Sistemas Operativos

1. El programa Make

El programa Make[1] es una utilidad que permite automáticamente construir programas ejecutables y librerías a partir de uno o más códigos fuente. La construcción se realiza a partir de los archivos fuentes y mediante la lectura de un programa llamado $Makefile$ que especifica la forma en que se debe realizar la construcción de los programas o librerías. Make que permite definir reglas de dependencia entre ficheros. Aunque puede utilizarse para diferentes fines, está especialmente orientado a la compilación de código.

Para el uso de make, se debe escribir el archivo $Makefile$ en texto plano y éste debe describir la relación que existe entre el programa a generar y los archivos fuente que se tienen. En un programa, usualmente el archivo ejecutable es actualizado a partir de los archivos objeto y éstos a su vez son obtenidos a partir de los archivos con los códigos fuente.

El propósito de make es determinar automáticamente qué piezas de un programa necesitan ser recompiladas y lanzar las órdenes necesarias para lograrlo.

2. Makefiles

Para utilizar make debe escribirse normalmente un fichero llamado GNUmakefile, makefile o Makefile (se aconseja esta última forma) que describe las relaciones de dependencia de los ficheros que forman un programa. En un programa normalmente el fichero ejecutable se genera a partir de ficheros objeto, los cuales a su vez se construyen mediante la compilación de los ficheros fuente. El fichero makefile contiene esencialmente variables y reglas.

2.1. Comentarios

Para escribir comentarios debe insertarse el carácter # al comienzo del texto en cuestión. Se pueden añadir al final de una lista de dependencias o de una orden. Si aparecen solos en una línea, deben comenzar en la primera columna.

2.2. Información al usuario

Es posible imprimir texto en la consola como complemento de información para el usuario. Para ello se utiliza la orden echo. Esta orden debe utilizarse en una regla como cualquier otra orden

[1]Parte de este tutorial tomado de http://arco.esi.uclm.es/ david.villa/doc/repo/make/make.html

2.3. Variables

Una variable es un nombre simbólico que será substituido por su valor en el momento de la aplicación de las reglas posteriores. Para definir una variable se utiliza la siguiente sintaxis:

```
<IDENTIFICADOR>=valor
```

```
Un ejemplo:
```

```
NOMBRE_PROGRAMA = uniq
```

Por convenio las variables tienen identificadores con todas las letras en mayúscula. Para .[ex]pandirüna variable, es decir, para obtener su valor se utiliza la siguiente sintaxis:

$(IDENTIFICADOR)

Make define algunas variables que es necesario conocer y que ya contienen valores por defecto. Algunas de ellas son:

```
AR: Programa de empaquetado. Por defecto es ar.
CC: Compilador de C. Por defecto es cc.
CXX: Compilador de C++. Por defecto es g++.
CPP: Preprocesador de C.
CFLAGS: Opciones de compilación
CCXXFLAGS: Opciones de compilación C++.
CPPFLAGS: Opciones para el preprocesador.
LDFLAGS: Opciones para el montador (enlazador).
LDLIBS: Librerías a utilizar en el montaje.
```

El valor de estas variables puede modificarse en el propio archivo makefile como se verá en los ejemplos, pero también pueden darse valores por línea de comandos:

$ make CFLAGS=-g

2.4. Reglas

La sintaxis de una regla es:

```
<objetivo>: <requisitos>
            <orden para generar objetivo a partir de requisitos>
```

Tanto $< objetivo >$ como $< requisitos >$ suelen ser nombres de ficheros. Cuando make evalúa una regla debe poder ejecutar la orden asociada para generar el objetivo a partir de los requisitos. Si falta alguno de los archivos requisito, make buscará una regla que le permita saber cómo generarlo; si no existe tal regla terminará informando del error.

El nombre del objetivo debe comenzar en la primera columna y debe estar seguido del carácter dos puntos (:) y al menos un espacio. A continuación se enumeran los ficheros requeridos (llamados dependencias).

