PART IV. CREATING C OR C++ APPLICATIONS

Red Hat offers multiple tools for creating applications using the C and C++ languages. This part of the book lists some of the most common development tasks.

CHAPTER 15. BUILDING CODE WITH GCC

This chapter describes situations where source code must be transformed into executable code.

15.1. RELATIONSHIP BETWEEN CODE FORMS

Prerequisites

- Understanding of the concepts of compiling and linking

Possible Code Forms

When using the C and C++ languages, there are three forms of code:

- Source code written in the C or C++ language, present as plain text files.
  The files typically use extensions such as **.c**, **.cc**, **.cpp**, **.h**, **.hpp**, **.i**, **.inc**. For a complete list of supported extensions and their interpretation, see the gcc manual pages:

  ```
  $ man gcc
  ```

- Object code, created by *compiling* the source code with a *compiler*. This is an intermediate form.
  The object code files use the **.o** extension.

- Executable code, created by *linking* object code with a *linker*.
  Linux application executable files do not use any file name extension. Shared object (library) executable files use the **.so** file name extension.

NOTE

Library archive files for static linking also exist. This is a variant of object code which uses the **.a** file name extension. Static linking is not recommended. See Section 16.2, “Static and dynamic linking”.

Handling of Code Forms in GCC

Producing executable code from source code requires two steps, which require different applications or tools. GCC can be used as an intelligent driver for both compilers and linkers. This allows you to use a single command **gcc** for any of the required actions. GCC automatically selects the actions required (compiling and linking), as well as their sequence:

1. Source files are compiled to object files.

2. Object files and libraries are linked (including the previously compiled sources).

It is possible to run GCC such that only step 1 happens, only step 2 happens, or both steps 1 and 2 happen. This is determined by the types of inputs and requested types of output.

Because larger projects require a build system which usually runs GCC separately for each action, it is helpful to always consider compilation and linking as two distinct actions, even if GCC can perform both at once.

Additional Resources

- Section 15.2, “Compiling Source Files to Object Code”

- Section 15.6, “Linking Code to Create Executable Files”

15.2. COMPILING SOURCE FILES TO OBJECT CODE

To create object code files from source files and not an executable file immediately, GCC must be instructed to create only object code files as its output. This action represents the basic operation of the build process for larger projects.

Prerequisites

- C or C++ source code file(s)
- GCC installed on the system

Procedure

1. Change to the directory containing the source code file(s).
2. Run **gcc** with the **-c** option:

   ```
   $ gcc -c source.c another_source.c
   ```

   Object files are created, with their file names reflecting the original source code files: **source.c** results in **source.o**.

   NOTE

   With C++ source code, replace the **gcc** command with **g++** for convenient handling of C++ Standard Library dependencies.

Additional Resources

- Section 15.5, “Hardening Code with GCC”
- Section 15.4, “Code Optimization with GCC”
- Section 15.8, “Example: Building a C Program with GCC”

15.3. ENABLING DEBUGGING OF C AND C++ APPLICATIONS WITH GCC

Because debugging information is large, it is not included in executable files by default. To enable debugging of your C and C++ applications with it, you must explicitly instruct the compiler to create debugging information.

Enabling the Creation of Debugging Information with GCC

To enable the creation of debugging information with GCC when compiling and linking code, use the **-g** option:

```
$ gcc ... -g ...
```

- Optimizations performed by the compiler and linker can result in executable code which is hard to relate to the original source code: variables may be optimized out, loops unrolled, operations merged into the surrounding ones etc. This affects debugging negatively. For an improved debugging experience, consider setting the optimization with the **-Og** option. However, changing the optimization level changes the executable code and may change the actual behaviour so as to remove some bugs.

- The **-fcompare-debug** GCC option tests code compiled by GCC with debug information and without debug information. The test passes if the resulting two binary files are identical. This test ensures that executable code is not affected by any debugging options, which further ensures that there are no hidden bugs in the debug code. Note that using the **-fcompare-debug** option significantly increases compilation time. See the GCC manual page for details about this option.

Additional Resources

- Section 20.1, “Enabling Debugging with Debugging Information”
- Using the GNU Compiler Collection (GCC) — 3.10 Options for Debugging Your Program
- Debugging with GDB — 18.3 Debugging Information in Separate Files
- The GCC manual page:

```
$ man gcc
```

15.4. CODE OPTIMIZATION WITH GCC

A single program can be transformed into more than one sequence of machine instructions. An optimal result can be achieved if more resources are allocated for analysis of the code during compilation.

Code Optimization with GCC

With GCC, it is possible to set the optimization level using the **-O*level*** option. This option accepts a set of values in place of the *level*.

Level	Description
0	Optimize for compilation speed - no code optimization (default)
1, **2**, **3**	Increasing optimization effort for code execution speed
s	Optimize for resulting file size
fast	Level 3 plus disregard for strict standards compliance to allow for additional optimizations
g	Optimize for debugging experience

For release builds, the optimization option **-O2** is recommended.

During development, the **-Og** option is more useful for debugging the program or library in some situations. Because some bugs manifest only with certain optimization levels, ensure to test the program or library with the release optimization level.

GCC offers a large number of options to enable individual optimizations. For more information, see the following Additional Resources.

Additional Resources

- Using GNU Compiler Collection — 3.11 Options That Control Optimization
- Linux manual page for GCC:

  ```
  $ man gcc
  ```

15.5. HARDENING CODE WITH GCC

When the compiler transforms source code to object code, it can add various checks to prevent commonly exploited situations and thus increase security. Choosing the right set of compiler options can help produce more secure programs and libraries, without changes to the source code.

Release Version Options

The following list of options is the recommended minimum for developers targeting Red Hat Enterprise Linux:

```
$ gcc ... -O2 -g -Wall -Wl,-z,now,-z,relro -fstack-protector-strong -D_FORTIFY_SOURCE=2 ...
```

- For programs, add the **-fPIE** and **-pie** Position Independent Executable options.
- For dynamically linked libraries, the mandatory **-fPIC** (Position Independent Code) option indirectly increases security.

Development Options

The following options are recommended to detect security flaws during development. Use these options in conjunction with the options for the release version:

```
$ gcc ... -Walloc-zero -Walloca-larger-than -Wextra -Wformat-security -Wvla-larger-than ...
```

Additional Resources

- Defensive Coding Guide
- Red Hat Developer blog post — Memory Error Detection Using GCC

15.6. LINKING CODE TO CREATE EXECUTABLE FILES

Linking is the final step when building a C or C++ application. Linking combines all object files and libraries into an executable file.

