Sistem Programlama ve İleri C Uygulamaları – II
Kurs Notları

Kaan ASLAN

C ve Sistem Programcıları Derneği

---

1. GUI Ortamlarında Kullanılan Mesaj Tabanlı Programlama Modelinin Temelleri

Bu bölümde bugün yaygın olarak kullanılan Windows gibi, XWindow gibi GUI ortamlarındaki mesaj tabanlı programlama modeli üzerinde durulacaktır. Bu bölümde kurs katılımcılarınına pencereli GUI (Graphical User Interface) sistemlerinde uygulama geliştirme becerisi kazandırma hedeflenmemektedir. Biz bu bölümde biz yalnızca sistem programcısı olarak mesaj tabanlı programlama modelinin alt yapısını aşağı seviyeli olarak ele almayı hedefliyoruz.

1.1. Grafik Kartları ve Ekran Belleği

Bilgisayar sistemlerine terminal bağlanması ilk kez 1957 yılında gerçekleştirilmiştir. Bu yıllar aynı zamanda programların zaman paylaşımlı (time sharing) olarak çalıştırılmaya başlandığı yıllardır. O zamana kadar insanlarlar bilgisayarlar doğrudan etkileşemiyordu. Programcı programını delikli kartlara delip bilgisayar operatörüne teslim ediyordu. Operatör de programı çalıştırıp sonucu programcıya kağıt çıktısı olarak veriyordu.

90'lı yılların ortalarına kadar terminal çalışması ağırlıklı olarak "text mode"da devam etmiştir. "Text mode" o zamanlar default çalışma moduydu. Bu çalışma modunda terminal ekranı yalnızca daktilodaki gibi karakterleri bir kalıp olarak ekrana basabiliyordu. Yani "text mode"da ekrana basılacak en küçük birim bir karakterdi.

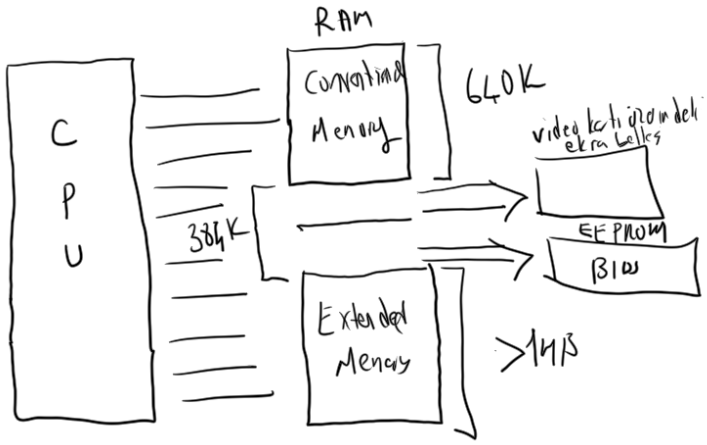
İlk grafik ekranlar 70 yıllarda ortaya çıkmaya başladıysa da grafik modun default duruma gelmesi 90'lı yıllarda olmuştur. Grafik modda ekrana çıkartılabilecek en küçük birime pixel (picture element sözcüklerinden kısaltma) denilmektedir. Ekran bu modda pixel'lerden oluşan bir matris gibidir. Her pixe'lin rengi diğerlerinden bağımsız olarak atanabilmektedir. Böylece grafik ekranda aslında bütün görüntüler pixel'lerin yan yana getirilmesiyle oluşturulmaktadır.

Pekiyi neden grafik çalışma nispeten daha sonraları başlamıştır? İşte grafik ekrandaki pixel'lerin görüntülenmesi ve bunların grafik kartında oluşturulması için belli bir teknoloji düzeyi gerekiyordu. Bugün artık hem ekranlar hem de grafik kartları teknolojik olarak çok ilerlemiştir. Örneğin 1980-1981 yıllarına IBM ilk kişisel bilgisayarı (IBM PC) piyasaya sürdüğünde oradaki ekran ancak kalıp karakterlkeri görüntüleme kapasitesine sahipti. Yani "text mode"da çalışıyordu. IBM daha sonra sırasıyla EGA, VGA ve SVGA (Super VGA) kartlarını geliştirdi. Bu kartlar pixel düzeyinde grafik işlem yeteneğine sahipti. Tabii EGA ve VGA kartları grafik görüntüyü gösterebilen monitörlerle kullanılabiliyordu. 80'li yıllarda ayrıca Hercules kartları da yoğun olarak kullanılmıştır.

Bugün kullandığımız PC'lerde (PC'yi burada genel bir terim olarak kullanıyoruz. Bu terim notebook'ları ve diziüstü bilgisayarları da kapsamaktadır) ekran pasif bir birimdir. Görüntü bilgisayarın içerisindeki grafik kartının belleği üzerinde oluşturulur. Grafik kartı da belli bir periyotta (buna "refresh rate" deniyor) bu belleğin içeriğini ekrana yollar. Grafik kartı üzerindeki görüntünün oluşturulduğu bu belleğe "ekran belleği (video RAM)" denilmektedir. Ekran belleği doğrudan CPU'nun adres alanı içerisindedir. Bugün kullandığımız

PC'lerde ilk 1 MB'nin son 384 K'sı BIOS'un içinde bulunduğu EEPROM belleğe ve ekran belleğine yönlendirilmiş durumdadır.

Anahtar Notlar: 8086 işlemcisiyle oluşturulmuş ilk PC'ler 1 MB belleği adresleyebiliyordu. Bu 1MB alanın son 384K'sı BIOS ve ekran belleğine yönlendirildiği için DOS işletim sistemi yalnızca bu 1 MB'nin 640K'sını kullanabiliyordu. Bu 640K'ya "geleneksel bellek (conventional memory)" deniliyordu.



Görüldüğü gibi RAM'in ilk 1MB'sinin son 384K'sında aslında RAM değil başka birimler bulunmaktadır. CPU orayı adreslerken orada RAM'in mi yoksa ekran belleğinin mi bulunduğunu bilmez. Eletrisksel olarak CPU'nun adres alanına farklı RAM blokalrı ve ROM'lar ya da başka aygıtlar bağlanabilir. Ekran belleği A0000 adresinden başlamaktadır. Biz C'de ya da sembolik makine dilinde oraya erişebiliriz. Ancak Windows gibi, Linux gibi işletim sistemlerinde ekran belleği korunmuştur. User mode prosesler oraya doğrudan erişemezler.

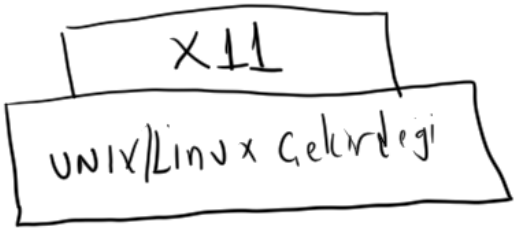
Bugün SVGA kartları çok gelişmiştir. Ekran kartının üzerindeki RAM'in bir bölümü ekran belleği için bir bölümü ise GPU'nun (Graphics Processing Unit) grafik işlemleri yapabilmesi için kullanılmaktadır. Bugünkü GPU'lar görüntü işleme konusunda rutinleri olan pek çok görüntü işlemini birinci elden yapabilen yetenekli işlemcilerdir. Örneğin bir şekil döndürmesini doğrudan GPU yapabilmektedir. Sonuç görüntüyü ekran belleğine aktarabilmektedir. Hatta GPU'lar paralel programlamada ayrı bir işlemci gibi bile bazı işlemlerde kullanılabilmektedir. Bugünkü ekran kartları ve GPU'lar bazı ayrıntılı özelliklere sahiptir. Kursumuzda bu ayrıntılardan bahsedilmeyecektir.

Anahtar Notlar: Genel olarak bilgisayar mimarisinde birtakım donanım aygıtlarının sanki RAM'in bir parçasıymış gibi gösterilmesine "memory mapped IO" denilmektedir. Grafik kartları tipik olarak "memory mapped IO"ya bir örnek oluşturmaktadır.

1.2. GUI Sistemlerine Özet Bir Bakış

Windows'un çekirdek (kernel) ile entegre edilmiş bir GUI alt sistemi vardır. Başka bir deyişle Windows'ta pencereli çalışma başka bir katman tarafından değil doğrudan işletim sisteminin bir parçası tarafından sağlanmaktadır. Windows'u GUI alt sistemi olmadan kullanmak mümkün olsa da anlamlı değildir.

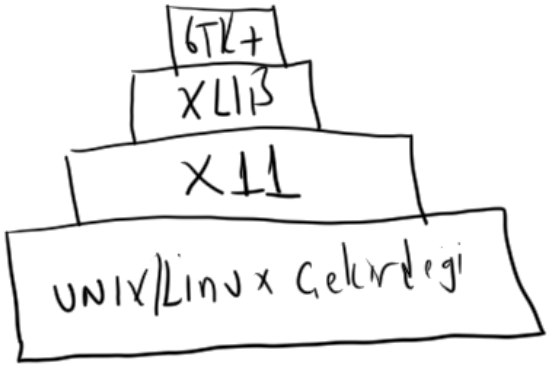
UNIX/Linux sistemlerinde grafik arayüz çekirdeğin üzerine oturtulan ve ismine X11 (ya da XWindow) denilen bir alt sistem tarafından sağlanmaktadır. Yani örneğin Linux'un çekirdeğinin kaynak kodlarında pencere kavramına ilişkin hiçbir şey yoktur. Ancak Windows'ta vardır.



Anahtar Notlar: Biz istersek bir Linux sistemini grafik arayüz olmadanda başlatabiliriz. Bunun için yapılması gereken tek şey "run level denilen bir değeri değiştirmektir. Bu işlem "/etc/init.d" ya da "/etc/inittab" dosyaları edit edilerek değiştirilebilir. (Örneğin "run level" değerini 3'e çekerek sistemin X11 olmadan başlatılmasını sağlayabiliriz.)

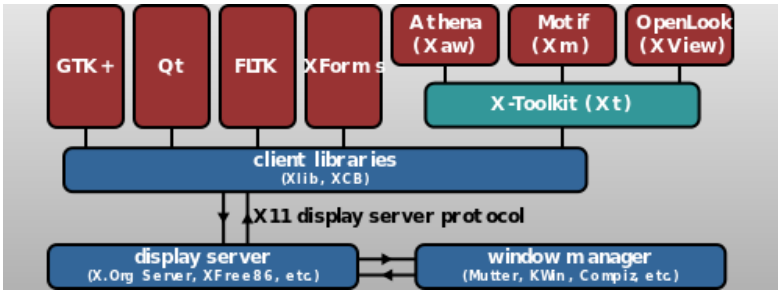
X11 grafik sistemi client-server tarzda çalışmaktadır. Yani sanki X11 bir server program gibi, pencere açmak ve pencereler üzerinde işlem yapmak isteyen programlar da client programlar gibidir. X11 sisteminde işlem yapabilmek için oluşturulmuş temel kütüphaneye XLIB denilmektedir. XLIB'i X11'in API kütüphanesi olarak düşünebiliriz. Son yıllarda XLIB'in XCB isimli daha modern bir versiyonu da oluşturulmuştur. XLIB ve XCB temelde C programlama dilinden kullanılmak için tasarlanmıştır. Ancak bu kütüphaneler başka dilelrden de kullanılabilmektedir.

Pek çok programcı için X11 sistemleri için tasarlanmış olan XLIB ve XCB oldukça aşağı seviyelidir. Bu kütüphanelerle kullanıcı arayüzü oluşturmak biraz zahmetlidir. İşte bu nedenle XLIB'den faydalanılarak daha yüksek seviyeli kütüphaneler de oluşturulmuştur. Bunların en yaygını GTK+'tır.



GTK+ da temelde bir C kütüphanesidir. GNOME pencere yöneticisi GTK+ kullanılarak yazılmıştır. GTK+'ın dışında ayrıca XLIB üzerine oturtulmuş Xt gibi, Motif gibi başka kütüphaneler de bulunmaktadır.

X11 sistemlerindeki kütüphaneler aşağıdaki şekille özetlenebilir:



Bu şekilden de gördüğünüz gibi Qt de tıpkı GTK+ gibi XLIB kullanılarak yazılmış bir framework ve kütüphanedir. Ancak GTK+ C Programlama Dilini için tasarlanmış olmasına karşın, Qt nesne yönelimli olarak C++ için tasarlanmıştır. Ayrıca Qt "cross platform" olmasından dolayı da pek çok programcı tarafından tercih edilmektedir.

1.3. GUI Ortamlarında Mesaj Tabanlı Programlama Modeli

Grafik arayüze sahip pencereli sistemlerde genel olarak mesaj tabanlı (message driven/event driven) çalışma modeli kullanılmaktadır. Mesaj tabanlı çalışma modelinin ayrıntıları sistemden sisteme değişebilmekle birlikte burada biz her sistemde geçerli olan bazı temel bilgileri vermekle yetineceğiz.

Mesaj tabanlı programlama modelinde klavye ve fare gibi aygıtlarda oluşan girdileri programcı kendisi almaya çalışmaz. Fare gibi, klavye gibi girdi aygıtlarını işletim sisteminin (ya da GUI alt sistemin) kendisi izler. Oluşan girdi olayı hangi pencereye ilişkinse işletim sistemi ya da GUI alt sistem, bu girdi olayını "mesaj" adı altında bir yapıya dönüştürerek o pencerenin ilişkin olduğu (yani o pencereyi yaratan) programın "mesaj kuyruğu (message queue)" denilen bir kuyuk sistemine yerleştirir. Mesaj kuyruğu içerisinde mesajların bulunduğu FIFO prensibiyle çalışan bir kuyruk veri yapısıdır. Sistemin daha iyi anlaşılması için süreci maddeler halinde özetlemek istiyoruz:

1. Her programın (ya da thread'in) "mesaj kuyruğu" denilen bir kuyruk veri yapısı vardır. Mesaj kuyruğu mesajlardan oluşmaktadır.

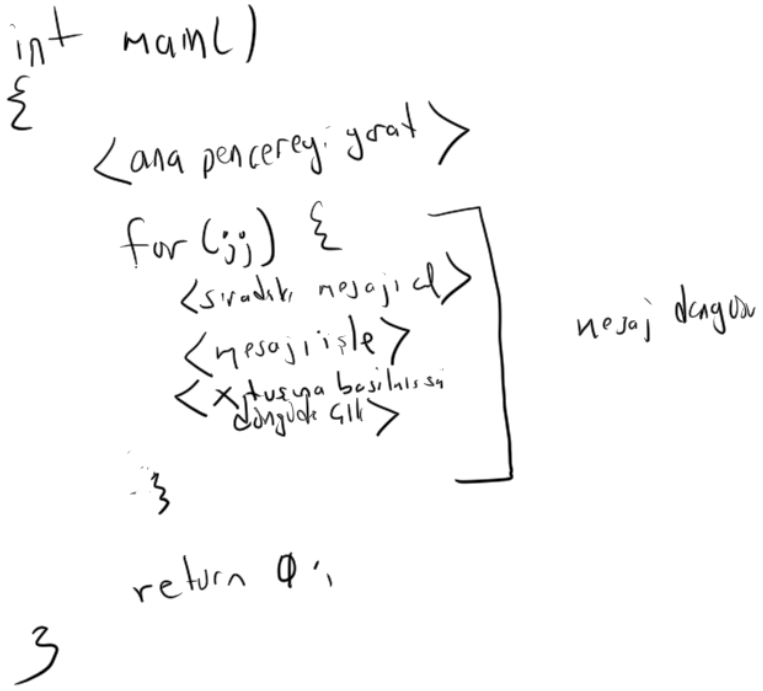
2. İşletim sistemi ya da GUI alt sistem gerçekleşen girdi olaylarını "mesaj (message)" adı altında bir yapı formatına dönüşürmekte ve bunu pencerenin ilişkin olduğu programın (ya da thread'in) mesaj kuyruğuna eklemektedir.

3. Mesajlar ilgili olayı betimleyen ve ona ilişkin bazı bilgileri barındıran yapı (structure) nesleridir. Örneğin Windows'ta mesajlar MSG isimli bir yapıyla temsil edilmişleridir. Bu yapının elemanlarında mesajın ne mesajı olduğu (yani neden gönderildiği) ve olaya ilişkin bazı bilgiler bulunur.

Görüldüğü gibi GUI programlama modelinde girdileri programcı elde etmeye çalışmamaktadır. Girdileri bizzat işletim sisteminin kendisi ya da GUI alt sistemi elde edip programcıya mesaj adı altında iletmektedir.

GUI programlama modelinde işletim sisteminin (ya da GUI alt sistemin) oluşan mesajı ilgili programın (ya da thread'in) mesaj kuyruğuna eklemenin dışında başka bir sorumluluğu yoktur. Mesajların kuyruktan alınarak işlenmesi ilgili programın sorumluluğundadır. Böylece GUI programcısının mesaj kuyruğuna bakarak sıradaki mesajı alması ve ne olmuşsa ona uygun işlemleri yapması gerekir. Bu modelde programcı kodunu şöyle düzenler: Bir döngü içerisinde sıradaki mesajı kuyruktan al, onun neden gönderildiğini belirle, uygun işlemleri yap, kuyrukta mesaj yoksa da blokede bekle". İşte GUI programlarındaki mesaj kuyruğundan mesajı alıp işleyen döngüye mesaj döngüsü (message loop) denilmektedir.

Bir GUI programının işleyişini tipik akışı aşağıdaki gibi bir kodla temsil edebiliriz:



Bu temsili koddan da görüldüğü gibi tipik bir GUI programında programcı bir döngü içerisinde mesaj kuyruğundan sıradaki mesajı alır ve onu işler. Mesajın işlenmesi ise "ne olmuş ve ben buna karşı ne yapmalıyım?" biçiminde oluşturulmuş olan kodlarla yapılmaktadır.

Pekiyi bir GUI programı nasıl sonlanmaktadır? İşte pencerenin sağındaki (bazı sistemlerde solundaki) X simgesine kullanıcı tıkladığında işletim sistemi ya da GUI alt sistem bunu da bir mesaj olarak o pencerenin ilişkin olduğu prosesin (ya da thread'in) mesaj kuyruğuna bırakır. Programcı da kuyruktan bu mesajı alarak mesaj döngüsünden çıkar ve program sonlanır.

GUI ortamımız ister .NET, ister Java, ister MFC olsun, isterse Qt olsun, işletim sisteminin ya da GUI alt sistemin çalışması hep burada ele açıklandığı gibidir. Yani örneğin biz .NET'te ya da Java'da işlemlerin sanki başka biçimlerde yapıldığını sanabiliriz. Aslında işlemler bu ortamlar tarafından aşağı seviyede yine burada anlatıldığı gibi yapılmaktadır. Bu ortamlar (frameworks) ya da kütüphaneler çeşitli yükleri üzerimizden alarak bize daha rahat bir çalışma modeli sunarlar. Ayrıca şunu da belirtmek istiyoruz: GUI programlama modeli özellikle nesne yönelimli programlama modeline çok uygun düşmektedir. Bu nedenle bu konuda kullanılan kütüphanelerin büyük bölümü sınıflar biçiminde nesne yönelimli diller için oluşturulmuş durumdadır.

Şimdi GUI programlama modelindeki mesaj kavramını biraz daha açalım. Yukarıda da belirttiğimiz gibi bu modelde programcıyı ilgilendiren çeşitli olaylara "mesaj" denilmektedir. Örneğin klavyeden bir tuşa basılması, pencere üzerinde fare ile tıklanması, pencere içerisinde farenin hareket ettirilmesi gibi olaylar hep birer mesaj oluşturmaktadır. İşletim sistemleri ya da GUI alt sistemler mesajları birbirinden ayırmak için onlara birer numara karşılık getirirler. Örneğin Windows'ta mesaj numaraları WM_XXX biçiminde sembolik sabitlerle kodlanmıştır. Programcılar da konuşurken ya da kod yazarken mesaj numaralarını değil, bu sembolik sabitleri kullanırlar. (Örneğin WM_LBUTTONDOWN, WM_MOUSEMOVE, WM_KEYDOWN gibi) Mesajların numaraları yalnızca gerçekleşen olayın türünü belirtmektedir. Oysa bazı olaylarda gerçekleşen olaya ilişkin bazı bilgiler de söz konusudur. İşte bir mesaja ilişkin o mesaja özgü bazı parametrik bilgiler de işletim sistemi ya da GUI alt sistem tarafından mesajın bir parçası olarak mesajın içerisine kodlanmaktadır. Örneğin Windows'ta biz klavyeden bir tuşa bastığımızda Windows WM_KEYDOWN isimli mesajı programın mesaj kuyruğuna bırakır. Bu mesajı kuyruktan alan programcı mesaj numarasına bakarak klavyenin bir tuşuna basılmış olduğunu anlar. Fakat hangi tuşa basılmıştır? İşte Windows basılan tuşun bilgisini de ayrıca bu mesajın içerisine kodlamaktadır. Örneğin WM_LBUTTONDOWN mesajını Windows farenin sol tuşuna tıklandığında kuyruğa bırakır. Ancak ayrıca basım koordinatını da mesaja ekler. Yani bir mesaj oluştuğunda yalnızca o mesajın hangi tür bir olay yüzünden oluştuğu bilgisini değil aynı zamanda o olayla ilgili bazı bilgileri de kuyruktaki mesajın içerisinden alabilmekteyiz.

GUI programlama modelinde bir mesaj oluştuğunda o mesajın bir an evvel işlenmesi ve akışın çok bekletilmemesi gerekir. Aksi takdirde programcı kuyruktaki diğer mesajları işleyemez bu da "program donmuş etkisi" yaratmaktadır. Eğer bir mesaj alındığında uzun süren bir işlem yapılmak isteniyorsa bir thread oluşturulup o işi o thread'e devretmek ve böylece mesaj döngüsünün işlemesini sağlamak gerekir.

GUI programlama modellerinde genel olarak mesaj kavramı pencere kavramıyla ilişkilendirilmiştir. Yani bir pencere yaratılmadıktan sonra bir mesajın oluşma durumu da yoktur. Bu nedenle mesaj döngüsüne girmeden önce programcının en az bir pencere (tipik olarak programın ana penceresi) yaratmış olması gerekir.

Windows gibi bazı sistemlerde thread'lerle ilişkilendirilmiştir. Bu sistemlerde prosesin tek bir mesaj kuyruğu yoktur. Her thread'in ayrı bir mesaj kuyruğu vardır. Bu durumda işletim sistemi ya da GUI alt sistem bir pencereye ilişkin bir işlem gerçekleştiğinde o pencerenin hangi prosesin hangi thread'i tarafından yaratılmış olduğunu belirler ve mesajı o thread'in mesaj kuyruğuna bırakır. Böylece biz bir thread oluşturup o thread'te de bir pencere yaratmışsak artık bizim de o thread'te o pencerenin mesajlarını işlemek için mesaj döngüsü oluşturmamız gerekir. Tabii eğer thread'imizde biz hiçbir pencere oluşturmamışsak böyle bir mesaj döngüsünü oluşturmamıza da gerek yoktur. (Örneğin Microsoft eğer bir thread bir pencere yaratmışsa böyle thread'lere "GUI thread'ler" yaratmamışsa "worker thread'ler" demektedir).

1.4. Windows Sistemlerinde GUI Programlama Modelinin Temelleri

Windows sistemlerinde GUI programlama için USER32.DLL içerisindeki API fonksiyonları kullanılmaktadır. (USER32.DLL dosyasının ismindeki 32 sizi yanıltmasın, 64 bit sistemlerde bu DLL aslında isminin aksine zaten 64 bittir.) Bu API fonksiyonları Windows GUI programlama modelinin temellerini bire bir yansıtmaktadır.

1.4.1. Windows Sistemlerinde İskelet GUI Programı

Windows sistemlerinde ekrana boş bir pencere çıkartan iskelet GUI programı şöyle yazılabilir:

```
#include <windows.h>

LRESULT CALLBACK WndProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM lParam);

int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, LPSTR lpszCmdParam, int nCmdShow)
{
    WNDCLASS wndClass;
    HWND hWnd;
    MSG message;

    if (!hPrevInstance) {
        wndClass.style = CS_HREDRAW | CS_VREDRAW;
        wndClass.cbClsExtra = 0;
        wndClass.cbWndExtra = 0;
        wndClass.hInstance = hInstance;
        wndClass.hIcon = LoadIcon(NULL, IDI_QUESTION);
        wndClass.hbrBackground = GetStockObject(WHITE_BRUSH);
        wndClass.hCursor = LoadCursor(NULL, IDC_ARROW);
        wndClass.lpszMenuName = NULL;
        wndClass.lpszClassName = "Generic";
        wndClass.lpfnWndProc = (WNDPROC)WndProc;
        if (!RegisterClass(&wndClass))
            return -1;
    }
    hWnd = CreateWindow("Generic", "Sample Windows", WS_OVERLAPPEDWINDOW, CW_USEDEFAULT, 0,
            CW_USEDEFAULT, 0, NULL, NULL, hInstance, NULL);
    if (!hWnd)
        return -1;
    ShowWindow(hWnd, SW_RESTORE);
```

```
    UpdateWindow(hWnd);
    while (GetMessage(&message, 0, 0, 0)) {
        TranslateMessage(&message);
        DispatchMessage(&message);
    }
    return message.wParam;
}

LRESULT CALLBACK WndProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM lParam)
{
    switch (message) {
        case WM_DESTROY:
            PostQuitMessage(0);
            break;
        default:
            return DefWindowProc(hWnd, message, wParam, lParam);
    }
    return 0;
}
```

Windows API programlamda kullanılan çeşitli typedef türleri ve sembolik sabitler vardır. Tüm bu typedef'ler ve sembolik sabitler ve API fonksiyonlarının prototipleri <windows.h> dosyasının içerisindedir. API programlamada ağırklı yazım biçimi olarak macar notasyonu (Hungarian Notation) kullanılmaktadır. Macar Notasyonunda değişkenler onların türlerini belirten küçük harf öneklerle başlatılır, sonra Pascal tarzı harflendirme ile devam edilir. Örneğin:

```
long lNumberOfSectors;
HINSTANCE hInstance;
DWORD dwFlags;
char szPath[MAX_PATH];
```

Türler için önemli kullanılan önekler şunlardır:

p ya da lp	Gösterici (lp long pointer'dan gelme. Eskiye uyum için hala kullanılıyor.)
l	long
w	WORD
dw	DWORD
h	HANDLE
sz	char * (fakat yazı gösterir)
b	BOOL
f	float
d	double

API programlamada kullanılan typedef tür isimleri hep büyük harflerle oluşturulmuştur. En çok kullanılanları şunlardır:

BYTE	Bir byte'lık işaretsiz tamsayı türü (unsigned char)
WORD	İki byte'lık işaretsiz tamsayı türü (unsigned short int)
DWORD	Dört byte'lık işaretsiz tamsayı türü (unsigned long int ya da unsigned int)
HANDLE	Handle türü (void *)
HXXXX	Handle türü (void *)
PXXX, LPXXX	XXX türünden adres türü (örneğin LPINT, LPVOID, PVOID, LPDWORD)
PCXXX,	XXX türünden gösterdiği yer const olan adres (Örneğin LPCVOID demek const void *
LPSTR	Yazıyı gösteren adres (char *)
LPTSTR	Yazıyı gösteren UNICODE destekli adres (char * ya da wchar_t *)

BOOL	int türünü belirtir. Fakat anlam olarak başarı ve başarısızlık düşünülmelidir. Geri dönüş değeri BOOL olan API fonksiyonları başarı durumunda sıfır dışı değere, başarızılık durumunda sıfır değerine geri dönerler.

Fonksiyon prototiplerinde parametre değişkenlerinin önündeki __in, __out ve __in_out makroları okunabilirliği artırmak için düşünülmüştür. Bu makrolar aşağıdaki gibi define edilmiştir:

```
#define __in
#define __out
#define __in_out
```

Görüldüğü gibi aslında bu makrolar önişlemci tarafından silinmektedir. __in makrosu fonksiyonun parametre değişkenindeki bilgiyi kullanacağı fakat ona bir değer yerleştirmeyeceği anlamına gelir. __out tam tersine fonksiyonun parametre değişkenindeki değeri değiştireceği anlamına gelmektedir. __in_out ise fonksiyonun hem parametre değişkenindeki değeri kullanacağı hem de ona yeni bir değer yerleştireceği anlamına gelir. Tabii bu makroların gerekliliği tartışılabilir. (Bilindiği gibi zaten gösterici olmayan parametre değişkenleri __in olmak zorundadır. Gösterici parametre değişkenlerinde __in ya da _out durumu göstericinin const olup olmamasıyla da zaten anlaşılmaktadır. O halde bu makrolar yalnızca __in_out durumunun anlaşılmasına katkıda bulunmaktadır.)

Yukarıdaki typedef isimlerinin dışında <Windows.h> dosyası üzerinde pek çok sembolik sabit ve makro da bulunmaktadır. Ancak bunların hepsini burada ele almamız imkansız. Fakat sembolik sabitler genel olarak onların hangi amaçla bulundurulduğuna ilişkin öneklere sahip olduğunu belirtelim (örneğin WS_CHILD, WM_COMMAND gibi). İleride kullanacağımız iki makroyu da burada tanıtmayı uygun görüyoruz. istiyoruz: LOWORD makrosu 4 byte'lık bir değerin bize düşük anlamlı 2 byte'ını, HIWORD makrosu da yüksek anlamlı 2 byte'ını vermektedir.

1.4.1.1. Windows Sistemlerinde İskelet GUI Programının Açıklaması

Windows GUI uygulamalarında programın başlangıç noktası (entry point) main isimli fonksiyon değil WinMain isimli fonksiyondur. WinMain fonksiyonunun dört parametresi vardır. Bu parametrelerin anlamları şöyledir:

HINSTANCE hInstance: Bu parametre programın (yani PE formatının) bellekteki yüklenme adresini belirtir. (16 bit Windows sistemlerinde bu adres prosesin "modül veritabanı (module database)" adresini belirtiyordu.)

HINSTANCE hPrevInstance: 16 bit Windows sistemlerinde program birden fazla kez çalıştırılmışsa bu parametre önceki çalıştırmaya ilişkin modül veritabanı adresini belirtiyordu. Ancak bu parametre 32 bit sistemlerde ve 64 bit sistemlerde bu değer her zaman NULL değerinde olmaktadır.

LPSTR lpszCmdParam: Bu parametre programın komut satırı argümanlarını belirtmektedir. Windows GUI uygulamalarında program ismi dahil olmak üzere tüm komut satırı argümanları tek bir yazı biçiminde WinMain fonksiyonuna aktarılmaktadır.

int nCmdShow: Bu parametre programın ana penceresinin nasıl görüntüleneceği konusunda bir tavsiye niteliğindedir. Bu değer PE formatından alınarak WinMain fonksiyonuna aktarılmaktadır.

İskelet GUI programında ilk olarak bir pencere sınıfının register ettirilmiştir:

```
if (!hPrevInstance) {
    wndClass.style = CS_HREDRAW | CS_VREDRAW;
    wndClass.cbClsExtra = 0;
    wndClass.cbWndExtra = 0;
    wndClass.hInstance = hInstance;
    wndClass.hIcon = LoadIcon(NULL, IDI_QUESTION);
    wndClass.hbrBackground = GetStockObject(WHITE_BRUSH);
    wndClass.hCursor = LoadCursor(NULL, IDC_ARROW);
```

```
        wndClass.lpszMenuName = NULL;
        wndClass.lpszClassName = "Generic";
        wndClass.lpfnWndProc = (WNDPROC)WndProc;
        if (!RegisterClass(&wndClass))
            return -1;
    }
```

16 bit Windows sistemlerinde bu register ettirme işlemi yalnızca programın ilk kopyası çalıştırıldığında yapılmak zorundaydı. Halbuki 32 bit ve 64 bit sistemlerde her zaman yapılmak zorundadır. (Zaten hPrevInstance değerinin 32 bit ve 64 bit Windows sistemlerine her zaman NULL olduğunu anımsayınız). Dolayısıyla buradaki if kontrolüne 32 bit ve 64 bit Windows programlarında artık hiç gerek yoktur. Ancak geriye doğru uyum için pek çok programcı bu kontrolü hala bulundurmaktadır.

Pencere sınıfının register ettirilmesi ne anlama gelir? Bir pencerenin pek çok özelliği vardır. Bu özelliklerden bazıları pencere sınıfı denilen bir yapı ile temsil edilir. Bir isim altında oluşturulur. Sonra pencere yaratılırken bu isim kullanılır. Böylece pencerenin bazı bilgileri önceden yapılmış bu belirlemelerden alınır. Pencere sınıfı (buradaki sınıfın C++'taki sınıf ile bir ilgisi yoktur) WNDCLASS isimli bir yapıyla temsil edilmiştir:

```
typedef struct {
    UINT style;
    WNDPROC lpfnWndProc;
    int cbClsExtra;
    int cbWndExtra;
    HINSTANCE hInstance;
    HICON hIcon;
    HCURSOR hCursor;
    HBRUSH hbrBackground;
    LPCTSTR lpszMenuName;
    LPCTSTR lpszClassName;
} WNDCLASS, *PWNDCLASS;
```

Bu yapıda pencereler için şu belirlemeler yapılmaktadır:

- Pencere fonksiyonunun hangi fonksiyon olacağı
- Fare pencere üzerindeyken görüntülenecek fare oku (cursor)
- Pencerenin simgesi (icon)
- Pencerenin zemin rengi
- Pencere stillerinin nasıl olacağı

İskelet programda pencere sınıfı RegisterClass API fonksiyonuyla sisteme register ettirilmiştir. RegisterClass API fonksiyonunun RegisterClassEx isimli genişletilmiş bir biçimi de vardır. Bu genişletilmiş biçim WNDCLASS yerine WNDCLASSEX yapısını parametre olarak almaktadır.

İskelet programda pencere sınıfı register ettirildikten sonra sıra ana pencerenin yaratılmasına gelmiştir. Windows'ta her türlü pencere (üst pencereler, ana pencereler ya da dialog pencereleri) CreateWindow API fonksiyonuyla yaratılmaktadır. CreateWindow fonksiyonunun daha sonradan CreateWindowEx isimli genişletilmiş bir biçimi de oluşturulmuştur. CreateWindow fonksiyonunun prototipi şöyledir:

```
HWND CreateWindow(LPCTSTR lpClassName,
    LPCTSTR lpWindowName,
    DWORD dwStyle,
    int x,
    int y,
    int nWidth,
    int nHeight,
    HWND hWndParent,
    HMENU hMenu,
    HINSTANCE hInstance,
```

```
    LPVOID lpParam
);
```

Fonksiyonun birinci parametresi pencerenin hangi pencere sınıfından yaratılacağını belirtir. (Anımsanacağı gibi pencere sınıfı bir isim altında register ettiriliyordu. İşte birinci parametre bu ismi alıyor.) İkinci parametre pencere yazısını belirtmektedir. Windows'ta her pencerenin bir pencere yazısı vardır. (Örneğin ana pencerelerde bu yazı pencere başlığında görüntülenecek olan yazıdır, düğmelerin (buttons) pencere yazısı düğmenin üzerindeki yazıdır. Seçenek kutularının yazısı küçük kutucuğun yanındaki yazıdır vs.) Üçüncü parametre pencere stillerini belirtir. Pencere stilleri <Windows.h> dosyası içerisinde WS_XXX biçimindeki sembolik sabitlerle bit düzeyinde bayraklarla define edilmiştir. Örneğin eğer alt pencere yaratılacaksa burada WS_CHILD bayrağının kullanılması gerekir. Fonksiyonun sonraki dört parametresi pencerenin ilk açıldığındaki konumunu ve büyüklüğünü alır. Fonksiyonun hWndParent parametresi ise eğer pencere ana pencereyse NULL olarak, alt pencereseyse onun üst penceresinin handle değeri olarak girilmelidir. hMenu parametresi pencerenin menüsü varsa o menü handle'ını belirtir. Penceresinin menüsü yoksa bu parametre de NULL biçiminde girilmelidir. hInstance parametresi WinMain'e geçirilen hInstance değerini alır. Son parametre isteğe bağlı bir değerdir. Bazı pencere sınıfları buraya bir yapı adresinin geçirilmesini isterler. Fonksiyonun ayrıntılı açıklaması için MSDN dokümanlarına başvurabilirsiniz. İskelet programda programın ana penceresi şöyle yaratılmıştır:

```
hWnd = CreateWindow("Generic", "Sample Windows", WS_OVERLAPPEDWINDOW, CW_USEDEFAULT, 0,
           CW_USEDEFAULT, 0, NULL, NULL, hInstance, NULL);
```

Windows'ta her pencerenin sistem genelinde tek olan bir handle değeri vardır. CreateWindow API fonksiyonu başarı durumunda bu handle değerine, başarısızlık durumunda ise NULL değerine geri döner.

Anahtar Notlar: Sistemdeki tüm prosesleri, pencereleri ve bu pencerelere gelen mesajları görüntüleyen Spy++ isimli bir araç vardır. Spy++ Microsoft tarafından geliştirilmiştir ve Visual Studio paketinin içerisindedir.

CreateWindow fonksiyonu ile ana pencere yaratıldıktan sonra o henüz görünür değildir. Onu görünür hale getirmek isiçin ShowWindow API fonksiyonu kullanılmalıdır. Ayrıca iskelet programdaki UpdateWindow çağrısının da mutlak gerekli olmadığını belirtelim. UpdateWindow fonksiyonu pencere fonksiyonunun çizim için çağrılmasına yol açmaktadır.

İskelet programda ana pencere görünür hale getirildikten sonra artık mesaj döngüsüne girilmiştir. Mesaj döngüsünün aşağıdaki gibi oluşturulmuştur:

```
while (GetMessage(&message, 0, 0, 0)) {
    TranslateMessage(&message);
    DispatchMessage(&message);
}
```

Mesaj döngüsünden sıradaki mesajı alan API fonksiyonu GetMessage fonksiyonudur. GetMessage mesajı kuyruktan alarak MSG isimli bir yapıya yerleştirmektedir. Windows'ta mesajlar en aşağı seviyeli olarak MSG isimli yapıyla temsil edilmektedir. GetMessage fonksiyonunun diğer parametreleri hangi aralıktaki mesajların alınacağına yöneliktir:

```
BOOL GetMessage(LPMSG lpMsg,
    HWND hWnd,
    UINT wMsgFilterMin,
    UINT wMsgFilterMax
);
```

MSG yapısı da şöyledir:

```
typedef struct {
    HWND hwnd;
    UINT message;
```

```
    WPARAM wParam;
    LPARAM lParam;
    DWORD time;
    POINT pt;
} MSG, *PMSG;
```

Yapının hWnd elemanı mesajın hangi pencereye gönderildiğini belirtir. Daha önceden de belirttiğimiz gibi Windows'ta mesajlar sistem tarafından belli bir pencere için gönderilmektedir. Windows'ta ister ana pencere olsun isterse herhangi bir alt pencere olsun bir thread'in yarattığı tüm pencerelerin mesajları aynı mesaj kuyruğuna (o thread'in mesaj kuyruğuna yerleştirilirler.

İskelet programda kuyruktan mesaj alındıktan sonra mesaj önce "translate" edilmiştir, sonra da "disptach" edilmiştir. Aslında mesajı "translate" etmek (yani TranslateMessage fonksiyonunu çağırmak) mutlak anlamda zorunlu değildir. Ancak mesaj DispatchMessage fonksiyonu çağrılarak "dispatchg edilmelidir. TranslateMessage WM_KEYDOWN mesajları için WM_CHAR mesajını oluşturarak bazı encoding dönüştürmelerini yapmaktadır. DispatchMessage fonksiyonunun kuyruktan alınan mesaj argüman yapılarak çağrıldığına dikkat ediniz. DispatchMessage ileride de göreceğimiz gibi pencere sınıfına ilişkin "pencere fonksiyonu" denilen bir fonksiyonun çağrılmasını sağlamaktadır.

Pekiyi mesaj döngüsünden nasıl çıkılmaktadır. İskelet programda ancak GetMessage fonksiyonu 0 ile (FALSE ile) geri dönerse mesaj döngüsünden çıkılabilmektedir. İşte GetMessage fonksiyonu kuyruktan WM_QUIT isimli mesajı aldığında 0 ile geri döner. O halde şöyle söyleyebiliriz: İskelet programda WM_QUIT mesajı kuyruğa bırakıldığında mesaj döngüsünden çıkılmaktadır. WM_QUIT mesajının oluşturulmasdı ve işlevi ileride ele alınacaktır. Önce pencere fonksiyonu üzerinde durmam istiyoruz.

1.4.2. Pencere Sınıflarına İlişkin Pencere Fonksiyonları

Bir pencereye bir mesaj gönderildiğinde çağrılması istenilen fonksiyona pencere fonksiyonu denilmektedir. Pencere fonksiyonun hangi fonksiyon olacağı pencere sınıfı register ettirilirken (WNDCLASS yapısında) belirlenir. Sonra pencere o sınıf kullanılarak yaratıldığında artık çağrılacak fonksiyon da belirlenmiş olmaktadır. Tabii kuyruktan mesaj alındığında pencere fonksiyonu otomatik çağrılmaz. Yukarıda da sözünü ettiğimiz gibi pencere fonksiyonun çağrılmasına DisptachMessage fonksiyonu yol açmaktadır. (Ancak DispatchMessage pencere fonksiyonun pencere fonksiyonunu çağırmanının dışında bazı diğer işlemleri de yaptığını belirtelim).

Pencere fonksiyonu herhangi bir fonksiyon olamaz. Pencere fonksiyonunun parametrik yapısının ve geri dönüş değerinin nasıl olması gerektiği WNDPROC isimli typedef türüyle belirlenmiştir:

```
typedef LRESULT (CALLBACK *WNDPROC)(HWND, UINT, WPARAM, LPARAM);
```

LRESULT long türünü belirtir. CALLBACK makrosu da __stdcall olarak define edilmiştir. (Yani pencere fonksiyonlarının çağırma biçimi (calling convention) __stdcall olmak zorundadır.)

Pencere fonksiyonunun birinci parametresi mesajın gönderildiği (yani ilişkin olduğu) pencerenin handle değerini belirtir. İkinci parametre gönderilen mesajın numarasıdır. Mesaj numaralarının <windows.h> içersinde WM_XXX biçiminde sembolik sabitlerle define edildiğini daha önce belirtmiştik. Üçüncü ve dördüncü parametreler mesaja ilişkin ekstra bilgileri belirtmektedir. Üçüncü parametre WPARAM türünden dörüdüncü parametre LPARAM türdendir. WPARAM 2 byte'lık işaretsiz tamsayı türü olarak, LPARAM ise 4 byte'lık işaretli tamsayı türü olarak typedef edilmişleridir. Pekiyi DispatchMessage pencere fonksiyonunu çağırırken bu parametreler için argümanları nasıl oluşturmaktadır? Aslında pencere fonksiyonuna aktarılan bu bilgilerin hepsi zaten kuyruktan alınan mesajın içerisinde (yani MSG yapısında) bulunmaktadır. DispatchMessage yalnızca onları mesajın içerisinden alarak pencere fonksiyonuna arüman yapmaktadır.

Bir pencereye mesaj geldiğinde o pencereye ilişkin pencere sınıfında belirtilen pencere fonksiyonunun çağrıldığını gördük. Pekiyi pencere fonksiyonu içerisinde mesajları nasıl işleyeceğiz? İşte bunun için en uygun yol mesaj numarasını switch içerisine almaktır. İskelet programdaki pencere fonksiyonunda da böyle yapıldığını görüyorsunuz:

```
LRESULT CALLBACK WndProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM lParam)
{
    switch (message) {
        case WM_DESTROY:
            PostQuitMessage(0);
            break;
        default:
            return DefWindowProc(hWnd, message, wParam, lParam);
    }
    return 0;
}
```

Windows'ta bazı mesajlarda bazı kritik işlemlerin yapılması gerekebilmektedir. Bu mesajların işlenmesi programcı için çok zahmetli olacağından DefWindowProc isimli bir fonksiyon bulundurulmuştur. Böylece programcı işlemediği mesajları DefWindowProc fonksiyonuna verir, mesajı onun işlemesini sağlar. Bazı mesajlar için DefWindowProc hiçbir şey yapmamaktadır. Ancak bazıları için yukarıda belirtildiği gibi bazı önemli işlemleri yapar. Pencere fonksiyonundan normal olarak sıfır ile geri dönülmelidir. Ancak bazı durumlarda bazı özel değerlerle geri dönülmesi gerekebilmektedir.

1.4.3. Windows GUI Programlarının Sonlandırılması

Windows'ta bir GUI programı tipik olarak şu adımlardan geçilerek sonlandırılır:

1) Kullanıcı ana pencerenin X simgesine tıklar ya da Alt + F4 tuşlarına basar. Bu durumda Windows kuyruğa o pencere için WM_CLOSE mesajını bırakır.

2) WM_CLOSE mesajını alan programcı tipik olarak (ancak zorunlu değil) DestroyWindow fonksiyonunu çağırır. (Bir pencere nasıl CreateWindow fonksiyonuyla yaratılıyorsa DestroyWindow fonksiyonuyla da yok edilmektedir.) Eğer programcı WM_CLOSE mesajını işlemezse DefWindowProc bu mesaj için zaten DestroyWindow fonksiyonunu çağırmaktadır. (Örneğin bu durumda örneğin X tuşuna basıldığında DestroyWindow fonksiyonun çağrılmasını engellersek pencere kapatılmaz.)

```
LRESULT CALLBACK WndProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM lParam)
{
    switch (message) {
        case WM_CLOSE:
            MessageBox(NULL, "You cannot close this window!", "Warning", MB_OK);
            break;
        case WM_DESTROY:
            PostQuitMessage(0);
            break;
        default:
            return DefWindowProc(hWnd, message, wParam, lParam);
    }
    return 0;
}
```

3) DestroyWindow fonksiyonu çağrıldığında bu fonksiyon pencereyi yok etmeden hemen önce pencereye WM_DESTROY mesajını yollar. Programcılar pencere yaratılırken eğer bazı tahsisatlar yapmışlarsa onları tipik olarak WM_DESTROY mesajında serbest bırakırlar.

4) WM_DESTROY mesajı için DefWindowProc birşey yapmaz. Tipik olarak eğer ana pencere yok edilmişse programcılar bu mesajda PostQuitMesssage API fonksiyonunu çağırırlar. PostQuitMessage fonksiyonu kuyruğa WM_QUIT mesajını bırakakmaktadır.

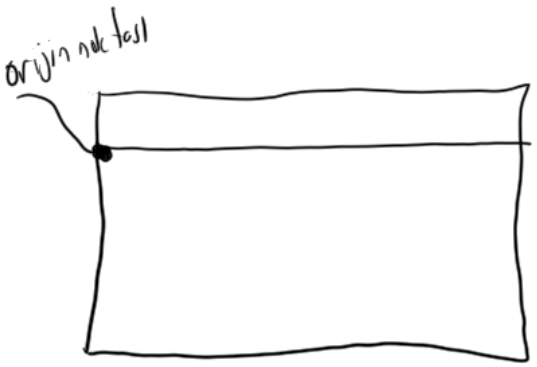
5) WM_QUIT mesajını alan GetMessage API fonksiyonu 0 ile geri döner ve mesaj döngüsünden çıkılır. Böylece WinMain sonlanır program da bitmiş olur.

1.4.4. Pencere Mesajlarının İşlenmesi

Daha önceden de belirttiğimiz gibi mesajlar tipik olarak pencere fonksiyonun içerisinde mesaj numarasının switch içerisine alınmasıyla işlenmektedir. Tabii okunabilirlik gereği switch deyiminin case bölümünde uzun işlemler yapılacaksa o işlemlerin bir fonksiyona havale edilmesi daha uygun olur. Böylece kod biraz daha okunabilir hale getirilebilir. Mesajların bu biçimde işlenmesi tipik olarak prosedürel programama tekniğini akla getirmektedir. Örneğin farenin sol tuşuna basıldığında bir MessageBox çıkartmak isteyelim:

```
void LButtonDownHandler(HWND hWnd, WPARAM wParam, LPARAM lParam)
{
    char buf[100];

    sprintf(buf, "X = %u, Y = %u", LOWORD(lParam), HIWORD(lParam));
    MessageBox(hWnd, buf, "Info", MB_OK);
}

LRESULT CALLBACK WndProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM lParam)
{
    switch (message) {
        case WM_LBUTTONDOWN:
            LButtonDownHandler(hWnd, wParam, lParam);
            break;

        case WM_DESTROY:
            PostQuitMessage(0);
            break;
        default:
            return DefWindowProc(hWnd, message, wParam, lParam);
    }
    return 0;
}
```

WM_LBUTTONDOWN mesajında lParam parametresinin düşük anlamlı WORD kısmı basım yerinin X koordinatını, yüksek anlamlı WORD kısmı basım yerinin y koordinatını belirtir. Buradaki değerler çalışma alanı orijinlidir. Yani orijin noktası pencere başlığının altındaki sınır çizgilerinin içerisindeki bölgenin sol üst köşesidir:



1.4.5. Alt Pencerelerin Yaratılması

Windows'ta tüm pencereler CreateWindow fonksiyonuyla yaratılırlar. Alt pencerelerin yaratımı sırasında CreateWindow fonksiyonunun hWndParent parametresi üst pencerenin handle değeri olarak girilir. Ayrıca fonksiyonun ikinci parametresinde WS_CHILD belirlemesinin yapılması gerekir. Alt pencerelerin için ayrı bir pencere sınıfı kullanılarak yaratılması daha uygundur. Bu durumda o alr pencereye gelen mesajlar ayrı bir pencere fonksiyonu tarafından işlenecektir. Alt pencerelerin yaratımı herhangi bir yerde yapılabilmektedir. Ancak en uygun yer üst pencerenin WM_CREATE mesajıdır. Bir pencere yaratılır yaratılmaz pencere henüz görünür halde değilken Windows onun üst penceresine WM_CREATE isimli bir mesajı gönderilmektedir.

Örnek bir alt pencere yaratımı şöyle yapılabilmektedir:

```
#include <windows.h>

LRESULT CALLBACK WndProcParent(HWND hWnd, UINT message, WPARAM wParam, LPARAM lParam);
LRESULT CALLBACK WndProcChild(HWND hWnd, UINT message, WPARAM wParam, LPARAM lParam);

HWND g_hWndParent;
HWND g_hWndChild;
HINSTANCE g_hInstance;

int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, LPSTR lpszCmdParam, int nCmdShow)
{
    WNDCLASS wndClassParent, wndClassChild;
    MSG message;

    if (!hPrevInstance) {
        wndClassParent.style = CS_HREDRAW | CS_VREDRAW;
        wndClassParent.cbClsExtra = 0;
        wndClassParent.cbWndExtra = 0;
        wndClassParent.hInstance = hInstance;
        wndClassParent.hIcon = LoadIcon(NULL, IDI_QUESTION);
        wndClassParent.hbrBackground = GetStockObject(WHITE_BRUSH);
        wndClassParent.hCursor = LoadCursor(NULL, IDC_ARROW);
        wndClassParent.lpszMenuName = NULL;
        wndClassParent.lpszClassName = "GenericParent";
        wndClassParent.lpfnWndProc = (WNDPROC)WndProcParent;
        if (!RegisterClass(&wndClassParent))
            return -1;

        wndClassChild.style = CS_HREDRAW | CS_VREDRAW;
        wndClassChild.cbClsExtra = 0;
        wndClassChild.cbWndExtra = 0;
        wndClassChild.hInstance = hInstance;
        wndClassChild.hIcon = LoadIcon(NULL, IDI_HAND);
        wndClassChild.hbrBackground = CreateSolidBrush(RGB(255, 0, 0));
        wndClassChild.hCursor = LoadCursor(NULL, IDC_CROSS);
        wndClassChild.lpszMenuName = NULL;
        wndClassChild.lpszClassName = "GenericChild";
        wndClassChild.lpfnWndProc = (WNDPROC)WndProcChild;
        if (!RegisterClass(&wndClassChild))
            return -1;
    }

    g_hInstance = hInstance;

    g_hWndParent = CreateWindow(GenericParent", "Sample Windows", WS_OVERLAPPEDWINDOW, CW_USEDEFAULT,
0,
        CW_USEDEFAULT, 0, NULL, NULL, hInstance, NULL);
    if (!g_hWndParent)
        return -1;
    ShowWindow(g_hWndParent, SW_RESTORE);
    UpdateWindow(g_hWndParent);

    while (GetMessage(&message, 0, 0, 0)) {
        TranslateMessage(&message);
```

```
        DispatchMessage(&message);
    }
    return message.wParam;
}

LRESULT CALLBACK WndProcParent(HWND hWnd, UINT message, WPARAM wParam, LPARAM lParam)
{
    switch (message) {
        case WM_CREATE:
            g_hWndChild = CreateWindow("GenericChild", "", WS_CHILD | WS_VISIBLE | WS_BORDER, 10, 10,
                100, 100, hWnd, NULL, g_hInstance, NULL);
            if (!g_hWndChild) {
                MessageBox(hWnd, "Cannot create child!", "Error", MB_OK);
                return -1;
            }
            break;
        case WM_LBUTTONDOWN:
            MessageBox(hWnd, "Parent WM_LBUTTONDOWN", "Message", MB_OK);
            break;
        case WM_DESTROY:
            PostQuitMessage(0);
            break;
        default:
            return DefWindowProc(hWnd, message, wParam, lParam);
    }
    return 0;
}

LRESULT CALLBACK WndProcChild(HWND hWnd, UINT message, WPARAM wParam, LPARAM lParam)
{
    switch (message) {
        case WM_LBUTTONDOWN:
            MessageBox(hWnd, "Child WM_LBUTTONDOWN", "Message", MB_OK);
            break;
        default:
            return DefWindowProc(hWnd, message, wParam, lParam);
    }
    return 0;
}
```

Alt pencerenin ayrı pencere sınıfı ile yaratıldığına ve dolayısıyla alt pencerenin ayrı pencere fonksiyonuna sahip olduğuna dikkat ediniz. Alt pencereyi yaratırken CreateWindow fonksiyonunun ikinci parametresi WS_CHILD|WS_VISIBLE|WS_BORDER biçiminde girildiğine dikkat ediniz. WS_VISIBLE yaratılan alt pencerenin yaratılır yaratılmaz görünür hale getirileceğini belirtmektedir. (Eğer yaratım sırasında WS_CHILD bayrağı kullanılmazsa alt pencerenin ShowWindow ile görünür hale getirilmesi gerekir). WS_BORDER ise yaratılan alt pencerenin sınır çizgilerine sahip olacağını belirtmektedir.

1.4.6. Pencerelere Mesaj Gönderilmesi

Yalnızca işletim sisteminin ya daa GUI alt sistemin kendisi değil, programcılar da isterlerse pencerelere mesajlar gönderebilirler. Mesaj göndermenin iki yolu vardır: SendMessage API fonksiyonu ile ya da PostMessage API fonksiyonu ile. (Windows dokümanlarında açıklamaklar yapılırken “sends” sözcüğü görüldüğünde mesajın SendMessage fonksiyonu ile “posts” sözcüğü görüldüğünde ise PostMessage ile gönderildiği anlaşılmalıdır.) SendMessage ve PostMesssage fonksiyonlarının çok benzerdir:

```
LRESULT SendMessage(HWND hWnd, UINT Msg, WPARAM wParam, LPARAM lParam);
BOOL PostMessage(HWND hWnd, UINT Msg, WPARAM wParam, LPARAM lParam);
```

SendMessage doğrudan pencere fonksiyonunu çağırarak mesajı işler. PostMessage ise mesajı mesaj kuyruğuna bırakır. SendMessage doğrudan pencere fonksiyonunun çağırdığı için akış SendMessage fonksiyonundan geri döndüğünde pencere fonksiyonu çalıştırımış ve mesaj işlenmiş olur. Örneğin:

```
SendMessage(hWnd, WM_USER, 0, 0);
...
```

Burada SendMessage fonksiyonu geri döndüğünde mesaj işlenmiş durumdadır. Halbuki PostMessage fonksiyonu mesajı ilgili pencereyi yaratan thread'in mesaj kuyruğuna bırakmaktadır:

```
PostMessage(hWnd, WM_USER, 0, 0);
```

Burada akış PostMessage fonksiyonundan çıktığında henüz mesaj işlenmiş değildir. Messaj hangi pencereye gönderilmişse ilgili mesaj döngüsü yoluyla mesaj kuyruktan GetMessage fonksiyonuyla alınıp "dispatch" edilerek işlenecektir. Dolayısıyla SendMessage mesajın "senkron" olarak, PostMessage ise "asenkron" olarak işlenmesine yol açar. (Örneğin CreateWindow fonksiyonu WM_CREATE mesajını, DestroyWindow ise WM_DESTROY mesajlarını SendMessage yoluyla göndermektedir.)

1.4.7. Standart Alt Pencerelerin Kullanılması

Windows'ta görsel arayüzü oluşturmak için sistem tarafından düğme gibi, edit alanı gibi, listeleme kutuları gibi çeşitli alt pencere sınıfları oluşturulup register ettirilmiştir. Bu pencere sınıflarına ilişkin pencere fonksiyonları Windows'un USER32.DLL dosyasının içerisindedir. Böylece programcı bu görsel elemanlara verilen sınıf isimlerini kullanarak bu standart alt pencereleri CreateWindow fonksiyonuyla yaratabilir. Aslında düğmeler, edit alanları, listeleme kutuları gibi görsel öğeler boş bir pencere üzerinde yapılan çizim işlemleriyle ve bazı mesajların işlenmesiyle gerçekleştirilmişlerdir. Örneğin "button" sınıfı normal düğmeleri (push button), radyo düğmelerini (radio buttons) ve seçenek kutularını (check boxes) yaratmak için, "edit" sınıfı edit alanı (edit box) yaratmak için, "static" sınıfı yalnızca yazı bulunan yazı penceresi (label) yaratmak için kullanılabilir. Örnek bir standart alt pencere uygulaması şöyle verilebilir:

```
#include <windows.h>

#define ID_BUTTONOK        100
#define ID_BUTTONCANCEL    101
#define ID_EDITNAME        102
#define ID_EDITNO          103

LRESULT CALLBACK WndProcParent(HWND hWnd, UINT message, WPARAM wParam, LPARAM lParam);
LRESULT CALLBACK WndProcChild(HWND hWnd, UINT message, WPARAM wParam, LPARAM lParam);

HWND g_hWndParent;
HWND g_hButtonOk;
HWND g_hButtonCancel;
HWND g_hEditBoxName;
HWND g_hEditBoxNo;
HWND g_hStaticName;
HWND g_hStaticNo;
HINSTANCE g_hInstance;

int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, LPSTR lpszCmdParam, int nCmdShow)
{
    WNDCLASS wndClassParent, wndClassChild;
    MSG message;

    if (!hPrevInstance) {
        wndClassParent.style = CS_HREDRAW | CS_VREDRAW;
        wndClassParent.cbClsExtra = 0;
        wndClassParent.cbWndExtra = 0;
        wndClassParent.hInstance = hInstance;
        wndClassParent.hIcon = LoadIcon(NULL, IDI_QUESTION);
        wndClassParent.hbrBackground = GetStockObject(WHITE_BRUSH);
        wndClassParent.hCursor = LoadCursor(NULL, IDC_ARROW);
        wndClassParent.lpszMenuName = NULL;
        wndClassParent.lpszClassName = "GenericParent";
```

```
        wndClassParent.lpfnWndProc = (WNDPROC)WndProcParent;
        if (!RegisterClass(&wndClassParent))
            return -1;
    }

    g_hInstance = hInstance;

    g_hWndParent = CreateWindow("GenericParent", "Sample Windows", WS_OVERLAPPEDWINDOW, CW_USEDEFAULT,
0,
        450, 200, NULL, NULL, hInstance, NULL);
    if (!g_hWndParent)
        return -1;
    ShowWindow(g_hWndParent, SW_RESTORE);
    UpdateWindow(g_hWndParent);

    while (GetMessage(&message, 0, 0, 0)) {
        TranslateMessage(&message);
        DispatchMessage(&message);
    }
    return message.wParam;
}

LRESULT CALLBACK WndProcParent(HWND hWnd, UINT message, WPARAM wParam, LPARAM lParam)
{
    switch (message) {
        case WM_CREATE:
            g_hButtonOk = CreateWindow("button", "Ok", WS_CHILD | WS_BORDER | WS_VISIBLE |
BS_PUSHBUTTON , 260, 120,
                70, 25, hWnd, (LPVOID)ID_BUTTONOK, g_hInstance, NULL);
            g_hButtonCancel = CreateWindow("button", "Cancel", WS_CHILD | WS_BORDER | WS_VISIBLE |
BS_PUSHBUTTON, 340, 120,
                70, 25, hWnd, NULL, g_hInstance, NULL);
            g_hStaticName = CreateWindow("static", "Adı Soyadı", WS_CHILD | WS_VISIBLE, 10, 10,
                100, 20, hWnd, (LPVOID)ID_BUTTONCANCEL, g_hInstance, NULL);
            g_hStaticNo = CreateWindow("static", "No", WS_CHILD | WS_VISIBLE, 10, 60,
                100, 20, hWnd, NULL, g_hInstance, NULL);
            g_hEditBoxName = CreateWindow("edit", "", WS_CHILD | WS_BORDER | WS_VISIBLE, 10, 30,
                200, 20, hWnd, (LPVOID)ID_EDITNAME, g_hInstance, NULL);
            g_hEditBoxNo = CreateWindow("edit", "", WS_CHILD | WS_BORDER | WS_VISIBLE, 10, 80,
                200, 20, hWnd, (LPVOID)ID_EDITNO, g_hInstance, NULL);
            break;
        case WM_LBUTTONDOWN:
            MessageBox(hWnd, "Parent WM_LBUTTONDOWN", "Message", MB_OK);
            break;
        case WM_DESTROY:
            PostQuitMessage(0);
            break;
        default:
            return DefWindowProc(hWnd, message, wParam, lParam);
    }
    return 0;
}
```

1.4.8. Alt Pencere Mesajlarının İşlenmesi

Windows'ta bir ister ana pencere olsun ister alt pencere olsun bir thread'in yarattığı tüm pencerelere gönderilen ("post" edilen) mesajlar aynı mesaj kuyruğuna bırakılırlar ve aynı mesaj döngüsü tarafından ele alınırlar. Ayrıca belli bir girdi olayı için mesajlar o girdi olayına ilişkin olan tek bir pencereye gönderilmektedir. Örneğin bir alt pencere üzerinde fareye tıkladığımızda fare mesajları yalnızca bu alt pencereye yollanacaktır.

