`

بسم الله الرحمن الرحيم

ANSWER ASSIGNMENT:

Q(P1): Code assembly program "Hello World!" using NASM assembler?

AS A FULFILLMENT OF CAO COURSE MIDTERM

Dr. Abdulwasea Alazani

Eng. Bakeel Azman

Student name: Ahmed Ehab Saleh Al-kataby

2023-2022

`

1. INTRODUCTION

1.1. About Assignment.

This documentation would explain significant fundamentals of Assembly Language in order to code a pure "Hello, World!" program that will be eventually brought to life by assembling, linking and executing processes. The used tools are NASM Assembler[1], and The GNU gold linker, Ld Linker (preinstalled linker on Linux). The application is going to be on Kali Linux OS and x86_64 Architecture. [1],[5]

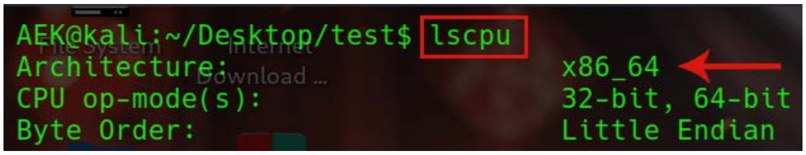
1.2. What Is NASM Assembler?

NASM is an open source x86 and x86_64 assembler developed by John S. Fine, Simon Tatham and Julian Hall in 1996. It supports most platforms in many object file formats, such as ELF and ELF64 on Linux, Win32 and Win64 on Windows OS. Its syntax is designed to be simple and easy to understand. It is case-sensitive for programmer identifiers, but not case-sensitive for the instruction and register names. NASM Assembler is easy to use on Linux rather than Windows OS; however, Windows OS commonly uses MASM[2]. [1],[2],[3]

2. DOWNLOADING AND INSTALLING NASM

2.1. Downloading and Installing NASM in Linux

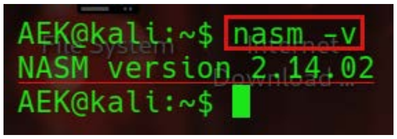
First, check the architecture of the CPU to make sure of its compatibility with this tutorial or not. That can be done by using the **"lscpu"** command on the Linux terminal. The first line that shown contains the architectural type. [7]



```
AEK@kali:~/Desktop/test$ lscpu
Architecture:                x86_64
CPU op-mode(s):              32-bit, 64-bit
Byte Order:                  Little Endian
```

2.2 check CPU architecture by Linux terminal

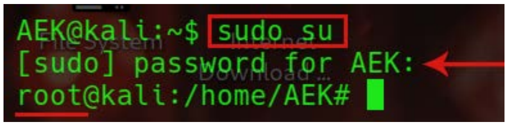
Second, write the **"nasm -v"** command, the option **"-v"** is to check if NASM is already installed or not by displaying either an error or NAMS's version edition. [3]



```
AEK@kali:~$ nasm -v
NASM version 2.14.02
AEK@kali:~$ █
```

2.1 check NASM version and its existing

Third, change to the *root mode (kernel mode)* by using the **"sudo su"** command on the terminal and then enter the password of the current user. After that, use the **"apt-get update; apt-get install nasm"** commands, firstly it is going to update the apt-get package to make sure not facing any issues by the **"apt-get update"** command and then download and install NASM simultaneously by



```
AEK@kali:~$ sudo su
[sudo] password for AEK:
root@kali:/home/AEK# █
```

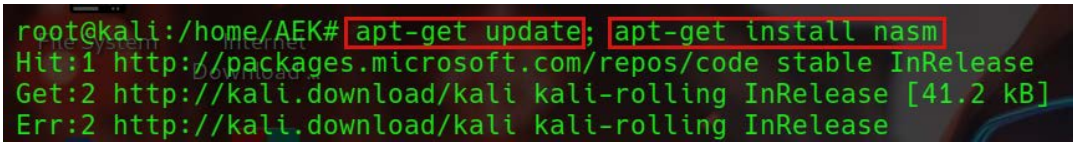
2.3 change to root mode

[1] NASM stands for Netwide Assembler (https://www.nasm.us).

[2] MASM stands for Microsoft Macro Assembler

`

the **“apt-get install nasm”** command. **NOTICE** that **“;”** is used to separate commands, also usually NASM is already installed on Kali Linux. After finishing all of that, turn back to *user mode* with the **“exit”** command. [6],[7]



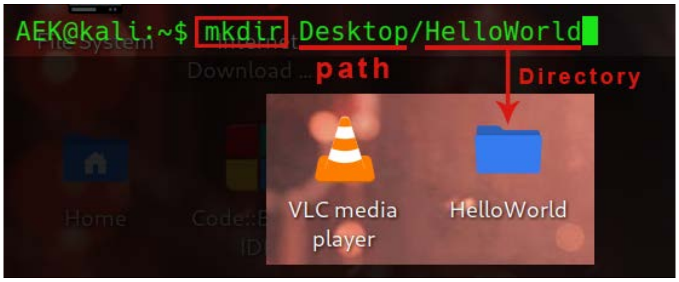
```
root@kali:/home/AEK# apt-get update; apt-get install nasm
Hit:1 http://packages.microsoft.com/repos/code stable InRelease
Get:2 http://kali.download/kali kali-rolling InRelease [41.2 kB]
Err:2 http://kali.download/kali kali-rolling InRelease
```