Prerequisites

- One or more object files
- GCC is installed on the system

Procedure

1. Change to the directory containing the object code files.
2. Run **gcc**:

   ```
   $ gcc ... objectfile.o another_object.o ... -o executable-file
   ```

- An executable file named ***executable-file*** is created from the supplied object files and libraries.

  To link additional libraries, add the required options before the list of object files. See Chapter 16, *Using Libraries with GCC*.

NOTE

With C++ source code, replace the **gcc** command with **g++** for convenient handling of C++ Standard Library dependencies.

Additional Resources

- Section 15.8, “Example: Building a C Program with GCC”
- Chapter 16, *Using Libraries with GCC*

15.7. C++ COMPATIBILITY OF VARIOUS RED HAT PRODUCTS

The Red Hat ecosystem includes several versions of the GCC compiler and linker, provided in Red Hat Enterprise Linux and Red Hat Developer Toolset. The C++ ABI compatibility between these is as follows:

- The *system compiler* based on GCC 4.8 and provided directly as part of Red Hat Enterprise Linux 7 supports only compiling and linking the C++98 standard (also known as C++03), and its variant with GNU extensions.
- Any C++98-compliant binaries or libraries built explicitly with the options **-std=c++98** or **-std=gnu++98** can be freely mixed, regardless of the version of the compiler used.
- Using and mixing the C++11 and C++14 language versions is supported only when using compilers from Red Hat Developer Toolset and only when all C++ objects compiled with the respective flag have been built using the same major version of GCC.
- When linking C++ files built with both Red Hat Developer Toolset and Red Hat Enterprise Linux toolchain, prefer the Red Hat Developer Toolset version of the compiler and linker.
- The default setting for compilers in Red Hat Enterprise Linux 6 and 7 and Red Hat Developer Toolset up to 4.1 is **-std=gnu++98**. That is, C++98 with GNU extensions.
- The default setting for compilers in Red Hat Developer Toolset 6, 6.1, 7, 7.1, 8.0, 8.1, 9.0, 9.1, and 10 is **-std=gnu++14**. That is, C++14 with GNU extensions.

Additional Resources

- Application Compatibility GUIDE
- Knowledge base solution — What gcc versions are available in Red Hat Enterprise Linux?
- Red Hat Developer Toolset User Guide — C++ Compatibility

15.8. EXAMPLE: BUILDING A C PROGRAM WITH GCC

This example shows the exact steps to build a sample minimal C program.

Prerequisites

- Understanding the use of GCC

Steps

1. Create a directory **hello-c** and change to its location:

```
$ mkdir hello-c
$ cd hello-c
```

2. Create a file **hello.c** with the following contents:

```
#include <stdio.h>

int main() {
  printf("Hello, World!\n");
  return 0;
}
```

3. Compile the code with GCC:

```
$ gcc -c hello.c
```

   The object file **hello.o** is created.

4. Link an executable file **helloworld** from the object file:

```
$ gcc hello.o -o helloworld
```

5. Run the resulting executable file:

```
$ ./helloworld
Hello, World!
```

Additional Resources

- Section 18.2, “Example: Building a C Program Using a Makefile”

15.9. EXAMPLE: BUILDING A C++ PROGRAM WITH GCC

This example shows the exact steps to build a sample minimal C++ program.

Prerequisites

- Understanding the use of GCC
- Understanding the difference between **gcc** and **g++**

Procedure

1. Create a directory **hello-cpp** and change to its location:

```
$ mkdir hello-cpp
$ cd hello-cpp
```

2. Create a file **hello.cpp** with the following contents:

   ```
   #include <iostream>

   int main() {
     std::cout << "Hello, World!\n";
     return 0;
   }
   ```

3. Compile the code with **g++**:

   ```
   $ g++ -c hello.cpp
   ```

   The object file **hello.o** is created.

4. Link an executable file **helloworld** from the object file:

   ```
   $ g++ hello.o -o helloworld
   ```

5. Run the resulting executable file:

   ```
   $ ./helloworld
   Hello, World!
   ```

CHAPTER 16. USING LIBRARIES WITH GCC

This chapter describes how to use libraries in code.

16.1. LIBRARY NAMING CONVENTIONS

A special file name convention is used for libraries: A library known as foo is expected to exist as the file **lib*foo*.so** or **lib*foo*.a**. This convention is automatically understood by the linking input options of gcc, but not by the output options:

- When linking against the library, the library can be specified only by its name foo with the **-l** option as **-l*foo***:

  ```
  $ gcc ... -lfoo ...
  ```

- When creating the library, the full file name **lib*foo*.so** or **lib*foo*.a** must be specified.

Additional Resources

- Section 17.2, “The soname Mechanism”

16.2. STATIC AND DYNAMIC LINKING

Developers have a choice of using static or dynamic linking when building applications with fully compiled languages. This section lists the differences, particularly in the contexti of using the C and C++ languages on Red Hat Enterprise Linux. To summarize, Red Hat discourages the use of static linking in applications for Red Hat Enterprise Linux.

Comparison of static and dynamic linking

Static linking makes libraries part of the resulting executable file. Dynamic linking keeps these libraries as separate files.

Dynamic and static linking can be compared in a number of ways:

Resource use

Static linking results in larger executable files which contain more code. This additional code coming from libraries cannot be shared across multiple programs on the system, increasing file system usage and memory usage at run time. Multiple processes running the same statically linked program will still share the code.
On the other hand, static applications need fewer run-time relocations, leading to reduced startup time, and require less private resident set size (RSS) memory. Generated code for static linking can be more efficient than for dynamic linking due to the overhead introduced by position-independent code (PIC).

Security

Dynamically linked libraries which provide ABI compatibility can be updated without changing the executable files depending on these libraries. This is especially important for libraries provided by Red Hat as part of Red Hat Enterprise Linux, where Red Hat provides security updates. Static linking against any such libraries is strongly discouraged.
Additionally, security measures such as load address randomization cannot be used with a statically linked executable file. This further reduces security of the resulting application.

Compatibility

Static linking appears to provide executable files independent of the versions of libraries provided by the operating system. However, most libraries depend on other libraries. With static linking, this dependency becomes inflexible and as a result, both forward and backward compatibility is lost. Static linking is guaranteed to work only on the system where the executable file was built.

