Bazı Temeller(tekrar)

- .h dosyaları
- “extern” anahtar kelimesinin anlamı
- “Compiling” ve “linking” kavramkarı
- Bir prosesin belleği nasıl organize edilir(yığın,yığıt,vs)
- Makefiles

Program Nedir ?

Bunun cevabı kişiye bağlıdır. Herkes çok farklı cevaplar verebilir. İşletim sisteminde bir program çalışmaya başladığı zaman biz bu programa “proses” adı veriyoruz.Programın kendisi(yazılma biçimi) birçok formda olabilir.Örneğin,makine kodu,assembly kodu,C,C++ kodu yada Java,Fortran, ...

İşlemciler sadece makine kodlarını işletirler. Makine kodu, ikili(0 ve 1’ler) “opcode” ve “operand” gibi ikili makine kelimeleriyle yazılmış koddur. Opcode’lar farklı talimatları temsil ederler,mesela aritmetik işlemler. Operandlar ise talimatlar tarafından işlenen verilerdir.Bir makine kodu şu şekildedir:

10001000110101001010101101101011

Yukarıdaki kodun ‘**10001000’ kısmı** opcode , ‘110101001010’ ve ‘101101101011’ iki operandıdır.Birçok insan bu formatta kod yazmak istemez. Bu yüzden bilgisayar bilimi öncüleri bizim için bir yol aradı ve tüm bu ikili sistemi temsil eden kısaltmaları tasarlasılar.Ve yukarıda gösterdiğimiz kod bu hali aldı:

```
ADD AX,BX
```

Bu kısaltmalar ve diğer bazı yönlendirme komutları hepsi beraber “assembly dili” olarak adlandırılır.Assembly programı oldukça okunabilir ve anlaşılabilirdir:

```
0000000000000100    MOV BX, VALUE1
0000000000001000    MOV AX, FACTOR
0000000000001100    MUL AX, BX
```

Bu gelişmelerle hayat biraz daha kolaylaştı fakat yeterince değil.Eğer hayatın sadece bir yükümlüğüğü olsaydı,sadece bu komutlarla birçok etkin program yazabilirdiniz.Ama eğer sizin klavyenizden uzakta bir hayat varsa,birçok farklı problemler vardır.Bunlar;

- Assembly dilindeki kısaltmalar makine kodlarıyla bire bir benzeşir ki bu işlemci bağımlıdır.Eğer Intel x86’dan Motorola 68k’ya taşıma yaparsanız,yeni bir program yazmak zorunda kalacaksınız.68k’da çalışan gerçek bir assemly programı:

```
        ORG    $1000
N       EQU    5
```

```
CNT1    DC.B   0
CNT2    DC.B   0
ARRAY   DC.B   2,7,1,6,3

        ORG    $1500
MAIN    LEA    ARRAY,A2
        MOVE.B #N,D1
        CLR    D6
        CLR    D7
        JSR    SORT
        STOP   #$2700

SORT    MOVE.B #0,D6
        MOVE.B #1,D7
LOOP    MOVE.B $0(A2,D6.W),D2
        MOVE.B $0(A2,D7.W),D3
        CMP.B  D2,D3
        BGT    EXCHANGE
        ADD.B #1,D7
        CMP.B #5,D7
        BLT    LOOP
        JMP    CHECK1
EXCHANGE  MOVE.B D2,$0(A2,D7.W)
        MOVE.B D3,(A2,D6.W)
        ADD.B #1,D7
        CMP.B #5,D7
        BLS   LOOP
CHECK1  ADD.B #1,D6
        MOVE.B D6,D7
```

```
CMP.B #4,D6

BLT   LOOP

RTS

END   $1500
```

Eğer yazdığınız C kodlarının assembly şeklini görmek isterseniz , ”cc –S xxx.x” yada “gcc –S xxx.x “ komutlarını kullanmayı deneyin.Bu komutlar .s uzantılı dosya oluşturur,ki bu dosya sizin kodunuzun assembly versiyonunu içerir.Sun ve Linux da bunu deneyerek farklılıklara(assambly dilinin nasıl sisteme bağlı olduğunu) daha yakından görebilirsiniz.

Not: Assembly kodu nekadar makine koduna benzerse benzesin birbirlerinden farklıdır.Assembler ile assemly kodu gerçek makine koduna dönüştürülür.

- Yukarıdaki programda adresler fiziksel adreslerdir,ki bunlar sizin bellek alanlarınıza karşılık gelen bytlerdır.Eğer bu adreslerin sabit kodlanmış adres değerlerine sahip olsaydınız,islemvinizde sadece bir program çalıstırabicektiniz

- Assebly dilinde program yazmak gerçekten çok sıkıcı.Sadece konsola bir karakter yazdırmak kodun 30 satırını alabilir, sistem çağrısı yapmamak kaydıyla.Ek olarak WordPerfect ve Word birbirlerine çok benzer iki programdır fakat biri assembly dilinde yazılmıştır,diğeri ise C dilinde.

- Kodunuz belli bir parçasını tekrar tekrar kullanmak isteyebilirsiniz,…Diğer insanların kodu kullandığını hayaledin.

Tüm bu problemlerin farkında olan birçok önder bilim adamlarının çalışmaları sonucunda, bize “compiler”, “linker” ,”işletim sistemi” ve yüksek seviye programlama dillerini C,C++,Fortran gibi dilleri verdi.Şimdi biz C compiler’ı ve linker’ı üzerinde duralım.

C dili ile programlama oldukça basit hale geldi.Eğer doğru zamanda doğduysanız, 3 aylık bir eğitim sonunda çok da zeki olmadan çok büyük paralar kazanabilirsiniz. Nasıl yaptığımıza bakalım ;

- Herhangi bir editörde programımızı yazalım ve adı bigmoney.c olsun

- “cc bigmoney.c” komutunu çalıştıralım.Bu bize çalıştırılabilir “a.out” dosyasını üretir.

- “a.out” tipindeki komut isteminden sonra “bigmoney” çalışır.

Herşey bukadar kısa ve kolay olsada biz bilgisayar bilimileri arka planda nelerin olup bittiğini bilmeliyiz.Şimdi buna kısaca bir bakalım:

- İlk olarak “compiler” çalışır ve sizin C programınızı ikili makine kodlarına ve genellikle “obje” adı verilen modüllere dönüştürür .Aslında compiler’ın çıkardığı herbir obje’nin içinde platform bağımlı makine kodu modülleri bulunur.Örneğin COFF (Common Object File Format – Yaygın obje dosya formatı).Ancak bir farklılık var,burada kodlanmış fiziksel adres bulunmaz.Buda modulu ana bellek üzerinde yerdeğiştirilebilir yapar.Böylelikle diske bigmoney.o olarak yazılmıştır.

