Makefile

Mauro Toscano,Dr. Mariano Méndez
Correcciones: Martin Dardis, Lorenzo Gimenez[1]

[1]Facultad De Ingeniería. Universidad de Buenos Aires

19 de agosto de 2018

1. Motivación

El proceso de desarrollo de software implica varias etapas. Una de estas etapas es la implementación de programas, también conocida como programación. Esta etapa además se divide en distintas tareas. Cada una de estas tareas se realiza cada vez que se desea compilar un programa. Una de las premisas de este curso es que no se utilizarán herramientas conocidas como IDE (Integrated Development Envirronment) que muchas veces realizan varias de estas tareas en forma automatizada para los programadores, la idea por el contrario es que se conozca en detalle que es lo que se está realizando en cada paso, en este caso del proceso de compilación.

Si se recuerda el ciclo esencial en el cual se trabaja cuando se resuelven problemas con una computadora, el mismo consiste en:

- Análisis del problema.
- Diseño de la solución.
- Implantación del programa.
- Compilación, Ejecución y Prueba.
- Mejora de la solución obtenida.

una vez que el programa se encuentra implementado en su totalidad o parcialmente, el desarrollador muy posiblemente desee probar parte de la solución construida. Para ello es necesario compilar el programa como ya se ha visto. En este proceso se ve involucrado el sistema de compilación, que esta compuesto por: pre-procesador, compilar, ensamblador y link-editor. En un sistema linux-like, en el cual se encuentra instalado GNU gcc como sistema de compilación, estos componentes son:

- El pre-procesador de C : cpp
- El compilador: gcc
- El ensamblador: as
- El link-editor: ld

Suponiendo que se tenga un programa muy básico escrito en lenguaje C, como el que se muestra a continuación llamado ex1.c:

```
1 #include<stdio.h>
2
3 int main (){
4         printf("hola, mundo \n");
5         return 0;
6 }
```

Mauro Toscano,Dr. Mariano Méndez
Correcciones: Martin Dardis, Lorenzo Gimenez

El proceso de compilación se realiza llamando a cada componente del sistema de compilación (preprocesador, compilador, ensamblador y link-editor), con los parámetros adecuados para lograr finalmente el archivo binario que una computadora puede entender. En este caso:

```
$ gcc ex1.c -o ex1
```

con este comando se le dice al sistema de compilación que pre-procese ex1.c , posteriormente lo compile, lo ensamble y por último lo link-edite generando un archivo ejecutable, el proceso exacto es el que se muestra en la Figura 1. Si bien el proceso una vez entendido no es extremadamente complejo, si puede ser extenuante, cuando nuestro software crece.

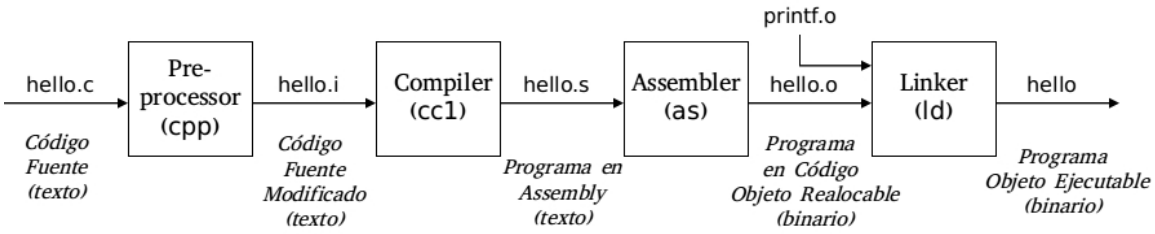

Figura 1: Proceso de Compilación

La cantidad de llamadas al compilador o sus parámetros irá aumentando a medida que el programa crezca, haciendo que sea difícil recordar exactamente todo lo que necesitamos pedirle que haga. Con el linker sucede algo similar.

Para empeorar aún mas la situación.

