Building a Cross-Compiler

i686-linux-gnu

Michele Bucca

Contents

1 How to build a cross-compiler that targets i686-linux-gnu

This work is based on *musl-cross-make's litecross*[1], the guide *How to Build a GCC Cross-Compiler*[2] and *Linux From Scratch*[3]

1.1 Explaining the creation process

A cross-toolchain is composed of a lot of tools: an assembler (**GNU as**), a linker (**GNU ld**), a tool to make and extract symbols from static libraries (**GNU ar**), a compiler (`cc`, `c++`), etc. The commands `ar`, `as`, `ld` are prvided by **GNU Binutils** while the c and c++ compilers are provided by thr **GNU Compiler Collection** (GCC for short).

Binutils is fairly easy to build and does not require a lot of dependencies, but GCC is a beast of its kind and will bring you into something we call dependecy hell. Why? Because in order to build GCC, not only we need to build MPC, MPFR and MPC, but we also need to have the kernel headers and a working libc!

The linux headers can be installed easily, at the beginning of the build, just like binutils, but to have a working libc we need to have a working cross-compiler in order to build it. We have a chicken and egg problem! How do we solve this? Usually we build a *simpler version of GCC*[4] with

[1] https://github.com/richfelker/musl-cross-make/blob/master/litecross/Makefile

[2] https://preshing.com/20141119/how-to-build-a-gcc-cross-compiler/

[3] https://linuxfromscratch.org/lfs/view/stable/

[4] https://linuxfromscratch.org/lfs/view/stable/chapter05/gcc-pass1.html

less features: then we *build the C library*[5] using that cross-compiler in order to be able to build the final compiler.

This requires building GCC two or three times, and if you have a slow computer it can take *ages*. I like to use the approach of richfelker[6] and the other website[7] to build GCC only **once**, one piece at a time.

1.2 Preparing for the build

1.2.1 Setting environment variables

Set the following environment variables

```
set +h
KERNEL_ARCH="x86"
TARGET="i686-linux-gnu"
DOWNLOADS=${PWD}/downloads
SOURCES=${PWD}/sources
OUTPUT="${PWD}/output/${TARGET}"
HOST=$(echo ${MACHTYPE} | sed "s/-[^-]*/-cross/")
MAKEFLAGS="-j$(nproc)"
PATH="${OUTPUT}/bin:${PATH}"
GCC_CONFIG_FOR_TARGET=""
COMMON_CONFIG=""

export PATH OUTPUT KERNEL_ARCH TARGET HOST MAKEFLAGS COMMON_CONFIG
```

The meaning ot the environment variables:

If you want to build different cross-compilers you need to customise these:

`KERNEL_ARCH="x86"`

this variable will be used when we will extract te ternel headers for our intel processors

`TARGET="i686-linux-gnu"`

We are targeting a 32bit system, but you can change it to x86_64-linux-gnu if you want.

`HOST=$(echo ${MACHTYPE} | sed "s/-[^-]*/-cross/")`

If the host and the target are the same a native compiler will be built instead. This prevents it. It uses the ${MACHTYPE} variable embedded in `bash` and turns the host target from **x86_64-linux-gnu** to **x86_64-cross-linux-gnu**. See the section Shell Variables from the bash manual

`MAKEFLAGS="-j$(nproc)"`

Makes sure that we use all our cores when compiling, this saves precious time

`PATH="${OUTPUT}/bin:${PATH}"`

Our cross-compiler tools need to be in our path in order to build successfully.

[5] https://linuxfromscratch.org/lfs/view/stable/chapter05/glibc.html
[6] https://github.com/richfelker/musl-cross-make/blob/master/litecross/Makefile
[7] https://preshing.com/20141119/how-to-build-a-gcc-cross-compiler/

set +h

Disables the hashing functionality of the shell. Usually the shell builds an hashing table to save time looking for our programs. This behavior can be harmful when compiling and building a cross-compiler. We need to use our new tools as soon as they are installed.

GCC_CONFIG_FOR_TARGET=""

Extra configuration for GCC. We could use it to optimize the compiler for a specific architecture but here we live it blank.

COMMON_CONFIG=""

Extra configuration for the toolchain, we leave this blank for now and we will use it later.

if we have a musl cross-compiler available we can build our toolchain statically!

```
_cc="gcc"
_cxx="g++"
musl_cxx="$(which -- $(uname -m)-linux-musl-g++)"
musl_cc="$(which -- $(uname -m)-linux-musl-gcc)"


if [ -n "$musl_cxx" ]; then
   _cc="$musl_cc -static --static"
   _cxx="$musl_cxx -static --static"
fi
```

To compile **binutils** and **GCC** we will use the compilers provided by the variable `${_cc}` and `${_cxx}`.