Pencere fonksiyonlarının pencere sınıflarında belirlendiğini anımsayınız. Bu durumda eğer biz yarattığımız bir alt pencereye ilişkin alt pencere sınıfını kendimiz oluşturmuşsak -onun pencere fonksiyonunu da kendimiz yazmış olacağımızdan- o alt pencereye gönderilen mesajları doğrudan işleyebiliriz. Pekiyi biz eğer Windows'un standart bir alt pencere sınıfını (ya da başkalarının oluşturup register ettirdiği) kullandığımızda o

pencereye gönderilen mesajları nasıl işleyebiliriz? Çünkü bu durumda o pencerenin pencere fonksiyonu bizim elimizde olmayacaktır. İşte standart pencereleri oluşturan Microsoft o pencerelerde bazı eylemler gerçekleştiğinde SendMessage yoluyla o standart pencerenin üst penceresine WM_COMMAND isimli bir mesaj göndermektedir. Böylece biz üst pencerede WM_COMMAND mesajı yoluyla alt penceredeki bazı mesajları alıp işleyebilmekteyiz.

WM_COMMAND mesajında HIWORD(wParam) alt pencereden gelen mesajın türünü, LOWORD(wParam) ise alt pencere "id değeri"ni belirtmektedir. Windows'ta handle değerlerinin yanı sıra alt pencerelerin bir de "id değerleri" vardır. Alt pencere id değerleri alt pencereleri birbirlerinden ayırmak için kullanılır. Bu nedenle bunlar sistem genelinde tek değildir, kardeş pencereler temelinde tektir. Alt pencerelerin id değerleri CreateWindow fonksiyonunda HMENU parametresi yoluyla girilir. Bunların dışında WM_COMMAND mesajındaki lParam parametresi de alt pencerenin handle değerini bulundurmaktadır. Aşağıda alt pencere mesajlarının işlenmesine yönelik bir örnek göreceksiniz.

```
#include <stdio.h>
#include <windows.h>

#define ID_BUTTONOK        100
#define ID_BUTTONCANCEL    101
#define ID_EDITNAME        102
#define ID_EDITNO          103

LRESULT CALLBACK WndProcParent(HWND hWnd, UINT message, WPARAM wParam, LPARAM lParam);
LRESULT CALLBACK WndProcChild(HWND hWnd, UINT message, WPARAM wParam, LPARAM lParam);

HWND g_hWndParent;
HWND g_hButtonOk;
HWND g_hButtonCancel;
HWND g_hEditBoxName;
HWND g_hEditBoxNo;
HWND g_hStaticName;
HWND g_hStaticNo;
HINSTANCE g_hInstance;

int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, LPSTR lpszCmdParam, int nCmdShow)
{
    WNDCLASS wndClassParent;
    MSG message;

    if (!hPrevInstance) {
        wndClassParent.style = CS_HREDRAW | CS_VREDRAW;
        wndClassParent.cbClsExtra = 0;
        wndClassParent.cbWndExtra = 0;
        wndClassParent.hInstance = hInstance;
        wndClassParent.hIcon = LoadIcon(NULL, IDI_QUESTION);
        wndClassParent.hbrBackground = GetStockObject(WHITE_BRUSH);
        wndClassParent.hCursor = LoadCursor(NULL, IDC_ARROW);
        wndClassParent.lpszMenuName = NULL;
        wndClassParent.lpszClassName = "GenericParent";
        wndClassParent.lpfnWndProc = (WNDPROC)WndProcParent;
        if (!RegisterClass(&wndClassParent))
            return -1;
    }

    g_hInstance = hInstance;

    g_hWndParent = CreateWindow("GenericParent", "Sample Windows", WS_OVERLAPPEDWINDOW, CW_USEDEFAULT,
0,
        450, 200, NULL, NULL, hInstance, NULL);
    if (!g_hWndParent)
        return -1;
    ShowWindow(g_hWndParent, SW_RESTORE);
    UpdateWindow(g_hWndParent);
```

```
    while (GetMessage(&message, 0, 0, 0)) {
        TranslateMessage(&message);
        DispatchMessage(&message);
    }
    return message.wParam;
}

LRESULT CALLBACK WndProcParent(HWND hWnd, UINT message, WPARAM wParam, LPARAM lParam)
{
    switch (message) {
        case WM_CREATE:
            g_hButtonOk   =   CreateWindow("button",   "Ok",   WS_CHILD   |   WS_BORDER   |   WS_VISIBLE   |
BS_PUSHBUTTON , 260, 120,
                70, 25, hWnd, (LPVOID)ID_BUTTONOK, g_hInstance, NULL);
            g_hButtonCancel  =  CreateWindow("button",  "Cancel",  WS_CHILD  |  WS_BORDER  |  WS_VISIBLE  |
BS_PUSHBUTTON, 340, 120,
                70, 25, hWnd, (LPVOID)ID_BUTTONCANCEL, g_hInstance, NULL);
            g_hStaticName = CreateWindow("static", "Adı Soyadı", WS_CHILD | WS_VISIBLE, 10, 10,
                100, 20, hWnd, NULL, g_hInstance, NULL);
            g_hStaticNo = CreateWindow("static", "No", WS_CHILD | WS_VISIBLE, 10, 60,
                100, 20, hWnd, NULL, g_hInstance, NULL);
            g_hEditBoxName = CreateWindow("edit", "", WS_CHILD | WS_BORDER | WS_VISIBLE, 10, 30,
                200, 20, hWnd, (LPVOID)ID_EDITNAME, g_hInstance, NULL);
            g_hEditBoxNo = CreateWindow("edit", "", WS_CHILD | WS_BORDER | WS_VISIBLE, 10, 80,
                200, 20, hWnd, (LPVOID)ID_EDITNO, g_hInstance, NULL);
            break;
        case WM_COMMAND:
            switch (LOWORD(wParam)) {
                case ID_BUTTONOK:
                    if (HIWORD(wParam) == BN_CLICKED) {
                        char name[120];
                        char no[120];
                        char text[240];
                        GetWindowText(g_hEditBoxName, name, 120);
                        GetWindowText(g_hEditBoxNo, no, 120);
                        sprintf(text, "Adı Soyadı: %s, No: %s", name, no);

                        MessageBox(hWnd, text, "Mesaj", MB_OK);
                    }
                    break;

                case ID_BUTTONCANCEL:
                    if (HIWORD(wParam) == BN_CLICKED) {
                        MessageBox(hWnd, "Cancel tuşuna tıklandı!", "Mesaj", MB_OK);
                    }
                    break;
            }
            break;

        case WM_LBUTTONDOWN:
            MessageBox(hWnd, "Parent WM_LBUTTONDOWN", "Message", MB_OK);
            break;
        case WM_DESTROY:
            PostQuitMessage(0);
            break;
        default:
            return DefWindowProc(hWnd, message, wParam, lParam);
    }
    return 0;
}
```

1.5. XWindow (X11) Sistemlerinde GUI Programlama Modelinin Temelleri

Bu bölümde XWindow sistemlerindeki GUI programlama modelinin ayrıntılarına Windows'taki kadar girmeyeceğiz. Yalnızca bir "Merhaba Dünya" programı eşliğinde bazı temel açıklamaları yapmakla yetineceğiz. Önceki bölümlerde de belirttiğimiz gibi bu sistemlerde çeşitli düzeylerde uygulama yapabilmek için kütüphaneler ve ortamlar (frameworks) bulunmaktadır. Ancak XWindow sistemlerinin aşağı seviyeli çalışmasını en iyi betimleyen kütüphaneler XLIB ve XCB kütüphaneleridir. Biz burada yalnızca temel bir XLIB örneği vereceğiz.

1.5.1. XWindow Sistemleri İçin XLIB Fonksiyonlarıyla "Merhaba Dünya" Programı

Daha önceden de belirttiğimiz gibi XWindow (X11) sistemleri client-server biçimde çalışmaktadır. Burada biz bu sistemlerde boş bir pencere çıkartan örnek bir program vereceğiz. XLIB seviye olarak yukarıda gördüğümüz Windows API sistemine göre biraz daha aşağı seviyeli gibi durmaktadır. XLIB'i kullanan GTK+ kütüphanesinin seviye olarak Windows API fonksiyonlarına daha çok benzediğini söyleyebiliriz. Aşağıda ekrana bir ana pencere çıkartan örnek bir XLIB programı görüyorsunuz:

```
/* xlib-helloworld.c */

#include <X11/Xlib.h>
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    Display *disp;
    Window w;
    XEvent e;
    int scr;

    disp = XOpenDisplay(NULL);
    if (disp == NULL) {
        fprintf(stderr, "Cannot open display\n");
        exit(1);
    }

    scr = DefaultScreen(disp);
    w = XCreateSimpleWindow(disp, RootWindow(disp, scr), 10, 10, 100, 100, 1,
            BlackPixel(disp, scr), WhitePixel(disp, scr));
    XSelectInput(disp, w, ExposureMask | KeyPressMask);
    XMapWindow(disp, w);

    for (;;) {
        XNextEvent(disp, &e);
        if (e.type == KeyPress)
            break;
    }

    XCloseDisplay(disp);

    return 0;
}
```

Merhaba dünya programındaki XOpenDisplay fonksiyonu XWindow sunucusu ile bağlantı yapmak için kullanılmaktadır. Bu fonksiyon başarı durumunda bize bir "Display handle" verir. Daha sonra biz bu handle'ı vererek bir ekran (screen) elde ederiz. Bu işlem DeafultScreen fonksiyonuyla yapılmaktadır. Örnek programımızda daha sonra ana pencere XCreateSimpleWindow fonksiyonuyla yaratılmıştır. Bu fonksiyon bize yaratılan pencereye ilişkin bir handle değerini Window * türü olarak vermektedir. Programda daha sonra mesaj döngüsüne girmeden önce hangi girdi olaylarının izleneceğini belirlemek için XSelectInput fonksiyonu çağrılmıştır. Mesaj döngüsünden sıradaki mesaj XNextEvent fonksiyonuyla elde edilmektedir. Bu fonksiyon bize kuyruktaki mesajı XEvent isimli bir yapı olarak verir. Örnek programımızda bir tuşa basıldığında mesaj döngüsünden çıkılmaktadır. Mesaj döngüsünden çıkıldığında XCloseDisplay fonksiyonu ile daha önce alınmış

olan ekran geri bırakılmıştır. Tabii ekran yok edildiğinde tüm pencereler de yok edilecektir. Ayrıca program sonlandığında X11 sistemi ile bağlantı da otomatik koparılmaktadır.

Bu bölümde XLIB programlamasına ilişkin başka ayrıntı verilmeyecektir. XLIB ile ilgili çeşitli kitaplardan ve dokümanlardan bilginizi ilerletilebilirsiniz. XLIB programları derlenirken libX11 kütüphanesinin link aşamasına dahil etmeyi unutmayınız. Derleme işlemi aşağıdaki gibi yapılabilir:

```
gcc -o xlib-helloworld xlib-helloworld.c -lX11
```

2. Proseslerin Heap Alanları ve Tahsisat Algoritmaları

Programlama dillerinde programın çalışma zamanı sırasında bellekte tahsisat yapan çeşitli fonksiyonlar bulunabilmektedir. Örneğin C'deki malloc, calloc, realloc ve free fonksiyonları, C++'taki operator new ve operator delete fonksiyonları, C# ve Java'daki new operatörleri programın çalışma zamanı sırasında tahsisat yapma amacıyla kullanılırlar. Genel olarak programlama dillerinde dinamik bellek tahsisatları için kullanılan potansiyel bellek alanına "heap" denilmektedir. Tabii "heap" genel bir kavramdır. Herhangi bir sistem belli bir bellek aalanını heap olarak olarak belirleyip bir veri yapısı eşliğinde orayı dinamik tahsisatlar için kullanabilir. Bu bölümde dinamik tahsisatların gerçekleştirilmesinde kullanılan veri yapıları ve algoritmalar üzerinde durulacaktır. Yani bu bölümde, "tahsisat algoritmaları bir yerin tahsis edilip edilmediğini nasıl belirlemektedir? Alan serbest bırakıldığında aslında neler yapılmaktadır? Tahsisat sistemlerinin performanslarına neler etki etmektedir?" gibi soruların yanıtları uygulamalı olarak ele alınacaktır.

Tahsisat algoritmalarının işletim sistemlerinin gerçekleştirilmesinde de önemli bir yeri vardır. Çünkü işletim sistemleri de çeşitli çekirdek (kernel) nesnelerini kendi oluşturdukları bir heap sistemi içerisinde tahsis ederler. Örneğin bir proses yaratıldığında Linux'ta proses kontrol bloğunu temsil eden bir task_struct yapısı oluşturulmaktadır. İşte bu task_struct yapısı çekirdeğin kendi heap alanından "dilimli tahsisat sistemi (slab allocator)" denilen bir dinamik tahsisat yöntemiyle tahsis edilmektedir. Benzer biçimde Linux'ta her dosya açıldığında çekirdek alanında dosya nesnesini temsil file isminde yapı tahsis edilir. Bu tahsisat da yine çekirdeğin heap alanında yapılmaktadır. Bunun gibi işletim sisteminin çekirdeğinde onlarca türden nesneler gerektiğinde tahsis edilip geri bırakılmaktadır.

2.1. Proseslerin Heap Alanları

Modern ve kapasiteli işletim sistemlerinde heap alanı genellikle proses temelinde oluşturulmaktadır. Yani klasik masaüstü ve mobil işletim sistemlerinde her prosesin ayrı bir heap alanı vardır. Proses yaratıldığında işletim sistemi proesin bellek alanında o proses için bir heap alanı da oluşturur. Proses sonlandığında prosesin bellek alanı boşaltılırken heap alanı da yok edilmektedir. Windows sistemlerinde, UNIX/Linux sistemlerinde ve Mac OS X sistemlerinde, Android ve IOS gibi mobil işletim sistemlerindeki heap kullanımı hep böyledir. Proseslerin heap alanlarının yönetilmesi pek çok sistemde çekirdeğin bir fonksiyonu değildir. Bu yönetim kullanıcı seviyesindeki (user level) kodlarla yapılmaktadır. Bu organizasyonları malloc, calloc, realloc ve free gibi kullanıcı düzeyindeki (user level) kütüphane fonksiyonları yapmaktadır.

Genel olarak bir proses yaratıldığında prosesin potansiyel kullanımına ayrılan belleğe "prosesin adres alanı (process address space)" denilmektedir. Prosesin adres alanının önemli bir kısmı çalıştırılabilen (executable) dosyadan alınarak oluşturulmaktadır. Örneğin Windows'un PE (Portable Executable) formatında ve UNIX/Linux sistemlerinde kullanılan ELF (Executable and Linkable Format) formatında çalıştırılabilen dosyalar bölümlerden (sections) oluşur. Programın tüm makine kodları çalıştırılabilen dosyanın "kod bölümünde", tüm global değişkenler "data" ve "bss" bölümlerinde bulunular. İşletim sisteminin yükleyicisi program için bir de stack alanı tahsis eder. İşte bu alanların dışında genellikle işletim sistemlerinde adres prosesin adres alanının geri kalan kısmı "heap" olarak düzenlenmektedir.

Yukarıdaki şekil kavramsal olarak çizilmiştir. Belli bir işletim sisteminde bu konuya ilişkin çeşitli ayrıntılar bulunmaktadır. Genel olarak heap alanının yerinin ve uzunluğunun sistemden sisteme değişebildiğini söyleyebiliriz.

Heap ile ilgili sıkça sorulan sorular şunlardır:

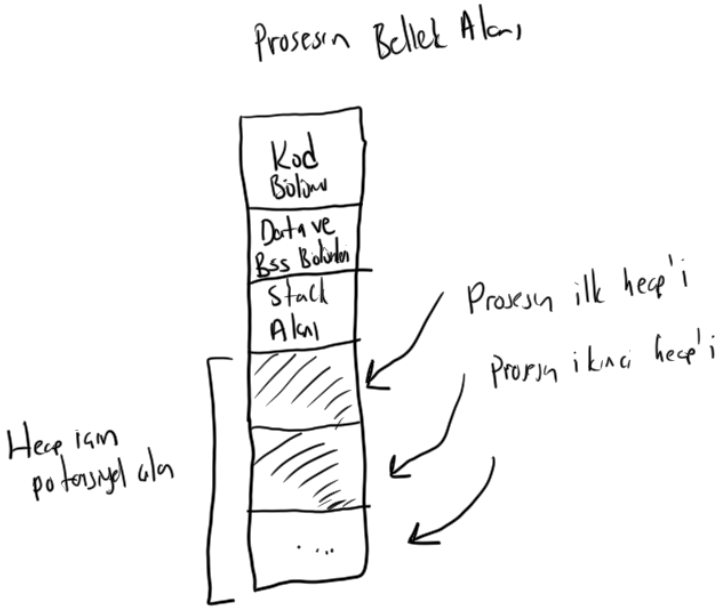
Soru: Heap alanı nerede oluşturulmaktadır?
Yanıt: Heap alanı prosesin adres alanı içerisinde oluşturulur. Yeri sistemden sisteme değişebilmektedir. Genellikle prosesin adres alanındaki geri kalan türm bölgeler heap olarak kullanılmaya uygundur.

Soru: Heap alanın büyüklüğü ne kadardır?
Yanıt: Heap alanının büyüklüğü sistemden sisteme değişebilmektedir. Bu büyüklük tabii prosesin o sistemdeki adres alanıyla da ilgilidir. Örneğin 32 bit Windows sistemlerinde prosesler zaten en fazla 2GB bellek alanına sahip olabilirler. Bu 2 GB'nin içerisinde ".text", ".data", ".bss" ve stack alanları da dahildir. O halde bu sistemlerde heap zaten en fazla 2 GB olabilir.

Soru: Heap alanı ortak bir alan mıdır, prosese özgü müdür?
Yanıt: Bu durum sistemden sisteme değişebilir. Ancak yaygın sistemlerin hemen hepsinde heap alanı prosesin adres alanı içerisinde proses özgü bir biçimde oluşturulmaktadır. Dolayısıyla proses sonlandığında o prosesin heap alanı da yok edilmektedir. Heap alanı bu sistemlerde thread'ler atarfından ortak erişilen bir alandır. Yani her thread'in ayrı bir heap alanı yoktur.

2.2. Windows Sistemlerinde Heap Organizasyonu

Windows sistemlerinde prosesler tek bir heap alanına sahip olmak zorunda değildir. Proses kendisi için birden fazla heap alanı yaratabilir.



Heap alanlarının yaratılması ve o alanlar üzerinde tahsisatların yapılabilmesi için HeapXXX biçiminde isimlendirilmiş API fonksiyonları bulunmaktadır. <bunların prototiplerini aşağıda görüyorsunuz:

```
HANDLE WINAPI HeapCreate(
  __in DWORD flOptions,
  __in SIZE_T dwInitialSize,
  __in SIZE_T dwMaximumSize
);

BOOL WINAPI HeapDestroy(
  __in HANDLE hHeap
);

LPVOID WINAPI HeapAlloc(
  __in HANDLE hHeap,
  __in DWORD dwFlags,
  __in SIZE_T dwBytes
);

LPVOID WINAPI HeapReAlloc(
  __in HANDLE hHeap,
  __in DWORD dwFlags,
  __in LPVOID lpMem,
  __in SIZE_T dwBytes
);

BOOL WINAPI HeapFree(
  __in HANDLE hHeap,
  __in DWORD dwFlags,
  __in LPVOID lpMem
);
```

HeapCreate fonksiyonu belli uzunlukta yeni bir heap alanı yaratır ve yaratılan heap alanının handle değeri ile geri döner. Yaratılan bir heap HeapDestroy fonksiyonuyla yok edilebilir. HeapAlloc fonksiyonu belli bir heap'ten tahsisat yapmaktadır. HeapReAlloc tahsis edilen alanın büyütülmesi ya da küçültülmesi için kullanılır. HeapFree ise alanı serbest bırakır.

Windows'ta bir proses yaratılırken işletim sistemi o proses için başlangıçta bir heap de yaratmaktadır. Buna prosesin default heap'i denir. Prosesin default heap'inin uzunluğu çalıştırılabilen dosyanın (PE dosyasının)

içerisine bağlayıcı (linker) tarafından yazılmaktadır. Bu uzunluk bağlayıcı ayarları ile belirlenebilir. Fakat Microsoft bağlayıcıları aksi belirtilmediği durumda bu uzunluğu default 1 MB olarak almaktadır. Prosesin default heap'ine ilişkin handle değeri GetProcessHeap API fonksiyonuyla elde edebilir:

```
HANDLE WINAPI GetProcessHeap(void);
```

Windows'ta örnek bir heap yaratarak ondan tahsisat yapan yalın bir program şöyle yazılabilir:

```
#include <stdio.h>
#include <stdlib.h>
#include <windows.h>


int main(void)
{
    HANDLE hHeap;
    char *str;

    if ((hHeap = HeapCreate(0, 0x100000, 0x100000)) == NULL) {
        fprintf(stderr, "cannot create heap!..\n");
        exit(EXIT_FAILURE);
    }

    if ((str = (char *)HeapAlloc(hHeap, 0, 100)) == NULL) {
        fprintf(stderr, "cannot allocate memory!..\n");
        exit(EXIT_FAILURE);
    }

    strcpy(str, "This is a test");
    puts(str);

    HeapFree(hHeap, 0, str);

    HeapDestroy(hHeap);

    return 0;
}
```

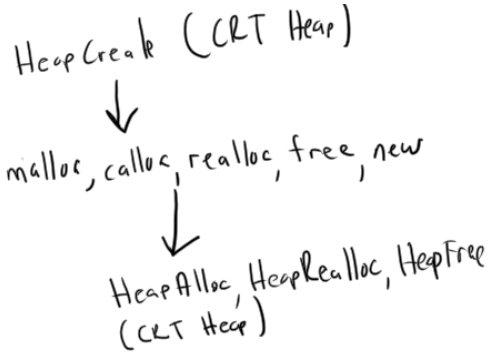
Bu örnekte proses için 1 MB uzunluğunda bir heap yaratılmıştır, sonra o heap'ten tahsisat yapılmıştır. Proses bittiğinde prosesin tüm heap alanları zaten yok edilmektedir.

Prosesin default heap'i proses yaratıldığında otomatik oluşturulduğu için hemen tahsisat işleminde kullanılabilir. Örneğin:

```
#include <stdio.h>
#include <stdlib.h>
#include <windows.h>

int main(void)
{
    HANDLE hHeap;
    char *str;

    if ((hHeap = GetProcessHeap()) == NULL) {
        fprintf(stderr, "cannot get default heap handle!..\n");
        exit(EXIT_FAILURE);
    }

    if ((str = (char *)HeapAlloc(hHeap, 0, 100)) == NULL) {
        fprintf(stderr, "cannot allocate memory!..\n");
        exit(EXIT_FAILURE);
    }
```

```
    strcpy(str, "This is a test");
    puts(str);

    HeapFree(hHeap, 0, str);

    return 0;
}
```

Pekiyi Windows sistemlerinde bizim C'de kullandığımız malloc, calloc, realloc ve free fonkisyonları, C++'taki operator new ve operator delete fonksiyonları arka planda hangi heap'ten tahsisat yapmaktadır? İşte akış main fonksiyonuna gelmeden önce derleyicinin yerleştirdiği başlangıç kodu (start up module) yoluyla ismine "CRT (C Runtime) Heap" denilen bir heap yaratılmaktadır. malloc, calloc ve realloc gibi standart C fonksiyonları bu heap'ten tahsisat yapmaktadır.



Pekiyi Windows sistemlerinde prosesin birden fazla heap'e sahip olmasının anlamı nedir? Heap alanında çok fazla tahsisat ve boşaltım yapıldığında "bölünme durumu (fragmentation)" oluşabilir. Bölünme ardışıl bellek miktarının azalmasına dolayısıyla da heap verimininin düşmesine yol açan bir olgudur. İşte birden fazla heap kullanımında bir heap'te bölünme olsa bile bu bölünme diğer heap'leri etkilememektedir. Yani heap'lerin birbirlerinden ayrılması bölünme olgusunu azaltıcı bir unsur oluşturabilmektedir.

Windows'ta Heap Kullanımına ilişkin sıkça sorulan sorular şunlar olabilir:

Soru: Windows'ta bir prosesin kaç heap'i vardır?
Yanıt: Windows'ta proses başlatıldığında default bir heap yaratılır. Ancak programcı isterse birden fazla heap yaratıp tahsisatlarını onların herhangi birinden yapabilir.

Soru: Windows'ta Microsoft'un C kütüphanesindeki malloc, calloc, realloc ve free fonksiyonları prosesin hangi heap'inden tahsisat yapmaktadır?
Yanıt: Bu fonksiyonlar derleyicilerin başlangıç kodları (start up code) tarafındna yaratılmış olan bir heap'ten (buna CRT heap deniyor) tahsisat yaparlar. (Tabii prosesin default heap'i de bu amaçla kullanılabilirdi. Ama Microsoft bu fonksiyonlar için ayrı bir heap yaratmayı uygun görmüştür.)

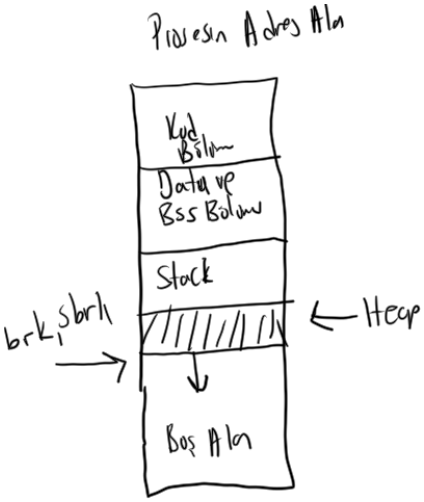
Soru: Prosesin default heap'inin uzunluğı nedir?
Yanıt: Bu uzunluk bağlayıcı ayarlarından değiştirilebilmekle birlikte bağlayıcı için default değer 1 MB'dir.

Soru: HeapCreate ile belli uzunlukta heap yaratmış olalım. Bu heap otomatik büyütülebilir mi?
Yanıt: Evet böyle bir olanak vardır. HeapCreate fonksiyonuyla heap alanı otomatik büyütülecek biçimde yaratılabilir. Örneğin prosesin default heap'i de CRT heap de böyle yaratılmışlardır.

2.2. UNIX/Linux Sistemlerindeki Heap Organizasyonu

UNIX/Linux sistemlerinde prosesin tek bir heap alanı vardır. Prosesin adres alanı başlangıçta çalıştırılabilen dosyanın içerisindeki bilgilerden oluşturulur. Fakat brk ve sbrk fonksiyonlarıyla prosesin adres alanı

büyütülebilmektedir. İşte malloc, calloc, realloc fonksiyonları heap alanı yetmediğinde bu brk ve sbrk fonksiyonlarını kullanarak prosesin adres alanını büyütürler.



2.3. Tahsisat Algoritmaları

Belli bir alanın heap olarak kullanılabilmesi için o alanın bir veri yapısı ile organize edilmesi gerekir. Örneğin malloc gibi bir fonksiyon heap'te bir alanı tahsis ettiğinde o alanın tahsis edilmiş olduğunu bir biçimde bilmektedir. Böylece yeni bir malloc çağrısı bize aynı alanı vermez. free fonksiyonu da alanı iade ettiğinde artık o alan boş olarak işaretlenmektedir. Pekiyi belli bir alan üzerinde tahsisat yapıp onları geri bırakan bir veri yapısı nasıl oluşturulmaktadır?..

Tahsisat sistemlerinde kullanılan veri yapıları ve algoritmaların bilinmesinin faydaları şunlar olabilir:

- Belli bir sistem için biz bir heap sistemini oluşturmak isteyebiliriz. Bu durumda bu sistemlerin nasıl oluşturulduğunu bilmemiz gerekir. Örneğin bir işletim sistemi yazacak olalım. İşletim sisteminin çekirdeğinin kullandığı bir heap sisteminin olması gerekir değil mi? Bu durumda işletim sistemini gerçekleştiren kişilerin bu tahsisat sistemlerini biliyor olmaları gerekir. Benzer biçimde biz heap sistemi olmayan küçük bir mikrodenetleyicide çalışıyor olabiliriz. Onun için bir heap sistemi oluşturmak isteyebiliriz.

- Tahsisat sistemlerinde kullanılan veri yapılarının ve algoritmaların bilinmesi bu sistemlerin daha iyi analiz edilmesini ve değerlendirilmesini sağlayabilir. Bu bilgi bizim bilinç düzeyimizi yükseltebilir. Bazı olayları daha iyi anlamamız için katkı oluşturabilir.

İyi bir tahsisat sisteminin performans ölçütleri ne olabilir? Yani örneğin iki ayrı sistemde iki malloc, realloc ve free fonksiyonları bulunuyor olsun. Bunlardan hangisinin daha iyi olduğuna yönelik ölçütler nelerdir? İşte tahsisat sistemlerinin performans ölçütlerini üç maddede özetleyebiliriz:

1) Hız en önemli ölçütlerden biridir. Örneğin iki farklı malloc gerçekleştirimi olsun. Bunlardan hangisi daha hızlı bize bir boş alan vermektedir? İki free fonklsiyonundan hangisi alanı daha çabuk serbest bırakmaktadır?

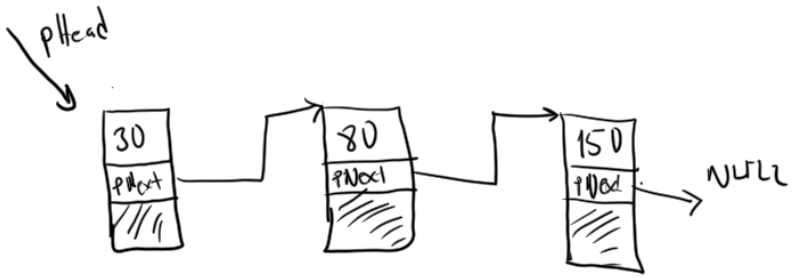
2) Bölünme (fragmentation) de önemli bir ölçüttür. Heap alanını daha az bölen, başka bir deyişle ardışıl bellek miktarını daha yüksek tutan sistemler diğerlerine tercih edilir.

3) Heap organizasyonu için oluşturulan veri yapısının kapladığı alanın (meta data alanının) küçüklüğü de bir performans ölçütüdür. Örneğin bir tahsisat alanını organize etmek için çok büyük bir alan kullanmak istemeyiz.

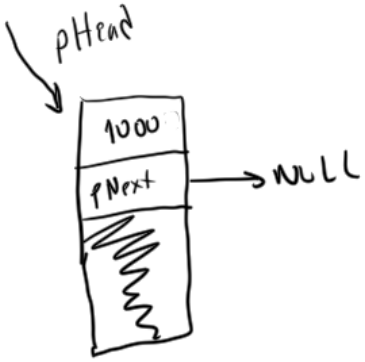
2.3.1. Klasik Yöntem: Boş Blokların Bağlı Listede Tutulması

Boş blokların bağlı listede tutulması en çok kullanılan tahsisat sistemi gerçekleştirim yöntemidir. Knuth bu yöntemi "The Art Of Computer Programming" kitabının birinci cildinde ele alınmıştır. Ayrıca Ritchie ve Kernigan'ın "The C Programming Language" kitabında da bu yöntem açıklanmaktadır. Bugün Windows sistemlerinde (yani HeapXXX fonksiyonlarında) ve UNIX/Linux sistemlerindeki C kütüphanelerinde temelk olarak bu yöntem kullanılmaktadır.

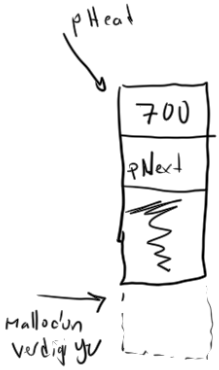
Bu yöntemde heap'teki boş alanlar bir bağlı listede tutulur. Bu boş alanların başında bir başlık kısmı vardır. Bu başlık kısmında bağlı listenin sonraki düğümünün yeri (next göstericisi) ve boş bloğun uzunluğu tutulmaktadır. Bu bağlı listeyi aşağıdaki şekille temsil edebiliriz:



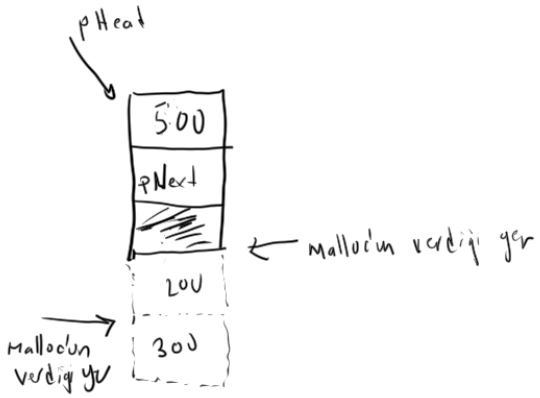
Bu şekilde bağlı listede sırasıyla 30 byte, 80 byte ve 150 byte boş blokların olduğunu görüyorsunuz. Pekiyi işin başında bu bağlı listenin durumu nasıldır? Örneğin heap'imizin 1000 byte olduğunu düşünelim. İşte işin başında bu listede tüm boş alanı tutan yalnızca tek bir eleman vardır:



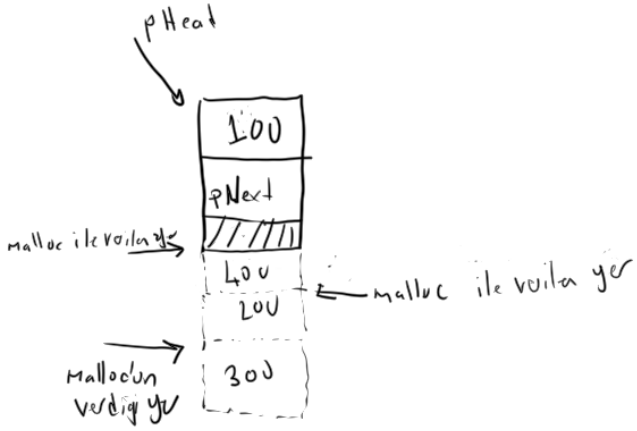
Sonra malloc gibi tahsisat fonksiyonları bu boş blok listesinde arama yaparak bize uygun bir blok verir. Bu yöntemde tahsis edilen blokların yerleri herhangi bir listede tutulmamaktadır. Yalnızca boş blokların kaydı bir bağlı listede tutulmaktadır. Örneğin böyle bir sistemde biz 300 byte tahsis etmiş olalım. Şimdi boş blok listesi şu hale gelecektir:



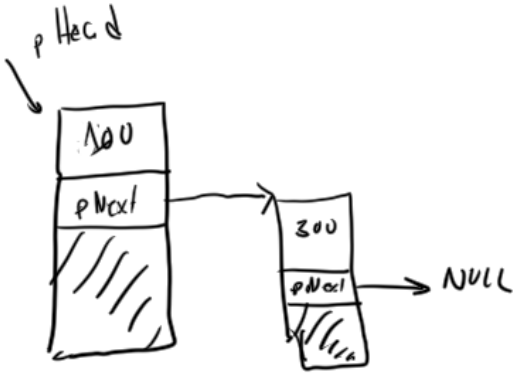
Şimdi de 200 byte daha tahsis etmiş olalım:



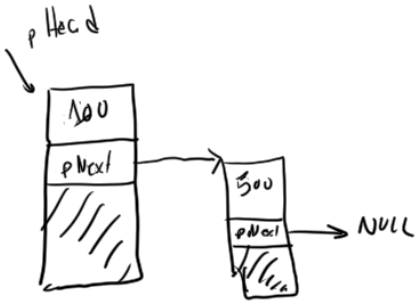
Şimdi 400 byte'lık bir tahsisat daha yapılmış olsun:



Şimdi 300'lük bloğun serbest bırakıldığını (free edildiğini) düşünelim. Artık bağlı listede iki eleman bulunacaktır:

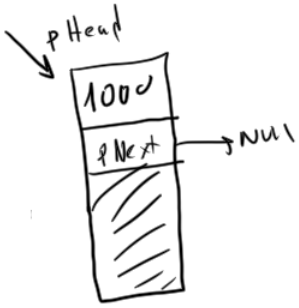


Tabii serbest bırakma işlemi yapılırken eğer serbest bırakılacakl blokla bağlı listedeki boş bir blok peşi sıra geliyorsa bunu ayrı blok olarak bağlı listede saklamak yerine bunları birleştirerek tek bir blok haline getirmek daha uygun olur. (Bunun nedenini düşününüz). Böylece ardışıl bloklar birleştirildiği için bağlı listedeki boş blokların hiçbiri ardışıl olmayacaktır. (Zaten ardışıl olanları birleştiriyoruz.) Örneğin şimdi biz daha önce tahsis etmiş olduğumuz 200 byte'lık bloğu serbest bırakmak isteyelim:



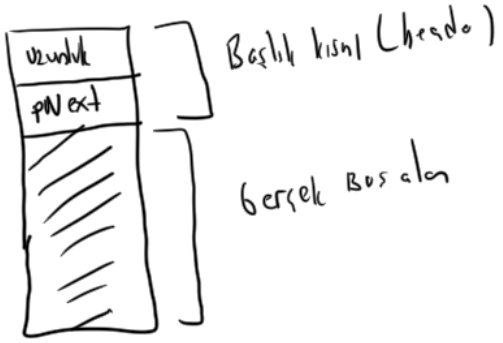
Gördüğünüz gibi 200 byte'lık blok 300 byte'lık blokla peşi sıra olduğu için birleştirilmiştir. Şimdi de 400 byte'lık bloğu serbest bırakalım:

,

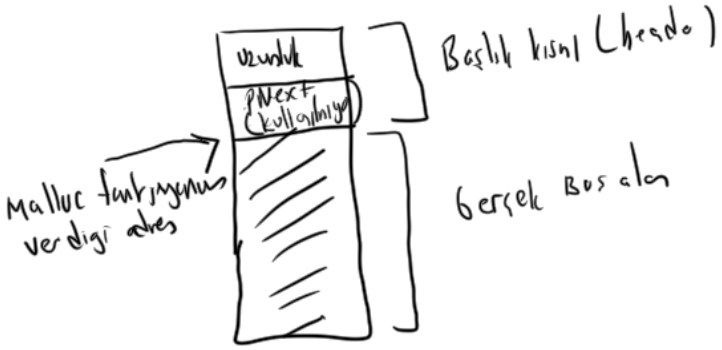


Gördüğünüz gibi tüm alanları serbest bırakınca ilk baştaki durumun aynısına geri döndük.

Şimdi veri yapısının ayrıntılarına girelim. Bu veri yapısında bağlı listedeki boş bloğun durumu şöyledir:



Gördüğünüz gibi boş bloğun başındaki başlık kısmında bloğun uzunluğu ve sonrakşi boş bloğun bağı (link) tutulmaktadır. Pekiyi tahsis edilmiş bir bloğun durumu nasıldır? Tahsistat fonksiyonu (örneğin malloc) bize boş blok listesinden uygun bloğu verirken o bloğun başlık kısmını ortadan kaldırmaz. Tahsisat fonksiyonunun bize verdiği adres bu başlık kısmından sonraki boş alanın adresidir. Dolayısıyla bize verilen adresten biraz yukarı çıktığımızda orada aslında orada tahsis edilen bloğun başlık kısmı vardır. O halde Tahsis edilmiş bloğun genel yapısı şöyledir:



Böylece bloğu serbest bırakan fonksiyon (örneğin free fonksiyonu) verilen adresin yukarısına bakarak tahsis edilmiş bloğun uzunluğunu görebilir. Tahsis edilmiş blok için artın bağ göstericisinin (next göstericisinin) bir anlamı kalmamıştır.

Pekiyi bu yöntemde tahsisat işlemini yapan fonksiyon nasıl başarısız oabilir mi? Evet, tahsisat fonksiyonu boş blok listesinde istenilen miktara eşit ya da ondan büyük hiç bir blok bulamazsa başarısız olacaktır.

Şimdi boş alanların bağlı listede tutulması tekniğinin gerçekleştirimini yapalım. Bu gerçekleştirimde heap olarak kullanacağımız alan bize başlangıç adresi ve uzunluğuyla verilmektedir. Biz de o alanı bu tekniğe göre organize edeceğiz. Gerçekleştirimde "handle tekniği" kullanılacaktır. Yani heap bilgileri bir yapıda tutulacak, onun başlangıç adresi bir handle olarak verilecek, bu handle kullanılarak da tahsisat işlemleri yapılacaktır.

Dinamik bellek tahsisatları genellikle byte düzeyinde belli bir değerin katlarında yaniş bir birim uzunlukl temelinde yapılmaktadır. Bu durum hizalama açısından fayda sağlamanın yanı sıra sistemin gerçekleştirimini de kolaylaştırmaktadır. Bizim gerçekleştirimimizde her boş bloğun başındaki başlık kısmının uzunluğu aynı zamanda birim uzunluk olarak alınacaktır. 32 bit sistemlerde bu birim uzunluk 8 byte'tır. Bu durumda örneğin biz 1 byte bile tahsis etmek istesek başlık kısmı haricinde aslında 8 byte'lık bir alan tahsis edilecektir.

Boş blokların başlık kısmı şağıdaki yapıyla temsil edilmektedir:

```
typedef struct tagBLOCK {
    size_t nunits;
    struct tagBLOCK *next;
} BLOCK;

#define UNIT_SIZE      sizeof(BLOCK)
```

Burada nunits başlık kısmı dahil olmak üzewre boş bloğun kaç birimden oluştuğunu belirtir. Bir birim sizeof(BLOCK) (yani 8 byte) uzunluğundadır.

Handle sistemi için aşağıdaki gibi bir yapı oluşturulabilir:

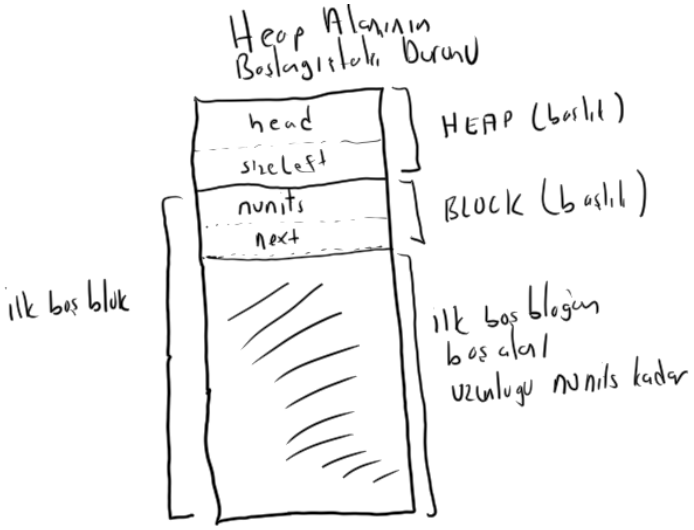
```
typedef struct tagHEAP {
    BLOCK *head;
    size_t sizeLeft;   /* byte cinsinden */
} HEAP, *HHEAP;
```

Bu HEAP yapısı aslında heap olarak ayrılan alanın başındaki başlık "header" kısmı gibidir.

Gerçekleştirimimizde heap alanını yaratan fonksiyon CreateHeap isimli fonksiyondur:

```
HHEAP CreateHeap(void *addr, size_t size)
{
    HHEAP hHeap;
    BLOCK *blk;

    if (size < sizeof(HEAP) + UNIT_SIZE)
        return NULL;

    hHeap = (HHEAP)addr;
    hHeap->sizeLeft = size - sizeof(HEAP);

    blk = (char *)addr + sizeof(HEAP);
    blk->nunits = hHeap->sizeLeft / UNIT_SIZE;
    blk->next = NULL;

    hHeap->head = blk;

    return hHeap;
}
```

Burada heap olarak kullanılacak alanın başındaki HEAP yapısıyla temsil edilen başlık kısmı oluştuurlmuştur. Bu başlığın hemen altından başlayan geri kalan tüm alan başlangıçtaki boş bloktur.



Sistemimizde test amaçlı heap'in boş bloklarını listeleyen yardımcı bir fonksiyonun bulunması faydalı olacaktır:

```
void DispHeap(HHEAP hHeap)
{
    BLOCK *blk;

    blk = hHeap->head;

    printf("Free Heap Size (byte): %u\n", hHeap->sizeLeft);
    while (blk != NULL) {
        printf("Unit Number = %d, Block Unit Size = %u\n",
            ((char *)blk - (char *)hHeap->head) / UNIT_SIZE, blk->nunits);
        blk = blk->next;
    }
    printf("-----------------------------------\n");
}
```

Burada "Unit Number" heap'in hemen başındaki birim (yani 8 byte'lık tahsisat birimi) 0 olmak üzere ilgili birimin numarasını belirtmektedir. "Block Unit Size" ise boş bloğun kaç birimden oluştuğunu belirtir.

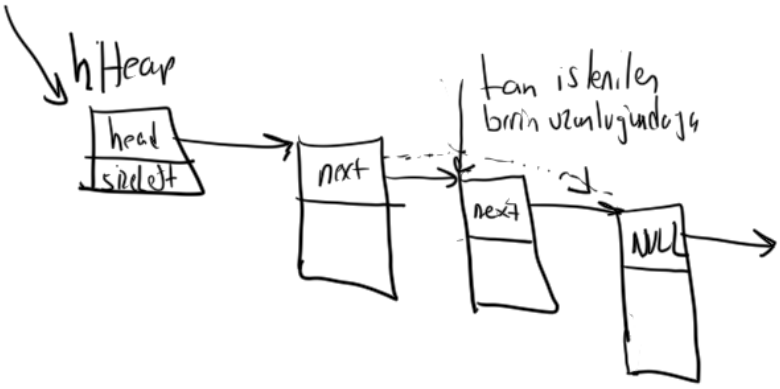
Şimdi en önemli fonksiyon olan tahsisat fonksiyonumuzu yazalım. Fonksiynumuzun ismi Malloc olsun:

```
void *Malloc(HHEAP hHeap, size_t size)
{
    size_t nunits;
    BLOCK *blk, *prevBlk;

    nunits = (size + UNIT_SIZE - 1) / UNIT_SIZE + 1;
    blk = prevBlk = hHeap->head;

    while (blk != NULL) {

        if (blk->nunits >= nunits) {
            if (blk->nunits == nunits) {
                if (blk == hHeap->head)
                    hHeap->head = blk->next;
                else
                    prevBlk->next = blk->next;
            }
            else {
                blk->nunits -= nunits;
                blk += blk->nunits;
                blk->nunits = nunits;
            }
            hHeap->sizeLeft -= nunits * UNIT_SIZE;

            return blk + 1;
        }
        prevBlk = blk;
        blk = blk->next;
    }

    return NULL;
}
```

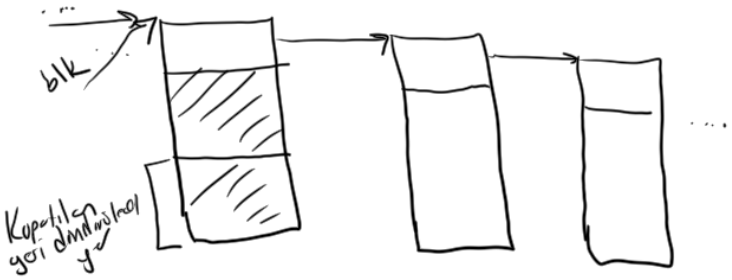
Burada önce tahsisat için gereken birim uzuluğu hesaplanmıştır:

```
nunits = (size + UNIT_SIZE - 1) / UNIT_SIZE + 1;
```

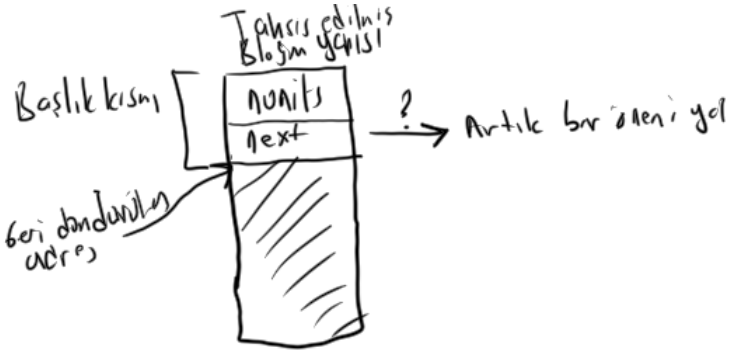
Daha sonra boş blok listesi bir döngü ile dolaşılmaktadır. Örneğimizdeki Malloc fonksiyonunda "First fit" algoritması uygulanmıştır. Bu durumda istenilen miktara eşit ya da ondan büyük olan ilk boş bloktan tahsisat yapılır. Bazı özel durumlara bakılmıştır. Örneğin boş blok tam olarak istenilen uzunluktaysa bu özel bir durumdur. Çünkü bu durmda önceki düğüm üzerinde değişikil yapılmalıdır:



Eğer boş blok büyükse o bloğun aşağısından gerekli boş alan kopartılmıştır.



Geri döndürülen adresin başlık kısmından bir birim ilerisi olduğuna dikkat ediniz. Böylece tahsis edilmiş olan bloğun hemen yukarısında bir başlık kısmı bulunmaktadır. Tabii bu başlık kısmının next göstericisinin bir önemi yoktur. Fakat birim uzunluğunun free işlemi için önemi varıdr:



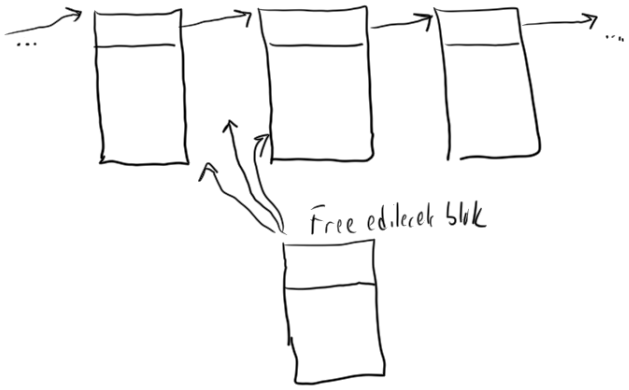
Şimdi sıra Free fonksiyonunu yazmaya geldi. Free fonksiyonunda dikkat eedilmesi gereken birkaç durum vardır:

1) Boş blokları tutan bağlı listedeki bloklar ardışıl değildir. Çünkü Free zaten bunları ardışıl hale getirecek biçimde yazılacaktır.

2) Free ile blok boş blok listesine hemen eklenmemeli. Bir birleştirme durumu var mı diye bakılmalıdır? Bölünme durumunu engellemek için mümkün olduğunca büyük blokların oluşturulması gerektiğini anımsayınız.

3) Boş blok listesindeki bloklar adrese göre sıralı durumdadır. Bu nedenle Free fonksiyonu hiç birleştirme yapmasa bile boş bloğu listenin sonuna değil adres sırasını bozmadan uygun yerine insert etmelidir.

Birleştirme için üç durum söz konusudur: Soldaki boş bloğun sağı ile birleştirme, Sağdaki boş bloğun solu ile birleştirme ve hem soldaki bloğun sağı hem de sağdaki bloğun solu ile tam birleştirme. Free algoritmasının bunu yapabilmesi gerekir.



Fonksiyon aşağıdaki gibi gerçekleştirilebilir:

```
void Free(HHEAP hHeap, void *addr)
{
    BLOCK *blk, *freeBlk;

    blk = hHeap->head;
    freeBlk = (BLOCK *)addr - 1;

    hHeap->sizeLeft += freeBlk->nunits * UNIT_SIZE;

    if (hHeap->head == NULL) {     /* special case 1: is list empty?*/
        hHeap->head = freeBlk;
        freeBlk->next = NULL;
        return;
    }

    if (hHeap->head > freeBlk) {   /* special case 2: add as a first node */
        if (freeBlk + freeBlk->nunits == hHeap->head) {
            freeBlk->nunits += hHeap->head->nunits;
            freeBlk->next = hHeap->head->next;
        }
        else
            freeBlk->next = hHeap->head;

        hHeap->head = freeBlk;

        return;
    }

    while (blk->next != NULL && blk->next < freeBlk)
            blk = blk->next;

    if (freeBlk + freeBlk->nunits == blk->next) {
        freeBlk->nunits += blk->next->nunits;
        freeBlk->next = blk->next->next;
    }
    else
        freeBlk->next = blk->next;

    if (blk + blk->nunits == freeBlk) {
        blk->nunits += freeBlk->nunits;
        blk->next = freeBlk->next;
    }
```

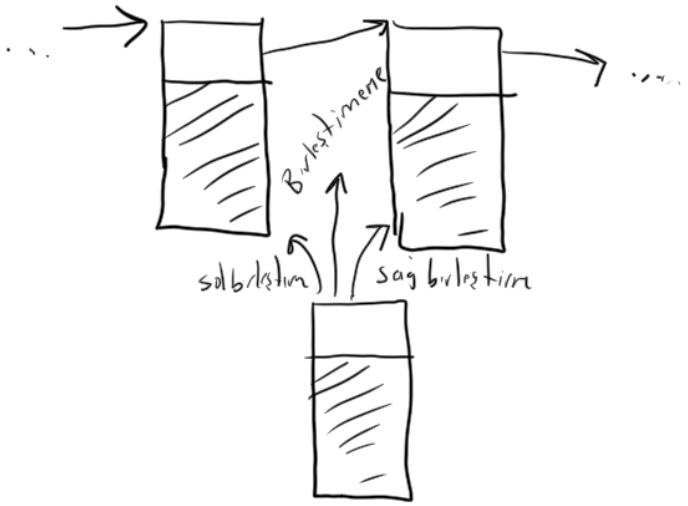
```
    else
        blk->next = freeBlk;
}
```

Burada önce iki özel duruma bakılmıştır:

1) Heap tamamen boş mudur durumu: Eğer böyleyse boş blok ilk blok olarak eklenip hHeap->head güncellenmiştir.

2) Ekleme hemen bağlı listenin başına mı yapılmaktadır? Bu durumda sağdaki düğümün solu ile birleştirme mümkünse yapılmış ve düğüm bağlı listenin başına eklenmiştir.

Gerçekleştirimde önce free edilecek bloğun bağlı listedeki yeri bulunmuştur. Sonra blok birleştirmesi durumuna bakılmıştır. Bunun için önce sağ tarafla birleştirme durumu kontrol edilmiştir. Sonra sol taraf ile birleştirmeye bakılmıştır.



Yukarıdaki gerçekleştirimde handle alanında (HEAP yapısında) yalnızca head göstericisi tutulmuştur. Eğer handle alaınında head göstericisi değil de bu niyetle BLOCK türünden bir nesne tutarsak özel durumlar ortadan kaldırılabilir ve özellikle Free algoritmaıs biraz da ha sade yazılabilir:

```
typedef struct tagHEAP {
    BLOCK head;         /* gösterici değil kendisi */
    size_t sizeLeft;    /* byte cinsinden */
} HEAP, *HHEAP;
```

Pekiyi boş listesinin çift bağlı liste (double linked list) olmasının bir faydası olabilir mi? Bilindiği gibi çift bağlı listeler iki amaçla tercih edilmektedir:

1) Tersten dolaşımı mümkün hale getirmek
2) Adresi bilinen bir düğümü sabit zaman karmaşıklıkta (O(1) karmaşıklıkta) silmek.

İşte çift bağlı liste yalnızca bizim tahsisat sırasında önceki düğümü tutmamızı engeller. Bu da ciddi bir fayda sağlamamaktadır. Üstelik çift bağlı listenin düğümleri (yani bloların başlıkları) daha fazla yer kaplayacaktır.

2.3.2. İkiz Blok Tahsisat Sistemi (Buddy Allocator)

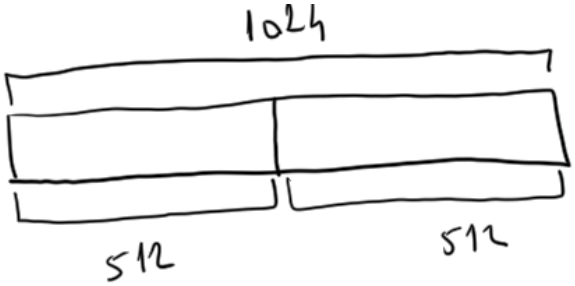
İkiz blok blok sistemi (buddy allocator) Linux başta olmak üzere bazı işletim sistemlerinin bellek yönetimlerinde kullanılmaktadır. Linux işletim sistemli sayfaları (page frames) bu tahsisat sistemine göre tahsis

eder. Linux'un çekirdek heap sistemini oluşturan "dilimli tahsisat sistemi (slab allocator)" de ikiz blok sistemi üzerine oturtulmuştur.

İkiz blok sisteminin en önemli avantajları tahsisatların ve geri bırakmaların hızlı bir biçimde yapılmasıdır. Bu sistemde dışsal bölünme (external fragmentation) oldukça iyi bir düzeyde tutulur. Ancak içsel bölünme (internal fragmentation) dolayısıyla oluşan kayıplar önemli boyutlara varabilmektedir. Tipik olarak tahsis edilen alanların %20 civarı içsel bölünme nedeniyle harcanmaktadır. İkiz blok sistemini gerçekleştirmek için kullanılan "metadata" alanları boş blok listesi sistemini gerçekleştirmek için kullanılan "metadata" alanlarından kimi durumlarda daha büyük olabilmektedir. Ayrıca bu sistemin gerçekleştirimi boş blok listesi sistemine göre daha zordur.

İkiz blok sisteminde tahsis edilebilecek blokların uzunlukları 2^n biçimindedir. Burada n değerine "mertebe (order)" denir. Sistemde genellikle başlangıçta 2^n'lik tek bir blok bulunur (ancak başlangıçta tek bloğun bulunması zorunlu değildir). Başlangıçtaki bu n değerine maksimum mertebe (maximum order) denir. İkiz blok sisteminde tahsis edilebilecek en küçük mertebenin (minimum order) de başlangıçta belirlenmesi gerekir. Teorik olarak en küçük blok $2^0 = 1$ byte olsa da, 1 byte tahsisat için çok küçük bir alandır. Örneğin en büyük mertebenin 10, en küçük mertebenin 3 olduğu bir sistem düşünelim. Bu durumda başlangıçtaki bellek miktarı $2^{10} = 1024$, tahsis edilebilecek en küçük blok uzunluğu da $2^3 = 8$ olacaktır.

İkiz blok sisteminde yan yana iki 2^k'lik 2^{k-1}'lik bir blok biçiminde ele alınmaktadır. Örneğin 1024'lük blok yan yana 2 tane 512'lik blok gibi ele alınabilir:



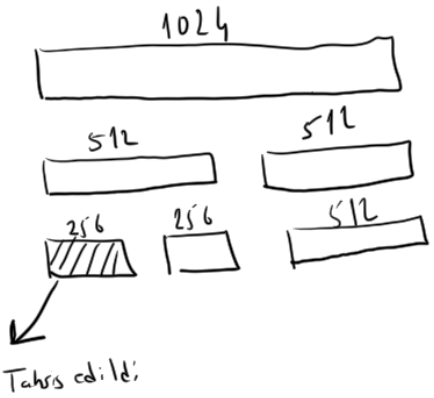
İşte yan yana iki tane 2^k'lık blok biribirlerinin ikizi (buddy'si) durumundadır. Belli bir anda ikiz bloklar için tahsisat durumu üç biçimde olabilir:

1) Bunlardan biri tahsis edilmiş biri edilmemiş olabilir.
2) Bunlardan her ikisi de tahsis edilmiş olabilir.
3) Bunların her ikisi de tahsis edilmemiş olabilir.

