GNU Make Aracının Kullanımı

GNU make aracı (bundan sonra kısaca yalnızca “Make” denilecektir) Linux sistemlerinde temel bir araç kabul edilmektedir. Dolayısıyla bu sistemleri kurduğumuzda zaten default olarak make aracı da kurulmuş durumda olacaktır. (Tabii biz daha yeni bir sürüm çıkınca onu kendimiz bu sistemlere kurmak durumunda kalabiliriz.) Mac OS X sistemlerinde de Make aracı derleyici araçlarının içerisinde bulunan doğal bir araç biçimindedir. Dolayısıyla bu sistemlerde de geliştirme araçları kurulduğunda Make aracı da kurulmuş olmaktadır. Windows sistemlerinde ise bilindiği gibi GCC derleyicilerinin MinGW isimli port edilmiş bir uyarlaması kullanılmaktadır. MinGW içerisinde Make programı da bulunmaktadır. Yani biz Windows sistemlerinde MinGW isimli paketi kurduğumuzda bu paket içerisinde GCC derleyicisi, "ld" bağlayıcısı, Make aracı gibi pek çok araç bulunmaktadır. Zaten pek çok açık kaynak kodlu C/C++ IDE’si Windows’ta GCC'nin MinGW sürümünü kullandığı için bu araçlar kurulduğunda da muhtemelen Windows sistemlerinize MinGW kurulmuş olmaktadır. Örneğin biz Qt platformunu GCC ile derleme yapacak biçimde kurduğumuzda bu kurulum ayrıca MinGW paketini de sisteme kurmaktadır. Tabii bunlardan bağımsız olarak Windows sistemlerinde biz de MinGW paketini ayrıca indirip kurabiliriz. MinGW’nin de 32 bit derleme yapan ve 64 bit derleme yapan iki ayrı uyarlaması vardır.

Make aracı kendine özgü bir dil kullanmaktadır. İçersinde Make kodları bulunan dosyalara da Make dosyaları (Make files) denilmektedir.

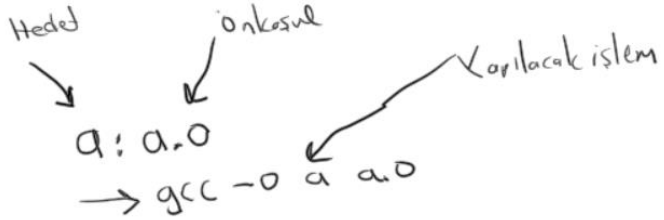
Bir Make dosyası kabaca kurallardan (rules) oluşmaktadır (rules). Bir kuralın genel biçimi şöyledir:

```
<hedefler (targets>) : [önkoşullar (prerequsites)] [; ilk yapılacak işlem]
    [yapılacak işlemler (recipes)]
```

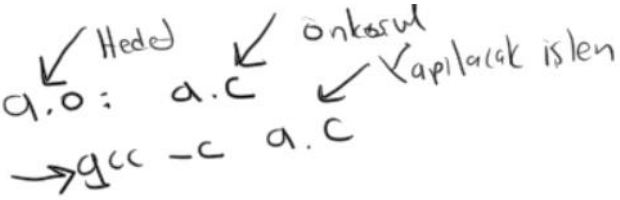
Kuralın yapılacak işlemler kısmının bir tab içeriden yazılması zorunludur. İlk yapılacak işlem istenirse önkoşullardan sonra aynı satırda ‘;’ atomundan sonra da belirtilebilir. Kurallar arasında boş satırlar bırakılabilir. Örneğin:

```
a: a.o
    gcc -o a a.o
a.o: a.c
    gcc -c a.c
```

Burada toplam iki tane kural vardır. Birinci kuralın hedefi “a” biçimindedir. Önkoşulları ise “a.o”dan oluşmaktadır. Yapılacak işlem ise “gcc -o a a.o” biçimindedir:



İkinci kural da benzer biçimde düzenlenmiştir:



Bir kuralın hedeflerinde ve önkoşullarında normal olarak dosya yol ifadeleri bulunur. Örneğin yukarıdaki birinci kuralda “a.o” ve “a” dosya belirten birer yol ifadesidir. İşte kuralın önkoşulunda belirtilen dosyanın (dosyaların) tarih ve zaman bilgisi kuralın hedefinde belirtilen dosyanın (dosyaların) tarih ve zaman bilgisinden daha yeniyse kuralda belirtilen işlemler çalıştırılmaktadır. Kuralın işlemleri komut satırından girilebilecek komutlardan oluşur. O halde birinci kural “eğer a.o dosyasının tarih ve zamanı a dosyasının tarih ve zamanından daha ileri ise gcc -o a.o komutunu çalıştır” anlamına gelmektedir. İkinci kural ise “eğer a.c dosyasının tarih ve zamanı a.o dosyasının tarih ve zamanından daha ileri ise gcc -c a.c komutunu çalıştır” anlamına gelir. Şimdi “a.c” dosyası üzeride bir değişikilik yaptığımızı düşünelim. Artık “a.c” dosyasının tarih ve zamanı “a.o” dosyasının tarih ve zamanından daha ileri olacaktır. Bu durumda “gcc -c a.c” komutu çalıştırılacak ve dosya derlenecektir. Bu derleme sonucunda “a.o” dosyası oluşacaktır. Bu durumda “a.o” dosyasının tarih ve zamanı “a” dosyasının tarih ve zamanından daha ileri olduğu için bu kez “gcc -o a.o” komutu çalıştırılacak ve çalıştırılabilir “a” dosyası elde edilecektir. Başka bir deyişle bu sistemde biz “a.c” dosyasında değişiklik yaptığımızda derleme ve link işlemi yapılıp yeniden çalıştırılabilir dosya elde edilecektir.

Pekiyi yazılan make dosyası nasıl işletilecektir? İşte bu dosyayı işletmek için Linux ve Mac OS X sistemlerinde “make” isimli programdan, Windows sistemlerinde de MinGW’deki “mingw32-make” isimli programdan ya da Microsoft’un “nmake” isimli programından faydalanılır. Make programının komut satırından en basit kullanımı şöyledir:

```
make
```

Hiç bir komut satırı argümanın girilmediği durumda make programı sırasıyla GNUmakefile, makefile ve Makefile isimli dosyaları o andaki geçerli dizinde arar. Bunlardan hangisini ilk kez bulursa make dosyası olarak onu işler. Genellikle programcılar make dosyasının ismini “Makefile” biçiminde verme eğilimindedir. Tabii make dosyasının ismi aslında istenildiği gibi de verilebilir. Bu durumda bizim bu dosyayı komut satırında “-f” ya da “--file” seçeneği ile belirtmemiz gerekir. Örneğin:

```
make -f test.make
```

Burada "test.make" isimli dosya işleme sokulacaktır.

Pekiyi make dosyası içerisindeki kurallar hangi sıraya göre işletilmektedir? Öncelikle bir make dosyasında nihai bir hedefin belirlenmesi gerekir. Nihai hedef en sonunda varılmak istenen hedeftir. Nihai hedef komut satırında seçeneksiz argüman biçiminde verilmektedir. Örneğin:

```
make a -f test.make
```

Burada nihai hedef "a" hedefidir, işlenecek make dosyası ise “test.make” isimli dosyadır. Örneğin:

```
make a
```

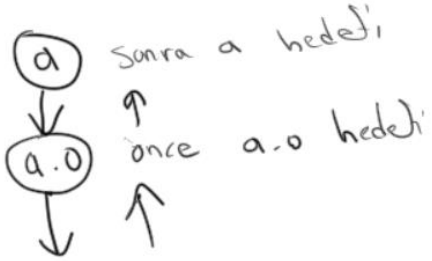
Buradaki nihai hedef yine "a" hedefidir. Ancak işlenecek Make dosyası sırasıyla dizindeki GNUmakefile, makefile ya da Makefile dosyalarından ilk bulunanıdır. Tabii komut satırında make programı çalıştırılırken nihai hedef hiç belirtilmeyebilir de. Bu durumda nihai hedef ilk kuralın ilk hedefi olur. Örneğin Makefile isimli aşağıdaki bir dosya bulunuyor olsun:

```
a: a.o
    gcc -o a a.o
a.o: a.c
```

```
    gcc -c a.c
```

Biz de komut satırında yalnızca “make” demiş olalım. Bu durumda nihai hedef “a” hedefi olacaktır.

