Binary Exploitation

An Introduction

Adwaith Gautham

Compiled on 8th May 2021

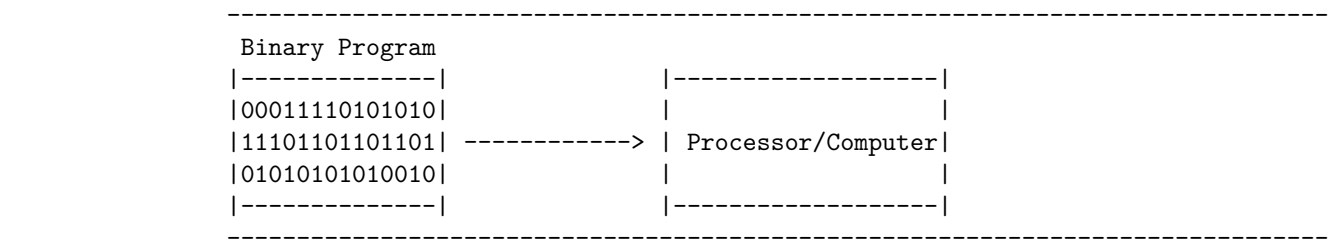
Contents

Chapter 1: Internals of Compiling

Summary
This chapter explores how a C program is converted into a file that can be run on the machine. The conversion is a multi-step process which involves lot of interesting things including the crazy Executable and Linkable Format(ELF). All of these are explored in detail.

1.1 Init

Let us start with a question. How was the first program ever written? Computers were mechanical machines back then. Handles and gears were used to program them. As time progressed, handles and gears were replaced by electrical signals. Later, a computer was designed to be programmed by just 0s and 1s - the binary language. Because this code is directly run on the machine, it is called **machine code**. This was amazing progress. But it was still not convenient because humans could not read that code, it was hard to debug and hard to write too. Just one mistake and it had to be written all over again. It looked like the following diagram.



What could be done to solve this problem? We need a programming language which is easy to write, read, debug and correct mistakes.

We need a programming language which is a level **higher** than the machine language. A programming language could be designed in such a way that when it is encoded, it produces 0s and 1s. Assume there are 4 operations the computer supports: Addition, Subtraction, Multiplication and Copying data from one place to another. The following is the encoding scheme.

1. 00 - `add`
2. 01 - `sub`
3. 10 - `mul`
4. 11 - `copy`

It is common to copy constants(or immediate values) into memory locations. That is why, let us consider one more type of copy operation. Let us call it `copyc` which stands for copy-constant. With this operation added, we need 3 bits to encode all the operations.

1. 000 - `add`
2. 001 - `sub`
3. 010 - `mul`
4. 011 - `copy`
5. 100 - `copyc`

Similarly, to write a full fledged programs with loops, functions etc., we would need a lot more operations like comparison, branching(jumping from one place to another) etc., Assume we can encode all the operations with 8 bits(ie., a maximum of 256 operations).

These operations need operands to work on. Assume length of each address is 16 bits and length of each data element is also 16 bits. This way, we would need(8+16+16)40 bits(or 5 bytes) to encode each operation. Let the result of the operation be stored in the first operand. Our operations now look like the following.

1. `add Address1, Address2`: Contents at Address1 and Address2 are added and stored back at Address1.
2. `sub Address1, Address2`: Difference between contents at Address1 and Address2 is computed and stored back into Address1.
3. `mul Address1, Address2`: Product of contents at Address1 and Address2 is computed and stored at Address1.
4. `copy Address1, Address2`: Contents at Address2 is copied to memory at Address1.
5. `copyc Address1, constant`: Copies a 2-byte constant into memory at Address1.

Note that contents mean 2 bytes at a specified address.

`add 0x1000, 0x1004` will be encoded as `00000000-0001000000000000-0001000000000100`. This operation adds 2 bytes present at address 0x1000 and 2 bytes present at address 0x1004 and stores the sum at 0x1000.

This high-level language given to the programmers is called **Assembly Language**.

We are at a stage where we can write a program using assembly language. Let us write a program to swap 2 given numbers - call it *swap.S*.

```
-------------------------------------------------------------------------swap.S
0x1000: 0x0010
        0x0020  ; present at 0x1002
        0x0000  ; present at 0x1004

0x0000: copy 0x1004, 0x1000
        copy 0x1000, 0x1002 ; present at 0x0005
        copy 0x1002, 0x1004 ; present at 0x000a

; At this point, the 2 values at 0x1000 and 0x1002 have been swapped.
-------------------------------------------------------------------------------
```

This program requests the operating system to store the variables at the address 0x1000 and code at address 0x0000.

All this does not happen on its own. Someone should convert these assembly programs into binary programs right? The computer even now understands only the binary language. Whatever high-level language we *invent* to write programs, they must be translated to machine language(0s and 1s) and then fed to the computer for execution. **We need a program/tool which does this translation**. Writing such a translator paved way to write better, cleaner programs quickly. A program which converts assembly language into machine language is known as an **Assembler**. An assembler **assembles** instructions to generate their the machine equivalent code.

This is great progress.

Currently, variables and code are present together. Separating them would make the program **cleaner** and **more secure**. While writing the program, it can be divided into 2 **sections**. One section can have variable declarations/definitions - call it **data section** and other can have assembly instructions - call it **text section**. The assembler takes out a two-section assembly program as input and gives a binary program as output which also has the 2 sections in binary form.

Now, our two-section *swap.S* looks like this.

```
-------------------------------------------------------------------------swap.S
section data

0x1000: 0x0010
        0x0020  ; present at 0x1002
        0x0000  ; present at 0x1004

section text

0x0000: copy 0x1004, 0x1000
        copy 0x1000, 0x1002 ; present at 0x0005
        copy 0x1002, 0x1004 ; present at 0x000a

; At this point, the 2 values at 0x1000 and 0x1002 have been swapped.
-------------------------------------------------------------------------------
```

We started with programs being a stream of bits entered into the computer. Now, we have divided the program into sections. The binary program would look like the following - *swap.bin*.

```
-------------------------------------------------------------------swap.bin
0001000000000000    ; Address 0x1000
0000000000010000    ; 0x0010
0000000000100000    ; 0x0020
0000000000000000    ; 0x0000

0000000000000000    ; Address 0x0000
00000011 0001000000000100 0001000000000000  ; copy 0x1004, 0x1000
00000011 0001000000000000 0001000000000010  ; copy 0x1000, 0x1002
00000011 0001000000000010 0001000000000100  ; copy 0x1002, 0x1004

; At this point, the 2 values at 0x1000 and 0x1002 have been swapped.
-------------------------------------------------------------------------------
```

Writing all this in hexadecimal will be a lot more easier to read.

```
-------------------------------------------------------------------swap.bin
1000
0010
0020
0000

0000
0310041000
0310001002
0310021004
```

```
--------------------------------------------------------------------------------
```

Consider the imaginary asssembler **myasm**. Run it on the two-section *swap.S* to get *swap.bin*.

```
--------------------------------------------------------------------------------
$ myasm swap.S -o swap.bin
--------------------------------------------------------------------------------
```

Now, let us think from the operating system's perspective. It should be able to process the binary program in an **unambiguous** manner. Let us build a few rules which the operating system can use to parse the binary program.

1. The binary program always starts with the data section.
2. The data section and text section is separated by 2 newlines.
3. Every section starts by specifying its starting address in memory.

Assuming every data element is 2 bytes and every instruction is 5 bytes, *swap.bin* can be parsed unambiguously and then executed.

Instead of using newlines and spaces to make processing unambiguous, we can have a header-like structure which has all the necessary **metadata** about the binary program. Let us see what metadata we need to put in the header.

1. Relative to the beginning of the file, where is the data section present - **data_section_offset** (in bytes).
2. Size of the data section - **data_section_size** (in bytes).
3. Relative to the beginning of the file, where is the text section present - **text_section_offset** (in bytes).
4. Size of the text section - **text_section_size** (in bytes).

Along with the above 4, we can specify the starting addresses of data and text sections in the header itself - **data_section_addr**, **text_section_addr**.

Assuming each of these to be 2 bytes in size, we can tell that the first 12 bytes will be the above metadata. Right after this metadata, the actual data and text sections are placed.

```
--------------------------------------------------------------------------------BFF
data_section_offset :   bytes 0-1
data_section_size   :   bytes 2-3
data_section_addr   :   bytes 4-5
text_section_offset :   bytes 6-7
text_section_size   :   bytes 8-9
text_section_addr   :   bytes 10-11

; Either data or text section right after the header
--------------------------------------------------------------------------------
```

We just designed a simple file format for binary programs. Let us call it **BFF - Binary File Format**. The following is how *swap.bin* in BFF looks like.

```
-----------------------------------------------------------------swap.bin in BFF
000c        ; data section offset = 12 bytes
0006        ; data section size = (2+2+2) = 6 bytes
1000        ; data section starting address = 0x1000
0012        ; text section offset = (12 + 6) = 18 bytes
000f        ; text section size = 15 bytes
0000        ; text section starting address = 0x0000
0010        ; data section starts
```

```
0020
0000
0310041000  ; text section starts
0310001002
0310021004
--------------------------------------------------------------------------------
```

The binary program is now self-describing. The part of the operating system that processes the binary program should be aware of the Binary File Format. Our **assembler** should be modified to generate a binary program in BFF.

Now, there is a definite structure to the generated binary programs, but there are a few problems here.

Handling raw addresses can lead to mistakes. Instead of writing `add 0x1000, 0x1004`, you write `add 0x0100, 0x1004`, the program definitely won't function properly, it might actually crash. And when dealing with numbers, these are common mistakes we do. How can this problem be solved? **Variable names** can be used instead of raw addresses. Map variable names to addresses in the beginning of the program, later only use the variable names throughout the program. The **assembler** should be updated with this feature.

Along with that, it would be better if the code can be divided into chunks each performing a particular task - similar to functions. Enter **labels**! Let us rewrite the two-section *swap.S* using variable names and labels.

```
-----------------------------------------swap.S with variable names and labels
section data

0x1000:

a: 0x0010       ; 0x1000
b: 0x0020       ; 0x1002
temp: 0x0000    ; 0x1004

section text

0x0000:

_start:             ; 0x0000
    jump swap_a_b   ; 0x0000

swap_ret:           ; 0x0005
    add a, b        ; 0x0005
    exit            ; 0x000a

swap_a_b:           ; 0x000f
    copy temp, a    ; 0x000f
    copy a, b       ; 0x0014
    copy b, temp    ; 0x0019
    jump swap_ret   ; 0x001e
--------------------------------------------------------------------------------
```

Let us write the BFF equivalent of the above *swap.S*. Assume `jump` is the 6th instruction, it is encoded as `00000101` and `exit` as the 256th instruction and is encoded as `11111111`.

```
--------------------------------------------------------------------------------
```

```
$ myasm swap.S -o swap.bin
--------------------------------------------------------------------------------
```

The below *swap.bin* is generated in the following manner by our imaginary assembler.

```
-----------------------------------------------------swap.bin written using BFF
000c        ; data section offset = 12 bytes
0006        ; data section size = 6 bytes
1000        ; data section starting address = 0x1000
0012        ; text section offset = 18 bytes
0023        ; text section size = 35 bytes
0000        ; text section starting address = 0x0000
0010        ; data section starts
0020
0000
050000000f  ; text section starts. swap_a_b = 0x0000000f
0010001002  ; add a, b
ff00000000  ; exit(0)
0310041000
0310001002
0310021004
0500000005  ; jump swap_ret
--------------------------------------------------------------------------------
```

Labels and variable names are the facilities given to the programmer. It is important to note that variable names and labels are simply **aliases** to addresses. The assembly program has used labels and variable names but the operating system does not care about names or labels. All it cares about are addresses. The assembler needs to be updated to replace all the labels and names with the corresponding memory addresses when converting an assembly program into binary.

Look at the convenience the programmer gets from variable names and labels. Consider the very first instruction - `jump swap_a_b`. Without the label, he should have written `jump 0x000f`. What if there are a lot of jumps? Or there are many instructions between `_start` and `swap_a_b` and the programmer made a mistake in calculating the address. All this grunt work is avoided and is loaded onto the assembler.

Note that we wrote *swap.bin* easily because there were hardly any instructions. Address of every label, variable name was easy to compute. But as discussed above, what happens when it is a long program? Computing the addresses for variable names and labels manually is a pain. This means the assembler needs to be updated to compute addresses and replace them when converting instructions to binary.

How should the assembler be modified in order to translate such an assembly program with sections, variable names and labels into binary program?

Let us see how it can be done.

Consider the instruction `jump swap_a_b`. Without the label, it should actually be `jump 0x000f`. Finally, it needs to be encoded to `00000101-00000000000000000000000000001111`. The `jump` instruction is encoded as `00000101`. 0x000f is encoded as a 32-bit number. Whenever the assembler encounters `swap_a_b`, it needs to be replaced with its address 0x000f. The exact thing needs to be done for all the other variables and labels too. Whenever **a** is encountered, it needs to be replaced with its address 0x1000. The variable names and labels are collectively called **Symbols**. To make this mechanism work, we can construct a simple table with Symbol as Key and its address as Value. Such a table is known as **Symbol Table**. The assembler takes the asssembly program as input and it processes it once top to bottom and constructs the table. The following is the symbol table.

Symbol	Address
a	0x1000
b	0x1002
temp	0x1004
_start	0x0000
swap_ret	0x0005
swap_a_b	0x000f

With if the variable name and a label are same? To resolve this, the symbol type can be stored.

Symbol	Type	Address
a	variable	0x1000
b	variable	0x1002
temp	variable	0x1004
_start	label	0x0000
swap_ret	label	0x0005
swap_a_b	label	0x000f

By processing the assembly program once(in **one pass**), the symbol table is constructed. The binary program is generated in the next pass. Because our assembler processes the assembly program twice(**2 passes**), this is a two-pass assembler.

The ease of programming has increased a lot at the cost of a complex assembler.

This is a lot of progress compared to binary language which was being used in the beginning.

We have come a long way, but there are still issues. Our assembler forces the programmer to write the complete program in a single file. But splitting the code into various files based on their functionality(or any other criteria) is helpful. Let us split our swapping program into 2 parts: *main.S* and *swap.S*.

```
-------------------------------------------------------------------------main.S
section data

0x1000:

a: 0x0010       ; 0x1000
b: 0x0020       ; 0x1002

section text

0x0000:

_start:             ; 0x0000
    jump swap_a_b   ; 0x0000

swap_ret:           ; 0x0005
    add a, b        ; 0x0005
    exit            ; 0x000a
-------------------------------------------------------------------------------

-------------------------------------------------------------------------swap.S
```

```
section data

0x2000:

temp: 0x0000        ; 0x2000

section text

0x3000:

swap_a_b:           ; 0x3000
    copy temp, a    ; 0x3000
    copy a, b       ; 0x3005
    copy b, temp    ; 0x300a
    jump swap_ret   ; 0x300f
--------------------------------------------------------------------------------
```

Slowly go through the above code. I want you to be convinced that splitting the sourcecode actually helps. *swap.S* can behave like a module. Any program which needs to swap 2 numbers - has to name those 2 variables **a** and **b** and they can readily use code present in *swap.S*.

Let us try to translate *main.S* using our assembler.

```
--------------------------------------------------------------------------------
$ myasm main.S -o main.bin
--------------------------------------------------------------------------------
```

What do you think? Will this successfully give *main.bin*?

Let us walk through the procedure. The assembler parses *main.S* once and constructs a symbol table.

Symbol	Type	Address
a	variable	0x1000
b	variable	0x1002
_start	label	0x0000
swap_ret	label	0x0005

In the second pass, every statement in *main.S* is translated into its binary equivalent. There are no issues with the data section. There are 2 symbols(a and b) are both are replaced by their address with the help of the symbol table. Then comes the instruction `jump swap_a_b`. As usual, the assembler searches for the symbol **swap__a__b** in the symbol table, but no luck. It is not present. It means `jump swap_a_b` cannot be assembled to its binary equivalent. The assembler gives an error and exits. The assembler was **unable to resolve a symbol into its address**.

Similar thing happens when you try assembling *swap.S*. A few symbols(a, b and swap_ret) won't get resolved. The reason is simple. Those symbols are simply not defined in the respective files. Our assembler currently has no mechanism to support our multiple files idea.

How can we solve this problem? I want you to think of ways to solve this.

We know that the missing symbols in *main.S* are found in *swap.S* and symbols missing in *swap.S* are found in *main.S*. We need to inform this information to the assembler. We need to design a mechanism for the assembler to resolve all the symbol-dependencies in both(or more) files.

Say we inform the assembler in this manner.

```
------------------------------------------------------------------------------
$ myasm main.S -o main.bin -depend swap.S
------------------------------------------------------------------------------
```

The assembler can parse both *main.S* and *swap.S* once and construct symbol tables for both. In *main.S*'s second pass, the assembler will find that there is no symbol table entry for **swap__a__b**. Instead of raising an error, we have another option. We search for **swap__a__b** in *swap.S*'s symbol table. If it is found, then go ahead and generate binary code for `jump swap_a_b`. If not found, then it needs to raise an error because there is no where else to search. The same should be done with the symbols not found in *swap.S*. When it encounters **a**, it won't find it in *swap.S*'s table. It needs to search for it in *main.S*'s table. Because there is an entry for **a** in *main.S*'s table, the assembler should be able to resolve it. This is one solution.

When you look at this, it seems like a working solution. But if you dig deeper, it has a few problems.

1. Say there are 100 files. We need to specify the starting addresses of data and text sections present in each of the 100 files. We specified 0x1000 to be starting address of *main.S*'s data section, 0x2000 to be the starting address of *swap.S*'s data section. What if the number of variables in *main.S* exceed 4096 bytes(0x2000-0x1000)? When there are 100 files, taking care of all this is very difficult. Programmer giving out addresses to each section might cause problems.

2. When there are 100 files, there will be 100 data and 100 text sections. Each of these sections have a different starting address. How should these 100+100 sections be organized? Should we change BFF to acoomodate it? Or should we combine all 100 data sections to give a single data section? If yes, how exactly do we do it?

These are problems we need to solve.

So far, the programmer was given the liberty to specify the addresses. It worked well in simple scenarios. But the scenario present above puts unnecessary burden on the programmer, which is common in case of large projects. Will the programmer have an advantage if he specifies the addresses for each data and text section? Not really. All that matters to him is that memory is allocated for these data and text sections. It is best to offload this job to another program or the operating system. Operating system's resources are precious and it needs to be spent carefully. When the operating system is trying to resolve symbols and could not find even a single symbol, it throws an error and all the work done by the operating system is a waste. It is best to write a program to get the job done instead of the operating system doing it All this new program needs to do is to automate the process of giving addresses to all variables and instructions.

The assembler can be extended to do this. But I think it is a stretch. The assembler is responsible for correct translation of assembly code to binary. Resolving unresolved symbols, fixing addresses, stitching all the sections of all sourcefiles to make the binary program is beyond its definition. Let us create another program which does exactly the above listed functions and finally generate the binary program.

How is a symbol resolved? It is used in one sourcefile but is defined in some other. Resolving is nothing but **linking** its usage to its definition. For this reason, let us call this program **linker**. The linker should do all the above listed functions and generate the binary program.

Let us list a few observations which will help us write the linker.

1. The programmer does not specify the addresses of variables, labels and instructions. The linker needs to give addresses.

2. When there are multiple sourcefiles, one of them should contain the **entry-point** to the program ie., the first instruction which should get executed by the operating system.

3. The symbol tables constructed for each sourcefile by the assembler will be helpful in resolving the symbols.

As of now, the assembler constructs a symbol table for each of the sourcefiles. Even if the asssembler has translated 1000s of instructions of a sourcefile into its binary equivalent, when it encounters an undefined symbol, it gives up. It throws an error and all of the work done so far is wasted. It should be noted that only the instructions with unresolved symbols needs to be worked on. Rest of the instructions can be translated to binary without any hustle. That is work gone waste. We want our assembler to do the best(translate all possible variables and instructions) and inform us about all the work it failed to do.

Let us change the assembler to our advantage. The asssembler takes up a file for translation. In the first pass, it generates symbol table of all the symbols defined in the sourcefile. In the second pass, it translates variable by variable, instruction by instruction. If it encounters a symbol not defined in the sourcefile(aka not present in the symbol table), create an empty symbol table entry for this undefined symbol and continue translation. At the end of it, we have a binary equivalent data and text section, along with a symbol table. Along with storing the binary equivalent of data and text sections in the binary file, store the symbol table too. This way, the assembler translated all the possible instructions to binary. Rest of the instructions which had unresolved symbols, information about it is stored in the symbol table.

In the above described process, a few things were abstract. If an undefined symbol is encountered(say in `jump swap_a_b`), create an entry for this undefined symbol(**swap_a_b**) and **continue translation**. We did not specify what **continue** exactly means? Ideally, `jump swap_a_b` would be translated into 5-byte binary, where the first byte specifies that the instruction is `jump` and the next 4 bytes - the address of **swap_a_b**. But now, the asssembler does not know **swap_a_b**'s address. What should the 5-byte binary contain then? The first byte is **0x05**, `jump`'s binary equivalent, but what about rest of the 4 bytes? That information is found only in the next stage when the linker gives addresses.

Consider our standard *main.S* and *swap.S* example.

```
-------------------------------------------------------------------------main.S
section data

a: 0x0010
b: 0x0020

section text

_start:
    jump swap_a_b

swap_ret:
    add a, b
    exit
-------------------------------------------------------------------------------

-------------------------------------------------------------------------swap.S
section data

temp: 0x0000

section text
```

```
swap_a_b:
    copy temp, a
    copy a, b
    copy b, temp
    jump swap_ret
--------------------------------------------------------------------------------
```

The assembler tries to translate *main.S* into a file which should contain 3 parts: data section, text section and symbol table.

1. This is how the data section looks like.

```
--------------------------------------------------------------------------------
0x0000:     ; data section
0010        ; a
0020        ; b
--------------------------------------------------------------------------------
```

The 0x0000 signifies the fact that programmer has not given any address. Relative to the beginning of data section, **a** is at an offset of 0x0000, **b** is at an offset of 0x0002. Unlike before, we do not know the absolute addresses of these variables. We only know their addresses(actually offsets) relative to the corresponding section beginning.

2. This is how text section would look like.

```
--------------------------------------------------------------------------------
0x0000:
05<??>      ; Don't know where swap_a_b is.
00<??><??>
ff00000000  ; exit(0)
--------------------------------------------------------------------------------
```

The assembler encounters `jump swap_a_b`. It sees that **swap__a__b** is not found in the symbol table. It creates an empty symbol table entry for **swap__a__b**. If the assembler knew **swap__a__b**'s absolute address(assume it is 0x9034), it would generate the binary code 0500009034. But now, the assembler does not know its address. What needs to be done?

The assembler should inform the **linker**(next stage) about this.

What we can do is, we can simply put `0500000000` - basically **0x00**s instead of the actual address and inform that 4 bytes at an offset of 0x0001 in the text section should be replaced by **swap__a__b**'s absolute address once found. Once the linker finds **swap__a__b** and gives it an address, `0500000000` is changed to `0500009034`.

Let us take up the next instruction `add a, b`. All the assembler can do is to translate it as `00-0000-0002`. It can put the relative addresses of **a** and **b**(with respect to the data section). Even this needs fixing. Ideally, absolute addresses of **a** and **b** should be present here. Even in this case, the assembler needs to inform the linker about this. Suppose the linker gives the data section the address 0x6000, then **a**'s address would be 0x6000+0x0000=0x6000(data section's address + offset at which **a** is present). **b**'s address would be 0x6002. Linker changes `0000000002` to `0060006002`.

You can see that there are 2 cases here. In the first case, linker had to find the label, give an address to it and then change the binary code. In the second case, the section had to be given an address which implicitly gives addresses to all the variables in it, then change the binary code of instructions which have used these variables. We discussed that the assembler needs to inform about these 2 cases to the linker. How can that be done? Think about how it can be solved.

3. Symbol table would look like this.

Symbol	Type	Address
a	variable	0x0000
b	variable	0x0002
_start	label	0x0000
swap_ret	label	0x0005
swap_a_b	undefined	——

The addresses in this symbol table are not the actual addresses. They are relative addresses of variables/labels with respect to the sections they belong to.

Coming back to the question at end of (2), how can the assembler inform the linker about what bytes need to be replaced by what? One solution is to create another table for this. Let us call it **fixing table** because it has information about what all bytes should be replaced by what.

1. In the first case, bytes 1-4 in the text section which are currently zeros should be replaced by the address of **swap_a_b**.
2. In the second case, bytes 0x0006 - 0x0007(2 bytes) in the text section whhich currently represent the relative address of **a** should be replaced by the absolute address of **a** once the linker has it. There is one more way to solve this case.
    - Instead of putting the relative addresses in the binary code and later replacing it with the actual address, it can have zeros and finally the linker can replace it with the actual address by taking the symbol table's help. Consider the variable b. In the add a, b instruction, currently it is translated to 00-0000-0002 – relative addresses of variables are put here. Later, when the data section is given an address(say 0x5000), 0000 is replaced by (0x5000+0x0000) and 0002 is replaced by (0x5000+0x0002). Either this can be done or the assembler can simply translate add a, b into 00-0000-0000 – basically zeros in places which need replacement. Later linker comes in and checks the fixing table. It sees that the first 0000 needs to be replaced by a's address. With symbol table's help, the linker updates it with absolute addresses.

The first case was an **unresolved label issue**, the second is an **unresolved variable name issue**. These are the only 2 cases we have and we have a solution for it. Let us design the fixing table. It needs to have the following details.

1. Offset at which bytes must be replaced. Observe that the offset is always with respect to the text section. Only the text section needs fixing because the programmer would have used undefined symbols there.
2. At the specified offset, how many bytes need to be replaced. In the first case, 4 bytes had to be replaced. In the second, for each variable, 2 bytes had to be replaced.
3. We have specified the location of replacement and how much to replace. We need to specify what it needs to be replaced **with**. In the first case, the 4 bytes should be replaced with **swap_a_b**'s address. In the second case, 2 bytes need to be replaced with **a**'s address, other 2 bytes with **b**'s address. Basically, we need to specify the symbol.

The fixing table would look like this for *main.S*.

Offset	Number of bytes	Symbol
1	4	swap_a_b
6	2	a
8	2	b

With the help of symbol table and fixing table, the linker should be able to fix all the instructions in the text section.

Note that the file output by the assembler is not a binary **program**. A program is something which can be run on the machine. It is a file which has a bunch of sections(data, text, symbol-table). It cannot be run because nothing is ready yet. Multiple symbols might be unresolved, its dependent files still need to be stitched with this file. Because these files will be used by our **linker** to generate the binary program, let us call them **linkable files**.

In the beginning, the assembler's output was a simple file with just a text and a data section. As we progressed, we designed a file format for the assembler's output – the Binary File Format which had a simple header. Now, we want the assembler to output a file which has the data section, text section, symbol table and the fixing table. Parts of the text section with unresolved variable names, labels will have zeros instead of their addresses. There will be empty symbol table entries, addresses in symbol table are all relative addresses. We call this file a **linkable file**.

We need to update the Binary File Format to match our linkable file requirements.

It has a data and text section, a symbol table and a fixing table. A section can contain anything. Both the tables can be present in the form of sections. The new BFF header would look like this.

```
-------------------------------------------------------------------------------
data_section_offset     :   bytes 0-1
data_section_size       :   bytes 2-3
data_section_addr       :   bytes 4-5
text_section_offset     :   bytes 6-7
text_section_size       :   bytes 8-9
text_section_addr       :   bytes 10-11
symtab_section_offset   :   bytes 12-13
symtab_section_size     :   bytes 14-15
fixtab_section_offset   :   bytes 16-17
fixtab_section_size     :   bytes 18-19

; Either data or text section right after the header.
-------------------------------------------------------------------------------
```

Note that the number of sections are steadily increasing. if we need to support program debugging, we can have one more section with debugging details. Then, we will have to add 2 more members to the BFF header. Whenever a new section needs to be introduced, BFF format itself needs changes. Any file format should be flexible to changes. How can BFF be designed in that way? Think about it.

Every section has some metadata related to it - section offset in the file, section size, starting address. Instead of having metadata of all sections in the BFF header, we have have a separate section-metadata header. Because we have a number of sections, wee can have an array of such section-metadata headers. When we process a section header, we should be able to get complete information about that section including what type of section it is - is it data, text, symtab or fixtab. Let us call the section-metadata headers `bff_sec_meta_t`. From now onwards, let us use C to describe these structures.

```
-------------------------------------------------------------------------------
typedef sec_meta
{
    unsigned short int type;
    unsigned short int offset;
    unsigned short int size;
    unsigned short int addr;
```

```
} sec_meta_t;
```

We now have 4 types of sections: text, data, symtab and fixtab. Let us write an enumerator for it.

```
typedef enum
{
    BFF_SEC_TEXT=0,
    BFF_SEC_DATA,
    BFF_SEC_SYMTAB,
    BFF_SEC_FIXTAB,
} bff_sec_type_t;
```

Let us discuss the symbol table. In all our previous discussions, we discussed about the table in a conceptual manner. We didn't think how to code a symbol table. Intuitively, it is an array of `<symbol, address>` pairs. Later, we added another attribute to it - **symbol-type**. A symbol structure would look something like this.

```
typedef struct symbol
{
    char name[256];
    unsigned short int type;
    unsigned short int addr;
} bff_sym_t;
```

There are 3 types of symbols as of now: variable, label and undefined.

```
typedef enum
{
    BFF_SYM_VAR=0,
    BFF_SYM_LABEL,
    BFF_SYM_UNDEFINED,
} bff_sym_type_t;
```

The symbol table should ideally be a hash table. If you input the symbol-name, you get the index where its structure is stored in the array of symbol structures. Let us discuss about the hash function and other details later.

Let us come to the fixing table. As discussed, each entry in the fixing table has 3 members: offset, number of bytes and symbol.

```
typedef struct fixtab_entry
{
    unsigned short int offset;
    unsigned short int nbytes;
    char symbol[256];
} bff_fix_ent_t;
```

With all these defined, we can decide the BFF header structure. All it needs to have is the number of sections. Everything else is defined in the section-metadata array.

```
--------------------------------------------------------------------------------
typedef struct bff_header
{
    unsigned short int bff_sec_n;
} bff_hdr_t;
--------------------------------------------------------------------------------
```

Now that we have all the fundamental elements, we can define the next version of BFF.

```
-----------------------------------------------------------------generic BFF file
bff_hdr_t                   ; Has number of sections
Array of bff_sec_meta_t     ; Array size depends on number of sections
; section1
; section2
.
.
.
; section bff_hdr_t.bff_sec_n
--------------------------------------------------------------------------------
```

This way, even if you need more sections later, you can add them without changing BFF. If you need one more section to contain debugging details, this version of BFF can easily handle it.

Out assembler **myasm** requires significant updates. By taking a single assembly sourcefile, it should generate a linkable file in BFF.

To firm up our discussion, let us take our *main.S* and *swap.S* example and generate 2 linkable files *main.link* and *swap.link* by hand. The following are the 2 sourcefiles.

```
-------------------------------------------------------------------------main.S
section data

a: 0x0010
b: 0x0020

section text

_start:
    jump swap_a_b

swap_ret:
    add a, b
    exit
--------------------------------------------------------------------------------


-------------------------------------------------------------------------swap.S
section data

temp: 0x0000

section text
```

```
swap_a_b:
    copy temp, a
    copy a, b
    copy b, temp
    jump swap_ret
-------------------------------------------------------------------------------
```

The assembler should take *main.S* and *swap.S* and generate *main.link* and *swap.link*. Let us go through one file at a time.

```
-------------------------------------------------------------------------------
$ myasm main.S
-------------------------------------------------------------------------------
```

1. Data section

```
-------------------------------------------------------------------------------
0x0000:     ; data section
0010        ; a
0020        ; b
-------------------------------------------------------------------------------
```

2. Text section

```
-------------------------------------------------------------------------------
0x0000:         ; text section
0500000000      ; jump swap_a_b
0000000000      ; add a, b
ff00000000      ; exit(0)
-------------------------------------------------------------------------------
```

Notice the first and second instructions. They both need fixing - assembler does not know the actual addresses of **swap_a_b** and **b**.

3. Symbol Table

Symbol	Type	Address
a	variable	0x0000
b	variable	0x0002
_start	label	0x0000
swap_ret	label	0x0005
swap_a_b	undefined	——

The symbol table has the symbols and their relative addresses. If the symbol is a variable, then its address is relative to the data section. If the symbol is a label, then its address is relative to the text section.

4. Fixing Table

Offset	Number of bytes	Symbol
1	4	swap_a_b
6	2	a
8	2	b

Let us now talk about *swap.S*.

1. Data section

```
--------------------------------------------------------------------------------
0x0000:     ; data section
0000        ; temp
--------------------------------------------------------------------------------
```

2. Text section

```
--------------------------------------------------------------------------------
0x0000:         ; text section
0300000000      ; copy temp, a
0300000000      ; copy a, b
0300000000      ; copy b, temp
0500000000      ; jump swap_ret
--------------------------------------------------------------------------------
```

This text section needs a lot of fixing.

3. Symbol Table

Symbol	Type	Address
temp	variable	0x0000
swap_a_b	label	0x0000
a	undefined	——
b	undefined	——
swap_ret	undefined	——

4. Fixing Table

Offset	Number of bytes	Symbol
0x1	2	temp
0x3	2	a
0x6	2	a
0x8	2	b
0xb(11)	2	b
0xd(13)	2	temp
0x11(16)	4	swap_ret

With all this, we have generated *main.link* and *swap.link*.

Now to the interesting part. Let us manually link these 2 files to generate an output program *a.out*. We know that a runnable program has only 2 sections - data and text.

Let us follow a very simple strategy. We have 2 data and 2 text sections. let us call them **main.data**, **swap.data** and **main.text**, **swap.text**.

Let us append all of these sections together. Assume we are dumping them into a file. **main.data** will be at the beginning - its file offset is 0x0000. **swap.data** will come right after **main.data** - this is at a file offset of **main.data.size**. **main.text** will be at an offset of (**main.data.size** + **swap.data.size**). **swap.text** will be present at an offset of (**main.data.size** + **swap.data.size** + **main.text.size**).

1. main.data.size = 4 bytes, main.data.off = 0x0000
2. swap.data.size = 2 bytes, swap.data.off = 0x0000+4 = 0x0004
3. main.text.size = 15 bytes, main.text.off = 0x0004+2 = 0x0006
4. swap.text.size = 20 bytes, swap.text.off = 0x0006+15 = 0x0015

Assume that the linker gives an address **0x1000** to the beginning. Automatically all the sections(and their members) get addresses. **addr = 0x1000**.

1. main.data.addr = addr + main.data.off = 0x1000
2. swap.data.addr = addr + swap.data.off = 0x1004
3. main.text.addr = addr + main.text.off = 0x1006
4. swap.text.addr = addr + swap.text.off = 0x1015

With these at hand, we can update the known symbols' addresses.Updating the addresses is very simple. We have the addresses of the sections above. Symbol table entries have the symbols' relative addresses. **Absolute-address = (section-address + relative-address)**. The following are the updated symbol tables.

1. *main.link*'s symbol table

Symbol	Type	Address
a	variable	0x0000
b	variable	0x0002
_start	label	0x0000
swap_ret	label	0x0005
swap_a_b	undefined	——

`converted to`

Symbol	Type	Address
a	variable	0x1000
b	variable	0x1002
_start	label	0x1006
swap_ret	label	0x100b
swap_a_b	undefined	——

2. *swap.link*'s symbol table

Symbol	Type	Address
temp	variable	0x0000
swap_a_b	label	0x0000
a	undefined	——
b	undefined	——
swap_ret	undefined	——

`converted to`

Symbol	Type	Address
temp	variable	0x1004
swap_a_b	label	0x1015
a	undefined	——

Symbol	Type	Address
b	undefined	——
swap_ret	undefined	——

Note that variable-type entries should be updated using **data.size**, label-type entries using **text.size**.

By combining 2 data sections, we have a single data section. By combining 2 text sections, we have a text section. Starting address of this new data section is **0x1000**, starting address of text section is **0x1006**. Size of data section is 6 bytes, size of text section is 35 bytes.

This is how it looks like.

```
--------------------------------------------------------------------------------
0010        ; a, 0x1000, data section
0020        ; b
0000        ; temp
0500000000  ; jump swap_a_b, 0x1006, text section
0000000000  ; add a, b
ff00000000  ; exit(0)
0300000000  ; copy temp, a
0300000000  ; copy a, b
0300000000  ; copy b, temp
0500000000  ; jump swap_ret
--------------------------------------------------------------------------------
```

This needs work. We need to fix everything to generate the program. Let us use the two fixing tables. Let us take a look at *main.link*'s fixing table.

Offset	Number of bytes	Symbol
1	4	swap_a_b
6	2	a
8	2	b

At offset 1, 4 bytes should be replaced with address of **swap__a__b**. From *swap.link*'s symbol table, we get address of **swap__a__b** = 0x1015. The offset is relative to **main.text**. In the file above, you go to that by adding the offset **1** with **main.text.off** computed above. We get 0x0006+1 = 0x0007. Go to byte 0x0007 and replace **00000000** with **00001015**. We have the following.

```
--------------------------------------------------------------------------------
0010        ; a, 0x1000, data section
0020        ; b
0000        ; temp
0500001015  ; <---UPDATED jump swap_a_b, 0x1006, text section
0000000000  ; add a, b
ff00000000  ; exit(0)
0300000000  ; copy temp, a
0300000000  ; copy a, b
0300000000  ; copy b, temp
0500000000  ; jump swap_ret
--------------------------------------------------------------------------------
```

Along with that, update **swap__a__b**'s symbol table entry in *main.link*. Its type is changed to **unde-**

fined to **label**. Its address is updated to 0x1015.

Let us move onto the next entry in the fixing table. It is at an offset of 6 bytes from **main.text**. This means, in the file above, it is present at the offset **main.text.off + 6** = 0x0006+6 = 0x000c. We need to replace the 2 bytes with **a**'s address. Go to the symbol table and get **a**'s address. It is 0x1000.

```
--------------------------------------------------------------------------------
0010        ; a, 0x1000, data section
0020        ; b
0000        ; temp
0500001015  ; jump swap_a_b, 0x1006, text section
0010000000  ; <---UPDATED add a, b
ff00000000  ; exit(0)
0300000000  ; copy temp, a
0300000000  ; copy a, b
0300000000  ; copy b, temp
0500000000  ; jump swap_ret
--------------------------------------------------------------------------------
```

The next entry in the fixing table. We need to go to byte **main.text.off + 8** = 0x0006+8 = 0x000e. We need to replace the 2 bytes with **b**'s address. Retrive it from the symbol table. Address if 0x1002.

```
--------------------------------------------------------------------------------
0010        ; a, 0x1000, data section
0020        ; b
0000        ; temp
0500001015  ; jump swap_a_b, 0x1006, text section
0010001002  ; <---UPDATED add a, b
ff00000000  ; exit(0)
0300000000  ; copy temp, a
0300000000  ; copy a, b
0300000000  ; copy b, temp
0500000000  ; jump swap_ret
--------------------------------------------------------------------------------
```

With that, all the *main.link*'s fixing table entries are satisfied.

Linker's job is to satisfy entries of all fixing tables. Let us move to *swap.link*'s fixing table.

Offset	Number of bytes	Symbol
0x1	2	temp
0x3	2	a
0x6	2	a
0x8	2	b
0xb(11)	2	b
0xd(13)	2	temp
0x11(16)	4	swap_ret

These offsets are relative to **swap.text**. Let us start with the first entry. It is at an offset of **0x1**. In the file above, we need to go to bytes **swap.text.off + 0x1** = 0x0015+1=0x0015. Let us go to byte 0x0016(22). The 2 bytes here needs to be replaced with **temp**'s address which is **0x1004** - retrieved from *swap.link*'s symbol table. After replacement,

```
--------------------------------------------------------------------------------
```

```
0010        ; a, 0x1000, data section
0020        ; b
0000        ; temp
0500001015  ; jump swap_a_b, 0x1006, text section
0010001002  ; add a, b
ff00000000  ; exit(0)
0310040000  ; <---UPDATED copy temp, a
0300000000  ; copy a, b
0300000000  ; copy b, temp
0500000000  ; jump swap_ret
--------------------------------------------------------------------------------
```

I hope you get an idea of what is being done. Simply continuing this process, we get the following.

```
--------------------------------------------------------------------------------
0010        ; a, 0x1000, data section
0020        ; b
0000        ; temp
0500001015  ; jump swap_a_b, 0x1006, text section
0010001002  ; add a, b
ff00000000  ; exit(0)
0310041000  ; <---UPDATED copy temp, a
0310001002  ; copy a, b
0310021004  ; copy b, temp
050000100b  ; jump swap_ret
--------------------------------------------------------------------------------
```

We are almost there. Everything is resolved, all the fixing tables' entries are satisfied. There is just one thing which needs to be sorted. Which is the first instruction run by the operating system? What is the entry-point to this program?

We know that the symbol `_start` which we used in *main.S* is the starting point. Its address is **0x1006**. It did not matter to the assembler and linker. But it matters to the operating system. The operating system will transferr control to the entry-point instruction and leave. We need to specify the **entry-point address**.

This can be done by the linker. The linker can search for `_start` symbol. The address aliased by the `_start` symbol is the entry-point address. Even when there are hundreds of sourcefiles, the linker will always search for `_start` to get the entry-point address. Without `_start`, the linker should throw an error.

With that, we have successfully stitched 2 linkable files into one single runnable program. This is what the linker needs to do. Let us summarize in short what the linker needs to do.

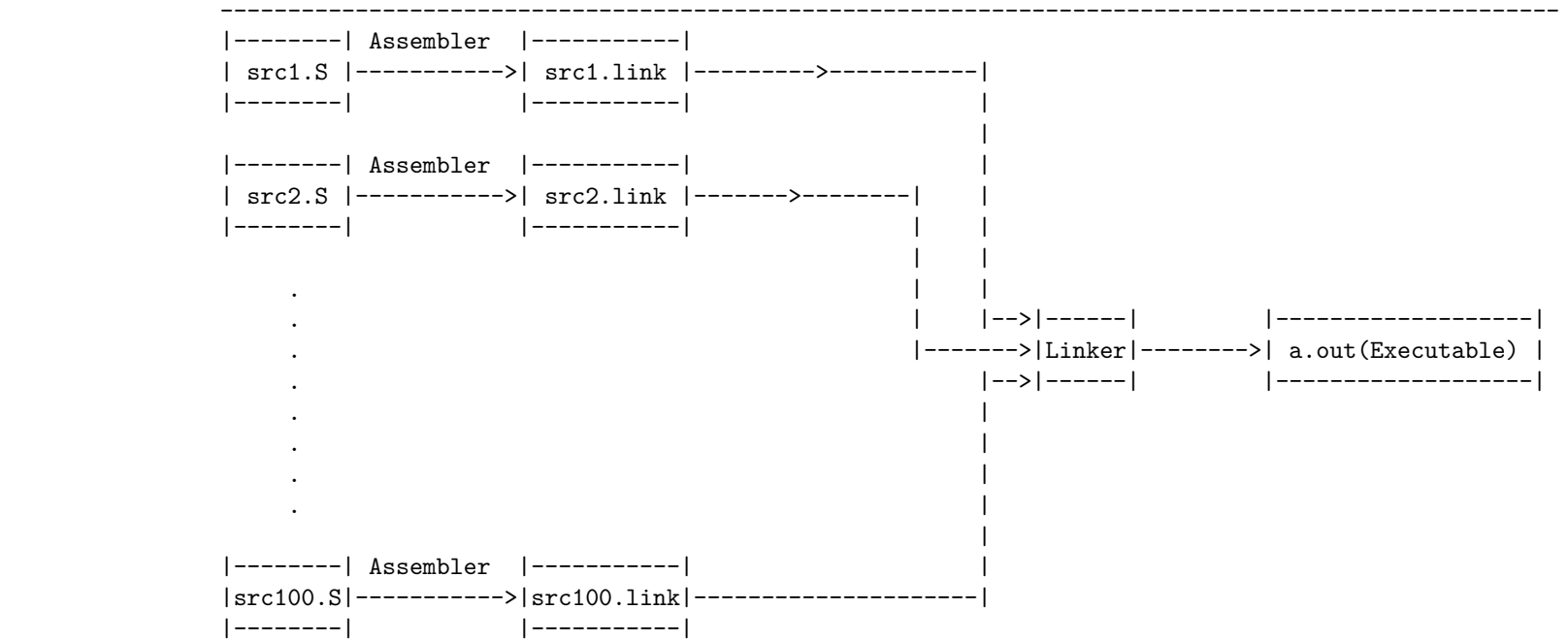
When multiple sourcefiles are given to the assembler, it generates multiple linkable files, one linkable file per sourcefile. Each linkable file has its own data, text, symtab and fixtab sections. The data section needs no fixing. But the text sections need fixing, specific bytes need replacement – everything that needs fixing is recorded by the assembler in the fixing table. The linker first absolute addresses to all the data and text sections. With that being done, update the symbol table entries' addresses. Once they are updated, start processing all the fixing tables' entries. With the help of symbol tables, ideally all the fixing tables' entries can be satisfied. Even if one fixing table entry is not satisfied, then a binary program cannot be generated, and we end up with a linking error. The linker searches for a symbol(here _start) which signifies the entry-point of the program.