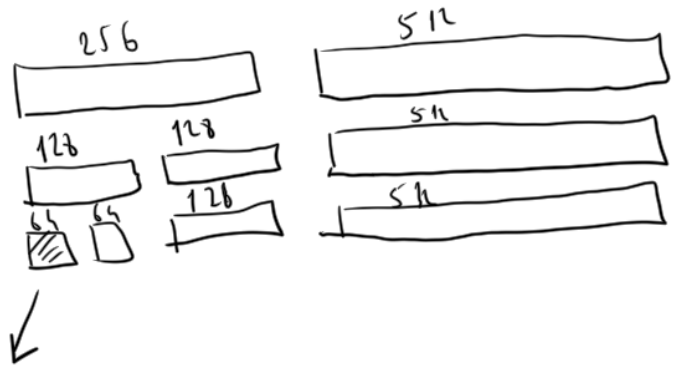
İşte ikizlerin her ikisi de tahsis edilmemiş duruma gelince bu bloklar birleştirilip daha yüksek mertebeye yükseltilmektedir. Bu durumda ikizlerden biri serbest bırakıldığında sistem diğer ikizin durumuna bakar; o da serbest bırakılmış durumdaysa onları birleştirir.

Yukarıda da belirttiğimiz gibi ikiz blok sisteminde genellikle başlangıçta tek bir blok vardır. Bir tahsisat yapılmak istendiğinde bu tek blok sürekli ikiye bölünerek istek karşılanmaya çalışılır. Örneğin maksimum mertebesi 10 olan (yani 1024 byte olan) bir sistemde 200 byte'lık bir alanın tahsis edilmek istendiğini düşünelim. İkiz blok sisteminde tahsis edilebilecek bloklar ikinin kuvvetleri uzunluğunda olmak zorundadır. Bu nedenle bu sistemde tam olarak 200 byte'lık bir blok tahsis edilemez. Onun yerine 200'den büyük olan 2'nin en küçük kuvveti kadar uzunlukta tahsisat yapılabilir. O da 256'dır. (Bu durumda 56 byte'ın içsel bölünme

nedeniyle harcandığına dikkat ediniz.) Bu tahsisat sırasında bloklar ikiye bölünerek istek karşılanmaya çalışılır. Önce 1024'lük tek blok iki ayrı 512'lik blok olarak bölünür. Sonra bu 512'lik bloklardan biri de 256'lık iki blok olarak bölünür. Bu 256'lıklardan biri de tahsis edilir.



Böylece elimizde bir tane 512'lik bir tane de 256'lık blok kalmıştır. Şimdi 50 byte'lık bir bloğun daha tahsis edilmek istendiğini düşünelim. 50'ye en yakın 2'nin kuvveti 64'tür. Bu durumda 64 byte tahsis edilecektir. Tahsisat işlemi eldeki en küçük bloktan hareketle yürütülür. Elimizdeki en küçük blok 256'lıktır. O halde bu blok önce iki 128 olarak bölünür. Sonra o 128'lerden biri yine ikiye bölünerek istek karşılanır:



Şimdi elimizde bir tane 64'lük, bir tane 128'lik bir tane de 512'lik blok bulunmaktadır. Şimdi bu son durumda 256'lık bloğun serbest bırakıldığını düşünelim. Yeni durumdaki serbest bloklar şöyle olacaktır:

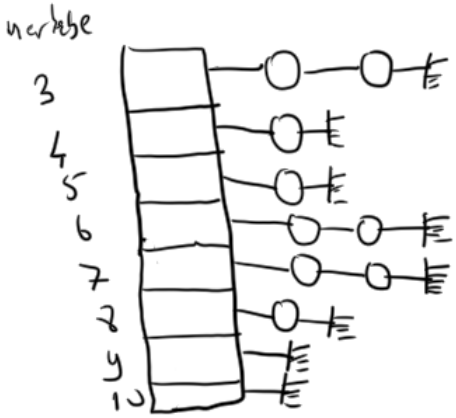
Bu sistemde nasıl tahsisat sırasında bloklar parçalanıyorsa serbest bırakma sırasında da mümkün olduğunca yan yaana bloklar birleştirilmeye çalışılır. Ancak örneğimizde 256'lık alanın serbest bırakılması herhangi bir birleştirmeye yol açmamaktadır. Çünkü birleştirme ancak ikizlerle (buddy'lerle) yapılır. Buradaki 256'lık bloğun ikizi tamamen serbest durumda değildir. Şimdi 64'lük bloğun da serbest bırakıldığını düşünelim. Bu 64'lük bloğun ikizi de serbest durumda olduğu için bunlar birleştirilir ve 128'lik tek blok haline getirilir. Bu 128'in de ikizi serbest olduğu için bunlar da birleştirilir ve 256'lık tek blok haline getirilir. Bu 256'nın da ikizi serbest durumda olduğu için bunlar da birleştirilir ve 512'lik tek blok oluşturulur. Nihayet bu 512'lik bloğun ikizi de serbest durumda olduğu için onlar da birleştirilecek ve 1024'lük tek blok elde edilecektir. Görüldüğü gibi ikiz blok sisteminde mümkün olduğunca ikizler birleştirilerek ardışıl büyük parçalar elde edilmeye çalışılmaktadır.

2.3.2.1. İkiz Blok Sisteminin Gerçekleştirilmesi

İkiz blok sistemi çok değişik biçimlerde gerçekleştirilebilmektedir. Ancak pek çok gerçekleştirim benzer veri yapılarından faydalanır. Gerçekleştirimde kullanılan veri yapılarının ana noktalı şunlardır:

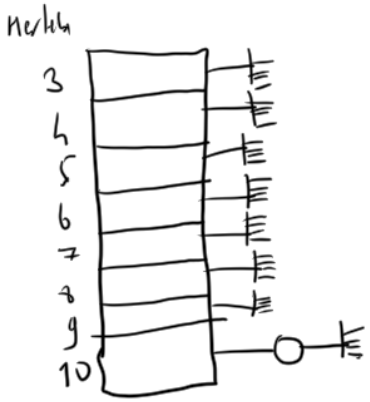
1) Her mertebe için o mertebedeki boş blokları tutan bir bağlı liste bulundurulabilir. Örneğin maksimum mertebesi 10 olan ve minimum mertebesi 3 olan bir sistem için toplamda 10 – 3 + 1 = 8 tane bağlı liste bulundurulacaktır. Bu bağlı listelerin her birinde o mertebedeki boş bloklar tutulacaktır. Örneğin eldeki boş bloklar şöyle olsun:

Şüphesiz aynı bağlı listedeki boş bloklar birbirlerinin ikizi değildir (çünkü ikizi olsalardı zaten birleştirilmiş olurlardı). Bu boş blokların oluşturduğu bağlı listeler şöyle gösterilebilir:

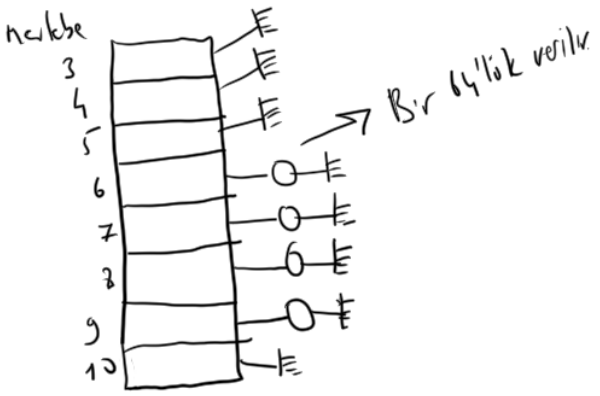


Pekiyi bu bağlı listelerin düğümleri (linkleri) nerededir? Aslında düğümler boş blok listesi tekniğinde olduğu gibi blokların içerisinde, onların başlarında tutulabilir. Blok tahsis edilince zaten bu düğümlere de gerek kalmayacaktır. Pekiyi bu bağlı listeler neden oluşturulmaktadır? İşte tahsisat sırasında önce hangi mertebeden tahsisat yapılacağı belirlenir, sonra da bu bağlı liste dizisinin ilgili elemanının belirttiği bağlı listenin hemen başından eleman alınır. Böylece tahsisat işlemi ek maliyetli sabit zamanlı (amortized constant time) bir işlem olarak yapılabilmektedir.

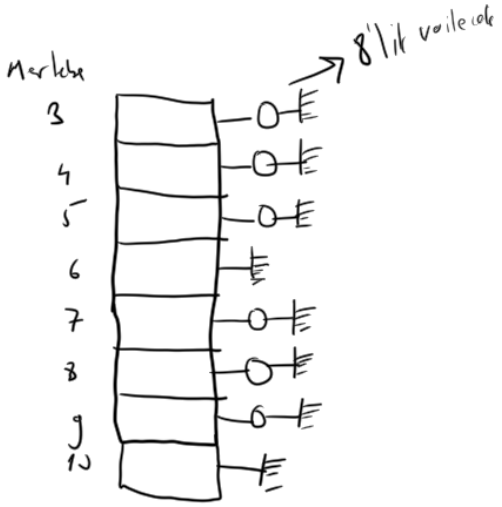
Yukarıda da belirttiğimiz gibi işin başında tipik olarak bu bağlı liste dizisinin içerisinde tek bir eleman vardır:



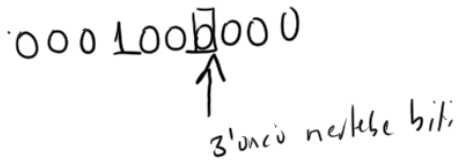
Belli bir mertebeden blok tahsisat istendiğinde bu bağlı liste dizisinde öncelikle o mertebeye ilişkin bağlı listenin boş olup olmadığına bakılır. Eğer o liste boşsa mertebe olarak yukarıya doğru ilk boş olmayan bağlı liste bulunur. Sonra o liste parçalanarak aşağıya doğru istediğimiz mertebeye kadar birer eleman eklene eklene geri gelinir. Örneğin ilk durumda 64'lük bir blok tahsis edilmek istenmiş olsun:



Burada 1024'lük blok bölünerek 64'lük bloğa kadar gelinmiştir. 64'lük bloğun ikizlerinden biri bağlı listeye eklenirken diğeri tahsis etmek isteyen kişiye verilmiştir. (1024 = 512 + 256 + 128 + 64 + 64 olduğuna dikkat ediniz). Artık burada dolu bağlı liste mertebelerinden tahsisat yapılmak istendiğinde hiç blok bölmesi yapılmadan doğrudan o listeden blok verilecektir. Pekiyi yukarıdaki son durumda biz 8'lik bir blok tahsis edilmek istense ne olur? Bu durumda 64'lük blok benzer biçimde bölünerek 8'lik bloklardan biri verilir:



Şimdi bir bloğun ikizinin yerinin belirlenmesi sürecine bakalım. Elimizde bir blok adresi varsa ve o bloğun mertebesini biliyorsak onun ikizinin yerini bulabilir miyiz? Örneğin heap'in 0'dan 1024'e kadar adreslendiğini varsayalım. 64 adresine sahip 3'üncü mertebeden bloğun ikizinin yeri neresidir? Öncelikle bu bloğun ikizi bu bloğun ya solundadır ya da sağındadır. Biz 64 değerini 2^3 değerine bölerek elde edilen değerin tek mi çift mi olduğuna bakabiliriz. Elde edilen değer çift ise bizim bloğumuzun ikizi onun sağındadır, tek ise solundadır. Örneğin 64 / 8 = 8'dir. 8 çift olduğu için bu bloğun ikizi onun sağındadır. Yani onun 8 byte ilerisindedir. Tabii bu işlemler bit düzeyinde çok daha pratik yapılabilmektedir. Şöyle ki: k adresindeki 2^n uzunluğundaki bloğun ikizinin yerini bulmak isteyelim. k değerinin ikilik sistemdeki n'inci bitinin durumu zaten onun ikizinin yeri hakkında bize bilgi vermektedir. Örneğin 64'üncü adresteki 2^3 = 8 uzunluğundaki bloğun ikizinin yerini bulmaya çalışalım:



İşte eğer mertebeye ilişkin bit 1 ise onun ikizi onu sıfır yaparak (yani 2^n kadar geriye giderek), 0 ise onu 1 yaparak (yani 2^n kadar ileriye gidilerek) elde edilebilir. EXOR işleminde 0'ın etkisiz eleman olduğunu 1'in ise evrik almakta kullanıldığını anımsayınız. O halde k adresinde ve 2^n uzunluğunda bir bloğun ikizinin yeri şöyle bulunabilir::

```
buddyAddr = (void *) ((unsigned long) k ^ 1 << n);
```

Pekiyi ikiz blok sisteminde bir blok serbest bırakılmak istendiğinde neler yapılacaktır? Öncelikle serbest hale getirilecek bloğu ilgili bağlı listeye eklemeden önce onun ikizinin serbest durumda olup olmadığına bakmak gerekir. Eğer bloğun ikizi de serbest durumdaysa ikizini bağlı listeden çıkartıp birleştirerek üst mertebedeki bağlı listeye eklemek gerekir. Fakat eklemeden önce yine o mertebedeki ikizin de serbest olup olmadığına bakılması gerekir. Bu biçimde birleştirme yapılamayana kadar ilerlenir.

Pekiyi serbest bırakılan bloğun ikizinin boş olup olmadığını nasıl anlayabiliriz? Bunu anlamak için ilgili mertebeye ilişkin bağlı listeyi dolaşmak etkin yöntem değildir. (Başarısız aramalarda bağlı listenin sonuna kadar gidileceğine dikkat ediniz.) Bu nedenle programcılar bloğun ikizinin durumu için genellikle ayrı bit dizileri kullanma yoluna giderler. Şöyle ki: Her mertebe için bir bağlı listenin yanı sıra o mertebedeki blokların durumunu gösteren bir bit dizisi de oluşturulur. Örneğin 64 numaralı adresteki 3'üncü mertebeden bloğun ikizinin bit dizisi içerisindeki yeri 64 / 8 = 8 + 1 = 9'dur. Biz bu bite bakarak onun boş olup olmadığını anlayabiliriz. Tabii tahsisat sırasında o bitin de set edilmesi ve serbest bırakma sırasında reset edilmesi gerekmektedir. Aslında buradaki bit dizilerinin ilgili mertebedeki blokların sayısı kadar değil onun yarısı kadar uzunlukta açılması da mümkündür. Örneğin Linux'taki gerçekleştirimde bit dizileri ilgili mertebedeki blok sayılarının yarısı kadardır. Yani her ikiz için tek bit tutulmaktadır. O bit 1 ise ikizlerden biri serbest durumdadır ve birleştirme yapılabilir. Sıfır ise birleştirme yapılamaz. İlgili bit sıfırken ikizlerden biri serbest bırakılmışsa o bit 1 yapılmaktadır. Bazı programcılar bloğun ikizinin boş olup olmadığını bit dizileriyle değil bizzat ikizin içerisinde onun baş kısmında da tutulabilmektedir. Böylece bir blok serbest bırakılırken onun ikizinin yeri belirlenir ve onun baş kısmına bakılarak bloğun boş olup olmadığına karar verilir. Tabii bu durumda bloğun başındaki başlık kısmı tahsis eden tarafından kullanılamayacaktır.

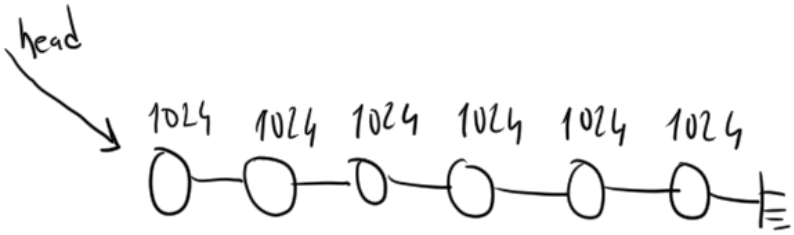
Şimdi başka bir soruna bakalım: İkiz blok sisteminin veri yapısı oluşturulurken bu metadata bilgileri nerede tutulur? Örneğin bağlı listeler ve bit dizileri nerede organize edilmektedir? İşte bu metadata'lar heap olarak belirlenen alanın başında bir yerde oluşturulabilir. Böylece geri kalan alan tahsisat için kullanılabilir. Ya da tamamen başka bir de oluşturulabilir. Yukarıda da belirtildiği gibi bağlı listelerin düğümleri doğrudan bloklar içerisinde, o blokların başında tutulabilmektedir. Fakat blok tahsis edilince o bağların bir önemi kalmadığı için bu düğüm alanı tahsis edilen bloğa dahil edilebilir.

Pekiyi bağlı liste dizilerindeki bağlı listeler tek bağlı mı çift bağlı mı olmalıdır? Serbest bırakma sırasında bloğun ikizinin yerini tepit edip onu bağlı listeden hızlı bir biçimde çıkartmak için listelerin çift bağlı olması daha uygundur. (Çift bağlı listelerde adresi bilinen bir düğümün sabit zamanlı olarak silinebildiğini anımsayınız.)

2.3.3. Dilimli Tahsisat Sistemi (Slab Allocator)

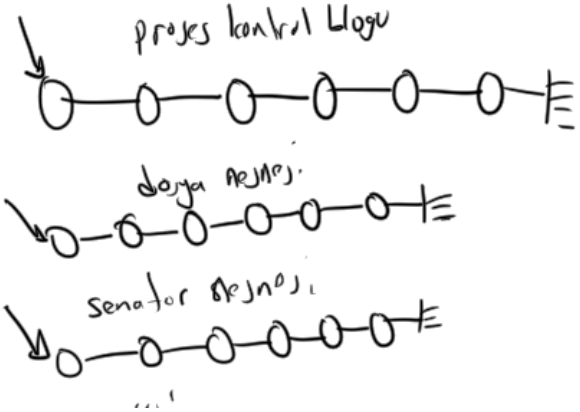
Dilimli tatsisat sistemi özellikle işletim sistemlerinin çekirdek heap sistemlerinin gerçekleştirilmesinde yaygın olarak kullanılmaktadır. Örneğin Solaris, Linux ve BSD sistemleri kendi çekirdekleri içerisindeki tahsisatlarda dilimli tahsisat sistemini kullanmaktadır. Dilimli tahsisat sistemi ilk kez Jeff Bonwick tarafından Sun OS 5.4 sistemlerinde uygulanmıştır. (Bonwick tarafından yazılan "The Slab Allocator: An Object-Caching Kernel Memory Allocator" makalesini inceleyiniz).

Dilimli tahsisat sisteminin dayandığı fikir basittir: Boş blokların bağlı listede tutulması tekniğinde eğer tüm boş bloklar aynı uzunlukta olsaydı tahsisat ve geri bırakma işlemleri çok hızlı yapılırdı değil mi? Çünkü bu durumda bu boş bağlı listede arama yapmaya gerek kalmazdı. Örneğin tüm boş blokların 1024 byte uzunluğunda olduğunu varsayalım. Şimdi biz 1024 byte tahsis etmek istediğimizde sistem hemen bağlı listenin önündeki bloğu bize verebilir:

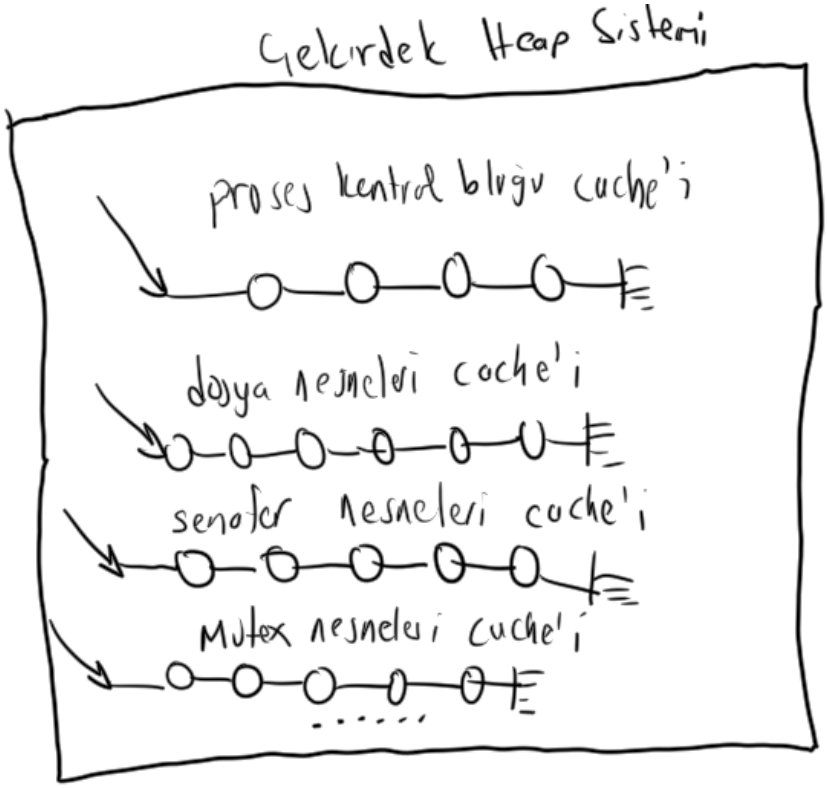


Tabii böyle bir sistemde tahsis edilecek blokların hep aynı uzunlukta olacağı varsayılmaktadır. Bu varsayım altında bir bloğun serbest bırakılması durumunda bir birleştirme yapmanın da gereksiz olduğu aşikardır.

Pekiyi hep aynı uzunlukta tahsisat yapma gerçekçi bir durum mudur? Daha büyük ya da daha küçük blok tahsis etmek istersek ne olacaktır? İşte dilimli tahsisat sistemi genel amaçlı bir sistem olmaktan çok özel amaçlı bir sistemdir. Örneğin işletim sistemlerinin çekirdeklerinde dinamik biçimde tahsis edilecek pek çok yapı vardır (proses kontrol blokları, dosya nesneleri, senkronizasyon nesneleri vs.) İşte uzunluğu zaten baştan belli olan bu nesneler için ayrı ayrı (her biri ayrı uzunlukta) dilimli tahsisat sistemleri oluşturulabilir. Böylece aşağıdaki gibi farklı uzunlukta nesneleri tutan ayrı boş blok listeleri elde edilmiş olacaktır:



Dilimli tahsisat sisteminde farklı uzunluklar için oluşturulmuş her tahsisat sistemlerine “cache” denilmektedir. Bu sistemde önce belli bir uzunluk belirtilerek bir “cache” oluşturulur. Sonra tahsis etme ve geri bırakma işlemleri cache belirtilerek belli bir cache'ten yapılır. Böylece örneğin çekirdek yazılımcısı bir tane “proses kontrol bloğu” tahsis edecekse onu “proses kontrol bloğu cache’inden” tahsis eder.



Linux işletim sisteminde yeni bir cache yaratmak için kmem_cache_create isimli çekirdek fonksiyonu kullanılmaktadır. Örneğin Linux 2.4 çekirdeğindeki kmem_cache_create fonksiyonunun prototipi şöyledir:

```
kmem_cache_t *kmem_cache_create (const char *name, size_t size, size_t offset,
        unsigned long flags, void (*ctor)(void*, kmem_cache_t *, unsigned long),
        void (*dtor)(void*, kmem_cache_t *, unsigned long));
```

Bu fonksiyon bir cache yaratır. Cache bilgileri kmem_cache_t isimli bir yapı ile temsil edilmektedir. Fonksiyon yaratılan cache'e ilişkin cache bilgilerinin tutulduğu yapının adresini bize bir handle değeri gibi vermektedir. (Bu handle değerine Linux terminoloisinde "cache descriptor" deniyor.) Linux'ta dilimli tahsisat sisteminde kullanılan cache'lerin handle alanları da (yani kmeme_cache_t yapıları da) yine dilimli tahsisat sistemi ile tahsis edilmektedir. Dilimli tahsisat sisteminin kendi veri yapıları için kullanılan cache "cache_cache" biçiminde isimlendirilmektedir. Yaratılan bir cache kmem_cache_destroy fonksiyonuyla serbest bırakılır.

Linux'ta belli bir cache'ten (yani belli uzunluk için yaratılmış cache'ten) tahsisat yapmak için kmem_cache_alloc fonksiyonu kullanılmaktadır:

```
void *kmem_cache_alloc(kmem_cache_t *cachep, int flags);
```

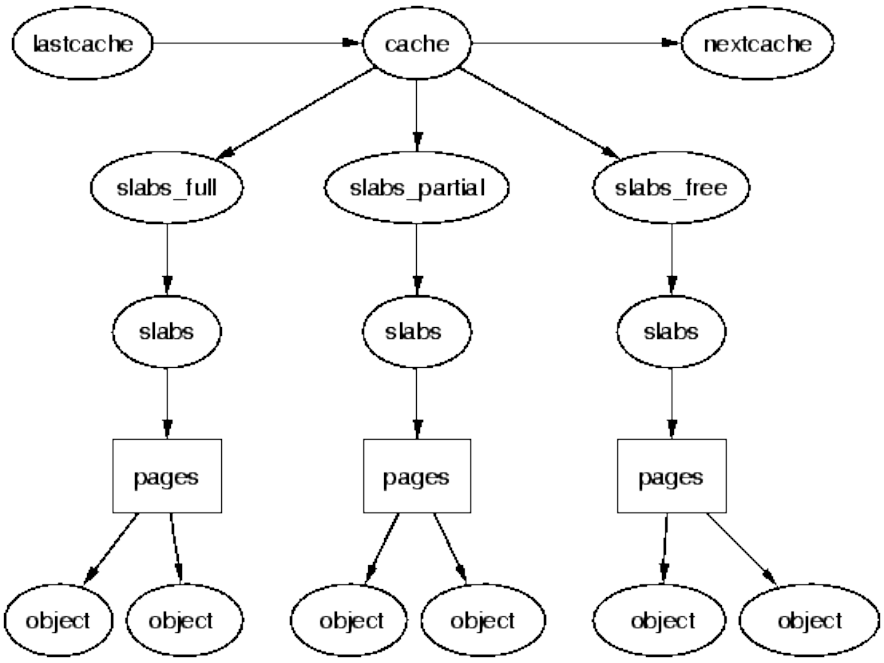
kmem_cache_alloc fonksiyonuyla tahis edilen blok kmem_cache_free fonksiyonuyla serbest bırakılmaktadır:

```
void kmem_cache_free(kmem_cache_t *cachep, void *objp);
```

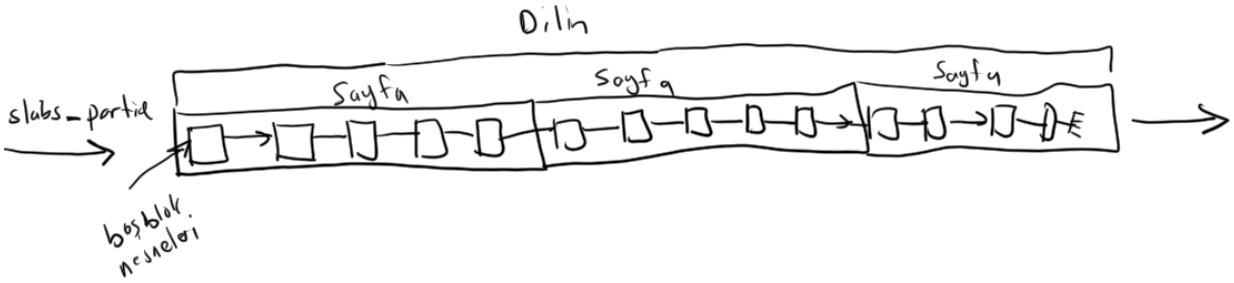
2.3.3.1. Dilimli Tahsisat Sisteminin Ayrıntıları

Dilimli tahsisat sistemi yukarıda da belirtildiği gibi özellikle işletim sistemlerinin çekirdek heap alanlarının organizasyonunda tercih edilmektedir. Öncelikle şu soruya yanıt arayalım: Dilimli tahsisat sistemindeki "dilim (slab)" terimi ne anlam ifade etmektedir? Bir dilim (slab) ardışıl n tane sayfadan (page) oluşan bir heap bloğudur. İşletim sistemi düzeyinde en aşağı seviyeli olarak tahsisatlar sayfa temelinde yapılmaktadır. Yukarıda da belirttiğimiz gibi sayfa tahsisatları için genellikle ikiz blok sistemi (buddy allocator) tercih edilmektedir. İşte dilimli tahsisat sistemi için bir cache oluşturulduğunda o cache de dilimlerden oluşmaktadır. Tahsis edilecek bloklar dilimlerin içerisindedir ve onlara bu terminolojide nesne (object) denir. Cache sistemi önce bir dilimle başlatılır, sonra yetmezse başka dilimler de cache'e dahil edilir. Bir dilimdeki nesnelerin hepsi serbest

bırakıldığında o dilim hemen çekirdeğin aşağı seviyeli sayfa tahsisat sistemiyle (ikiz blok sistemine) hemen iade edilmez. Daha sonra gereksinim duyulabilir diye bekletilir. Dilimli tahsisat sistemi için bir cache sistemini aşağıdaki gibi bir şekille temsil edebiliriz:

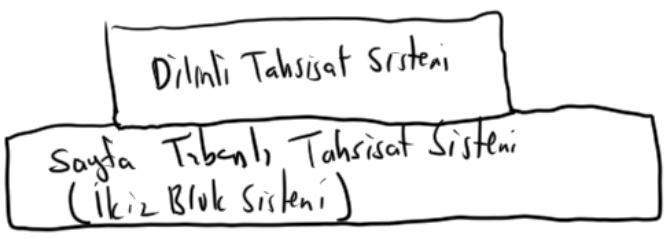


Bu şekil Linux işletim sisteminin çekirdeğini anlatan bir dokümandan alınmıştır. Linux çekirdeği bir dilimli tahsisat sistemi için bir cache yarattığında o cache içerisinde 3 tür dilimi bağlı listeler halinde tutmaktadır: Tam boş olan dilimler (slabs_free), tam dolu olan dilimler (slabs_full) ve tam dolu olmayan (ya da tam boş olmayan) dilimler (slabs_partial). Cache içerisindeki boş blok listeleri dilimlerin dolayısıyla sayfaların içerisindedir. Örneğin:



Dilimli tahsisat sistemindeki sayfalar, dilimler ve nesneler ile ilgili önemli sorular ve yanıtları şöyledir:

Soru: Dilimli tahsisat sistemi çekirdeğin en aşağı seviyeli tahsisat sistemi midir?
Yanıt: Hayır. Dilimli tahsisat sistemi işletim sistemlerinde sayfa tabanlı tahsisat işlemlerini yapan tahsisat sistemini (tipik olarak ikiz blok sistemini) kullanan daha yüksek seviyeli bir tahsisat sistemidir.



Yani dilimli tahsisat sistemi tahsisatta kullanacağı dilimleri (yani heap olarak kullanacağı alanı) sayfa düzeyinde tahsisat yapan alt sistemden istemektedir.

Soru: Dilimli tahsisat sisteminde cache'te neden birden fazla dilim (slab) vardır örneğin neden büyük ve tek bir dilim yoktur?
Yanıt: Cache'in dilimlerden oluşmasının nedeni belli dilimler tamamen boşaltıldığında sayfa tabanlı tahsisat sistemine onların geri verilmesini sağlamaktır. Eğer cache'te yalnızca bir tane büyük bir dilim bulunsaydı o büyük dilim sürekli bir kısmı iade edilemeden bekletilirdi.

Soru: Bir dilim neden ardışıl n tane sayfadan oluşmaktadır da bir tane sayfadan oluşmamaktadır?
Yanıt: Şüphesiz bu durum mümkün olabildi. Ancak bir dilimin bir sayafadan oluşması bazı bakımlardan etkin değildir. Çünkü bir sayfa nispeten küçük bir birimdir (Intel'de 4K). Eğer bir dilim 1 sayfadan oluşturulsa bu kez cache içerisinde pek çok dilim birikir ki, bunların da bağlı liste içeirisnde tutulması ve işlenmesi zor olur. Ayrıca dilimlerin tek sayfadan oluşması durumunda tahsisat sistemi için gereken toplam "metadata" alanları da oransal olarak büyüyecektir.

Soru: Dilimli tahsisat sisteminde gerçek tahsisatı yapan fonksiyonlar (örneğin Linux'taki kmem_cache_alloc) tahsisatı hangi dilimlerden yapmaktadır?
Yanıt: Cache'teki hepsi dolu olmayan (slabs_partial) dilimleri içerisindeki dilimlerden.

Soru: Bir cache'te aynı uzunluktaki nesneleri tutan bağlı listelerden kaç tane vardır?
Yanıt: Hepsi dolu olmayan her dilim için bu listeden bir tane vardır. Ayrıca Linux sistemlerindeki dilimli tahsisat sistemi gerçekleştiriminde boş blokların kendileri değil onların indeks numaraları bağlı listede tutulmaktadır (Bunun için çekirdek kodlarını inceleyiniz. Çekirdek kodları içerisindeki slab_t yapısının kmem_bufctl_t elemanı aslında int türden indekslerden oluşan bir bağlı liste dizisidir.)

Soru: Cache içerisinde tamamen boşaltılmış dilimler ne zaman sayfa tabanlı tahsisat sistemine iade edilmektedir?
Yanıt: Boşaltılmış dilimler cache'te daha sonra gereksnimin duyulur diye bekletilirler. Zaten buna "cache" denmesinin bir nedeni de budur. Çekirdek başka amaçlarla sayfalara gereksinim duyduğunda bu boş sayfalar iade edilmektedir.

Soru: Belli bir anda tüm dilimler doluyken tahsisat yapılmak istenirse ne olur?
Yanıt: Bu durumda sayfa tabanlı tahsisat sisteminden yeni bir dilim tahsis edilerek cache'e dahil edilir.

Dilimli tahsisat sisteminin önemli bir özelliği de tahsis edilen nesneler için yapılan başlangıç ve bitiş işlemlerinin (bunları nesne yönelimli teknikteki "constructor" ve "destructor"lara benzetebilirsiniz) mümkün olduğunca az yapılmasını sağlamasıdır. Şöyle ki: Bu sistemde bir nesne tahsis edildiğinde o nesnenin elemanlarını için yapılan birtakım ilkdeğer verme işlemleri ve diğer işlemler (örneğin nesnenin bir elemanı "semaphore" olabilir, işin başında bu "semaphore"un yaratılması gerekir) yalnızca bir kez yapılmaktadır. Yani nesne ilk kez tahsis edildiğinde bu ilk işlemler bir kez yapılır. Sonra nesne boşaltıldığında bitiş işlemleri uygulanmadan nesne cache'te bekletilir. Daha sonraki bir tahsisatta sistem bu nesneyi yeniden verdiğinde bu ilk işlemler gereksiz biçimde yeniden yapılmayacaktır. Benzer biçimde nesnenin yok edilmesi sırasında yapılacak son işlemler de nesne serbest bırakılırken değil, tüm dilim sayfa tabanlı tahsisat sistemine iade edilirken yalnızca bir kez yapılmaktadır. Daha önce vermiş olduğumuz kmem_cache_create fonksiyonun prototipine bir kez daha bakınız:

```
kmem_cache_t *kmem_cache_create (const char *name, size_t size, size_t offset,
        unsigned long flags, void (*ctor)(void*, kmem_cache_t *, unsigned long),
       void (*dtor)(void*, kmem_cache_t *, unsigned long));
```

Bu fonksiyondaki ctor fonksiyon göstericisi bir nesne tahsis edildiğinde o nesneyi ilklemek (initialize etmek) için yalnızca bir kez çağrılacak fonksiyonu, dtor fonksiyon göstericisi de dilim sayfa tabanlı tahsisat sistemine iade edildiğinde bitiş işlemleri için yalnızca bir kez çağrılacak fonksiyonu belirtmektedir.

Linux işletim sisteminde (dilim tahsisat sistemini kullanan diğer sistemlerde de böyle) belli veri yapıları için oluşturulmuş cache'lerin dışında bir de 32, 64, 128, 256, 512, 1024, 2048, 4096, 8192, 16384, 32768, 65536 ve 131072 uzunlukta genel amaçlı cache'ler bulundurmaktadır. Bu cache'ler sistem açılırken yaratılırlar. Böylece örneğin bir aygıt sürücüsü yazan programcı kendi amaçları için çekirdeğin heap'inden n byte tahsis etmek isterse bu n değerinden büyük olan en küçük cache'ten tahsisatını yapar. Aslında Linux'ta bu işlemi tek hamlede yapan kmalloc isimli bir çekirdek fonksiyonu bulunmaktadır:

```
void *kmalloc (size_t size, int flags);
```

Fonksiyonun "cache descriptor" istemediğine dikkat ediniz. Fonksiyon doğrudan birinci parametresiyle belirtilen uzunluktan büyük olan en küçük cache'i hesaplar ve tahsisatını o cache'ten yapar. kmalloc ile tahsis edilen blok kfree fonksiyonuyla geri bırakılabilmektedir:

```
void kfree (const void *objp);
```

2.3.3.2. Dilimli Tahsisat Sisteminin Performans Kıyaslaması

Bu sistemde belli bir cache'ten tahsisat yapmak ve onu serbest bırakmak sabit karmaşıklıkta yani çok hızlıdır. Zaten bu sistemin işletim sistemlerinin çekirdekleri tarafından tercih edilmesinin en önemli nedeni hızlı oluşudur. Dilimli tahsisat sisteminde bir cache'ten yapılan tahsisatlarda içsel bölünme sıfır düzeyindedir. Tüm nesneler aynı uzunlukta olduğu için dışsal bölünmenin de sıfır düzeyinde olduğu söylenebilir. Ancak dilimli tahsisat sistemi uzunluğu baştan bilinen veri yapıları için düşünülmüştür. Dolayısıyla bu sistemde her veri yapısı için ayrı cache'in oluşturulması gerekir. Bu nedenle bu sistem "user mode" programlar için genel ve etkin bir tahsisat sistemi olamaz. (Anımsanacağı gibi genel tahsisat fonksiyonları klasik boş blok bağlı liste tekniğini kullanmakatdır.) Ayrıca belki belirtmemize gerek yok fakat dilimli tahsisat sisteminin gerçekleştirimi biraz zordur.

3. Derleyicilerin ve Yorumlayıcıların Tasarımları ve Gerçekleştirimleri

Bu bölümde derleyicilerin ve yorumlayıcıların tasarım ve gerçekleştirim prensipleri ele alınacak ve açıklanacaktır. Bu bölümün sonunda kurs katılımcılarının çeşitli araçları kullanarak basit yorumlayıcılar yazabilmesi öngörülmektedir.

3.1. Dil Olgusu ve Dillerin Sınıflandırılması

Dil karmaşık bir olgudur. Pek çok bilimin çalışma alanı içerisine girmektedir. (Örneğin dilbilim (linguistics), bilişsel bilimler (cognitive science), psikoloji (psychology), sosyoloji (sociolgy) vs.) Bu nedenle dilin basit bir tanımını yapmak zordur. Pek çok teorisyen ve düşünür değişik tanımlar yapmışlardır. Ancak bu tanımların hepsinde ortak özelliklerden biri dilin "iletişimde kullanılan bir araç" olduğudur. Bir dilin pek çok kural topluluğu söz konusu olabiliyorsa da en temel iki kural kümesinden bahsedilebilir: Sentaks ve semantik. Sentaks ve semantik dili dil yapan ve tüm dillerde var olan ortak özelliklerdir.

Sentaks dili oluşturan en yalın öğelerin (bunlara atom (token) denir) doğru yazılma ve dizilme kurallarıdır. Örneğin aşağıdaki İngilizce cümlede öğeler doğru dizilmemiştir:

```
I going am school to
```

Bu İngilizce'ye göre bir sentaks hatasıdır. Aşağıdaki cümlede de Türkçe'ye göre bir sentaks hatası vardır:

```
Herkez çok neşeliydi
```

"Herkez" sözcüğü yanlış yazılmıştır. Sentaks yalnızca doğal dillerde değil programlama dillerinde de söz konusu olan bir kurallar kümesidir. Örneğin:

```
if )a == 10)
   printf(“ok\n”);
```

Burada if anahtar sözcüğünden sonra ‘(‘ atomunun gelmesi gerekirdi. Halbuki ‘)‘ atomu gelmiştir. Bu da bir sentaks hatasıdır.

Semantik doğru yazılmış ve dizilmiş öğelerin ne anlam ifade ettiğine ilişkin kurallardır. Yani örneğin “I am going to school” doğru yazılmış ve dizilmiştir. Fakat ne anlam ifade etmektedir? Ya da örneğin:

```
if (a == 10)
   printf(“Ok\n”);
```

Doğru yazılmış ve dizilmiştir fakat ne anlam ifade etmektedir?

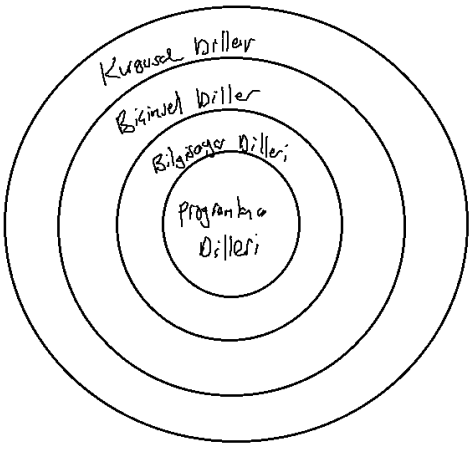
Bir olgunun dil olarak nitelendirilmesi için onun en azından sentaks ve semantik kurallara sahip olması gerekir. Sentaks ve semantik kuralların dışında bazı diller başka kural topluluklarına da sahip olabilirler. Örneğin doğal dillerde “fonetik” teleaffuza ilişkin kurallarla ilgilidir.

3.1.1. Doğal Diller, Kurgusal ve Biçimsel Diller

Yaşantı sonucuyla oluşmuş dillere doğal diller denir. Doğal diller son derece karmaşık sentaks ve semantik yapıya sahiptir. Doğal dillerin sentaks kuralları tam olarak matematiksel biçimde ifade edilememektedir. İnsanların belli bir amaç için tasarladığı dillere ise kurgusal diller (constructed languages) denilmektedir. Sentaksları tam olarak matematiksel biçimde ifade edilebilen diller ise ise biçimsel diller olarak isimlendirilir. Biçimsel dillerde doğal dillerdeki gibi istisnalar yoktur. Çünkü istisnalar aslında sentaks kurallarını bozucu bir etki yapmaktadır. Doğal dilleri öğrenirken kuralları olduğunu sandığımız pek çok yapının aslında çok fazla istisnalarının olduğunu görmüşüzdür. İstisnalar ise öğrenmeyi zorlaştırmaktadır.

3.1.2. Bilisayar Dilleri ve Programlama Dilleri

Bilgisayar sistemlerinde kullanılmak üzere tasarlanmış dillere bilgisayar dilleri 8computer languages) denilmektedir. Yani bir olgunun “bilgisayar dili” olarak nitelendirilmesi için onun bilgisayar dünyası için tasarlanmış olması ve sentaks, semantik kurallara sahip olması gerekir. Bu bakımdan örneğin XML bir bilgisayar dilidir. (XML "Extensible Markup Language" sözcüklerinden kısaltılmıştır.) Bu dilde sentaks ve semantik kurallar vardır. Bu dil bilgisayar sistemleri tarafından kullanılmak üzere tasarlanmıştır. Bilgisayar dillerinde bir “akış (flow)” olması zorunlu değildir. Eğer bir bilgisayar dilinde bir akış da varsa bu tür dillere “programlama dilleri (programming languages)” denilmektedir. Örneğin C bir programlama dilidir, ancak XML bir programalam dili değildir.



Pekiyi örneğin UML (Unified Modeling Language) bir dil midir? İsminden de anlaşılabileceği gibi UML bir dildir. UML'in özellikle nesne yönelimli projeleri modellemek için kullanılan diyagramlardan oluşmuş bir yapısı vardır. Bu diagramları çizmenin bir kuralı vardır. Bu kural UML'in sentaksını oluşturur. Tabii bu diagramların anlamları da vardır. Bu da dilin semantiğini oluşturmaktadır. UML bazılarına göre bir bilgisayar dili olarak nitelendirilebilir bazılarına göre ise kurgusal bir dil olarak sınıflandırılmaktadır (çünkü UML bilgisayarla hiçbir ilgisi olmayan endüstri alanlarında da kullanılmaktadır).

3.1.3. Biçimsel Dillerin Teorik Altyapısı

Eskiden diller matematiksel bir modelle ele alınmıyordu. Biçimsel diller (formal languages) dillerin matematiksel bir bakış açısı ile ele alınması süreci sırasında ortaya çıkmıştır. Bazı çalışmalar çok daha eskiye dayanıyorsa da bu konudaki modern altyapı büyük ölçüde Noam Chomsky'nin çalışmalarıyla oluşturulmuştur. Chomsky dillerin sentakslarını matematiksel terimlerle açıklamış ve "üretici gramer (generative grammar)" kavramını ortaya atmıştır. Gerçekten bilgisayar dillerinin resmi sentakslarını açıklayan BNF ve EBNF gibi notasyonlar Chomsky'nin bu çalışmalarından ilham alınarak geliştirilmişlerdir.

Chomsky dilleri sentaks üretim biçimlerine göre dört bölüme ayırmıştır:

Type 0: Bu dillerin sentaksları oldukça karmaşıktır. Doğal diller bu tür sentaks yapıları içerirler. Bu dillere "serbest (free)" sentakslar da denilmektedir.

Type 1: Bu sentaks biçimine sahip dillere "bağlam bağımlı (context sensitive)" diller de denilmektedir. Bu tür gramerlerde bir öğenin açılımı (ve anlamı) hangi bağlamda bulunduğuna bağlı olarak değişebilmektedir.

Type 2: Bu sentaks yapısına sahip olan dillere de "bağlam bağımsız (context free)" diller de denilmektedir. Bağlam bağımsız dillerin sentakslarında bir öğe hangi bağlamda olursa olsun hep aynı biçimde açılmaktadır (yani hep aynı anlama gelmektedir). Modern programlama dillerinin pek çoğu bağlam bağımsız bir sentaksla ifade edilebilmektedir. Bu nedenle bu dillere "bağlam bağımsız diller (context free langueages)" de denilmektedir.

Type 3: Bu tür sentakslara Chomsky "düzenli gramer (regular grammar)" demektedir. Pek çok kütüphanede kullanılan düzenli ifadeler (regular expressions) düzenli gramere örnek verilebilir.

3.1.4. Biçimsel Dillerin Matematiksel İfadeleri

Diller matematiksel olarak kümeler teorisi kullanılarak ifade edilebilmektedir. Bir dil aslında bir semboller kümesidir. Bir dili oluşturan en yalın elemanlara "son semboller (terminal symbols)" ya da "alfabe (alphabet)" denir. Alfabeyi oluşturan küme sigma işareti ile gösterilmektedir. Örneğin:

$$\sum = \{a, b, c\}$$

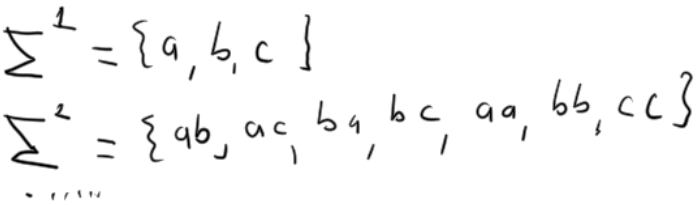
Alfabedeki karakterlerin art arda getirilmesiyle oluşan dizilimlere "string" denilmektedir. Örneğin yukarıdaki sonlu sembol kümesindeki bazı dizilimler şunlar olabilir:

```
aabb
abcca
abccab
aaaaaaaaaa
```

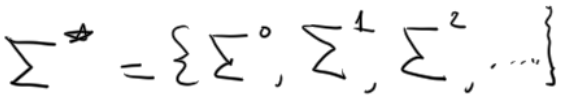
Sonlu semboller kümesi sayılabilir sonlu bir kümedir. Ancak bundan elde edilebilecek tüm string'lerin kümesi sonlu bir küme değildir.

Bir alfabede her zaman boş kümeye karşı gelen bir elemanın bulunduğu kabul edilir. Bu eleman λ ile gösterilmektedir.

Bir alfabedeki n elemanlı string'lerin kümesi sigma karalterinin sağ köşesine n sayısı yazılarak gösterilebilir. Örneğin:



Sigma * tüm bu kümelerin birleşimini ifade eder:



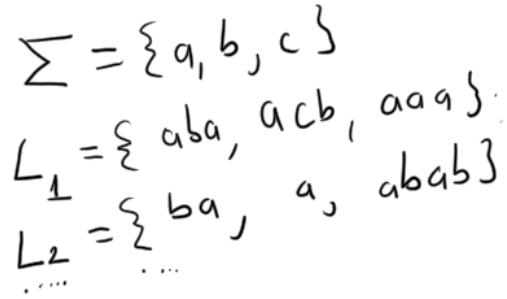
Örneğin:

Sigmanın a ve b elemanlarından oluştuğunu kabul edelim. Bu durumda Sigma * aşağıdaki gibi bir küme olacaktır:

$$\Sigma^* = \{\lambda, a, b, aa, ab, ba, bb, aaa, aab, ...\}\,.$$

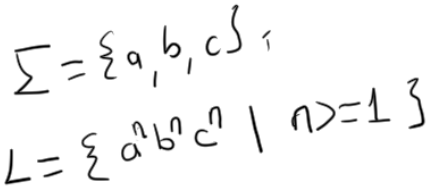
Başka bir deyişle sigma * alfabeden teorik olarak elde edilecek tüm stringlerin kümesidir. Yukarıda da belirttiğimiz gibi sigma * kümesinin eleman sayısı sonsuzdur.

İşte sigma * kümesinin herhangi bir alt kümesine "dil (language)" denilmektedir. Örneğin:



O halde bir dili tanımlayabilmek için öncelikle bir alfabenin tanımlanmış olması gerekir. Örneğin Türkçe için alfabe 29 harften oluşmaktadır. C Programlala Dilinin alfabesi ise pek çok alfa numerik karakterlerden oluşmaktadır.

Bilindiği gibi matematikte bir kümenin eleman sayısı çok fazla ise artık listeleme tekniği gösterim için elverişsiz olmaktadır. Onun yerine ortak özellik (öyle ki) tekniği kullanılır. Örneğin:



Buradaki L dilinin bazı elemanları şunlardır: abc, aabbcc, aaabbbccc, ... Görüldüğü gibi buradaki L dilinin elemanları sonsuz sayıdadır. Sonlu sayıda elemandan oluşan alfabeden sonsuz sayıda elemana sahip dil elde edilebildiğine dikkat ediniz.

Ancak bir dilin eleman sayısı çok fazlaysa ya da sonsuz sayıdaysa onu her zaman ortak özellik yöntemine göre ifade edemeyiz. İşte dillerin elemanlarını belirlemek için Chomsky "üretici gramer (generative grammar)" kavramını ortaya atmıştır.

Şimdi C Programlama Dilini biçimsel bir dil olarak ele almaya çalışalım. C'nin alfabesi temel alfanümerik karakterlerden oluşmaktadır. Bu karakterlerin tüm yan yana getirilmiş hallerinin kümesi sigma *'dır. C'de sigma * kümesinin bir alt kümesidir. C geçerli olan tüm C programlarının (stringlerinin) oluşturduğu bir kümedir. Geçerli olan her C programı C dilini oluşturan kümedeki bir string'tir. Tabii yukarıda da belirttiimiz gibi C gibi karmaşık bir dilin tüm elemanlarını ortak özellik yöntemine göre yazmak mümkün değildir. İşte C gibi bir dilin tüm elemanlarını üretici gramer tekniği ile ifade edebiliriz.

3.1.5. Üretici Gramer (Generative Grammar) Kavramı

Üretici gramer bir dilin tüm elemanlarını üreten bir otomat (automata) olarak düşünülebilir. Üretici gramerin dört öğresi vardır:

```
G = {V, T, S, P}
```

Buradaki harflerin kısaltması şöyledir:

V: Variable (genelikle büyük harflerle gösterilmektedir).Ggrameri ifade ederken kullanılan ara sembolleri belirtmektedir.
T: Terminal Symbol (ya da alfabe). T yerine sigma da kullanılabilir. (Kursumuzda "terminal symbol" yerine Türkçe "son sembol" terimini de kullanacağız.)
S: Start Symbol (variable kümesi içerisindeki bir sembol olmak zorundadır.)
P: Production

Gramer başlangıç sembolünden (start symbol) başlar. Her aşamadan bir üretimle (production) geçilir. Üretimler değişken (variable) denilen ara sembolleri kullanır. Bir üretim bir grup sembolün yerine başka bir sembolün yerleştirilmesi anlamına gelmektedir. Bu biçimde üretim devam ettirilir. Ta ki her şey alfabedeki elemanlardan (yani "terminal symbol"lardan) oluşana kadar. Üretimler bir okla gösterilmektedir. Tipik olarak Ok işaretinin solunda bir değişken, sağında da onun nasıl açılacağı (yani onun yerine ne yerleştiriliceği) bilgisi vardır. Aslında ok işaretinin solunda tek bir değişken bulunmak zorunda değildir. Bağlam bağımlı ve serbest dillerin gramerlerinde ok işaretinin solunda birden fazla ara sembol ve son sembol de bulunabilemktedir.

Şimdi üretici bir gramer örneği verelim:

```
G = {{X}, {a, b}, {X}, P}
```

P üretimleri şunlardır:

```
X -> aXb
X -> λ
```

Bu üretici gramerden ne anlamalıyız? Burada başlangıç sembolü X'tir. Dilin alfabesi (yani sigma kümesi) ise {a, b}'den oluşmaktadır. Değişkenler kümesi de yalnızca X'ten oluşmaktadır. Şimdi biz ok sembolünün solundaki X yerine onun sağındaki dizilimi yerleştirerek bu işleme yalnızca alfabedeki semboller kalana kadar devam edersek dilin bir elemanını buluruz. Bu işlemi her yinelediğimizde de dilin başka bir elemanı elde ederiz. Bu üretici gramer kümesel yöntem yetersiz kaldığı için Chomsky tarafından düşünülmüştür. Örneğin yukarıdaki G dilinin bir elemanı şöyle elde edilebilir:

```
1) X -> aXb     (aXb)
2) X -> λ       (ab)
```

Buradan elde edilen string "ab" dir. Şimdi başka bir elemanı elde edelim:

```
1) X -> aXb     (aXb)
2) X -> aXb     (aaXbb)
3  X -> aXb     (aaaXbbb)
4) X -> λ       (aaabbb)
```

Buradan elde edilen string de "aaabbb" olacaktır. Burada hiçbir sembol içermeyen bir string'in de (bu λ ile gösteriliyor) bu dile dahil olduğuna dikkat ediniz.

Yukarıdaki üretici gramerle açıklanmış olan basit dil aslında -basit olduğu için- ortak özellik yöntemiyle de gösterilebilirdi:

$$G = \{ a^n b^n \mid n >= 0 \}$$

Biçimsel diller ve üretici gramerler konusuyla ilgili tipik soru ve yanıtlar şunlardır:

Soru: Bir dilin alfabesi ne demektir?
Yanıt: Dildeki en yalın elemanların oluşturduğu kümeye "alfabe (alphabet)" ya da "son semboller (terminal symbols)" kümesi denir ve büyük harf sigma karakteri ile gösterilir.

Soru: Biçimsel dil terminolojisindeki string nedir?
Yanıt: Alfabedeki sembollerin peşi sıra getirilmesiyle elde edilen her dizilime string denir. Bir dil aslında string'lerden oluşan bir kümedir.

Soru: $\sum^*$ (Sİgma yıldız) ne anlama gelmektedir?
Yanıt: $\sum^*$ alfabedeki karakterlerin istenildiği kadar birbirleriyle birleştirilmesiyle oluşturulan tüm string'lerin kümesidir. Dil (language) bu kümenin bir alt kümesi olarak tanımlanmaktadır.

Soru: Dil bir string kümesi ise onu nasıl ifade edebiliriz?
Yanıt: Eğer bu kümenin eleman sayısı az ise onu doğrudan listeleme yöntemiyle ifade edebiliriz. Örneğin: L = {ababa, aaaaa, bbab} gibi. Ancak eleman sayısı fazla (hatta sonsuz) olan dillerin listeleme yöntemiyle ifade edilmesi olanaksızdır. Ortak özellik yöntemi ise ancak kuralları çok belirgin olan bazı dilleri ifade edebilmektedir. Örneğin: L = $\{a^n b^n \mid n >= 1\}$. İşte bu iki yöntem yetersiz olduğundan dolayı "üretici gramer (generative grammar)" yöntemi geliştirilmiştir. Bu yöntemde soldaki sembol sağdaki ile açılarak ilerlenir ta ki tüm semboller alfabedeki sembollerden oluşuna kadar.

Soru: Üretici Gramerin resmi ifadesi nasıldır?
Yanıt: Üretici gramerin resmi gösterimi G = {V, T, S, P} biçimindedir. Burada V açılacak başlangıç sembolünü belirtir. T ise alfabeyi belirtmektedir. S açılımın başlatılacağı semboldür. P de açılımda kullanılacak

üretimlerin kümesidir. Başlangıç sembolünden başlanarak P üretimlerimleri yoluyla elde edilen açılımlardaki tüm semboller yalnızca alfabedeki semboller olana kadar işlem devam ettirilir.

Soru: Üretici gramer bir dildeki stringlerin kümesini nasıl oluşturmaktadır?
Yanıt: Başlangıç sembolünden hareketle açılım yapıla yapıla elde edilen tüm stringler dili oluşturmaktadır. Benzer biçimde elde edilemeyen stringler de o dilin bir elemanı değildir.

Soru: Bir programlama dili (örneğin C için) yukarıdaki tanımların anlamı nedir?
Yanıt: Aslında C sonsuz sayıda geçerli C programlarının oluşturduğu kümedir. Yani geçerli bir C programı yukarıdaki terminolojiye göre bir string'tir. C standartlarında dili oluşturan tüm üretimler listelenmiştir.

Soru: Yukarıdaki bilgilerin derleyici tasarımı ve gerçekleştirimi ile ilgisi nedir?
Yanıt: Derleyici tasarımını ve gerçekleştirimini anlayabilmek için daha teknik yaklaşımların kullanılması gerekmektedir. Bu nedenle yukarıdaki terminoloji ve biçimsel dillerle ilgili malzemeleri biz tasarım ve gerçekleştirim sürecinde kullanacağız. Ayrıca derleyici yazımında kullanılan araçların pek çoğu grameri bizden üretici gramer biçiminde istemektedir.

3.1.6. BNF ve EBNF Notasyonları

Üretici gramer fikri Noam Chomsky tarafından 1956'da ortaya atılmıştır. Fakat Chomsky bir bilgisayar bilimcisi değildi. Daha çok doğal diller ve biçimsel diller konusunda çalışmalar yapıyordu. Dünyanın ilk yüksek seviyeli programlama dili olan Fortan 1954-1957 yılları arasında geliştirildi. Fortan'ın tasarımı büyük ölçüde John Backus tarafından yapılmıştı. Fortran'dan sonra onu Algol dili (Algol 60) izledi. Algol de John Backus ve Peter Naur gibi kişilerin öncülüğünde geliştirildi. İşte Backus ve Naur birbirlerinden bağlantısız bir biçimde Chomsky'nin üretici gramerinden esinlenerek programlama dilleri için üretici gramer yöntemleri geliştirdiler. İkisinin ayrı ayrı geliştirdiği bu yöntemler birbirlerine de benziyordu. Bu yöntemler daha sonraları birleştirilerek BNF (Backus- Naur Form) notasyonu doğdu. Bugün programlama dillerinin resmi gramerleri BNF notasyonu ve onun türevleriyle ifade edilmektedir. BNF notasyonu ISO tarafından genişletilerek EBNF (Extended BNF) ismiyle standardize edilmiştir. (Kurs dokümanlarında bu standartları bulabilirsiniz). Ancak C, C++, Java, C# gibi dillerin sentaksları EBNF ile değil klasik BNF notasyonunun türevleriyle açıklanmış durumdadır.

3.1.6.1. BNF Notasyonun Temel Özellikleri

BNF notasyonu standardize edilmediği için birbirlerine benzeyen pek çok biçimi kullanılmaktadır. Notasyonun temel özellikleri şöyledir:

1) Üretimlerde açılacak ara sembollerden sonra ':' karakteri yerleştirilir. Bunun yanına ya da aşağısına da açımdan elde edilecek hedef semboller yerleştirilmektedir. Örneğin:

```
declaration:
    declaration-specifiers init-declarator-listopt ;
```

2) Semboller son semboller (terminal symbols) ve ara semboller (non terminal symbols) olmak üzere ikiye ayrılmaktadır. Son semboller atomları (ya da alfabedeki elemanları) belirtir. Genellikle ara semboller italik olarak yazılırlar. Son semboller ise genellikle bold bir biçimde belirtilmektedir. Son sembollerin bold yerine tek tırnak içerisinde yazılması da yaygındır.

