MITOCW | ocw-6.033-lec4

0:00:00 All right, so what we're going to do today is continue our 0:00:03.74 discussion of modularity and how you hook software modules up 0:00:07.678 together. And what we did the last time 0:00:10.172 was talked about how you share data between programs and between 0:00:14.044 users. And we went through the design of 0:00:16.276 the UNIX file system, or at least a particular aspect 0:00:19.689 of the UNIX file system where we talked about how we built a file 0:00:23.889 system out of layers and layers. And each layer was essentially 0:00:27.958 doing name resolution in order for us to take a UNIX path name 0:00:31.962 and eventually get the data blocks corresponding to the data 0:00:35.834 in that file. What we're going to do today is 0:00:39.795 to move away a little bit from the memory abstraction, 0:00:42.864 which is what we spend our time on the last time, 0:00:45.643 and start talking about the second abstraction, 0:00:48.191 the interpreter abstraction. And we're going to do that in 0:00:51.492 the context of software libraries, and basically 0:00:54.213 understand how software libraries are put together, 0:00:57.109 and how you can build large software programs out of 0:01:00.062 individual software modules that get hooked together. 0:01:04 So that's the plan for today. And in addition to seeing the 0:01:07.341 actual mechanism for how you take all these modules in the 0:01:10.625 form of libraries and other software pieces, 0:01:13.102 hook them up together, what we are really going to 0:01:15.925 focus on is the mechanism by which these different modules 0:01:19.209 actually run on your computer. OK, so it's going to involve a 0:01:22.666 fair amount of mechanism going down to the lowest layer until 0:01:26.123 you actually have something that's a single, 0:01:28.6 executable file that runs on your computer. 0:01:32 And then what we're going to do is to step back and think about 0:01:36.043 the kind of modularity that we will have gotten from this 0:01:39.695 approach. And that's the kind of 0:01:41.717 modularity that we're going to call soft modularity. 0:01:45.043 So, that's the plan for today. And you'll see why this is 0:01:48.695 called soft modularity. And it has to do with the way 0:01:52.086 in which propagation, a force, occurs and in 0:01:55.086 particular the kind of modularity with libraries that 0:01:58.608 we are going to discuss today has the property that a problem 0:02:02.521 in one module, for example, 0:02:04.217 an infinite loop or a crash in one module will turn out to 0:02:07.934 affect the whole program. And the whole thing will come 0:02:12.615 crumbling down. OK, and that's why it's going 0:02:15.065 to be called soft modularity. Of course, we don't like that 0:02:18.296 kind of modularity, although it is useful to have 0:02:20.969 soft modularity. So the next three lectures 0:02:23.308 after today, we're going to talk about hardening the soft 0:02:26.427 modularity using a variety of different techniques. 0:02:29.212 So that's the plan for the next three to four lectures. 0:02:33 OK, so let's take some examples to start with of where you end 0:02:36.669 up using these modules, these softer modules, 0:02:39.317 to build bigger software systems. 0:02:41.242 And the first one is actually what you see in programming 0:02:44.611 language when you use C or C++ or Java or Pearl or any of 0:02:48.16 these programs that you're familiar with, 0:02:50.567 you end up actually building large programs out of taking 0:02:53.936 lots of little programs, usually programmed in different 0:02:57.245 files, and running a compiler on it, and it'll turn out to 0:03:00.674 require other system software to take programs that are written 0:03:04.404 in different modules and hook them all up together into a 0:03:07.773 bigger program. And this is actually going to 0:03:12.065 be, we're going to learn today with examples from the C or C++ 0:03:16.123 programming language to find out how we build C modules together. 0:03:20.681 But, by no means are we restricted to these programming 0:03:24.526 languages. In fact, there's a lot of 0:03:27.018 different examples. Database systems provide a 0:03:30.222 great example of modularity. So if you take a system like, 0:03:34.804 you might've heard of Oracle or Sybase, or any of IBM's DB2, 0:03:38.474 all of these things are fairly complicated pieces of software. 0:03:42.269 But what they invariably allow you to do is to put in your own 0:03:46.064 software, your own code, that will run as part of the 0:03:49.299 database system. So if you, for example, 0:03:51.725 come up with, you know, normally databases 0:03:54.276 allow you to run queries on tables of data. 0:03:56.889 We'll be back after these messages. 0:04:00 OK, so every database system allows you to upload modules 0:04:03.281 into it that run as part of the database system. 0:04:06.035 So if you come up with a different kind of data type, 0:04:09.083 for example, some geometric data type on 0:04:11.368 which you asked whether there is a bunch of points inside a 0:04:14.767 polygon, a regular database system won't actually support 0:04:18.049 that. But what you can do is write 0:04:19.982 code for that, and upload it, 0:04:21.623 and run it as a module in a database system. 0:04:24.143 Another example, a common example these days are 0:04:26.897 Web servers, particularly something like Apache -- 0:04:31 0:04:35 -- where you can actually include as part of Apache when 0:04:38.735 you run it a bunch of different modules for doing a variety of 0:04:42.878 new functions that weren't previously running. 0:04:45.934 And all of these different kinds of systems will turn out 0:04:49.737 to use essentially variants of the same basic idea in terms of 0:04:53.88 solving the problem of taking these different modules together 0:04:58.023 and composing a single, large program that you can 0:05:01.351 execute on your computer. So what we're going to do today 0:05:06.348 is discuss this method, this way of doing modularity in 0:05:10.439 the context of, a particular example, 0:05:13.166 it's easiest to do this with an example. 0:05:16.121 And then we'll step back and talk about the general 0:05:19.909 principles. So, we're going to focus on the 0:05:23.09 Linux operating system running the new tools. 0:05:26.424 And in particular, we're going to focus on how you 0:05:30.136 do this in a language like C or C++. 0:05:34 And so, most of you are probably familiar with some of the tools 0:05:37.702 that you use to build programs. And we are going to use a 0:05:41.157 compiler called GCC that will turn out to use some other 0:05:44.551 pieces of software to build modular programs. 0:05:47.266 So, that's kind of what we're going to start with. 0:05:50.29 And the basic approach in terms of how these modules are going 0:05:54.054 to be written is if you want to write a program, 0:05:56.954 you typically design what you want to do, design the system, 0:06:00.595 and then decide on how you decompose the functionality of 0:06:04.05 your system into a bunch of different modules. 0:06:08 And typically you write one or more files for each module and 0:06:11.949 system, and then you need a way to bring all these modules, 0:06:15.766 these different files together to build a bigger program. 