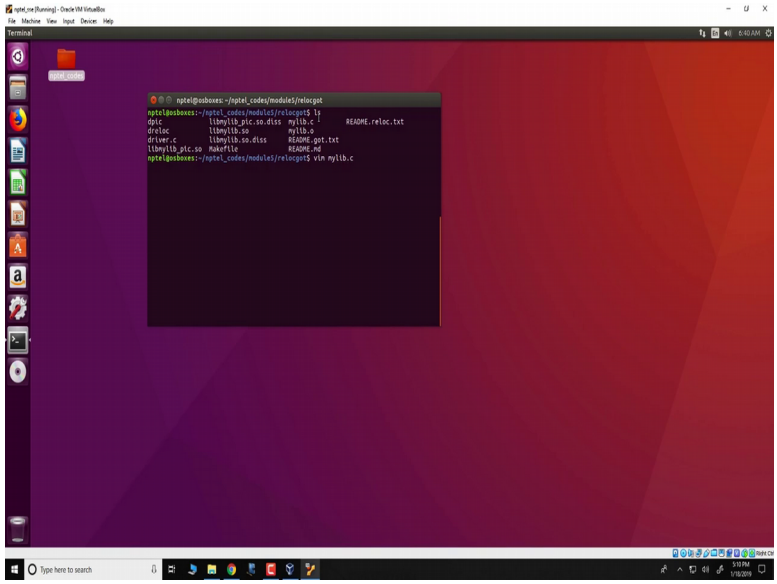
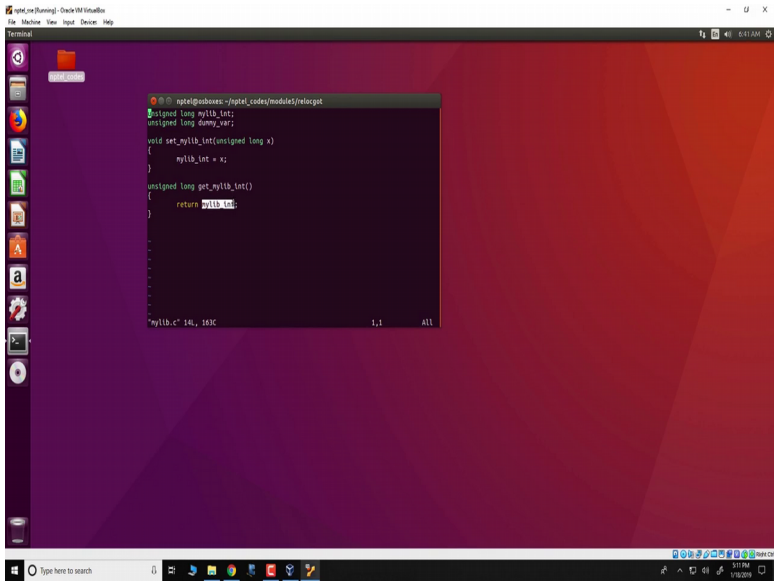
Information Security 5 Secure System Engineering
Prof. Chester Rebeiro
Indian Institute of Technology Madras
Demonstration of Load Time Relocation
Mod03_Lec18

Hello and welcome to this demonstration in the course for secure system engineering, in this particular course we will look at a load time relocatable techniques which is essentially one of the ways we could actually have address space layout randomization.

(Refer Slide Time: 0:34)

