First Things First: The Basics

Before start creating 539kernel, we need to learn some basics, let's start with practical aspects and get our hands dirty!

Computer Architecture

[MQH] CPU and Main Memory (The program should be here to be executed), the program counter

x86 Assembly Language Overview

To build a boot loader, we need to use assembly language. The program that takes a source code which is written in assembly language and transforms this code to the machine language is known as *assembler* [1]. There are many assemblers available for x86 but the one that we are going to use is Netwide Assembler (NASM). However, the concepts of x86 assembly are the same, they are tight to the architecture itself, also the instructions are the same, so if you grasp the basics it will be easy to use any other assembler [2]. Don't forget that the assembler is just a tool that helps us to generate an executable x86 machine code out of an assembly code.

In this section I don't aim to examine the details of x86 and NASM, you can consider this section as a quick start on both x86 and NASM, the basics will be presented to make you familiar with x86 assembly language, more advanced concepts will be presented later when we need them. If you are interested in x86 assembly for its own sake, there are multiple online resources and books that explain it in details.

Registers

In any CPU architecture, and x86 is not an exception, a register is a small memory inside the CPU chip. Like any other type of memories (e.g. RAM), we can store data inside a register and we can read data from it, it is too small and too fast.

The CPU's architecture provides us with a number of registers, and in x86 there are two types of registers: general purpose registers and special purpose registers. In general purpose registers we can store any kind of data we want, while the special purpose registers are provided by the architecture for some

[1] While the program that transforms the source code which is written in high-level language such as C to machine code is known as *compiler*.

[2] Another popular open-source assembler is GNU Assembler (GAS). One of main differences between NASM and GAS that the first uses Intel's syntax while the second uses AT&T syntax.

specific purposes, we will see the second type latter in our journey of creating 539kernel.

x86 provides us with eight general purpose registers and we refer to them by their names in assembly code. The names of these registers are: `EAX`, `EBX`, `ECX`, `EDX`, `ESI`, `EDI`, `EBP`, and `ESP`. The size of these registers is 32-bit (4 bytes) and due to that, they are available only on 32-bit x86 CPUs [3] such as Pentium 4 for instance, that is, you can't use the register EAX (for example) in Intel 8086 because it is a 16-bit CPU and not 32-bit.

The registers ESI, EDI, EBP and ESP are considered as general purpose in x86 architecture, but we will see latter that they store some important data in some cases and it's better to use them carefully if we are forced to.

These registers are 32-bit registers as we have mentioned before, but we can use only the first 16-bit of them by referring to them by their old names which have been used in 16-bit x86 CPUs. These 16-bit names are AX, BX, CX and DX for respectively EAX, EBX, ECX, DX. In old days, when 16-bit x86 CPUs were dominant, assembly programmers used a register called AX (for instance) which has the size 16-bit, but when 32-bit x86 CPUs came, this register (and the others of course) has been *extended* to have the size 32-bit and its name changed to EAX, though, the old register name AX is also available (at least for compatibility reasons) in 32-bit CPUs with the same old size, but now, it is a part of the bigger register EAX. And that's holds for all other general purpose registers.

Furthermore, the 16-bit registers can be divided into two 8-bit parts, the first 8 bits of the register are called the *low* bits, while the second 8 bits are called the *high* bits. For example, AX register is a 16-bit register, and it is a part of 32-bit EAX register. AX is divided into two more parts, A**L** for the **l**ow 8 bits and A**H** for the **h**igh 8 bits. And the same concept holds for BX, CX and DX.

Instruction Set

The assembly language is really simple. There is a bunch of instructions provided by the CPU for the programmer, each instruction performs something, and it may takes operands which the instruction works on it somehow, depending on the instruction, the operands can be a static values (e.g. a number), a register or a memory location. An assembly code is simply a sequence of those instructions which executed sequentially. For example, the following is an assembly code, don't worry about the meaning now, you will understand what it does eventually:

```
mov ah, 0Eh
mov al, 's'
int 10h
```

[3] Also they are available in **64-bit** x86 CPUs such as Core i7 for instance.

You can see that each line performs an instruction, and depending on the instruction there will be one or two operands.

We can say an instruction is same as a function [4] in a high-level languages which is provided by some third-party library (in our case the CPU) and may takes some parameters (in our case the operands) to operate on. We know that each function does something useful, but we don't necessarily know how does it work inside.

For example, x86 has an instruction called *add* which takes two operands and add the value of the first operand to the values of the second operand and stores the result in the first operand. That means, the first operand of *add* should be something that we can store values on it. Yes! you are right, the first operand in this case can be a register or a memory location and not a static value.

To clarify the previous analogy, in high-level languages world, let's say in C programming language, *add* instruction will be a function which takes two parameters (operands), adds the two values and returns the result to the caller [5].

