How to Write a Shared Library

Adam Midvidy

Who am I?

- Engineer on Platforms Team @ MongoDB
  - Help maintain the C++ Driver
- Co-organizer of C++ Developers Group

Who are you?

- Have developed C++ applications
- Have used libraries - but not written them

Caveats

- Examples/code are Linux specific
  - Will work on OSX as well.
  - ...but not Windows.
- I am *not* the world's foremost expert on shared libraries.
  - If you are, this is probably not the talk for you.

Talk Overview

1. Refresher on shared libraries
2. Why should you care?
3. Demo the example library
4. The “Shared Library Design Checklist”
5. Resources
6. Q/A

Refresher - Shared vs Static

Static Library

- code linked into application at compile time
- bigger binaries, faster startup time

Shared Library

- code linked into application at runtime
- smaller binaries, slower startup time

How do I make one?

```
$ cat ex.cpp

#include “ex.hpp”
int getRandomNumber() { return 4; }

$ c++ ex.cpp -fpic -shared -o libex.so
$ file libex.so
libex.so: ELF 64-bit LSB shared object....
```

How do I use one?

```
$ cat app.cpp
#include <iostream>
#include "ex.hpp"
int main() { std::cout << getRandomNumber() << std::endl; }
$ c++ app.cpp -lex -L. -I. -o app
$ LD_LIBRARY_PATH=. ./app # Don't actually use LD_LIBRARY_PATH!
4
$ readelf -d app
Dynamic section at offset 0xe08 contains 26 entries:
  Tag        Type                         Name/Value
 0x0000000000000001 (NEEDED)             Shared library: [libex.so]
... (and more) ...
```

Why Shared Libraries?

- Can update without recompiling
- Quicker link times during development
- Can load dynamically with `dlopen(3)` and friends
- More efficient memory usage (still important for mobile)

Why are shared libraries hard?

- If you want all the benefits of shipping shared libraries - you need to do it right.
- If you don’t do it right from the start, you will make your life difficult for a long time.

ABI vs API

- API - Application *Programming* Interface
  - data types, functions, classes, etc.
  - content of your library's header files
  - consumed by compilers & humans
- ABI - Application *Binary* Interface
  - exported symbols, structure layout, etc.
  - content of your library's .so files
  - consumed by linkers

ABI

- ABI is versioned separately from ABI
  - more on this later
- ABI changes
  - ABI addition - does *not* affect forwards compatibility
  - ABI break - modification or removal of symbols.
    - This needs to be handled carefully, and should be avoided if possible.

Introducing libfactorial

- Strawman example library
- Will modify it as we go
- Code overview

libfactorial ABI (v0)

```
% nm -g --defined-only libfactorial.so | grep ‘ T ’ | c++filt
0000000000001180 T factorial::calculator::calculate(unsigned long long)
0000000000000e30 T factorial::calculator::calculator(factorial::options)
00000000000010f0 T factorial::calculator::calculator(factorial::options)
0000000000000ea0 T factorial::memoizer::get(unsigned long long)
0000000000000e60 T factorial::memoizer::store(unsigned long long,
unsigned long long)

.... (other stuff)
```

Notice two identical constructors - an artifact of the C++ Itanium ABI

Shared Library Design Checklist

1. Adopt a sane versioning scheme.
2. Set a soname to reflect your ABI version.
3. Export a minimal ABI surface.
4. Be mindful when inlining functions.
5. Ensure multiple versions can coexist.

Adopt a sane versioning scheme

- ABI and API should be versioned **separately**
- Semver for API (Major.Minor.Patch)
- Single number for ABI - bump on break only
  - note: autotools uses a more complicated scheme

Separate Marketing Version?

```
// Marketing-driven product version
#define TBB_VERSION_MAJOR 4
#define TBB_VERSION_MINOR 3

// Engineering-focused interface version
#define TBB_INTERFACE_VERSION 8001
#define TBB_INTERFACE_VERSION_MAJOR TBB_INTERFACE_VERSION/1000
```

