Yet Another CROSSTOOL Writing Tutorial

Chip Hogg edited this page on 11 Dec 2017 · 8 revisions

Let's say that for some reason we need to configure our C++ toolchain in Bazel. This is currently quite a frustrating task. To quote one of many:

> This is a fabulously difficult project that causes hardened engineers to stare blankly at screens in defeat.

Hopefully this wiki page will improve the situation. I'll try to do this the incremental, one error at a time way.

DISCLAIMER: We are very well aware that specifying C++ toolchains in Bazel is suboptimal for many reasons and we are actively working on improving the story. We hope that this wiki page will be soon obsolete and replaced by something much nicer to work with. In the meantime, bear with us, and don't hesitate to ask on bazel-discuss if you struggle with something.

Goal

We have our C++ application that we already build with gcc, clang, and msvc. We need to build it with emscripten as well. We want to be able to run:

```
bazel build --config=asmjs test/helloworld.js
```

Pages 9

Wiki

- Who is using Bazel?
- Project Ideas

Website

- Docs
- Contribute

Connect

- Blog
- Stack Overflow
- Twitter

Clone this wiki locally

https://github.com/bazelb

on a linux machine and we expect to get our application built using emscripten targeting asm.js.

Workspace

We'll start with an empty bazel workspace - empty WORKSPACE and BUILD files. To be able to use `--config` likewe desire to, we need to add `.bazelrc` :

```
# .bazelrc

# We will use new CROSSTOOL file for our toolchain
build:asmjs --crosstool_top=//toolchain:emscripten

# Since emscripten can target both asmjs and web assembly,
# we'll use --cpu as differentiator. We will not configure
# web assembly toolchain in this tutorial though.
build:asmjs --cpu=asmjs

# Bazel uses internal tools many of which are written in
# C++ (such as `process-wrapper`). Therefore we still need a sane
# C++ toolchain for these tools.
build:asmjs --host_crosstool_top=@bazel_tools//tools/cpp:toolchain
```

This is our application:

```
// helloworld.cc
#include <stdio.h>

int main() {
  printf("Hello, World!\n");
  return 0;
}
```

And the corresponding entry in the BUILD file:

```
# BUILD
cc_binary(
    name = "helloworld.js",
    srcs = ["helloworld.cc"],
)
```

Configuring C++ toolchain

Now we'll repeatedly run `bazel build --config=asmjs helloworld.js`, showing the error message, fixing the problem, and trying again.

```
no such package 'toolchain': BUILD file not found on package path
```

We specify `--crosstool_top=//toolchain:emscripten` in the `.bazelrc`. Package `toolchain` doesn't yet exist. We fix it by creating a directory and empty BUILD file in it.

```
no such target '//toolchain:emscripten': target 'emscripten' not declared in
package 'toolchain'
defined by .../toolchain/BUILD
```

The package `toolchain`, exists, but it doesn't define `emscripten` target. Let's make things simple for now by defining an empty filegroup:

```
# toolchain/BUILD
package(default_visibility = ['//visibility:public'])

filegroup(name = "emscripten")
```

```
The crosstool_top you specified was resolved to '//toolchain:emscripten',
which does not contain a CROSSTOOL file.
```

Currently we don't specify `CROSSTOOL` file as an input of any rule, bazel just assumes that whatever you set `--crosstool_top` to, there will be a `CROSSTOOL` file in that package. Let's add an empty one.

```
Could not read the crosstool configuration file 'CROSSTOOL
file .../toolchain/CROSSTOOL', because of an incomplete protocol
buffer (Message missing required fields: major_version, minor_version,
default_target_cpu)
```

Bazel reads content of the `CROSSTOOL` file but is not pleased. Let's add only those required fields.

```
# toolchain/CROSSTOOL
major_version: "1"
minor_version: "0"
# Legacy, deprecated field, that hlopko@ should finally remove.
default_target_cpu: "asmjs"
```

```
No toolchain found for cpu 'asmjs'. Valid cpus are: []
```

