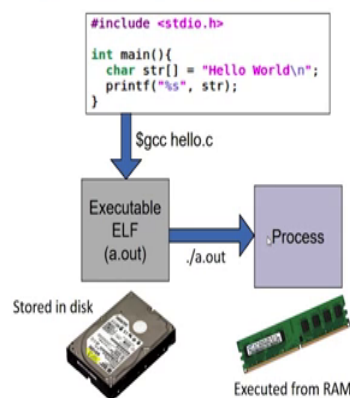


Secure Systems Engineering
Program Binaries
Prof. Chester Rebeiro
Indian Institute of Technology Madras
Mod_01 Lec_02

Hello and welcome to this lecture in the course for Secure System Engineering, during this course will be studying a lot of wonderibilities in C and C++ programs, now in order to appreciate these wonderibilities and how attackers have been able to utilise these wonderibilities to create exploits and malware, it is important for us to be able to understand C and C++ program binaries, so in this course we will be restricting ourselves to just see binaries, but a lot of this what we study here can be extended to C++ binaries as well.

(Refer Slide Time: 0:55)

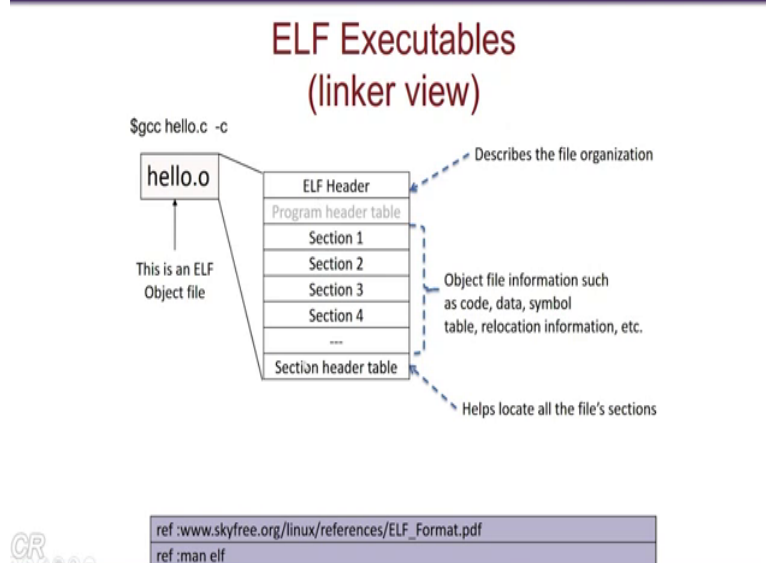
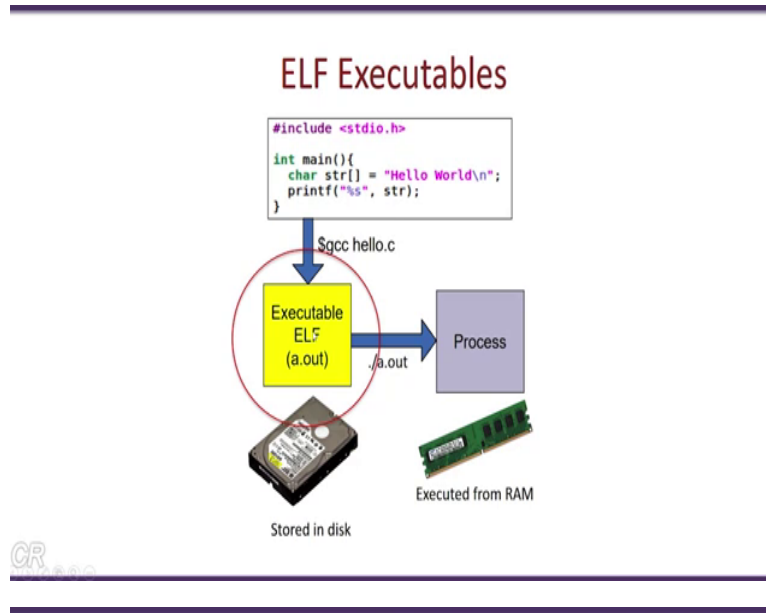
Executables and Processes



So let us start with a small C program, so as we know this program over here defines a string called which is initialised to hello world, and then invokes the print F function to print that particular string. Now, in order to execute this particular program. The first thing we do is enter these program in a text editor, save it as later say hello dot C, then the user compiler like this GCC hello dot C which would create an A dot out executable, so this executable is stored in the disk, so this executable has a particular format known as the ELF format or Executable Linker Format, so at a later time when you want to execute this particular program, you run the command dot/a dot out from your shell, when this happens, the operating system gets invoked it loads the executable files from the hard disk and creates a process out of it which is present on the RAM.

The process is then made to execute and you would get the string hello world printed on your terminal, now what we could do during this lecture is to understand details about, some details about what this Elf format is all about and we will also need to know some details about what this process contains.

