Static and Dynamic Libraries in Yocto

Presented by: Manjunath E L

What is Static Library and Dynamic Library?

Static Library:

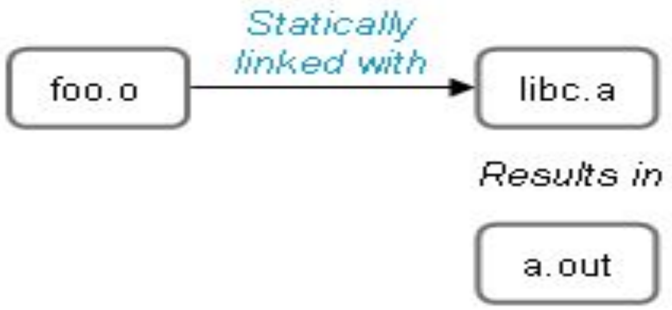
- A static library is a collection of object files (compiled code) bundled together into a single archive file.
- It is linked at compile time, and the entire library becomes part of the executable.
- Each program using a static library has its own copy of the library, resulting in larger executable sizes.
- Static libraries have the file extension ".a" on Unix-like systems or ".lib" on Windows.

Dynamic Library (Shared Library):

- A dynamic library is a compiled binary file that is linked at runtime when a program is executed.
- It is loaded into memory once and shared among multiple processes, reducing memory usage.
- Dynamic libraries are more flexible as they can be updated without recompiling the entire application.
- They have file extensions like ".so" (shared object) on Unix-like systems or ".dll" (dynamic-link library) on Windows.

Static Linking

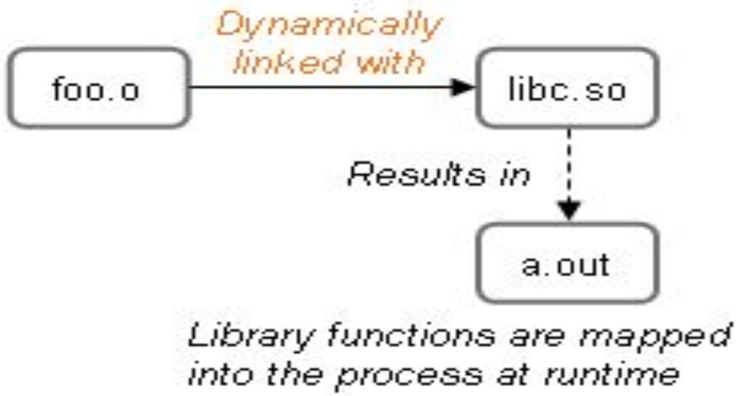
Static linking combines your work with the library into one binary.



The executable is statically linked because a copy of the library is physically part of the executable.

Dynamic Linking

Dynamic linking creates a combined work at runtime.



The executable is dynamically linked because it contains filenames that enable the loader to find the program's library references at runtime.

Why we need Static Library and Dynamic Library

Static Library:

Predictability: By including all necessary code during compile time, static libraries ensure predictable behavior as the executable doesn't rely on external factors that might vary.

Performance: Linking a static library at compile time can result in faster execution since the entire code is present in the executable, eliminating the need for dynamic linking at runtime.

Dynamic Library:

Resource Efficiency: Dynamic libraries reduce memory usage by loading into memory once and being shared among multiple processes. This is particularly beneficial when multiple applications use the same library.

Ease of Updates: Dynamic libraries can be updated independently of the applications using them. This facilitates bug fixes, security updates, or the introduction of new features without recompiling all dependent applications.

Creating the Static Library

A static library is basically a set of object files that were copied into a single file with the suffix .a

The basic tool used to create static libraries is a program called ar (archiver).

This program can be used to create static libraries (also known as archive files),modify object files in the static library, list the names of object files in the library, etc.

In order to create a static library, we have to perform two steps:

1. Create the object files from the source files of the project

2. Create the static library (the archive file) from the object files

Step 1: Creating the object file

The GCC compiler to create object files from the source files "arith.c" and "print.c". The "-c" option instructs GCC to compile the source files into object files without linking.