Para simplificar este proceso, es que se creo una herramienta que llamada "Make". La misma nos permitirá hacer un pequeño programa que se encargue de realizar todo lo necesario para llevar nuestro código a un binario. Con la sencillez de tan solo indicar la palabra "make.[en] la terminal.

2. Algo de historia, y además, ¿Qué es make?

Make fue desarrollado por Stuart Feldman en los laboratorios Dell, en el año 1976, y comenzó a utilizarse en PWB/UNIX 1.0 en el año 77.

Su creador había participado en en el grupo que desarrollo UNIX, y es también el autor del primer compilador de FORTRAN 77.

La herramienta, es principalmente un automatizado para el proceso de compilación y enlazado de un programa. Sirviendo adicionalmente para crear pequeños programas que nos pueden interesar para el mismo proceso.

3. ¿Como usamos Make?

3.1. Reglas

Para utilizar make, necesitamos entender como funciona algo que dentro de nuestro programa se llamaran "reglas". Las reglas constatan de algo a lo que queremos llegar,que llamaremos objetivo, objetos que necesitamos, que llamaremos pre-requisitos , y una receta para hacerlo. Esto se ve de la siguiente forma

```
objetivo ... : prerrequisitos ...
receta
...
...
```

Mauro Toscano,Dr. Mariano Méndez
Correcciones: Martin Dardis, Lorenzo Gimenez

3.2. Primer make

Supongamos que tenemos nuestro pequeño hola_mundo.c que queremos compilar. Debemos entonces crear un archivo llamado makefile donde colocaremos nuestra primer regla que hará lo deseado. La misma tiene el objetivo de llegar a un ejecutable que llamaremos hola_mundo. Para esto requiere hola_mundo.c. Y llegara con el código al ejecutable llamando a gcc. Nuestra regla entonces sera:

```
holaMundo.exe: hola_mundo.c
	gcc hola_mundo.c -o hola_mundo -std=c99 -Wall
```

Y en este caso muy sencillo, es todo en el makefile. Para compilar nuestro programa, podemos ahora escribir make en la terminal

3.3. Compilando múltiples bibliotecas propias sencillamente

Ahora nuestro programa ha crecido, y ya tiene diversos archivos, que deben ser compilados para que funcione el programa. En este caso sencillo, todas las bibliotecas son estandar o las hemos hechos nosotros. Para mostrar el ejemplo, tendremos nuestro main en "hola_mundo.c". Además, "hola_mundo.c", utilizará las bibliotecas que desarrollamos en "hola.c", y "mundo.c". A cada una además le creamos un header con los mismos nombres y terminación .h

La solución mas sencilla con lo que sabemos hasta ahora es delegar a gcc la mayor parte del trabajo.Y entonces hacer todo con una sola receta. Esto es sencillo, pero puede no ser la mejor forma de resolver el problema:

```
hola_mundo: hola_mundo.c hola.c hola.h mundo.c mundo.h
	gcc hola_mundo.c hola.c mundo.c -o hola_mundo -std=c99 -Wall -O2
```

Esta implementación no nos proporcionara una ventaja fundamental de make, que es solo recompilar los modulos que han cambiado. Por otro lado, eventualmente sera dificil entender todos los parametros del gcc. Ambas problemáticas seran solucionadas en las próximas implementaciones.

3.4. Compilando multiples bibliotecas detalladamente

Para lograr que no se re-compilen las bibliotecas que no han cambiado, deberemos agregar para cada una regla que las compile. Luego cada biblioteca compilada sera utilizada para armar el programa principal.

Notese que podríamos compilar con diferentes parámetros diferentes bibliotecas.

La solución nos queda:

```
hola_mundo: hola_mundo.c hola.o mundo.o hola.h mundo.h
	gcc hola_mundo.c hola.o mundo.o -o hola_mundo -std=c99 -Wall -O2 -Werror

hola.o: hola.c
	gcc -o hola.o -c hola.c -std=c99 -Wall -O2 -Werror

mundo.o: mundo.c
	gcc -o mundo.o -c mundo.c -std=c99 -Wall -O2 -Werror
```

Podemos ver que con el comando -c detrás del nombre del .c compilamos cada biblioteca por separado.