0:06:19.452 And when you run GCC in particular on a dot O file, 0:06:22.743 the modules themselves want to compile a C file. 0:06:25.836 As you know, you produce an object file with 0:06:28.666 a name like file dot O, and that actually contains in 0:06:32.089 it, that's an object file that with some more work you can 0:06:35.84 arrange to run on your computer. And what we're going to do 0:06:40.783 today is figure out how when you have a large number of object 0:06:44.966 files, and it will also turn out, we'll talk about something 0:06:49.013 called a library, which is a collection of object 0:06:52.305 files put into a single archive, or how you can take those 0:06:56.214 object files and produce an executable that will run. 0:06:59.78 So if this works, let me show you an example of 0:07:02.935 what we're going to be talking about. 0:07:05.404 So I have two little files, a little trivia program. 0:07:10 I don't know if this is visible. 0:07:12.214 OK, so all this is doing is computing the square of the 0:07:16.071 number. And it uses here, 0:07:17.785 this is a very trivial program. It defines a few variables, 0:07:21.928 and then just calls on modular function called SQR, 0:07:25.5 which isn't actually in this file. 0:07:27.857 It's in a separate file. But one thing you can do is run 0:07:31.785 GCC-C which just tells it to produce the object file, 0:07:35.5 and there's a program called object dump which is a useful 0:07:39.571 program if you want to see what's inside your object file. 0:07:45 It's a binary file, so you're not going to be able 0:07:49.758 to check it. But you could run it with, 0:07:53.741 we've already reached the bottom, all right. 0:07:57.917 OK, so the part to worry about for the moment are shown in the 0:08:03.841 last three lines that you could see here starting main, 0:08:09.086 SQR, and print F. And those are the three 0:08:13.452 functions that are being invoked by the program I showed you. 0:08:17.242 And you can see, for SQR and print F, 0:08:19.515 this object file doesn't actually know where it is or 0:08:22.8 where its definition is. And that's why you see UND for 0:08:26.21 undefined. And part of what we're going to 0:08:28.8 figure out today is how we can find out where those other 0:08:32.336 modules are defined, and hook up all those different 0:08:35.557 modules together to build a program that will actually run 0:08:39.157 to do what we want it to do. So the other thing here is 0:08:43.777 square dot C, which does the obvious thing of 0:08:46.786 just multiplying two numbers. And you could do the same thing 0:08:50.888 with square dot C. So I have a little thing called 0:08:54.239 show object which is just an alias project dump. 0:08:58 0:09:02 And you can see here that there aren't actually any undefined 0:09:06.664 symbols or undefined functions because square, 0:09:10.163 in turn, didn't invoke anything that was outside and defined 0:09:14.75 somewhere else. And underneath here, 0:09:17.471 you see the actual disassembled assembler code for this 0:09:22.214 machine text. So going back to M dot O, 0:09:25.168 I just want to draw your attention to a couple of things. 0:09:29.522 Yeah? 0:09:31 0:09:36 Shows up fine on my screen, I know. 0:09:38.956 [LAUGHTER] All right, good. 0:09:41.217 OK, so there's a few sections to this object file here. 0:09:45.913 And it's important to understand a little bit how an 0:09:50.347 object file is laid out. It has in it a few different 0:09:54.869 sections as I mentioned. One of the sections is called 0:09:59.478 the text section. The text section basically 0:10:03.569 contains the machine code. Ideally, once you hook all these 0:10:07.01 modules together, that machine code will actually 0:10:09.907 just run on your computer. It also has a section, 0:10:12.804 I think you can see it up there, called auto data. 0:10:15.762 That's the second section that stands for read only data. 0:10:19.142 And one of the things I had here in that program was 0:10:22.221 something that said print F with a string inside. 0:10:25.118 And, that string is read only data. 0:10:27.171 I mean, you don't want, the program doesn't actually 0:10:30.249 modify it. And that's the kind of thing 0:10:33.534 you could see it on the right. There is a comment there: 0:10:36.493 the square of something is something. 0:10:38.465 And that's an example of read only data. 0:10:40.602 And then it has a section for data which really corresponds to 0:10:43.945 the global variables used in the program. 0:10:46.136 And for those of you who remember 6.004, 0:10:48.273 and if not we'll talk about this next time. 0:10:50.575 The look of variables in your modules are not actually in the 0:10:53.863 data section. They're typically on the stack. 0:10:56.273 So we're not going to worry about that for today. 0:11:00 And then there's a section called a symbol table which is 0:11:04.595 shown here, I think, as sym tab not on the 0:11:08.206 screen. Let me go back. 0:11:10.012 So that's the symbol table. And what the symbol table shows 0:11:14.772 is the different global variables and functions that are 0:11:19.285 either defined in the module in this daughter file or are 0:11:23.881 referenced by this module. OK, and for the symbols that 0:11:28.775 are defined in this module, what the symbol table tells you 0:11:32.452 is the address at which you can find it in the module. 0:11:35.813 OK, and for the things that are not defined, it says it's 0:11:39.364 undefined. And hopefully once the compiler 0:11:41.963 sees all of the different object files involved, 0:11:44.943 it can piece together these different object files and build 0:11:48.684 a larger program module. So when you compile each of 0:11:51.918 these things in this example, M dot O and SQR dot O, 0:11:55.152 you will find that the object file that's produced for both 0:11:58.829 files starts with address zero. OK, so obviously when you hook 0:12:03.878 these modules up together, you can't have both modules run 0:12:07.569 at address zero. So one of the things you need 0:12:10.484 to be able to do is to take these different object files 0:12:14.046 together, and somehow join them so that the addresses are no 0:12:17.867 longer colliding with one another. 0:12:20.005 So that's one problem that we are going to have to solve. 0:12:23.632 So that's what we're going to do today. 0:12:27 So there's basically three steps that we are going to talk about 0:12:30.597 today. The first step is figuring out 0:12:32.719 all these different symbols. So when you have SQR or if you 0:12:36.14 were using square root in a different program, 0:12:38.793 print F is another example, figuring out where the 0:12:41.683 definitions of these symbols are, and where the definitions 0:12:45.103 of the global variables are in your program. 0:12:47.638 And that's a step called symbol resolution. 0:12:51 0:12:55 OK, and the plan is going to be that each object file is going 0:12:59.131 to have inside it a table like here that shows for the symbols 0:13:03.262 that have been locally defined, where in the local module they 0:13:07.394 are. I'm going to use the word 0:13:09.358 module for these dot O files. And for symbols that are not in 0:13:13.422 the same dot O file, it just says it's undefined. 0:13:16.673 And I need it. OK, and that's what the dot O 0:13:19.585 contains. And, when you take all these 0:13:22.091 dot O's together and produce a bigger program, 0:13:25.139 those symbols are going to be resolved. 0:13:29 And that's the step called symbol resolution. 0:13:31.918 The second thing we have to do before we get a single big 0:13:35.632 program is to do something called relocation.