We look for a musl cross-compiler in our path and, if present, binutils and GCC will be compiled staticaly.

1.2.2 Download and extract the sources

Set the version of the software that we want to download:

```
    binutils_ver=2.45
    gcc_ver=15.2.0
    glibc_ver=2.42
    gmp_ver=6.3.0
    mpc_ver=1.3.1
    mpfr_ver=4.2.2
    linux_ver=6.16.1
```

Download the source tarballs:

```
    mkdir -p ${SOURCES} ${DOWNLOADS}; cd ${DOWNLOADS}
    wget -c https://sourceware.org/pub/binutils/releases/binutils-$binutils_ver.
tar.xz
    wget -c https://ftp.gnu.org/gnu/gcc/gcc-$gcc_ver/gcc-$gcc_ver.tar.xz
    wget -c https://ftp.gnu.org/gnu/glibc/glibc-$glibc_ver.tar.gz
    wget -c https://ftp.gnu.org/gnu/gmp/gmp-$gmp_ver.tar.xz
    wget -c https://ftp.gnu.org/gnu/mpc/mpc-$mpc_ver.tar.gz
```

```
wget -c https://ftp.gnu.org/gnu/mpfr/mpfr-$mpfr_ver.tar.xz
wget -c https://www.kernel.org/pub/linux/kernel/v6.x/linux-$linux_ver.tar.xz
```

Extract every tar archive into the sources directory:

```
for f in *.tar*; do tar xf $f -C ${SOURCES} ; done
```

1.3 Create directory structure

```
mkdir -p ${OUTPUT}/${TARGET}
ln -s . ${OUTPUT}/${TARGET}/usr || true
```

`ln -s . ${OUTPUT}/${TARGET}/usr || true`

We use an "or pipe" with `true` when creating a symlink to avoid the script to abort if we set the shell to exit on errors.

1.4 Linux Headers

Enter in the linux kernel source tree and type:

```
cd ${SOURCES}/linux-$linux_ver
make mrproper
ARCH=${KERNEL_ARCH} make headers
find usr/include -type f ! -name '*.h' -delete
cp -rv usr/include ${OUTPUT}/${TARGET}
```

`make mrproper`

Cleans the kernel source tree

`ARCH=${KERNEL_ARCH} make headers`

extracts the kernel headers

`find usr/include -type f ! -name '*.h' -delete`

this command will find every file that is not a header and delete it. There are a lot of `.cmd` files that are useless for us

`cp -rv usr/include \${OUTPUT}/${TARGET}`

copies the headers to the toolchain sysroot

1.5 Binutils

Create a build directory for binutils

```
mkdir -p ${SOURCES}/build-binutils
cd ${SOURCES}/build-binutils
```

Configure Binutils:

```
../binutils-$binutils_ver/configure \
  --enable-gprofng=no \
  --disable-nls \
  --disable-separate-code \
  --disable-werror \
  --target=${TARGET} --prefix= \
  --disable-multilib \
```

```
--with-sysroot=/${TARGET} \
--enable-deterministic-archives \
--build=${HOST} --host=${HOST} \
${COMMON_CONFIG} CC="${_cc}" CXX="${_cxx}"
```

The meaning of the configure options:

`--enable-gprofng=no`
: disables gprofng, it may fail to build and it's not needed

`--disable-nls`
: disables internationalization, we are not interested building the translations of the program

`--disable-werror`
: This prevents the build from stopping in the event that there are warnings from the host's compiler

`--build=${HOST} --host=${HOST}`
: This cross compiler is built and needs to run on ${HOST}. We set the environment variable at the beginning

`--target=${TARGET}`
: This specifies the target of the cross compiler. We set the environment variable at the beginning: *i686-linux-gnu*

`--disable-multilib`
: We don't need multilib support, we keep it simple. This argument does nothing on cross-compilers that target 32bit systems

`--prefix= --with-sysroot=/${TARGET}`
: These options make the cross-compiler portable, you can move it around without breaking it

`--enable-deterministic-archives`
: GNU ar adds UIDs, GIDs file modes and timestamps to .a files. Because of this, two builds from the same sources yield different results if they include static archives. When the option above is used, if `ar` is used with identical options and identical input files, multiple runs will create identical output files regardless of the input files' owners, groups, file modes, or modification times.

`CC="${_cc}" CXX="${_cxx}"`
: `CC` and `CXX` environment variables will determine which compiler to use. the musl one, for static compiling or normal one installed in the system

Build and install the package:

```
make
make install-strip DESTDIR=${OUTPUT}
```

The `make install-strip` command removes debugging sybols from the executables in order to make them smaller.

