



connect

San Francisco 2015

# SFO15-310: Advanced Toolchain Usage Parts 5 & 6

**Presented by**

Adhemerval Zanella,  
Maxim Kuvyrkov, Ryan S. Arnold

**Date**

Wednesday 23 September 2015

**Event**

SFO15

Adhemerval Zanella,  
Maxim Kuvyrkov, Ryan S. Arnold

# Overview

- “Advanced Toolchain Usage” is a series of presentations given by Linaro toolchain experts on toolchain black-magic
- Continuation of “Advanced Toolchain Usage” series
  - <http://www.slideshare.net/linaroorg/lcu14-307-advanced-toolchain-usage-parts-1>
  - <http://www.slideshare.net/linaroorg/hkg15207-advanced-toolchain-usage-part-3>
- Suggestions for topics to cover or extend on in future presentations are welcome!

[linaro-toolchain@linaro.org](mailto:linaro-toolchain@linaro.org)



connect

San Francisco 2015

## Part 5

- Search Paths
- Symbol Versioning

## Part 6

- Timers, timestamps and the VDSO
- What the compiler can't know
- Out-of-order vs in-order execution

# Dynamic-Linker vs Static-Linker

## Static-linker

- Also called the *link-editor*, or just *the linker*.
- *binutils ld* or *gold* in the GNU toolchain
- LLD in the LLVM toolchain.
- This resolves most/many relocations at link-time and combines relocatable objects into shared-objects and/or executables, constructs the ELF object files, etc.

## Dynamic-linker

- Also called *the loader*.
- *ld.so*, or *ld-linux.so* on GNU/Linux systems.
- There is no LLVM equivalent.
- This loads libraries, resolves symbols in remote libraries, resolves a few relocations, etc., all at run-time.

# Search Paths

- The term *search paths* refers to how the dynamic-linker searches for dependencies (shared-objects) at run-time.
- You can see where the dynamic-linker looks for libraries for a particular program using LD\_DEBUG, e.g., LD\_DEBUG=libs ./foo

# Dynamic-Linker Search Order

- LD\_PRELINK (circumvents searching)
- DT\_RPATH dynamic array tag of library.
- DT\_RPATH dynamic array tag of executable.
- LD\_LIBRARY\_PATH environment variable.
- DT\_RUNPATH dynamic array tag of each individual shared object or executable
- /etc/ld.so.cache
- base library directories
  - /lib/[hwcap] and /usr/lib/[hwcap]
  - /lib and /usr/lib

# The setup

- We have a .c file, foo.c which calls a function bar () that is in the shared object libbar.so.
- foo.c is located at /home/ryanarn/foo.c
- libbar.so is located at /home/ryanarn/libs/libbar.so

# Using -L

- Since the shared library, `libbar.so`, is in `/home/ryanarn/libs/` we need to tell the static linker where to find it when we compile `foo.c` and link it to `libbar.so` so that the symbols are resolved during linking:

```
$ gcc -L /home/ryanarn/libs foo.c -o foo -lbar
```



connect

San Francisco 2015

# LD\_LIBRARY\_PATH

- When we try to execute the previous example we get the following:

```
./foo: error while loading shared libraries: libbar.so: cannot open  
shared object file: No such file or directory
```

- Since the shared library is in `/home/ryanarn/libs/` we need to tell the dynamic linker where to find it when the application is loaded and dependent shared objects are resolved.

```
$ export LD_LIBRARY_PATH=/home/ryanarn/libs:$LD_LIBRARY_PATH
```

- Relying on setting the `LD_LIBRARY_PATH` env variable is a support nightmare.



connect

San Francisco 2015

# DT\_RPATH Dynamic Array Tag

- We can use the `-rpath` linker option to get around having to use `LD_LIBRARY_PATH` by setting the `DT_RPATH` dynamic array tag:

```
-rpath=dir
```

```
Add a directory to the runtime library search path.
```

- What's this mean?