WARNING

Applications linking static libraries from the GNU C library (glibc) still require glibc to be present on the system as a dynamic library. Furthermore, dynamic library variant of glibc available at the application's run time must be a bitwise identical version of the one present while linking the application. As a result, static linking is guaranteed to work only on the system where the executable file was built.

Support coverage

Most static libraries provided by Red Hat are in the *Optional* channel and not supported by Red Hat.

Functionality

Some libraries, notably the GNU C Library (glibc), offer reduced functionality when linked statically. For example, when statically linked, glibc does not support threads and any form of calls to the **dlopen()** function in the same program.

As a result of the listed disadvantages, static linking should be avoided at all costs, particularly for whole applications and the glibc and libstdc++ libraries.

NOTE

The compat-glibc package is included with Red Hat Enterprise Linux 7, but it is not a run time package and therefore not required for running anything. It is solely a development package, containing header files and dummy libraries for linking. This allows compiling and linking packages to run in older Red Hat Enterprise Linux versions (using compat-gcc-* against those headers and libraries). For more information on use of this package, run: **rpm -qpi compat-glibc-***.

Reasons for static linking

Static linking might be a reasonable choice in some cases, such as:

- A library which is not enabled for dynamic linking
- Fully static linking can be required for running code in an empty chroot environment or container. However, static linking using the **glibc-static** package is not supported by Red Hat.

Additional Resources

- Red Hat Enterprise Linux 7: Application Compatibility GUIDE

16.3. USING A LIBRARY WITH GCC

A library is a package of code which can be reused in your program. A C or C++ library consists of two parts:

- The library code
- Header files

Compiling Code That Uses a Library

The header files describe the interface of the library: The functions and variables available in the library. Information from the header files is needed for compiling the code.

Typically, header files of a library will be placed in a different directory than your application's code. To tell GCC where the header files are, use the **-I** option:

```
$ gcc ... -Iinclude_path ...
```

Replace *include_path* with the actual path to the header file directory.

The **-I** option can be used multiple times to add multiple directories with header files. When looking for a header file, these directories are searched in the order of their appearance in the **-I** options.

Linking Code That Uses a Library

When linking an executable file, both the object code of your application and the binary code of the library must be available. The code for static and dynamic libraries is present in different forms:

- Static libraries are available as archive files. They contain a group of object files. The archive file has the file name extension **.a**.
- Dynamic libraries are available as shared objects. They are a form of executable file. A shared object has the file name extension **.so**.

To tell GCC where the archives or shared object files of a library are, use the **-L** option:

```
$ gcc ... -Llibrary_path -lfoo ...
```

Replace *library_path* with the actual path to the library directory.

The **-L** option can be used multiple times to add multiple directories. When looking for a library, these directories are searched in the order of their **-L** options.

The order of options matters: GCC cannot link against a library foo unless it knows the directory of this library. Therefore, use the **-L** options to specify library directories before using the **-l** options for linking against libraries.

Compiling and Linking Code Which Uses a Library in One Step

When the situation allows the code to be compiled and linked in one **gcc** command, use the options for both situations mentioned above at once.

Additional Resources

- Using the GNU Compiler Collection (GCC) — 3.16 Options for Directory Search
- Using the GNU Compiler Collection (GCC) — 3.15 Options for Linking

16.4. USING A STATIC LIBRARY WITH GCC

Static libraries are available as archives containing object files. After linking, they become part of the resulting executable file.

NOTE

Red Hat discourages the use of static linking for various reasons. See Section 16.2, “Static and dynamic linking”. Use static linking only when necessary, especially against libraries provided by Red Hat.

Prerequisites

- GCC is installed on your system
- Understanding of static and dynamic linking
- A set of source or object files forming a valid program, requiring some static library foo and no other libraries
- The foo library is available as a file **libfoo.a**, and no file **libfoo.so** is provided for dynamic linking.

NOTE

Most libraries which are part of Red Hat Enterprise Linux are supported for dynamic linking only. The steps below only work for libraries which are *not* enabled for dynamic linking. See Section 16.6, “Using Both Static and Dynamic Libraries with GCC” .

Procedure

To link a program from source and object files, add a statically linked library foo, which is named **libfoo.a**:

1. Change to the directory containing your code.
2. Compile the program source files with headers of the foo library:

   ```
   $ gcc ... -Iheader_path -c ...
   ```

   Replace *header_path* with the path to the directory containing the header files for the foo library.
3. Link the program with the foo library:

   ```
   $ gcc ... -Llibrary_path -lfoo ...
   ```

   Replace *library_path* with the path to the directory containing the file **libfoo.a**.
4. To execute the program, run:

   ```
   $ ./program
   ```

CAUTION

The **-static** GCC option related to static linking forbids all dynamic linking. Instead, use the **-Wl,-Bstatic** and **-Wl,-Bdynamic** options to control linker behavior more precisely. See Section 16.6, “Using Both Static and Dynamic Libraries with GCC”.

16.5. USING A DYNAMIC LIBRARY WITH GCC

Dynamic libraries are available as standalone executable files, required at both linking time and run time. They stay independent of your application's executable file.

Prerequisites

- GCC is installed on the system
- A set of source or object files forming a valid program, requiring some dynamic library foo and no other libraries
- The foo library available as a file *libfoo.so*

Linking a Program Against a Dynamic Library

To link a program against a dynamic library foo:

```
$ gcc ... -Llibrary_path -lfoo ...
```

When a program is linked against a dynamic library, the resulting program must always load the library at run time. There are two options for locating the library:

- Using an **rpath** value stored in the executable file itself
- Using the **LD_LIBRARY_PATH** variable at runtime

Using an **rpath** Value Stored in the Executable File

The **rpath** is a special value saved as a part of an executable file when it is being linked. Later, when the program is loaded from its executable file, the runtime linker will use the **rpath** value to locate the library files.

While linking with GCC, to store the path *library_path* as an **rpath**:

```
$ gcc ... -Llibrary_path -lfoo -Wl,-rpath=library_path ...
```

The path *library_path* must point to a directory containing the file *libfoo.so*.

CAUTION

There is no space after the comma in the **-Wl,-rpath=** option

To run the program later, execute:

```
$ ./program
```

Using the LD_LIBRARY_PATH Environment Variable

If no **rpath** is found in the program's executable file, the runtime linker will use the **LD_LIBRARY_PATH** environment variable. The value of this variable must be changed for each program according to the path where the shared library objects are located.