2.4 update apt-get package and install NASM

3. WRITING “Hello World!” PROGRAM

3.1. Setting up Project.

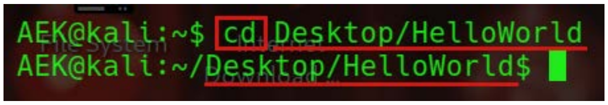
First, create a directory (folder) to keep the work organized with the code, intermediate file (*Object File*) and *Executive File* on the same path. To make this use the **“mkdir”** command followed by the path where the directory will be placed with the name of the directory. **NOTICE** the *space* between the **“mkdir”** command and the path, and **“/”** between the path and the directory. [6]



```
AEK@kali:~$ mkdir Desktop/HelloWorld
```

3.1 create a directory on Linux terminal

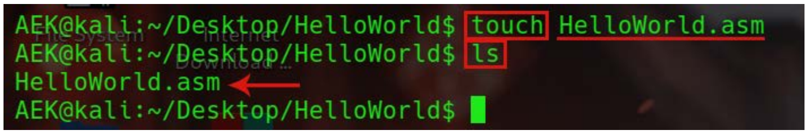
Second, change the terminal path from */home* into the new directory in order to make the work easier by **“cd”** command. [6]



```
AEK@kali:~$ cd Desktop/HelloWorld
AEK@kali:~/Desktop/HelloWorld$
```

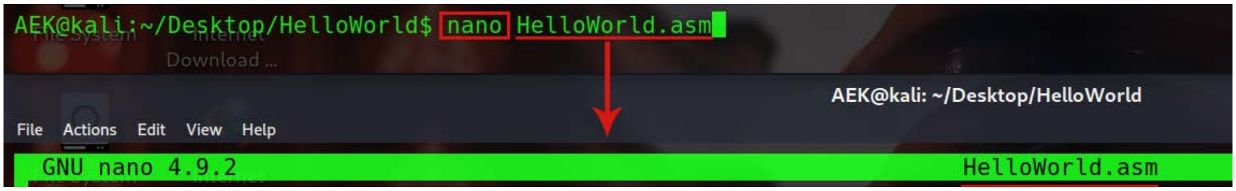
3.2 change a directory on Linux terminal

Third, create the Assembly File for coding by the **“touch”** command with the file name and (.asm) extension. After that, open the file using any text editor or IDE[3], in this tutorial *Nano Editor* (popular text editor on Linux) is used. **NOTICE** that **“ls”** command used to show a list of current files and directories on a specific or the current path. [6]



```
AEK@kali:~/Desktop/HelloWorld$ touch HelloWorld.asm
AEK@kali:~/Desktop/HelloWorld$ ls
HelloWorld.asm
AEK@kali:~/Desktop/HelloWorld$
```

3.3 create assembly file and show current directory contains by Linux terminal



```
AEK@kali:~/Desktop/HelloWorld$ nano HelloWorld.asm
```

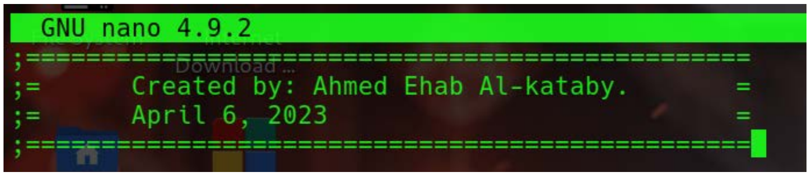
```
  GNU nano 4.9.2                                              HelloWorld.asm
```

3.4 open an assembly file with Nano editor

[3] IDE stands for Integrated Development Environment

`

3.2. Comments



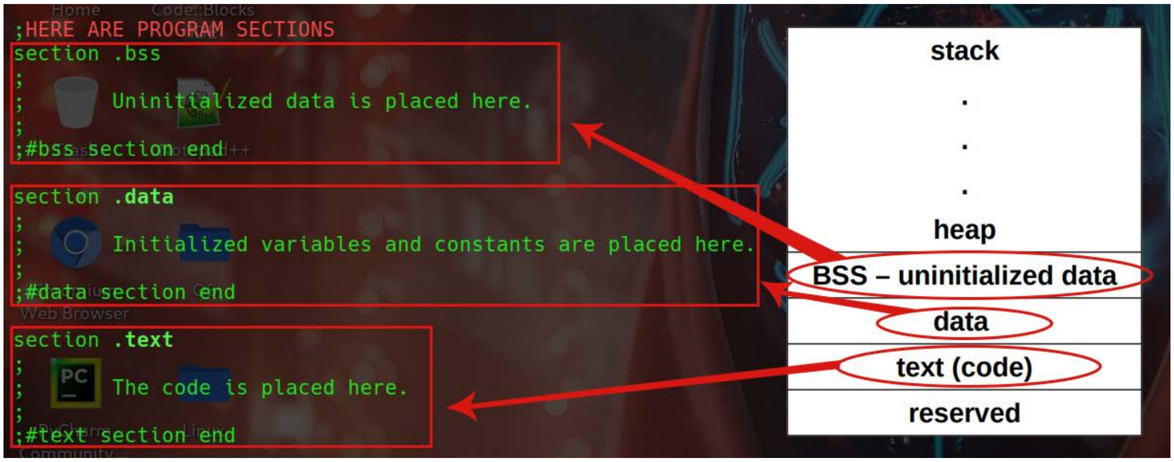
```
 GNU nano 4.9.2
;==================================================
;=        Created by: Ahmed Ehab Al-kataby.       =
;=        April 6, 2023                           =
;==================================================
```

