Uso básico de make
CC31A

Prof.: José Miguel Piquer
Auxs.: S. Kreft, P. Valenzuela

11 de agosto de 2007

Introducción a *make*

El programa *make* es de mucha utilidad a la hora de compilar otros programas, sobretodo si estos son extensos y su código fuente está dividido en varios archivos. El objetivo de estas notas es mostrar de manera breve la utilización de *make* en la compilación de programas en C.

Se debe tener en cuenta que la utilidad de *make* es mucho mayor de lo que pueda aparentar al mirar estas notas, dado que sus usos son variados, tanto en labores de compilación en distintos lenguajes de programación, procesamiento de documentos, o alguna labor automatizable.

Makefile

Generalmente, *make* trabaja con los llamados makefiles, los que contienen conjuntos de reglas y macros que ayudan a definir o a aplicar las reglas. Por omisión el archivo de donde son leídas las reglas se llama *Makefile*. Sin embargo es posible utilizar algún otro archivo especificando la opción *make -f archivo*. Las reglas poseen la siguiente estructura,

```
objetivo: dependencias
    comandos
```

En cada regla, *objetivo* es el archivo que se desea generar, *dependencias* son los archivos u otras reglas necesarias para poder generar el *objetivo*, y *comandos* son, precisamente, los comandos que deben ser ejecutados para, una vez satisfechas las dependencias, generar el *objetivo*.

Es importante notar que las líneas que contienen los comandos, *deben* comenzar con una tabulación.

Un ejemplo de regla simple es,

```
hola: hola.c hola.h
    gcc -Wall -ansi -pedantic -o hola hola.c
```

La regla anterior indica que para construir el objetivo *hola*, debe existir en el directorio actual los archivos hola.c y hola.h. Si estás dependencias se cumplen, entonces se procede a compilar y generar el ejecutable *hola*.

Reglas

Anteriormente se mencionó que dentro de las dependencias de un objetivo puede encontrarse otra regla. Un ejemplo en que una regla es una dependencia de otra regla, es el siguiente,

```
default: tarea

tarea : estructuras.o funciones.o programa.o
    gcc estructuras.o funciones.o programa.o -o tarea

estructuras.o : estructuras.c estructuras.h
    gcc estructuras.c -o estructuras.o -c

funciones.o : funciones.c funciones.h
    gcc funciones.c -o funciones.o -c

programa.o : programa.c programa.h
    gcc programa.c -o programa.o -c

clean :
    rm -rf *.o tarea
```

Este Makefile incluye la regla *default*, la cual se usa cuando make se llama sin parámetros. En la regla *default*, podemos ver que hay otra regla como dependencia, *tarea*. De este mismo modo, la regla *tarea* posee como dependencia otras tres reglas, la cuales siguen el formato de las reglas vistas en los primeros ejemplos.

Adicionalmente, se define la regla *clean*, la cual no tiene intrucciones de compilación, pero sí de limpieza. Para ejecutar esta regla se debe usar en la línea de comandos *make clean*.

Macros

Es posible definir macros en un makefile. Más aún, su uso se recomienda debido a que es más simple modificar una macro que cambiar todas las reglas en que se use una macro. Por ejemplo, para efectos del curso es necesario utilizar varias opciones al momento de compilar. En vez de definir en cada regla estas opciones, podemos crear una macro que las contenga.

```
CC=gcc
CFLAGS=-pedantic -Wall -ansi
```

```
default: tarea

tarea : estructuras.o funciones.o programa.o
    $(CC) $(CFLAGS) estructuras.o funciones.o programa.o -o tarea

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

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

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

clean :
    rm -rf *.o tarea
```

De esta forma, si decidimos usar una versión distinta de gcc o dejar de usar la opción -ansi, no será necesario cambiar todas las reglas del makefile, sino únicamente modificar la macro correspondiente.

Como convención se usa las macros CC, CFLAGS, LD y LDFLAGS para hacer referencia al compilador de C, las opciones de compilación, el linker, y las opciones del linker, respectivamente. Si usted decide usar estos nombres con otros propósitos, los programadores que lean sus makefiles, lo odiarán profundamente a usted y a sus makefiles.

Más Macros

Si uno está en este departamento, gusta entonces de las declaraciones cortas y elegantes, aunque todo el mundo diga que en realidad es mejor ser explícito, como en la guía de teléfonos. De todas maneras, miles de líneas de código como *if(!puntero)* se escriben al año y los programadores son felices ante tales expresiones.

Como *make* no es menos, podemos hacer nuestros makefiles más compactos. Retomemos el ejemplo largo de la sección anterior, donde hay muchas reglas casi idénticas donde sólo cambia el nombre de archivo, pero los sufijos son los mismos tanto en los objetivos como en las dependencias.

Podemos escribir,

```
CC=gcc
CFLAGS=-pedantic -Wall -ansi
OBJ=estructuras.o funciones.o programa.o
BIN=tarea

default: tarea
```

```
tarea: $(OBJ)
    $(CC) $(CFLAGS) $^ -o $(BIN)

%.o: %.c %.h
    $(CC) $(CFLAGS) -c -o $@ $<

clean :
    rm -rf *.o tarea
```

Donde se define la macro OBJ para contener todas las dependencias del programa a generar. También se utiliza una regla relativa a los sufijos de un archivo, que indica que para crear un archivo .o, se necesita un archivo con el mismo nombre base, pero con sufijo .c y otro con sufijo .h.

También se introducen las macros especiales $@ y $^, donde la primera representa el objetivo de la regla actual (por lo que en el caso de la regla de los sufijos, los objetivos son del tipo *archivo.o* y son usados para especificar el archivo de salida de gcc), y la segunda indica todas las dependencias de un objetivo (en el ejemplo, *estructuras.o funciones.o programa.o*). La macro $< indica la primera dependencia de un objetivo.

Si no se tiene cuidado, el deseo de compactar el makefile puede convertirse en un vicio, y se puede escribir expresiones del tipo,

```
CC=gcc
CFLAGS=-pedantic -Wall -ansi
OBJECTS := $(patsubst %.c,%.o,$(wildcard *.c))

tarea : $(OBJECTS)
    $(CC) $(CFLAGS) $(OBJECTS) -o $@

%.o: %.c %.h
    $(CC) $(CFLAGS) -c -o $@ $<

clean :
    rm -rf *.o tarea
```

Donde se define la macro OBJECTS, la cual contiene los nombres de todos los archivos de sufijo .o que se puede generar a partir de un archivo de sufijo .c. Para tales efectos, se utiliza la funcion *$(wildcard patrón)*, que devuelve una lista de archivos en el directorio actual cuyo nombre calza con el patrón. En el ejemplo son devueltos todos los archivos de sufijo .c.

Luego, dada la lista de archivos de sufijo .c, se reeemplaza dicho sufijo por el sufijo .o utilizando la función *$(patsubst pattern,replacement,text)*.

Finalmente, se establece una regla que utiliza la macro OBJECTS como dependencia. El resultado es que se compila todos los archivos fuente de un directorio, obteniendo un único ejecutable a partir de ellos. Ideal para flojos.

Como esto es una introducción y además hace sueño, el resto se lo dejamos a google.