GNU Paket Yapılandırma Sistemi

Makefile, autoconf, automake kullanımı

Yazan:
Murat Demirten

6 Şubat 2007

Özet

Bu belge *NIX sistemler üzerinde uygulama geliştiren, geliştirmek isteyenlere yardımcı olabilmek amacıyla hazırlanmıştır. Belge kapsamında `Makefile` dosyalarının nasıl hazırlanabileceği, büyük projeler için birden fazla `Makefile` dosyası yazma/yazmama, bunun yerine **`autoconf`** ve **`automake`** kullanarak `Makefile` dosyalarının otomatik üretilebilmesi, **`autoconf`**'un özelliklerini kullanarak taşınabilir kod geliştirme gibi konular üzerinde durulacaktır.

Konu Başlıkları

Geçmiş

Versiyon 1.0.0	21 Haziran 2003	murat
İlk versiyon, bir gece vakti uykum yok, başlıyorum		

Yasal Uyarı

1. Giriş

*NIX sistemlerde uygulama geliştirmek gerçekten eğlenceli bir iştir. Öğrenilecek o kadar çok şey var ki. Uzun zamandır Linux ile uğraşmama rağmen sürekli çok şeyler öğrenmeye devam ediyorum, işin en güzel yanı da bu. Üç yıl öncesine kadar geliştirdiğim uygulamalarda yeniden derleme işlemini tek tek komutlarla yapıyordum. Fakat programlar büyüdükçe bu işi yapmak inanılmaz zor hale geldi. Hele bir de yazdığınız programların başkaları tarafından da derlenmesi söz konusu olunca işler daha da karışmaktadır. O zaman `Makefile` yazmam gerektiğini anladım. Oturdum 3–5 saatlik bir araştırmadan sonra **make** kullanımıyla ilgili pek çok bilgi edindim ve orada kaybettiğim (!) 3–5 saat bana şimdiye dek kat kat fazlasını kazandırdı. Bu belgeyi okuduğunuza göre bir şekilde siz de bu konulara ilgi duyuyorsunuzdur. Eğer henüz yeni iseniz, bir iki saatinizi ayırıp burada yazılanları uygulamanızı şiddetle öneririm.

Bu belgede önce nasıl kendi `Makefile` dosyalarımızı oluşturabileceğimizden bahsedeceğiz. Temel **make** kullanımını öğreneceğiz. Ardından taşınabilirlik özelliğine sahip, daha büyük uygulamaların derlenebilmesi için tek tek tüm `Makefile` dosyalarını elle oluşturmanın zorluğundan bahsedecek ve sizleri **autoconf** ve **automake** kullanmaya zorlayacağız. Sizi ikna ettikten sonra ise örneklerle bu araçların da kullanımından bahsedeceğiz.

2. Makefile Kullanımı

Uygulama geliştirirken sıklıkla nesne dosyalarımızı yeniden ve yeniden oluşturmak zorunda kalırız. Yerine göre **gcc**, **ld**, **ar** vb. uygulamaları tekrar tekrar aynı parametrelere çağırırız. İşte **make** ugulaması, programların yeniden derlenme sürecini otomatik hale getirmek, sadece değişen kısımların yeniden derlenmesini sağlamak suretiyle zamandan kazanmak ve işlemleri her zaman otomatik olarak doğru sırada yapmak için tasarlanmıştır.

2.1. Temel Kurallar

make uygulaması çalıştırıldığında, bulunulan dizinde sırasıyla `GNUmakefile`, `makefile` ve `Makefile` dosyalarını arar. Alternatif olarak `-f` seçeneği ile `Makefile` olarak kullanacağınız dosyayı da belirlemeniz mümkün olsa da standartların dışına çıkmamakta fayda var. **make** neyi nasıl yapacağını bu dosyalardan öğrenecektir. Eğer bulunduğunuz dizinde bir `Makefile` yok ise aşağıdaki gibi bir çıktı alacaksınız demektir:

```
$ make
make: *** No targets specified and no makefile found.  Stop.
```

İpucu

Genel kabul görmüşlüğü ve göz alışkanlığı açısından dosya adı olarak alternatiflerin yerine `Makefile` kullanmanızı öneririm.

Bir `Makefile` aslında işlemlerin nasıl yapılacağını gösteren kural tanımlamalarından oluşmaktadır. Genel olarak dosyanın biçimi aşağıdaki gibidir:

```
hedef: bağımlılıklar
<TAB> komut
<TAB> komut
<TAB> ...
Diğer kurala geçmeden bir boş satır
...
```

Burada en sık yapacağımız hata <**TAB**> tuşuna basmayı unutmak olacaktır. `Makefile` dosyasını hazırladığınız metin düzenleyiciden kaynaklanan bir sorun da olabilir. En iyisi **emacs** kullanarak *makefile–mode* ile yazmaktır, böylece hata yapma olasılığınız oldukça azalacaktır.

Kurallar arasında bir satır boş bırakılması *GNU make* için zorunlu olmamakla birlikte bazı Unix sürümleriyle uyumluluk için boşluk bırakılması gereklidir.