In the `.bazelrc` we specify `--crosstool_top` and `--cpu`. In the most advanced case we would also specify `--glibc` and `--compiler`, forming a triplet based on which bazel selects the `toolchain` from CROSSTOOL. In the case of full triplet, the error message would be `No toolchain found for --cpu='asmjs' --compiler='emscripten' --glibc='unknown'. Valid toolchains are: []`. Let's fix it by adding `toolchain` section into `CROSSTOOL`:

```
# toolchain/CROSSTOOL
# ...
toolchain {
    toolchain_identifier: "asmjs-toolchain"
    host_system_name: "i686-unknown-linux-gnu"
    target_system_name: "asmjs-unknown-emscripten"
    target_cpu: "asmjs"
    target_libc: "unknown"
    compiler: "emscripten"
    abi_version: "unknown"
    abi_libc_version: "unknown"
}
```

With this bazel will correctly consume the triplet. If we insist on providing only `--cpu`, we need to add the defaults:

```
# toolchain/CROSSTOOL
# ...
default_toolchain {
    cpu: "asmjs"
    toolchain_identifier: "asmjs-toolchain"
}
```

`toolchain_identifier`s from `default_toolchain` and `toolchain` must match. There is no other use for these fields.

```
The specified --crosstool_top '//toolchain:emscripten' is not a valid
cc_toolchain_suite rule
```

Bazel finally discovered that our `--crosstool_top` doesn't point to the `cc_toolchain_suite` rule. We have to remove the empty filegroup and add `cc_toolchain_suite` instead:

```
# toolchain/BUILD
# ...
cc_toolchain_suite(
    name = "emscripten",
    toolchains = {
         "asmjs|emscripten": ":asmjs_toolchain",
    },
)
```

Magically looking `toolchains` attribute just maps cpu and compiler values to cc_toolchains. This is needed to map what BUILD file sees with what is in the CROSSTOOL. We have no `cc_toolchain` yet as bazel reminds us right away.

```
The label '//toolchain:asmjs_toolchain' is not a cc_toolchain rule
```

Now we come to the important milestone. We need to define `cc_toolchain` targets for all toolchains in the CROSSTOOL. This is where we need to specify all the files that the toolchain consists of, so bazel can correctly setup the sandbox. Let's start with all empty toolchain.

```
# toolchain/BUILD
# ...
filegroup(name = "empty")
cc_toolchain(
    name = "asmjs_toolchain",
    all_files = ":empty",
    compiler_files = ":empty",
    cpu = "asmjs",
    dwp_files = ":empty",
    dynamic_runtime_libs = [":empty"],
    linker_files = ":empty",
    objcopy_files = ":empty",
    static_runtime_libs = [":empty"],
    strip_files = ":empty",
    supports_param_files = 0,
)
```

```
.../BUILD:1:1: C++ compilation of rule '//:helloworld.js' failed (Exit 1)
src/main/tools/linux-sandbox-pid1.cc:421:
    "execvp(toolchain/DUMMY_GCC_TOOL, 0x11f20e0)": No such file or directory
Target //:helloworld.js failed to build
```

At this point bazel is happy enough to try to compile things. At the time of writing this wiki page we were in the process of updating the old, legacy way of specifying toolchains in the CROSSTOOL using fields like `compiler_flag` or `linker_flag` into new, `action_config` and `feature` - algebra based configuration. For the migration purpose, Bazel inspects all the tools defined by the CROSSTOOL and used them to initialize what we call feature configuration. Let's talk about details some other day. To more forward, we need to give bazel little bit more information about the toolchain by describing tools that it can use.

```
# toolchain/CROSSTOOL
# ...
tool_path {
    name: "gcc"
    path: "emcc.sh"
}
tool_path {
    name: "ld"
    path: "emcc.sh"
}
tool_path {
    name: "ar"
    path: "/bin/false"
}
tool_path {
    name: "cpp"
    path: "/bin/false"
}
tool_path {
    name: "gcov"
    path: "/bin/false"
}
tool_path {
    name: "nm"
    path: "/bin/false"
}
tool_path {
    name: "objdump"
    path: "/bin/false"
}
tool_path {
    name: "strip"
    path: "/bin/false"
}
```

As you noticed, I already expect to use wrapper script named `emcc.sh`. Relative paths in the CROSSTOOL are usually relative to the folder containing the CROSSTOOL. Let's start with naive wrapper script (don't forget to set executable flag):

```
#!/bin/bash
# toolchain/emcc.sh
set -euo pipefail

# Run emscripten to compile and link
python external/emscripten_toolchain/emcc.py "$@"
```

Here we delegate to the `external/emscripten_toolchain/emcc.py` file. But this file doesn't exist yet. It's distributed with the emscripten toolchain, for example here. We have 2 options:

1. check in the emscripten toolchain
2. use repositories and download emscripten toolchain from the web

Both have their pros and cons, and in this tutorial I'll use repositories. We'll also create a repository for clang distribution that has support for emscripten enabled. Our `WORKSPACE` file will look like this:

```
# WORKSPACE
new_http_archive(
    name = 'emscripten_toolchain',
    url = 'https://github.com/kripken/emscripten/archive/1.37.22.tar.gz',
    build_file = 'emscripten-toolchain.BUILD',
    strip_prefix = "emscripten-1.37.22",
)

new_http_archive(
    name = 'emscripten_clang',
    url = 'https://s3.amazonaws.com/mozilla-
games/emscripten/packages/llvm/tag/linux_64bit/emscripten-llvm-e1.37.22.tar.gz',
```

```
    build_file = 'emscripten-clang.BUILD',
    strip_prefix = "emscripten-llvm-e1.37.22",
)
```

In the real world we would want to only include files that are needed during the build, and we would split them depending on whether they are needed for compilation, linking, stripping, etc. Here we'll keep things simple by including all files ditributed in archives.

```
# emscripten-toolchain.BUILD
package(default_visibility = ['//visibility:public'])

filegroup(
    name = "all",
    srcs = glob(["**/*"]),
)
```

and

```
# emscripten-clang.BUILD
package(default_visibility = ['//visibility:public'])

filegroup(
    name = "all",
    srcs = glob(["**/*"]),
)
```

```
"execvp(toolchain/emcc.sh, 0x12bd0e0)": No such file or directory
```

Surprise. We added the `emcc.sh` file, we added emscripten distribution, be we didn't tell bazel about them. We mentioned `emcc.sh` only in the CROSSTOOL, not in the `BUILD` file. Using a tool in the CROSSTOOL will not add the tool as a dependency of the corresponding `cc_toolchain`. Our new toolchain/BUILD will look like this:

```
# toolchain/BUILD
package(default_visibility = ['//visibility:public'])

cc_toolchain_suite(
    name = "emscripten",
    toolchains = {
        "asmjs|emscripten": ":asmjs_toolchain",
    },
)

filegroup(name = "empty")
filegroup(
    name = "all",
    srcs = [
        "emcc.sh",
        "@emscripten_toolchain//:all",
        "@emscripten_clang//:all"
    ],
 )

cc_toolchain(
    name = "asmjs_toolchain",
    all_files = ":all",
    compiler_files = ":all",
    cpu = "asmjs",
    dwp_files = ":empty",
    dynamic_runtime_libs = [":empty"],
    linker_files = ":all",
    objcopy_files = ":empty",
    static_runtime_libs = [":empty"],
```

```
    strip_files = ":empty",
    supports_param_files = 0,
)
```

```
ERROR: .../BUILD:1:1: C++ compilation of rule '//:helloworld.js' failed (Exit 1)
Traceback (most recent call last):
File "external/emscripten_toolchain/emcc.py", line 32, in <module>
  from tools import shared, jsrun, system_libs
File ".../external/emscripten_toolchain/tools/shared.py", line 959, in <module>
  check_vanilla()
File ".../external/emscripten_toolchain/tools/shared.py", line 935, in
check_vanilla
  is_vanilla_file = temp_cache.get('is_vanilla', get_vanilla_file,
extension='.txt')
File ".../external/emscripten_toolchain/tools/cache.py", line 90, in get
  self.acquire_cache_lock()
File ".../external/emscripten_toolchain/tools/cache.py", line 43, in
acquire_cache_lock
  self.filelock.acquire(60)
File ".../external/emscripten_toolchain/tools/filelock.py", line 242, in acquire
  self._acquire()
File ".../external/emscripten_toolchain/tools/filelock.py", line 362, in _acquire
  fd = os.open(self._lock_file, open_mode)
OSError: [Errno 30] Read-only file system: '/home/hlopko/.emscripten_cache.lock'
```

Congratulations, we are finally using emscripten toolchain to compile C++. Our tutorial is almost over, but in the real world now would be when the most of the work begins. We need to persuade the toolchain itself to be deterministic, to only touch files it is supposed to, and make sure it doesn't write temp files outside of the sandbox. We have to make sure it doesn't assume the existence of your `HOME` directory with configuration files, or it doesn't depend on unspecified environment variables. I'm not going to explain emscripten specifics (and I'm not in any means an emscripten expert so I mostly have no idea what I did to make it work :) I'll just provide my hacks that allowed me to compile and link the hello world.