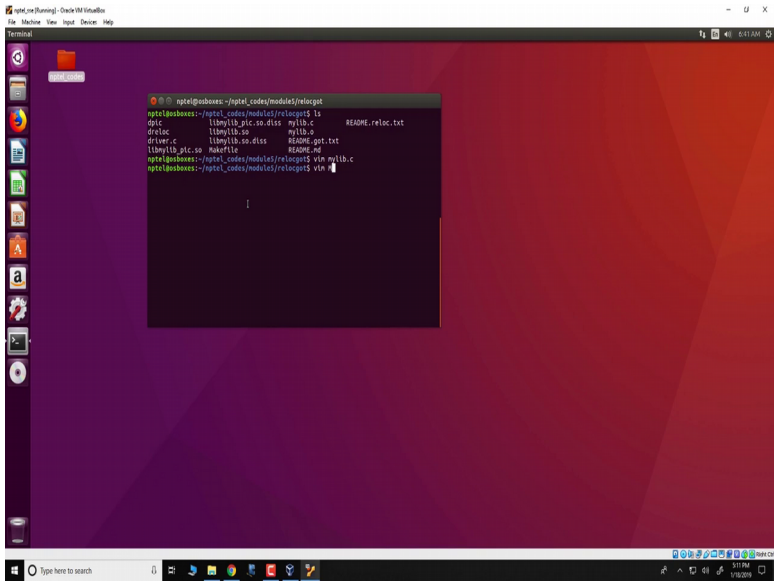


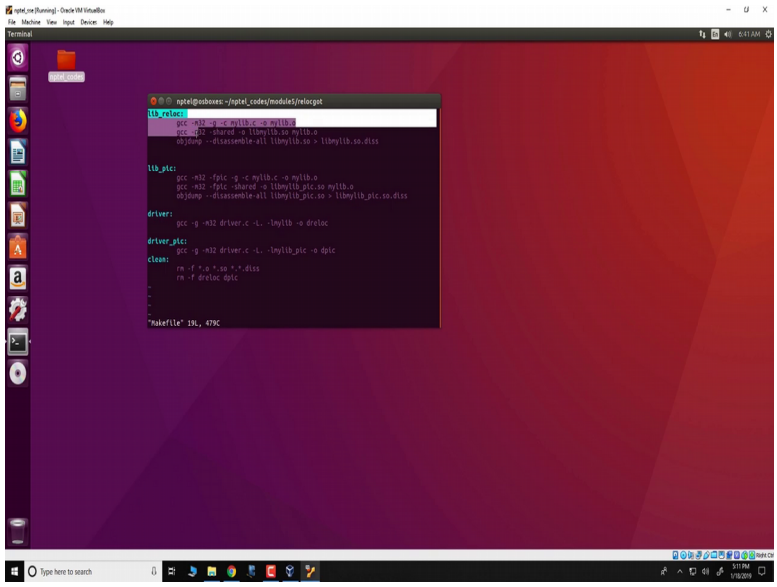


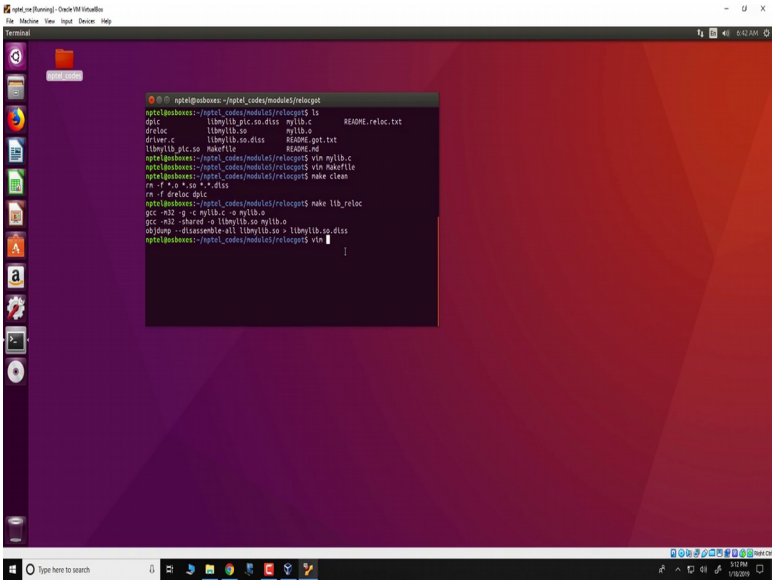
So the course that we use are available as part of this particular course and once you download and install the virtual box, the codes will be available in a model 5, direct subdirectory relocgot, so in this particular directory we would actually have two source files,