İlk satırda *hedef*'in oluşturulmasında etkili olan, bağımlılık yaratan dosyalar birbirinden boşluk ile ayrılmış olarak tek satırda listelenir. Eğer bağımlılık kısmında yer alan dosyalardan en az birinin son değiştirilme tarihi, hedef'ten daha yeni ise, hedef yeniden oluşturulur. Diğer durumda hedefin yeniden oluşturulmasına gerek olmadığı anlaşılır, çünkü hedefin bağımlı olduğu dosyalarda son oluşturmadan sonra bir değişiklik olmamıştır. Sonraki satırlarda bağımlılık yaratan bu dosyalardan hedefin oluşturulabilmesi için gerekli komutlar yer alır. Şimdi basit bir örnek yapalım:

```
  test: test.c
      gcc -o test test.c
```

Bu örnekte hedef olarak **test** uygulaması derlenecektir. Uygulamanın bağımlı olduğu dosya `test.c`'dir. `test.c` dosyasında herhangi bir değişiklik olduğunda veya **test** silindiğinde, **gcc -o test test.c** komutu çalıştırılacak ve **test** yeniden oluşturulacaktır. Şimdi daha karışık bir örnek yapalım:

```
CC = gcc
CFLAGS = -O2 -Wall -pedantic
LIBS = -lm -lnsl

test: test.o
    $(CC) $(CFLAGS) $(LIBS) -o test test.o

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

clean:
    rm -f test *.o

install: test
    cp test /usr/bin
```

İlk satırda `CC` değişkenine kullanacağımız derleyiciyi atıyoruz. `Makefile` dosyaları içerisinde bu şekilde değişken tanımlaması yapıp, değişkeni dosya içerisinde `$(değişken)` olarak kullanabiliriz. İkinci satırda ise derleyiciye vereceğimiz bazı seçenekleri `CFLAGS` değişkenine atıyoruz. Üçüncü satırda uygulamamızın kullandığı kütüphaneleri listeledik. Ardından ilk kuralımız geliyor. **test** uygulaması `test.o` dosyasına bağımlı olarak belirtilmiş ve `test.o`'dan **test**'in oluşturulabilmesi için gerekli komut hemen altında listelenmiştir. Değişkenlerin değerlerini yerine koyduğumuzda komutumuz **gcc -O2 -Wall -pedantic -lm -lnsl -o test test.o** şeklinde olacaktır.

İkinci kuralımız `test.o`'nun nasıl oluşturulacağını belirtmektedir. Aslında bu iki kural bir önceki örnekte olduğu gibi birleştirilebilir, ancak mantığı anlatabilmek için burada ikiye bölünmüştür. `test.c` dosyasında bir değişiklik olduğunda `test.o` dosyası hemen altında listelenen komutla yeniden oluşturulur: **gcc -O2 -Wall -pedantic -c test.c**

Üçüncü kuralımızda çalıştığımız dizinde nasıl temizlik yapacağımızı belirtiyoruz. **make clean** komutunu çalıştırdığımızda **test** dosyası ve `.o` ile biten nesne dosyaları silinecektir. Bir sonraki kuralımız ise **install**. Bu kuralda da **test** dosyasında bir değişme olduğunda **cp test /usr/bin** komutu ile dosyayı `/usr/bin` dizini altına kopyalıyoruz.

`Makefile` içerisindeki her bir kural **make** uygulamasına seçenek olarak verilebilir ve ayrıca işletilebilir. Yukarıdaki gibi bir `Makefile` dosyasına sahipsek **make test.o** komutuyla sadece `test.o` için verilen kuralın çalıştırılmasını sağlayabiliriz. Veya **make install** komutuyla sadece **install** kuralının çalışmasını sağlayabiliriz. Ancak **install** aynı zamanda **test**'e bağımlı olduğundan **test**'in kuralı da çalışır. Aynı şekilde

test de test.o'ya bağlı olduğundan test.o kuralı da çalışacaktır. Komutu seçenek vermeden sadece **make** şeklinde çalıştırdığınızda ise Makefile dosyasını okur ve bulduğu ilk kuralı işler. Bizim örneğimizde ilk kural test olduğu için test dosyasının oluşturulabilmesi için gerekli işlemleri yapacaktır. Bu nedenle Makefile dosyalarında ilk kural çoğu zaman all: test install gibi olur. Böylece her defasında **make xxx** yazmak yerine sadece **make** yazarak hız kazanmış oluruz.

Bu örneği iyice anlamadan sonraki bölümlere devam etmeyiniz. **make** uygulamasının bu basit ama bir o kadar da güçlü mantığını tam olarak anladığınızda onu sadece kodunuzu derlemek için değil, çok farklı amaçlar için de kullanabileceğinizi göreceksiniz. Hemen bir örnek verelim, bir sanaldoku (web) uygulamanız var ve buradan isim:telefon şeklinde bir metin dosyasına giriş yapılıyor. Eğer bu metin dosyası değiştiğinde çalışacak şekilde bir kural tanımlarsanız, mesela metin dosyası her değiştiğinde bu dosyayı okuyup ayrıştırarak veritabanına kayıt edecek bir uygulamanın çalıştırılması sağlanabilir. Örneğimiz pek işe yarar bir şey olmadı ama eminim mantığı anlamışsınızdır.

Bilgi

Aslında **make** için verilebilecek en iyi örneklerden bir tanesi de Debian sanalyöresidir. Debian sanalyöresi, tamamen statik HTML sayfalardan oluşur. Bu sayede yansılanması daha kolay hale gelir ve statik sayfalar sanaldoku sunucusuna çok az yük getirir. Ancak binlerce sayfadan oluşan Debian sanalyöresi, statik olmasına rağmen çok hızlı güncellenebilmektedir. Aynı zamanda yöreyi ziyaret ettiyseniz farketmiş olacağınız gibi, sanaldoku istemciniz dil ayarına göre sayfanın o dile çevirilmiş bir sürümü mevcut ise karşınıza o getirilmektedir. Tüm bu dinamiklik alt tarafta kullanılan, çoğunluğu wml, binlerce dosya tarafından sağlanmaktadır. Her 3–4 saatte bir CVS'de bulunan kaynak kodu çekilerek **make** ile wml dosyalarından HTML dosyaları üretilmekte, sayfalar arası aşamalar düzenlenmekte, farklı dillere çevirilen sayfalar kontrol edilmekte, bazı programlar ve betikler çalıştırılmaktadır. Kısaca özetlemek gerekirse böyle ama gerçekte tüm yörenin yeniden oluşturulması için gerçekten oldukça karmaşık işlemler yapılmaktadır. İlgilenenler http://www.debian.org/devel/website/desc adresine bakabilir.