3.5 make comments in Assembly language

“;” symbol used to make single-line (stand-alone) comments in Assembly language. It can be placed anywhere within the code including steps explanation, code sections and after instructions. NASM assembler ignores what comes after **“;”** until the line end. [5],[8]

3.3. Sections

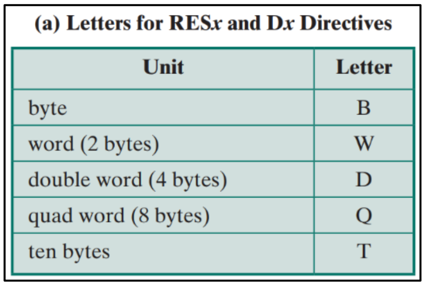
Assembly program is divided into sections which each has its missions to imply. Sections refer to a specific segment in main memory including data, code (text), bss, etc. A space is required to place between **“section”** instruction and aimed *segment* that refers to main memory (RAM). Uninitialized data is declared in **“section .bss”**; however, Initialized variables and constants are declared is declared in **“section .data”**. The code is written in **“section .text”** also called code section, each line represents just one instruction. [5]



```
;HERE ARE PROGRAM SECTIONS
section .bss
;
;    Uninitialized data is placed here.
;
;#bss section end
section .data
;
;    Initialized variables and constants are placed here.
;
;#data section end
section .text
;
;    The code is placed here.
;
;#text section end
```

3.6 sections syntax in NASM. Contains form [5]

3.4. Variables and Constants



(a) Letters for RES*x* and D*x* Directives

Unit	Letter
byte	B
word (2 bytes)	W
double word (4 bytes)	D
quad word (8 bytes)	Q
ten bytes	T

3.7 variable directives in NASM. Source [9]

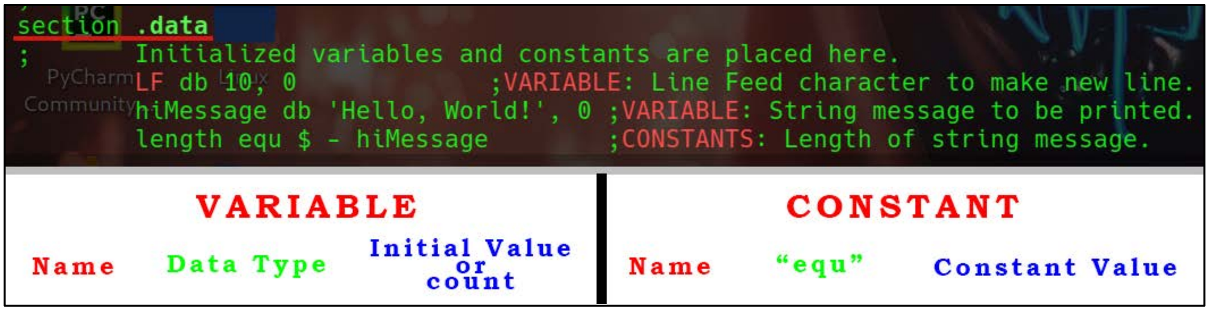
Variables in Assembly language are two types, *initialized* and *uninitialized*. Initialized variables are defined in **“.data section”** whereas uninitialized variables are in **“.bss section”**. A variable name must start with a letter, followed by letters or numbers with the possibility of including some special characters, such as **“_”**. Initialized variable is defined by **“d”** directive (data type) whereas uninitialized variable is defined by **“res”** directive. The directive must be associated with size of each individual item which can be byte, word, double word, …etc. String is a sequence of characters, all in a row in memory which last character have to be

`

"NUL (0 in ASCII[4])". Also, either single quote **(')** or double quote **(")** characters may be used to delineate a string. [1],[5],[9]

Constants in Assembly language are defined with **"equ"** followed by a value that cannot be changed during program execution and they are substituted for their values during the assembly process. **"$"** returns the current address for the line it appears on. [2],[5]

The tutorial's code has two variables and one constant in **".data section"**. **"LF"** variable is one-byte size to store the ASCII (10) to be used later for printing a new line. **"hiMessage"** is a string of one byte characters that contains the message to be printed. **"length"** constant stores the size of **"hiMessage"** by subtracing the current address from the first address of the string. [9]

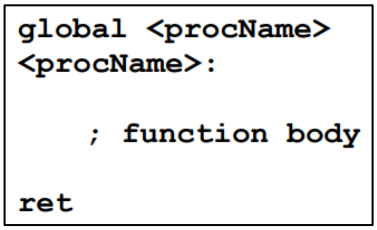


```
section .data
;       Initialized variables and constants are placed here.
        LF db 10, 0               ;VARIABLE: Line Feed character to make new line.
        hiMessage db 'Hello, World!', 0 ;VARIABLE: String message to be printed.
        length equ $ - hiMessage        ;CONSTANTS: Length of string message.
```