To run the program without **rpath** set, with libraries present in the *library_path*, execute:

```
$ export LD_LIBRARY_PATH=library_path:$LD_LIBRARY_PATH
$ ./program
```

Leaving out the **rpath** value offers flexibility, but requires setting the **LD_LIBRARY_PATH** variable every time the program is to run.

Placing the Library into the Default Directories

The runtime linker configuration specifies a number of directories as a default location of dynamic library files. To use this default behaviour, copy your library to the appropriate directory.

A full description of the dynamic linker behavior is out of scope for this document. For more information, see the following resources:

- Linux manual pages for the dynamic linker:

  ```
  $ man ld.so
  ```

- Contents of the **/etc/ld.so.conf** configuration file:

  ```
  $ cat /etc/ld.so.conf
  ```

- Report of the libraries recognized by the dynamic linker without additional configuration, which includes directories:

  ```
  $ ldconfig -v
  ```

16.6. USING BOTH STATIC AND DYNAMIC LIBRARIES WITH GCC

Sometimes it is required to link some libraries statically and some dynamically.

Prerequisites

- Understanding of static and dynamic linking

Introduction

gcc recognizes both dynamic and static libraries. When the **-l*foo*** option is encountered, gcc will first attempt to locate a shared object (a **.so** file) containing a dynamically linked version of the foo library, and then look for the archive file (**.a**) containing a static version of the library. Thus, the following situations can result from this search:

- Only the shared object is found and gcc links against it dynamically
- Only the archive is found and gcc links against it statically
- Both the shared object and archive are found; gcc selects by default dynamic linking against the shared object
- Neither shared object nor archive is found and linking fails

Because of these rules, the best way to select the static or dynamic version of a library for linking is having only that version found by gcc. This can be controlled to some extent by using or leaving out directories containing the library versions when specifying the **-L*path*** options.

Additionally, because dynamic linking is the default, the only situation where linking must be explicitly specified is when a library with both versions present should be linked statically. There are two possible solutions:

- Specifying the static libraries by file path instead of the **-l** option

- Using the **-Wl** option to pass options to the linker

Specifying the static libraries by file

Usually, gcc is instructed to link against a library *foo* with the **-l*foo*** option. However, it is possible to specify the full path to the file **lib*foo*.a** containing the library instead:

```
$ gcc ... path/to/libfoo.a ...
```

From the file extension **.a**, gcc will understand that this is a library to link with the program. However, specifying the full path to the library file is a less flexible method.

Using the -Wl option

The gcc option **-Wl** is a special option for passing options to the underlying linker. Syntax of this option differs from the other gcc options. The -Wl option is followed by a comma-separated list of linker options, while other gcc options require a space-separated list of options. The ld linker used by gcc offers the options **-Bstatic** and **-Bdynamic** to specify whether libraries following this option should be linked statically or dynamically, respectively. After passing **-Bstatic** and a library to the linker, the default dynamic linking behaviour must be restored manually for the following libraries to be linked dynamically with the **-Bdynamic** option.

To link a program, linking a library first statically (**libfirst.a**) and second dynamically (**libsecond.so**), run:

```
$ gcc ... -Wl,-Bstatic -lfirst -Wl,-Bdynamic -lsecond ...
```

NOTE

gcc can be configured to use linkers other than the default ld. The **-Wl** option applies to the gold linker, too.

Additional Resources

- Using the GNU Compiler Collection (GCC) — 3.15 Options for Linking
- Documentation for binutils 2.32 — 2.1 Command Line Options

CHAPTER 17. CREATING LIBRARIES WITH GCC

This chapter describes steps for creating libraries and explains the necessary concepts used by the Linux operating system for libraries.

17.1. LIBRARY NAMING CONVENTIONS

A special file name convention is used for libraries: A library known as foo is expected to exist as the file **lib*foo*.so** or **lib*foo*.a**. This convention is automatically understood by the linking input options of gcc, but not by the output options:

- When linking against the library, the library can be specified only by its name foo with the **-l** option as **-l*foo***:

  ```
  $ gcc ... -lfoo ...
  ```

- When creating the library, the full file name **lib*foo*.so** or **lib*foo*.a** must be specified.

Additional Resources

- Section 17.2, “The soname Mechanism”

17.2. THE SONAME MECHANISM

Dynamically loaded libraries (shared objects) use a mechanism called *soname* to manage multiple compatible versions of a library.

Prerequisites

- Understanding of dynamic linking and libraries
- Understanding of the concept of ABI compatibility
- Understanding of library naming conventions
- Understanding of symbolic links

Problem Introduction

A dynamically loaded library (shared object) exists as an independent executable file. This makes it possible to update the library without updating the applications that depend on it. However, the following problems arise with this concept:

- The identification of the actual version of the library
- The need for multiple versions of the same library present
- The signalling of ABI compatibility of each of the multiple versions

The soname Mechanism

To resolve this, Linux uses a mechanism called soname.

A library **foo** version *X.Y* is ABI-compatible with other versions with the same value of *X* in the version number. Minor changes preserving compatibility increase the number *Y*. Major changes that break compatibility increase the number *X*.

The actual library **foo** version *X.Y* exists as the file **libfoo.so.*x.y***. Inside the library file, a soname is recorded with the value **libfoo.so.x** to signal the compatibility.

When applications are built, the linker looks for the library by searching for the file **libfoo.so**. A symbolic link with this name must exist, pointing to the actual library file. The linker then reads the soname from the library file and records it into the application executable file. Finally, the linker creates the application so that it declares dependency on the library using the soname, not name or file name.

When the runtime dynamic linker links an application before running, it reads the soname from the executable file of the application. This soname is **libfoo.so.*x***. A symbolic link with this name must exist, pointing to the actual library file. This allows loading the library, regardless of the *Y* component of the version, because the soname does not change.

NOTE

The *Y* component of the version number is not limited to just a single number. Additionally, some libraries encode the version in their name.

Reading soname from a File

To display the soname of a library file **somelibrary**:

```
$ objdump -p somelibrary | grep SONAME
```

Replace *somelibrary* with the actual file name of the library you wish to examine.

17.3. CREATING DYNAMIC LIBRARIES WITH GCC

Dynamically linked libraries (shared objects) allow resource conservation through code reuse and increased security by easier updates of the library code. This section describes the steps to build and install a dynamic library from source.