The linker's output can be run on the Operating System. Such a runnable program is called an executable.

You can see that the BFF can be used for the executable too. An executable has 2 sections - data, text and nothing more. There is just one addition needed in BFF. it needs to have the entry-point address. It can be added to the BFF header. The following is the executable in the new BFF.

```
-------------------------------------------------------------------------------
0002        ; bff_hdr_t.bdd_sec_n = number of sections = 2
1006        ; bff_hdr_t.bff_entry_addr = 0x1006
0001        ; > bff_sec_meta_t.type = BFF_SEC_DATA
0014        ; data.offset = 20 bytes
0006        ; data.size = 6 bytes
1000        ; data.addr = 0x1000
0000        ; > bff_sec_meta_t.type = BFF_SEC_TEXT
001a        ; text.offset = 26 bytes
0023        ; text.size = 35 bytes
1006        ; text.addr = 0x1006
0010        ; data section starts
0020
0000
0500001015  ; text section starts
0010001002
ff00000000
0310041000
0310001002
0310021004
050000100b
-------------------------------------------------------------------------------
```

Compare this with the first BFF format. There is almost no difference at all. But the way it was generated was entirely different. As new features came in, complexity of converting an assembly program into an executable increased. Finally, we split the conversion process into 2 steps: Assembling and Linking. The following summarizes the progress so far.



This is good progress. The programmer can write code the way he wants – variables, functions, multiple files. He does not have to take care of anything Assembler and Linker will take care of everything.

All programs in general have a few common properties. They will need file manipulation(creating new files, writing to files, reading from files, deleting old files etc.,), the may need standard input/output functions to take in values from the user, to output results to the user. They may also have lot of string related operations etc., I write my own file manipulation code. You write your own file manipulation code. All the programmers write their own file manipulation code. This is a waste of time and effort right? It would be so good if one programmer writes a **library** and shares it with everyone else. A **library** is basically a bunch of functions which are present to facilitate certain functionality. A threading library is present to facilitate multi-threaded programs. Standard I/O library facilitates taking input from users and displaying results to the user.

Let us take the example of file manipulation. We need to write a library. Let us decide on the function names.

1. `file_create()`: Creates a new file
2. `file_open()`: Opens up a specified file
3. `file_read()`: Reads from a specified file
4. `file_write()`: Writes to a specified file
5. `file_close()`: Closes a specified file

Say we create 6 sourcefiles. 5 files with implementations of each of the above functions and one sourcefile with all the data structures, variables needed. Let us call them **file_create.S**, **file_open.S**, **file_read.S**, **file_write.S**, **file_close.S** and **file_core.S**.

Share the above 6 files with everyone. Programmers can assemble and link them with their projects.

If the linkable files are shared, they can be linked with the linkable files of our project to generate an executable.

This library has 6 files. Say the generated executable is 800 bytes. Consider a huge library with 100 files. This generates an executable of 11000 bytes. When this is run, 11000 bytes of main memory is allocated for this executable to run. Consider another program you wrote, even that uses the same library. Even its executable is about 10000 bytes. When it is run, a 10000 bytes is allocated. Essentially, there are 2 copies of the library in main memory at this point. This is sheer waste. How can this problem be solved?

Common sense tells us that only one copy of the library code and variables should be present in main memory. The programs we write should **share** it.

If we use the library's linkable files while linking our project, we would get one fact executable. We want our executable to remain small and we want the library to be separate. Assuming the library code and data is in main memory, when our executable is run, it should be able to access the library's code without hustle. We need to device a mechanism for it. Look at this. We want our executable to communicate during runtime ie., when our executable calls a function in the library, that function should be called. Whatever we have done so far is everything before running the program. Assembling, Linking labels, functions are all before the program is run. But now, there are functions, labels, variables in some library which our executable should be able to call. How can we make this work?

Let us start with what we know. Let us consider our file manipulation libraryThink what all a library file has. As usual, it should a text section and a data section. Unlike an executable, it does **not** have an entry-point. Instead, it has 5 labels(file_create, open, read, write, close) which needs to be **exposed** to other executables. How do you inform the world about a bunch of labels and thier addresses you have? Through a symbol table! Along with data section and text section, the library can have a

symtab section with details of these 5 labels – their names, types and addresses. This means, a library can be generated in BFF format.

The below is how the file library would look like.

```
-------------------------------------------------------------------------------
0003; bff_hdr_t.bff_sec_n, number of sections = 3
0000; bff_hdr_t.bff_entry_addr = 0x0000, no use of entry
0001; > bff_sec_meta_t.type = BFF_SEC_DATA
XXXX; data.offset
XXXX; data.size
XXXX; data.addr
0000; > bff_sec_meta_t.type = BFF_SEC_TEXT
XXXX; text.offset
XXXX; text.size
XXXX; text.addr
0002; > bff_sec_meta_t.type = BFF_SEC_SYMTAB
XXXX; symtab.offset
XXXX; symtab.size
XXXX; symtab.addr
DDDD; data section starts
DDDD
.
.
DDDD
TTTTTTTTTT; text section starts
TTTTTTTTTT
.
.
TTTTTTTTTT
SSSSSSSSSSSSSSS; symbol table section starts
SSSSSSSSSSSSSSS
.
.
SSSSSSSSSSSSSSS
-------------------------------------------------------------------------------
```

The symbol table would look like this.

Symbol	Type	Address
file_create	label	0x4000
file_open	label	0x4123
file_read	label	0x4345
file_write	label	0x4456
file_close	label	0x4567

Out linker should be able to generate a library, with a few modifications.

```
-------------------------------------------------------------------------------
$ mylinker *.link -o file.lib -lib -expose=file_create,file_write,file_read,file_open,file_close
-------------------------------------------------------------------------------
```

The linker should be modified to generate a library. If the linker is invoked with the **-lib** argument, it

should not search for `_start`(basically the entry-point). Other than that, everything else is the same. In the end, create a symbol table with the labels/variables specified using the **-expose** argument. This can easily be generated using the updated symbol tables used during linking.

We have our library ready. Now coming to our executable. Say we have used the function **file_create**, **file_write**, **file_close** in our project. These symbols will be present in various symbol tables of linkable files. The linker tries to resolve all these symbols only to find out they are not present in our project. It gives a linking error.

We know that these functions are actually present in the library. Suppose we have an instruction `jump file_create`. Because assembler does not find **file_create**, it translates the instruction into `05-00000000`. In all the previous cases, the linker converted `05-00000000` to `05-XXXXXXXX`, where XXXXXXXX is **file_create**'s address. But now, even linker does not find file_create in any of the other linkable files. These unresolved instructions need **fixing**. What it can do is to make a list of all these linker-unresolved symbols and create a symbol table in the executable. As programmers, we can specify in which library these symbols are present in.

Symbol	Type	Address	Library
file_create	undefined	0x0000	file.lib
file_write	undefined	0x0000	file.lib
file_close	undefined	0x0000	file.lib

This is a special version of the symbol table. It contains the library a symbol is present in.

We discussed that the instructions which use the above symbols need fixing. We would need a fixing table too. A fixing table would have the offset at which the bytes need fixing, the number bytes which need fixing and what it needs to be replaced with – basically the symbols.

Offset	Number of bytes	Symbol
X	4	file_create
Y	4	file_write
Z	4	file_close

This means, an executable which uses library functions has 3 sections: data, text, special symbol table and fixing table.

Now let us talk about how these symbols are resolved at runtime.

We run the executable. The Operating System copies all the 4 sections. Simply transferring control to entry-point and running the program won't help. Because when `jump file_create`(**05-00000000**) is executed, **00000000** may not contain any code and the program might crash. Before running, all the fixing table entries should be satisfied. Let us do that.

First entry is at an offset of X bytes into the text section. The 4 bytes need to be replaced by **file_create**'s address. The OS looks at the symbol table. **file_create** is present in the library *file.lib*. If this library is not present in main memory, it is first put into main memory. Check *file.lib*'s symbol table and get **file_create**'s address. With that, replace **00000000** with **file_create**'s address.

Do the same with the other 2 entries. Once all the fixing table entries are satisfied, the program is ready to go. With a little bit of work after linking, and a couple of tables in the executable, we are able to save multiple copies of the same library in the main memory.

What did we do? We performed linking here, but at runtime.This way, programmers can write libraries and reuse the code everywhere. BFF can be modified to fit linkable files, executables and libraries.

We started with programs written in binary language(0s and 1s). Then came assembly language, assembler, symbols, symbol table, fixing table, linkable files, linker, executable, library, runtime linking.

This is one thread of progress. Assembly language and the stuff that came later made programming lot more convenient. We have explored this thread to a certain extent.

Let us explore another very interesting thread.

An **assembly language** is specific to a processor. There are a lot of processors out there - Intel 32-bit, Intel 64-bit, AMD, ARM, Sparc etc., Every processor manufacturer offers an **Instruction Set** to the programmer.

1. Intel i7: x64 Instruction Set
2. AMD Ryzen: x64 Instruction Set
3. Intel Pentium: x86 Instruction Set
4. IBM's PowerPC: PPC, PPC64 Instruction Set
5. Oracle's Sparc: SPARC Instruction Set

It is basically all the instructions and other specifications which can be used to program that particular processor.

Say you want to write a sorting program. You are currently working on an Intel Pentium machine. It is a 32-bit architecture. Intel would have published a Programmer's manual which you can use to write the sorting program. Suppose you want to run a sorting program on an ARM machine. Can you run the program you wrote for the pentium machine on this new ARM machine? By definition, an assembly language is specific to a processor. Because of that, a program written for a Pentium machine cannot be run on an ARM machine. You end up writing a sorting program specifically for the ARM machine. Later, you may have to write the same program for a Sparc machine. Do you observe what is happening? The same program is written again and again with different assembly languages. This type of code is known as **non-portable code**. Code intended for one machine cannot be run on another. This is some time and effort which can be saved if there is a way to run a program you write on all the machines. You write it once and run on all machines. How can this problem be solved?

Back then, operating systems, assemblers, linkers, system utilities everything was written in assembly. Assembly language is way more structured than binary language. But think about implementing complex data structures like trees, graphs, lists in assembly. A doubly linked list's node has 3 members: data(say an integer) and 2 references(to next and previous nodes). At assembly level, a node's members would look like this.

```
--------------------------------------------------------------------------------
data: 0x0000
prev: 0x0000
next: 0x0000
--------------------------------------------------------------------------------
```

If we have an array of structures, it would look like this.

```
--------------------------------------------------------------------------------
data1: 0x0000
prev1: 0x0000
next1: 0x0000
data2: 0x0000
prev2: 0x0000
next2: 0x0000
```

```
.
.
.
dataN: 0x0000
prevN: 0x0000
nextN: 0x0000
--------------------------------------------------------------------------------
```

They are all separate variables, it would help if it is more structured. Having high level data structures like arrays, structs would help.

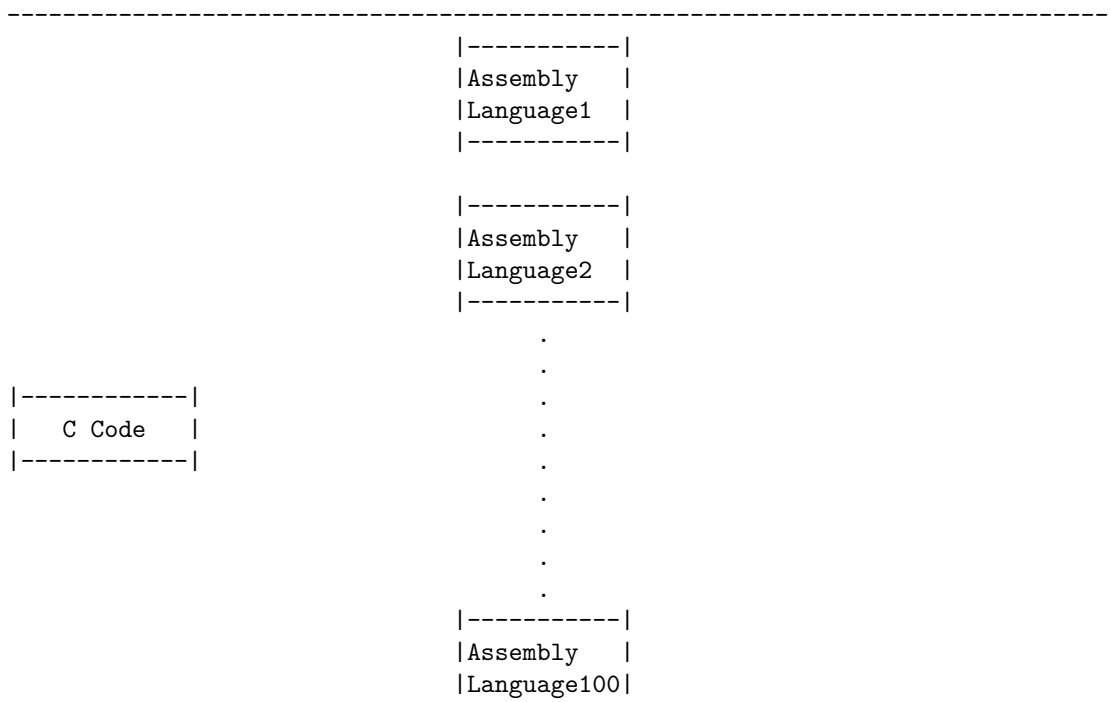
```
--------------------------------------------------------------------------------
node
{
    data;
    prev_ref;
    next_ref;
};
--------------------------------------------------------------------------------
```

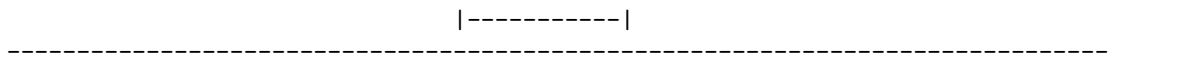
There is a need for a higher level language. It can have helpful, often used programming constructs and data structures. Can that language be portable? Can a program written in this language be run on various architectures?

Enter **Fortran**, **COBOL**, **ALGOL**, **C**. The C programming language was invented in 1969 by the god Dennis Ritchie. The UNIX operating system was completely rewritten in C by Dennis Ritchie and Brian Kernighan. It became a very popular language.

How can code written in one of these high-level languages be run on all the architectures? Think about it.

The following diagram explains the problem.





From the diagram, we can quickly figure out that code written in high-level languages like C can be run on any architecture if **they can be translated into all these assembly languages**. Once there is assembly code, then we know how it is run on the machine.

The way we designed a translator to translate assembly code to machine code, we now need a translator to translate C code to these assembly languages. Such translators are called **compilers**. High-level code is **compiled** into a specified assembly language.

Let us take a an example of a simple `for` loop.

```
for(i = 0; i < 100; i++)
{
    ; Body of the loop
}
```

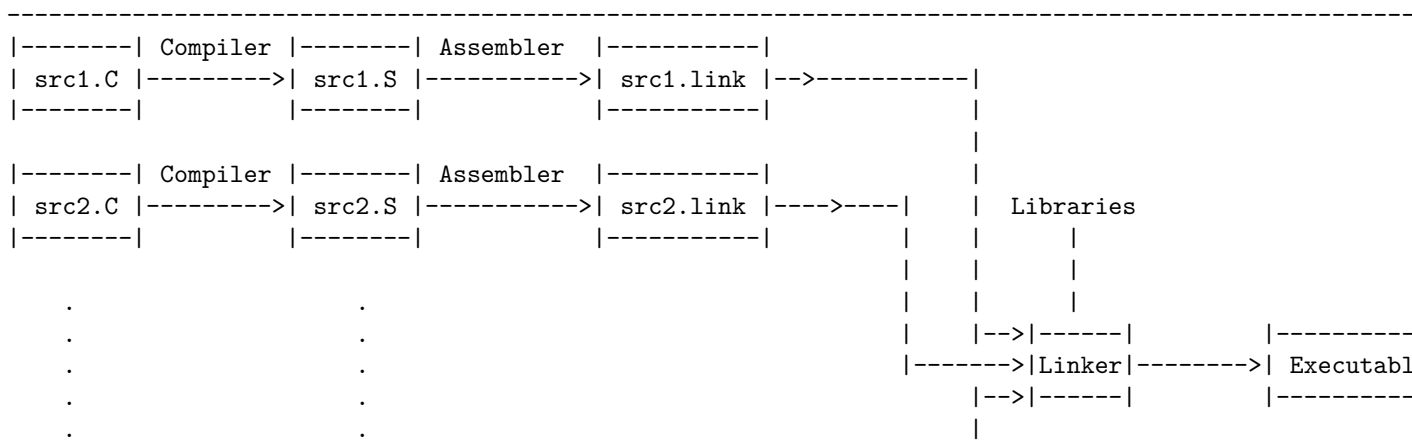
Try writing its assembly equivalent.

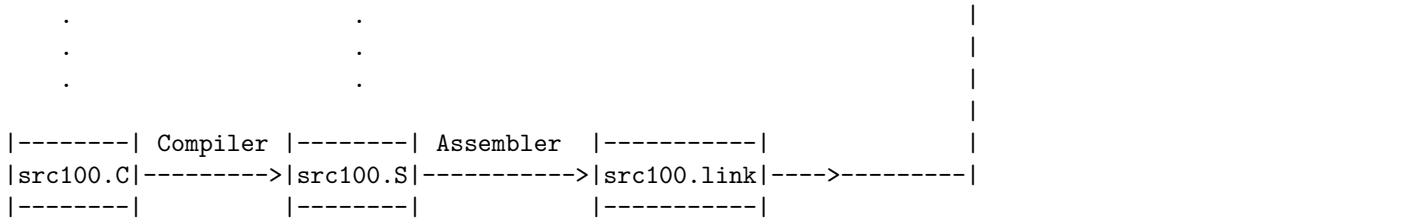
```
    copyc i, 0  ; i = 0
loop:
    cmp i, 100      ; Compare i with 100
    jifeq loop_out  ; if i == 100, jump to loop_out
    ; Body
    ; of
    ; the
    ; loop
    inc i           ; Increment i
    jump loop       ; Simply go back to the label loop

loop_out:
    ; Rest of the program
```

To design portable languages, we need a new program - the **compiler**.

With that, let us summarize this section.





1.2 Overview

In the previous section, we started our exploration with the binary language and then saw high-level languages which can be run on any architecture present. We designed our own toy assembly language, toy binary file format to understand how translation happens.

In this section, we will skim through the process of converting C programs to executables and libraries. We will also see what ELF is in short.

Consider a simple C program *code1.c*.

code1.c

```c
#include <stdio.h>
#include <stdio.h>

#define NUMBER 100

int a = 10;     // Global variable 'a' initialized to 10
int b;          // Global uninitialized variable

int main()
{
        int c = 123, number = NUMBER;
        char d = 'x';

        printf("Hello World!\n");

        return 0;
}
```

We will use **gcc**(GNU C Compiler) for our exploration. Compile our C program and see what we get.

```
chapter1$ gcc-4.8 code1.c -o code1
chapter1$ ls -l
total 16
-rwxr-xr-x 1 dell dell 8440 Mar 22 20:04 code1
-rw-r--r-- 1 dell dell  295 Mar 22 20:04 code1.c
```

I used gcc version 4.8 to generate the executable.

`file` is a command which gives a description about a specified file. Let us use it and check out the executable.

```
--------------------------------------------------------------------------------
chapter1$ file code1
code1: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), dynamically linked,
interpreter /lib64/l, for GNU/Linux 3.2.0,
BuildID[sha1]=db06fa5d56ff35837b7c96ca06294c76261e7468, not stripped
--------------------------------------------------------------------------------
```

This gives good amount of information about the generated executable.

1. It is an ELF file.
2. It targets x86-64 architecture. This implies it targets a 64-bit processor.

In the previous section, we came up with a binary file format(**BFF**) for linkable files, executables, libraries. In *NIX like systems(eg. Linux), a binary file format called **Executable and Linkable Format(ELF)** is used. You can see from its name why it is called so. ELF is used in a variety of operating systems running on a variety of processors.

The `file` command output has a lot of other details. It says it is **dynamically linked**, it talks about some **interpreter**, BuildID. It also says it is **not stripped**.

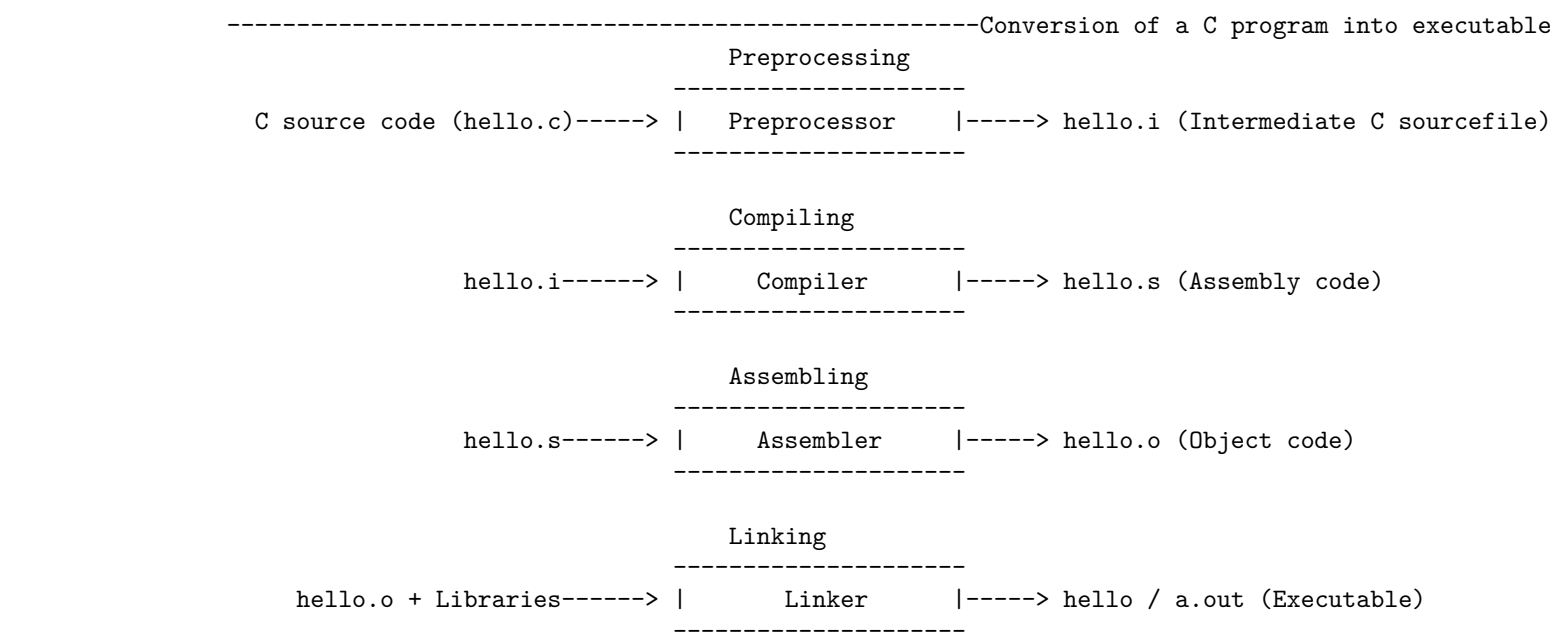
In case you used later versions of gcc(like 6, 7, 8...), your `file` output would look like this.

```
--------------------------------------------------------------------------------
chapter1$ file code1
code1: ELF 64-bit LSB shared object, x86-64, version 1 (SYSV), dynamically linked,
interpreter /lib64/l, for GNU/Linux 3.2.0,
BuildID[sha1]=d2a757dca057ec6d4405f2dc15d130274b3acc2e, not stripped
--------------------------------------------------------------------------------
```

This describes *code1* as a **shared object**, not an executable. Understanding the difference needs some background that will be discussed in later chapters.

Open up the executable with a text editor. See if you can understand anything. It is mostly binary, but has a few character-strings.

Conversion of a C sourcefile to an executable is a **multi-step** process.



The conversion process constitutes of 4 sub-processes. They are **Preprocessing**, **Compiling**, **Assembling** and **Linking**.

We saw in the previous section that when a C program is compiled, few files other than the executable(or library) are generated. But **gcc** obviously generates all those files but it stores the end-product(executable/library) on the filesystem. Let us request the compiler to store all the intermediate files between C sourcefile and executable.

```
--------------------------------------------------------------------------------
chapter1$ gcc code1.c -o code1 --save-temps
chapter1$ ls
code1  code1.c  code1.i  code1.o  code1.s
--------------------------------------------------------------------------------
```

In the next few sections, each of these sub-processes are explored.

1.3 Preprocessing

The **C-Preprocessor**(**CPP**) does the Preprocessing.

CPP takes a normal C sourcefile(*code1.c*) as input and generates an **intermediate** sourcefile(*code1.i*). CPP does a lot of work before sourcefile goes to next sub-process. **gcc** is invoked to generate the executable, it internally invokes the preprocessor, which is a separate program. What exactly does the preprocessor do? Let us take a look at **CPP**'s manual page.

```
------------------------------------------------------------cpp's manual page
chapter1$ man cpp
CPP(1)                                GNU                                CPP(1)

NAME
       cpp - The C Preprocessor
.
.
.
DESCRIPTION
The C preprocessor, often known as cpp, is a macro processor that is used
automatically by the C compiler to transform your program before compilation.
It is called a macro processor because it allows you to define macros,
which are brief abbreviations for longer constructs.
.
.
.
--------------------------------------------------------------------------------
```

The manual page gives all the details about the Preprocessor. It tells that it allows us to define **Macros**. As you read along, you will come to know it helps in doing lot more than that.

Let us use **CPP** on our C sourcefile(*code1.c*) and see what we get.

```
-------------------------------------------------------------------------code1.i
# 1 "code1.c"
# 1 "<built-in>"
# 1 "<command-line>"
```

```
# 31 "<command-line>"
# 1 "/usr/include/stdc-predef.h" 1 3 4
.
.
# 868 "/usr/include/stdio.h" 3 4
# 2 "code1.c" 2

int a = 10;
int b;

int main()
{
        int c = 123, number = 100;
        char d = 'x';

        printf("Hello World!\n");

        return 0;
}
--------------------------------------------------------------------------------
```

The above code has the first few and last few lines of the complete output. You can check *code1.i* for the complete output. The following are the notable differences between the above and the actual C program present in *code1.c*.

1. A lot of stuff is added - stuff which was never part of the original C program.
2. The 2 `#include <stdio.h>` present in *code1.c* have vanished.
3. The macro definition `#define NUMBER  100` is not present.
4. The comments next to the global variables have vanished.
5. There is `number = 100` instead of `number = NUMBER`. This means that macro has been replaced with its actual value.

Let us take up one difference at a time.

All the header files are present in **/usr/include** directory. Checkout **stdio.h**. It has a bunch of **header files**, **macros**, **typedefs**, **library functions**, **declarations** etc., as shown below.

```
------------------------------------------------------------------------stdio.h
/* The possibilities for the third argument to `fseek'.
   These values should not be changed.  */
#define SEEK_SET        0       /* Seek from beginning of file.  */
#define SEEK_CUR        1       /* Seek from current position.  */
#define SEEK_END        2       /* Seek from end of file.  */
.
.
.
typedef __off_t off_t;
.
typedef __off64_t off_t;
.
.
/* Write formatted output to STREAM.

   This function is a possible cancellation point and therefore not
```

```
   marked with __THROW.  */
extern int fprintf (FILE *__restrict __stream,
                    const char *__restrict __format, ...);
/* Write formatted output to stdout.

   This function is a possible cancellation point and therefore not
   marked with __THROW.  */
extern int printf (const char *__restrict __format, ...);
```

Compare the intermediate file(*code1.i*) with *stdio.h*. You will find that stdio.h is **copied** into *code1.i* with a few changes. You will find the above stuff even in *code1.i*.

Now, moving to the next part.

stdio.h starts with the following.

```
#ifndef _STDIO_H
#define _STDIO_H        1
```

and ends with this.

```
#endif /* <stdio.h> included.  */
```

The comment says `"stdio.h included"`. What does this mean? What is `#ifndef`, `#endif`?

We have stumbled upon a piece of preprocessing. It is called **Conditional Compilation**. If you observe, *stdio.h* is included twice in *code1.c*. The header file has macros, typedefs, variables and function declarations. If you include the same file twice, all these macros, variables, declarations will be copied twice. What happens when the same variable(or function) is declared more than once? What happens when the same macro is defined more than once? Check it out. Write a simple program with a variable declared twice. The compiler throws an **error**. This means when the same header is included twice, the **compiler** in the next stage will throw an error. But the executable got generated without any error. How did it happen?

It is because of these.

```
#ifndef _STDIO_H
#define _STDIO_H        1
.
.
.
#endif /* <stdio.h> included.  */
```

When the preprocessor encounters `#include <stdio.h>` for the first time, it starts preprocessing *stdio.h*. Now, preprocessor is inside *stdio.h*. Let us carefully understand what the above lines are doing. `#ifndef _STDIO_H` requests the preprocessor to check if the macro `_STDIO_H` is not defined. It is short for "**if _STDIO_H n**ot** def**ined". If that is true, the preprocessing continues. The next line is `#define _STDIO_H 1`. Now,** _STDIO_H** has a value **1**. The `#ifndef` is ended with an `#endif`. Only if that macro is not defined, everything between the `#ifndef` and its corresponding `#endif` is preprocessed. If the macro is already defined, then preprocessing for *stdio.h* stops here.

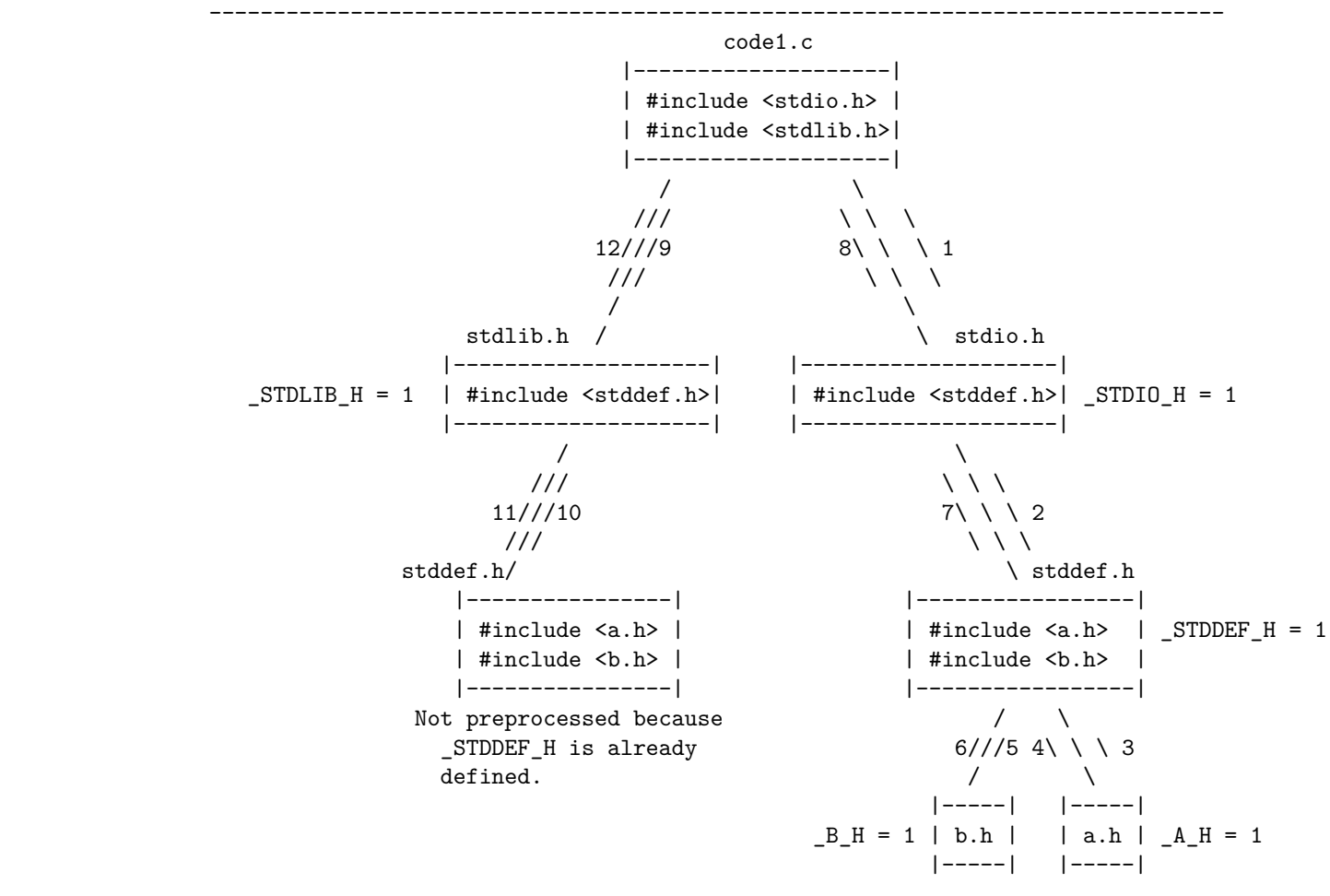
The preprocessor goes back to *code1.c* and it encounters `#include <stdio.h>` for the second time. It starts preprocessing *stdio.h* again. The very first statement is `#ifndef _STDIO_H`. But **_STDIO_H** already has a value **1**. Because its already defined, preprocessing for *stdio.h* stops here.

This way, *stdio.h* is **not** copied into *code1.i*. For the secodn time and thus **no errors**. THis was just one example for Conditional Compilation. Checkout *stdio.h*(or any header file). It will have a lot of such directives - `#ifdef`, `#ifndef`, `#endif`, `#pragma` etc., Preprocessor resolves all these preprocessor-directives.

Let us move on to the next part of preprocessing. The `#define` macros are resolved by the preprocessor. In *code1.c*, whenever the macro **NUMBER** is used, it is replaced by its value **100**.

It removes the comments because they are useless from compilation perspective.

The following diagram summarizes the preprocessing sub-process.



Let us end this section by getting a high-level understanding of preprocessing. Preprocessing happens in a **depth-first** manner. Preprocessor encounters `#include <stdio.h>` in *code1.c*. Preprocessor jumps to preprocess *stdio.h*. Say *stdio.h* has `#include <stddef.h>`. Preprocessor jumps to preprocess *stddef.h*. Once *stddef.h* is preprocessed completely, it comes back to preprocess *stdio.h*. Once preprocessing of *stdio.h* is done, preprocessor comes back to *code1.c* and say it encounters `#include`

`<stdlib.h>`. Preprocessor takes *stdlib.h* for preprocessing. This happens till *code1.c* is entirely preprocessed and a final *code1.i* is generated. *code1.i* will contain contents of *stdio.h*, *stddef.h*, *stdlib.h* etc., in a preprocessed manner - with all preprocessor directives resolved.

At the end of this sub-process, we have a modified C sourcefile which has the all the header files in preprocessed manner and the code in the original C sourcefile.

1.4 Compiling

We generally call the complete converion or a C sourcefile to an executable as Compiling / Compilation. it should be noted that Compiling is one step in the multi-step process.

As we discussed in the Overview, the Compiler takes in *code1.i*(preprocessed C sourcefile, which is also a C sourcefile) as input and gives *code1.s*, as assembly sourcefile as output.

Let us checkout *code1.s* with the `file` command.

```
--------------------------------------------------------------------------------
chapter1$ file code1.s
code1.s: assembler source, ASCII text
--------------------------------------------------------------------------------
```

Let us checkout its contents.

```
--------------------------------------------------------------------------------code1.s
        .file   "code1.c"
        .globl  a
        .data
        .align 4
        .type   a, @object
        .size   a, 4
a:
        .long   10
        .comm   b,4,4
        .section        .rodata
.LC0:
        .string "Hello World!"
        .text
        .globl  main
        .type   main, @function
main:
.LFB0:
        .cfi_startproc
        pushq   %rbp
        .cfi_def_cfa_offset 16
        .cfi_offset 6, -16
        movq    %rsp, %rbp
        .cfi_def_cfa_register 6
        subq    $16, %rsp
        movl    $123, -8(%rbp)
        movl    $100, -4(%rbp)
        movb    $120, -9(%rbp)
        movl    $.LC0, %edi
```

```
        call    puts
        movl    $0, %eax
        leave
        .cfi_def_cfa 7, 8
        ret
        .cfi_endproc
.LFE0:
        .size   main, .-main
        .ident  "GCC: (Ubuntu 4.8.5-4ubuntu8) 4.8.5"
        .section        .note.GNU-stack,"",@progbits
```

This is x86-64 assembly language which is used to program Intel and AMD processors. Let us understand this sourcefile line by line. Please go through *code1.c* again before this analysis.

1. It first specifies the C sourcefile used to generate it.

```
.file   "code1.c"
```

2. This is something interesting. It specifies that the symbol **a** belongs to the **global scope** through the line `.globl  a`.

```
        .globl  a
```

What does this mean? In C, we know that a global variable can be read and written into by any function written in the same sourcefile. It means that the symbol **a** is visible to everyone in the **.text** section. Thinking from assembler's perspective, at this point it does not know anything about **a** except that it is a symbol and it belongs to **global scope**. It does not know that it is a variable.

3. The data section

```
        .data
        .align 4
        .type   a, @object
        .size   a, 4
a:
        .long   10
        .comm   b,4,4
```

The **.data** signifies the beginning of the **.data** section. `.align 4` informs the assembler that this section is 4-byte aligned. What does this mean? Alignment informs us(or tools) how the memory should be viewed and how data needs to be stored in that piece of memory. Take the data section. It is 4-byte aligned. It should be viewed as a piece of memory made of 4-byte chunks.

```
.data:
    <------4 bytes------>
    |----|----|----|----|
    |    |    |    |    |
    |----|----|----|----|
```

```
    |----|----|----|----|
    |    |    |    |    |
    |----|----|----|----|

    |----|----|----|----|
    |    |    |    |    |
    |----|----|----|----|
              .
              .
-------------------------------------------------------------------------------
```

When the program is put into memory, **a** is stored in the first 4 bytes, **b** is stored in the next 4 bytes(starting from byte 5).

If a piece of memory specified to be 8-byte aligned, then it should be viewed as an array of 8-byte chunks.

```
-------------------------------------------------------------------------------
.data:
    <-----------------8 bytes--------------->
    |----|----|----|----|----|----|----|----|
    |    |    |    |    |    |    |    |    |
    |----|----|----|----|----|----|----|----|

    |----|----|----|----|----|----|----|----|
    |    |    |    |    |    |    |    |    |
    |----|----|----|----|----|----|----|----|

    |----|----|----|----|----|----|----|----|
    |    |    |    |    |    |    |    |    |
    |----|----|----|----|----|----|----|----|
                         .
                         .
                         .
-------------------------------------------------------------------------------
```

Continuing, variable **a** is defined.

```
-------------------------------------------------------------------------------
        .type   a, @object
        .size   a, 4
a:
        .long   10
-------------------------------------------------------------------------------
```

a is of the type **object** and its size is 4 bytes. It is then defined to be an integer **10**. `.long` is equivalent to the `int` datatype in C.

Next comes **b**, a uninitialized global variable. It is not defined the way **a** is defined.

```
-------------------------------------------------------------------------------
.comm   b,4,4
-------------------------------------------------------------------------------
```

This informs the assembler that **b** is a **common** symbol, whose size is 4 bytes and should be 4-byte aligned. The first 4 signifies the size of the variable, the second 4 signifies the alignment. Because **b**

is just declared by not defined by a particular value, this information is enough.

`.byte` is an assembler directive lets you specify a byte value. `.quad` lets you specify an 8-byte value.

That was about the data section.

4. Next section is a new one to us.

```
--------------------------------------------------------------------------------
        .section        .rodata
.LC0:
        .string "Hello World!"
--------------------------------------------------------------------------------
```

This section's name is **.rodata**. It stands for **Read-Only Data**. All the read-only data - mostly strings. In this program, the only string present is **Hello World!**. To identify the string, the label `.LC0` is used.

5. Now comes the heart of the program, the **.text** section.

```
--------------------------------------------------------------------------------
        .text
        .globl  main
        .type   main, @function
main:
.LFB0:
        .cfi_startproc
        pushq   %rbp
        .cfi_def_cfa_offset 16
        .cfi_offset 6, -16
        movq    %rsp, %rbp
        .cfi_def_cfa_register 6
        subq    $16, %rsp
        movl    $123, -8(%rbp)
        movl    $100, -4(%rbp)
        movb    $120, -9(%rbp)
        movl    $.LC0, %edi
        call    puts
        movl    $0, %eax
        leave
        .cfi_def_cfa 7, 8
        ret
        .cfi_endproc
.LFE0:
        .size   main, .-main
--------------------------------------------------------------------------------
```

It declares a symbol **main** of type **function**. It is defined as a label. This is x86_64 assembly language, which is used to program 64-bit Intel and AMD processors.

In the end, it's size is calculated with `.size main, .-main`.

6.The compiler version

```
--------------------------------------------------------------------------------
        .ident  "GCC: (Ubuntu 4.8.5-4ubuntu8) 4.8.5"
--------------------------------------------------------------------------------
```

7. In the end, there is one more section **.note.GNU-stack**.

```
------------------------------------------------------------------------------
        .section        .note.GNU-stack,"",@progbits
------------------------------------------------------------------------------
```

We will see what this section is when we discuss about ELF.

What all happens when a C sourcefile is compiled? The following are a few things which happen.

1. C source is converted to a particular assembly source.
2. It performs various types of optimizations, which helps in generating least amount of assembly code for a given C source.
3. It eliminates dead code. Sometimes, there will be pieces of code in your source which will never get executed. Such code is never included in the assembly sourcefile.
4. Probably the most significant one. It **strips** or **removes** names and datatypes of local variables. In *code1.c*, the `main()` function has 3 local variables - `int c = 123, number = NUMBER` and `char d = 'x'`. If you look at assembly code, you will not find any names or explicit datatypes.

We will explore points 2, 3 and 4 in future chapters.

At the end of the compiling sub-process, the intermediate C sourcefile generates an assembly sourcefile which has a data section, a read-only data section, text section and the .note.GNU-stack section.

1.5 Assembling

We now have an assembly sourcefile *code1.s*. It is assembled to give a linkable file *code1.o*. The linkable file is also known as **Object file** or **Relocatable file**.

What does an object file have? In Init, we saw that it has a bunch of sections - text, data, symbol table, fixing table. Open up this file in a text editor. It is binary and hard to understand. For that reason, let us use **readelf** - a tool which will help analyze ELF files to read through *code1.o*.

```
------------------------------------------------------------------------------
chapter1$ readelf -S code1.o
There are 13 section headers, starting at offset 0x2e0:

Section Headers:
  [Nr] Name              Type             Address           Offset
       Size              EntSize          Flags  Link  Info  Align
  [ 0]                   NULL             0000000000000000  00000000
       0000000000000000  0000000000000000           0     0     0
  [ 1] .text             PROGBITS         0000000000000000  00000040
       000000000000002b  0000000000000000  AX       0     0     1
  [ 2] .rela.text        RELA             0000000000000000  00000230
       0000000000000030  0000000000000018   I      10     1     8
  [ 3] .data             PROGBITS         0000000000000000  0000006c
       0000000000000004  0000000000000000  WA       0     0     4
  [ 4] .bss              NOBITS           0000000000000000  00000070
       0000000000000000  0000000000000000  WA       0     0     1
  [ 5] .rodata           PROGBITS         0000000000000000  00000070
       000000000000000d  0000000000000000   A       0     0     1
  [ 6] .comment          PROGBITS         0000000000000000  0000007d
       0000000000000024  0000000000000001  MS       0     0     1
```

```
  [ 7] .note.GNU-stack   PROGBITS         0000000000000000  000000a1
       0000000000000000  0000000000000000           0     0     1
  [ 8] .eh_frame         PROGBITS         0000000000000000  000000a8
       0000000000000038  0000000000000000   A       0     0     8
  [ 9] .rela.eh_frame    RELA             0000000000000000  00000260
       0000000000000018  0000000000000018   I      10     8     8
  [10] .symtab           SYMTAB           0000000000000000  000000e0
       0000000000000138  0000000000000018          11     9     8
  [11] .strtab           STRTAB           0000000000000000  00000218
       0000000000000017  0000000000000000           0     0     1
  [12] .shstrtab         STRTAB           0000000000000000  00000278
       0000000000000061  0000000000000000           0     0     1
Key to Flags:
  W (write), A (alloc), X (execute), M (merge), S (strings), I (info),
--------------------------------------------------------------------------------
```

ELF also has a concept of section headers. Each section header has metadata about a section. In this object file, there are 13 sections. A few which we know are

.text: Has machine code (which may need fixing).

