[redacted]

A fresh look at toolchains in 2021

Bernhard "bero" Rosenkränzer

<bero@lindev.ch>

FOSDEM 2021 -- February 6, 2021

The current situation

There is a well known, well documented (but slightly complicated) way to build crosscompilers -- some alternatives (some even good) have sprung up, but are not yet as widely known.

The traditional way

- Build binutils
- Build a minimal gcc crosscompiler (C only, no threads, ...)
- Build glibc
- Build gcc again with all needed features (including libstdc++ if you want C++ support)

Advantages

- By far the most widespread -- most 3rd party libraries and applications have been tested in this setup, and chances are people on IRC, mailing lists, forums, ... have done this (and can likely help)

binutils

- binutils is a collection of tools that deal with object files -- you might not have used them directly, your compiler uses them.
- There are 3 major implementations, and you're likely using all of them already:
  - GNU binutils (the "standard" implementation)
  - elfutils (shipped with almost all distributions to get libelf)
  - LLVM binutils (part of LLVM, built everywhere for Mesa etc.)

binutils

- elfutils provides good implementations of the tools it provides, but is lacking a linker -- you still need a linker from GNU binutils or LLVM.
- Tools in GNU binutils and LLVM binutils are pretty much interchangable and use mostly the same parameters
- Big advantage of LLVM tools: Crosscompilers built in. "`llvm-objdump --disassemble`" on an ARM binary works just fine on an x86 box (or vice versa) while you need per-architecture tools in GNU binutils

Alternative compilers: Clang

- There's another good compiler: clang (from LLVM). LLVM can also replace binutils. In LLVM 10, the lld linker (up until recently, the main blocker for replacing binutils) has become good enough and compatible enough to replace ld.bfd and ld.gold for almost everything. And it's getting better with 11 and 12 (while gold is seeing little attention).

Alternative compilers: Clang

- Clang is a crosscompiler by design: You don't have to build special per-target crosscompilers, clang can target any supported platform.
  Instead of having to build e.g. `aarch64-linux-gnu-gcc`, you can use `clang -target aarch64-linux-gnu` -- for any architecture.
- It's easier to get into Clang's code than into gcc's code.

Alternative compilers: Clang

- Many targets - including some GPUs - supported
- Performance of clang-built binaries is similar to gcc-built binaries. There are special cases where clang performs better, and others where gcc performs better. On average, their performance is similar.
- Clang tries to be a drop-in replacement for gcc, implementing many gcc extensions
- Initial release was in 2012 (gcc: 1987)

Alternative compilers: Clang

- Apache 2.0 licensed (with both the advantages and drawbacks compared to gcc's GPL)
- There's a good chance you'll need LLVM anyway (it is used, among other things, by Mesa) -- but on the other hand, depending on some other decisions, you may need GCC anyway even if you opt for clang (libstdc++, libgcc_s)

Alternative compilers: Clang

- Fortunately, clang and gcc are binary compatible. You can link a gcc-built binary to a clang-built library and vice versa. In fact, you can even

```
gcc -O2 -o test1.o -c test1.c
clang -O2 -o test2.o -c test2.c
gcc [or clang] -o test test1.o test2.o
```

- If you want to mix compilers that way, you have to use gcc's support libraries (libgcc rather than compiler-rt) - clang can use gcc's, but not vice versa (at least without -nostdlib trickery)

Alternative compilers: TinyCC

- TinyCC is what its name implies - probably just about the smallest possible implementation of a full C99 compiler -- the compiler's source is smaller than 4 MB, and it takes mere seconds to compile.
  It has interesting uses (e.g. embedding inside an application), but doesn't optimize as strongly as clang or gcc.
  It is also limited to C (no C++). It might be interesting for small embedded devices.

Compilers: OpenArk (not yet ready)

- Announced by Huawei, but so far not usable:
  The OpenArk compiler is supposed to become a C, C++, Java, Kotlin and JavaScript compiler generating native code.
  So far, the released code can compile Java to aarch64 assembly, but the largest component is a binary blob for now. **This is being fixed.**
- May be interesting in the future, but not there yet.

Compilers: BSPs

- Many BSPs (Board Support Packages) that come with development boards contain a compiler.
  This compiler is usually a fork of an outdated version of gcc or clang (both of which, in the mean time, have typically added much better support for the hardware in question).
  Unless you're working on a very special device (not yet supported by the upstream compilers), it's usually good advice to ignore the BSP and build your own clang or gcc.
- Sometimes that means adding a few kernel patches to support newer toolchains - those patches are usually already written and relatively easy to find (try the kernel git repository).
- Try to avoid anything not based on gcc >= 8 or clang >= 9.