3) Bir ara sembolün seçenekleri ya '|' sembolleriyle ya da alt alta belirtilir. Örneğin:

Digit:
    '0'
    '1'
    '2'

‘3’
‘4’
‘5’
‘6’
‘7’
‘8’
‘9’

Diğer bir yazım biçimi şöyle olabilirdi:

Digit: ‘0’ | ‘1’ | ‘2’ | ‘3’ | ‘4’ | ‘5’ | ‘6’ | ‘7’ | ‘8’ | ‘9’

4) Bir sembolün olup olmamasının isteğe bağlı olması genellikle alt indis olarak “opt” sözcüğü ile ifade edilmektedir. Örneğin:

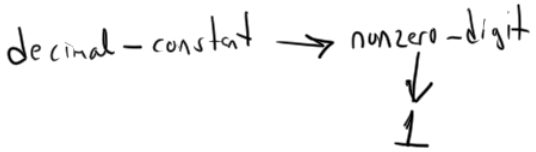
```
declaration:
    declaration-specifiers init-declarator-listopt ;
```

Burada “declaration” “declarator-specifiers” ve isteğe bağlı bir “init-declararor-list”ten oluşmaktadır. Yani başka bir deyişle “declaration”da “init-declarator-list” bulunmak zorunda değildir.

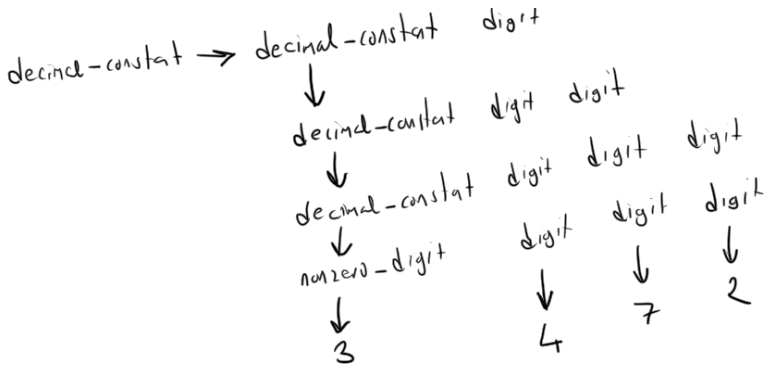
Şimdi BNF için bazı örnekler verelim:

```
decimal-constant:
    nonzero-digit
    decimal-constant   digit

nonzero-digit: '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'

digit: '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'
```

Şimdi "decimal-constant" başlangıç ara sembolünden hareketle çeşitli açımlar yapalım:



Burada 1 sayısı açılımdan elde edilmiştir. Örneğin:

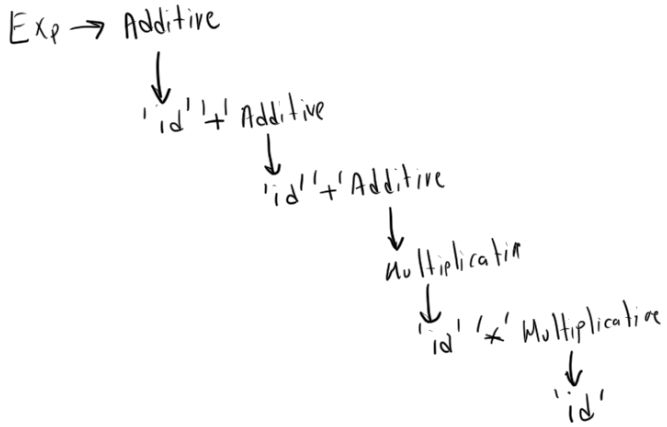


Burada 3472 sayısı elde edilmiştir. "decimal-constant" gramerinde başı sıfır ile başlamayan her türlü tamsayının elde edilebileceğine dikkat ediniz.

Şimdi aşağıdaki gramere bakalım:

```
Exp:
    Additive

Additive:
    ‘id’ ‘+’ Additive
    Multiplicative

Multiplicative:
    ‘id’ ‘*’ Multiplicative
    ‘id’
```

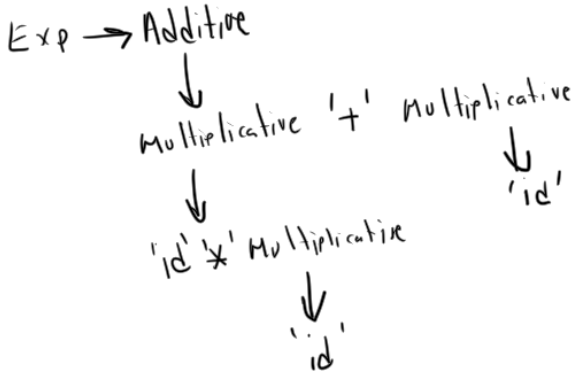
Bu gramerde başlangıç sembolü “Expression” sembolüdür. Burada biraz tersten giderek şu çalışmayı yapalım. Yukarıdaki gramer 'id' + 'id' + 'id' * 'id' açılımını yapabilir mi? Evet aşağıdaki gibi bir açılımla istenilen ifade elde edilebilir.



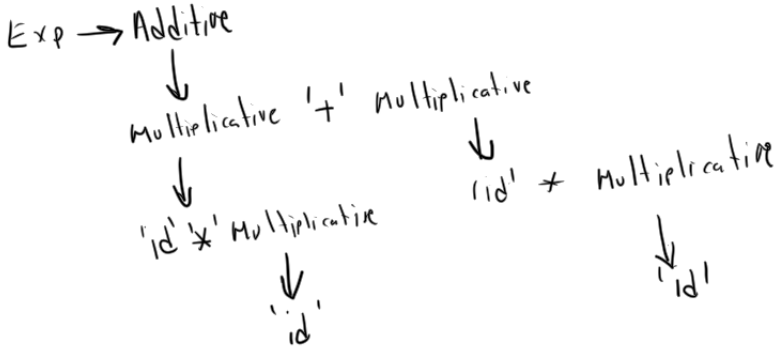
Pekiyi yukarıdaki "Expression" grameri ile 'id' * 'id' + 'id' ifadesi açılabilir mi? Yanıt hayır! Bu grameri aşağıdaki gibi değiştirelim:

```
Exp:
    Additive

Additive:
    Multiplicative ‘+’ Multiplicative
    Multiplicative

Multiplicative:
    ‘id’ ‘*’ Multiplicative
    ‘id’
```

Şimdi bu gramerle ile 'id' * 'id' + 'id' ifadesini açmaya çalışalım:



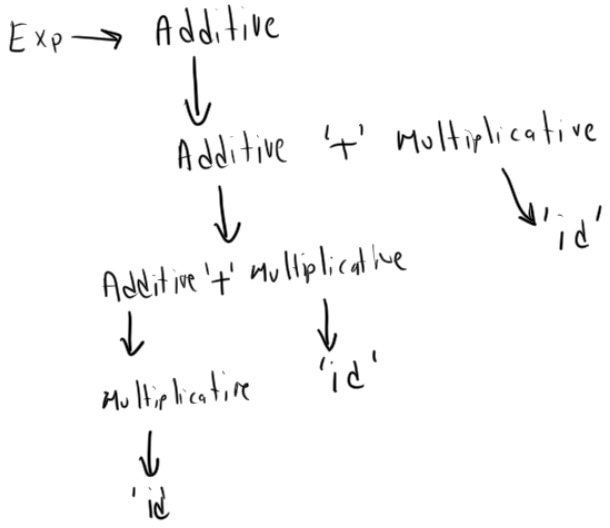
Şimdi de 'id' * 'id' + 'id' * 'id' ifadesini açmaya çalışalım:



Şimdi de 'id' + 'id' + 'id' ifadesini bu gramerle açmaya çalışalım. Açabilir miyiz? Yanıt hayır! Pekiyi grameri biraz daha değiştirelim:

```
Exp:
    Additive

Additive:
    Additive ‘+’ Multiplicative
    Multiplicative

Multiplicative:
    ‘id’ ‘*’ Multiplicative
    ‘id’
```

Artık 'id' + 'id' + 'id' ifadesini açabiliriz:



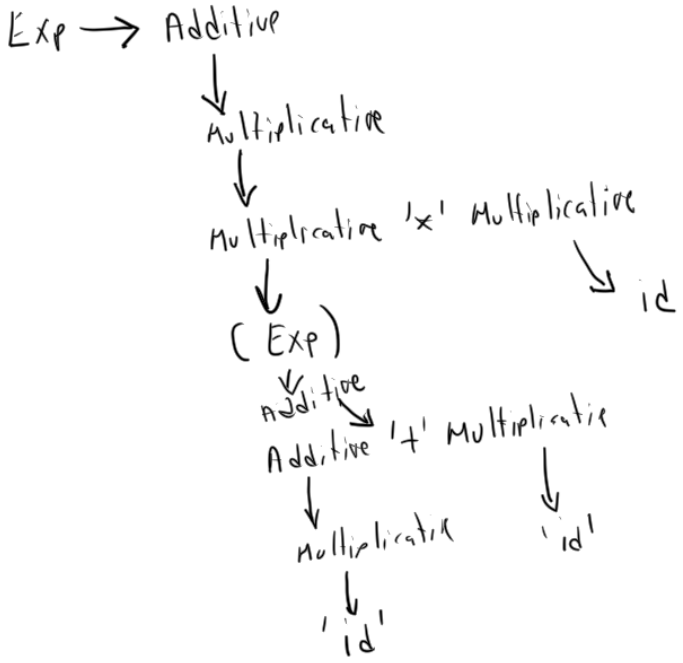
Bu son gramerin aşağıdaki tüm ifadeleri açabileceğine dikkat ediniz:

```
'id'
'id' + 'id' * 'id'
'id' * 'id' * 'id'
'id' + 'id' + 'id'
'id' * 'id' + 'id' * 'id'
```

Şimdi grameri biraz daha geliştirelim:

```
Exp:
    Additive

Additive:
    Additive ‘+’ Multiplicative
    Additive ‘-‘ Multiplicative
    Multiplicative

Multiplicative:
    Multiplicative ‘*’ Multiplicative
    ‘id’
    ‘(‘ Exp ’)’
```

Şimdi de ('id' + 'id') * 'id' bir ifadeyi bu gramerle açmaya çalışalım:



Bu gramerle aşağıdaki ifadelerin hepsinin açılabileceğine dikkat ediniz:

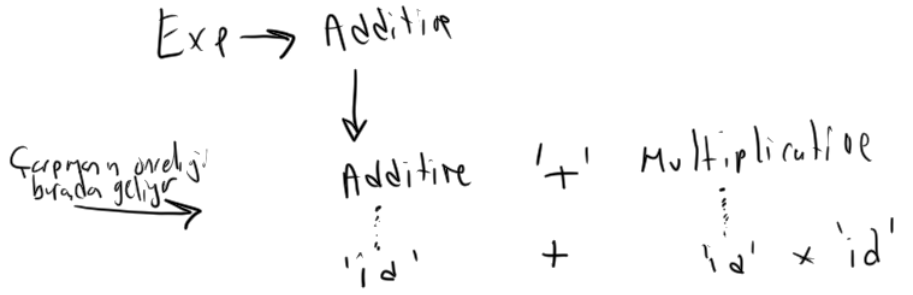
```
'id'
'id' + 'id'
'id' * ('id' + 'id')
('id' + 'id') * ('id' + 'id')
('id' * ('id' + 'id')) * ('id' + 'id')
```

Bu gramer programlama dillerindeki “ifade” kavramını bire bir yansıtmaktadır.

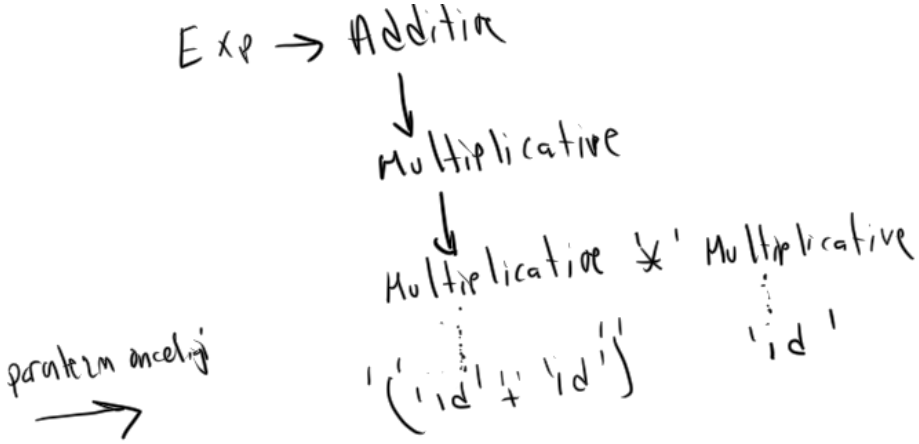
Programlama dillerindeki işlem önceliği ve operatörlerin öncelik tablosu aslında programcıların kolay anlaması için uyfurulan kavramlardır. Gerçekten de C ve C++ standartlarında operatör önceliği diye bir kavram yoktur. Gramerin kendisi zaten bizzat bu önceliği belirtmektedir. Şöyle ki:

```
'id' + 'id' * 'id'
```

Bu ifade yukarıdaki gramere göre açıldığında 'id' ile 'id' * 'id'nin toplanacağı anlaşılır. id + id ile id'nin çarpılması bu gramere göre zaten münkün değildir. Çünkü açım bu gramere göre ancak şöyle yapılabilir:



Parantezlerin önceliği de gramerin içerisinden çıkartılacak bir sonuçtur. Örneğin ('id' + 'id') * 'id' ifadesinde açılım ancak şöyle olabilir:



Başka bir deyişle operatör önceliği zaten gramer tarafından dolaylı olarak belirtilmiş durumdadır. Ayrıca bir öncelik tablosuna gereksinin yoktur. Gerçekten de C ve C++ standartlarına baktığınızda operatör önceliği diye bir konunun olmadığını göreceksiniz. Çünkü örneğin 'id' + 'id' * 'id' ifadesi 'id' + 'id' ile 'id'nin çarpımı biçiminde bu gramer tarafından açılamamaktadır. Yani bu ifade aslında yukarıdaki gramere göre şöyle açılmaktadır:

```
İ1: id * id
İ2: id + İ1
```

C Programlam dilinin standartlarında belirtilen başlangıç sembolü "translation-unit" isimli semboldür:

```
translation-unit:
    external-declaration
    translation-unit external-declaration
```

Buradan "translation-unit" (yani kaynak dosya) sembolünün bir ya da birden fazla "external-declaration"dan oluştuğunu söyleyebiliriz. external-declaration sembolü de şöyle belirtilmiştir:

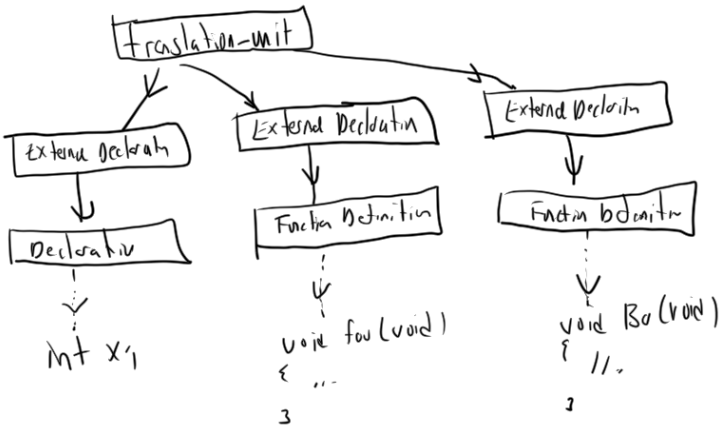
```
external-declaration:
    function-definition
    declaration
```

Görüldüğü gibi bir "external-decalaration" ya bir fonksiyon tanımlasından ya da bir bildirimden oluşmaktadır. Örneğin aşağıdaki bir C programı olsun:

```
int x;
```

```
void foo(void)
{
    ...
}

void bar(void)
{
    ...
}
```

Bu program gramerden şöyle elde edilebilir:



3.1.6.2. EBNF Notasyonunun Temel Özellikleri

Yukarıda da belirttiğimiz gibi BNF Notasyonu standardize edilmemişti. ISO BNF notasyonunu geliştirerek EBNF (Extended BNF) ismiyle standardize etti (ISO/IEC-14977: 1996). EBNF yazımı kolaylaştımrak için birtakım meta karakterler kullanmaktadır. Meta karakterler son sembol olmayan, özel anlama gelen karakterlerdir.

EBNF notasyonun temel özellikleri şunlardır:

1) Ara sembolü açıklamak için ‘=’ meta karakteri kullanılmaktadır.

2) Son semboller (terminal symbols) iki tırnak içerisinde belirtilirler. Ara sembol tanımlamasını bitirmek için ‘;’ meta karakteri yan yana gelen atomları ayırmak için ise ‘,’ meta karakteri kullanılmaktadır.

3) Seçenekler BNF'de olduğu gibi yine ‘|’ meta karakterleriyle belirtilirler. Örneğin:

```
digit = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9" ;
```

4) ‘? ‘ metakarakteri tek bir ifadenin isteğe bağlı olduğunu, ‘*’ meta karakteri ilgili ifadenin sıfır tane ya da daha fazla yineleneceğini, ‘+’ meta karakteri ise ilgili ifadenin bir ya da daha fazla yinelenceğini belirtmektedir.

5) Köşeli parantezler içerisindeki öğeler isteğe bağlı (optional) öğeleri belirtir. Gruplama için normal parantezler kullanılır.

5) Tekrarlamalar EBNF’de daha kolay bir biçimde küme parantezleriyle belirtilmektedir. Küme parantezleri içerisindeki öğeler bir ya da birden fazla kez yinelenebilir. Örneğin:

```
identifier = letter , { letter | digit | "_" }
```

Burada "identifier", " letter" ve "digit" ara sembollerdir. Bir "identifer" tek bir "letter"dan oluşabilir. Ya da bir "letter" ile başlayıp bir "letter", "digit" ya da "_" karakterlerinden istenildiği kadar yan yana getirilerek oluşturulabilir.

Aşağıdaki "en.wikipedia.org"den alınmış örneği inceleyiniz:

```
letter = "A" | "B" | "C" | "D" | "E" | "F" | "G"
       | "H" | "I" | "J" | "K" | "L" | "M" | "N"
       | "O" | "P" | "Q" | "R" | "S" | "T" | "U"
       | "V" | "W" | "X" | "Y" | "Z" | "a" | "b"
       | "c" | "d" | "e" | "f" | "g" | "h" | "i"
       | "j" | "k" | "l" | "m" | "n" | "o" | "p"
       | "q" | "r" | "s" | "t" | "u" | "v" | "w"
       | "x" | "y" | "z" ;

digit = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9" ;

symbol = "[" | "]" | "{" | "}" | "(" | ")" | "<" | ">"
       | "'" | '"' | "=" | "|" | "." | "," | ";" ;

character = letter | digit | symbol | "_" ;


identifier = letter , { letter | digit | "_" } ;

terminal = "'" , character , { character } , "'"
         | '"' , character , { character } , '"' ;


lhs = identifier ;

rhs = identifier
     | terminal
     | "[" , rhs , "]"
     | "{" , rhs , "}"
     | "(" , rhs , ")"
     | rhs , "|" , rhs
     | rhs , "," , rhs ;


rule = lhs , "=" , rhs , ";" ;

grammar = { rule } ;
```

Bu gramerden şu sonuçları çıkartabiliriz: Buradaki gramer "rule"lardan oluşmaktadır. Bir rule kabaca şu biçimdedir:

```
lhs = rhs;
```

lhs (left hand side) identifier olmak zorundadır. Ancak rhs çeşitli biçimlerde açılabilmektedir.

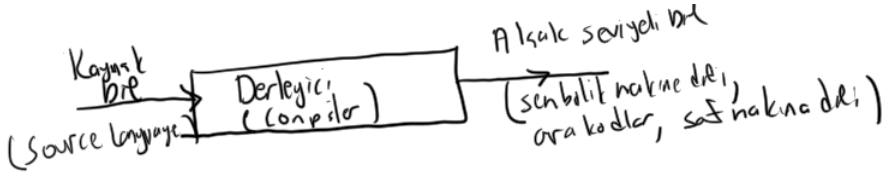
XML standartlarında (Ecma-357) gramer EBNF ile açıklanmıştır. Bu standartları gözden geçirerek EBNF notasyonu konusunda uzmanlaşabilirsiniz.

3.2. Programlama Dilleri Üzerinde İşlemler Yapan Araçlar

Bir programlama dilinde yazılmış olan bir programı eşdeğer olarak başka bir programlama diline çeviren programlara "çevirici programlar (translators)" denilmektedir.

Bir çevirici programda çevrilmek istenen programın diline “kaynak dil (source language)”, çevirme sonucunda elde edilen programın diline de “hedef dil (target language)” denilmektedir. Örneğin bir C# programını eşdeğer olarak VB.NET programına dönüştüren çevirici programın kaynak dili C#, hedef dili ise VB.NET’tir. Şüphesiz her dili her dile dönüştüren bir çevirici programın yazılabilmesi mümkün olmayabilir.

Bir çevirici programda hedef dil alçak seviyeli bir dilse (yani sembolik makine dili, ara kod ya da saf makine dili) böyle çevirici programlara “derleyici (compiler)” denilmektedir.



Gördüğünüz gibi bir çevirici programın derleyici biçiminde isimlendirilmesi kaynak dile değil hedef dile bağlıdır. Örneğin sembolik makine dilinde yazılmış programları da saf makine diline dönüştüren programlara derleyici denilmektedir. Benzer biçimde Java ve .NET platformlarında bu platformların arakodlarını gerçek makine kodlarına dönüştüren sistemler de derleyici olarak tanımlanırlar. Bu tür derleyicilere özel olarak “Just In Time Compiler” denilmektedir. Özel olarak sembolik makine dilinde yazılmış programı saf makine diline dönüştüren programlara “assembler” da denilmektedir. “Assembler” terimi aslında sembolik makine dili derleyicisi anlamına geliyorsa da bu terim yanlış kullanımlarla "sembolik makine dilinin (assembly language)” kendisini de anlatır hale gelmiştir.

Alçak seviyeli dilleri girdi olarak alıp bunları yüksek seviyeli dillere dönüştüren çevirici programlara “decompiler” denilmektedir. "Decompiler"ların işlevsel olarak derleyicilerin tam tersi bir işlemi yaptıklarına dikkat ediniz. Örneğin .NET’in arakodunu yeniden C#’a dönüştüren pek çok “decompiler” vardır (Reflector, ILSpy, Salamander, Dis# gibi). Maalesef saf makine dillerinen yüksek seviyeli dillere etkin dönüştürme yapan "decompiler"lar etkin biçimde yazılamamaktadır.

Saf makine dilinden sembolik makine dillerine dönüştürme yapan çevirici programlara “disassembler” denilmektedir. Örneğin çalıştırılabilen bir programı sembolik makine dilinde bize gösteren pek çok "disassember" vardır. Ancak “disassembler” terimi daha çok yalnızca makine komutlarının dönüşümünü yapan programlar için tercih edilmektedir. Aslında pek çok kaynak saf makine dilinden belli bir sembolik makine dili çıktısı üreten programları belirtmek için yine “decompiler” terimini tercih etmektedir.

Bazen derleme işleminin yapıldığı makinenin işlemci ailesi ile derleme sonucunda üretilen kodun çalıştırılacağı işlemci ailesi birbirinden farklı olabilmektedir. İşte bu biçimde çalıştırıldığı ortamdaki işlemcinin kodunu değil de başka bir işlemcinin kodunu üreten derleyicilere “çapraz derleyiciler (cross compilers)” denilmektedir. Örneğin kişisel bilgisayarlarımızda Intel ailesi işlemciler kullanılıyor. Biz böyle bir bilgisayarda cep telefonlarında çalıştırılmak üzere ARM kodu üreten bir derleyici kullanıyorsak bu derleyici bir çapraz derleyicidir. Örneğin mikrodenetleyici kodları hemen her zaman çapraz derleyicilerle derlenmektedir.

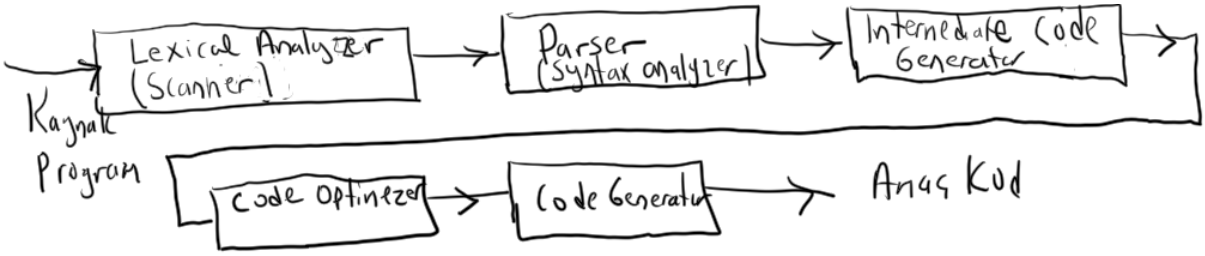
Yorumlayıcılar (interpreters) çevirici programlar değillerdir. Yani yorumlayıcılar kod üretmezler. Bir yorumlayıcı kaynak kodu okur, onu hiç kod üretmeden doğrudan çalıştırır.

Anhtar Notlar: “Translator” İngilizce çevirmen anlamına gelmektedir. Yazılı metni çeviren kişilere çevirmen denir. Interpreter ise İngilizce “mütercim tercüman” anlamına gelmektedir. Konuşmayı çevirenlere mütercim tercüman denir. İngilizce “interpreter” aslında mütercim tercüman anlamından yazılım dünyasına uyarlanmıştır. Fakat biz “interpret” sözcüğünün Türkçe karşılığı için "mütercim tercüman" yerine “yorumlayıcı” sözcüğünü kullanacağız.

3.3. Derleme ve Yorumlama İşleminin Aşamaları

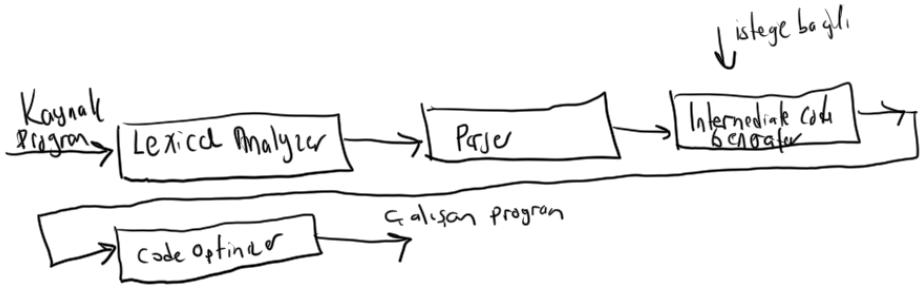
Hem derleyiciler hem de yorumlayıcıların gerçekleştirimlerinde birtakım ortak aşamalar vardır. Ancak yorumlayıcılar kod üretimi yapmadıkları için onların gerçekleştirilmeleri derleyicilere göre daha kolaydır.

Bir derleyici tipik olarak şu aşamalardan geçilerek gerçekleştirilmektedir:



Kaynak kod derleyicinin "lexical analiz (lexical anaylzer)", "tarayıcı (scanner)" ya da "atom ayrıştırıcısı (tokenizer)" denilen modülü tarafından ele alınır. Bu modülün görevi kaynak kodu atomlarına ayırmaktır. Bu modülden sonra "parser" modülü devreye girer. "Parser" modülüne "syntax analyzer" da denilmektedir. Parser modülü atomları girdi olarak alır ve bu atomların dizilişlerinin dilin gramerine uygun olup olmadığını denetler. Parser modülü ürün olarak "parse ağacı (parse tree)" denilen bir ağaç oluşturmaktadır. "Parse ağacı" kaynak kodun işlenebilir bir veri yapısı haline dönüştürülmüş biçimidir. Yani program artık bir yazı olmaktan çıkmış bir veri yapısı olarak ifade edilmiştir. Sonraki modül parse ağacını özyinelemeli biçimde dolaşır ve ağaçtaki elemanlar için arakodlar (intermediate codes) üretir. Arakod gerçek makine kodu değildir. Onu temsil eden bir geçiş kodudur. Bu işlemi yapan modüle "ara kod üreticisi (intermediate code generator)" denilmektedir. Daha sonra bu ara kodlar optimize edilir. Bu modül de "kod eniyileyicisi (code optimizer)" olarak isimlendirilmektedir. Nihayet optimize edilmiş arakodlardan gerçek makine kodları oluşturulur. Bu modül de "kod üreticisi (code generator)" olarak isimlendirilmektedir.

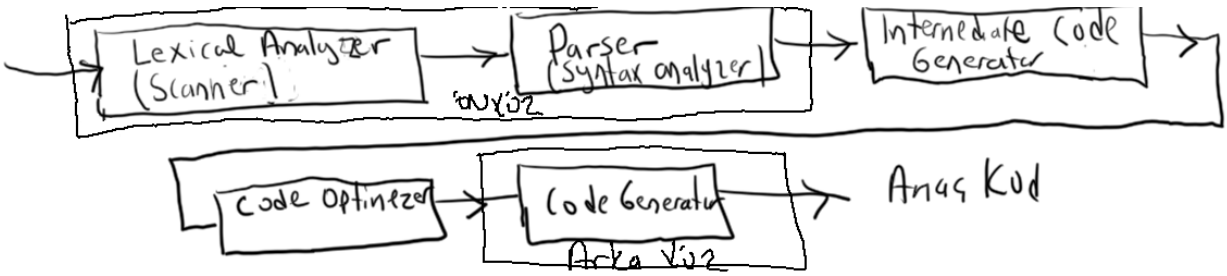
Yorumlayıcılar da benzer aşamalardan geçerek gerçekleştirilirler. Aslında bir yorumlayıcı "kod üretici (code generator)" kısmı olmayan bir derleyici gibi düşünülebilir. Bazı yorumlayıcılar hiç arakod üretmezler, doğrudan parse ağacı üzerindeişlem yaparlar.



Derleyicilerin kaynak program üzerinde işlem yaptığı modüllere "ön yüz (frontend)", makine kodunu ürettiği modüllere "arka yüz (back end)" ve aradaki diğer modüllere de "orta yüz (middle end)" denilmektedir. Ön yüz kaynak dile, arka yüz ise hedef dile bağlıdır. Derleyicileri yazanlar önyüzlerin çıktılarını (parse ağacını vs.) standart hale getirmeye çalışırlar. Böylece derleyicinin port edilmesi kolaylaşır. Şöyle ki: Bir firma düşünelim. Bu firma N tane dili M tane makine dili için derleyecek derleyiciler yazmak istesin. Normalde bunun için kaç derleyicinin yazılması gerekir? Yanıt: N * M tane değil mi? Halbuki firma N tane dil için önyüz, M tane dil için de arka yüz yazarsa toplamda N + M tane faaliyetle bunları birleştirebilir.
Örneğin biz bir dil tasarlamış olalım. Ancak bunun derleyicisi için GCC derleyicisinden faydalanmak isteyelim. Mademki GCC işin önemli kısmını zaten yapmaktadır. O halde biz GCC için kendi dilimizi atomlarına ayıran ve onu parser modülüne veren bir önyüz yazabiliriz. Bu durumda kod optimizasyonunu ve kod üretimini GCC'nin zaten var olan modülleri yapacaktır. İşte bu faaliyete GCC için "fron-tend" yazma faaliyeti denilmektedir. Şimdi biz yeni bir işlemci için C derleyicisi yazmak isteyelim. Ve bunun için yine GCC

derleyicisinden faydalanmak isteyelim. Bu durumda GCC'nin C frontend'i doğrudan kullanılabilir. Fakat bizim GCC için bir hedeflediğimiz işlemcinin kodunu üreten bir "arkayüz (back-end)" yazmamız gerekir.

Derleyicilerin ya da yorumlayıcıların "Lexical analiz" ve "Parser" modülleri onların önyüzlerini oluşturmaktadır. "kod üretici modülleri ise onların arka yüzünü oluşturur. Geri kalan kısımlar modüller ise orta yüze ilişkindir. Yorumlayıcılar kod üretmimi yapmakdıkları için bunların arkayüzleri yüzlerinin olmadığını yeniden anımsatalım:



Bu konuda sıkça sorulan sorular ve yanıtları şöyledir:

Soru: Derleyicilerde ve yorumlayıcılarda önyüz (frontend) nedir?
Yanıt: Önyüz derleyicilerin ve yorumlayıcıların kaynak dil üzerinde işlem yapan modüllerdir. Kaynak kodun atomlarına ayrılması ve parse edilmesi doğrudan kaynak dilin sentaksı ve semantiği ile ilgilidir. Bu nedenle "lexical analiz" ve "parser" modülleri önyüze ilişkin modüllerdir.

Soru: Derleyicilerde arkayüz nedir?
Yanıt: Arkayüz hedef koda yünelik işlem yapan modülden oluşur. Bu da tipik olarak "kod üretici (code generator)" modülüdür. Önyüzü yazmak için kaynak dili, arkayüzü yazmak için ise hedef dili bilmek gerekir.

Soru: Derleyicilerin gerçekleştirme aşaması bakımından yorumlayıcılardan ne farkı vardır?
Yanıt: Derleyiciler arkayüze sahiptir halbuki yorumlayıcılar kod üretmedikleri için sahip değildir.

Soru: Derleme ve yorumlama sürecindeki arakod kavramı nedir?
Yanıt: Arakod gerçek hedef kodu temsil eden fakat hedef koddan bağımsız bir koddur. Optimizasyonların çoğu arakodlar üzerinde yapılmaktadır. Böylece derleyiciler port edilirken arakod değişmeyeceği için optimizasyon işlemleri büyük ölçüde hedef koddan bağımsız hale getirilmiş olur.

Soru: Bir derleyiciyi "port etmek" ne anlama gelmektedir?
Yanıt: Belli bir mikroişlemci için hedef kod üreten derleyiciyi başka bir işlemci için hedef kod üretecek hale getirme sürecine "port etmek" denilmektedir. Örneğin GCC derleyicileri yalnızca Intel ailesi için değil pek çok mikroişlemci ailesi için kod üretir durumdadır. Ancak port etmek önyüz değiştirme faaliyeti için de kullanılabilmektedir.

3.4. Kaynak Kodun Atomlarına Ayrılması (Lexical Analysis / Scanning / Tokenizing)

Derleyicilerin ve yorumlayıcıların ilk aşaması kaynak kodun atomlarına ayrılmasıdır. Bu sürece İngilizce "lexical analysis", "scanning" ya da "tokenizing" denilmektedir. Derleyicilerde bu süreci gerçekleştiren modüller de benzer biçimde "lexical analyzer", "scanner" ya da "tokenizer" biçiminde isimlendirilmektedir. Biz kursumuzda bu sürece yarı İngilizce yarı Türkçe "lexical analiz" diyeceğiz. Derleyiciler ve yorumlayıcılar tarafından kaynak kod yalnızca "lexical analiz" aşamasında okunmaktadır. Dolayısıyla bu aşama bir dosya işlemi gerektirdiğiiçin bu işlemlerin yürütülme biçimi de derleyici ya da yorumlayıcının performansını etkilemektedir.

“Lexical analiz” modülünün aslında tek bir fonksiyondan oluştuğunu söyleyebiliriz. Bu fonksiyon kaynak kodda kalınan yerden sonraki ilk atomun karakterlerini ve türünü bize verir. Bu işlemi yapan fonksiyonun arayüzü şöyle olabilir:

```
char g_token[MAX_TOKEN];

int GetNextToken(void);
```

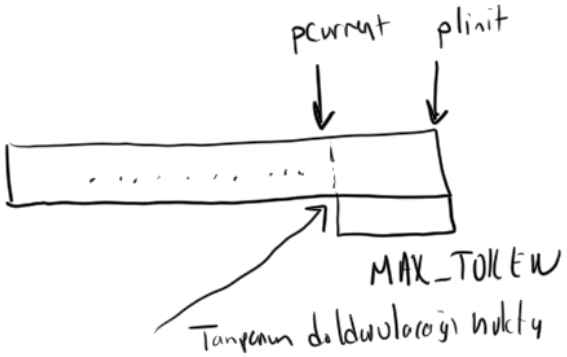
Burada GetNextToken bize kalınan yerden sonraki ilk atomu veren fonksiyondur. Bu fonksiyon atomun yazısını g_token isimli char türden diziye yerleştirip atomun türüyle geri dönmektedir. Tipik olarak atom türlerinin her biri bir sayıyla temsil edilmektedir. Tabii yukarıdaki verdiğimiz arayündeki fonksiyonların ve global değişkenin isimlendirme ve harflendirme (capitalization) biçimine takılmayınız. Bu isimlendirme biçimlerini istediğiniz gibi değiştirebilirsiniz. Bu arayüzde global bir nesnenin kullanılması da sizi şaşırtmasın. Bildiğiniz gibi global nesnelere erişim bilgisayar zamanı bakımından daha hızlıdır. Bu nedenle derleyici aşamalarının etkin bir biçimde gerçekleştirilmesi için global nesnelerden faydalanılmaktadır. Tabii aşağıdaki gibi bir arayüz de söz konusu olabilirdi:

```
int GetNextToken(char *token);
```

Bu arayüzde GetNextToken fonksiyonu sıradaki atomu parametresiyle aldığı adrese yerleştirmektedir. Fakat biz daha çok birinci arayüzü tercih edeceğiz.

3.4.1. Atomlara Ayırma İşlemi İçin Tamponlama Mekanizması

Lexical analiz modülü kaynak kodu mümkün olduğunca etkin bir biçimde okumalıdır. Oysa karakterleri fgetc ya da getc gibi bir fonksiyonla tane tane okumak (her ne kadar standart C fonksiyonları tamponlama yapıyorsa da) göreli olarak zaman kaybına yol açar. Bu nedenle bir tamponlama mekanizmasının kullanılması tercih edilir. Yani dosyanın karakterleri bir tampona okunur, lexical analiz modülü de bu tampondan karakterleri alır. Tampondaki karakterler bitince tampon yeniden doldurur. Ancak lexical analiz modülü atomlarına ayırma işlemi sırasında atomu tespit edebilmek için sonraki karakterlere de bakmak zorunda kalabilmektedir. İşte bir atomun bazı karakterlerinin tamponda olması bazılarının olmaması gerçekleştirimi zorlaştırmaktadır. Bu nedenle tamponda en az bir atomun uzunluğu kadar karakterin bulundurulması yoluna gidilir. Yani başka bir deyişle tamponun doldurulması tamponun sonuna gelince değil sondan MAX_TOKEN gibi bir seviyeye gelince yapılır.

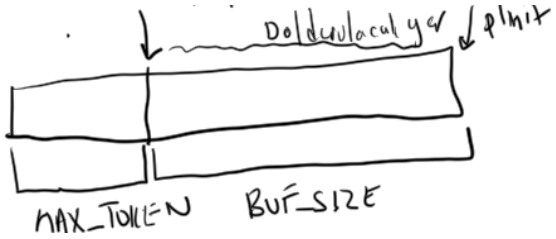


Şekildeki pcurrent tamponda kalınan yeri, plimit de tamponun sonunu göstermektedir. Tamponun doldurulması şöyle bir kontrolle yapılabilir:

```
if (plimit - pcurrent < MAX_TOKEN) {
    FillBuffer();
    ...
}
```

Pekiyi buaradaki MAX_TOKEN hangi değerde olmalıdır? Örneğin C'de bir atomun maksimum uzunluğu nedir? İşte pek çok atomun uzunluğu bir ya da birkaç karakterdir. Ancak istisna olarak değişkenlerin (identifer), sabitlerin (literals), iki tırnak ifadelerinin (strings) ve yorum (remark) kısımlarının uzunlukları çok fazla olabilmektedir. Bu dört istisna durum özel durumlar olarak değerlendirilebilir. Dolayısıyla MAX_TOKEN bu üç atom grubununun dışındaki atomların maksimum uzunluğudur.

Pekiyi tamponun doldurulması nasıl yapılmalıdır? Burada kullanılan tampon bir kuyruk sistemine benzetilebilir. Dolayısıyla tamponun gerçekleştirimi döngüsel bir kuyruk sistemi ile yapılabilir. Ancak döngüsel kuyruk sistemlerinde her karakterde tamponun sonuna gelindi mi diye bir kontrolün yapılması gerekmektedir. İşte bu kontrolden kurtulmak için genellikle tamponun sonındaki MAX_TOKEN kadar karakter tamponun başına kopyalanıp oradan devam edilir. Pekiyi okuma ne kadar uzunlukta yapılacaktır? Dosya okumalarının belli değerlerin katları (örneğin sektörlerin) kadar yapılması okuma verimliliğini yükseltebilmektedir. Dolayısıyla okuma miktarı BUF_SIZE ile temsil edilirse bizim bu kadarlık okumayı yapabilmemiz için gerçek tamponun bundan MAX_TOKEN kadar daha büyük olması gerekir.

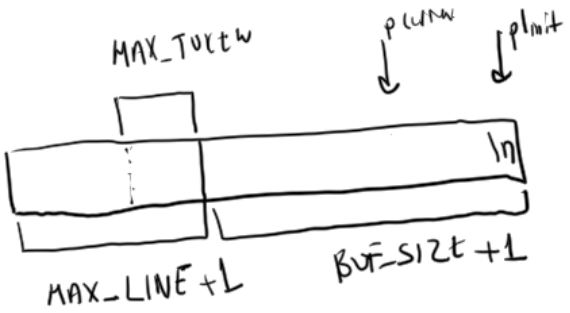


Burada bir noktaya dikkatinizi çekmek istiyoruz.-Dosyadaki karakterler BUF_SIZE'ın katlarından küçük olabilir. Bu durumda son okuma tamponu tamamen doldurmayacaktır.

Karakterler tampondan tek tek okurken tamponun ya da dosyanın sonuna gelinip gelinmediği kontrol edilmelidir. İşte benzer kontrollerin ayrı ayrı yapılması yerine onlar birleştirebilir. Şöyle ki: Tamponun ve dosyanın sonu için tampona '\n' karakteri yerleştirilir. Bu '\n' karakteri okunduğunda ya yeni bir satıra geçilmiştir ya da dosyanın sonuna gelinmiştir. Böylece sona gelindi mi kontrolü yalnızca '\n' karakteri görüldüğünde yapılır. Zaten pek çok lexical analiz modülü atomlara ayırma işlemi sırasında hata mesajları için satır numaralarını saklar. Bu durumda '\n' karakteri görüldüğünde ya tamponun sonuna, ya dosyanın sonuna ya da satırın sonuna gelinmiştir.

Yukarıda programlama dillerindeki pek çok atomun birkaç karakterden oluştuğunu söyledik ve bu atomların maksimum uzunluğunu MAX_TOKEN değeri ile temsil ettik. Ancak yine yukarıda MAX_TOKEN değerinden daha uzun olabilecek üç tür atomun olduğunu da belirttik. Bunlar değişkenler (identifiers), sabitler, iki tırnak ifadeleri (strings) ve yorumlama (remark) alanlarıydı. Şimdi bu atomların elde edilmesi için nasıl bir tampon strateji izleneceği üzerinde duralım. Değişkenler ve iki tırnak ifadeleri pek çok dilde en fazla bir satırın uzunluğu kadar olabilmektedir. (Çünkü satırın sonunda '\n' karakteri vardır ve bu karakter zaten atomu sonlandırmaktadır.) İşte derleyicilerin çoğu işlemleri kolaylaştırmak için kaynak koddaki maksimum satır uzunluğu konusunda bir ön belirleme yapmaktadır. Biz de burada maksimum satır uzunluğunun MAX_LINE kadar olduğunu varsayalım. Böylece eğer boşluklar atıldıktan sonraki ilk karakter alfabetik bir karakterse, sayısal bir karakterse ya da iki tırnak karakteriyse gelmekte olan atom en kötü olasılıkla MAX_LINE kadar olabilir. Bu üç durumda tamponun doldurulma noktası MAX_TOKEN değil MAX_LINE olmalıdır. Satırın sonunda ve dosyanın sonunda '\n' karakterleri bulunacağından tamponun da maksimum uzunluğu MAX_LINE + 1 + BUFSIZE + 1 kadar olacaktır:

```
if (plimit - pcurrent < MAX_LINE) {
    FillBuffer();
    ...
}
```



Yorum alanlarının koddan atılması sırasında tamponun yine satır uzunluğu kadar doldurulması uygun olabilir. Tabii yorum alanları birden fazla satırdan oluşlabilmektedir. Buy durum bir istisna olarak ele alınabilir.

3.4.2. Lexical Analiz İşleminin Algoritmik Yapısı

Lexical analiz işleminde sıradaki atomun türü ve yazısal temsili tipik olarak şöyle bir algoritmik yapıyla gerçekleştirilmektedir:

1) Boşluk karakterleri pek çok dilde atom ayıracı olarak kullanıldığından dolayı önce boşluk klarakterleri geçilir. Boşluk karakterlerinin geçilmesi "boşluk karakteri olduğu sürece ilerleme yapan" bir while döngüsü ile gerçekleştirilebilir.

2) İlk boşluk karakteri olmayan karakter bulunır ve switch içerisine sokulur. Bu ilk karakter atomun türü hakkında bize bit ip ucu vermektedir. Bundan sonra duruma göre bu karakterin yanındaki karakterlere bakılarak atomun türü ve onu oluşturan karakterler net olarak belirlenir.

Örneğin C için bir lexical analiz modülü yazacak olalım. g_cp göstericisi tampondaki son kalınan yeri belirtiyor olsun. Biz oradaki karakterin '+' karakteri olduğunu düşünelim. Bu noktada biz henüz bunun + operatörü olduğu sonucunu çıkaramayız. Pekala bu operatör ++ ya da += operatörü de olabilir. Bunu belirlemek için bizim '+' karakterini gördüğümüzde onun yanındaki karakterlere de bakmamız gerekir.

```
switch (*g_cp++) {
    ...
    case '+':
        if (*g_cp == '+')
            return OPERATOR_PLUS_LUS;
        if (*g_cp == '=')
            return OPERATOR_PLUS_EQUAL;
        return OPERATOR_PLUS;
    ...
}
```

İşte lexical analiz işlemi "boşlukları atıp sıradaki karaktere bak, duruma göre onun yanındakilere de bakarak atomu oluşturan karakterleri ve atomun türünü tespit et" biçiminde bir algoritmik yapıya sahiptir. (Bu tür parse işlemine LL(k) tipi parse işlemi denilmektedir. Bu konu "parse" işlemlerinde ele alınacaktır.)

Kaynak kodu tararken karakterlerin türlerini belirlemek için bazı karşılaştırma işlemleri gerekir. Örneğin bir karakterterin boşluk karakteri olup olmadığı aşağıdaki gibi bir if deyimiyle belirlenebilir:

```
if (ch == ' ' || ch == '\t' || ch == '\n' || ch = '\v') {
    ...
}
```

Ancak lexical analiz modüllerinde işlemleri hızlandırmak için genellikle bir "lookup" tablosundan faydalanılabilmektedir. Şöyle ki: 256 elemanlı bir karakter dizisi oluşturulur. Bu karakter dizisinin karakter

kodlarına karşı gelen elemanlarına bitsel olarak özellikler atanır. Sonra da tek bir bitsel işlemle karakterin türü belirlenir. Örneğin böyle bir "lookup" tablosu aşağıdaki gibi oluşturulabilir:

```
enum CHAR_CLASS {
    BLANK = 0x01,
    NEWLINE = 0x02,
    ALPHA = 0x04,
    DIGIT = 0x08,
    OCTAL = 0x10,
    HEX = 0x20,
    OTHER = 0x40
};

static unsigned char g_cmap[256] = {
         /* 000 nul */     0,
         /* 001 soh */     0,
         /* 002 stx */     0,
         /* 003 etx */     0,
         /* 004 eot */     0,
         /* 005 enq */     0,
         /* 006 ack */     0,
         /* 007 bel */     0,
         /* 010 bs  */     0,
         /* 011 ht  */     BLANK,
         /* 012 nl  */     NEWLINE,
         /* 013 vt  */     BLANK,
         /* 014 ff  */     BLANK,
         /* 015 cr  */     0,
         /* 016 so  */     0,
         /* 017 si  */     0,
         /* 020 dle */     0,
         /* 021 dc1 */     0,
         /* 022 dc2 */     0,
         /* 023 dc3 */     0,
         /* 024 dc4 */     0,
         /* 025 nak */     0,
         /* 026 syn */     0,
         /* 027 etb */     0,
         /* 030 can */     0,
         /* 031 em  */     0,
         /* 032 sub */     0,
         /* 033 esc */     0,
         /* 034 fs  */     0,
         /* 035 gs  */     0,
         /* 036 rs  */     0,
         /* 037 us  */     0,
         /* 040 sp  */     BLANK,
         /* 041 !   */     OTHER,
         /* 042 "   */     OTHER,
         /* 043 #   */     OTHER,
         /* 044 $   */     0,
         /* 045 %   */     OTHER,
         /* 046 &   */     OTHER,
         /* 047 '   */     OTHER,
         /* 050 (   */     OTHER,
         /* 051 )   */     OTHER,
         /* 052 *   */     OTHER,
         /* 053 +   */     OTHER,
         /* 054 ,   */     OTHER,
         /* 055 -   */     OTHER,
         /* 056 .   */     OTHER,
         /* 057 /   */     OTHER,
         /* 060 0   */     DIGIT|OCTAL,
         /* 061 1   */     DIGIT|OCTAL,
         /* 062 2   */     DIGIT|OCTAL,
         /* 063 3   */     DIGIT|OCTAL,
         /* 064 4   */     DIGIT|OCTAL,
         /* 065 5   */     DIGIT|OCTAL,
         /* 066 6   */     DIGIT|OCTAL,
```

```
/* 067 7   */      DIGIT|OCTAL,
/* 070 8   */      DIGIT,
/* 071 9   */      DIGIT,
/* 072 :   */      OTHER,
/* 073 ;   */      OTHER,
/* 074 <   */      OTHER,
/* 075 =   */      OTHER,
/* 076 >   */      OTHER,
/* 077 ?   */      OTHER,
/* 100 @   */      0,
/* 101 A   */      ALPHA|HEX,
/* 102 B   */      ALPHA|HEX,
/* 103 C   */      ALPHA|HEX,
/* 104 D   */      ALPHA|HEX,
/* 105 E   */      ALPHA|HEX,
/* 106 F   */      ALPHA|HEX,
/* 107 G   */      ALPHA,
/* 110 H   */      ALPHA,
/* 111 I   */      ALPHA,
/* 112 J   */      ALPHA,
/* 113 K   */      ALPHA,
/* 114 L   */      ALPHA,
/* 115 M   */      ALPHA,
/* 116 N   */      ALPHA,
/* 117 O   */      ALPHA,
/* 120 P   */      ALPHA,
/* 121 Q   */      ALPHA,
/* 122 R   */      ALPHA,
/* 123 S   */      ALPHA,
/* 124 T   */      ALPHA,
/* 125 U   */      ALPHA,
/* 126 V   */      ALPHA,
/* 127 W   */      ALPHA,
/* 130 X   */      ALPHA,
/* 131 Y   */      ALPHA,
/* 132 Z   */      ALPHA,
/* 133 [   */      OTHER,
/* 134 \   */      OTHER,
/* 135 ]   */      OTHER,
/* 136 ^   */      OTHER,
/* 137 _   */      ALPHA,
/* 140 `   */      0,
/* 141 a   */      ALPHA|HEX,
/* 142 b   */      ALPHA|HEX,
/* 143 c   */      ALPHA|HEX,
/* 144 d   */      ALPHA|HEX,
/* 145 e   */      ALPHA|HEX,
/* 146 f   */      ALPHA|HEX,
/* 147 g   */      ALPHA,
/* 150 h   */      ALPHA,
/* 151 i   */      ALPHA,
/* 152 j   */      ALPHA,
/* 153 k   */      ALPHA,
/* 154 l   */      ALPHA,
/* 155 m   */      ALPHA,
/* 156 n   */      ALPHA,
/* 157 o   */      ALPHA,
/* 160 p   */      ALPHA,
/* 161 q   */      ALPHA,
/* 162 r   */      ALPHA,
/* 163 s   */      ALPHA,
/* 164 t   */      ALPHA,
/* 165 u   */      ALPHA,
/* 166 v   */      ALPHA,
/* 167 w   */      ALPHA,
/* 170 x   */      ALPHA,
/* 171 y   */      ALPHA,
/* 172 z   */      ALPHA,
/* 173 {   */      OTHER,
/* 174 |   */      OTHER,
```

```
        /* 175 }   */     OTHER,
        /* 176 ~   */     OTHER,
};
```

Böylece kaynak koddan ch karakteri çekildiğinde bu karakterin ASCII tablosundaki sıra numarası bu diziye indeks yapılacak ve g_cmap[ch] değeri de ilgili türlerle "bit and" işlemine sokulacaktır. Örneğin boşlukları g_cp göstericisi tamponda kaynak koddaki kalınan yeri gösteriyor olsun. Boşluk karakterlerini geçmek için aşağıdaki gibi bir döngü oluşturulabilir:

```
while (g_cmap[*g_cp] & BLANK)
    ++g_cp;
```

Görüldüğü gibi karakter haritasının oluşturulmasının amacı atomun türünü hızlı bir biçimde belirlemektir.

Şimdi aşağıdaki gibi basit bir dili atomlarına ayırmak isteyelim:

```
Small-Lang:
    Expression
    Expression Small-Lang
Expression:
       identifier '=' Additive ';'
Additive:
    Additive '+' Multiplicative
    Additive '-' Multiplicative
    Multiplicative

Multiplicative:
    Multiplicative '*' Multiplicative
    Multiplicative '/' Multiplicative
    identifier
    '(' Expression ')'

Identifier:
    Alfa
    AlfaNumeric  Identifer

Alfa:
    'a' | 'b' .....
AlfaNumeric:
    Alfa | 0 | 1 ....
```

Bu dilin birkaç elemanı şöyle olabilir:

```
a = b * c * d;
a = (b + c) * d;
a = b + c – d / e;
```

Bütün programın global bir g_prog isimli bir dizide bulunduğunu ve tamponlama yapılmadığını düşünelim. g_cp göstericisi de işin başında bu dizinin başlangıcını gösteriyor olsun. Gramerde gösterilmemiş olsa da atomlar arasında istenildiği kadar boşluk karakterlerinin bulunabildiğini varsayalım. Bu koşullar altında atomlarına ayırma işlemi aşağıdaki gibi bir kodla yapılabilir:

```
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#include <string.h>

/* Token Types */

#define TOKEN_IDENTIFIER            1
#define TOKEN_OPERATOR_PLUS         2
```

```
#define TOKEN_OPERATOR_MINUS        3
#define TOKEN_OPERATOR_MULTIPLY     4
#define TOKEN_OPERATOR_DIVIDE       5
#define TOKEN_LEFT_PARANTHESIS      6
#define TOKEN_RIGHT_PARANTHESIS     7
#define TOKEN_ASSIGNMENT            8
#define TOKEN_SEMICOLON             9


#define MAX_TOKEN                   512

#define BLANK       0x01
#define NEWLINE     0x02
#define ALPHA       0x04

static unsigned char g_cmap[256] = {
    /* 000 nul */   0,
    /* 001 soh */   0,
    /* 002 stx */   0,
    /* 003 etx */   0,
    /* 004 eot */   0,
    /* 005 enq */   0,
    /* 006 ack */   0,
    /* 007 bel */   0,
    /* 010 bs  */   0,
    /* 011 ht  */   BLANK,
    /* 012 nl  */   BLANK | NEWLINE,
    /* 013 vt  */   BLANK,
    /* 014 ff  */   BLANK,
    /* 015 cr  */   0,
    /* 016 so  */   0,
    /* 017 si  */   0,
    /* 020 dle */   0,
    /* 021 dc1 */   0,
    /* 022 dc2 */   0,
    /* 023 dc3 */   0,
    /* 024 dc4 */   0,
    /* 025 nak */   0,
    /* 026 syn */   0,
    /* 027 etb */   0,
    /* 030 can */   0,
    /* 031 em  */   0,
    /* 032 sub */   0,
    /* 033 esc */   0,
    /* 034 fs  */   0,
    /* 035 gs  */   0,
    /* 036 rs  */   0,
    /* 037 us  */   0,
    /* 040 sp  */   BLANK,
    /* 041 !   */   0,
    /* 042 "   */   0,
    /* 043 #   */   0,
    /* 044 $   */   0,
    /* 045 %   */   0,
    /* 046 &   */   0,
    /* 047 '   */   0,
    /* 050 (   */   0,
    /* 051 )   */   0,
    /* 052 *   */   0,
    /* 053 +   */   0,
    /* 054 ,   */   0,
    /* 055 -   */   0,
    /* 056 .   */   0,
    /* 057 /   */   0,
    /* 060 0   */   0,
    /* 061 1   */   0,
    /* 062 2   */   0,
    /* 063 3   */   0,
    /* 064 4   */   0,
    /* 065 5   */   0,
```

```
    /* 066 6   */   0,
    /* 067 7   */   0,
    /* 070 8   */   0,
    /* 071 9   */   0,
    /* 072 :   */   0,
    /* 073 ;   */   0,
    /* 074 <   */   0,
    /* 075 =   */   0,
    /* 076 >   */   0,
    /* 077 ?   */   0,
    /* 100 @   */   0,
    /* 101 A   */   ALPHA,
    /* 102 B   */   ALPHA,
    /* 103 C   */   ALPHA,
    /* 104 D   */   ALPHA,
    /* 105 E   */   ALPHA,
    /* 106 F   */   ALPHA,
    /* 107 G   */   ALPHA,
    /* 110 H   */   ALPHA,
    /* 111 I   */   ALPHA,
    /* 112 J   */   ALPHA,
    /* 113 K   */   ALPHA,
    /* 114 L   */   ALPHA,
    /* 115 M   */   ALPHA,
    /* 116 N   */   ALPHA,
    /* 117 O   */   ALPHA,
    /* 120 P   */   ALPHA,
    /* 121 Q   */   ALPHA,
    /* 122 R   */   ALPHA,
    /* 123 S   */   ALPHA,
    /* 124 T   */   ALPHA,
    /* 125 U   */   ALPHA,
    /* 126 V   */   ALPHA,
    /* 127 W   */   ALPHA,
    /* 130 X   */   ALPHA,
    /* 131 Y   */   ALPHA,
    /* 132 Z   */   ALPHA,
    /* 133 [   */   0,
    /* 134 \   */   0,
    /* 135 ]   */   0,
    /* 136 ^   */   0,
    /* 137 _   */   ALPHA,
    /* 140 `   */   0,
    /* 141 a   */   ALPHA,
    /* 142 b   */   ALPHA,
    /* 143 c   */   ALPHA,
    /* 144 d   */   ALPHA,
    /* 145 e   */   ALPHA,
    /* 146 f   */   ALPHA,
    /* 147 g   */   ALPHA,
    /* 150 h   */   ALPHA,
    /* 151 i   */   ALPHA,
    /* 152 j   */   ALPHA,
    /* 153 k   */   ALPHA,
    /* 154 l   */   ALPHA,
    /* 155 m   */   ALPHA,
    /* 156 n   */   ALPHA,
    /* 157 o   */   ALPHA,
    /* 160 p   */   ALPHA,
    /* 161 q   */   ALPHA,
    /* 162 r   */   ALPHA,
    /* 163 s   */   ALPHA,
    /* 164 t   */   ALPHA,
    /* 165 u   */   ALPHA,
    /* 166 v   */   ALPHA,
    /* 167 w   */   ALPHA,
    /* 170 x   */   ALPHA,
    /* 171 y   */   ALPHA,
    /* 172 z   */   ALPHA,
};
```

```c
/* Function Prototypes */

int GetNextToken(void);

/* Global Variables */

char g_prog[4096 + 1];      /* array in which source code resides */
char *g_cp;                 /* current pointer for the lexer */
char *g_limit;              /* points to the end of the code */
int g_lineNo;

char g_token[MAX_TOKEN];
char *g_tokenTypes[] = { "", "Identifier", "Operatör Plus", "Operator Minus", "Operetor Multiply", "Operator
Divide", "Operator Left Paranthesis", "Operator Right Paranthesis", "Operator Assignment", "Delimiter
Semicolon"};

/* Function Definitions */

int main(int argc, char *argv[])
{
    FILE *f;
    int tokenId;
    size_t n;

    if (argc != 2) {
        fprintf(stderr, "wrong number of arguments!..\n");
        exit(EXIT_FAILURE);
    }

    if ((f = fopen(argv[1], "r")) == NULL) {
        fprintf(stderr, "cannot open file!...\n");
        exit(EXIT_FAILURE);
    }

    n = fread(g_prog, 1, 4096, f);

    if (ferror(f)) {
        fprintf(stderr, "cannot read file!..\n");
        exit(EXIT_FAILURE);
    }
    fclose(f);

    g_prog[n] = '\n';
    g_limit = &g_prog[n];
    g_cp = g_prog;

    while ((tokenId = GetNextToken()) != -1)
        printf("Token Type: %s, Token: %s\n", g_tokenTypes[tokenId], g_token);

    return 0;
}

int GetNextToken(void)
{
    char *cp;
    int count;
    int tokType;

    while ((g_cmap[*g_cp] & BLANK)) {
        if (g_cmap[*g_cp] & NEWLINE) {
            if (g_cp == g_limit)
                return -1;
            ++g_lineNo;
        }
        ++g_cp;
    }

    if (g_cmap[*g_cp] & ALPHA) {
        cp = g_cp + 1;
```

```
        while (g_cmap[*cp] & ALPHA)
            ++cp;
        count = cp - g_cp;
        memcpy(g_token, g_cp, count);
        g_token[count] = '\0';
        g_cp = cp;

        return TOKEN_IDENTIFIER;
    }

    switch (*g_cp) {
        case '+':
            tokType = TOKEN_OPERATOR_PLUS;
            break;
        case '-':
            tokType = TOKEN_OPERATOR_MINUS;
            break;
        case '*':
            tokType = TOKEN_OPERATOR_MULTIPLY;
            break;
        case '/':
            tokType = TOKEN_OPERATOR_DIVIDE;
            break;
        case '(':
            tokType = TOKEN_LEFT_PARANTHESIS;
            break;
        case ')':
            tokType = TOKEN_RIGHT_PARANTHESIS;
            break;
        case ';':
            tokType = TOKEN_SEMICOLON;
            break;
        case '=':
            tokType = TOKEN_ASSIGNMENT;
            break;
    }

    g_token[0] = *g_cp++;
    g_token[1] = '\0';

    return tokType;
}
```

Buradaki kodda birkaç noktanın altını çizmek istiyoruz:

- Her ne kadar kullanılmamış olsa da yukarıdaki kodda satır sayılarının tutulduğunu görüyorsunuz. Pek çok durumda (hata mesajlarında vs.) o andaki satır numarasının tutulması gerekebilmektedir.

