Guía básica de la herramienta `make`

Grado en Ingeniería Informática

Escuela Politécnica Superior
Universidad de Alcalá

Departamento de Automática

Índice

1. Introducción

El desarrollo de aplicaciones, que están formadas por muchos archivos diferentes, puede implicar el uso de opciones de compilación muy diversas y complejas, de tal modo que puede convertirse en una tarea muy laboriosa, sensible a errores e incluso provocar que la reconstrucción de nuevas versiones de la aplicación sea un proceso lento.

`make` es una herramienta que permite automatizar la gestión de las órdenes de compilación y facilita la tarea de creación de ejecutables. Su funcionamiento, basado en reglas de compilación, permite definir una sola vez las opciones de compilación de los módulos que forman parte del programa; asimismo, es capaz de llevar un control de los cambios que se han realizado en los archivos fuente y ejecutables. Estas características evitan tener que recompilar los módulos del programa que no hayan sido modificados y, por lo tanto, optimizar el proceso de compilación.

2. Archivo *Makefile*

Un archivo *Makefile* es un archivo de texto que utiliza `make` para llevar a cabo la gestión de la compilación de programas. Este archivo contiene, en forma de reglas, las dependencias entre los diferentes archivos de un proyecto.

2.1. Cómo invocar a `make`

Al ejecutar la orden `make` desde la línea de órdenes, se busca por defecto el archivo *makefile* o *Makefile*. Si no se encuentra ninguno de estos archivos en el directorio actual de trabajo se producirá un error y `make` se detendrá.

Se puede modificar el nombre del archivo que `make` va a buscar invocando a make de la siguiente forma:

```
make -f <nombre_archivo>
```

2.2. Las reglas del `Makefile`

En general, las reglas de un *Makefile* especifican qué es lo que hay que hacer para obtener un archivo concreto de la aplicación. La sintaxis y semántica de las reglas es la siguiente:

```
objetivo: dependencias
< TAB > orden_1
< TAB > ...
< TAB > orden_n
```

objetivo Normalmente, es un archivo (o varios) que se quiere(n) crear o actualizar.

dependencias Son nombres de archivos u otros objetivos necesarios para actualizar el objetivo de la regla.

orden$_i$ Orden necesaria para reconstruir el objetivo de la regla. Puede ser cualquier orden del *shell* o intérprete de órdenes. Debe comenzar necesariamente por el carácter `<TAB>` (tabulador).

Un objetivo necesita actualización si y sólo si cualquiera de sus dependencias cambia.

Ejemplo:

```
archej: archl.o arch2.o
<TAB> gcc principal.o practica1.o entrada.o -o practica1
```

En el ejemplo anterior, para obtener `archej` es necesario que se hayan creado `arch1.o` y `arch2.o`. A su vez, para cada uno de estos archivos habrá una regla en el mismo archivo *Makefile*.

Para facilitar la comprensión de los archivos *Makefile* se pueden usar comentarios que empiezan por el carácter `'#'`. Las líneas de comentarios son ignoradas cuando se ejecute `make`.

2.3. Reglas implícitas

En algunos casos, los objetivos de un *Makefile* pueden no tener una lista de órdenes asociadas para poder ser generados, es decir, hay reglas que sólo indican las dependencias necesarias y es el propio `make` el que "sabe" cómo se logran los objetivos correspondientes porque posee para ellos reglas implícitas.

Ejemplo:

```
arch1.o: arch1.h
```

Si la regla anterior se ha definido en un *Makefile* y se ejecuta `make`, esta herramienta recorre las reglas y, en concreto, para generar el archivo objeto `arch1.o` no existen órdenes. En este caso, `make` presupone que para obtener ese archivo, como cualquier otro archivo objeto, es necesario compilar su archivo fuente asociado con la opción adecuada. Es decir, implícitamente ejecuta la regla siguiente:

```
arch1.o: arch1.c
<TAB> gcc -c arch1.c
```

2.4. Reglas ficticias

Además de las reglas normales, es habitual en los archivos *Makefile* el uso de reglas ficticias (o virtuales). Estas reglas no generan un archivo específico sino que se utilizan para realizar una determinada tarea dentro del desarrollo de la aplicación. Generalmente, estas reglas suelen tener un objetivo (denominado *objetivo ficticio*) pero ninguna dependencia.