- Ayrı bir “linker” programı çağırılmış obje dosyasını herhangi bir fonksiyon çağrısı yada çözümlenmemiş değişkenlerin varlığını inceler.Örnegin,”printf” fonksiyonunu kullandınız.”Compiler” bu fonksiyonun nasıl çalıştığını bilmez çünkü bu fonksiyon sistem kütüphanesi tarafından sağlanır.Bu fonksiyonun makine kodlarını bizim makine kodlarımıza eklemek “linker” ın görevidir.Linker görevini tamamladıktan sonra çalıştırılabilir dosya elde edersiniz.

- Sonra “a.out” tipinde komut isteminizle, işletim sistemindeki “loader” “a.out” u diskten okur ve ana belleğe yerleştirir. a.out’un her bir satırın fiziksel adresi bu zaman diliminde sağlanır.Sonra işletim sistemi programın giriş noktasını bulur(bigmoney.c’nin içindeki main fonksiyonu) ve işletmeye buradan başlar.

Niçin .h dosyalarına ihtiyaç duyarız ?

Birçok geliştiriciler büyük bir uygulama geliştirirken işletim sisteminde bulunan kütüphaneleri yada başkalarının yazmış olduğu kütüphaneleri kullanırlar. Modüler programlamanın gereği budur. Compiler’lar tür denetimi yapar ve kütüphane çağrılarınızın ve diğer insanların tanımladıkları fonksiyonların doğruluğunu kontrol eder. Çağırdığınız fonksiyonun kaynak kodlarına erişmenin bir önemi yoktur siz fonksiyonu ya çağırırsınız yada çağırmazsınız. Kodunuzun içine fonksiyon tanımlarını koymak iyi bir fikir değildir(neden?). .h dosyalarını kullanmak(genellikle fonksiyon prototipi, veri tipleri ve makrolardan oluşur) sorunu çözer. Bir diğer soru .h dosyası içine fonksiyonların tanımları yapılamaz(neden ?).

Derleme & Bağlama

Compiler her bir kaynak kodu obje dosyası haline dönüştürür. Linker ise üretilen tüm obje dosyalarını, aynı zamanda ilişkili sistem kütüphanelerini alır ve çalıştırılabilir dosya üretir ve bunu disk üzerine depolar. Aslında bu farklı dillerde yazılmış kodlardan üretilen obje modüllerini bağlamayı mümkün kılar.

Bir prosesin belleği nasıl organize ediliyor ?

Bir programın daha yakından incelersek;

İşletim sistemi “a.out” dosyasını diskten ana belleğe yükledikten sonra “a.out” un çalışması için bu segmentler oluşmak zorunda :

Çalıştırılabilir text(executable text):

İkili(binary) gerçek çalıştırılabilir bilgilerden oluşur. Bu ikili disk üzerinden eşlenir ve sadece okunabilir.

Çalıştırılabilir veri(executable data):

Çalıştırılmadan önce ilk değeri atanmış değişkenleri(sabitleri) içerir. İkili disk üzerinden eşlenir ve isteğe göre okunabilir, yazılabilir, private da olabilir.(private eşlemesi değişkenin çalışma esnasında değilmemesini gerektirir)

Heap(yığıt) uzayı:

"Mallock" fonksiyonu tarafından tahsil edilmiş bellek. İsimsiz(anonymous) bellek olarak tanımlanır. Çünkü dosya sisteminde eşlemesi yok.

Stack (yığın):

İsimsiz bellekten tahsil edilmiş bellek.

Tam olarak prosesin bellek adres uzayı düzeni sistemden sisteme değişir. Ancak farklı segmentlerin isimlendirme kuralı mevcuttur,bu kuralı her sistem takip eder(BSD).BSD Unix'e bakalım

BSD segmentleri sanal uzyın bitişik bölgeleri olarak tanımlar.Proseslerin adres alanları 5 temel segmentten oluşur:

- Text segment, çalıştırılabilir kodu tutar.
- İnitialized data segmenti, sıfır olmayan ve çalışma başlamadan önce ataması yapılan değişkenler.
- Bss segmenti başlangıçta sıfır olması gereken değerleri ilklendirir.
- Heap segmenti, ilklendirirlmemiş verileri içerir. Program çalışma zamanında değişken ataması ve bellek ayırması için kullanılır.
- Stack

Stack belleğin yanında birde çekirdeğin stack belleği (sistem çağırısı geldiğinde işletim sisteminin o anki değerlerini tutmak için) ve kullanıcı yapısı(struct)

0 2GB

text	İnit.data	bss	heap	…. ...	stack	Kernel stack,u struct

Bu bir BSD'nin her bir prosesin sanal bellek uzayındaki görüntüsü. Text, İnitialized data, Bss, Heap bitişik olarak bellek alanındadır ve düşük adres değerinden başlarlar. Stack yüksek adres değerinden başlar ancak aşağıya doğru genişler. Gerektiği kadar yığının altı ve üstü arasında boş öbek tahsil edilir

Fonksiyon ve Yığın Segmenti

Fonksiyonlardan bahsediyorsak Lifetime dan sözetmeliyiz. Lifetime bir değişkenin program çalıştığı süre içerisindeki görünür(aspect) olma durumudur. Global bir değişken işletim sistemi programı çalıştırıp kapatana ve hafızadan silene kadar var olmaktadır. Bir fonksiyonun içerisinde tanımlanan bir değişken yalnızca fonksiyon çağırıldığında var olmaktadır. Bir fonksiyon çağrısı olduğunda aşağıdaki olaylar meydana gelmektedir:

1. 1. İşletim sistemi ana programda kullanılmakta olan adresi yığına **ekler(push eder).**
2. 2. CPU registerlerinde bulunan ana programın tüm gerekli verileri geri getirmek (recover etmek) için yığına push eder.
3. 3. Fonksiyon çağırısı içerisindeki tüm argümanlar yığına bir bir push eder.
4. 4. Fonksiyonun text segmentinin başlangıç adresini bulur ve execute etmeye başlar.
5. 5. Fonksiyonun çalışırken, yığında yerel değişkenler için yeterince hafıza alanı ayırılır.
6. 6. Eğer malloc() çağırılmışsa heap den de hafıza ayırılır.
7. 7. Fonksiyonunun işleyişi sona erer.
8. 8. Eğer dinamik hafıza ayırmaları varsa bu nu geri bırakmalıyız(iyi programlama alışkanlığı).
9. 9. Tüm lokal değişkenler yığından pop edilir.
10. 10. Tüm fonksiyon argümanları yığından pop edilir.
11. 11. Stackden orjinal çalışma durumu recover edilir.
12. 12. Stackden alınan adresten tekrar execute e başlanır.

Soru : Bir fonksiyonun geri dönüş değeri ana programa nasıl döner ?

Kapsam ve Görünürlük

Bir tanımlayıcının görünürlüğü programın bir kapsam içerisinde referans edilebildiği kısımlarını belirler.

Bir tanımlayıcı sadece kapsamının sınırladığı alanlarda görünürdür. Bu alan dosyaya, fonksiyona, bloka yada fonkksiyon prototipine göre sınırlandırılmış olabilir. Bir tanımlayıcının kapsamı programda isminin kullanılabildiği yerlerdir. Bu bazen lexical kapsam olarak adlandırılır. Dört tür kapsam vardır : Fonksiyon, dosya, blok ve fonksiyon prototipi.

