, that contains the length of hello\_world string with equ, which means equal, and $-, which means evaluate the current line.

In the .text section, as we previously explained, we move 1 to rax, which indicates the write syscall number, then we move 1 to rdi as an indicator that the file descriptor is set to standard output, then we move the address of the hello\_world string to rsi, and we move the length of the hello\_world string to rdx, and finally, we invoke syscall, which means execute.

Now, let's assemble and link the object code, as follows:

**$ nasm -felf64 hello-world.nasm -o hello-world.o**

**$ ld hello-world.o -o hello-world**

**$ ./hello-world**

**Example (Full code):**

global \_start

section .text

\_start:

mov rax, 1

mov rdi, 1

mov rsi, hello\_world

mov rdx, length

syscall

mov rax, 60

mov rdi, 1

syscall

section .data

hello\_world: db 'hello world',0xa

length: equ $-hello\_world

**Stack**

As we discussed in the previous chapter, a **stack** is a space allocated for each running

application and is used to store variables and data. A stack supports two operations (push and pop); a **push** operation is used to push an element to the stack, and that will cause the stack pointer to move to a lower memory address (a stack grows from high memory to low memory) and point to the top of the stack, whereas **pop** takes the first element at the top of the stack and extracts it

Let's take a look at a simple example:

global \_start

section .text

\_start:

mov rdx,0x1234

push rdx

push 0x5678

pop rdi

pop rsi

mov rax, 60

mov rdi, 0

syscall

section .data

This code is very simple; let's compile and link it:

**$ nasm -felf64 stack.nasm -o stack.o**

**$ ld stack.o -o stack**.