If you are interested on the available instructions on x86, there is a four-volumes manual named "Intel® 64 and IA-32 architectures software developer's manual" provided by Intel that explains each instruction in details [6].

Assigning Values with "mov"

You can imagine a register same as a variable in high-level languages. We can assign values to a variable, we can change its old value and we can copy its value to another variable. In assembly language, these operations can be performed by the instruction *mov* which takes the value of the second operand and stores it in the first operand.

You have seen in the previous examples the following two lines that use *mov* instruction:

```
mov ah, 0Eh
mov al, 's'
```

Now you know that the first instruction copy the value *0Eh* to the register *ah*, and the second instruction copy the character *s* to the register *al*. Also you can see that the value *0Eh* ended with the character *h*, for NASM that means *0E* is a number in **h**exadecimal numbering system which is 14 in the decimal numbering system.

The instruction *mov* is one of most important instructions that we will use in our work. Of course, we are going to use more instructions to accomplish our goal,

[4] Or a procedure for people who work with Algol-like programming languages.
[5] More accurately, stores the result in the same location of the first parameter.
[6] https://software.intel.com/en-us/articles/intel-sdm

but in this quick overview we will stop here. We have learned enough basics to proceed and the more advanced concepts will be examined in its right place.

NASM

Netwide Assembler (NASM) is an open-source source assembler for x86 architecture which uses Intel's synatx of assembly language. We can use it through command line to assemble x86 assembly code and generate the corresponding machine code. The basic use of NASM command is:

```
nasm -f <format> <filename> [-o <output>]
```

The argument *format* decides the binary format of the generated machine code. There are multiple binary format available, each operating system uses a specific binary format for its executable (and object) files, for example, Linux uses a format called Executable and Linkable Format (ELF) binary format, and there are many different binary format there such as: Mach-O which is used by Mach-based [7] and Portable Executable (PE) which is used by Microsoft Windows.

A binary format is basically a specification which give a blueprint of how a binary file is organized and what are the features are available, that is, it describes how a binary file is structured, the generated machine code is one part of a binary format. Note that each executable file uses some binary format no matter what is the programming language that has been used to create the software [8], for example in Linux, if we create a software either by C or assembly, the last executable result will be and ELF binary file.

Each operating system's kernel knows its own binary format well, and knows how the binary file that uses this format is structured, and how to seek the binary file to find the machine code that should be loaded into memory and executed by the CPU. For example, when you run an ELF executable file in GNU/Linux system, the Linux kernel knows it is an ELF executable file and assumes that it is organized in a specific way, by using the specification of ELF, Linux kernel will be able to locate the machine code of the software inside the ELF file and load it into memory to be ready for execution.

In any binary format, one major part of the binary file that uses this format is the machine code that has been produced by compiling or assembling some source code, the machine code is specific to a CPU architecture, for example,

[7] Mach is an operating system's kernel which is well-known for using *microkernel* design. It has been started as a research effort in Carnegie Mellon University in 1985. Current Apple's operating systems macOS and iOS are both based on an older operating system known as NeXTSTEP which uses Mach as its kernel,

[8] Of course the programming language should be a *compiled* programming language such as C and Rust and not an *interpreted* such as Python or a one that uses a virtual machine such as Java.

the machine code that has been generated for x64 [9] cannot run on x86. Because of that the binary files are distributed according to the CPU architecture which can run on, for example, GNU/Linux users see the names of software packages in the following format *nasm_2.14-1_i386.deb*, the part *i386* tells the users that the binary machine code of this package is generated for *i386* architecture, which is another name for x86 by the way, that means this package cannot be used in a machine that uses ARM CPU such as Raspberry Pi for example. Because of that, to distribute a binary file of the same software for multiple CPU's architecture, a separate binary file should be generated for each architecture, there are some binary format such as FatELF that gathers the machine code of multiple CPU's architecture in one binary file and the suitable machine code will work according to the current running CPU, this type of binary format named *fat binary*, their size will be bigger that a binary file of the same software which is oriented to one architecture, and because of that we call it *fat*.

However, if our goal of using assembly language is to produce an executable file for Linux for example, we will use *elf* as a value for *format* argument. But we are working with low-level kernel development, so our binary files should be *flat* and the value of *format* should be *bin* to generate a *flat binary* file.

The second argument is the *filename* of the assembly file that we would like to assemble, and the last option and argument are optional, we use them if we want to specify another name for the generated binary file by NASM, the default name will be same as the filename with different extension.

GNU Make

GNU Make is a build automation tool. Well, don't panic because of this fancy term! the concept behind it is too simple.

When we create a kernel of an operating system we are going to write some assembly code and due to that we are going to use an assembler (e.g. NASM), also, we are going to write C code and due ti that we are going to use a C compiler (e.g. GNU GCC), the ultimate goal of doing that is to generate the executable file of our kernel.