Tüm tanımlayıcıların harici etiketleri deklarasyonun meydana geldiği seviyede belirlenen kendi kapsamlarına sahiptir.

Her çeşit kapsam için aşağıdaki kurallar bir programın içerisinde tanımlayıcıların görünürlüğünü yönetir.

Dosya Kapsamı

Dosya kapsamı olan bir tanımlayıcı için bildirici yada tür belirteci herhangi bir blok yada parametre listesinde görünür ve deklerasyondan sonra herhangi bir alandan erişilebilirdir. Tanımlayıcı isimleri

dosya kapsamıyla genel olarak global yada external olarak adlandırılır. Bir global tanımlayıcı tanımlandığı yada deklare edildiği noktadan başlar ve çeviri ünitesinin sonunda sonlanır.

Fonksiyon Kapsamı

Bir etiket fonksiyon kapsamı olan bir tanımlayıcının bir türüdür. Etiket bir alan içerisinde kullanımıya dolaylı olarak deklare edilir. Etiket isimleri bir fonksiyon içerisinde benzersiz(uniqe) olmalıdır.

Blok kapsamı

Blok kapsamı içerisinde tanımlanmış bir tanımlayıcı blok içinde görünür yada fonksiyon tanımında formal parametre bildirimleri listesi içinde görünür.Saece kendi bildirimi yada tanımını içerek bloklar arasında görülebilir.Kapsamı kendi bloğu yada iç içe geçmiş ilişkili bloğun sonuna kadardır. Bu tanımlayıcılara "local değişken"ler denilir.

Fonksiyon-prototipi kapsamı

Fonksiyon tanımı yapılırken tanımlanan tanımlayıcılardır ve sadece fonksiyon prototip tanımı yapılırken üretilir ve fonksiyon prototip tanımı bittiği yere kadar geçerlidirler.(fonksiyon tanımı içinde geçerli değil sadece prototip tanımlamada)

Ancak, statik depolama sınıfı belirteci ile dış düzeyinde bildirilen değişkenler ve fonksiyonlar sadece tanımlandığı kaynak dosya içinde görülebilir. Diğer tüm işlevler global görünür.

C'de "extrern" ve "static" anahtar kelimeleri

Tüm dış değişkenler ve fonksiyonlar varsayılan extern tiptedir. Dış değişkenler dosya kapsam değişkenlerdir. Değişken bildirimlerinin dış yüzeydeki (bildirimleri tanımlayan) değişkenlerin tanımları veya başka yerlerde tanımlanan değişkenlerin referansları (referans bildirimleri) vardır.

Harici değişken bildiriminde ayrıca başlangıçta değişkenin (örtülü veya açık) değişken bildirimi tanımlanır. Dış düzeyinde bir tanımı çeşitli biçimler alabilir:

- Statik ile bildirilen bir değişken. Açık statik değişken sabit bir ifadeyle örtülenebilir. Eğer başlangıçta atama yaparsanız değer o olur aksi takdirde değişken 0 varsayılan değerini alır.

```
static int k = 16;
static int k;
```

- Örtülü bir değişken dış düzeyde başlatıldığında. Örneğin, int j=3; j'nin bir değişken tanımıdır.

Değişken bir dış düzeyde tanımlandıktan sonra, bu birim için geri kalanı boyunca görülebilir. Değişken bazı kaynak dosyalarda önceki bildirimlerde görünür değildir. Bu yüzden programdaki diğer kaynak dosyalarda görünür değildir, ayrıca aşağıda açıklandığı gibi bir referans bildirimi görünür yapabilir.

***static** declarator (statik bildirici)*:

Değişken değiştirirken, static anahtar sözcüğü değişken statik süresi (program başlatılıp tahsis edilip program bitirilip ayrılana kadar) vardır ve başka bir değer belirtilmemişse 0 olarak başlatılır.

Dosya kapsamında bir değişken veya fonksiyon değiştirirken static anahtar sözcüğü değişken veya fonksiyonun (bildirildiğinden dosya dışına kadar görünür değil) iç bağlantı olduğunu belirtir.

```
// static anahtar kelimesine örnek
static int i; // Değişken sadece bu dosyadan erişilebilir
static void func(); // Fonksiyon sadece bu dosyadan erişilebilir
int max_so_far( int curr )
{
static int biggest; // Değişken değeri korunur.
// Her bir fonksiyon çağrısı arasında
if( curr > biggest )
biggest = curr;
return biggest;
        }
```

Bir program içinde sadece bir kez dış seviyesinde bir değişken tanımlayabilirsiniz. Farklı bir çeviri biriminde aynı adı taşıyan başka bir statik değişkeni tanımlayabilirsiniz, ancak, hiçbir çakışma oluşmuyorsa.

Bir işlev statik bildirirseniz, adını bu ilan edildiği dosya dışında görünmez. Static olarak tanımlanan bir fonksiyon, sadece tanımlandığı kaynak dosya içinde görünür. Fonksiyonların aynı kaynak dosyasında statik fonksiyonu çağırabilirsiniz, ama diğer kaynak dosyalarında fonksiyonlara adıyla doğrudan erişilemez. Çatışma olmuyorsa farklı bir kaynak dosyasıyla aynı ada sahip başka bir statik fonksiyon bildirebilirsiniz.

***extern** declarator (extern bildirici)*:

extern anahtar kelime bildirimlerinde bir değişkenin veya fonksiyon ve türlerinin dış bağlantıları (bildirildiği ve bulunduğu dosyalarda ismi görünür) vardır. Değişken değiştirilirken extern değişkenlerinin türlerinin statik sürekliliği vardır(Program başlatılıp tahsis edildikten, ayrılıp program sonlandırılıncaya kadar).Değişken ya da fonksiyon farklı bir kaynak dosya da yada aynı dosya da farklı zamanlarda bildirilmiş olabilir.

extern anahtar kelime bildirimleri bir değişkenin referans edildiği herhangi bir yerin bildirilmesidir. Farklı kaynak dosya görünürlüğünün bildirimini yaparken ya da aynı kaynak dosyada çakıştığında görünürlüğünü yaparken extern bildirimini kullanabiliriz. Önce dış yüzeyde değişkenin referansı bildirilmeli, değişken beyan edilen referansla oluşan çeviri birimi kalanı boyunca görülebilir.

Bir dış başvuru geçerli olabilmesi için, değişken dış düzeyinde, sadece bir kez tanımlanmalı ve bir kez olmalıdır. Bu tanım (extern depolama sınıfı olmadan) programı oluşturan çeviri birimleri herhangi birinde olabilir.

Fonksiyonlar programın tüm kaynak dosyaları (daha sonra statik gibi bir işlevi redeclare sürece) çerçevesinde görülebilir. Herhangi bir fonksiyon extern işlevini çağırabilir.Fonksiyon bildirimleri varsayılan extern olarak depolu-sınıf türlerini kapsamaz.