- Tek karakterli atomların türleri için doğrudan onların ASCII karakter karşılıkları kullanılabilir. (Örneğin ‘;’ atomunun türü ‘;’ karakterinin ASCII karşılığı olarak alınabilir.) Böylece yukarıdaki kodun son bölümü çok daha basit hale gelecektir.

- Kodda sürekli “dosyanın sonuna gelindi mi?” kontrolünün yapılmadığına bunun yalnızca ‘\n’ karakteri görüldüğünde yapıldığına dikkat ediniz.

Bir C programının atomlarına ayrılması yukarıdaki örnekten daha karmaşıktır. Ancak temel yapı aynıdır. Kursumuzda Src dizinini altında “008-C-Lexer” dizininde bu işlemi yapan kodlar verilmiştir. Bu kodlarda yukarıda ele alınan tamponlama da uygulanmış durumdadır.

3.4.3. Düzenli İfadeler (Regular Expressions)

Düzenli ifadeleri "yazısal kalıpların ifade edilmesinde kullanılan küçük bir dil" olarak tanımlayabiliriz. Gerçekten de özellikle yazılarda belli kalıpların aranması sürecinde düzenli ifadelerden sıkça faydalanılmaktadır. Örneğin bir text editörde “dd/mm/yyyy” kalıbına uyan tarihleri ya da xxxxx@yyyyy.com

kalıbına uyan e-posta adreslerini bulmak isteyebiliriz. Bu kalıpların editörlerdeki klasik metin arama özellikleriyle bulunamayacağına dikkat ediniz. İşte düzenli ifadeler böyle kalıpların ifade edilmesini sağlayan kurallar topluluğundan oluşmaktadır. Gelişmiş pek çok kelime işlemci düzenli ifadeler yoluyla arama işlemi yapabilmektedir. Lexical analiz araçlarında da atomların belirlenmesi sürecinde düzenli ifadelerden faydalanılmaktadır.

Düzenli ifadeler üzerinde işlem yapan araçların kullandıkları kodlara "düzenli ifade motorları (regular exprtession engines)" denilmektedir. Malesef düzenli ifadelerin kurallarına ilişkin bir standart yoktur. Bu nedenle düzenli ifade motorlarının da tamamen aynı kurallara sahip olduklarını söyleyemeyiz. Ancak pek çok motor büyük ölçüde birbirlerine benzemektedir.

Düzenli ifadeler yukarıda da belirtildiği gibi kalıpların ifade edilmesinde kullanılmaktadır. Düzenli ifadeleri kullanan tipik araçlardan bazıları şunlardır:

- Gelişmiş kelime işlemciler. (Microsoft Word, Libre Office vs.)

- Bazı komut satırı araçları. Örneğin bunların en ünlüsü "grep"tir.

- awk, sed gibi text işlemlerinde kullanılan küçük diller.

- Programlama dillerindeki kütüphane fonksiyonları ve sınıflar. Örneğin POSIX'in regex fonksiyonları ya da boost'tan alınarak C++11'e dahil edilmiş olan regex sınıfları.

- Lexical analiz işlemlerini yapan araçlar.

3.4.3.1. Düzenli İfadelerin Oluşturulması

Düzenli ifadelerde iki tür karakter kümesi vardır: Normal karakterler ve meta karakterler. Normal karakterler kalıpta karakter olarak bulunması gereken öğelerdir. Yani kalıptaki normal bir karakter başak bir şeyi değil kendisini temsil eder. Meta karakterler ise kalıpta kendisini temsil etmeyen, özel anlama gelen karakterlerdir. Örneğin '+' bir meta karakterdir. '+' karakterinin düzenli ifadelerde özel başka bir anlamı vardır. Bu karakter onun solundaki karakterden "bir tane ya da daha fazla bulunma" durumunu belirtir. Örneğin "ab+c" kalıbı aşağıdaki yazılarla uyuşabilir:

```
abc
abbbbc
abbbbbbbbc
abbbbbbbbbbbbc
```

İşte '+' gibi değişik anlamlara gelen pek çok meta karakter bulunmaktadır. Zaten düzenli ifade dilinin öğrenilmesi büyük ölçüde bu meta karakterlerin öğrenilmesi sürecidir.

Düzenli ifade motorlarının kullandığı tipik meta karakterler ve anlamları şunlardır:

Meta Karakterler	Anlamı
. (nokta)	'\n' dışındaki herhangi bir karakter
?	Solundaki karakterden 0 tane ya da 1 tane
*	Solundaki karakterden 0 tane ya da çok tane
+	Solundaki karakterden 1 tane ya da çok tane
{n}	Burada n bir sayıdır. Solundaki karakterden tam olarak n tane anlamına helir.
{n,}	Burada n bir sayıdır. Solundaki karakterden tam olarak en az n tane anlamına helir.
{n,m}	Burada n ve m birer sayıdır. Solundaki karakterden en az n tane en fazla m tane anlamına gelir.
[ ]	Köşeli parantez içerisindeki karakterlerden herhangi birisi anlamına gelir.

[x-y]	x ve y aralığındaki herhangi bir karakter
[^]	Köşeli parantez içerisindeki karakterlerden olmayan herhangi bir karakter
\w	Herhangi bir alfanümerik karakter
\W	Herhangi bir alfanümerik olmayan karakter
\s	Herhangi bir boşluk karakteri
\S	Herhangi bir boşluk olmayan karakter
$	Satırın sonunun belli karakterlerle sonlanması durumu (Örneğin “kaan$”)
^	Satırın başı belli karakterlerle sonlanması durumu (Örneğin ^kaan”)
(..)	Gruplama amacıyla kullanılır. Böylece bunun sağındaki metakarakterler bu grup için anlam kazanır.
|	Veya anlamına gelmektedir. Örneğin "ali|veli" yazı içerisindeki "ali" veya "veli" ile uyşur.

Düzenli ifadelerde kullanılan tüm meta karakterlerin bunlarla sınırlı olmadığını belirtelim. Diğer meta karakterler için kurs dokümanları içerisindeki kitaplardan faydalanabilirsiniz.

Parantezlerin gruplama amacıyla kullanıldığında dikkat ediniz. Örneğin ([a-z]_){3} kalıbında {3} ‘a’ ile ‘z’ arasındaki karakterlerden biri ile ‘_’ karakterinin birleşimlerinde üç tane olacağı anlamına gelmektedir (örneğin “x_y_z_” gibi).

Kalıp içerisindeki meta karakterlerin normal karakterlerle karışmaması için düzenli ifade motorları iki yöntem kullanabilmektedir:

1) Meta karakterlerle çakışan normal karakterlerin önüne ters bölü karakteri getirme yöntemi. Örneğin: "\.+" kalıbı bir ya da birden fazla '.' karakteri ile uyuşur. Ters bölüden dolayı artık kalıptaki '.' karakteri bir meta karakter olarak değil '.' karakterinin kendisi olarak ele alınır.

2) Meta karakterlerlerin önüne ters bölü karakteri getirme yöntemi. Örneğin bu yöntemde ".\+" kalıbı bir ya da birden fazla '.' karakteriye uyuşacaktır. Bu yöntemde çakışan karakterlerin default olarak normal karakter kabul edildiğine dikkat ediniz.

Düzenli arama motorları genellikle birinci yöntemi kullanmaktadır. Ancak bazıları kullanıcının her iki yöntemden birini seçmesine de olanak sağlar.

Şimdi bu meta karakterlerin anlamlarına ilişkin bazı örnekler verelim:

Kalıp	Neyle Uyuşur?
`[_a-zA-z]+`	'_' karakterinden ve alfabetik karakterlerden oluşan karakter dizileriyle uyuşur. Köşeli parantez içerisindeki [a-z] gibi bir kalıbın 'a' ile 'z' arasındaki herhangi bir karakter anlamına geldiğini anımsayınız.
`[+-]?[0-9]+\.?[0-9]*`	Gerçek sayı kalıplarıyla uyuşur. Örneğin “123”, “123.45”, “-1” gibi. (Ancak “.12” ya da “.12” gibi kalıplarla uyuşmaz)
`([0-9]{1,3}\.){3}[0-9]{1,3}`	Bölümleri “.” ile ayrılmış IP numaralarıyla uyuşur. Örneğin “192.160.0.100” gibi. Burada parantezler gruplama amacıyla kullanılmıştır. Dolayısıyla kalıbın [0-9]{1,3} kısmı 0’dan 9’a kadar karakterlerden 1 ya da 2 ya da 3 tane olacağını belirtir. kalıbın ([0-9]{1,3}\.){3} kısmı ise üç basamağa kadar sayı ve noktalaran toplamda üç tane bulunacağını belişrtmektedir.
`^\w*`	Satırların başındaki sözcüklerle uyuşur

3.4.3.2. UNIX/Linux Sistemlerindeki grep Programı

grep (globally search a regular **e**xpression) UNIX/Linux sistemlerinde çok sık kullanılan POSIX standartlarında da tanımlı olan bir komut satırı aracıdır. grep programının genel kullanım biçimi şöyledir:

```
grep  <kalıp>     [dosya yol ifadeleri]
```

Kalıp eğer boşluk içermiyorsa tırnak içerisine alınmayabilir. Ancak boşluk içeriyorsa tırnaklanmalıdır. Tırnaklama tek tırnak ya da çift tırnak ile yapılabilir. Komutta bir ya da birden fazla "dosya yol ifadesi" belirtilebilir. Bu durumda grep kalıbı sırasıyla bu dosyalarda arar. Eğer komutta hiç "dosya yol ifadesi" belirtilmezse grep arama yapacağı yazıyı stdin dosyasından okur. Örneğin:

```
grep -E “([0-9]{1,3}\.){3}[0-9]{1,3}” test.txt
```

Burada grep kalıbı "test.txt" dosyasında arayacaktır.

grep default durumda metin içerisinde kalıbı bulduğunda onun bulunduğu satırın tamamını ekrana (stdout dosyasına) yazdırmaktadır. Kalıp içerisinde meta karakterler ile çakışan normal karakterlerin hangilerinin ters bölü ile yazılacağı komut satırından –E seçeneği ile belirlenir. grep default durumda kalıptaki her karakteri normal karakter olarak değerlendirir. Bunların meta karakter olarak değerlendirilmesi için ters bölülenmesi gerekir. Halbuki –E seçeneği bunun tam tersine yol açmaktadır. –E seçeneği girilirse kalıptaki meta karakterlerle çakışan karakterler meta karakterler kabul edilir. Bu durumda bunları meta karakter olmaktan çıkartmak için ters bölülemek gerekecektir. grep'in "–E" kullanımı çok yaygın olduğu için UNIX/Linux sistemlerinde bu işlemi yapan "egrep" komutu da bulundurulmaktadır. Yani "egrep" ile "grep -E" aynı işlemeyol açmaktadır.

grep borulama işlemleriyle de çok sık kullanılmaktadır. Örneğin:

```
ps -e | grep "tty"
```

Burada borulama sayesinde ps komutunun stdout dosyasına yazdıklarını grep stdin’den okuyormuş gibi bir etki oluşacaktır. Sonuç olarak bu komutla birlikte proses listesindeki içinde "tty" geçen satırlar elde edilecektir. Örneğin:

```
ls -l /usr/include | grep "Nov"
```

Burada /usr/iclude dizininde “Nov” geçen satırlar listelenmektedir.

Yukarıda da belirtildiği gibi grep default olarak kalıbın bulunduğu tüm satırı ekrana (stdout dosyasına) yazdırmaktadır. Fakat eğer grep “–o” seçeneğiyle kullanılırsa kalıbın bulunduğu tüm satırı değil yalnızca bulunan kalıbın kendisini ekrana yazdırır. “–c” seçeneği kalıbın toplamda kaç satırda bulunduğu bilgisini, “-b” seçeneği kalıbın dosyanın kaçıncı offset’lerinde bulunduğu bilgisini, “-n” seçeneği de kalıbın bulunduğu satır numaralarını vermektedir.

3.4.3.3. C’de Düzenli İfadelerle İşlemler

C’nin standart kütüphanesinde düzenli ifadeler üzerinde işlem yapan fonksiyonlar yoktur. Ancak POSIX standartlarında düzenli ifadeler üzerinde işlem yapan C fonksiyonları bulunmaktadır. POSIX fonksiyonlarının UNIX türevi işletim sistemlerinde bulunması öngörülmüş olan fonksiyonlar olduğunu anımsayınız. Bu nedenle düzenli ifadelere ilişkin POSIX fonksiyonlarını da yalnızca UNIX/Linux türevi sistemlerde ve Mac OS X sistemlerinde kullanabilirsiniz.

Düzenli ifadeler üzerinde işlem yapan POSIX fonksiyonları şunlardır:

```
#include <sys/types.h>
#include <regex.h>
```

```
int regcomp(regex_t *preg, const char *regex, int cflags);
int regexec(const regex_t *preg, const char *string,
           size_t nmatch, regmatch_t pmatch[], int eflags);
size_t regerror(int errcode, const regex_t *preg, char *errbuf,  size_t errbuf_size);
void regfree(regex_t *preg);
```

regcomp fonksiyonu düzenli ifadeyi oluşturan ana fonksiyondur. Bu fonksiyondan regext_t türüyle temsil edilen bir handle değeri elde edilmektedir. regcomp fonksiyonun birinci parametresi regex_t türünden içi doldurulacak nesnenin adresini alır. Fonksiyonun ikinci parametresi düzenli ifade kalıbını almaktadır. Son parametre ise bazı seçeneklerden oluşmaktadır. Fonksiyon başarı durumunda sıfır, başarısızlık durumunda hata kodunun kendisine geri dönmektedir.

Kalıbı bulan asıl fonksiyon regexec fonksiyonudur. Bu fonksiyonun birinci parametresi regcomp fonksiyonundan elde edilen handle değerini, ikinci parametresi de arama yazısını almaktadır. Üçücüncü parametre eşleşen kaç kalıbın ve alt kalıpların sayısını belirtir. Bulunan kalıplar regmatch_t türünden bir dizinin içerisine yerleştirilmektedir. Son parametre yine bazı seçenekleri belirtmektedir.

Düzenli ifadelerle işlemler bittiğinde regcomp fonksiyonunda yapılan bazı tahsisatları geri almak için regfree fonksiyonu çağrılmalıdır. Örneğin:

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <regex.h>

#define BUF_SIZE  10000

int main(int argc, char *argv[])
{
    regex_t rex;
    FILE *f;
    size_t n;
    char buf[BUF_SIZE + 1];
    regmatch_t matches[1];
    int result, i, k, beg = 0;

    if (argc < 2 || argc > 3) {
        fprintf(stderr, "wrong number of arguments!\nusage mygrep <pattern> [path]\n");
        exit(EXIT_FAILURE);
    }

    if (argc == 2)
        f = stdin;
    else {
        if ((f = fopen(argv[2], "r")) == NULL) {
            fprintf(stderr, "cannot open file!..\n");
            exit(EXIT_FAILURE);
        }
    }
    n = fread(buf, 1, BUF_SIZE, f);
    if (n == 0 && ferror(f)) {
        fprintf(stderr, "cannot read file!..\n");
        exit(EXIT_FAILURE);
    }
    buf[n] = '\0';

    if ((result = regcomp(&rex, argv[1], REG_EXTENDED)) != 0) {
        fprintf(stderr, "%s\n", strerror(result));
        exit(EXIT_FAILURE);
```

```
    }

    while (regexec(&rex, buf + beg, 1, matches, REG_NOTEOL) != REG_NOMATCH) {
        printf("%.*s\n", matches[0].rm_eo - matches[0].rm_so, &buf[beg + matches[0].rm_so]);
        beg += matches[0].rm_eo + 1;
    }

    regfree(&rex);

    return 0;
}
```

Biz burada yalnızca temel bazı açıklamalar eşliğinde küçük bir örnek verdik. Ancak POSIX'in düzenli ifadeler üzerinde işlem yapan fonksiyonlarının bazı ayrıntıları da vardır. Bu ayrıntılar hakkında bilgileri POSIX dokümanlarından ya da "man" sayfalarından elde edebilirsiniz.

3.4.3.4. C++'ta Düzenli Düzenli İfadelerle İşlemler

C++'ta düzenli ifadeler ile işlemler için ağırlıklı olarak "boost" kütüphanesindeki regex sınıfları kullanılıyordu. Bu sınıflar daha sonraları üzerinde küçük değişiklikler yapılarak C++11'de standartlara dahil edilidi. Bu kurs C++ bilgisi gerektirmediği için biz burada yalnızca C++11'in regex sınıflarına yönelik küçük bir örnekle yetineceğiz.

```
#include <iostream>
#include <regex>

using namespace std;

int main()
{

    regex re("(\\+|-)?[0-9]+");
    string text;

    cout << "Bir yazi giriniz:";
    cin >> text;

    if (regex_match(text, re))
        cout << "yazida bir sayi var\n";
    else
        cout << "yazida sayi yok!\n";

    return 0;
}
```

Burada bir regex nesnesi oluşturulmuştur. Bu nesne oluşturulurken düzenli ifade kalıbı sınıfın başlangıç fonksiyonunda (constructor) belirtilmiştir. Sonra regex_match fonksiyonu ile verilen yazının bu kalıba uyup uymadığına bakılmıştır. Yazı içerisindeki kalıbın aranması işlemi ise regex_search fonksiyonuyla yapılmaktadır. Örneğin:

```
#include <iostream>
#include <regex>

using namespace std;

int main()
{
    regex re("(\\+|-)?[0-9]+");
    string text;
    smatch match;

    cout << "Bir yazi giriniz:";
```

```
    cin >> text;
    if (regex_search(text, match, re)) {
        cout << "prefix: " << match.prefix() << ", suffix: " <<
                match.suffix() << endl;
        cout << "bulunan kalip:";
        for (int i = 0; i < match.size(); ++i)
            cout << match[i];
        cout << endl;
    }
    else
        cout << "not ok\n";

    return 0;
}
```

C++11'in regex kütüphanesinin ayrıntıları için ilgili dokümanları inceleyiniz.

3.5. Lexical Analiz İşlemini Yapan Araçlar

Lexical analiz işlemleri manuel olarak yapılabileceği gibi bazı araçlar kullanılarak da yapılabilmektedir. Lexical analiz işlemini gerçekleştiren pek çok araç bulunmaktadır. Ancak bunlardan en yaygın kullanılanının "lex" ve onun modern versiyonu olan "flex" olduğu söylenebilir. ANTLR diğer bir seçenek olarak düşünülebilir. ANTLR gittikçe yaygınlaşmaktadır.

Lexical analiz araçları lexical analiz işlemini yapan program kodlarını üretmektedir. Programcı da üretilen bu kodları kendi projelerine ekleyerek kullanabilmektedir. lex 1975 yılında tasarlanmış olan standart haline gelmiş bir lexical analiz aracıdır. Flex ise 1987 yılında lex'in geliştirilmiş bir biçimi olarak tasarlanmıştır. flex ile lex pek çok bakımından uyumludur. Flex organik bağlantı olsa da resmi olarak GNU projesi kapsamında değildir.

3.5.1. Flex (Lex) Aracının Kullanımı

Flex orijinal olarak UNIX/Linux sistemleri için düşünülmüşür. Ancak bu aracın Windows versiyonu da oluşturulmuş durumdadır. Flex Linux sistemlerinde "binutil" paketi içersinde işletim sisteminin temel bir aracı olarak bulundurulmaktadır. Yani Linux sistemlerinde flex'in kurulumu zaten yapılmış gibidir. Ancak Flex'in Windows sürümleri numara olarak daha geriden gelmektedir. Flex'in orijinal dokümanları kendi sitesinden indirilebilir. Bunları kurs dokümanları içerisinde "Doc/EBooks/Flex-Bison" dizininde bulabilirsiniz.

3.5.1.1. Flex'in Kurulumu

Yukarıda da belirtildiği gibi Flex Linux sistemlerinin temel bir aracı olarak kabul edilmektedir. Eğer sistemde GCC derleyicisi yüklü ise (temel kurulumlarda bile GCC yüklenir) Flex de zaten yüklenmiş durumda olacaktır. Tabii biz bu sistemlerde kurulum programlarıyla bunların yeni versiyonlarını indirip kurabiliriz. Örneğin:

```
sudo apt-get install flex
```

Yukarıda da belirttiğimiz gibi Flex'in Windows versiyonu da oluşturulmuştur. Projenin "sourceforge.net" sayfasının adresi şöyledir: http://gnuwin32.sourceforge.net/packages/flex.htm. Bu sayfadan kurulum dosyası indirilerek kurulum yapılabilir. Tabii Flex komut satırından kullanılan bir araç olduğu için kurulumdan sonra "PATH" çevre değişkeninin ayarlanması gerekmektedir. Ancak Flex'in bu Windows versiyonunda uzun komut satırı seçenekleri konusunda (örneğin –header-file gibi) sorunlar vardır. Buna alternatif olarak https://sourceforge.net/projects/winflexbison/ projesindeki binary dosyalar da kullanılabilir. Bu projedeki Flex programının ismi "win_flex.exe" biçimindedir.

Mac OS X sistemlerinde çalışıyorsanız Flex'i "brew" utility'si yardımıyla kurabilirsiniz. Tabii önce "brew" utility'sini kurmair. Bu işlem şöyle yapılabilir:

```
ruby -e "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/master/install)" < /dev/null
2> /dev/null
```

Daha sonra "brew" ilw kurulum şöye yapılabilir:

```
brew install flex
```

3.5.1.2. Flex'in Genel Sentaksı

Bir Flex kaynak dosyası üç bölümden oluşmaktadır: "Tanımlamalar (Definitions)", Kurallar (Rules)" ve "Kullanıcı Kodları (Uer Codes)". Kaynak kodun tepesinden ilk %% karakterlerine kadarki bölüm "tanımlamaları", ilk %% karakterlerinden ikinci %% karakterlerine kadar olan bölüm "kuralları" ve ikinci %% karakterlerinden dosya sonuna kadarki bölüm de "kullanıcı kodlarını" oluşturmaktadır:

```
Tanımlamalar (definitions)
%%
Kurallar (Rules)
%%
Kullanıcı Kodları  (User Codes)
```

Flex kodları herhangi bir text editör kullanılarak oluşturulabilir. Geleneksel olarak Flex dosyalarının uzantısı ".l" biçiminde verilmektedir. Tabii aslında uzantı herhangi bir biçimde olabilir.

Flex dosyası oluşturulduktan sonra dosya işlenmek üzere komut satırından flex programına sokulur. Bu işlemin en yalın hali şöyledir:

```
flex <flex kaynak dosyası>
```

Örneğin:

```
flex sample.l
```

Flex bu işlem sonucunda bize lexical analiz işlemini yapan bir C dosyası verecektir. Flex'in ürettiği bu C dosyası default olarak "lex.yy.c" ismindedir. Artık biz bu dosyayı bir C derleyicisi ile derleyerek lexical analiz işlemini yapan programımızı çalışabilir hale getirebiliriz. İleride de ele alacağımız gibi flex bize default biçimde yazılmış iki fonksiyonu bir kütüphane içerisinde vermektedir. Bunlardan birincisi programın başlangıç noktası olan main fonksiyonu (yanlış görmüyorsunuz main fonksiyonu da istenirse kütüphaneye yerleştirilebilir) diğeri de yywrap isimli fonksiyondur. Bu iki fonksiyon "libfl.a" isimli kütüphanedir. Eğer main ve yywrap fonksiyonalrını biz yazmayacaksak link işlemine bu kütüphanenin de dahil edilmesi gerekir. Bu dahil etme işlemini GCC derleyicilerinde "-lfl" seçeneği ile yapabilirsiniz. Örneğin:

```
gcc –o sample lex.yy.c –lfl
```

Windows'ta GCC derleyicisiyle derleme yapılırken eğer "-lfl" seçeneği kullanılacaksa libfl.a'nın yeri önemlidir. Bu yerin –L seçeneğiyle belirtilmesi gerekir. Örneğin:

```
gcc -o sample lex.yy.c -LC:\Program Files (x86)\GnuWin32\lib -lfl
```

Ya da Windows'ta hiç "-lfl" seçeneği kullanılmadan doğrudan ".a" uzantılı kütüphane dosyasının yeri de belirtilebilir. Örneğin:

```
gcc –o sample lex.yy.c "C:\Program Files (x86)\GnuWin32\lib\libfl.a"
```

Windows'ta Microsoft'un "cl.exe" derleyicisi ile derleme yapılırken bu "libfl.a" kütüphanesi benzer biçimde link işlemine dahil edilmelidir.

```
cl /output: sampl.exe lex.yy.c "C:\Program Files (x86)\GnuWin32\lib\libfl.a"
```

Burada bir noktaya dikkatinizi çekmek istiyoruz. Eğer main fonksiyonunu programcının kendisi yazarsa (zaten biz hep böyle yapacağız) ve yywrap fonksiyonunu da yazmayacağını beyan ederse "libfl.a" dosyasının link işlemine dahil edilmesinin gerekliliği ortadan kalkmaktadır. yywrap fonksiyonunun bulunmayacağı bilgisi flex kaynak dosyasında "%option noyywrap" direktifi ile belirtilmektedir. Bu durumda iskelet bir flex programı şöyle olabilir:

```
%option noyywrap

%%
%%

int main(void)
{
   yylex();

   return 0;
}
```

Artık derleme işleminde "libfl.a" dosyasının belirtilmesine gerek yoktur. Hem Linux, hem Mac OS X hem de Windows sistemlerinde flex ve derleme işlemleri şöyle yapılabilir:

```
flex sample.l
gcc –o sample lex.yy.c
```

Windows'ta Microsoft'un "cl.exe" derleyicisi ile derleme işlemini şöyle yapabilirsiniz:

```
cl /output:sample.exe lex.yy.c
```

Yukarıda da belirttiğimiz gibi default olarak flex programının ürettiği C program dosyası "lex.yy.c" ismindedir. Ancak biz "–o" komut satırı argümanıyla üretilecek dosyanın ismini belirleyebiliriz. Örneğin:

```
flex –osample.c sample.l
gcc –o sample sample.c
```

flex klasik GNU argüman seçenekleri kurallarına uymamaktadır. Bu nedenle –o seçeneği ile dosya ismi arasında boşluk karakterleri bırakmaynız.

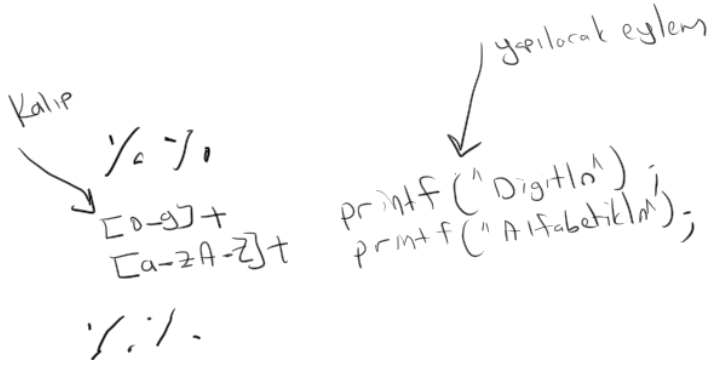
3.5.1.3. Flex Kaynak Dosyasındaki Bölümlerin Anlamı

Flex aracının ürettiği koddaki lexical analiz işlemini yapan temel fonksiyon yylex isimli fonksiyondur. Bu fonksiyon bizim önceki konularda ele aldığımız GetNextToken fonksiyonuna benzetilebilir. yylex tek çağırmada her şeyi yapacak biçimde ya da her çağırmada sıradaki atomu verecek biçimde de kullanılabilir. İskelet Flex programında main fonksiyonu içerisinde yylex fonksiyonun yalnızca bir kez çağrıldığına dikkat ediniz.

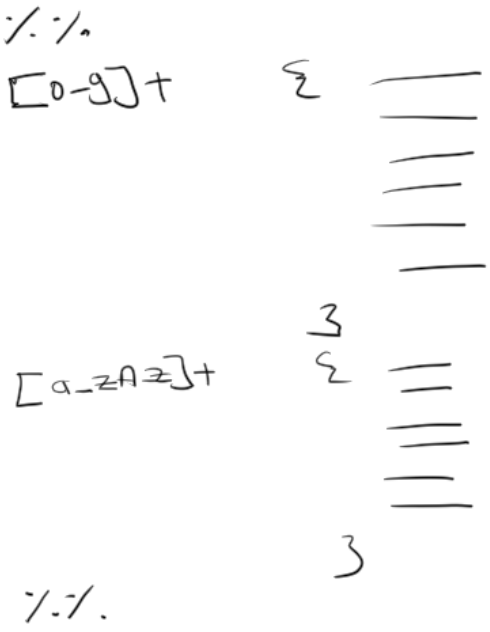
Flex kaynak dosyasndaki en önemli bölüm "Kurallar (Rules)" bölümüdür. Kurallar bölümünün iki %% karakterlerinin arasındaki bölüm olduğunu anımsayınız. Bu bölüm "kalıp (pattern)" ve "yapılacak eylem (action)" çiftlerinden oluşmaktadır:

```
kalıp (pattern)   yapılacak eylem (action)
```

Kalıp en soldaki sütuna dayalı olarak (unindented) biçimde yazılmak zorundadır. (Yani kalıpları yazarken soldan boşluk karakterleri bile veremeyiz.) Kalıp düzenli ifadelerden oluşur. Kalıptan sonraki boşluk karakterleri atılarak ilk boşluksuz karakterden satır sonuna kadarki karakterler "yapılacak eylemi (action)" olarak belirlenmektedir. Örneğin:



Yapılacak eylem C kodlarından oluşmaktadır. Başka bir deyişle "yapılacak eylem (action)" ilgili kalıp bulunduğunda çalıştırılacak C kodlarını belirtir. Eğer "yapılacak eylem" bir satırdan fazlaysa blok açılarak kodlar istenildiği kadar uzatılabilir. Örneğin:



Aslında bir kalıp için bir "yapılacak eylem" kısmının bulunması da zorunlu değildir. eğer kalıp için bir "yapılacak eylem" belirtilmediyse o kalıp bulunduğunda hiçbir şey yapılmaz.

Flex'in ürettiği kod ("lex.yy.c" dosyasındaki kod) default durumda lexical analiz işlemine tutulacak kaynak metni "stdin" dosyasından (yani klavyeden) okumaktadır. Eğer programcı girişlerin "stdin" dosyasından değil de kendi istediği bir dosyadan alınması istiyorsa ilgili dosyayı fopen fonksiyonyla açıp dosya bilgi göstericisini de Flex'in global yyin isimli FILE * türünden değişkenine atamalıdır. Örneğin:

```
%option noyywrap

%{
#include <stdio.h>
#include <stdlib.h>
%}
```

```
%%
[0-9]+      {
            printf("digit\n");
            }
\n
.
%%

int main(void)
{
    if ((yyin = fopen("test.txt", "r")) == NULL) {
        fprintf(stderr, "cannot open file!\n");
        exit(EXIT_FAILURE);
    }

    yylex();

    return 0;
}
```

Burada artık "test.txt" dosyası lexical analiz işlemine sokulacaktır. Kodu aşağıdaki gibi derleyebilirsiniz:

```
flex –osample.c sample.l
gcc –o sample sample.c (ya da Microsoft derleyicilerinde “cl sample.c”)
```

Burada “tanımlamalar (definitions)” bölümündeki %{ ve %} kısmında iki include işleminin yapıldığını görüyorsunuz. Tanımlamalar bölümünü biraz ileride ele alacağız. yyin değişkeni default olarak “stdin” dosyasına ilişkin dosya bilgi göstericisini tutmaktadır. Biz örneğimizde yyin değişkenine “test.txt” dosyasına ilişkin dosya bilgi göstericisini atayarak yylex fonksiyonunun bu dosyadan okuma yapmasını sağladık. Şimdi de aşağıdaki örneği inceleyiniz:

```
%option noyywrap

%{
#include <stdio.h>
#include <stdlib.h>
%}

%%
[0-9]+      {
            printf("digit\n");
            }
\n
.
%%

int main(int argc, char *argv[])
{
    if (argc > 2 ) {
        fprintf(stderr, "wrong number of arguments!\n");
        exit(EXIT_FAILURE);
    }
    if (argc == 2)
        if ((yyin = fopen(argv[1], "r")) == NULL) {
            fprintf(stderr, "cannot open file!\n");
            exit(EXIT_FAILURE);
        }

    yylex();

    return 0;
}
```

Kodu aşağıdaki gibi derleyebilirsiniz:

```
flex –osample.c sample.l
gcc –o sample sample.c (ya da Microsoft derleyicilerinde “cl sample.c”)
```

Bu örnekte komut satırı argümanıyla verilen dosya fopen fonksiyonuyla açılarak dosyaya ilişkin dosya bilgi göstericisi yyin değişkenine atanmıştır. Böylece programın girdilerinin komut satırı argümanı ile verilen dosyadan okunması sağlanmıştır. Programı aşağıdaki gibi komut satırı argümanı vererek çalıştırmalısınız:

```
sample test.txt       (Windows)
./sample test.txt     (Linux/Mac OS X)
```

Flex default durumda kalıba uymayan bütün karakterleri “stdout” dosyasına yazdırmaktadır. Bu durumda örneğin aşağıdaki flex programı dosya içerisindeki “ankara” yazılarını diğer karakterleri değiştirmeden “ANKARA” olarak ekrana yazdıracaktır:

```
%option noyywrap

%{
#include <stdio.h>
#include <stdlib.h>
%}

%%
ankara  printf("ANKARA");
%%

int main(int argc, char *argv[])
{
    if (argc > 2 ) {
        fprintf(stderr, "wrong number of arguments!\n");
        exit(EXIT_FAILURE);
    }
    if (argc == 2)
        if ((yyin = fopen(argv[1], "r")) == NULL) {
            fprintf(stderr, "cannot open file!\n");
            exit(EXIT_FAILURE);
        }

    yylex();

    return 0;
}
```

Eğer kalıba uymayan karakterlerin “stdout” dosyasına yazdırılmasını istemiyorsak “kurallar” kısmının sonuna “yapılacak eylem (action)” kısmı olmayan “.” ve “\n” kalıplarını yerleştirebiliriz. (Anımsanacağı gibi düzenli ifadelerde “.” meta karakteri “\n dışındaki herhangi bir karakter” anlamına gelmektedir.) Örneğin:

```
%%
ankara  printf("ANKARA");
\n
.
%%
```

Artık “ankara” kalıbına uymayan her karakter “.” ya da “\n” kalıbına uymaktadır. Bu kalıplar için birşey yapılmadığından “ankara” kalıbının dışındaki karakterlerin “stdout” dosyasına yazılması engellenmiş olur.

Flex’te “kurallar” kısmında belirtilen kalıplar için iki önemli nokta vardır:

1) Eğer yazıdaki bir kısım birden fazla kuraldaki kalıba uyuyorsa en uzun uygun kalıba ilişkin eylem işletilir. Örneğin:

```
%%
\+      printf(“+ operatörü\n”);
\+=     printf(“+= operatörü\n”);
%%
```

Burada kalıptaki ‘+’ karakteri meta karakter değil normal ‘+’ karakteri olduğu için ters bölülenmiştir. Yukarıdaki kurallarda yazıda eğer tek bir + görülmüşse birinci kalıp bununla uyuşur (match eder). Fakat += görülmüşse ikinci kalıp daha uzun olduğu için ikinci kalıp uyuşur. Benzer biçimde:

```
%%
ali         printf(“ali bulundu\n”);
aliye       printf(“aliye bulundu\n”);
%%
```

Burada yazı içerisinde “alim” gibi bir dizilim varsa bununla birinci kalıp uyuşacaktır. Ancak “aliyem” gibi bir yazıyla ikinci kalıp uyuşacaktır.

2) Eğer yazıdaki bir kısım birden fazla kuraldaki kalıba uyuyorsa ve bunlar da aynı uzunluktaysa daha yukarıda yazılmış olan kalıbın “yapılacak eylem (action)” kısmı işletilir. Örneğin:

```
%%
123         printf("birinci kural\n");
[0-9]+      printf("ikinci kural\n");
.
\n
%%
```

Burada örneğin yazıda “123” gibi bir dizilim görülürse bu aslında eşit uzunluklu olarak birinci kalıba da ikinci kalıba da uymaktadır. Ancak Flex bu durumda daha yukarıya yazılmış kalıbın eylem kısmını işletecektir.

Kurallar bölümünün sentaksı maddeler halinde aşağıdaki gibi özetlenebilir:

1) Kurallardaki kalıplar en soldaki sütuna dayalı (unindented) yazılmalıdır.

2) Kurallardaki kalıplar içını “yapılacak eylem (action)” kısmı hiç belirtilmeyebilir. Bu durumda bu kalıp ile uyuşma durumunda birşey yapılmayacaktır.

3) Kuralların “yapılacak eylem (action)” kısımları tek bir satırda bulunmak zorundadır. Eğer bunların birden fazla satırda yer alması isteniyorsa bloklama yapılmalıdır. Blokların içleri istenildiği gibi yazılabilir. Bu durumda kalıbın “yapılacak eylem (action)” kısmının blok kapatıldığında sonlandığı kabul edsilmektedir.

4) Kurallardaki kalıplar en soldaki sütuna dayalı olarak yazılmalıdır. Eğer kalıplar en soldaki sütuna dayalı olarak yazılmazlarsa arık bunlar kalıp olmaktan çıkmaktadır. Flex dokümanları kalıp olarak ele alınmayan bu karakterlerin sanki C koduymuş gibi ele alınacağını ve üretilen C koduna aktarılacağını belirtmektedir. Ancak dokümanlar bu karakterlerin üretilen C kodunun neresine aktarılacağı konusunda bir garanti vermemektedir. O halde bir satıra kalıbı olmayan bir kod yazmak iyi bir teknik değildir. Ancak Flex dokümanları özel bir durum olarak “kurallar kısmının” başındaki kodların yylex fonksiyonunun ana bloğunun başına aktarılacağı konusunda garanti vermektedir. (Yani biz örneğin “kurallar” kısmının başında bildirim yaparsak bu bildirilen değişkenler yylex fonksiyonunun yerel değişkenleri gibi olacaktır.)

5) Kurallar kısmında (ileride de görüleceği gibi “tanımlamalar” kısmında da) %{ ile %} arasındaki bölümler yine üretilen hedef koda aktarılmaktadır. %{ ve %} karakterleriin en soldaki sütuna dayalı olarak yazılması (unidented) gerekmektedir. Ancak bunların arasındaki kodlar istenildiği gibi yazılabilirler. Örneğin:

```
%%
%{
    int x = 0;
%}
[0-9]+ {
    if (x == 0) {
        /* ...
    }
}
%%
```

Aslında biz yukarıdaki Flex kodunu %{ ve %} karakterlerini kullanmadan şöyle de yazabilirdik:

```
%%
    int x = 0;
[0-9]+ {
    if (x == 0) {
        /* ...
    }
}
%%
```

Ancak %{ ... %} bloklaması ile birden fazla satıra yayılmış biçimde kodlar oluşturulabilmektedir.

Şimdi de Flex kaynak dosyasının başında bulunması gereken "tanımlamalar (definitinos)" bölümü üzerinde duralım. Tanımlamalar bölümü flex kaynak dosyasının başından ilk %% karakterlerine kadarki bölümdür. Tanımlamalar bölümünde üç şey bulunabilmektedir:

1) %option ile başlayan Flex direktifleri
2) Flex Makroları
3) Üretilecek koda yerleştirilecek global düzeydeki C kodları

%option ile başlayan satırlara Flex direktifleri denilmektedir. Flex direktifleri Flex kaynak dosyasının Flex programına bazı yönergeler vermek için kullanılmaktadır. Zorunlu olmasa da Flex direktifleri genellikle kaynak kodun tepesine yerleştirilirler. Örneğin yukarıdaki iskelet programımızda da yywrap fonksiyonunu kullanmak istemediğimizi belirtmek için kaynak kodun tepesine aşağıdaki direktifi yerleştirmiştik:

```
%option noyywrap
```

%option Fleks direktifleri en soldaki sütuna dayalı olarak (unindented) yazılmak zorundadır.

Flex makroları yazımı kolaylaştırmak ve okunabilirliği artırmak için kullanılmaktadır. Makroların genel formatı şöyledir:

```
isim (name)    kalıp (pattern)
```

Makrolar da en soldaki sütuna dayalı olarak (unindented) yazılmak zorundadır. İsimden sonra boşluk karakterleri atılarak elde edilen ilk boşluksuz yazı kümesi kalıbı oluşturmaktadır. Örneğin:

```
Constant    [\+-]?[0-9]+
Identifer   [_a-zA-Z]+
```

Kurallar bölümündemakrodaki isimler küme parantezlerine alınırsa bu isimler yerine ona karşı gelen kalıpların kullanıldığı kabul edilir. Örneğin:

```
%option noyywrap

Constant       [\+-]?[0-9]+
Identifier     [_a-zA-Z]+

%%

{Constant}         printf("digit\n");
{Identifier}       printf("identifier\n");
.
\n
%%
```

Tanımlamalar kısmında sola dayalı olarak yazılmayan (indented) her şey Flex tarafından C kodu olarak ele alınmaktadır. Buradaki C kodları Flex tarafından üretilen kodun global alanına yerleştirilmektedir. Programcılar Flex'te kullanacakları global değişkenlerin ve fonksiyon prototiplerini tipik olarak tanımlamalar bölümünde bu biçimde yaparlar. Örneğin:

```
%option noyywrap

    int g_x;       /* Global değişken tanımlaması */

Constant       [\+-]?[0-9]+
Identifier     [_a-zA-Z]+

%%
```

Yine tanımlamalar bölümünde %{ ile %} karakterleri arasındaki kısım C kodu olarak üretilen kodda global alana aktarılmaktadır. %{ ve %} karakterleri soldaki sütuna dayalı olarak (unindented) yazılmak zorundadır. Ancak bunların içi herhangi bir biçimde yazılabilir. %{ ve %} içerisine C kodlarının yazılması daha serbest ve düzenli bir görünüm sağlamaktadır. Örneğin:

```
%option noyywrap

%{
        #include <stdio.h>
        #include <stdlib.h>

        int g_x;       /* Global değişken tanımlaması */

%}

Constant       [\+-]?[0-9]+
Identifier     [_a-zA-Z]+

%%
```

İstisna olarak C kodları için yorumlama başlangıcı (yani /* karakterleri) en soldaki sütuna dayalı biçimde (unindented) yazılabilmektedir. Yorumlama kısımları da üretilen koda doğrudan aktarılmaktadır. Örneğin:

```
%option noyywrap

%{
        #include <stdio.h>
        #include <stdlib.h>
%}

/* Global Tanımlamalar */
    int g_x;       /* Global değişken tanımlaması */
```

```
Constant        [\+-]?[0-9]+
Identifier      [_a-zA-Z]+

%%
```

Burada /* Global Tanımlamalar */ yorumlamasının en soldaki sütuna dayalı olarak yazılabildiğine dikkat ediniz. Genel olarak Flex'te her zaman C kodları için C stili yorumlama kullanılabilmektedir. Ancak makro satırlarında youmlama yapılamaz ve kurallar bölümünde de sola dayalı olarak yorumlama yapılamaz.

Flex kaynak dosyasında ikinci %% karakterlerinden sonraki bölüme "kullanıcı Kodları (user codes)" bölümü denilmektedir. Bu bölüm tamamen C kodlarına ayrılmıştır. Burada biz her türlü global tanımlamaları ve bildirimleri (örneğin fonksiyon ve global nesne tanımlamaları, tür bildirimleri gibi) yapabiliriz. Tabii bu kısımda bildirilmiş olan değişkenleri kalıplar kısmında kullanamazsınız. Eğer birtakım değişkenleri her yerde kullanmak istiyorsanız onların bildirimlerini "tanımlamalar (definitions)" bölümünde yapmalısınız. Ayrıca eğer istenirse "kullanıcı kodları (user codes)" bölümü tamamen boş da bırakılabilmektedir.

3.5.1.4. Flex'in Önemli Global Değişkenleri

Flex ne zaman yyin değişkeni ile belirlenen dosyada kalıba uygun bir atom bulsa o atomu yytext isimli bir değişkenin belirttiği adrese -sonunda '\0' karakterini de ekleyerek- yerleştirir. yytext Flex tarafından char türden bir dizi ya da bir gösterici olarak bildirilebilmektedir. (Tabii her iki durumda da yytext adresinin gösterdiği yer tahsis edilmiş durumdadır.) Default durumda yytext değişkenini char türden bir göstericidir. Ancak bu durum %array ve direktifi ile değiştirilebilir. Aşağıdaki örneği inceleyiniz:

```
%option noyywrap

%{
        #include <stdio.h>
        #include <stdlib.h>
%}

Constant        [\+-]?[0-9]+
Identifier      [_a-zA-Z]+

%%
{Constant}          printf("Constant: %s\n", yytext);
{Identifier}        printf("identifier: %s\n", yytext);
\n
.
%%

int main(int argc, char *argv[])
{
    if (argc > 2 ) {
        fprintf(stderr, "wrong number of arguments!\n");
        exit(EXIT_FAILURE);
    }
    if (argc == 2)
        if ((yyin = fopen(argv[1], "r")) == NULL) {
            fprintf(stderr, "cannot open file!\n");
            exit(EXIT_FAILURE);
        }

    yylex();

    return 0;
}
```

Kodu şöyle derleyebilirsiniz:

```
flex –osample.c sample.l
gcc –o sample sample.c (ya da Microsoft derleyicilerinde “cl sample.c”)
```

Kodun aşağıdaki gibi çalıştırıldığını varsayalım:

```
sample test.txt   (ya da Linux ve Mac OS X'te “./sample test.txt”)
```

Örneğin buradaki "test.txt" dosyasının içeriği şöyle olsun:

```
ankara izmir 123 456789, ali, vel, selami
```

Programın çıktısı da şöyle olacaktır:

```
identifier: ankara
identifier: izmir
Constant: 123
Constant: 456789
identifier: ali
identifier: vel
identifier: selami
```

Flex’in int türden yyleng isimli global değişkeni, bulunan atomun karakter uzunluğunu bize verir. Yani başka bir deyişle yyleng bize strlen(yytext) değerini vermektedir. Biz yine bunu istediğimiz yerde kullanabiliriz. Örneğin:

```
%option noyywrap

%{
        #include <stdio.h>
        #include <stdlib.h>
%}

Constant        [\+-]?[0-9]+
Identifier      [_a-zA-Z]+

%%
{Constant}          printf("Constant: %s (%d)\n", yytext, yyleng);
{Identifier}        printf("identifier: %s (%d)\n", yytext, yyleng);
\n
.
%%

int main(int argc, char *argv[])
{
    if (argc > 2 ) {
        fprintf(stderr, "wrong number of arguments!\n");
        exit(EXIT_FAILURE);
    }
    if (argc == 2)
        if ((yyin = fopen(argv[1], "r")) == NULL) {
            fprintf(stderr, "cannot open file!\n");
            exit(EXIT_FAILURE);
        }

    yylex();

    return 0;
}
```

Nasıl yyin atomlarına ayrılacak dosyayı belirtiyorsa, yyout da kalıp ile uyuşmayan karakterlerin yazdırıldığı dosyayı belirtmektedir. Default durumda yyout stdout dosyasına ilişkindir. Yani kalıplara uymayan karakterler default durumda stdout dosyasına yazdırılırlar.

3.5.1.5. yylex Fonksiyonunun Atomlar İçin Tek Tek Çağrılması

Yukarıdaki örneklerde biz main fonksiyonu içerisinde yalnızca bir kez yylex fonksiyonunu çağırdık. Bu fonksiyon başından sonuna kadar yyin ile belirtilen dosyadaki yazıyı girdi olarak kullanıyordu. Halbuki biz yylex fonksiyonunun her çağrıda sıradaki atomu vermesini sağlayabiliriz. Bunun için "kurallar (rules)" bölümünde kalıplara karşı gelen "yapılacak eylem (action)" kısımlarında yylex'in return deyimi ile sonlandırılması gerekir. Örneğin:

```
%option noyywrap

%{
#include <stdio.h>
#include <stdlib.h>

#define CONSTANT           1
#define IDENTIFIER         2
%}

Constant       [\+-]?[0-9]+
Identifier     [_a-zA-Z]+

%%
{Constant}         return CONSTANT;
{Identifier}       return IDENTIFIER;
\n
.
%%

int main(int argc, char *argv[])
{
    int tokenId;

    if (argc > 2 ) {
        fprintf(stderr, "wrong number of arguments!\n");
        exit(EXIT_FAILURE);
    }
    if (argc == 2)
        if ((yyin = fopen(argv[1], "r")) == NULL) {
            fprintf(stderr, "cannot open file!\n");
            exit(EXIT_FAILURE);
        }

    while ((tokenId = yylex()) != 0) {
        switch (tokenId) {
            case CONSTANT:
                printf("Constant: %s (%d)\n", yytext, yyleng);
                break;
            case IDENTIFIER:
                printf("Identifier: %s (%d)\n", yytext, yyleng);
                break;
        }
    }

    return 0;
}
```

Flex kaynak kodunda da görüldüğü gibi yylex fonksiyonu her çağrıldığında bir atomu bulur bulmaz return deyimi ile sonlandırılmıştır. yylex dosyanın sonuna geldiğinde 0 ile geri dönmektedir. Yukarıdaki Flex dosyasının "kullanıcı kodu" kısmındaki while döngüsünün "yylex fonksiyonu sıfır değeri ile geri dönmediği sürece" devam ettirildiğini görüyorsunuz. yylex atomu bulursa bizim belirlediğimiz değerle geri dönmektedir. Örnek kodda daha sonra yylex'in geri dönüş değerinin switch içerisine sokularak ele alındığını görüyorsunuz.

Bazen kurallarda yapılacak işlemler çok uzun olabilir. Bu durumda bu işlemlerin kurallar kısmında yapılması yerine bir fonksiyona yaptırılması daha uygun olur. Örneğin bir sabitle karşılaşıldığında sabitin istenilen limit dışında olup olmadığı kontrol edilmek istensin. Bu işlem aşağıdaki gibi yapılabilir:

```
%option noyywrap

%{
#include <stdio.h>
#include <stdlib.h>
#include <limits.h>

#define CONSTANT           1
#define INVALID_CONSTANT   2
#define IDENTIFIER         3

int CheckConstant(void);

%}

Constant       [\+-]?[0-9]+
Identifier     [_a-zA-Z]+

%%
{Constant}         return CheckConstant();
{Identifier}       return IDENTIFIER;
\n
.
%%

int CheckConstant(void)
{
    double result;

    result = atof(yytext);

    if (result > LONG_MAX || result < LONG_MIN)
        return INVALID_CONSTANT;
    return CONSTANT;
}

int main(int argc, char *argv[])
{
    int tokenId;

    if (argc > 2 ) {
        fprintf(stderr, "wrong number of arguments!\n");
        exit(EXIT_FAILURE);
    }
    if (argc == 2)
        if ((yyin = fopen(argv[1], "r")) == NULL) {
            fprintf(stderr, "cannot open file!\n");
            exit(EXIT_FAILURE);
        }

    while ((tokenId = yylex()) != 0) {
        switch (tokenId) {
            case CONSTANT:
                printf("Constant: %s (%d)\n", yytext, yyleng);
                break;
            case INVALID_CONSTANT:
                printf("Invalid Constant: %s (%d)\n", yytext, yyleng);
                break;
            case IDENTIFIER:
                printf("Identifier: %s (%d)\n", yytext, yyleng);
                break;
```

```
        }
    }

    return 0;
}
```

CheckConstant fonksiyonunun prototip bildiriminin dosyanın "tanımlamalar" kısmında yapıldığına dikkat ediniz. Tabii Flex bildirimleri çok uzunsa biz bu bildirimleri bir başlık dosyasına yerleştirip onu da include edebiliriz. Örneğin:

```
/* sample.l */

%option noyywrap

%{
#include "sample.h"
%}

Constant      [\+-]?[0-9]+
Identifier    [_a-zA-Z]+

%%
{Constant}        return CheckConstant();
{Identifier}      return IDENTIFIER;
\n
.
%%

int CheckConstant(void)
{
    double result;

    result = atof(yytext);

    if (result > LONG_MAX || result < LONG_MIN)
        return INVALID_CONSTANT;
    return CONSTANT;
}

int main(int argc, char *argv[])
{
    int tokenId;

    if (argc > 2 ) {
        fprintf(stderr, "wrong number of arguments!\n");
        exit(EXIT_FAILURE);
    }
    if (argc == 2)
        if ((yyin = fopen(argv[1], "r")) == NULL) {
            fprintf(stderr, "cannot open file!\n");
            exit(EXIT_FAILURE);
        }

    while ((tokenId = yylex()) != 0) {
        switch (tokenId) {
            case CONSTANT:
                printf("Constant: %s (%d)\n", yytext, yyleng);
                break;
            case INVALID_CONSTANT:
                printf("Invalid Constant: %s (%d)\n", yytext, yyleng);
                break;
            case IDENTIFIER:
```

```
                printf("Identifier: %s (%d)\n", yytext, yyleng);
                break;
        }
    }

    return 0;
}

/* sample.h */

#ifndef SAMPLE_H_
#define SAMPLE_H_

#include <stdio.h>
#include <stdlib.h>
#include <limits.h>

/* Symbolic Constants */

#define CONSTANT          1
#define INVALID_CONSTANT  2
#define IDENTIFIER        3

/* Function Prototypes */

int CheckConstant(void);

#endif
```

Yine derleme işlemi şöyle yapılabilir:

```
flex –osample.c sample.l
gcc –o sample sample.c (ya da Microsoft derleyicilerinde “cl sample.c”)
```

Programı çalıştırmadan önce bir test dosyası hazırlamalısınız. Örneğin:

```
sample test.txt   (Linux ve Mac OS X sistemlerinde “./sample test.txt)
```

3.5.1.6. Flex’in Ürettiği Kodlarla Programcının Kodlarının Farklı Dosyalarda Bulundurulması

Flex default olarak içerisinde atomlara ayırma işlemini yapan yylex fonksiyonun bulunduğu “lex.yy.c” isminde bir C kaynak kod dosyası üretmektedir. Anımsanacağı gibi biz Flex dosyasının “Kullanıcı Kodları (User Codes)” bölümünde main fonksiyonunu ve diğer fonksiyonları bulundurabiliyorduk. Aslında bir Flex dosyasının “Kullanıcı Kodları (User Codes)” kısmı tamamen ayrı bir C kaynak dosyasına da taşınabilir. (Bu bölümün Flex dosyasında boş olarak bulundurulabileceğini anımsayınız.) Tabii bizim “kullanıcı kodlarını” ayrı bir dosyaya taşımamız durumunda Flex değişkenlerinin extern bildirimlerinin de bu dosyada bulunması gerekmektedir. Bu bildirimleri tek tek elle yazmak yerine Flex’ten bunların bulunduğu bir başlık dosyasını üretmesini isteyebiliriz. Bu işlem iki biçimde yapılabilmektedir: Birinci seçenek Flex kaynak kodu işleme sokulurken “–header-file=dosya ismi” seçeneğinin eklenmesidir. (Ancak maalesef Flex’in Windows GNU versiyonu henüz bunu desteklememektedir. Bunun için flex’in diğer bir Windows nuyarlaması olan “win_flex.exe” programını kullanmalısınız) İkinci seçenek Flex kaynak kodunun “Tanımlamalar (Definitions)” kısmında %option header-file=”dosya ismi” tanımlamasının yapılmasıdır. (Ancak maalesef Flex’in Windows GNU versiyonu bunu da henüz desteklememektedir. Bunun için Flex’in diğer Windows sürümü olan “win_flex.exe” programını kullanmalısınız).

Örneğin bir Flex uygulamasını Visual Studio kullanarak oluşturmak isteyelim (010-FlexWithVisualStudio). Bu işlemi şu adımlardan geçerek yapabiliriz:

1) Flex kaynak dosyası (örneğimizde sampleflex.l) "--header-file=yylex.h" seçeneği ile aşağıdaki gibi işleme sokulur:

```
win_flex --wincompat --header-file=lex.yy.h sampleflex.l
```

Buradan i "lex.yy.c" ve "lex.yy.h" isminde iki dosya elde edilecektir:. "lex.yy.c" dosyasında atomlara ayırma işlemini yapan C kodları, "lex.yy.h" dosyasında ise Flex değişkenlerinin extern bildirimleri bulunacaktır. (Komut satırındaki --wincompat seçeneği ileride ele alınacaktır.)

2) Şimdi "Kullanıcı Kodları (User Codes)" blümündekileri başka bir C dosyasına alabiliriz. (Örneğin bu dosyamızın ismi "sampleflexemplc" olsun.) Tabii bu dosya içerisinde Flex'in değişkenleri kullanılacağı için bu dosyadan "lex.yy.h" dosyasının include edilmesi gerekmektedir.

3) Eğer programcı isterse yine kendi dosyası için de bir başlık dosyası hazırlayabilir. Mademki Flex'in kendisi de programcının birtakım bildirimlerini kullanmaktadır. O halde bu bildirimler ortak bir başlık dosyasında toplanabilir. (Örneğimizde bu başlık dosyası "sampleflexemplh" ismindedir.

4) Visual Studio'da yaratılacak projenin içerisine "lex.yy.c" ve "sampleflexemplc" dosyalarının eklenmesi gerekir. Başlık dosyaları include edildiği için onların doğrudan projeye eklenmesine gerek yoktur. Ancak başlık dosyalarında bir değişiklik yapıldığında kaynak kodların yeniden derlenmesi isteniyorsa bunlar da projeye eklenebilir.

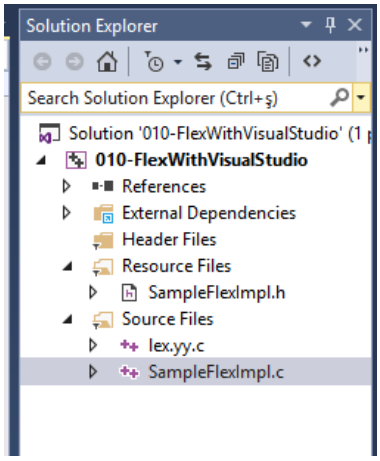
Maalesef Flex'in "win_flex.exe" Windows uyarlamasında üretilen başlık dosyasında (örneğimizdeki "lex.yy.h" dosyası) UNIX/Linux sistemlerindeki bazı POSIX fonksiyonlarının bulunduğu "unistd.h" dosyası include edilmiştir. Windows sistemlerinde bu dosya olmadığı için derlemede sorunlar oluşabilmektedir. Bunu sorun üretilen "lex.yy.c" ve "sampleflexemplc" dosyalarının başına YY_NO_UNISTD_H makrosunun define edilmesi ile engellenebilir. (Bu define işlemini Visual Studio'da proje özelliklerindeki "C-C++/Preprocessor" seçeneğiseçeneği ile de yapabilirsiniz.)

Örneğimizde kullanılan dosyalar şöyledir:

```
/* sampleflex.l */

%option noyywrap
%{
#include "sampleflexemplh"
%}

Constant      [\+-]?[0-9]+
Identifier    [_a-zA-Z]+

%%
{Constant}        return CheckConstant();
{Identifier}      return IDENTIFIER;
\n
.
%%
```

```
/* sampleflexemplh */

#ifndef SAMPLEFLEXEMPL_H_
#define SAMPLEFLEXEMPL_H_

#include <stdio.h>
#include <stdlib.h>
#include <limits.h>

/* Symbolic Constants */
```

```
#define CONSTANT          1
#define INVALID_CONSTANT  2
#define IDENTIFIER        3

/* Function Prototypes */

int CheckConstant(void);

#endif

/* samplefleximpl.c */

#include <stdio.h>
#include <stdlib.h>
#include "samplefleximpl.h"
#include "lex.yy.h"

int CheckConstant(void)
{
    double result;

    result = atof(yytext);

    if (result > LONG_MAX || result < LONG_MIN)
        return INVALID_CONSTANT;
    return CONSTANT;
}

int main(int argc, char *argv[])
{
    int tokenId;

    if (argc > 2) {
        fprintf(stderr, "wrong number of arguments!\n");
        exit(EXIT_FAILURE);
    }
    if (argc == 2)
        if ((yyin = fopen(argv[1], "r")) == NULL) {
            fprintf(stderr, "cannot open file!\n");
            exit(EXIT_FAILURE);
        }

    while ((tokenId = yylex()) != 0) {
        switch (tokenId) {
        case CONSTANT:
            printf("Constant: %s (%d)\n", yytext, yyleng);
            break;
        case INVALID_CONSTANT:
            printf("Invalid Constant: %s (%d)\n", yytext, yyleng);
            break;
        case IDENTIFIER:
            printf("Identifier: %s (%d)\n", yytext, yyleng);
            break;
        }
    }

    return 0;
}
```



Yukarıdaki işlemleri Linux ya da Mac OS X'te de şöyle yapabiliriz. (Aynı dosya isimlerinin küçük harflerle oluşturulduğunu varsayalım)

1) Flex kaynak dosyası aşağıdaki gibi derlenir:

```
flex --header-file=lex.yy.h sampleflex.l
```

2) "sampleflexImpl.c" dosyası oluşturulur. Yine ortak bildirimler "sampleflexImpl.h" dosyasında bulundurulur. "sampleflexImpl.c" dosyasından sampleflex.h ve yylex.h dosyaları include edilir.

3) Oluşturulan bu iki C dosyası aşağıdaki gibi derlenerek birlikte bağlanır:

```
gcc -o sampleflex yylex.c sampleflexImpl.c
```

Anahtar Notlar: Flex ve Bison ile Visual Studio'da hiç komut satırına geçmeden de çalışabilir. Bunun için öncelikle "Flex" ve "Bison" araçlarının Visual Studio build sistemine tanıtılması gerekir. Bu tanıtım Flex ve Bison'un diğer Windows uyarlaması olan "WinFlexBison" dağıtımında önceden hazırlanmış dosyalarla yapılabilmektedir. Visual Studio'da proje seçenekleri üzerinde bağlam menüsünden "Build Dependencies/Build Custimization" seçilir. Buradan da "Find Existing" düğmesi ile "WinFlexBison" dağıtımındaki "custom_build_tools" dizinindeki dosyalar seçilir. Artık biz projeye doğrudan ".l" ve ".y" uzantılı dosyaları ekleyebiliriz. Bu dosyalar "win_flex.exe" ve "win_bison.exe" programları tarafından çalıştırılacaktır. Tabii burada "win_flex .exe" ve "win_bison.exe" programları tarafından üretilen dosyaların da ayrıca projeye manuel olarak eklenmesi gerekir.

3.5.1.7. C Programlama Dilini Atomlarına Ayıran Flex Kodu

Internet'te standart pek çok programlama dili için başkaları tarafından yazılmış olan "flex" kodlarını bulabilirsiniz. (Bunun için Google'da "C grammar lex" gibi bir arama yapılabilir.) Örneğin https://www.lysator.liu.se/c/ANSI-C-grammar-l.html#comment sitesinde " ANSI C Grammar, Lex Specification" isimli başlık altında aşağıdaki gibi bir flex (ya da lex) kodu verilmiştir.