(Refer Slide Time: 2:30)

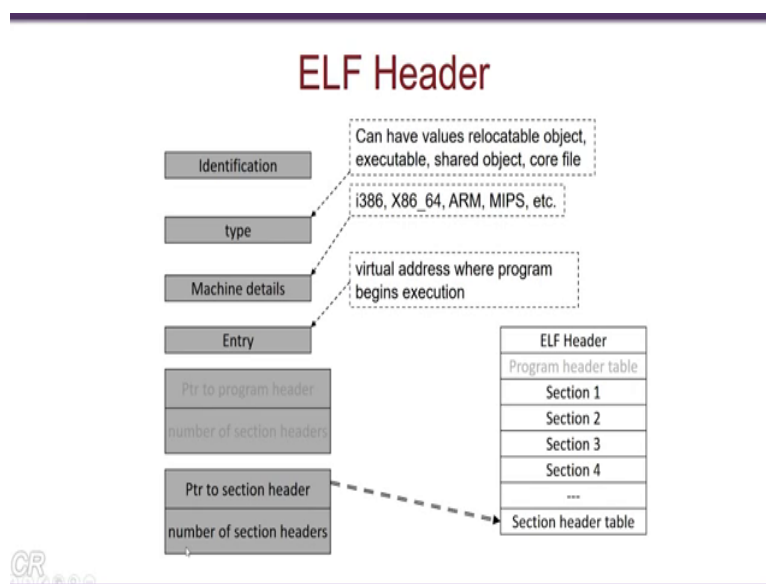


So we will start with the Elf format or the Executable Linker Format, Elf format describes a structure by which object files and executables need to be stored, so there are two views for the Elf format, one is the linker view and the other one is the executable view, the linker view is applicable for object files, while executables have the executable review, so let us start with the linker view. So, and object file that can for hello dot C can be created by this particular command, you do a GCC hello dot C, have a minus C option, it creates a hello dot O. Now

hello dot O is an Elf object file, it has a structure as shown over here, at the start you have an Elf header which describes the entire file organisation, then you have various sections which contain the code, the data, the symbol table, relocation information and so on and you also have a section header table.

So, in the section header table, essentially there is a structure which would help you locate the various sections present in the Elf object file. There is also a structure known as a program header table, but this is typically not present in the object file, so we look at more details about each of these headers, especially the Elf header and the section header.

(Refer Slide Time: 4:06)



Let us start with the Elf header, so the Elf header defines a structure with various parameters. So here we have actually said some of the few parameters present in the Elf header. It starts off with an identifier, so this identifier is a magic number which can be used to determine whether the file is an Elf file, so for example all Elf objects, Elf executables, libraries and so on would start off with this particular identifier. Then, there is entry in the Elf header which describes the type of this particular file, whether the file is an object, an executable a shared object or a core file, so this information is present in type.

On other entry is the Machine details, which processor was this particular file compiled for, it could have entries like i386, X86, 64, ARM, MIPS, and so on. What this means, for example if I have an entry, if the entry is let us say, arm, it means that this particular object file or executable was compiled for the arm processor.

Then you have an entry which is known as Entry, which describes the virtual address, where program has to begin execution. So this is more applicable for executables rather than object files or libraries.

So another important component in the Elf header is this pointer to the section header table, so as we said in the previous slide the section header table is present as part of the Elf image and it contains pointers to the various sections.

Also, there is an other entry called the number of section headers present in that particular file.

(Refer Slide Time: 5:50)


Hello World' s ELF Header

```
#include <stdio.h>

int main(){
    char str[] = "Hello World\n";
    printf("%s", str);
}
```

```
$ gcc hello.c -c
$ readelf -h hello.o
```

```
Chester@optiplex:~/tmp$ readelf -h hello.o
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:                      REL (Relocatable file)
  Machine:                   Advanced Micro Devices X86-64
  Version:                   0x1
  Entry point address:       0x0
  Start of program headers:  0 (bytes into file)
  Start of section headers:  368 (bytes into file)
  Flags:                      0x0
  Size of this header:       64 (bytes)
  Size of program headers:   0 (bytes)
  Number of program headers: 0
  Size of section headers:   64 (bytes)
  Number of section headers: 13
  Section header string table index: 10
```

 8

So let us see the Elf header for our program that we have written, so what we do is we say, while the program GCC hello dot C, give it the minus C option by which we get the executable hello dot O then we run this command readelf minus H, hello dot O, the output would look something like this, this output tells you the Elf header information for the file hello dot O. Note that it has the magic number which essentially is the Elf identification, then it has various other aspects, including the machine details, here it says that this object file was compiled for AMD X86, 64, that is. This object file can be only used by AMD an Intel machines which are configured for 64-bit.

Then you have the entry point address an importantly for us, you have this start of the section headers which is an offset of 368 bytes into the file, also the number of section headers that are present is in this particular case is 13, so let us look a little more detail into the section headers.