We can dump the contents of a section using **readelf**.

```
--------------------------------------------------------------------------------
chapter1$ readelf --hex-dump .text code1.o

Hex dump of section '.text':
 NOTE: This section has relocations against it, but these have NOT been applied to this dump.
  0x00000000 554889e5 4883ec10 c745f87b 000000c7 UH..H....E.{....
  0x00000010 45fc6400 0000c645 f778bf00 000000e8 E.d....E.x......
  0x00000020 00000000 b8000000 00c9c3            ...........
--------------------------------------------------------------------------------
```

This is pure binary. We know that these are binary equivalent of the assembly instructions. Let us get the assembly equivalent of the above machine code. To get the assembly code back, you **disassemble** the assembled machine code. **objdump** is a useful tool which can used to disassemble assembled code.

```
------------------------------------------------------------------------
chapter1$ objdump -d code1.o

code1.o:     file format elf64-x86-64


Disassembly of section .text:

0000000000000000 <main>:
   0:   55                      push   %rbp
   1:   48 89 e5                mov    %rsp,%rbp
   4:   48 83 ec 10             sub    $0x10,%rsp
   8:   c7 45 f8 7b 00 00 00    movl   $0x7b,-0x8(%rbp)
   f:   c7 45 fc 64 00 00 00    movl   $0x64,-0x4(%rbp)
  16:   c6 45 f7 78             movb   $0x78,-0x9(%rbp)
  1a:   bf 00 00 00 00          mov    $0x0,%edi
  1f:   e8 00 00 00 00          callq  24 <main+0x24>
  24:   b8 00 00 00 00          mov    $0x0,%eax
```

```
  29:   c9                      leaveq
  2a:   c3                      retq
-------------------------------------------------------------------
```

Compare the above assembly code with assembly code present in *code1.s*. There will be some differences. The assembly code in *code1.s* used labels, directives. But this code is derived from assembled code. It means, this code needs fixing. Look at line **0x1f**. Its machine code is **b8-00-00-00-00**. It says `callq 24`. Go back and look at *code1.s* - it is actually `call puts`. Instead of `call`(or `callq`) is encoded as **b8**. The rest of the 4 bytes needs fixing. Those 4 bytes which are zeros need to be replaced with **puts**'s address.

That was about text section.

.data: Data section - Has all the global **initialized** variables.

Lets dump its contents using **readelf**.

```
-------------------------------------------------------------------------------
chapter1$ readelf --hex-dump .data code1.o

Hex dump of section '.data':
  0x00000000 0a000000                             ....
-------------------------------------------------------------------------------
```

0x0a is **10** in hexadecimal system. This is the global variable **a**. **b** is also a global variable. But it is not present here. Where is it?

.rodata: Read-Only data section

The following is its dump.

```
-------------------------------------------------------------------------------
chapter1$ readelf --hex-dump .rodata code1.o

Hex dump of section '.rodata':
  0x00000000 48656c6c 6f20576f 726c6421 00       Hello World!.
-------------------------------------------------------------------------------
```

It has the one read-only element in our program, the **Hello World!** string.

.symtab: Symbol Table

Let us take a look at the symbol table's binary form.

```
-------------------------------------------------------------------------------
chapter1$ readelf --hex-dump .symtab code1.o

Hex dump of section '.symtab':
  0x00000000 00000000 00000000 00000000 00000000 ................
  0x00000010 00000000 00000000 01000000 0400f1ff ................
  0x00000020 00000000 00000000 00000000 00000000 ................
  0x00000030 00000000 03000100 00000000 00000000 ................
  0x00000040 00000000 00000000 00000000 03000300 ................
  0x00000050 00000000 00000000 00000000 00000000 ................
  0x00000060 00000000 03000400 00000000 00000000 ................
  0x00000070 00000000 00000000 00000000 03000500 ................
  0x00000080 00000000 00000000 00000000 00000000 ................
  0x00000090 00000000 03000700 00000000 00000000 ................
```

```
  0x000000a0 00000000 00000000 00000000 03000800 ................
  0x000000b0 00000000 00000000 00000000 00000000 ................
  0x000000c0 00000000 03000600 00000000 00000000 ................
  0x000000d0 00000000 00000000 09000000 11000300 ................
  0x000000e0 00000000 00000000 04000000 00000000 ................
  0x000000f0 0b000000 1100f2ff 04000000 00000000 ................
  0x00000100 04000000 00000000 0d000000 12000100 ................
  0x00000110 00000000 00000000 2b000000 00000000 ........+.......
  0x00000120 12000000 10000000 00000000 00000000 ................
  0x00000130 00000000 00000000                   ........
--------------------------------------------------------------------------------
```

We can't understand anything from it. Let us dump it in human readable form.

```
--------------------------------------------------------------------------------
chapter1$ readelf -s code1.o

Symbol table '.symtab' contains 13 entries:
   Num:    Value          Size Type    Bind   Vis      Ndx Name
     0: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 0000000000000000     0 FILE    LOCAL  DEFAULT  ABS code1.c
     2: 0000000000000000     0 SECTION LOCAL  DEFAULT    1
     3: 0000000000000000     0 SECTION LOCAL  DEFAULT    3
     4: 0000000000000000     0 SECTION LOCAL  DEFAULT    4
     5: 0000000000000000     0 SECTION LOCAL  DEFAULT    5
     6: 0000000000000000     0 SECTION LOCAL  DEFAULT    7
     7: 0000000000000000     0 SECTION LOCAL  DEFAULT    8
     8: 0000000000000000     0 SECTION LOCAL  DEFAULT    6
     9: 0000000000000000     4 OBJECT  GLOBAL DEFAULT    3 a
    10: 0000000000000004     4 OBJECT  GLOBAL DEFAULT  COM b
    11: 0000000000000000    43 FUNC    GLOBAL DEFAULT    1 main
    12: 0000000000000000     0 NOTYPE  GLOBAL DEFAULT  UND puts
--------------------------------------------------------------------------------
```

It has 13 symbols. **Sourcefile-name** is itself a symbol. Come to entry 9. Symbol name is **a**. It is an object of 4 bytes and its scope is **GLOBAL**. Look at entry 11 - it belongs to **main**. It is a function of size 43 bytes with global scope. The most interesting one is entry 12- it belongs to **puts**. There was a `call puts` instruction in *code1.s*, but assembler didn't find **puts** anywhere. It created a symbol table entry of type **NOTYPE** and left.

Now, let us take up the other sections.

.rela.text: This is ELF's version of fixing table. This table has all the entries to fix the .text section.

Object files are also called relocatable files. Fixing formally is called **Relocation**. Take a look at the Relocation records for .text section.

```
--------------------------------------------------------------------------------
chapter1$ objdump -r code1.o

code1.o:     file format elf64-x86-64

RELOCATION RECORDS FOR [.text]:
OFFSET           TYPE              VALUE
000000000000001b R_X86_64_32       .rodata
```

```
0000000000000020 R_X86_64_PC32     puts-0x0000000000000004
-----------------------------------------------------------------------------
```

Take a look at the first entry. It is at an offset **0x1b** bytes into the .text section. Type is **R_X86_64_32** - this has 2 pieces of information. First is that this is a relocation(fixing) specific to x86_64 architecture. Second is 32 bits(4 bytes) need fixing. Let us look at the disassembly and see what is present at byte **0x1b**.

```
-----------------------------------------------------------------------------
Disassembly of section .text:

0000000000000000 <main>:
   0:   55                      push   %rbp
   1:   48 89 e5                mov    %rsp,%rbp
   4:   48 83 ec 10             sub    $0x10,%rsp
   8:   c7 45 f8 7b 00 00 00    movl   $0x7b,-0x8(%rbp)
   f:   c7 45 fc 64 00 00 00    movl   $0x64,-0x4(%rbp)
  16:   c6 45 f7 78             movb   $0x78,-0x9(%rbp)
  1a:   bf 00 00 00 00          mov    $0x0,%edi
  1f:   e8 00 00 00 00          callq  24 <main+0x24>
  24:   b8 00 00 00 00          mov    $0x0,%eax
  29:   c9                      leaveq
  2a:   c3                      retq
-----------------------------------------------------------------------------
```

Byte 0x1b belongs to the instructon **bf-00-00-00-00** - `mov $0x0, %edi`. Take a look at *code1.s*. The 4 bytes need to be replaced with **Hello World!**'s address. That is what the table tells. The value which needs to be put in the place of those 4 zeros is address of .rodata which is the address of the string.

The next entry is to resolve undefined symbol **puts**.

.comment: This section has compiler version information.

You can use **readelf** to dump its contents.

```
-----------------------------------------------------------------------------
chapter1$ readelf --string-dump .comment code1.o

String dump of section '.comment':
  [     1]  GCC: (Ubuntu 4.8.5-4ubuntu8) 4.8.5
-----------------------------------------------------------------------------
```

.strtab: String Table

This section has all the strings used in this object file. The symbols, filename etc., are all stored in this section. Let us take a look at it.

```
-----------------------------------------------------------------------------
chapter1$ readelf --string-dump .strtab code1.o

String dump of section '.strtab':
  [     1]  code1.c
  [     9]  a
  [     b]  b
  [     d]  main
```

```
  [    12]  puts
-------------------------------------------------------------------------------
```

I It has the file name **code1.c** and the 4 symbols we had used - **a**, **b**, **main** and **puts**.

shstrtab: Table of all the section names.

This section is another string table which contains only section names like .text, .data etc.,

```
-------------------------------------------------------------------------------
chapter1$ readelf --string-dump .shstrtab code1.o

String dump of section '.shstrtab':
  [     1]  .symtab
  [     9]  .strtab
  [    11]  .shstrtab
  [    1b]  .rela.text
  [    26]  .data
  [    2c]  .bss
  [    31]  .rodata
  [    39]  .comment
  [    42]  .note.GNU-stack
  [    52]  .rela.eh_frame
-------------------------------------------------------------------------------
```

It is strange that it does not contain **.text**.

That was about the assembling sub-process. It takes in an assembly sourcefile as input and outputs a linkable / relocatable file. The relocatable file contains information in the form of sections. We looked at a few important sections.

1.6 Linking

The final step of the conversion process is **linking**. We discussed that linking is key in resolving symbol references, giving addresses, stitching all the sections in the linkable files to generate the executable or library. Because libraries are shared by various programs, they are called **shared libraries**.

It takes in one or more relocatable files as input and generates an executable or a library as output - based on what is requested.

Before moving forward, let us read the GNU linker's manual page.

```
-------------------------------------------------------------------------------
man ld
LD(1)           GNU Development Tools            LD(1)

NAME
       ld - The GNU linker

SYNOPSIS
       ld [options] objfile ...

DESCRIPTION
       ld combines a number of object and archive files, relocates their data and ties
       up symbol references. Usually the last step in compiling a program is to run ld.
```

```
        .
        .
-------------------------------------------------------------------------------
```

The manual page is a gold mine. It is full of interesting information.

We discussed in Init that linker takes takes essential sections like data, text of one or more relocatable files, give addresses to them, stitch them together, resolve all the symbol references and generates metadata which helps in runtime linking of library functions. In the previous section, we took a look at symbol table, fixing table(relocatable record) used by the linker.

Linker generates an executable *code1*. Let us take a look at the entry-point address. The executable is also an ELF file. The entry-point address for an executable will be present in the ELF header (similar to BFF header). The ELF header has the metadata about the complete ELF file - it is a **roadmap** to the entire file.

```
-------------------------------------------------------------------------------
chapter1$ readelf -h code1
ELF Header:
  Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF64
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              EXEC (Executable file)
  Machine:                           Advanced Micro Devices X86-64
  Version:                           0x1
  Entry point address:               0x400400
  Start of program headers:          64 (bytes into file)
  Start of section headers:          6520 (bytes into file)
  Flags:                             0x0
  Size of this header:               64 (bytes)
  Size of program headers:           56 (bytes)
  Number of program headers:         9
  Size of section headers:           64 (bytes)
  Number of section headers:         30
  Section header string table index: 29
-------------------------------------------------------------------------------
```

The ELF header has a lot of metadata. It tells that this file is a 64-bit executable, following the little-endian byte ordering. It also has a type member. Because relocatable files, executables and libraries all are generated as ELF files, a type member is necessary. The type tells that this is an executable. The machine(architecture) it targets is AMD64. **Its entry-point address is 0x400400**. This means when the program is run, the first instruction will be at the address **0x400400** and control is transfered to that instruction.

There is something interesting here. So far, we discussed quite a bit about sections, section headers. We saw that relocatable files are **divided** into several **sections**. Executables and libraries can be copied onto memory and can be run. Such type of **runnable** files are divided into what are called **program segments**. Sections help the linker to generate the executable/library. Segments help the operating system run them. What do these segments contain? Do they have headers? Let us explore them now.

From the ELF header, it can be seen that there are **9** number of **Program Headers**. Program

Headers are short form of **Program Segment Headers**. Each of these headers describe a particular segment. Let us list the 9 segments present in this file. Again, **readelf** can be used.

```
--------------------------------------------------------------------------------
chapter1$ readelf --segments code1

Elf file type is EXEC (Executable file)
Entry point 0x400400
There are 9 program headers, starting at offset 64

Program Headers:
  Type           Offset             VirtAddr           PhysAddr
                 FileSiz            MemSiz              Flags  Align
  PHDR           0x0000000000000040 0x0000000000400040 0x0000000000400040
                 0x00000000000001f8 0x00000000000001f8  R      0x8
  INTERP         0x0000000000000238 0x0000000000400238 0x0000000000400238
                 0x000000000000001c 0x000000000000001c  R      0x1
      [Requesting program interpreter: /lib64/ld-linux-x86-64.so.2]
  LOAD           0x0000000000000000 0x0000000000400000 0x0000000000400000
                 0x0000000000000700 0x0000000000000700  R E    0x200000
  LOAD           0x0000000000000e08 0x0000000000600e08 0x0000000000600e08
                 0x000000000000022c 0x0000000000000238  RW     0x200000
  DYNAMIC        0x0000000000000e20 0x0000000000600e20 0x0000000000600e20
                 0x00000000000001d0 0x00000000000001d0  RW     0x8
  NOTE           0x0000000000000254 0x0000000000400254 0x0000000000400254
                 0x0000000000000044 0x0000000000000044  R      0x4
  GNU_EH_FRAME   0x00000000000005c4 0x00000000004005c4 0x00000000004005c4
                 0x000000000000003c 0x000000000000003c  R      0x4
  GNU_STACK      0x0000000000000000 0x0000000000000000 0x0000000000000000
                 0x0000000000000000 0x0000000000000000  RW     0x10
  GNU_RELRO      0x0000000000000e08 0x0000000000600e08 0x0000000000600e08
                 0x00000000000001f8 0x00000000000001f8  R      0x1

 Section to Segment mapping:
  Segment Sections...
   00
   01     .interp
   02     .interp .note.ABI-tag .note.gnu.build-id .gnu.hash .dynsym .dynstr .gnu.version .gnu.versio
   03     .init_array .fini_array .jcr .dynamic .got .got.plt .data .bss
   04     .dynamic
   05     .note.ABI-tag .note.gnu.build-id
   06     .eh_frame_hdr
   07
   08     .init_array .fini_array .jcr .dynamic .got
--------------------------------------------------------------------------------
```

There are different types of segments - **PHDR**, **INTERP**, **LOAD**, **DYNAMIC** etc., The **linker** generates these program headers and puts it into the executable. See if relocatable files have program headers.

Dividing an executable/library into segments helps in running it. To analyze them, it is still useful to divide into sections. That is why, generally an executable/library is divided into both segments and sections. A segment would generally contains one or more sections. Look at that Section to Segment

mapping. Segment-1 contains some section called the **.interp**. Segment-2 is a collection of lot of sections.

Below is the list of sections.

```
--------------------------------------------------------------------------------
chapter1$ readelf -S code1
There are 30 section headers, starting at offset 0x1978:

Section Headers:
  [Nr] Name              Type             Address           Offset
       Size              EntSize          Flags  Link  Info  Align
  [ 0]                   NULL             0000000000000000  00000000
       0000000000000000  0000000000000000           0     0     0
  [ 1] .interp           PROGBITS         0000000000400238  00000238
       000000000000001c  0000000000000000   A       0     0     1
  [ 2] .note.ABI-tag     NOTE             0000000000400254  00000254
       0000000000000020  0000000000000000   A       0     0     4
  [ 3] .note.gnu.build-i NOTE             0000000000400274  00000274
       0000000000000024  0000000000000000   A       0     0     4
  [ 4] .gnu.hash         GNU_HASH         0000000000400298  00000298
       000000000000001c  0000000000000000   A       5     0     8
  [ 5] .dynsym           DYNSYM           00000000004002b8  000002b8
       0000000000000060  0000000000000018   A       6     1     8
  [ 6] .dynstr           STRTAB           0000000000400318  00000318
       000000000000003d  0000000000000000   A       0     0     1
  [ 7] .gnu.version      VERSYM           0000000000400356  00000356
       0000000000000008  0000000000000002   A       5     0     2
  [ 8] .gnu.version_r    VERNEED          0000000000400360  00000360
       0000000000000020  0000000000000000   A       6     1     8
  [ 9] .rela.dyn         RELA             0000000000400380  00000380
       0000000000000030  0000000000000018   A       5     0     8
  [10] .rela.plt         RELA             00000000004003b0  000003b0
       0000000000000018  0000000000000018  AI       5    23     8
  [11] .init             PROGBITS         00000000004003c8  000003c8
       0000000000000017  0000000000000000  AX       0     0     4
  [12] .plt              PROGBITS         00000000004003e0  000003e0
       0000000000000020  0000000000000010  AX       0     0     16
  [13] .text             PROGBITS         0000000000400400  00000400
       00000000000001a2  0000000000000000  AX       0     0     16
  [14] .fini             PROGBITS         00000000004005a4  000005a4
       0000000000000009  0000000000000000  AX       0     0     4
  [15] .rodata           PROGBITS         00000000004005b0  000005b0
       0000000000000011  0000000000000000   A       0     0     4
  [16] .eh_frame_hdr     PROGBITS         00000000004005c4  000005c4
       000000000000003c  0000000000000000   A       0     0     4
  [17] .eh_frame         PROGBITS         0000000000400600  00000600
       0000000000000100  0000000000000000   A       0     0     8
  [18] .init_array       INIT_ARRAY       0000000000600e08  00000e08
       0000000000000008  0000000000000008  WA       0     0     8
  [19] .fini_array       FINI_ARRAY       0000000000600e10  00000e10
       0000000000000008  0000000000000008  WA       0     0     8
  [20] .jcr              PROGBITS         0000000000600e18  00000e18
```

```
       0000000000000008  0000000000000000  WA       0     0     8
  [21] .dynamic          DYNAMIC          0000000000600e20  00000e20
       00000000000001d0  0000000000000010  WA       6     0     8
  [22] .got              PROGBITS         0000000000600ff0  00000ff0
       0000000000000010  0000000000000008  WA       0     0     8
  [23] .got.plt          PROGBITS         0000000000601000  00001000
       0000000000000020  0000000000000008  WA       0     0     8
  [24] .data             PROGBITS         0000000000601020  00001020
       0000000000000014  0000000000000000  WA       0     0     8
  [25] .bss              NOBITS           0000000000601034  00001034
       000000000000000c  0000000000000000  WA       0     0     4
  [26] .comment          PROGBITS         0000000000000000  00001034
       0000000000000023  0000000000000001  MS       0     0     1
  [27] .symtab           SYMTAB           0000000000000000  00001058
       0000000000000630  0000000000000018          28    46     8
  [28] .strtab           STRTAB           0000000000000000  00001688
       00000000000001e4  0000000000000000           0     0     1
  [29] .shstrtab         STRTAB           0000000000000000  0000186c
       0000000000000108  0000000000000000           0     0     1
Key to Flags:
  W (write), A (alloc), X (execute), M (merge), S (strings), I (info),
  L (link order), O (extra OS processing required), G (group), T (TLS),
  C (compressed), x (unknown), o (OS specific), E (exclude),
  l (large), p (processor specific)
--------------------------------------------------------------------------------
```

Again note that segments are the ones which help in running the executable/library. Sections are just another way to partition it - which helps in analyzing the file better.

Let us start with something we have discussed. Lets starts with the type of symbols present in *code1*. Use **readelf** to dump them.

```
--------------------------------------------------------------------------------
chapter1$ readelf -s code1

Symbol table '.dynsym' contains 4 entries:
   Num:    Value          Size Type    Bind   Vis      Ndx Name
     0: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND puts@GLIBC_2.2.5 (2)
     2: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND __libc_start_main@GLIBC_2.2.5 (2)
     3: 0000000000000000     0 NOTYPE  WEAK   DEFAULT  UND __gmon_start__

Symbol table '.symtab' contains 66 entries:
   Num:    Value          Size Type    Bind   Vis      Ndx Name
     0: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 0000000000400238     0 SECTION LOCAL  DEFAULT    1
     2: 0000000000400254     0 SECTION LOCAL  DEFAULT    2
     3: 0000000000400274     0 SECTION LOCAL  DEFAULT    3
     4: 0000000000400298     0 SECTION LOCAL  DEFAULT    4
     5: 00000000004002b8     0 SECTION LOCAL  DEFAULT    5
     6: 0000000000400318     0 SECTION LOCAL  DEFAULT    6
     7: 0000000000400356     0 SECTION LOCAL  DEFAULT    7
     8: 0000000000400360     0 SECTION LOCAL  DEFAULT    8
```

```
 9: 0000000000400380     0 SECTION LOCAL  DEFAULT    9
10: 00000000004003b0     0 SECTION LOCAL  DEFAULT   10
11: 00000000004003c8     0 SECTION LOCAL  DEFAULT   11
12: 00000000004003e0     0 SECTION LOCAL  DEFAULT   12
13: 0000000000400400     0 SECTION LOCAL  DEFAULT   13
14: 00000000004005a4     0 SECTION LOCAL  DEFAULT   14
15: 00000000004005b0     0 SECTION LOCAL  DEFAULT   15
16: 00000000004005c4     0 SECTION LOCAL  DEFAULT   16
17: 0000000000400600     0 SECTION LOCAL  DEFAULT   17
18: 0000000000600e08     0 SECTION LOCAL  DEFAULT   18
19: 0000000000600e10     0 SECTION LOCAL  DEFAULT   19
20: 0000000000600e18     0 SECTION LOCAL  DEFAULT   20
21: 0000000000600e20     0 SECTION LOCAL  DEFAULT   21
22: 0000000000600ff0     0 SECTION LOCAL  DEFAULT   22
23: 0000000000601000     0 SECTION LOCAL  DEFAULT   23
24: 0000000000601020     0 SECTION LOCAL  DEFAULT   24
25: 0000000000601034     0 SECTION LOCAL  DEFAULT   25
26: 0000000000000000     0 SECTION LOCAL  DEFAULT   26
27: 0000000000000000     0 FILE    LOCAL  DEFAULT  ABS crtstuff.c
28: 0000000000600e18     0 OBJECT  LOCAL  DEFAULT   20 __JCR_LIST__
29: 0000000000400440     0 FUNC    LOCAL  DEFAULT   13 deregister_tm_clones
30: 0000000000400470     0 FUNC    LOCAL  DEFAULT   13 register_tm_clones
31: 00000000004004b0     0 FUNC    LOCAL  DEFAULT   13 __do_global_dtors_aux
32: 0000000000601034     1 OBJECT  LOCAL  DEFAULT   25 completed.7098
33: 0000000000600e10     0 OBJECT  LOCAL  DEFAULT   19 __do_global_dtors_aux_fin
34: 00000000004004d0     0 FUNC    LOCAL  DEFAULT   13 frame_dummy
35: 0000000000600e08     0 OBJECT  LOCAL  DEFAULT   18 __frame_dummy_init_array_
36: 0000000000000000     0 FILE    LOCAL  DEFAULT  ABS code1.c
37: 0000000000000000     0 FILE    LOCAL  DEFAULT  ABS crtstuff.c
38: 00000000004006fc     0 OBJECT  LOCAL  DEFAULT   17 __FRAME_END__
39: 0000000000600e18     0 OBJECT  LOCAL  DEFAULT   20 __JCR_END__
40: 0000000000000000     0 FILE    LOCAL  DEFAULT  ABS
41: 0000000000600e10     0 NOTYPE  LOCAL  DEFAULT   18 __init_array_end
42: 0000000000600e20     0 OBJECT  LOCAL  DEFAULT   21 _DYNAMIC
43: 0000000000600e08     0 NOTYPE  LOCAL  DEFAULT   18 __init_array_start
44: 00000000004005c4     0 NOTYPE  LOCAL  DEFAULT   16 __GNU_EH_FRAME_HDR
45: 0000000000601000     0 OBJECT  LOCAL  DEFAULT   23 _GLOBAL_OFFSET_TABLE_
46: 00000000004005a0     2 FUNC    GLOBAL DEFAULT   13 __libc_csu_fini
47: 0000000000601020     0 NOTYPE  WEAK   DEFAULT   24 data_start
48: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND puts@@GLIBC_2.2.5
49: 0000000000601038     4 OBJECT  GLOBAL DEFAULT   25 b
50: 0000000000601034     0 NOTYPE  GLOBAL DEFAULT   24 _edata
51: 00000000004005a4     0 FUNC    GLOBAL DEFAULT   14 _fini
52: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND __libc_start_main@@GLIBC_
53: 0000000000601020     0 NOTYPE  GLOBAL DEFAULT   24 __data_start
54: 0000000000000000     0 NOTYPE  WEAK   DEFAULT  UND __gmon_start__
55: 0000000000601028     0 OBJECT  GLOBAL HIDDEN    24 __dso_handle
56: 00000000004005b0     4 OBJECT  GLOBAL DEFAULT   15 _IO_stdin_used
57: 0000000000400530   101 FUNC    GLOBAL DEFAULT   13 __libc_csu_init
58: 0000000000601040     0 NOTYPE  GLOBAL DEFAULT   25 _end
59: 0000000000400430     2 FUNC    GLOBAL HIDDEN    13 _dl_relocate_static_pie
60: 0000000000400400    43 FUNC    GLOBAL DEFAULT   13 _start
```

```
    61: 0000000000601030     4 OBJECT  GLOBAL DEFAULT   24 a
    62: 0000000000601034     0 NOTYPE  GLOBAL DEFAULT   25 __bss_start
    63: 00000000004004fd    43 FUNC    GLOBAL DEFAULT   13 main
    64: 0000000000601038     0 OBJECT  GLOBAL HIDDEN    24 __TMC_END__
    65: 00000000004003c8     0 FUNC    GLOBAL DEFAULT   11 _init
-------------------------------------------------------------------------------
```

We discussed in Init that an executable will contain symbols that need to be resolved at runtime. This type of runtime linking is called **dynamic linking**. We discussed a type of symbol resolution where all the symbols are resolved before the program is run. But in **dynamic linking**, a symbol is resolved only when it is referenced. For example, there is a call to `printf`. In the method we discussed before, its address is obtained before running the program. Better method is to resolve only when it is referenced. What if it is never called? It never has to be resolved. Be lazy, do work only if needed, when needed. This type of linking is also called **lazy linking**.

Observe the above readelf output. It dumped 2 types of symbol tables.

1. **.dynsym**:

```
-------------------------------------------------------------------------------
Symbol table '.dynsym' contains 4 entries:
   Num:    Value          Size Type    Bind   Vis      Ndx Name
     0: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND puts@GLIBC_2.2.5 (2)
     2: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND __libc_start_main@GLIBC_2.2.5 (2)
     3: 0000000000000000     0 NOTYPE  WEAK   DEFAULT  UND __gmon_start__
-------------------------------------------------------------------------------
```

As expected, it has **puts**. It has figured out that **puts** is present in the C library. The other 2 entries are added by GNU Linker.

2. **.symtab**: The normal symbol table

This symbol table has lots of symbols. It has undefined symbols like **puts**, **___libc_start_main** - basically which are present in **.dynsym**. It also has symbols which are resolved and have absolute addresses - it has **a**, **b**, **main**. When they are already resolved, why are they present? It helps in debugging. They are **not necessary** to run it. Even if you remove it, the program should run just fine. Let us test that. Run the program once. Also make note of the executable's size.

```
-------------------------------------------------------------------------------
chapter1$ ls -l code1
-rwxr-xr-x 1 dell dell 8440 Mar 23 00:53 code1
chapter1$ ./code1
Hello World!
-------------------------------------------------------------------------------
```

It is running properly. There is a tool called **strip** which can be used to remove(or strip) any specified section. Let us remove the **.symtab** section.

```
-------------------------------------------------------------------------------
chapter1$ strip --remove-section .symtab code1
dell@adwi:~/Documents/projects/books/binary-exploitation/exp/chapter1$ ls -l code1
-rwxr-xr-x 1 dell dell 6224 Mar 23 01:15 code1
-------------------------------------------------------------------------------
```

Look at how fat the symbol table was. List all the sections and make sure this section is not present in the executable. If you checkout the symbols using `readelf -s` command, you will get only the

.dynsym which **must** be present.

Along with a symbol table containing unresolved symbols, a fixing table(relocation records) should be present which will help in runtime linking.

```
------------------------------------------------------------------------------
chapter1$ readelf -r code1

Relocation section '.rela.dyn' at offset 0x380 contains 2 entries:
  Offset          Info           Type           Sym. Value    Sym. Name + Addend
000000600ff0  000200000006 R_X86_64_GLOB_DAT 0000000000000000 __libc_start_main@GLIBC_2.2.5 + 0
000000600ff8  000300000006 R_X86_64_GLOB_DAT 0000000000000000 __gmon_start__ + 0

Relocation section '.rela.plt' at offset 0x3b0 contains 1 entry:
  Offset          Info           Type           Sym. Value    Sym. Name + Addend
000000601018  000100000007 R_X86_64_JUMP_SLO 0000000000000000 puts@GLIBC_2.2.5 + 0
------------------------------------------------------------------------------
```

There you go. For every unresolved symbol in **.dynsym**, there is a relocation record.

This is not enough. We need to know the symbol-library relationship - basically information which tells that **puts** is defined in a particular library. These strings are present in the **.dynstr** section.

```
------------------------------------------------------------------------------
chapter1$ readelf --string-dump .dynstr code1

String dump of section '.dynstr':
  [     1]  libc.so.6
  [     b]  puts
  [    10]  __libc_start_main
  [    22]  GLIBC_2.2.5
  [    2e]  __gmon_start__
------------------------------------------------------------------------------
```

That was about information required for proper dynamic linking.

Checkout the .data and .text sections too.

Going back to exploring segments, the following is the list.

1. PHDR
2. INTERP
3. LOAD
4. LOAD
5. DYNAMIC
6. NOTE
7. GNU_EH_FRAME
8. GNU_STACK
9. GNU_RELRO

Each of these segments has an essential piece of the executable/library without which it cannot be run. The **LOAD** type segments are directly copied into the main memory. The **DYNAMIC** segment has information about dynamic linking - The symbol table with unresolved symbols, library-list of functions/variables, a hash-table which helps in accessing symbol table efficiently and more. The **NOTE** has some information about the file. The **GNU_STACK** and **GNU_RELRO** are necessary to administer important security features. Other segment types will be discussed in future chapters.

We discussed that the section-wise division of an executable does not help in running it. How do we verify it?

The section-headers has information about how the executable is divided into sections - what part of executable is what section. Without section-headers, all this information is lost. Even with that information lost, we should be able to run it. Let us delate all the section-headers in *code1*.

The size of *code1* is 8448 bytes. From the ELF header, the section-headers array starts at a byte offet 6520. There are 30 section headers each of size 64 bytes. Total size of section-headers array is 30X64 = 1920 bytes. By observation, 6520+1920 = 8440 = *code1*'s size. This means, the section-headers array starts from byte 6520 and goes till the end of the file. We will simply chop it off and see if the program can still run. We can use this command called **dd** to get the job done.

```
--------------------------------------------------------------------------------
chapter1$ dd if=code1 of=code1.nosec bs=1 count=6520
6520+0 records in
6520+0 records out
6520 bytes (6.5 kB, 6.4 KiB) copied, 0.0357093 s, 183 kB/s
--------------------------------------------------------------------------------
```

The first 6520 bytes are copied into *code1.nosec*. Try running this new file.

```
--------------------------------------------------------------------------------
chapter1$ chmod u+x code1.nosec
dell@adwi:~/Documents/projects/books/binary-exploitation/exp/chapter1$ ./code1.nosec
Hello World!
--------------------------------------------------------------------------------
```

Look at that. Section headers are not needed.

Experiment with program headers. You will see that corrupting these headers won't allow you to run the program. You will mostly probably end up with a runtime error.

That was about linking. We skimmed through contents of an executable, introduced the concept of segments and did some experimentation. Successful linking of the given bunch of relocatable files will give you an executable/library which can be run. Linker also adds information essential for dynamic linking.

1.7 Further exploration

1.7.1 Entry Point of a program

We saw that there should be an entry-point for any executable. For a C program, it is logical to think that the `main()` function is its entry-point. Is it really the entry point? Let us dig deep.

Using *code1* for this. We saw that the ELF header has the entry-point address. Let us check it out.

```
--------------------------------------------------------------------------------
chapter1$ readelf -h code1
ELF Header:
  Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF64
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
```

```
  ABI Version:                       0
  Type:                              EXEC (Executable file)
  Machine:                           Advanced Micro Devices X86-64
  Version:                           0x1
  Entry point address:               0x400400
  Start of program headers:          64 (bytes into file)
  Start of section headers:          6520 (bytes into file)
  Flags:                             0x0
  Size of this header:               64 (bytes)
  Size of program headers:           56 (bytes)
  Number of program headers:         9
  Size of section headers:           64 (bytes)
  Number of section headers:         30
  Section header string table index: 29
-------------------------------------------------------------------------------
```

The entry-point address is **0x400400**. Let us see if we can get a symbol in the symbol table for this address.

```
-------------------------------------------------------------------------------
chapter1$ readelf -s code1
.
.
    58: 0000000000601040     0 NOTYPE  GLOBAL DEFAULT   25 _end
    59: 0000000000400430     2 FUNC    GLOBAL HIDDEN    13 _dl_relocate_static_pie
    60: 0000000000400400    43 FUNC    GLOBAL DEFAULT   13 _start
    61: 0000000000601030     4 OBJECT  GLOBAL DEFAULT   24 a
    62: 0000000000601034     0 NOTYPE  GLOBAL DEFAULT   25 __bss_start
    63: 00000000004004fd    43 FUNC    GLOBAL DEFAULT   13 main
.
.
-------------------------------------------------------------------------------
```

The `_start` symbol is given the address **0x400400**. `main`'s address is **0x4004fd**. This means `_start` is the starting point of this executable. But we never defined the `_start`. All we did is define the `main` function. How do we explain this?

Compare the disassembly of *code1.o* and *code1*. The text section of *code1.o* has `main` but *code1* has lot of other things including the `_start*`.

```
-------------------------------------------------------------------------------
Disassembly of section .text:

0000000000400400 <_start>:
  400400:   31 ed                   xor    ebp,ebp
  400402:   49 89 d1                mov    r9,rdx
  400405:   5e                      pop    rsi
  400406:   48 89 e2                mov    rdx,rsp
  400409:   48 83 e4 f0             and    rsp,0xfffffffffffffff0
  40040d:   50                      push   rax
  40040e:   54                      push   rsp
  40040f:   49 c7 c0 a0 05 40 00    mov    r8,0x4005a0
  400416:   48 c7 c1 30 05 40 00    mov    rcx,0x400530
  40041d:   48 c7 c7 fd 04 40 00    mov    rdi,0x4004fd
```

```
  400424:   ff 15 c6 0b 20 00       call   QWORD PTR [rip+0x200bc6]        # 600ff0 <__libc_start_mai
  40042a:   f4                      hlt
  40042b:   0f 1f 44 00 00          nop    DWORD PTR [rax+rax*1+0x0]
--------------------------------------------------------------------------------
```

This code was **added** by the linker. Execution starts with `_start`, later `main` is called. `main` is the entry-point from programmer's perspective, `_start` is the entry-point from system's perspective.

Linker's manual page has some details related to this.

```
--------------------------------------------------------------------------------
       -e entry
       --entry=entry
           Use entry as the explicit symbol for beginning execution of your program, rather than the
           point.  If there is no symbol named entry, the linker will try to parse entry as a number,
           as the entry address (the number will be interpreted in base 10; you may use a leading 0x
           or a leading 0 for base 8).
--------------------------------------------------------------------------------
```

There is an option to specify the entry-point. Let us try that out.

```
--------------------------------------------------------------------------------
chapter1$ gcc-4.8 code1.c -o code1.newstart --entry=main
--------------------------------------------------------------------------------
```

Checkout the ELF header.

```
--------------------------------------------------------------------------------
chapter1$ readelf -h code1.newstart
ELF Header:
  Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF64
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              EXEC (Executable file)
  Machine:                           Advanced Micro Devices X86-64
  Version:                           0x1
  Entry point address:               0x4004fd
  Start of program headers:          64 (bytes into file)
  Start of section headers:          6520 (bytes into file)
  Flags:                             0x0
  Size of this header:               64 (bytes)
  Size of program headers:           56 (bytes)
  Number of program headers:         9
  Size of section headers:           64 (bytes)
  Number of section headers:         30
  Section header string table index: 29

chapter1$ readelf -s code1
.
.
    60: 0000000000400400    43 FUNC    GLOBAL DEFAULT   13 _start
    61: 0000000000601030     4 OBJECT  GLOBAL DEFAULT   24 a
```

```
    62: 0000000000601034     0 NOTYPE  GLOBAL DEFAULT   25 __bss_start
    63: 00000000004004fd    43 FUNC    GLOBAL DEFAULT   13 main
.
.
--------------------------------------------------------------------------------
```

Now, entry-point is **main**. Lets run it.

```
--------------------------------------------------------------------------------
chapter1$ ./code1.newstart
Hello World!
Segmentation fault (core dumped)
--------------------------------------------------------------------------------
```

The program ran, but it crashed.

To understand this, we will have to see what was executed between **_start** and **main**.

1.7.2 Can intermediate files be generated separately?

There are 3 intermediate files which are generated when a C sourcefile is compiled - preprocessed sourcefile(**.i**), assembly sourcefile(**.s**) and relocatable file(**.o**). They were stored on the filesystem by using the –**save-temps** compiler option.

It helps sometimes to generate each of these files separately. Let us see how it can be done.

When **gcc** is invoked to generate the executable/library, it feels that it is generating all the intermediate files. But that is not the case. It is invoking other programs which generate these files.

1. Preprocessed sourcefile:

The **cpp** can be separately invoked and any code can be just preprocessed. Take a look.

```
--------------------------------------------------------------------------------
chapter1$ cpp
#define NUMBER 100
int a = NUMBER;
int main()
{
    int b = NUMBER;
    printf("Hello!\n");
    return 0;
}
^D
# 1 "<stdin>"
# 1 "<built-in>"
# 1 "<command-line>"
# 31 "<command-line>"
# 1 "/usr/include/stdc-predef.h" 1 3 4
# 32 "<command-line>" 2
# 1 "<stdin>"

int a = 100;
int main()
{
```

```
 int b = 100;
 printf("Hello!\n");
 return 0;
}
```

Look at that. Preprocessed. Invoking **cpp** on *code1.c **will give** code1.i**.

2. Assembly sourcefiles

gcc actually compiles the preprocessed sourcefile to assembly file. The **-S** option can be used to generate just the **.s** file.

```
chapter1$ ls code3*
code3.c

chapter1$ gcc code3.c -S

chapter1$ ls code3*
code3.c  code3.s
```

3. Relocatable files

There are 2 ways to generate **.o** files. **gcc** along with **-c** option can be used to generate just the relocatable file.

```
chapter1$ rm code3.s

chapter1$ gcc -c code3.c

chapter1$ ls code3*
code3.c  code3.o
```

1.7.3 How are libraries made?

We focused on executable generation in previous sections. Let us take a look at how a shared object is generated.

Consider the following example.

```
lib.c
#include <stdio.h>

void lib_func()
{
    printf("Inside lib_func()\n");
}
```

It is a dummy library with one function. Let us generate a library out of it.

1. Generate the corresponding relocatable file with a special option **fPIC**.

```
-------------------------------------------------------------------------------
chapter1$ ls lib*
lib.c
chapter1$ gcc-4.8 lib.c -c -fPIC
chapter1$ ls lib*
lib.c  lib.o
-------------------------------------------------------------------------------
```

PIC stands for **Position Independent Code**. For executables, we saw that linker gave addresses. Here, we are requesting the operating system to give addresses using the **fPIC** option. Why is this necessary? This will be discussed in one of the future chapters. Position Independent Code itself is a very interesting topic and there are a lot of other concepts attached to it.

Now, let us request the linker to generate a shared library using the –**shared** option.

```
-------------------------------------------------------------------------------
~/Documents/projects/books/binary-exploitation/exp/chapter1$ gcc-4.8 lib.o -o libdummy.so --shared
chapter1$ file libdummy.so
libdummy.so: ELF 64-bit LSB shared object, x86-64, version 1 (SYSV), dynamically linked, BuildID[sha1
-------------------------------------------------------------------------------
```

With that, we have our shared library ready. A shared library is also called a shared object. I want you to analyze *libdummy.so* using readelf, objdump etc., the way we analyzed *code1*.

Every library has corresponding header file(s). The header files informs the programmer what functions, data structures are offered by that library. Let us write a small header file *lib.h*.

```
-------------------------------------------------------------------------------lib.h
#ifndef _LIB_H
#define _LIB_H 1

void lib_func();

#endif  /* _LIB_H */
-------------------------------------------------------------------------------
```

Let us use the above library and write a program.

```
-------------------------------------------------------------------------code2.c
#include "lib.h"
#include <stdio.h>

int main()
{
    printf("Inside main, before lib_func()\n");
    lib_func();
    printf("Inside main, after lib_func()\n");

    return 0;
}
-------------------------------------------------------------------------------
```

Try compiling *code2.c*.

```
-------------------------------------------------------------------------------
chapter1$ gcc-4.8 code2.c -o code2
/tmp/cczqD3tj.o: In function `main':
```

```
code2.c:(.text+0x16): undefined reference to `lib_func'
collect2: error: ld returned 1 exit status
--------------------------------------------------------------------------------
```

As expected. We got an **undefined reference** error. The symbol **lib_func** is used, but it is not defined anywhere in *code2.c*. We expect this to be resolved at runtime. We know that it is defined in *libdummy.so*, but we need to inform the linker about this right? We need to specify where our library is present and the name of the library.

```
--------------------------------------------------------------------------------
chapter1$ gcc-4.8 code2.c -o code2 -L. -ldummy
chapter1$ ls -l code2
-rwxr-xr-x 1 dell dell 8336 Mar 23 12:54 code2
--------------------------------------------------------------------------------
```

-L option is used to specify the location of the library. **-l** option is used to specify the library. It successfully compiled to give the executable. Try running it.

```
--------------------------------------------------------------------------------
chapter1$ ./code2
./code2: error while loading shared libraries: libdummy.so: cannot open shared object file: No such f
--------------------------------------------------------------------------------
```

You compiled properly. The loader is responsible for loading all the libraries a program depends on. The loader does not know where *libdummy.so* is. Let us inform it. The `LD_LIBRARY_PATH` environment variable can be updated. We can update it this way.

```
--------------------------------------------------------------------------------
chapter1$ export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:.
--------------------------------------------------------------------------------
```

The **.** is representative of the present working directory. Add it to the environment variable. Now let us run the program.

```
--------------------------------------------------------------------------------
chapter1$ ./code2
Inside main, before lib_func()
Inside lib_func()
Inside main, after lib_func()
--------------------------------------------------------------------------------
```

Check the dynamic linking information present in *code2* using readelf, objdump.

1.7.4 Other binary file formats?

In this chapter, we created a crude binary file format to understand certain concepts. We also explored parts of ELF. Are there other binary file formats? What format is used for Windows executables? for macOS?

There are lot of other binary file formats out there.

Windows uses an Executable Format called **Portable Executable(PE)** and object file format called **Common Object File Format(COFF)**. Apple uses **Mach object file format(Mach-O)**.

These are the popular formats. Many operating systems use their own formats.

