Nano-Kernel : A Bare Metal OS

The first prerequisite is that you will need your own Linux environment. This guide assumes that you have VMWare Workstation (or Fusion) and a freshly installed Ubuntu 18.04. Normally, we’d want to use the 64-bit version of Ubuntu, but for this first step, it is probably easier to install the 32-bit version of Linux. See: <http://bfy.tw/GIou> for instructions on how to accomplish a generic install. The only changes I recommend are to modify the workstation configuration to 4GB of RAM (or as much as you can reasonably allocate) and 2 (or more as many as you can spare) CPU’s.

After the installation is finished, it is helpful to execute a “sudo reboot” command to reset the VMWare video emulator to resize and fit your window size.

After a default installation you will need to install the following packages:

ubuntu:~$ **sudo apt-get install xorriso qemu bless texinfo**

## Cross Compiler

From: <https://wiki.osdev.org/GCC_Cross-Compiler> and   
<http://www.ifp.illinois.edu/~nakazato/tips/xgcc.html>

One of the first steps is to build a cross-compiler to build the rest of the system. The GNU tools use a “target triplet” naming convention to determine the target of the cross-compiler:

machine-vendor-operating\_sys

For example, The GCC that ships with your newly installed Ubuntu system is:

x86\_64-unknown-linux-gnu

or simply

x86-linux-gnu

We are seeking to build a cross-compiler for an operating system that is yet to be written, and for the 32-bit intel processors. Our **target triplet** will be:

i686-none-elf

Or

i686-elf

The GNU build system uses this target triplet to direct how to build the cross compiler. We’ll set some environment variables to make this easier later on:  
$ **export TARGET=i686-elf**

$ **export PREFIX=/opt/cross**

$ **export PATH=$PATH:$PREFIX/bin**

**$ sudo mkdir $PREFIX**

**$ sudo chown opsys $PREFIX**

### GNU Binutils

These instructions use an “out-of-tree” build. This means that there is a separate build directory from the source code. This is a best-practice. It allows you to build multiple cross-compilers from the same source, or to remove a build directory and recreate it without risking removing parts of the downloaded package.

1. Download the source code for GNU “binutils” - a set of utilities that enable the creation of executable files, especially the assembler and linker: <http://www.gnu.org/software/binutils/>. The latest at the time of this writing is : binutils-2.30.tar.xz
2. Extract the downloaded file:  
   ubuntu:~$ **tar xJfv ~/Downloads/binutils-2.30.tar.xz**
3. Make a build directory:  
   ubuntu:~$ **mkdir binutils-build**

ubuntu:~$ **cd binutils-build**

1. Build binutils:

ubuntu:binutils-build$ **../binutils-2.30/configure \   
 --target=$TARGET --prefix=$PREFIX**

ubuntu:binutils-build$ **make -j*12* all**

ubuntu:binutils-build$ **make install**

### GNU GCC

The bootstrap build is used to create a stand-alone C compiler that can be used to compile code for libraries that are necessary for the full C compiler (and other languages).

1. Download the source code for GNU GCC - the C compiler: [http://www.gnu.org/softwaregcc/](http://www.gnu.org/software/binutils/). I chose the latest of the version 6 series of the C compiler, so at the time of this writing, I used “gcc-6.4.0.tar.xz”
2. Extract the downloaded file:  
   ubuntu:~$ **cd ~**  
   ubuntu:~$ **tar xJfv ~/Downloads/gcc-6.4.0.tar.xz**
3. Fix missing pre-requisites:  
   ubuntu:~$ **cd gcc-6.4.0**  
   ubuntu:gcc-6.4.0$ **contrib/download\_prerequisites**
4. Make a build directory:  
   ubuntu:~$ **cd ~**  
   ubuntu:~$ **mkdir gcc-build**

ubuntu:~$ **cd gcc-build**

1. Build the compiler

ubuntu:gcc-build$ **../gcc-6.4.0/configure \   
 --target=$TARGET --prefix=$PREFIX \  
 --without-headers --with-newlib \**

