CHAPTER 1

C PROGRAM A SYSTEM VIEW

I assume that all of you would have got the setup of running a C program ready. Some clarifications on my previous instructions:

- $Turbo\ C$ is a 16 bit compiler made for $MS-DOS$. Some school student may say that it's easy to start with but I say it is difficult to program in $Turbo\ C$. Now, we are living in the world of 64-bit compilers and so we should learn to program for that.
- Why linux? Absolutely no need to use linux for learning C. But I would like to give an overview of the whole computer system (in this chapter itself), and it would be based on linux. As a computer science student if you don't know how to use linux, you are like a purse without money. So, start using at least now.
- Why AWS? It is not needed if you are having some linux distribution on your machine. But otherwise it is a very good idea to use AWS. It gives a shell on a linux system and you have full control of the OS. And it is free also (for normal usage). Moreover, AWS is used by many companies, and this would be a simple exercise to get used to it.

Now, if you are having a gcc or icc compiler on windows, its very much okay. But just one clarification: We won't be using the word "Turbo C" further in any of the discussions.

1.1 What a C program does?

First of all why are we using C language? We can start from there:

Digital Circuits We all know that the most important thing inside our CPU is the processor. It is made up of digital components like flip-flops and those who have studied digital circuits would understand that the digital circuits produce output from the input, when supplied with power and a clock. The speed at which the output is produced is determined by the clock speed of the CPU, because the clock determines the speed of transfer of bits between different units (and that's why a 3 GHz processor is faster than a 1 GHz processor). But this will be effective only if the processor has the input data available. (Most times there is delay for the CPU to get the input data and that's why the actual running time of a process on a 1GHz processor is not 3 $\times$, compared to the same process running on a 3GHz processor.)

Computer Organization The reason for the above problem is that data is given to processor from memory and the time to take a data from memory to processor is like 100 $\times$, the speed at which the processor works. So, we use many techniques like buffering, caching etc. to minimize this effect.

Okay, so now our aim is to give data to the CPU and it has these circuits called Functional Units to perform the specified tasks. Each functional unit does some job like addition, multiplication etc. and they give the result when provided with the input(s). But these input and output are in binary, as all these units are digital. That is, the input to CPU will be a series of 0's and 1's and the output will also be a series of 0's and 1's. So, what happens to the CPU when we give some string of 0's and 1's? Not all such strings will produce a valid output and this is entirely dependent on the processor. The processor manufacturer clearly specifies what's the meaning of each combination 0's and 1's and that's called the language of the processor which is entirely determined by its architecture.

Computer Architecture Most of our systems are $x86$ architecture and it has its own Instruction set [1] . So, lets take an example string of 0's and 1's in it:

$00000101 \;\; 00000000 \;\; 00000000 \;\; 00000000 \;\; 00000001$

This set of strings will add 1 to the content of EAX register (a register is a very fast memory inside the processor for holding data, and EAX is one among them in $x86$ architecture) which is a 32-bit register . The first byte is 00000101 which is given to the Instruction decode unit inside the processor which tells the CPU that it must add the next 32 bits to the contents of EAX register. Now, the next 32-bits will be given to the ADD unit which will add it to the contents of EAX register. In our example, the 32 bits represents just 1 and so the addition results in an increment of 1.

[Small question: Instead of ADD, if we use INC instruction, it results in better performance. How?]

So, this is how the CPU works- everything is in binary. We can get the meaning of each binary string from the Instruction manual [1]

To make things slightly easier they encode the bit-strings as HEX codes. So, the above bit string will become

$05 \;\; 00 \;\; 00 \;\; 00 \;\; 01$

Even then, it is difficult to write a program like this. Because human mind is not good at remembering numbers. So, then came Assembly language. Here, we use mnemonics to represent each operation. For example ADD is the mnemonic to do addition, SUB is the mnemonic to do subtraction etc. So, instead of the above binary string, now we can write

ADD EAX, 01

and the assembler will translate it into

```
05 00 00 00 01
```

Ahh! Much easier world for a programmer. But imagine writing an assembly program to print the factorial of n. How much time is needed to write it using these kinds of mnemonics, for each operation in the algorithm? Wouldn't it be better if we say the algorithm and then that is translated to binary string by $< someone >$? Yes, that's where C language comes. We can straight away represent most algorithms in C, and the C compiler will translate it into the binary string in the same way an assembler would do for assembly language.

So, lets write the C code for the above addition.

```
int a;
a = a + 1;
```

This will do the same job as

```
ADD EAX, 01
```

assuming a is the value we had in EAX.