[illegible]
0:13:38.551 So what relocation is, is remember I told you when you 0:13:42.066 compile a program to a dot O file, all of the dot O files all 0:13:46.045 have addresses starting from zero. 0:13:48.234 You can't run them all together unless you actually modify the 0:13:52.28 different addresses. And any time you see a 0:13:55.066 reference to a variable in one of the dot O files, 0:13:58.316 you have to modify that address to point to the actual address 0:14:02.362 at which the instruction would run or the data object is 0:14:06.01 present. And that's called relocation. 0:14:09.745 And it'll turn out that the object files that are produced 0:14:13.175 by the compiler are what are called relocatable object files. 0:14:16.787 Pretty much every object file these days is relocatable. 0:14:20.097 But the relocatable object file allows you to do this relocation 0:14:23.888 easily because it has an information that tells whoever 0:14:27.138 is composing these modules together how to modify the 0:14:30.268 addresses that are referenced within each module. 0:14:34 And the third step once you do all this and you produce a 0:14:38.307 single big program that's ready to execute, is actually to run 0:14:43 the program. And that's a step called 0:14:45.769 loading or program loading. So you create a big executable 0:14:50.153 file, and you type it on the command line. 0:14:53.307 What happens? So, those are the three steps 0:14:56.538 that we are going to be talking about today. 0:14:59.846 And the first two steps typically are called linking. 0:15:05 OK, so most of today is going to be talking about how linking 0:15:08.829 works. And the piece of software, 0:15:10.872 system software, that does linking is called a 0:15:13.744 linker. And in new Linux the 0:15:16.042 operating system, it's a program called LD. 0:15:18.787 That's the linking program. And so when you run GCC to take 0:15:22.489 a bunch of modules and produce a program out of it, 0:15:25.68 underneath inside not visible to you but underneath, 0:15:28.936 GCC actually invokes a bunch of other programs including LD as 0:15:32.893 one of the last steps to produce an actual file that will run. 0:15:38 So the linker takes as input, object files, 0:15:41.452 and produces as output a program that you can run. 0:15:45.479 And then the loader takes over and runs the program. 0:15:49.671 So there are actually three kinds of object files that the 0:15:54.356 linker takes as argument. The first kind of object file 0:15:58.794 is a relocatable object file. 0:16:02 0:16:08 OK, actually there's two kinds. The first kind is a relocatable 0:16:11.831 object file. And that's what we're going to 0:16:14.426 be spending most of our time on. The second kind of object file 0:16:18.257 is something called a shared object file. 0:16:20.667 We are going to get to this a little bit at the end. 0:16:23.818 And the reason we're going to wait until the end is that the 0:16:27.464 reason for a shared object file is that if you have a program, 0:16:31.233 a module like print F or some math library like using 0:16:34.446 functions like square root, if you just do linking the 0:16:37.721 normal way, which is the easy way we're going to talk about 0:16:41.305 for most of today, you will end up with a program 0:16:44.271 that includes in its text that includes in the binary all of 0:16:47.916 the modules corresponding to all of the libraries at, 0:16:51.129 you know, the text corresponds to the different modules that 0:16:54.775 you use. So now, if you have 100 0:16:58.133 different programs running on your system, all of which use 0:17:00.992 print F, it will turn out that every program is quite big. 0:17:04 And so, shared object files allow us to not actually have to 0:17:07.464 include the text of print F in every binary that we produce, 0:17:10.928 but just to maintain a pointer to something that says, 0:17:13.629 when you need print F, go and get this, 0:17:15.86 and include it as part of your program. 0:17:18.091 So we get to that in the end. But for the most part, 0:17:21.086 we are going to be focusing on relocatable object files, 0:17:24.315 that is, object files that are produced by something like GCC 0:17:27.838 that I showed you, which include in it information 0:17:30.715 for modifying the addresses that are being referenced inside the 0:17:34.414 module. And once the linker takes these 0:17:38.423 as arguments, what it produces as output, 0:17:42.144 out comes an executable file. And these daughters don't 0:17:47.076 actually run. The executable is a thing 0:17:50.365 that's capable of actually running. 0:17:54 0:18:01 OK, so the first step we had to talk about is symbol resolution. 0:18:05 0:18:12 And so the input is a set of object files. 0:18:14.468 And the output is going to be something that allows LD, 0:18:17.719 the linker program, to know which symbol is located 0:18:20.73 where in all the different modules. 0:18:22.777 And if your linking succeeds, then it means that every symbol 0:18:26.39 that's been referenced by any of the modules that's on the 0:18:29.821 command line of this LD program, there are no undefined symbols 0:18:33.554 remaining. So that's what symbol 0:18:36.276 resolution does. And the input to this is really 0:18:39.134 it looks something like you have a bunch of files. 0:18:42.113 I'm going to call them F1 dot O, F2 dot O, all the way through 0:18:45.821 FN dot O. OK, and for those of you who 0:18:48.071 know what libraries, don't worry about them now. 0:18:50.928 We are just going to take these individual modules, 0:18:53.968 and we are going to produce out of it a single program that 0:18:57.494 includes in it all of these different modules such that 0:19:00.777 there are no unresolved symbols, OK? 0:19:04 So, SQR is one file, and M dot C contains references 0:19:07.743 SQR, we want to make it so the actual program, 0:19:11.045 when the reference to SQR comes in, you know where it is. 0:19:15.155 So, what each file contains when you do GCC on F1 dot CN, 0:19:19.266 you get F1 dot O, it contains in it already a 0:19:22.495 local symbol table. And you saw that in the example 0:19:26.165 I showed you. And the symbol table has the 0:19:29.893 following structure. There's two kinds of data in 0:19:33.138 the symbol table on a local module. 