From Intel’s Threading Building Blocks

Shared Library Design Checklist

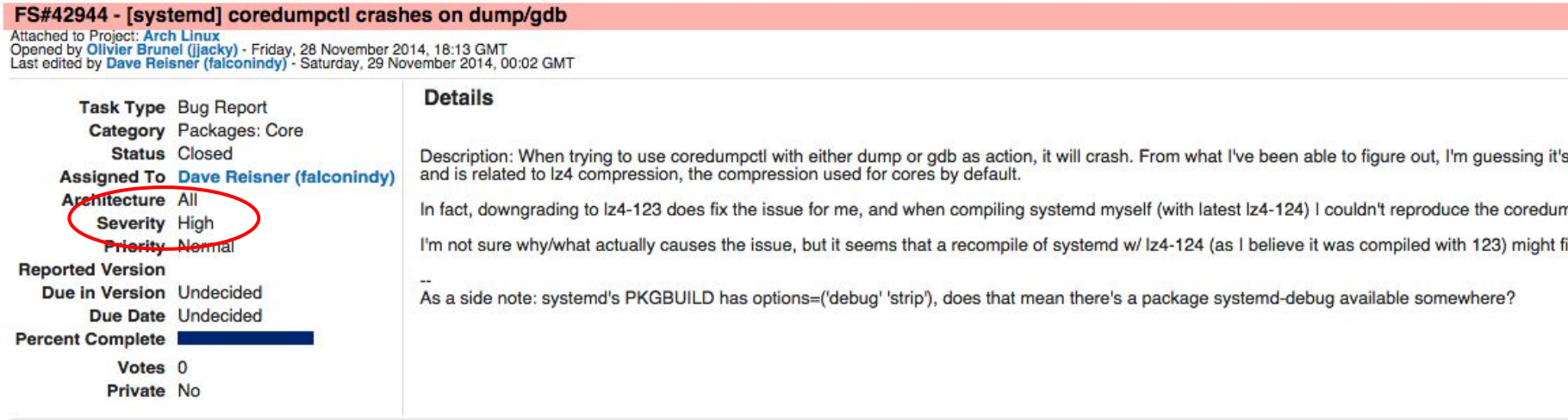
1. ~~Adopt a sane versioning scheme.~~
2. Set a soname to reflect your ABI version.
3. Export a minimal ABI surface.
4. Be mindful when inlining functions.
5. Ensure multiple versions of can coexist.

Set a soname to reflect your ABI version.

- Soname encodes the ABI version of library
- Specify soname when building library
  - Recorded in DT_SONAME field of ELF header
- Pass `-Wl,soname,mysoname.1` to compiler
- At compile time, soname will be embedded in executable
  - Recorded in DT_NEEDED field of ELF header

If you break ABI - you must change your library's soname.

LZ4 ABI Break



FS#42944 - [systemd] coredumpctl crashes on dump/gdb

Attached to Project: Arch Linux
Opened by Olivier Brunel (jjacky) - Friday, 28 November 2014, 18:13 GMT
Last edited by Dave Reisner (falconindy) - Saturday, 29 November 2014, 00:02 GMT

Task Type	Bug Report
Category	Packages: Core
Status	Closed
Assigned To	Dave Reisner (falconindy)
Architecture	All
Severity	High
Priority	Normal
Reported Version	
Due in Version	Undecided
Due Date	Undecided
Percent Complete	
Votes	0
Private	No

Details

Description: When trying to use coredumpctl with either dump or gdb as action, it will crash. From what I've been able to figure out, I'm guessing it's and is related to lz4 compression, the compression used for cores by default.

In fact, downgrading to lz4-123 does fix the issue for me, and when compiling systemd myself (with latest lz4-124) I couldn't reproduce the coredum

I'm not sure why/what actually causes the issue, but it seems that a recompile of systemd w/ lz4-124 (as I believe it was compiled with 123) might fi

--
As a side note: systemd's PKGBUILD has options=('debug' 'strip'), does that mean there's a package systemd-debug available somewhere?

LZ4 ABI Break

```
236  236
237       -#define LZ4_STREAMDECODESIZE_U32 4
     237  +#define LZ4_STREAMDECODESIZE_U32 8
238  238   #define LZ4_STREAMDECODESIZE     (LZ4_STREAMDECODESIZE_U32 * sizeof(unsigned int))
239  239   /*
240  240    * LZ4_streamDecode_t
241  241    * information structure to track an LZ4 stream.
242  242    * important : init this structure content using LZ4_setStreamDecode or memset() before first use !
243  243    */
244  244 + typedef struct { unsigned int table[LZ4_STREAMDECODESIZE_U32]; } LZ4_streamDecode_t;
```

LZ4 ABI Break

New issue | Search | Open issues | for | Search | Advanced sea

Issue 147: lz4 r124 breaks ABI without bumping soname

6 people starred this issue and may be notified of changes.

Status: WontFix
Owner: ----
Closed: Dec 24

Type-Defect
Priority-minor

Add a comment below

Reported by d...@falconindy.com, Nov 28, 2014

```
You can't make changes like this without increasing lz4.so's soname version:

https://code.google.com/p/lz4/source/diff?spec=svn124&r=124&format=side&path=/

ABI changes this like cause programs which link to lz4.so to behave strangely,

https://bugs.archlinux.org/task/42944

Please be considerate of your downstream consumers.
```

Soname symlinks

- physical library
  - file, libexample.so.1.2.3
- soname symlink
  - symlink libexample.so.1 -> libexample.so.1.2.3
- dev symlink
  - symlink libexample.so -> libexample.so.1

How does this all work?

