Android Dynamic Linker - Marshmallow

WANG Zhenhua, i@jackwish.net

Abstract

Dynamic linker, links shared libraries together to be able to run, has been a fundamental mechanism in modern operating system and rich software ecosystem over the past decades. Dynamic linker is always highly platform-customized since it's coupled with binary format of a system. This article introduces the basic conception of dynamic linker and takes Android (Marshmallow) dynamic linker as example to discuss the implementation.

Introduction

Dynamic Link

Open any programming language book, "Hello World" is usually the first code example. The C example below which we are familiar with is related with dynamic link. The life cycle of this code is as Figure 1 (memory related part is in blue while symbol related is in purple).

```
#include<stdio.h>

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

As we know, functions need to be declared and defined before use. For the "Hello World" example, `printf()` is declared in `stdio.h` and the implementation is in shared library `libc.so`. The procedure of locating the declaration is *compiling* (pre-process more precisely) while locating the implementation is *linking*.

There are two categories of linking - *static linking* and *dynamic linking* - of which the difference is the time the *linking* procedure is performed, as Figure 1 demonstrated.

Static linking is performed by compiler tool-chain, `gcc example.c -static` on Linux is an example. A static linked binary runs without the need to relocate

symbol - `printf` is as its self-defined function. When the binary `hello.elf` is executed, operating system only needs to load it into memory.

Dynamic linking is performed at runtime by dynamic linker. At compile time, compiler tool-chain generates dynamic linked binary `hello.elf` which contains the information that it depends on `libc.so` for the implementation of `printf`. At runtime, dynamic linker loads `hello.elf`, reads the dependent data, loads `libc.so` into memory, and fills the address of `printf` into `hello.elf`. In this way, the `main` function can correctly calls `printf`.

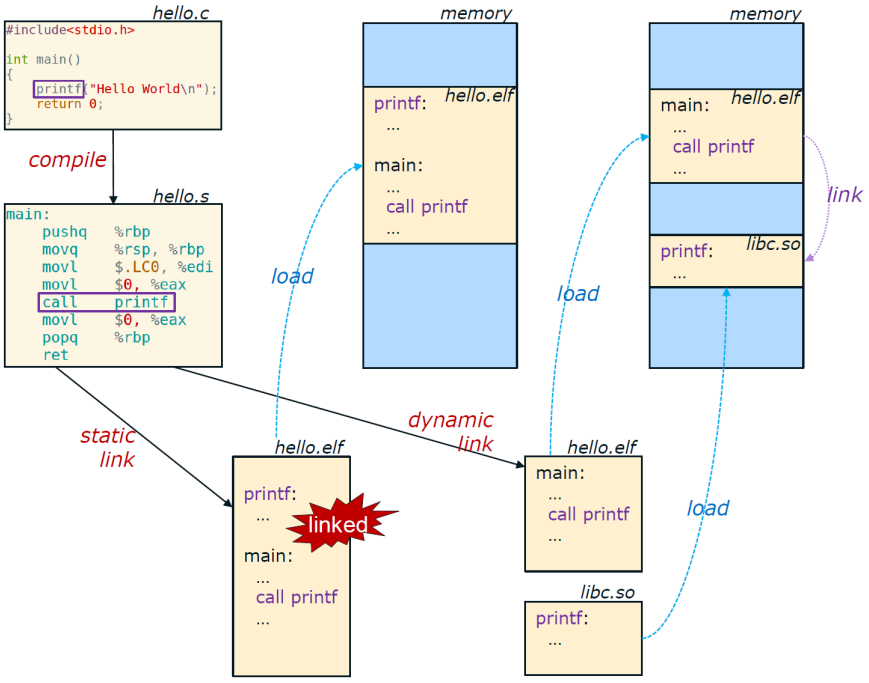

Figure 1: `printf` Example of Link

With the ability of dynamic linking, developers could create and share libraries. Library author could update internal implementation without need to inform users, while library users don't need to re-deploy their applications unless library interfaces have change. This is the infrastructure of API (Application Program Interface) !

Library Dependency

Diverse programs running on modern computer system are constructed by libraries to diminish development effort and memory and storage consumption.

Dynamic linker is responsible for locating libraries from storage, loading them into memory and linking the reference of them.