0:19:35.437 The first is a set of items, D1, that correspond to global 0:19:39.291 variables or functions that are defined in the module F1 dot C, 0:19:43.483 and therefore are defined in the module F1 dot O. 0:19:46.729 So, this just tells the linker, you know, if you see the symbol 0:19:50.921 being invoked by somebody else, it's being defined in this 0:19:54.775 module. OK, and it contains in it 0:19:56.939 information about where in the local module it does it so that 0:20:01.064 you know at what address it's being defined and so on. 0:20:06 Likewise, you have here another set of symbols that you want 0:20:10.879 which corresponds to things that are not defined in F1 dot O. 0:20:15.842 And likewise, for each of these things you 0:20:19.233 have D2 and U2, and so on, OK? 0:20:21.631 So, the way the particular implementation of the linker 0:20:26.097 under new Linux works, and there are many ways to 0:20:30.067 implement linkers, and there are very complicated 0:20:34.037 linkers, and obviously simple linkers as well. 0:20:39 The Linux one is particularly simple at least in terms of this 0:20:42.789 way of linking, resolving all of the symbols 0:20:45.461 here. You just scan the command line 0:20:47.636 from left to right and you start building up three sets. 0:20:51.053 OK? So

we're going to build up 0:20:52.855 three sets in this algorithm when we scan from left to right. 0:20:56.582 The first set is a set I'm going to call O. 0:21:00 And the idea in O is going to be all of object files that go 0:21:03.374 into the program, that go into the output of the 0:21:06.063 linking step. In this case, 0:21:07.55 in the end, it's going to be pretty straightforward. 0:21:10.467 It's going to be F1 dot O, union F2, all the way up to 0:21:13.499 union FN. It's going to get a little more 0:21:15.786 complicated when we talk about libraries. 0:21:18.074 But for now, it's a very easy set. 0:21:19.962 When you scan from left to right, you look at the next 0:21:22.994 object module or the object file, and all you have to do is 0:21:26.311 O goes to O union FI dot O. OK, this is the I'th step of 0:21:30.8 the algorithm. OK, now we have to go to the 0:21:33.502 next step, which is get to the defined symbols. 0:21:36.46 So as we go from left to right, we build up a set of defined 0:21:40.255 symbols. Initially it's null, 0:21:42.056 and then we start with D1, and then we do D1 union and D2, 0:21:45.721 and all the way. So up to the I'th step, 0:21:48.23 what we're going to end up with is obviously at the I'th 0:21:51.767 iteration, we are going to have some running set of defined 0:21:55.497 symbols that have been defined in the modules so far. 0:22:00 And we're just going to do D goes to D union DI where DI is 0:22:03.678 the set of symbols that are defined in module number I. 0:22:07.103 OK, in the last set, we're going to calculate, 0:22:09.957 and the linker is going to calculate, is a set U. 0:22:12.938 And U is the set of all undefined symbols so far. 0:22:15.983 So you have a set of undefined symbols. 0:22:18.393 And you are at the I'th stage of this process where up to here 0:22:22.262 you have a set U of undefined symbols. 0:22:24.608 And now you're seeing this new module a bunch more undefined 0:22:28.35 symbols. So clearly what you have to do 0:22:31.909 first is do union UI, correct, because you now have 0:22:35.092 built up a bigger set of undefined symbols. 0:22:37.765 But then, notice that there might be symbols that were 0:22:41.139 previously undefined that are now being defined in the I'th 0:22:44.83 module, right? Otherwise, I mean, 0:22:46.867 if this set kept growing, then all you would end up with 0:22:50.368 is a big undefined set at the end. 0:22:52.469 So clearly there are symbols that are being defined and 0:22:55.906 subsequent modules. So you have to subtract out a 0:22:58.961 set. And the set you have to 0:23:00.68 subtract obviously is the undefined set intersection 0:23:03.926 whatever is being defined in this module. 0:23:08 OK, and so the linker does this from left to right. 0:23:11.26 It's actually a pretty simple linker because it doesn't go 0:23:14.978 back and do anything. And it doesn't actually have to 0:23:18.369 at least for this way of hooking up object files together. 0:23:22.086 And what you get at the end are three sets: O, 0:23:25.021 D, and U, OK? And, the linker outputs 0:23:27.369 success, it then gets to the relocation stage if U is null. 0:23:32 If there are no undefined symbols at the end, 0:23:34.037 you know that now you can then relocate by modifying patching 0:23:36.814 the different addresses together to produce your big program. 0:23:39.592 But if the set is not null, you know that no matter what 0:23:42.138 you do to relocate, there is some symbol that is 0:23:44.314 not being defined, which means that module won't 0:23:46.49 run, which means the program won't run, OK? 0:23:48.435 So that's kind of what this linking program does. 0:23:50.657 It just goes from left to right and resolves all symbols. 0:23:53.25 And although it's done in the context of, we've discussed this 0:23:56.074 in the context of a particular example of how new Linux 0:23:58.898 does its linking, the basic idea is the same. 0:24:02 Essentially, all symbol resolution will turn 0:24:05.583 out to use some variant of something that greatly resembles 0:24:10.416 this method. There is actually only one 0:24:13.583 problem with what I described so far, and it's wrong. 0:24:17.916 So, what's the problem? Yeah? 0:24:21 0:24:29 Right. If the symbol is undefined in 0:24:30.775 F1 and defined in F2, you're fine because let's take 0:24:33.362 this example. I had M dots [SOUND OFF/THEN 0:24:35.442 ON] it's OK. We're just building up sets. 0:24:37.471 So you can have a symbol that's defined. 0:24:39.449 You are saying, what is the symbol is defined 0:24:41.681 in one of the files and undefined in the next file? 0:24:44.217 No, it doesn't matter for this. It's going to matter when we 0:24:47.21 talk about something called a library, but it's not going to 0:24:50.202 matter here because we've built up these sets of what are 0:24:53.043 defined and what are undefined? So let's say you define a 0:24:55.934 symbol here, and you come here and you find that it references a 0:24:59.079 symbol that's not locally defined, but has been previously 0:25:01.92 defined, right? That's been built up in the 0:25:05.528 set, D. So, in the end -- 0:25:07 0:25:11 Oh I see, so the code is a little wrong because I had to 0:25:14.689 actually update. You're right. 