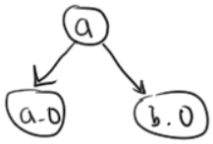
Make işlemlere başlamadan önce nihai hedefe varabilmek için gereken bağımlılık ağacı (dependency tree) oluşturulur, sonra işlemler aşağıdan yukarıya doğru yapılır. Örneğin yukarıdaki Make dosyasında nihai hedef olan “a” için bağımlılık ağacı şöyle oluşturulacaktır:



Şimdi aşağıdaki gibi bir Make dosyası söz konusu olsun:

```
a: a.o b.o
   gcc -o a a.o b.o
a.o: a.c
   gcc -c a.c
b.o: b.c
   gcc -c b.c
```

Eğer bir kuralda birden fazla önkoşul varsa bu önkoşullara ilişkin dosyaların herhangi birinin tarih ve zamanı hedefe ilişkin dosyanın tarih ve zamanından daha ileride ise belirtilen işlemler yapılır. Burada “a” hedefi için iki önkoşul verilmiştir. Bunlardan herhangi biri daha yeniyse yeniden link işlemi yapılacaktır. Buradaki hedeflerin bağımlılık ağacı şöyledir:



Burada kesinlikle önce “a.o” ya da “b.o” hedefi gerçekleştirilmeye çalışılacaktır. Çünkü “a” hedefi bu hedeflere bağlıdır. Fakat "a.o" hedefinin mi yoksa "b.o" heedefinin mi daha önce yapılacağının bir önemi yoktur.

Bir make dosyasında aslında kuralların yazım sırasının bir önemi yoktur. Ancak nihai hedefe dikkat edilmelidir. Örneğin yukarıdaki make dosyasında kuralları aşağıdaki gibi yazmış olalım:

```
b.o: b.c
    gcc -c b.c
a.o: a.c
    gcc -c a.c
a: a.o b.o
    gcc -o a a.o b.o
```

Burada aslında “a” hedefi için değişen birşey yoktur. Ancak nihai hedef değişmiştir. Dolayısıyla bizim make işlemini artık hedef belirterek yapmamız uygun olur:

```
make a
```

Eğer biz make işleminde hedef belirtmeseydik bu durumda nihai hedef “b.o” olacağı için yalnızca o hedef yapılacaktı. O halde okunabilirlik ve kolay kullanım bakımından nihai hedefi en yukarıya yazmak iyi bir tekniktir.

Bir kuralda hedefte belirtilen dosya yoksa ne olur? İşte bu durumda her zaman hedefin güncelliğini kaybettiği varsayılır ve ilgili işlemler yapılır. Zaten yukarıdaki örneklerde de aslında işin başında "a" hedefine ilişkin "a.o" ve "b.o" hedeflerine ilişkin dosyalar yoktur. Ancak kuralın önkoşulunda belirtilen dosyalar yoksa ya da bu dosyalar başka bir kuralın hedefi değilse bu durum hata olarak değerlendirilmektedir.

Bir kuralda önkoşul olmak zorunda değildir. Bu durumda bu kural işletildiğinde kuralda işlemler her zaman yapılır. Örneğin:

```
a: a.o b.o
    gcc -o a a.o b.o
b.o: b.c
    gcc -c b.c
a.o: a.c
    gcc -c a.c
clean:
    rm -f *.o a
```

Burada "clean" isimli hedef amaç dosyaları ve çalıştırılabilir dosyayı silmektedir (-f seçeneği dosya yoksa rm komutunun uyarı mesajı vermesini engellemek için kullanılmıştır). Bu durumda biz:

```
make clean
```

biçiminde make programını çalıştırdığımızda bu dosyalar silinecektir. Dolayısıyla bir daha make yaptığımızda tüm işlemler baştan sona yenilecektir. Pek çok Make dosyasında özel bir hedef (bunun ismi genellikle “install” biçimindedir) üretilen dosyaları belli dizinlere kopyalamaktadır. Örneğin:

```
a: a.o b.o
    gcc -o a a.o b.o
b.o: b.c
    gcc -c b.c
a.o: a.c
    gcc -c a.c
clean:
    rm -f *.o a
install:
    cp a /home/test/a
```

Yazımı kolaylaştırmak için Make dilinede makrolar (ya da değişkenler) da kullanılabilmektedir. Makro bildiriminin genel biçimi şöyledir:

<değişken> = <değer>
<değişken> := <değer>

Makro tanımlarken “=” ya da “:=” arasında özyineleme bakımından farklılık vardır. Bunun dışında iki biçim de aynıdır. Biz burada “=” karakterini kullanacağız. Örneğin:

```
CFLAGS = -c -Wall -g
SOURCE = a.c b.c c.c d.c
```