**--disable-shared --disable-nls \**

**--with-gnu-as --with-gnu-ld --enable-c99**

ubuntu:gcc-build$ **make -j12 all-gcc**

ubuntu:gcc-build$ **make install-gcc**

ubuntu:gcc-build$ **make -j12 all-target-libgcc**

ubuntu:gcc-build$ **make install-target-libgcc**

\*Note: I gave the machine 16GB of RAM and 12 CPU’s. “**make -j all-gcc**” uses all of them in this build, and I ran out of memory. I know that because I received an error that “g++: internal compiler error: Killed (program cc1plus)” I simply re-ran the same “**make -j12 all-gcc**” command again and it completed without error, but only using 12 CPUs - select the number appropriate to your virtual machine.

### GNU Newlib

Newlib is a set of system-independent C libraries useful for low-level / bare-metal programming.

1. Download source: <http://www.sourceware.org/newlib/>
2. Extract source:  
   ubuntu:~$ **cd ~**ubuntu:~$ **tar xvfz ~/Downloads/newlib-3.0.0.tar.gz**
3. Make a build directory:  
   ubuntu:~$ **cd ~**  
   ubuntu:~$ **mkdir newlib-build**

ubuntu:~$ **cd newlib-build**

1. Build the compiler

ubuntu:newlib-build$ **../newlib-3.0.0/configure \**

**--target=$TARGET --prefix=$PREFIX**

ubuntu:~$ **make -j12 all**

ubuntu:~$ **make install**

### GNU Debugger (GDB)

Newlib is a set of system-independent C libraries useful for low-level / bare-metal programming.

1. Download source: <http://ftp.gnu.org/gnu/gdb/>
2. Extract source:  
   ubuntu:~$ **cd ~**ubuntu:~$ **tar xvfJ ~/Downloads/gdb-8.1.tar.xz**
3. Make a build directory:  
   ubuntu:~$ **cd ~**  
   ubuntu:~$ **mkdir gdb-build**

ubuntu:~$ **cd gdb-build**

1. Build the compiler

ubuntu:newlib-build$ **../gdb-8.1/configure \**

**--target=$TARGET --prefix=$PREFIX**

ubuntu:~$ **make -j12 all**

ubuntu:~$ **make install**

## Cross-Compiler Test Program

Next, I built a small cross-compiler test program, first the assembler. Copy the following to a file called test.s:

Figure 1 - test.S, a simple assembly file.

.section .bss  
.align 16  
stack\_bottom:  
.skip 16384 # 16 KiB  
stack\_top:

.section .text

.global \_start

.type \_start, @function

\_start:

cli

1: hlt

jmp 1b

#.size \_start, . - \_start

Then, to test the assembler:

opsys@ubuntu:~/cross-test$ **i686-elf-as boot.s -o test.o**

opsys@ubuntu:~/cross-test$ **file test.o**

test.o: ELF 32-bit LSB relocatable, Intel 80386, version 1 (SYSV), not stripped

Then, to test the C compiler:

Figure 2 - test.c a Simple C file

int main()

{

}

opsys@ubuntu:~/cross-test$ **i686-elf-gcc -c test.c**

opsys@ubuntu:~/cross-test$ **file test.o**

test.o: ELF 32-bit LSB relocatable, Intel 80386, version 1 (SYSV), not stripped

# Deliverables and Demos

Arrange a time for us to meet, and show be prepared to show me the following:

1. Use your cross-assembler to compile the test.S of Figure 1
2. Use your cross-compiler to compile the test.c of Figure 2
3. When we meet, answer the following questions:
   1. What is the “hlt” instruction for?
   2. Can you run these executables under Linux? Why or Why not?

Points: \_\_\_\_\_\_\_\_\_ / 10