0:25:16.634 I had to modify that U line. This is wrong. 0:25:19.451 Good. So I have to modify the set of 0:25:21.798 undefined symbols to include the things that were previously 0:25:25.756 defined. So actually it should probably 0:25:28.304 have been U intersection D. And that will probably fix it. 0:25:33 Good. Any other errors? 0:25:34 [PASUE] Yeah? 0:25:38.362 Right. So, there were two. 0:25:40.175 This one I actually didn't realize. 0:25:42.64 But it's actually right. So it should be U intersection 0:25:46.555 D. The other one is that we just 0:25:48.803 keep doing D goes to D union DI. But now, if I define a 0:25:52.719 function, SQR here, and I define the same function 0:25:56.271 SQR again here, we're not going to know which 0:25:59.462 SQR to actually use. So we have a duplicate symbol. 0:26:03.526 And so, in fact, what the algorithm actually 0:26:05.714 ought to do is while it's doing this union competition, 0:26:08.461 it better make sure that any symbol that's being defined in a 0:26:11.514 subsequent module has not already been defined in a 0:26:14.057 previous module. And, if there is a duplicate 0:26:16.296 definition, it'll tell you that there's a duplicate definition. 0:26:19.45 And I'm not going to show you an example, but if you just go 0:26:22.452 and type out a little U's SQR twice and you run the compiler, 0:26:25.505 GCC, on it what you'll find is that it'll tell you that this 0:26:28.507 symbol is multiply defined. OK, so we're not going to want 0:26:33.079 that either. So once you obtain these 0:26:35.576 different sets, what you'll end up with is 0:26:38.418 information that will tell the linker everything about all of 0:26:42.578 the different symbols, and where they have been 0:26:45.768 defined, and in which object file they've been defined. 0:26:49.511 And so, the step that it has to do after that is the relocation 0:26:53.81 step. So this was the first step. 0:26:57 0:27:05 It will turn out that this step is also pretty straightforward 0:27:08.607 because what happens when GCC runs on a single C file and 0:27:11.919 produces a single dot O file is it contains in it information on 0:27:15.645 how to relocate all of the variables and all of the 0:27:18.602 functions that are defined in that module. 0:27:21.026 And there is a section of the object file that I haven't shown 0:27:24.634 you called the relocation section that tells you.

0:27:27.473 for any given variable in the text, or any given line of code 0:27:31.021 in the text area, all of the load instructions 0:27:33.682 and all of the variables, how they get remapped inside. 0:27:38 So that's maintained. Basically the approach is to 0:27:41.617 maintain the local table. And, it's called the relocation 0:27:45.536 section of your program, OK? 0:27:47.571 So, we're going to get to loading in a little bit. 0:27:51.339 But before we get to loading, I want to get back to symbol 0:27:55.634 resolution. So here we just talked about 0:27:58.573 taking a bunch of dot O files and producing a final program. 0:28:04 But there is another notion of something called a library. 0:28:07.137 So, how you do symbol resolution with a library -- 0:28:10 0:28:15 So an example of a library here is lib C dot A, 0:28:17.45 which is the standard C library. 0:28:19 And that's the library that contains the definition of print 0:28:21.95 F. So if you use print F or any of 0:28:23.599 these other standard functions, they define the C library. 0:28:26.45 Another example is the math library, where you have the 0:28:29.15 square root function and a bunch of other mathematical functions. 0:28:33 So we are going to want to know how to resolve with the library. 0:28:37.156 So first, we have to know what a library is. 0:28:39.993 And what a library is, is just you take a bunch of 0:28:43.226 object files together and basically just concatenate them 0:28:46.921 together. It's not literally 0:28:48.703 concatenation because the library also maintains an index 0:28:52.397 that says, has information about what modules, 0:28:55.367 what included dot O files contain which symbol 0:28:58.336 definitions. But essentially it's just a 0:29:00.909 concatenation of a bunch of dot O files, OK? 0:29:05 And they get put together in a library. 0:29:07.047 And there is some index information in front that says 0:29:09.903 what is where inside the library. 0:29:11.627 OK, so the approach to resolving symbols with a library 0:29:14.537 is almost the same as what we have so far, except with one 0:29:17.608 twist. And the twist is that often 0:29:19.386 libraries are extremely big, much bigger than a single dot O 0:29:22.565 file. And, one approach to resolving 0:29:24.451 with libraries would be to apply the same approach here. 0:29:27.415 So when you have something that's defined in the standard C 0:29:30.54 library like lib C or the math library like lib M, 0:29:33.342 you could just include the entire text of the library 0:29:36.144 inside to build your program. But that just makes things 0:29:40.577 extremely bloated. I mean, think about if you just 0:29:43.489 wanted to use print F in your program like we had in this 0:29:46.887 example and you had to include the entire C library, 0:29:49.981 which is megabytes long, that seems like not a good way of 0:29:53.317 doing the linking. So what we're going to do with 0:29:56.23 the resolution of libraries is essentially the idea is to only 0:29:59.93 include the dot O files that are in the library in which 0:30:03.267 undefined symbols that were previously encountered are 0:30:06.483 defined. And if you think a little bit 0:30:09.84 about what I said, that's one reason why it 0:30:12.505 usually a good idea when you do GCC and use the linker, 0:30:15.931 use LD, to specify the libraries at the end because 0:30:19.104 what we're going to do is we're going to take all the daughter 0:30:22.974 files, and then they're going to be things like dash LM. 0:30:26.464 I mean, the standard C library is usually, by default, 0:30:29.827 already included on the command line. 0:30:33 But if you have functions like square root and so on, 0:30:35.748 here what we're going to do is we're going to build up. 0:30:38.497 What the linker is going to do is it is going to build up a set 0:30:41.775 of undefined symbols until it gets to the end. 0:30:44.154 And there's going to be undefined symbols remaining. 