What We Take For Granted

sin-ack

2025-12-04 Thu

Introduction (1/2)

- Build systems!
  - Just a big, executable dependency graph
  - Take some inputs, produce some outputs
- What are my inputs?
  - Source code
  - Other outputs
    - For this machine (custom build-time tools, code generators etc.)
    - For the target machine (built .o files/shared objects etc.)
  - Compiler/interpreter (the toolchain)
  - The operating system environment

Introduction (2/2)

- Most build systems are non-hermetic
- Inputs are not captured accurately
  - Assume toolchains/libraries/headers are available on the system
    - Assume the correct version is installed (sometimes impossible due to conflicts)
  - Assume build-time executables exist
  - Assume `BUILD == HOST`
    - Fun fact: You can't cross-compile most of GTK, thanks to introspection!
- Why is cross-compiling almost always a nightmare?
  - Because we don't accurately define our DAG
  - Because we ignore that the environment is an input

Absolute paths are evil

- Why is packaging portable software always painful?
- Why does Nix need everything to be in `/nix/store`?
- Why can't you install multiple versions of the same software?
  - Because everything abuses absolute paths
- Just a reality of Unix (Plan 9 was too late); have to actively fight it

The tyranny of the "system toolchain"

- "System toolchains" should not be used for anything serious!
  - For source-based distributions: meant for system packages only
  - For binary-based distributions: ideally should not exist at all
- Most system toolchains love to hardcode system paths
  - Accurate tracking of dependencies is nigh-impossible
  - Bazel puts in a lot of effort here (but it's still a mess)
- What's allowed by one compiler/interpreter version is not allowed by another!
  - GCC 14 removed a bunch of implicit header includes
  - Clang 19 removed the `std::char_traits` base template
  - Python regularly deprecates and removes stuff from its stdlib
  - And so on...

My tar is not your tar!

- Many different `tar` implementations out there
- When we say `tar`, we think about `gnutar`. Not the case when we're building on macOS
  - The `bsdtar` man page is around 6000 words
  - The `gnutar` documentation is at around 90000 words and weighs 1.3MB in HTML format
- In fact, `bsdtar` has features that `gnutar` doesn't, like mtree
- You can't make reproducible archives if you don't control the tar version
- Not limited to tar: Any tool you run should be controlled

At least coreutils are guaranteed, right..?

- macOS didn't ship `realpath` until Ventura
- BSDs and macOS won't ship (recent) Bash because of licensing
- Let's not even talk about Windows
- Even Bazel makes the same mistake, `genrule` targets don't work with a hermetic sandbox!
- If you don't include your tools in your build graph, your build graph is not accurate!

Does Nix fix this?

- The obvious question: Why not just use Nix?
- Nix works great if your only issue is building hermetically
- Awful DX :(
  - Nix eval gets very slow as your dependency graph grows
  - Nix works at package granularity, not file/module granularity (slow rebuilds)
  - "Fiddle until it works"
  - REPL becomes unusable if you eval the wrong thing, no way to cancel a build

How to stay sane

- Use a hermetic build system, please!
  - Higher up-front cost, but it pays off!
- My recommendation: Use Bazel with a proper toolchain setup
  - Not perfectly hermetic like Nix, but much better DX
- Assume the least from the system you're running under
  - Ideally, your software package should only depend on the dynamic linker, `libc` and (if necessary) vendor libraries like `libGL`
  - Make your standalone packages ship with relative `RUNPATH` values
  - But also: don't forget about system integration (for hermetic build systems this means sysroots/linking by `SONAME` of libs)
- What about `zig build`?
  - `zig build` is good at what it does, but doesn't generalize
  - On Zig/C/C++ repos, use `zig build` to your heart's content. Otherwise see above