1. Compile "arith.c" into an object file:

```
$ gcc -c arith.c
```

   This command compiles the "arith.c" source file and generates an object file named "arith.o" (assuming the default output naming convention).

2. Compile "print.c" into an object file:

```
$ gcc -c print.c
```

   This command compiles the "print.c" source file and generates an object file named "print.o"

After running these commands, you will have two object files, "arith.o" and "print.o," which can later be linked together to create an executable or used in the building of a larger program.

Step 2:Create the static library

```
ar rcs libphy.a arith.o print.o
```

- **ar:** This is the archive command in Unix-like systems, and it's used for creating and maintaining archives, or libraries.
- **rcs:**
  - **r:** Replace or add the specified files to the archive. If the files already exist in the archive, they will be replaced with the new versions.
  - **c:** Create the archive if it doesn't already exist. This option ensures that the archive is created if it's not present.
  - **s:** Write an object-file index into the archive. This index helps speed up symbol-lookup operations when using the library.
- **libphy.a:** This is the name of the archive file that will be created or updated. The ".a" extension is a convention for static libraries.

- **arith.o print.o:** These are the object files that will be added to the library. The ar command combines these object files into a single archive file for ease of distribution and linking.

After running this command, you'll have a static library file named "libphy.a" containing the object files "arith.o" and "print.o." This library can then be linked with other programs to provide the functionality implemented in these object files.

Linking static library into application

$ gcc userprog.c -o userprog -lphy -L

The -L flag indicates (a non standard) directory where the libraries can be found, else you can copy this in standard location (/usr/lib)

The -l flag indicates the name of the library

Note the -larith will be converted to libarith.a by the compiler

$ gcc userprog.c -o userprog -I ../ -lphy -L ..

if don't want to mention -I and -L then copy mylib.h and libphy.a in std location

sudo cp ../mylib.h /usr/include/

sudo cp ../libphy.a /usr/lib/

$ gcc userprog.c -o userprog -lphy

Example of Static Library Recipe

```
DESCRIPTION = "Simple helloworld application"
LICENSE = "MIT"
LIC_FILES_CHKSUM = "file://${COMMON_LICENSE_DIR}/MIT;md5=0835ade698e0bcf8506ecda2f7b4f302"
SRC_URI = "file://print.c \
           file://arith.c \
           file://mylib.h"
S = "${WORKDIR}"
do_compile() {
         ${CC} -c print.c arith.c
         ${AR} rcs libphy.a print.o arith.o
do_install() {
        install -d ${D}${libdir}
        install -m 0755 libphy.a ${D}${libdir}
        install -d ${D}${includedir}
```

```
 install -m 0644 mylib.h ${D}${includedir}

}ALLOW_EMPTY:${PN}="1"
```

What is the purpose of ALLOW_EMPTY:${PN}='1' in a recipe ?

Answer:The ALLOW_EMPTY:${PN}=1 in a Yocto Project recipe is used to indicate that the recipe can be empty without causing a build error.

How to create the dynamic library

Step1: Create object files using the below command

```
$ gcc -fPIC -c print.c arith.c
```

The -fPIC flag stands for Position Independent Code, a characteristic required by shared libraries Position-independent code is code that can be executed at any memory address, which is necessary for shared libraries (dynamic link libraries) because they can be loaded at different memory locations in different processes.

Step2: Create the library

```
$ gcc -shared -Wl,-soname,libphy.so.1 -o libphy.so.1.0 *.o
```

The -shared key tells the compiler to produce a shared object which can then be linked with other objects to form an executable.

-Wl flag passes an options to linker with following format -Wl,options   in case of our example it sets the name of library, as it will be passed to the linker.

-Wl,-soname,libphy.so.1: This is a linker option (-Wl) passed to GCC. It specifies that the shared library should have a soname of libphy.so.1. The soname is used to identify the library and its version, allowing for version compatibility when linking against it.

1.Create a symbolic link named "libphy.so.1" pointing to "libphy.so.1.0":

```
ln -s libphy.so.1.0 libphy.so.1
```

- This command creates a symbolic link named "libphy.so.1" that points to the actual shared library file "libphy.so.1.0." Symbolic links help manage library versions and provide flexibility for dynamic linking.