(Refer Slide Time: 7:12)

Section Headers

Contains information about the various sections

```
$ readelf -S hello.o
```

```
chester@optiplexi:~/work/SSE/sse/src/elfs$ readelf -S hello.o
There are 13 section headers, starting at offset 0x138:

Section Headers:
 [Nr] Name              Type              Addr              Off              Size              ES              Flg              Lk              Inf              Al
 [ 0]                      NULL              00000000          00000000         00000000         00              00              00              00              00
 [ 1] .text                PROGBITS          00000000          000034           00003C           00              AX              00              00              01
 [ 2] .rel.text           REL               00000000          000010           000010           00              11              14              00              00
 [ 3] .data                PROGBITS          00000000          000070           000000           00              WA              00              00              01
 [ 4] .bss                 NOBITS           00000000          000070           000000           00              WA              00              00              01
 [ 5] .rodata              PROGBITS          00000000          000070           000003           00              A              00              00              01
 [ 6] .comment             PROGBITS          00000000          000073           00002C           01              M5              00              00              01
 [ 7] .note.GNU-stack      PROGBITS          00000000          00009f           000000           00              00              00              00              00
 [ 8] .eh_frame            PROGBITS          00000000          0000a8           000039           00              A              00              00              04
 [ 9] .rel.eh_frame        REL               00000000          000010           000000           00              11              04              00              00
 [10] .shstrtab            STRTAB            00000000          000060           00005f           00              00              00              00              00
 [11] .symtab              SYMTAB            00000000          000340           00000a           10              12              04              00              00
 [12] .strtab              STRTAB            00000000          000370           000015           00              00              00              00              00
```

Key to Flags:
W (write), A (alloc), X (execute), M (merge), S (strings)
T (info), l (link order), G (group), T (TLS), E (exclude), x (unknown)
O (extra OS processing required) o (OS specific), p (processor specific)

Virtual address where the Section should be loaded ('0' all 0 because this is a o file)

Offset and size of the section

Type of the section
PROGBITS : information defined by program
SYMTAB : symbol table
NULL : inactive section
NOBITS : Section that occupies no bits
RELA : Relocation table

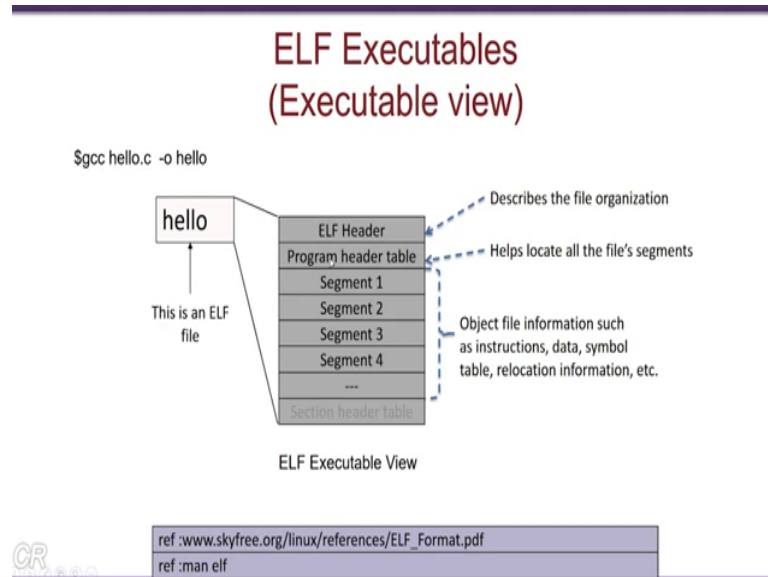
The section header table for this particular program can be obtained by running readelf with the minus S option as shown over here that is readelf minus S hello dot O, the output looks as follows, so these are the names, this particular column tells you the names of the various sections present in the hello dot O object file, then you would have the type of the section and as we see over here, there are various types of the section from starting from programming bits which essentially contains the program or the code that was written, you could have symbol table, you have no bits the allocation table and null and so on.

You have an other entry which is known as the address, so this corresponds to the virtual address for the various sections within the object file, so note that since this is a hello dot O, this is an object file, therefore each of this sections are relocatable, therefore the addresses present over here are all zero and then you have two columns, one for the offset and other for the size, the offset specifies, the offset within that particular Elf object where you could find this specific section, for example, the dot text section, these two columns specify the offset and the size for the various sections present in the object file. For example, the dot text section is present at an offset of 34 and has a size of 3C.

So 3C here is the hexadecimal notation, so in addition to all of this columns, there are other columns like this. For example, you have a flag column which specifies the various flags for this particular section, so for example A and X implies that this is a stands for Alloc, allocated while X stands for Executable, which means that the section contains executable code and can be executed.

In a similar way, for example, you have the data section, which is 3 over here and you notice that 3 has the flag WA, so W stands for Write, so note that the data segment is writable but cannot be executed.

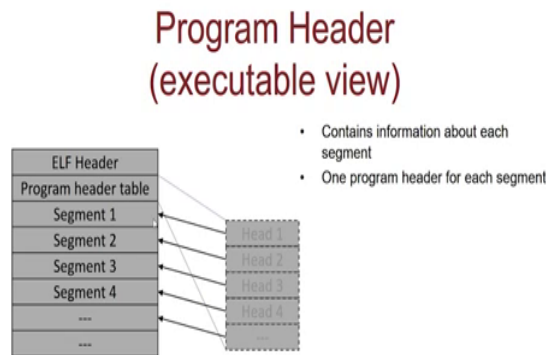
(Refer Slide Time: 9:40)



Now that we have seen the linker view of an Elf format, we would now look at the executable view, so the executable view is applicable for Elf executables, to generate an Elf executables for hello dot C, you could run this command like this or GCC hello dot C minus O hello, so this creates an Elf executable, a very similar to the A dot out, which is called Hello, so the hello executable has a format like this, so note that this format is very similar to the linker view, except that there are few changes.