VARIABLE			CONSTANT		
Name	Data Type	Initial Value or count	Name	"equ"	Constant Value

3.8 data in the "Hello, World!" code and structure of data statements

3.5. Procedures and Entry Points



```
global <procName>
<procName>:

    ; function body

ret
```

3.9 procedure structure in NASM. Source [5]

Procedures in Assembly language are like void functions in C language. They are placed in **".text section"**, and used to split the code into small parts in order to make it easier for coding, debugging and maintaining. Procedures are declared by using **"global"** directive to be accessed from external files. Most procedures have **"ret"** instruction which pops the address from the top of the stack and transfers control to the address. The **"call"** instruction is used to call the procedure by its label in any place of code that happens by pushing the address *(return address)* of the instruction following the call onto the stack and to transfer control to the address associated with the procedural label. [5],[8]

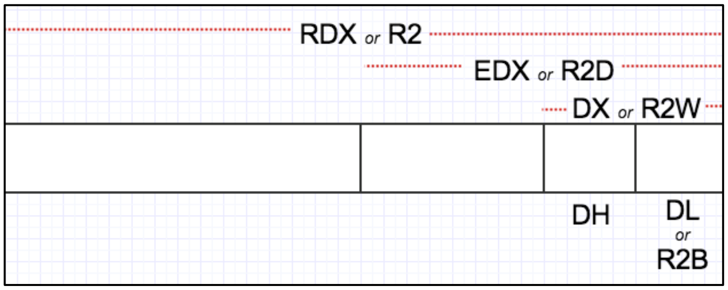
The tutorial's code has two procedures **"exit"** and **"_start"**. **"_start"** is the basic *entry point* of the program (Linux program), so Ld linker will know this label by default when the program is linking. **"exit"** is a procedure to end up the program by calling the kernel by using *80h interrupt* to answer the request in **"rax"** (64-bit register)[5] which in this case **"1"** (sys_exit) (Linux system calls)[6]. In addition, **"rdx"** register is used to return 0 to the system in successfully ending. After that, just call **"exit"** in **"_start"**. [8],[10]

[4] ASCII stands for American Standard Code for Information Interchange

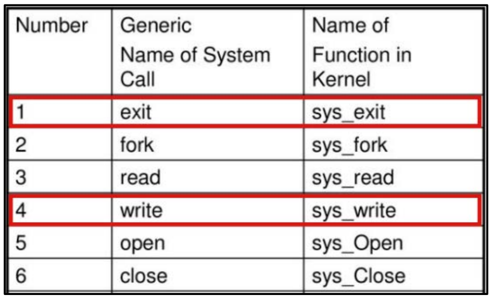
[5] 3.10 picture from "Loyola Marymount University NASM Tutorial (https://cs.lmu.edu/~ray/notes/nasmtutorial)

[6] 3.11 picture from *"Chapter 2: Operating System Structures from Bilkent University (https://slideplayer.com/slide/16415093), slide 12"*

`

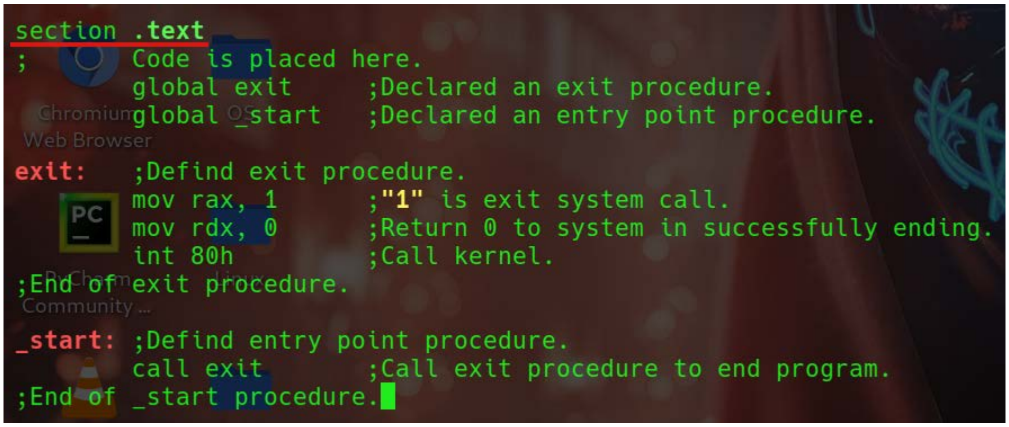


3.10 x-bit registers. Source 5



Number	Generic Name of System Call	Name of Function in Kernel
1	exit	sys_exit
2	fork	sys_fork
3	read	sys_read
4	write	sys_write
5	open	sys_Open
6	close	sys_Close