0:30:46.85 If you use the square root program, the square root 0:30:49.493 function, and you didn't write your own square root function, 0:30:52.665 it's in the math library. Then you might ask for the math 0:30:55.625 library to be included, and you want to pull the 0:30:58.11 definition of square root from the math library. The way in 0:31:01.123 which the linker does to pull the right dot O file is that the 0:31:04.348 math library is going to have information that says which 0:31:07.308 object file contains what symbols, and anytime you see an 0:31:10.268 undefined symbol at this stage, we are going to scan this 0:31:13.229 archive looking for the symbol that's been undefined, 0:31:15.977 in particular looking at this example for the square root 0:31:18.938 symbol. OK, and when we find the square 0:31:22.869 root symbol somewhere inside in some dot O file, 0:31:25.704 we're going to take that dot O file, and not the rest of the 0:31:29.263 library, and then use this algorithm that we did. 0:31:32.158 So take that dot O file alone and push that as input to this 0:31:35.717 algorithm. That's the way in which we're 0:31:38.069 going to build it up. So that's why at least in this 0:31:41.74 particular implementation of the algorithm, the linker which is very simple, 0:31:45.22 if you put the LM way up in front, you are a little bit in 0:31:48.527 trouble because if FN dot O is the file that actually uses 0:31:51.834 square root, and the math library was included well in 0:31:54.908 front, then square root would not yet have been part of the 0:31:58.273 undefined set of symbols until then. 0:32:01 And there are other ways to deal with it. 0:32:03.222 You can have linkers that go in more passes that presumably can 0:32:06.666 deal with this problem. But this is just an example of 0:32:09.611 how new Linux does this, and that's just worth knowing. 0:32:12.777 So this idea of linking and symbol resolution and 0:32:15.444 relocation, people have worked on this for a very long time. 0:32:18.722 And almost every software system uses it. 0:32:20.944 In fact, if you use LATEC to build your 0:32:23.666 design papers and so on, it does symbol resolution as 0:32:26.555 well because there is all sorts of cross-references that are 0:32:29.833 there and later. And it uses essentially the 0:32:33.416 same kinds of algorithms, go over the files and go over 0:32:36.25 the documents in multiple passes and resolve the symbol. 0:32:39.135 So it's a pretty general algorithm that we described here 0:32:42.074 of building these different sets to ultimately figure out whether 0:32:45.432 there are undefined references remaining at the end or not. 0:32:49 0:33:02 So I want to step back for a minute and generalize on the 0:33:05.821 different approaches that we've seen so far for doing symbol 0:33:09.847 resolution. And today we saw one approach 0:33:12.576 for doing symbol resolution. But in fact, 0:33:15.305 last time we saw a couple of different approaches to resolve 0:33:19.331 names whose values we didn't actually know. 0:33:22.197 So, we're going to step back and generalize with a couple of 0:33:26.223 different techniques that we saw from the different examples. 0:33:31 So the general problem here, the specifics are how you can 0:33:35.854 find out where these undefined symbols are defined, 0:33:40.112 or in the UNIX example, how you can take a big path 0:33:44.37 name and identify which blocks contain the files or contain the 0:33:49.651 data for the file that was named in the path. 0:33:53.398 But the general problem is you have a set of names. 0:33:57.656 And associated with each name is value. 0:34:02

0:34:06 And these names get associated with or mapped to the different 0:34:09.269 values. In fact, the names get bound 0:34:11.32 to the different values. And in this example, 0:34:13.759 the linker needed to take a name like the definition of a 0:34:16.862 symbol and needed to identify, what's the value associated 0:34:20.021 with that symbol? The value here in this context 0:34:22.626 is, where is this name, the square root function? 0:34:25.286 Where is it actually defined? At what location is it? 0:34:29 OK, and so the way in which that resolution is going to be 0:34:32.828 done is done in general using something called the name 0:34:36.522 mapping algorithm. 0:34:38 0:34:48 And the mapping of the name, the value being done by the 0:34:51.869 name mapping algorithm takes into account, 0:34:54.753 or takes as input something called the context. 0:34:57.989 And I'll describe this with an example in a minute. 0:35:02 So somebody has found a name to value. 0:35:03.947 And when you are a linker or when you are part of the UNIX 0:35:06.947 file system, and you're trying to identify the value associated 0:35:10.21 with the name, you're going to have to resolve 0:35:12.578 that name to find a value. And that resolution is going to 0:35:15.526 be done in a particular context. So, for example, 0:35:18.052 you might have two files with the same name in two different 0:35:21.052 directories, and that's fine. The same name can have two 0:35:23.947 different values because that resolution from the name to the 0:35:27.105 correct value, and the direct case it's an 0:35:29.263 inode number would be done in the context of the 0:35:32.315 directory in which the resolution is being done. 0:35:36 OK, so what we've seen over the last couple of days, 0:35:39.954 the last lecture, and today, are three different 0:35:43.598 ways of doing it. And it turns out that in almost 0:35:47.32 every system, or in every system that I know 0:35:50.655 of, anyway, there's basically three ways of doing this name 0:35:55.152 resolution. The first way is a table 0:35:57.866 lookup. Within the context of a dot O 0:36:00.735 file, which has a set of defined symbols. 0:36:05 Taking one of the symbols, the name, in that case, 0:36:07.803 this input, and finding out where it's been defined in that 0:36:11.121 dot O file is basically a table lookup. 0:36:13.296 That's what that symbol table section describes. 0:36:15.985 From the disk example from last time, the inode table is an 0:36:19.475 example of a table lookup. I mean, there's a portion of 0:36:22.564 disk that has in it the mapping between inode numbers and the 0:36:25.997 corresponding inodes. And that's just a table lookup. 