AFRAME DOCUMENTATION(official tutorial?)

I’m working on the rest, so yes the pointers lesson is still not done.

Make sure that

1. you use Linux lol
2. your system is x86_64 or “64-bit”
3. you read this

Installing software

to install the necessary software (nasm, gcc linker, git) on debian-based distros like ubuntu, Linux Mint, pop_os! Run this command:

```
sudo apt install -y nasm gcc git
```

if you use rhel what are you doing? And if you use Arch, you should know how to install all of that.

Installing Aframe

when that is done, run this:

```
git clone https://github.com/umikali222/aframe
```

then you can go into the directory with

```
cd aframe
```

then, to install aframe to your computer, run:

```
sudo cp aframe.py /bin/aframe
```

than to see if it runs properly, run

```
aframe hw.py
```

If that returns Hello, World! Then everything is installed correctly.

storing static values

To define a static value in aframe you need to start the .data section like so:

```
section .data
```

ant to actually define a value you put the name, size, and value like so:

```
section  .data
    funnyNumber db 69
```

Here we store only a byte, but you can store more by replacing db with dw, dd, dq which store 2 bytes, 4 bytes, and 8 bytes respectively.
Note: you need to put all this before _start, for the program to not throw any errors.

You can also store more than one number of the same size under a different name like so:

```
section .data
    funnyNumbers db 69, 420, 17
```

now, can you figure out why this program won't work? If you can, congrats, but if you can't, the problem is that the number 420 is too big for a byte, because a byte can store numbers from 1 to 255, if you want this to work , you will need to replace db with dw, but if you want to store ASCII, which we will need to do, you can just stick to db. Now ASCII text automatically gets converted to decimal, so we can easily store ASCII text, to start our hello world program we will add this at the start of an empty file:

```
section .data
    text db "Hello, World!",10
```

the reason for the ,10 at the end is that 10 in ASCII is a newline, so we don't want to print
"Hello, Worldumikali@komputeras-informejtika:~/aframe$"

Linux syscalls

Syscalls (or system calls) is what the kernel is for. It takes some values which are stored the CPU, and the kernel executes them like an instruction you can find system call tables online, which show you what values to store, and where to store them. So where can you store those values? In registers. You can think of them as short-term memory , the things you are thinking about. The basic registers are: rax, rbx, rcx, rdx. These are the absolute most basic registers, but I don't recommend using them, because aframe uses them constantly, but for syscalls it's just fine. There are a few more registers, that I still don't recommend using for anything except syscalls, like: rdi, rsi … . Now to make your 1st syscall you will need to start executing instructions.

Hello World program

Come back to the program you made in the last lesson, and at the end add:

```
_start:

```

that will be the start of your program. Now to move the values into the registers you will need the mov instruction, which you will add to your program like so:

```
_start:
    mov rax, 1
```

Great, this program will put the number 1 into the register rax this, for the kernel means sys_write, now you want specify stdout like so:

```
_start
    mov rax, 1
    mov rdi, 1
```

for now we only specified the instruction, so it won't do anything, we can also add the text that we're trying to print like so:

```
_start
    mov rax, 1
    mov rdi, 1
    mov rsi, text
```

but this still won't work because we need to add 2 more commands, which are:

```
mov rdx, 14
syscall
```

the 1st command tells the kernel how long the text is, there the ,10 also counts. Why it needs to do this, You will understand after the pointers lesson, but I don't recommend skipping ahead. Now this program will work, but not fully, because the program still crashes, because we never told the kernel the program has stopped. But we can do that with just 3 simple commands

```
mov rax, 60
mov rdi, 0
syscall
```

where 60 in rax tells the kernel we're exiting the program, and 0 in rdi tells the kernel that everything went ok, and that nothing crashed. Now here's what the program should look like:

```
section .data
    text db "Hello, World!",10

_start:
    mov rax, 1
    mov rdi, 1
    mov rsi, text
    mov rdx, 14
    syscall

    mov rax, 60
    mov rdi, 0
    syscall
```

btw, running syscall doesn’t reset the registers, so you can run syscall multiple times, and it will print the string multiple times.