Developers create program or library based on different libraries. One typical dependency of library is as Figure 2 (`libcutils.so` of Android, `libdl.so` is ignored). A library author knows which libraries is depended upon by his library, and records the dependency in the library (`DT_NEEDED` table for ELF format library). At runtime, dynamic linker re-builds the dependency of an executable or shared library and links the binary against its dependency.

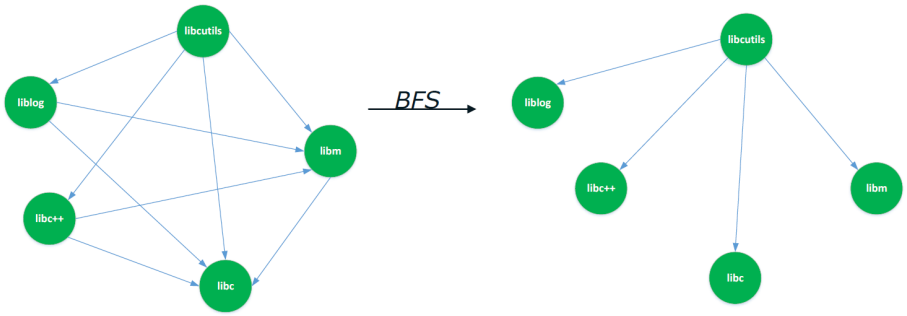

Figure 2: Dependency of `libcutils.so`

The dependency is mostly a DAG (Directed Acyclic Graph). For dynamic system supporting recursive dependent, the dependency could be a DCG (Directed Cyclic Graph). No matter what kind the graph is, dynamic linker can simply visit all nodes in the graph, locate, load and link them.

In a dependency graph, there is one and only one node which has no entry edge, called *root*. The re-building procedure of a dependency is traversing the graph in DFS (Depth First Search) or BFS (Breadth First Search) order starting with *root*. Figure 2 is a BFS example. In this article, our discussion is based on the BFS dependency.

Document Structure

We take the dynamic linker of Android as the implementation example. It is part of bionic, the standard C library developed by Google for its Android operating system. Android is based on Linux of which the shared library format is ELF. Dynamic linker provides linking service for Android system and applications deployed with JNI capability.

We firstly introduce the basic mechanism of dynamic linker of Android. Then talk about how special features is implemented. After that, the bootstrap of dynamic linker is discussed. And, at last, coming to the tricky part of *library dependency*.

Dynamic Linking Mechanism

In Android, dynamic linker is invoked when `System.loadLibrary()` is executed in Java or `dlopen()` is executed in native code. For Java code, Dalvik/Android-runtime calls into dynamic linker just like `dlopen()` eventually.

Starting with Lollipop MR1 (Our discussion is based on Marshmallow), Android dynamic linking is two-phase: library loading and library relocation. As Figure 3 shows, the left half is loading while the right half is linking.

During the library loading procedure, dynamic linker re-builds the *library dependency*, loads all libraries of it into memory. The library relocation procedure links the dependency. We talk about the important data structure of Android dynamic linker firstly.

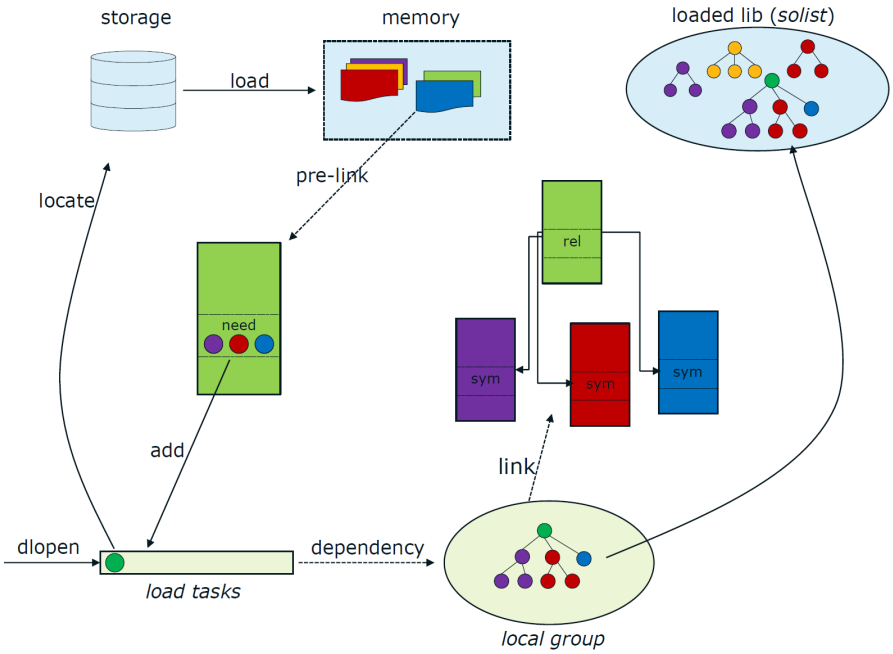

Figure 3: Workflow of Android Dynamic Linker

Data Structure

Persistent Data Structure

Dynamic linker of Android has two persistent data structure during the lifetime of an application/program - *LSPath* (Library Search Paths) and *ALList* (Already-loaded Library List).