*Libraries in a fixed location can have the path to them embedded in a shared-object or executable directly at link time so the dynamic-linker knows where to find the library at runtime.*

# DT\_RPATH Dynamic Array Tag (cont)

- Let's use the `-rpath` linker option in our example:

```
$ gcc -L/home/ryanarn/libs/ -Wl,-rpath=/home/ryanarn/libs -o foo foo.c -lfoo
```

- It gives us the following dynamic array tag:

```
$ objdump -x test | grep PATH
RPATH                /home/ryanarn/libs/
```

- This is useful if your product builds several shared libraries and you know ahead of time where they're located.



connect

San Francisco 2015

# \$ORIGIN with DT\_RPATH

- We can use DT\_RPATH in a way that doesn't require absolute paths:

```
$ gcc -L/home/ryanarn/libs/ -Wl,-rpath='$ORIGIN/libs' -o foo foo.c -lbar
```

- This will give us:

```
$ objdump -x hello | grep PATH
RPATH                $ORIGIN/libs
```

- This will direct the dynamic linker to look for `libbar.so` in the `libs/` directory relative to the directory where the executable is located.

# LD\_RUN\_PATH Environment Variable

- LD\_RUNPATH=/home/ryanarn/libs is equivalent to using -Wl,-rpath=/home/ryanarn/libs

```
$ export LD_RUN_PATH=/home/ryanarn/libs
```

```
$ gcc -L/home/ryanarn/libs -Wall -o foo foo.c -lbar
```

```
$ objdump -x foo | grep PATH
RPATH                /home/ryanarn/libs
```

- The \$ORIGIN trick works here as well

```
$ export LD_RUN_PATH='$ORIGIN/libs'
```

- *Note: this is not a runtime environment variable. It is only understood by the static-linker, not by the dynamic-linker.*



# DT\_RPATH and LD\_LIBRARY\_PATH

- What if you link with `-rpath` and at runtime use `LD_LIBRARY_PATH`?

*The dynamic-linker will search for the dependencies in the directories specified by the **DT\_RPATH** dynamic array tag **before** searching those specified in `LD_LIBRARY_PATH`. This means you cannot override the library that is selected by using `LD_LIBRARY_PATH`.*

# DT\_RUNPATH Dynamic Array Tag

- As described on the slide titled [DT\\_RPATH and LD\\_LIBRARY\\_PATH](#), it was shown that the DT\_RPATH is always searched before LD\_LIBRARY\_PATH.
- In other words, the user can't override what's been encoded in the binary by the developer if DT\_RPATH was set.
- The `--enable-new-dtags` static linker flag with `-rpath` solves this problem by creating a new DT\_RUNPATH dynamic array tag.

# DT\_RUNPATH Dynamic Array Tag

- When `--enable-new-dtags` is passed to the linker along with `-rpath` a dynamic array tag called `DT_RUNPATH` is added to the dynamic section along with the expected `DT_RPATH` dynamic array tag.

```
$ gcc -L/home/ryanarn/libs/ -Wl,--enable-new-dtags \
    -Wl,-rpath='$ORIGIN/libs' -o foo foo.c -lbar
```

```
$ objdump -x foo | grep PATH
RPATH          $ORIGIN/libs
RUNPATH        $ORIGIN/libs
```

- This has a very subtle but important property that differs from when `-rpath` is used without `--enable-new-dtags` (as described in the next slide)

# DT\_RUNPATH and LD\_LIBRARY\_PATH

- What if you link with `-rpath` and `--enable-new-dtags` and at runtime use `LD_LIBRARY_PATH`?