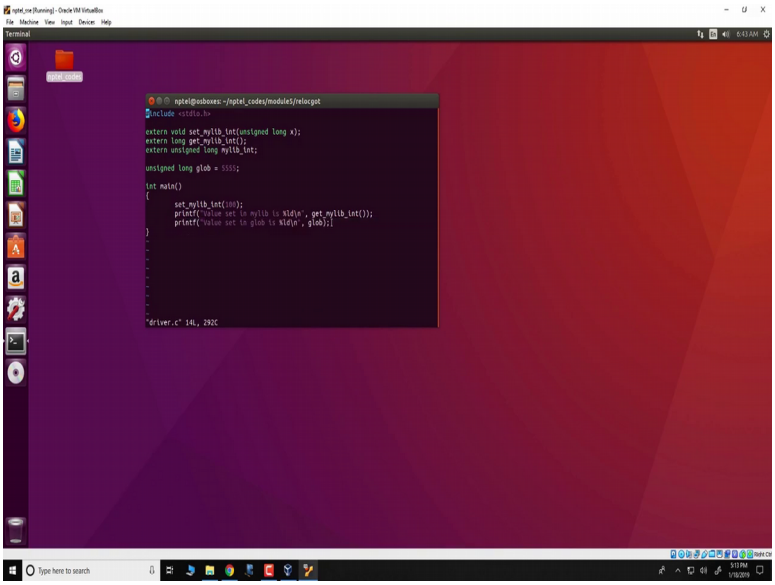
one is known as the driver.c and the other one is known as mylib.c, so first let us look at mylib.c which essentially creates a library, so this is a very simple library it comprises of a global variable mylib_int, it has two functions setmylib_int and getmylib_int, the setmylib_int takes unsigned long argument X and just copies it to this global variable and getmylib_int returns this global variable mylib_int.

(Refer Slide Time: 1:43)









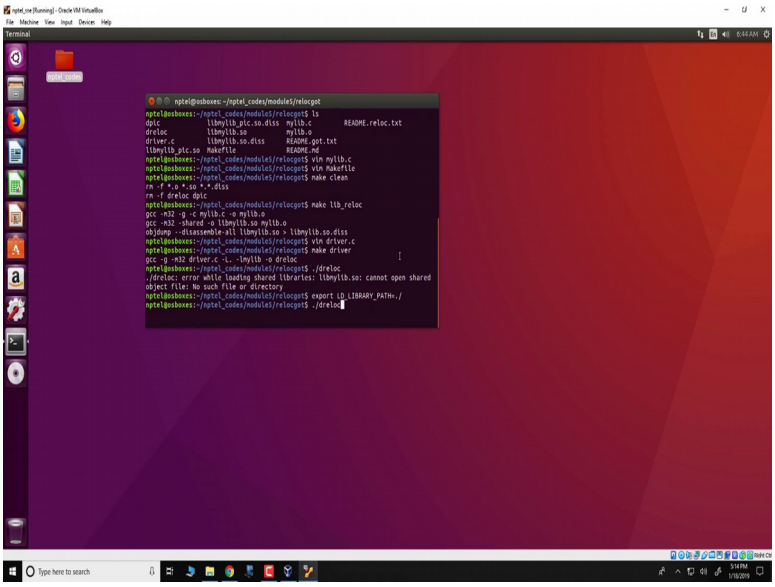
In order to create our library what we do is we have a makefile and this makefile we have several options but what we will be seeing right now is a this one, make a library which is relocatable, so in order to make the library we to make clean and then make lib_reloc, so what happens over here is that we compile other source code mylib.c, create an object file mylib.o and then create our library, the library is call libmylib.so and it comprises of this object file mylib.o, so what we also do is to and objdump and disassemble the entire mylib.so.

