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!

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] even if it uses other syntax than NASM. 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, so, any suitable assembler that we use to reach our goal will be enough.

In this section I don't aim to examine the details of x86 or 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 processor architecture, and x86 is not an exception, a register is a small memory inside the processor's chip. Like any other type of memories (e.g. RAM), we can store data inside a register and we can read data from it, the registers are too small and too fast. The processor architecture provides us with multiple registers. 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 specific purposes, we will encounter the second type latter in our journey of creating 539kernel.

x86 provides us with eight general purpose registers and to use them in order to read from or write to them 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`. While the registers `ESI`, `EDI`, `EBP` and `ESP` are considered as general purpose

[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.

registers in x86 architecture [3], 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. The size of each one of these general purpose registers is `32` bits (`4` bytes) and due to that, they are available only on x86 processors that supports `32-bit` architecture [4] such as Pentium 4 for instance. These `32-bit` registers are not available on x86 processors that support only `16-bit` architecture or lower, so, for example, you can't use the register `EAX` in Intel 8086 because it is a `16-bit` x86 processor and not `32-bit`.

In old days, when `16-bit` x86 processors were dominant, assembly programmers used the registers `AX`, `BX`, `CX` and `DX` and each one of them is of size `16` bits (`2` bytes), but when `32-bit` x86 processors came, these registers have been extended to have the size `32-bit` and their names were changed to `EAX`, `EBX`, `ECX` and `EDX`. In fact, the first letter `E` of the new names means *extended*. However, the old names are still usable in `32-bit` x86 processors and they are used to access and manipulate the first `16` bits of the corresponding register, for instance, to access the first `16` bits of the register `EAX`, the name `AX` can be used. Furthermore, the first `16` bits of these registers can be divided into two parts and each one of them is of size`8` bits (`1` bytes) and has it's one name that can be referred to in the assembly code. The first `8` bits of the register are called the *low* bits, while the second `8` bits are called the *high* bits. Let's take one of these register as an example:`AX` register is a `16-bit` register which is a part of the bigger `32-bit` `EAX` register in `32-bit` architecture. `AX` [5] is divided into two more parts, `AL` for the **l**ow `8` bits as the second letter indicates and `AH` for the **h**igh `8` bits as the second letter indicates. And the same holds for `BX`, `CX` and `DX`.

Instruction Set

The processor's architecture provides the programmer with a bunch of *instructions* that can be used in assembly code. Processor's instructions resemble functions [6] in a high-level languages which are provided by the libraries, in our case, we can consider the processor as the ultimate library for the assembly code. As with functions in high-level programming languages, each instruction has a name and performs a specific job, also, it can take parameters which are called *operands*. Depending on the instruction itself, the operands can be a static value (e.g. a number), a register name that the instruction is going to fetch the stored value of it to be used or even a memory location.

The assembly language is really simple. An assembly code is simply a sequence of instructions which will be executed sequentially. The following is an assembly code, don't worry about its functionality right now, you will understand what it does eventually.

[3] According to Intel's manual

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

[5] Or in other words for `32-bit` architecture: The first `16` bits of `EAX`.

[6] Or a procedure for people who work with Algol-like programming languages.

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

As you can see that each line starts with an instruction which is provided to us by x86 architecture, in the first two lines use an instruction named `mov` and as you can see, this instruction receives two operand. In the current usage of this instruction we can see that the first operand of it is a register name which the second operand is a static value. The third line uses another instruction named `int` which receives one operand. When this code is running, it will be executed by the processor sequentially, starting from the first line until it finishes in the last line.

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 [7].

Assigning Values with `mov`

You can imagine a register 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 can tell that the first line copies the value `0Eh` to the register `ah`, and the second line copies the character `s` to the register `al`. The single quotation is used in NASM to represent strings or characters and that's why we have used it in the second line, based on that, you may noticed that the value `0Eh` is not surrounded by a single quotation though it contains characters, in fact, this value isn't a string, it is a number that is represented by using hexadecimal numbering system and due to that the character `h` was put in the end of that value, to tell NASM to move a value `0E` which is a hexadecimal number to the register `al`, the equivalent number of `0E` in the decimal numbering system which, we human, are used to is `14`, that is `0E` and `14` are the exactly the same, but the are represented in two different numbering system[8].

NASM

Netwide Assembler (NASM) is an open-source source assembler for x86 architecture which uses Intel's syntax of assembly language, the other well-known

[7] https://software.intel.com/en-us/articles/intel-sdm

[8] Numbering systems will be discussed in more details later.

syntax for assembly language is AT&T syntax and, of course, there are some differences between the two, the first syntax is used in the official manuals of Intel. NASM can be used through command line to assemble [9] x86 assembly code and generate the corresponding machine code. The basic usage of NASM command is the following.

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

The argument `format` decides the binary format of the generated machine code, the binary format will be discussed in more details in a moment. 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, the default name will be same as the filename with a different extension.