Örnek

Dış bildirimlerin örneklendirilmesi örneği

```
/*****************************************************************
KAYNAK DOSYA BİR
*****************************************************************/
extern int i; /* Reference to i, defined below */
void next( void ); /* Function prototype */
```

```
void main()
{
i++;
printf( "%d\n", i ); /* i equals 4 */
next();
}
int i = 3; /* Definition of i */
void next( void )
{
i++;
printf( "%d\n", i ); /* i equals 5 */
other();
}
/*************************************************************************
KAYNAK DOSYA İKİ
*************************************************************************/
extern int i; /* Reference to i in */
/* first source file */
void other( void )
{
i++;
printf( "%d\n", i ); /* i equals 6 */
        }
```

Bu örnekte iki kaynak dosyada i toplam üç dış bildiri içerir. Sadece bir bildirimi bir tanımlama beyanıdır.Bildiri;

```
int i = 3;
```

Global değişken i tanımlanır ve başlangıç değeri 3 ile başlatır. Referans extern i kullanarak ilk kaynak dosyasının en üstünde ilanı öncesinde dosyasında kendi tanımlayan beyan görünür global değişken yapar. İkinci kaynak dosyasında i referans bildirisi, ayrıca, kaynak dosyasındaki değişkeni görünür kılar. Eğer bildiri örneği çeviri birimi için görünürlük sağlamıyorsa, derleyici bunun var olduğunu sayar;

```
extern int x;
```

referans bildirimi ve belirleyici bir referans

```
int x = 0;
```

programda baka bir referans bildirimi görünür.

Her üç fonksiyon ana, gelecek ve diğer aynı görevi gerçekleştirir: i'yi arttır ve yazdır. Değerleri 4,5 ve 6 olarak yazıldı. i değişkeni başlatılmamış olsaydı, bunu otomatik olarak 0'a ayarlanmış olurdu. Bu durumda, değerleri 1, 2, ve 3 yazılı olurdu.

Lifetime(Ömür)

Bir değişkenin ya da fonksiyonun çalışmaya başlamasından çıkılıncaya kadar geçen süredir|lifetime.Statik süresini (global lifetime) veya otomatik süresini (yerel lifetime) olduğunu tanımlayıcının depolama süresi belirler.

Statik depolu-sınıf-türleriyle bir değişken bildiriminde statik depolama bildirimi vardır. Statik saklama süresince (aynı zamanda sözde-global) tanımlayıcıların depolama ve bir program süresince tanımlı bir değeri vardır. Depolama ayrılmıştır ve tanımlayıcının depolanan değeri programı devreye almadan önce, sadece bir kez başlatılır. İç veya dış bağlantı ile bildirilen bir kimliği de statik saklama süresi vardır.

Statik hafıza sınıf belirteci bir işlev veya blok içinde bildirilirse, otomatik depolama süresine sahip olmayan bir tanımlayıcı ilan edilir. Otomatik depolama süresi ile bir tanımlayıcı (yerel tanımlayıcı) depolama ve sadece tanımlayıcı olarak tanımlanan veya bildirilmiş bloğu içinde tanımlanmış bir değeri vardır. Otomatik değişken yeni depolandığı değer programın bloğuna girdiğinde tahsis edilir ve program bloğundan çıktığında depoladığı değer serbest bırakılır. Tanımlayıcılar bağlantı yokken otomatik depolama süresine sahip olan bir fonksiyon ilan etti.

Aşağıdaki kurallar bir tanımlayıcı global (statik) veya yerel (otomatik) ömrünün olduğunu belirtir:

Global lifetime: Tüm fonksiyonlar genel kullanım ömrüne sahiptir. Dolayısıyla programın çalışması sırasında her zaman var. Tanıtıcılar dış düzeyi (bu işlev tanımları aynı düzeyde programındaki tüm blokları dışında, böyle) her zaman global (statik) bir ömrü olduğu bildirilmiştir.

Local lifetime: Yerel değişkenin bir başlangıcı varsa, değişken (o statik olarak ilan edilmediği sürece) oluşturulduğu her zaman başlatılır. Fonksiyon parametreleri de yerel ömrüne sahiptir. Onun bildiriminde statik depolama sınıfı belirteci dahil ederek bir blok içinde bir tanımlayıcı için global lifetime belirtebilirsiniz. Sonra girdiği bloklarda ilk bildirilen statik, değerini korur.

Global lifetime bir tanımlayıcının kaynak programından (örneğin, harici olarak ilan değişken veya statik anahtar sözcüğü ile bildirilmiş bir yerel değişken) icrası boyunca var olsa da, bu programın her yerinde görünür olmayabilir.

Pointerlar

Pointer aslında bellekte bir adres temsil eden bir tam sayıdır.Diğer yandan çok sayıda programlama dillerinde pointer kavramı kullanılmaz(gerek duyulmaz)Ancak,esnekliği olan C'nin en büyük avantajlarından biri pointer kullanımıdır.Dizi kavramı ise C dilinde olduğu gibi birçok programlama dilinde olan bir kavramdır.

Şu olay gerçekten çok ilginc :

```
int myarray[30];
```

Değişken myarray'ın gerçek tipi (*int)'dir.C dizi elemanlarına pointerlar kullanılarak erişilir:

myarray[5] dizisi *(myarray+5) olarak çevirilebilir.

Dinamik bir fonksiyon içinde tahsis edilen diziler, bu fonksiyonun dışarı aktarılmasına izin verir.

Ayrıca,biz işaretçileri,işaretçi işaretçileri olarak tanımlayabiliriz, çünkü bir işaretçi sadece bir sayının adresini temsil eder.Örneğimize bir göz atalım.

```
int myarray[30];
    main()
        {
          int * myptr;
          int ** mydblptr;
          myptr = myarray;
```

```
                                        mydblptr = &myarray;
                                }
```

myarray int türünde pointer bir değişkendir.Veri segmenti,yani bss bölümünden 30 tamsayı tahsis edilmiştir.Bss içinde 120 byte yani 30 tamsayı koyma yeri vardır ve myarray bu 120 byte'lık başlık adresine eşittir. myarray =0000F0CF oduğunu düşünelim.Şimdi yukarıdaki koda bakarak myptr ve mydblptr değerlerini söyleyin?

Hepimiz yukarıdakileri anlamak kaydıyla bunu deneyelim :

```
void f1(int * ptr, int len)
{
int i;
ptr = (int*) malloc (sizeof(int)*len);
for (i = 0; i < len; i ++)
ptr[i] = i;
}
main()
{
int i, * array;
f1(array, 5);
for (i = 0; i < 5; i ++)
printf("got value %d\n", array[i]);
return;
}
```

Bu kodu çalıştırdığınızda 'segment fault ' hatası alamadığınız için çok şanslısınız.Neden Segmentation fault nedir Bus error nedir bunu bilen biriyle tartısabiliriniz.