LSPath are the directories where libraries are stored. Dynamic linker traverses these paths to hunt for a library. These paths are critical to the library locating and are sequenced in priority.

ALList is a list of *soinfo* which is used to maintain metadata of loaded libraries (ELF data and memory layout for example). Dynamic linker obtains data from *ALList* across different library loading and linking. *ALList* grows and shortens when library is loading and unloading respectively.

Persistent data structures are at the top side of Figure 3. "Storage" in Figure 3 means *LSPath* somehow.

Temporary Data Structure

Naturally, many temporary data structures are used during loading library. Among them, the most important two are *load_tasks* and *local_group*. Both of them present the *library dependency*.

load_tasks is a queue containing the libraries to be loaded - a subset libraries of *library dependency* which have NOT been loaded into memory yet. *load_tasks* dequeues when linker begins to search a library and enqueues the dependent libraries just parsed (from `DT_NEEDED` table of an ELF format library).

After all the *tree* has been loaded (*load_tasks* is empty at this time), ***local_group*** is constructed and used for relocation. *local_group* is a queue of *soinfo* and represents *library dependency* in BFS order. (Another similar data structure called *global_group* will be discussed in "Special Features" section.)

Temporary data structures are listed at the bottom side of Figure 3.

Library Loading Procedure

At the beginning, the library requested by operating system - *root* - is added to *load_tasks*, as Figure 3. During the procedure of library loading, dynamic linker continually loads all libraries in *load_tasks* and updates it if necessary, as the left half part of Figure 3. All libraries in the *library dependency* will be loaded when this procedure is finished.

Library Locating

Dynamic linker extracts one name/path from *load_tasks*, and opens directly if it's absolute path or traverses *LSPath* to hunt for the library otherwise.

When the library is located and opened, it could be a *system library* or a *app library*. *system library* are libraries loaded from system library paths

like `/system/lib`; *app library* are loaded from application library paths like `/data/data/com.example.app/lib`.

Before application is forked from Zygote, dynamic linker only searches for library under system library paths. After application is forked and library paths are set, dynamic linker searches under application paths firstly, then system library paths.

Library Loading

After library is opened from storage, and before loads the library into memory, linker wants to verify whether the file just opened is a valid shared library. It performs check based on ELF data: magic number, 32/64 bit, little/big endian, machine type and so on. If anything wrong, this library and the *library dependency* will be unloaded.

If validation passes, dynamic linker reads the library header and loads all loadable segments into memory. It calculates the needed memory size of the library by checking `PT_LOAD` tables of program header. The memory allocation is simply via `mmap`. (In and before *Jelly Bean*, the library memory is managed by a buddy memory allocation system system)

Library *Pre-link*

"Pre-link" intends to build one more level of *library dependency* by reading the dependency (dynamic `DT_NEEDED` section) of a library. All library names recorded in `DT_NEEDED` table are added to *load_tasks* and to be loaded.

It's easily to see that same library (name) may be added to *load_tasks* many times when loads a library. Dynamic linker traverses *ALList* to check if the library has already be loaded into memory by name and i-node, before and after open that library. If found, dynamic linker drops that *load_tasks* node and fetches next. So there are no duplicate loaded libraries in *ALList*.

The occurrence time of reading dependency of a library has changed across the development of Android. Before *Lollipop-MR1*, library linking is DFS which loads and links the *library dependency* recursively. Beginning with *Lollipop-MR1*, library linking changes to BFS. This change makes the library linking a two-stage procedure, all libraries in a *library dependency* has been loaded into memory before any of them has been relocated.

