Guía de desarrollo para otras arquitecturas

Mariano Street – diciembre de 2018 – Licenciatura en Ciencias de la Computación

Tabla de Contenido

Las arquitecturas de computadoras que existen son muchas. Cuando todo nuestro desarrollo se limita a una específica y las máquinas de desarrollo mismas tienen esta arquitectura, no hay mayor problema. El caso más común es x86: nuestra PC es x86, el software que hacemos es para x86, entonces todo funciona.

Pero ¿qué pasa cuando pasamos a desarrollar software para otras arquitecturas? Tomemos como ejemplo la arquitectura ARM. Queremos hacer un programa para ARM. Hay dos opciones: una es conseguirnos una máquina de desarrollo que sea también ARM, entonces usamos todo cómodamente como en el caso anterior. La solución funciona, pero tiene sus inconvenientes: requiere conseguir una máquina nueva y abandonar las posibles comodidades de la anterior.

Un enfoque alternativo es tener la máquina de desarrollo en una arquitectura y desarrollar para otra. Es decir, por ejemplo, mantener la PC con x86 para hacer programas para ARM. Para hacer esto, las herramientas que venimos usando (GCC, GDB y demás) nos siguen sirviendo, pero hay que ajustarlas un poco. Además, en caso de no tener tampoco hardware con la arquitectura de destino para ejecutar los programas, necesitamos encontrar alguna otra forma de probarlos.

Compilación

Compilador cruzado

Entra en juego lo que se denomina un **compilador cruzado**. Se trata de un compilador que está construido para una arquitectura dada pero que genera código para otra.

GCC es capaz de generar código para muchas arquitecturas distintas. Sin embargo, esto no es configurable pasándole opciones al ejecutarlo: una vez compilado GCC, este genera código para solo una arquitectura. La misma recibe el nombre de **arquitectura objetivo** (en inglés, *target architecture*). Se le indica como una opción a GCC al ser compilado.

Los paquetes de GCC que uno instala comúnmente en su sistema GNU/Linux están hechos para compilar para la arquitectura nativa (la de la máquina donde se compila). Es decir, no sirven para lo que buscamos.

En muchas distribuciones de GNU/Linux hay disponibles paquetes de GCC cruzado para cierta arquitectura. Por ejemplo, para ARMv7 (ARM de 32 bits) se tienen, entre otros, los siguientes paquetes:

Debian GNU/Linux, Ubuntu

`gcc-arm-linux-gnueabi`, `gcc-arm-linux-gnueabihf`, `gcc-arm-none-eabi`

Arch Linux

`arm-none-eabi-gcc`, `arm-linux-gnueabi-gcc` (AUR), `arm-linux-gnueabihf-gcc` (AUR), `gcc-arm-none-eabi-bin` (AUR)

OpenSUSE

`cross-arm-linux-gnueabi-gcc`, `cross-arm-none-eabi-gcc`

Una vez instalado el paquete, pasa a estar disponible un nuevo comando para GCC, por lo general con un prefijo que indica la plataforma objetivo. Por ejemplo `arm-linux-gnueabihf-gcc`. Se lo usa con exactamente los mismos argumentos que el GCC al que estamos acostumbrados. Por ejemplo:

```
$ arm-linux-gnueabihf-gcc -o hola hola.c
```

Podemos ejecutar `file` sobre el archivo ejecutable generado, y veremos que efectivamente es para la arquitectura objetivo.

```
$ file hola
hola: ELF 32-bit LSB executable, ARM, EABI5 version 1 (GNU/Linux), ...
```

Utilidades cruzadas

Además de GCC, hay muchas otras herramientas que intervienen en el proceso de desarrollo y dependen de la arquitectura. Dos de ellas son fundamentales: `as` (ensamblador) y `ld` (enlazador). Otras son complementarias y en muchos casos resultan útiles, por ejemplo `objdump`. Se trata de un conjunto de herramientas de GNU agrupado bajo el nombre **Binutils**.