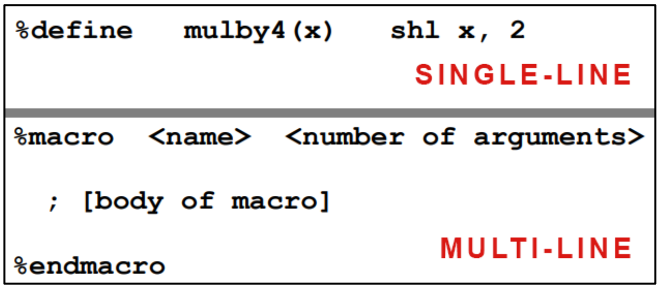
3.11 some Linux system calls. Source 6



```
section .text
;       Code is placed here.
        global exit     ;Declared an exit procedure.
        global _start   ;Declared an entry point procedure.

exit:   ;Defind exit procedure.
        mov rax, 1      ;"1" is exit system call.
        mov rdx, 0      ;Return 0 to system in successfully ending.
        int 80h         ;Call kernel.
;End of exit procedure.

_start: ;Defind entry point procedure.
        call exit       ;Call exit procedure to end program.
;End of _start procedure.
```

3.12 exit procedure and _start entry point in the “Hello, World!” code

3.6. Macros



```
%define   mulby4(x)   shl x, 2
```

SINGLE-LINE

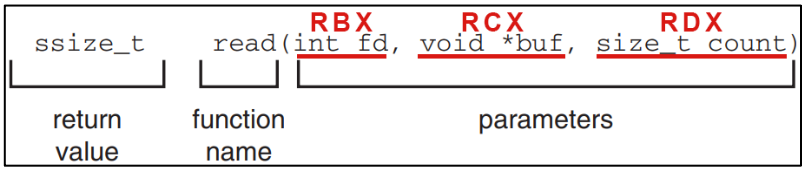
```
%macro  <name>  <number of arguments>

  ; [body of macro]

%endmacro
```

MULTI-LINE

3.13 single-line and multi-line macros forms. Source [5]

Macros in Assembly language are two types single-line and multi-line. Single-line macros are defined using the **“%define”** directive which preprocessor directives are started with **“%”** (like **“#”** in C and C++ languages). Multi-line macros are defined using the **“%macro”** and ended with **“%endmacro”**, **“%macro”** is followed by the macro name then the numbers of parameters. To reach parameters in macro use “%” with the parameter index in ascending order. For example, in tutorial’s code **“%2”** is used to pass second parameter into **“print”** macro. **NOTICE** that 0 after macro name means there is **NO** parameters. In order to use the macro, write its name in **“.text section”** followed by arguments *in order*, use **“,”** to separate between arguments. [2],[5]

The tutorial’s code has two macros **“print”** and **“newLine”**. **“print”** is used to print “Hello, World!” on screen whereas **“newLine”** is used to make a new line. Parameters of **“read()”** (function that is available in Linux) are **“rbx, rcx, rdx”** in order, **“rbx”** is the file descriptor to be read, **“rcx”** is a buffer where the data will

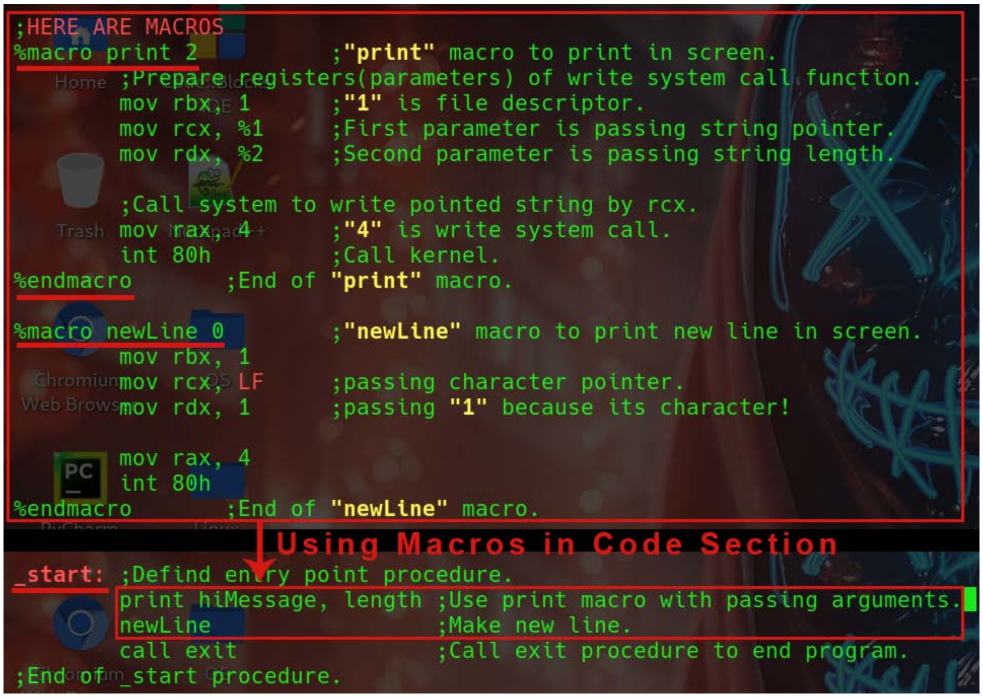


```
ssize_t    read(int fd, void *buf, size_t count)
```

