

ORDEN MAKE

INTRODUCCIÓN

make es un programa de propósito general que construye un objetivo o meta a partir de prerequisites o dependencias. El objetivo puede ser un programa ejecutable, un documento PostScript o cualquier otra cosa. Los prerequisites pueden ser un código C, un fichero de texto T_EX, etc.

Supongamos la siguiente situación:

```
gcc -o foo foo.c bar.c baz.c
```

¿Por qué recompilarlo todo si sólo se modifica uno? ⇒ Pérdida de tiempo

La solución es guardar lo siguiente en un fichero que se debe llamar makefile o Makefile.

```
# Los comentarios se escriben de esta manera.  
  
edimh: main.o edit.o  
<Tab>      gcc -o edimh main.o edit.o  
  
main.o: main.c  
<Tab>      gcc -c main.c  
  
edit.o: edit.c  
<Tab>      gcc -c edit.c
```

En la primera línea edimh es un objetivo y main.o y edit.o son dependencias. El significado es: “ para obtener el fichero edimh necesitas los ficheros main.o y edit.o. Una vez que los tengas, edimh se construye con

```
gcc -o edimh main.o edit.o”.
```

Es muy importante observar el uso del tabulador. De lo anterior se deduce que en lugar de gcc se podría colocar cualquier otra orden.

Tras crear el fichero anterior debemos ejecutar

```
make
```

o bien

```
make edimh
```

En el segundo caso se indica qué objetivo se debe hacer. En el primero, se hace el objetivo por defecto, que es el primer objetivo que se indique en el fichero Makefile. El orden de los objetivos es indiferente, con la excepción del primero que se toma por defecto si sólo se ejecuta make.

MACROS

Un ejemplo de macro es `OBJECTS=main.o edit.o`. Y se utiliza como, por ejemplo,
`gcc -o edimh $(OBJECTS)`.

Otro ejemplo:

```
DRIVERS= drivers/block/scsi/scsi.a

ifdef CONFIG_SCSI
    DRIVERS := $(DRIVERS) drivers/scsi/scsi.a
endif
```

`CONFIG_SCSI` se define con `CONFIG_SCSI=yes` dentro del fichero Makefile, con `make CONFIG_SCSI=yes` objetivo, o con un `export CONFIG_SCSI=yes` en `.bashrc` (si se utiliza `bash` como shell).

Observar el uso de `:=` en lugar de `=`. Esto se hace para poder utilizar en la asignación la propia macro.

REGLAS DE SUFIJOS Y REGLAS DE PATRONES

Imaginemos que en un fichero Makefile aparecen unas líneas como:

```
.c.o:
    gcc -c $(FLAGS) $<
```

La primer línea es una regla de sufijos y significa que para obtener un fichero con sufijo “.o”, es necesario tener un fichero del mismo nombre pero con sufijo “.c”. “\$<” es una forma críptica de decir “el fichero de entrada”. `$(FLAGS)` es otro ejemplo de uso de macros, en este caso, para definir opciones de compilación.

Se podría ejecutar algo así como:

```
make CFLAGS="-O -g" edit.o
```

y en la pantalla aparecería algo como

```
gcc -c -O -g edit.c
```

Imaginemos que en un fichero Makefile, aparecen unas líneas como

```
%.o: %.c
    gcc -c -o $@ $(CFLAGS) $<
```

La primera línea es una regla de patrones, que tiene el mismo significado que la regla de sufijos anterior. El carácter % representa a cualquier cadena. La macro predefinida `$@` hace referencia al fichero de salida y `$<` tiene el mismo significado de antes.

EJECUCIÓN DE COMANDOS

Cualquier comando shell se puede ejecutar en un fichero Makefile. ¡Cuidado!, make ejecuta cada comando en un shell distinto. Por tanto, lo siguiente no hace lo esperado:

```
cd obj
HOST_DIR=/home/e
mv *.o $HOST_DIR
```

Sin embargo, si funcionará con

```
cd obj; HOST_DIR=/home/e; mv *.o $HOST_DIR
```

o con

```
cd obj;\
HOST_DIR=/home/e;\
mv *.o $HOST_DIR
```

OTRAS CONSIDERACIONES

make puede ser recursivo. La macro predefinida \$(MAKE) hace que se invoque a make.

HOST_NAME=\$(shell uname -n). El make GNU permite que la salida de una orden se guarde en una macro.

@: este símbolo al principio de una línea hace que no se produzca eco de la misma cuando se ejecuta el Makefile (igual que en un fichero de procesamiento por lotes de MS-DOS).

-: un guión al principio de una orden permite que continúe la ejecución del fichero Makefile, aunque falle la orden (útil para órdenes como cp o mv). Por defecto, make termina inmediatamente si una orden falla.