0:36:30.174 So when you want to go from an inode number to an inode, 0:36:34.032 you do a table lookup. And that's the simplest base 0:36:37.54 form, base case of how this name resolution is done. 0:36:41.117 The second way of doing name resolution is something called a 0:36:45.256 path name resolution. We didn't see an example of 0:36:48.623 this today, but we saw an example of path name resolution 0:36:52.552 the last time. If you take a big UNIX path 0:36:55.428 name slash home slash foo slash bar, what we did was start 0:36:59.567 left to right along that path and added down our resolution of 0:37:03.846 the file to get to the block that we wanted while going 0:37:07.634 through a path, sorry, not a search path, 0:37:10.44 but going to the path that names the item. 0:37:15 And a third way of doing name resolution is what we saw today. 0:37:19.201 And that's an example of searching through contexts. 0:37:22.714 There's really no path here. I just tell you, 0:37:25.744 here's a set of dot O files, and here's a set of libraries. 0:37:29.739 And the symbols that are undefined are defined in 0:37:33.045 different modules of my program, or FO2 in different modules 0:37:37.316 of my program are defined somewhere among these modules. 0:37:42 And I'm not really going to tell the linker what's been 0:37:44.954 defined where. It's up to the linker to figure 0:37:47.416 it out, and it does that by basically 0:37:50.37 running a search through a variety of different contexts. 0:37:53.434 So, each dot O file and each library is a new context in 0:37:56.443 which a search for previously undefined symbol happens. 0:38:00 And in this particular case, the search within each context 0:38:03.879 takes the form of the table lookup, OK? 0:38:06.421 So, those are the three techniques for how you do name 0:38:09.966 resolution in general. And we saw two of them today, 0:38:13.377 and we saw two of them, the first two, 0:38:15.852 last time when we talked about the UNIX file system. 0:38:20 0:38:32 So the last step in the process of what a linker does after 0:38:35.093 symbol resolution and relocation, relocation in this 0:38:37.813 particular kind of linking, I didn't use the term, 0:38:40.426 but this form of linking is called static linking because 0:38:43.413 we're going to take all of these different object files and 0:38:46.506 defined on the command line or on the library, 0:38:48.906 and build together a single big binary that has been linked once 0:38:52.266 up front where the linker's called. 0:38:54.239 That's called static linking. Internal relocation in that 0:38:57.226 context is pretty straightforward. 0:39:00 But so we're not going to talk more about that except to note 0:39:04.043 that this relocation table is maintained in each object file. 0:39:08.086 But the third step is a little more slightly more complicated, 0:39:12.197 and actually varies a lot depending on the system. 0:39:15.5 And that's program loading. And the problem that solved by 0:39:19.341 loading is that the linker produces an output program 0:39:22.845 that's executable. And you can run that program. 0:39:26.147 And in UNIX, you run it by typing something on a command line. 0:39:31 Or in Windows, you click on something, 0:39:33.336 and effectively that causes a shell to execute a program. 0:39:36.873 So somebody has to do the work of when you type something on 0:39:40.599 the command line, somebody has to do the work of 0:39:43.568 looking at what file has been typed, taking the contents of 0:39:47.231 the file, loading it up into memory, passing control to 0:39:50.642 something that can then start running the program. 0:39:53.736 And typically the place where that control is passed is the 0:39:57.4 interpreter corresponding to the program. 0:40:01 So all of this work is done in UNIX by a program called 0:40:04.151 EXECVE. So the actual loader in UNIX is 0:40:06.544 a program called EXECVE. And its job, 0:40:08.645 once you type it on the command line, is the shell invokes it. 0:40:12.206 And what it does is to do what I said, which is look at the 0:40:15.591 file name, take the contents of it, load it up into memory, 0:40:18.918 and pass control to basically the first line of the program. 0:40:22.303 Often, modern object files are a little more complicated. 0:40:25.571 They actually have something that says who the interpreter of 0:40:29.073 the program is. So you pass control to that 0:40:32.955 interpreter, which in turn goes through a bunch of steps, 0:40:36.649 and then invokes the first line in main. 0:40:39.691 And that's what the C loader, the way in which loading 0:40:43.53 program that's written in C works. 0:40:45.92 So, so far what we've seen is, as I mentioned, 0:40:49.179 an example of linking called static linking where you take 0:40:53.308 all these object files and libraries, and extract the right 0:40:57.509 object files out of it and build a big program. 0:41:02 So, what you'll find is that even small programs like this 0:41:07.016 one, all it's doing is multiplying two numbers. 0:41:11.063 If I compile it -- 0:41:13 0:41:25 -- it's pretty big, almost 400 kB. 0:41:26.922 I mean, it's multiplying two numbers and it takes 400 kB to 0:41:30.302 multiply it. And the reason is that I made 0:41:32.691 the mistake of including, I have to show the output to 0:41:35.779 the user, so I call it print F. And print F happens with part 0:41:39.275 of a big object file that defines a lot of other things. 0:41:42.48 And that whole thing got built as part of the program. 0:41:45.568 Now, if you have a lot of programs, so it's not just that 0:41:48.831 the files are big. I mean, disks, 0:41:50.696 as a mentioned a couple lectures ago, 0:41:52.793 disks are cheap. And so that's not really the 0:41:55.357 problem as much. The problem is that this is the 0:41:58.096 entire program, so it has to get loaded in. 0:42:02 So if you have some machine on which a hundred processors run, 0:42:06.161 and each of those processors has print F's in a few places or 0:42:10.255 even one print F, each of those programs is going 0:42:13.53 to be extremely big. So the way in which you deal 0:42:16.805 with this problem is to do something that's pretty obvious 0:42:20.693 in retrospect. Why have multiple copies of the 0:42:23.764 same module? Why don't we just have one copy 0:42:26.697 of that module running and all the programs that use that 0:42:30.518 module? And that's done using an idea 0:42:32.974 called a shared object or shared modules. 