Now that you have a general understanding of ELF, you can go ahead and try reading about PE, Mach-O. They are all similar in a sense: All have the basic sections, symbol table, string table, relocatable table, file-header etc., Even if there are differences, you should be able to understand it.

1.8 Fini

With that, we have come to the end of this chapter.

We explored the 4 steps involved in converting C programs into executables. The 4 steps are Pre-processing, Compiling, Assembling and Linking. We vaguely explored the Executable and Linkable Format(ELF).

A few important concepts were conveniently avoided during the chapter. When we were discussing about addresses and memory in Init, it was not specified what type of addresses we are talking about - physical or virtual addresses. In the swap program we wrote, all 3 were global variables. I did not mention anything about local variables. And a few things were pushed to future chapters because they needed background. ELF is a complex file format. It is a tightly knit pack of variety of data structures and exploring it in detail will have to be done in another chapter. Because binary file formats are closely bound to the compilation process, A few details of ELF was introduced to understand the internals of compiling. Reading and understanding assembly code was not the aim of this chapter, so it was avoided. I kept telling that the OS copies program into memory. Today's operating system does the bare minimum to copy a program into memory. It checks the validity of the program(does it abide by the rules of file format, OS etc.,). If it does, then it invokes a helper program to do the rest - copy the complete program into memory, check its dependent libraries and copy them, transfer control to entry-point of the program, dynamic linking etc.,. Exploring all this will take up a complete chapter. To compile *code1*, gcc-4.8 was used but my system comes with gcc-7. The differences between their outputs is difficult to explain without background - so gcc-4.8 was used.

When I first tried to understand Internals of Compiling with gcc, readelf, objdump etc., it felt pretty heavy and hard to understand. Understanding ELF was difficult because it has lots of things in it. Simply listing that a relocatable file has text, data, symbol table, fixing table etc., felt dry. Even once I knew why a symbol table is used, why a fixing table is used, it was not satisfying. Why is something present? What problem did it solve? Needed an answer to that. The Init section was a very joyous experiment. Life started out with 0s and 1s. We now have a complex binary file format which tells how programs should be. I know the starting point and the ending point. There was a direction - whatever was done, it was to make programmers' life easier, run programs faster, easy debugging. With that, a path was followed in Init to slowly build from 0s and 1s to a binary file format. I don't know if this path was taken to develop ELF starting from 0s and 1s, but I hope that the Init section gives atleast an intuition in that direction.

1.9 Further Reading

1. Manual pages of
   - elf
   - as
   - ld
   - any tool you use. There is always something interesting!
2. ELF related resources
   - ELF Specification
   - Linux Foundation Referenced Specifications

- Introduction to ELF by RedHat

3. Resources related to other binary file formats
   - Mach-O File format reference
   - Official doc for Windows' PE

Chapter 2: Introduction to Assembly Programming

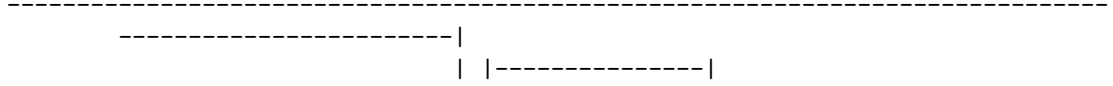
Summary
Every architecture offers an Instruction Set which can be used to program it. This chapter is about understanding what an Instruction Set is and learning the x86/x64 Instruction Set in particular. Finally writing programs for Intel/AMD processors.

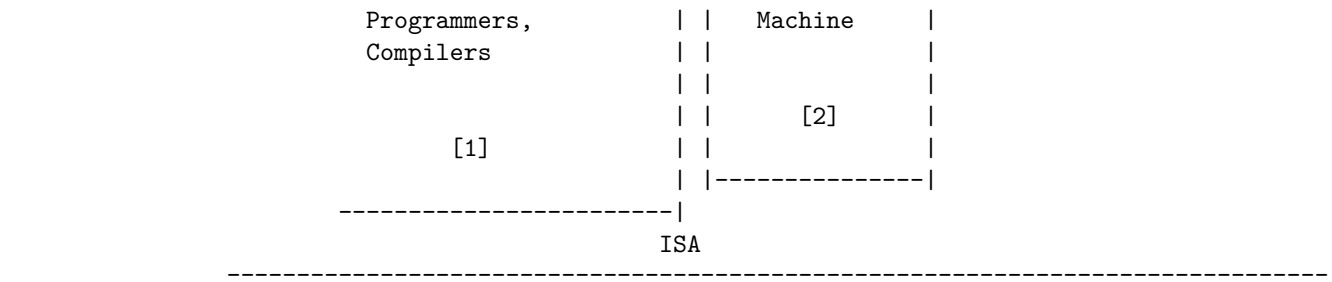
2.1 Init

Let us start with a question. What is a programming language? A programming language is a set of instructions and constructs which can be used to program machines. That machine can be hardware, it can be implemented in software(virtual machines). In this chapter, we will be exploring the type of programming language used to program processors.

In Chapter1's Init, we defined our own set of instructions and rules to program an imaginary processor. It had 256 instructions - which can be encoded with 1 byte. We just didn't specify instructions, we also specified a lot of other things. We specified a way to access data - only through addresses. We specified that operands of most instructions(add, sub, mul, copy, jump etc.,) are memory addresses. We defined a datatype of size 2 bytes. We can go ahead and define all 256 instructions, how they are encoded, their operands, datatypes etc., With all this, a programmer can write programs which can be run on that processor. All this is collectively called an **Instruction Set**. You write a program using this well defined instruction set, but then what would you want to do? You obviously want to run it. Just having a well defined instruction set is not enough. There should be a **machine** on which you can run that program. Basically, that machine should understand the instruction set and run the instructions listed in the program. For this to happen, that machine should **implement** this instruction set. What is that machine? Is it a piece of hardware? Is it another software program?

The machine which implements that Instruction Set be seen from 2 sides. One is from programmer's side. All the programmer sees is the Instruction Set - whatever is given to him to write programs. This programmer's perspective of a machine is called **Architecture**. That is why, it is generally refered to as **Instruction Set Architecture(ISA)**. Another view is the actual implementation of the machine - how is each instruction implemented? How are the arithmetic instructions implemented? What algorithms are used? etc., This view is called **Organization**. Take a look at the following diagram.



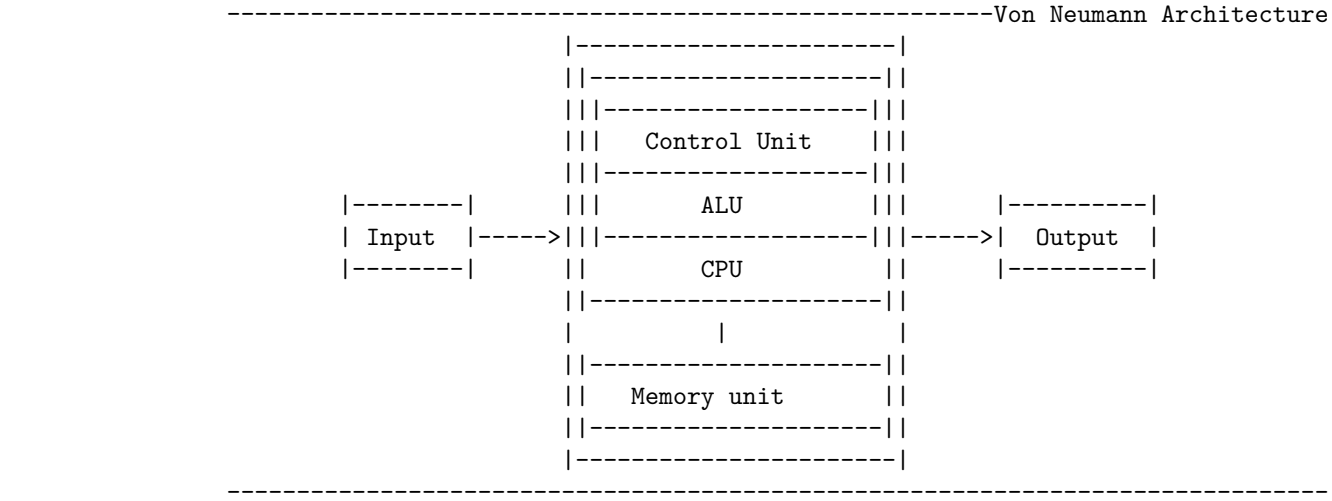


Let us take a simple example.

In our instruction set, the `mul` instruction multiplies contents at the 2 operands and stores it back into the memory pointed by the first operand. Its syntax is `mul address1, address2`.

Consider two processor manufacturers(you and me) implement the same instruction set. Consider the `mul` instruction's implementation. There are various multiplication algorithms which can be implemented at the hardware level. I decide to implement algorithm1, you decide to implement algorithm2. There are 2 perspectives. From programmer's perspective, the internal implementation does not matter. The processor manufacturer gives an instruction to multiply 2 numbers and thats all the programmer is concerned about.

Let us go a bit deeper into what what this machine contains which makes it capable of running programs. All modern machines follow an architecture called as **Von Neumann Architecture**.



Let us first understand the design.

1. It has one compute unit - the CPU which is present to execute instructions.
2. It has a memory unit to store instructions, data, results etc.,
3. It has Input and Output - the way users interact with the system.

Based on this, we can design what all instructions need to be included in the Instruction Set.

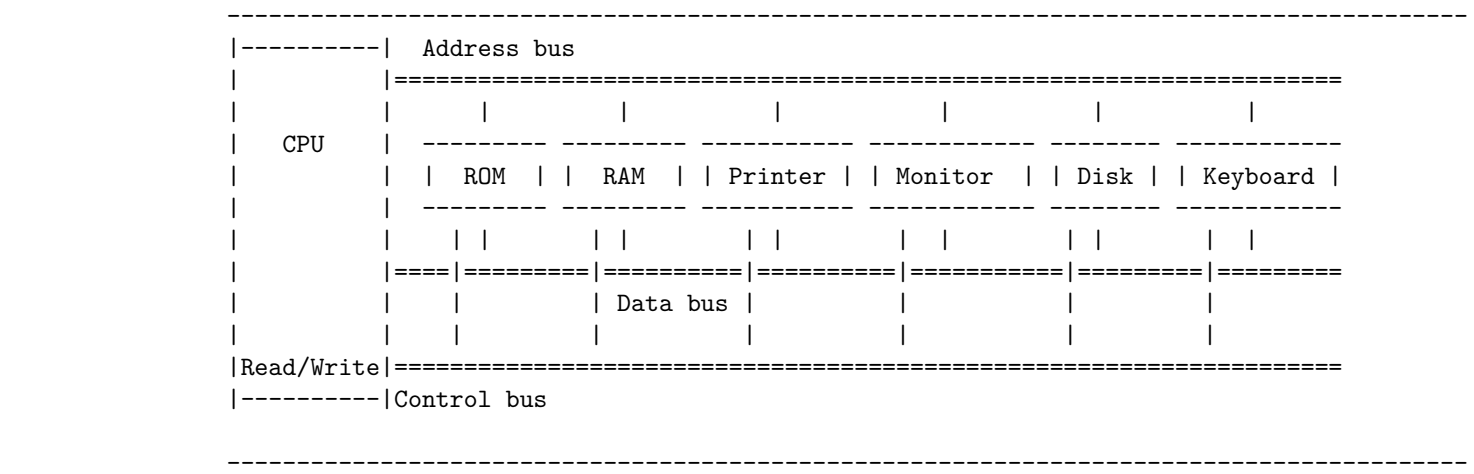
1. There is an **Arithmetic and Logic Unit(ALU)**. This means the CPU has hardware for Arithmetic and Logical operations like Addition, Multiplication, Division, Bit manipulation etc., The instruction set should contain ALU-type instructions through which programmers can use the ALU.

2. There is a connection between the **Memory Unit** and CPU. This means, there should be

- **Memory Access Instructions**: Which are required to load values from the memory unit and store back value/results back to the memory unit.
- Some memory manipulation instructions.

3. To write programs with conditional statements, loops, functions, interrupts etc., there is a need for **Control Instructions**. These instructions handle the control flow of a program.

How does the CPU(processor) communicate with the memory unit, input-output devices? For any device(memory unit or input-output device) to communicate with the CPU, it should be identified first by the CPU. CPU does that by giving **unique addresses** to the memory unit and input-output devices. The CPU is connected to these devices through a series of wires called **bus**. There are 3 types of buses: **Address bus**, **Data bus** and **Control bus**. Got this beautiful diagram from here.



Let us discuss the above diagram in detail.

1. **Read Only Memory(ROM)**: This is a piece of memory present in the Motherboard. As the name suggests, it is read-only. An exception is raised if we try writing into it.
2. **Random Access Memory(RAM)**: This it the main memory present in our computers. Every byte in it has a unique address(aka it is **byte addressable**). Data can be read from and written to it.
3. **Printer** and **Monitor** are output devices. Data can only be written to these devices. The output devices process our input and generates suitable output. For example, we send a PDF file to the printer to print it - this is the input. The printer processes that input and prints it - this is the output.
4. **Disk** is a storage device: Data can be read from and written to it.
5. **Keyboard** is an input device. Data can only be read from it. As we type something, the CPU can take that as data-input.

When a device is connected, the CPU gives a unique set of addresses to that device. These addresses are used to communicate with that device.

Let us take an example. Consider a system where

1. ROM size is 1MiB.
2. RAM size is 4GiB.
3. Assume printer would need 1Ki addresses.
4. Monitor would need 1Ki addresses.
5. Disk would need 1Ki of addresses.

6. Keyboard would need 512 addresses.

The CPU starts alloting the requested number of addresses to these devices. Starting from **0**, Address space of

1. ROM: 0x0 - 0xfffff, size = 0x100000 = 1,048,576
2. RAM: 0x100000 - 0x1000fffff, size = 0x100000000 = 4,294,967,296
3. Printer: 0x100100000 - 0x1001003ff, size = 0x400 = 1,024
4. Monitor: 0x100100400 - 0x1001007ff, size = 0x400 = 1,024
5. Disk: 0x100100800 - 0x100100bff, size = 0x400 = 1,024
6. Keyboard: 0x100100c00 - 0x100100dff, size = 0x200 = 512

Complete address space: 0x0 - 0x100100dff, total address space size = 0x100100e00. This address space is called **Physical Address Space**.

The following is my laptop's physical address space.

```
------------------------------------------------------------------------------
00000000-00000fff : Reserved
00001000-00057fff : System RAM
00058000-00058fff : Reserved
00059000-0005efff : System RAM
0005f000-0005ffff : Reserved
00060000-0009efff : System RAM
0009f000-0009ffff : Reserved
000a0000-000bffff : PCI Bus 0000:00
000c0000-000cfdff : Video ROM
000d0000-000d0fff : Adapter ROM
000f0000-000fffff : System ROM
00100000-bd336fff : System RAM
bd337000-bd7b2fff : Reserved
bd7b3000-d9dbafff : System RAM
d9dbb000-d9ef2fff : Reserved
d9ef3000-d9f15fff : ACPI Tables
d9f16000-da686fff : System RAM
da687000-dade2fff : ACPI Non-volatile Storage
dade3000-dba62fff : Reserved
dba63000-dbafefff : Unknown E820 type
dbaff000-dbaffffff : System RAM
dbb00000-dbffffff : RAM buffer
dd000000-df7fffff : Reserved
  dd800000-df7fffff : Graphics Stolen Memory
df800000-feafffff : PCI Bus 0000:00
  df800000-df9fffff : PCI Bus 0000:01
  dfa00000-dfbfffff : PCI Bus 0000:01
  dfc00000-dfdfffff : PCI Bus 0000:02
  dfe00000-dfffffff : PCI Bus 0000:03
  e0000000-efffffff : 0000:00:02.0
  f0000000-f01fffff : PCI Bus 0000:04
  f0200000-f03fffff : PCI Bus 0000:04
  f6000000-f6ffffff : 0000:00:02.0
  f7000000-f70fffff : PCI Bus 0000:03
    f7000000-f7001fff : 0000:03:00.0
      f7000000-f7001fff : iwlwifi
```

```
  f7100000-f71fffff : PCI Bus 0000:02
    f7100000-f7103fff : 0000:02:00.0
    f7104000-f7104fff : 0000:02:00.0
      f7104000-f7104fff : r8169
  f7200000-f720ffff : 0000:00:14.0
    f7200000-f720ffff : xhci-hcd
  f7210000-f7213fff : 0000:00:1b.0
    f7210000-f7213fff : ICH HD audio
  f7214000-f7217fff : 0000:00:03.0
    f7214000-f7217fff : ICH HD audio
  f7218000-f72180ff : 0000:00:1f.3
  f7219000-f72197ff : 0000:00:1f.2
    f7219000-f72197ff : ahci
  f721a000-f721a3ff : 0000:00:1d.0
    f721a000-f721a3ff : ehci_hcd
  f721c000-f721c01f : 0000:00:16.0
    f721c000-f721c01f : mei_me
  f7fe0000-f7feffff : pnp 00:05
  f7ff0000-f7ffffff : pnp 00:05
  f8000000-fbffffff : PCI MMCONFIG 0000 [bus 00-3f]
    f8000000-fbffffff : Reserved
      f8000000-fbffffff : pnp 00:05
fec00000-fec00fff : Reserved
  fec00000-fec003ff : IOAPIC 0
fed00000-fed03fff : Reserved
  fed00000-fed003ff : HPET 0
    fed00000-fed003ff : PNP0103:00
fed10000-fed17fff : pnp 00:05
fed18000-fed18fff : pnp 00:05
fed19000-fed19fff : pnp 00:05
fed1c000-fed1ffff : Reserved
  fed1c000-fed1ffff : pnp 00:05
    fed1f410-fed1f414 : iTCO_wdt.0.auto
    fed1f800-fed1f9ff : intel-spi
fed20000-fed3ffff : pnp 00:05
fed45000-fed8ffff : pnp 00:05
fed90000-fed90fff : dmar0
fed91000-fed91fff : dmar1
fee00000-fee00fff : Local APIC
  fee00000-fee00fff : Reserved
ff000000-ffffffff : Reserved
  ff000000-ffffffff : INT0800:00
    ff000000-ffffffff : pnp 00:05
100000000-11f7fffff : System RAM
  101800000-102600e60 : Kernel code
  102600e61-10305063f : Kernel data
  10330b000-1037fffff : Kernel bss
11f800000-11fffffff : RAM buffer
------------------------------------------------------------------------------
```

Look at that. There are a bunch of PCI buses, there is Video ROM, System RAM and lot more.

Coming back, CPU has now alloted addresses to all these devices. How exactly does communication

happen? Communication is nothing but moving data to and from the device. Suppose I want to **write** some **data** into a particular **address**. The following is done.

1. Set Address Bus = **address**
2. Set Data Bus = **data**
3. Control Bus can be either set to **Read/Write**. In this example, we set it to **Write**.

Suppose the CPU wants to **read** from an **address**. The Address Bus is set to the **address** and control bus is set to **Read**. The device which has that address places **data** at that address.

In short, this is how communication happens.

Obviously, the CPU can't support infinite number of addresses. How many addresses can a CPU support? ie., What is the size of the physical address space it can support? **This depends on the size of the Address Bus**. If the address bus' size is 4 bits, then the size of physical address space is (2^4) = 16. If the address bus is 32 bits, then size of physical address space is 2^32 = 4294967296.

The size of the data bus decides what type of processor it is. If the length of data bus is 4-bits, then it is a 4-bit processor. Now a days, we mostly deal with 32-bit and 64-bit processors.

Coming back to our discussion on Instruction Sets, we will be exploring the x86 instruction set. The x86 instruction set is a 32-bit instruction set - the data-unit size is 32-bits. Intel and AMD implement these instruction sets in their processors. This instruction sets defines a rich set of instructions, a bunch of datatypes, various addressing modes, exception handling instructions, I/O methods and more. All these can be used by the programmer or can be used to build a compiler.

2.2 What is x86?

To answer this question, let us take a quick look at history of Intel's processors.

- **Intel 4004**: Intel's one of the first processors. It was a 4-bit processor.
- **Intel 8080**: 8-bit processor.
- **Intel 8086**: A 16-bit processor. Intel provided an Instruction Set. The 8080 was named to 808**6** because it is a 1**6**-bit processor. This is a very interesting processor. We will specifically talk about this at the end of this chapter.
- **Intel 80186**, **Intel 80286**: These are also 16-bit processors.
- **Intel APX 432**: First 32-bit microprocessor by Intel. This design was discontinued.
- **Intel 80386**: A 32-bit microprocessor. This because very famous in the market. It made its mark.
- **Intel 80486**: 80386's successor.
- Then came the legendary **Intel 80586** or **Pentium**!
- Then came Pentium Pro, Pentium II, Pentium III etc.,
- Soon after that, 64-bit processors came into market.

The point is, 8086 had a 16-bit ISA. When Intel introduced 32-bit microprocessors(80386 and later), they came up with a 32-bit ISA which was an **Extension** of the older 16-bit ISA.

8086, 80186, 80286, 80386, 80486, 80586 are the series of microprocessors. Later, they named the ISA as **x86** ISA where **86** stands for the actual 86 in the series and x is like a variable here.

What does extensions mean?

A few new instructions are added to the old ISA to support the new hardware design. This means, all the old instructions will be able to run on the new processor unless they have removed it from the new ISA for some reason. This is called **Backward Compatibility**.

2.3 The x86 ISA

2.3.1 Registers

Before moving to instructions, we need to understand what a **Register** is.

A **Register** is a small storage space **on the chip** of the microprocessor. Generally, there will be multiple registers on the chip. All the registers collectively is known as **Register File**.

Why do you need such storage when there is a proper memory unit? Because the memory unit is outside the microprocessor, it takes certain amount of time(aka latency) to access it, it is better to have on-chip storage for fast access.

The x86 ISA provides **8** registers each of size **32-bits**. They are

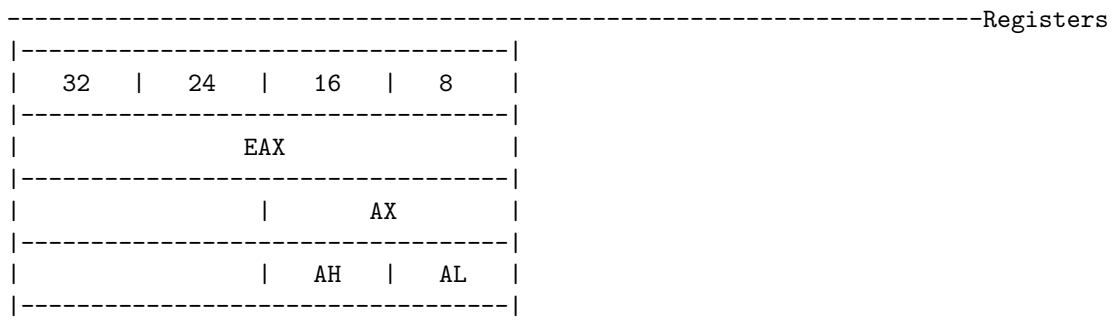
1. **eax**: The **a** stands for **accumulator**. An accumulator is a register which is used to store results of certain operations.
2. **ecx**: **c** stands for **counter**. Certain instructions used this register as counter(or iterator) in loops.
3. **edx**: **d** stands for **data**.
4. **ebx**: **b** stands for **base**. Certain instructions store the base value in this register.
5. **esi**: **Source Index** register. Used in certain string related instructions.
6. **edi**: **Destination Index** register. Used in certain string instructions.
7. **ebp**: **Base Pointer**
8. **esp**: **Stack Pointer**

It should be noted that **only** certain instructions use each of these registers in a particular manner which has been mentioned.

These registers are known as **General Purpose Registers(GPRs)**, but **ebp** and **rsp** are almost never used for general purposes. They have very specific purposes.

Along with the above registers, there are 2 more registers - **eip(Instruction Pointer)** and **eflags**. They have very specific purpose and that is why they are called **Special Purpose Registers**.

Some history about the ISA. When 8-bit processors came into the market, registers were named **a**, **b**, **c**, **d**. They were renamed as **ax**, **bx**, **cx**, **dx** for 16-bit processors. The **x** stands for e**x**tended. When 32-bit processors were designed, the 16-bit set was extended. The **e** in each of those 32-bit registers means **e**xtended.



A programmer(or compiler) can use 8-bit, 16-bit versions of registers in 32-bit programs. For example, **ax**, **si**, **al** can be used in 32-bit programs. You will understand this better when we write programs.

Now coming to the special purpose registers,

1. **eip**:

- Instruction Pointer
- The register stores the address of the next instruction to be executed.

2. **eflags**:

- This is a set of **status** flags(or bits). Each flag can either be 0(not set) or 1(set). The following are a few flags. * **zf** - The zero flag: This is set when the result of an operation is zero. If the result is not zero, it is cleared. * **cf** - The carry flag: This is set when the result of an operation is too small or too large for destination operand. * **sf** - The sign flag: This is set when the result of an operation is negative. If the result is positive, it is cleared. The flag value is equal to the **most-significant bit** of the result(in 2's complement representation).

2.3.2 Instructions

Instructions can be broadly classified into 4 types:

1. Arithmetic and Logical istructions: `add`, `sub`, `imul`, `idiv`, `or`, `xor`, `and` etc.,
2. Data flow instructions: `mov`, `lea`, `push`, `pop`.
3. Control Flow instructions: `cmp`, `jmp` and its siblings, `call`, `ret`.
4. Interrupts: `syscall`, `int`, `sysenter`.

We will take a look at each instruction in detail in the next section.

2.4 x86 Assembly Language

2.4.1 Difference between ISA and Assembly language

What we discussed so far was about x86 ISA. What is the difference between ISA and assembly language?

The ISA is a document which has a list of all the registers, instructions, addressing modes etc., Assembly language is a programming language made out of the ISA. To understand this, let us go back to our toy ISA in Chapter1's Init. It has 256 instructions, operands must be memory addresses, 1 datatype of 2 bytes. We took that Instruction Set and slowly came up with an assembly language. We split the program into data and text sections, programmers were allowed to use variable-names, labels and the assembler took care of it. Similar to this, modern assemblers have a lot more features - macros, directives, different syntax of the same ISA. A standard is converted into a language.

Though there is only one x86 ISA, there are a variety of assemblers which offer their own versions of the x86 assembly language. There are a few x86 assemblers out there like Netwide Assembler(nasm), Microsoft Assembler(masm), Turbo Assembler(tasm), Yasm Modular Assembler(yasm), GNU Portable Assembler(as) etc., Each of these assemblers offers a variety of features. We will be using **nasm** throughout the chapter.

Installing **nasm** is just one command away.

```
--------------------------------------------------------------------------------
$ sudo apt-get install nasm
--------------------------------------------------------------------------------
```

2.4.2 Datatypes

At the assembly level, we will be dealing with bytes. Datatypes like **char**, **int**, **long int** etc., are not present at assembly level. Then, how are these datatypes translated to their assembly equivalent? This is done by **accessing the specific number of bytes a particular datatype in C represents**. Let us take a few examples to understand this.

In 32 systems, `sizeof(short int)` = **2 bytes**. As the assembly level, there is no integer datatype. There is only a stream of bytes. Suppose **a** is an ***short int** variable whose address is present in the register **ebx**. If you want to load the contents of variable **a** into another register(say **cx**), this is how its done.

```
--------------------------------------------------------------------------------
mov cx, word [ebx]
--------------------------------------------------------------------------------
```

- `mov` is the instruction.
- `cx` is the destination register.
- A **word** means **2** bytes.
- The instruction tells the assembler to consider **ebx** to be a word pointer. That is, assume that it points to 2 bytes. So, when it is used in an instruction, 4 bytes pointed by ebx is loaded.

Let us take another example.

Let **b** be the **int** variable. In a 32-bit machine, `sizeof(int)` = **4 bytes**. Suppose you want to perform an operation `b = b + 0x123`. Assuming address of **b** is present in **rax**, this is how its done.

```
--------------------------------------------------------------------------------
add dword[rax], 0x123
--------------------------------------------------------------------------------
```

- `add` is the instruction.
- 0x123 is the constant to be added.
- **dword** is short for **double word** which is 4 bytes.
- This instruction tells the assembler to consider **rax** to be a double word pointer. That is, consider the first 4 bytes pointed by it. So, when it is used in this instruction, all 4 bytes are taken, 0x123 is added to it and then put back into the memory.

Let us put these in a formal manner.

1. How is the size of data measured at assembly level?

- **1 byte** is the small piece of memory that can be accessed.
- **byte** stands for 1 byte.
- **word** stands for 2 bytes.
- **dword / double word** stands for 4 bytes.

Remember that the data-bus size of a 32-bit system is 32-bits. This means you can bring in a maximum of 32-bits of data at a time from RAM. 8-bits, 16-bits of data can also be brought in using a 32-bit long data bus. That is the idea.

2. The following are the methods to access memory.

- `byte[REG]`: This tells the assembler to consider **REG** as a byte pointer. Consider the first byte it is pointing to. Any operation performed with this as an operand will take only 1 byte directly pointed by **REG**.
- `word[REG]`: This tells the assembler to consider **REG** as a word pointer.
- `dword[REG]`: Tells the assembler to consider 4 bytes pointed by **REG**.

2.4.3 Instructions

In the previous section, we listed a few instructions. Now, let us see how exactly to use them, their operands.

Most of the instruction operand on operand(s). These operands can be

1. **Registers**: These are mostly general purpose registers discussed in the previous section. There are a few special registers which are also used, we will discuss about them later.
2. **Immediate Values**: These are constants like 100, 0x1234, 0x9484 etc.,
3. **Addresses**: There are 2 ways to represent addresses.

- A Immediate value - a direct number which can be used as an address.
- A pointer: The address is loaded into one of the registers and then used as explained in the previous subsection.

Most of the instructions are of the following form.

```
-------------------------------------------------------------------------
Ins Destination, Source
-------------------------------------------------------------------------
```

The first operand is Destination and second is the source.

Let us start with the Arithmetic and Logical instructions.

2.4.3.1 Arithmetic and Logical instructions

1. add: Used to add 2 operands. The general syntax is as follows.

- `add Reg, Reg`
- `add Reg, Imm`
- `add Mem, Reg`
- `add Reg, Mem`
- `add Mem, Imm`

Examples:

- `add eax, ebx`: Adds values in **eax** and **ebx** and stored it back in `eax`. Note that `eax` is the destination. `eax = eax + ebx`.
- `add rax, 0x123`: Adds 0x123 to value in rax and stores it in `rax`. `rax = rax + 0x123`.
- `add dword[ebx], eax`: Adds value in register **eax** with 4 bytes at memory pointed by **ebx**. The result is also 4 bytes stored back at the memory pointed by **ebx**.
- `add eax, dword[ebx]`: Adds value in **eax** with 4 bytes of memory pointed by **ebx**. Stores back the value into **eax**.

It is important to note that both operands cannot be memory locations.

2. sub: This is an instruction to find the difference between 2 operands. The syntax is similar to `add` instruction.

3. imul: Integer multiplication.

- `imul` can have 2 or 3 operands.

The syntax is as follows.

- `imul op1, op2`: **op1** and **op2** are multiplied and then stored back in **op1**. Note that **op1** must be a register.
- `imul op1, op2, op3`: **op2** and **op3** are multiplied and then stored in **op1**. **op1** must be a register and **op3** must be an immediate value.

5. or, **xor**, **and** instructions do bitwise operations between 2 operands. The pair of operands are the same as that for the add instruction.

6. inc: Increment the operand by **1**. * The syntax is `inc Operand`, the operand can be a register or a memory location.

7. dec: Decrement the operand by **1**. Syntax is same as **inc**.

These are the instructions which are used in mostly every program.

2.4.3.2 Data Flow instructions

Let us take at 4 very common instructions: **mov**, **lea**, **push** and **pop**.

1. mov: Though the instruction name is **mov**, it is used to copy data from source to operand.

This is the syntax: `mov Destination, Source`.

- `mov Reg, Reg`
- `mov Reg, Imm`
- `mov Reg, Mem`
- `mov Mem, Reg`
- `mov Mem, Imm`

Note that there is no **memory to memory** move instruction. At the assembly is accessed through pointers. Generally, the **memory address is loaded into a register** and then it is accessed. In some cases, direct valid addresses are used.

Here are a few examples.

- Accessing a **char** variable.
  - Suppose we have a C variable `char a = 'x'`. **a** is a character variable => Its size is **1 byte**. At the assembly level, this is how it is accessed.
  - Load the address of variable **a** into any register(say eax).
  - `byte[eax]`: Refers to 1 byte of memory pointed by eax.
  - `mov bl, byte[eax]`: Copies 1 byte pointed by eax into **bl** => **bl** will have the value of **a** after this instruction is executed.
- Accessing a **short int** variable.
  - Suppose we have a variable **short int s = 123**. The size of short int is **2 bytes**.
  - Load the address of variable **s** into a register(say ecx).
  - `word[ecx]`: Refers to 2 bytes of memory pointed by ecx. As size of **s** is 2 bytes, `word[ecx]` points to **s**.
  - `mov ax, word[ecx]` copies variable **s** into **ax**.
- Similar to this, **int** variable can be copied using **dword**.

If you observe the above **mov** instructions, the **size of both operands are always same**. This is important. If there is a size mismatch, you get an assembler error.

Consider `mov dx, dword[ecx]`: The assembler refuses to assemble this because size of **dx** is 2 bytes and **dword** means 4 bytes. You are trying to copy 4 bytes into a 2-byte register.

2. lea: Load Effective Address

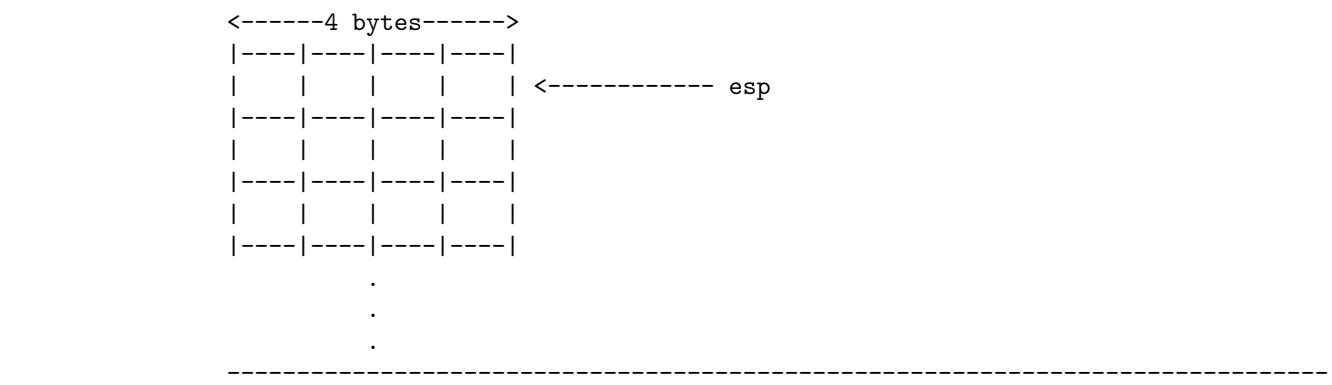
While discussing the **mov** instruction, we assumed that the address of a variable is already in a register. Later, we used to access the data. But how do we load the address of a variable into a given register? You can use the **lea** instruction.

The syntax is `lea Destination, Source`. A few examples.

- Suppose **a** is a variable and its address needs to be loaded to **rbx**. It can done like this: `lea rbx, [a]`.

3. push

Every process is given a **runtime stack** which can be used as a scratch-pad by the process. One example of this is local variables of a function. They should be in the main memory only till that function is executing. Once it returns, they should be deallocated. They can be pushed before the function starts executing and can be popped when the function is done. In 32-bit systems, the stack is 4-byte aligned. The register **esp**(Stack Pointer) always points to the top of this stack.



Its syntax is `push Operand`. Whatever is pushed onto stack, it is pushed as 4 bytes because the stack is 4-byte aligned. Suppose the value in **ax** is 0x1234 and `push ax` is executed. Top of stack would be 0x00001234. It should be noted that **esp** is changed accordingly when push is executed. Note that **esp** always points to the top of the stack.

4. pop

You pop 4 bytes from the stack into the operand. The syntax is `pop operand`.

2.4.3.3 Control Flow instructions

All the instructions are stored as an array in memory. During execution, the instruction pointer(eip) always points to the next instruction. It is important to note that the instruction next to the current instruction in memory may not be the next instruction to get executed. That is what happens in conditional statements, loops, function calls etc., There should be instructions which will help implement these conditional statements, loops, functions etc.,

1. cmp: Compares 2 operands

- Syntax: `cmp op1, op2`
- Checks **op1** with respect to **op2** and sets the appropriate flags(in eflags).

2. jmp and its derivatives

- Syntax: `jmp Address`
- The address can be a number which is a valid address. While writing code, labels are used.
- This instruction is like **goto** at C level.
  - Example: `jmp Label`: When this is executed, the instruction at **Label** is executed. It is an **unconditional jump** instruction.

Obviously, mostly we would want conditional jumps ie., we want to jump if some condition is true or false. That is where **jmp**'s derivatives come in.

- The following are a few conditional jump instructions which should be used after a **cmp** instruction. `cmp op1, op2`. Then,
  - **je**: Jump if op1 == op2
  - **jne**: Jump if op1 != op2
  - **jle**: Jump if op1 <= op2
  - **jge**: Jump if op1 >= op2
  - **jg**: Jump if op1 > op2
  - **jl**: Jump if op1 < op2

3. call: Executed to call a function

- Syntax: `call function_name`
- The call instruction is a sequence of 2 instructions.

```
--------------------------------------------------------------------------------
push return_address
jmp function_name
--------------------------------------------------------------------------------
```

4. ret: Executed by **callee** function to return back to caller function. * Syntax: `ret` * The ret instruction actually means

```
--------------------------------------------------------------------------------
pop hidden_reg
jmp hidden_reg
--------------------------------------------------------------------------------
```

With that, we have covered some common x86 instructions. The x86 ISA has unbelievably large number of instructions. There are variants of the same instruction, there are floating point instructions, there are vector-instructions and more. This is just an introduction. Learning about new, crazy instructions is an exercise.

2.4.4 Variables

We will be using the nasm assembler. It provides a certain way to declare, define variables.

1. Global initialized variables - these go into the .data section.

- `var1: db 0x12` : var1 is a variable of size 1 byte initialized to a value 0x12.
- `var2: dw 0x1234`: var2 is a 2-byte variable initialized to a value 0x1234.
- `var3: dd 0x120a0b33`: var3 is a 4-byte variable initialized to a value 0x120a0b33.
- `var4: dq 0x1223344556677889`: var4 is a 8-byte variable initialized to that value.

This is how you initialize strings(NULL-terminated character array). * `str: db "Hello World!, I am learning assembly", 0x0a, 0x00`.

2. Uninitialized variables

- `buffer: resb 1000`: Reserve 1000 bytes
- `buffer: resw 1000`: Reserve 1000 words - 2000 bytes
- `buffer: resb 1`: Reserve 1 byte - probably to store a character.
- `buffer: resd 1`: Reserve a double word - probably to store an integer.

Note that there are no datatypes like char, int etc., These are just array of bytes. We have defined int to be 4 bytes. This means, we can reserve a dword in memory and store something there, which we call it an integer.

With that, we have discussed a bunch of instructions, variable declaration-definition techniques.

2.4.5 Passing arguments and returning values

To write programs, we need more. We write functions to get work done. We don't know how arguments are passed to the caller, how values are returned to the callee. The following are standard rules.

- Push the arguments onto the stack.
- Put the return value in **eax** and execute **ret**.

There is a lot to discuss about this. We will take this up in one of the future chapters.

2.5 Hello World!

We are ready to write our first assembly program! It is a simple hello world program.

Let us write the code in a *hello.s*. It is going to have 2 sections. A .rodata section consisting of the string and a .text section with code.

1. Let us define our string.

```
-------------------------------------------------------------------------------
section .rodata

str: db "Hello World!", 0x00
-------------------------------------------------------------------------------
```

2. Let us write some code.

- Our program begins with **_start**. The only thing we need to is print the string and exit.
- Let us write the instructions to print the string.
  - **puts** is a common string printing function. It takes in one argument - pointer to a constant character string. Take a look at its manual page.

```
-------------------------------------------------------------------------------
section .text
    global _start

_start:
    push str    ; Arg: Pointer to our string
    call puts   ; Call the function
-------------------------------------------------------------------------------
```

Once it is printed, let us exit out of the program - `exit(0)`.

```
-------------------------------------------------------------------------------
    push 0      ; Argument for exit
    call exit   ; Call it
-------------------------------------------------------------------------------
```

Let us assemble it using nasm.

```
-------------------------------------------------------------------------------
chapter2$ nasm hello.s -f elf32
hello.s:10: error: symbol `puts' undefined
hello.s:13: error: symbol `exit' undefined
-------------------------------------------------------------------------------
```

The **-f** option is used to specify the relocatable file format. Here, we want to generate a 32-bit ELF executable in the end => We need to generate a 32-bit ELF relocatable file. You can run **nasm -fh** to list the bunch of target formats.

Here, we are getting 2 undefined symbols error. we need to inform the assembler that these 2 are found in some other relocatable file or in some library. The assembler should put relocation records(fixing table entries) for these 2 symbols.

You can use the **extern** keyword in nasm to inform it that these symbols are defined elsewhere and you want the linker to take care of it.

The following is the complete listing of the program.

```
-------------------------------------------------------------------------hello.s
section .rodata

str: db "Hello World!", 0x00

extern puts
extern exit

section .text
    global _start

_start:
    push str    ; Arg: Pointer to our string
    call puts   ; Call the function

    push 0      ; Arg: 0
    call exit   ; Call it
-------------------------------------------------------------------------------
```

Let us assemble it.

```
-------------------------------------------------------------------------------
chapter2$ nasm hello.s -f elf32
chapter2$ file hello.o
hello.o: ELF 32-bit LSB relocatable, Intel 80386, version 1 (SYSV), not stripped
-------------------------------------------------------------------------------
```

It generated *code1.o*, an 32-bit ELF relocatable file.Now let us link it.

```
-------------------------------------------------------------------------------
chapter2$ ld hello.o -o hello -melf_i386
```

```
hello.o: In function `_start':
hello.s:(.text+0x6): undefined reference to `puts'
hello.s:(.text+0xd): undefined reference to `exit'
-------------------------------------------------------------------------------
```

ld is the GNU Linker. The **-m** option specifies the type of executable we want. We want an i386(basically Intel 32-bit) executable.

We got a linking error. We know that the linker should either find the definitions in other relocatable files or libraries. In our case, there are no other relocatable files. If they are present in the libraries, they linker will add fixing table entries and leave to to get resolved at runtime. At runtime, who resolves these symbols? To do this runtime linking, there is a helper program called the **dynamic linker**. As the name suggests, it does dynamic linking. We know that **puts** and **exit** belong to the standard C library. While linking, we need to specify 2 things. The library these symbols can be found and the dynamic linker.

The dynamic linker is found at `/lib/ld-linux.so.2` and the library we want to link our program **against** is **libc**. Let us go ahead and do that.

```
-------------------------------------------------------------------------------
chapter2$ ld hello.o -o hello --dynamic-linker /lib/ld-linux.so.2 --library c -melf_i386
chapter2$ file hello
hello: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV), dynamically linked,
interpreter /lib/ld-, not stripped
-------------------------------------------------------------------------------
```

You specify the dynamic-linker using the –**dynamic-linker** option, the library using the –**library** option. If you don't get any errors, you got yourself an executable. Try running it.

```
-------------------------------------------------------------------------------
chapter2$ ./hello
Hello World!
-------------------------------------------------------------------------------
```

There you go! Your first assembly program in action!

I want you to inspect *hello.o*(the relocatable file) and *hello*. Take a look at their sections, whats added - the symbol table, fixing table etc.,

2.5.1 Hello World! - Part2

Let us a modified version of the hello program, one which prints one character at a time.

Just have in mind how you would a C program to do this. You can have a while loop, a character variable traversing through the string till it hits NULL.

Lets call the program *hello1.s*.

Let us start with the variables. We would have a the hello string and the **format string** for printf.

```
-------------------------------------------------------------------------------
section .rodata

str: db "Hello World!", 0x00
format_string: db "%c", 0x0a, 0x00
-------------------------------------------------------------------------------
```

Now onto the .text section. Let us start with getting a pointer the string to a string and initializing a register to 0.

```
-------------------------------------------------------------------------------
_start:
    xor ebx, ebx        ; Clean up the register
    lea esi, [str]      ; Get a pointer to our string
-------------------------------------------------------------------------------
```

Let us start with our loop.

```
-------------------------------------------------------------------------------
loop:
    mov bl, byte[esi]   ; Get the byte
    cmp bl, 0x00        ; Compare it with NULL
    je loop_out     ; If it is NULL, then we have hit the end. Get out!
-------------------------------------------------------------------------------
```

That was the loop initialization. Let us come to the body of the loop.

```
-------------------------------------------------------------------------------
    ; Print the character   ; printf("%c\n", c);
    push ebx        ; Second argument = the character itself
    push format_string  ; First argument = address of "%c\n"
    call printf     ; Call it
-------------------------------------------------------------------------------
```

Look at the order in which the arguments are pushed. The last argument is pushed first. This is a standard followed.