⚠️ **Caution!**
We use the `DESTDIR` variable with an *absolute path* to tell `make install-strip` where to install the program. Without it, an umpredictable behavior might occur. If you don't specify the `prefix`, as we did here, build things as root, and forget to pass the `DESTDIR` variable. **It might break your system!**

1.6 GCC (Core)

in order to build GCC, we need GMP, MPFR and MPC.

Enter in the source directory of GCC and create a symbolic link to their sources:

```
cd ${SOURCES}/gcc-$gcc_ver
ln -sf ../gmp-$gmp_ver gmp
ln -sf ../mpfr-$mpfr_ver mpfr
ln -sf ../mpc-$mpc_ver mpc
```

Create a build directory for gcc:

```
mkdir -p ${SOURCES}/build-gcc
cd ${SOURCES}/build-gcc
```

Configure GCC:

```
../gcc-$gcc_ver/configure --target=${TARGET} \
    --host=${HOST} --build=${HOST} \
    --enable-languages=c,c++ \
    --target=${TARGET} --prefix= \
    --disable-multilib \
    --disable-libsanitizer \
    --with-sysroot=/${TARGET} \
    --with-build-sysroot=${OUTPUT}/${TARGET} \
    ${GCC_CONFIG_FOR_TARGET} \
    ${COMMON_CONFIG} CC="${_cc}" CXX="${_cxx}"
```

The meaning of the configure options:

`--enable-languages=c,c++`
: We only requiure the C and C++ compiler at the moment, so this will build only `gcc` and `g++`

`--disable-libsanitizer`
: We disable libsanitizer becasue it fails to build under certain architectures.

`--with-build-sysroot=${OUTPUT}/${TARGET}`
: the sysroot of our cross-compiler that we specified previously is relative to the location of the `gcc` and `g++` binary. This will cause the build to fail, because the compiling process calls these executables, while they are not yet installed in their proper location, to build some libraries

that will run on the target. Usually this is not a problem, because the compiling process looks into the `<prefix>/<sysroot>` directory, but in these case the prefix is empty and the compiler can't find anything becasue it looks in the wrong place. We explicitly tell the build system which directory to use as sysroot to avoid this problem.

Build only core of GCC

```
make all-gcc
make install-strip-gcc DESTDIR=${OUTPUT}
```

1.7 Glibc Headers

Create a build directory for glibc

```
mkdir -p ${SOURCES}/build-glibc
cd ${SOURCES}/build-glibc
```

Configure and install the libc headers, they are needed in order to build `libgcc` and the rest of the compiler:

```
../glibc-$glibc_ver/configure --prefix=/usr \
    --host=$TARGET --disable-nls --disable-werror --disable-nscd
make DESTDIR=${OUTPUT}/${TARGET} install-bootstrap-headers=yes install-
headers
```

The meaning of the configure options:

`--disable-nscd`

Do not build the name service cache daemon which is no longer used.

We need to install the crt* files in order to build other libraris such as `libgcc`:

```
make csu/subdir_lib
install csu/crt1.o csu/crti.o csu/crtn.o  ${OUTPUT}/${TARGET}/lib
```

We install a dummy, empty, `libc.so`. This is needed to build the libgcc shared library successfully:

```
${TARGET}-gcc -nostdlib -nostartfiles -shared -x c /dev/null -o ${OUTPUT}/
${TARGET}/lib/libc.so
```

We also need a `gnu/subs.h` file to be present inside the include directory of our sysroot, so we create it now:

```
touch ${OUTPUT}/${TARGET}/include/gnu/stubs.h
```

Keep the source and build directory intact

```
cd ${SOURCES}
```

1.8 GCC compiler support library (libgcc.a/libgcc.so)

Build and install `libgcc`:

```
cd ${SOURCES}/build-gcc
make all-target-libgcc
make install-strip-target-libgcc DESTDIR=${OUTPUT}
```

Leave the build directory intact. We will use it later

1.9 Glibc

Move to the glibc directory

```
cd ${SOURCES}/build-glibc
```

Finish building the rest of the C library and install it:

```
make
make install DESTDIR=${OUTPUT}/${TARGET}
```

1.10 GCC (Final)

Now we can finally finish building GCC.

Enter in the build directory of GCC:

```
cd ${SOURCES}/build-gcc
```

Run the following commands to finish the build of GCC and install it:

```
make
make install-strip DESTDIR=${OUTPUT}
```

We should now have a working cross-compiler!