- c++ app.cpp -lexample -o app
  - linker looks for libexample.so on the lib include path
  - finds dev symlink, which points to physical library
  - embeds soname from DT_SONAME field of library into DT_NEEDED field of executable at compile time
- ./app
  - runtime linker (ld.so) looks for filename matching soname, finds soname symlink, which points to physical library

Libfactorial Example

Link

Shared Library Design Checklist

1. ~~Adopt a sane versioning scheme.~~
2. ~~Set a soname to reflect your ABI version.~~
3. Export a minimal ABI surface.
4. Be mindful when inlining functions.
5. Ensure multiple versions of your library can coexist.

Export a Minimal ABI Surface

- So I can’t break my ABI…. how do I prevent that?

ABI Stability in C++

- (Incomplete list) of things that can break ABI
  - Adding/removing a virtual method
  - Adding/removing member variables (incl. private)
  - Changing declaration order of member variables
  - Changing an inheritance hierarchy
  - Changing an inline function (if old version doesn’t continue to work)
  - redefining an existing method inline
  - …lots more...

Expose a Minimal ABI Surface

- Only expose the minimal set of symbols.
- A symbol's visibility determines whether it will be exported to users of the library
- **By default, everything is exported**
- To change the default, use the `-fvisibility` flag

Expose Symbols Selectively

- Use `-fvisibility=hidden` to hide everything by default
- Export symbols explicitly with visibility attributes
- Old way: `__attribute__((visibility(“default”))`
- C++11: `[[gnu:visibility(“default”)]]`

Libfactorial Example

Link

PIMPL your classes

- “....another level of indirection”
- Hide implementation of class behind an opaque ‘implementation pointer’
- Frees us from worrying about the size of our types at the cost of (minor) runtime performance

Libfactorial Example

Link

Shared Library Design Checklist

1. ~~Adopt a sane versioning scheme.~~
2. ~~Set a soname to reflect your ABI version.~~
3. ~~Export a minimal ABI surface.~~
4. Be mindful when inlining functions.
5. Ensure multiple versions of your library can coexist.

Be mindful when inlining functions

- When you inline a function, any types used in its definition are exposed to consumers
- When you inline a function, you can’t de-inline it
- If you change it, the old version still has to work

Ensure Inline Functions Are Inlined

“The rationale for the use of always_inline in libc++ is to control the ABI….compilers use different heuristics from release to release on making the inline/outline decision. This can cause code to be silently added to and removed from a dylib” - **Howard Hinnant**

Libfactorial Example

Link

Shared Library Design Checklist

1. ~~Adopt a sane versioning scheme.~~
2. ~~Set a soname to reflect your ABI version.~~
3. ~~Export a minimal ABI surface.~~
4. ~~Force inline functions to be inlined.~~
5. Ensure multiple versions of your library can coexist.

Inline Namespaces

- Added in C++11

```
library: inline namespace v0 { int bar(); }
app: bar(); // don’t need to qualify with v0
```

- `bar` is part of ABI symbol, not API
- Allows you to version symbols!
- Enables using multiple incompatible ABI versions of a library in one application

Libfactorial Example

Link

Parallel Install Directories

- Install headers to versioned subdirectory

BAD: /usr/local/lib/example/foo.hpp

GOOD:/usr/local/lib/example/**v0.0/**foo.hpp

- Allows multiple versions of your headers to coexist on the same machine

pkg-config

- Distribute a pkg-config file with your library
- Before:

```
c++ app.cpp -I/usr/local/example/v0.0/ -I/usr/local/mydep/v0.0/
-I/usr/local/example/myotherdep/v0.0/ -lexample -lmydep
-lmyotherdep
```

- After:

```
c++ app.cpp $(pkg-config --libs --cflags example)
```

pkg-config example

```
prefix=/usr/local
includedir=${prefix}/include
libdir=${prefix}/lib

Name: factorial
Version: 0.1.0
Description: A factorial library
URL: http://github.com/amidvidy/shared-library-talk
Cflags: -I${includedir}/factorial/v0.0
Libs: -L${libdir} -lfactorial
```

Shared Library Design Checklist

1. ~~Adopt a sane versioning scheme.~~
2. ~~Set a soname to reflect your ABI version.~~
3. ~~Export a minimal ABI surface.~~
4. ~~Be mindful when inlining functions.~~
5. ~~Ensure multiple versions of your library can coexist.~~

Review:

1. Adopt a sane versioning scheme.
2. Set a soname to reflect your ABI version.
3. Export a minimal ABI surface.
4. Be mindful when inlining functions.
5. Ensure multiple versions of your library can coexist.

Resources

- Ulrich Drepper - How to Write Shared Libraries
- Sun Linker and Libraries Guide
- libabigail - library and tools for manipulating and analyzing ABIs

Questions?

Fin.

- Twitter - @amidvidy
- Email - amidvidy@gmail.com

- Let me know if you are interested in giving a C++ meetup talk!