Fuchsia

LLVM Runtimes Build

phosek@google.com

C-family Language Toolchain

Tools
Compiler (Clang)
Assembler (LLVM)
Linker (LLD)

Runtimes
Compiler runtimes (compiler-rt)
C standard library (LLVM libc)
Unwind library (LLVM libunwind)
C++ ABI library (libc++abi)
C++ standard library (libc++)

3 ways to build LLVM runtimes

1. External build
2. LLVM projects build
3. LLVM runtimes build

External build

This means building each runtime separately, typically using custom scripts (see for example Android and Chromium).

This gives you a lot of flexibility, but also requires a lot of maintenance.

It involves some duplication across projects.

LLVM projects build

This means building selected runtimes by listing them in `-DLLVM_ENABLE_PROJECTS="<name>;..."`

In this case, the runtimes are built as part of the LLVM build using the host compiler.

It's simple, straightforward... and if you're not careful also dangerous. There's no guarantee that runtimes built by one compiler will work with another one.

LLVM runtimes build

This means building selected runtimes by listing them in `-DLLVM_ENABLE_RUNTIMES="<name>;..."`

In this case, the runtimes are built as part of the LLVM build using the compiler just built.

This guarantees correct ABI between the compiler and the runtime.

It involves the use of child CMake builds which introduces some additional overhead and complexity.

Example usage

Projects build

```
cmake -G Ninja -S llvm-project/llvm -B build \
  -DLLVM_ENABLE_PROJECTS="clang;libcxx;..."
```

Runtimes build

```
cmake -G Ninja -S llvm-project/llvm -B build \
  -DLLVM_ENABLE_PROJECTS="clang;..." \
  -DLLVM_ENABLE_RUNTIMES="libcxx;..."
```

Support for multiple targets

Building a cross-compiling toolchain can be challenging and often involves custom scripts.

We need to build runtimes in the right order for the target platform with the just-built Clang and LLVM tools.

LLVM runtimes build supports cross-compiling runtimes for multiple targets (and multilibs).

Each target has a separate child CMake build with its own set of flags.

Builtins

We need to build compiler-rt builtins first.

```
set(targets "x86_64-fuchsia;aarch64-fuchsia")

foreach(target ${targets})
  set(BUILTINS_${target}_CMAKE_SYSROOT
      "${FUCHSIA_${target}_SYSROOT}" CACHE PATH "")
  set(BUILTINS_${target}_CMAKE_SYSTEM_NAME
      "Fuchsia" CACHE STRING "")
endforeach()

set(LLVM_BUILTIN_TARGETS
    "${targets}" CACHE STRING "")
```

See Fuchsia-stage2.cmake

Runtimes

We can build the rest of runtimes second.

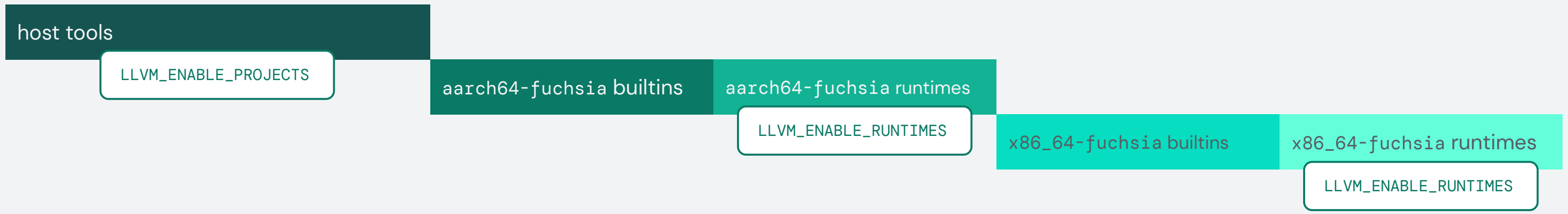
```
set(targets "x86_64-fuchsia;aarch64-fuchsia")

foreach(target ${targets})
  set(RUNTIMES_${target}_CMAKE_SYSROOT
      "${FUCHSIA_${target}_SYSROOT}" CACHE PATH "")
  set(RUNTIMES_${target}_CMAKE_SYSTEM_NAME
      "Fuchsia" CACHE STRING "")
  set(RUNTIMES_${target}_LIBCXX_ABI_VERSION
      "2" CACHE STRING "")
  ...
endforeach()

set(LLVM_RUNTIME_TARGETS
    "${targets}" CACHE STRING "")
```

See Fuchsia-stage2.cmake

Build stages



Support for multilibs

Multilib enables the use of different application binary interfaces (ABIs) on the same target.

The multilib support in the runtimes build uses separate library directories for non-native ABIs.

Multilibs

Multilib ABI builds inherit flags from the native one.

```
set(targets "x86_64-fuchsia;aarch64-fuchsia")

foreach(target ${targets})
  set(RUNTIMES_${target}+asan_LLVM_USE_SANITIZER
      "Address" CACHE PATH "")
  ...
endforeach()

set(LLVM_RUNTIME_MULTILIBS
    "asan" CACHE STRING "")
set(LLVM_RUNTIME_MULTILIB_asan_TARGETS
    "${targets}" CACHE STRING "")
```

See Fuchsia-stage2.cmake

Per-target runtime layout

We need a way to install runtimes for all enabled targets side-by-side.

`-DLLVM_ENABLE_PER_TARGET_RUNTIME_DIR` enables the use of multiarch layout to achieve that.

This option is enabled by default with the runtimes build.

Layout

We use the normalized target names.

```
bin/
  clang
include/
  c++/v1/
  aarch64-unknown-fuchsia/
    c++/v1/
      __config_site
lib/
  clang/14.0.0/
    include/
    lib/
     aarch64-unknown-fuchsia/
       libclang_rt.builtins.a
  aarch64-unknown-fuchsia/
    libc++.so
    asan/
      libc++.so
```

What could be improved?

Reuse CMake check results across runtimes.

Combine builtins and runtimes build by using custom CMake checks that don't rely on builtins being ready.

Leverage `CMAKE_NINJA_OUTPUT_PATH_PREFIX` to avoid Ninja having to invoke child Ninja builds.

Use Ninja Multi-Config for multilibs without requiring separate CMake builds.

External build
Flexible but complex.

LLVM projects build
Simple but limited (and potentially dangerous).

LLVM runtimes build
Simple and powerful.

Summary

Questions?

Please post them to git.io/Jud1j

Fuchsia toolchain team is looking for a software engineer to help us with runtime testing.
Please reach out if you interested in solving problems similar to the ones covered in this talk.