Library Relocation Procedure

After library loading procedure, the dependent relationship of libaries are recorded in *soinfo*. Dynamic linker reads *soinfo* beginning with *root* to build

local_group. Relocation is performed on *local_group*. Main loop of relocation dequeues a library from *local_group* and relocates it. *local_group* is BFS built, so the relocation is BFS too.

When resolving a symbol of a library, dynamic linker walks the *Relocation Section*, a table of all things needed to be relocated(`DT_REL` or `DT_RELA` of ELF), of a shared library. For each relocation entry, linker reads the symbol index and converts it to symbol name. With the name, linker searches the definition of it in the dependency tree - begin with the library itself, then *global_group*(see "Extension of Dynamic Linker") and *local_group*. When searches a symbol definition in a library, dynamic linker checks the symbol table(`DT_SYMTAB` of ELF) of it. There is acceleration methods for the table lookup, `DT_HASH` of ELF is a hash list which contains all the "exported" or "imported" symbol of a library.

The library relocation procedure is intuitive. When it's done, dynamic linker calls all library constructors in the dependency. After constructors are finished, the library is loaded, dynamic linker returns a handler of this library to user.

Extension of Dynamic Linker

Dynamic linking has some extensions to support various scenarios, and Android has extended dynamic linking functions for specific purpose.

Generic Dynamic Link

Global Library

When a library is declared as a "global library", loaded with the flag `RTLD_GLOBAL`, the definition of the symbol of the library has the highest priority for all libraries loaded after it.

Android dynamic linker builds the *global_group* every time at the beginning when load a library. When relocating a symbol, the *global_group* is firstly looked up - "global library" can overlays the symbol definition of the libraries to be loaded afterwards.

Preload Library

When a binary executed with flag `LD_PRELOAD`. These libraries will be loaded before the binary is really executed. Android dynamic linker preloads these libraries when it initializes. These libraries will carry the flag `RTLD_GLOBAL`. After that, "preload library" is just like "global library". `LD_PRELOAD` only takes effect for pure native programs.

Android Extended Dynamic Link

Android system extends dynamic linking to improve experience when loads libraries from both Java and native. The API is `android_dlopen_ext()`. Till M, features of this extension is as below, most of them are pretty easy to understand. Just copy from the source file....

The data structure of this extension is like this:

```
typedef struct {
  uint64_t flags;
  void*   reserved_addr;
  size_t  reserved_size;
  int     relro_fd;
  int     library_fd;
  off64_t library_fd_offset;
} android_dlextinfo;
```

Library Memory Related

`ANDROID_DLEXT_RESERVED_ADDRESS`: When set, the `reserved_addr` and `reserved_size` fields must point to an already-reserved region of address space which will be used to load the library if it fits. If the reserved region is not large enough, the load will fail.

`ANDROID_DLEXT_RESERVED_ADDRESS_HINT`: As DLEXT_RESERVED_ADDRESS, but if the reserved region is not large enough, the linker will choose an available address instead.

Library Opening Related

`ANDROID_DLEXT_USE_LIBRARY_FD`: Instruct dlopen to use `library_fd` instead of opening file by name. The filename parameter is still used to identify the library.

`ANDROID_DLEXT_USE_LIBRARY_FD_OFFSET`: If opening a library using `library_fd` read it starting at `library_fd_offset`. This flag is only valid when `ANDROID_DLEXT_USE_LIBRARY_FD` is set.

`ANDROID_DLEXT_FORCE_LOAD`: When set, do not check if the library has already been loaded by file stat(2)s. This flag allows forced loading of the library in the case when for some reason multiple ELF files share the same filename (because the already-loaded library has been removed and overwritten, for example). Note that if the library has the same dt_soname as an old one and some other library has the soname in `DT_NEEDED` list, the first one will be used to resolve any dependencies.

Library Relocation Related

`ANDROID_DLEXT_WRITE_RELRO`: When set, write the GNU RELRO section of the mapped library to `relro_fd` after relocation has been performed, to allow it to be reused by another process loading the same library at the same address. This implies `ANDROID_DLEXT_USE_RELRO`.

`ANDROID_DLEXT_USE_RELRO`: When set, compare the GNU RELRO section of the mapped library to `relro_fd` after relocation has been performed, and replace any relocated pages that are identical with a version mapped from the file.