Compilers: Conclusions

- gcc and clang are both good options. There is no clear winner.
- Both have been used to compile full systems (including the kernel). Most Linux distributions are built mostly with gcc, some (OpenMandriva, Android) and the BSDs are built mostly with clang. Some build-from-source distributions offer both choices.
  OpenHarmony will likely use a clang based toolchain in the future (currently inherits gcc from Yocto).

Compilers: Conclusions

- clang makes it easier (and, unless you're very familiar with gcc's code base, faster) to add new architectures and new languages, and is mostly built as a library. If you're planning to add architectures and new language, or to embed the compiler in your own projects, give clang a try.
- If you're using glibc, you need gcc to build it (for now). If you don't need any of the extras offered by clang, you may want to go with gcc for everything.

libc: glibc

- For the system libc, glibc is the default option:
  - most widespread
  - most complete/most standards compliant
  - very well tested
  - most complete arch support (aarch64, arm, x86, x86_64, x32, RISC-V 64, alpha, C-Sky, hppa, ia64, m68k, microblaze, mips, powerpc, S/390, sh, SPARC)
- But:
  - code not very readable
  - compiles only with gcc (patches to make an older version compile with clang exist in the google/grte/v6-2.29/master branch of glibc)
  - not very optimized for small systems
  - rather big (roughly 4 MB for ld.so, libc, libm, libpthread)

libc: musl

- Complete, fast and relatively small (785 kB)
- Designed for C11+ and POSIX 2008+, with many glibc, Linux and BSD extensions
- Supports aarch64, arm, x86, x86_64, x32, RISC-V 64, m68k, microblaze, mips, mips64, mipsn32, or1k, powerpc, powerpc64, s390x, sh
- Readable code
- Started 2011

libc: uClibc-ng

- Complete, fast and relatively small (1 MB in full config)
- Can be stripped down easily
- Focused on embedded systems
- Supports many processor types, including MMU-less: Aarch64, Alpha, ARC, ARM, AVR32, Blackfin, CRIS, C-Sky, C6X, FR-V, H8/300, HPPA, i386, IA64, LM32, M68K/Coldfire, Metag, Microblaze, MIPS, MIPS64, NDS32, NIOS2, OpenRISC, PowerPC, RISCV64, Sparc, Sparc64, SuperH, Tile, X86_64 and XTENSA

libc: klibc

- Written for the early bootup process, used in the initramfs of Debian and some derivates
- Subset of libc functions, optimized for size over performance
- More direct use of kernel structures avoids some type conversion (e.g. between different ideas of "struct stat")
- Extremely small (75 kB)
- But not powerful enough as a real world libc - might be an option for some embedded systems
- Uses GPL kernel headers, resulting license situation not 100% clear.

libc: LLVM libc (not yet complete)

- In its early stages, but some code is there.
- Potentially interesting in the future because:
  - Designed to work with sanitizers and fuzz testing from the start
  - Targeting C17 and up - not carrying around ancient cruft
  - Design goal: "Use source based implementations as far possible rather than assembly. Will try to fix the compiler rather than use assembly language workarounds."
  - The LLVM project has a track record of delivering good toolchain options

libc: bionic (Android)

- Originally based on the BSD libc, bionic is the libc used in Android.
- Currently supports ARM (32 and 64) and x86 (32 and 64)
- Rather well optimized because of vendor support for Android
- Used to be unusable for a regular Linux system - lacking e.g. SysV SHM needed for X11 - but has largely caught up
- Unfortunately, at the same time added some Android-isms that make it harder to use outside of a full Android system (APEX, system properties etc.), build system tied to the Android tree

libc: bionic (Android)

- Potentially makes it possible to use closed drivers written for Android in a regular Linux system without having to go through hacks like libhybris
- May be interesting to build Linux/Android hybrid systems

Other potential libc options

- newlib is limited to static linking - if you don't need dynamic linking, it may be for you.
  Most Zephyr builds use newlib.
- dietlibc is optimized for small size and static linking - but not very actively maintained, and on something as low level as a libc, its GPL (not LGPL) license may be a problem if your system will allow building/installing/running custom code.

libc: Conclusions

- There are many interesting options - for now:
- If you need **maximum compatibility** with other systems, go with **glibc**.
- If you need a full fledged, but **smaller** and **more memory efficient** libc, go with **musl**.
- If you need a subset of libc and want to **strip out unneeded components**, try **uClibc-ng**.
- If you want to experiment with **Android** features on regular Linux, try **Bionic**.