0:42:37 And this stuff is extremely hot. 0:42:38.811 If you look at recent activity in almost every modular software 0:42:42.433 system like you look at Apache or you look at many database 0:42:45.822 systems, and also you look at new Linux, there's been a ton of 0:42:49.386 activity over the past five or six years on different ways of 0:42:52.892 optimizing things so that, the idea is a very old idea. 0:42:56.047 But there's still been a lot of activity making it efficient and 0:42:59.728 practical over the last five to ten years. 0:43:03 The problem with shared objects is the following. 0:43:05.891 And this has become a little more clear when we talk about 0:43:09.325 actually how something called an address space a couple lectures 0:43:13.12 from now. But the basic problem is that 0:43:15.409 instructions are associated with memory locations, 0:43:18.361 and they run each thing in the object file. 0:43:21.072 And the binary has an instruction location. 0:43:23.421 And data has a particular location associated with it as 0:43:26.734 well. And the problem is that when 0:43:29.821 you have two programs that each want to use a shared module, 0:43:33.658 unless you're really careful about how you design it, 0:43:37.04 it's going to be very hard for you to ensure that for both of 0:43:40.943 those programs, this module that's going to be 0:43:43.869 shared has exactly the same addresses because if you have in 0:43:47.707 one program the module being called from address 17, 0:43:51.024 and in another program the module is written up as address 255, 0:43:54.536 then that object cannot be shared, right? 0:43:57.138 It's two different objects. And that's what happens with 0:44:01.892 static linking. If you take the SQR.O module 0:44:04.799 and you include that in two different programs, 0:44:07.773 the addresses that get associated with it in the two 0:44:11.221 different programs are going to be completely different. 0:44:14.939 So one challenge, and a significant one in the 0:44:17.981 shared object is objects that are shared by different programs 0:44:22.104 running at the same time is to generate code that is what is 0:44:26.093 called position independent. So it doesn't have in it 0:44:30.623 anything that's different for the different programs. 0:44:34.141 And so this is called position independent code. 0:44:37.321 And so the idea is when you call the module on your 0:44:40.704 computer, the program counter is going to point to something. 0:44:44.763 And all of the addresses inside that module are going to have 0:44:48.822 addresses. Obviously, they're going to 0:44:51.325 have addresses, but they're going to be 0:44:53.896 relative to the program counter. So when you jump to a location, 0:44:58.158 I'm going to jump to 317. You're going to jump to 0:45:02.346 something that says five locations from where you are 0:45:05.39 now. So it's a kind of addressing 0:45:07.264 called PC relative addressing. 0:45:09.137 And that's not going to be the only thing that's used. 0:45:12.24 But by and large, a requirement for position 0:45:14.757 independent code is that all of the addressing be relative to, 0:45:18.329 say, the program counter. And once you have this kind of 0:45:21.549 position independent code, what you have to do in your 0:45:24.651 program, if you're using something like square root 0:45:27.579 program, let's say, OK, in the math library when 0:45:30.33 you do the linking, you don't have to include the 0:45:33.14 object file in which square root dot square root is defined. 0:45:38 All that your object file has to do now is at the time it was 0:45:42.21 linked, there is some library, runtime library, 0:45:45.438 that you know when you link the program has the definition of 0:45:49.649 square root. So what the program contains 0:45:52.456 when you run this F1 dot O, F2 dot O, all the way through 0:45:56.385 the library is it's just going to maintain a pointer, 0:46:00.035 a name actually. It's going to maintain a name 0:46:03.41 to where the square root function is defined. 0:46:05.549 It's going to be a filename. OK, and so this is why 0:46:07.98 sometimes when you type in a program, and somebody has 0:46:10.557 changed the configuration on your machine and built before, 0:46:13.377 and somebody changes the configuration, 0:46:15.225 sometimes you get an error message that while you're 0:46:17.705 running the program, you get an error message that 0:46:20.087 says some library dot SO not found. 0:46:21.74 It worked two days ago. It doesn't work now. 0:46:23.831 And the reason for that is somebody may have moved things 0:46:26.554 around and you get an error not when you compile the program, 0:46:29.471 but when you run the program. And many executions of program 0:46:34.122 may not actually trigger the error at all. 0:46:37.023 It may get triggered only when the actual object is needed. 0:46:41.127 And that's called dynamic linking. 0:46:43.462 And again, the way in which we do dynamic linking is to use the 0:46:47.849 same name and constants. Rather than embed the entire 0:46:51.528 object file corresponding to the module corresponding to a name 0:46:55.915 of a function that's being defined elsewhere, 0:46:59.028 maintain a reference to it. And load that reference up at 0:47:03.808 runtime. Now, in order to enable for 0:47:05.991 that object to be shared between different programs, 0:47:09.047 all of the addressing inside that has to be relative. 0:47:12.29 That is, it has to be independent of what the PC's 0:47:15.346 value for the starting point of that module is. 0:47:18.215 And that's called position dependent code. 0:47:20.772 And so, that's the basic story behind how linking works and 0:47:24.389 almost all software systems involving libraries and modules 0:47:28.006 ends up having a linking in it. I mentioned LATEC as an 0:47:32.615 example before. Even document systems have 0:47:35.163 linking in it. And it's a pretty fundamental 0:47:37.835

[illegible]
algorithm that we talked about. It's specific to new Linux, 0:47:41.439 but the basic idea is pretty common. 0:47:43.614 Now, what we'll see next time is that this way of modularizing 0:47:47.404 has a lot of nice properties, allows you to build big 0:47:50.636 programs, but at the same time has pretty bad fault isolation properties 0:47:54.364 that we'll talk about next time.