Let us write the last part of the loop.

```
-------------------------------------------------------------------------------
    inc esi         ; Increment the pointer
    jmp loop        ; Jump back to the beginning
-------------------------------------------------------------------------------
```

You increment the pointer and jump back the beginning of the loop.

loop_out has a call to the exit function.

The following is the complete listing.

```
-------------------------------------------------------------------------hello1.s
section .rodata

str: db "Hello World!", 0x00
format_string: db "%c", 0x0a, 0x00

extern printf
extern exit

section .text
    global _start

_start:
    xor ebx, ebx        ; Clean up the register
    lea esi, [str]      ; Get a pointer to our string
```

```
loop:
    mov bl, byte[esi]   ; Get the byte
    cmp bl, 0x00        ; Compare it with NULL
    je loop_out     ; If it is NULL, then we have hit the end. Get out!

    ; Print the character   ; printf("%c\n", c);
    push ebx        ; Second argument = the character itself
    push format_string  ; First argument = address of "%c\n"
    call printf     ; Call it

    inc esi         ; Increment the pointer
    jmp loop        ; Jump back to the beginning

loop_out:
    ; Exit
    push 0      ; Arg: 0
    call exit   ; Call it
--------------------------------------------------------------------------------
```

Assemble and link it.

```
--------------------------------------------------------------------------------
chapter2$ nasm hello1.s -f elf32
chapter2$ ld hello1.o -o hello1 --dynamic-linker /lib/ld-linux.so.2 --library c -melf_i386
--------------------------------------------------------------------------------
```

And run it!

```
--------------------------------------------------------------------------------
chapter2$ ./hello1
H
e
l
l
o

W
o
r
l
d
!
--------------------------------------------------------------------------------
```

That was the second.

With that, we have come to the end of this section. I encourage you to practice because we will be reading lot of assembly code and writing some too.

2.6 The x86_64 ISA

So far, we explored the 32-bit x86 instruction set. Now, let us take a look at its 64-bit extension - the x86_64 ISA.

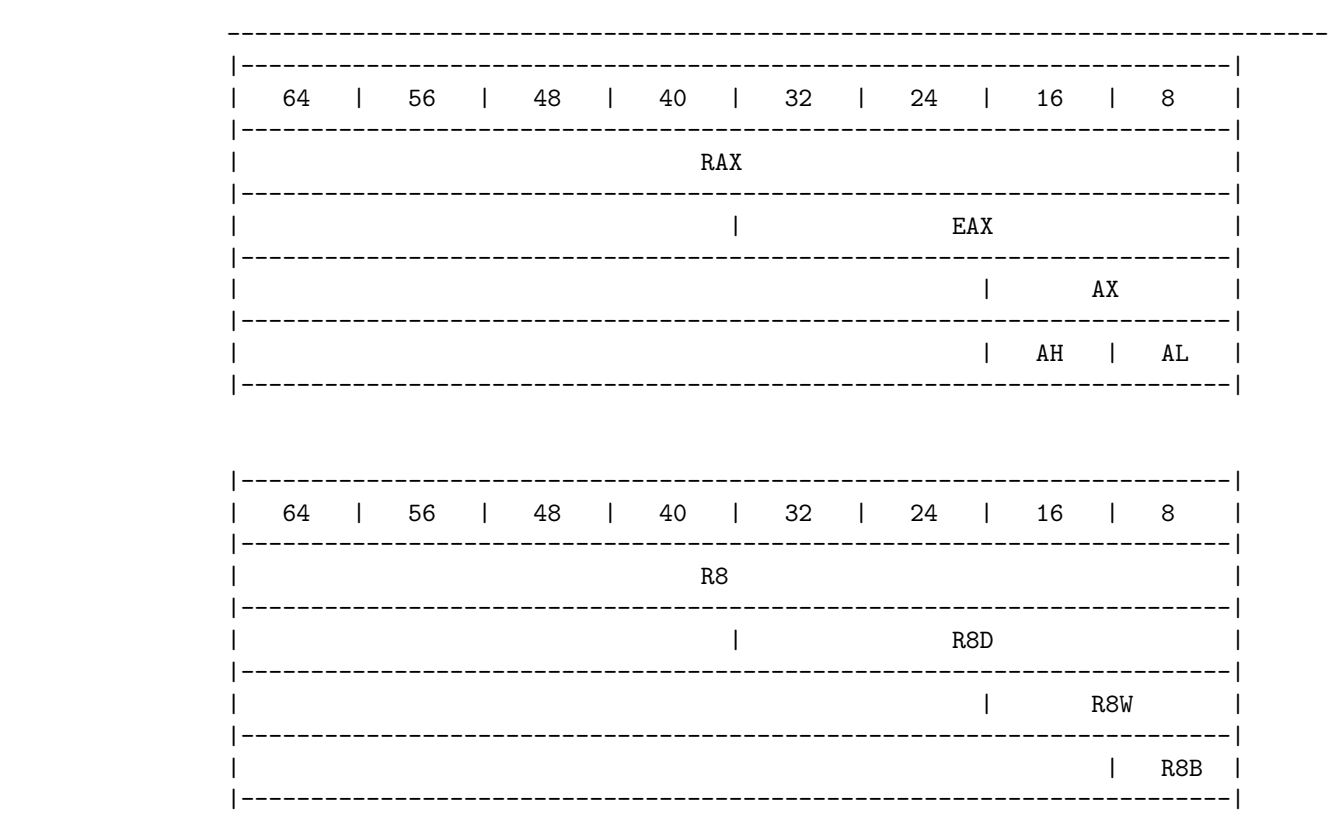
2.6.1 Registers

The x86 ISA offered 8 general purpose registers along with 2 special purpose registers. The x86_64 ISA provides 16 general purpose registers with 2 special registers.

Size of each of these GPRs is **8** bytes. They are **rax**, **rcx**, **rdx**, **rbx**, **rsp**, **rbp**, **rsi**, **rdi**, **r8**, **r9**, **r10**, **r11**, **r12**, **r13**, **r14** and **r15**. The first 8 are direct 64-bit extensions of their corresponding 32-bit registers. The last 8 are newly added to the ISA.

The 2 special purpose registers are **rip** and **eflags**. **rip** is the instruction pointer, 8 bytes in size. **eflags** is still a 32-bit register.

The **r** simply stands for register.



2.6.2 Datatypes

We have seen **byte**, **word** and **dword**. One more addition to this. Because the size of data bus is 8 bytes. a new type is added - **qword / quad word** which is 8 bytes. Note that all the 4 can be used in a 64-bit program.

2.6.3 Instructions

Most of the instructions discussed in the previous chapter are part this ISA too. There are a few changes, extensions we need to look at.

2.6.3.1 push and pop

The runtime stack in a 64-bit system is 8-byte aligned. The stack can be viewed like this.

```
-------------------------------------------------------------------------------
<---------------8 bytes----------------->
|----|----|----|----|----|----|----|----|
|    |    |    |    |    |    |    |    | <-----rsp
|----|----|----|----|----|----|----|----|
|    |    |    |    |    |    |    |    |
|----|----|----|----|----|----|----|----|
|    |    |    |    |    |    |    |    |
|----|----|----|----|----|----|----|----|
                    .
                    .
                    .
-------------------------------------------------------------------------------
```

Because the stack is 8-byte aligned, only 8-byte operands can be pushed onto stack and whatever you pop, the destination should be 8 bytes long. Registers rax, r8, r15 can be pushed and popped into but using eax, cl, bx gives an assembler error.

Rest of the instructions we discussed have no changes. Note that data can be 8 bytes long. You can use **qword** in the data movement instructions.

2.6.4 Passing arguments and returning values

In x86, passing arguments was very easy. You had to push arguments onto stack. In 64-bit, this is how it is done. Because there are a lot more registers, the first few arguments are passed through registers. Rest of them(if there are so many) are pushed to the stack.

1. Arg1 into **rdi**
2. Arg2 into **rsi**
3. Arg3 into **rdx**
4. Arg4 into **rcx**
5. Arg5 into **r8**
6. Arg6 into **r9**
7. If there are more, push them onto stack.

2.6.5 Hello World in x64

Let us rewrite the second hello program *hello1.s* in x64 assembly language. Lets start with defining the necessary strings in the .rodata section.

```
-------------------------------------------------------------------------------
section .rodata
```

```
str: db "Hello World!", 0x00
format_string: db "%c", 0x0a, 0x00
--------------------------------------------------------------------------------
```

Let us head to the .text section. Start with initializing registers.

```
--------------------------------------------------------------------------------
section .text
    global _start

_start:
    xor r15, r15        ; Clean up the register
    lea r14, [str]      ; Get a pointer to our string
--------------------------------------------------------------------------------
```

Let us start the loop.

```
--------------------------------------------------------------------------------
loop:
    mov r15b, byte[r14] ; Get the byte
    cmp r15b, 0x00      ; Compare it with NULL
    je loop_out     ; If it is NULL, then we have hit the end. Get out!
--------------------------------------------------------------------------------
```

Now is the interesting part, the body of the loop.

```
--------------------------------------------------------------------------------
    ; Print the character   ; printf("%c\n", c);
    lea rdi, [format_string]; Arg1: Address of format string
    mov rsi, r15        ; Arg2: The character
    call printf     ; Call it
--------------------------------------------------------------------------------
```

Look at the way arguments are passed. They are being passed through registers, unlike through stack in the 32-bit program. Remember the register order in which arguments are passed.

End the loop and exit.

```
--------------------------------------------------------------------------------
    inc r14         ; Increment the pointer
    jmp loop        ; Jump back to the beginning

loop_out:
    ; Exit
    push 0      ; Arg: 0
    call exit   ; Call it
--------------------------------------------------------------------------------
```

Note that **printf** and **exit** are part of **libc**. Use the **extern** keyword and inform the assembler. The following is the complete listing.

```
----------------------------------------------------------------------hello1_64.s
section .rodata

str: db "Hello World!", 0x00
format_string: db "%c", 0x0a, 0x00
```

```
extern printf
extern exit

section .text
        global _start

_start:
        xor r15, r15            ; Clean up the register
        lea r14, [str]          ; Get a pointer to our string
loop:
        mov r15b, byte[r14]     ; Get the byte
        cmp r15b, 0x00          ; Compare it with NULL
        je loop_out             ; If it is NULL, then we have hit the end. Get out!

        ; Print the character   ; printf("%c\n", c);
        lea rdi, [format_string]; Arg1: Address of format string
        mov rsi, r15            ; Arg2: The character
        call printf             ; Call it

        inc r14                 ; Increment the pointer
        jmp loop                ; Jump back to the beginning

loop_out:
        ; Exit
        push 0          ; Arg: 0
        call exit       ; Call it
--------------------------------------------------------------------------------
```

Let us assemble it.

```
--------------------------------------------------------------------------------
chapter2$ nasm hello1_64.s -f elf64
--------------------------------------------------------------------------------
```

Now to linking. The way there was a dynamic linker for 32-bit programs, there is a dynamic linker for 64-bit programs too. By default, `/lib/x86_64-linux-gnu/ld-linux-x86-64.so.2`. Let us go ahead and link it up.

```
--------------------------------------------------------------------------------
chapter2$ ld hello1_64.o -o hello1_64 -lc --dynamic-linker
/lib/x86_64-linux-gnu/ld-linux-x86-64.so.2
--------------------------------------------------------------------------------
```

Go ahead and run it!

```
--------------------------------------------------------------------------------
chapter2$ ./hello1_64
H
e
l
l
o

W
o
```

```
r
l
d
!
--------------------------------------------------------------------------------
```

2.7 A couple of things

2.7.1 Measuring bytes and spaces

Hardware manufacturers and programmers, operating system designers use different units to measure memory. 1 GB harddisk means it is a 1 Gigabytes harddisk. Here, 1 GB = 10^9 bytes. In operating systems, everything is a power of 2. If someone mentions 1GB, he probably means $(2^{10})3$ bytes. This is an official unit called the **Gibibyte(GiB)**

- 1KB = 1,000B, 1KiB = 1,024B
- 1MB = 1,000,000B, 1MiB = 1024*2 = 1,048,576B
- 1GB = 1,000,000,000B, 1GiB = 1024^3 = 1,073,741,824B

This continues: Tebibytes, Pebibytes, Exbibytes…

2.7.2 On the 8086 Intel processor

The Intel 8086 processor is very interesting. It was originally a 16-bit processor => Its data bus size was 16 bits. Its address bus size was also 16 bits. This meant it could address a maximum of 2^16 = 65536 bytes - 64KiB. Programmers wanted more system space, more memory but the processor could handle only 64KiB. There was a need for the processor manufacturers to support more memory. That is when they came up with the concept of **segmentation**. In this new design, the size of the address was 20-bits ie ie., it could address a maximum of 1MiB which is 16 times of 64KiB. Let us have a quick look at how this works.

The complete address space is divided into **segments**. Each byte in the address space has an address of the form **Segment : Offset** where **Segment** contains that byte at an offset of **Offset**.

This is when the x86 ISA was introduced with 4 new registers:

1. **ds** - Data Segment
2. **cs** - Code Segment
3. **ss** - Stack Segment
4. **es** - Extra Segment

These are called **segment registers**. Each of these were supposed to point to their segments.

Later, 2 more were introduced - **fs** and **gs**.

Now, segmentation is outdated but a few of these registers are still used for special purposes.

2.7.3 What all does the ISA contain?

What we discussed was just tip of the iceburg. The x86 ISA has lots of instructions for floating point operations, vector operations, string operations, cryptographic instructions like sha1, aes etc.,

It has a bunch of other general purpose registers too which are there to handle floating point operations.

It should be noted that the ISA is what the programmer/compiler can see. The processor has a lot of registers, buffers, storage on the chip of the microprocessor which are not exposed to the programmers. **Control Registers** is a class of registers which controls the behavior of a CPU. These control registers have info about paging, interrupts etc.,

This is the official link where you will find x86 software developer manuals. It has the instruction set, it has a system programming guide where it describes memory management, protection, multi-processing, virtualization and a lot more.

2.7.4 Different Syntaxes

There are 2 prominent assembly language syntaxes out there. They are **AT&T Syntax** and the **Intel Syntax**. We used the Intel syntax in this chapter. GNU toolchain by default uses the AT&T syntax. We used Intel Syntax because it is simple and code is less noisy. Read this article to know more.

2.7.5 What exactly is the dynamic linker?

For me, the operating system is like god, your program is a baby, the dynamic linker is like the god-given parent to the baby. The dynamic linker is a helper program that makes sure your program runs properly. It loads your program into memory, the dependent libraries into memory, prepares the program to run and it runs it. It resolves links symbols dynamically whenever necessary. We will specifically explore the dynamic linker in one of the future chapters. To learn more, you can read its manual page - `man ld.so`. Try running the dynamic linker and you will get more details.

2.8 Fini

We have come to the end of this chapter.

In this chapter, we explored the x86 and x64 ISA and the x86 assembly language provided by the nasm assembler. We explored how CPU communicates with devices, phyical address spaces, bunch of common instructions, registers and finally wrote 2 programs.

We tied up a few loose ends present in the previous chapter but there are still a lot of them. As we progress, we will tie up the rest.

2.8 Further Reading

1. Abstract Machines for programming language implementation: Whatever we are exploring is basically part of programming languages. This is an interesting paper.
2. The 8051 Microcontroller and Embedded Systems: Using assembly and C
3. The Intel Software Developer Manuals

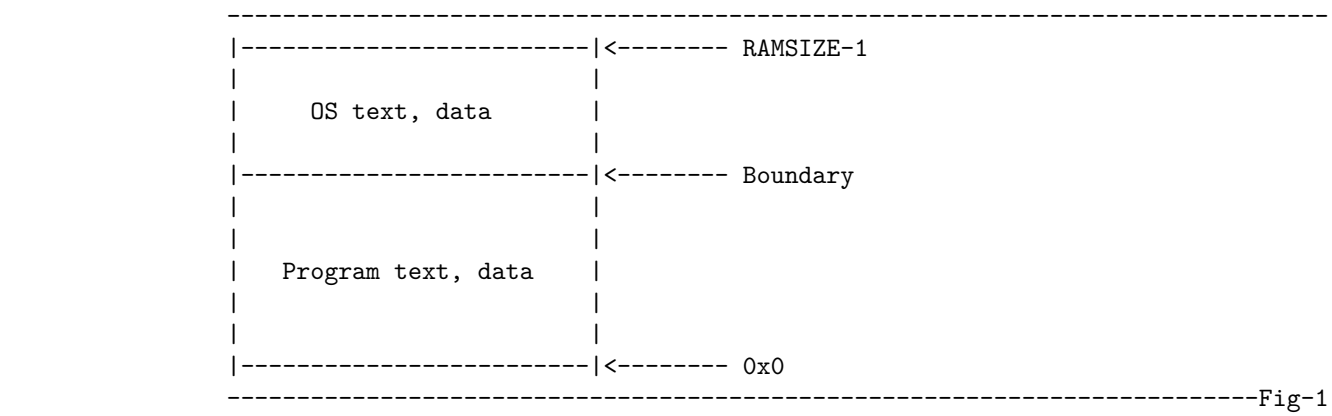
Chapter 3: Address Spaces

Summary
In this chapter, we will look at a process - a progam with life. The program on file is a collection of sections. We will see how it looks like in memory, when it is under execution. We will explore a process's memory layout and address spaces.

3.1 Init

Let us start with a question. What is a process? A process is a program under execution. A program is a dead file sitting in the secondary storage. But a process is full of life, present in main memory and is running.

Let us start with a very simple operating system. An operating system where a user can run only one program at a time - a single-tasking operating system. There can be multiple program(files) stored in the secondary storage. But only one program can be executed at once ie., only one process can exist in memory at once. Figure-1 shows us how the RAM would look like when a process is running.



One portion of main memory will always have operating system text and data. The operating system is always present in main memory. It is supposed to be an efficient resource manager. When a program is run, parts of the program file essential for it to run are copied to the main memory. In the previous chapter, we discussed about Physical Address Space. If the processor has to access something on the main memory, it will use the physical address of that content and fetch it. Early versions of MS-DOS were single-user single-tasking operating system. Let us talk in terms of address spaces. A part of the processor's physical address space is used to address the entire main memory. Let us call it the main memory space. In the above scenario, the main memory space is divided into 2 parts: operating system

space(**os-space**) and user program space(**user-space**). The operating system being the supervisor, its text should be protected from any process run on this machine. A program run by some user should not be able to change the operating system's text, data or anything. The operating system should enforce a strict mechanism defining the boundaries. In current day operating systems, if a user process tried to access some OS-space memory in the os-space, the process will be terminated for doing it.

Let us move a step forward. Consider a multi-user operating system. Assume there are currently 5 users on this system and each of them run a program. With the single-tasking nature, the operating system should generate an order to execute the 5 programs. The following is one order of execution.

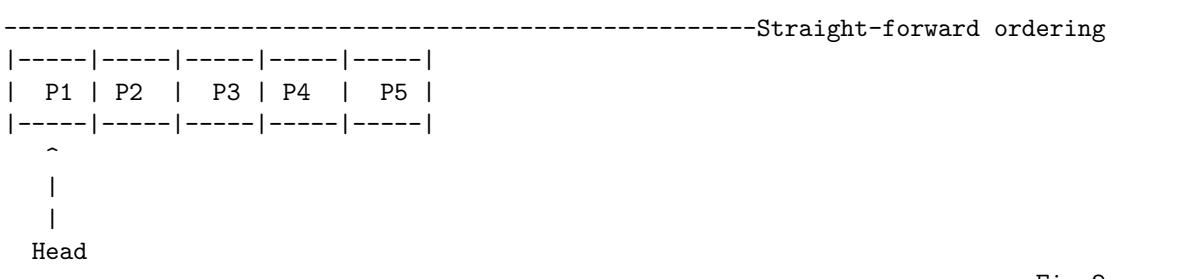


Fig-2

Let us take a closer look at how the 5 programs get executed. Initially, P1 is copied onto main memory. It is executed. Once it is done, the user-space is cleaned up, P2 is copied and executed. In the same way, P3, P4 and P5 are executed. This is one way of ordering(or **scheduling**) the programs for execution. The operating system can choose different orders of execution. It can order based on program-priority, if the user executing the program is root or not etc., The straight-forward ordering gets the job done but there are problems. When P1 is executing, only U1 will be busy interacting with his program - input, output, computation while the other 4 users wait for their opportunity. U5 waits for the longest amount of time. It would be best if all the 5 processes run simultaneously. But that is not possible because there is only 1 CPU and only 1 process can run on the CPU at any given moment. Can we do something to give the users an **illusion** of these 5 processes running simultaneously? How can this be implemented? Instead of the above straight-forward order of execution, consider a **round-robin** ordering. Total execution time is divided into time-intervals of equal lengths T units of time. P1 runs for T units of time. Once T units are over, P1 stops running and P2 starts running. P2 runs for T units. Once T units are done, P2 stops and P3 starts. Continuing, once P5 runs for T units, P5 is stopped and P1 is brought back. This keeps happening till all processes come to an end. Take a look at the following diagram.

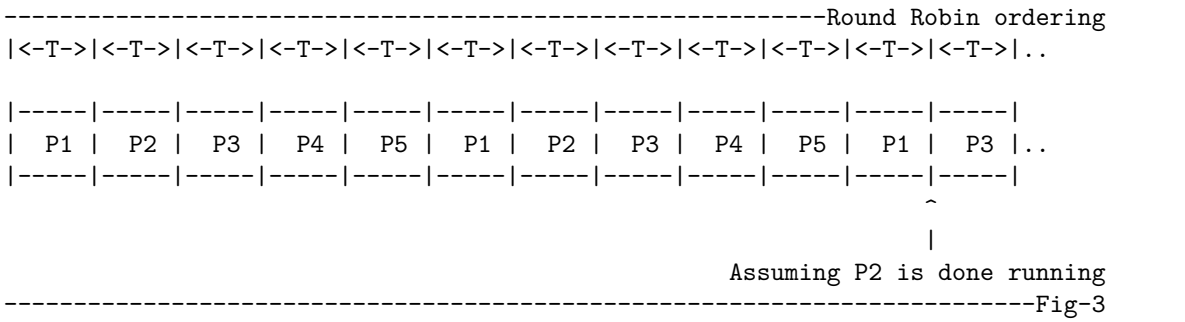


Fig-3

You might ask even here U5 waits till P1, P2, P3 and P4 finishes its T units of time? That is true. The time interval T is generally a small quantity. Say T is 5ms. Excluding the first round, each process waits for (4 * T) units of time. This means each user will also have to wait for (4 * T) = 20ms. From a process's point of view, 5ms is a lot of time. A lot of instructions get executed in 5ms. From a user's point of view, 20ms is a really small amount of time. Waiting for 20ms is almost equivalent to not waiting. This way, the users **feel** that their programs are run simultaneously but even now only one

process is run at any given point in time. This idea of providing an **illusion** of simultaneous execution of programs is called **Time-Sharing**. When the concept of time-sharing is used to to run programs of various users, it is known as a multi-tasking operating system. The original UNIX operating system was a time-sharing operating system.

That is the concept. Now let us see how it can be implemented in the operating system. A fresh copy of text, data etc., are loaded into memory the first time a program(say P1) comes under execution. When it stops running after T units of time, it will come back some time later. When it comes back, it needs to start exactly at the point where execution had stopped the last time. Because it was executed for T units of time, the processor's register values might be different, the variables' values might be different, execution probably had stopped in the middle of some function, the runtime stack had some contents etc., basically, the **process state** would have changed. The operating system should take a **snapshot** of P1 before P2 is put under execution. When its P1's turn again, the operating system must load the snapshot and not a fresh copy. How can this snapshot mechanism be implemented? The snapshot is nothing but the user-space after P1 runs for T units of time. This snapshot can be copied onto the secondary storage. Later when its P1's turn, this snapshot will be copied back onto the main memory. This technique is called **swapping**. You swap out P1 into secondary storage when its time-interval expires and swap in P2 into the main memory when its time-interval starts. The part of secondary storage used for swapping is called **swap-space**. The swap-space here should be a multiple of size of user-space. Because the swap-space is used by the operating system, it should be protected from users. The RAMSIZE is generally very small compared to the size of secondary storage. That is why we can make use of a piece of secondary storage for swapping.

Note that the memory manager part of the operating system was very simple when it implemented the straight-forward ordering. Time-sharing increased the complexity. Let us calculate the total execution times of the 2 types of ordering. Let T1, T2, T3, T4 and T5 be the execution times of P1, P2, P3, P4 and P5 respectively. Each program needs to be loaded into memory and once execution is over, memory needs cleanup(Why?). Amount of time needed for loading and cleanup depends on the size of the program. Loading is nothing but copying the program from secondary storage to main memory. Bigger the program, more time it takes to load. Similarly, bigger the program, more time is needed to cleanup. Let it take T_Load units of time to load X bytes from secondary storage to main memory and T_Clean units of time to clean X bytes of main memory. With that, we can calculate total runtimes for both orderings. Total runtime = Total Program Execution time + Total Load time + Total Cleanup time

1. Straight-forward Ordering:
   - Total runtime = (T1 + T2 + T3 + T4 + T5) + [Ceil(sizeof(Pi) / X) * T_Load][i=1 to 5] + [Ceil(sizeof(Pi) / X) * T_Clean][i = 1 to 5]
2. Round-Robin ordering
   - Total runtime = (T1 + T2 + T3 + T4 + T5) + [Ceil(Ti/T) * Ceil(sizeof(Pi)/X) * T_Load][i = 1 to 5] + [Ceil(Ti/T) * Ceil(sizeof(Pi)/X) * T_Clean][i = 1 to 5].

In a practical scenario, there won't be a fixed number of programs. Programs will be coming in for execution and leaving after execution. It is highly dynamic scenario. In the straight-forward ordering system, the new program is put at the end of the execution queue and it has to wait till every program before it in the queue is executed. But in a time-sharing system, it is a different scene. It is given its T units of time as soon as possible. Then it becomes part of the round-robin ordering.

Observe that the total runtime of the time-sharing system is significantly higher than the straight-forward ordering system. The operating system also has to do a lot more work in the time-sharing scenario. That is the cost paid in return for fairness and a time-sharing system.

That was one thread of exploration to which we will come back later. Let us take up another interesting thread.

The linker is the one which gives absolute addresses to text and data. When the program is run, these sections are copied to the memory at the addresses alloted by the linker. With almost all the examples we have seen, the text starts at 0x0 and data starts right after text ends. The linker started with the first unused address(which is 0x00) and gave it to text. Once all bytes in text are given addresses, it took the next unused address(which is 0x00 + sizeof(text)-1) and gave it to data. Consider the following assembly program. Assume that we are working on a 64-bit system and addresses are 64-bits in size.

```
-------------------------------------------------------------------------hello.s
section .data

str: db "Hello World!", 0x00
ptr: dq 0x0000000000000000              ; pointer

section .text
        global _start

_start:
        xor r15, r15            ; Clean up the register
        lea r14, [str]          ; Get the starting address of the string
        mov qword[ptr], r14     ; ptr is a pointer
loop:
        mov r14, qword[ptr]     ; Load the pointer into a register
        mov r15b, byte[r14]     ; Get the byte, r15 = *(char *)r14;
        cmp r15b, 0x00          ; Compare it with NULL
        je loop_out             ; If it is NULL, then we have hit the end. Get out!

        mov r8, r15             ; Just load the character into register r8

        inc qword[ptr]          ; Increment the pointer
        jmp loop                ; Jump back to the beginning

loop_out:
        mov rax, 0x01
-------------------------------------------------------------------------------
```

Don't worry about what the program does. It is assembled and linked with 0x00 as text's starting address and data coming right after text. This is how the program would look like.

```
-------------------------------------------------------------------------------
Disassembly of section .text:

0000000000000000 <_start>:
  000000:   4d 31 ff                xor    r15,r15
  000003:   4c 8d 34 25 36 00 00    lea    r14,0x36
  00000a:   00
  00000b:   4c 89 34 25 43 00 00    mov    QWORD PTR 0x43,r14
  000012:   00

0000000000000013 <loop>:
  000013:   4c 8b 34 25 43 00 00    mov    r14,QWORD PTR 0x43
  00001a:   00
  00001b:   45 8a 3e                mov    r15b,BYTE PTR [r14]
```

```
  00001e:   41 80 ff 00             cmp    r15b,0x0
  000022:   74 0d                   je     0x31 <loop_out>
  000024:   4d 89 f8                mov    r8,r15
  000027:   48 ff 04 25 43 00 00    inc    QWORD PTR 0x43
  00002e:   00
  00002f:   eb e2                   jmp    0x13 <loop>

0000000000000031 <loop_out>:
  000031:   b8 01 00 00 00          mov    eax,0x1

Disassembly of section .data:

0000000000000036 <str>:
  000036:   48 65 6c 6c 6f 20 57 6f 72 6c 64 21 00

0000000000000043 <ptr>:
    000043: 00 00 00 00 00 00 00 00
--------------------------------------------------------------------------------
```

Figure-4 shows how the program looks like in memory.

```
--------------------------------------------------------------------------------
|-------------------| <-- 0x00000000bfffffff // Highest available user-space address
|                   |
|                   |
|     unused        |
|                   |
|-------------------| <-- 0x000000000000004a // data ends
|                   |
|      data         |
|                   |
|-------------------| <-- 0x0000000000000036 // Starting of data
|                   |
|                   |
|      text         |
|                   |
|-------------------| <-- 0x0000000000000000 // Starting of text
--------------------------------------------------------------------------Fig-4
```

And the program runs properly. If the linker had the information about the amount of user-space physical memory available, it could have alloted different addresses to text and data other than 0x00 and (0x00 + sizeof(text)-1). Assume that the linker now knows that there is 3GiB of user-space physical memory available. It can allot some other addresses for both text and data. Assume that it gave the addresses 0x4000b0 to the text and 0x6000e8 to data. The program would look like this.

```
--------------------------------------------------------------------------------
chapter3$ objdump -Mintel -D hello

hello:     file format elf64-x86-64


Disassembly of section .text:

00000000004000b0 <_start>:
```

```
  4000b0:   4d 31 ff                xor    r15,r15
  4000b3:   4c 8d 34 25 e8 00 60    lea    r14,0x6000e8
  4000ba:   00
  4000bb:   4c 89 34 25 f5 00 60    mov    QWORD PTR 0x6000f5,r14
  4000c2:   00

00000000004000c3 <loop>:
  4000c3:   4c 8b 34 25 f5 00 60    mov    r14,QWORD PTR 0x6000f5
  4000ca:   00
  4000cb:   45 8a 3e                mov    r15b,BYTE PTR [r14]
  4000ce:   41 80 ff 00             cmp    r15b,0x0
  4000d2:   74 0d                   je     4000e1 <loop_out>
  4000d4:   4d 89 f8                mov    r8,r15
  4000d7:   48 ff 04 25 f5 00 60    inc    QWORD PTR 0x6000f5
  4000de:   00
  4000df:   eb e2                   jmp    4000c3 <loop>

00000000004000e1 <loop_out>:
  4000e1:   b8 01 00 00 00          mov    eax,0x1

Disassembly of section .data:

00000000006000e8 <str>:
  6000e8:   48 65 6c 6c 6f 20 57 6f 72 6c 64 21 00

00000000006000f5 <ptr>:
    6000f5: 00 00 00 00 00 00 00 00
--------------------------------------------------------------------------------
```

The process would look like the following in memory.

```
--------------------------------------------------------------------------------
|-------------------| <-- 0x00000000bfffffff // Highest available user-space address
|                   |
|      unused       |
|-------------------| <-- 0x00000000006000fc // data ends
|      data         |
|-------------------| <-- 0x00000000006000e8 // data starts
|                   |
|      unused       |
|-------------------| <-- 0x00000000004000e1 // text ends
|                   |
|      text         |
|-------------------| <-- 0x00000000004000b0 // text starts
|      unused       |
|                   |
|-------------------| <-- 0x0000000000000000 // Lowest available user-space address
--------------------------------------------------------------------------------Fig-5
```

Even with this, the program should run properly. Important point to note is that the number of user-space physical addresses are limited. The linker cannot allot addresses greater than 0x00000000bfffffff because there are only 3GiB of user-space memory available.

Now, let us discuss something interesting. Consider a simple single-tasking operating system which

uses the straight-forward ordering. Maximum process size is RAMSIZE - OSSIZE. We assumed that the process size will be atmost RAMSIZE - OSSIZE. Assume you have a 4GiB RAM, 64-bit CPU. Assuming OSSIZE to be 1GiB, user-space is 3GiB. But I have a program which requires 5GB of main memory to run. Is it somehow possible to run this big process on our system? Obviously, the current memory manager cannot handle such cases. Can you come up with a mechanism to handle this? Consider a hypothetical system with a 64-bit CPU along with just the amount of user-space needed to store 1 instruction and 1 data element. In x64 architecture, instruction size can range from 1 byte to 15 bytes. Maximum data size is 8 bytes. Note that the OS is still in the RAM, just the user-space is equal to 23 bytes - 15 bytes for instruction, 8 bytes for data.

```
-------------------------------------------------------------------------
|-------------------------|<-------- RAMSIZE-1
|                         |
|     OS text, data       |
|                         |
|-------------------------|<-------- Boundary
|                         |<-- 0x16 //  Data space ends here
|                         |.
|                         |.
|                         |<-- 0x10
|     User-space          |<-- 0x0f // Data space starts here
|                         |<-- 0x0e // Instruction space ends here
|                         |.
|                         |.
|                         |.
|                         |<-- 0x01
|-------------------------|<-- 0x00 // Instruction space starts here
-------------------------------------------------------------------------Fig-6
```

With the above system, you need to execute *hello*, the executable generated from *hello.s*. This program traverses through the hello-world string. It is exactly how we do it in C. We use a pointer, dereference it to get the character, we are simply loading that character into register r8 and then incrementing the pointer. When the character is NULL, you exit the loop.

If the linker knew that there is only 23 bytes of user-space available, it would probably give an error. But let us ignore it and generate the executable *hello*. An address is a unique identifier of a memory location. An address has meaning only when it actually points to a memory location right? With a 3GiB user-space system, the highest address available is 0x00000000bfffffff. If I am talking about 0xd1234567 or any address above 0xbfffffff, it is meaningless. We have the exact same situation now. We have 23 bytes of user-space physical memory which means the only valid addresses are 0x00 to 0x16. If the process's size is greater than 23 bytes, then with the current system, we can't run it. But challenge is to come up with a mechanism to run it. For fun, let us use the executable with different addresses for text and data. The following is the executable.

```
-------------------------------------------------------------Disassembly of hello
Disassembly of section .text:

00000000004000b0 <_start>:
  4000b0:   4d 31 ff                xor    r15,r15
  4000b3:   4c 8d 34 25 e8 00 60    lea    r14,0x6000e8
  4000ba:   00
  4000bb:   4c 89 34 25 f5 00 60    mov    QWORD PTR 0x6000f5,r14
  4000c2:   00
```

```
00000000004000c3 <loop>:
  4000c3:   4c 8b 34 25 f5 00 60    mov    r14,QWORD PTR 0x6000f5
  4000ca:   00
  4000cb:   45 8a 3e                mov    r15b,BYTE PTR [r14]
  4000ce:   41 80 ff 00             cmp    r15b,0x0
  4000d2:   74 0d                   je     4000e1 <loop_out>
  4000d4:   4d 89 f8                mov    r8,r15
  4000d7:   48 ff 04 25 f5 00 60    inc    QWORD PTR 0x6000f5
  4000de:   00
  4000df:   eb e2                   jmp    4000c3 <loop>

00000000004000e1 <loop_out>:
  4000e1:   b8 01 00 00 00          mov    eax,0x1

Disassembly of section .data:

00000000006000e8 <str>:
  6000e8:   48 65 6c 6c 6f 20 57 6f 72 6c 64 21 00

00000000006000f5 <ptr>:
    6000f5: 00 00 00 00 00 00 00 00
--------------------------------------------------------------------------------
```

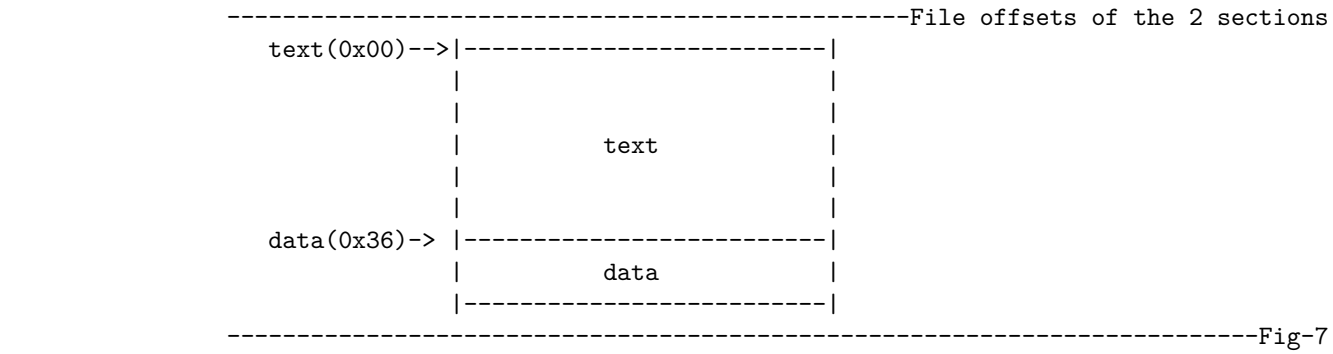
The above disassembly tells us what byte will be present at what address IF the complete program is loaded into main memory - that is not happening in our case. Observe all the information it gives about the program. It gives a relative positioning of every byte, every instruction and data element. IF text's starting address is 0x4000b0, then the first instruction is situated at 0x4000b0, second instruction at (0x4000b0 + 3) = 0x4000b3, third at (0x4000b3 + 8) = 0x4000bb and so on. IF data's starting address is 0x6000e8, then str is situated at 0x6000e8, ptr is situated at (0x6000e8 + 13) = 0x6000f5. Even though these are **fake addresses** and don't have real meaning because there is no physical memory to back it up, it provides complete structure to the program.

At this point, we have 2 types of addresses. One is the real, physical addresses which actually point to memory locations. We have 23 of them - 0x00 to 0x16, but there are only 2 relevant physical addresses - 0x00 where a single instruction can be placed and 0x0f where a single data element can be placed. Secondly, we have the fake addresses which gives complete structure of the program but addresses don't refer to any memory locations.

In our discussions so far, processors used physical addresses and executed the program - quite natural. The processor understood the program structure from the physical addresses. If physical address 0x1234 is the first instruction's address and the first instruction is 5 bytes long, this meant that the second instruction was stored at physical address 0x1238. But now, there are just 2 relevant physical addresses. With that, the processor cannot understand anything about the program. With physical address 0x00, the program does not know what instruction comes after what or with 0x0f, processor cannot figure out where str or ptr is present. The fake addresses has all this information. Infact, it has all the information about the program. Qn: Instead of making the processor work on physical addresses, can we make it work on these fake addresses and execute programs?

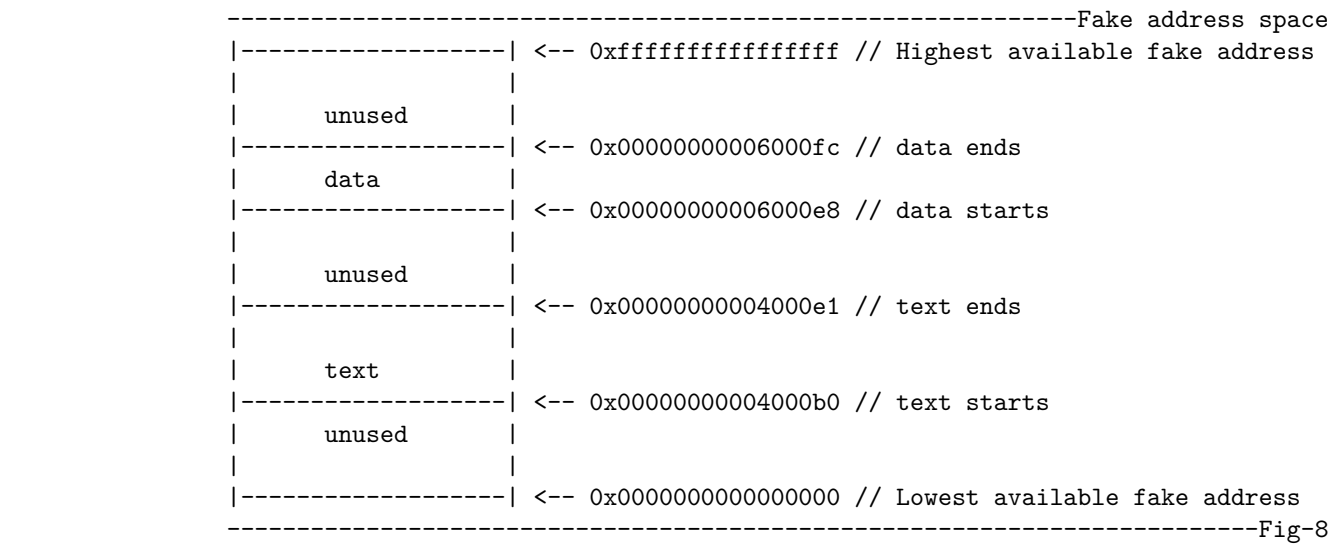
We have memory for 1 instruction(max size = 15 bytes) and 1 data element(max size = 8 bytes). Before program execution, the complete program is in secondary storage. We can execute the program instruction by instruction. We load the first instruction from secondary storage into physical memory at address 0x00. Execute it. If some data element is needed, fetch it from secondary storage and put it into the physical memory alloted data element. That is the plan.

On file, the program looks like this.



On file, the program's size is still sizeof(text) + sizeof(data) = 54 + 21 = 75 bytes. The file-offset of text is 0x00 ie., text is at the beginning of the file. data's File-offset is 0x36.

On the fake address range, the program looks like this. Because we are dealing with a 64-bit CPU, highest fake address is 0xffffffffffffff.



The operating system loads the fake address of the first instruction(which is 0x00000000004000b0) into the instruction-pointer register(**rip**) and leaves. The processor looks at the physical address 0x00 where an instruction should be present. How does it know if the bytes present at physical address 0x00 are the instruction bytes or some garbage? It may actually pick some garbage value present at physical address 0x00 assuming them to be instruction bytes and continue execution - which is incorrect. What can be do to make sure that does not happen? We can store the fake address whose content(an instruction in this case) is present at physical address 0x00. The processor can use that to verify what instruction is present at physical address 0x00. We can a similar entry for the data element. Lets form a small 2-entry table for this. Initially, this is how the table looks like.

Physical Address	Fake Address
0x00	———
0x0f	———

Call this table address-mapping table. Because it maps the two physical addresses to Fake addresses.

The processor refers to the mapping table and sees that the physical address 0x00(the location where instruction is present) has an empty fake address entry. This means no instruction is present at physical address 0x00. It looks at the **rip** register and requests the operating system to fetch the instruction at the fake address 0x00000000004000b0. Can you tell what instruction situated at the fake address 0x00000000004000b0? How will you fetch it? You know that the text starts with fake address 0x00000000004000b0. The first instruction would be situated at a file-offset of (0x00000000004000b0 - 0x00000000004000b0) = 0 into the file. With the file-offset, you can fetch the instruction bytes without any problem. The operating system fetches and loads **4d-31-ff**(xor r15, r15) at the physical address 0x00. The operating system also updates the corresponding table entry. The processor fetches the instruction from the physical address and executes it. The table and user-space memory looks like this.

Physical Address	Fake Address
0x00	0x00000000004000b0
0x0f	———

```
--------------------------------------------------------------------------------
0x00: 4d 31 ff 00 00 00 00 00 00 00 00 00 00 00 00  ; xor r15, r15
0x0f: 00 00 00 00 00 00 00 00                       ; No data element
--------------------------------------------------------------------------------
```

The instruction is 3-byte long and is situated at the fake address 0x00000000004000b0. This means that the next instruction is at the fake address (0x00000000004000b0 + 3) = 0x00000000004000b3. The processor looks at the table, makes sure that the instruction at fake address 0x00000000004000b3 is not present in physical memory and requests the operating system to fetch the instruction. I want you to calculate the file-offset of the instruction with fake address 0x00000000004000b3. It is present at a file-offset of 3. The OS fetches the second instruction **4c-8d-34-25-e8-00-60-00**(lea r14, 0x6000e8) and loads it into physical memory. The operating system updates the table entry. Note that the second instruction replaces the first instruction in the physical memory. The processor executes the second instruction. The table and physical memory looks like this.

