Chimera Linux

Building a Linux distro with LLVM

Introduction

- General-purpose Linux distribution
- Particular focus on desktop/workstations
- Robustness, security hardening, good defaults
- Lightweight and transparent
- Pragmatic and versatile

Overview

- LLVM as system toolchain
- FreeBSD as core userland
- Musl as libc
- Apk-tools as package manager
- Binary packaging, clean source build infra

Overview

- Bootstrappable
- Can cross-compile (official repos are native)
- Supports many architectures:
  - Aarch64, ppc64le, ppc64, riscv64, x86_64
- System-wide LTO, UBSan, CFI, ...

Getting started

- Early 2021
- Cbuild: source package build system
- Template: describes one individual piece of software
- Host system: Void Linux, architecture: ppc64le
- Sandbox: runs its own build environment

Bootstrap process

- 4 stages
- Stage 0: use outside toolchain
- Stage 1: use packages from stage 0 (containerized)
- Stage 2: enable LTO and remaining build options
- Stage 3: clean rebuild (with stage 2 packages)

Build environment

- Bare minimum packages needed to build self
- Libc, compiler, core userland, package manager
- Build dependencies for software being packaged
- Overall a small Chimera container
- Running unprivileged (Linux user namespaces)

Why use LLVM?

- More modern compiler design
- Security: state of the art sanitizers (e.g. CFI)
- Easier to build and bootstrap, simpler cross-build
- Thin LTO, better performance
- Often less buggy

Why not use LLVM?

- Very occasionally worse compatibility
- Fewer architectures supported
- Some of the supported ones are less maintained
- Takes longer to build (bigger, idiomatic C++)
- Very rarely worse performance

Toolchain structure: typical

- A C library (usually glibc or musl)
- GNU Binutils: assembler, linker, ELF utilities
- GCC: C/C++ compiler, core runtime (crt+libgcc), C++ standard library, possibly other languages
- One build of binutils+gcc per target

Userland ABI: typical

- Part statically linked – builtins (libgcc.a), early crt
- Unwinder + dynamic builtins: libgcc_s.so.1
- C++ standard library: libstdc++.so.6
- Base C++ ABI: libstdc++.so.6 or libsupc++
- CRT is part libc, part GCC

Toolchain structure: LLVM

- LLVM: compiler, linker, assembler, binutils, all-in-one
- The only separate component is libc (glibc/musl)
- One compiler for all native and cross targets
- Only the runtime and target libraries are per-target
- Simplified bringup

Userland ABI: native LLVM

- Not used in most distros (LLVM uses GCC's env)
- Builtins are static (libclang_rt.builtins.a) - compiler_rt
- Unwinder: libunwind.so.1
- C++ library and base ABI: libc++.so.1, libc++abi.so.1
- Standalone (no GCC dependency)

ABI compatibility

- Libunwind ABI matches most of libgcc_s
- Builtins can be compiled into a makeshift libgcc_s
- Libc++ stdlib ABI is different (won't run GCC programs)
- However, libc++ and libstdc++ can live in one process
  - In theory (need libstdc++ using libc++'s ABI library)

Choosing a libc

- Glibc: could not build with Clang until recently
  - Still incompatible with native LLVM: dlopens libgcc_s.so.1
- Musl: builds and just works as-is
- There are others but generally domain-specific (e.g. embedded) or somehow worse than the other two

The allocator

- LLVM comes with yet another thing: Scudo
- The memory allocator used in Android, hardened
- Modular design: can mix and match components, configure them, or even provide custom ones
- Very unassuming (no mandatory ELF TLS, etc.)

The allocator

- Musl allocator: bad performance in threaded programs
- Replaced with Scudo: e.g. 3x faster lld LTO link perf
- No ELF TLS within libc: custom TSD registry
  - Plugs directly into the pthread structure, manual mmap
- Unfortunately very high virt mem usage :(

Cross-compilation

- Cbuild can cross-compile
- Cross targets need cross runtimes
- Includes compiler-rt, musl, libunwind, libc++(abi)
- Slightly tricky bootstrap
- Installed into sysroot

Bootstrapping cross runtimes

- Build compiler-rt builtins+crtbegin/end first
  - Force static libs (avoid conftests) and disable sanitizers
  - Requires libc headers, so give it some (musl in temp place)
- Build and install libc (needs only the above)
- Build and install libunwind and libc++(abi)

Bootstrapping cross runtimes

- Libunwind and libc++ can (should) be done all at once
  - Still needs explicit -nostdlib in CXXFLAGS (don’t have one)
- Now compile the rest of compiler-rt
  - This is mainly sanitizers
- Once in sysroot, this is the full cross runtime

Practical experiences

- Makes system-wide LTO possible
  - Far lower resource usage and near universal compatibility
- Security hardening
  - We deploy a subset of UBSan (signed overflows are crashes…)
  - Partial CFI; breaks too many projects to deploy universally

Practical experiences

- Toolchain patching is in line with GCC
  - Still heavier than I would like… (~30 patches)
- On Linux, often geared towards GCC-style environment
  - Upstream should consider dogfooding more
- The build system can be an impenetrable mess

Practical experiences

- Major release updates can be daunting
  - Every release breaks some third party software
  - Usually for a good reason (legacy C misfeatures…)
  - Still causes an unfortunate amount of fallout
- Community is good and helpful

Conclusion

- Generally a great toolchain
- Some pain points, but generally practical
- Can build just about any Linux software
  - Given GCC's history, an amazing feat
- Should not be reduced to "that GCC drop-in"

Thanks for listening!

- https://chimera-linux.org
- https://github.com/chimera-linux
- https://floss.social/@chimera_linux
- #chimera-linux @ OFTC (irc.oftc.net)
- #chimera-linux:matrix.org