The great wide world, part 2

Build rule generators

- autotools
  - archaic
- CMake
  - build language sucks
- Meson
- Soong
  - for Android

Sampling: Multi-language

GNU Autotools: Autoconf, Automake, and `./configure`

- Part of the "GNU Build System," along with Make
- Designed to abstract away differences in shells, compilers, kernels, system packages, and most other things you can imagine
- Vastly overcomplicated by modern standards
- Processes input files named `configure.ac` and `Makefile.am` into a plethora of intermediate files
- Intermediate shell script named `configure` generates a Makefile
  - By convention, a script named `autogen.sh` creates `configure` when it's not present.
- `./configure && make` will build most projects that use Autotools

CMake

Generates either Makefile or Ninja file.

Abstracts away OS and platform differences.

```
$ cmake -GNinja -B build \
  -DCMAKE_BUILD_TYPE=DebugOpt
$ ninja -C build
$ rm -r build # clean
or
$ ninja -C build clean
```

CMakeLists.txt

```
cmake_minimum_required(VERSION 3.16)
project(BuildDemo)
SET(CMAKE_C_FLAGS  "${CMAKE_C_FLAGS} -Wall -Wextra")

add_executable(main main.c)
target_link_libraries(main PUBLIC log)

add_library(log OBJECT log.c log.h)
```

Architectural note: we make `log.c` its own library, since we don't want each thing that depends on it to care how it's implemented.

This is CMake's way of specifying optimization flags, including -O2. We also don't need to specify -Wall and -Wextra, since CMake adds them by default.

Meson

Supports C, C++, D, Fortran, Rust, and others.

Doesn't run builds: emits Ninja, Visual Studio, or XCode rule definitions.

Out-of-tree builds are default and required.

Automatically gets C header dependencies from compiler.

```
$ meson setup somedir
$ ninja -C somedir
$ rm -r somedir # clean
```

meson.build (for C)

```
project('myproject', 'c',
  default_options: [
      'c_std=gnu99',            # Adds -std=gnu99
      'warning_level=2',        # Adds -Wall, -Wextra
      'optimization=2',         # Adds -O2
      'default_library=static', # Links libraries statically
  ],
)

log_lib = library('log', 'log.c')

executable('main', 'main.c', link_with: log_lib)
```

Automatically adds the library source directory as an include path for dependents.

Can be a list

meson.build (for Rust)

```
project('myproject', 'rust')

executable('main', 'src/main.rs')
```

Don't need to mention `src/log.rs`, since the Rust compiler, rustc, is what resolves `mod` statements pointing to other files in the same crate.

Bazel

Built by Google -- open source version of their internal tool "Blaze". Meant to build millions of files across a large monorepo.

Can take advantage of multiple processes, machines.

Runs a server in the background.

```
$ bazel build //:main
$ bazel clean # clean
```

BUILD

```
common_flags = ["-Wall", "-Werror", "-Wextra", "-pedantic"]

cc_binary(
      name="main",
      srcs=["main.c"],
      copts=common_flags,
      deps=[":logger"],
)

cc_library(
      name="logger",
      srcs=["log.c"],
      hdrs=["log.h"],
      copts=common_flags,
)
```

Other language-agnostic build systems

- SCons
- Rake
- Ninja
  - hard to write by hand
- Pants
- Just
- Please
- tup
  - leaves state on disk
- redo
  - many files spread out

Sampling: Language-specific

Pip

Package download & install

Version solving

Dependency isolation

Kind of hard to use

```
$ cat requirements.txt
pyOpenSSL==0.13.1
pyparsing==2.0.1
python-dateutil==1.5
pytz==2013.7
scipy==0.13.0b1
six==1.4.1
virtualenv==16.3.0
$ python3 -m pip install -r requirements.txt
...
$
```

setuptools/distutils

Bundle Python files into distributions like .tar.gz, .whl

```
$ python3 -m pip install --upgrade pip build twine

packaging_tutorial/
├── LICENSE
├── pyproject.toml
├── README.md
├── setup.cfg
├── src/
│   └── mypackage/
│       ├── __init__.py
│       └── example.py
└── tests/

$ python3 -m build
$ python3 -m twine upload --repository testpypi dist/*
$ python3 -m pip install mypackage
$ python3
>>> from mypackage import example
>>> example.add_one(4)
5
>>>
$
```

poetry

Package download & install

Create packages

Version solving

Dependency isolation

```
$ python3 -m pip install --upgrade poetry
$ poetry new packaging_tutorial

packaging_tutorial/
├── pyproject.toml
├── README.rst
├── mypackage/
│   ├── __init__.py
│   └── example.py
└── tests/

$ poetry build
$ poetry config repositories.testpypi https://test.pypi.org/legacy/
$ poetry publish -r testpypi
$ python3 -m pip install mypackage
$ python3
>>> from mypackage import example
>>> example.add_two(4)
6
>>>
$
```

Cargo

Build system and package manager for Rust.

Most rules are auto-discovered from source files and directory structure.

Cargo.toml config file often only includes basic metadata and a list of dependencies, which Cargo downloads and makes available.

```
$ cargo build
$ cargo clean # clean
```

Cargo.toml

```
[package]
name = "mypackage"
version = "1.0.0"
edition = "2021"

# [dependencies] section can declare external packages.

# All .rs files inside src/ are built. main.rs is a special
# name that includes functions at the crate root.
```

src/main.rs

```
mod log;

fn main() {
  log::log_message("Hello, world!");
}
```

src/log.rs

```
pub fn log_message(msg: &str) {
  eprintln!("{}", msg);
}
```

Web things

For when you want a task runner. Or a JS->JS compiler. Or an asset minification pipeline. Or a combination of the above.

- Webpack
- Grunt
- Gulp
- Babel

Disclaimer: neither of us are very familiar with frontend dev tooling for web.

The package manager-build system continuum

- Sometimes hard to tell where "build system" stops and "package manager" starts
- Is fetching dependencies part of the build system's job?
- What about reconciling versions?
- Sometimes the tools need to work together

http://blog.ezyang.com/2015/12/the-convergence-of-compilers-build-systems-and-package-managers/

Build systems that also manage packages

- npm (JS/TS)
- Cargo (Rust)
- Cabal/Stack (Haskell)
- go build (Go)
- setuptools/pip/poetry (Python)
- ant/maven/gradle (Java/Scala/Kotlin)
- all the web ones (webpack, grunt, gulp, babel, …) (JS)
- Dune/opam (OCaml)

How to choose a build system

If your language has its own package ecosystem
Use whatever the currently-recommended build tool for integrating with that ecosystem is (see previous slide).

If your project is small and you want it to be buildable on any POSIX system
Use Make. The Make implementation on BSD and macOS isn't GNU Make, so for broadest compatibility, only use POSIX Make syntax.

If your code belongs to a project or company with established tooling
Use that. The benefits of doing your own thing are almost never worth it.

The ~*environment*~ is not clean

- Hermetic and reproducible builds
- Debian reproducible build efforts
- NixOS

"Build systems" that build entire OS images

Containers / VMs

- Docker
- Vagrant

Provisioning software

- Ansible
- Chef
- Puppet
- Terraform

Hermetic environments

- Python virtualenv
- Ruby rbenv
- JS npm/yarn/esy

Parting words about build systems

Tools for adjacent problems

- Why rebuild files if only their mtime changed? ccache
- What happens if you have enormous amounts of software that take too long to compile on one computer? distcc/icecc
- Software breaks frequently? Run builds with every change on CI

It's not just about building programs

- Distributed rendering
- Machine learning