First, you would notice that the sections in the linker view now called segments, further you will note that the program header table is now applicable now, while in the linker view this program header table was not, although it was present, it was not used, while in the executable view on the other hand, they section header table is not used. So as before the Elf header describes the file organisation of this executable and has a very same structure as what we have seen previously for the linker view. The Program header table helps to locate the various file segments, while the various segments present could be used for code, an instructions, data, symbol table, relocation information and so on.

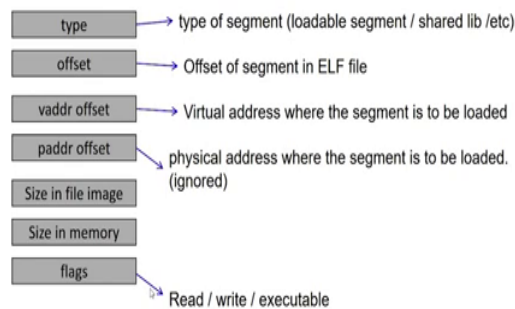
(Refer Slide Time: 11:16)



So we will look how the program header table looks like, so the program header table essentially contains various programs headers for the various segments, there would be one header corresponding to each segment.

(Refer Slide Time: 11:30)

Program Header Contents



So the contents of a program header would look something like this, so it is also define by a structure and has various entries and each program header is associated with one program segment, so the contents of the program header is as follows, there is a type entry which tells you the type of that particular segment, whether that segment is a loadable segment is a shared library and so on.

There is an entry called the offset which tells you the offset in the Elf file, where that segment is present.

There is a virtual address entry which tells you where that segment has to be loaded in the process during the execution time, at what location should that segment be loaded in the entire virtual space.

There is also physical address offset which specifies, which physical address that the segment should be loaded and most of the cases. This particular entry is ignored and we have the memory management unit of the processor which takes care of managing the physical address.

Beside this, you have the size of this particular segment in the file and also the size of the segment when it is loaded into memory and additionally you have flags for this particular segment, which specifies whether that segment can be read, written to or can be executed.

(Refer Slide Time: 12:59)

Program headers for Hello World

```
$ readelf -l hello
```

```
chester@optiplxri~/.work/SSR/ssa/src/elfs$ readelf -l hello
Elf file type is EXEC (Executable file)
Entry point 0x0040320
There are 9 program headers, starting at offset 52

Program Headers:
Type           Offset             VirtAddr           PhysAddr          FileSiz          MemSiz          Flg Align
PHDR           0x000034           0x00040034         0x00040034        0x00120          0x00120         R E 0x4
INTERP        0x000154           0x00040154         0x00040154        0x00013          0x00013         R 0x1
              [Requesting program interpreter: /lib/ld-linux.so.2]
LOAD          0x000000           0x00040000         0x00040000        0x00500          0x00500         R E 0x1000
LOAD          0x000f00           0x00040f00         0x00040f00        0x00110          0x0011c         RW 0x1000
DYNAMIC       0x000f14           0x00040f14         0x00040f14        0x00000          0x00000         RW 0x4
NOTE         0x000160           0x00040160         0x00040160        0x00044          0x00044         R 0x4
GNU_EH_FRAME 0x0004f4           0x000404f4         0x000404f4        0x0002c          0x0002c         R 0x4
GNU_STACK    0x000000           0x00000000         0x00000000        0x00000          0x00000         RWE 0x10
GNU_RELRO    0x000f00           0x00040f00         0x00040f00        0x000f0          0x000f0         R 0x1

Section to Segment mapping:
Segment Sections...
00
01  .interp
02  .interp .note.ABI-tag .note.gnu.build-id .gnu.hash .dynsym .dynstr
1  .rodata .eh_frame_hdr .eh_frame
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
```

Mapping between segments and sections

So let us take the hello dot C file again, create an executable hello and then read the program header information, so the program header information for the hello program could be write like this, so readelf with the minus L option and then the executable and you get an output like this way, so note that there are 9 program headers, these are the 1 to 9 headers, it contains an offset into the file where these segment, for each of the various segments, it has a virtual address, where each of these segments should be loaded when executed, this physical addresses also present, but it is not used, it also has the file size and the memory size and the flags present.

So for example you see that there is this section over here, which is read and executables, so this is a text section, so this is a, this particular section has code and you would be executing from this particular section, additionally, an important aspect over here is this particular part that creates a mapping from the section to the segment, so for example the segment 2 contains all of this sections.

Similarly, section 5, as this section dot note dot ABI tag, dot note gnu dot build and so on, so essentially every section present in the program gets mapped to a particular segment.