Yukarıdaki Makefile örneğimize tekrar dönelim. **make clean** komutunu çalıştırdığımızda derleme sonrasında oluşan dosyalar silinmektedir. Peki, bulunduğumuz dizinde ismi clean olan bir dosya mevcut ise ne olur?

```
$ make clean
make: 'clean' is up to date.
```

Gördüğünüz gibi clean adında bir dosya var olduğu ve clean için bağımlılık listesi olmadığından dolayı kuralın güncelliğini koruduğunu ve alttaki komutların çalıştırılmaması gerektiğini düşündü. İşte bu gibi durumlar için özel bir kural mevcuttur: .PHONY

Yukarıda anlatılan sorunu giderebilmek için Makefile dosyamızın içeriğine aşağıdaki kuralı da eklemeliyiz:

```
.PHONY: clean
```

Böylelikle **make clean** komutunun, bulunulan dizinde clean adında bir dosya olsa bile düzgün olarak çalışmasını sağlamış olduk.

2.2. Daha Karmaşık Makefile Dosyaları

Önceki bölümde temel olarak **make** kullanımı üzerinde durduk. Örnek bir Makefile hazırladık. Ancak tek bir kaynak dosyasından oluşturulan bir uygulama için **make** o kadar da yararlı bir şey değil. Zaten gerçekte de en küçük uygulama bile onlarca kaynak dosyadan oluşur. Şimdi böyle bir uygulama için Makefile hazırlayalım.

Örnek 1. Soyut kurallar kullanılmamış Makefile

```
CC = g++
CFLAGS = -O2 -Wall -pedantic
LIBS = -lnsl -lm
INCLUDES = -I/usr/local/include/custom

all: server client

server: ortak.o server.o list.o que.o \
            data.o hash.o
    $(CC) $(CFLAGS) $(LIBS) -o server ortak.o server.o \
            list.o que.o data.o hash.o

client: ortak.o client.o
    $(CC) $(CFLAGS) $(LIBS) -o client ortak.o client.o

ortak.o: ortak.cpp ortak.h
    $(CC) $(CFLAGS) $(INCLUDES) -c ortak.cpp

server.o: server.cpp server.h ortak.h
    $(CC) $(CFLAGS) $(INCLUDES) -c server.cpp

client.o: client.cpp client.h ortak.h
    $(CC) $(CFLAGS) $(INCLUDES) -c client.cpp

list.o: list.cpp list.h
    $(CC) $(CFLAGS) $(INCLUDES) -c list.cpp

que.o: que.cpp que.h
    $(CC) $(CFLAGS) $(INCLUDES) -c que.cpp

data.o: data.cpp data.h
    $(CC) $(CFLAGS) $(INCLUDES) -c data.cpp

hash.o: hash.cpp hash.h
    $(CC) $(CFLAGS) $(INCLUDES) -c hash.cpp

install: client server
    mkdir -p /usr/local/bin/test
    cp client /usr/local/bin/test
    cp server /usr/local/bin/test

uninstall:
    rm -rf /usr/local/bin/test

clean:
    rm -f *.o server client

.PHONY: clean
```

Kullandığımız derleyici, derleyici seçenekleri, kütüphaneler gibi değerleri değişkenlere atamakla neler kazandığımıza bir bakalım. Derleyici parametrelerini değiştirmeye karar verdiğimizde değişken kullanmıyor olsaydık 9 farklı yerde bu değişikliği el ile yapmak zorunda kalacaktır. Fakat şimdi ise sadece `CFLAGS` değişkeninin değerini değiştirmemiz yeterli olacaktır.

Ancak gene de yukarıdaki gibi bir `Makefile` yazmak uzun sürecek bir işlemdir. Eğer uygulamanız 60 cpp dosyasından oluşuyorsa ve 60 farklı nesne için tek tek kuralları yazmak zorunda kalıyorsanız bu hoş olmaz.

Çünkü tüm `.o` dosyalarını üretebilmek için vereceğimiz komut aynı: `$(CC) $(CFLAGS) $(INCLUDES) -c xxx.cpp`. Oysa biz 60 defa bu komutu tekrar yazmak zorundayız. İşte bu noktada soyut kurallar (*abstract rules*) imdadımıza yetişir.

Bir soyut kural `*.u1` uzantılı bir dosyadan nasıl `*.u2` uzantılı bir dosyanın üretileceğini tanımlar. Genel olarak kullanımı aşağıdaki gibidir:

```
.u1.u2:
    komutlar
    komutlar
...
```

Burada `u1` kaynak dosyanın uzantısı iken, `u2` hedef dosyanın uzantısıdır. Bu tür kullanımda dikkat ederseniz bağımlılık tanımlamaları yer almamaktadır. Çünkü tanımladığımız soyut genel kural için bağımlılık belirtmek çok anlamlı değildir. Bunun yerine `.u1` uzantılı bir dosyadan `.u2` uzantılı dosya üretmede istisnai olarak farklı bağımlılıkları olan kurallar da ileride vereceğimiz örnekte olduğu gibi belirtilebilir.

Soyut kurallar tanımlarken aşağıdaki özel değişkenleri kullanmak gerekecektir:

- **$<** Değiştiği zaman hedefin yeniden oluşturulması gereken bağımlılıkları gösterir.
- **$@** Hedefi temsil eder.
- **$^** Geçerli kural için tüm bağımlılıkları temsil eder.