3.14 Linux read() function signature. Source [10]

`

be read into, and **“rdx”** is the number of bytes to be read into the buffer. **“rbx”** is passed with 1, whereas **“rcx”** with a pointer to the message, and **“rdx”** holds the message length. After that, call kernel by using *80h interrupt* to answer the request in **“rax”** which in this case **“4”** (sys_write). However, **“newLine”** macro is the same as **“print”** macro the only change is the parameters are already known, **“rcx”** is passed with **“LF”**, while **“rdx”** is passed with 1 because **“LF”** points to a single character. The final step is to use the macros under **“_start”** label. [1],[10]



```
;HERE ARE MACROS
%macro print 2             ;"print" macro to print in screen.
        ;Prepare registers(parameters) of write system call function.
        mov rbx, 1         ;"1" is file descriptor.
        mov rcx, %1        ;First parameter is passing string pointer.
        mov rdx, %2        ;Second parameter is passing string length.

        ;Call system to write pointed string by rcx.
        mov rax, 4         ;"4" is write system call.
        int 80h            ;Call kernel.
%endmacro          ;End of "print" macro.

%macro newLine 0           ;"newLine" macro to print new line in screen.
        mov rbx, 1
        mov rcx, LF        ;passing character pointer.
        mov rdx, 1         ;passing "1" because its character!

        mov rax, 4
        int 80h
%endmacro          ;End of "newLine" macro.
```

```
_start: ;Defind entry point procedure.
        print hiMessage, length ;Use print macro with passing arguments.
        newLine                 ;Make new line.
        call exit               ;Call exit procedure to end program.
;End of _start procedure.
```

3.15 “print” and “newline” macros, and using them in the “Hello, World!” code

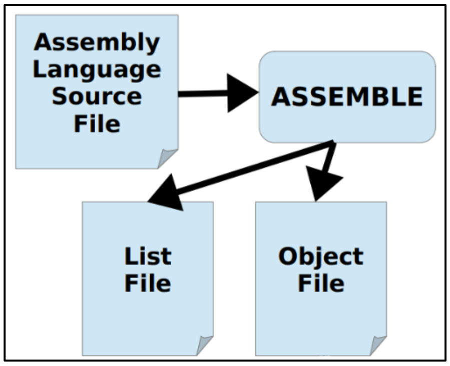
3.7. Conclusion

Now, the code is done! press **“Ctrl + x”** to save the code and exit from *Nano editor*. The **“Hello, World!”** code is available on *AEKStudio GitHub Profile*[7].

4. ASSEMBLING PROCESS AND RECOMMENDED NASM OPTIONS

4.1. Assembling Process

Basically, *Assembling process* is done by *an assembler* which is a utility software that takes an assembly code file **(.asm)** as input and creates an object file (binary file) **(.o)** as output. The assembler views this file as a memory block starting at location of 0. While the assembler reads lines of source code, it will build a symbol table for summarizing items’ names, and if an error occurs, it will display the misunderstood line followed by an error message. [1],[5],[9]



4.1 assembling process. Source [5]

[7] Ahmed Ehab Al-kataby’s “Hello, World!” program in Github (https://github.com/AEKStudio2019/Hello-World).

`

4.2. Assembling "Hello, World!" Code

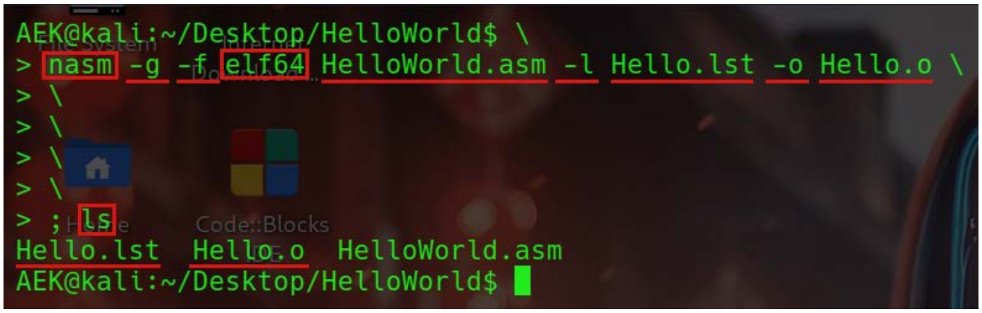
To assemble **"HelloWorld.asm"** code, use *NASM assembler* by writing on Linux terminal **"nasm"** command followed by needed options (options are case-sensitive). For example, **"-g"** option is used to enable debugging information in the output object file, It will increase the object file size, but it is necessary to allow effective debugging; however, it is optional. **NOTICE** that **"-h"** option gets further usage instructions for typing. [3],[5]

"-f" option is used to specify the output file format by mentioning the file format after it, in Linux with x86_64 architecture use **"elf64"** format and then write the assembly file name with **(.asm)** extension **"HelloWorld.asm"**, without using this option, the default is **"bin"** format. [3],[5]

"-l" option creates a listing file of a given name, so write a name for the list file with **(.lst)** extension **"Hello.lst"**. the list file is optional to be created, it shows how the code was assembled, such as the line numbers, the relative addresses, the machine language version of instructions (including variable references) and the source code lines, besides it is very useful while using debugger. [2],[5]