(Refer Slide Time: 14:32)

Contents of the Executable

```
$ objdump --disassemble-all hello > hello.lst
```

```
#include <stdio.h>

int main(){
    char str[] = "Hello World\n";
    printf("%s", str);
}
```

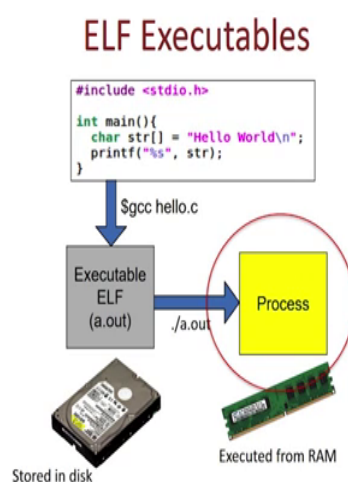
0004041d <main>:			
0004041d:	55	push	%ebp
0004041e:	89 e5	mov	%esp,%ebp
00040420:	83 e4 f0	and	\$0xffffffff,%esp
00040423:	83 ec 20	sub	\$0x20,%esp
00040426:	c7 44 24 13 48 65 6c	movl	\$0x6c6548,0x13(%esp)
0004042d:	6c		
0004042e:	c7 44 24 17 6f 20 57	movl	\$0x6f57206f,0x17(%esp)
00040435:	6f		
00040436:	c7 44 24 1b 72 6c 64	movl	\$0xa646c72,0x1b(%esp)
0004043d:	0a		
0004043e:	c6 44 24 1f 00	movb	\$0x0,0x1f(%esp)
00040443:	8d 44 24 13	lea	0x13(%esp),%eax
00040447:	89 44 24 04	mov	%eax,0x4(%esp)
0004044b:	c7 04 24 f0 84 04 08	movl	\$0x0404f0,(%esp)
00040452:	e8 99 fe ff ff	call	000402f0 <printf@plt>
00040457:	c9	leave	
00040458:	c3	ret	
00040459:	66 90	xchg	%ax,%ax
0004045b:	66 90	xchg	%ax,%ax
0004045d:	66 90	xchg	%ax,%ax
0004045f:	90	nop	

So now let us go a little more detail into the contents of the Elf executable, so what we do is we take the hello executable that we have created and we could actually create the disassemble for that particular executable, so by the running the command like objdump disassemble all hello and saving the output in this file called hello dot list, we would be able to get the complete disassembly for the Elf executable, so what here is a small snippet of that hello dot list file and it is only dealing with this particular main program over here.

So what we see over here is the assembly level instructions for this main program, so what we see at this particular, in this particular column is the virtual address, while in this columns are the corresponding instructions, so notice that over here, there is a call to printf at the LT, we will see what this actually means in a later class, but for now, we can understand this is the call to the printf, which is made over here, similarly, there are other instructions which are used in this main particular, main function.

Another important aspect for us is this column over here, which gives you a series of numbers like 59, 89, E5 and so on. What these numbers actually represent are the machine level codes corresponding to each of these functions, so for example this number 55 implies push percentage EBP, so in another words, when the processor sees an instruction encoded as 55, it would imply that it has to push this register EBP into this type, similarly, if it sees the sequence of numbers 89, EC and 20 present in the instruction it would imply that the instruction is subtract 20X from the stack pointer.

(Refer Slide Time: 16:43)

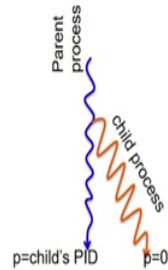


OR

Okay, so now we would move from the Elf executable to processes, so we would see what happens when you do, when you run `dot/a dot out` on your shell we would see how the process gets created and we would see how the Elf executable gets transferred from the disk into your RAM.

(Refer Slide Time: 17:03)

Creating a Process by Cloning (using fork system call)



```
int p;
p = fork();
if (p > 0){
    printf("Parent : child PID = %d", p);
    p = wait();
    printf("Parent : child %d exited\n", p);
} else{
    printf("In child process");
    execlp("hello", "", NULL);
    exit(0);
}
```

CR

18

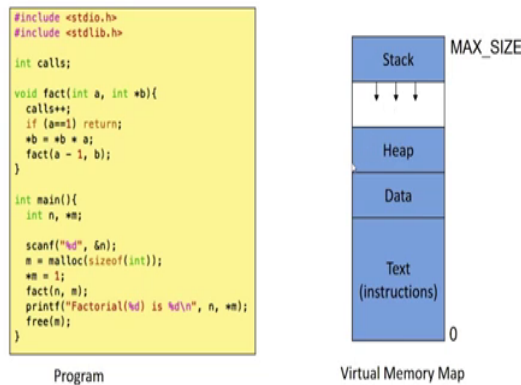
So when you give the command/a dog out and you shell let us see what happens internally, how your program gets executed. So when the shell receives your command, it would run a function which looks something like this, the function would invoke a fork, which is a system call and it invokes the operating system, the OS would then create a child process like this, so this predictor trail shows the your shell and when the fork system calls get created is, at child process gets created.