Prerequisites

- Understanding of the soname mechanism
- GCC is installed on the system
- Source code for a library

Procedure

1. Change to the directory with library sources.

2. Compile each source file to an object file with the position-independent code option **-fPIC**:

   ```
   $ gcc ... -c -fPIC some_file.c ...
   ```

   The object files have the same file names as the original source code files, but their extension is **.o**.

3. Link the shared library from the object files:

   ```
   $ gcc -shared -o libfoo.so.x.y -Wl,-soname,libfoo.so.x some_file.o ...
   ```

   The used major version number is X and the minor version number is Y.

4. Copy the ***libfoo.so.x.y*** file to an appropriate location where the system's dynamic linker can find it. On Red Hat Enterprise Linux, the directory for libraries is **/usr/lib64**:

   ```
   # cp libfoo.so.x.y /usr/lib64
   ```

   Note that you need root permissions to manipulate files in this directory.

5. Create the symlink structure for the soname mechanism:

   ```
   # ln -s libfoo.so.x.y libfoo.so.x
   # ln -s libfoo.so.x libfoo.so
   ```

Additional Resources

- The Linux Documentation Project — Program Library HOWTO — 3. Shared Libraries

17.4. CREATING STATIC LIBRARIES WITH GCC AND AR

Creating libraries for static linking is possible through the conversion of object files into a special type of archive file.

NOTE

Red Hat discourages the use of static linking for security reasons. Use static linking only when necessary, especially against libraries provided by Red Hat. See Section 16.2, "Static and dynamic linking".

Prerequisites

- GCC and binutils are installed on the system
- Understanding of static and dynamic linking
- A source file with functions to be shared as a library

Procedure

1. Create intermediate object files with GCC.

   ```
   $ gcc -c source_file.c ...
   ```

   Append more source files as required. The resulting object files share the file name but use the **.o** file name extension.

2. Turn the object files into a static library (archive) using the **ar** tool from the **binutils** package.

   ```
   $ ar rcs libfoo.a source_file.o ...
   ```

   The file **libfoo.a** is created.

3. Use the **nm** command to inspect the resulting archive:

   ```
   $ nm libfoo.a
   ```

4. Copy the static library file to the appropriate directory.

5. When linking against the library, GCC will automatically recognize from the **.a** file name extension that the library is an archive for static linking.

```
$ gcc ... -lfoo ...
```

Additional Resources

- The Linux manual page for the **ar** tool:

```
$ man ar
```

CHAPTER 18. MANAGING MORE CODE WITH MAKE

The GNU Make utility, commonly abbreviated as Make, is a tool for controlling the generation of executables from source files. Make automatically determines which parts of a complex program have changed and need to be recompiled. Make uses configuration files called Makefiles to control the way programs are built.

18.1. GNU MAKE AND MAKEFILE OVERVIEW

To create a usable form (usually executable files) from the source files of a particular project, perform several necessary steps. Record the actions and their sequence to be able to repeat them later.

Red Hat Enterprise Linux contains GNU **make**, a build system designed for this purpose.

Prerequisites

- Understanding the concepts of compiling and linking

GNU **make**

GNU **make** reads Makefiles which contain the instructions describing the build process. A Makefile contains multiple *rules* that describe a way to satisfy a certain condition (*target*) with a specific action (*recipe*). Rules can hierarchically depend on another rule.

Running **make** without any options makes it look for a Makefile in the current directory and attempt to reach the default target. The actual Makefile file name can be one of **Makefile**, **makefile**, and **GNUmakefile**. The default target is determined from the Makefile contents.

Makefile Details

Makefiles use a relatively simple syntax for defining *variables* and *rules*, which consists of a *target* and a *recipe*. The target specifies the output if a rule is executed. The lines with recipes must start with the tab character.

Typically, a Makefile contains rules for compiling source files, a rule for linking the resulting object files, and a target that serves as the entry point at the top of the hierarchy.

Consider the following **Makefile** for building a C program which consists of a single file, **hello.c**.

```
all: hello

hello: hello.o
	gcc hello.o -o hello

hello.o: hello.c
	gcc -c hello.c -o hello.o
```

This specifies that to reach the target **all**, the file **hello** is required. To get **hello**, one needs **hello.o** (linked by **gcc**), which in turn is created from **hello.c** (compiled by **gcc**).

The target **all** is the default target because it is the first target that does not start with a period. Running **make** without any arguments is then identical to running **make all**, if the current directory contains this **Makefile**.

Typical Makefile

A more typical Makefile uses variables for generalization of the steps and adds a target "clean" which removes everything but the source files.

```
CC=gcc
CFLAGS=-c -Wall
SOURCE=hello.c
OBJ=$(SOURCE:.c=.o)
EXE=hello

all: $(SOURCE) $(EXE)

$(EXE): $(OBJ)
        $(CC) $(OBJ) -o $@

%.o: %.c
        $(CC) $(CFLAGS) $< -o $@

clean:
        rm -rf $(OBJ) $(EXE)
```

Adding more source files to such a Makefile requires adding them to the line where the SOURCE variable is defined.

Additional resources

- GNU make: Introduction — 2 An Introduction to Makefiles
- Chapter 15, *Building Code with GCC*

18.2. EXAMPLE: BUILDING A C PROGRAM USING A MAKEFILE

Build a sample C program using a Makefile by following the steps in the example below.

Prerequisites

- Understanding of Makefiles and **make**

Procedure

1. Create a directory **hellomake** and change to this directory:

   ```
   $ mkdir hellomake
   $ cd hellomake
   ```

2. Create a file **hello.c** with the following contents:

   ```
   #include <stdio.h>

   int main(int argc, char *argv[])
     { printf("Hello, World!\n");
     return 0;
   }
   ```

3. Create a file **Makefile** with the following contents:

   ```
   CC=gcc
   CFLAGS=-c -Wall
   SOURCE=hello.c
   ```

```
OBJ=$(SOURCE:.c=.o)
EXE=hello

all: $(SOURCE) $(EXE)

$(EXE): $(OBJ)
	$(CC) $(OBJ) -o $@

%.o: %.c
	$(CC) $(CFLAGS) $< -o $@

clean:
	rm -rf $(OBJ) $(EXE)
```

CAUTION

The Makefile recipe lines must start with the tab character. When copying the text above from the browser, you may paste spaces instead. Correct this change manually.

4. Run **make**:

```
$ make
gcc -c -Wall hello.c -o hello.o
gcc hello.o -o hello
```

This creates an executable file **hello**.