Segment fault hatasından bahsedersek, sayfa hatası anımsatması olabilir.Aslında, sayfa hatasını açıklamak çok kolaydır.Bir işlemde OS sanal bellek adres alanında bir sayfaya erişmeye çalıştığını anladığında bir sayfa hatası üretilir, fakat o sayfanın fiziksel belleği değildir.Sonuç olarak , talep edilen sayfa içinde okunana kadar OS bu süreci durdurur.
Sayfa hatası, çoğu durumda bir hata değildir.

Segment fault çoğu zaman hatadır,tersini açıklamak çok daha karmaşık ve genellikle işlemci bağımlı.İşte pentiumlar üzerindeki olası nedenlerin kısa bir listesi :

-- CS, DS, ES, FS, veya GS ile segment sınırını aşan
--Bir tanımlayıcı tabloya başvururken segment sınırını aşan
--Segment kontrol aktarma sırasında yürütülebilir değil
--Salt okunur veri segmenti veya bir kod kesimi yazılıyor
--Okuma işlemi bir tek segmenti yürütebilir
--Salt okunur bir tanımlayıcı ile SS yükleniyor
--Bir sistem segmenti bir tanımlayıcı ile SS, DS, ES, FS veya GS yükleniyor
--Okunabilir değil yürütülebilir segment tanımlayıcısı olan DS,ES,FS veya GS yükleniyor
--Yürütülebilir bir segment tanımlayıcısı olan SS yükleniyor
--Segment kayıt NULL seçici içerdiğinde DS, ES, FS veya GS belleğe erişir

Bus error donanım içinde değişik yerlere adım adım yerleşir. Tüm bellek R / W adres yolu üzerinde gerçek gerilim sinyallerine çevrilmesi için bir donanım adresi dekoder geçmesi gerekir. Bellek ve her I / O cihaz aynı küresel adres alanına eşleştirilir. Genel adres alanlarında bazı bölümler ayrılmış,

bazıları korunan ve bazısı boş, bazıları açıktır. Eğer donanım kod çözücüsü verdiği gösterici ile bir yasal sinyal dönüştürme yapamıyorsa, o zaman geriye bir donanım kesmesi atar. -"Bus Error" .

MAKE

Bu konu ile ilgili biraz bilgi var . GNU on-line da fazlasıyla dersler var.Ayrıntılar için bakınız. GNU Make Manual yapılan bir kaynak.

Dosyaların çok sayıda proje yönetimi her zaman zahmetlidir. Komut satırında derleyici ve bağlayıcı komutları yazarak hata budama olur. Make bir yardımcı program yönetim sürecinde yardımcı olmak içindir,büyük projelerde güncellemesini sağlar.Bu tür ihtiyaçların belirtilen kurallar kümesine sahip olunmasını otomatik işleme ile elde edilebilir.Bu kurallar genellikle bir makefile da belirtilir.Varsayılan olarak ne zaman makefile yapılır burada sırayla şu isimler çalışır: makefile ve Makefile. Normalde makefile'ını makefile yada Makefile şeklinde çağırabilirsin.Eğer make bu isimlerin hiçbirini bulamazsa,herhangi bir makefile kullanmaz.Eğer makefile için standart bir isim kullanmak istiyorsanız,dosya seçeneğini -f yada --file olarak belirtebilirsiniz. '-f name' veya '--file name' argümanları 'make' e ; 'makefile ' anlamına gelen 'name ' dosyasını okumayı sağlar.Eğer sen '-f' veya '--file ' seçeneği kullanıyorsan,birkaç tane 'makefile' tanımlayabilirsin.Bütün 'makefile' lar düzenlenmiş sırada etkin bir şekilde sıralanabilir. '-f' veya '--file' belirlediysen,varsayılan 'makefile' adlarından 'makefile' ve 'Makefile' otomatik olarak seçilmez.

Bir Kural Neye Benzer(What does a Rule Look Like)

Basit bir makefile aşağıdaki kalıpları içeren kurallardan(rules) oluşur:

```
hedef ... :bağımlılıkları ...
komut
...
...
```

Bir Hedef, genellikle bir program tarafından oluşturulan bir dosya ismidir;Örneğin hedefleri yürütülebilir veya nesne dosyaları olabilir.Bir hedef gerçekleştirilecek bir hareketin ismi de olabilir,'clean' gibi.Dependency,oluşturulan hedefe veri girilen bir klasördür. Bir hedef,genelikle birkaç dosyaya bağlıdır.Command,eylemi gerçekleştirmek için yapılır.Bir kuralın, herbiri kendi satırında, birden fazla komut olabilir.

Lütfen dikkat: Her komut satırının başında bir sekme karakteri koymak gerekir!Dikkatsiz yakalandığımızda bir bilinmezlik olur.

Genellikle bir komut bağımlılıkları olan bir kural olduğunu ve bağımlılıklarda değişiklik varsa bir hedef dosyası oluşturmak için hizmet vermektedir.Ancak,hedef için komutları belirtir bağımlılıkları bulunması gerekmemektedir.Örneğin,hedef 'clean' ile ilişkili bir komut silme içeren bir kural bağımlılıklara sahip değildir. 'rule',daha sonra , belirli bir kural hedefi olan bazı dosyaları nasıl ve ne zaman yapacağını açıklar.'make' hedef oluşturmak veya güncellemek için bağımlı komutları yürütür.Bir kural ne zaman ve nasıl bir eylem gerçekleştireceğini açıklar.'makefile' kuralları dışında başka bir metin içerebilir,fakat basit bir makefile sadece kuralları içeren ihtiyaçtır.Kurallar bu şablondan daha karmaşık görülebilir,fakat az veya çok bütün kalıplar sığacak.

Burada sekiz nesne dosyaları üzerinde 'edit' adında çalıştırılabilir dosyaya bağlı basit bit makefile var,sırayla,sekiz C kaynak ve üç başlık dosyasına bağlıdır.Bu örnekte, bütün 'defs.h' dosyalarını içermektedir,ancak sadece düzenleme komutlarını tanımlayan 'command.h' içerir ve sadece

düzenleyici tampon değiştirmek için düşük düzeydeki dosyaları 'buffer.h' içerir.

```
edit : main.o kbd.o command.o display.o \
        insert.o search.o files.o utils.o
       cc -o edit main.o kbd.o command.o display.o \
       insert.o search.o files.o utils.o

main.o : main.c defs.h
       cc -c main.c
       kbd.o : kbd.c defs.h command.h
       cc -c kbd.c

command.o : command.c defs.h command.h
       cc -c command.c

display.o : display.c defs.h buffer.h
       cc -c display.c

insert.o : insert.c defs.h buffer.h
      cc -c insert.c

search.o : search.c defs.h buffer.h
      cc -c search.c

files.o : files.c defs.h buffer.h command.h
      cc -c files.c

utils.o : utils.c defs.h
      cc -c utils.c

clean :  rm edit main.o kbd.o command.o display.o \
          insert.o search.o files.o utils.o
```

Biz ters slash kullanarak her uzun satır bölmede iki satır kullanılıyor;uzun bir hat kullanılıyor,fakat okunması daha kolaydır.