So now you have two processes, in the parent process, the value written by fork that is P has the child's PID and this is a value which is greater than zero, now in the child process P has a value equal to 0, therefore, the parent process would execute this part of the code while its child process would execute this part, which is present in green.

Now, in the child part of the code, there is a second system called that is get invoked which is the egsig system where the executable hello is specified, so this is the hello world program that you need to execute at the same time, the parent process invokes the weight system called so as to wait until this child process completes its execution.

(Refer Slide Time: 18:42)

Process Virtual Memory Map



CR

So let us look a little more in detail about the egsig called when it is executed by the operating system, so when the egsig system called is invoked in the operating system, the OS would create a new virtual address base for the new process. It would then load a segments from the executable file stored on the hard disk and copy them into the virtual address base, for example, all the segment comprising of the instruction and code are copied into this text segment, all the data, the global data and the static data are copied into this data segment.

Now we have taken here a small program, so this particular program computes the factorial of an integer in by this function fact which is essentially a recursive function, when this program is compiled and Elf executable gets created and when this program gets run the virtual address space for this program gets created by the operating system, the OS would load the code segments of this particular program into this text segment, it would load the global data such as calls into the data segment and whenever there is a malloc like this, which is dynamically allocated data, so this data gets allocated into the heap.

So finally we have this text segment, which comprises of the local variables, so in the function main, the local variables are N and M, so this local variables are present in the stack, so besides these, the stack also contains various aspects of function invocation and parameters processing from one function to another, so details about this virtual address base can be obtained from the proc directory present in your Linux operating systems.

(Refer Slide Time: 20:28)

Process Virtual Memory Map

```
chester@optiplex:~$ ps -ae | grep hello
6757 pts/25  00:00:00 hello
chester@optiplex:~$ sudo cat /proc/6757/maps
00049000-0004a000 r-xp 00000000 00:07 2491006 /home/chester/work/SSE/sse/src/elf/hello
00049000-0004a000 r-xp 00000000 00:07 2491006 /home/chester/work/SSE/sse/src/elf/hello
0004a000-0004b000 rwxp 00001000 00:07 2491006 /home/chester/work/SSE/sse/src/elf/hello
f759f000-f75a0000 rwxp 00000000 00:00 0
f75a0000-f774b000 r-xp 00000000 00:06 208150 /lib/i386-linux-gnu/libc-2.19.so
f774b000-f774c000 r-xp 001aa000 00:06 208150 /lib/i386-linux-gnu/libc-2.19.so
f774c000-f774e000 rwxp 0012c000 00:06 208150 /lib/i386-linux-gnu/libc-2.19.so
f774e000-f7751000 rwxp 00000000 00:00 0
f7773000-f7778000 rwxp 00000000 00:00 0
f7778000-f7779000 r-xp 00000000 00:00 0 [vdso]
f7779000-f7790000 r-xp 00000000 00:06 208150 /lib/i386-linux-gnu/ld-2.19.so
f7790000-f7799000 r-xp 0001f000 00:06 208150 /lib/i386-linux-gnu/ld-2.19.so
f7799000-f779a000 rwxp 00020000 00:06 208150 /lib/i386-linux-gnu/ld-2.19.so
f805000-f80a000 rwxp 00000000 00:00 0 [stack]
chester@optiplex:~$
```

Virtual address memory range

flags

Device details
(offset in file; device number; inode)

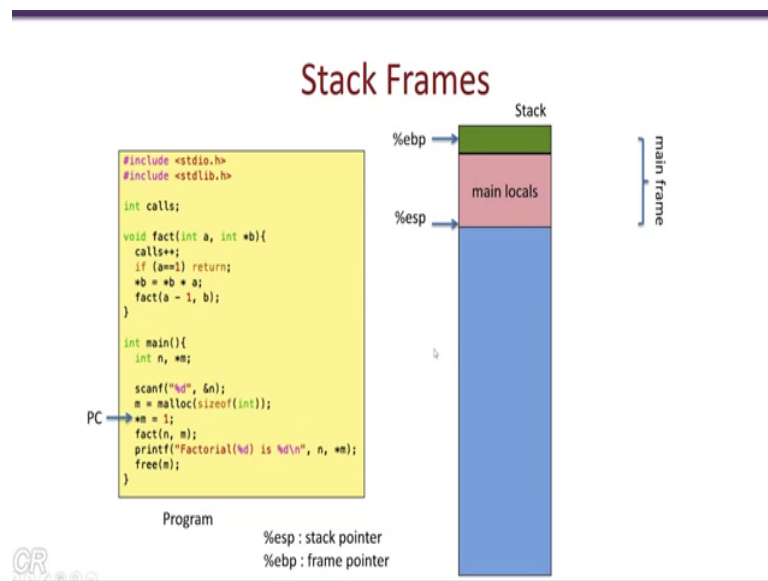
On a Linux system, we would be able to look at the virtual address base by the hello world program is executing, so this is done as follows, first we need to get the PID of the hello world program, so for that we could run PS minus AE and grep hello, so we get a line like this, where we see that the process hello is present and it has a PID 6757, now the file/proc 6757/maps contains the virtual address base for this PID 6757, which corresponds to our hello executable, so the virtual address map looks something like this way, so as you can see that there are various segments that are present, some which are present in the hello program itself, some which are present in the Lipsey library while the other are presenting in the loader library, also present is the stack and some internal segments like the VDSO.