En la siguiente línea se describe la orden que ha de generar el objetivo. Dicha línea debe comenzar obligatoriamente con un carácter TABULACIÓN, en caso contrario make informará de un error de sintáxis. Cada regla puede contener varias líneas de órdenes que seguirán el mismo formato. Cada regla debe ir separada de la anterior por medio de una línea en blanco y al final del archivo también debe aparecer una línea vacía.

Veamos un ejemplo de makefile:

Listing 1: Ejemplo de Makefile

```
1
2  CC=gcc
3  CFLAGS=-Wall -g
4
5  hola: hola.o auxhola.o
6          $(CC) -o hola hola.o auxhola.o
7
8  hola.o: hola.c
9          $(CC) $(CFLAGS) -c -o hola.o hola.c
10
11 auxhola.o: auxhola.c
12         $(CC) $(CFLAGS) -c -o auxhola.o auxhola.c
13
14 clean:
15         $(RM) *.o hola core
```

En el ejemplo se pretende construir la aplicación hola que consta de los módulos hola.o y auxhola.o. Cuando se ejecuta sin parámetros, make siempre empieza evaluando la primera regla que aparece en el fichero, en este caso la regla hola.

Si alguno de los dos fichero objeto no está en el directorio, make ejecutará la regla correspondiente que permite generarlo.

Una característica muy importante de make es que antes de ejecutar una regla comprueba que realmente sea necesario. Por medio de las marcas de tiempo que el sistema operativo almacena en los descriptores de fichero, make averigua que órdenes necesita ejecutar. Siempre que el fichero objetivo tenga una fecha más reciente que cualquiera de los ficheros requisito, la regla se evaluará pero no se ejecuta la orden. Esto permite que cuando trabajamos con grandes proyectos formados por muchos módulos ahorremos mucho tiempo ya que make sólo recompila los ficheros necesarios para obtener la versión actualizada del ejecutable.

En el ejemplo aparece una regla adicional llamada clean; esta regla elimina todos los ficheros objeto y binarios dejando sólo los fuentes. Por convenio se la suele denominar clean aunque no es obligatorio. También suele definirse una regla all que provoca que se evaluen todas las demás reglas de construcción. Para ejecutar una regla en concreto debe indicarsele a make del siguiente modo.

$ make clean

3. Variables automáticas

Se pueden utilizar algunos indicadores especiales que permiten hacer ficheros *makefile* más simples:

```
$@ Es el objetivo de la regla en la que aparece.
$^ En la lista de requisitos de la regla en que aparece.
$?  Es la lista de requisitos más recientes que el objetivo.
$< Es el primer requisito de la regla.
```

Veamos como aplicarlos al ejemplo anterior:

Listing 2: Ejemplo de Makefile

```
1  CFLAGS=-Wall -g
2
3  hola: hola.o auxhola.o
4          $(CC) -o $@ $^
5
6  hola.o: hola.c
7          $(CC) $(CFLAGS) -c -o $@ $^
8
9  auxhola.o: auxhola.c
10         $(CC) $(CFLAGS) -c -o $@ $^
11
12 clean:
13         $(RM) *.o hola core
```

4. Reglas implícitas

Make es capaz de averiaguar por si mismo reglas sencillas. Por ejemplo, la regla del tipo:

hola.o: hola.c

puede darse por implícita y make entenderá que si necesita un fichero hola.o y tiene un fichero hola.c debe compilarlo. De esta forma podemos simplificar el fichero makefile anterior, quedando algo como:

```
CC=gcc
CFLAGS=-Wall -g

hola: hola.o auxhola.o

clean:
      $(RM) *.o hola core
```

Aunque las reglas para compilar hola.c y auxhola.c no aparecen si se ejecutarán en caso necesario y se utilizarán las variables definidas al igual que ocurría en el caso anterior. Es decir, las variables definidas por defecto aún si se modifican se expanden también en las reglas implícitas.