*The dynamic-linker will search for the dependencies in the directories specified by **LD\_LIBRARY\_PATH** before looking in the directories specified by the `DT_RUNPATH` dynamic array tag. The `DT_RPATH` dynamic array tag is ignored. This means you can override the library that is selected by using `LD_LIBRARY_PATH`*

# Dynamic-linker treatment of DT\_RPATH vs DT\_RUNPATH

- **DT\_RUNPATH (with DT\_RPATH)**
  - Ignore DT\_RPATH, search LD\_LIBRARY\_PATH first, then DT\_RUNPATH.
- **DT\_RPATH alone**
  - search DT\_RPATH first then search LD\_LIBRARY\_PATH
- ***Per the ELF ABI DT\_RPATH is deprecated and superseded by DT\_RUNPATH***
  - *This is because it's more useful for the user to override the search path than it is to enforce the hard-coded search path first.*
  - *In reality there are some problems, particularly when using dlopen.*



# Dynamic Array Tag Transitivity

- **Transitive** - ... the property that if the relation holds between a first element and a second and between the second element and a third, it holds between the first and third elements <equality is a transitive relation>.
- A shared-object or executable with a transitive dynamic array tag may have the search paths therein applied to any other shared-object or executable.
- A shared-object or executable with a non-transitive dynamic array tag may only have the search paths therein applied to the shared-object or executable to which it was defined.

# DT\_RPATH Is Transitive

- Let's use an example from the binutils bugzilla <sup>[1]</sup> to discuss transitivity of DT\_RPATH vs DT\_RUNPATH:

```
$ cat foo.c
#include <dlfcn.h>

int main () {
    void* handle = dlopen("libso1.so", RTLD_LAZY);
    typedef int (*hello_t)();
    hello_t hello1 = (hello_t)dlsym(handle, "hello1");
    hello1();
    return 0;
}
```

```
$ cat so1.c
extern void hello2 (void);
void hello1 (void) {
    hello2();
}
```

```
$ cat so2.c
#include <stdio.h>
void hello2 () {
    printf ("hello\n");
}
```

[1] [https://sourceware.org/bugzilla/show\\_bug.cgi?id=15096#c0](https://sourceware.org/bugzilla/show_bug.cgi?id=15096#c0)

# DT\_RPATH Is Transitive (cont)

- Even though DT\_RPATH specified only when linking `libso1.so` the dynamic-linker will find `libso2.so` because DT\_RPATH is processed transitively by the dynamic-linker, meaning it is able to apply the DT\_RPATH dynamic array tag to all shared object searches:

```
$ gcc -fPIC -O2 -shared -o libso2.so so2.c
$ gcc -fPIC -O2 -shared -o libso1.so so1.c libso2.so
$ gcc -o foo1 foo.c libso1.so -Wl,-rpath,. -ldl
$ readelf -d foo1 | grep PATH
0x000000000000000f (RPATH)          Library rpath: [.]
$ ./foo1
hello
```

# DT\_RUNPATH Is Non-Transitive

- The ELF spec <sup>[1]</sup> says:

The set of directories specified by a given DT\_RUNPATH entry is used to find only the immediate dependencies of the executable or shared object containing the DT\_RUNPATH entry. That is, it is used only for those dependencies contained in the DT\_NEEDED entries of the dynamic structure containing the DT\_RUNPATH entry, itself. One object's DT\_RUNPATH entry does not affect the search for any other object's dependencies.

[1] [http://uw714doc.sco.com/en/SDK\\_cprog/OF\\_ShObjDependencies.html](http://uw714doc.sco.com/en/SDK_cprog/OF_ShObjDependencies.html)

# DT\_RUNPATH Is Non-Transitive (cont)

- Let's link the previous example with DT\_RUNPATH and see what happens:

```
$ gcc -o foo2 foo.c libso1.so -Wl,--enable-new-dtags -Wl,-rpath,. -ldl
$ readelf -d foo2 | grep PATH
0x000000000000000f (RPATH)          Library rpath: [.]
0x000000000000001d (RUNPATH)       Library runpath: [.]
$ ./foo2
./foo2: error while loading shared libraries: libso2.so: cannot open shared object
file: No such file or directory
```

- Per the ELF ABI, DT\_RUNPATH is non-transitive, i.e., the dynamic-linker is prohibited from searching the directories specified in one DT\_RUNPATH dynamic array tag for other shared-objects.

# DT\_RUNPATH Is Non-Transitive (cont)

- The solution many have decided upon is to use LD\_LIBRARY\_PATH:

```
$ LD_LIBRARY_PATH=.:$LD_LIBRARY_PATH ./foo2
$ ./foo2
hello
```

- This, of course means all of the other problems with LD\_LIBRARY\_PATH apply.