Bu bilgiler ışığında hemen bir örnek verelim. Uzantısı `.cpp` olan bir kaynak kodundan nesne kodunu üretebilmek için aşağıdaki gibi bir kural tanımlayabiliriz:

```
.cpp.o:
    g++ -c $<
```

Şimdi biraz daha açıklık getirelim. Kaynak dosyamızın adı `helper.cpp` ve amacımız `helper.o` nesne dosyasını üretmek olsun. Yukarıdaki kural kaynak dosyamız için çalıştığında `.cpp.o:` satırı yüzünden `helper.cpp` oluşacak `helper.o` için bir bağımlılık durumunu alır. Bu nedenle $< değişkeni `helper.cpp`'yi gösterir. Bu sayede `helper.o` dosyası üretilmiş olacaktır.

Şimdi aynı mantıkla nesne dosyalarından çalıştırılabilir programımızı üretelim.

```
.o:
    g++ $^ -o $@
```

Bu biraz daha karışık çünkü çalıştırılabilir dosyamızın uzantısı olmayacak. Eğer tek bir uzantı verilmiş ise bunun birinci uzantı olduğu ve ikincinin boş olduğu düşünülür.

Soyut kurallar tanımladığımızda yapmamız gereken iki işlem daha var. Bunlardan birincisi kullandığımız uzantıların neler olduğunu belirtmektir. Bu işlem için `.SUFFIXES` özel değişkeni kullanılır:

```
.SUFFIXES: .cpp .o
```

Diğer yapmamız gereken işlem ise üretilecek çalıştırılabilir dosyamızın hangi nesne dosyalarına, nesne dosyalarımızın ise hangi kaynak dosyalarına bağımlı olduğunu belirtmek olacaktır. İşin en güç tarafı budur. Her zaman doğru değerleri yazmak o kadar kolay olmayabilir. Bu noktada **gcc** derleyicisi `-MM` seçeneğiyle bize yardımcı olacaktır. Aşağıdaki ekran çıktısına bakalım:

```
$ g++ -MM -c server.cpp
server.o: server.cpp server.h ortak.h
$
```

Görüldüğü gibi `server.o` için gerekli `Makefile` kuralını bizim için hatasız olarak verdi. Tek yapmamız gereken bu satırları kopyalayıp `Makefile` içerisine yapıştırmaktır. Şimdi bölümün başında verdiğimiz `Makefile` dosyasını bu yöntemle yeniden yazalım:

Örnek 2. Soyut kuralların kullanıldığı Makefile

```
CC = g++
CFLAGS = -O2 -Wall -pedantic
LIBS = -lnsl -lm
INCLUDES = -I/usr/local/include/custom
SERVER_nesneCTS = ortak.o server.o list.o que.o data.o hash.o
CLIENT_nesneCTS = ortak.o client.o

all: server client

.SUFFIXES: .cpp .o

.cpp.o:
    $(CC) $(CFLAGS) $(INCLUDES) -c $<

.o:
    $(CC) $(CFLAGS) $(LIBS) $^ -o $@

server: $(SERVER_nesneCTS)
client: $(CLIENT_nesneCTS)
ortak.o: ortak.cpp ortak.h
server.o: server.cpp server.h ortak.h
client.o: client.cpp client.h ortak.h
list.o: list.cpp list.h
que.o: que.cpp que.h
data.o: data.cpp data.h
hash.o: hash.cpp hash.h

install: client server
    mkdir -p /usr/local/bin/test
    cp client /usr/local/bin/test
    cp server /usr/local/bin/test

uninstall:
    rm -rf /usr/local/bin/test

clean:
    rm -f *.o server client

.PHONY: clean
```

3. Autoconf ve Automake Kullanımı

GNU Paket Kurgulama Sistemi iki temel amacın gerçekleştirilebilmesi için geliştirilmiştir: Programları platformlar arası daha rahat taşınabilir hale getirmek ve kaynak koddan program kurulumlarını mümkün olduğu kadar basite indirgeyebilmek.

Taşınabilir kod yazmak gerçekten oldukça zahmetli bir iştir. Hedef mimarinin ayrıntılı olarak özelliklerinin bilinmesi çoğu zaman mümkün değildir. Bir önceki bölümde örnek olarak yazdığımız `Makefile` dosyasında **`mkdir -p /usr/local/bin/test`** komutunu kullanmıştık. Oysa **`mkdir`** komutunun `-p` seçeneği tüm Unix sis-

temlerde aynı şekilde çalışmaz. Bu ve bunun gibi pek çok farklılık yüzünden her Unix sisteminde çalışabilecek bir `Makefile` yazmak çok zor iştir. Kullanılan kütüphanelerin sistemler arasındaki farklılıkları ise apayrı bir konudur. İşte GNU Paket Kurgulama Sistemi tüm bu zorlukların üstesinden gelebilmek için oluşturulmuştur. Kdevelop gibi programlar yeni proje oluşturduğunuzda b sistemi de otomatik olarak oluşturmaktadırlar. Ancak oluşan dosyalar fazlasıyla karışık olduğundan bu bölümde çok daha basit örneklerle yapıyı anlatmaya çalışacağım. Buradaki temel bilgilerden yararlandıktan sonra Kdevelop gibi programların ürettiği veya örütbağdan indirmiş olduğunuz herhangi bir uygulamanın kaynak kodu içerisinde gezinerek farklı kullanımları inceleyebilirsiniz.