Se puede ver además que en este último listado ni siquiera aparece la instrucción para realizar el montaje de los dos ficheros objeto. Esto es porque también se emplea otra regla implícita.

De hecho si nuestro programa consta de un único fichero es posible compilarlo con make sin siguiera escribir un makefile. Para ello debemos indicar a make el nombre del ejecutable que deseamos y él buscará un fichero del mismo nombre pero con extensión .c que tomará como fuente del programa. Para compilar y enlazar un fichero adios.c con el fin de obtener un ejecutable adios simplemente debemos indicar:

```
$ make adios
```

5. Makefiles jerárquicos

Cuando en la construcción de una aplicación o sistema intervienen varios ejecutables, librerías o módulos resulta mucho más cómodo organizarlos en directorios.

Para cada módulo se escribe un makefile que se encarga de construir sus objetivos específicos y además se escriben otros makefiles en los directorios superiores que construirán objetivos de más alto nivel a partir de los anteriores.

Por ello, los módulos de los niveles superiores del árbol de directorios necesitan ejecutar los makefiles de los niveles inferiores. Esto se puede hacer con la opción -C que permite que indicando un directorio se procese el makefile que se encuentre en él. Veamos un ejemplo:

```
libs:
	$(MAKE) -C dir1
	$(MAKE) -C dir2

clean:
	$(RM) *.o
	$(MAKE) -C dir1 clean
	$(MAKE) -C dir2 clean
```

El valor de las variables en el fichero makefile no se propaga a la ejecución de los makefiles que se encuentren en los subdirectorios. La opción -C también puede indicarse desde línea de comandos.

6. Opciones.

Make acepta varios modificadores por línea de comando relativos a su comportamiento y la visualización de información durante la ejecución.

Estos son algunos de ellos:

```
-n  Imprime las órdenes que make efectúa pero sin ejecutarlas.

-f  <fichero>Indica a makeque debe utilizar <Fichero> como si fuera un makefile aunque tenga

-p  Imprime las definiciones de macros y las reglas implícitas

-C  dir Cambia al directorio y dir procesa el makefile que allí se encuentre.
```

7. Práctica.

Para crear un *Makefile* con múltiples archivos y generar un sólo ejecutable, basta con indicar el de las fuentes que se tienen. El siguiente ejemplo muestra la forma en que se pueden compilar 3 fuentes y generar un sólo ejecutable llamado hola. Si se requiere otro fuente, bastará con añadir su nombre a la variable *SOURCES* y automáticamente será compilado e insertado su código objeto al enlazado del ejecutable.

Listing 3: Ejemplo de Makefile

```
14 CC = gcc
15 SOURCES = hola.c chao.c mensaje.c
16 EXECUTABLE = hola
17 OBJECTS = $(SOURCES:.c=.o)
18 CFLAGS =
19
20 .c.o:
21         $(CC) $(CFLAGS) -c $< -o $@
22
23 $(EXECUTABLE): $(OBJECTS)
24         $(CC) $(OBJECTS) -o $(EXECUTABLE)
```

1. Elabora un programa que tenga dos archivos fuente, uno principal (llamado *hola.c*) que llame a una función (llamada *mensaje*) para imprimir en pantalla un texto que es pasado como argumento. El segundo archivo fuente contendrá la función mensaje y éste deberá llamarse *mensaje.c*.
2. Elabora un archivo *Makefile* con variables automáticas que compile el programa de la práctica 1. El programa deberá tener al menos 3 archivos fuente, el principal deberá llamarse *main.c*

8. Referencias.

[1] Escuela Superior de Informática, Curso de Aplicaciones de desarrollo en GNU/Linux.

[2] Página de manual en línea de GNU/Make..

[3] Guía de usuario de la Utilidad Make v1.0, Eduardo Dominguez, Carlos Villarrubia, Escuela Superior de Informática.

[4] The GNU Make Manual, Free Software Foundation.