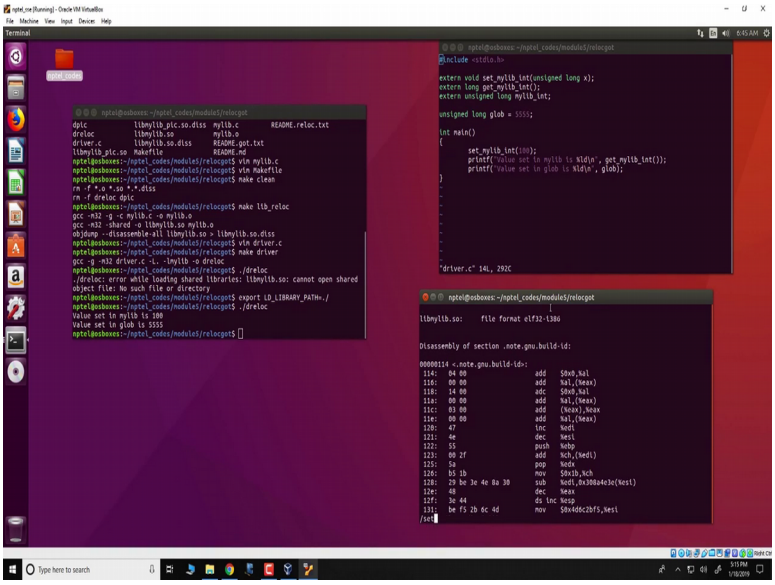
So the entire mylib.so this assembly is present in this file libmylib.so.disc, now in order to use this library we have written a driver program which is present in driver.c and what we see over here is we define externs two functions setmylib_int and getmylib_int according to the library which we have generated and also defined is extern unsigned long mylib_int which

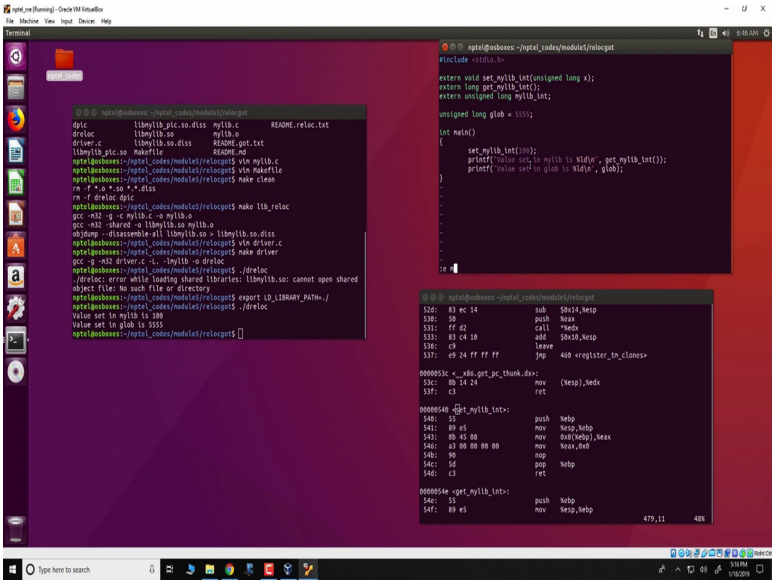
essentially is not very important, we invoked this function as follows setmylib_int which would internally invoke the library, said the value of mylib_int to 100 because we are passing the argument 100.

And in the second line we have this getmylib_int which would just print the value of mylib_int which should be 100, in a later part of the video we will see the use of this global data which is set to a value of 5555 and we print this value of 5555 over here.

(Refer Slide Time: 4:03)