Physical Address	Fake Address
0x00	0x00000000004000b3
0x0f	———

```
--------------------------------------------------------------------------------
0x00: 4c 8d 34 25 e8 00 60 00 00 00 00 00 00 00 00  ; lea r14, 0x6000e8
0x0f: 00 00 00 00 00 00 00 00                       ; No data element
--------------------------------------------------------------------------------
```

The processor knows that the second instruction is 8 bytes long. It calculates the fake address of the next instruction: (0x00000000004000b3 + 8) = 0x00000000004000bb. 0x00000000004000bb is loaded into **rip**. This instruction is interesting because it has a data access. As usual, the instruction is placed in the main memory by the operating system and the table is updated. The processor tries to execute it. The instruction is **4c-89-34-25-f5-00-60-00**(mov qword[0x6000f5], r14) - contents of register r14 is written into the data element at fake address 0x6000e5. This is how the table looks and physical memory looks like now.

Physical Address	Fake Address
0x00	0x00000000004000bb

Physical Address	Fake Address
0x0f	———

```
--------------------------------------------------------------------------------
0x00: 4c 89 34 25 f5 00 60 00 00 00 00 00 00 00 ; mov qword[0x6000f5], r14
0x0f: 00 00 00 00 00 00 00 00                   ; No data element
--------------------------------------------------------------------------------
```

The processor goes ahead and executes the instruction. It tries to access the data element and checks the mapping table to see if if the data element at fake address 0x6000f5 is present in physical memory. It comes to know that the data element is **not** present in physical memory. It stops the execution temporarily and requests the operating system to fetch it from secondary storage and put it into physical memory. How will you fetch a data element at fake address 0x6000f5? You know that the data's starting fake address is 0x6000e8. This means the data element we want is present at an offset of (0x6000f5-0x6000e8) = 13. You also know that data is present at a file-offset of 0x36. With that, you can calculate the file-offset of the required data element as (0x36 + 13) = 0x43. The operating system fetches 8 bytes present at the file-offset 0x43 and loads it into physical memory at 0x0f and it updates the mapping table entry. The table and memory looks like this.

Physical Address	Fake Address
0x00	0x00000000004000bb
0x0f	0x00000000006000f5

```
--------------------------------------------------------------------------------
0x00: 4c 89 34 25 f5 00 60 00 00 00 00 00 00 00 ; mov qword[0x6000f5], r14
0x0f: 00 00 00 00 00 00 00 00                   ; 8 bytes at 0x6000f5 are 0s
--------------------------------------------------------------------------------
```

Now that the data is present in main memory, the processor goes ahead and executes the instruction. This instruction copies contents of register r14(which is 0x00000000006000e8) into the data element. After the instruction is executed, this is how the user-space memory looks like.

```
--------------------------------------------------------------------------------
0x00: 4c 89 34 25 f5 00 60 00 00 00 00 00 00 00 ; mov qword[0x6000f5], r14
0x0f: 00 00 00 00 00 60 00 e8                   ; After execution
--------------------------------------------------------------------------------
```

That was a data-write. Later when another data element is accessed by the processor, this data is overwritten. But when the processor changes the value of a data element, this change in value should be reflected in the secondary storage right? Whenever data-write happens, it needs to be **written back** to the secondary storage. How can this write-back mechanism be implemented? It is quite simple. We can have a single bit of information which tells us if the data has changed by the processor. If it is changed(or written by the processor), let us set that bit to 1, else it will remain 0. The table has a new column now. Before the processor wrote into the data element, the table looked like this.

Physical Address	Fake Address	Changed bit
0x00	0x00000000004000bb	FALSE
0x0f	0x00000000006000f5	FALSE

The processor wrote the value 0x00000000006000e8 into the data element. The processor must update

the data element's Changed-bit. After updating,

Physical Address	Fake Address	Changed bit
0x00	0x00000000004000bb	FALSE
0x0f	0x00000000006000f5	TRUE

The processor computes the fake address of the next instruction: (0x00000000004000bb + 8) = 0x00000000004000c3. It updates **rip** and checks if that fake address present in the table. It is not present, so it requests the operating system to fetch the instruction at fake address 0x00000000004000c3. The instruction is **4c-8b-34-25-f5-00-60-00**(mov r14,qword[0x6000f5]). It is loaded into physical memory and processor tries to execute it. This instruction has a data access. It needs 8 bytes present at the fake address 0x00000000006000f5. It checks the table to see if physical memory has the data element at fake address 0x00000000006000f5 and yes, it does. The processor goes ahead and reads the value and executes the instruction. The table and physical memory looks like this,

Physical Address	Fake Address	Changed bit
0x00	0x00000000004000c3	FALSE
0x0f	0x00000000006000f5	TRUE

```
--------------------------------------------------------------------------------
0x00: 4c 8b 34 25 f5 00 60 00 00 00 00 00 00 00 ; mov r14, qword[0x6000f5]
0x0f: 00 00 00 00 00 60 00 e8                   ; Updated by third instruction
--------------------------------------------------------------------------------
```

Processor computes the fake address of the next instruction: (0x00000000004000c3 + 8) = 0x00000000004000cb. The instruction is **45-8a-3e**(mov r15b, byte[r14]). Processor updates **rip** and checks if the instruction is present in main memory. It sees that the instruction is not present. It requests the operating system to fetch the instruction and place it in physical memory. Once fetching is done, this is how the table and main-memory looks like.

Physical Address	Fake Address	Changed bit
0x00	0x00000000004000cb	FALSE
0x0f	0x00000000006000f5	TRUE

```
--------------------------------------------------------------------------------
0x00: 45 8a 3e 00 00 00 00 00 00 00 00 00 00 00 ; mov r15b, byte[r14]
0x0f: 00 00 00 00 00 60 00 e8                   ; Updated by third instruction
--------------------------------------------------------------------------------
```

The processor goes ahead and executes the instruction. But note that the instruction requests access to a byte of data at r14(ie., at fake address 0x6000e8). It checks the table and sees that data at fake address 0x6000e8 is not present in the RAM. It stops the execution and requests the operating system to fetch the data element at fake address 0x6000e8. The operating system first takes a look at the table. It sees that the data has been changed by some old instruction. This means that the operating system needs to write back the data element back to its respective fake address(here 0x6000f5) and only then load the new data element. The operating system does the write back and brings in the new data element. Now, the table and memory looks like the following.

Physical Address	Fake Address	Changed bit
0x00	0x00000000004000cb	FALSE
0x0f	0x00000000006000e8	FALSE

```
--------------------------------------------------------------------------------
0x00: 45 8a 3e 00 00 00 00 00 00 00 00 00 00 00 ; mov r15b, byte[r14]
0x0f: 48 65 6c 6c 6f 20 57 6f                   ; 8 bytes at fake Address 0x6000e8
--------------------------------------------------------------------------------
```

The operating system by default fetches 8 bytes. Depending on the instruction, the processor decides how many bytes it should access. In this case, the instruction is `mov r15b, byte[r14]` => processor reads only the first byte and copies it into r15b.

With that, I will stop here. I request you to continue this simulation till the complete program is executed.

Look at how the write-back mechanism works. The operating system does not write back the changed data element the moment it was changed by the processor. The operating system waits for as long as possible - till another data element needs to be fetched. Only when it is high time, it writes back the changed data element. The operating system is being **lazy** and non-proactive here. It does work only when it needs to. You will find this lazy concept in a lot of places.

A few important observations.

1. Before the above exercise, the processor was a simple one. Text and data both were present in physical memory. The processor used the **physical address** and execute the program - straightforward. But in the above exercise, the processor did **not** work on physical addresses, instead it used the **fake addresses**. The CPU used the fake addresses to request instructions and data.
2. The CPU is still aware of physical memory and physical addresses. Even now, it has to fetch instruction and data from physical memory itself. It uses the mapping table to translate the fake address into the physical address, put the physical address in the address bus and fetch the instruction/data. Just that from a program execution perspective, physical addresses are not relevant.
3. Note that there was absolutely no relation between the fake addresses and physical addresses.
4. Observe that the only constraint on the fake address range is the CPU type. All the fake addresses are 64-bits(8 bytes) because a 64-bit CPU was used. This means the largest usable fake address is 0xffffffffffffffff(2^64-1).
5. The text and data can be placed at any fake address and the program would work without any problems. We placed text at 0x4000b0 and 0x6000e8 and we were able to simulate the program properly. Similarly, with any other addresses, it can be run.
6. What can you tell about the maximum size of a process which can be executed? The program we ran above was 75 bytes big - 54 bytes of text and 21 bytes of data. 23 bytes of physical memory was given for program execution. This means that we were able to execute a program whose size is greater than the size of available physical memory for program execution. Earlier, we had a system with 64-bit CPU and 3GiB of user-space memory. The process size was constrained by the amount of physical memory available. On that system, the size of the largest process is 3GiB and nothing more could be handled. But look at it now. We have a 64-bit CPU and 23 bytes in user-space memory. With the old methods, we could not have run a process whose size is more than 23 bytes. But with a few (significant) changes to the CPU, a couple of data structures and changes in the memory manager, we were able to run a 75-byte process on it. What is the size of the largest process that can be run on our new system? You can see that the program size is now constrained by the **size of the fake address range**. Because it is a 64-bit CPU, the size of fake address range is 2^64 or 18,446,744,073,709,551,616 bytes which is unbelievably big!

Virtually, there is no limit on the process size.

Our original problem to solve was this: Can we run a program whose size is larger than the available user-space physical memory? We now have a definitive answer for that. Yes, we can run large processes on systems with small physical memories, the size of the processes is constrained by the size of the fake address range.

Before we move forward, let us put whatever we discussed so far in a formal manner.

There are 2 address spaces now. **Physical Address Space** and **Fake Address Space**.

1. **Physical Address Space**: As discussed on the previous chapter, the address bus length decides the size of the physical address space. Devices like ROM, RAM, Keyboard etc., are part of the physical address space. All the data elements in each of these devices is given a unique physical address. A physical address can either be unused or it can be used to access some data element of some device.

- The current-day Intel 64-bit CPUs have a 64-bit address bus - the physical address space has 2^64 addresses. The following is the physical address space of my system.

```
--------------------------------------------------------------------------------
chapter3$ sudo cat /proc/iomem
00000000-00000fff : Reserved
00001000-00057fff : System RAM
00058000-00058fff : Reserved
00059000-0005efff : System RAM
0005f000-0005ffff : Reserved
00060000-0009efff : System RAM
0009f000-0009ffff : Reserved
000a0000-000bffff : PCI Bus 0000:00
000c0000-000cfdff : Video ROM
000d0000-000d0fff : Adapter ROM
000f0000-000fffff : System ROM
00100000-bb96b017 : System RAM
bb96b018-bb978857 : System RAM
bb978858-bb979017 : System RAM
bb979018-bb989057 : System RAM
bb989058-bd336fff : System RAM
bd337000-bd7b2fff : Reserved
bd7b3000-cd1f3fff : System RAM
bda00000-be800e80 : Kernel code
be800e81-bf2507bf : Kernel data
bf50b000-bf9fffff : Kernel bss
cd1f4000-cd238fff : Reserved
cd239000-d9dbafff : System RAM
d9dbb000-d9ef2fff : Reserved
d9ef3000-d9f15fff : ACPI Tables
d9f16000-da686fff : System RAM
da687000-dade2fff : ACPI Non-volatile Storage
dade3000-dbafefff : Reserved
dbaff000-dbafffff : System RAM
dbb00000-dbffffff : RAM buffer
dd000000-df7fffff : Reserved
dd800000-df7fffff : Graphics Stolen Memory
df800000-feafffff : PCI Bus 0000:00
df800000-df9fffff : PCI Bus 0000:01
```

```
dfa00000-dfbfffff : PCI Bus 0000:01
dfc00000-dfdfffff : PCI Bus 0000:02
dfe00000-dfffffff : PCI Bus 0000:03
e0000000-efffffff : 0000:00:02.0
f0000000-f01fffff : PCI Bus 0000:04
f0200000-f03fffff : PCI Bus 0000:04
f6000000-f6ffffff : 0000:00:02.0
f7000000-f70fffff : PCI Bus 0000:03
f7000000-f7001fff : 0000:03:00.0
f7000000-f7001fff : iwlwifi
f7100000-f71fffff : PCI Bus 0000:02
f7100000-f7103fff : 0000:02:00.0
f7104000-f7104fff : 0000:02:00.0
f7104000-f7104fff : r8169
f7200000-f720ffff : 0000:00:14.0
f7200000-f720ffff : xhci-hcd
f7210000-f7213fff : 0000:00:1b.0
f7210000-f7213fff : ICH HD audio
f7214000-f7217fff : 0000:00:03.0
f7214000-f7217fff : ICH HD audio
f7218000-f72180ff : 0000:00:1f.3
f7219000-f72197ff : 0000:00:1f.2
f7219000-f72197ff : ahci
f721a000-f721a3ff : 0000:00:1d.0
f721a000-f721a3ff : ehci_hcd
f721c000-f721c01f : 0000:00:16.0
f721c000-f721c01f : mei_me
f7fe0000-f7feffff : pnp 00:05
f7ff0000-f7ffffff : pnp 00:05
f8000000-fbffffff : PCI MMCONFIG 0000 [bus 00-3f]
f8000000-fbffffff : Reserved
f8000000-fbffffff : pnp 00:05
fec00000-fec00fff : Reserved
fec00000-fec003ff : IOAPIC 0
fed00000-fed03fff : Reserved
fed00000-fed003ff : HPET 0
fed00000-fed003ff : PNP0103:00
fed10000-fed17fff : pnp 00:05
fed18000-fed18fff : pnp 00:05
fed19000-fed19fff : pnp 00:05
fed1c000-fed1ffff : Reserved
fed1c000-fed1ffff : pnp 00:05
fed1f410-fed1f414 : iTCO_wdt.0.auto
fed1f800-fed1f9ff : intel-spi
fed20000-fed3ffff : pnp 00:05
fed45000-fed8ffff : pnp 00:05
fed90000-fed90fff : dmar0
fed91000-fed91fff : dmar1
fee00000-fee00fff : Local APIC
fee00000-fee00fff : Reserved
ff000000-ffffffff : Reserved
ff000000-ffffffff : INT0800:00
```

```
ff000000-ffffffff : pnp 00:05
100000000-11f7fffff : System RAM
11f800000-11fffffff : RAM buffer
-------------------------------------------------------------------------------
```

- It starts with physical address 0x0000000000000000 and goes till 0x000000011ffffff. 4,831,838,208 (0x120000000) physical addresses have been used. But the highest address is 0xffffffffffffffff(2^64-1). It can still accomodate a lot of physical devices.
- Consider a 32-bit Intel System. It has a 32-bit address bus. The physical address space is 4GiB(2^32) in size. Address range is 0x00000000 to 0xffffffff. I am running a 32-bit ubuntu virtual machine. The following is its physical address space.

```
-------------------------------------------------------------------------------
vm/chapter3$ sudo cat /proc/iomem
00000000-00000fff : reserved
00001000-0009fbff : System RAM
0009fc00-0009ffff : reserved
000a0000-000bffff : PCI Bus 0000:00
    000a0000-000bffff : Video RAM area
000c0000-000c7fff : Video ROM
000e2000-000ef3ff : Adapter ROM
000f0000-000fffff : reserved
    000f0000-000fffff : System ROM
00100000-3ffeffff : System  RAM
    01000000-017e7e53 : Kernel code
    017e7e53-01bcad3f : Kernel data
    01cbe000-01d85fff : Kernel bss
3fff0000-3fffffff : ACPI Tables
40000000-fdffffff : PCI Bus 0000:00
    e0000000-e0ffffff : 0000:00:02.0
    f0000000-f001ffff : 0000:00:03.0
        f0000000-f001ffff : e1000
    f0400000-f07fffff : 0000:00:04.0
        f0400000-f07fffff : vboxguest
    f0800000-f0803fff : 0000:00:04.0
    f0804000-f0804fff : 0000:00:06.0
        f0804000-f0804fff : ohci_hcd
    f0806000-f0807fff : 0000:00:0d.0
        f0806000-f0807fff : ahci
fec00000-fec00fff : reserved
fee00000-fee00fff : Local APIC
    fee00000-fee00fff : reserved
fffc0000-ffffffff : reserved
-------------------------------------------------------------------------------
```

If you need more details about physical address space, about how the processor can read/write into a particular physical address and more, please refer to previous chapter's init.

2. **Fake Address Space**: This is the concept which helped us run processes with size greater than available usable main memory. The various sections of a program are laid down by the linker on this fake address space. This address space is not real aka it is **virtual**. It is also known as **Virtual Address Space** and the fake addresses are known as **Virtual Addresses**. The current day 64-bit Intel CPU supports 64-bit virtual address space. This means that a process can be 2^64 bytes big.

This was an short introduction to Virtual Address Space. The programmer(and tools like linker) are given an illusion that virtually infinite memory is present in the system because a huge address space is given for the program. This is called **Virtual Memory**. Its called virtual memory because so much memory does not exist in the system, but the operating system presents itself as if so much memory is available. Because of Virtual Address Space, the linker does not have to care about the addresses given to the program. It can give any address to it irrespective of the amount of physical memory present - this is one of biggest advantages of Virtual Memory. Virtual Address Space, Virtual Memory are all unbelievably awesome concepts. We will slowly discuss all of them as we move forward.

We solved the problem of running programs with size greater than the available user-space memory. But the solution came at some cost and complexity. The fundamental nature of CPU was changed - it was working with real, physical addresses but now it is working with virtual addresses. For every instruction, every data element, the processor had to stop and request the operating system for the instruction(or data) and the operating system accesses the secondary storage and fetches the instruction(or data). For every instruction(or data), the processor had to refer to the new data structure, the mapping table. We currently have a costly solution. Either we need to reduce the cost or discard the solution and come up with something else.

The hand-simulation we did was hypothetical. A practical system will never have memory as small as 23 bytes. Let us consider a practical system and see if the cost can be reduced. Consider a system with 4GiB RAM and a 64-bit CPU which supports virtual addresses. Assume that 3GiB of RAM is given to run processes. Assume that the physical addresses 0x00-00-00-00-00-00-00-00 to 0x00-00-00-00-bf-ff-ff-ff can be used to refer to the 3GiB of user-space RAM. Because it is a 64-bit CPU which can understand virtual addresses, the size of virtual address space is 2^64(0x0000000000000000-0xffffffffffffffff). Task: How do we run a 5GiB(2GiB of text, 3GiB of data) large process? Let us assume that text starts from virtual address 0x400000. Because its size is 2GiB, virtual address range for text = 0x0000000000400000 - 0x00000000803fffff. Data starts at 0x0000000080400000 and goes till 0x00000001403fffff. Note that text and data don't have to be contiguous. We have kept them contiguous for the sake of simplicity.

With 3GiB of RAM in hand, we don't have to execute the bring in one instruction at a time from secondary storage to RAM and then execute it. Because there is lot of RAM available, parts of text and data can be fetched in before hand in bulk so that we can save time on fetching instructions/data from secondary storage again and again. But note that the fetch/write-back will not stop because the process is larger than the available user-space memory. To start with, let us place first 1GiB of text and 2GiB of data in physical memory. Once this is placed, we need to have a mapping table for all the valid virtual addresses. At any point, the table should know whether a virtual address is present in main memory or in secondary storage - just like the table we used in the previous example. But there, it was only a matter of 1 instruction and 1 data element. Here, there are way more physical addresses. So we can have a table like this. Note that first 1GiB of text and first 2GiB of data are in RAM.

Virtual Address	Present in RAM	Physical Address
0x0000000000400000	YES	0x0000000000000000
0x0000000000400001	YES	0x0000000000000001
.	.	.
.	.	.
.	.	.
0x00000000403fffff	YES	0x000000003fffffff
0x0000000040400000	NO	———
0x0000000040400001	NO	———
.		
.		
.		

Virtual Address	Present in RAM	Physical Address
0x00000000803fffff	NO	———
0x0000000080400000	YES	0x0000000040000000
0x0000000080400001	YES	0x0000000040000001
.	.	.
.	.	.
.	.	.
0x00000001003fffff	YES	0x00000000bfffffff
0x0000000100400000	NO	———
0x0000000100400001	NO	———
.	.	.
.	.	.
.	.	.
0x00000001403fffff	NO	———

Slowly go through the table. It has entries for every single valid virtual address. Note that only valid virtual addresses are 0x0000000000400000 - 0x00000001403fffff because text and data fall into that range of virtual addresses. Other virtual addresses are invalid.

Let us start the execution. The first instruction is at the virtual address 0x400000. The processor checks the mapping table to see if it is present. The byte at 0x0000000000400000 is present at physical address 0x0000000000000000. Assuming that the instruction is 8 bytes, the processor checks if all 8 bytes are present in main memory. It confirms that it is present. It executes the instruction.

Assume that a large function is about to be executed. Assume that its first instruction is 2 bytes long and is present at virtual address 0x00000000403ffffe. The processor checks and executes the instruction. It calculates the virtual address of the next instruction: (0x00000000403ffffe + 2) = 0x0000000100400000. The processor refers the table and comes to know that the next instruction's bytes are not present in the physical memory. Now, the processor requests the operating system to fetch the instruction at virtual address 0x0000000100400000 and put it in physical memory. Assume that the next instruction is placed at the physical address 0x0000000000000000. Assuming that the next instruction is 4 bytes in size, this is how the table would look like now. Only the updated entries are put below.

Virtual Address	Present in RAM	Physical Address
0x0000000000400000	NO	———
0x0000000000400001	NO	———
0x0000000000400002	NO	———
0x0000000000400003	NO	———
0x0000000000400004	YES	0x0000000000000004
0x0000000000400005	YES	0x0000000000000005
0x0000000000400006	YES	0x0000000000000006
0x0000000000400007	YES	0x0000000000000007
.	.	.
.	.	.
.	.	.
0x00000000403fffff	YES	0x000000003fffffff
0x0000000040400000	YES	0x0000000000000000
0x0000000040400001	YES	0x0000000000000001
0x0000000040400002	YES	0x0000000000000002
0x0000000040400003	YES	0x0000000000000003

Virtual Address	Present in RAM	Physical Address
0x0000000040400004	NO	———
.		

The next instruction is in physical memory, the processor executes it. Question is, on what basis did we overwrite the bytes at physical address 0x0000000000000000? We could have overwritten some other bytes too?

The processor gets the virtual address of the next instruction: (0x0000000040400000 + 4) = 0x0000000040400004. It checks the table and sees that those bytes are not present in the memory. Request the operating system and wait till the bytes are loaded at some physical address. This is the case with rest of the function's instructions. Even though we have enough physical memory to load (say) all the instructions of the function in bulk, we are not doing it. We are doing it the old-fashioned way - fetching one instruction at a time which is very costly. Similarly, assume that a large array needs to be traversed and only the first byte is present in the physical RAM. For the rest of the array, we being in element by element. We changed the initial load mechanism by loading lots of text and data. But once the process is running, we are still using the one element/request mechanism - which is inefficient. We need to be able to bring in lots of bytes at a time so that the cost of fetching reduces. Over the years, a lot of programs have been written and their behavior has been studied in great detail. One of the characteristics which will help us is the memory access patterns of a program. Two observations about memory accesses of programs are

1. If a program accesses(reads from/writes into) a memory location, it is highly likely that the program will access the same memory location again.
2. If a program accesses a memory location, it is highly likely that the program will access its neighboring memory locations.

Read these observations again. They reinforce our examples in the above paragraph. When the processor accesses an instruction, it is highly likely that the instruction next to it will be accessed next. If an array element A[i] is accessed, it is highly likely that A[i-1] or A[i+1] gets accessed next. These are not rules, but just observations and are generally true. There are definitely exceptions to it. If the instruction is a **jmp** or **call**, then our observation mostly fails because it is highly unlikely that the function or label is the next instruction(there is an exception to this too :P). Consider the iterator used to access that array. It will be accessed again and again either to read some array element or for increment/decrement etc.,

Let us talk about the mapping table. Currently, every valid virtual address has an entry. This requires a significant part of the operating system's memory.

Based on the above arguments, we need to modify our memory manager. Instead of fetching instruction(or data) one by one, let us swap in or swap out a large number of bytes. Break the 3GiB of user-space memory into pieces of X bytes each. It can be viewed as a huge array where each array element is X bytes in size.

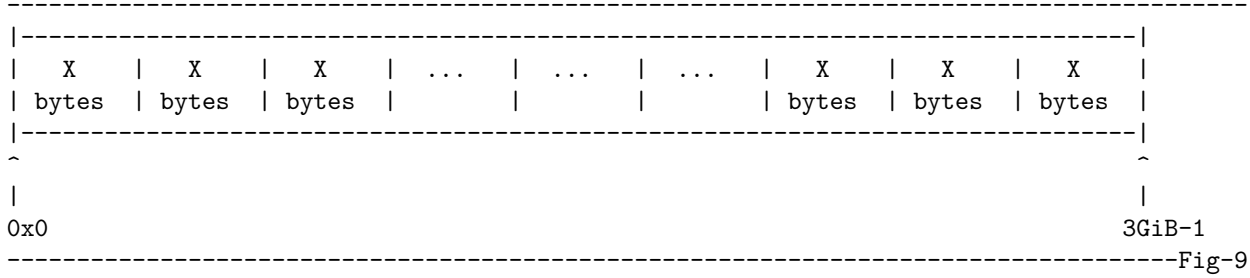
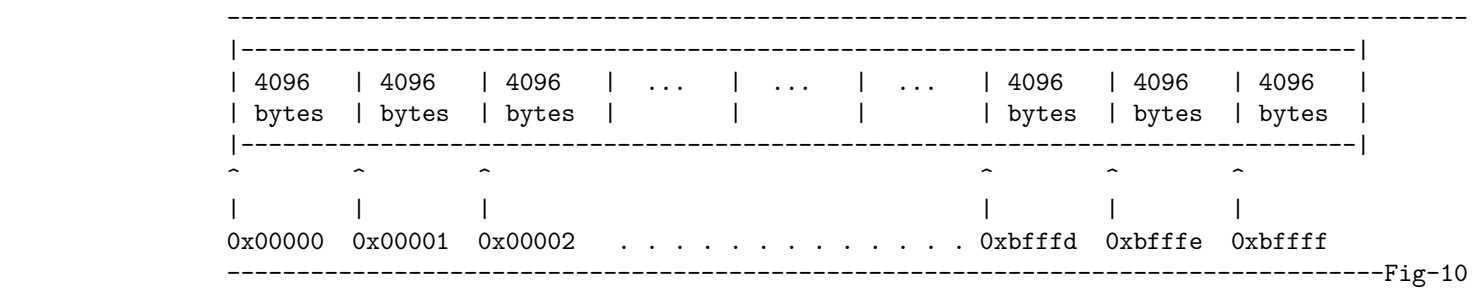


Fig-9

Each of these pieces is called a **frame**. How many such frames are present in 3GiB user-space memory? We will have (3GiB / X) number of frames. Assuming X to be 4KiB(4096 bytes) , there are 786432(or 0xc0000) frames. We can give a unique identifier to each frame - 0x00000 to 0xbffff. This is how it looks now.



In the exact same way, let us break the virtual address space into pieces. Call these **virtual pages**. Idea is that each virtual page will get copied onto a physical frame. Size of the virtual address space is 2^64. It can be divided into 4,503,599,627,370,496(or 0x10000000000000) pages. But the virtual pages which containing parts of our program are relevant to us. Given a virtual address, how do you calculate the page it is in? You simply divide the virtual address by size of the virtual page. Take 0x0000000000400000 for example. It belongs to virtual page number 1024(or 0x0000000000400). It is as simple as that. A virtual address / physical address here is 64-bits long. With a page/frame size of 4KiB(4096 bytes), what is the size of the page/frame ID? Its 52-bits in size.

Let us list all the valid virtual pages now. First valid virtual address is 0x0000000000400000 => page ID is 0x0000000000400. Last valid virtual address is 0x00000001403fffff => page ID is 0x000000001403ff. The virtual page IDs of our program 0x0000000000400 to 0x00000001403ff.

With that, let us discuss how transactions happen. In the simulation, individual instructions/data elements were fetched. Later, byte was the unit of transaction. After some discussion, now we a page. A page is the unit of transaction. A page gets fetched from secondary storage into physical memory and is copied onto some frame, a page gets swapped out of physical memory making space for some other page. Here, the page size is 4096 bytes. Just to make it clear, we have 3 things here: **Physical frame** is part of the physical memory, **Virtual page** is part of the virtual address space and **page** is a unit of measurement, the unit of transaction. Before the program executes, we put a few text virtual pages and data virtual pages onto physical memory. We put 1GiB of text and 2GiB of data just like before. But remember that everything will be in terms of pages, even the mapping table. We will know which virtual page is mapped to which physical frame if it is present. This is how the mapping table looks like.

Virtual Page number	Present in RAM	Physical Frame number
0x0000000000400	YES	0x0000000000001
0x0000000000401	YES	0x0000000000002
.	.	.
.	.	.
.	.	.
0x00000000403ff	YES	0x000000003ffff
0x0000000040400	NO	—
0x0000000040401	NO	—
.	.	.
.	.	.
.	.	.
0x00000000803ff	NO	—
0x0000000080400	YES	0x0000000040000

Virtual Page number	Present in RAM	Physical Frame number
0x0000000080401	YES	0x0000000040001
.	.	.
.	.	.
.	.	.
0x00000001003ff	YES	0x00000000bffff
0x0000000100400	NO	—
0x0000000100400	NO	—
.	.	.
.	.	.
.	.	.
0x00000001403ff	NO	.

You can observe that the number of entries in the table have reduced and number of bytes needed to store a single entry has also reduced.

With that, we are ready to go. If the processor wants to check if a virtual address is present in memory, then what should it do? It needs to find the virtual page that virtual address is in and then check the mapping table. You know how to get the virtual page from a virtual number. From here, program execution will be a bit faster and amount of metadata(like tha table) takes lesser space than before.

There is one question left: When a virtual page needs to be swapped in, some other page needs to get swapped out. Of all the physical frames, which one should we swap out? Think about it.

With that, let us end this thread.

We had 2 discussion threads: The first thread introduced the concept of time sharing and multi-tasking, swapping etc., The above thread introduced the concept of Virtual Address Space, frames, pages, mapping table etc., Each of the threads tried to solve 2 different problems.

I want to combine these two threads now.

We ended the first thread by calculating the cost of time-sharing vs straight-forward ordering. The following is what we found when there were 5 programs to run. Note that in the first thread, we were using simple processors which worked on physical addresses and programs' size was not greater than user-space memory.

1. Straight-forward Ordering:
   - Total runtime = (T1 + T2 + T3 + T4 + T5) + [Ceil(sizeof(Pi) / X) * T_Load][i=1 to 5] + [Ceil(sizeof(Pi) / X) * T_Clean][i = 1 to 5]
2. Round-Robin ordering
   - Total runtime = (T1 + T2 + T3 + T4 + T5) + [Ceil(Ti/T) * Ceil(sizeof(Pi)/X) * T_Load][i = 1 to 5] + [Ceil(Ti/T) * Ceil(sizeof(Pi)/X) * T_Clean][i = 1 to 5].

Before we start any discussion, let us assume a couple of things. Consider a 3GiB user-space memory and a 64-bit CPU. The following describes each of the 5 processes.

1. P1: 512MiB, 384MiB text, 128MiB of data
2. P2: 1GiB, 256MiB text, 768MiB of data
3. P3: 1.5GiB, 1GiB of text, 512MiB of data
4. P4: 2GiB, 512MiB of text, 1.5GiB of data
5. P5: 2.5GiB, 1.5GiB of text, 1GiB of data

Text will always start at physical address 0x0000000000000000 and data will follow the text. With that, we are ready to go.

Total runtime of round-robin ordering is significantly higher because of the sheer loads and cleanups done. Time-sharing is a very useful concept, but at the moment a costly one. We need to reduce its cost. Cost is direct related to the number of loads and cleanups. We reduce the number of loads and cleanups, we reduce the cost. How do we do it? Try taking hints from the second thread.

The operating system starts with executing P1. Currently, complete P1 will be copied to main memory. Later, complete P1 is removed once its T units of time is over and again when its turn comes, complete P1 is copied back. When P1 starts its execution, the first instruction is accessed. It is highly likely that it accesses the first few instructions when the execution has just begun. For data, it is quite unpredictable. One idea is to break the complete program into pages and load them on a **demand** basis. When the program is about to run, let us load the first few pages from text because it is highly likely that they will be accessed. Data access is unpredictable. We have 2 choices - We can be conservative about loading data pages and not load any data pages in the beginning. Once the CPU accesses a data element, let us load the page containing that data element. Or load a couple of data pages in the beginning. Assuming page size to be 4KiB(4096 bytes), exactly how many text pages should we load? Let us calculate a rough number. Assuming the processor can execute 1 billion instructions per second(1,000,000,000 instructions/second), T to be 5ms, number of instructions executed in 5ms = (1,000,000,000 instructions/second * 5ms) = 5,000,000 instructions. Size of a x64 instruction can be anywhere from 1 byte to 15 bytes. We should ideally take the weighted average to get the average instruction size, but to calculate the weighted average, we need the actual program and instructions. We don't have the program. Let us simply take the average. Average instruction size is (1 + 15)/2 = 8 bytes/instruction. Number of bytes to load initially = (5,000,000 instructions * 8 bytes/instruction) = 40,000,000 bytes. With page-size = 4096 bytes/page, 40,000,000 bytes is equivalent to (40,000,000 bytes/(4096 bytes/page) = 9765.625 pages. Let us round off and load the first 10,000 text pages. Along with that, let us load 10,000 data pages too. There is no need for a mapping table here. We are not mapping addresses. We just need to know if a page is present or not. Let us have set of page numbers of the pages present in main memory. That should suffice. Size of P1 is 512MiB which amounts to 131072(or 0x20000) pages. Out of those pages, text falls under pages 0x0000000000001 to 0x0000000017fff and data under 0x0000000018000 to 0x000000001ffff. The following is the list.

List of page numbers
0x0000000000000
0x0000000000001
0x0000000000002
.
.
.
0x000000000270f
0x0000000018000
0x0000000018001
.
.
.
0x000000001a70f

At the end of P1's 5ms, let there be 20,500 pages. Now, it is P2's turn. Same analysis can be done on P2 and 20,000(10,000 text and 10,000 data) pages can be loaded. Before loading, all of P1's pages should be swapped back to secondary storage and the operating system should preserve P1's set of page numbers so that when its P1's turn again, the operating system knows what pages to bring back. All the physical frames where P1's pages were present needs to be cleaned up. Similar to P1, P2 also

gets executed. Assume that in P2's 5ms, there were 21,000 pages in memory. Similar number of pages were loaded during P3, P4 and P5's execution. When its P1's turn again, the operating system takes a look at at P1's set of page numbers and loads them. Obviously, the operating system will have to load newer pages as the process runs. As time progresses, number of pages of a process in memory steadily increases. It is highly likely that many of these pages are never used - because the new pages are used and not the old ones. This means that the number of pages increases steadily. This unnecessarily increases the load and cleanup time. This means, the operating system should keep track of what pages are being accessed, how frequently these pages are being accessed. The unused ones, seldom used pages can be swapped out of the main memory which will contribute in reducing the load and cleanup time. Note that the operating system should not carelessly swap out frequenty used, **hot** pages - this will prove to be very costly. It should strategically swap out unused, **cold** pages.

Assume that the above concept is implemented in the operating system. Say an average of 20,000 pages were always present in the main memory. You can easily calculate the load-cleanup cost. 20,000 pages amounts to 81,920,000 bytes = 78.125MiB. Compare loading and cleaning 78.1215MiB after every process with loading and cleaning complete processes with sizes 0.5GiB, 1GiB, 1.5GiB, 2GiB and 2.5GiB. Cost is definitely reduced. This is good. But observe the average amount of memory used. There is 3GiB of user-space memory available and about 80MiB of it is being used. This is an case of extreme underutilization of resources. What can be done about this?

Why is the cleanup done? It is done so that a process's data remains private and no other processes can get hold of its data. Currently, relevant pages of a process is loaded, the process runs for T units of time, it is cleaned up and the relevant pages of the next process is loaded. These relevant pages are placed in the same physical memory. The same physical memory is cleaned up when its the next process's turn. Instead of that, we can place relevant pages of different processes in different parts of physical memory. 3GiB user-space memory means we have 786432(0xc0000) frames in hand. If we allocate 20,000 frames for each process, a total of 100,000 frames will be used. Instead of cleanup, the operating system should protect a process's frames against other processes. No one other than that process should be access its frames. This way, we won't even need cleaning at the end of T units of time. All the relevant pages of all the processes will always be in memory - obviously once a particular process is done running, all its frames are cleaned up. In fact, more than 20,000 pages per process can be allocated now.

If this can be done, we can strike a balance between underutilization and the cost of loads and cleans. But with the current setup, do you think the above memory allocation can be done?

At the moment, our linker assumes that the complete user-space memory(3GiB: 0x0000000000000000 - 0x00000000bfffffff) is given to one single process and allots addresses. P1 is given addresses (0 to 512MiB-1). P2 is given addresses (0 to 1GiB-1) and so on. Unless each of these processes are copied at the linker-alloted addresses, they won't run properly. Eg: The linker would have given a data element the address 0x12345678 and instructions would be using this address to refer to that data element. When the program is about to run, that data element must be loaded at the physical address 0x12345678. If it is loaded at some other address, the programs won't run properly. This is exactly what will happen for P2, P3, P4 and P5. Each of these processes want physical memory from (0x00 to 512MiB-1). But with the above memory allocation, everything goes haywire. It is impractical to generate executables based on the unused physical memory left at that moment. The amount of unused physical memory obviously keeps changing - it is dynamic in nature. Programs should run irrespective of how much physical memory is allocated for it. Programs should run irrespective of where it is placed in the physical memory. The linker allotment of addresses should be independent of the availability of physical addresses. Do you know of any concept which will help us solve this problem?

Virtual Address Space and Virtual Memory can help us solve it. We used the concept of Virtual Memory to solve the problem of running large programs on systems with smaller main memory. It solved the problem are given 5 different physical memory allocations.and it also brought in a lot of

other features.

We have 5 processes in hand, with the above mentioned physical memory allocation. Instead of thinking this as 1 single problem, let us break it into 5 problems - we need to run 5 individual processes.

Let us take up Process P1. Its size is 512MiB(384MiB text and 128MiB data). The linker is now aware of the huge 64-bit virtual address space it is given. It does not have to worry about any aspect of physical memory now. Assume that the linker starts the text at virtual address 0x400000 and data following it. Virtual address range of text is 0x0000000000400000 - 0x00000000183fffff(384MiB), range of data is 0x0000000018400000 - 0x00000000203fffff(128MiB). text falls under the virtual pages 0x0000000000400 - 0x00000000183ff(98304 virtual pages), data falls under the virtual pages 0x0000000018400 - 0x00000000203ff(32768 virtual pages). As planned, let us place 20,000 virtual pages into the main memory. Note that the allocation need not be contiguous ie., a process's frames don't have to adjacent to each other. They can be scattered across the physical memory. There are 786432 physical frames with us and we can choose any 20,000 frames and copy P1's 20,000 virtual pages onto them. Create a mapping table for P1. Once all this is done, P1 is ready for execution.

P1's T units of time is done. The operating system takes it out of execution and its P2's turn. Let us repeat whatever we did for P1. P2 has its own 64-bit virtual address space in which its text and data resides. The linker may have given P2's text and data the same virtual addresses given to P1's text and data. Virtual address range of P2's text is 0x0000000000400000 - 0x00000000103fffff(256MiB) and data range is 0x0000000010400000 - 0x00000000403fffff(768MiB). P2's text falls under the virtual pages 0x0000000000400 - 0x00000000103ff(65536 virtual pages), data falls under virtual pages 0x0000000010400 - 0x00000000403ff(196608 virtual pages). There are (786,432 - 20,000) = 726,432 physical frames available. Choose any 20,000 frames and copy P2's 20,000 virtual pages onto them. Create a new, different mapping table for P2. Once all this is done, P2 is ready for execution.

Similarly, P3, P4 and P5 can be run.

Let us summarize whatever we discussed so far.

The idea of pages and Virtual Memory helped in accomodating processes with size greater than the available physical memory. In the first thread, we discussed about time-sharing and the cost paid. To reduce the the cost, we started by cutting the program and physical memory into pages. It reduced the cost but ended in memory underutilization. We wanted to increase memory utilization. Using Virtual Address Space did the job. Every process has its own Virtual Address Space, its own mapping table.

3.2 Virtual Memory and Security

All the programs we considered so far was divided into two parts - text and data. The text has pure machine code, data has variables, data structures. Because the text has instructions, the processor should be able to **read** it and then **execute** it. Basically the complete text should be readable and executable. Take data now. It has variables and data structures. The processor might have to read them, update(write into) them. There is no meaning in executing a data element. This means data should be readable and writable. These are basically security permissions - basically what can you do with a given section.

When the linker generates the executable, it should also generate security permissions for each of these sections. Let us take *hello.s*, the program we used in the previous section. Let us assemble and link it and inspect the permissions.

```
--------------------------------------------------------------------------------
chapter3$ nasm hello.s -f elf64
```

```
chapter3$ ld hello.o -o hello
```

As discussed in the first chapter, the executable's segments help in program execution. Let us check them out using readelf.

```
chapter3$ readelf --segments hello

Elf file type is EXEC (Executable file)
Entry point 0x4000b0
There are 2 program headers, starting at offset 64

Program Headers:
  Type           Offset             VirtAddr           PhysAddr
                 FileSiz            MemSiz              Flags  Align
  LOAD           0x0000000000000000 0x0000000000400000 0x0000000000400000
                 0x00000000000000e6 0x00000000000000e6  R E    0x200000
  LOAD           0x00000000000000e8 0x00000000006000e8 0x00000000006000e8
                 0x0000000000000015 0x0000000000000015  RW     0x200000

 Section to Segment mapping:
  Segment Sections...
   00     .text
   01     .data
```

Look at the first segment. Its starts at the virtual address 0x0000000000400000. There is also physical address given which is the same as the virtual address. As discussed, the linker does not have any information about physical addresses. So it fills in the virtual address itself in the place of physical address. Take a look at the **Flags** entry. Its **RE** which stands for **Readable** and **Executable**. THe first segment is nothing but the .text section. The second is the .data section which has flags **RW** which stands for **Readable** and **Writable**.

How do we implement this now? The executable has information about the segments' permissions. When the program is run, the main memory containing these segments should have the corresponding permissions. The frames containing the text section should be Readable and Executable and the data section frames should be readable and writable. If you think about it, the permissions about a particular page can be stored in the mapping table itself. The following is the modified mapping table.

Virtual Page number	Present in RAM	Physical Frame number	Permissions
0x0000000000400	YES	0x0000000000001	R-X
0x0000000000401	YES	0x0000000000002	R-X
.	.	.	.
.	.	.	.
.	.	.	.
0x00000000403ff	YES	0x000000003ffff	R-X
0x0000000040400	NO	—	R-X
0x0000000040401	NO	—	R-X
.	.	.	.
.	.	.	.
.	.	.	.
0x00000000803ff	NO	—	R-X

Virtual Page number	Present in RAM	Physical Frame number	Permissions
0x0000000080400	YES	0x0000000040000	RW-
0x0000000080401	YES	0x0000000040001	RW-
.	.	.	.
.	.	.	.
.	.	.	.
0x00000001003ff	YES	0x00000000bffff	RW-
0x0000000100400	NO	—	RW-
0x0000000100400	NO	—	RW-
.	.	.	.
.	.	.	.
.	.	.	.
0x00000001403ff	NO	.	RW-

Any violation of the security permissions ends up in process termination. You cannot write into text frames or execute data elements. It might actually seem crazy to execute data elements or write into text frames, but there lies the fun part! A lot of future chapters are about the same and you will see how important these permissions are. I have observed a strict rule in computer systems. Do/Give whatever is just needed, when needed. Anything more is unnecessary and can end up in screwing/helping attackers screw the system.

3.3 Shared Libraries and Position Independent Code

The processes we considered so far had 2 segments: text and data. These segments are private to a process. No other process should be able to access my text and data. In the first chapter, we discussed about code which can be compiled into a **shared library** / **shared object** which can be shared among various processes as the name suggests. Every shared object also has text and data - code to run and variables, data structures. With whatever we know, how can we implement this sharing mechanism?