'edit ' adında bir makefile dosyası oluşturmamız için ;

make

Dizinde çalıştırılabilir ve tüm nesne dosyalarını silmek için kullanılan makefile dosyası,

make clean

Örneğin makefile,edit dosyası dahil hedefleri çalıştırabilir ve nesne dosyaları main.o ve kbd.o 'dur.'main.c' ve 'defs.h' gibi dependencies dosyaları,aslında her .o dosyası bir hedef ve bağımlılıktır.'cc -c main c' ve 'cc -c kbd c' komutları da dahildir.

Bir hedef dosya oluştuğunda çekirdekler ve bağımlı değişim varsa relinked gerekir.Buna ek olarak,kendi otomatik ürettiği bağımlılık için güncellemesi gerekir.Örneğin;edit sekiz nesne dosyalarına bağlıdır;nesne dosyası main.o kaynak dosya main.c ve başlık dosyası defs.h ' a bağlıdır.

Bir kabuk komutunu,bir hedef ve bağımlılık içeren her bir çizgi izlemektedir.Bu kabuk komutları hedef dosyasını nasıl güncellenmesini söylüyor.Bir sekme karakteri makefile,diğer çizgilerden komutları ayırmak için her komut satırının başında gelmelidir.(Make komutları çalışmaları hakkında

hiç bir şey bilmiyor unutmayın.Doğru hedef dosyalarını güncelleyecek komutlar size kalmıştır.Hedef dosyası güncellenmesi gerektiği zaman bütün belittiğiniz make kural komutlarını çalıştırmanız gerekli.)

Hedef clean sadece bir eylemin adıdır, bir dosya değil.Normalde bu hedef dosyalarını istemediğimden,clean diğer kurallar için bağımlılık değil.Sonuç olarak,make olmadığı takdirde hiçbir şey yapılmaz.Unutmayın bu kural bir bağımlılık değil,bu kuralın tek amacı komutları çalıştırmaktır.Hedefler dosyalar ile ilgili değildir fakat sadece eylemleri sahte eylemlerdir denilebilir.

BİR MAKE FİLE İŞLEMİ NASIL YAPILIR ?

Varsayılan olarak make, ilk kural ile başlar(isim hedefi ile başlayan kuralları saymassak). Bu varsayılan hedef çağrılır.(Hedeflerin make hedefi olarak güncellenmesi içinçaba sarfedilmelidir.) Önceki bölümdeki basit örnekte varsayılan hedefi çalıştırılabilir programa güncellemek için öncelikle edit kuralını koyduk. Böylece make'e ne zaman komut verirsek, make makefile'ın geçerli dizinini okur ve ilk kural işletilmeye başlar. Örneğin, edit kuralına yeniden bağlanmak için önce make bu kuralı tam olarak işleyebilmelidir. Edit'le bağlantılı dosyalar için işlem kurallarına uyulmalıdır. Bu durumda hangi dosyaların nesne dosyası olduğu, dosyaların herbiri kendi kuralına göre işlenir. Bu kurallar her güncelleme için .o ile kaynak dosyasını derler. Eğer kaynak dosyası derlenmeksizin yapılması gerekiyorsa veya birkaç başlık dosyası bağlı olduğu proses ile isimlendirilirse, nesne dosyası daha yeniyse yada nesne dosyası yoksa, diğer kurallar işletilir. Çünkü, hedeflerine bağımlı oldukları hedef gibi görünürler. Bazı kurallar hedefe bağlanmassa(yada bazıları bağlanırsa…vs) bu kurallar işletilmez, make'e ne yapacağını make'e anlatmadıkça (make clean gibi bir komut ile) yeniden derlemeden önce bir nesne dosyası, make gördüğünde kaynak ve başlık dosyalarını günceller. Bu makefile .c ve.h dosyaları için yapılacak bir şey belirtmez .c ve .h dosyaları herhangi bir kuralın hedefi değildir. Make bu dosyalar için bir şey yapmaz fakat, make'i otomatik olarak üretilen c programları güncelleştirir. Şu anda kendi kurallarına göre Bisen veya Yacc tarafından yapılmış gibiyeniden derlendikten sonra nesne dosyaları hangisine ihtiyaç duyarsa (make: kararvermek, edit:yeniden bağlanmak) dosya düzenleme ve çıkış yapılmayacaksa yada nesne dosyası kuıllanılmayacaksa bu yapılmalıdır.

Eğer nesne dosyası sadece yeniden derlendiyse, edit daha yeni olur. Edit yeni bağlanmıştır. Eğer eklenen dosyayı değiştirirsek .c ile make'i çalıştırırız make.o ekleyerek günceller ve derleyerek çalıştırır ve edit bağlantısı oluşur. Eğer command.h ile dosyayı değiştirip make'i çalıştırırsak make, kbd.o, command.o ve files.o nesne dosyalarıyla yeniden derlenerek edit dosyasına bağlanır.

Makefile' ları Değişkenlerle Kolaylaştırmak

Örneğimizde; bütün nesne dosyalarını düzenlemek için gerekli kuralları inceleyeceğiz:

```
edit : main.o kbd.o command.o display.o \
        insert.o search.o files.o utils.o
    cc -o edit main.o kbd.o command.o display.o \
        insert.o search.o files.o utils.o
```

Bu şekilde tanımlama yapmak hataya çok meyillidir;Eğer sisteme yeni bir obje dosyası eklyecek olursak,bunu herbirine eklememiz gerekecek ve biz belki birkaçına eklemeyi unutabiliriz. Bu riskleri

ortadan kaldırmak ve işimizi kolaylaştırmak için değişkenleri kullanırız .Değişkenleri bir yerde metin dizesi olarak tanımlayarak daha sonra istediğimiz yerde tanımlayabiliriz.

