6.4.1 Shared Libraries

Click one of the buttons to take you to that part of the video.

Shared Libraries 0:00-1:40

In this lesson, we're going to discuss the concept of shared libraries. Understand that the applications that are running on the Linux system can share code elements. These shared code elements are called shared libraries.

This is actually really useful. Shared libraries make it such that software developers don't have to rewrite the same code every time they write a new program.

If you think about it, there are many functions that are commonly used across many different programs.

For example, the process for opening a file, saving changes to a file, and then closing that file after you're done saving changes to it will be the same no matter what application is being used. Doesn't matter if you're running a word processing application, or a spreadsheet, or an image editing application--the process for opening, saving, and closing a file is basically the same.

Without shared libraries the programmers who wrote these applications would have to include the code for completing the open, save, and close task within every single application that they write. Really, that would just be a waste of time and resources. Instead, with shared libraries, the software developers can focus on the software code elements within their application that are unique to that individual application.

For example, if the programmers are writing a word processing application, they write the word processing application; and the spreadsheet developer develops the spreadsheet application. Then, for any tasks that are common among many different applications--the shared task across many applications--they simply link their code to the pre-written code in a shared library. They don't really worry about rewriting the code.

Dynamic Shared Libraries 1:41-3:43

Now, using shared libraries has many benefits. Obviously, it speeds up the development of applications and it also makes the programs being written smaller and leaner. There are two different types of shared libraries on Linux.

The first type is called a dynamic shared library. Now, dynamic shared libraries exist as files in the Linux system. In this example we have a shared library 1, and a shared library 2. These are not real file names, it's just an example file name.

Now, when a programmer writes an application the programmer simply inserts links to the functions within these shared libraries in their program code. In this case we're using the write to disk function, and the read from disk function. These again are not real function names, they're just examples.

Then, when the program here--the applications here--actually run and we need to, in this case, write to disk or read from disk, these functions are called from within these two dynamic shared libraries.

This is the important thing to understand, these functions--read from disk and write to disk--are not actually integrated into the application itself, they stay over here in the shared libraries. The application just calls them from the shared library.

Therefore, in order for this to work we have to have a configuration file on the system that contains a list of all the installed dynamic shared library files and where they're stored in the file system. However, using dynamic shared libraries does have one weakness, and that is the fact that it creates dependencies. In other words, in order for this application to run, it's got to have access to this shared library file and this shared library file.

If something happens to these shared library files, let's say somebody actually deleted it, or they moved it, or they renamed it, or something like that happens such that these links are broken, then the application is going to error out--it's going to malfunction because it will not be able to get to the code that it needs to run.

Static Shared Library 3:44-9:10

Now there's a second type of shared library that's commonly used on Linux, and those are static shared libraries. As this name implies, static shared libraries are linked statically into the application itself when it's compiled. In essence, with a static library the actual code from the shared library for the functions that are called within the application itself are integrated directly into the application.

It's included into the application. This has some benefits and some weaknesses. Probably the key weakness is the fact that it makes the application larger. And as a result, you have code within applications that's redundant across many applications. It does have one key benefit, and that is the fact that the applications are independent.

Remember when we were talking about dynamic shared libraries that dependencies exist and, if a shared library is missing, then the application malfunctions. It can't get to the code it needs.

With a static shared library--because we've integrated the function directly into the application itself, compiled it right in there-- that application is not dependent upon that shared library being available in the file system of the Linux system where the application is running.

Which type of shared library is best? Well, it actually depends upon the situation. Understand that most of the applications that you're going to use on a Linux system day to day will use dynamic shared libraries. And this is done for a very particular reason. It's because it allows them to have a lot of functionality without a big foot print on your hard disk drive.

Because they use shared dynamic libraries, they don't use up a whole lot of disk space. However, be aware that there are many applications that have to use static libraries instead. This is especially true for applications that are designed to help you rescue a malfunctioning system.

Now, instead of linking to a dynamic shared library, which in the case of a system recovery may not be available, these applications are completely self-contained and can run independently in a very minimal Linux environment, which is what you want if you're troubleshooting a system that isn't working properly.

A good example of an application that uses static shared libraries is the stand alone shell, or sash. It's designed to help you rescue a Linux system that has gone belly up. And because it has many functions built into the shell itself it can run independently. It's not dependent upon the availability of dynamically linked shared libraries. This way if the partition where the shared libraries reside is toast, the program can still run and allow you to rescue the system.