"-o" option identifies the output object file name followed by **(.o)** extension. In this tutorial, the output file is named **"Hello.o"**. However, It is optional, so by default, the output file has the same name as the source assembly file in this case the output file would be like **"HelloWorld.o"**. [3]

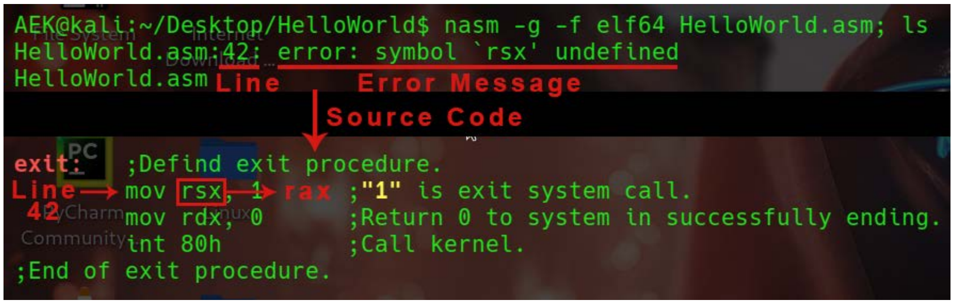
To see if **"Hello.lst"** (list file) and **"Hello.o"** (object file of the source code) are made in the current path successfully, just use **"ls"** command to show them. [6]



```
AEK@kali:~/Desktop/HelloWorld$ \
> nasm -g -f elf64 HelloWorld.asm -l Hello.lst -o Hello.o \
> \
> \
> \
> \
> ; ls
Hello.lst   Hello.o   HelloWorld.asm
AEK@kali:~/Desktop/HelloWorld$ 
```

4.2 assembling the "Hello, World!" code by NASM

If there is an error, it will be shown on the terminal with specifying the misunderstood line. For instance, replacing **"rax"** register in *line 42* with **"rsx"** word which does not mean anything in Assembly language, that is resulting *an undefined error* in *line 42*, and the output file has not been created. [1]



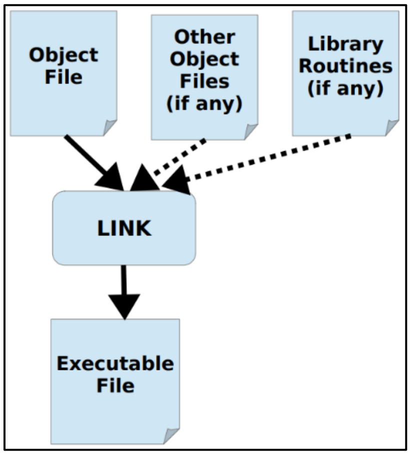
```
AEK@kali:~/Desktop/HelloWorld$ nasm -g -f elf64 HelloWorld.asm; ls
HelloWorld.asm:42: error: symbol `rsx' undefined
HelloWorld.asm

exit:     ;Defind exit procedure.
          mov rsx, 1     ;"1" is exit system call.
          mov rdx, 0     ;Return 0 to system in successfully ending.
          int 80h        ;Call kernel.
;End of exit procedure.
```

4.3 NASM undefined symbol error while assembling process

`

5. LINKING PROCESS

5.1. Linking Process



5.1 linking process. Source [5]

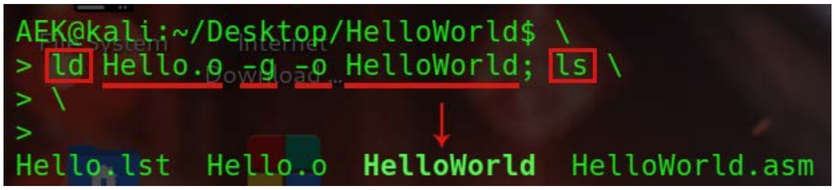
Generally, the *linking process* is the process of combining the machine code and data in object files (a collection of one or more) and library files together to create a single executable file, this process is done by the *linker* (sometimes referred to as linkage editor). *Dynamic linking* is supported on Linux OS, which allows for postponing the resolution of some symbols until a program is executed. Plenty of instructions are not in executable file, but in dynamic libraries to be resolved and accessed at run-time. On Linux, the extension of *dynamic libraries* is **(.so)**. While linking file process, the linker might face some issues, so it will have a fatal error message and the executable file will not be generated. [1],[2],[5],[9]

5.2. Linking "Hello.o" File

To link the **"Hello.o"** object file, use the *Ld linker* by writing on Linux terminal **"ld"** command followed by the **"Hello.o"** object file name and needed options (options are case-sensitive). [5]

"-g" option enables debugging information in the executable file, it will increase the executable file size, but it is necessary to allow effective debugging; however it is optional. [5]

"-o" option creates the executable file name (there is no specification for the extension). It is optional, so by default the output file is named as **"a.out"**. In this tutorial, the output execution file is named as **"HelloWorld"**, so it came after **"-o"**. [5]

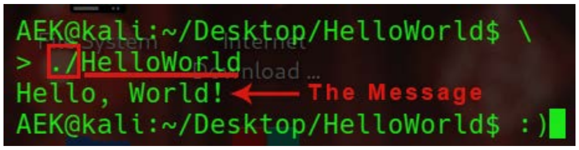


```
AEK@kali:~/Desktop/HelloWorld$ \
> ld Hello.o -g -o HelloWorld; ls \
> \
>
Hello.lst  Hello.o  HelloWorld  HelloWorld.asm
```