Emscripten uses a cache for standard library files. To not waste time compiling stdlib for every action, and to prevent it storing temporary files who knows where, I checked in precompiled bitcode files into `toolchain/emscripten_cache`. They were created by calling `embuilder.py build dlmalloc libcxx libc gl libcxxabi libcxx_noexcept wasm-libc` in the `emscripten_clang` repository. I updated the `toolchain/BUILD` file to also add these files into sandbox.

```
# toolchain/BUILD
filegroup(
    name = "all",
    srcs = [
        "emcc.sh",
        "@emscripten_toolchain//:all",
        "@emscripten_clang//:all",
        ":emscripten_cache_content"
    ],
)

filegroup(
    name = "emscripten_cache_content",
    srcs = glob(["emscripten_cache/**/*"]),
)
```

Then I had to update our `emcc.sh` wrapper. In addition to setting up the cache it also configures various bits of emscripten:

```
#!/bin/bash
# toolchain/emcc.sh

set -euo pipefail

export LLVM_ROOT='external/emscripten_clang'
export EMSCRIPTEN_NATIVE_OPTIMIZER='external/emscripten_clang/optimizer'
export BINARYEN_ROOT='external/emscripten_clang/'
export NODE_JS=''
export EMSCRIPTEN_ROOT='external/emscripten_toolchain'
export SPIDERMONKEY_ENGINE=''
export EM_EXCLUSIVE_CACHE_ACCESS=1
export EMCC_SKIP_SANITY_CHECK=1
export EMCC_WASM_BACKEND=0

mkdir -p "tmp/emscripten_cache"
export EM_CACHE="tmp/emscripten_cache"
export TEMP_DIR="tmp"

# Prepare the cache content so emscripten doesn't try to rebuild it all the time
cp -r toolchain/emscripten_cache/* tmp/emscripten_cache
# Run emscripten to compile and link
python external/emscripten_toolchain/emcc.py "$@"
# Remove the first line of .d file (emscripten resisted all my attempts to make
# it realize it's just the absolute location of the source)
find . -name "*.d" -exec sed -i '2d' {} \;
```

Now bazel should be able to compile our helloworld.cc, but... :)

```
.../BUILD:1:1: undeclared inclusion(s) in rule '//:helloworld.js':
```

```
this rule is missing dependency declarations for the following files included by
'helloworld.cc':
  '.../external/emscripten_toolchain/system/include/libcxx/stdio.h'
  '.../external/emscripten_toolchain/system/include/libcxx/__config'
  '.../external/emscripten_toolchain/system/include/libc/stdio.h'
  '.../external/emscripten_toolchain/system/include/libc/features.h'
  '.../external/emscripten_toolchain/system/include/libc/bits/alltypes.h'
```

At this point we successfully compiled our source. This error message is shown because bazel uses `.d` file produced by the compiler to verify that all includes were declared (and of course to prune action inputs, but that is not important right now). And in this `.d` file bazel discovered that our source included some headers that we didn't declare in the BUILD file. In this case there is nothing wrong with it, these are system headers that we don't want to explicitly declare all the time. To fix it, we add these folders as `-isystem` directories by appending this:

```
compiler_flag: "-isystem"
compiler_flag: "external/emscripten_toolchain/system/include/libcxx"
compiler_flag: "-isystem"
compiler_flag: "external/emscripten_toolchain/system/include/libc"
```

into our toolchain in the CROSSTOOL. And with this last change, we finally compile and link. \o/

A lot is still missing. We cannot produce libraries with this toolchain yet (we use /bin/false for archiver, but emar.sh should be used instead). I'm not even sure if emscripten supports shared libraries. We don't specify any custom flags for compilation or linking. From my experience what we went through in this tutorial is *the* frustrating part of dealing with crosstool. Specifying flag using `compiler_flag` or `linker_flag` and friends (or if you live at the edge using action_configs and features) is much easier, since we at least have something that works and we can improve it incrementally.

But isn't it cool that "all" (very big quotes :) we need to do to support new toolchain or to target new platform is to write the CROSSTOOL? Of course in the real world it is more complicated than that, but bazel does allow you to encapsulate compilation and linking specifics of a platform solely in the CROSSTOOL.

+ Add a custom footer