2.Create another symbolic link named "libphy.so" pointing to "libphy.so.1":

```
ln -s libphy.so.1 libphy.so
```

- This command creates a symbolic link named "libphy.so" that points to "libphy.so.1." This link is often used by applications expecting a generic library name without a version suffix.

3.List all files in the current directory, including details about symbolic links:

```
ln -al
```

  - This command displays a detailed list of files in the current directory, including information about symbolic links such as their target and permissions.

After running these commands, you'll have two symbolic links, "libphy.so.1" and "libphy.so," pointing to the shared library file "libphy.so.1.0." These links facilitate dynamic linking by providing a consistent name for applications to use.

Step4: Compile the userprog.c using dynamic library (if it is present in same directory)

```
$ gcc userprog.c -o userprog -lphy -L.
$ ./userprog
```

Step5: Compile the userprog.c using dynamic library (if it present in different folder let say temp)

```
$ gcc userprog.c -o userprog -lphy -L.. -I..
$ export LD_LIBRARY_PATH=../:
```

Export: This command is used in Unix-like operating systems to set environment variables. The export keyword makes the variable available to child processes.

LD_LIBRARY_PATH=../: This sets the LD_LIBRARY_PATH variable to the parent directory (../). The dynamic linker will now look for shared libraries in the specified directory.

```
$ ./userprog
```

Shared Library Names

Dynamic libraries follow certain naming conventions on running systems so that multiple versions can co-exist.

Linux shared library can have three names. Which are:

Linker name (eg: libexample.so)

Soname (eg : libexample.so.1.2)

Real Name (eg : libexample.so.1.2.3)

Linker Name:

Linker Name is the name that is requested by the linker when another code is linked with your library (with –lexample linker option).

Linker name typically starts with the prefix lib name of the library the phrase .so

Soname:

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 Eg. libphy.so.1

Real Name: Real Name is the actual name of the shared library file.

Real Name = soname + minor version number

Eg. libphy.so.1.0

It can also be libphy.so.1.0.1.3

Command to read the soname

```
$ readelf -d libphy.so
```

- **readelf:** This is a command-line tool that displays information about ELF(Executable and Linkable Format) files.
- **-d:** This option specifies that you want to display the dynamic section of the ELF file.
- **libphy.so:** This is the name of the shared library for which you want to inspect the dynamic section.

Running the command readelf -d libphy.so will provide detailed information about the dynamic section of "libphy.so." This information typically includes details about shared library dependencies, version information, and various attributes related to dynamic linking.

Example of Dynamic Receipe:

```
DESCRIPTION = "Simple C prog using Dynamic Library"

LICENSE = "MIT"

LIC_FILES_CHKSUM = "file://${COMMON_LICENSE_DIR}/MIT;md5=0835ade698e0bcf8506ecda2f7b4f302"

SRC_URI = "file://print.c \

      file://arith.c \

      file://mylib.h"

S = "${WORKDIR}"

do_compile() {

     ${CC} -c -fPIC print.c arith.c

     ${CC} ${LDFLAGS} -shared -o libphy.so *.o}

do_install() {

          install -d ${D}${libdir}

          install -m 0755 libphy.so ${D}${libdir}

          install -d ${D}${includedir}

          install -m 0644 mylib.h ${D}${includedir}

}SOLIBS = ".so"

FILES_SOLIBSDEV = ""
```

Summary:

STATIC LIBRARIES

- Are part of the build environment
- Object files are added to the executable file
- Have the .a extension

ADVANTAGES

- Static libraries are faster, since all modules are in the same file
- Their distribution and installation is easier (all in the same file).
- Avoid the dependency hell.

DRAWBACKS

- Use more memory space: create a copy by each executable file.
- Slower compilation process: all source code is re-compiled.

DYNAMIC LIBRARIES

- Are part of the run-time environment
- Address of the object files are added to the executable file
- Have the .so extension

ADVANTAGES

- Use less memory space: only one copy to use multiple times.
- Source code is not re-compiled.
- Faster compilation process.

DRAWBACKS

- Could cause dependency problems in applications.
- Compatibility problems if the library is removed.

THANK YOU