If you have worked previously with GNU GCC for example, you can recall that each time you need to generate the executable file, you have to write the same command in the terminal over and over again, and mention the list of all source files that you would like to compile, and sometimes you use a tool called the *linker* to link the generated object file and produce the final executable file. This process of compiling and linking is called *building* process.

In our case of creating 539kernel, not only GNU GCC is used to *build* the final executable file of our kernel, also NASM should be used to generate the object file of the portion which is written by using assembly. Furthermore, a bootable

[9] The x86 architecture that supports 64-bit.

image of the kernel should be generated. Well, you can see that the build process of our kernel is too tedious. To save our time (and ourselves from boredom of course) we don't want to write all these commands over and over again. Here where GNU Make comes to the rescue, it *automates* the *building* process.

What we should do to use GNU Make is simply create the *makefile* for our kernel. The make file is a simple text file written in a way GNU Make can understand. It tells GNU Make what to do to produce the final bootable image of our kernel, that means by using make file we tell GNU Make to compile the assembly files by using NASM, and compile the C files by using GCC, use the linker to link the output of these two steps, then create the final bootable image of our kernel. After writing the make file, what we need to do to compile our kernel is simply run the command *make* in the terminal! That's it!

Makefile

A makefile is a text file that tells GNU Make what to do, that is, in most cases, how to create an executable file from a source code. In makefile, we define a number of rules, that is, a makefile has a list of rules that defines how to create the executable file. Each rule has the following format:

```
target: prerequisites
    recipe
```

When we run the command *make* without specifying a defined target name as an argument, GNU Make is going to start with the first rule in makefile, if the first rule's target name doesn't start with dot, otherwise, the next rule will be considered. The name of a target can be a general name or filename. Assume that we defined a rule with the target *foo* and it's not the first rule in makefile, we can tell GNU Make to execute this rule by running the command *make foo*. One of well-known convention in makefiles is to define a rule with target name *clean* that deletes all object files and binaries that have been created in the building process. We will see after a short time the case where the name of a target is a filename instead of general name.

The *prerequisites* part of a rule is what we can call the dependencies, those dependencies can be either filenames (the C files of the source code for instance) or other rules in the same makefile. To run the a specific rule successfully, the dependencies of this rule should be ready, if there is another rule in the dependencies, it should be executed successfully first, if there is a filename in the dependencies and there is no rule which its target has the same name of the file, then this file will be checked and used in the recipe of the rule.

Each line in the *recipe* part should start with a tab and it contains the commands that is going to run when the rule is executed. These commands are normal Linux commands, so in this part of a rule we are going to run the C compiler for

example to compile a bunch of C source files. Any arbitrary command can be used in the recipe as we will see later when we create the makefile of 539kernel.

Consider the following C source files:

file1.c:

```
#include "file2.h"

int main()
{
    func();
}
```

file2.h

```
void func();
```

file2.c

```
#include <stdio.h>

void func()
{
    printf( "Hello World!" );
}
```

By using these three files, let's take an example of a makefile with filenames that have no rules with same target's name:

Makefile

```
build: file1.c file2.c
    gcc -o ex_file file1.c file2.c
```

The target name of this rule is *build*, and since it is the first and only rule in the makefile, that it can be executed either by the command *make* or *make build*. It depends on two C files, file1.c and file2.c, they should be available on the same directory. The the recipe tells GNU Make to compile and link these two files and generate the executable file under the name *ex_file*.

The second example of makefile that has multiple rules:

Makefile

```
build: file2.o file1.o
    gcc -o ex_file file1.o file2.o

file1.o: file1.c
    gcc -c file1.c

file2.o: file2.c file2.h
    gcc -c file2.c file2.h
```

Here, the first rule depends on the two *object files* [10] file1.o and file2.o. We know these two files aren't available in the source code directory, therefore, we have defined a rule for each one of them. The rule *file1.o* is going to generate the object file *file1.o* and it depends on *file1.c*, the object file will be simple generated by compiling *file1.c*. The same happens with *file2.o* but this rule depends on two files instead of only one.

GNU Make also supports variables which can simply be defined as the following:

```
foo = bar
```

and they can be used in the rules as the following:

```
$(foo)
```

Let's now redefine the second makefile by using the variables which is a good practice for several reasons:

Makefile

```
c_compiler = gcc
buid_dependencies = file1.o file2.o
file1_dependencies = file1.c
file2_dependencies = file2.c file2.h
bin_filename = ex_file

build: $(buid_dependencies)
    $(c_compiler) -o $(bin_filename) $(buid_dependencies)

file1.o: $(file1_dependencies)
    gcc -c $(file1_dependencies)

file2.o: $(file2_dependencies)
    gcc -c $(file2_dependencies)
```

[10] An object file is a machine code of a source file and it is generated by the compiler. The object file is not an executable file and in our case at least it is used to be linked with other object files to generate the final executable file.