Toolchain

To build a working Linux system, you will require to put together (or use already built) a minimum of three components:

- **a bootloader** (minimally, to load the Linux kernel into RAM and jump to it's entry point)
- **a Linux kernel image** (the core OS/drivers)
- **a Linux root filesystem** (rootfs: the set of directories under root "/"; its a standard layout, refer to the FHS – the *Filesystem Hierarchy Standard*).

What is a Toolchain

To build these three components from source code, you will require a *toolchain*.

Toolchain : set of components, usually comprising systems software, to generate binary executables as well as object files, libraries, kernel image, kernel modules, drivers, etc on a given platform.

Best example of a component: the `gcc / clang` compilers;
The compiler is not a stand-alone tool; several tools "under the hood" are invoked.

The toolchain will include a compiler, a linker and runtime libraries.

The toolchain that is pre-installed on a regular Linux distro is called the *"native" toolchain*; it works for that hardware (usually x86_64) and software platform.

Cross Toolchain

What if we want to compile a program on our Intel x86_64 *host* system, but transfer onto and run it on an ARM-based Linux *target* device.

This is precisely why we require a *cross toolchain* – same as the toolchain, except that it runs on your host but generates code for a given target.
Eg. x86-to-ARM, x86-to-MIPS, x86-to-PPC, PPC-to-x86, etc.

Toolchain Naming Convention

Source: http://web.eecs.umich.edu/~prabal/teaching/eecs373-f12/notes/notes-toolchain.pdf

...
Unix cross compiler naming conventions can seem mystifying. If you search for

an ARM compiler, you might stumble across the following toolchains:

`arm-none-linux-gnueabi`, `arm-none-eabi`, `arm-eabi`, and `arm-softfloat-linux-gnu` , among others. This might leave you wondering about the method to the naming madness. Unix cross compilers are loosely named using a convention of the form `arch[-vendor][-os]-abi` .

The `arch` refers to the target architecture, which in our case is ARM. The `vendor` nominally refers to the toolchain supplier. The `os` refers to the target operating system, if any, and is used to decide which libraries (e.g. newlib, glibc, crt0, etc.) to link and which syscall conventions to employ. The `abi` species which application binary interface convention is being employed, which ensures that binaries generated by different tools can interoperate.

Some examples will illustrate the rough naming convention:

`arm-none-eabi`
is the toolchain we use in this class. This toolchain targets the ARM architecture, has no vendor, does not target an operating system (i.e. targets a “bare metal" system), and complies with the ARM EABI.

`i686-apple-darwin10-gcc-4.2.1`
is the version of GCC on my MacBook Pro. This toolchain targets the Intel i686 architecture, the vendor is Apple, and the target OS is Darwin version 10.

`arm-none-linux-gnueabi`
is the toolchain that can be installed in Debian-based systems using a package manager like `apt` (the package is called `gcc-arm-linux-gnueabi`). This toolchain targets the ARM architecture, has no vendor, creates binaries that run on the Linux operating system, and uses the GNU EABI. In other words, it is used to target ARM-based Linux systems.

`arm-eabi`
(Android ARM compiler).

...

Take a look at some practical (and recent, as of this writing), notes here on installing a toolchain for generic x86-to-ARM development:
https://github.com/kaiwan/seals/wiki/HOWTO-Install-required-packages-on-the-Host-for-SEALS

A good resource on Toolchains: https://elinux.org/Toolchains .

Example :: Installing a modern x86-to-ARM Toolchain

Source: SEALs (Simple Embedded ARM Linux System) Wiki page

NOTE- Quite often, folks install a toolchain using the distro's package management software (like apt, dnf or yum). Well, that does install a basic toolchain but I find that it's incomplete in an important aspect: the toolchain 'sysroot' libraries, sbin and usr components!

For this reason, I strongly suggest you don't take the shortcut and just install a toolchain via the package manager; rather, follow the steps below to properly install the **Linaro x86_64-to-ARM GNU EABI5 toolchain on Linux**:

The links provided below are representative; they're fine as of this writing (July 2018) but may change in future.

Base URL: https://releases.linaro.org/components/toolchain/binaries/latest/arm-linux-gnueabihf/

1. The toolchain xz file:
   URL: https://releases.linaro.org/components/toolchain/binaries/latest/arm-linux-gnueabihf/gcc-linaro-7.3.1-2018.05-x86_64_arm-linux-gnueabihf.tar.xz

   a) [Older]
   https://releases.linaro.org/components/toolchain/binaries/latest/arm-linux-gnueabihf/gcc-linaro-7.2.1-2017.11-x86_64_arm-linux-gnueabihf.tar.xz

2. The 'sysroot' xz (holds the runtime libraries, etc):
   URL: https://releases.linaro.org/components/toolchain/binaries/latest/arm-linux-gnueabihf/sysroot-glibc-linaro-2.25-2018.05-arm-linux-gnueabihf.tar.xz

   a) [Older]
   https://releases.linaro.org/components/toolchain/binaries/latest/arm-linux-gnueabihf/sysroot-glibc-linaro-2.25-2017.11-arm-linux-gnueabihf.tar.xz

3. Extract the above .tar.xz files (I extracted them into my ~/Downloads dir) (Its not mandatory to extract the *sysroot* just yet).

4. Setup your PATH environment variable to point to the toolchain binaries:

```
export PATH=${PATH}:/home/seawolf/Downloads/arm-linaro_2018.05/gcc-linaro-7.3.1-2018.05-x86_64_arm-linux-gnueabihf/bin/
```

5. Test the cross toolchain: ensure a ‘C’ program compiles and links correctly.