Binary Format

A *binary format* is basically a specification which gives a blueprint of how a binary file is organized, in other words, it describes how a binary file is structured, there are multiple parts in a binary file that is formatted by a specific binary format and the generated machine code is one part. Note that each executable file uses some binary format to organize its content and to make a specific operating system understands its content. There is no difference between the programming languages in the matter of the binary format [10] that will be used in the last output of the compiling process, for example in Linux, if we create a software either by C, Rust or assembly, the last executable result will be a binary file that is formatted by using a binary format known as *Executable and Linkable Format* (ELF) which is the default in Linux. There are many other binary formats, Mach-O is one example which is used by Mach-based [11], another example is Portable Executable (PE) which is used by Microsoft Windows.

Each operating system knows its own binary format well, and knows how a 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 processor. 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.

[9] The process of transforming an assembly source code to machine code is known as *assembling*.

[10] 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.

[11] 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 used Mach as its kernel,

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 processor architecture, for example, the machine code that has been generated for x64 [12] cannot run on x86. Because of that the binary files are distributed according to the processor 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 processor such as Raspberry Pi for example. Due to that, to distribute a binary file of the same software for multiple processor's architectures, a separate binary file should be generated for each architecture, to solve this problem, a binary format named `FatELF` was presented. In this binary format, the software machine code of multiple processor architectures are gathered in one binary file and the suitable machine code will be loaded and run based on the type of the system's processor. Naturally, the size of the files that use such format will be bigger than the files that uses a binary format that is oriented for one processor architecture. Due to the bigger size, this type of binary formats is known as *fat binary*.

Getting back to the `format` argument of NASM, 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 which doesn't use any specification, instead, in flat binary files, the output is stored as is with no additional information or organization, only the output machine language of our code. Using flat binary does make sense and that's because the code which is going to load [13] our binary file doesn't understand any binary format to interpret it and fetch the machine code out of it, instead, the content of the binary file will be loaded to the memory as is.

GNU Make

GNU Make is a build automation tool. Well, don't let this fancy term make you panic! the concept behind it is too simple. When we create a kernel of an operating system [14] we are going to write some assembly code and C code and both of them need to be assembled and compiled (for the C code) to generate the machine code as binary files out of them. With each time a modification is made in the source code, you need to recompile (or reassemble) the code over and over again through writing the same commands in the terminal in order to generate the last binary output of your code. Beside the compiling and recompiling steps,

[12] The x86 architecture that supports 64-bit.
[13] Which is BIOS as we will see latter.
[14] Or any software with any other compiled programming languages.

an important step needs to take its place in order to generate the last output, this operation is known as *linking*, which links the different *object files* [15] with each other to generate one binary file out of these multiple object files. These operations which are needed to generate the last binary file out of the source code is known as *building process*, which, as mentioned earlier, involves executing multiple commands such as compiling, assembling and linking. This makes the building process a tedious job and error-prone process. To save our time (and ourselves from boredom of course) we don't want to write all these commands over and over again in order to generate the last output, we need an alternative and here where GNU Make [16] comes to the rescue, it *automates* the *building* process by gathering all required commands in a text file known as `Makefile`, once the user runs this file through the command `make`, GNU Make is going to run these commands sequentially, furthermore, it checks whether a code file is modified since the last building process or not, if the case is that the file is not modified then it will not be compiled again and the generated object file from the last building process is used instead, which of course minimize the needed time to finish the building process.

Makefile

A `makefile` is a text file that tells GNU Make what are the needed steps to complete the building process of a specific source code. There is a specific syntax that we should obey when writing `makefile`. A number of *rules* may be defined, we can say that a `makefile` has a list of rules that define 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 the `makefile` only 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 name `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 when writing a `makefile` is to define a rule with target name `clean` that deletes all object files and binaries that have been created in the last 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 list of dependencies, those dependencies can be either filenames (the C files of the source code for instance) or other rules in the same `makefile`. For GNU Make, to run the a

[15] 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.

[16] And any other building automation tool.

specific rule successfully, the dependencies of this rule should be fulfilled, if there is another rule in the dependencies, it should be executed successfully first, if there is a filename in the dependencies list and there is no rule that has the same filename as a target name, 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 being executed. These commands are normal Linux commands, so in this part of a rule we are going to write the compiling commands to compile the C source files, assembling commands for the assembly source files and linking command that links the generated object 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, the first one is `file1.c`, the second one is `file2.h` and the third one is `file2.c`.

```
#include "file2.h"

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

```
void func();
```

```
#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.

```
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` which its name doesn't start with a dot, then it will be executed directly once the command `make` is issued, another way to execute this rule is by mentioning its name explicitly as an argument to `make` command as the following: `make build`.

The rule `build` depends on two C files, `file1.c` and `file2.c`, they should be available on the same directory. The the recipe uses GNU GCC to compile and link these two files and generate an executable file named `ex_file`. The following is an example of a `makefile` that has multiple rules.

```
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
```

In this example, the first rule `build` depends on the two object files `file1.o` and `file2.o`. Before running the first building process, these two files will not be available in the source code directory [17], 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.

```
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)
```

[17] Since they are a result of one step of the building process which is the compiling step which has not been performed yet.