5. Run the executable file **hello**:

```
$ ./hello
Hello, World!
```

6. Run the Makefile target **clean** to remove the created files:

```
$ make clean
rm -rf hello.o hello
```

Additional Resources

- Section 15.8, “Example: Building a C Program with GCC”
- Section 15.9, “Example: Building a C++ Program with GCC”

18.3. DOCUMENTATION RESOURCES FOR MAKE

For more information about **make**, see the resources listed below.

Installed Documentation

- Use the **man** and **info** tools to view manual pages and information pages installed on your system:

```
$ man make
$ info make
```

Online Documentation

- The GNU Make Manual hosted by the Free Software Foundation
- The Red Hat Developer Toolset User Guide — GNU make

CHAPTER 19 SHARED LIBRARIES INTRODUCTION

Shared libraries are libraries that are loaded by programs when they start. When a shared library is installed properly, all programs that start afterwards automatically use the new shared library. It's actually much more flexible and sophisticated than this, because the approach used by Linux permits you to:

- update libraries and still support programs that want to use older, non-backward- compatible versions of those libraries;
- override specific libraries or even specific functions in a library when executing a particular program.
- do all this while programs are running using existing libraries.

19.1 **Conventions**

For shared libraries to support all of these desired properties, a number of conventions and guidelines must be followed. You need to understand the difference between a library's names, in particular its ``soname'' and ``real name'' (and how they interact). You also need to understand where they should be placed in the filesystem.

19.1.1 **Shared Library Names**

Every shared library has a special name called the ``soname''. The soname has the prefix ``lib'', the name of the library, the phrase ``.so'', followed by a period and a version number that is incremented whenever the interface changes (as a special exception, the lowest-level C libraries don't start with ``lib''). A fully-qualified soname includes as a prefix the directory it's in; on a working system a fully-qualified soname is simply a symbolic link to the shared library's ``real name''.

Every shared library also has a ``real name'', which is the filename containing the actual library code. The real name adds to the soname a period, a minor number, another period, and the release number. The last period and release number are optional. The minor number and release number support configuration control by letting you know exactly what version(s) of the library are installed. Note that these numbers might not be the same as the numbers used to describe the library in documentation, although that does make things easier.

In addition, there's the name that the compiler uses when requesting a library, (I'll call it the ``linker name''), which is simply the soname without any version number.
The key to managing shared libraries is the separation of these names. Programs, when they internally list the shared libraries they need, should only list the soname they need. Conversely, when you create a shared library, you only create the library with a specific filename (with more detailed version information). When you install a new version of a library, you install it in one of a few special directories and then run the program ldconfig(8). ldconfig examines the existing files and creates the sonames as symbolic links to the real names, as well as setting up the cache file /etc/ld.so.cache (described in a moment).

ldconfig doesn't set up the linker names; typically this is done during library installation, and the linker name is simply created as a symbolic link to the ``latest'' soname or the latest real name. I would recommend having the linker name be a symbolic link to the soname, since in most cases if you update the library you'd like to automatically use it when linking. I asked H. J. Lu why ldconfig doesn't automatically set up the linker names. His explanation was basically that you might want to run code using the latest version of a library, but might instead want *development* to link against an old (possibly incompatible) library.
Therefore, ldconfig makes no assumptions about what you want programs to link to, so installers must specifically modify symbolic links to update what the linker will use for a library.

Thus, `/usr/lib/libreadline.so.3` is a fully-qualified soname, which ldconfig would set to be a symbolic link to some realname like `/usr/lib/libreadline.so.3.0`. There should also be a linker name, `/usr/lib/libreadline.so` which could be a symbolic link referring to `/usr/lib/libreadline.so.3`

19.1.2 **Filesystem Placement**

Shared libraries must be placed somewhere in the filesystem. Most open source software tends to follow the GNU standards; for more information see the info file documentation at info:standards#Directory_Variables. The GNU standards recommend installing by default all libraries in /usr/local/lib when distributing source code (and all commands should go into
/usr/local/bin). They also define the convention for overriding these defaults and for invoking the installation routines.

The Filesystem Hierarchy Standard (FHS) discusses what should go where in a distribution (see http://www.pathname.com/fhs). According to the FHS, most libraries should be installed in /usr/lib, but libraries required for startup should be in /lib and libraries that are not part of the system should be in /usr/local/lib.

There isn't really a conflict between these two documents; the GNU standards recommend the default for developers of source code, while the FHS recommends the default for distributors (who selectively override the source code defaults, usually via the system's package management system). In practice this works nicely: the ``latest'' (possibly buggy!) source code that you download automatically installs itself in the ``local'' directory (`/usr/local`), and once that code has matured the package managers can trivially override the default to place the code in the standard place for distributions. Note that if your library calls programs that can only be called via libraries, you should place those programs in /usr/local/libexec (which becomes /usr/libexec in a distribution). One complication is that Red Hat-derived systems don't include /usr/local/lib by default in their search for libraries; see the discussion below about /etc/ld.so.conf. Other standard library locations include
/usr/X11R6/lib for X-windows. Note that /lib/security is used for PAM modules, but those are usually loaded as DL libraries (also discussed below).

19.2 **How Libraries are Used**

On GNU glibc-based systems, including all Linux systems, starting up an ELF binary executable automatically causes the program loader to be loaded and run. On Linux systems, this loader is named /lib/ld-linux.so.X (where X is a version number). This loader, in turn, finds and loads all other shared libraries used by the program.

The list of directories to be searched is stored in the file /etc/ld.so.conf. Many Red Hat- derived distributions don't normally include /usr/local/lib in the file /etc/ld.so.conf. I consider this a bug, and adding /usr/local/lib to /etc/ld.so.conf is a common ``fix'' required to run many programs on Red Hat-derived systems.

If you want to just override a few functions in a library, but keep the rest of the library, you can enter the names of overriding libraries (.o files) in /etc/ld.so.preload; these ``preloading'' libraries will take precedence over the standard set. This preloading file is typically used for emergency patches; a distribution usually won't include such a file when delivered.

Searching all of these directories at program start-up would be grossly inefficient, so a caching arrangement is actually used. The program ldconfig(8) by default reads in the file

/etc/ld.so.conf, sets up the appropriate symbolic links in the dynamic link directories (so they'll follow the standard conventions), and then writes a cache to /etc/ld.so.cache that's then used by other programs. This greatly speeds up access to libraries. The implication is that ldconfig must be run whenever a DLL is added, when a DLL is removed, or when the set of DLL directories changes; running ldconfig is often one of the steps performed by package managers when installing a library. On start-up, then, the dynamic loader actually uses the file /etc/ld.so.cache and then loads the libraries it needs.