Una regla de este tipo y muy utilizada en los archivos *Makefile* es la siguiente:

```
clean:
<TAB>rm -f archej *.o
```

Cuando se ejecute `make clean` en el directorio donde se encuentra el *Makefile*, la orden `rm` de la regla anterior se ejecuta y por lo tanto, se borran el archivo ejecutable `archej` y los archivos objeto del directorio de trabajo actual. El propósito de la regla: rehacer todo la próxima vez que se invoque a `make`.

Observe que, al no presentar esta regla ficticia dependencias, si existiera ya un archivo con el nombre `clean` en el directorio donde se encuentra el *Makefile*, al ejecutar `make` consideraría que el objetivo ya está realizado, no ejecutaría la orden `rm` asociada y mostraría un mensaje similar a: `make: 'clean' ya está actualizado`. Para evitar este efecto, se puede usar un objetivo especial, `.PHONY`. Las dependencias asociadas a este objetivo ignorarán la existencia de un archivo que coincida con el nombre de este objetivo y, por lo tanto, se ejecutará la orden (u órdenes) correspondientes. Así pues, para evitar este problema, en el ejemplo anterior se añadiría previamente la siguiente línea:

```
.PHONY: clean
```

2.5. Uso de variables en el *Makefile*

En un archivo *Makefile* se puede usar variables que facilitan su portabilidad a diferentes plataformas y entornos así como su modificación.

Las variables se definen de la siguiente forma:

```
nombre=valor
```

Generalmente, el nombre de la variable está escrito en mayúsculas.

Ejemplo:

```
CC = gcc
CFLAGS = -c -Wall
```

En el ejemplo anterior se observa el uso de las variables `CC` y `CFLAGS`. Son, entre otras, variables predefinidas de `make`. Nosotros simplemente las hemos

redefinido para representar el compilador de C usado y las opciones del compilador, respectivamente.

También existen variables de `make` no configurables. Son variables que no deberían ser modificadas. Por ejemplo:

SRCS Archivos fuente de la aplicación.

OBJS Archivos objeto de la aplicación.

HDRS Archivos cabecera de la aplicación.

...

Para obtener el valor de una variable, se pone el signo '$' y el nombre de la variable entre paréntesis o llaves.

Adicionalmente, también se pueden usar las variables de entorno accesibles desde el intérprete de órdenes (por ejemplo, la variable HOME). Ejemplo:

```
SRC = $(HOME)/src
INCLUDE = $(HOME)/include


arch1.o: $(SRC)/arch1.c $(INCLUDE)/arch1.h
<TAB>  $(CC) $(CFLAGS) $(SRC)/arch1.c
```

Además, en un *Makefile* se puede usar las denominadas variables automáticas, que se evalúan en cada regla. A continuación, se muestran algunos ejemplos de este tipo de variables[1]:

$@ Se sustituye por el nombre del objetivo de la regla actual.

$* Se sustituye por la raíz de un nombre de archivo.

$< Se sustituye por la primera dependencia de la regla actual.

$^ Se sustituye por una lista separada por espacios de cada una de las dependencias de la regla actual.

...

Ejemplo:

```
arch1.o: arch1.c arch1.h
<TAB>  gcc -c -Wall $<
```

En la orden de la regla anterior se sustituye `$<` por la primera dependencia, `arch1.c`.

[1]El uso de estas variables simplifican el *makefile* pero, a la vez, pueden hacerlo menos legible. No es obligatorio su uso en el desarrollo de las prácticas.

3. Un ejemplo de cómo funciona `make` con un *Makefile* sencillo

A continuación, se muestra un ejemplo sencillo de *Makefile* y a partir de él, cómo funciona el `make` al ser invocado.

```
#Makefile de ejemplo
CC=gcc
CFLAGS=-Wall -g

archej: arch1.o arch2.o libej.a
  $(CC) $(CFLAGS) arch1.o arch2.o -L./ -lej -o archej

arch1.o: arch1.c arch1.h
  $(CC) $(CFLAGS) -c arch1.c

arch2.o: arch2.c arch1.h
  $(CC) $(CFLAGS) -c arch2.c

.PHONY: clean
clean:
  rm -f archej *.o
```

Cuando se invoca a `make`, esta herramienta decide cuáles son los archivos que deben ser actualizados. Para ello, se basa en las dependencias existentes en las reglas del archivo *Makefile* así como en las fechas de la última modificación que el sistema operativo guarda para cada archivo. De este modo, para cada uno de los objetivos existentes, ejecuta las órdenes apropiadas especificadas en la regla correspondiente.