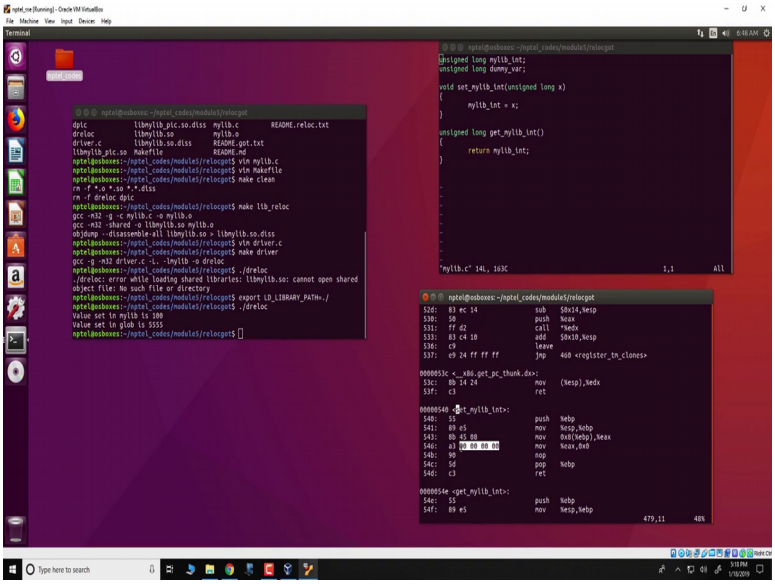


So let us actually compile this driver and as a make driver, so what you see here is that while compilation we specify minus L mylib, so this means that we are trying to link to the library that we have created, this minus capital L. Is the search path for this particular library, since we put dot over here it would mean that you want to search for the library in this particular directory.

So if you run this program, the program is called d relock then we would get expected output, yes we would get an error like this and this error occurs because we have not set the path for the LD, for the library, so we can do to as follows export LD path equal to dot/ and then run the executable and we see as expected that the value set to mylib_int is 100 and the value in global is 5555, this is as expected because that is what is present in the driver.c.

Okay, so now we will investigate why this particular code is relocatable, so in order to do that we will look at the disassembly of the library that we have created, so we collect that this disassembly is present in libmylib.so.disc, so let us open that up, we can operate in a separate thing, search for the function setmylib_int, here we can do mylib.c.

(Refer Slide Time: 6:32)

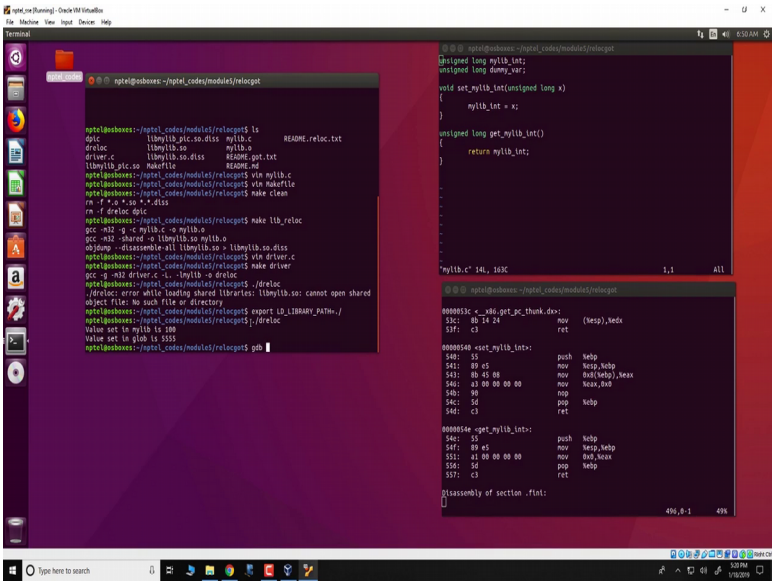


So what we see over here is the C function for setmylib_int and the assembly equivalent is here, so the first two instructions push EBP and move ESP to EBP are the usuals thing to actually create the stack frame and then importantly is this function, this particular instruction loads from an offset in the stack into a location EAX, so what is the here is that the argument X which is present at an offset of 8 bytes from the frame pointer is loaded into EAX.

Therefore at after execution of this instruction the EAX register contains the value of 100 which is the argument that we specified during our execution over here, now the next instruction is a store instruction, where it stores a value of EAX to mylib_int, so this is due to this statement in C, so this statement where the value of X is stored in mylib_int is executed in this instruction where EAX which comprises of X is stored in mylib_int.

But one thing you will notice over here is that the address for mylib_int which was supposed to be over here is filled with zeros, so we have 0000 0000 and this should be actually filled with the address of mylib_int.