3.1. Gerekli Araçlar

GNU Paket Kurgulama Sistemi için gerekli araçlar ve kullanım alanları aşağıdaki gibidir:

1. **autoconf** yapılandırma için kullanılacak `configure` betiğini üretir. Kodun taşınabilir olmasını etkileyecek özellikleri, üzerinde çalıştığı platform için denetler. Elde ettiği değerleri, daha önceden belirtilmiş şablonlara uygun şekilde birleştirerek özelleştirilmiş `Makefile`, başlık dosyaları vb. oluşturur. Bu sayede programı derleyecek kullanıcı tek tek elle bu değişiklikleri yapmak zahmetinden kurtulur.

2. **automake**, **autoconf** için kullanılacak `Makefile` şablonlarını (`Makefile.in`) temel alarak `Makefile.am` dosyalarını üretir. **automake** tarafından üretilen `Makefile` dosyaları GNU makefile standartlarına uygun olup, kullanıcıyı elle `Makefile` dosyası oluşturma zahmetinden kurtarır. **autoconf**'un çalışabilmesi için öncelikle **automake**'in düzgün olarak çalışması gereklidir.

3. **libtool** özellikle paylaşımlı kütüphanelerin taşınabilir bir yapıda oluşturulabilmesi için gereken pek çok ayrıntıyı kullanıcıdan soyutlar. Kullanımı için **autoconf** veya **automake** gerekli değildir, tek başına da kullanılabilir. **automake** ise **libtool**'u destekler ve onunla birlikte çalışabilir.

4. **Autotools** GNU kodlama standartlarına uygun, taşınabilir kod üretmede yardımcı olur.

GNU Paket Kurgulama Sistemi tarafından gerçekleştirilen temel görevler şunlardır:

1. Çok sayıda alt dizin içeren kaynak kodlardan uygulamaları üretebilir. Her bir dizin için ayrıca **make** komutunu çağırmak zahmetinden geliştiriciyi kurtarır. Bu sayede tüm kaynak kodları aynı dizinde bulundurmak yerine aşamaları daha belirgin bir dizin yapısı kullanabilirsiniz.

2. Yapılandırma işlemini otomatik olarak yapar. Kullanıcıların `Makefile` dosyalarını düzenlemelerine gerek kalmaz.

3. `Makefile` dosyalarını otomatik olarak üretir. `Makefile` yazımı büyük projelerde sürekli tekrar gerektirir ve aynı zamanda hata yapmaya elverişli bir yapıdır. GNU Paket Kurgulama Sistemi için sadece `Makefile.am` şablonunun yazımı yeterlidir. Bu sayede hata yapma olasılığı azalır ve yönetimi kolay hale gelir.

4. Hedef platform için özel testler yapabilme imkanı sunar. `Makefile.am` dosyasına eklenecek bir kaç satırla hedef platformda programın derlenebilesi için aranan özelliklerin var olup olmadığı kontrol edilebilir.

5. Paylaşımlı kütüphanelerin oluşturulması statik kütüphanelerin oluşturulması kadar kolay hale gelir.

GNU Paket Kurgulama Sistemi için gerekli olan bu araçların sadece geliştirmenin yapıldığı sistemde kurulu olması yeterlidir. Bu programlar çalıştıktan sonra her platformda çalışabilecek betik programları üretirler. Bu sayede uygulamanızın kaynak kodunu indirip kurmak isteyen biri, **autoconf**, **automake** gibi araçları da sistemine kurmak zorunda kalmaz.

3.2. Basit Bir Autoconf, Automake Örneği

Şimdi aşağıdaki test programımız için GNU Paket Kurgulama Sistemini nasıl kullanacağımızı öğrenelim.

```
#include <stdio.h>

int main()
{
  printf("Çalışıyor\n");
  return 0;
```

Programımızı `test.c` olarak kaydedelim. Şimdi programın derlenmesi işlemlerini **autoconf** ve **automake** ile yapmaya başlayalım. Bunun için öncelikle aşağıdaki `Makefile.am` dosyasını oluşturalım:

```
in_PROGRAMS = test
test_SOURCES = test.c
```

Ardından aşağıdaki gibi bir `configure.in` dosyası oluşturalım:

```
AC_INIT(test.c)
AM_INIT_AUTOMAKE(test,0.1)
AC_PROG_CC
AC_PROG_INSTALL
AC_OUTPUT(Makefile)
```

Dosyaları oluşturup kaydettikten sonra şimdi aşağıdaki komutları çalıştıralım:

```
$ aclocal
$ autoconf
$ ls -l
total 88
-rw-r--r--    1 demirten demirten    16626 Haz 22 17:32 aclocal.m4
-rwxr-xr-x    1 demirten demirten    50233 Haz 22 17:32 configure
-rw-r--r--    1 demirten demirten       90 Haz 22 17:31 configure.in
-rw-r--r--    1 demirten demirten       43 Haz 22 17:29 Makefile.am
-rw-r--r--    1 demirten demirten       71 Haz 22 17:28 test.c
```