Makroların değerleri $(değişken ismi) ile elde edilmektedir. Örneğin:

```
CFLAGS=-c -g -Wall
OBJS=a.o b.o

a: $(OBJS)
    gcc -o a $(OBJS)
b.o: b.c
    gcc $(CFLAGS) b.c
a.o: a.c
    gcc $(CFLAGS) a.c
clean:
    rm -f *.o a
install:
    cp a /home/test/a
```

Bazı makrolar önceden tanımlanmıştır (predefined) ve bunların default değerleri vardır. Ancak programcı isterse bunları değiştirebilir. Önceden tanımlanmış önemli olanlarından bazıları aşağıdaki tabloda belirtilmektedir:

Makro	Default Anlamı
CC	cc (bu da zaten gcc'ye sembolik link yapılmış)
CXX	g++ (C++ derleyicisi)
CPP	$(CC) -E (C dosyasını yalnızca önişlemciye sokmak için)
LEX	lex (bu da flex'e sembolik link yapılmış)
YACC	yacc (bu da bison'a sembolik link yapılmış)
CFLAGS	cc için derleme seçenkleri. Default durum boş.
CXXFLAGS	g++ için derleme seçenekleri. Defulet durum boş.
AS	as (GNU sembolik makine dili derleyicisi)
AR	ar (statik kütüphane oluşturmak için araç)
RM	rm -f (remove komutu)

Genel olarak kurallarda joker karakterleri kullanılabilir. Örneğin:

```
$(EXECUTABLE): *.o
    gcc -o *.o
```

Ancak joker karakterleri makroların değer kısımlarında kullanılırsa açım yapılmaz. Bunun için "wildcard" belirlemesinin kullanılması gerekmektedir. Örneğin:

```
OBJS=*.o
```

gibi bir makro aşağıdaki gibi kullanılmış olsun:

```
$(EXECUTABLE): $(OBJS)
    gcc -o $(OBJS)
```

Burada istemnilen işlem yapılmaz. Yani $(OBJS) açılımı o dizindeki tüm ".o" dosyaları anlamına gelmez. Sanki "*.o" isimli bir dosya gibi ele alınır. İşte makrolarda joker karakterlerinin kullanımı aşağıdaki gibi yapılmalıdır:

```
OBJS=$(wildcard *.o)
```

Bazı makro isimlerine make terminolojisinde "otomatik değişkenler (automatic variables)" denilmektedir. Bunların özel bazı anlamları vardır:

Otomatik Makro (Değişken)	Anlamı

$@	Kuralın hedefindeki dosya ismini belirtir. Örneğin:
	`a.o: a.c`
	`  $(CC) -o $@ $(CFLAGS) a.c`
	
	Burada $@ a.o anlamına gelmektedir.

$?	Kuralın önkoşul kısmında güncel olmaktan çıkmış (yani tarih ve zamanı hedeften daha ileri hale gelmiş) dosyaların listesini belirtir. Örneğin:
	
	`print: *.c`
	`  lpr -p $?`
	`  touch print`
	
	Burada print hedefi çalıştırıldığında (yani make print denildiğinde) son print işleminden sonra değişltirilmiş olan C dosyaları print edilmektedir.

$<	İlk önkoşulu belirtir. Örneğin:
	
	`b.o: b.c`
	`  $(CC) $(CFLAGS) $<`
	
	Burada $< değişkeni "b.c"yi belirtmektedir.

$^	Tüm ön koşulları belirtir

Kuralların önkoşullarında birden fazla dosyanın bulunması bazı işlemleri kolaylaştırmaktadır. Örneğin:

```
a.o : a.c a.h x.h y.h z.h
    gcc -c $<
```

Burada kuraldaki önkoşullara ilişkin herhangi bir dosya değiştiğinde komut çalıştırılacaktır.

Kuralların hedefleri birden fazla kez yinelenbilir. Örneğin:

```
a.o:    a.c
    gcc -c a.c

a.o: a.h x.h y.h z.h
    gcc -c a.c
```

Bu durumda ilgili dosyalar güncellendiğini kaybettiğinde o hedefe ilişkin belirtilen tüm kurallar gerçekleştirilir.

Make ile ilgili daha detaylı bilgi için "GNU Make Reference" dokümanlarına başvurulabilir.