# DT\_RUNPATH Is Non-Transitive (cont)

- Another option is to specify DT\_RUNPATH when linking `libso1.so` so that the directory of `libso2.so` is embedded in the shared-object dynamic array tag.

```
$ gcc -fPIC -O2 -shared -o libso2.so so2.c
$ gcc -fPIC -O2 -shared -o libso1.so so1.c libso2.so \
  --enable-new-dtags -Wl,-rpath,'$ORIGIN/.'
$ gcc -o foo2 foo.c libso1.so -Wl,--enable-new-dtags -Wl,-rpath,. -ldl
$ readelf -d foo2 | grep PATH
0x000000000000000f (RPATH)          Library rpath: [.]
0x000000000000001d (RUNPATH)       Library runpath: [.]
./foo2
hello
```



connect

San Francisco 2015

# DT\_RUNPATH Is Non-Transitive (cont)

- Unfortunately you don't always have control over how library dependencies are linked, especially when `dlopen` is used.
- Many libraries will use `dlopen` to open an older version of libraries that were linked under the older dynamic tags methodology in order to bridge ABI changes, etc.
- In the future this should continue to be less of a problem as more Linux distributions adopt the newer dynamic array tags by default.

# ld.so.conf

- There is a lot written on the internet ld.so.conf and how to specify search paths so we won't go over that here.

# Symbol Versioning

- Compiler and linker directives aimed to provide a way to create a stable API and ABI
  - This avoids build and runtime issues such as undefined symbols or linking errors
- At build time the linker tries to satisfy undefined references and match the version against the ones provided by the DSO
- At runtime the loader will match and bind the specified version in the binary or DSO with the ones provided by the installed DSOs

# Stable API and ABI

- Means that the interface provided by the DSO remains without modification that breaks either build and runtime over the version lifecycle
  - It includes both functions and classes definition along with data structure size and layout
- Changes are incorporated by creating new revisions while either providing a new ABI or changing the defaults
  - Bugfixes not always breaks the ABI/API, although they might

# Scenarios

- One needs to fix a DSO function and it requires changing the API
  - The DSO is shared among multiple binaries and world rebuild is not an option
- One needs to provide a new binary with an updated DSO API
  - But old binaries must continue to work
- A bugfix is required on a DSO interface to fix a specific binary
  - But old binaries/DSO are not required to further validation, thus old interface must exist until new binaries are deployed

# Versioning - Example 1

- Simple DSO versioning using file system name
- Update without ABI/API breakage is just a matter of recompiling the binary
  - Any breakage require full rebuild
- It increase the waste of resources specially at runtime when running application need more than one version of the DSO

# Versioning - Example 2

- Simple example showing how to create and use different version for a same symbol
- The default symbol can be changed while providing old symbols
- For instance, given a two binaries and a DSO one can update the DSO and add a new version of 'foo' with a different interface and provide a third binary that uses this new version without require to rebuild all the binaries

# Versioning - Example 3

- It is basically the same as previous example but with a hack to use a different version than default
- Not a suggested approach of symbol versioning, but sometimes it requires
  - For instance the Intel GLIBC memcpy/memmove implementation

# Versioning - Example 4

- The objective is to show how to update a function prototype without breaking old binaries
  - The arguments is changed, either by changing its type, size, etc.
- First the initial binary is build against the old interface. Then the new library is built and replace the first version. This new library provides both the old function version and a new implementation with a new version.
- Both new and old binaries works as intended, each one binded to different symbol versions.

