Herramienta de compilación GNU Make

Programación para Ingeniería Eléctrica

April 25, 2014

El propósito del comando **make** es el de automatizar la compilación de proyectos medianos a grandes de desarrollo. En este tutorial veremos un uso muy básico de esta herramienta de modo de cubrir las necesidades mínimas del curso. Dicho esto, con lo que se verá aquí alcanza para hacer proyectos razonablemente grandes si se trabaja de manera prolija.

1 Introducción

Supongamos, a lo largo de este tutorial, que tenemos implementada un aplicación `simulador` que consta de los siguientes módulos: `calculo`, `bloques`, `interfaz`, y `utilidades`. Cada uno de estos módulos consta de un archivo fuente .c y un encabezado **.h**. El `main` se encuentra en `interfaz.c`. Finalmente, la aplicación utiliza funciones de la biblioteca de matemática del sistema, `libm`, y de la biblioteca *GNU Scientific Library* (GSL), que se compone de varias bibliotecas. En este caso usaremos `libgsl`, que a su vez depende de `libgslcblas` (CBLAS es C Basic Linear Algebra System, funciones de álgebra lineal básicas como producto interno, matriz por vector, etc.).

La herramienta `make` utiliza como entrada un archivo de descripción del proyecto cuyo nombre por defecto es **Makefile** o **makefile**. Si se utiliza otro nombre, debe especificarse mediante una opción en su línea de comandos. El makefile más sencillo para compilar este proyecto sería algo así:

```
simulador: interfaz.o calculo.o bloques.o utilidades.o
	cc -o simulador interfaz.o calculo.o bloques.o utilidades.o -lgsl -lgslcblas -lm

interfaz.o: interfaz.c
	cc -c -Wall -O2 interfaz.c

calculo.o: calculo.c
	cc -c -Wall -O2 calculo.c

utilidades.o: utilidades.c
	cc -c -Wall -O2 utilidades.c

bloques.o: bloques.c
	cc -c -Wall -O2 bloques.c

clean:
	-rm -f *.o
	-rm -f simulador
```

Luego de ejecutado el comando (simplemente escribiendo `make` y luego enter en la linea de comandos), tendremos, si no hay problemas de compilación, el ejecutable **simulador** compilado. Si volvemos a correr `make` sin cambiar ningún archivo, se nos indicará que no hay nada que compilar, ya que no han habido cambios en los archivos fuente. Si en cambio, tocamos un archivo cualquiera. Por ejemplo, le actualizamos artificialmente la última fecha de modificación mediante

```
$ touch interfaz.c
```

y luego corremos `make`, veremos que sólo se recompila **interfaz.c** y se linkea **simulador**, pero el resto no es tocado. En proyectos grandes, este ahorro hace una diferencia enorme en tiempo de desarrollo.

2 Sintaxis del Makefile

Ahora veamos, cómo es que se define en el Makefile qué y cómo debe compilares. Observando el listado anterior, vemos que el archivo se divide en 6 bloques de estructura similar, separados por una o más lineas en blanco. Cada bloque de estos se llama *regla* (rule) de compilación.

```
objetivo: dependencia1 dependencia2 dependencia3 ...
        comando1
        comando2
        ...
        comandon
```

El primer bloque del Makefile de ejemplo tiene como *objetivo* (target) al ejecutable `simulador`, como *dependencias* a todos los archivos objeto que conforman al ejecutable `simulador`, y como único comando a la linea que, a partir de las dependencias, genera el objetivo. La semántica de la regla es pues la siguiente: define que para generar el *objetivo*, se necesita previamente de una serie de *dependencias*; teniendo éstas, se genera el objetivo en base a las dependencias mediante los *comandos* listados inmediatamente abajo.

Es *fundamental* notar que los objetivos empiezan en la **primera** columna del archivo de texto, mientras que los comandos empiezan luego de **1 tab** o bien **8 espacios**. Otra cosa importante es que **no deben haber espacios** entre el nobmre del objetivo y ":". El formato de un Makefile es muy rígido; si no se respeta, el comando `make` lo rechaza como inválido.

Cada dependencia puede ser un archivo u otro objetivo en el Makefile. En el primer caso, es necesario que dicho archivo exista en el sistema. En el segundo caso, `make` ejecutará los comandos de la regla correspondiente a la dependencia *previamente* a ejecutar los del objetivo que depende de ella. De este modo, al ejecutar la regla de `simulador`, primero se ejecutarán las reglas de los archivos objeto `interfaz.o`, etc., para tenerlos todos disponibles.