Gördüğünüz gibi **aclocal** ve **autoconf** uygulamaları çalıştıktan sonra `aclocal.m4` ve `configure` dosyaları üretildi. Şimdi ise **automake** programını çalıştıracağız:

```
$ automake -a
automake: configure.in: installing `./install-sh'
automake: configure.in: installing `./mkinstalldirs'
automake: configure.in: installing `./missing'
automake: Makefile.am: installing `./INSTALL'
automake: Makefile.am: required file `./NEWS' not found
automake: Makefile.am: required file `./README' not found
automake: Makefile.am: installing `./COPYING'
automake: Makefile.am: required file `./AUTHORS' not found
automake: Makefile.am: required file `./ChangeLog' not found
automake: configure.in: installing `./depcomp'
```

automake ilk çalıştırıldığında öncelikle `install-sh`, `mkinstalldirs` ve `missing` dosyalarının daha önceden oluşturulup oluşturulmadığını kontrol eder. Eğer yoksa bu dosyaları oluşturur (Bendeki sistemde dosyaları oluşturmak yerine `/usr/share/automake` dizini altındaki asıllarına bağ veriyor). Bu dosyalar **automake** tarafından üretilen `Makefile` dosyaları için gereklidir. Ayrıca GNU kodlama standartlarına göre `INSTALL`, `NEWS`, `COPYING`, `README`, `AUTHORS` ve `ChangeLog` dosyalarının da bu dizinde bulunması gereklidir. Bu dosyalar olmadığı için **automake** uyarı vermektedir. **make distcheck** komutunun hata vermemesi için bu dosyaları oluşturalım, sonra içlerini nasıl olsa doldururuz. Bendeki sistemde **automake** ilk çalıştırıldığında `INSTALL` ve `COPYING` dosyalarını da bağ olarak oluşturduğu için önce onları siliyorum:

```
$ rm INSTALL COPYING
$ touch NEWS INSTALL README COPYING AUTHORS ChangeLog
```

automake programını bu dosyalardan varlığından haberdar edelim:

```
$ automake -a
$ ls
aclocal.m4  configure     depcomp     Makefile.am  mkinstalldirs  test.c
AUTHORS     configure.in  install-sh  Makefile.in  NEWS
ChangeLog   COPYING       INSTALL     missing      README
```

Bu dizin yapısı size tanıdık gelmiş olmalı. Kaynak paketimiz bu haliyle artık son kullanıcının karşısına çıkmaya hazır, hemen paketleyip dağıtabiliriz. Şimdi kendimizi bu programi Genel Ağ'dan indirip bilgisayarına kurmak isteyen birinin yerine koyalım. Yapmamız gerekenler aşağıdaki gibidir:

```
$ ./configure
creating cache ./config.cache
checking for a BSD compatible install... /usr/bin/install -c
checking whether build environment is sane... yes
checking for mawk... mawk
checking whether make sets ${MAKE}... yes
checking for gcc... gcc
checking whether the C compiler (gcc  ) works... yes
checking whether the C compiler (gcc  ) is a cross-compiler... no
checking whether we are using GNU C... yes
checking whether gcc accepts -g... yes
checking for style of include used by make... GNU
checking dependency style of gcc... gcc
checking for a BSD compatible install... /usr/bin/install -c
updating cache ./config.cache
creating ./config.status
creating Makefile
$ make
source='test.c' nesnect='test.o' libtool=no \
depfile='.deps/test.Po' tmpdepfile='.deps/test.TPo' \
depmode=gcc /bin/sh ./depcomp \
gcc -DPACKAGE=\"test\" -DVERSION=\"0.1\"  -I. -I. -g -O2 -c `test -f test.c
¬ || echo './'`test.c
gcc  -g -O2   -o test  test.o
$ ./test
Çalışıyor
```

Uyarı

Aşağıda programı nasıl sisteminize kuracağınız anlatılmaktadır. Öntanımlı olarak **test** uygulaması `/usr/local/bin` dizini altına kopyalanır. Ancak bu dizinde test adında bir uygulamanız zaten varsa ve bunu kaybetmek istemiyorsanız veya yanlışlıkla *prefix* vererek uygulamayı varolan `/usr/bin/test` üzerine kopyalamamak için dikkatli olunuz.

Bu noktada dilerseniz uygulamayı sisteme kurabilirsiniz. Bunun için `root` kullanıcı haklarına sahip olmalısınız.

```
$ make install
make[1]: Entering directory `/tmp/test'
/bin/sh ./mkinstalldirs /usr/local/bin
  /usr/bin/install -c test /usr/local/bin/test
make[1]: Nothing to be done for `install-data-am'.
```

```
make[1]: Leaving directory '/tmp/test'
```

Uygulamayı kaldırmak içinse **make uninstall** komutunu kullanabilirsiniz:

```
$ make uninstall
rm -f /usr/local/bin/test
```

Uygulamanızın artık hazır olduğuna inandığınızda **make distcheck** komutu ile onu paket haline getirebilirsiniz (Ekran çıktısı biraz uzun olduğundan burada listelenmemiştir). Bu komut işini tamamladığında bulunduğunuz dizinde `test-0.1.tar.gz` adında bir dosya oluşacaktır. Artık bu dosya ile programınızın dağıtımını yapabilirsiniz.

Şimdi biraz da yaptığımız bu örneği biraz daha açıklayalım. `Makefile.am` içerisinde mantıksal bir dil kullandık. Yazdığımız hiç bir satır çalıştırılmadı. Diğer yandan `configure.in` içerisinde kullandığımız dil prosedüreldir, yazdığımız her satır çalıştırılacak bir komutu göstermektedir. `Makefile.am` dosyası içerisindeki ilk satır programın ismini belirtirken ikinci satır programı oluşturan kaynak kodları belirtmektedir. Şimdi daha karışık olan `configure.in` içerisindeki komutlara sırasıyla bakalım:

- **AC_INIT** komutu `configure` betiği için ilklendirmeleri yapar. Parametre olarak kaynak dosyaların adlarını alır.
- **AM_INIT_AUTOMAKE** komutu, **automake** kullanacağımızı gösterir. Parametre olarak programın ismini ve sürümünü alır. Eğer `Makefile.in` dosyalarını elle hazırlayacak olsaydık bu komutu kullanmamıza da gerek olmayacaktı.
- **AC_PROG_CC** komutu kullanılan C derleyicisinin ne olduğunu belirler.
- **AC_PROG_INSTALL** komutu BSD uyumlu `install` uygulamasına sahip olup olmadığımızı denetler. Eğer yoksa bu işlem için `install-sh`'ı kullanır.
- **AC_OUTPUT** komutu `configure` betik programının `Makefile` dosyalarını `Makefile.in` dosyalarından üretmesi gerektiğini belirtir.

