Intro To Cross Compilation

Intro To Cross Compilation

What is compilation?

We need a compiler to translate code that we write in a **high level language** (say C, C++, or go) to a **low level language** (usually machine code)

```
┌──────────────────────┐  compiler  ┌──────────────────────┐
│ High Level Language  │ ──────────> │ Low Level Language   │
└──────────────────────┘             └──────────────────────┘
```

An example

As an example, here's a simple compilation of the following C program

```
path: example1.c
lang: c
```

An example (continued)

We use the compiler `gcc` here, present on most Linux systems

More about it later

```
command: sh -il
rows: 3
init_text: gcc -g -c example1.c -o example1
init_wait: "1$"
```

Now we get the compiled program `example1`

```
ls
```

Note:

Here I've just compiled the program, I haven't linked external libraries to it, or made it into an executable that can be run by the operating system

Usually, we use external libraries like `stdio.h`, functions from which are attached to our executable as needed

An example (continued)

```
objdump -d -S example1
```

This newly compiled file is a binary file in a format called **ELF** (Executable Linkable Format)

An example (continued)

Now, we look at the same example compiled as an executable, with some initialization code added and the external libraries attached

```
path: ./example2.dump
lang: asm
```

What did the last couple of examples tell us?

- Compilation generates machine code for the machine we run the compiler on (x86_64 in this case)
- Init code is OS specific
- Linking external libraries is environment dependent

This means that compilation itself is very environment dependent

We can describe the environment using 3 things

- Machine (architecture)
- Vendor
- Operating System

This forms what we call a *target triplet*

```
machine-vendor-os
```

Each of these 3 things take up a field (OS may take 2), and sometimes the vendor field may be omitted

To find the target triplet for the current machine, we can run a simple command `gcc -dumpmachine`

And this is my laptop's target triplet

```
gcc -dumpmachine
```

Where does cross compilation come in?

We use cross compilation to generate machine code for machines different than ours

Terminology

Build Platform: This is the platform on which the compilation tools are executed

Host Platform: This is the platform on which the code will eventually run

Target Platform (for a compiler): This is the platform that the compiler will generate code for

What do we need for cross compiling code?

- Cross compiler
- Source code/headers for libraries on the target

Cross compilation example

We'll compile a program for the target `arm-none-eabi`

Breaking down the triplet tells us what the target is

`arm`: The target's architechure is *ARM*

`none`: The vendor is none, this is mostly irrelevant

`eabi`: The OS field is *EABI* (Embedded-application binary interface), basically means no OS

We use a cross compiler here, `arm-none-eabi-gcc`

```
command: sh -il
rows: 3
init_text: arm-none-eabi-gcc -g -c example1.c -o example3
init_wait: "1$"
```

Cross compilation example (continued)

Disassembling the file, we see the assembly for our target platform (ARM)

```
path: ./example3.dump
lang: asm
```

Cross compilation example (continued)

If we try to compile this into a complete executable...

```
command: sh -il
rows: 20
init_text: arm-none-eabi-gcc -g example1.c -o example3
init_wait: "1$"
```

We get this huge error!

Cross compilation example (continued)

This happens because the library we tried to include in our program isn't available on our build platform

Generally, when we compile for other targets, we also need to implement C's *standard library* for the target For some targets, implementing and compiling the entire thing (GNU, or even musl) might not be possible because of various reasons

It also gets tedious when you need to compile and implement it for multiple targets

To solve this issue, there exists a minimal implementation of the standard library called *newlib* This library can be configured according to our needs while linking our compiled program to make our executable, and is usually the most commonly used library for cross compilation newlib has been *implemented for a lot of targets*, ranging from Linux on x86, ARM, RISC-V, to bare metal on ARM, RISC-V, and much more We denote the implementations of newlib using target triplets too

For example, the implementation of newlib for `arm-none-eabi` is called `arm-none-eabi-newlib`

Cross compilation example (continued)

Searching the Arch repositories for implementations for newlib shows us just how much it has been ported

```
paru -Ssq newlib
```

```
sh -c "paru -Ssq newlib"
```

Conclusion

We need cross compilation when:

- Build and target *architectures* are different
- Build and target *operating systems* are different
- Certain build and target *environments* differ

We can specify these quantities using a *target triplet*

```
machine-vendor-os
```

For cross compilation, we need available on our build platform:

- A *cross compiler* for the given target
- The required *libraries* for the target

That's it!

Reading material

- Wikipedia - Compiler
- Wikipedia - Cross compiler
- OSDev Wiki - Target triplet
- OSDev Wiki - GCC Cross Compiler
- Zero to main() series