Now shared library files use a special naming format in order to help you identify what kind of shared library it is. The syntax is shown here. It always starts with lib, and then it's followed by the name of the shared library itself, and then we identify its type.

The type part of the file name identifies whether the shared library is a static library or a dynamically shared library. If you see so in the type part of a file name, you know that file is a dynamic shared library. On the other hand, if you see an a in the type part of the file name, you know that is a static shared library.

Finally, we include in the file name, the version number. This simply identifies which version of the shared library the file is. An example is shown right here, libfreetype.so.6.4.0. This is an example of a dynamic shared library because it uses so in the type part of the file name.

As I talked about earlier, Linux uses a special configuration file, /etc/ld.so.conf, to tell the applications running on the system where they can find the dynamic shared library files that they are dependent upon. Using this type of configuration provides application developers with a lot of independence.

Basically, they don't have to worry about where those dynamic shared libraries will reside on the Linux system--where their programs are running. They basically just let the configuration file tell the application where those shared libraries are, wherever that happens to be on a particular Linux distribution.

Now this ld.so.conf file contains a list of paths in the file system where shared library files are stored. Any application that's linked to a function in one of these files can search through these paths to locate the appropriate library.

If you want to view a list of all the dynamic shared libraries that are available on your system, you use the ldconfig command. Type 'ldconfig -p' at the shell prompt and you see a list of all the different shared library files that are currently on your system.

Now, the ldconfig command displays all of the shared libraries on the system. There may be times when you need to view just the shared libraries that are required by a particular application running on your system.

In this case, you can use the ldd command instead. The syntax is to run ldd -v followed by the full path and file name of the executable in question. In this case we run ldd -v /bin/bash to see what shared libraries are available by the bash shell. Here you can see a list of all the shared libraries that are required in order for the bash shell to function properly.

Shared Library Dependencies 9:11-12:13

One of the most important uses of the ldd command is to check for shared library dependencies. Basically, it's going to determine whether all of the libraries required by the application have been installed or not. If they have, then you don't see any error messages.

But if a library file is missing, then an error message is going to be displayed. Then you can locate and install the missing library. When you do, all is well in the world.

How does an application know which directories to look in when they're trying to find a shared library? Understand that the applications running on your system don't actually look at the /etc/ld.so.conf file.

Instead, what they do is check the library cache, and they also look at the 'LD_LIBRARY_PATH' environment variable, which points to the library cache, which is /etc/ld.so.cache. This file contains a list of all the system libraries, and it's refreshed whenever the system is turned on and booted up.

Now, it's important to remember this fact because if you add a new directory to your file system that contains dynamic shared libraries, and you make the necessary entry in your ld.so.conf file, well you're going to have problems. When you try to run the applications that are linked to the libraries in that new directory, it's not going to work.

The reason for that is because the library cache has not been updated with the new path information. In order to fix this, you have the two options shown on this slide. The first one is to use the ldconfig command. The ldconfig command is used to rebuild the library cache manually. The other option is to set the value of your LD_LIBRARY_PATH environment variable.

What you'll probably want to do is add the new path to the dynamic shared library directory to the end of the existing list of directories that are already included within the environment variable.

Remember, if you just set the value of this environment variable to the new path, it will overwrite whatever paths that are already in here. And that's probably bad because you're going to break other applications in the process. They'll not be able to find their dynamic shared libraries.

What you want to do is preserve the existing directories within the path, and then just add a semicolon, and then the new path onto the end of the environment variable. Also remember that when you're done setting the value of the library path variable, you need to then export it so it will be usable across multiple shell sessions.

If the new directory containing dynamic libraries that you've added is one that you want to continue to use, then you should probably add the commands necessary to add it to your library path environment variable to one of your bash configuration files. That way, it will always be available--even if you reboot the system.

Generally speaking, the first option is probably the best one. This ensures that the shared libraries will always be available to the applications that need them, even if the system gets rebooted. Usually, I only set the value of the ld library path in situations where I have two versions of the same shared library installed within different directories, and I want to use one version instead of the other.

Summary 12:13-12:26

That's it for this lesson. In this lesson we discussed shared libraries. We discussed what a shared library is and how it works. We talked about how to view shared libraries that are required by an application. And then we ended this lesson by discussing how to manage shared libraries.