La lista completa de comandos incluidos es: `ld`, `as`, `addr2line`, `ar`, `c++filt`, `dlltool`, `gold`, `gprof`, `nlmconv`, `nm`, `objcopy`, `objdump`, `ranlib`, `readelf`, `size`, `strings`, `strip`, `windmc`, `windres`.

De nuevo, hay paquetes cruzados y es necesario instalar uno de ellos. Los principales para varias distribuciones, de nuevo considerando ARMv7, son:

Debian GNU/Linux, Ubuntu

`binutils-arm-linux-gnueabi`, `binutils-arm-linux-gnueabihf`, `binutils-arm-none-eabi`

Arch Linux

`arm-none-eabi-binutils`, `arm-linux-gnueabi-binutils` (AUR), `arm-linux-gnueabihf-binutils` (AUR)

OpenSUSE

`cross-arm-binutils`, `cross-arm-linux-gnueabi-binutils`, `cross-arm-none-eabi-binutils`

Ejecución

Puede ser que queramos compilar de forma cruzada pero después ejecutemos de forma nativa. Una razón posible es que la máquina objetivo la tengamos disponible pero no sea idónea para el desarrollo. En tal caso, no hacemos nada nuevo: movemos el archivo generado a la máquina objetivo y ejecutamos ahí directamente.

Pero también puede ser que queramos ejecutar sin una máquina nativa. En este caso, hay que usar un **emulador**. Es decir, un programa que recree, por software, una arquitectura dada; de esta forma se tiene una máquina virtual para la arquitectura destino.

El emulador recomendado es **QEMU**. Este está disponible en la mayoría de distribuciones de GNU/Linux. Tiene dos modos de emulación:

1. Sistema completo.
2. Espacio de usuario.

Vamos a ver el segundo. La sintaxis básica del comando es así:

```
$ qemu-<arq> <ejecutable>
```

Por ejemplo, si en `hola.c` tenemos un programa *hola, mundo*:

```
$ arm-linux-gnueabihf-gcc -static -o hola hola.c
$ qemu-arm hola
¡Hola, mundo!
```

PRECAUCIÓN

Hay que compilar de forma estática, por eso la opción `-static`. De lo contrario, QEMU busca bibliotecas que no están instaladas para la arquitectura en cuestión. Alternativamente, se las podría instalar y así no haría falta esto.

Depuración

Sabemos que con GDB se puede depurar un programa que se ejecuta nativamente. ¿Cómo hacer cuando no es nativo? Sería bueno contar con una forma también.

Hay formas de integrar GDB con QEMU. Veremos una de ellas. Antes que nada, hay que tener instalado un GDB multiarquitectura. En algunas distribuciones, esto viene en un paquete dedicado:

Debian GNU/Linux, Ubuntu

`gdb-multiarch`

Arch Linux

`gdb-multiarch` (AUR)

Hay algunos paquetes que vienen para arquitecturas específicas; en el caso de ARMv7:

Debian GNU/Linux, Ubuntu

`gdb-arm-none-eabi`

Arch Linux

`arm-none-eabi-gdb`

Luego, tenemos que compilar el programa con la opción `-g` de GCC, como haríamos nativamente. Usando el compilador cruzado, eso sí. Asumamos que el ejecutable generado se llama `hola`.

Hecho esto, abrimos dos terminales. En una ejecutamos QEMU con la opción `-g`. Por ejemplo:

```
qemu-arm -g hola
```

La opción `-g` hace que el emulador se quede esperando una conexión de GDB por un puerto del sistema. Por defecto es el 1234. Esto se puede cambiar, si se desea, indicándolo tras la opción:

```
qemu-arm -g [<puerto>] <ejecutable>
```

Por ejemplo:

```
qemu-arm -g 1337 hola
```

> **PRECAUCIÓN** En algunas versiones de QEMU, es obligatorio especificar el puerto.

En la otra terminal ejecutamos GDB.

```
$ gdb-multiarch <archivo>
> target remote <dirección>:<puerto>
```

Por ejemplo:

```
$ gdb-multiarch hola
> target remote localhost:1234
```

Y nos queda todo listo para una sesión de depuración.

Ultima actualización 2018-12-02 00:46:54 -0300