# Versioning - Resources

- How to Write Shared Libraries - <http://www.akkadia.org/drepper/dsohowto.pdf>
- Versioning a Structure - [https://sourceware.org/glibc/wiki/Development/Versioning\\_A\\_Structure](https://sourceware.org/glibc/wiki/Development/Versioning_A_Structure)
- Examples used for this presentation - <https://github.com/zatrazz/sfo15/tree/master/versioning>
- libabigail - <http://dodji.seketeli.net/talks/gnu-cauldron-2015/libabigail-state-of-the-onion.pdf>

# Timers and Timestamps

- An identifier when a certain event occurred
  - Different accuracy and formats
- Provided either by the operational system or by direct hardware access
  - Different APIs (POSIX, languages)
  - Different precisions
  - Different latencies

# Timers and Timestamps - POSIX

- Different API with different precisions
  - time, ftime
    - second resolution
  - gettimeofday
    - microsecond resolution
  - clock\_gettime
    - nanoseconds resolution
      - Relies on hardware precision
    - Different clocks (CLOCK\_REALTIME, CLOCK\_MONOTONIC, etc.)

# Timers and Timestamps - POSIX time

- Multiple function to transform to human readable form (days, hours, minutes, etc.)
  - asctime, ctime, gmtime, localtime, mktime
- Usually implemented through more precise timers
  - GLIBC uses gettimeofday

# Timers and Timestamps

- `gettimeofday`
  - POSIX.1-2008 marks `gettimeofday()` as obsolete, recommending the use of `clock_gettime`
  - Although for most of cases it suffice
  - Usually implemented through a syscalls or a vDSO symbol
- `clock_gettime`
  - Some SMP issues one must take care of
  - Different clocks with different internal implementations
    - Some are handled through syscalls, by hardware, or vDSO



# Timers and vDSO

- vDSO: A common shared library attach by the kernel on every running process to provide common functionalities
- Latency compared to syscalls (aarch64 box):

clock-gettime-monotonic-coarse: syscall: 149 nsec/call

clock-gettime-monotonic-coarse: libc: 21 nsec/call

clock-gettime-monotonic-coarse: vdso: 17 nsec/call

clock-gettime-realtime: syscall: 163 nsec/call

clock-gettime-realtime: libc: 62 nsec/call

clock-gettime-realtime: vdso: 59 nsec/call

clock-gettime-realtime-coarse: syscall: 134 nsec/call

clock-gettime-realtime-coarse: libc: 18 nsec/call

clock-gettime-realtime-coarse: vdso: 16 nsec/call

gettimeofday: syscall: 171 nsec/call

gettimeofday: libc: 89 nsec/call

gettimeofday: vdso: 86 nsec/call



connect

San Francisco 2015

# Timer and Timestamps

- C++11 provides a high level API
  - chrono header
- Hardware might provide specialized instruction to access to high resolution timers in userspace
  - Required for microbenchmarks and latency sensitive workloads
  - aarch64: cntvct\_el0 provides a fixed frequency counter to user space
    - Required for Server Base System Architecture Level 0, but not might be present in all hardwares

# What the compiler can't know

- Unit-at-a-time compilation
- Whole-program compilation
- Run-time effects
- Algorithmic optimizations
- Undocumented CPU features

# Unit-at-a-time compilation

- Unreachable functions
- Unused variables
- Constant variables
- Function binding
- Big functions
- Big applications

# Whole-program compilation

- Unreachable functions
- Unused variables
- Constant variables
- **Function binding**
- **Big functions**
- **Big applications**

# Runtime effects

- Layout and alignment of code and data
  - Data arrays fill cache lines differently
- Branch prediction
  - Branch hinting is discouraged for smart cores
- Balance between CPU and RAM speeds

# Algorithmic optimizations

- Compiler will diligently generate code for stupid algorithms
  
- Simple recursion can be converted to loop

# Undocumented CPU features

- Undocumented CPU features
  - Artifacts of cache / memory sub-system
- Sparse information on CPU bottlenecks
  - Instruction decoder

# Out-of-order vs in-order execution

- Comparison of different ARM cores
  - big vs LITTLE
- OOO cores still benefit from instruction scheduling
- Which `-mcpu=` value to use for big.LITTLE
  - `-mcpu=cortex-a57.cortex-a53`

# Out-of-order vs in-order execution

- A-profile
  - big cores are out-of-order
  - LITTLE cores are in-order
- R-profile
  - mostly in-order cores
- M-profile
  - almost all in-order cores