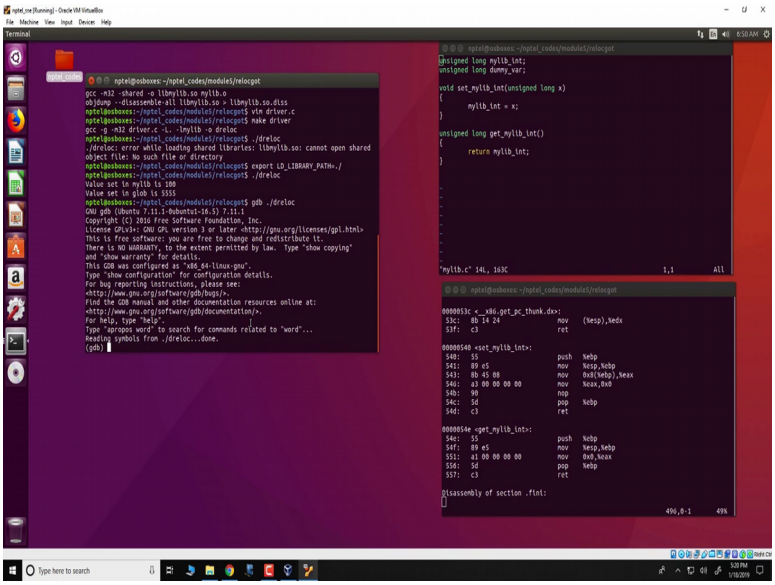
(Refer Slide Time: 8:21)

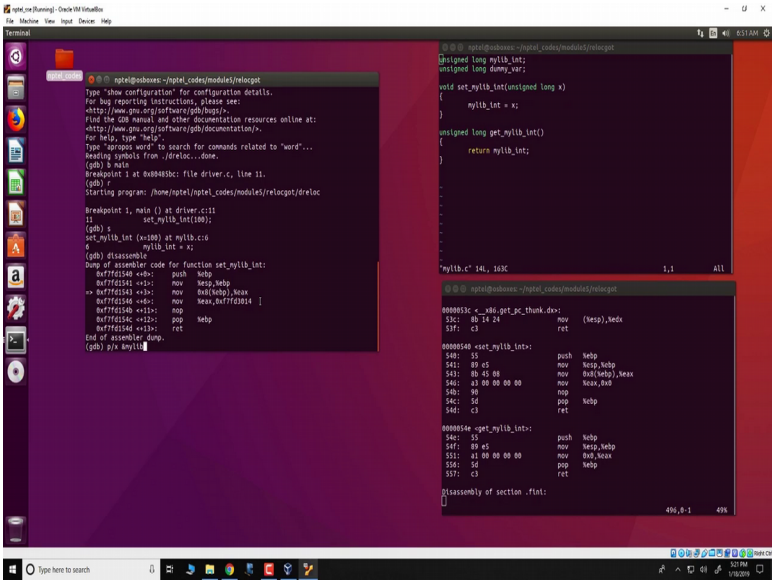


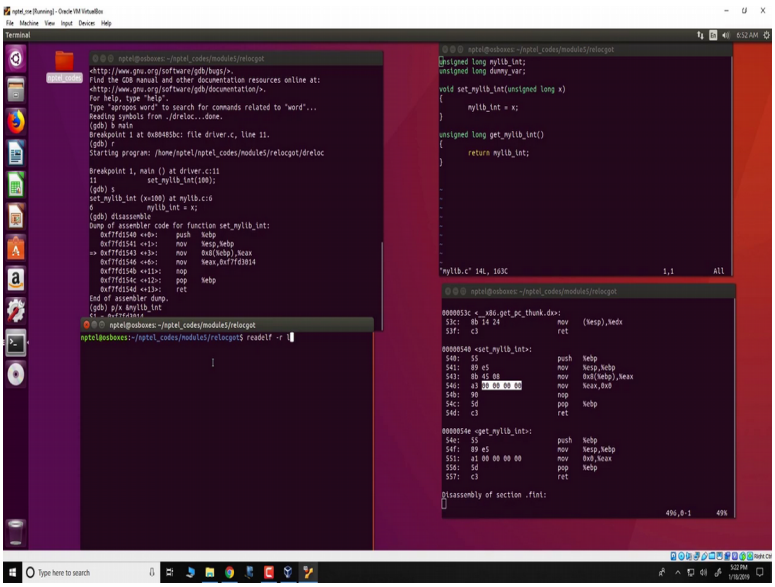
Similarly in the next function get mylib_int which returns the value of mylib_int, this is done by this statement where the contents of the global variable mylib_int is stored in the EAX register okay, now again what we see over here is just like setmylib_int, the address for the mylib_int is all set to 0, so node this is what the compiler inserts, now in the load time relocatable technique what happens is when we eventually load this program the loader would identify the address of mylib_int has to be fixed.