Bootstrap of Dynamic Linker

Dynamic linker is designed to "link" all relocatable binaries, and must make itself looks like `libdl.so` to relocatables - the `libdl.so` binary is just a dummy library which makes `ld` of compiler tool-chain happy. Dynamic linker is statically linked at compile time and doesn't depend on any other resources except system call. The self-relocating and faking `libdl.so` is *Bootstrap*.

The bootstrap of Android dynamic linker is divided into two steps:

1. Initialize: hard coded to relocate linker itself.
2. Post-initialize: prepare "linker runtime" for library loading.

Initialize

During this stage, all executed code is statically relocated. No extern variable, extern function, or GOT access. Called from `begin.S` and will call Post-initialize functions afterwards. Primary operations are relocating linker itself and creating the dummy `libdl.so` *soinfo*.

Relocating linker itself is a sad story, every thing is hand-obtained. After the *soinfo* is well-setuped (memory related mostly), the real relocation is conducted. Then constructors of linker are called to initialize linker's global variables.

Creating dummy `libdl.so` *soinfo* is mainly set and update the reference of the *soinfo* to hard-coded array, symbol table for example. This *soinfo* node of `libdl.so` is always the first node of *ALList*.

With these work done, linker is relocated.

Post-initialize

After self-relocated, dynamic linker relocates *somain* - the main process - Zygote.

Before relocates Zygote, linker asks for runtime variable from system like `LD_LIBRARY_PATH` and `LD_PRELOAD`. And then, it relocates Zygote. With Zygote relocated, load all libraries declared in `LD_PRELOAD`. With everything done, linker finishes *Bootstrap* and jumps to Zygote.

Library Dependency

As discussed in the beginning, one task of dynamic linker is to re-build *library dependency*. The re-building procedure is sensitive to runtime environment in some corner scenario.

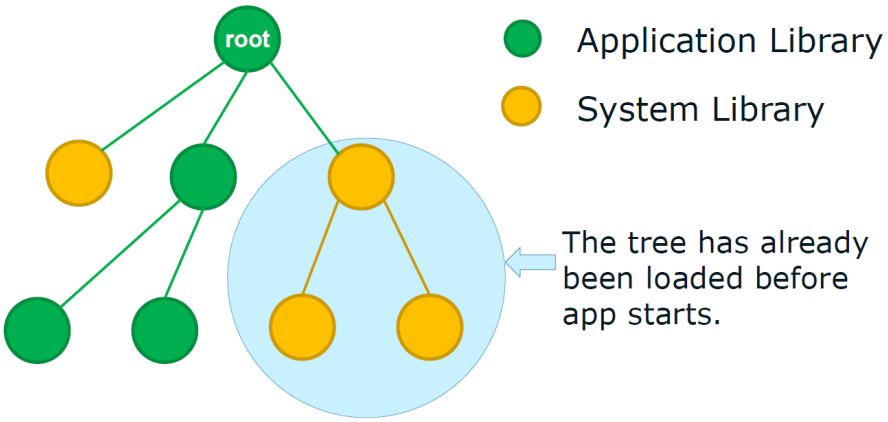

Figure 4: Tricky Library Depdendency Generation of Android

Consider that there are two sets of libraries - *set1* and *set2*. Some libraries in these two sets share same name but have different definitions. At the beginning, only *set1* can be loaded, and then both *set1* and *set2* can be loaded. The trick is, in any dependency, the library loaded in phase 1 could only depends on library in *set1*, as Figure 4. This is because whenever library in *set1* is needed, dynamic linker simply reuses the *soinfo* of it.

`LD_PRELOAD` in traditional Linux and libraries loaded before Zygote forks in Android are such scenarios. This is fine for most developers, but could impact some emulation system.

Summary

Dynamic linker re-builds the dependency of executables, locates, loads and links it. It's fundamental infrastructure of modern operating system and sensitive to running environment. Dynamic linking is usually high platform customized and requires *bootstrap*.

Android N includes namespace changes to prevent loading of non-public APIs. This feature heavily impacts the ecosystem of Android. In theory, *namespace* enables "virtualization" in dynamic linking. The dynamic linking we discussed in this document in "process internal", while *namespace* can build several virtual space - *namespace* - for dynamic linking in one process, making the dynamic link "namespace internal". We will refer to *namespace* in the future.