Örneğin;

```
objects = main.o kbd.o command.o display.o \
        insert.o search.o files.o utils.o
```

Şeklinde objects değişken nesnesini oluşturarak kullanmak istediğimiz farklı yerlerden ($objects) yazdığımızda ;bu değişken nesnesinin değerinin yerini alır.Aşağıda nesne dosyaları için değişken kullandığımızda, basit bir makefile ın nasıl olduğuna bakalım;

```
objects = main.o kbd.o command.o display.o \
    insert.o search.o files.o utils.o

edit : $(objects)
    cc -o edit $(objects)
main.o : main.c defs.h
    cc -c main.c
kbd.o : kbd.c defs.h command.h
    cc -c kbd.c
command.o : command.c defs.h command.h
    cc -c command.c
display.o : display.c defs.h buffer.h
    cc -c display.c
insert.o : insert.c defs.h buffer.h
    cc -c insert.c
search.o : search.c defs.h buffer.h
    cc -c search.c
files.o : files.c defs.h buffer.h command.h
    cc -c files.c
utils.o : utils.c defs.h
    cc -c utils.c
clean :
```

```
	rm edit $(objects)
```

İcra Edilebilir Komutların Anlaşılması

C kaynak dosyalarını derlemek için komutları telaffuz etmeye gerek yoktur.cc-c komutu .c yerine .o ile güncellenebilir. Örneğin cc –c main.c –o main.o komutu main.c nin içindeki main.o şeklinde derlenebilir. Bu yüzden nesne dosyaları için komut ve kuralları göz ardı ederiz.Bir c dosyası bu şekilde otomatik olarak kullanıldığında, aynı zamanda bağlantıların listesi eklenir.Bu yüzden c dosyalarının kurallarını ihmal edebiliriz. Buradaki örnek açıklandığı gibi değişken nesneleriyle icra edilebilir komutların ikisini anlatmaktadır:

```
objects = main.o kbd.o command.o display.o \
	insert.o search.o files.o utils.o

edit : $(objects)
	cc -o edit $(objects)

main.o : defs.h

kbd.o : defs.h command.h

command.o : defs.h command.h

display.o : defs.h buffer.h

insert.o : defs.h buffer.h

search.o : defs.h buffer.h

files.o : defs.h buffer.h command.h

utils.o : defs.h


.PHONY : clean

clean :
	-rm edit $(objects)
```

Makefile nesneleri sadece örüntülü kurallarla oluşturulduğunda, alternatif bir makefile stili mümkündür. Bu stildeki makefile tarafından grup girişleri hedeflerin yerine geçer. Burada görüldüğü gibi;

```
objects = main.o kbd.o command.o display.o \
	insert.o search.o files.o utils.o

edit : $(objects)
```

cc -o edit $(objects)

$(objects) : defs.h

kbd.o command.o files.o : command.h

display.o insert.o search.o files.o : buffer.h

Burada defs.h bütün dosyaların örüntüsü olarak verilmiştir;command.h ve buffer.h belirli nesne dosylarının örüntüleri için listelenir,tüm bilgileri daha açıktır fakat bazı insanlar bundan hoşlanmaz çünkü hedef ile ilgili bilgileri net bir yerde bulmak isterler.

Dizin Temizlik Kuralları

Bir program derlemek kuralları yazmak için isteyeceğiniz tek şey değildir.Makefile lar yaygın bir program derlemenin yanı sıra bütün nesne dosyalarının nasıl silinebileceğini ve icra edilebilir dizinlerin silinebileceğini söyler.

Burada örnek dizin temizlemenin nasıl olacağına bakacağız;

Clean;

Rm edit $(objects)

Uygulamada beklenmedik durumlarla başa çıkabilmek için kuralı biraz daha karmaşık yazmak istersek;

PHONY:clean

Clean:

Rm edit $(objects)

Bu rm hatalarına rağmen devam edebilmek için gerçek bir dosyayla karıştırılmasını önler.Varsayılan olarak bunu çalıştırmak istemiyoruz.Çünkü bu kural makefile nin başına konulmamalıdır.Böylece örnek makefile düzenleme,kural ve varsayım hedefin bulunmasını istiyoruz.Temizleyerek düzenleme bir bağımlılık olmadığı için bu komutun elemanları olmadan çalıştırırsak bu kurallar çalışmaz.Kuralın çalışması için temiz halini yazmalıyız.

Bağımsız Değişken Hedefleri Belirtme

Hedefleri güncellemek için çaba sarfedilmelidir.Onlar güncellendiğinde bağımlı oldukları diğer hedeflerde güncellenir, yada güncel görünür.Varsayılan olarak ilk makefile hedef makefile dır.Bu nedenle makefile lara genellikle ilk hedef de tüm programların derlenmesi sağlanmalıdır.Eğer makefile nin ilk kuralında birkaç hedef varsa tüm liste değil sadece ilk hedef gelir.Bağmız değişkenler ile farklı bir hedef yada hedefler belirliyebiliriz.Bağımsız değişken olarak hedef adı kullanılır.Eğer birkaç hedef belirtirsek, işlemlerin herbirini isim sırasıyla yapmalıyız.Makefile da herhangi bir hedef amaç olarak belirtile bilir.Hatta makefile değil hedef belirtilebilir,örüntü kurallarına nasıl ulaşabiliriz.Eğer programın bir kısmını derleyecek olursak yada birkaç programdan birini

derleyebiliriz.Hedef olarak yeniden yapmak istediğimiz her dosyayı biçok program içeren dizinden örneğin makefile ile başlangıç hedefini belirtin.

PHONY:all

All: size nm /d ar as

Programın boyutunda çalışıyorsanız,sadece dosyaların yeniden derlenmesini sağlayarak boyutu ayarlamak isteyebilirsiniz.

Normalde yapılmayan makefile ları başka bir kullanımda hedef haline getirebilirsiniz.Örneğin derleme çıktısı bir dosya olabilir yada test için derlenmiş bir program sürümü varsayılan hedefe bir kuralla bağlı değildir.

Başka bir kullanımda sahte yada boş bir hedefteki komutları çalıştırmak için bir çok makefile sahete bir hedefi temizlemek için kaynak dosyaları dışındaki herşeyi sileriz.Siz istediğinizde bu temizliği açıkça yapabilirsiniz.Aşağıda tipik sahte ve boş hedef isim listesi verilmiştir.Örnek olarak gnu nun yazılım paketlerini kullanan hedef isimler verilmiştir.

all:Makefilede bilinen tüm üst düzey hedeler.

Clean: Çalışırken oluşturulan tüm dosyaları siler.

İnstall: Genelllikle kullanıcıların aradığı komutların ve ek dosyaların kopyasını çalıştırabilir di,zinlere kopyalayabilir.

Tar: Kaynak dosyalar için bir tar dosyası oluşturur.Kaynak dosyalar için bir dağıtım dosyası oluşturur.Bu bir tar dosyası,shar dosyası yada sıkıştılmış bir versiyonu olabilir.

Dist:Kaynak dosyalar için bir dağıtım(distribitution) dosyası oluşturur.Bu bir tar dosyası , shar dosyası yada sıkıştırılmış bir versiyonu olabilir.