Now this particular column present here specifies the various a virtual address ranges for the various segments, so for example we have in the hello executable, we have a segments which starts at 08049000 to 0804a000, the second column specifies the various flags for each segment, so for example this particular segment which is present in hello has the, is readable, writable and executable while they segment cannot be written to, it is only read and write flags are set.

Now these three columns specifies details about from where the segment was actually loaded from, so for example this columns specifies the offset in the Elf file from a particular segment was loaded, now this particular columns specifies the hard disk number or the device number, which is a major and the minor number for that particular disk, from where this particular library was loaded and this one, this last column over here specifies the inode number, which is essentially an identifier in the disk for this particular library.

So we have all of this information completely specifying the virtual address map for a particular process, so one thing to note is that for this text segment, we see that the offset in the file, device number and the inode is zero, so this is because the text segment is created dynamically at runtime and the Elf executable and the Elf object files do not have any motion about stack, now we will look a little more in detail about how the stack is managed in the program.

(Refer Slide Time: 23:47)



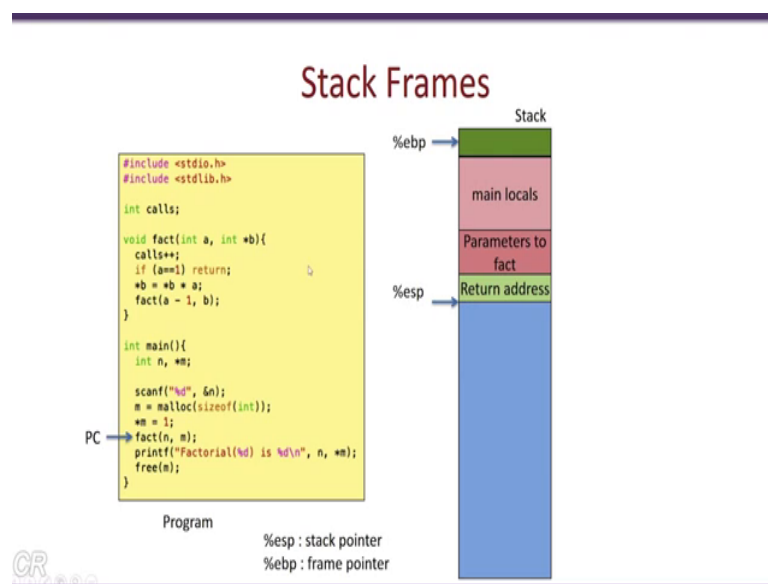
So now we will see how the stack of a program is managed during the execution of the program, so we will take as an example, this particular program, which comprises of two functions, a main function and fact function, objective of this particular program is to determine the factorial of this number N which is specified by the user and the way this factorial is computed is by the function fact which is a recursive function that gets invoked until a becomes one, so note that fact takes two parameters, one is int A and another one is a pointer and the other one is an int star B, to understand how the stack is manage we would have to know about two registers, one is the stack pointer register which is denoted as ESP and the other one is the frame pointer register which is denoted EBP, now we have represent this stack by this particular diagram, so we have a frame pointer, pointing to a location in the stack and we have a stack pointer, pointing lower down in the stack, every time we push something onto this stack, the stack pointer address reduces as we keep pushing into the stack.

While as we pop from the stack, the stack pointer address keeps incrementing again, so we further define something known as an active frame which comprises of the region between

the base pointer to the stack pointer, when the main function gets executed and let us assume that the program counter is pointing to this particular location, now when the main function is executing and the program counter is pointing to this particular instruction, then we have this thing as the active frames, so we called this as a main stack frame and it is the active frame because it is between the base pointer and the stack pointer, so this region is the active frame.

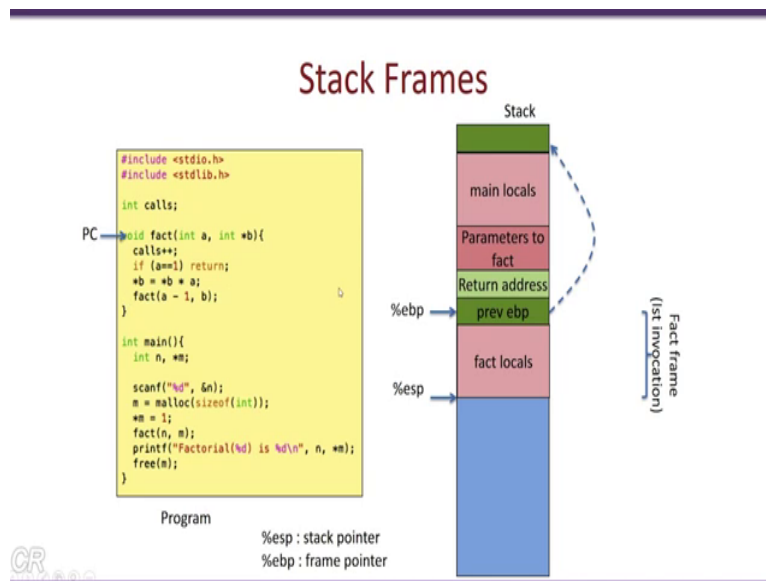
Now in this region, we have all the locals that are present in main to the present over here that is the locals N and M are mapped to regions to memory regions present in this area, now let us see what happens when the function fact gets invoked.

