6.033 Lecture 3
Operating Systems Part III

Americo De Filippo

May 16, 2023

1 How the gcc compiler works with modularity?

The first two are also called the Linking program (LD). So when you run gcc for produce an actual file that will run it will run in the back others programs that will do the follwing thing. The Linker takes as argument two kinds of object file: **Relocatable obj file** and a **Shared obj file** and produces as output and **executable file**

1.1 Symbol resolution and Relocation, Static Linking

This steps find for unresolved symbols, meaning that are not specified in the modules. Each file f_n contains a local sysbol table, that contains a set of items D_n that correspnd to local variable or functions that are defined in the module f_n. Then you have a set of thing U_n that are not defined in f_n.

Implmenting linkers (as in linux) we are going to scan from left to right building three sets.
The first set is called O (all the object file that go into the output O U f_i)
the second step is D (defined symbol) D U D_i, and the last is the U (undefined symbol so far) U u U_i - (the defined symbols in the current file that we are anlyzing). The linking success if the U set is null at the end. A problem of this implementation is a duplicate definition module. So it has to be careful finding some duplicate defintion methods or variable.

Symbol resolution with libraries A library is a bunch of obj file together, essentialy is just a concatenation of a lot of .o files. The Approc is essentialy the same with just one different. The main different is the size of the libraries are much bigger than our own programs. So the process of scanning all the library would be really expensive, for this reason we are going to include the .o file where are defined the remains items inside the U set.

The problem now is how to find the undefined symbols? We have a set of names and a set of values (where is defined). This is resolved through a name-mapping algorithm, that takes a context as input (seen in before lectures) There are Three ways of doing this:
The first way is a table lookup, (inode table or the table of defined symbol that very .o file takes with him)
The Second way is a path-name resolution. The third way (seen today) searching throught modules (search in all the file which is done in a form of table lookup).

1.2 Relocation

Relocation is pretty streightforward, in static linking cause it simply maintain a relocation table in each file.

1.3 Loading / Program Loading

The third varies a lot depending on the system. The problem solved by program loading is looking a what file has been requested, taking the context, putting in the memory and passing the control. In UNIX is done by 'execve', it pass the control to an interpreter that will invoce the first line in main (in C).

2 Shared object, Dynamic Linking

This idea solves the problem of including a library just one time when is called multiple times. Why have multicopies of the same module?

Problems with them The basic problem is that instruction are allocated with memory allocation, the problem is that when you have two programs that want to access at the same module at the same time, is very hard to share the object (is position dependent by where is called)

Solved by: Position Indipendent Code When you call the module and he allocates memory he will have allocation dependent to the Program Counter and not to the code has been called from. When you have the object indipendent code, you do not have to include the .o file where let's printf() is defined. Instead the object file has to maintain the file name from where printf() is defined (a pointer to the file). Thats called dynamic linking (instead of including the complete module that contains the defintion of a function we are going to maintain a reference to the position of the defintion and we invoce it when needed).