Örneğimizdeki `configure.in` dosyası içerisinde yer almayan ama sıklıkla kullanacağımız bazı komutlar da şunlardır:

- **AC_PROG_RANLIB** komutuyla bir kütüphane geliştiriyorsak **ranlib**'in sistemde nasıl kullanılacağını öğrenebiliriz.
- **AC_PROG_CXX** komutuyla sistemdeki C++ derleyicisinin ne olduğunu öğrenebiliriz.
- **AC_PROG_YACC** ve **AC_PROG_LEX** komutlarıyla kaynak kodlarımız `lex` veya `yacc` dosyaları içeriyorsa bu uygulamaların sistemde varlığını denetleyebiliriz.
- Eğer alt dizinlerde başka `Makefile` dosyalarımız da olacaksa bunu

  ```
  AC_OUTPUT(Makefile          \
            dizin1/Makefile   \
            dizin2/Makefile   \
            )
  ```

  komutlarıyla belirtebiliriz.

Dosyaların içeriğinden bahsettikten sonra şimdi de biraz önce yaptığımız örnekte çalıştırdığımız komutlardan sonra neler olduğuna tekrar bakalım.

- **aclocal** komutu çalıştıktan sonra aclocal.m4 dosyası üretilir. Bu dosya içerisinde **autoconf** tarafından kullanılacak olan makrolar yer almaktadır (kendi özel makrolarımızı nasıl hazırlayacağımıza ileride değinilecektir).
- **autoconf** komutuyla aclocal.m4 ve configure.in dosyaları işlenerek configure betik programı oluşturulur.
- **automake** komutu Makefile.am dosyasını temel alan bir Makefile.in oluşturur. Ayrıca GNU kodlama standartlarına göre eksik olan dosyalar için örnek birer kopya üretir.
- **./configure** komutuyla çalıştırılan betik programı daha önceden belirtilen özellikler için sistemimizi test eder ve Makefile.in dosyasını örnek alarak Makefile dosyalarını oluşturur. **AC_OUTPUT()** ile belirtilen tüm dosyalardaki **@FOO@** şeklindeki kayıtları FOO için elde edilen değerlerle değiştirir (örneğin C derleyicisinin ne olduğu gibi).

3.3. Yapılandırma Başlık Dosyalarının Kullanımı

Çoğu zaman derleme anında bazı makrolar tanımlamak isteriz. **-D** seçeneği ile derleyiciye bildirilen bu değerleri programımız içerisinden kullanarak ilgili kod parçacığının çalışma şeklini değiştirebiliriz. **autoconf** kullandığımız bir uygulama için böylesi seçenekleri kullanmanın yolu yapılandırma başlık dosyası, config.h kullanmaktan geçmektedir.

config.h mantığını kullanabilmemiz için test.c programımızın en başına aşağıdaki üç satırı eklemeliyiz:

```
#ifdef HAVE_CONFIG_H
#include <config.h>
#endif
```

Burada unutulmaması gereken önemli bir nokta, config.h dosyasının mutlaka ilk olarak *include* edilmesidir.

Program kaynak kodunu bu şekilde değiştirdikten sonra configure.in dosyasını da aşağıdaki duruma getirmeliyiz:

```
AC_INIT(test.c)
AM_CONFIG_HEADER(config.h)
AM_INIT_AUTOMAKE(test,0.1)
AC_PROG_CC
AC_PROG_INSTALL
AC_OUTPUT(Makefile)
```

Ve çalıştırdığımız komutlara bir yenisini aşağıdaki sırada ekleyelim:

```
$ aclocal
$ autoconf
$ touch NEWS README AUTHORS ChangeLog
$ autoheader
$ automake -a
```

Burada yaptıklarımızın bir önceki örneğimizden farkı **autoheader** programını da çalıştırmaktan ibarettir. **autoheader**, configure.in dosyasını tarayarak bir config.h.in şablon dosyası oluşturur. Bu dosyanın içerisine configure.in dosyasından öğrendiği, tanımlanabilecek değerleri ve açıklamalarını yerleştirir. Aynı zamanda kaynak kodumuzdan config.h dosyasının *include* edilebilmesi için gerekli -I seçeneklerini de derleyiciye geçirir. **./configure** komutundan sonra ise gerçek config.h dosyası oluşur. Şimdi örneğimizde oluşan config.h dosyasına bir bakalım.

```
/* config.h.  Generated automatically by configure.  */
/* config.h.in. Generated automatically from configure.in by autoheader 2.13. */
```

```
/* Name of package */
#define PACKAGE "test"

/* Version number of package */
#define VERSION "0.1"
```

3.4. Automake ile ilgili ayrıntılar

Projemiz büyüdükçe kaynak kodların bulunduğu yerler gittikçe karışmaya başlar. Bunları düzenleyebilmek amacıyla daha hiyerarşik dizin yapıları kurarız. Ancak bu defa da her bir dizin için uygun `Makefile` dosyalarını üretmemiz gerekecektir. Bunun için bu bölümde `Makefile.am` dosyalarından daha ayrıntılı bir şekilde bahsedeceğiz.

`Makefile.am` dosyalarının genel biçimi `değişken = değer` şeklindedir. Ancak aynı zamanda, geleneksel `Makefile` mantığındaki gibi hedef ve soyut kural tanımlamalarını da destekler. Şimdi `Makefile.am` dosyalarında sıklıkla kullanacağımız satırlara bir bakalım.

