D and buck2

Build systems

- Automate our processes
- Capture dependency information

Examples

- `make`
- But also Microsoft Excel.

Many, many, more

- All subtly different.
- Excellent paper called "build systems à la carte"

My point

- Much discussion of faster horses in this space.
- Untapped value in not trying to copy (say) npm, or even CMake.
- Analogy with testing - tests are a collection of assumptions about program behaviour rather.
- Our build systems can be more "general" in this sense, but currently aren't.
- Look towards interesting alternatives.

Common problems.

Some patterns that often happen in projects:

- Slow. Doing too much work.
- Tests: Often brittle, but under powered, poor feedback loops
- Reliant on host machine being just-so.
- "Bits versus Atoms" but just within the bits.

Why

"Terminal complexity bubble crisis"

- We aren't being strict enough. There isn't enough information for tools to use.
- `make` underrated, makes you write stuff down.
- Some patterns encourage bloat e.g. hormesis
- The abstractions underlying the tools are also weak.
- `make` overrated, lots of typing.

A solution

- People have thrown money at solving this problem before.
- So-called "declarative build systems".
- A handful exist (in public), in particular Bazel, buck2, and pants.

Input looks like this

```
cxx_library(
    name = "foo",
    src = glob(["src/lib/*.cpp"])
)

d_binary(
    name = "program",
    src = "src/main.d"
    deps = [:foo]
)
# and so on
```

Output is ...

- object files and executables. Shocker.
- But also (say) tests, as part of our build graph.

What's the difference.

- Explicit
- Enforcing rules under the hood: Builds should be hermetic.
- e.g. Use a file you don't say you need -> fail.

Questions we can now ask, things we can do

- *Exactly* which files could this rule access
- For this diff which rules do we run, without having already run it e.g. faster test suites
- Have we built this before: Cache.
-

Why buck2 quickly.

Why buck2 now:

- buck2 is relatively new, the others are priced in.
- Someone, not entirely sure who, has already done D rules for bazel.

Why buck2 in general vs Bazel:

- Cleaner theoretical model. Bazel splits builds into three phases, buck2 hides this.
- buck2 has no rules built in.
- buck2 is a single static binary (afaict bazel isn't)
- buck2 starlark can be statically type checked.

How do we teach buck2 new tricks

- Starlark language.
- `rules`, `providers` and so on.
- Rules are passed an `AnalysisContext`, output `DefaultInfo()`, `RunInfo()` and so on.
- We have to write everything down.
- Good and bad to this. Hormesis.
- Ogilvy on advertising.

A starlark example

- Actually not building anything.
- Format / lint check
- Why is this not usually part of the build system?

Rules for a javascript linter.

```
BiomeToolchain = provider(
    fields = {
        "biome_binary": provider_field(RunInfo)
    }
)

def _biome_toolchain_impl(ctx: AnalysisContext) -> list[Provider]:
    urlToFetch = ctx.attrs.biome_url
    shaShouldBe = ctx.attrs.biome_sha256

    downloadTo = ctx.actions.declare_output(ctx.label.name)
    ctx.actions.download_file(downloadTo, urlToFetch, sha256 = shaShouldBe, is_executable = True)

    return [DefaultInfo(), BiomeToolchain(
        biome_binary = RunInfo(args = [downloadTo])
    )]

biome_toolchain = rule(
    impl = _biome_toolchain_impl,
    attrs = {
        "biome_url": attrs.string(),
        "biome_sha256": attrs.string()
    },
    is_toolchain_rule = True
)
```

We then use like this:

```
load("@rules//biome_linter.bzl", "biome_toolchain")

biome_toolchain(
    name="biome_toolchain",
    biome_url = "https://github.com/biomejs/biome/releases/download/cli%2Fv1.9.1/biome-linux-x64",
    biome_sha256 = "931aa434bdee3aca1ddb3119e97f1028b0b11cdc206107d9415e537f4dd8e27f",
    visibility = ["PUBLIC"]
)
```

- Note the integrity check.

Running the tool now we've downloaded it

```
def _biome_check_impl(ctx: AnalysisContext) -> list[Provider]:
    runThis = ctx.attrs.biome_toolchain[BiomeToolchain].biome_binary
    onThis = ctx.attrs.file

    return [DefaultInfo(), ExternalRunnerTestInfo(
        type = "format",
        command = [runThis, "ci", onThis]
    )]

biome_check = rule(
    impl = _biome_check_impl,
    attrs = {
        "file": attrs.source(),
        "biome_toolchain": attrs.toolchain_dep(default = "toolchains//:biome_toolchain", providers = [BiomeToolchain])
    }
)
```

To use it:

```
load("@rules//biome_linter.bzl", "biome_check")

biome_check(
    name = "lint_js",
    file = "src/file.js"
)
```

To run:

```
buck2 test :lint_js
```

- We can also query for all rules touching js of `kind == "format"`

Output

```
File changed: root//.buckconfig
X Fail: root//:lint_js (0.4s)
---- STDOUT ----
Checked 1 file in 110ms. No fixes applied.
Found 1 error.

---- STDERR ----
src/file.js:1:1 lint/style/useConst FIXABLE ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━

  ✖ This let declares a variable that is only assigned once.

  > 1 | let x = 0;
      | ^^^
    2 |
    3 | function ClosesOver() {

  i 'x' is never reassigned.

  > 1 | let x = 0;
      |     ^
    2 |
    3 | function ClosesOver() {

  i Safe fix: Use const instead.

    1   | - let·x·=·0;
      1 | + const·x·=·0;
    2 2 |
    3 3 |   function ClosesOver() {


ci ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━

  ✖ Some errors were emitted while running checks.



Build ID: e41c0864-f740-414e-bcbc-34dfb7c600cd
Network: Up: 0B  Down: 27MiB
```

Why not:

- A google search yields a link to the article:
  "Why Declarative Build Systems Aren’t Popular"
- "Not built for open source." This is basically fair.
- "Closely coupled with monorepo architecture." Also fair.
- "Not helpful or detrimental for small projects." Also fair.

First and last points aren't anywhere near as bad with buck2 than bazel IMO.

In particular the on-ramp for a new project should be (in theory) much smoother if one willing to be creative.