La última regla tiene como objetivo algo que no es un archivo sino simplemente un nombre. A esto se le llaman `objetivos engañosos`, ya que nunca pueden ser compilados, y su propósito es automatizar ciertas tareas típicas de un proyecto como, por ejemplo, limpiar archivos intermedios generados en la compilación. Éste es el caso del objetivo `clean`. El nombre no tiene nada de especial. Podría haberse llamado `apocalipsis` o `miabuelaentanga` que da lo mismo.

3 Variables, reglas automáticas

Incluso para el proyecto anterior, si uno desea modificar por ejemplo las opciones del compilador (hasta ahora `-Wall -O2`), es muy engorroso trabajar con el Makefile tal como se presentó; ni que hablar para proyectos más grandes. La sintaxis de Makefile por suerte es mucho más rica y permite definir variables y todo tipo de automatizaciones para simplificarnos la vida. Veamos algunas de las más sencillas y útiles. Veamos un ejemplo más elaborado:

```
OBJ=interfaz.o calculo.o bloques.o utilidades.o
CFLAGS=-Wall -O2
LIBS=-lgsl -lgslcblas -lm
CC=gcc

simulador: $(OBJ)
	$(CC) -o simulador $(OBJ) $(LIBS)

interfaz.o: interfaz.c
	$(CC) -c $(CFLAGS) interfaz.c

calculo.o: calculo.c
	$(CC) -c $(CFLAGS) calculo.c

utilidades.o: utilidades.c
	$(CC) -c $(CFLAGS) utilidades.c

bloques.o: bloques.c
	$(CC) -c $(CFLAGS) bloques.c
```

```
clean:
	-rm -f $(OBJ)
	-rm -f simulador
```

En el ejemplo anterior se definieron 4 variables: `OBJ`, `CFLAGS`, `CC` y `LIBS`. Una vez definidas (debe ser en la primera linea, tampoco pueden haber espacios entre el nombre de la variable y el "="), se pueden utilizar en otros lugares posteriores a su definición en el archivo mediante la sintaxis `$(nombre de variable)`. Aquí usamos `CC` para definir en un sólo lugar qué compilador vamos a usar (`gcc`). `OBJ` suele usarse para almacenar la lista de archivos objeto del programa. `CFLAGS` contiene banderas y opciones del compilador. Finalmente `LIBS` se utilizó para listar las bibliotecas externas de que depende el programa. Notar nuevamente que essos nombres son convenciones, arbitrarios, podrían llamarse cualquier otra cosa.

Make también define un número de *variables predefinidas* para hacer cómodas ciertas tareas. Por ejemplo, `$@` refiere siempre al nombre del objetivo siendo generado, mientras que `$<` contiene la primera de las dependencias. Las primera y segunda reglas quedaría por ejemplo así:

```
...
simulador: $(OBJ)
	$(CC) -o $@ $(OBJ) $(LIBS)

interfaz.o: interfaz.c
	$(CC) -c $(CFLAGS) $<
...
```

4 Reglas generales

Otra cosa fastidiosamente repetitiva en el Makefile que tenemos ahora es que las reglas para todos los archivos objeto son idénticas salvo el nombre del archivo. Esto es casi siempre así para los archivos objeto. Por lo tanto, en lugar de definir una regla por cada archivo `.o`, podemos definir una regla general para todos los objetivos que cumplan un cierto patrón en su nombre; en particular, la última versión del Makefile que mostramos debajo lo hace para para todos los que tengan como extensión `.o`:

```
OBJ=interfaz.o calculo.o bloques.o utilidades.o
CFLAGS=-Wall -O2
LIBS=-lgsl -lgslcblas -lm
CC=gcc
HDR=interfaz.h calculo.h bloques.h utilidades.h

simulador: $(OBJ)
	$(CC) -o $@ $(OBJ) $(LIBS)

.c.o: $(HDR)
	$(CC) -c $(CFLAGS) $<

clean:
	-rm -f $(OBJ)
	-rm -f simulador
```

La sintaxis `.c.o` indica que, cuando se quiera compilar `interfaz.o`, se buscará una dependencia de nombre `interfaz.c` y dependencias secundarias luego del ":" (en este caso en la variable `HDR`, que contiene todos los encabezados – siempre es buena idea incluir los encabezados como dependencias!), y se ejecutará el comando correspondiente a sustituir todas las variables `CC`, `CFLAGS` y la variable especial `$<`.