In the first chapter, to understand shared objects, we used the following.

```
-------------------------------------------------------------------------------
chapter3$ cat lib.c
#include <stdio.h>

void lib_func()
{
    printf("Inside lib_func()\n");
}

chapter3$ cat lib.h
#ifndef _LIB_H
#define _LIB_H 1

void lib_func();

#endif  /* _LIB_H */
-------------------------------------------------------------------------------
```

lib.c is the simple shared library. It has one function `lib_func()` which has a simple print statement which when executed will prove that the shared library's code is executed.

Generate the shared library.

```
--------------------------------------------------------------------------------
chapter3$ gcc-4.8 lib.c -c -fPIC
chapter3$ gcc-4.8 lib.o -o libdummy.so --shared

--------------------------------------------------------------------------------
```

This generates a shared library with name *libdummy.so*.

Now, how do we actually share the shared object among various processes? Let us start by making the shared object part of one, single process. The executable just has the libraries' names it depends on. The shared libraries are separate files. When the executable is run, this is how its virtual address space looks like. The following is just an example.

```
--------------------------------------------------------------------------------
|------------------| <-- 0xffffffffffffffff // Highest virtual address
|                  |
|                  |
|                  |
|                  |
|                  |
|                  |
|      unused      |
|                  |
|                  |
|                  |
|                  |
|                  |
|------------------| <-- program's data ends
|       data       |
|------------------| <-- 0x0000000000600000 // program's data starts
|       unused     |
|------------------| <-- program's text ends
|       text       |
|------------------| <-- 0x0000000000400000 // program's text starts
|       unused     |
|------------------| <-- 0x0000000000000000 // Lowest virtual address
--------------------------------------------------------------------------------
```

Idea of a library is that it has functions, data structures which can be used by the program(and ideally shared among multiple programs). For the program to be able to access something, it must be part of the process's virtual address space. For the program to access the library, it should become part of the process's virtual address space. Basically, we should **map** the shared library's text and data onto the process's virtual address space. It should look something like this.

```
--------------------------------------------------------------------------------
|------------------| <-- 0xffffffffffffffff // Highest virtual address
|                  |
|                  |
|       unused     |
|                  |
```

```
|                   |
|-------------------|
|   dummy's data    |
|-------------------| <-- 0x0000123456900000 // libdummy.so's data starts
|   dummy's text    |
|-------------------| <-- 0x0000123456789012 // libdummy.so's text starts
|                   |
|       unused      |
|-------------------| <-- program's data ends
|       data        |
|-------------------| <-- 0x0000000000600000 // program's data starts
|       unused      |
|-------------------| <-- program's text ends
|       text        |
|-------------------| <-- 0x0000000000400000 // program's text starts
|       unused      |
|-------------------| <-- 0x0000000000000000 // Lowest virtual address
--------------------------------------------------------------------------------
```

The library has just one function `lib_func()`. Assume it is present at 0x0000123456789012, the beginning of the library's text. The program now can access that function through the virtual address 0x0000123456789012. This way, the program has access to the library. How will the mapping table look like now? It will have the library's entries too. Now, the library is part of this process. The library's text and data's virtual pages should be accounted for.

Now, one process has access to the library. There comes another process which uses the same library. Now, the library(or atleast some part of it) is already in memory and on demand, the necessary parts of the library can be brought back to memory. With that, we should obviously not load another copy for the sake of the new process. The new process needs to get access to the already present copy of the library. How can that be done? First of all, look at what parts of the library can be shared among multiple processes and what parts need to be private to a particular process. The library has 2 parts - text and data. We know that the text is readable and executable, it is **never** changed because it is not writable. But data is not like that. Data is writable. This means if a variable in the data segment is written into by a process, it should not be changed and must not have access to other processes. It is safe to conclude that all the non-writable parts can be shared among the processes and each process should have its own copy of the writable part of the library. Consider the Standard C library - **libc**. Almost all the processes use it. Its text is shared among all the processes but not the data segment.

That is the idea, but how do we implement it? Take some time and come up with an idea. Lets talk about just the sharable part of the library. Suppose it is already in the memory system(meaning parts of it are in main memory and other parts in secondary storage) and it is mapped to one process's virtual address space. When another process comes in which requires the same library, the same physical frames can be mapped to the second process's virtual space too. Basically, there is one copy of text present in physical memory but it is mapped onto two process's virtual address space. Regarding data segment, new memory needs to be allocated specifically for the second process and that needs to be kept private to the second process - the first process should not be able to access it. With this arrangement, how will the mapping tables of the two processes look like? There won't be much difference. The second process should also have new entries for the virtual pages of library's text and data. Again, note that only certain parts of a shared library can be shared.

We just specified that the text of the library needs to be mapped onto all the necessary processes' virtual address spaces and each process gets a private data segment. We didn't talk about **where** exactly in the virtual address space will the libraries be mapped? Should the library be mapped at the same virtual address ranges of all the processes? or how do you think it should be done? Is it practical

to map the library to the same virtual address ranges of all the processes? It is not. We need a flexible mechanism. There are 2 sides to this problem. One is the capability of the operating system to map the library onto any free(available) virtual address range of the process- the operating system should be able to map to any free(available) virtual address range of the process. How do we solve this problem? All we need to do is to make sure that the virtual addresses should not overlap with the program's virtual addresses. We can choose a really large virtual address as a **shared libraries' base virtual address** and start mapping each library from there. Suppose there are 2 libraries **libdummy1.so** and **libdummy2.so**. Let us map the first one now.

```
-------------------------------------------------------------------------------
|------------------| <-- 0xffffffffffffffff // Highest virtual address
|                  |
|                  |
|       unused     |
|------------------|<------- Shared Libraries' base address
|   dummy1's data  |
|------------------| <-- 0x0000123456900000 // libdummy1.so's data starts
|   dummy1's text  |
|------------------| <-- 0x0000123456789012 // libdummy1.so's text starts
|                  |
|                  |
|                  |
|                  |
|                  |
|                  |
|       unused     |
|------------------| <-- program's data ends
|       data       |
|------------------| <-- 0x0000000000600000 // program's data starts
|       unused     |
|------------------| <-- program's text ends
|       text       |
```

Where can we place *libdummy2.so*? Its pretty straight-forward. Instead of placing at some random place, it can be placed right after dummy1's text ends. The operating system can maintain the **next free virtual address**. With just that information, the problem is solved. dummy2's data and text can be placed at that next free virtual address. Once placed, that address will be updated. This way, the operating system has a simple mechanism to place the shared libraries.

Now, the other side of the problem: Any library can be placed at any random virtual address. This means, the code should work irrespective of where it is placed in the virtual address space. Do you think some special kind of code needs to be generated or the type of code generated for executables is enough for libraries too? Let us dig a little deeper. So far, we have generated only executables which are placed at known addresses and the linker generated the code keeping these addresses in mind. Consider the following program.

```
-------------------------------------------------------------------------prog.c
chapter3$ cat prog.c
#include <stdio.h>

int
main()
{
        char *str = "Hello World!";
```

```
        printf("main: %s\n", str);

        return 0;
}
```

We saw that the linker had alloted virtual addresses to Let us look at the **LOAD**able segments of this program. Let us see at what virtual addresses these loadable segments need to be mapped.

```
chapter3$ readelf --segments prog

Elf file type is EXEC (Executable file)
Entry point 0x400400
There are 9 program headers, starting at offset 64

Program Headers:
  Type           Offset             VirtAddr           PhysAddr
                 FileSiz            MemSiz              Flags  Align
  .
  .
  LOAD           0x0000000000000000 0x0000000000400000 0x0000000000400000
                 0x0000000000000708 0x0000000000000708  R E    0x200000
  LOAD           0x0000000000000e08 0x0000000000600e08 0x0000000000600e08
                 0x0000000000000228 0x0000000000000230  RW     0x200000
  .
  .
```

The first load segment needs to be loaded at the virtual address 0x400000. You can observe that this segment has a lot of sections in it - .text and .rodata(read-only data) sections are two of them.

Let us see in the virtual address range the linker has alloted to the above program's .text and .rodata. *readelf* tool can be used here.

```
chapter3$ readelf -S prog
.
.
[13] .text             PROGBITS         0000000000400400  00000400
       00000000000001a2  0000000000000000  AX       0     0     16
.
.
[15] .rodata           PROGBITS         00000000004005b0  000005b0
       000000000000001b  0000000000000000   A       0     0     4
```

Note that there are a lot of other sections too, but these two are enough to understand the scenario. Each and every detail above is interesting. The .text section is at an offset of 0x400 bytes from the segment-base(whose address is 0x400000). The .rodata section is at an offset of 0x5b0 bytes from the segment-base.

.text needs to be mapped at the virtual address 0x0000000000400400 and it is 0x1a2 bytes in size. The virtual address range of **.text** would be 0x400400 - 0x4005a2 if it is mapped at 0x400400. Similarly, **.rodata** needs to be mapped at the virtual address 0x00000000004005b0 and it takes 0x1b bytes of

memory and its virtual address range is 0x4005b0 - 0x4005cb. Let us take a quick look at `main()`'s disassembly.

```
-------------------------------------------------------------------------------
chapter3$ objdump -Mintel -d prog
.
.
00000000004004fd <main>:
  4004fd:       55                      push   rbp
  4004fe:       48 89 e5                mov    rbp,rsp
  400501:       48 83 ec 10             sub    rsp,0x10
  400505:       48 c7 45 f8 b4 05 40    mov    QWORD PTR [rbp-0x8],0x4005b4
  40050c:       00
  40050d:       48 8b 45 f8             mov    rax,QWORD PTR [rbp-0x8]
  400511:       48 89 c6                mov    rsi,rax
  400514:       bf c1 05 40 00          mov    edi,0x4005c1
  400519:       b8 00 00 00 00          mov    eax,0x0
  40051e:       e8 cd fe ff ff          call   4003f0 <printf@plt>
  400523:       b8 00 00 00 00          mov    eax,0x0
  400528:       c9                      leave
  400529:       c3                      ret
  40052a:       66 0f 1f 44 00 00       nop    WORD PTR [rax+rax*1+0x0]
.
.
-------------------------------------------------------------------------------
```

and the read-only data section.

```
-------------------------------------------------------------------------------
4005b0:  01 00 02 00 48 65 6C 6C 6F 20 57 6F 72 6C 64 21
4005c0:  00 6D 61 69 6E 3A 20 25 73 0A 00 00
-------------------------------------------------------------------------------
```

We know that the program will definitely run properly if these sections are mapped to the addresses the linker has requested to. Will the program work if they are mapped at addresses other than 0x400400 and 0x4005b0 respectively? Say I map the segment containing .text and .rodata at 0x800000. This means, .text will be present at the virtual address 0x800400 and .rodata at 0x8005b0. Will the program still run properly?

Read through `main()`'s disassembly again. Look at the 4th instruction.

```
-------------------------------------------------------------------------------
400505:       48 c7 45 f8 b4 05 40 00   mov    QWORD PTR [rbp-0x8],0x4005b4
-------------------------------------------------------------------------------
```

It is clearly visible that 0x4005b4 is the address given to the string **Hello World!** given by the linker. Take a look at the instruction's bytes. The last 4 bytes(**b4-05-40-00**) are the little-endian representation of **0x004005b4**. That this virtual address is hardcoded in that instruction.

Now the two sections are mapped at 0x800400 and 0x8005b0 respectively. This means that the **Hello World!** string is actually present at the virtual address 0x8005b4 but the above instruction still uses the linker-given virtual address 0x4005b4. There is probably nothing mapped at 0x4005b4 and dereferencing it will crash the program.

This means mapping that segment at the new virtual address 0x800000 was a bad idea. The program won't run properly.

Now consider our shared library *libdummy.so*. We already discussed that a shared library should work irrespective of where it is mapped in the virtual address space. It has a function `lib_func` which prints a string literal. When the library is linked, the linker allots an virtual address to that string literal. `lib_func` would also contain an assembly instruction similar to the one above. That instruction generated also uses string literal's virtual address. This means that the string literal's virtual address is hardcoded into that instruction. With an instruction containing a hardcoded virtual address, will the shared library work irrespective of where it is mapped?

We did the same experiment with *prog*. In *prog*'s `main()`, a virtual address was hardcoded into an instruction and we mapped it at some other address. We proved that the program would crash. The exact same thing would happen with the library too.

Instructions with hardcoded addresses won't allow shared libraries to function properly if it is mapped elsewhere. But we saw that a shared library needs to work irrespective of where it is mapped. How can this problem be solved? How do we not hardcode addresses in instructions but make it work?

Let us consider *prog* and come up with a solution. First of all, lets list everything we know about *prog*.

1. The first loadable segment is mapped at 0x400000.
2. .text section is present at an offset of 0x400 bytes => It will be mapped at a virtual address 0x400400.
3. .rodata section is present at an offset of 0x5b0 bytes => It will be mapped at a virtual address 0x4005b0.
4. If the loadable segment is mapped at 0x100000, then .text will be present at 0x100400 and .rodata at 0x1005b0.
5. Observe the relative addresses of bytes in .text and .rodata with respect to the segment base. They never change. The string **Hello World!** in *prog* is always at an offset of 0x5b4 bytes from the segment base. The instruction we considered is always at an offset of 0x505 bytes from segment base. Whichever virtual address you map the segment, the relative positions of bytes with respect to the segment base always remains the same.

With the above details, can you think of a solution?

Consider the string literal and the instruction.

```
--------------------------------------------------------------------------------
String literal:
4005b4:  48 65 6C 6C 6F 20 57 6F 72 6C 64 21    ; Hello World!

Instruction:
400505:       48 c7 45 f8 b4 05 40 00   mov    QWORD PTR [rbp-0x8],0x4005b4
--------------------------------------------------------------------------------
```

It is currently like this. The address 0x4005b4 is used to refer to the string literal. But we need a mechanism to refer to that string without hardcoding it in the instruction.

The string literal is always at an offset of 0x5b4 bytes from the segment base. The instruction is always at an offset of 0x505 bytes. This means, the gap between the string literal and the instruction is always (0x5b4 - 0x505) = 0xaf bytes. At runtime, let the segment be loaded at the virtual address V_base. The instruction will be present at V_i = (V_base + 0x505). The string literal will be present at V_s = (V_i + 0xaf). The instruction now can easily refer the string just by adding an offset of 0xaf to its own virtual address. We need something like this.

```
--------------------------------------------------------------------------------
String literal:
V_base + 0x5b4:  48 65 6C 6C 6F 20 57 6F 72 6C 64 21    ; Hello World!
```

```
Instruction:
V_base + 0x505:         mov    QWORD PTR [rbp-0x8], my_own_virtual_address + 0xaf
-------------------------------------------------------------------------------
```

Our problem has been transfered from hardcoding the string literal's address to hardcoding instruction's address. At runtime, how can the instruction know its address? When this instruction is getting executed, what does the Instruction Pointer(rip) contain? It contains the next instruction's address. Assuming the above instruction's encoding is 8 bytes, address of the next instruction = (V_base + 0x505 + 8). This means, when this instruction is getting executed, rip = (V_i + 8) => V_i = (rip - 8). There you go! We now have the instruction's virtual address. Now, the instruction looks like this.

```
-------------------------------------------------------------------------------
String literal:
V_base + 0x5b4:  48 65 6C 6C 6F 20 57 6F 72 6C 64 21    ; Hello World!

Instruction:
V_base + 0x505:         mov    QWORD PTR [rbp-0x8], (rip - 8) + 0xaf
-------------------------------------------------------------------------------
```

The instruction is now totally independent of V_base. Now, the segment can be placed at any position in the virtual address space, this code will work irrespective of that. Such code is called **Position Independent Code(PIC)**. You place it at 0x400000, it will work. You place it at 0x800000 it will work.

When the compiler generates code for a shared library, it must position independent code like above. Code which relies on runtime information rather than link time. This way, a shared library will work irrespective of its position in the virtual address space.

We simply constructed some hypothetical instruction there which will make that instruction position independent. But it will work only if the processor ISA supports such an instruction and yes it does. The x64 ISA supports this type which allows you to refer something relative to the Instruction Pointer(rip). This is called RIP-relative addressing mode. You are calculating the address of something using **rip**.

Now, let us look at *libdummy.so*'s code. It should contain code like above. Generate its disassembly and inspect it.

```
-------------------------------------------------------------------------------
chapter3$ objdump -Mintel -D libdummy.so > dummy.obj
.
.
0000000000000665 <lib_func>:
 665:   55                      push   rbp
 666:   48 89 e5                mov    rbp,rsp
 669:   48 8d 3d 11 00 00 00    lea    rdi,[rip+0x11]        # 681 <_fini+0x9>
 670:   e8 eb fe ff ff          call   560 <puts@plt>
 675:   5d                      pop    rbp
 676:   c3                      ret
.
.
0000000000000681 <.rodata>:
 681:   49 6e 73 69 64 65 20 6c 69 62 5f 66 75 6e 63 28 29 00
-------------------------------------------------------------------------------
```

Look at the third instruction: `lea rdi,[rip+0x11]`. That 0x11 is the gap between the fourth instruction and the string literal. The instruction is essentially `lea rdi, [rip - 7 + 0x18]` where

(rip-7) is the third instruction's address and 0x18 is the gap between the third instruction and the string literal.

I want you to read through the disassembly thoroughly. You won't find a single reference to a hardcoded address in it. Everything will be relative to something else which is decided at runtime.

3.4 Position Independent Executable (PIE)

We saw that the linker had alloted virtual addresses to *prog*'s segments. The first loadable segment need to be mapped at virtual address 0x400000 and the second at 0x601000. It contained instructions which had virtual addresses hardcoded in it. I want you to go through *prog*'s disassembly thoroughly. The two loadable segments **must** be loaded at 0x400000 and 0x601000 for the program to work properly.

If you think about it, even executables can be made position independent. The compiler should generate position independent code. Executables which are position independent are called Position Independent Executables(PIE). But what is a the need to make it position independent? There are a couple of advantages. It aids in implementing a highly effective security technique called ASLR which we will discuss in one of the future chapters.

This is the reason why **gcc-4.8** was used so far. It generates position dependent executables. Explaining PIC and PIE right in the beginning would be hard and would not have made much sense, so I used an older version of the gcc compiler. Modern versions(6, 7, 8) all generate position independent executables by default. Let us check it out.

```
-------------------------------------------------------------------------------
chapter3$ gcc --version
gcc (Ubuntu 7.5.0-3ubuntu1~18.04) 7.5.0
Copyright (C) 2017 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.  There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
-------------------------------------------------------------------------------
```

The default compiler version in my system is 7.5.0. Let me compile *prog.c* using this.

```
-------------------------------------------------------------------------------
chapter3$ gcc prog.c -o prog.7.5.0
chapter3$ ./prog.7.5.0
main: Hello World!
-------------------------------------------------------------------------------
```

It will run fine. Let us use the **file** command and check *prog* and *prog.7.5.0*.

```
-------------------------------------------------------------------------------
chapter3$ file prog
prog: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), dynamically linked, interpreter /lib64/l,
chapter3$ file prog.7.5.0
prog.7.5.0: ELF 64-bit LSB shared object, x86-64, version 1 (SYSV), dynamically linked, interpreter /
-------------------------------------------------------------------------------
```

file recognizes *prog.7.5.0* as a shared object because it is position independent. But we know that it is an executable. Then what exactly is the difference between a position independent executable and a shared object? One **defining** difference is that an executable has an **entry point**. If you try running *libdummy.so*, it will segfault because it does not have an entry point. We have seen that both executables and shared libraries can be shared objects. When we refer to something as shared object,

it simply means it has position independent code. If it has a well-defined entry point, then it is an executable and if it doesn't, it is a shared library.

Let us try something interesting. Position Independent Executables(PIEs) have all the properties of a shared library. So can it be used as a shared library? Let us check it out.

Let us write a normal C program with a function in it like the following. I will be using my default compiler from now on.

```
-------------------------------------------------------------------------libdummy2.c
chapter3$ cat libdummy2.c
#include <stdio.h>

void
func()
{
        printf("Inside func()\n");
}

int
main()
{
        printf("Inside main!\n");
        func();
}
chapter3$ gcc libdummy2.c -o libdummy2.so
-------------------------------------------------------------------------------
```

It has a function `func()` which is called by `main()`. Note that *libdummy2.so* is a PIE. My question is can this `func()` now be called by some other program if it is linked against *libdummy2.so*? Let us write *prog2.c* which calls `func()`.

```
-------------------------------------------------------------------------prog.c
chapter3$ cat prog2.c
#include <stdio.h>

int
main()
{
        func();
        return 0;
}
-------------------------------------------------------------------------------
```

It is a very simple program. Let us compile it and link it against *libdummy2.so*.

```
-------------------------------------------------------------------------------
chapter3$ gcc prog2.c -o prog -L. -ldummy2
/tmp/ccjPom8V.o: In function `main':
prog2.c:(.text+0xa): undefined reference to `func'
collect2: error: ld returned 1 exit status
-------------------------------------------------------------------------------
```

Nope! The linker gave an **undefined reference** error. Even though *libdummy2.so* has the function definition in it. the linker could not find it. This is one more difference between a PIE and a shared library. Although both of them are shared objects, only functions present in a shared library can

actually be accessed by other programs. This implies that position independent code(PIC) is a necessary condition to become a shared object, it is not a sufficient condition. Then what is the sufficient condition?

libdummy.so has `lib_func` function and we have used it in an external program *prog*. *libdummy2.so* has `func` and we couldn't use it. Function definitions are present in both binaries but we could link *prog* against *libdummy.so* but the linker gave an undefined reference error when we tried to link *prog2* against *libdummy2.so*. This means, there should be something in *libdummy.so* which informs the linker that it has `lib_func` and other programs can use it. In first chapter's Init, we saw how shared libraries are built. Just having function definitions is not sufficient. The linker should know that these function definitions exist in the shared library. A shared library has a list of all the functions exposed to the world whereas a PIE does not have that list.

The Dynamic Symbol Table(**.dynsym** section) present in these objects have the details. Let us checkout *libdummy.so*'s (shared library's) .dynsym section.

```
-------------------------------------------------------------------------------
chapter3$ readelf --dyn-syms libdummy.so

Symbol table '.dynsym' contains 13 entries:
   Num:    Value          Size Type    Bind   Vis      Ndx Name
     0: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 0000000000000000     0 NOTYPE  WEAK   DEFAULT  UND _ITM_deregisterTMCloneTab
     2: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND puts@GLIBC_2.2.5 (2)
     3: 0000000000000000     0 NOTYPE  WEAK   DEFAULT  UND __gmon_start__
     4: 0000000000000000     0 NOTYPE  WEAK   DEFAULT  UND _Jv_RegisterClasses
     5: 0000000000000000     0 NOTYPE  WEAK   DEFAULT  UND _ITM_registerTMCloneTable
     6: 0000000000000000     0 FUNC    WEAK   DEFAULT  UND __cxa_finalize@GLIBC_2.2.5 (2)
     7: 0000000000201028     0 NOTYPE  GLOBAL DEFAULT   23 _edata
     8: 0000000000201030     0 NOTYPE  GLOBAL DEFAULT   24 _end
     9: 0000000000000665    18 FUNC    GLOBAL DEFAULT   12 lib_func
    10: 0000000000201028     0 NOTYPE  GLOBAL DEFAULT   24 __bss_start
    11: 0000000000000530     0 FUNC    GLOBAL DEFAULT    9 _init
    12: 0000000000000678     0 FUNC    GLOBAL DEFAULT   13 _fini
-------------------------------------------------------------------------------
```

The 10th entry is about `lib_func`.

Now take a look at *libdummy2.so*'s (PIE's) .dynsym.

```
-------------------------------------------------------------------------------
chapter3$ readelf --dyn-syms libdummy2.so

Symbol table '.dynsym' contains 7 entries:
   Num:    Value          Size Type    Bind   Vis      Ndx Name
     0: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 0000000000000000     0 NOTYPE  WEAK   DEFAULT  UND _ITM_deregisterTMCloneTab
     2: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND puts@GLIBC_2.2.5 (2)
     3: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND __libc_start_main@GLIBC_2.2.5 (2)
     4: 0000000000000000     0 NOTYPE  WEAK   DEFAULT  UND __gmon_start__
     5: 0000000000000000     0 NOTYPE  WEAK   DEFAULT  UND _ITM_registerTMCloneTable
     6: 0000000000000000     0 FUNC    WEAK   DEFAULT  UND __cxa_finalize@GLIBC_2.2.5 (2)
-------------------------------------------------------------------------------
```

There you go! This does not have any entry about `func()`.

Obvious question: Now how do we convert a position independent executable into a shared library? We have a conceptual answer to it. We neeed to expose the functions present in the PIE. How can that be done? Entries about these functions can be added to the **.dynsym** of the PIE. It can be done using the **-rdynamic** option. Take a look.

```
--------------------------------------------------------------------------------
chapter3$ gcc libdummy2.c -o libdummy2.so -rdynamic
--------------------------------------------------------------------------------
```

Now let us inspect the new *libdummy2.so*'s .dynsym section.

```
--------------------------------------------------------------------------------
chapter3$ readelf --dyn-syms libdummy2.so

Symbol table '.dynsym' contains 20 entries:
   Num:    Value          Size Type    Bind   Vis      Ndx Name
     0: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 0000000000000000     0 NOTYPE  WEAK   DEFAULT  UND _ITM_deregisterTMCloneTab
     2: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND puts@GLIBC_2.2.5 (2)
     3: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND __libc_start_main@GLIBC_2.2.5 (2)
     4: 0000000000000000     0 NOTYPE  WEAK   DEFAULT  UND __gmon_start__
     5: 0000000000000000     0 NOTYPE  WEAK   DEFAULT  UND _ITM_registerTMCloneTable
     6: 0000000000000000     0 FUNC    WEAK   DEFAULT  UND __cxa_finalize@GLIBC_2.2.5 (2)
     7: 0000000000201010     0 NOTYPE  GLOBAL DEFAULT   23 _edata
     8: 0000000000201000     0 NOTYPE  GLOBAL DEFAULT   23 __data_start
     9: 0000000000201018     0 NOTYPE  GLOBAL DEFAULT   24 _end
    10: 0000000000201000     0 NOTYPE  WEAK   DEFAULT   23 data_start
    11: 00000000000008e0     4 OBJECT  GLOBAL DEFAULT   16 _IO_stdin_used
    12: 0000000000000860   101 FUNC    GLOBAL DEFAULT   14 __libc_csu_init
    13: 0000000000000720    43 FUNC    GLOBAL DEFAULT   14 _start
    14: 0000000000201010     0 NOTYPE  GLOBAL DEFAULT   24 __bss_start
    15: 000000000000083d    33 FUNC    GLOBAL DEFAULT   14 main
    16: 00000000000006d8     0 FUNC    GLOBAL DEFAULT   11 _init
    17: 00000000000008d0     2 FUNC    GLOBAL DEFAULT   14 __libc_csu_fini
    18: 00000000000008d4     0 FUNC    GLOBAL DEFAULT   15 _fini
    19: 000000000000082a    19 FUNC    GLOBAL DEFAULT   14 func
--------------------------------------------------------------------------------
```

Great! Now the `func()` present in *libdummy2* is exposed to the real world for use. Let us try linking *prog2.c* against *libdummy2.so*.

```
--------------------------------------------------------------------------------
chapter3$ gcc prog2.c -o prog2 -L. -ldummy2
chapter3$ ./prog2
Inside func()
--------------------------------------------------------------------------------
```

There you go! *libdummy2.so* is now both an executable and a shared library. A PIE is also a shared library.

Try running standard shared libraries like libc.so, ld.so etc., You will find that a couple of them are PIEs which are also shared libraries.

From now onwards, I will be using the default compiler on my system (gcc 7.5.0) now that we know what PIC is.

If a binary is a shared object, it means that it is made up of Position Independent Code(PIC). A binary made up of PIC does not imply that it is a shared library or a Position Independent Executable(PIE). Difference between any executable and a shared library is that the shared library has PIC and its symbols are exposed to the real world whereas the executable may or may not have PIC and its symbols are not exposed by default. Most importantly, the executable has an entry-point. A PIE can be converted into a shared library by exposing its symbols. To avoid confusion, let us stick to 4 terms. PIE means it is just an executable. Shared Library means it is a shared library and not PIE. An Executable Shared Library means it is a PIE and a Shared Library. Shared object is any binary which contains Position Independent Code - it can be PIE or Shared Library.

3.5 Memory Layout of a process

In all the previous sections, we focused on specific parts of the entire virtual address space. In Init, we always considered that a program is made up of only 2 parts - text and data. In Section 3.3, we focused completely on shared libraries and in Section 3.4, we came back to executable's sections and saw how it can be made position independent. At this point, we still don't have an idea of how the entire virtual address space of a process looks like, what all does it contain. We know that it contains the program's text, data and shared libraries' text and data. In this section, we will be exploring all the elements present in a process's virtual address space.

What all should be present in a process's virtual address space for it to run properly?

Every program has 3 compulsory parts - text, data and read-only data. It can be dependent on multiple shared libraries where each of those shared library has text, data and read-only. Along with that, each of them has its own metadata. Generally programs would want to allocate memory at runtime. Some part of address space needs to be given to that. We saw in previous chapters that a program needs a stack to run properly. These are the basic elements present in the address space. Let us now consider a program and inspect its elements.

```
-------------------------------------------------------------------------prog3.c
chapter3$ cat prog3.c
#include <stdio.h>

int main()
{
        printf("Hello!\n");
        while(1);
}
chapter3$ gcc prog3.c -o prog3
chapter3$ ./prog3
Hello!

^C
chapter3$
-------------------------------------------------------------------------------
```

Its a hello program ending which enters an infinite loop after printing the string. The **/proc** directory has details of every process. Let us make use of the information present in that directory. This is what the **/proc** directory looks like.

```
-------------------------------------------------------------------------------
/proc$ ls
1      1300  142   1495  170    19721  2359   2442  2535  2711   33    404   6135  7126  988
```

```
10      1307  1422  1496  1771   1985   2380   2446  2540  277    34    405   6153  7165  992
1066    1308  1425  15    1774   2      2387   245   2543  278    347   406   6188  7832  997
11      1314  1427  1504  1783   20577  23915  2454  2548  28     35    407   6195  830   999
11413   1333  143   1505  1794   2063   2392   246   2549  28004  352   41    6202  833   acpi
1190    1357  1436  1506  18     21     23978  247   2552  28879  3538  411   6247  834   asound
12      136   144   1507  1808   21159  24     2472  2553  28906  36    42    6248  9     buddyinfo
1200    1365  1451  151   1809   21163  240    2481  2561  29     361   4255  6251  952   bus
1207    137   1452  1515  1871   22     24038  2486  2563  2902   368   43    6253  954   cgroups
1209    1370  1463  1516  189    2207   2404   2490  2611  2905   374   432   6354  955   cmdline
1212    1372  1471  1522  1916   2208   241    2496  2619  2926   38    438   6367  957   consoles
1225    1376  1472  1533  1948   2221   2417   2500  2630  29833  384   450   6740  961   cpuinfo
1247    1377  1475  154   1955   2229   2419   2509  2642  3      39    473   6756  966   crypto
1263    138   1479  155   19703  2231   242    2510  2653  30     396   5024  6757  973   devices
1268    1380  1483  156   19704  2238   2423   2511  2656  3078   398   5029  6758  976   diskstats
1275    139   1486  1584  19716  2241   2425   2515  2678  3120   4     5042  6773  977   dma
1283    14    1487  159   19717  23     243    2518  2685  31449  40    5123  6788  978   driver
1295    140   1488  16    19718  2349   2430   2522  27    31542  400   5318  6818  979   execdomains
1297    141   149   1623  19719  2352   2437   2530  2702  31757  401   539   6827  984   fb
13      1419  1492  17    19720  2357   244    2534  2709  328    402   5685  7120  985   filesystems
--------------------------------------------------------------------------------
```

Each of those numbers are ProcessIDs. Apart from that, the **/proc** directory also has some interesting details about the operating system. Let us run *prog3* and get into its directory. Note that a directory for a process will be present in **/proc** till that process is running. Once it is terminated, the directory goes away. The infinite loop makes sure the process doesn't terminate. Lets go!

```
--------------------------------------------------------------------------------
chapter3$ ./prog3
Hello!

--------------------------------------------------------------------------------
```

Leave it alone, open up a new terminal. Lets get its PID and go to its directory.

```
--------------------------------------------------------------------------------
chapter3$ ps -e | grep prog3
 7881 pts/4    00:00:44 prog3
/proc/7630$ ls
arch_status      environ    mountinfo      personality   statm
attr             exe        mounts         projid_map    status
autogroup        fd         mountstats     root          syscall
auxv             fdinfo     net            sched         task
cgroup           gid_map    ns             schedstat     timers
clear_refs       io         numa_maps      sessionid     timerslack_ns
cmdline          limits     oom_adj        setgroups     uid_map
comm             loginuid   oom_score      smaps         wchan
coredump_filter  map_files  oom_score_adj  smaps_rollup
cpuset           maps       pagemap        stack
cwd              mem        patch_state    stat
--------------------------------------------------------------------------------
```

A process has a lot of metadata. We are currently interested in how its virtual address space looks like, what all is present in it. The **maps** file has the necessary details.

```
--------------------------------------------------------------------------------
```

```
56460cb7e000-56460cb7f000 r-xp 00000000 08:05 10775454 XXXXX/chapter3/prog3
56460cd7e000-56460cd7f000 r--p 00000000 08:05 10775454 XXXXX/chapter3/prog3
56460cd7f000-56460cd80000 rw-p 00001000 08:05 10775454 XXXXX/chapter3/prog3
56460ea06000-56460ea27000 rw-p 00000000 00:00 0        [heap]
7f6ed70a6000-7f6ed728d000 r-xp 00000000 08:05 13112024 /lib/x86_64-linux-gnu/libc-2.27.so
7f6ed728d000-7f6ed748d000 ---p 001e7000 08:05 13112024 /lib/x86_64-linux-gnu/libc-2.27.so
7f6ed748d000-7f6ed7491000 r--p 001e7000 08:05 13112024 /lib/x86_64-linux-gnu/libc-2.27.so
7f6ed7491000-7f6ed7493000 rw-p 001eb000 08:05 13112024 /lib/x86_64-linux-gnu/libc-2.27.so
7f6ed7493000-7f6ed7497000 rw-p 00000000 00:00 0
7f6ed7497000-7f6ed74be000 r-xp 00000000 08:05 13111996 /lib/x86_64-linux-gnu/ld-2.27.so
7f6ed768f000-7f6ed7691000 rw-p 00000000 00:00 0
7f6ed76be000-7f6ed76bf000 r--p 00027000 08:05 13111996 /lib/x86_64-linux-gnu/ld-2.27.so
7f6ed76bf000-7f6ed76c0000 rw-p 00028000 08:05 13111996 /lib/x86_64-linux-gnu/ld-2.27.so
7f6ed76c0000-7f6ed76c1000 rw-p 00000000 00:00 0
7fffc5f19000-7fffc5f3b000 rw-p 00000000 00:00 0        [stack]
7fffc5f74000-7fffc5f77000 r--p 00000000 00:00 0        [vvar]
7fffc5f77000-7fffc5f78000 r-xp 00000000 00:00 0        [vdso]
ffffffffff600000-ffffffffff601000 --xp 00000000 00:00 0 [vsyscall]
--------------------------------------------------------------------------------
```

Let us talk about the first 3 entries. The first virtual address range is readable-executable. You can guess that it contains the text. It also contains other metadata useful for execution. Next is a read-only address range. It contains all read-only elements like string literals. Third is a read-write address range. It has the data and some metadata useful for execution. To understand what this metadata is, we will have to explore the ELF in detail. Look at the 5th column. It is the **inode-number** of **prog3**. That can be verified.

```
--------------------------------------------------------------------------------
chapter3$ ls -i prog3
10775454 prog3
--------------------------------------------------------------------------------
```

The **inode-number** is a unique number given to a file, that is how the filesystem internally refers to that file.

The 4th column mentions where that file is present. Every hard-drive partition is identified by a pair of numbers - **Major number : Minor number**. The **/dev** directory has all the hardware and software devices.

```
--------------------------------------------------------------------------------
/dev$ ls -l sda*
brw-rw---- 1 root disk 8, 0 May  5 22:57 sda
brw-rw---- 1 root disk 8, 1 May  5 22:57 sda1
brw-rw---- 1 root disk 8, 2 May  5 22:57 sda2
brw-rw---- 1 root disk 8, 3 May  5 22:57 sda3
brw-rw---- 1 root disk 8, 4 May  5 22:57 sda4
brw-rw---- 1 root disk 8, 5 May  5 22:57 sda5
--------------------------------------------------------------------------------
```

Harddrives(specifically SCSI devices) are identified by the major number **8**. This page documents all the major numbers. Each of the partitions are identified by the minor numbers.

The last column tells us what that address range is.

The 4th entry is something called **heap**. Any dynamically allocated memory(using malloc, calloc, realloc) comes from this address range. This heap has nothing to do with the heap data structure(binary

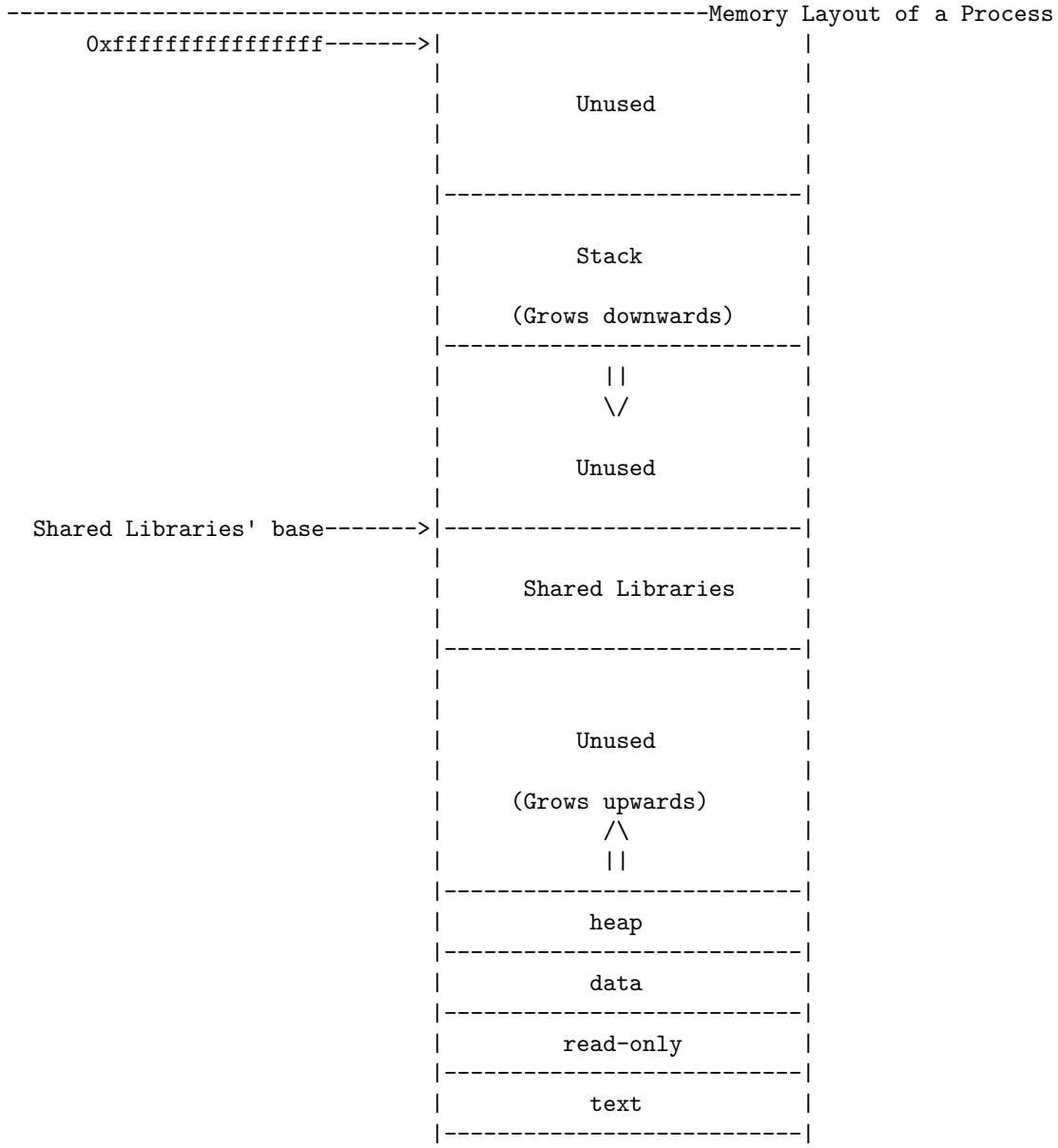
heap, binomial heap, fibonacci heap etc.,). This heap is a heap of memory just like a heap/pile of clothes.

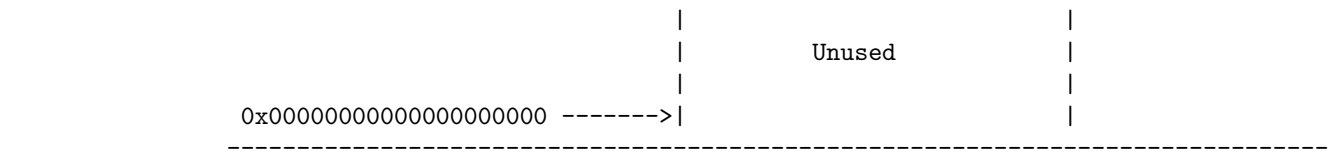
The first 4 ranges are close to each other. There is a huge gap between heap and the next entry. The next 4 entries belong to the C standard library(libc). There are 4 address ranges belonging to libc. This is where the shared libraries are mapped. Because this is a simple program, it depends only on the C library.

There are a couple of address ranges which are readable-writable but don't have any description. Let us discuss about these later.

Then comes the dynamic linker **ld**. Then comes the runtime **stack**. The rest of the ranges are **vvar**, **vdso** and **vsyscall**. Each of these need some explanation which we will discuss later.

We have seen the memory layout of a process. Let us generalize it. It looks like this.





I want you to go through the **/proc/PID/maps** file again and cross-check the above diagram.

You take any process, its memory layout will be similar to this.

3.6 The mmap() system call

In the previous section, we saw that various files(or its parts) being part of the virtual address space - it could be the executable itself(text, data and read-only) or it could be segments of other shared objects like the dynamic linker and C library. How exactly do these become part of the virtual address space? Does being part of the virtual address space mean memory is allocated? What exactly does mapping mean? Does it mean making something part of the physical memory or virtual address space? Let us explore these questions in this section.

Consider a file of size 10,000 bytes and you want to do some processing on it. Where do you start? There are various ways to do it. You can use the file-related functions and load the part of the file to be processed into a buffer and then process it. Another way is to include the file into the process's virtual address space. Once that is done, you can use virtual addresses to access any part of the file and then process it. Consider the second method. How would you include the file into the process's virtual address space? Can you think of a mechanism to make this work?

Let us start simple. However big the file is, let us first allocate actual physical memory required to load the entire file(here 10000 bytes). To access this memory, we also need to allocate a patch of virtual addresses - an address space of 10000 addresses. Then create mapping table entries to link the physical memory with virtual addresses. From here, it is straight-forward and should work. There is only one downside to this mechanism. We allocate physical memory for the entire file and then proceed. We have already seen cases when we do not have sufficient physical memory and have already solved that problem using **swapping**. So the only change is this. To start with, let us just allocate a patch of virtual addresses. Create entries in the mapping table. As of now, in the mapping table, all these new virtual pages **don't** have corresponding physical frames. When a virtual address is accessed, the corresponding piece of the file can be put into physical memory, the mapping table can be updated and then let the process use it. Let the first virtual address given be 0x12340000. This internally points to the first byte of the file. Say the process tries to access the first 4 bytes at virtual address 0x12340000. These 4 bytes are still not in memory. The first page of the file currently present in secondary storage is loaded into main memory, the mapping table entry is updated. This way, any piece of file is loaded into memory on a **demand-basis** which is a simple way to use memory efficiently. But one question. When the process tries to access 0x12340000, how will we(the operating system) know that the first byte of this file is being accessed? The operating system will have to store the file descriptor along with the virtual address patch - it is basically a mapping between the file and that virtual address patch.

This is exactly what the **mmap()** system call does. Here, I think it would be wrong to tell that mmap maps the file into memory because it is not mapping the entire file into memory. It breaks the file into pages and loads only the pages needed by the process. When it is not needed, the page-frame mechanism throws pieces of file into swap space. What I can tell is that the file is mapped entirely onto the virtual address space. For every byte in the file, there is already a virtual address to access it.

I hope you can appreciate this mechanism. A patch of virtual addresses may be allocated and is linked to a file. But it may not be linked to any physical memory behind. Only when you try accessing it, memory is given to only that part of the file being accessed.