By the way, FreeBSD uses slightly different filenames for this cache. In FreeBSD, the ELF cache is /var/run/ld-elf.so.hints and the a.out cache is /var/run/ld.so.hints. These are still updated by ldconfig(8), so this difference in location should only matter in a few exotic situations.

19.3 **Environment Variables**

Various environment variables can control this process, and there are environment variables that permit you to override this process.

19.3.1 **LD_LIBRARY_PATH**

You can temporarily substitute a different library for this particular execution. In Linux, the environment variable LD_LIBRARY_PATH is a colon-separated set of directories where libraries should be searched for first, before the standard set of directories; this is useful when debugging a new library or using a nonstandard library for special purposes. The environment variable LD_PRELOAD lists shared libraries with functions that override the standard set, just as /etc/ld.so.preload does. These are implemented by the loader /lib/ld- linux.so. I should note that, while LD_LIBRARY_PATH works on many Unix-like systems, it doesn't work on all; for example, this functionality is available on HP-UX but as the environment variable SHLIB_PATH, and on AIX this functionality is through the variable LIBPATH (with the same syntax, a colon-separated list).
LD_LIBRARY_PATH is handy for development and testing, but shouldn't be modified by an installation process for normal use by normal users; see ``Why LD_LIBRARY_PATH is Bad'' at http://www.visi.com/~barr/ldpath.html for an explanation of why. But it's still useful for development or testing, and for working around problems that can't be worked around otherwise. If you don't want to set the LD_LIBRARY_PATH environment variable, on Linux you can even invoke the program loader directly and pass it arguments. For example, the following will use the given PATH instead of the content of the environment variable LD_LIBRARY_PATH, and run the given executable:

```
/lib/ld-linux.so.2  --library-path  PATH  EXECUTABLE
```

Just executing ld-linux.so without arguments will give you more help on using this, but again, don't use this for normal use - these are all intended for debugging.

19.3.2 **LD_DEBUG**

Another useful environment variable in the GNU C loader is LD_DEBUG. This triggers the dl* functions so that they give quite verbose information on what they are doing. For example:

```
export
LD_DEBUG=files
command_to_run
```

displays the processing of files and libraries when handling libraries, telling you what dependencies are detected and which SOs are loaded in what order. Setting LD_DEBUG to
``bindings'' displays information about symbol binding, setting it to ``libs'' displays the library search paths, and setting ti to ``versions'' displays the version depdendencies.

Setting LD_DEBUG to ``help'' and then trying to run a program will list the possible options. Again, LD_DEBUG isn't intended for normal use, but it can be handy when debugging and testing.

19.3.3 **Other Environment Variables**

There are actually a number of other environment variables that control the loading process; their names begin with LD_ or RTLD_. Most of the others are for low-level debugging of the loader process or for implementing specialized capabilities. Most of them aren't well-documented; if you need to know about them, the best way to learn about them is to read the source code of the loader (part of gcc).

Permitting user control over dynamically linked libraries would be disastrous for setuid/setgid programs if special measures weren't taken. Therefore, in the GNU loader (which loads the rest of the program on program start-up), if the program is setuid or setgid these variables (and other similar variables) are ignored or greatly limited in what they can do. The loader determines if a program is setuid or setgid by checking the program's credentials; if the uid and euid differ, or the gid and the egid differ, the loader presumes the program is setuid/setgid (or descended from one) and therefore greatly limits its abilities to control linking. If you read the GNU glibc library source code, you can see this; see especially the files elf/rtld.c and sysdeps/generic/dl-sysdep.c. This means that if you cause the uid and gid to equal the euid and egid, and then call a program, these variables will have full effect. Other Unix-like systems handle the situation differently but for the same reason: a setuid/setgid program should not be unduly affected by the environment variables set.

19.4 **Creating a Shared Library**

Creating a shared library is easy. First, create the object files that will go into the shared library using the gcc -fPIC or -fpic flag. The -fPIC and -fpic options enable ``position independent code'' generation, a requirement for shared libraries; see below for the differences. You pass the soname using the -Wl gcc option. The -Wl option passes options along to the linker (in this case the -soname linker option) - the commas after -Wl are not a typo, and you must not include unescaped whitespace in the option. Then create the shared library using this format:

```
gcc  -shared  -Wl,-soname,your_soname  \
    -o  library_name  file_list  library_list
```

Here's an example, which creates two object files (a.o and b.o) and then creates a shared library that contains both of them. Note that this compilation includes debugging information (-g) and will generate warnings (-Wall), which aren't required for shared libraries but are recommended. The compilation generates object files (using -c), and includes the required -fPIC option:

```
gcc -fPIC -g -c -
Wall a.c gcc  -fPIC
-g  -c  -Wall  b.c
gcc  -shared  -Wl,-soname,libmystuff.so.1  \
    -o  libmystuff.so.1.0.1  a.o  b.o  -lc
```

Here are a few points worth noting:

- Don't strip the resulting library, and don't use the compiler option -fomit-frame- pointer unless you really have to. The resulting library will work, but these actions make debuggers mostly useless.
- Use -fPIC or -fpic to generate code. Whether to use -fPIC or -fpic to generate code is target-dependent. The -fPIC choice always works, but may produce larger code than - fpic (mnenomic to remember this is that PIC is in a larger case, so it may produce larger amounts of code). Using -fpic option usually generates smaller and faster code, but will have platform-dependent limitations, such as the number of globally visible symbols or the size of the code. The linker will tell you whether it fits when you create the shared library. When in doubt, I choose -fPIC, because it

always works.

- In some cases, the call to gcc to create the object file will also need to include the option ``-Wl,-export-dynamic''. Normally, the dynamic symbol table contains only symbols which are used by a dynamic object. This option (when creating an ELF file) adds all symbols to the dynamic symbol table (see ld(1) for more information). You need to use this option when there are 'reverse dependencies', i.e., a DL library has unresolved symbols that by convention must be defined in the programs that intend to load these libraries. For ``reverse dependencies'' to work, the master program must make its symbols dynamically available. Note that you could say ``-rdynamic'' instead of ``-Wl,export-dynamic'' if you only work with Linux systems, but according to the ELF documentation the ``-rdynamic'' flag doesn't always work for gcc on non-Linux systems.