5.2 linking the "Hello.o" file by using ld linker

6. EXECUTING PROCESS

6.1. Executing "HelloWorld" Program



```
AEK@kali:~/Desktop/HelloWorld$ \
> ./HelloWorld
Hello, World!
AEK@kali:~/Desktop/HelloWorld$ :)
```

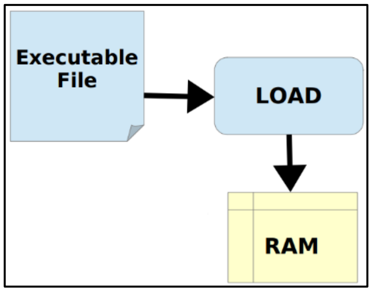
6.1 executing "HelloWorld" program in terminal

In the same path of the executable file, write **"./"** command associated with the executable file name which in this tutorial is "HelloWorld" then press **"Enter"** to run the program. Finally, the terminal will display the **"Hello, World!"** message. [6],[5]

`

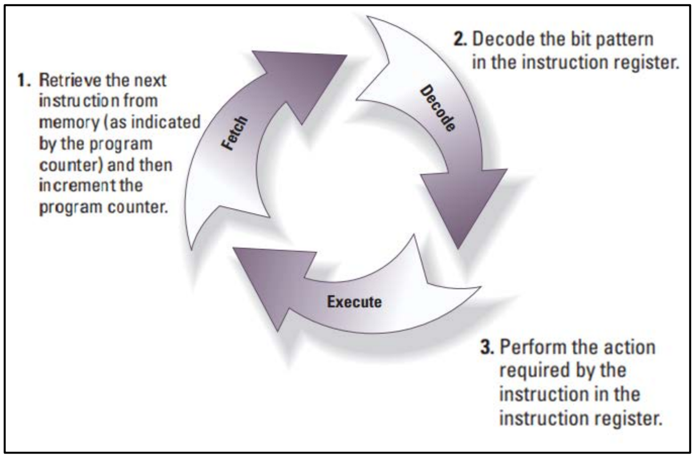
6.2. Executing Process

Loading process is a process that is done before the executing process. This process is done by the *loader software* which is responsible for creating a new process in memory, and loading from the hard disk each segment of the executable file to its place in main memory (RAM). The operating system scheduler[8] will make the decisions about which and when the process is executed. [5]



6.2 loading process. Source [5]

After the loading process, the executing process will be started. The CPU performs this process through a three-stage process known as the *machine cycle*, the stages are fetching, decoding then executing. In *fetching*, the CPU copies the next instruction (already in program counter register) from RAM into instruction register and then increments the program counter to be ready for the next instruction. **REMEMBER** that program counter and instruction register are special-purpose registers. In *decoding*, the CPU resolves the instruction by breaking the operand field into its components based on the op-code. In *executing*, the CPU performs the required action on a specific circuity by the decoded instruction inside instruction register. [4]


1. Retrieve the next instruction from memory (as indicated by the program counter) and then increment the program counter.
2. Decode the bit pattern in the instruction register.
3. Perform the action required by the instruction in the instruction register.
Fetch
Decode
Execute

6.3 machine cycle. Source [4]

7. References

[1] Assembly Language Step-by-Step: Programming with Linux 3rd Edition by Jeff Duntemann, 2009, (pages xxxi, 131, 132, 136, 139, 243).

[2] PC Assembly Language by Paul A. Carter, 2019, (page vi) and (sections 1.3.5, 1.4.5, 1.4.6, 4.2).

[3] The NASM Manual version 2.16.01 (NASM - The Netwide Assembler), (sections 1.1, 2.1, 2.1.1, 2.1.2, 2.1.27).

[4] Computer Science: An Overview 13th edition by Glenn Brookshear and Dennis Brylow, 2020, (section 2.3).

[5] x86-64 Assembly Language Programming with Ubuntu by Dr. Ed Jorgensen, 2022, (secions 2.5, 4.1, 4.3 - 4.6, 5.1, 5.2.1, 5.2.2, 5.3, 5.3.3, 5.5, 11.1 - 11.3, 12.0, 12.4) and and (illustrations 4, 6).

[6] The Linux Command Line Fifth Internet Edition by William Shotts, 2019, (sections 1.1, 1.2, 1.4, 2.11, 3.14, 3.17, 4.24).

[7] The Kali Linux Website, (util-linux | Kali Linux Tools), (Enabling Root | Kali Linux Documentation).

[8] Introduction to 64 Bit Intel Assembly Language Programming for Linux by Ray Seyfarth, 2011, (sections 1.4, 9.2, 9.3).

[9] Computer Organization And Architecture 10th Edition by William Stallings, 2016, (sections B.1 - B.3) and (table B.2).

[10] Operating System Concepts 9th Edition by Abraham Silberschatz, Peter B. Galvin and Greg Gagne, 2013, (section 2.3).

[8] For more information about CPU scheduling, read “Chapter 6 from Operation System Concepts: 9th Edition”