3.7 Conclusion

With that, let us conclude this chapter.

We discussed quite a lot in this chapter. Started with simple memory models, broke the memory into pieces, then brought in the concept of virtual memory to make room to run big programs. Then went on to discuss what position independent code is, discussed the differences between shared objects, shared libraries and position independent executables. We then saw how the virtual address space of a generic process looks like and ended the chapter with exploring the very interesting mmap system call.

If it was just about understanding vulnerabilities like buffer overflow or format string, this chapter was not required. We could have done away with this chapter. But the memory subsystem is a very interesting one(the most interesting according to me). A while before writing this chapter, I read the papers on meltdown and spectre vulnerabilities. Reading those papers made me understand the intricacies and nuances of the memory subsystem. If you want to understand those vulnerabilities, you will require a solid understanding of the memory subsystem. That is why, this chapter was included.

This chapter contains a lot of theory and some practicals. I urge you to write small programs, dissect them, check if they are PIE or not, look at their **/proc/PID/maps** files and see how the virtual address space looks like, see if you can dig up something interesting.

3.8 Further Reading

1. Opal: A single address space Operating System: Complete design, documentation, papers, reports related to a single address space OS.
2. Before Memory was Virtual by Peter J. Denning
3. Complete Fair Scheduling(CFS): The linux process scheduling algorithm

Chapter 4: x64 Program Execution Internals

Summary
In this chapter, we explore how programs are executed. We will be analyzing various C constructs at assembly level. We explore how function calls work, how local variables are stored, how parameters are passed to a function and more in 64-bit programs.

4.1 Init

In the previous chapters, we discussed a lot of things revolving around a program - in what file format the program should be stored, how does the x86 assembly language look like, how does the program look like once it is loaded into memory for execution. We have not taken a deep look at the program itself. A C program we write can have a lot of things - variables, pointers, switch blocks, if-else-if code, loops, function calls, structures and more. What does each of these C constructs look like at assembly level? This chapter explores this question. We will go through each C construct and explore it at the assembly level.

We will be focusing on x64 architecture in this chapter. You are expected to know C. Also suggest you to read Chapter2 before proceeding with this chapter.

4.2 How does a function call work?

In C, all the code is divided into meaningful procedures(or functions). Consider the following program.

code1.c

```
1 void func()
2 {
3         return;
4 }
5
6
7 int main()
8 {
9         // Call it
10        func();
```

```
11
12          return 0;
13 }
```

It is a very simple program. The `main()` function calls the `func()` function. Once `func()` is done running, it returns the control back to `main()` and `main()` should continue from where it stopped. By looking at the C code, we could figure this out easily. Take a look at the following diagram.

```
           main (line 8)
            ||
            ||
            \/
           call func() (line 10)
            ||
            ----------->----------->-----------||
                                               ||
                                               \/
                                              func (line 2)
                                               ||
                                               || (func gets executed.)
                                               \/
            ||-----------------------------------
            main (line 11 - code present right after the call to func())
            ||
            ||
            \/
            main is done.
```

Note that after `func()` is done executing, control should be transfered to `main()` but not to the starting of `main()`. It needs to transfer control to line 11 - the line right after `func()` is called in `main()`. This needs to be implemented at assembly level. How can this be implemented? Take some time and come up with a mechanism. Idea is, when `func()` is done execution, it should transfer control to some line in `main()`. To do this, `func()` should know which line in `main()` it needs to transfer the control to. What can be done is, we store the line number in `main()` to which control has to be transfered at some fixed memory location and then **jmp** to `func()`. Once `func()` is done, it accesses that fixed memory location, gets the line number and **jmp**s to that line. We are basically storing the **return line number**.

Now, lets take a slightly complex example.

code2.c

```
 1 void func3()
 2 {
 3         return;
 4 }
 5
 6 void func2()
 7 {
 8         func3();
 9         return;
10 }
```

```
11
12 void func1()
13 {
14         func2();
15         return;
16 }
17
18
19 int main()
20 {
21         func1();
22         return 0;
23 }
```

The program is simple to understand. There is a series of function calls and returns.

1. `main()` calls `func1()`.
2. `func1()` calls `func2()`.
3. `func2()` calls `func3()`.
4. `func3()` executes and returns back to `func2()`.
5. `func2()` returns back to `func1()`.
6. `func1()` returns back to `main()`.

We need to come up with a mechanism to make this work. Try observing how the control flows. One method is to extend our previous mechanism. Have 3 fixed memory locations and store all the return line numbers. When `main()` calls `func1()`, it stores line number 22 in the first fixed location. When `func1()` calls `func2()`, it stores the line number 15 in the second fixed location. When `func2()` calls `func3()`, it stores the line number 9 in the third fixed location. How will a function know in which fixed location it needs to store the return line number? When the return-streak starts, how will `func3()` know it needs to return to the line stored in the third fixed location? You can see that this mechanism is not really flexible. When there are hundreds of calls happening (which is a very normal), the caller and callee are forced to remember the fixed locations - all this is a lot of runtime overhead.

We need to come up with a mechanism which is flexible and has low runtime overhead. Which data structure is suitable for the job? I want you to try all the data structures you know of - stack, queue, list etc., Which one works?

You can see that the stack data structure fits like magic in our case. Say the stack is empty when `main()` is called.

```
-------------------------------------------------------------------------return-stack
                    |                   |
                    |                   |
                    |                   |
                    |                   |
                    |                   |
                    |                   |
                    |                   |
                    |                   |
---------------------------------------------------------------------------------
```

Now, before `main()` **jmp**s to `func1()`, it pushes the return line number onto the stack - it pushes the number 22 and then **jmp**s to `func1()`. Now, the stack looks like this.

```
-------------------------------------------------------------------return-stack
```

```
                    |                 |
                    |                 |
                    |                 |
                    |                 |
                    |                 |
                    |                 |
                    |                 |
             TOP--->|        22       |
-------------------------------------------------------------------------
```

`func1()` pushes its return line number (line 15) before **jmp**ing to `func2()`. The stack looks like the following.

```
-----------------------------------------------------------------return-stack
                    |                 |
                    |                 |
                    |                 |
                    |                 |
                    |                 |
                    |                 |
             TOP--->|        15       |
                    |        22       |
-------------------------------------------------------------------------
```

Now, control is in `func2()`. Before it **jmp**s to `func3()`, it needs to push the return line number - line number 9 onto stack.

```
-----------------------------------------------------------------return-stack
                    |                 |
                    |                 |
                    |                 |
                    |                 |
                    |                 |
             TOP--->|         9       |
                    |        15       |
                    |        22       |
-------------------------------------------------------------------------
```

Now, we are inside of `func3()`. Now, we need to return back to line number 9 which is part of `func2()`. All `func3()` has to do is to look at the top of the stack. It reads it, pops it off of the stack and **jmp**s to that return line. So, control is transfered to line number 9 - which is part of `func2()`. The return-stack looks like this now.

```
-----------------------------------------------------------------return-stack
                    |                 |
                    |                 |
                    |                 |
                    |                 |
                    |                 |
                    |                 |
             TOP--->|        15       |
                    |        22       |
-------------------------------------------------------------------------
```

Once `func2()` is done executing, it looks at the top of stack, reads it, pops it and **jmp**s to that

return line. Control is transfered to line number 15 which is part of `func1()`. The following is the return-stack.

```
-------------------------------------------------------------------------return-stack
                     |                   |
                     |                   |
                     |                   |
                     |                   |
                     |                   |
                     |                   |
                     |                   |
                     |                   |
             TOP---->|        22         |
-------------------------------------------------------------------------------------
```

Now, we are in `func1()`. It looks at the top of stack and jumps back to `main()`'s line number 22. The stack is back being empty.

```
-------------------------------------------------------------------------return-stack
                     |                   |
                     |                   |
                     |                   |
                     |                   |
                     |                   |
                     |                   |
                     |                   |
                     |                   |
-------------------------------------------------------------------------------------
```

That worked fine. Having a stack makes control flow a lot easier. I am assuming this is why a **runtime stack** is present for every process.

Let us step into a more realistic scenario. In real, we don't deal with line numbers. We deal with addresses and assembly instructions. How will this mechanism look like at the assembly level? It is similar. Instead of pushing the next line number, you push the **address of the next instruction** and later **jmp** to that callee function. Once the callee is done executing, it looks at the top of the stack, pops it and **jmp**s to the instruction present at that address. The **address of the next instruction** is the **return address**. But how exactly do we get the address of the next instruction? We get it from the **rip** register. The Instruction Pointer register always points to the next instruction.

To make things easier, the x64 architecture has two instructions - **call** and **ret**. The **call** instruction does two things. It **push**es the address of the next instruction(the return address) onto the runtime stack and **jmp**s to the callee function. The **call** instruction is a combination of two other instructions.

```
-------------------------------------------------------------------------------call
push rip (which always points to the next instruction)
jmp func
-------------------------------------------------------------------------------------
```

Once the callee is done executing, it looks at the stack. It needs to **pop** that address and then **jmp** to that address. The **ret** instruction does exactly that. It is a combination of two other instructions.

```
-------------------------------------------------------------------------------ret
pop rip
jmp rip
-------------------------------------------------------------------------------------
```

We will see **call** and **ret** in action later in the chapter when we do some practicals.

That is how the control flow part of function calls work. A runtime stack is used to push and pop return addresses. We still need to see how arguments are passed to the callee function, how a callee function returns a value and more. We will explore those in later parts of the chapter.

4.3 What is a StackFrame?

Functions use different types of variables.

1. All the global, extern and static variables lie in the data segment. These variables are active and present throughout the lifetime of the program. We don't have to manage their lifetimes. They come to life when the program comes to life and dies when the program dies.

2. Then comes dynamically allocated memory - the memory allocated on heap by malloc, calloc or realloc. Lifetime of this memory is completely in the hands of the programmer. We free it when we no longer need it.

3. Another type of variable is the **local variable**. They are called local for a reason. They are local to a function. They have meaning only when the function they belong to is executing. Once that function is done running, the memory allocated to these local variables need to be deallocated. They are also called function's **private variables**. Its lifetime is the lifetime of the function. With such a characteristic, which part of the virtual address space should these variables belong to? Should they be allocated in heap or data or where? How do we control the lifetimes of these variables?

In this section, we will be exploring all the above questions related to local variables.

Let us take a program to explore.

code3.c

```
 1 void func()
 2 {
 3         short int x = 5;
 4         int y = 10;
 5         long int z = 20;
 6
 7         return;
 8 }
 9
10 int main()
11 {
12         func();
13         return 0;
14 }
```

This is a simple program. `main()` calls a function `func()` which defines 3 **local variables** x, y and z. Again, remember their characteristic - they come to life when `func()` starts its execution and dies when `func()` returns back its control to `main()`. Think like a compiler. What type of code should it emit to bring about this behavior?

Let us start with a simple idea. Let all the local variables be stored in heap. How will `func()`'s code look like? Take some time and think about it.

1. The compiler knows the total amount of memory needed for all the local variables in a function In our case, `func()` requires $(2 + 4 + 8) = 14$ bytes. Let us implement `func()` at assembly level. Let us start with allocating memory for local variables on heap.

```
-------------------------------------------------------------------------code3.asm
func:
       call malloc(14)      ; Allocation
-------------------------------------------------------------------------------
```

2. `malloc` returns a virtual address pointing to those 14 bytes of memory. At assembly level, let a function call return a value through the register **rax**. After `malloc` is called, on success **rax** has the address pointing to the 14 bytes. Let us load that address into register **r15**.

```
-------------------------------------------------------------------------code3.asm
func:
       call malloc(14)      ; Allocation
       mov r15, rax         ; r15 now points to the heap memory
-------------------------------------------------------------------------------
```

3. Now, first 2 bytes is the variable **x**. It needs to be initialized with the value 5. Lets do that.

```
-------------------------------------------------------------------------code3.asm
func:
       call malloc(14)      ; Allocation
       mov r15, rax         ; r15 now points to the heap memory
       mov word[r15], 5     ; short int x = 5;
-------------------------------------------------------------------------------
```

4. The next 4 bytes is the variable **y** which needs to be initialized with the value 10.

```
-------------------------------------------------------------------------code3.asm
func:
       call malloc(14)      ; Allocation
       mov r15, rax         ; r15 now points to the heap memory
       mov word[r15], 5     ; short int x = 5;
       mov dword[r15 + 2], 10      ; int y = 10;
-------------------------------------------------------------------------------
```

5. The next 8 bytes is the variable **z** which needs to be initialized with the value 20.

```
-------------------------------------------------------------------------code3.asm
func:
       call malloc(14)      ; Allocation
       mov r15, rax         ; r15 now points to the heap memory
       mov word[r15], 5     ; short int x = 5;
       mov dword[r15 + 2], 10      ; int y = 10;
       mov qword[r15 + 6], 20      ; long int z = 20;
-------------------------------------------------------------------------------
```

6. The local variables are now initialized in heap in the following manner.

```
-------------------------------------------------------------------------------
r15 -> | 00 | 05 | 00 | 00 | 00 | 0a | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 14 |

       |----x----|---------y---------|-------------------z------------------|
-------------------------------------------------------------------------------
```

7. This function is a simple one. It does not have any other code. It has code only to initialize the variables. Once they are initialized, they can be used without any problem.
8. Once the function is over, before it returns we need to deallocate it. We can use `free` to free it up.

```
-------------------------------------------------------------------------code3.asm
func:
       call malloc(14)       ; Allocation
       mov r15, rax          ; r15 now points to the heap memory
       mov word[r15], 5      ; short int x = 5;
       mov dword[r15 + 2], 10       ; int y = 10;
       mov qword[r15 + 6], 20       ; long int z = 20;
       ; Body

       free(r15)             ; Deallocation
-------------------------------------------------------------------------------
```

9. We are done cleaning up, lets return.

```
-------------------------------------------------------------------------code3.asm
func:
       call malloc(14)       ; Allocation
       mov r15, rax          ; r15 now points to the heap memory
       mov word[r15], 5      ; short int x = 5;
       mov dword[r15 + 2], 10       ; int y = 10;
       mov qword[r15 + 6], 20       ; long int z = 20;
       ; Body

       free(r15)             ; Deallocation
       ret                   ; Return back to the caller
-------------------------------------------------------------------------------
```

Note that the programmer does not have to explicitly call `malloc` and `free`. That is code added automatically by the compiler. If this needs to be implemented, the compiler needs to emit code which also handles the case when `malloc` fails.

Now, let us take another approach. Let us see if the runtime stack can be used to store the local variables.

So far, we have used the runtime stack only for storing return addresses. What can we do to to accomodate the local variables?

Every architecture offers a special register - the **Stack Pointer** which always points to the top of the stack. Assume that we are in a state where `main()` just **call**ed `func()` and we are inside `func()` now. The stack looks like this.

```
-------------------------------------------------------------------------return-stack
                    |  |  |  |  |  |  |  |  |
                    |  |  |  |  |  |  |  |  |
                    |  |  |  |  |  |  |  |  |
                    |  |  |  |  |  |  |  |  |
                    |  |  |  |  |  |  |  |  |
                    |  |  |  |  |  |  |  |  |
                    |  |  |  |  |  |  |  |  |
              SP--->|    Return Address     |
```

```
--------------------------------------------------------------------------------
```

We know that the stack has a width. Assume the width is 8 bytes. Any changes we do to the stack pointer, it should adhere to the 8-byte boundary. Even though we need only 14 bytes, we will allocate 16 bytes. Let us start implementing `func()`.

```
--------------------------------------------------------------------------code3.asm
func:
       add sp, 16           ; Allocation
--------------------------------------------------------------------------------
```

In a normal stack data structure you implement or you see in any library, you can't really change the stack pointer directly. You can only push and pop objects - that will indirectly change the stack pointer. Even here, we could do the same. We can push 0 twice to allocate 16 bytes and to move the stack pointer up. But we can do simple arithmetic on the stack pointer in most of the architectures. So, we will use add/sub to adjust the stack pointer whenever needed.

The stack now looks like this.

```
--------------------------------------------------------------------------stack
                     |  |  |  |  |  |  |  |  |
                     |  |  |  |  |  |  |  |  |
                     |  |  |  |  |  |  |  |  |
                     |  |  |  |  |  |  |  |  |
                     |  |  |  |  |  |  |  |  |
               SP--->|  |  |  |  |  |  |  |  |
               SP-8->|  |  |  |  |  |  |  |  |
                     |    Return Address     |
--------------------------------------------------------------------------------
```

Now, let us initialize the 3 variables. The first free byte is pointed by **sp-8**. Let the 2 bytes pointed by **sp-8** be variable x. Let us initialize it to 5.

```
--------------------------------------------------------------------------code3.asm
func:
       add sp, 16           ; Allocation
       mov word[sp-8], 5    ; short int x = 5;
--------------------------------------------------------------------------------
```

The stack looks like this.

```
--------------------------------------------------------------------------stack
                     |  |  |  |  |  |  |  |  |
                     |  |  |  |  |  |  |  |  |
                     |  |  |  |  |  |  |  |  |
                     |  |  |  |  |  |  |  |  |
                     |  |  |  |  |  |  |  |  |
               SP--->|  |  |  |  |  |  |  |  |
               SP-8->|00|05|  |  |  |  |  |  |
                     |    Return Address     |
--------------------------------------------------------------------------------
```

The next four bytes is belong to **y**. Let us init it.

```
--------------------------------------------------------------------------code3.asm
func:
       add sp, 16           ; Allocation
```

```
       mov word[sp-8], 5    ; short int x = 5;
       mov dword[sp-6], 10  ; int y = 10;
--------------------------------------------------------------------------------
```

The stack looks like this.

```
------------------------------------------------------------------------------stack
                     |  |  |  |  |  |  |  |  |
                     |  |  |  |  |  |  |  |  |
                     |  |  |  |  |  |  |  |  |
                     |  |  |  |  |  |  |  |  |
                     |  |  |  |  |  |  |  |  |
               SP--->|  |  |  |  |  |  |  |  |
               SP-8->|00|05|00|00|00|0a|  |  |
                     |    Return Address     |
--------------------------------------------------------------------------------
```

Where do we place the variable z? It can be placed at the 8 bytes pointed by **sp**.

```
------------------------------------------------------------------------code3.asm
func:
       add sp, 16           ; Allocation
       mov word[sp-8], 5    ; short int x = 5;
       mov dword[sp-6], 10  ; int y = 10;
       mov qword[sp], 20    ; long int z = 20;
--------------------------------------------------------------------------------
```

The stack looks like this now.

```
------------------------------------------------------------------------------stack
                     |  |  |  |  |  |  |  |  |
                     |  |  |  |  |  |  |  |  |
                     |  |  |  |  |  |  |  |  |
                     |  |  |  |  |  |  |  |  |
                     |  |  |  |  |  |  |  |  |
               SP--->|00|00|00|00|00|00|00|14|
               SP-8->|00|05|00|00|00|0a|  |  |
                     |    Return Address     |
--------------------------------------------------------------------------------
```

There are 2 unused bytes. They will always be present, irrespective of the way the 3 variables are arranged in those 16 bytes.

Now comes the body of the function.

```
------------------------------------------------------------------------code3.asm
func:
       add sp, 16           ; Allocation
       mov word[sp-8], 5    ; short int x = 5;
       mov dword[sp-6], 10  ; int y = 10;
       mov qword[sp], 20    ; long int z = 20;
       ; Body of the function
--------------------------------------------------------------------------------
```

Once the body is done, we need to deallocate the 16 bytes. How do we deallocate it? It is fairly simple. We simply shift the stack pointer back by 16 bytes. It is one assembly instruction.

```
-------------------------------------------------------------------------code3.asm
func:
       add sp, 16           ; Allocation
       mov word[sp-8], 5    ; short int x = 5;
       mov dword[sp-6], 10  ; int y = 10;
       mov qword[sp], 20    ; long int z = 20;
       ; Body of the function
       sub sp, 16           ; Deallocation
-------------------------------------------------------------------------------
```

The stack looks like this.

```
-------------------------------------------------------------------------stack
                     |  |  |  |  |  |  |  |  |
                     |  |  |  |  |  |  |  |  |
                     |  |  |  |  |  |  |  |  |
                     |  |  |  |  |  |  |  |  |
                     |  |  |  |  |  |  |  |  |
                     |00|00|00|00|00|00|00|14|
                     |00|05|00|00|00|0a|  |  |
               SP--->|    Return Address     |
-------------------------------------------------------------------------------
```

Note that the data is still there, but the memory does not belong to the function anymore. The stack pointer now points to the return address. We can simply **ret** back to the caller.

```
-------------------------------------------------------------------------code3.asm
func:
       add sp, 16           ; Allocation
       mov word[sp-8], 5    ; short int x = 5;
       mov dword[sp-6], 10  ; int y = 10;
       mov qword[sp], 20    ; long int z = 20;
       ; Body of the function
       sub sp, 16           ; Deallocation
       ret                  ; Return back to the caller
-------------------------------------------------------------------------------
```

This shows that even the runtime stack can be used to store the local variables. I want you to stop here and compare the stack and heap techniques to store the local variables.

Consider the following stack state.

```
-------------------------------------------------------------------------stack
                     |  |  |  |  |  |  |  |  |
                     |  |  |  |  |  |  |  |  |
                     |  |  |  |  |  |  |  |  |
                     |  |  |  |  |  |  |  |  |
                     |  |  |  |  |  |  |  |  |
               SP--->|00|00|00|00|00|00|00|14|
               SP-8->|00|05|00|00|00|0a|  |  |
                     |    Return Address     |
-------------------------------------------------------------------------------
```

Note that the 24 bytes starting from the byte pointed by **sp** till the byte pointed by **sp-23** - they are all private to the function `func()`. These 24 bytes form the **stack frame** of `func()`. A Stack frame is

simply the part of the runtime stack which belongs to a particular function till it is alive and running. Once the function is done running, that stack frame is descructed(or deallocated).

To understand this better, let us consider a slightly complex example.

code4.c

```
 1 void func2()
 2 {
 3         int a = 25;
 4         unsigned int b = 35;
 5
 6         return;
 7 }
 8
 9
10 void func1()
11 {
12         short int x = 5;
13         int y = 10;
14         long int z = 20;
15
16         return;
17 }
18
19 int main()
20 {
21         func1();
22         return 0;
23 }
```

Let us go through the above program - write assembly code and make sure we understand how stack is used to store local variables.

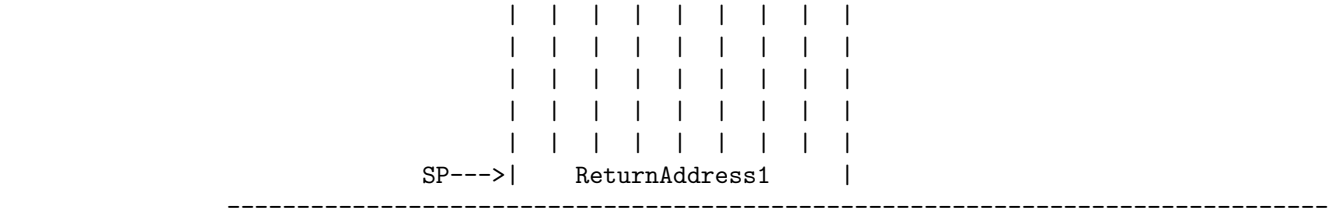
`main()`'s code would be pretty simple. It simply calls `func1()` and returns. The following is `main()`.

code4.asm

```
main:
       call func1           ; func1();
       ret                  ; return
```

When `main()` calls `func1()`, it pushes the address of the next instruction(`ret` here) and then **jmp**s to the label `func1()`. The stack would look like this once control is transfered to `func1()`.

stack



Now we are in `func1()`. It has 3 local variables x, y and z and the total memory required is (2 + 4 + 8) = 14 bytes. This means, we will have to allocate 16 bytes and lay the variables the way we did in

the previous example. The assembly code for `func1()` looks like this.

```
-------------------------------------------------------------------------code4.asm
func1:
       add sp, 16           ; Allocation
       mov word[sp-8], 5    ; short int x = 5;
       mov dword[sp-6], 10  ; int y = 10;
       mov qword[sp], 20    ; long int z = 20;

main:
       call func1           ; func1();
       ret                  ; return
-------------------------------------------------------------------------------
```

And the stack would look like this if those instructions were executed.

```
-------------------------------------------------------------------------------stack
                    |  |  |  |  |  |  |  |  |
                    |  |  |  |  |  |  |  |  |
                    |  |  |  |  |  |  |  |  |
              SP--->|00|00|00|00|00|00|00|14|    -
              SP-8->|00|05|00|00|00|0a|  |  |    | func1's 24-byte stack-frame
                    |    ReturnAddress1     |    -
-------------------------------------------------------------------------------
```

Look at the above stack state very carefully. `func1()`'s **stack-frame** is at the top of the stack. The size of that stack-frame is 24 bytes - 16 bytes for local variables and 8 bytes for ReturnAddress1.

Now is the time to call `func2()` from inside of `func1()`. Once that call is done, `func1()` returns.

```
-------------------------------------------------------------------------code4.asm
func1:
       add sp, 16           ; Allocation
       mov word[sp-8], 5    ; short int x = 5;
       mov dword[sp-6], 10  ; int y = 10;
       mov qword[sp], 20    ; long int z = 20;

       call func2           ; func2();


main:
       call func1           ; func1();
       ret                  ; return
-------------------------------------------------------------------------------
```

When the `call func2` instruction is executed, the address of the next instruction is pushed onto the stack and control is transfered to `func2()`. After the control is in `func2()`, the stack looks like the following.

```
-------------------------------------------------------------------------------stack
                    |  |  |  |  |  |  |  |  |
                    |  |  |  |  |  |  |  |  |
              SP--->|    ReturnAddress2     |    -> func2's new stack-frame
                    |00|00|00|00|00|00|00|14|    -
                    |00|05|00|00|00|0a|  |  |    | func1's 24-byte stack-frame
                    |    ReturnAddress1     |    -
```

We are now in `func2()`. It has 2 variables a and b of types int and unsigned int. Each of these variable needs 4 bytes => we need a total of 8 bytes. The following is the initialization code for `func2()`.

```
-------------------------------------------------------------------------code4.asm
func2:
       add sp, 8            ; Allocation
       mov dword[sp], 25    ; int a = 25;
       mov dword[sp+4], 35  ; unsigned int b = 35;

func1:
       add sp, 16           ; Allocation
       mov word[sp-8], 5    ; short int x = 5;
       mov dword[sp-6], 10  ; int y = 10;
       mov qword[sp], 20    ; long int z = 20;

       call func2           ; func2();


main:
       call func1           ; func1();
       ret                  ; return
-------------------------------------------------------------------------------
```

The stack looks like this now.

```
-------------------------------------------------------------------------stack
                     |  |  |  |  |  |  |  |  |
               SP--->|00|00|00|19|00|00|00|23|    -
                     |    ReturnAddress2     |    | func2's 16-byte stack-frame
                     |00|00|00|00|00|00|00|14|    -
                     |00|05|00|00|00|0a|  |  |    | func1's 24-byte stack-frame
                     |    ReturnAddress1     |    -
-------------------------------------------------------------------------------
```

Look at the above stack state. The 16 bytes pointed by the stack pointer make up `func2()`'s **stack-frame**. Because `func2()` is currently running, it's stack-frame is at the top of the stack. Right below it's stack-frame is `func1()` stack-frame.

In general, the runtime stack might have a lot of stack-frames. But the stack-frame which is at the top of the stack belongs to the function currently active and running.

`func2()` does not do more. It initializes variables and then returns. So, now we need to write the clean-up code which deallocates the stack memory. It is just a line - we need to reduce the stack pointer by 8 bytes. The code looks like this now.

```
-------------------------------------------------------------------------code4.asm
func2:
       add sp, 8            ; Allocation
       mov dword[sp], 25    ; int a = 25;
       mov dword[sp+4], 35  ; unsigned int b = 35;
       sub sp, 8            ; Deallocation

func1:
       add sp, 16           ; Allocation
```

```
        mov word[sp-8], 5     ; short int x = 5;
        mov dword[sp-6], 10   ; int y = 10;
        mov qword[sp], 20     ; long int z = 20;

        call func2            ; func2();


main:
        call func1
        ret
-------------------------------------------------------------------------------
```

Now that the clean-up is done, the stack looks like this.

```
-------------------------------------------------------------------------stack
                    |  |  |  |  |  |  |  |  |
                    |00|00|00|19|00|00|00|23|
               SP--->|    ReturnAddress2     |
                    |00|00|00|00|00|00|00|14|
                    |00|05|00|00|00|0a|  |  |
                    |    ReturnAddress1     |
-------------------------------------------------------------------------------
```

The data is still there, but it does not matter. Now, let us return using the **ret** instruction. It pops the ReturnAddress2 at the top of stack and then **jmp**s to it. The following is the code.

```
-------------------------------------------------------------------code4.asm
func2:
        add sp, 8             ; Allocation
        mov dword[sp], 25     ; int a = 25;
        mov dword[sp+4], 35   ; unsigned int b = 35;
        sub sp, 8             ; Deallocation
        ret                   ; Return back to the caller

func1:
        add sp, 16            ; Allocation
        mov word[sp-8], 5     ; short int x = 5;
        mov dword[sp-6], 10   ; int y = 10;
        mov qword[sp], 20     ; long int z = 20;

        call func2            ; func2();


main:
        call func1
        ret
-------------------------------------------------------------------------------
```

Once that is done, we are back in `func1()` and the stack looks like this.

```
-------------------------------------------------------------------------stack
                    |  |  |  |  |  |  |  |  |
                    |  |  |  |  |  |  |  |  |
                    |  |  |  |  |  |  |  |  |
               SP--->|00|00|00|00|00|00|00|14|
```

```
                    |00|05|00|00|00|0a|  |  |
                    |    ReturnAddress1     |
-------------------------------------------------------------------------------
```

Look how beautifully it fits. Compare the above stack state with the one before `func2()` is called. They are identical. Its as if `func2()` was never called. Now, we need to write the clean-up instructions for `func1()`. We need to deallocate 16 bytes. The following does it.

```
-------------------------------------------------------------------------code4.asm
func2:
       add sp, 8              ; Allocation
       mov dword[sp], 25      ; int a = 25;
       mov dword[sp+4], 35    ; unsigned int b = 35;
       sub sp, 8              ; Deallocation
       ret                    ; Return back to the caller

func1:
       add sp, 16             ; Allocation
       mov word[sp-8], 5      ; short int x = 5;
       mov dword[sp-6], 10    ; int y = 10;
       mov qword[sp], 20      ; long int z = 20;

       call func2             ; func2();

       sub sp, 16             ; Deallocation


main:
       call func1
       ret
-------------------------------------------------------------------------------
```

Once that deallocation instruction(`sub sp, 16`) is executed, the stack looks like the following.

```
-------------------------------------------------------------------------stack
                    |  |  |  |  |  |  |  |  |
                    |  |  |  |  |  |  |  |  |
                    |  |  |  |  |  |  |  |  |
                    |  |  |  |  |  |  |  |  |
                    |  |  |  |  |  |  |  |  |
               SP--->|    ReturnAddress1     |
-------------------------------------------------------------------------------
```

Finally, we **ret** back to `main()`. That completes `func1()`'s code.

```
-------------------------------------------------------------------------code4.asm
func2:
       add sp, 8              ; Allocation
       mov dword[sp], 25      ; int a = 25;
       mov dword[sp+4], 35    ; unsigned int b = 35;
       sub sp, 8              ; Deallocation
       ret                    ; Return back to the caller

func1:
       add sp, 16             ; Allocation
```

```
        mov word[sp-8], 5     ; short int x = 5;
        mov dword[sp-6], 10   ; int y = 10;
        mov qword[sp], 20     ; long int z = 20;

        call func2            ; func2();

        sub sp, 16            ; Deallocation
        ret                   ; Return back to the caller

main:
        call func1
        ret
-------------------------------------------------------------------------------
```

The stack is back to being empty - the way it was before `func1()` was called in `main()`.

There are a couple of observations.

1. Whatever function is running right now, it's stack-frame will always be at the top of the runtime stack.
2. Stack-Frame is constructed when a function is called by moving the stack pointer by suitable amount. It is desctructed when the function execution comes to an end by moving the stack pointer back to its original position.
3. Here, I have assumed that the ReturnAddress is part of the **callee**'s stack-frame. I think it is a matter of definition.

At this point, I hope you have an idea of how function calls work, how stack-frames are constructed and destructed for each function. Everything there was pretty generic. We did not deal into the nitty-gritty details of the x64 architecture. The next section will be about that - how exactly does function calls, stack-frames work in x64?

4.4 Function calls and stack-frames in x64

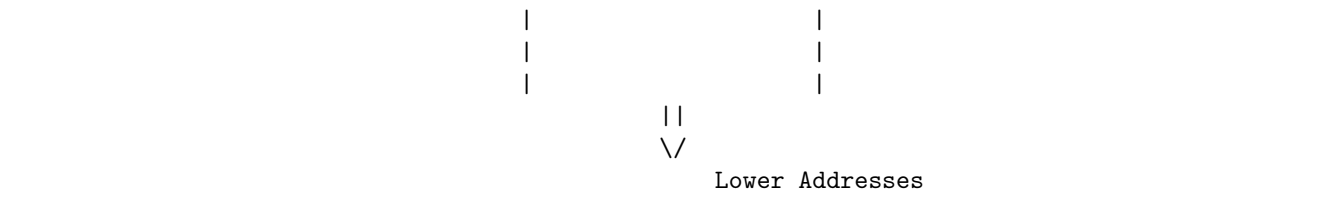
Function calls in x64 happens exactly the way we discussed in Section4.2. The runtime stack is used to make function calls work. In this section, we will mainly discuss about stack-frames in x64 architecture.

In the previous section, we discussed the construction and destruction of stack-frames as and when control is transfered into and out of a function respectively. There are certain things specific to x64 which we will discuss in this section.

4.4.1 Growth direction of the runtime stack

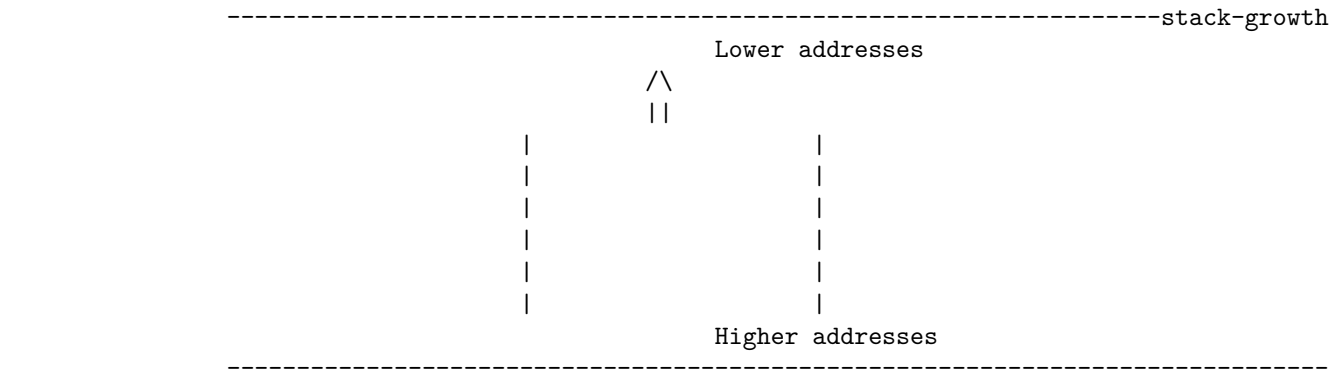
Whenever a new stack-frame is constructed, the runtime stack is growing. In the previous section, all the stack diagrams indicated that the stack grows upwards - like any other stack. If you take a closer look at the memory layout of a process presented in previous chapter's Section3.5, the runtime stack actually grows downwards - towards smaller virtual addresses. This can be visualized in two ways.

```
-------------------------------------------------------------------stack-growth
                                  Higher addresses
                        |                   |
                        |                   |
                        |                   |
```



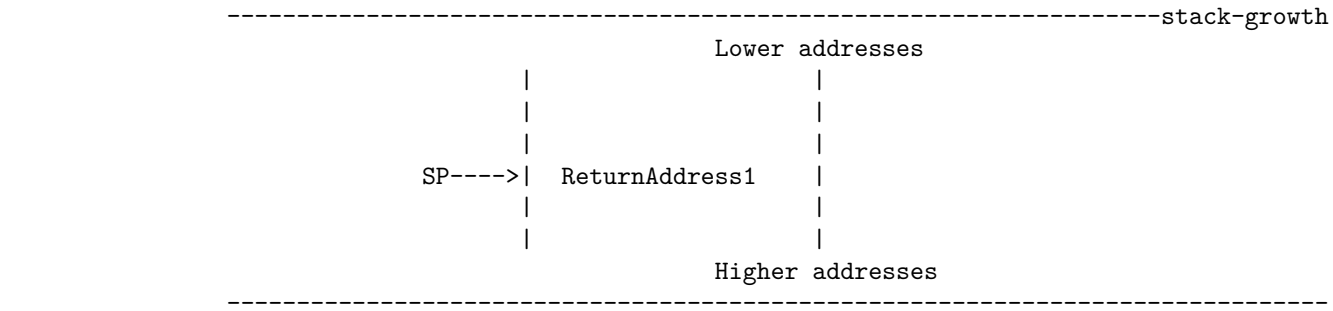
The above is the straight-forward way - it shows the runtime stack the way it is. It grows towards lower addresses.

Because we are accustomed to the stack growing upwards, we can simply show the stack upside-down like the following.

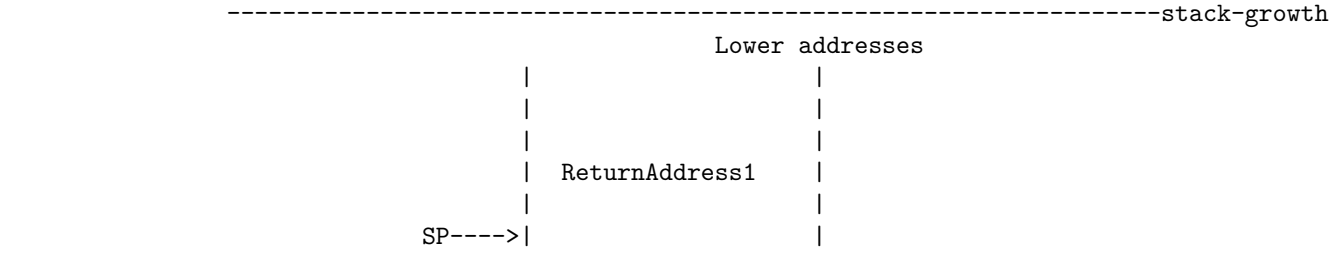


We will be using the second diagram throughout the chapter unless specified otherwise.

In the previous section, to allocate memory on the runtime stack, we did this - `add sp, 16`. This works only when the stack grows towards higher addresses. In our case, the stack grows towards lower addresses. So, to allocate memory on the runtime stack, we should not add an immediate value. We should instead subtract an immediate - `sub sp, 16` will do. Consider the following stack state.



Now, we need to allocate 16 bytes on the stack. Executing `add sp, 16` will change the stack to the following.



```
                                    Higher addresses
--------------------------------------------------------------------------------
```

This actually messed up the stack-state. When the stack is growing downwards(towards lower addresses), one should subtract an immediate value to allocate more memory - `sub sp, 16`. The following is the result.

```
--------------------------------------------------------------------stack-growth
                                    Lower addresses
                      |                   |
                SP---->|                   |
                      |                   |
                      |                   |
                      |  ReturnAddress1   |
                      |                   |
                      |                   |
                                    Higher addresses
--------------------------------------------------------------------------------
```

I can understand that this is a little confusing. You might have questions like why is the stack even growing downwards? Why didn't they design it to grow upwards and make everything simple and straight-forward? We will reason out these questions later in the chapter. For now, I want you to remember that the stack in Intel/AMD processors always grows downwards - towards lower addresses.

4.4.2 Data Storage and encoding

Consider the following function.

```
--------------------------------------------------------------------------------func2
func2:
       sub rsp, 8              ; Allocation
       mov dword[rsp], 25      ; int a = 25;
       mov dword[rsp+4], 35    ; unsigned int b = 35;

       ; No body

       add rsp, 8              ; Deallocation
       ret
--------------------------------------------------------------------------------
```

Say some function called `func2()` and the first instruction `sub rsp, 8` is already executed. The stack looks like the following.

```
--------------------------------------------------------------------------------stack
                                    Lower addresses
                      |  |  |  |  |  |  |  |  |
                      |  |  |  |  |  |  |  |  |
                      |  |  |  |  |  |  |  |  |
                rsp--->|  |  |  |  |  |  |  |  |
                rsp+8->|    ReturnAddress1     |
                      |  |  |  |  |  |  |  |  |
                                    Higher addresses
--------------------------------------------------------------------------------
```

Executing the instruction `mov dword[rsp], 25` will change the stack to the following.

```
--------------------------------------------------------------------------stack
                                  Lower addresses
                      |  |  |  |  |  |  |  |  |
                      |  |  |  |  |  |  |  |  |
                      |  |  |  |  |  |  |  |  |
                rsp--->|00|00|00|19|  |  |  |  |
                rsp+8->|    ReturnAddress1     |
                      |  |  |  |  |  |  |  |  |
                                  Higher addresses
-------------------------------------------------------------------------------
```

The hexadecimal equivalent of 25 is 0x00000019. This is the natural way to store a number. **rsp** points to the Most Significant byte of the number 0x00000019. What if I tell you that this is not the way in which the number is stored in real memory? It is stored as shown in the following stack diagram.

```
--------------------------------------------------------------------------stack
                                  Lower addresses
                      |  |  |  |  |  |  |  |  |
                      |  |  |  |  |  |  |  |  |
                      |  |  |  |  |  |  |  |  |
                rsp--->|19|00|00|00|  |  |  |  |
                rsp+8->|    ReturnAddress1     |
                      |  |  |  |  |  |  |  |  |
                                  Higher addresses
-------------------------------------------------------------------------------
```

The number is stored in a reverse manner. **rsp** points to the Least Significant byte of the number. This is once again a bit confusing. Why on earth did we decide to do everything in the reverse manner?

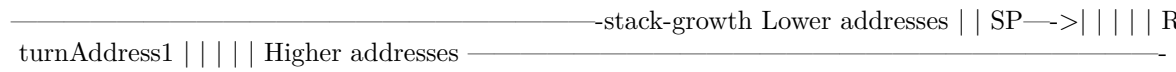
With that in mind, lets take *code4.c* and trace the assembly code and stack for it.

```
--------------------------------------------------------------------------code4.c
 1 void func2()
 2 {
 3         int a = 25;
 4         unsigned int b = 35;
 5
 6         return;
 7 }
 8
 9
10 void func1()
11 {
12         short int x = 5;
13         int y = 10;
14         long int z = 20;
15
16         return;
17 }
18
19 int main()
20 {
21         func1();
22         return 0;
```

```
23 }
--------------------------------------------------------------------------------
```

In x64, the register **rsp** is the stack pointer. From now onwards, we will be using the symbol **rsp** instead of just **sp**.

Consider the following empty runtime stack.



4.4.1 The Base Pointer

In the previous section, we discussed the entire concept of stack-frames with just the stack-pointer. The x64 architecture offers two registers: **rsp** or the Stack Pointer register and **rbp** or the Base Pointer register. **rsp** always points to the top of the stack(or the active stack-frame). **rbp** can be made to point to the base of the stack-frame. But what exactly is the base of a stack-frame?

Consider the following stack-frame.

```
--------------------------------------------------------------------------stack
                |  |  |  |  |  |  |  |  |
           SP--->|00|00|00|19|00|00|00|23|    -
                 |    ReturnAddress2     |    | func2's 16-byte stack-frame
                 |00|00|00|00|00|00|00|14|    -
                 |00|05|00|00|00|0a|  |  |    | func1's 24-byte stack-frame
                 |    ReturnAddress1     |    -
--------------------------------------------------------------------------------
```

Some resources

1. Something on 16-byte alignment - https://sourceforge.net/p/fbc/bugs/659/
2. https://raw.githubusercontent.com/wiki/hjl-tools/x86-psABI/x86-64-psABI-1.0.pdf - Latest AMD64 ABI copy
3. https://stackoverflow.com/questions/49391001/why-does-system-v-amd64-abi-mandate-a-16-byte-stack-alignment - good discussion on why stack is 16-byte aligned.
4. https://patchwork.kernel.org/patch/9507697/ - another discussion on why stack should be 16-byte aligned.
5.

The stack pointer is pointing to the top of the stack. You can observe that the top of the stack is nothing but top of `func2()`'s stack-frame. From this, we can tell that the **rsp** always points to the top of the **active stack-frame** - the stack-frame of the function currently under execution.