During development, there's the potential problem of modifying a library that's also used by many other programs -- and you don't want the other programs to use the ``developmental''library, only a particular application that you're testing against it. One link option you might use is ld's ``rpath'' option, which specifies the runtime library search path of that particular program being compiled. From gcc, you can invoke the rpath option by specifying it this way:

```
-Wl,-rpath,$(DEFAULT_LIB_INSTALL_PATH)
```

If you use this option when building the library client program, you don't need to bother with LD_LIBRARY_PATH (described next) other than to ensure it's not conflicting, or using other techniques to hide the library.

19.5 **Installing and Using a Shared Library**

Once you've created a shared library, you'll want to install it. The simple approach is simply to copy the library into one of the standard directories (e.g., /usr/lib) and run ldconfig(8).

First, you'll need to create the shared libraries somewhere. Then, you'll need to set up the necessary symbolic links, in particular a link from a soname to the real name (as well as from a versionless soname, that is, a soname that ends in ``.so'' for users who don't specify a version at all). The simplest approach is to run:

```
ldconfig  -n  directory_with_shared_libraries
```

Finally, when you compile your programs, you'll need to tell the linker about any static and shared libraries that you're using. Use the -l and -L options for this.

If you can't or don't want to install a library in a standard place (e.g., you don't have the right to modify /usr/lib), then you'll need to change your approach. In that case, you'll need to install it somewhere, and then give your program enough information so the program can find the library... and there are several ways to do that. You can use gcc's -L flag in simple cases. You can use the ``rpath'' approach (described above), particularly if you only have a specific program to use the library being placed in a ``non-standard'' place. You can also use environment variables to control things. In particular, you can set LD_LIBRARY_PATH, which is a colon-separated list of directories in which to search for shared libraries before the usual places. If you're using bash, you could invoke my_program this way using:

```
LD_LIBRARY_PATH=.:$LD_LIBRARY_PATH    my_program
```

If you want to override just a few selected functions, you can do this by creating an overriding object file and setting LD_PRELOAD; the functions in this object file will override just those functions (leaving others as they were).

Usually you can update libraries without concern; if there was an API change, the library creator is supposed to change the soname. That way, multiple libraries can be on a single system, and the right one is selected for each program. However, if a program breaks on an update to a library that kept the same soname, you can force it to use the older library version by copying the old library back somewhere, renaming the program (say to the old name plus ``.orig''), and then create a small ``wrapper'' script that resets the library to use and calls the real (renamed) program. You could place the old library in its own special area, if you like, though the numbering conventions do permit multiple versions to live in the same directory. The wrapper script could look something like this:

```
#!/bin/sh
export
LD_LIBRARY_PATH=/usr/local/my_lib:$LD_LIBRARY_PATH  exec
/usr/bin/my_program.orig  $*
```

Please don't depend on this when you write your own programs; try to make sure that your libraries are either backwards-compatible or that you've incremented the version number in the soname every time you make an incompatible change. This is just an ``emergency'' approach to deal with worst-case problems.

You can see the list of the shared libraries used by a program using ldd(1). So, for example, you can see the shared libraries used by ls by typing:

```
ldd /bin/ls
```

Generally you'll see a list of the sonames being depended on, along with the directory that those names resolve to. In practically all cases you'll have at least two dependencies:

- /lib/ld-linux.so.N (where N is 1 or more, usually at least 2). This is the library that loads all other libraries.
- libc.so.N (where N is 6 or more). This is the C library. Even other languages tend to use the C library (at least to implement their own libraries), so most programs at least include this one.

Beware: do *not* run ldd on a program you don't trust. As is clearly stated in the ldd(1) manual, ldd works by (in certain cases) by setting a special environment variable (for ELF objects, LD_TRACE_LOADED_OBJECTS) and then executing the program. It may be possible for an untrusted program to force the ldd user to run arbitrary code (instead of simply showing the ldd information). So, for safety's sake, don't use ldd on programs you don't trust to execute.

19.6 **Incompatible Libraries**

When a new version of a library is binary-incompatible with the old one the soname needs to change. In C, there are four basic reasons that a library would cease to be binary compatible:

1. The behavior of a function changes so that it no longer meets its original specification,
2. Exported data items change (exception: adding optional items to the ends of structures is okay, as long as those structures are only allocated within the library).
3. An exported function is removed.
4. The interface of an exported function changes.

If you can avoid these reasons, you can keep your libraries binary-compatible. Said another way, you can

keep your Application Binary Interface (ABI) compatible if you avoid such changes. For example, you might want to add new functions but not delete the old ones.
You can add items to structures but only if you can make sure that old programs won't be sensitive to such changes by adding items only to the end of the structure, only allowing the library (and not the application) to allocate the structure, making the extra items optional (or having the library fill them in), and so on. Watch out - you probably can't expand structures if users are using them in arrays.
For C++ (and other languages supporting compiled-in templates and/or compiled dispatched methods), the situation is trickier. All of the above issues apply, plus many more issues. The reason is that some information is implemented ``under the covers'' in the compiled code, resulting in dependencies that may not be obvious if you don't know how C++ is typically implemented. Strictly speaking, they aren't ``new'' issues, it's just that compiled C++ code invokes them in ways that may be surprising to you. The following is a (probably incomplete) list of things that you cannot do in C++ and retain binary compatibility, as reported by Troll Tech's Technical FAQ:

1. add reimplementations of virtual functions (unless it it safe for older binaries to call the original implementation), because the compiler evaluates SuperClass::virtualFunction() calls at compile-time (not link-time).
2. add or remove virtual member functions, because this would change the size and layout of the vtbl of every subclass.
3. change the type of any data members or move any data members that can be accessed via inline member functions.
4. change the class hierarchy, except to add new leaves.
5. add or remove private data members, because this would change the size and layout of every subclass.
6. remove public or protected member functions unless they are inline.
7. make a public or protected member function inline.
8. change what an inline function does, unless the old version continues working.
9. change the access rights (i.e. public, protected or private) of a member function in a portable program, because some compilers mangle the access rights into the function name.

Given this lengthy list, developers of C++ libraries in particular must plan for more than occasional updates that break binary compatibility. Fortunately, on Unix-like systems (including Linux) you can have multiple versions of a library loaded at the same time, so while there is some disk space loss, users can still run ``old'' programs needing old libraries.