Now, to start executing a program we need an entry point to the code. This is often named as $main$. The program starts executing from the first instruction inside $main$. So, to make our C code complete, we do the following

```
#include<stdio.h>
int main () {
 int a;
 a = a + 1;
 printf(`` a = %d\n", a);
}
```

For now lets assume that the $printf$ statement prints the value of a. But, C language has a restriction that before any variable is used, its type must be specified. So, before using $printf$, we must tell its type. Its type is written in a file called "$stdio.h$" so we can include that file (which will make all the contents of that file as part of our file) or we can just give the correct type of $printf$ as

```
int printf(const char *, ...);
```

Directly writing the declaration of $printf$ is not recommended and I just gave it to show the functionality of $\#include < stdio.h >$. Many people still think that code of printing is inside "$stdio.h$" and that's completely wrong. Code of $printf$ is part of C library which is available as $libc$ (there must be a file called $libc.so$ in linux and $printf$ code is inside that). The usage of $libc.so$ file is that the same code can be used by many programs (all linking to the same library code) so that reduces the main memory required to run the programs.

So, this usage of sharing $libc$ saves the total memory required when these 4 programs are concurrently executing. (If you want to see how many processes are executing at this moment on your system, just type *top* in a shell).

By default gcc will link to any function inside $libc$, if we call them in our program. But if we use any other library function, we have to explicitly tell gcc to link to that library. For example, to use $sin(x)$ in a code we have to link to math library ($libm$) as follows:

4 programs using the same code of *printf* inside *libc.so*, which is loaded in memory.

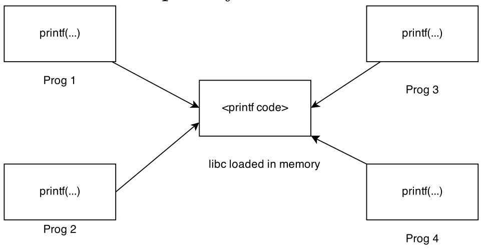

Figure 1.1 Programs sharing *printf* code

```
gcc prog.c −o prog −lm
(l is written instead of lib , so libm becomes lm,
libc becomes lc ...)
```

Once we compile the code *gcc* will be producing the output in a file called *prog* which is given with the $-o$ option. If we don't give any $-o$ option, output by default goes to a file called *a.out*. This will be in binary format and we cannot see it as text. This binary contains the bitstrings to be given to the processor as we discussed in the beginning. But some codes like that of *printf* is not inside this binary and is at a common location, which is called by our binary. Can we make copy the *printf* code and other library functions to be inside our binary? Yes, we can with the following command:

```
gcc prog.c −o progS −lm −static
(progS is just a different name)
```

Now, just see the size difference of the two binaries using *ls* command

```
ls −l prog
```

Now, to get the output by running the executable, we have to do

```
./prog
(./ just tells that prog is in the current directory)
```

Once the binary is produced by the compiler, before we get the output, there are many stages:

- Loading: Copying the content of the binary file to memory
- Linking: Fixing the calls in the binary which are to shared libraries as in the case of *printf*
- Process starts: Now the *OS* makes a process for our program (it gets a *pid*) and it'll be in *ready* state.
- When the turn comes, our process will get executed (it can be stopped in between to give other processes their turn)
- When our process is being executed, the binary strings inside it (which are instructions to the processor) are send one by one to the processor, which executes them
- Memory Management: For all the memory required by the process, the Memory Management Unit (*MMU*) ensures that the memory addresses used inside the object code

(which are virtual addresses) are properly mapped to physical memory locations on the RAM

So, even after the compilation (which of course includes its own phases) there are so many phases before we get the output of a C program.

We'll stop this chapter after mentioning about how we get segmentation fault in our programs.

1.1.1 Segmentation Fault

When a process is made by the OS, it allots some memory to it. This can be increased during its execution, upon request to the OS, and can go up to a limit set by the OS. So, when this process goes to execution state, it can only access the memory allotted to it. (This is done by giving a $pagetable$ to each process and all memory accesses are done through it). Whenever a process tries to access a memory which is not allotted to it, $segmentationfault$ occurs. Segmentation fault also occurs, if a process tries to write something to a read only memory area. For example, a program memory consists of many parts called segments and there are code segment, data segment and stack segment. Of these, the data segment is again divided into Read Only (RO) data segment and Read Write (RW) data segment. Among these segments only the RW data segment and stack segment are allowed to be modified by a process. (Some systems allow code segment also to be writable and can be used for writing self modifiable code) If a process tries to modify any other segment, then also segmentation fault happens. (Segmentation fault also happens due to some special hardware instructions, but we can ignore them as this won't happen for general programs compiled in a normal way.)

In this first chapter we have skimmed across compilers, memory management, process management, computer organization and computer architecture, which covers the basics of a Computer System. So, in order to run a very simple program itself we require all these. If you understand the basic functioning of these topics, that will be enough for an exam like $GATE$. From next chapter onwards, we'll go inside C.