(Refer Slide Time: 26:15)



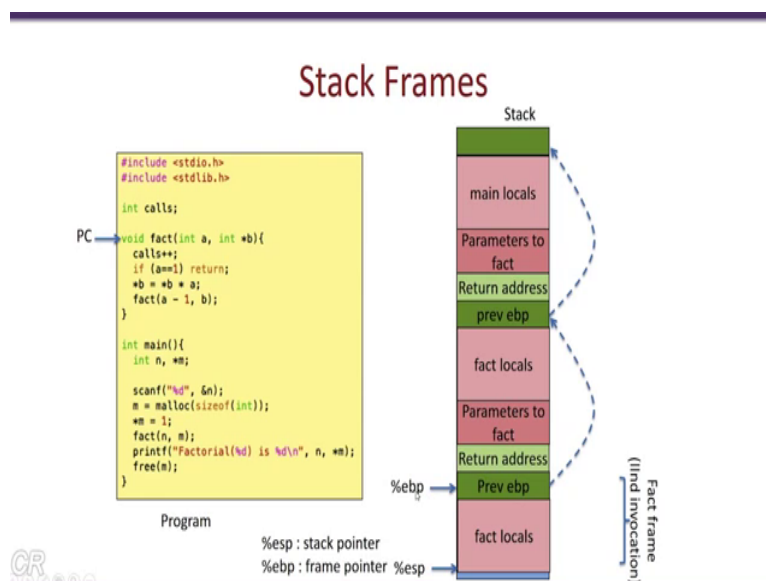
The first thing that main would do is to push the parameters N and M onto the stack and then it would call the fact function, the call instruction would automatically push the return address onto the stack, so the return address would be the instruction just following the fact function and it would indicate the instruction to be executed soon after fact returns, so therefore after the call to the fact function, you would have a frame, an active frame which looks like this, there are the main locals of the main function that is comprising of N and M, you would have a parameters to the fact function, which here again would be a copy of N and M, and then you would have the return address.

(Refer Slide Time: 27:06)

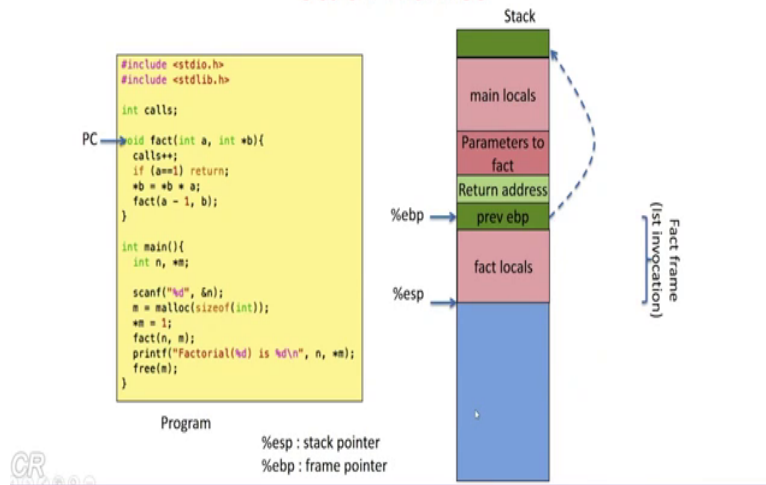


The next thing, what happens is that the fact function starts to execute, so the first thing that the factors is to copy the frame pointer which was previously pointing to this location, so this frame pointer gets copied onto the stack and then the frame pointer is moved to this location, so now what we have seen is that we have got a new frame and this is now the active frame and it is we call it as the stack frame for the fact function, the locals for the fact function are then present on the stack and its bit, these locals are present between the base pointer and the stack pointer, so as the fact function executes it will recursively invoke the fact function, so in a very similar way as we have seen before.

(Refer Slide Time: 27:59)



Stack Frames



When the `fact` function gets invoked again parameters to the `fact` return address and the previous `EBP` gets pushed onto the stack and similarly there is space present for this `fact` locals, so on the first return from the stack the previous base pointer which is pointing to the previous frame gets loaded into the base pointer and therefore we would get the new `fact` frame.

With this, we have given a small introduction to Elf loader and executable formats and then we have also seen how processes execute and load this executes Elf executables and create a virtual address basis and finally we have seen how the stack in the program operates, so one thing for you to think about is about the command line arguments, so as we know the `main` function actually takes a two parameters, one is an `argc` and an `argv`, so one thing that you could think about is how are these arguments actually passed from the command line that is from your shell to your program. Thank you.