MODEL KURALLARINA GİRİŞ;

Bir desen hedefin içinde %(tam olarak bunlardan) karakteri içerir, ayrıca tam olarak sıradan bir kural gibi görünür. Hedef dosya adları ile eşleşen bir desen % herhangi bir boş olmayan alt dizin ile eşleşir. Diğer karakterler sadece kendilerini işlem sırasında eşlerler. Örneğin, %c ile biten dosya adı .c.s%.c ile eşlenir. S. Deseni herhangi bir dosya adı ile başlar. .c ile sonlanır ve 5 karakter uzunluğundadır..(bir karakter % olmak zorunda) kök dizinler sistem tarafından % ile çağrılır. % desen kurallarında hedefte aynı kök ile örtüşmektedir. Desen kuralı uygulamak için, hedef desen incelenen dosya adı ile eşleşmeli ve bağımlı desenler mevcut dosyaları adlandırabilmeli yada yapabilmelidir. Hedeften bağımlı dosyalar geldiğinde, form kuralı; %.o.%.c; command…

n.o, bir dosyayı nasıl belirler; farklı bir dosya n.c ile bağımlılığı, n.c mevcut olursa yapılabilir. Belki bazı bağımlılıklarda % kullanılmaz. Böyle bir bağımlılık desen kuralının her dosyaya verdiğini yapar. Bu değişmeyen bağımlılıklar kullanıldığında yararlıdır. Bir desen kuralının herhangi bir % bağımlılık kuralına ihtiyacı yoktur. Aslında herhangi bir bağımlılıkta hepsi bulunur. Bu kural böyle genel olarak etkili bir jokerdir. Bu hedef desen herhangi bir dosya yapmak için bir yol sağlar. Desen kurallarında

birden fazla hedef olabilir. Pek çok farklı kural bağımlı oldukları komutlar ile aynı hareket etmediğinden, normal kurallar beğenilmez.

Eğer bir desen kuralı çoklu hedefler içeriyorsa, make, kural komutlarını bilmelidir. Bu komutlar sadece bütün hedefler yapıldığında icra edilebilir. Hedef ile eşleştirmek için bir desen kuralı arandığı zaman, hedef desenlerin kuralı eşleştirilen diğer hedefler için önemsizdir. Make, özel dosyaların hedef ve bağımlılıklarını verme konusunda endişelenir. Ancak bu dosyaların komutları çalıştırıldığında, diğer hedefleri kendine işaretli şekilde güncellenir.

Bir desen kuralının derleyicisine a.c dosyasına a.o yazdığımızı varsayarsak; doğru kaynak dosyası adında cc komutunu nasıl yazarız? Bu komut adını yazamayız çünkü, örtülü kurallar uygulanır yani adı her zaman farklıdır. Otomatik değişkenler için make'te ne özelliği kullanırız? Bu değişkenler yürütülür ve her kural için baştan değerleri hesaplanır, hedef ve bağımlılık kurallarına göre çalıştırılır. Örneğin, nesne dosyası adı için$@ ve kaynak dosyası adı için $< kullanırsak otomatik değer tablosu;

$@:

Dosya adı ve hedef kuralları arasındadır. Eğer hedef bir arşiv üyesi ise, arşiv dosyasının adı $@ dir. Çoklu hedeflerde, desen kuralında $@ hedef kural komutlarını hangisinin çalıştırdığını belirler.

$%:

Hedef bir arşiv üyesi olduğunda hedef adıdır.örneğin foo.a(bar.o)'dan sonra $%, bar.o ve $@ foo.a.$% hedefi arşiv üyesi değilse boştur.

$<:

İlk bağımlılığın dıdır. Hedefte örtük kuralların içerdiği komutlar varsa, örüntü kuralları bu ilk bağımlılıkta eklenecektir. $? Aralarında boşluk olan tüm bağımsızlık isimlerinin hedefi yenidir.Arşive üye olan bağımlılıklar için,yalnızca adlandırılmış üye kullanılır.

$^

Aralarında boşluk olan tüm bağımsızlık isimleridir.Arşiv üye bağımlılıkları için,sadece üye adları kullanılır.Bir hedef her bir bağımlı dosyadan sadece birine bağlıdır,her dosya kaç kez olursa olsun bir kere listelenir.Eğer bir hedef birden fazla bağımlılığı listeliyorsa ,yani $^ adı sadece bir değer kopyasını içerir.

$+

Bu $^ gibi,ama birden fazla kez listelenen bağımlılıklardır bunlar makefile listelenerek çoğaltılır.Bu öncelikle belli bir düzen içinde kütüphane dosyası adlarını tekrarlamak anlamlı olduğu komutlarla bağlantı kurmak için faydalıdır.

$*

Örtük bir kural ile eşleşen kökdür.Eğer hedef dir/a.foo.b ve a.%.b hedef bir kalıptır o zaman kök dizini dir/foo dur. Kök dosya isimlerini oluşturmak için yararlıdır.Statik bir desen kural,kök hedef kalıbı içerisinde bir dosya adı parçasıdır.

Açık bir kuralda hiçbir kök yoktur;böylece $* bu şekilde kontrol edilemez.Bunun yerine ,eğer hedef ismi tanınan bir hedef ile bitiyorsa, $* hedef ismi eksi son eki ayarlanmışmı.Örneğin,hedef adı foo.c ise c nin bir son ek olduğunu $*,foo.c ayarlanır .GNU make yalnızca bu uyumluluk için bu tuhaf şeyi yapar,make ise diğer uygulamaları ile.Siz genellik örtük kurallar veya statik desenler dışında $* kullanmaktan kaçınmalısınız.

Açık bir kural hedef ismi tanınan bir ek ile bitmiyorsa, $*,bu kural için boş bir dizi ayarlanmıştır.

$?

Sadece bağımlılıkları çalıştırmak istediğinizde bile açık kurallar yararlıdır.Örneğin lib adlı bir arşiv birkaç nesne dosyasını tanımladığı varsayılır.Bu kural sadece arşiv değişen nesne dosyalarını kopyalar:

```
lib: foo.o bar.o lose.o win.o

ar r lib $?
```

Burada yapacağını aslında önceden tanımlanmış model kurallarına bir örnek.Bu kural tüm .c dosyalarını .o dosyalarına derler:

%.o : %.c

$(CC) -c $(CFLAGS) $(CPPFLAGS) $< -o $@

x.c herhangi dosyasını x.o yapabilirsiniz.$@ komutu otomatik değişkenler kullanır ve $< hedef dosya isimleri yerine ve her durumda kaynak dosyası kurallı geçerlidir.

Jokerler(wildcards)

Jokerler kabuk tarafından genişletilmiş olarak bir kuralın komutlarında kullanılabilir.Örneğin,bu komut tüm obje dosyalarını silen kuraldır:

Clean:

rm-f *.o

Jokerler ayrıca bir kuralın bağımlılıklarında kullanışlıdır.Aşagıdaki makefile içerisindeki kural ile , "make print" tüm .c dosyalarını yazdıracak:

print: *c

lpr -p $?

touch print

Bu kural "print" hedef dosyayı boş gibi kullanır.Otomatik değişken $? Sadece bu dosyaları yazdırmak için kulanılır.

Not:Bir değişken taımladığınız zaman, joker(wildcard) genişlemesi oluşmaz.Eger bu kodu yazarsanız:

objects = *.o

"objects" değişkenin değeri mevcut string *.o'dur.Ancak "objects"in değerini hedefde,bağımlılıkta yada komutta kullanırsanız,joker genişlemesi(wildcard expansion) aynı zamanda meydana gelir.Yukarıdakı kodun yerine bunu kullanmalıyız :

Objects:=$(wildcard *.o)