Build System

Introduction

Modularising the build of Swift is a large undertaking that will simplify the infrastructure used to build the compiler, runtime, standard library, as well as the core libraries. Additionally, the work will enable a unified, uniform build system that reduces the barrier to entry for new contributors.

Current State

Currently, the Swift build is setup similar to the way that the GCC build was setup. This requires the understanding of the terms `build`, `host`, `target` as used by autotools.

build

the system where the package is being configured and compiled
x86_64-unknown-linux-gnu

host

the system where the package runs (by default the same as *build*)
x86-64-unknown-windows-msvc

target

the system for which the compiler tools generate code
aarch64-unknown-linux-android
NOTE this is not really supported properly in the original build system

Desired State

It is preferable to instead consider the full package build as a set of builds where you are doing normal builds with a differing set of `build` and `host` values. That is, you can consider the regular host build (e.g. Linux) as:

```
[
  /* compiler */ (build: "x86_64-unknown-linux-gnu", host: "x86_64-unknown-linux-
gnu"),
  /* runtime */  (build: "x86_64-unknown-linux-gnu", host: "x86_64-unknown-linux-
gnu"),
  /* stdlib */   (build: "x86_64-unknown-linux-gnu", host: "x86_64-unknown-linux-
gnu"),
]
```

Or for more complicated scenarios such as Darwin where you may be compiling for Darwin and iOS as:

```
[
  /* compiler */ (build: "x86_64-apple-macosx10.14", host: "x86_64-apple-
macosx10.14"),
  /* runtime */  (build: "x86_64-apple-macosx10.14", host: "x86_64-apple-
macosx10.14"),
  /* stdlib */   (build: "x86_64-apple-macosx10.14", host: "x86_64-apple-
macosx10.14"),
  /* runtime */  (build: "x86_64-apple-macosx10.14", host: "armv7-apple-ios12.3"),
  /* stdlib */   (build: "x86_64-apple-macosx10.14", host: "armv7-apple-ios12.3"),
]
```

This simplifies the build terminology by having to only deal with the terms `build` and `host` which removes some of the confusion caused by the original build system's implementation creating multiple terms and being designed around the needs of the Darwin build.

This also generalises to allow for multiple cross-compilations in a single build invocation which allows the functionality currently available only on Darwin to be applied to all the targets that Swift supports (and may support in the future).

Doing so also enables the build system to be trimmed significantly as we can use CMake as it was designed to while retaining the ability to cross-compile. The cross-compilation model for CMake involves invoking `cmake` multiple times with `CMAKE_SYSTEM_NAME` and `CMAKE_SYSTEM_PROCESSOR` set appropriately to the host that you would like to build for. This ensures that host specific behaviour is explicitly handled properly (e.g. import libraries on Windows - something which was shoehorned into the original CMake implementation in CMake) and corrects the behaviour of the cmake `install` command.

Another bit of functional flexibility that this system enables is the ability to allow developers to focus entirely on the area that they are working in. The runtime and standard library can be separated and built with a prebuilt (nightly) release of the compiler toolchain, or focus entirely on building the standard library for a certain set of targets.

By organising the standard library build as a regular library, we enable the ability to use the LLVM runtimes build to cross-compile the standard library for multiple targets as long as the dependent system libraries are available at build time.

The ability to build the runtime components for different hosts allows building the Swift SDK for various hosts on a single machine. This enables the build of the android SDK and Linux SDKs on Windows and vice versa.

Note that none of the changes described here prevent the workflows that are possible with build-script today.