`INCLUDES = -I/usr/local/include -I/usr/custom/include ...`
: Nesne kodlarını oluştururken derleyiciye *include* edilen dosyaları hangi dizinlerde araması gerektiğini belirtir. Ayrıca proje kaynak kod yapısı içerisindeki bir dizin seçenek olarak verildi ise alt dizinlerde yer alan kaynak programların hepsinin bu dosyalara erişebilmelerini sağlamak amacıyla tanımlama **`INCLUDES = -I$(top_srcdir)/src/libxxx`** şeklinde yapılmalıdır. Buradaki `$top_srcdir` değişkeni kaynak kod yapısı içerisindeki en üst dizini tutar.

`LDFLAGS = -L/usr/local/lib ...`
: Derleyici çalıştırılabilir dosyaları üretirken ihtiyaç duyduğu kütüphaneleri hangi dizinlerde araması gerektiğini bu tanımla öğrenecektir.

`LDADD = test.o ... $(top_builddir)/lib/libfoo.a ... -lfoo ...`
: Tüm oluşacak çalıştırılabilir dosyalara sembolik bağlamak istediğiniz sisteme kurulu olan ve olmayan nesne dosyaları burada listelenir. Eğer listelenen nesne dosyası sistemde kurulu değilse dosyanı tam adresi verilmelidir (`$top_builddir/lib/libfoo.a` örneğindeki gibi).

`EXTRA_DIST = dosya1 dosya2 ...`
: Kaynak kod paketinizde bulunmasınıistediğiniz her türlü dosyayı burada listeleyebilirsiniz.

`SUBDIRS = dizin1 dizin2 ...`
: Bulunulan dizin için işlem yapmadan önce kuralların çalıştırılması gereken dizinlerdir. **`make`** uygulaması bulunulan dizinde işleme başlamadan önce, burada belirtilen dizinlerdeki `Makefile` kurallarını çalıştırır ve listelenen tüm dizinler için işlemleri bitirdikten sonra bu dizine geri döner.

`bin_PROGRAMS = test test2 ...`
: **`make`** komutu çalıştıktan sonra üretilecek ve **`make install`** komutuyla belirli bir dizin altına kopyalanacak program adları burada listelenir.

`lib_LIBRARIES = libfoo1.a libfoo2.a ...`
: **`make`** komutu çalıştıktan sonra üretilecek ve **`make install`** komutuyla belirli bir dizin altına kopyalanacak kütüphane dosyalarının adları burada listelenir.

`check_PROGRAMS = program1 program2 ...`
: **`make`** komutunun çalışması esnasında üretilmeyip, sadece **`make check`** komutuyla üretilecek, programınızın tümünü veya bir kısmını test edecek uygulamaların çalıştırılabilir dosya adları listelenir.

`TESTS = program1 program2 ...`

make check komutu sonrasında test amaçlı çalıştırılacak dosya adlarını listeler. Çoğu durumda `TEST = $(check_PROGRAMS)` şeklinde bir tanımlama yapabilirsiniz.

include_HEADERS = foo1.h foo2.h ...
`/prefix/include` dizini altına kurulmasını istediğiniz başlık dosyalarını burada listelemelisiniz.

bin_PROGRAMS
Bu değişkende listelediğiniz her bir program için aşağıdaki tanımlamaları da yapmalısınız (*program* kelimesi yerine programın adını yazmalısınız):

*program*_**SOURCES = test.c test1.c test2.c test.h test1.h test2.h ...**
automake programı burada belirtmiş olduğunuz dosya adları için, C, C++ ve Fortran dillerine özgün soyut `Makefile` kurallarını oluşturur. Eğer başka bir dil kullanılıyorsa gerekli kuralları siz vermelisiniz.

*program*_**LDADD = $(top_builddir)/lib/libfoo.a -lnsl ...**
Burada programınızla ilintilenmesi gereken kütüphaneleri listelemelisiniz.

*program*_**LDFLAGS = -L/dizin1 ...**
`program_LDADD` ile belirttiğiniz kütüphanelerin hangi dizinlerde aranması gerektiği burada listelenir.

*program*_**DEPENDENCIES = dep1 dep2 ...**
Programınızın derlenebilmesi için bağımlı olduğu diğer hedefleri burada listelemelisiniz.

4. Yararlı Belgeler

Daha ayrıntılı bilgi almak için aşağıdaki belgeleri inceleyebilirsiniz:

- GNU Make Kılavuzu[B3]
- GNU Autoconf Kılavuzu[B4]
- GNU Automake Kılavuzu[B5]

Notlar

a) Belge içinde dipnotlar ve dış bağlantılar varsa, bunlarla ilgili bilgiler bulundukları sayfanın sonunda dipnot olarak verilmeyip, hepsi toplu olarak burada listelenmiş olacaktır.

b) Konsol görüntüsünü temsil eden sarı zeminli alanlarda metin genişliğine sığmayan satırların sığmayan kısmı ¬ karakteri kullanılarak bir alt satıra indirilmiştir. Sarı zeminli alanlarda ¬ karakteri ile başlayan satırlar bir önceki satırın devamı olarak ele alınmalıdır.

[B3] `http://www.gnu.org/manual/make/html_sect1/make_toc.html`

[B4] `http://www.gnu.org/manual/autoconf/`

[B5] `http://www.gnu.org/manual/automake/`

Bu dosya (makefile–nasil.pdf), belgenin XML biçiminin TEXLive ve belgeler-xsl paketlerindeki araçlar kullanılarak PDF biçimine dönüştürülmesiyle elde edilmiştir.

6 Şubat 2007