Según el funcionamiento expuesto, cada vez que se modifique algún archivo fuente y se invoque a `make`, se recompilará el programa sin necesidad de compilar todo de nuevo. Por ejemplo, si un archivo de cabecera (.h) se modifica, sólo se recompilarán los archivos fuente que incluyan esa cabecera, en función de las dependencias especificadas en la regla correspondiente del *Makefile*, y se actualizarán los archivos objeto. Una vez actualizados estos archivos, deben enlazarse todos los módulos objeto, ya recién actualizados o procedentes de anteriores compilaciones, para generar la nueva versión del ejecutable.

Cuando se invoca a `make`, por defecto procesa siempre la primera regla que encuentra en el *Makefile*. Por lo tanto, se pone en primer lugar la regla para el ejecutable (en el ejemplo, `archej`). Antes de actualizar este ejecutable, `make` debe procesar las reglas para sus dependencias, es decir, las reglas para todos sus módulos objeto. Cada uno de ellos tienen su propia regla en el *Makefile* del ejemplo, que especifican cómo se deben obtener los correspondientes archivos objeto (`.o`) mediante la compilación de sus archivos fuente correspondiente. La recompilación deberá efectuarse si el archivo fuente o cualquiera de los archivos de cabecera especificados como dependencias son más recientes que el archivo objeto a actualizar, o bien si éste no existe. Igualmente, antes de actualizar los archivos objeto que lo necesiten, `make` considera cómo actualizar sus dependencias, es decir, el archivo fuente y los archivos de cabecera. El archivo *Makefile* ejemplo no especifica cómo hacerlo porque no son archivos *objetivo* de ninguna regla, así que `make` no hace nada al respecto (puesto que los archivos fuente son modificados directamente por el programador mediante un editor).

Tras todas las compilaciones precisas, `make` debe decidir si actualizar el ejecutable `archej` o no. Deberá realizarse esta tarea bien si el archivo no existe (es la primera vez que se genera), o bien si alguno de sus módulos objeto es más reciente (si un módulo objeto acaba de ser generado, es más reciente que el ejecutable). Por ejemplo, si modificamos el archivo fuente `arch1.c` y a continuación ejecutamos `make`, se compilará este archivo fuente para obtener `arch1.o` (regla segunda), y luego se enlazan todos los módulos `.o` y la biblioteca `libej.a` para generar el nuevo ejecutable `archej` (regla primera). Si, por el contrario, se modifica el archivo `arch1.h`, se compilarán los dos archivos fuente, `arch1.c` y `arch2.c`, para obtener los archivos `arch1.o` y `arch2.o` respectivamente (reglas segunda y tercera), y luego serían enlazados junto con la biblioteca `libej.a` para generar el ejecutable `archej` (regla primera). Todo ello, de acuerdo a la ejecución de las reglas asociadas en el *makefile*.

Debido a este funcionamiento, se dice que `make` es un *shell script* inteligente; sólo ejecuta las acciones necesarias en la generación de archivos ejecutables.

Mediante el uso de parámetros y opciones pasados a `make`, se puede controlar qué archivos recompilar o cómo. Por ejemplo:

```
make arch1.o
```

En el caso anterior, sólo se actualizaría `arch1.o` (si se ha modificado alguna de sus dependencias, `arch1.h` o `arch1.c`), sin procesar el resto de reglas.

Igualmente sucedería al invocar `make` con *objetivos ficticios*. Por ejemplo:

```
make clean
```

Mediante el uso de los diversos elementos de la sintaxis de un *Makefile*, la forma de especificar las reglas puede ser variada. Por ejemplo, también se podrían haber sustituido las reglas para los archivos objeto `arch1.o` y `arch2.o` por la siguiente regla con múltiples objetivos.

```
arch1.o arh2.o: arch1.h
```

En este caso, como ya hemos visto anteriormente (sección 2.3), `make` tiene incorporada una regla implícita que le dice cómo construir un archivo objeto a partir del archivo fuente asociado, de tal forma que se puede omitir en la regla correspondiente la orden específica para generar los archivos objeto; sólo es necesario especificar las dependencias.