C++ support: libstdc++

- libstdc++ is part of gcc, used by almost all Linux distributions including some that use clang as their primary compiler (notable exception: Android)
- This is what almost everything is developed against - the easiest option if you don't want to tweak code to add missing `#include`s that happen to be ignored by libstdc++.

C++ support: libc++

- libc++ is an optional part of LLVM/Clang.
- It's newer and smaller than libc++, carries less cruft to support ancient code. Most benchmarks also show it performing better.
- Problem: You can't mix libstdc++ and libc++ (for obvious reasons, they export the same symbols). You can't e.g. compile Qt against libc++ and expect a binary built with Qt/libstdc++ (such as pretty much any non-free software out there) to work.
- 3rd party applications (Chromium etc.) increasingly use libc++

C++ support: uClibc++

- uClibc++ is (was?) an attempt to write an STL implementation to go along with uClibc - a good idea (certainly you can strip out some parts of the STL when building an embedded system), but the last commit was in 2016.

C++ support - conclusions

- If binary compatibility with other Linux distributions is a big concern, go with libstdc++.
- If you're using clang and you care about performance and memory efficiency, try libc++.

Debuggers

- gdb has been the debugger to go to for a long time - initially released in 1986, and kept up to date (latest release: December 2020
- More recently, lldb - a part of the LLVM project - has come along. Initially released in 2003, it has become a realistic replacement for gdb by now.
- Both tools do pretty much the same job, and both do it well.

Debuggers: GDB and LLDB

- LLDB provides many command aliases for gdb compatibility.
- LLDB's native syntax tends to be cleaner (designed 20 years later - less need to retrofit new features), but also more verbose
- LLDB has the edge in C++ support, and evaluating expressions in the LLVM JIT
- Good news for remote debugger users: gdbserver and lldb-server speak the same protocol. You don't have to force users to use a specific debugger when deciding what (if any) debug server/stubs you put into a BSP/distro

Doing the right thing in a distribution

- Given there is no clear "best option for everything", a distribution should try to support developers opting for all options.
- Some things we have done and are planning to do in OpenMandriva (and you can do in your favorite distribution as well):

Keep crosscompilers up to date

- Many distributions decide to package crosscompilers at some point - and then forget about them when updating the native compiler (or adding an important patch). It's easy to take care of this -- OpenMandriva toolchain packages automatically build crosscompilers for all supported targets.
  If you'd like to know how to do it, take a look at the OpenMandriva gcc package (using rpm, but other package managers can do something similar)

Filesystem changes

- The traditional /usr/lib, /usr/lib64, /usr/lib32 split is no longer sufficient. There's multiple ABIs, and there's qemu-binfmt making it possible to run code for different CPUs seamlessly -- opening the door to e.g. running x86 Windows applications on ARM Linux with qemu-binfmt and an x86 wine package... If only there was a better place to put all the libraries wine needs...

Filesystem changes

- Debian derivates (and possibly others) have recently addressed this by creating /usr/lib/<triplet>, e.g. /usr/lib/x86_64-linux-gnu, /usr/lib/aarch64-linux-gnu
- That's a step in the right direction, but we prefer going for /usr/<triplet>/lib instead:
  - Allows combining the real filesystem with a crosscompiler sysroot
  - Allows for /usr/<triplet>/include and /usr/<triplet>/bin overrides where necessary (thankfully, only needed for a few libraries)

Filesystem changes

- Remaining compatible with previous releases and other distributions through symlinks is possible just as easily:

```
cd /usr
ln -s x86_64-openmandriva-linux-gnu/lib lib64
ln -s i686-openmandriva-linux-gnu/lib lib
...
```

Filesystem changes

- As an extra bonus, this enables mixing libcs and STL implementations as well - someone so inclined can keep the main system on e.g. `aarch64-openmandriva-linux-musl` while providing `aarch64-openmandriva-linux-gnu` libraries to fulfill the needs of relevant applications that can't be recompiled (non-free games etc).

Questions? Feedback? Bags of cash? ;)

- If you have any of the above, ask in the FOSDEM chat -- or email me at bero@lindev.ch or bernhard.rosenkraenzer.ext@huawei.com

HUAWEI

OSTC
Open Source Technology Center

Lin Dev

OpenMandriva
Association