```
D           [0-9]
L           [a-zA-Z_]
H           [a-fA-F0-9]
E           [Ee][+-]?{D}+
FS          (f|F|l|L)
IS          (u|U|l|L)*

%{
#include <stdio.h>
#include "y.tab.h"

void count();
%}

%%
"/*"            { comment(); }
```

```
"auto"          { count(); return(AUTO); }
"break"         { count(); return(BREAK); }
"case"          { count(); return(CASE); }
"char"          { count(); return(CHAR); }
"const"         { count(); return(CONST); }
"continue"      { count(); return(CONTINUE); }
"default"       { count(); return(DEFAULT); }
"do"            { count(); return(DO); }
"double"        { count(); return(DOUBLE); }
"else"          { count(); return(ELSE); }
"enum"          { count(); return(ENUM); }
"extern"        { count(); return(EXTERN); }
"float"         { count(); return(FLOAT); }
"for"           { count(); return(FOR); }
"goto"          { count(); return(GOTO); }
"if"            { count(); return(IF); }
"int"           { count(); return(INT); }
"long"          { count(); return(LONG); }
"register"      { count(); return(REGISTER); }
"return"        { count(); return(RETURN); }
"short"         { count(); return(SHORT); }
"signed"        { count(); return(SIGNED); }
"sizeof"        { count(); return(SIZEOF); }
"static"        { count(); return(STATIC); }
"struct"        { count(); return(STRUCT); }
"switch"        { count(); return(SWITCH); }
"typedef"       { count(); return(TYPEDEF); }
"union"         { count(); return(UNION); }
"unsigned"      { count(); return(UNSIGNED); }
"void"          { count(); return(VOID); }
"volatile"      { count(); return(VOLATILE); }
"while"         { count(); return(WHILE); }

{L}({L}|{D})*     { count(); return(check_type()); }

0[xX]{H}+{IS}?    { count(); return(CONSTANT); }
0{D}+{IS}?      { count(); return(CONSTANT); }
{D}+{IS}?       { count(); return(CONSTANT); }
L?'(\\.|[^\\'])+' { count(); return(CONSTANT); }

{D}+{E}{FS}?      { count(); return(CONSTANT); }
{D}*"."{D}+({E})?{FS}?    { count(); return(CONSTANT); }
{D}+"."{D}*({E})?{FS}?    { count(); return(CONSTANT); }

L?\"(\\.|[^\\"])*\"   { count(); return(STRING_LITERAL); }

"..."           { count(); return(ELLIPSIS); }
">>="           { count(); return(RIGHT_ASSIGN); }
"<<="           { count(); return(LEFT_ASSIGN); }
"+="            { count(); return(ADD_ASSIGN); }
"-="            { count(); return(SUB_ASSIGN); }
"*="            { count(); return(MUL_ASSIGN); }
"/="            { count(); return(DIV_ASSIGN); }
"%="            { count(); return(MOD_ASSIGN); }
"&="            { count(); return(AND_ASSIGN); }
"^="            { count(); return(XOR_ASSIGN); }
"|="            { count(); return(OR_ASSIGN); }
">>"            { count(); return(RIGHT_OP); }
"<<"            { count(); return(LEFT_OP); }
"++"            { count(); return(INC_OP); }
"--"            { count(); return(DEC_OP); }
"->"            { count(); return(PTR_OP); }
```

```
"&&"            { count(); return(AND_OP); }
"||"            { count(); return(OR_OP); }
"<="            { count(); return(LE_OP); }
">="            { count(); return(GE_OP); }
"=="            { count(); return(EQ_OP); }
"!="            { count(); return(NE_OP); }
";"             { count(); return(';'); }
("{"|"<%")      { count(); return('{'); }
("}"|"%>")      { count(); return('}'); }
","             { count(); return(','); }
":"         { count(); return(':'); }
"="         { count(); return('='); }
"("         { count(); return('('); }
")"         { count(); return(')'); }
("["|"<:")      { count(); return('['); }
("]"|":>")      { count(); return(']'); }
"."         { count(); return('.'); }
"&"         { count(); return('&'); }
"!"         { count(); return('!'); }
"~"         { count(); return('~'); }
"-"         { count(); return('-'); }
"+"         { count(); return('+'); }
"*"         { count(); return('*'); }
"/"         { count(); return('/'); }
"%"         { count(); return('%'); }
"<"         { count(); return('<'); }
">"         { count(); return('>'); }
"^"         { count(); return('^'); }
"|"         { count(); return('|'); }
"?"         { count(); return('?'); }

[ \t\v\n\f]       { count(); }
.                 { /* ignore bad characters */ }

%%

yywrap()
{
    return(1);
}

comment()
{
    char c, c1;

loop:
    while ((c = input()) != '*' && c != 0)
        putchar(c);

    if ((c1 = input()) != '/' && c != 0)
    {
        unput(c1);
        goto loop;
    }

    if (c != 0)
        putchar(c1);
}


int column = 0;

void count()
```

```
{
    int i;

    for (i = 0; yytext[i] != '\0'; i++)
        if (yytext[i] == '\n')
            column = 0;
        else if (yytext[i] == '\t')
            column += 8 - (column % 8);
        else
            column++;

    ECHO;
}


int check_type()
{
/*
* pseudo code --- this is what it should check
*
*   if (yytext == type_name)
*       return(TYPE_NAME);
*
*   return(IDENTIFIER);
*/

/*
*   it actually will only return IDENTIFIER
*/

    return(IDENTIFIER);
}
```

Yukarıdaki Flex programını kullanan bir main fonksiyonu şöyle yazılabilir (“Src/011-CProgrammingLanguageFlex”):

```
#include <stdio.h>
#include <stdlib.h>
#include "cflex.h"
#include "lex.yy.h"

int main(int argc, char *argv[])
{
    int tokenId;

    if (argc > 2) {
        fprintf(stderr, "wrong number of arguments!\n");
        exit(EXIT_FAILURE);
    }
    if (argc == 2)
        if ((yyin = fopen(argv[1], "r")) == NULL) {
            fprintf(stderr, "cannot open file!\n");
            exit(EXIT_FAILURE);
        }

    while ((tokenId = yylex()) != 0)
        printf("%s\n", yytext);

    return 0;
}
```

İlgili Flex dosyasında tek karakterli atomların id’lerinin doğrudan o karakterlerin karakter tablosundaki sayısal değeri olarak alındığına dikkat ediniz. Bu durum kodlama bakımından bazı pratiklikler sağlamaktadır.

3.5.1.8. Flex'in Bazı Ayrıntıları

Flex normal olarak yyin ile belirtilen dosyadaki karakterleri atomlarına ayırmaya çalışır. (Anımsanacağı gibi default durumda yyin değişkeni "stdin" dosyasını belirtiyordu. İşte Flex bize yyrestart isimli bir fonksiyon da vermektedir. Bu fonksiyonun prototipi şöyledir:

```
void yyrestart(FILE *f);
```

Fonksiyon parametre olarak FILE * türünden dosya bilgi göstericisini alır. yyrestart fonksiyonu çağrıldığında artık yylex bu yeni dosyanın başından itibaren işlemine devam eder. Böylece bir Flex dosyası bittiğinde işlemlerin başka bir Flex dosyasından devam ettirilmesi sağlanabilmektedir:

```
%option noyywrap

%{
    #include <stdio.h>
    #include <stdlib.h>

    int g_lines;
    int g_words;
    int g_chars;
%}

%%
[0-9A-Za-z]+   { ++g_words; g_chars += yyleng;}
.              { ++g_chars; }
\n             { ++g_lines; ++g_chars; }
%%

int main(char argc, char *argv[])
{
    int i;
    FILE *f;

    if (argc == 1) {
        yylex();
        printf("%d %d %d\n", g_lines, g_words, g_chars);
        exit(EXIT_SUCCESS);
    }

    for (i = 1; i < argc; ++i) {
        if ((f = fopen(argv[i], "r")) == NULL) {
            fprintf(stderr, "cannot read file!..\n");
            exit(EXIT_FAILURE);
        }
        yyrestart(f);
        yylex();
        printf("%d %d %d %s\n", g_lines, g_words, g_chars, argv[i]);
        g_words = g_lines = g_chars = 0;
    }

    return 0;
}
```

Flex'in yywrap isimli fonksiyonundan daha önce söz etmiştik. Bu fonksiyon dosyanın sonuna gelindiğinde yylex tarafından çağrılmaktadır. Bazen dosya sonuna gelindiğinde bazı ayarlamaların yapılması istenebilir. İşte yywrap fonksiyonu bu amaçla kullanılmaktadır. Bu fonksiyonu yazan programcı eğer 0 ile geri dönerse yylex işlemine devam eder. (Örneğin bunun içerisinde yyrestart yapılmış olabilir.) Eğer sıfır dışı bir değerle geri dönerse yylex işlemini bitirir. Ancak biz yylex fonksiyonunun yywrap fonksiyonunu çağırmasını da Flex kaynak dosyasının başına "%option noyywrap" direktifini yerleştirerek engelleyebiliriz. Bu durumda yylex yywrap fonksiyonunu çağırmayacağı için bizim de bu fonksiyonu tanımlamamıza gerek kalmaz.

Flex'in ürettiği C kodu normal olarak atomlara ayıracağı dosyayı tamponlamaktadır. Yani Flex'in ürettiği kod dosyayı karakter karakter okumak yerine onun bir bölümünü belleğe çekip karakterleri oradan hızlı bir biçimde almaktadır. İşte biz bazen onun kullandığı bu tamponun değiştirilmesini de isteyebiliriz. Tampon YY_BUFFER_STATE isimli bir türle tespit edilmiştir. Belli bir uzunlukta yeni bir tampon yaratmak için yy_create_buffer fonksiyonu kullanılmaktadır. Örneğin:

```
YY_BUFFER_STATE bp;
...
bp = yy_create_buffer(yyin, 4096 );
```

Burada yy_create_buffer fonksiyonunun iki parametre aldığına dikkat ediniz. Birinci parametre FILE * türündendir ve atomlara ayrılacak dosyaya ilişkin dosya bilgi göstericisini alır, ikinci parametre ise tamponun uzunluğunu belirtmektedir. yylex'in yaratılan bu yeni tamponu kullanması da yy_switch_tobuffer fonksiyonuyla sağlanmaktadır. Örneğin:

```
yy_switch_to_buffer(bp);
```

Flex'in ürettiği yylex fonksiyonunun atomlarına ayıracağı yazı her zaman bir dosyada mı bulunmak zorundadır? Yanıt hayır. İşte yy_scan_string isimli fonksiyon yylex'in dosyadan değil de sonu '\0' ile biten yazıyı atomlarına ayırmasını sağlamaktadır. Fonksiyonun prototipi şöyledir:

```
YY_BUFFER_STATE yy_scan_string(yyconst char *yy_str);
```

yy_scan_string fonksiyonunun kullanımına ilişkin şöyle bir örnek verebiliriz:

```
%option noyywrap

%{
    #include <stdio.h>
    #include <stdlib.h>

    int g_lines;
    int g_words;
    int g_chars;
%}

%%
[0-9A-Za-z]+   { ++g_words; g_chars += yyleng;}
.              { ++g_chars; }
\n             { ++g_lines; ++g_chars; }
%%

int main(void)
{
    char str[] = "bu bir denemedir\nevet denemedir";
    YY_BUFFER_STATE bs;

    bs = yy_scan_string(str);
    yy_switch_to_buffer(bs);
    yylex();
    printf("%d %d %d\n", g_lines, g_words, g_chars);

    return 0;
}
```

yy_scan_string fonksiyonun yanı sıra sonu '\0' karakter ile bitmeyen yazıların atomların a ayrılması için yy_scan_buffer isimli bir fonksiyon da bulundurulmaktadır. Bu fonksiyon belli bir adresten belli miktarda karakterin taranmasını sağlar:

```
YY_BUFFER_STATE yy_scan_buffer(char *base, yy_size_t size);
```

Flex ile ilgili daha başka ayrıntılar da vardır. Ancak biz burada bu ayrıntıları ele almayacağız. Bunlar için Flex'in dokümanlarına bakabilirsiniz.

3.6. Derleyiciler ve Yorumlayıcılarda String Tablolarının Oluşturulması

Pek çok derleyici ya da yorumlayıcı ayrıştırdığı atomlar içerisinde değişkenleri ve sabitleri string tablosu denilen bir tabloya yerleştirmektedir. Örneğin aşağıdaki gibi bir C kodu atomlarına ayrılacak olsun:

```
int count, total;

count = 10;
total = 0;
```

Bu kod içerisinde count ve total değişken (identifer) isimlerinin birden fazla kez kullanıldığını görüyorsunuz. İşte lexical analiz modülü tarafından bu isimler ilk görüldüğünde string tablosu denilen bir tabloya yerleştirilmekte ve yeniden görüldüklerinde de artık o tablodan elde edilmektedir. Tabii aslında string tablolarında değişken isimleri doğrudan tutulmamaktadır. Değişken isimleri dinamik olarak tahsis edilen bir alana kopyalanmakta, string tablolarında yalnızca onların adresleri tutulmaktadır. String tablolarına yalnızca değişken isimleri değil sabitler de yerleştirilebilmektedir.

String tablolarının kullanılma gerekçesi iki maddeyle özetlenebilir:

1) String tabloları sayesinde aynı değişken isimlerinin ve sabitlerin gereksiz biçimde birden fazla kez parse ağacı içerisinde (ya da sembol tabloları) yer kaplaması engellenmiş olur.

2) String tablolarında her değişken ismi yalnızca bşir kez bulunduğundan parser modülü değişkenleri onları isimsel olarak yalnızca adreslerini kullanrak karşılaştırabilmektedir.

String tablolarının gerçekleştirilmesinde veri yapısı olarak genellikle "hash tabloları (hash tables)" kullanılmaktadır. (Çünkü hash tabloları arama konusunda oldukça hızlıdır. "Hash Tabloları" konusu "Sistem Programlama ve İleri C Uygulamaları-1" isimli kursta ele alınmıştı.) Burada bir uyarıda bulıunmak istiyoruz: String tablosunu "Sembol Tablosu (Symbol Table)" ile karıştırmamalısınız. Sembol tabloları "parser" modülü tarafından kullanılmaktadır ve sembol tablolarında değişkenlerin türleri ve diğer özellikleri tutulmaktadır.

Örnek bir string tablosu şöyle gerçekleştirilebilir:

```
/* StringTable.h */

#ifndef STRINGTABLE_H_
#define STRINGTABLE_H_

#include <stddef.h>

/* Symbolic Constants */

#define TABLE_SIZE    1000

/* Type Declarations */

typedef struct tagSTRNODE {
    char *str;
    size_t size;
    struct tagSTRNODE *next;
} STRNODE;
```

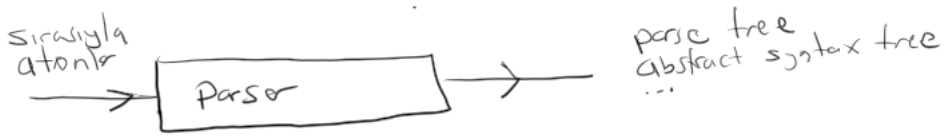
```c
/* Function Prototypes */

void InitStringTable(void);
char *LookupStrSize(const char *str, size_t size);
char *LookupStr(const char *str);
void DestroyStringTable(void);

#endif

/* StringTable.c */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "StringTable.h"

/* static Function prtotypes */

static size_t hashFunc(const char *str, size_t size);

/* Global Data Definitions */

STRNODE *g_hashTable[TABLE_SIZE];

/* Function Definitions */

static size_t hashFunc(const char *str, size_t size)
{
    unsigned long hash = 5381;
    int i;

    for (i = 0; i < size; ++i)
        hash = ((hash << 5) + hash) + str[i];

    return hash % TABLE_SIZE;
}

void InitStringTable(void)
{
    int i;

    for (i = 0; i < TABLE_SIZE; ++i)
        g_hashTable[i] = NULL;
}

char *LookupStrSize(const char *str, size_t size)
{
    size_t hash;
    STRNODE *node;

    hash = hashFunc(str, size);
    node = g_hashTable[hash];
    while (node != NULL) {
        if (size == node->size && !strncmp(str, node->str, size))
            return printf("bulundu\n"), node->str;
        node = node->next;
    }
    if ((node = (STRNODE *)malloc(sizeof(STRNODE))) == NULL)
        return NULL;
    if ((node->str = (char *)malloc(size + 1)) == NULL) {
        free(node);
        return NULL;
    }
    strncpy(node->str, str, size + 1);
    node->size = size;

    node->next = g_hashTable[hash];
    g_hashTable[hash] = node;
```

```
    return node->str;
}

char *LookupStr(const char *str)
{
    char *end = str;

    while (*end != '\0')
        ++end;

    return LookupStrSize(str, (size_t)(end - str));
}

void DestroyStringTable(void)
{
    int i;
    STRNODE *node, *tempNode;

    for (i = 0; i < TABLE_SIZE; ++i) {
        node = g_hashTable[i];
        while (node != NULL) {
            tempNode = node->next;
            free(node->str);
            free(node);
            node = tempNode;
        }
    }
}

#if 1

int main(void)
{
    char text[1024];
    char *str;

    InitStringTable();

    for (;;) {
        printf("Bir yazi giriniz:");
        gets(text);
        if (!strcmp(text, "quit"))
            break;
        if ((str = LookupStr(text)) == NULL) {
            fprintf(stderr, "cannot lookup string!..\n");
            exit(EXIT_FAILURE);
        }
        printf("%s (%x)\n", str, str);
    }

    DestroyStringTable();

    return 0;
}

#endif
```

3.7. Parse İşlemleri (Parsing)

“Parsing” İngilizce “ayrıştırma”, “gramer olarak çözümleme” anlamına gelen bir sözcüktür. Bilgisayar bilimlerinde bir yazının (biçimsel dil terminolojisinde bir string’in) verilen bir gramere (sentaksa) uygunluğunun denetlenmesi ve bunun bir veri yapısı biçiminde ifade edilmesi sürecine “parsing” denilmektedir. Bu işi yapan dil işlemcileri (language processors) de “parser” olarak isimlendirilir. Parser modülünün giridisi atomlardır. (Örneğin bu girdi “lexical analiz” aşamasında gerçekleştirilen GetNextToken gibi bir fonksiyonla ya da flex’in oluşturduğu yylex fonksiyonuyla elde edilebilir.) Parser modülünün çıktısı ise ilgili girdinin gramere göre çözümlenerek bir veri yapısı haline getirilmiş biçimidir. Bu çıktı da genellikle

"parse ağacı (parse tree)" ya da "soyut sentaks ağacı (abstract syntax tree)" biçiminde isimlendirilir. Tabii derleyicilerin ve yorumlayıcıların parser modülleri grameri temsil eden başka bir veri yapısı da üretebilir.



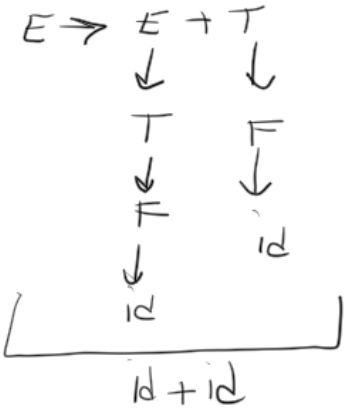
Şüphesiz "parse" işlemi için öncelikle elimizde biçimsel (resmi) bir gramerin (formal grammar) üretici bir gramer (generative grammar) biçiminde ifade ifade edilmesi gerekir. Dilin üretici grameri (sentaksı) matematiksel bir biçimde ya da BNF notasyonuyla betimlenebilmektedir. Biz burada üretici grameri bazen matematiksel gösterimle bazen de BNF notasyonuyla ifade edeceğiz. Örneğin aşağıda bir üretici gramerin matematiksel olarak gösterimini görüyorsunuz (örnek "Compiler Principles and Techniques" isimli kitabından alınmıştır):

$$
\begin{array}{lcl}
E & \rightarrow & E + T \mid T \\
T & \rightarrow & T * F \mid F \\
F & \rightarrow & (E) \mid \textbf{id}
\end{array}
$$

Burada son semboller (terminal symbols) ya da atomlar "id", "+", "*", "(" ve ")" karakterlerinden oluşmaktadır. E, T ve F ise ara sembollerdir. Matematiksel gösterimde (ve BNF notasyonunda) '|' karakterinin "veya" anlamına gelen bir meta karakter olduğunu anımsayınız. Aynı üretici gramer BNF notasyonu ile aşağıdaki gibi ifade edilebilir.

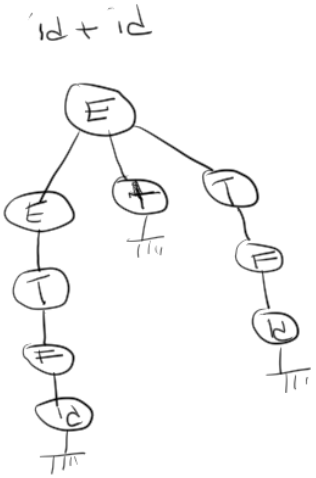
E: E '+' T
   T
T: T '*' F
   F
F: '(' E ')'
   'id'

Örneğin bu gramere göre üretilen bir çıktı (string) şöyle olabilir:

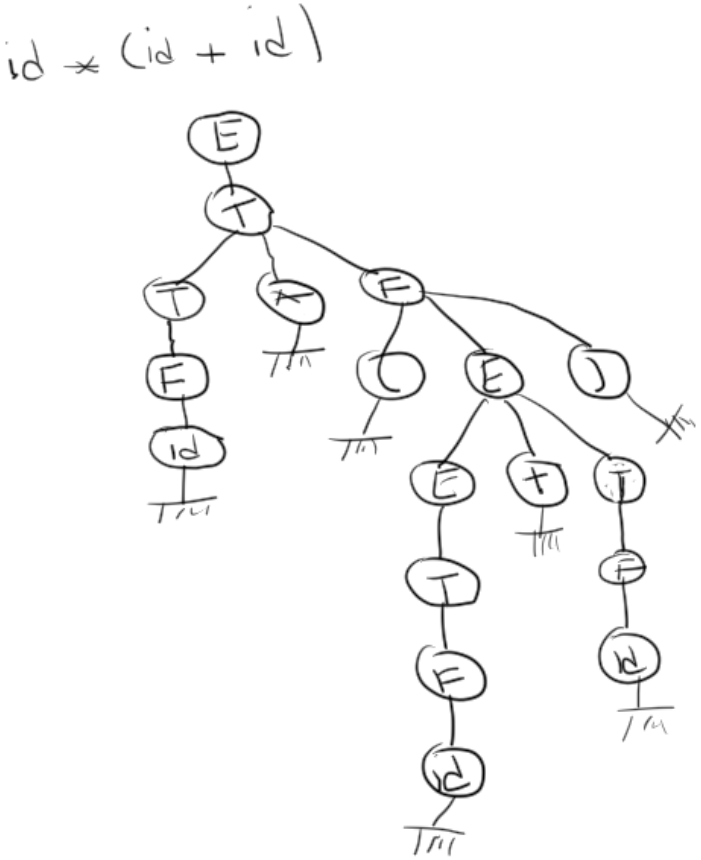


Bu gramerin örneğin id * (id + id) gibi bir string de üretebileceğine dikkat ediniz.

Parse ağacı ("Somut Sentaks Ağacı ("Concrete Sytax Tree") de denilmektedir) bir string'in üretici gramere göre açılımını gösteren bir veri yapısıdır. Parse ağacında ara semboller ve son semboller (atomların) birer düğüm olarak ifade edilirler. Parse ağacının yapraklarında (leaves) son semboller (atomlar) bulunur. Örneğin yukarıdaki gramerde "id + id" yazısı için oluşturulabilecek parse ağacı şöyledir:

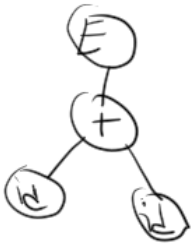


Şimdi de “id * (id + id)” string’i için parse ağacını oluşturalım:

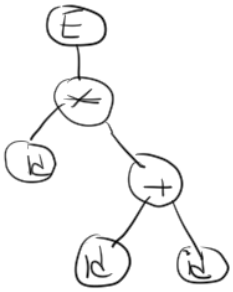


Bir string için tek bir “parse ağacı” oluşturulabileceğine dikkat ediniz.

Soyut sentaks ağacı (abstract syntax tree) gereksiz kural geçişlerinin elimine edildiği, parse ağacının daha sade hale getirilmiş bir biçimidir. Soyut sentaks ağacında son semboller yapraklarda bulunmak zorunda değildir. Bundan dolayı soyut sentaks ağacı “parser” modülünün çıktısı olmaya daha adaydır. Ancak soyut sentaks ağacı kuralları belirlenmiş olan, her zaman aynı biçimde oluşturulan resmi bir ağaç değildir. Dolayısıyla aynı string için farklı programcılar farklı soyut sentaks ağaçları oluşturabilirler. Örneğin “id + id” string’i için aşağıdaki gibi bir soyut sentaks ağacı oluşturulabilir:

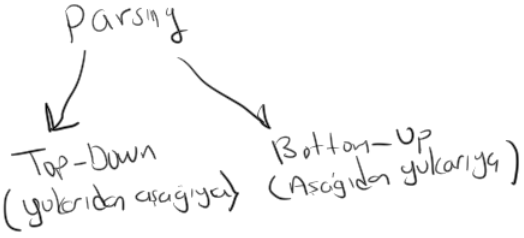


Görüldüğü gibi soyut sentaks ağacında gereksiz geçişler elimine edilmiştir. Şimdi de "id * (id + id)" string'i için soyut sentaks ağacı oluşturalım:



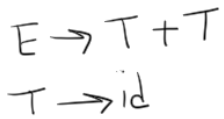
3.7.1. Parse İşlemleri İçin Teorik Temel

Parse işlemleri atomların sırasıyla elde edilip incelenmesi yoluyla yapılmaktadır. Tabii bu süreç için pek çok teknik oluşturulmuştur. Genel olarak parse işlemi "yukarıdan aşağıya (top down)" ya da "aşağıdan yukarayıa (bottom up)" denilen iki temel teknikle yapılabilmektedir.

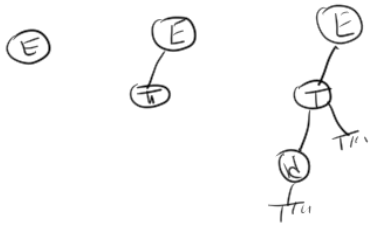


3.7.1.1. Yukarıdan Aşağıya Parse (Top-Down Parse) İşlemi

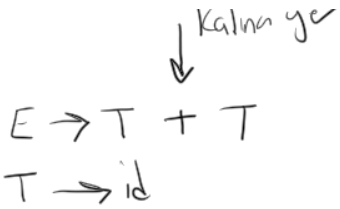
Yukarıdan aşağıya parse işleminde string'te sıradaki atom alınır ve bu atom bulunana kadar üretici gramerde aşağıya doğru inilir. Sonra diğer atom alınır bu atom da bulunana kadar kalının yerden aşağıda doğru inilir. Örneğin aşağıdaki gibi basit bir üretici gramer söz konusu olsun:



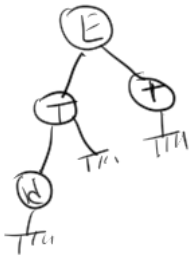
Şimdi biz "id + id" biçiminde bir string'i yukarıdan aşağıya doğru parse etmek isteyelim. Burada önce ilk atom olan id'nin yerini bulmak gerekir. Bunun için gramerin tepesinden sırasıyla id bulunana kadar aşağıya inilir (zaten "yukarıdan aşağıya (top-down) parse" ismi buradan gelmektedir):



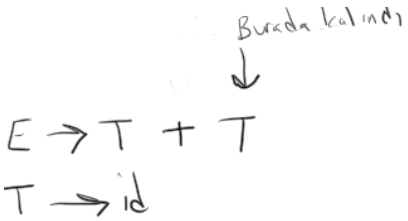
Pekiyi ilk “id” bulunduktan sonra ağaçta kalınan yer neresidir? T’den başka gidilecek yer olmadığına göre kalınan yer E’de T’den sonraki yerdir.

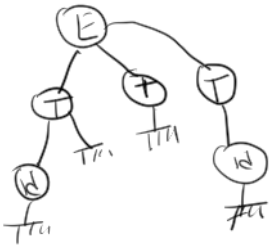


Şimdi ikinci atom çekilir. Bu atom ‘+’ atomudur. Bu atom da kalınan yerden itibaren aşağıya doğru bulunmaya çalışılır ki, zaten birinci kuralda bulunacaktır:



Şimdi sıradaki atom yine elde edilir. Kalınan yer E’deki ‘+’dan sonraki yerdir:





Yukarıdan aşağıya parse işlemini şöyle de anlatabiliriz: Atomlar sırasıyla elde edilir, üretici gramerin tepesinden aşağıya inilerek bunların yeri bulunur ve ağaç oluşturulur. Sonraki atomlar hep kalınan yerden itibaren aranmaktadır. Bu yöntemde parse ağacının oluşturulması “depth first” dolaşım sistemine uygundur. (Buna İngilizce “leftmost derivation” da denilmektedir.) Aşağıda Aho’nun “Compiler Design and Implementation” kitabından alınmış bir “yukarıdan aşağıya parse” örneğini görüyorsunuz:

$$
\begin{array}{lcl}
E & \rightarrow & T\ E' \\
E' & \rightarrow & +\ T\ E' \mid \epsilon \\
T & \rightarrow & F\ T' \\
T' & \rightarrow & *\ F\ T' \mid \epsilon \\
F & \rightarrow & (\ E\ ) \mid \mathbf{id}
\end{array}
$$

Burada epsilon boş küme anlamına gelmektedir. Yani epsilon görülğünde artık o ara sembol sonlandırılmış olur. Şimdi bu üretici gramer için "id + id * id" string'inin parse ağacını oluşturalım:

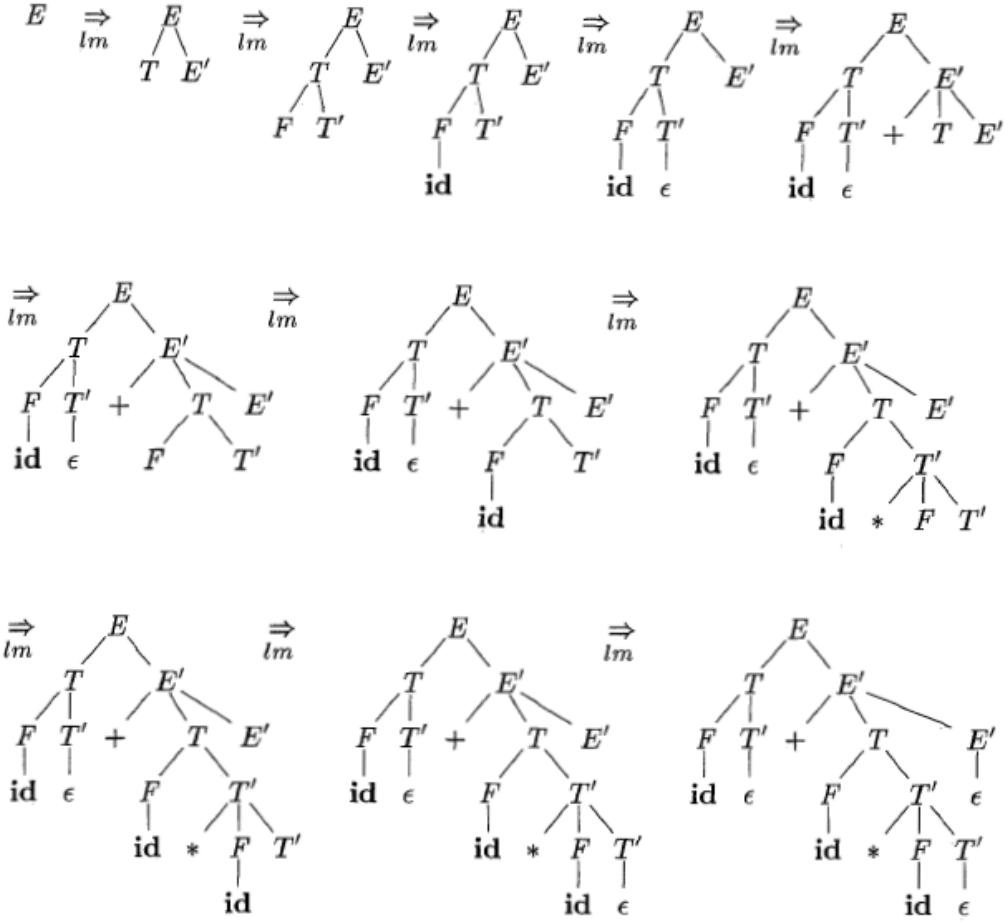


Figure 4.12: Top-down parse for $\mathbf{id} + \mathbf{id} * \mathbf{id}$

Pekiyi böyle bir yukarıdan aşağıya parse işlemini yapan genel bir kod nasıl yazılabilir? Aslında yukarıdan aşağıya parse işlemi basit bir biçimde şöyle genelleştirilebilir:

- Tüm ara semboller (non-terminal symbols) için birer fonksiyon yazılır. Örneğin yukarıdaki gramerde T, E', F, T' için birer fonksiyon yazılır. (İstenirse son semboller için de fonksiyonlar yazılabilir.)

- Ara sembollere ilişkin kurallardaki ara sembollere ilişkin fonksiyonlar bir diziye yerleştirilir.

- Çözümlenecek string'ten sıradaki atom elde edilir. Bu atom kök ara sembolden başlayarak fonksiyon çağrılarıyla bulunmaya çalışılır. Atom bulunduğunda string'teki sonraki atomdan devam edilir. Zaten fonksiyon çağrıları kalınan yerden devamı kendiliğinden sağlayacaktır. Aho'nun "Compiler Design and Implementation" kitabında bu işlemin semboli kodları (pseudo codes) şöyle verilmiştir:

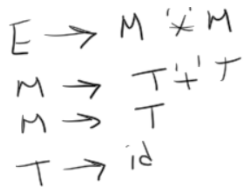
4.4.1 Recursive-Descent Parsing

```
    void A() {
1)      Choose an A-production, A → X1X2 ··· Xk;
2)      for ( i = 1 to k ) {
3)          if ( Xi is a nonterminal )
4)              call procedure Xi();
5)          else if ( Xi equals the current input symbol a )
6)              advance the input to the next symbol;
7)          else /* an error has occurred */;
        }
    }
```

Figure 4.13: A typical procedure for a nonterminal in a top-down parser

Bu algoritmayla oluşturulan parser'a "recursive-descent parser" da denilmektedir. Bu algoritmanın basit bir gramere uygulanmış hali kurs notlarında "Src\015-RecursiveDescentParser" klasöründe bulunmaktadır.

Pekiyi bir kuralın sol tarafında aynı ara sembolden birden fazla kez bulunduğu durumda (yani seçeneklerin söz konusu olduğu durumda) hangi ara sembole gitmek gerekir? Örneğin:



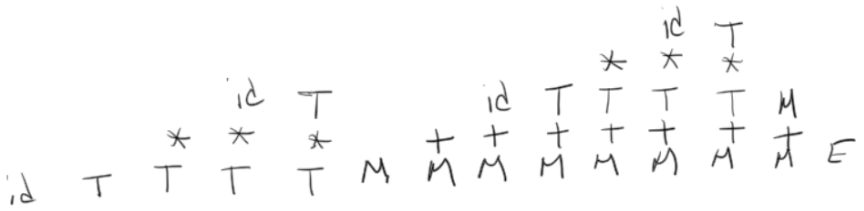
Burada M ara sembolü iki seçeneklidir. Yukarıdan aşağıya parse işleminde önce bunlardan biri denenip gramerle uyuşum olmadığında diğerini denemek akla uygun gelebilir. Bu stratejiye "parsing" terminolojisinde "geriye dönme (backtracking)" denilmektedir. Şüphesiz geriye dönme performansı azaltıcı bir unsur oluşturur. Seçenekli durumlarda geriye dönme sonraki birkaç atoma bakılarak elimine edilebilmekteedir. Böylece daha işin başında doğru seçeneğe yönelim sağlanmaktadır. İşte bu tür parser'lara "ön kestirimli (predicative) yukarıdan aşağıya parser"lar denilmektedir. Ön kestirimli yukarıdan aşağıya parser'lar sıklıkla "LL(k)" biçiminde gösterilirler. Burada k kaç ileriye bakılacağını belirtmektedir. Bağlam bağımsız gramerlerin çoğu yukarıdan aşağıya yalnızca bir sonraki atoma bakılarak yani LL(1) parser'larla parse edilebilmektedir. (Özel olarak k tane sonraki atoma bakılarak parse edilebilen gramerlere de LL(k) gramer denilmektedir. LL(k) teriminde ilk L "Left To Right" ikinci L ise "Left Most Derivation" sözcüklerinden gelmektedir.)

3.7.1.2. Aşağıdan Yukarıya Parse (Bottom-Up Parse) İşlemi

Aşağıdan yukarıya parse işleminde tam ters bir yol izlenmektedir. Sıradaki atom alınır, gramerin en aşağısından başlanarak üste doğru çıkılıp gramere uygunluk denetlenir. Bu denetleme sırasında bir stack'e gereksinim duyulduğu için bu yönteme "Shift-Reduce" Parsing de denilmektedir. (Buradaki Shift "push" olarak, Ruduce ise "pop" olarak düşünülebilir. (Tipik olarak aşağıdan yukarıya parse işlemi şu sistematikle yapılmaktadır: Sıradaki atom alınır, aşağıdan başlanarak gramere uydurulmaya çalışılır. Eğer atom gramere uymazsa stack'a push (shift) edilir. Sonra yeni atom alınır. Bu atomda da aynı işlemler yapılır. Ancak stack'e push işleminden sonra stack'teki durum eğer bir indirgeme (reduce) gerektiriyorsa (yani bir ara sembolle ifade edilebiliyorsa) stack'ten indirgenecek semboller alınır ve indirgenmiş yeni ara sembol stack'e atılır. Bu biçimde ilerlenerek kök sembole ulaşılmaya çalışılır. Örneğin aşağıdaki gibi bir gramer olsun:

```
E -> M '+' M
M -> T '*' T
T -> id
```

Bu gramer için “id * id + id * id” string’i verilmiş olsun, “shift” ve “reduce” işlemleri aşağıdaki gibi yapılmaktadır:



Aşağıdan yukarıya parse işlemi de bir kuralda alternatifler olduğunda “geri dönmeyi (backtracking)” gerektirebilmektedir. Geriye dönme işlemini ortadan kaldırabilmek için yine burada da sonraki atomun ele alınması gerekebilir. Aşağıdan yukarıyya parse işleminin atomların soldan sağa alınarak yapılmasından dolayı bu tür parser’lara “LR Parser” da denilmektedir. (Buradaki L harfi “Left To Right” sözcüğünden, R harfi ise “Rightmost derivation” sözcüğünden gelmektedir. LR parser’lar alternatif durumlarında ilerideki kaç elemana bakılacağına göre LR(k) biçiminde isimlendirilmektedir. Pek çok bağlam bağımsız dil LR(1) aşağıdan yukarıya parser’larla ayrıştırılabilmektedir.

LR parser’ların özel bir türüne LALR parser’lar denilmektedir. Örneğin Bison LALR(1) tarzı parser kullanmaktadır. Bu tür parser’lar için teorik bilgi için önerilen kaynaklara başvurabilirsiniz.

3.7.2. Parse İşlemininin Yapılmasına Ön Ayak Olan Araçlar

Nasıl flex (eski ismiyle lex) atomlarına ayırma işinde bize yardımcı oluyorsa parse işleminde de çeşitli yardımcı araçlardan faydalanılabilinmektedir. Genel olarak bu araçlara “parser generator” denilmektedir. Bu araçlar genellikle bizden dilin gramerini BNF veya türevleriyle alırlar ve parse işlemini yapan kodu üretirler. Bu kod derlenerek amaç doğrultusunda kullanılabilmektedir. Bu araçlardan en eskisi ve ünlüsü “yacc (yet another compiler compiler)” isimli araçtır. GNU projesi kapsamında “yacc” aracının “bison” isimli daha modern bir versiyonu oluşturulmuştur. Bison büyük ölçüde “yacc” ile uyumludur. Bunların dışında yine “ANTLR” isimli araç parse işlemleri için oldukça yoğun kullanılmaktadır. ANTLR’nin scanner ve parser üreticileri “Java” ve “C#” dillerine ilişkin kod üretmektedir. Maalesef ANTLR'nin C ve C++ versiyonları henüz etkin biçimde oluşturulamamıştır. Ancak çalışmalar devam etmektedir. Bunların dışında da daha az tercih edilen çeşitli “parser generator” araçları mevcuttur.

3.8. Bison Aracının Kullanımı

Bison’un kurulumu oldukça basittir. https://www.gnu.org/software/bison/ sitesinden Bison indirilebilir. Fakat zeten Linux sistemlerinde Bison tıpkı Flex gibi temel bir araçtır ve default kurulumlarda zaten yüklenmektedir. Bison’un Windows versiyonu için yine seçenek vardır. Birincisi GNU’nun kendi yazılımıdır. Bu http://gnuwin32.sourceforge.net/packages/bison.htm adresinden indirilebilir. İkincisi de WinFlexBison aracıdır. Bu araçta bison programı “win_bison.exe” ismiyle bulunmaktadır.

Bison flex ile beraber kullanılabilecek biçimde tasarlanmıştır. Bison parse işlemini yaparken default olarak atomları yylex isimli fonksiyonu çağırarak elde eder.

Bir bison dosyası tıpkı Flex’te olduğu gibi üç bölüme ayrılmaktadır:

```
%{
    Giriş (Prologue)
%}

Bison Bildirimleri (Bison Declarations)

%%
```

```
Gramer Kuralları (Grammar Rules)

%%

Kullanıcı Kodları (User Codes / Epilogue)
```

Bison dosyalarının uzantıları geleneksel olarak ".y" biçimindedir. Tıpkı Flex'te olduğu gibi Bison dosyaları da %% karakterleri ile üç bölüme ayrılmıştır. Dosyanın "Giriş" bölümünde tıpkı Flex'te olduğu gibi %{ .. %} atomlarının arasına C'ce anlamlı olabilecek bildirimler ve tanımlamalar yerleştirilir. Bu bölüme yerleştirilen kodlar yine Bison'un ürettiği C kaynak dosyasının başına yerleştirilmektedir. Yine ilk %% atomunun yukarısındaki bölüme %{ ... %} bloğunun dışına Bison'ca anlamlı % karakteri ile başlayan bildirimler yerleştirilebilmektedir. Bu bildirimlerin bazıları ilerleyen kısımlara ele alınacaktır. Aslında ilk %% karakterlerinin yukarısındaki "Giriş" ve "Bison Bildirimleri" kısımları birden fazla kez bu bölüm içerisinde bulunabilir.

İki %% atomlarının arasına BNF notasyonuna uygun biçimde gramer kuralları yerleştirilir. Gramerdeki ara sembollerin seçenekleri '|' karakteriyle belirtilmektedir. Kurallar birden fazla satıra yayılmış olarak yazılabilirler. Ancak seçeneksiz kuralların tek bir satıra yazılması ve seçenekli kuralların her seçeneğinin ayrı bir satıra yazılması çok kullanılan bir yazım biçimidir. Bison'da kuralların son sembolleri (ya da atomları) tek tırnak ya da çift tırnak ile belirtilebilmektedir. Dosyanın "Kullanıcı Kodları" bölümüne istenildiği kadar C kodu yerleştirilebilir. Örneğin main fonksiyonu tipik olarak burada tanımlanmalıdır. Bison'da Parse işlemini yyparse isimli fonksiyon yapmaktadır. Dolayısıyla main fonksiyonu içerisinde bu fonksiyonun çağrılması gerekir.

Tipik olarak bir bison programı şöyle işleme sokulmaktadır:

```
bison <bison dosyasının yol ifadesi>
```

Örneğin:

```
bison sample.y
```

Ya da Windows sistemlerinde win_bison kullanılıyorsa derleme işlemi şöyle de yapılabilir:

```
win_bison <bison dosyasının yol ifadesi>
```

Örneğin:

```
win_bison sample.y
```

Bison dosyanın ismi "x.y" olmak üzere Bison default durumda çıktı olarak "x.tab.c" isimli bir C kaynak dosyası üretir. Ancak "-o" komut satırı seçeneği ile biz dosyanın ismini de istediğimiz belirleyebiliriz. Örneğin:

```
bison –o sample.c sample.y
```

Yukarıda da belirttiğimiz gibi Bison gramer kontrolü için atomları yylex isimli fonksiyonu çağırarak elde etmektedir. Bu durumda Bison dosyasını derlerken ve bağlarken bizim bir biçimde yylex fonksiyonunu bulundurmamız gerekir. İşte yylex fonksiyonunu biz manuel olarak yazılabiliriz ya da genellikle tercih edildiği gibi bunu Flex'e yazdırabiliriz. Bu durumda bizim Flex tarafından üretilen kaynak dosyayla Bison tarafından üretilen kaynak dosyayı birlikte derleyip bağlamamız (link etmemiz) gerekir. Bison'un çağırdığı yylex fonksiyonunun protoipi aşağıdaki gibi olmalıdır:

```
int yylex(void);
```

Bison tarafından üretilen C kodu bir gramer hatası ile karşılaşıldığında yyerror isimli bir fonksiyonu çağıracak biçimde oluştuşturulmuştur. Bu fonksiyonun tanımlanması da Bison programcısının sorumluluğundadır. yyerror fonksiyonunun prorotipi şöyledir:

```
void yyerror(const char *s);
```

Özetle programcının Bison tarafından üretilen kaynak dosyayı derleyip bağlayabilmesi için yylex ve yyerror isimli iki fonksiyonu bulundurması gerekmektedir. Burada bir noktaya daha dikkatinizi çekmek istiyoruz: Bison'un ürettiği kod tarafından çağrılan yylex ve yyerror dosyalarının prototipleri Bison'un ürettiği kaynak dosyalarda bulunmamaktadır. Bu nedenle bu fonksiyonların prototip bildirimlerinin Bison dosyasının yukarısındaki "Giriş" bölümünde programcı tarafından yapılması gerekir.

Şimdi Bison dosyasında gramerin nasıl belirtildiği üzerinde duralım. Bison dosyasının "Gramer Kuralları" bölümüne yerleştirilecek BNF gramerinde uygulanacak kurallar şunlardır:

1) Kuralların sol tarafı ara sembollerden oluşur. Ara sembol isimlerinden sonra bir tane ':' atomunun bulunması gerekir. Kuralların sağ tarafı ise ara semboller ve son sembollerden (atomlardan) oluşturulabilir. Örneğin:

```
A:  B '+' C;
```

Bu kuralda açılmak istenen ara sembol A'dır. Burada B ve C ara semboller, '+' ise son semboldür.

2) Kurallardaki boşluk karakterlerinin bir önemi yoktur. Örneğin kuralın sağ tarafında istenildiği kadar boşluk karakteri bırakılabilir. (Boş satırların da boşluk karakterleri ile oluşturulduğuna dikkat ediniz.) Örneğin:

```
A:
    B    '+'    B;
```

3) Kuralların sonu noktalı virgül meta karakteri ile bitirilir.

4) Eğer bir kuralın sol tarafındaki ara sembol için seçenek varsa bunlar farklı kurallar biçiminde ya da '|' meta karakterleri kullanılarak tek bir kural biçiminde de oluşturulabilir. Örneğin:

```
A: B '+' B;
A: B '-' B;
```

Bu iki kural şöyle de yazılabilirdi:

```
A:  B '+' B
    | B '-' B;
```

5) Bison'da son semboller (atomlar) üç biçimde belirtilirler:

a) Tek tırnak içerisinde. Tek tırnağın içerisinde tek bir karakter bulunmak zorundadır. Bu durumda bu tek tırnağın içerisindeki karakter atomun türü olarak yylex fonksiyonundan elde edilen değer olur. Örneğin Bison '+' gibi bir sembol gördüğünde bu '+' sembolünün ASCII tablosundaki sıra numarası 97 olduğu için bununla yylex fonksiyonun geri döndürdüğü 97 değerini eşleştirecektir.

b) Bir karakterden uzun olan atomlar %token direktifi ile dosyanın "Bison Bildirimleri" kısmında belirtilmelidir. Bu durumda Bison bu direktifle bildirilen atomun türüne ilişkin değeri kendisi belirler. Tabii yylex fonksiyonunun da bu atom için aynı değerle geri döndürülmesi gerekir. Örneğin:

```
%token SQRT
...
Expression:
    SQRT '(' Expression ')';
```

Eğer istenirse atomun tür değeri de belirtilebilir. Örneğin:

```
%token SQRT   260
```

Bison %token direktifi ile bildirilen atomların değerlerini ürettiği “xxx.tab.h” (burada xxx Bison dosyasının ismini belirtiyor) isimli dosyada #define önişlemci komutuyla sembolik sabit biçiminde bildirmektedir. Böylece Flex dosyasında bu “xxx.tab.h” dosyası include edilerek yylex fonksiyonunun uygun atomlarda uygun değerlerle geri dönmesi sağlanabilir .

c) Atomlar okunabilirliği artırmak için iki tırnak içerisinde de belirtilebilirler. Ancak iki tırnak içerisinde belirtilen atomların yine %token direktifi ile aşağıdaki gibi bildirilmeleri gerekir:

```
%token SQRT “sqrt”
...
Expression:
    “sqrt” ‘(‘ Expression ‘)’
```

Bu biçimde istenirse atom için tür numarası da verilebilir:

```
%token SQRT 260 “sqrt”
```

Eğer son semboller tek tırnak içerisinde belirtilmemiş fakat %token direktifi ile belirtilmişe bunları büyük harflerle isimlendirmek bir gelenektir.

6) Bison’da kuralların sağ tarafına { ... } blokları içerisinde istenildiği kadar C kodları yerleştirilebilir. Bu kodlara Bison manüelinde “kuralın yapılacak işlemler kısmı (action)” denilmektedir. Örneğin:

```
A:
    B ‘+’ { printf(“test1\n”); } B { printf(“test\n”); };
```

Kurallardaki C kodları gramer çözümlenirken ilgili kurala uyuşum sağlandığında çalıştırılmaktadır. Örneğin yukarıdaki kuralda '+' atomu elde edildiğinde "test1" yazısı, B ara sembolüğ elde edildiğinde de "test2" yazısı ekrana yazdırılacaktır.

7) Bison’da ikinci %% karakterinden sonra yine programcının C kodları bulundurulabilir. (Tabii bu biçimde dosyanın sonunda tanımlanan fonksiyonlar Bison gramerinde çağrılacaksa bunların prototip bildirimlerinin "Giriş (Prologue)" bölümünde bildirilmesi gerekir.)

8) Bison’da parse işlemini yapan ana fonksiyon yyparse isimli fonksiyondur. Bu nedenle Bison uygulamasındaki main fonksiyonunun bir biçimde bu yyparse fonksiyonunu çağrılması gerekir.

3.8.1. Bison İle Flex’in Birlikte Kullanılması

Yukarıda da belirtildiği gibi Bison sıradaki atomu yylex fonksiyonunu çağırarak elde eder. yylex fonksiyonu da Flex tarafından sağlanabildiği için genellikle Bison aracı Flex ile birlikte kullanılmaktadır. Anımsanacağı gibi yylex fonksiyonunun geri dönüş değeri atomun türüdür. Bu durumda yylex’in geri döndürdüğü atom türleriyle Bison’daki türlerin sayısal bakımdan uyuşması gerekir. Atom değerleri Bison’da tek tırnak içerisinde verilmişse Flex’te de yylex fonksiyonu aynı değerle geri döndürülerek uyum sağlanabilir. Ancak Bison’da atomlar %token direktifi ile belirtilmişse ve atoma değer verilmemişse bu durumda Bison atomun tür değerini kendisi tespit etmektedir. İşte Bison tüm %token direktifleriyle belirtilen atomlara ilişkin sembolik sabit bildirimlerini istenirse bir başlık dosyasında toplayabilmektedir. Bunun için Bison “-d” seçeneğiyle çalıştırılmalıdır:

```
bison -d <bison dosyasının yol ifadesi>
```

Bu işlemden default durumda Bison dosyasının ismi “xxx.y” olmak üzere “xxx.tab.h” isimli bir dosya elde edilecektir. Örneğin:

```
bison -d sample.y
```

Burada ürün olarak “sample.tab.h” dosyası elde edilir. (Ayrıca bu örnekte “-o” seçeneği kullanılmadığı için Bison tarafından üretilecek C kaynak dosyasının “sample.tab.c biçiminde olacağına dikkat ediniz. Bu durumda Bison ile Flex birlikte kullanılacaksa önce Bison dosyası işleme sokulup “xxx.tab.h” dosyası elde edilmeli, bu dosya da Flex’te include edilerek yylex fonksiyonun aynı atom tür değerleriyle geri dönmesi sağlanmalıdır.

Anımsanacağı gibi yylex atomu bulduğunda onun tür koduyla geri dönmeden önce bulduğu atomun yazısını da yytext adresiyle belirtilen char türden diziye yerleştirmekteydi. Ancak Bison yylex fonksiyonu çağırdıktan sonra atomun türünü elde edince onun değerini yytext değişkeninden değil yylval isimli bir değişkenden almaktadır. yylval değişkeni YYSTYPE isimli bir türdendir. Bu tür default olarak int biçiminde typedef edilmiştir. Ancak ileride de görüleceği üzere bu türün atomdan atoma ve kuraldan kurala değişebilmesi nedeniyle bir birlik (union) olarak bildirilmesi gerekmektedir. Bunun için %union direktifi kullanılır. %union direktifi kullanıldığında yylval değişkeni de bu direktife belirtilen union türünden olacaktır. Programcının Flex tarafında atomu bulunca -eğer onun yalnızca türü değil değeri de önemliyle- atomun değerini de yylval içerisine yerleştirmesi gerekir. Çünkü yukarıda da belirttiğimiz gibi Bison onu yylval değişkeninden almaktadır.

%union direktifi kullanıldığında atomların türleri birbirlerinden farklı olabileceği için %token direktifinde de onların türleri açısal parantezler içerisinde belirtilir. Bu belirtme %union direktifindeki değişken isimleri kullanılarak yapılır. Ancak başka seçenekler de vardır. Örneğin:

```
%union {
    double val;
    const char *id;
};

%token NUMBER<val>
%token IDENTIFIER<id>
```

Burada dolaylı olarak NUMBER isimli atom türünün double türünden olduğu ve yylval isimli birliğin val elemanından alınacağı, IDENTIFIER isimli atom türünün de const char * türünden olduğu ve yylval birliğinin id isimli elemanından alınacağı belirtilmektedir.

3.8.2. Ara Sembollerin Türleri ve Değerleri

Bison’da yalnızca atomların değil her ara sembolün de bir değeri vardır. Atomların değerlerinin türleri %token direktifi ile açısal parantezler kullanılarak belirtiliyordu. Ara sembollerin türleri ise %type direktifiyle belirtilmektedir. Örneğin:

```
%union {
    double val;
    const char *id;
};

%token NUMBER<val>
%token IDENTIFIER<id>
%type Expression<val>
```

Burada Expression ara swembolü double türündendir ve yylval birliğinin val elemanından elde edilmektedir.

Bison'da ara sembollerin ya da atomların değerleri blok içerisindeki C kodlarında $n sembolüyle kullanılabilir. Buradaki n değeri kuralın sağ tarafındaki atom ya da ara sembolün 1’den başlayan pozisyon numarasıdır. Kuralın sol tarafındaki ara sembolün değeri ise $$ ile belirtilmektedir. Örneğin:

```
%union {
```

```
    double val;
    const char *id;
};

%token NUMBER<val>
%token IDENTIFIER<id>
%type AdditiveExpression<val>
%type MultiplicativeExpression<val>