So therefore it would identify that at locations, this particular location 547 in the executable the actual address of mylib_int should be replaced, similarly over here the actual value of mylib_int should be placed, so let us see this happening in practice, so note at the OBJ dump that we have obtained is from the compiler, now what we will do is that we will look the output at runtime from GDB.

(Refer Slide Time: 10:01)



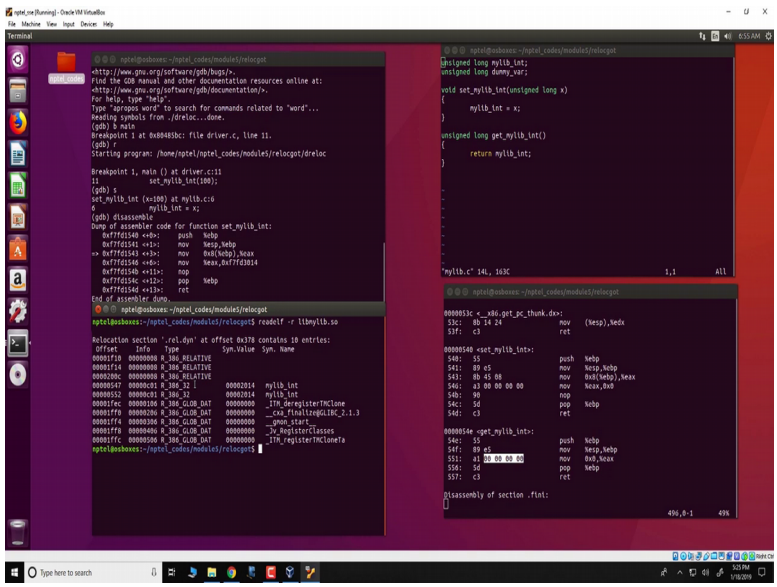




So we do this as follows we run GDB, the reloc and we put a breakpoint at min and run the program and single step into mylib_int and disassemble it and what we see is that this is the disassembly of mylib_int, the disassembly and the instructions use are exactly same as what the compiler has put in except for the fact that the zero which is present here is replaced with the actual address of mylib_int.

So if I print the address of mylib_int as follows, we see that it has the value X7 FT 3014, what is happening here is that the when the library is getting loaded into the process would determine that the address of mylib_int has to be fixed and therefore it would fixed it in this function, similarly the getmylib_int would also be fixed in a very similar manner, next thing to actually think of is how does the loader know where these locations are should be fixed, so that can be identified by a table present in the executable and we can use the command readelf minus R mylib_so.

(Refer Slide Time: 11:50)



So what you see here is that there are two entries for mylib_int, so it defines it that at an offset of 547 and 552 a 32-bit integer needs to be fixed, so if you look at this particular dump we see that at an offset 547 is essentially these four zeros and therefore the loader will look into this relocation table and determine that 4 bytes have to be fixed and the address is that of mylib_int, similarly at an offset of 552 which corresponds to these 4 bytes and getmylib_int, the address of mylib_int has to be fixed, so in this way the loader would determine at the time of loading that these regions in memory have to be fixed with the correct address, this achieves a relocatable code, the advantage of this code is that it is very simple to understand.

And however it makes a load time extremely complex, especially if you have a large number of such variables then the load time would actually be take quite long and also it requires the loader to actually go and modify executable code which is not what is actually required, so in the next demonstration what we will actually look at is another way of relocatable code using PIC a position independent code. Thank you.