3.5. Variables

Aunque no fue el caso anterior, en general tendremos muchas reglas. Y seguramente, muchas tengan cosas en común. Por ejemplo, quizás compilemos siempre con GCC y con un set específico de flags. Y en algunas recetas en particular pongamos flags de debug. Estos parámetros suelen repetirse. Por lo que no es práctico escribirlos una y otra vez.

Para solucionar esto, podemos usar variables. Estas se definen directamente con una etiqueta, y se les asigna un valor con el signo $=$. Luego donde se necesita usarlas se escribe el signo $ y se coloca el nombre de la variable entre paréntesis.

Mauro Toscano,Dr. Mariano Méndez
Correcciones: Martin Dardis, Lorenzo Gimenez

Agregando variables tenemos ahora:

```
CC=gcc
CFLAGS= -std=c99 -Wall -O2 -Werror

hola_mundo: hola_mundo.c hola.o mundo.o hola.h mundo.h
	$(CC) hola_mundo.c hola.o mundo.o -o hola_mundo

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

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

Notese que a la variable que almacena el compilador compilador se lo llama CC por convención, y a las flags CFLAGS por el mismo motivo.

3.6. Función clean

Se suele querer luego de compilar, limpiar todos los archivos creados para volver al estado previo a la compilación. Para hacer esto, podemos crear una pequeña regla al final que se encargue de borrar todos los archivos, y se escribe de esta manera:

```
.PHONY : clean
clean:
	-rm hola_mundo
	-rm *.o
```

Teniendola, podemos invocar a **make clean**, para que borre nuestros archivos. Se puede usar una variable que contenga todo lo que se quiere borrar.

A pesar de ser pequeña, es una regla que tiene muchos detalles importantes que pueden pasar desapercibidos.

Make parte de la primera regla, y usa las demás en tanto se necesiten para completarla. Por lo tanto nunca se debe poner como la regla del make esta función, puesto a que si no se correría el clean cuando se trate de usar el make.

Hay una linea llamada .PHONY : clean. Esto hace un clean mas robusto. Si existiera un archivo que se llamara clean, y no la pusieramos, vamos a tener un comportamiento indefinido.

Por comodidad hacemos uso de las wildcards que soporta la consola de bash. El *.c es reemplazado por todos los archivos que terminen en .c.

3.7. Variables automáticas

Si miramos las reglas para compilar las bibliotecas, podemos notar que son similares. Siempre para crear un .o se utiliza un archivo con el mismo nombre pero que termina en .c, y con esos dos nombres se arma la misma linea de gcc. Para resumir el trabajo, se pueden utilizar variables automáticas, que expresaran lo mismo.

La lista de variables automáticas que existen es extensa, así que brindaremos la solución en nuestro ejemplo y explicaremos solo las usadas. Referirse a la documentación de make para averiguar sobre otras.

```
CC=gcc
CFLAGS= -std=c99 -Wall -O2 -Werror

hola_mundo: hola_mundo.c hola.o mundo.o hola.h mundo.h
	$(CC) $^ -o $@

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

.PHONY : clean
clean:
	-rm hola_mundo
	-rm *.o
```

Mauro Toscano,Dr. Mariano Méndez
Correcciones: Martin Dardis, Lorenzo Gimenez

%.o indica que es la regla para cualquier cosa terminada en .o. Del otro lado indicamos que lo que necesitara para la receta es un archivo del mismo nombre %, pero terminado en .c. Esto queda %.c.

Luego la receta es idéntica, pero usamos las dos variables automáticas propiamente dichas. $<indica que ahí se escribirá allí el nombre del primer requisito. En este caso el único .c que se necesita. Por otro lado, $@ indica que ahí se escribirá el nombre de lo que queremos generar, el target de la regla.

`https://www.gnu.org/software/make/manual/make.html`