...
AdditiveExpression:
    MultiplicativeExpression                                { $$ = $1; }
    | AdditiveExpression '+' MultiplicativeExpression       { $$ = $1 + $3; }
    | AdditiveExpression '-' MultiplicativeExpression       { $$ = $1 - $3; }
    ;
```

Yukarıda anlatılan bilgiler size oldukça soyut ve anlamsız gelebilir. Ancak bu bilgiler örnek bir Bison dosyasının oluşturulabilmesi için gereken en temel bilgilerdir. Yulkarıdaki açıklamalarda neyin ne anlama geldiği örneklerle yavaş yavaş anlaşılabilecektir.

3.8.3. yyparse Fonksiyonun Çalışma Biçimi

Bison'daki yyparse fonksiyonu tıpkı Flex'teki yylex fonksiyonu gibi birden fazla kez çağrılabilmektedir. yyparse her çağrıldığında kaynak dosyada kalınan yerden devam eder. Programcı da kaynak kodu ayrıştırma işlemini yyparse fonksiyonunu birden fazla kez çağırarak da yapabilir. yyparse bir sentaks hatası ile karşılaştığında yyerror fonksiyonunu çağırdıktan sonra kendisini sonlandırmaktadır. Bu durumda biz yyparse fonksiyonunu yeniden çağırarak işlemlerin devam etmesini sağlayabiliriz. (Aslında bir hata olduğunda yyerror fonksiyonun çağrılması da engellenebilmektedir. Bu konu ileride ele alınacaktır.)

3.8.4. Örnek Bir Komut Satırı Hesap Makinesi

Komut satırında her bir girişte bir ifadenin değerini hesaplayıp ekrana yazdıran bir uygulama yapmak isteyelim. Örneğin bu uygulama sayesinde:

```
3 * (2 + 7) * sqrt(100) + 2
```

gibi bir ifadenin sonucu ENTER tuşuna basıldığında aşağıda görüntülenecek olsun. "quit " gibi bir komut verildiğinde de programın sonlanmasını isteyelim.

Önce bu uygulama için atomlarına ayırma işlemini yapan bir Flex dosyası hazırlayalım. Aslında yukarıda da belirttiğimiz gibi Bison'dan elde edilen atom değerlerinin Flex'e verilmesi gerekir. Ancak biz bunun yapıldığını varsayarak Bison'dan elde edilecek dosyayı yukarıya include edelim. Basit bir hesap makinesi için Flex kodu şöyle olabilir:

```
/* calc.l */

%option noyywrap
%{
    #include <math.h>
    #include "calc.tab.h"
%}

%%
[0-9]+        { yylval.val = atof(yytext); return NUMBER; }
[+\-*()/\n]   { return *yytext; }
exit          { return EXIT; }
[ \t]
.             { return *yytext; }
```

```
%%
```

Şimdi bu Flex kodunu açıklayalım. Kodun başında iki dosyanın include edildiğini görüyorsunuz. Kod içerisinde atof fonksiyonu kullanıldığı için standart <math.h> dosyası da include edilmiştir. “calc.tab.h” Bison tarafından üretilecek olan dosyadır. Bu dosyanın içerisinde NUMBER sembolik sabitinin ve yylval değişkeninin extern bildirimleri bulunur. yylval bizim Bison dosyamıza göre bir birlik türündendir. Bu birlik türünün val isimli bir elemanı vardır. Bison yylex fonksiyonunu çağırdığında atomun türünün NUMBER olduğunu görünce onun değerin yylval nesnesinin val elemanından alacaktır. Örnek Flex dosyasında boşluk ve tab karakterlerinin geçildiğini görüyorsunuz. Boşluk ve tab karakterler elde edildiğinde yylex fonksiyonunun geri döndürülmediğine dikkat ediniz. (Dolayısıyla parser modülünün bu karakterlerin olduğundan bile haberi olmayacaktır.) ‘+’, ‘-‘, ‘*’, ‘(‘, ‘)’, ‘/’ ve ‘\n’ karakterleri ayrı birer atom olarak değerlendirilmiş ve bunların ASCII karşılıkları atom türü olarak belirlenmiştir. (Köşeli parantezlerin içerisindeki *, +, (,) karakterleri meta karakterler olarak değerlendirilmemektedir. Ancak ‘-‘ karakteri aralık belirttiğinden köşeli parantez içerisinde ters bölülenmiştir.)

Şimdi bu basit hesap makinesi için Bison dosyasını oluşturalım:

```
/* calc.y */

%{
    #include <stdio.h>
    #include <stdlib.h>

    int yylex(void);
    void yyerror(const char *str);

%}

%union {
    double val;
};

%token <val> NUMBER
%token EXIT
%type <val> Expression
%type <val> AdditiveExpression
%type <val> MultiplicativeExpression
%type <val> UnaryExpression
%type <val> PrimaryExpression

%%
Calc:
    Line Calc
    |
    ;

Line:
    Expression '\n'  { printf("%f\n", $1); } Line
    | '\n'
    | EXIT         { return 0; }
    |
    ;

Expression:
    AdditiveExpression     { $$ = $1; }
    ;

AdditiveExpression:
    MultiplicativeExpression   { $$ = $1; }
    | AdditiveExpression '+' MultiplicativeExpression {$$ = $1 + $3; }
    | AdditiveExpression '-' MultiplicativeExpression {$$ = $1 - $3; }
    ;
```

```
MultiplicativeExpression:
    UnaryExpression                                 { $$ = $1; }
    | MultiplicativeExpression '*' UnaryExpression  {$$ = $1 * $3; }
    | MultiplicativeExpression '/' UnaryExpression  {$$ = $1 / $3; }
    ;

UnaryExpression:
    PrimaryExpression      { $$ = $1; }
    | '-' UnaryExpression  { $$ = -$2; }
    | '+' UnaryExpression  { $$ = +$2; }
    ;

PrimaryExpression:
    NUMBER     { $$ = $1; }
    | '(' Expression ')'       { $$ = $2; }
    ;

%%

int main(void)
{
    yyparse();

    return 0;
}

void yyerror(const char *str)
{
    fprintf(stderr, "syntax error!\n");
}
```

Kodun “Giriş (Prologue)” kısmında gerekli C başlık dosyalarının include edildiğini ve ayrıca yylex ve yyerror fonksiyonlarının prototip bildirimlerinin yapıldığını görüyorsunuz:

```
%{
    #include <stdio.h>
    #include <stdlib.h>

    int yylex(void);
    void yyerror(const char *str);
%}
```

Daha önceden de belirttiğimiz gibi Bison yylex ve yyeror fonksiyonlarını kendi ürettiği kodda kullanmaktadır ancak bunların prototip bildirimlerini kendisi yapmamaktadır. Bu nedenle bizim dosyanın “Giriş” kısmında bu bildirimleri yapmamız gerekir.

Dosyanın “Giriş” kısmından sonra “Bison Bildirimleri” kısmında aşağıdaki bildirimler yapılmıştır:

```
%union {
    double val;
};

%token <val> NUMBER
%token EXIT

%type <val> Expression
%type <val> AdditiveExpression
%type <val> MultiplicativeExpression
%type <val> UnaryExpression
%type <val> PrimaryExpression
```

Buradaki %union bildirimi yylval isimli nesnenin türünü belirtmektedir. Eğer bu bildirim yapılmazsa yylval default olarak int türden kabul edilir. %union bildirimi yylval nesnesinin burada belirtilen C tarzı union

türünden olduğunu belirtir. yylval aslında Bison tarafından bizim aracılığımızla kullanılmaktadır. Biz Bison kodundaki kurallarda $n ifadesiyle bunları kullanırız. Benzer biçimde Bison'da her ara sembolün de bir türü vardır. Bu tür de yine bizim tarafımızdan $n ifadeleriyle kullanılmıştır. İşte %type direktifi bu ara sembollerin türünü belirtmektedir. Ara semboller de yine yylval nesnesi yoluyla bize verilmektedir.

%token ve %type direktiflerindeki tür açısal parantezler içerisinde dolaylı olarak belirtilmektedir. Eğer bu direktiflerde açısal parantezler kullanılmazsa default tür int olarak kabul edilir. Açısal parantezlerin içerisinde union bildirimindeki eleman isimleri yazılır. Bu durumda bu elemanın türü (örneğimizde val) o atomun ya da ara sembolün türü olur.

Hesap mekinesi uygulamasında "Kurallar" bölünmü aşağıdaki gibi oluşturulmuştur:

```
Calc:
    Line Calc
    |
    ;

Line:
    Expression '\n'  { printf("%f\n", $1); } Line
    | '\n'
    | EXIT         { return 0; }
    |
    ;

Expression:
    AdditiveExpression    { $$ = $1; }
    ;

AdditiveExpression:
    MultiplicativeExpression { $$ = $1; }
    | AdditiveExpression '+' MultiplicativeExpression { $$ = $1 + $3; }
    | AdditiveExpression '-' MultiplicativeExpression { $$ = $1 - $3; }
    ;

MultiplicativeExpression:
    UnaryExpression                                    { $$ = $1; }
    | MultiplicativeExpression '*' UnaryExpression     { $$ = $1 * $3; }
    | MultiplicativeExpression '/' UnaryExpression {$$ = $1 / $3; }
    ;

UnaryExpression:
    PrimaryExpression     { $$ = $1; }
    | '-' UnaryExpression { $$ = -$2; }
    | '+' UnaryExpression { $$ = +$2; }
    ;

PrimaryExpression:
    NUMBER     { $$ = $1; }
    | '(' Expression ')'     { $$ = $2; }
    ;
```

Burada "Calc" kuralı "Line"lardan, "Line"lar da "Expression" ve \n' atomlarından oluşmaktadır. Expression kuralı klasik C'nin gramer kurallarına benzetilerek oluşturulmuştur. Biz burada kuralların "Yapılacak Eylem (Action)" kısımları üzerinde duracağız. Her kural belirlendiğinde Bison o noktada küme parantezleri içerisindeki C kodlarının çağrılmasını sağlar. (Tabii küme parantezleri aslında kuralların her yerine yerleştirilebilir. Yani onların en sağında bulunmak zorunda değildir.) Küme parantezleri içerisindeki C kodlarında $$ meta karakterleri o kuralın sol tarafındaki ara sembole atanacak değeri, $n meta karakterleri ise ara sembolün sağ tarafındaki ara sembol ya da atomun değerini belirtmektedir. Bison'un aşağıdan yukarıya

(Bottom-Up) bir parser algoritması uyguladığını anımsayınız. Bu durumda ifade içerisindeki atomlar birleştirile birleştirile yukarıya doğru çıkılacak ve en yukara Expression tek bir değer olarak elde edilecektir.

Şimdi yukarıdaki hesap makinesini biraz daha geliştirelim. Örneğin ona çeşitli fonksiyonlar (“built-in” fonksiyonlar) ekleyelim:

```
/* calc.l */

%option noyywrap
%{
    #include <math.h>
    #include "calc.tab.h"
%}

%%
[0-9]+         { yylval.val = atof(yytext); return NUMBER; }
[+\-*()/\n!%]  { return *yytext; }
exit           { return EXIT; }
sqrt           { return SQRT; }
pow            { return POW; }
[ \t]
.              { return *yytext; }
%%

/* calc.y */

%{
    #include <stdio.h>
    #include <stdlib.h>
    #include <math.h>

    int yylex(void);
    void yyerror(const char *str);
    double factorial(double n);

%}

%union {
    double val;
};

%token <val> NUMBER
%token SQRT
%token EXIT
%token POW
%type <val> Expression
%type <val> AdditiveExpression
%type <val> MultiplicativeExpression
%type <val> UnaryExpression
%type <val> PostfixExpression
%type <val> PrimaryExpression

%%
Calc:
    Line Calc
    |
    ;

Line:
    Expression '\n'  { printf("%f\n", $1); } Line
    | '\n'
    | EXIT         { return 0; }
    |
```

```
    ;

Expression:
    AdditiveExpression    { $$ = $1; }
    ;

AdditiveExpression:
    MultiplicativeExpression { $$ = $1; }
    | AdditiveExpression '+' MultiplicativeExpression {$$ = $1 + $3; }
    | AdditiveExpression '-' MultiplicativeExpression {$$ = $1 - $3; }
    ;

MultiplicativeExpression:
    UnaryExpression                                 { $$ = $1; }
    | MultiplicativeExpression '*' UnaryExpression {$$ = $1 * $3; }
    | MultiplicativeExpression '/' UnaryExpression {$$ = $1 / $3; }
    | MultiplicativeExpression '%' UnaryExpression {$$ = (int)$1 % (int)$3; }
    ;

UnaryExpression:
    PostfixExpression     { $$ = $1; }
    | '-' UnaryExpression { $$ = -$2; }
    | '+' UnaryExpression { $$ = +$2; }
    ;

PostfixExpression:
    PrimaryExpression        { $$ = $1;}
    | SQRT '(' Expression ')'{ $$ = sqrt($3); }
    | POW  '(' Expression ',' Expression ')'   { $$ = pow($3, $5); }
    | PostfixExpression '!'  { $$ = factorial($1); }
    ;

PrimaryExpression:
    NUMBER     { $$ = $1; }
    | '(' Expression ')'     { $$ = $2; }
    ;
%%

int main(void)
{
    yyparse();

    return 0;
}

double factorial(double n)
{
    int i;
    double fact = 1;

    for (i = 1; i <= n; ++i)
        fact *= i;

    return fact;
}

void yyerror(const char *str)
{
    fprintf(stderr, "syntax error!\n");
}
```

Burada gramere % operatörünün, ! (faktöryel) operatörünün ve sqrt ve pow fonksiyonlarının eklendiğine dikkat ediniz. Aşağıda hesap makinesinin örnek bir kullanımını görüyorsunuz:

```
sqrt(100) + 3! + pow(2, 3) * (2 + 3)
56.000000
```

Yuklarıdaki örneği Windows'ta şöyle derleyip bağlayabilirsiniz:

```
win_flex --wincompat calc.l
win_bison -d calc.y
cl /Fe:calc.exe lex.yy.c calc.tab.c
```

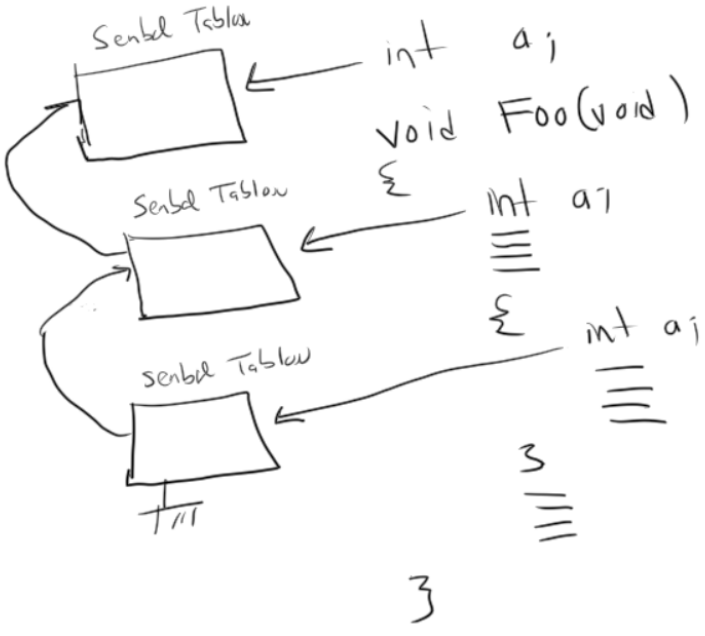
Linux sistemlerinde de aynı işlem şöyle yapılabilir:

```
flex  calc.l
bison -d calc.y
gcc -o calc lex.yy.c calc.tab.c
```

3.8.5. Derleyicilerin ve Yorumlayıcıların Kullandıkları Sembol Tabloları (Symbol Tables)

Derleyiciler ve yorumlayıcılar değişkenleri (identifiers) ilk gördüklerinde (dillerin çoğunda bunlar ilk kez bildirim yoluyla görülürler) bunları sembol tablosu denilen bir tabloya yerleştirmektedir. Sembol tablolarında değişkenlerin isimleri, türleri, faaliyet alanları, deposal özellikleri (örneğin static, extern gibi), değiştirilebilirliği (const olma durumu), kaynak koddaki bildirim yerleri gibi bilgiler tutulmaktadır. Yorumlayıcılarda değişkenlerin değerleri de yine sembol tablolarında tutulabilmektedir. Sembol tabloları genellikle hash tabloları biçiminde gerçekleştirilir. Değişkenlerin adı da anahtar olarak kullanılır. Tabii sembol tablolarının içeriği dilden dile yani o dile bağlı olarak değişebilmektedir. Sembol tabloları ile string tablolarını karıştırmamak gerekir. String tablolarından amaç bir değişkenin (identifier) isminin yalnızca bir kez depolanmasını sağlamaktır. Halbuki sembol tablolarında değişkenlere ilişkin tüm bilgiler tutulmaktadır.

C gibi iç içe faaliyet alanlarının bulunduğu dillerde farklı faaliyet alanlarında aynı isimli değişkenler bildirilebilmektedir. Değişken ismi sembol tablosuna anahtar yapıldığına göre ve anahtarın tek olması gerektiğine göre sembol tabloları faaliyet alanlarına göre nasıl gerçekleştirilecektir? İşte genellikle bu tür durumlarda her faaliyet alanı için ayrı bir sembol tablosu oluşturulma yoluna gidilmektedir. Sembol tabloları da içten dışarıya doğru bağlı liste biçiminde bir arada tutulur.



Böylece değişken önce aktif olan en içteki sembol tablosunda aranır, orada bulunursa yukarıya çıkılmaz. Eğer orada bulunamazsa yukarıya çıkılarak dış sembol tablolarında da aranır.

Pekiyi en basit bir sembol tablosu gerçekleştirimi nasıl olabilir? Eğer biz değişken isimlerini bir karakterle kısıtlarsak o karakterin karakter tablosundaki sıra numarasını bir hash değeri olarak kullanabiliriz. Örneğin tasarladığımz dilde yalnızca tek bir küçük harften değişken isimleri oluşturulabiliyor olsun. Değişkenlerin bilgilerinin ID_INFO isimli bir yapıyla temsil edildiğini düşünelim. Bu durumda sembol tablomuz şöyle olabilir:

```
struct ID_INFO symbolTable[26];
```

Tabii değişken isimleri uzun olabilecekse mecburen hash tablosu kullanmak gerekir. Şimdi daha önce yapmış olduğumuz hesap makinesine değişken de ekleyelim. Sembol tablosunda da değişken bilgisi olarak yalnızca değişkenin değeri bulunuyor olsun:

```
/* calc.l */

%option noyywrap
%{
    #include <math.h>
    #include "calc.tab.h"
%}

%%
[a-z]              { yylval.id = *yytext; return IDENTIFIER; }
[0-9]+             { yylval.val = atof(yytext); return NUMBER; }
[+\-*()/\n!%=]     { return *yytext; }
exit               { return EXIT; }
sqrt               { return SQRT; }
pow                { return POW; }
[ \t]
.                  { return *yytext; }
%%

/* calc.y */

%{
    #include <stdio.h>
    #include <stdlib.h>
    #include <math.h>

    /* Type Declarations */

    typedef struct tagID_INFO {
        double val;
    } ID_INFO;

    /* Function Prototypes */

    int yylex(void);
    void yyerror(const char *str);
    double factorial(double n);
    double get_identifier_value(char id);
    double set_identifier_value(char id, double val);


    /* Global Definitions */

    ID_INFO g_symbolTable[26];

%}

%union {
```

```
    double val;
    char id;
};

%token <val> NUMBER
%token <id> IDENTIFIER
%token SQRT
%token EXIT
%token POW
%type <val> AssignmentExpression
%type <val> Expression
%type <val> AdditiveExpression
%type <val> MultiplicativeExpression
%type <val> UnaryExpression
%type <val> PostfixExpression
%type <val> PrimaryExpression

%%
Calc:
    Line Calc
    |
    ;

Line:
    Expression '\n'  { printf("%f\n", $1); } Line
    | '\n'
    | EXIT         { return 0; }
    |
    ;

Expression:
    AssignmentExpression      { $$ = $1; }
    ;

AssignmentExpression:
    AdditiveExpression        { $$ = $1; }
    | IDENTIFIER '=' AdditiveExpression      { $$ = set_identifier_value($1, $3); }
    ;

AdditiveExpression:
    MultiplicativeExpression  { $$ = $1; }
    | AdditiveExpression '+' MultiplicativeExpression {$$ = $1 + $3; }
    | AdditiveExpression '-' MultiplicativeExpression {$$ = $1 - $3; }
    ;

MultiplicativeExpression:
    UnaryExpression                             { $$ = $1; }
    | MultiplicativeExpression '*' UnaryExpression {$$ = $1 * $3; }
    | MultiplicativeExpression '/' UnaryExpression {$$ = $1 / $3; }
    | MultiplicativeExpression '%' UnaryExpression {$$ = (int)$1 % (int)$3; }
    ;

UnaryExpression:
    PostfixExpression     { $$ = $1; }
    | '-' UnaryExpression  { $$ = -$2; }
    | '+' UnaryExpression  { $$ = +$2; }
    ;

PostfixExpression:
    PrimaryExpression         { $$ = $1;}
    | SQRT '(' Expression ')'  { $$ = sqrt($3); }
    | POW  '(' Expression ',' Expression ')'  { $$ = pow($3, $5); }
    | PostfixExpression '!'    { $$ = factorial($1); }
    ;

PrimaryExpression:
```

```
    NUMBER      { $$ = $1; }
    | IDENTIFIER { $$ = get_identifier_value($1); }
    | '(' Expression ')'       { $$ = $2; }
    ;
%%

int main(void)
{
    yyparse();

    return 0;
}

double factorial(double n)
{
    int i;
    double fact = 1;

    for (i = 1; i <= n; ++i)
        fact *= i;

    return fact;
}

double get_identifier_value(char id)
{
    return g_symbolTable[id].val;
}

double set_identifier_value(char id, double val)
{
    return g_symbolTable[id].val = val;
}

void yyerror(const char *str)
{
    fprintf(stderr, "syntax error!\n");
}
```

Windows'ta derleme işlemi aynı biçimde yapılabilir:

```
win_flex --wincompat calc.l
win_bison -d calc.y
cl /Fe:calc.exe lex.yy.c calc.tab.c
```

Linux ve Mac OS X sistemlerinde de şöyle yapılabilir:

```
flex  calc.l
bison -d calc.y
gcc -o calc lex.yy.c calc.tab.c
```

Tabii normal olan durum sembol tablolarının hash tablosu (hash table) yoluyla gerçekleştirilmesidir. Şimdi hash tablosu kullanarak değişken isimlerini tek karakter olmaktan çıkartalım:

```
/* stringtable.h */

#ifndef STRINGTABLE_H_
#define STRINGTABLE_H_

#include <stddef.h>

/* Symbolic Constants */

#define TABLE_SIZE     1000
```

```c
/* Type Declarations */

typedef struct tagSTRNODE {
    char *str;
    size_t size;
    struct tagSTRNODE *next;
} STRNODE;

/* Function Prototypes */

char *lookup_str_size(const char *str, size_t size);
char *lookup_str(const char *str);
void destroy_string_table(void);

#endif

/* stringtable.c */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "stringtable.h"

/* static Function prtotypes */

static size_t hashFunc(const char *str, size_t size);

/* Global Data Definitions */

STRNODE *g_hashTable[TABLE_SIZE];

/* Function Definitions */

static size_t hashFunc(const char *str, size_t size)
{
    unsigned long hash = 5381;
    int i;

    for (i = 0; i < size; ++i)
        hash = ((hash << 5) + hash) + str[i];

    return hash % TABLE_SIZE;
}

char *lookup_str_size(const char *str, size_t size)
{
    size_t hash;
    STRNODE *node;

    hash = hashFunc(str, size);
    node = g_hashTable[hash];
    while (node != NULL) {
        if (size == node->size && !strncmp(str, node->str, size))
            return node->str;
        node = node->next;
    }
    if ((node = (STRNODE *)malloc(sizeof(STRNODE))) == NULL)
        return NULL;
    if ((node->str = (char *)malloc(size + 1)) == NULL) {
        free(node);
        return NULL;
    }
    strncpy(node->str, str, size + 1);
    node->size = size;
```

```
    node->next = g_hashTable[hash];
    g_hashTable[hash] = node;

    return node->str;
}

char *lookup_str(const char *str)
{
    char *end = str;

    while (*end != '\0')
        ++end;

    return lookup_str_size(str, (size_t)(end - str));
}

void destroy_string_table(void)
{
    int i;
    STRNODE *node, *tempNode;

    for (i = 0; i < TABLE_SIZE; ++i) {
        node = g_hashTable[i];
        while (node != NULL) {
            tempNode = node->next;
            free(node->str);
            free(node);
            node = tempNode;
        }
    }
}


/* calc.l */

%option noyywrap
%{
    #include <math.h>
    #include "calc.tab.h"
    #include "stringtable.h"
%}

%%
exit                { return EXIT; }
sqrt                { return SQRT; }
pow                 { return POW; }
[a-z][a-z0-9]*      { yylval.id = lookup_str(yytext); return IDENTIFIER; }
[0-9]+              { yylval.val = atof(yytext); return NUMBER; }
[+\-*()/\n!%=]      { return *yytext; }
[ \t]
.                   { return *yytext; }
%%

/* calc.y */

%{
    #include <stdio.h>
    #include <stdlib.h>
    #include <math.h>
    #include "stringtable.h"

    /* Symbolic Constants */

    #define TABLE_SIZE     1000

    /* Type Declarations */
```

```
    typedef struct tagID_INFO {
        char *id;
        double val;
        struct tagID_INFO *next;
    } ID_INFO;

    /* Function Prototypes */

    int yylex(void);
    void yyerror(const char *str);
    double factorial(double n);
    double get_identifier_value(const char *id);
    double set_identifier_value(const char *id, double val);
    static size_t hashFunc(const char *str);
    void destroy_symbol_table(void);

    /* Global Definitions */

    ID_INFO *g_symbolTable[TABLE_SIZE];

%}

%union {
    double val;
    const char *id;
};

%token <val> NUMBER
%token <id> IDENTIFIER
%token SQRT
%token EXIT
%token POW
%type <val> AssignmentExpression
%type <val> Expression
%type <val> AdditiveExpression
%type <val> MultiplicativeExpression
%type <val> UnaryExpression
%type <val> PostfixExpression
%type <val> PrimaryExpression

%%
Calc:
    Line Calc
    |
    ;

Line:
    Expression '\n'  { printf("%f\n", $1); } Line
    | '\n'
    | EXIT         { return 0; }
    |
    ;

Expression:
    AssignmentExpression      { $$ = $1; }
    ;

AssignmentExpression:
    AdditiveExpression        { $$ = $1; }
    | IDENTIFIER '=' AdditiveExpression      { $$ = set_identifier_value($1, $3); }
    ;

AdditiveExpression:
    MultiplicativeExpression   { $$ = $1; }
    | AdditiveExpression '+' MultiplicativeExpression {$$ = $1 + $3; }
```

```
    | AdditiveExpression '-' MultiplicativeExpression {$$ = $1 - $3; }
    ;

MultiplicativeExpression:
    UnaryExpression                                 { $$ = $1; }
    | MultiplicativeExpression '*' UnaryExpression {$$ = $1 * $3; }
    | MultiplicativeExpression '/' UnaryExpression {$$ = $1 / $3; }
    | MultiplicativeExpression '%' UnaryExpression {$$ = (int)$1 % (int)$3; }
    ;

UnaryExpression:
    PostfixExpression      { $$ = $1; }
    | '-' UnaryExpression  { $$ = -$2; }
    | '+' UnaryExpression  { $$ = +$2; }
    ;

PostfixExpression:
    PrimaryExpression          { $$ = $1;}
    | SQRT '(' Expression ')'  { $$ = sqrt($3); }
    | POW  '(' Expression ',' Expression ')'  { $$ = pow($3, $5); }
    | PostfixExpression '!'    { $$ = factorial($1); }
    ;

PrimaryExpression:
    NUMBER     { $$ = $1; }
    | IDENTIFIER { $$ = get_identifier_value($1); }
    | '(' Expression ')'       { $$ = $2; }
    ;
%%

int main(void)
{
    while (yyparse() != 0)
        ;

    destroy_string_table();
    destroy_symbol_table();

    return 0;
}

double factorial(double n)
{
    int i;
    double fact = 1;

    for (i = 1; i <= n; ++i)
        fact *= i;

    return fact;
}

double get_identifier_value(const char *id)
{
    ID_INFO *idInfo;
    size_t hash;

    hash = hashFunc(id);
    idInfo = g_symbolTable[hash];

    while (idInfo != NULL) {
        if (id == idInfo->id)
            return idInfo->val;
        idInfo = idInfo->next;
    }
```

```
    fprintf(stderr, "warning: variable not found: '%s', zero value used instead\n", id);

    return 0;
}

double set_identifier_value(const char *id, double val)
{
    ID_INFO *idInfo;
    size_t hash;

    hash = hashFunc(id);
    idInfo = g_symbolTable[hash];

    while (idInfo != NULL) {
        if (id == idInfo->id) {
            idInfo->val = val;
            return val;
        }
        idInfo = idInfo->next;
    }

    if ((idInfo = (ID_INFO *)malloc(sizeof(ID_INFO))) == NULL) {
        fprintf(stderr, "cannot allocate memory!\n");
        exit(EXIT_FAILURE);
    }
    idInfo->id = id;
    idInfo->val = val;
    idInfo->next = g_symbolTable[hash];
    g_symbolTable[hash] = idInfo;

    return val;
}

void destroy_symbol_table(void)
{
    int i;
    ID_INFO *idInfo, *tempIdInfo;

    for (i = 0; i < TABLE_SIZE; ++i) {
        idInfo = g_symbolTable[i];
        while (idInfo != NULL) {
            tempIdInfo = idInfo->next;
            free(idInfo);
            idInfo= tempIdInfo;
        }
    }
}

static size_t hashFunc(const char *str)
{
    unsigned long hash = 5381;
    int i;

    for (i = 0; str[i] != '\0'; ++i)
        hash = ((hash << 5) + hash) + str[i];

    return hash % TABLE_SIZE;
}

void yyerror(const char *str)
{
    fprintf(stderr, "syntax error!\n");
}
```

Burada string tablosunun da kullanıldığına dikkat ediniz. Sembol (değişken ismi) lexical analiz modülü tarafından bulunduğunda string tablosuna yerleştirilmiştir. Parser modülü de string tablosundaki bu isme referans etmektedir.

Şimdi de yukarıdaki hesap makinesi örneğini basit bir yorumlayıcı haline getirelim. Yani Flex girişleri klavyeden (stdin) değil, bizim belirlediğimiz bir dosyadan alsın. Komutları peşi sıra çalıştırsın. print isimli isimli özel bir komut da ifadenin değerini ekrana yazdırsın. Bunun için "calc.y" dosyası içindeki main fonksiyonunu aşağıdaki gibi değiştirmemiz yeterli olacaktır:

```
int main(int argc, char *argv[])
{
    if (argc == 2) {
        if ((yyin = fopen(argv[1], "r")) == NULL) {
            fprintf(stderr, "cannot open file: %s\n", argv[1]);
            exit(EXIT_FAILURE);
        }
    }
    else if (argc > 2) {
        fprintf(stderr, "too many arguments!..\n");
        exit(EXIT_FAILURE);
    }
    while (yyparse() != 0)
        ;

    destroy_string_table();
    destroy_symbol_table();

    return 0;
}
```

Programda komut satırından girilen dosya fopen fonksiyonuyla açılmış ve dosya bilgi göstericisi yyin değişkenine atanmıştır. Böylece artık kaynak program stdin dosyasından değil bu dosyadan alınacaktır. Derleme işleminden sonra programı çalıştırırken işletilecek dosyayı komut satırı argümanı ile vermeyi unutmayınız. Örnek basit bir dosya şöyle olabilir:

```
/* test.calc */

x = 99 ;
y = 1;
print sqrt(x + y);
```

3.8.6. Bison'da Kurallar Arası Geçişler

Anımsanacağı gibi Bison aşağıdan yukarıya LALR(1) türü bir parser algortiması kullanmaktaydı. Aşağıdan yukarıya parse işlemi parse ağacının oluşturulmasını kolaylaştırmaktadır. Çünkü bu yöntemde önce alt düğümler sonra üst düğümler elde edilmektedir. Burada küçük bir örnekle Bison'un aşağıdan yukarıya parse işlemi sırasında kuraldan kurala nasıl geçtiği ele alınacaktır. Aşağıdaki gibi basit bir gramer oluşturmuş olalım:

```
/* test.y */

%{
    #include <stdio.h>

    /* Function Prototypes */

    int yylex(void);
    void yyerror(const char *str);

%}

%token NUMBER
%token IDENTIFIER
```

```
%type <val> AssignmentExpression
%type <val> Expression
%type <val> AdditiveExpression
%type <val> MultiplicativeExpression
%type <val> PrimaryExpression

%union {
    int val;
}

%%

Expression:
    AssignmentExpression  '\n' { printf("AssignmentExpression\n"); }
    |
    ;

AssignmentExpression:
    AdditiveExpression          { printf("AdditiveExpression\n"); }
    | IDENTIFIER '=' AdditiveExpression     {  printf("IDENTIFIER '=' AdditiveExpression\n");}
    ;

AdditiveExpression:
    MultiplicativeExpression{  printf("MultiplicativeExpression\n"); }
    | AdditiveExpression '+' MultiplicativeExpression { printf("AdditiveExpression '+'
MultiplicativeExpression\n"); }
    | AdditiveExpression '-' MultiplicativeExpression { printf("AdditiveExpression '-'
MultiplicativeExpression\n"); }
    ;

MultiplicativeExpression:
    PrimaryExpression                   { printf("PrimaryExpression\n"); }
    | MultiplicativeExpression '*' PrimaryExpression { printf("MultiplicativeExpression '*'
PrimaryExpression\n"); }
    | MultiplicativeExpression '/' PrimaryExpression { printf(" MultiplicativeExpression '/'
PrimaryExpression\n"); }
    | MultiplicativeExpression '%' PrimaryExpression { printf("MultiplicativeExpression '%%'
PrimaryExpression\n"); }
    ;

PrimaryExpression:
    NUMBER      { printf("NUMBER\n"); }
    | IDENTIFIER {   printf("IDENTIFIER\n"); }
    | '(' Expression ')'    { printf("'(' Expression ')'\n"); }
    ;
%%

int main(int argc, char *argv[])
{

    yyparse();

    printf("Ok\n");

    return 0;
}


void yyerror(const char *str)
{
    fprintf(stderr, "syntax error!\n");
}
```

Şimdi programı çalıştırıp stdin dosyasından şşağıdaki gibi bir girişin yapıldığını düşünelim:

```
a + b
```

Ekrana aşağıdaki yazıların sırasıyla yazıldığını göreceksiniz:

```
IDENTIFIER
PrimaryExpression
MultiplicativeExpression
IDENTIFIER
PrimaryExpression
AdditiveExpression '+' MultiplicativeExpression
AdditiveExpression
AssignmentExpression
```

Burada Bison önce a'yı bulur ve onun IDENTIFIER olduğunu belirler. Sonra '+' atomu elde edildiğinde bunun artık "Additive '+' Multiplicative" kuralı olabileceğini düşünür. Sonra b'yi çektiğinde bunun da bir IDENTIFIER olduğunu anlar ve bu kuralı oluşturarak yukarıya doğru gider. Şimdi de aşağıdaki gibi bir giriş yapılmış olduğunu varsayalım:

```
a = 10 + 20
```

Burada Bison önce 'a' atomunu görecek bunun IDENTIFIER olduğunu belirler sonra '=' atomunu gördüğünde artık bunun "IDENTIFIER '=' AdditiveExpression" kuralına uygun olduğunu anlar. Tabii henüz indirgeme (reduce) işlemini yapmaz. Sonra 10 atomunu çeker sonra '+' atomunu ve sonra da 20'yi çekerek grameri tamamlar. Ekranda şınlar görünecektir:

```
NUMBER
PrimaryExpression
MultiplicativeExpression
NUMBER
PrimaryExpression
AdditiveExpression '+' MultiplicativeExpression
IDENTIFIER '=' AdditiveExpression
AssignmentExpression
```

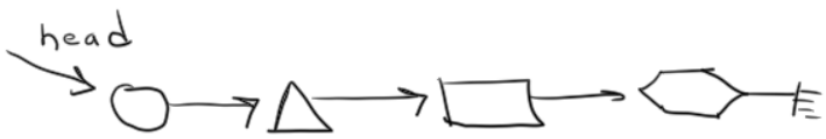
Görüldüğü gibi Bison'un aşağıdan yukarıya parse işlemi yapması programcının bir parse ağacı oluşturmasını kolaylaştırmaktadır.

3.9. Küçük Bir Dilin Tasarlanması ve Onun İçin Bir Yorumlayıcının Gerçekleştirilmesi

Bu bölümde küçük bir dil tasarlayıp o dil için bir yorumlayıcı yazacağız. Ancak yorumlayıcı yazımı için soyut bir sentaks ağacının da kurulması gerekmektedir. Bu sentaks ağacı heterojen düğümlerden oluşan bir ağaç biçimindedir. Bu nedenle öncelikle heterojen düğümlere sahip bağlı liste ve ağaç gibi veri yapılarının nasıl oluşturulacağı üzerinde duracağız.

3.9.1. Heterojen Düğümlere Sahip Bağlı Listelerin ve Ağaçların Oluşturulması

Soyut Sentaks Ağacı (Abstract Syntax Tree) oluşturulurken pek çok bağlı liste ve ağaç yapısının kullanılması gerekebilmektedir. Ancak bu bağlı liste ve ağaçlar homojen (yani aynı türden) düğümlere sahip değillerdir. Örneğin bir tek bağlı liste düşünelim. Fakat onların gösterdiği yerdeki düğümlerin içerikleri farklı olsun:



Buradaki şekiller farklı türdeki düğümleri temsil etmektedir. İşte heterojen düğümlere sahip bağlı listeleri ya da ağaçları temsil etmek için birkaç yöntem kullanılabilmektedir. Bu yöntemlerin hepsinin bazı avantajları ve dezavantajları söz konusu olabilmektedir.

1. Yöntem : Birliklerden Faydalanarak Tür Farklılığını Ortadan Kaldırmak

Bu yöntemde düğümler sanki aynı türdenmiş gibi ele alınır. Ortak elemanlar yapının başına yerleştirilir. Farklı elemanlar da bir birlik içerisinde ifade edilir. Tabii bu teknikte ayrıca o düğümün hangi türden olduğunu tutan bir type alanının da ortak elemanlar içerisine yerleştirilmesi gerekir. Örneğin:

```
enum TYPES {
    XNODE, YNODE, ZNODE
};

struct NODE {
    int type;
    struct NODE *next;
    union {
        struct XNODE xnode;
        struct YNODE ynode;
        struct ZNODE znode;
    } u;
};
```

Burada artık bir düğüm adresi elde edildiğinde onun içeriğine type elemanı yoluyla aşağıdaki gibi erişilebilecektir:

```
switch (node->type) {
    case XNODE:
        /* node->u.xnode  */
        break;
    case YNODE:
        /* node->u.ynode  */
        break;
    case ZNODE:
        /* node->u.znode  */
        break;
}
```

Bu yöntemin avantajı basit olmasıdır. Ancak birlik içerisindeki elemanların türleri farklı olduğundan birlik için ayrılan alan birliğin en büyük elemanı kadar olacak dolayısıyla da toplam kullanılan bellek miktarı da büyüyecektir. (Bir birlik için o birliğin en büyük elemanı kadar yer ayrılacağını anımsayınız.)

2. Yöntem : Farklı Türden Düğümlerin Ortak Kısımlarının Aynı Türle İfade Edilmesi

Bu yöntemde farklı türdeki düğüm yapılarının başındaki ortak kısım bir yapıyla temsil edilir. Bu ortak kısımda yine sonraki düğümü gösteren göstericiler ve düğümün türünü belirten type elemanı tutulur. Örneğin:

```
struct NODE {
    int type;
    struct NODE *next;
};

struct XNODE {
    int type;
    struct NODE *next;
    ....
};

struct YNODE {
    int type;
    struct NODE *next;
    ....
};
```

```
struct ZNODE {
    int type;
    struct NODE *next;
    ....
};
```

Burada XNODE, YNODE ve ZNODE yapılarının başlangıç kısımları aynıdır. Bu başlangıç kısımları NODE yapısındaki gibidir. O halde bu farklı yapılar sanki NODE türündenmiş gibi ifade edilebilirler. Tabii gerektiğinde bu NODE adreslerinin yeniden type elemanı yoluyla gerçek düğüm türlerine dönüştürülmesi gerekecektir. Örneğin:

```
switch (node->type) {
    case XNODE:
        {
            struct XNODE *xnode = (struct XNODE *)node;
            ...
        }
        break;
    case YNODE:
        {
            struct YNODE *ynode = (struct YNODE *)node;
            ...
        }
        break;
    case ZNODE:
        {
            struct ZNODE *znode = (struct ZNODE *)node;
            ...
        }
        break;
}
```

Ancak bu yöntemde C standartları bakımından küçük bir sorun vardır. C standartlarında yapının ilk elemanının adresinin yapı nesnesinin bütünsel adresiyle aynı olacağı belirtilmiştir. Ancak yapının sonraki elemanlarının artan adreslerde olduğu belirtilmişse de hizalama yüzünden elemanlar arasında kontrollü boşluklar bulunabilmektedir. Gerçi derleyiciler varsayılan durumda tüm yapılar için aynı hizalama biçimini kullanıyor olsalar da standartlar bağlamında bu durum garanti edilmemiştir. O halde yukarıdaki yapıyı standartlara uydurmak için ortak elemanların tek bir yapı elemanıyla ifade edilmesi gerekir:

```
struct NODE {
    int type;
    struct NODE *next;
};

struct XNODE {
    struct NODE node;
    ....
};

struct YNODE {
    struct NODE node;
    ....
};

struct ZNODE {
    struct NODE node;
    ....
};
```

Aslında bu yöntem nesne yönelimli programlama tekniğindeki türetme işlemine benzetilebilir. Çünkü orada da zaten nesnenin taban sınıf kısmı tipik olarak düşük adreste (daha yukarıda) tutulmaktadır ve türemiş sınıftan taban sınıfa adres dönüştürmesi yapılabilmektedir.

Bu ikinci yöntemde farklı düğümler yalnızca kendileri kadar yer kaplarlar. Ancak gerçekleştirilmeleri biraz daha zahmetlidir.

3. Yöntem : Farklı Türden Düğümlerin Elemanlarını Ayrı Bir Biçimde Tahsis Edip Onların Adreslerini Düğümde Tutmak

Bu yöntemde yine tüm düğümler sanki aynı türdenmiş gibi aynı yapıyla temsil edilir. Ancak bunların bir gösterici elemanı vardır ve o eleman heterojen türe ilişkin bilgileri tutan yapıyı gösterir. Örneğin:

```
struct NODE {
    int type;
    struct NODE *next;
    void *data;
};
...
switch (node->type) {
    case XNODE:
        {
            struct XNODE *xnode = (struct XNODE *)node->data;
            ...
        }
        break;
    case YNODE:
        {
            struct YNODE *ynode = (struct YNODE *)node->data;
            ...
        }
        break;
    case ZNODE:
        {
            struct ZNODE *znode = (struct ZNODE *)node->data;
            ...
        }
        break;
}
```

Bu daha kolay anlaşılabilir olsa da her düğüm için iki ayrı dinamik tahsisat gerekmektedir. Bu da heap alanının bölünmesini (fragmentation) artırabilmektedir.

3.9.2. Örnek Bir Dil Tasarımı

Şimdi bir dil tasarlayıp onun için bir yorumlayıcı yazalım. Örnek dilimiz prosedürel programlamayı destekleyen, içerisinde fonksiyonların, if gibi while ve for gibi kontrol değimlerinin bulunduğu bir dil olsun. Dilimizde iki veri türü bulunacaktır. Bunlar int ve double türleridir. void bir tür belirleyicisi olarak yalnızca fonksiyonların geri dönüş değerlerinde kullanılabilir. Bu durum fonksiyonun bir değer geri döndürmeyeceği anlamına gelir. Programın başlangıç noktası C ve C++'taki gibi main isimli fonksiyondur. Dilde iç içe bloklar bulunabilir. Bildirimler blokların başlarında yapılmak zorunda değildir. Bu dile bu dokümanlarda CSD dili denilecektir. CSD diline ilişkin örnek bir program şöyle olabilir:

int g_x;

```
void Foo(int  n)
{
    int  i;
```

```
    i = 0;
    while (i < n) {
        print i;
        ++i;
    }

    g_x = n;
}

void main()
{
    Foo(10);
    Foo(20);

    print x;
}
```

3.9.3. CSD Dilinin Grameri

Şimdi CSD dilinin gramerini Flex ve Bison ile ifade etmeye çalışalım. CSD Dilini atomlarına ayıran Flex kodu aşağıdaki gibidir:

```
/* csd.l */

%option noyywrap
%{
    #include "csd.tab.h"
    #include "stringtable.h"
%}

%%
int                 { return INT; }
double              { return DOUBLE; }
void                    { return VOID; }
if                  { return IF; }
else                { return ELSE; }
while               { return WHILE; }
for                 { return FOR; }
do                  { return DO; }
return              { return RETURN; }
println             { return PRINTLN; }
\+\+                    { return PLUSPLUS; }
--                  { return MINUSMINUS; }
==                  { return EQUAL; }
!=                  { return INEQUAL; }
\<\=                    { return LESSOREQU; }
\>\=                    { return GREOREQU; }
[a-zA-Z][a-zA-Z0-9]*    { yylval.id = lookup_str(yytext); return IDENTIFIER; }
[0-9]+              { yylval.numberInt = atoi(yytext); return NUMBER_INT; }
[0-9]+\.[0-9]+      { yylval.numberDouble = atof(yytext); return NUMBER_DOUBLE; }
\"(\\.|[^\\"])*\"   { yytext[yyleng - 1] = '\0'; yylval.str = lookup_str(yytext + 1); return STRING; }
[+\-*()/%=;{}]      { return *yytext; }
[ \t\n]
.                   { return *yytext; }
%%
```

CSD dilinin Bison Grameri de aşağıdaki gibi oluşturulmuştur:

```
/* csd.y */

%{
    #include "common.h"
    #include "csd.h"
```

```
%}

%expect 2

%union {
     void *astNode;
     void *llHead;
     char *id;
     char *str;
     int numberInt;
     int typeSpecifier;
     double numberDouble;
};

%token INT
%token DOUBLE
%token VOID
%token IDENTIFIER
%token NUMBER_INT
%token NUMBER_DOUBLE
%token STRING
%token IF
%token ELSE
%token WHILE
%token FOR
%token DO
%token EQUAL
%token INEQUAL
%token GREOREQU
%token LESSOREQU
%token PLUSPLUS
%token MINUSMINUS
%token RETURN
%token PRINTLN

%type <astNode> CSD
%type <llHead> DefinitionList
%type <astNode> Definition
%type <astNode> VariableDefinition
%type <astNode> FunctionDefinition
%type <typeSpecifier> TypeSpecifier
%type <astNode> VariableList
%type <astNode> InitDeclarator
%type <llHead> ParameterList
%type <llHead> ArgumentList
%type <astNode> Argument
%type <astNode> Parameter
%type <astNode> Statement
%type <llHead> StatementList
%type <astNode> WhileStatement
%type <astNode> DoWhileStatement
%type <astNode> DefinitionStatement
%type <astNode> IfStatement
%type <astNode> ForStatement
%type <astNode> SimpleStatement
%type <astNode> CompoundStatement
%type <astNode> ReturnStatement
%type <astNode> PrintlnStatement
%type <astNode> Expression
%type <astNode> ForExpression
%type <astNode> AssignmentExpression
%type <astNode> EqualityExpression
%type <astNode> RelationalExpression
%type <astNode> AdditiveExpression
%type <astNode> MultiplicativeExpression
%type <astNode> UnaryExpression
%type <astNode> PostfixExpression
%type <astNode> PrimaryExpression
%type <astNode> Identifier
%type <astNode> String

%%

CSD:
     DefinitionList                     { g_astRoot = $1; }
     |                                  { $$ = NULL;  }
     ;

DefinitionList:
```

```
    Definition                          { $$ = ProcessList(NULL, $1); }
    | DefinitionList Definition     { CheckFunctionDefinition($1, $2); $$ = ProcessList($1, $2); }
    ;

Definition:
    VariableDefinition         { $$ = $1;  }
    | FunctionDefinition       { $$ = $1; }
    ;

FunctionDefinition:
    TypeSpecifier Identifier '(' ParameterList ')'    CompoundStatement { $$ = ProcessFunctionDefinition($1, $2, $4, $6);
}
    ;

ParameterList:
    Parameter                           { $$ = ProcessList(NULL, $1);}
    | ParameterList ',' Parameter       { $$ = ProcessList($1, $3); }
    | VOID                                  { $$ = ProcessList(NULL, NULL); }
    |                                       { $$ = ProcessList(NULL, NULL); }
    ;

Parameter:
    TypeSpecifier Identifier            { $$ = ProcessParameter($1, $2);}
    ;

VariableDefinition:
    TypeSpecifier VariableList ';'      { $$ = ProcessVariableDefinition($1, $2); }
    ;

VariableList:
    InitDeclarator                              { $$ = ProcessList(NULL, $1);}
    | VariableList ',' InitDeclarator       { $$ = ProcessList($1, $3); }
    ;

InitDeclarator:
    Identifier                              { $$ = ProcessInitDeclarator($1, NULL); }
    | Identifier '=' Expression         { $$ = ProcessInitDeclarator($1, $3); }
    ;

ArgumentList:
    Argument                        { $$ = ProcessList(NULL, $1);}
    | ArgumentList ',' Argument     { $$ = ProcessList($1, $3); }
    |                                   { $$ = NULL; }
    ;
Argument:
    Expression        { $$ = $1; }
    ;

TypeSpecifier:
    INT                         { $$ = TYPE_INT;}
    | DOUBLE               { $$ = TYPE_DOUBLE;}
    | VOID                      { $$ = TYPE_VOID;}
    ;

StatementList:
    Statement                       { $$ = ProcessList(NULL, $1);}
    | StatementList Statement       { $$ = ProcessList($1, $2); }
    ;

Statement:
    SimpleStatement                     { $$ = $1; }
    | CompoundStatement                 { $$ = $1; }
    | DefinitionStatement           { $$ = $1; }
    | IfStatement                   { $$ = $1; }
    | WhileStatement                { $$ = $1; }
    | ForStatement                      { $$ = $1; }
    | DoWhileStatement              { $$ = $1; }
    | ReturnStatement               { $$ = $1; }
    | PrintlnStatement              { $$ = $1; }
    ;

DefinitionStatement:
    VariableDefinition     { $$ = $1; }
    ;

CompoundStatement:
    '{' StatementList '}'  { $$ = ProcessCompoundStatement($2); }
    | '{' '}'              { $$ = ProcessCompoundStatement(NULL);}
    ;
```

```
SimpleStatement:
    Expression ';'            { $$ = ProcessSimpleStatement($1); }
    | ';'                     { $$ = ProcessSimpleStatement(NULL); }
    ;

IfStatement:
    IF '(' Expression ')' Statement                          { $$ = ProcessIfStatement($3, $5, NULL); }
    | IF '(' Expression ')' Statement ELSE Statement      { $$ = ProcessIfStatement($3, $5, $7); }
    ;

WhileStatement:
    WHILE '(' Expression ')' Statement      { $$ = ProcessWhileStatement($3, $5);}
    ;

ForStatement:
    FOR '(' ForExpression ';' ForExpression ';' ForExpression ')' Statement     { $$ = ProcessForStatement($3, $5, $7,
$9);}
    ;

DoWhileStatement:
    DO Statement WHILE '(' Expression ')' ';'    { $$ = ProcessDoWhileStatement($2, $5); }
    ;

ReturnStatement:
    RETURN Expression ';'          { $$ = ProcessReturnStatement($2);}
    | RETURN ';'                   { $$ = ProcessReturnStatement(NULL); }
    ;

PrintlnStatement:
    PRINTLN Expression ';'         { $$ = ProcessPrintlnStatement(1, $2); }
    | PRINTLN String ';'           { $$ = ProcessPrintlnStatement(0, $2); }
    ;

ForExpression:
    Expression                         { $$ = $1; }
    |                                  { $$ = NULL; }
    ;

Expression:
    AssignmentExpression                                  { $$ = $1; }
    ;

AssignmentExpression:
    EqualityExpression                                    { $$ = $1; }
    | Identifier '=' AssignmentExpression                 { $$ = ProcessBinaryOperator(OPERATOR_ASSIGNMENT,  $1, $3); }
    ;

EqualityExpression:
    RelationalExpression                                  { $$ = $1; }
    | EqualityExpression EQUAL RelationalExpression       { $$ = ProcessBinaryOperator(OPERATOR_EQUAL,  $1, $3); }
    | EqualityExpression INEQUAL RelationalExpression { $$ = ProcessBinaryOperator(OPERATOR_INEQUAL,  $1, $3); }
    ;

RelationalExpression:
    AdditiveExpression                                    { $$ = $1; }
    | RelationalExpression '<' AdditiveExpression     { $$ = ProcessBinaryOperator(OPERATOR_LESS,  $1, $3); }
    | RelationalExpression '>' AdditiveExpression     { $$ = ProcessBinaryOperator(OPERATOR_GREATER,  $1, $3); }
    | RelationalExpression GREOREQU AdditiveExpression    { $$ = ProcessBinaryOperator(OPERATOR_GREOREQU,  $1, $3); }
    | RelationalExpression LESSOREQU AdditiveExpression { $$ = ProcessBinaryOperator(OPERATOR_LESSOREQU,  $1, $3); }
    ;

AdditiveExpression:
    MultiplicativeExpression                              { $$ = $1; }
    | AdditiveExpression '+' MultiplicativeExpression { $$ = ProcessBinaryOperator(OPERATOR_PLUS,  $1, $3); }
    | AdditiveExpression '-' MultiplicativeExpression { $$ = ProcessBinaryOperator(OPERATOR_MINUS,  $1, $3); }
    ;

MultiplicativeExpression:
    UnaryExpression                                       { $$ = $1; }
    | MultiplicativeExpression '*' UnaryExpression    { $$ = ProcessBinaryOperator(OPERATOR_MULTIPLY,  $1, $3); }
    | MultiplicativeExpression '/' UnaryExpression  { $$ = ProcessBinaryOperator(OPERATOR_DIVIDE,  $1, $3); }
    | MultiplicativeExpression '%' UnaryExpression  { $$ = ProcessBinaryOperator(OPERATOR_MODULUS,  $1, $3); }
    ;

UnaryExpression:
    PostfixExpression                  { $$ = $1; }
    | '-' UnaryExpression              { $$ = ProcessUnaryOperator(OPERATOR_PREFIX_MINUS, $2);}
    | '+' UnaryExpression              { $$ = ProcessUnaryOperator(OPERATOR_PREFIX_PLUS, $2);}
```

```
    | PLUSPLUS Identifier              { $$ = ProcessUnaryOperator(OPERATOR_PREFIX_PLUSPLUS, $2);}
    | MINUSMINUS Identifier               { $$ = ProcessUnaryOperator(OPERATOR_PREFIX_MINUSMINUS, $2);}
    ;

PostfixExpression:
    PrimaryExpression                  { $$ = $1; }
    | Identifier '(' ArgumentList ')'  { $$ = ProcessFunctionCallOperator(OPERATOR_FUNCTION_CALL, $1, $3); }
    | Identifier PLUSPLUS              { $$ = ProcessUnaryOperator(OPERATOR_POSTFIX_PLUSPLUS, $1);}
    | Identifier MINUSMINUS                { $$ = ProcessUnaryOperator(OPERATOR_POSTFIX_MINUSMINUS, $1);}
    ;

PrimaryExpression:
    NUMBER_INT                { $$ = ProcessNumberInt(yylval.numberInt); }
    | NUMBER_DOUBLE           { $$ = ProcessNumberDouble(yylval.numberDouble); }
    | String              { $$ = $1; }
    | Identifier          { $$ = $1; }
    | '(' Expression ')'  { $$ = $2; }
    ;

Identifier:
    IDENTIFIER                { $$ = ProcessIdentifier(yylval.id); }
    ;
String:
    STRING                    { $$ = ProcessString(yylval.str); }
    ;

%%
```

3.9.4. CSD Dili İçin Soyut Sentaks Ağacının Tasarımı

CSD Dili için soyut sentaks ağacınını oluşturan düğümleri parça parça belirleyelim. Aşağıdaki düğümlerin hepsi o düğümün türünü belirten ortak bir type elemanı ile başlamaktadır. Düğümlere ilişkin yapılar ASTNODE_XXX isimli yapılarla temsil edilmiştir. ASTNODE isimli yapı ise yalnızca type elemanı olan genel bir yapıyı temsil etmektedir:

```
typedef struct tagASTNODE {
    int type;
    double number;
} ASTNODE;
```

Şimdi ağacı oluşturan füğümleri tek tek ele alalım.

1) NUMBER_INT atomuna ilişkin düğüm aşağıdaki gibi bir yapı ile temsil edilebilir:

```
typedef struct tagASTNODE_NUMBER_INT {
    int type;
    int number;.
} ASTNODE_NUMBER_INT;
```

Burada number sabit olan tamsayı değeri belirtmektedir.

2) NUMBER_DOUBLE atomuna ilişkin düğüm aşağıdaki gibi bir yapı ile temsil edilebilir:

```
typedef struct tagASTNODE_NUMBER_DOUBLE {
    int type;
    double number;
} ASTNODE_NUMBER_DOUBLE;
```

Burada number sabit olan double türden değeri belirtir.

3) IDENTIFER atomuna ilişkin düğüm aşağıdaki gibi bir yapıyla temsil edilebilir:

```
typedef struct tagASTNODE_IDENTIFIER {
    int type;
    char *id;
```

```
} ASTNODE_IDENTIFIER;
```

Burada id değişklen isminin bulunduğu alanın adresini tutmaktadır. Değişken isimlerinin yalnızca bir kez string tablolarına yerleştirildiğini anımsayınız.

4) STRING atomuna ilişkin düğüm aşağıdaki gibi bir yapıyla temsil edilebilir:

```
typedef struct tagASTNODE_STRING {
    int type;
    const char *str;
} ASTNODE_STRING;
```

Burada str stringin belirttiği yazıyı göstermektedir.

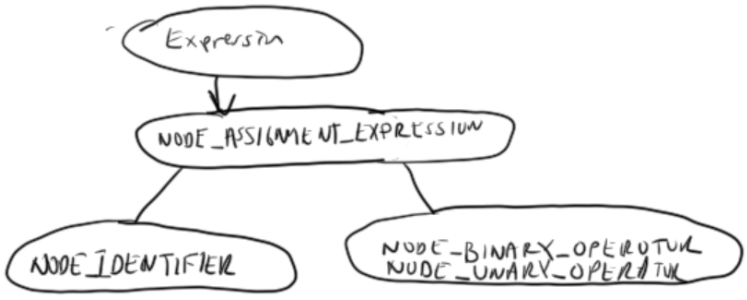
5) Tek operandlı tüm operatörlere ilişkin düğümler NODE_UNARY_OPERATOR isimli bir yapıyla temsil edilmektedir. Bu yapının içerisinde operatörün türü ve ilgili operandın adresleri vardır:

```
typedef struct tagASTNODE_UNARY_OPERATOR {
    int type;
    int opType;
    ASTNODE *exp;
} ASTNODE_UNARY_OPERATOR;
```

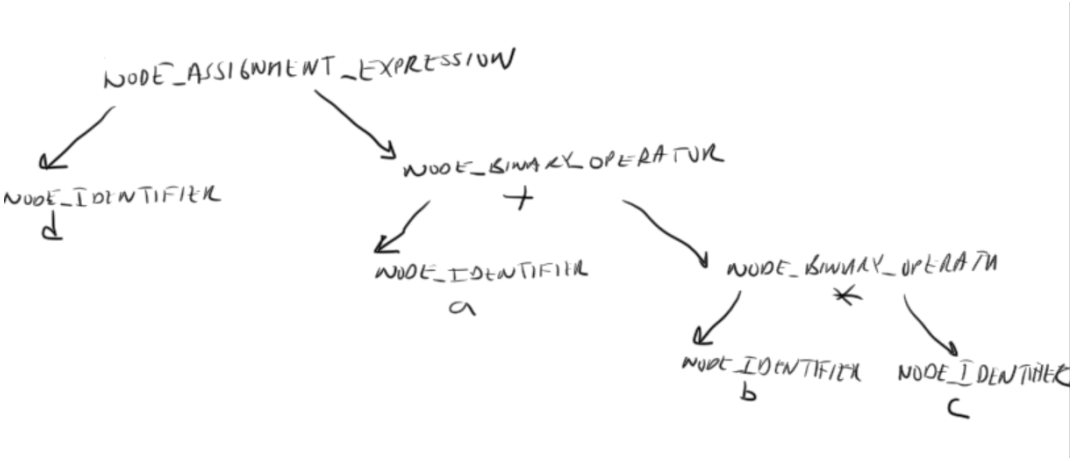
6) İki operandlı tüm operatörlere ilişkin düğümler NODE_BINARY_OPERATOR isimli bir yapıyla temsil edilmektedir. Bu yapının içerisinde operatörün türü ve iki operanda ilişkin düğümlerin adresleri vardır:

```
typedef struct tagASTNODE_BINARY_OPERATOR {
    int type;
    int opType;
    void *leftExp;
    void *rightExp;
} ASTNODE_BINARY_OPERATOR;
```

Tek operand'lı ve iki operand'lı operatörleri aynı yapılarla ifade etmek soyut sentaks ağacını oldukça sadeleştirmektedir. Bu durumda soyut sentaks ağacındaki Expression özet olarak şöyle bir ağaçla ifade edilebilmektedir:



Örneğin d = a + b * c gibi bir ifadenin soyut sentkas ağacı aşağıdaki gibi olacaktır:



7) return deyimi ASTNODE_RETURN_STATEMENT isimli yapıyla temsil edilebilir:

```
typedef struct tagASTNODE_RETURN_STATEMENT {
    int type;
    void *exp;
} ASTNODE_RETURN_STATEMENT;
```

Burada exp return anahtar sözüğünün yanındaki ifadeyi temsil etmektedir.

8) while döngüsü ASTNODE_WHILE_STATEMENT isimli yapıyla temsil edilmiştir:

```
typedef struct tagASTNODE_WHILE_STATEMENT {
    int type;
    void *statement;
} ASTNODE_WHILE_STATEMENT;
```

9) DO-while döngüsü ASTNODE_doWHILE_STATEMENT isimli yapıyla temsil edilmiştir:

```
typedef struct tagASTNODE_DOWHILE_STATEMENT {
    int type;
    void *exp;
    void *statement;
} ASTNODE_DOWHILE_STATEMENT;
```

10) For döngüsü ASTNODE_FOR_STATEMENT isimli yapıyla temsil edilebilir:

```
typedef struct tagASTNODE_FOR_STATEMENT {
    int type;
    void *exp1;
    void *exp2;
    void *exp3;
    void *statement;
} ASTNODE_FOR_STATEMENT;
```

11) İf deyimi ASTNODE_IF_STATEMENT iismli yapıyla temsil edilebilir:

```
typedef struct tagASTNODE_IF_STATEMENT {
    int type;
    void *exp;
    void *statementTrue;
    void *statementFalse;
} ASTNODE_IF_STATEMENT;
```

12) Basit deyim ASTNODE_SIMPLE_STATEMENT isimli yapıyla temsil edilebilir:

```
typedef struct tagASTNODE_SIMPLE_STATEMENT {
```

```
    int type;
    void *exp;
} ASTNODE_SIMPLE_STATEMENT;
```

13) Bileşik deyimler ASTNODE_COMPOUND_STATEMENT isimli yapıyla temsil edilebilir:

```
typedef struct tagASTNODE_COMPOUND_STATEMENT {
    int type;
    void *sllist;
} ASTNODE_COMPOUND_STATEMENT;
```

14) Bildirim deyimleri ASTNODE_VARIABLE_DEFINITION isimli yapıyla temsil edilebilir:

```
typedef struct tagASTNODE_VARIABLE_DEFINITION {
    int type;
    int typeSpecifier;
    void *sllist;
} ASTNODE_VARIABLE_DEFINITION;
```

15) Fonksiyon tanımlamaları ASTNODE_FUNCTION_DEFINITION isimli yapıyla temsil edilebilir:

```
typedef struct tagASTNODE_FUNCTION_DEFINITION {
    int type;
    int typeSpecifier;
    void *identifier;
    void *sllist;
    void *compoundStatement;
} ASTNODE_FUNCTION_DEFINITION;
```

16) Fonksiyon parametreleri ASTNODE_PARAMETER isimli yapıyla temsil edilebilir:

```
typedef struct tagASTNODE_PARAMETER {
    int type;
    int typeSpecifier;
    void *identifier;
} ASTNODE_PARAMETER;
```

17) PRINT deyimi de ASTNODE_PRINT_STATEMENT isimli bir yapıyla temsil edilebilir:

```
typedef struct tagASTNODE_PRINT_STATEMENT {
    int type;
    int expFlag;
    void *exp;
} ASTNODE_PRINT_STATEMENT;
```

Burada expFlag “print” anahtar sözcüğünün yanındakilerin bir ifade mi yoksa string mi olduğunu belirtmektedir.

Buradaki dile ilişkin yorumlayıcı "Src/022-CSDInterpreter" dizininde gerçekleştirilmiştir.

Bison’da Gramer Hatalarının Yakalnması ve Rapor Edilmesi

Anımsanacağı gibi Bison gramere uymayan bir atomun farkına vardığında yyerror isimli fonksiyonu çağırıp prosesi sonlandırmaktadır. Gerçekten de pek çok yorumlayıcı derleme sırasında ya da çalışma zamanı sırasında ilk hata görüldüğünde hatayı rapor ederek işlemini sonlandırır. Derste örnek olarak yazdığımız CSD diline ilişkin yorumlayıcıda da böyle yapılmıştır. Halbuki özellikle derleyici yazımında bir hata görüldüğünde derleme işleminin devam etmesi ve tüm hataların bir liste halinde rapor edilmesi gerekmektedir.

Derleyici ya da yorumlayıcılardaki hata ele alımında iki duruma dikkat edilmelidir:

1) Hatanın yapıldığı yer rapor edilmelidir.
2) Hata mesajları hatanın kaynağını açıklayacak biçimde oluşturulmalıdır.

Bazen programcının yaptığı tek bir hata zincirleme olarak onlarca hatanın oluşmasına yol açabilir. Bu tür durumlarda mümkün olduğu kadar az hata mesajının verilmesi uygun olur.

Sentaks hatalarının ortaya çıktığı yer aslında yalnızca kaynak dosyayı okuyan lexical analiz modülü tarafından (yani flex tarafından) bilinmektedir. Bu nedenle Bison'un bu bilgiyi Flex'ten alması gerekir. Flex son atomun bulunduğu yeri yylineno isimli bir global değişkende tutar. Ancak Flex'in bu son atomun çekildiği satır numarasını yylineno değişkenine yerleştirmesi için %option yylineno bildiriminin yapılmış olması gerekmektedir. Programcı yylineno isimli global değişkene Bison modülünden erişebilir.

Flex'in yylineno global değişkenin yanı sıra bir de Bison'un yyloc isimli global değişkeni vardır. Tipik olarak Bison içerisinde Flex'in yylineno değişkeninin doğrudan kullanılması yerine Flex'te Bison'un yyloc değişkeninin içinin doldurması yöntemi tercih edilmektedir. yyloc değişkeni YYLTYPE isimli bir yapı türündendir:

```
typedef struct YYLTYPE
{
    int first_line;
    int first_column;
    int last_line;
    int last_column;
} YYLTYPE;
```

Burada first_line hatanın başladığı ilk satırı, last_line son satırı, first_column ilk sütunu ve last_column da son sütunu belirtmektedir. Bu noktada Flex ile ilgili bir özellik daha devreye girmektedir. Flex her atomu bulduğunda henüz belirtilen kodu çalıştırmadan önce eğer YY_USER_ACTION isimli makro define edilmişse o makroyu çağırmaktadır. İşte biz de o makroda yyloc değişkenini set edebiliriz. Örneğin:

```
int g_column = 1;

#define YY_USER_ACTION yylloc.first_line = yylloc.last_line = yylineno; \
yylloc.first_column = g_column; yylloc.last_column = g_colunn + yyleng -1; \
g_column += yyleng;
```

Burada makro içerisinde yylineno değişkeninden hareketle hataya yol açan atomun başlangıç ve bitiş sütunları ve başlangıç satırı yyloc değişkenin içerisine yerleştirilmiştir. Ayrıca Bison'un yyloc değişkenini kullanabilmesi için komut satırı argümanı olarak --location seçeneğinin girilmesi gerektiğini belirtelim.

Build Araçları (Build Automation Tools)

Birden fazla kaynak dosya ile proje geliştirirken bu dosyaların ayrı ayrı derlenmesi ve link edilmesi programcıya ciddi bir yük oluşturmaktadır. Çünkü projedeki bir kaynak dosyada değişiklik yapıldığında yalnızca o kaynak dosyanın yeniden derlenerek diğer derlenmiş kaynak dosyalarla birlikte hep beraber link işlemine sokulması gerekir. Bu rutin işlemler de programcının üretkenliği düşürebilmektedir. Bazen de bir projede farklı build işlemleri yapılabilmektedir. Örneğin projede önce bir DLL'in oluşturulması sonra o DLL'i kullanan bir uygulamanın derlenmesi gerekebilir. Hatta bazı projelerde birden fazla uygulamanının build edilmesi de gerekebilmektedir. İşte bu işlemleri kolaylaştıran araçlara "build araçları (build tools)" denilmektedir. Build araçları yalnızca C ve C++ için değil Java gibi .NET gibi ortamlarda da gereksinim duyulan araçlardandır.

Değişik platformlar ve diller için değişik build araçları gerçekleştirilmiştir. Bazı build araçları programlama dilinden bağımsız bir biçimde (yani her programlama dili için kullanılabilecek biçimde) tasarlanmışlardır. Bazıları ise belli bir programlama diline oldukça bağımlıdırlar. Build araçlarının listesi Wikipedia'dan incelenebilir (https://en.wikipedia.org/wiki/List_of_build_automation_software).

C ve C++ için en çok tercih edilen build araçları Make, CMake ve qmake isimli araçlardır. Make oldukça klasiktir ve en eski build araçlarındandır. Bugün bile hala en fazla tercih edilen build aracının Make olduğu söylenebilir. Make'in pek çok versiyonu oluşturulmuştur. GNU projesi kapsamında geliştirilen Make aracına "GNU Make" denilmektedir. Microsoft Windows sistemleri için Make aracının değişik bir uyarlamasını oluşturmuştur. Buna "nmake" denilmektedir. Klasik Make ile nmake arasında küçük farklılıklar vardır.

Make aracının öğrenilmesi ve kullanılması biraz zordur. Bunun için zamanla değişik build araçları da geliştirilmiştir. Örneğin CMake son zamanlarda çok popüler hale gelen bir build aracıdır. CMake platform bağımsız ve basit bir sentaks içeren bir build aracıdır. Bu build aracı build işlemini kendisi yapmaz, ilgili platforma yönelik build dosyaları oluşturur. (Örneğin Linux, Mac OS X gibi sistemlerde CMake normal GNU Make dosyası oluşturur. Sonra bu make dosyası Make işlemine sokulur. Ya da Windows sistemlerinde CMake Micrsoft'un solution dosyası da üretebilmektedir.)

C ve C++'taki diğer bir build aracı da "qmake" denilen araçtır. Aslında qmake Qt isimli GUI ortamı için düşünülmüş bir build aracıdır. Ancak bu ortamın dışında da klasik C ve C++ projelerinde kullanılabilmektedir. Qmake de build işleminin kendisini yapmamakta CMake gibi ilgili platforma özgü bir make dosyası oluşturmaktadır.

Java dünyasında "Ant" isimli build aracı çok yoğun kullanılmaktadır. Bunun alternatifi olarak "Maven" da Java geliştiricileri tarafından tercih edilebilmektedir. .NET dünyasında ise ağırlıklı olarak Microsoft'un "MSBuild" aracı kullanılmaktadır. MSBuild MSBuild XML tabanlıdır ve Visual Studio IDE'sinin kullandığı default build aracıdır.

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.

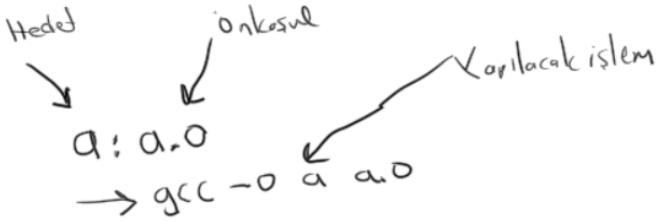
Bir Make dosyası kabaca kurallardan (rules) oluşmaktadır (rules). Bir kuralın genel biçimi şöyledir:

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

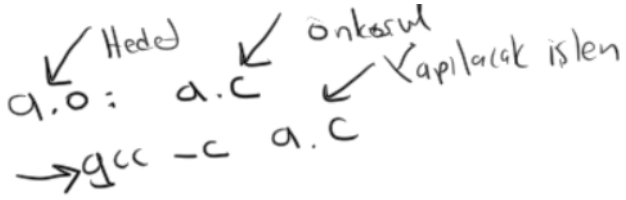
Kuralın yapılacak işlemler kısmının bir tab içeriden yazılması zorunludur. İlk yapılacak işlem istenirse önkoşullardan sonra aynı satırda ‘;’ atomundan sonra da belirtilebilir. Kurallar arasında boş satırlar bırakılabilir. Örneğin:

```
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:

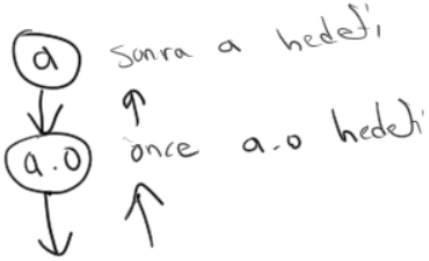
```
make a
```

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.

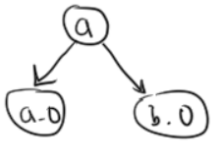
Make işlemlere başlamadan önce nihai hedefe varabilmek için gereken bağımlılık ağacı (dependency tree) oluşturulur, sonra işlemler aşağıdan yukarıya doğru yapılır. Örneğin yukarıdaki Make dosyasında nihai hedef olan “a” için bağımlılık ağacı şöyle oluşturulacaktır:



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 kuraldaki 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 sll 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 kurallar gerçekleştirilir.

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

CMake Aracının Kullanımı

CMake son yıllarda gittikçe daha poüler olmuştur. Genel kullanımı Make aracına göre daha kolaydır. Ancak giriş kısmında da belirtildiği gibi CMake aslında platforma bağlı olarak o platformda kullanılan build kaynak dosyalarını üretmektedir. Yani örneğin tipik olarak biz build işlemini CMake ile organize ederiz. CMake bize bir make dosyası üretir. Biz de onu make işlemine sokarız.

CMake hem Linux hem Windows hem de Mac OS X sistemlerinde kullanılabilen cross platform bir araçtır. Kurulumu oldukça kolaydır. Bunun için cmake.org sistesinde "download" kısmına gelinir. İlgili platformdaki kurulum dosyası indirilir. Örneğin Windows için ".msi" uzantılı "install dosyası" vardır. Linux sistemlerinde doğrudan ilgili dağıtımın sunduğu paket yöneticilerinden faydalanılabilir. Örneğin apt-get programı ile kurulum aşağıdaki gibi yapılabilmektedir:

```
sudo apt-get install cmake
```

Fakat zaten Linux sistemlerinde pek çok dağıtımda "cmake" temel bir araç olarak zaten kurulmuş durumda olmaktadır. Tabii biz onu güncellemek isteyebiliriz. Mac OS X sistemlerinde de kurulum dosyası cmake.org sitesinden indirilebilir. Her ne kadar cmake aslında komut satırından çalıştırılan bir araçsa da bir GUI ortamı da ayrıca bulunmaktadır. Windows kurulumunda hem komut satırı aracı hem de GUI aracı birlikte kurulmaktadır.

CMake'in de ayrı bir dili vardır. Bu dil komutlardan (commands) oluşmaktadır. Komutlar arasında istenildiği kadar boşluk karakterleri bırakılabilir. Komutların genel sentaks biçimi şöyledir:

```
<komut ismi> (argüman listesi)
```

Argüman listesi virgül atomlarıyla değil boşluk karakterleriyle birbirlerinden ayrılmaktadır. CMake komutlarında büyük harf-küçük harf duyarlılığı yoktur. Fakat küçük harf ağırlıklı bir yazım tercih edilmektedir.

Geleneksel olarak CMake dosyası "CMakeLists.txt" ismiyle bulundurulmaktadır. Bu isim cmake programı tarafından dizin içerisinde aranmaktadır. Tabii biz aslında CMake dosyasına istediğimiz bir ismi de verebiliriz.

En önemli komutlardan biri add_executable isimli komuttu. Bu komut istenildiği kadar çok argüman alabilir. İlk argüman hedef dosyanın ismidir. Sonraki argümanlar projeye dahil olan kaynak dosyaları belirtir. Örneğin:

```
add_executable(a a.c b.c)
```

Burada oluşturulacak hedef "a" isimli çalıştırılabilir dosyadır. Bunun için "a.c" ve "b.c" dosyaları derlenerek bağlanacaktır. Komuttaki argümanların başluk karakterleriyle birbirlerinden ayrıldığına dikkat ediniz.

project isimli komut projenin ismini belirtmektedir. Her projeye bir isim vermek zorunlu olmasa da tavsiye edilmektedir. Örneğin:

```
project(a)
```

Bu durumda minimal bir CMake dosyası aşağıdaki gibi oluşturulabilir:

```
project(a)
add_executable(a a.c b.c)
```

Bu CMake dosyası nasıl işleme sokulur? Bunun için komut satırında CMake dosyasının ismi verlerek cmake programı çalıştırılabilir. Örneğin:

```
cmake CMakeLists.txt
```

Ya da dosya ismi yerine bir dizin ismi verilirse cmake o dizinde CMakeLists.txt dosyasını arar. Bulursa zaten onu işleme sokar. Örneğin proje dizininde olduğumuzu varsayalım:

```
cmake .
```

CMake bize default durumda hangi platformdaysak o platforma ilişkin default build dosyası üretmektedir. Örneğin Linux ortamında cmake bize MakeFile isimli GNU make dosyası üretecektir. O halde bizim cmake işleminden sonra make işlemi yaparak build işlemini yapmamız gerekir. Tabii geliştirme sırasında projeye yeni bir kaynak dosya eklemediysek yeniden cmake işlemini yapmamıza gerek yoktur. Yalnızca make işlemi yapabiliriz.

CMake dosyalarının başında genellikle bir cmake_minimum_required komutu bulunur. CMake aracı zamanla farklı özelliklere sahip olduğu için CMake dosyasının eski bir versiyon ile işleme sokulmasını engellemek amacıyla bu komutun bulundurulması tavsiye edilmektedir. Komutun genel biçimi şöyledir:

```
cmake_minimum_required(VERSION x.x.x)
```

Örneğin:

```
cmake_minimum_required(VERSION 2.8.9)
project(a)
add_executable(a a.c b.c)
```

Projelerde genellikle include dosyaları ayrı dizinlerde tutulmaktadır. İşte derleyicinin o include dizinlerine bakmasının sağlanması için include_directories komutu bulundurulmuştur. Komutun yalın genel biçimi şöyledir:

```
include_directories(dizin1 dizin2 dizin3....)
```

Dizinlerin yol ifadeleri göreli ya da mutlak verilebilir. Örneğin:

```
cmake_minimum_required(VERSION 2.8.9)
project(a)
include_directories(inc)
add_executable(a a.c b.c)
```

cmake programı pek çok doğal build aracı için kod üretebilmektedir. Eğer komut satırında -G seçeneği ile bu belirleme yapılmamışsa default durum ele alınır. Default durum PATH çevre değişkeninde bulunan derleme arfaçlarıyla değişebilmektedir. Windows'ta default olarak Microsoft'un "msbuild" aracı için ".sln" uzantılı "solution" ve buna bağlı proje dosyaları üretilmektedir. Linux sisteemlerinde default durum "GNU Make" dosyasının üretilmesidir. Tabii biz -G seçeneği ile bu default durumu değiştirebiliriz. Örneğin Windows'ta gcc'nin MinGW port'u ile mingw32-make aracı için kod üretilmesini -G "MinGW Makefiles" seçeneği ile sağlayabiliriz. Örneğin:

```
cmake -G "MinGW Makefiles" .
```

Diğer seçenekleri görmek için komut satırında -G seçeneğini vererek yardım alabilirsiniz:

```
cmake -G
```

CMake dosyası içerisinde tıpkı Make aracında olduğu gibi makrolar (değişkenler) kullanılabilir. Bir makroya değer yerleştirmek için set komutu kullanılmaktadır. set komutunun genel biçimi şöyledir:

```
set(<makro ismi> <değer1> <değer2> ...)
```

Örneğin:

```
set(SOURCES a.c b.c c.c d.c)
```

Makroların değerleri ${<makro ismi>} ile elde edilir. Örneğin:

```
cmake_minimum_required(VERSION 2.8.9)
project(a)
include_directories(inc)
set(SOURCES a.c b.c)
add_executable(a ${SOURCES})
```

CMake dosyasında '#' karakteri yorumlama anlamına gelmektedir. Bu karakterden satır sonuna kadarki tüm karakterler cmake programı tarafından dikkate alınmamaktadır.

Birden fazla dosya ismini joker karakteri kullanarak bir makroya atamak için file komutu kullanılmaktadır. Eğer file komutu bu amaçla kullanılacaksa komutun önekinde GLOB belirlemesinin yapılması gerekir. Örneğin:

```
Örneğin:

cmake_minimum_required(VERSION 2.8.9)
project(a)
include_directories(inc)
file(GLOB SOURCES *.c)
add_executable(a ${SOURCES})
```

Burada dizindeki bütün C dosyaları build işlemine dahil edilmiştir.

Kütüphane dosyası yaratmak için add_library komutu kullanılmaktadır. Komutun yalın genel biçimi şöyledir:

```
add_library(<isim> [STATIC | SHARED] <dosya1> <dosya2> ...)
```

Default durum static kütüphane biçimindedir. Bir kütüphane ya da çalıştırılabilen dosyanın link aşamasında bir kütüphaneyi kullanabilmesi için target_link_libraries komutunun uygulanması gerekir. Örneğin:

```
cmake_minimum_required(VERSION 2.8.9)
project(a)
include_directories(inc)
add_library(b STATIC b.c)
add_executable(a a.c)
target_link_libraries(a b)
```

Burada "a" isimli çalıştırılabilen dosya elde edilmek istenmiştir. Ancak "b" isimli bir static kütüphane yapılmış ve a'nın o static kütüphaneyi kullanması sağlanmıştır. Benzer biçimde dinamik kütüphane de aşağıdaki gibi oluşturulabilir:

```
cmake_minimum_required(VERSION 2.8.9)
project(a)
include_directories(inc)
add_library(b SHARED b.c)
```

```
add_executable(a a.c)
target_link_libraries(a b)
```

cmake programının ürettiği build dosyalarında zaten "clean" isimli bir hedef vardır. Ancak install hedefi install komutuyla oluşturulmaktadır. install komutunun genel biçimi şöyledir:

```
install(<tür> <isim> DESTINATION <yer>)
```

Tür olarak TARGETS kullanılırsa isim de bir hedef ismi olur. Bu durumda ilgili hedef ile üretilen dosya DESTINATION ile belirtilen dizine kopyalnır. Örneğin:

```
cmake_minimum_required(VERSION 2.8.9)
project(a)
include_directories(inc)
add_library(b SHARED b.c)
add_executable(a a.c)
target_link_libraries(a b)
set(CMAKE_INSTALL_PREFIX .)
install(TARGETS b DESTINATION bin)
install(TARGETS a DESTINATION bin)
```

DESTINATION ile belirtilen dizin göreli ise SMake onun kök dizinini CMAKE_INSTALL_PREFIX isimli makronun belirttiği yerde aramaktadır. Bunun da platforma bağlı olarak default bir değeri vardır. Ancak yukarıdaki örnekte biz set komutuyla bu yeri değiştirdik. Şimdi artık komut satırında "make install" ya da "mingw32-make install" yaptığımız zaman kütüphane dosyaları ile çalıştırılabilen dosyalar bin dizinine kopyalanacaktır.

CMake dilinde tıpkı GNU Make dilinde olduğu gibi IF gibi deyimler de vardır. if deyiminin genel biçimi şöyledir:

```
if ([Tür] <ifade>)
...
else
....
endif
```

Buradaki tür EXISTS gibi, IS_DIRECTORY gibi bazı operatörlerden oluşturulabilmektedir. Bunların listesi için cmake dokümanlarına bakılabilir. Örneğin EXISTS bir özelliğin var olup olmadığını sorgulamakta kullanılır. Örneğin:

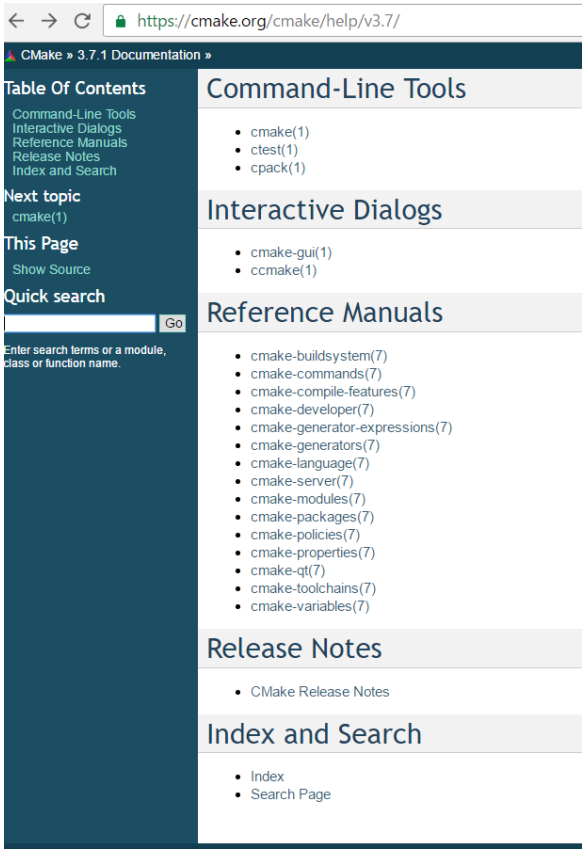
```
if (MINGW)
   ....
endif
```

Burada eğer MinGW build sistemi kullanılıyorsa ilgili komutun dahil edilmesi sağlanmaktadır. MINGW gibi pek çok önceden tanımlanmış makro vardır. Bu makrolar hangi sistem söz konusuysa ona göre true ya da false değeri verirler. Örneğin:

```
if (WIN32)
   ...
endif
```

if komutunun dışında döngü komutları da vardır.

Burada biz temel bir CMake kullanımını ele aldık. Halbuki bu dilin pek çok ayrıntısı da vardır. Bunun CMake'in orijinal dokümanlarından ya da kurstaki EBook dizininde bulunan "Mastering CMake" kitabından faydalanılabilir. CMake'in orijinal kaynakları çeşitli kategoriler altında dokümante edilmiştir.



QMake Aracının Kullanılması

QMake aslında Qt platformu için geliştirilmiş olan bir build aracıdır. Ancak bu platformun dışında da kendisine kullanılma alanı bulmuştur. Kurulumu Qt platformunun (framewok'ünün) kurulumuyla otomatik yapılmaktadır. Ancak yanızca QMake de daha az dosyanın diske çekilmesi ile de kurulabilmektedir. Qt Creator IDE'sinin default build sistemi QMake'tir. QMake bir bakımdan CMake'e benzetilebilir. QMake de tıpkı CMake gibi aslında GNU Make ya da NMake dosyası üretmektedir. Bu dosyanın make işlemine sokulmasıyla proje build edilir.

QMake dosyası da komutlardan oluşmaktadır. En önemli iki komut SOURCES ve HEADERS komutlarıdır. Örneğin:

```
HEADERS = myclass.h login.h  mainwindow.h
HEADERS += test.h
SOURCES = myclass.cpp login.cpp mainwindow.cpp
SOURCES += test.cpp
```

QMake dosyalarının uzantıları geleneksel olarak ".pro" biçimindedir. Sonra bu pro dosyası qmake işlemine sokulur:

```
qmake sample.pro
```

Bunun sonucunda qmake bize bir make dosyası üretir. Bu make dosyası da make (ya da nmake) işlemine sokulmaktadır. Biz bir dizinde qmake programını "-project" seçeneğiyle çalıştırırsak qmake bize o dizindeki C, C++ kaynak ve başlık dosyalarından bir qmake dosyasını oluşturmaktadır.

MSBuild Aracının Kullanımı

Microsoft uzun süre build aracı olarak “make” aracının kendisine özgü “nmake” isimli bir versiyonunu kullandı. Daha sonra msbuild isimli build sistemine geçti. Bugün Microsoft’un Visual Studio IDE’leri temel olarak MSBuild sistemini kullanmaktadır.

MSBuild XML tabanlı bir build dosyası kullanmaktadır. Build dosyasında önceden tanımlanmış pek çok tag bulunmaktadır. Bu tag’ların anlamları Microsoft’un dokümanlarındanb öğrenilebilir. Tabii daha önceden de belirttiğimiz gibi aslında Visual Studio IDE’sinde görsel olarak bir proje oluşturulduğunda zaten IDE bu build dosyasını bizim için otomatik oluşturmaktadır. MSBuild aracı komut satırından da kullanılabilir. Örneğin:

```
msbuild sample.sln
```

Microsoft’un dışında MonoDevelop gibi bazı IDE’ler de MSBuild sistemini default build sistemi olarak kullanmaktadır.

C’de Veritabanı Yönetim Sistemleriyle İşlemler

Her ne kadar veritablarının kullanılması sistem programalama faaliyeti değilse de pek çok sistem programında bir biçimde ufak da olsa veritabanı kullanma gereksinimi oluşmaktadır. Bu bölümde biz saf C ile Veritabanı Yönetim Sistemleriyle İlgili İşlemler Yapma konusunu ele alacağız.

Kaydedilen bilgilere hızlı erişmek için oluşturulmuş olan veri topluluklarına veritabanı (database) denilmektedir. Şüphesiz veritabanları işletim sistemlerinin sunduğu dosya sistemi ile ele alınıp kontrol edilmektedir. Ancak veritabanları bilginin hızlı elde edilmesi için algoritmik bir organizasyon oluşturmaktadır. Biz de dosya işlemlerini kullanarak uygun veri yapıları ve algoritmaları kullanarak kendi veritabanlarımızı kendimiz oluşturabiliriz. Veritabanlarının organizasyonunda özellikle “B Tree” gibi, “hash tabloları” gibi veri yapıları tercih edilmektedir. “Sistem Programlama ve İleri C Uygulamaları - 1” kursunda biz bu veri yapılarını temel düzeyde görmüştük. O kursa “B Tree” teorik olarak ele alınmış ancak gerçekleştirimi yapılmamıştı.

Ticari uygulamaların çok büyük çoğunluğu veritabanı kullanmaktadır. Veritabanı işlemleri pek çok ticari yazılımın performanslarını belrleyen önemli bir etken olmaktadır. Veritabanı işlemleri kabaca üç biçimde yapılabilir:

1) Programcı veri yapıları ve algoritmalara ilişkin teorik altyapıya sahipse veritabanı işlemlerini yapan kütüphaneleri kendisi oluşturabilir.

2) Programcı çeşitli kurumların ve firmaların oluşturulmuş olduğu veritabanı kütüphanelerini kullanabilir. (Örneğin tarihsel açıdan bakıldığında DBVista, Btrieve, CTree gibi pek çok veritabanı kütüphanesi kullanılmıştır.)

3) Programcı veritabanı işlemlerini yapmak için oluşturulmuş ismine "Veritabanı Yönetim Sistemi (Database Management System)" denilen özel yazılımları kullanabilir. Bugün veritabanı işlemleri ağırlıklı olarak VTYS'ler kullanılarak yapılmaktadır.

Veritabanı Yönetim Sistemleri (Database Management Systems)

VTYS'ler veritabanı işlemlerini yapmak için geliştirilmiş özel yazılımlardır. Tipik olarak VTYS yazılımlarının özellikleri şunlardır:

- VTYS'lerde aşağı seviyeli dosya işlemleriyle kullanıcının ilişkisi kesilmiştir. Kullanıcılar VTYS'lerle çalışırken yüksek seviyeli soyutlamalar kullanırlar. Örneğin VTYS kullanıcıları bilgilerin hangi dosyalarda nasıl tutulduğu gibi konularla ilgilenmezler.

- VTYS'ler client-server mimariye uygun olarak tasarlanırlar. Yani onlara birden fazla kişi aynı anda erişip işlem yaptırabilir. VTYS'ler çok sayıda veritabanını barındırarak kullanıcılara sunabilmektedir.

- VTYS'ler belli bir güvenlik mekanizmasına sahiptir. Böylece bir kullanıcı başka bir kullanıcının veritabalarına erişip oradaki bilgileri kullanamaz. Yani bunlara erişmek için "user name", "password" gibi bilgilere sahip olmak gerekir.

- VTYS'ler bilgilerin bozulmasına karşı dirençli biçimde tasarlanmışlardır. Örneğin elektrik kesilmesi gibi bir durumda sistem kendini onarabilmektedir.

- VTYS'ler bize ilave bazı araçlar da sunarlar. Örneğin backup-restore gibi utility'lere sahiplerdir.

- VTYS'ler işleri kolay yapmak için "yönetim konsollarına" da sahiptirler. Yani bunlar üzerinde komut satırından ya da görsel olarak işlemler yapılabilmektedir.

- VTYS'ler kullanıcıdan istekleri yüksek seviyeli deklaratif diller yoluyla almaktadır. Örneğin SQL bu amaçla kullanılan bir dildir. Biz VTYS'nin veritabanına kayıt eklemesi için bir SQL komutunu ona veririz. VTYS o SQL komutunu parse eder ve bizim istediğimiz işlemi arka planda C/C++ ile yazılmış olan kodları çalıştırarak yapar. (SQL yalnızca bizim VTYS'ye istekte bulunmamız için kullanılmaktadır. Yoksa VTYS aşağı seviyeli disk işlemlerini C/C++'ta yazılmış motor kısmıyla yapar.)

Bugün için çok kullanılan DBMS'ler şunlardır:

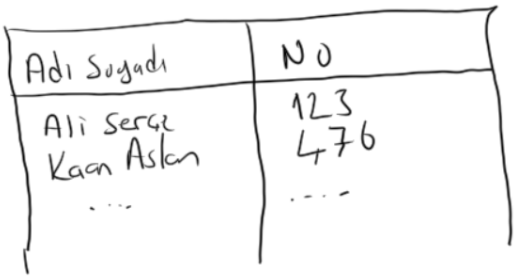
- DB2 (IBM)
- Oracle (Oracle)
- Sql Server (Microsoft)
- MySql (Open Source, fakat Oracle'ın artık)
- PostgreSql (Open Source)
- H2 (Open Source)
- SqLite (Open Source)
- Access (Jet Motor) (Microsoft)

Veritabanı Modelleri

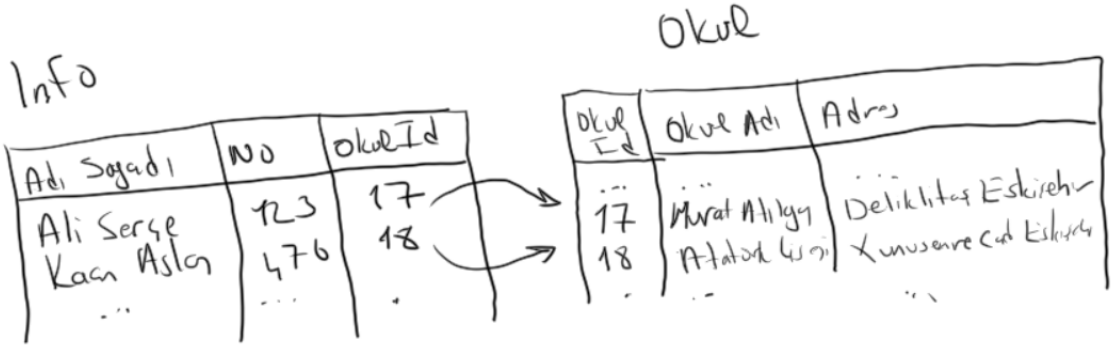
Veritabanları için pek çok model tasarlanmıştır. Bugün için endüstride en çok kullanılan model "ilişkisel veritabanı modeli (relational database model)"dir. Bunun dışında "nesne yönelimli", "hiyerarşik" modeller de belli yaygınlıkta kullanılmaktadır. Ancak "büyük veri (bid data)" konusunun yaygınlaşmasıyla özellikle kalıpların (ya da yazıların) saklanması ve aranmasını kolaylaştıran yeni modeller de kullanılmaya başlanmıştır. Genel olarak bu modellere "No SQL" denilmektedir. Bazı uygulamalarda birden fazla veritabanı modeli de kullanılmaktadır. Örneğin projede bazı veriler için ilişkisel veritabanlarında saklanırken bazı veriler ise "no sql" tarzı veritabanlarında saklanabilmektedir.

İlişkisel Veritabanı Yönetim Sistemleri (Relational Database Management Systems)

Veritabanlarının gerçekleştirilmesinde için çeşitli modeller (paradigmalar) kullanılmaktadır. Günümüzde en çok kullanan model ilişkisel modeldir. İlişkisel modelde veritabanı kullanıcıya tablolar biçiminde gösterilir. Bir tablo satır ve sütunlardan oluşur. Tablonun sütunlarına alan (field) satırlarına genel olarak kayıt (record) denilmektedir. İlişkisel veritabanlarında örnek bir tabloyu şöyle gösterebiliriz:



İlişkisel veritabanlarında bir veritabanı birden fazla tablodan oluşabilmektedir. Her tablo farklı bilgileri tutar. Böylece bilgiler çeşitli tablolara yayılmış olarak bulunur. Örneğin:



İlişkisel veritabalarında ideal olarak veri tekrarı yapılmaz. Yani bir bilgi yalnızca tek bir tabloda bulunur. Tablolar arasında geçiş yapmak için ortak bir anahtar kullanılır. Yukarıdaki örnekte bu ortak anahtar “Okul Id”si sütunudur. Verilerin tek bir tabloda bulunacak biçimde organize edilmesine “normalizasyon” denilmektedir. Bugün VTYS tablolarının tasarlanmasında ve normalize edilmesinde çeşitli araçlar da kullanılabilmektedir.

Büyük işletmeler ve kurumlar çok büyük veritabanlarına sahip olabilmektedir. Büyüklük burada tablo sayısının ve kayıt sayısının çokluğuyla tanımlanır. Örneğin “amazon.com” sitesinin, “GSM servis sağlayıcılarının” veritabaları “e-devlet” veritabanı büyük veritabanlarıdır. Hizmet sektöründe ve üretim sektöründe faaliyer gösteren pek çok kurum büyük veritabanlarıyla çalışabilmektedir. Bu veritabanlarının organizasyonu ve işletilmesi artık tamamen farklı bir uzmanlık alanı haline gelmiştir. Bu alanda çalışanlara “Database Administrator (DBA)” denilmektedir. Ancak sistem programlamada kullanılan veritabanları genellikle küçük ya da orta ölçeklidir. Bu nedenle bunların oluşturulması ve idaresi temel bir veritabanı bilgisiyle yapılabilmektedir.

Gömülü VTYS Kavramı (Embedded DBMS)

Bir VTYS'nin kendisinin kurulması zaman alan bir süreçtir. Ayrıca VTYS'ler arka planda servis olarak çalıştıklarından belli bir sistem kaynağını da kullanırlar. Bazı küçük ve orta ölçekli uygulamalarda bir VTYS'nin kurulması istenmeyebilir. Örneğin küçük bir rehber uygulaması için MySql gibi bir VTYS'nin hedef bilgisayara kurulması ve konfigüre edilmesi zahmetl bir süreçtir. Bu tür uygulamalarda VTYS gibi davranan fakat aslında tek bir kütüphaneden oluşan (DLL'den oluşan) VTYS'ler kullanılmaktadır. Bunlara gömülü (embedded) VTYS denir. Gömülü VTYS'ler gömülü sistemlerde de yoğun olarak kullanılmaktadır. Gömülü VTYS'ler client-server biçiminde çalışma sunmazlar. Aslında bunlar yapı bakımından veritabanı kütüphanelerine benzemektedir. Ancak VTYS'lerin bazı özelliklerini barındırmaktadır. Gömülü VTYS'lerin en çok kullanılanı SQLite'tır. Microsoft'un Jet Motoru da Windows sistemlerinde kullanılmaktadır. (Örneğin Access bu Jet motorunu kullanıyor. Bu yüzden bu VTYS'ye access de denilmektedir.)

MySql'in Kurulumu

MySql'i kurmak için tek yapılacak şey server programı http://dev.mysql.com/downloads/ sitesnden indirip yüklemektir. Kurulum oldukça basittir. Birtakım sorular default değerlerle geçilebilir. Ancak kurulum sırasında MySql kurulum programı bizden "root" isimli yetkili kullanıcının parolasını istenecektir. Diğer Bu parola yetkili olarak VTYS'ye bağlanmak için gerekir. Server programın yanı sıra bir yönetim ekranı elde etmek için ayrıca "MySql Workbench" programı da kurulabilir.

Sql Server'ın Kurulumu

SqlServer paralı bir üründür. Fakat bunun da "Express Edition" isminde bedava bir sürümü vardır. Bu sürüm Microsoft'un sayfasından indirilip kurulabilir. Tıpkı MySql'de olduğu gibi Sql Server'da da yönetim konsol programı da vardır. Buna "Sql Server Management Studio" denilmektedir. Bunun da indirilip kurulması tavsiye edilir.

SQLite'ın Kurulumu

SQLite zaten tek bir DLL'den oluşmaktadır. Dolayısıyla aslında kurulumu diye bir durum söz konusu değildir. Fakat biz burada C için örnekler yaparken SQLite başlık dosyalarına ve SQLite DLL'inin import kütüphanesine sahip olmak zorundayız. Bunların nasıl elde edileceği sonraki konularda ele alınacaktır. SQLite yönetim konsolu olarak pek çok alternatif vardır. Bunlardan biri "FireFox Add On olarak çalışmaktadır. İkinci bir seçenek ise "SQLite Studio" aracıdır. Cross Platform olan bu araç ilgili web sayfasından indirilerek kurulabilir.

SQL Veri Türleri

SQL ISO tarafından standardize edilmiş bir dildir. Ancak VTYS'ler bu standarları desteklemekle birlikte kendilerine özgü eklentilere ve komutlara da sahip olabilmektedir. Bu nedenle örneğin MySql'deki SQL ile SqlServer'daki SQL arasında ayrıntılarda farklılıklar olabilmektedir.

SQL veri türleri tablo sütunlarını oluştururken o sütunlardaki bilginin formatını belirlemekte kullanılmaktadır. Standart SQL veri türlerinin önemli olanları şunlardır:

INTEGER: Tamsayısal bilgileri tutan bir türdür. İstenirse kaç digitlik sayıların tutulacağı da belirtilebilir.

INT: Tipik olarak 4 byte uzunluğunda işaretli tamsayı türüdür. (Örneğin bu tür C'deki int türü ile temsil edilebilir.)

SMALLINT: Tipik olarak 2 byte'lık işaretli tamsayı türüdür. (Örneğin bu tür C'deki short türü ile temsil edilebilir.)

BIGINT: Tipik olarak 8 byte uzunluğunda işaretli tamsayı türüdür. (Örneğin bu tür C'deki long long türü ile temsil edilebilir.)

FLOAT: Tipik olarak 4 byte'lık gerçek sayı türüdür. (Örneğin bu tür C'deki float türü ile temsil edilebilir.)

DOUBLE: Tipik olarak 8 byte'lık gerçek sayı türüdür. (Örneğin bu tür C'deki double türü ile temsil edilebilir.)

TIME: Zaman bilgisini saklamak için kullanılan türdür.

DATE: Tarih bilgisini saklamak için kullanılan türdür.

CHAR(n): n karakterli yazıyı tutmak için kullanılan türdür.

VARCHAR(n): En fazla n karakterli bir yazıyı tutmak için kullanılan türdür.

TINYTEXT: Yazısal bilgileri tutmak için kullanılan türdür. (Tipik olarak 256 byte'a kadar)

TEXT: Yazısal bilgileri tutmak için kullanılan türdür. (Tipik olarak 64K'ya kadar)

LONGTEXT: (Tipik olarak 4GB'ye byte'a kadar)

TINYBLOB: Binary bilgileri tutmak için kullanılan türdür. (Tipik olarak 256 byte'a kadar)

BLOB: Binary bilgileri tutmak için kullanılan türdür. (Tipik olarak 64K'ya kadar)

LONGBLOB: Binary bilgileri tutmak için kullanılan türdür. (Tipik olarak 4GB'ye byte'a kadar)

Yukarıdaki türler pek çok VTYS'de vardır. Bunların dışında başka standart SQL veri türleri de bulunmaktadır. Ayrıca yukarıda belirtildiği gibi her VTYS'de diğerlerinde olmayan o VTYS özgü türler de bulunuyor olabilir.

Gömülü VTYS'ler dışındaki VTYS'ler (MySql, Oracle, SqlServer, Postgre SQL gibi) yukarıda da belirtildiği gibi client-server tarzda çalışmaktadır. Bunlar bazı seçenekler söz konusu olsa da temel olarak TCP/IP soket haberleşmesiyle haberleşirler. Yani bir client program VTYS server'ına arka planda bir soket açarak "connect" fonksiyonuyla bağlanmaktadır. Client ile Server arasındaki haberleşme protokolü VTYS'den VTYS'ye değişebilmektedir. Tabii VTYS'ye SQL komutları gönderen programcılar bu aşağı seviyeli protokolü bilmek zorunda değilllerdir. Çeşitli platformlarda yazılmış sınıflar ya da C'de sunulmuş olan fonksiyonlar arka planda belirlenmiş olan bu haberleşme protokolüne uygun olarak SQL cümlelerini göndererek sonucu almaktadır.

Temel SQL Komutları

Bu bölümde temel bazı SQL komutları yüzeysel olarak ele alınacaktır. SQL kolay bir dildir. Şüphesiz bu dilin ayrıntıları ne kadar iyi SQL bilinirse o kadar iyidir. Ancak biz kursumuzda temel SQL bilgisiyle işlerimizi yürüteceğiz. SQL büyük harf duyarlılığı olan bir dil değildir. Ancak geleneksel olarak anahtar sözcükler büyük harflerle yazılırlar. Veritabanı isimleri, tablo isimleri, alan isimleri vs. genellikle küçük harflerle isimlendirilmektedir. SQL komutlarının sonunda ';' atomu bulunur. Ancak pek çok VTYS bunu zorunlu tutmamaktadır. Tabii eğer peşi sıra birden fazla SQL komutu verielecekse bu durumda ';' komutlar aarasındaki sonlandırıcı olarak zorunlu hale gelir.

Anahtar Notlar: Bazı işlemler SQL komutlarıyla değil VTYS'ler için yazılmış yönetim programlarıyla görsel olarak da yapılabilmektedir. Aslında bu yönetim programları arka planda yine SQL komutlarını kullanarak işlemleri yaparlar. Fakat çoğu zaman bazı işlemler için (örneğin veritabanı yaratma, tablo oluşturma gibi) yönetim ekranlarını kullanmak daha pratiktir.

CREATE DATABASE Komutu: İlişkisel veritabanlarında "veritabanı" tablolardan oluşmaktadır. Bu nedenle önce bir veritabanının yaratılması, sonra da onun içerisinde tabloların yaratılması gerekir. Veritabanlarını yaratmak için CREATE DATABASE komutu kullanılır. Komutun genel biçimi şöyledir:

```
CREATE DATABASE <isim>;
```

Örneğin:

```
CREATE DATABASE student;
```

USE Komutu: Belli bir veritabanı üzerinde işlemler yapmak için öncelikle onun seçilmesi gerekir. Bu işlem USE komutuyla yapılır. Komutun genel biçimi şöyledir:

```
USE <isim>;
```

SHOW DATABASES Komutu: Bu komut VTYS'de yaratılmış olarak bulunan veritabanlarını gösterir. Komutun genel biçimi şöyledir:

```
SHOW DATABASES;
```

CREATE TABLE Komutu: Bu komut veritabanı için bir tablo yaratmak amacıyla kullanılır. Komutun temel genel biçimi şöyledir:

```
CREATE TABLE <isim> (<isim> <tür>, <isim> <tür>, <isim> <tür>... );
```

Aslında bu komutun bazı ayrıntıları vardır. Bu ayrıntılar ilgili dokümanlardan öğrenilebilir.

Örneğin:

```
CREATE TABLE person(person_name VARCHAR(45), person_telno CHAR(11), person_bdate INTEGER);
```

Bir tabloda tekrarlanması yasaklanmış olan sütunlara “birincil anahtar (primaey key)” denilmektedir. Tablodaki kayıtların hepsinin birincil anahtar sütunları farklı olmak zorundadır. Başka bir deyişle biz birtablya orada zaten var olan birincil anahtar değerine ilişkin bir kayıt ekleyemeyiz. Her tabloda bir tane birincil anahtarın olması tavsiye edilmektedir. Birincil anahtarın tablo yaratılırken sCREATE TABLE komutunda belirtilme biçimi çeşitli VTYS’lerde farklı olabilmektedir.

DROP TABLE Komutu: Bu komut tabloyu silmek için kullanılır. Genel biçimi şöyledir:

```
DROP TABLE <isim>;
```

Örneğin:

```
DROP TABLE person;
```

INSERT INTO Komutu: Bu komut bir tabloya bir satır eklemek için kullanılır. Komutun temel genel biçimi şöyledir:

```
INSERT INTO <tablo ismi> (sütun1, sütun2, sütun3,...)  VALUES (değer1, değer2, değer3,...);
```

Tabloya satır eklerken aslında her sütun bilgisinin belirtilmesi gerekmez. Bu durumda o sütun için tablo yaratılırken (CREATE TABLE komutunda) belirlenmiş olan default değerler kullanılır. Komutun ayrıntılı genel biçimi için ilgili dokğmanlara başvurabilirsiniz.

Örneğin:

```
INSERT INTO bilgiler(adi_soyadi, tel_no, eposta) VALUES('Sami Akyol', '5323153440', 'akyol@csystem.org')
```

Değerler girilirken yazılar ve tarihler tek tırnak içerisinde belirtilmelidir.

WHERE Cümleciği: Pek çok komut bir WHERE kısmı içermektedir. Where cümleciği koşul belirtmek için kullanılır. Koşullar karşılaştırma operatörleriyle oluşturulur. Mantıksal operatörlerle birleştirilebilir. Örneğin:

```
WHERE yas > 20 AND dogum_yeri = 'Eskişehir'
```

LIKE operatörü joker karakterleri kullanılarak belli bir kalıba uyan yazı koşulu oluşturur. Örneğin:

```
WHERE adi_soyadi LIKE 'A%'
```

Burada adi_soyadi 'a' ile başlayanlar koşulu verilmiştir. % karakteri "geri kalanı herhangi biçimde olabilir" anlamına gelir. Örneğin:

```
WHERE adi_soyadi LIKE '%an'
```

Burada sonu 'an' ile bitenler koşulu verilmiştir.

WHERE cümleciğinin bazı detayları vardır. Bu detaylar ilgili dokümanlardan öğrenilebilir.

DELETE FROM Komutu: Bu komut bir tablodan satır silmek için kullanılır. Genel biçimi şöyledir:

```
DELETE FROM <tablo ismi> <WHERE cümleciği>;
```

Örneğin:

```
DELETE FROM ogrenci WHERE adi_soyadi = 'Kaan Aslan' AND no = 12345
```

Burada öğrenci tablosundan adi_soyadi 'Kaan Aslan' ve numarası 12345 olan kayıt silinecektir. Bu komut kullanılırken dikkat etmek gerekir. Çünkü koşulu sağlayan ne kadar kayıt varsa hepsi tek hamlede silinmektedir.

UPDATE Komutu: Update komutu belli kayıtların alan bilgilerini değiştirmek amacıyla kullanılır. Örneğin ismi "Kağan" olan bir kaydı "Kaan" olarak değiştirmek isteyebiliriz. Ya da bir müşterinin bakiyesini değiştirmek isteyebiliriz. Komutun genel biçimi çöyledir:

```
UPDATE <tablo ismi> SET alan1 = değer1, alan2 = değer2, ... WHERE <koşul>;
```

Örneğin:

```
UPDATE student_info SET student_name = 'Kaan Kaplan' WHERE student_name = 'Kaan Aslan'
```

SELECT Komutu: Veritabanında belirli koşulları sağlayan kayıtlar SELECT komutuyla elde edilir. SELECT geniş bir komuttur. Genel biçimi oldukça karmaşıktır. Burada SELECT komutunun tipik kullanımlarını ele alacağız.

SELECT komutunun yalın kullanımı şöyledir:

```
SELECT <alan listesi> FROM <tablo ismi> WHERE <koşul>;
```

Örneğin:

```
SELECT student_name FROM student_info WHERE student_name LIKE 'K%';
```

Burada ismi K ile başlayan tüm kayıtların yalnızca isimleri elde edilmiştir. Birden fazla sütun aralarına ',' konularak beirtilir. Örneğin:

```
SELECT school_name, school_address FROM school_info WHERE school_name LIKE '%Eskişehir%';
```

Burada okul ismi içerisinde "Eskişehir" geçen okulların isimleri ve adresleri elde edilmiştir.

Sütun listesi yerine '* ' karakteri kullanılırsa "tüm sütunlar" kastedilmiş olur. Örneğin:

```
SELECT * FROM school_info WHERE school_name LIKE '%Eskişehir%';
```

Eğer SELECT komutunda WHERE cümleciği yoksa tüm kayıtlar listelenir. Örneğin:

```
SELECT school_name FROM school_info;
```

Burada tüm okulların isimleri elde edilmiştir.

SELECT komutuna ORDER BY cümleciği eklenebilir. ORDER BY anahtar sözcüklerini sütun listesi izler. Böylece ilgili kayıtlar burada belirtilen alanlara göre sıraya dizilir. ORDER BY cğmleceğini ASC ya da DESC izleyebilir. Default dizilim küçükten büyüğe (ASC) biçimindedir. Örneğin:

```
SELECT school_name, school_address FROM school_info ORDER BY school_name DESC;
```

ORDER BY cümleciği birden fazla alan içerebilir. Örneğin:

```
SELECT student_name, school_id FROM student_info WHERE student_name LIKE 'K%' ORDER BY student_name ASC,
school_id DESC;
```

Burada sıralama öğrenci ismine göre artan sırada yapılmaktadır. Ancak aynı isimli öğrenciler varsa bunlar da kendi aralarında okul id'lerine göre büyükten küçüğe elde edilecektir.

İlişkisel Veritabanlarında Bire Çok İlişkisi ve Join İşlemleri

İlişkisel veritabanlarında bilgiler birden fazla tabloya yayılmış olarak tutulurlar. Bu nedenle veritabanı yöneticisi birden fazla tablodan “yabancı anahtarlar (foreign keys) yardmıyla istediği bilgileri toplayabilmektedir. Örneğin MySql’in örnek “world” isimli veritabanında “city” tablosunda şehirlerin bilgileri bulunur. Ancak bu tabloda o şehirlerin ilişkin olduğu ülke bilgileri bulunmamaktadır. Yalnızca ülkenin ülke kodu bilgisi bulunmaktadır. İşte bu ülke kodu bilgisi kullanılarak diğer bir tablo olan “Country” tablosundan şehrin ilişkin olduğu ülkenin bilgileri elde edilebilir. İşte tablolar arasında yabancı anahtarlarla bilgi toplama işlemine SQL’de JOIN işlemi denilmektedir.

Join işlemi kartezyen çarpım işlemi biçiminde ele alınarak açıklanabilir. Bilindiği gibi iki kümenin kartezyen çarpımı sıralı ikililerden oluşmaktadır. Bu sıralı ikililerin ilk terimleri soldaki kümeden, ikinci terimleri sağdaki kümeden oluşturulur:

```
A X B = { (a, b) | a € A ve b € B}
```

İşte biz iki tabloyu bu biçimde kartezyen çarpım işlemine sokarsak iki tablonun eleman sayılarının çarpımı kadar kayıt elde etmiş oluruz. Sonra bu kayıtlardan WHERE cümlesi ile belirtilen koşulu sağlayanlar seçilirse bu işleme INNER JOIN denilmektedir. INNER JOIN sentaksı şöyledir:

```
SELECT <sütun listesi> FROM table1 INNER JOIN table2 ON <koşul>;
```

Sütun listesi ve koşul kısımlarında her iki tablonun sütunları bulundurulabileceğinden dolayı bir çakışma söz konusu olabilir. Çakışma durumunda sütun isimleri tablo isimleriyle araya ‘.’ karakteri konularak niteliklendirilebilir. Aslında SQL kullanıcıları çakışma

olmasa da sütunları hep tablo isimleriyle niteliklendirmektedir. Örneğin MySQL'in örnek "world" veritabanı için aşağıdaki sorgulamayı yapıyor olalım:

```
SELECT city.Name, country.Name FROM city INNER JOIN country ON city.CountryCode = country.Code;
```

Burada biz sonuç olarak city tablosundaki isimler ile country tablosoundaki isimleri beraber görüntülemek istemekteyiz. Ancak bu iki tablonun kartezyen çarpımındaki tüm satırlar için bu işlemler yapılmayacaktır. Yalnızca ON kısmında belirtilen koşulların sağlandığı satırlar elde edilecektir. Bu işlemin sonucunda da biz tüm şehirlerin hangi ülkeye ilişkin olduğuna ilişkin bir liste elde ederiz.

Name	Name
Oranjestad	Aruba
Kabul	Afghanistan
Qandahar	Afghanistan
Herat	Afghanistan
Mazar-e-Sharif	Afghanistan
Luanda	Angola
Huambo	Angola
Lobito	Angola
Benguela	Angola

Örneğin:

```
SELECT country.Name, countrylanguage.Language, countrylanguage.Percentage FROM country INNER JOIN
countrylanguage ON country.Code = countrylanguage.CountryCode
```

Name	Language	Percentage
Tonga	English	0.0
Tonga	Tongan	98.3
Trinidad and Tobago	Creole English	2.9
Trinidad and Tobago	English	93.5
Trinidad and Tobago	Hindi	3.4
Tunisia	Arabic	69.9
Tunisia	Arabic-French	26.3
Tunisia	Arabic-French-English	3.2
Turkey	Arabic	1.4
Turkey	Kurdish	10.6
Turkey	Turkish	87.6

INNER JOIN işlemi için alternatif bir sentaks daha vardır. Bu sentaks doğrudan birden fazla tablonun isminin geçtiği SELECT cümlesi sentaksıdır. Örneğin:

```
SELECT city.Name, country.Name FROM city INNER JOIN country ON city.CountryCode = country.Code
```

INER JOIN işleminin eşdeğeri şöyle de yazılabilir:

```
SELECT city.Name, country.Name FROM city, country WHERE city.CountryCode = country.Code
```

Örneğin:

```
SELECT country.Name, countrylanguage.Language, countrylanguage.Percentage FROM country INNER JOIN
countrylanguage ON country.Code = countrylanguage.CountryCode
```

INNER JOIN işleminin de eşdeğeri şöyle yazılabilir:

```
SELECT country.Name, countrylanguage.Language, countrylanguage.Percentage FROM country,
countrylanguage WHERE country.Code = countrylanguage.CountryCode
```

LEFT JOIN işleminde sol taraftaki tablonun tüm satırları ve ON koşulunu sağlayan satırlar alınır. Sol taraftaki tablonun ON koşulunu sağlamayan satırlarının sağ taraf sütunları boş (NULL) biçimdedir. Örneğin:

```
SELECT city.Name, country.Name FROM city LEFT JOIN country ON city.CountryCode = country.Code AND
country.Population > 50000000
```

Name	Name
Naogaon	Bangladesh
Sirajganj	Bangladesh
Narsinghdi	Bangladesh
Saidpur	Bangladesh
Gazipur	Bangladesh
Bridgetown	NULL
Antwerpen	NULL
Gent	NULL
Charleroi	NULL

RIGHT JOIN ise LEFT JOIN işleminin tersidir. Yani sağ taraftaki tablonun tüm satırları ve ON koşulunu sağlayan satırlar alınır. Sağ taraftaki tablonun ON koşulunu sağlamayan satırlarının sol taraf sütunları boş (NULL) biçimdedir. Örneğin:

```
SELECT city.Name, country.Name FROM city RIGHT JOIN country ON city.CountryCode = country.Code
AND country.Population > 50000000
```

Name	Name
NULL	Belgium
NULL	Benin
NULL	Burkina Faso
Dhaka	Bangladesh
Chittagong	Bangladesh
Khulna	Bangladesh
Rajshahi	Bangladesh

FULL JOIN pek çok VTYS tarafından desteklenmemektedir. Bu işlemde sol taraftaki ve sağ taraftaki tabloların bütün satırları ayrıca bir de koşulu sağlayan satırlar elde edilir. Ancak koşulu sağlamayan satırların diğer tablo karşılıkları boş (NULL) olarak elde edilir.

Aslında SQL burada anlatılanlardan daha ayrıntılı bir dildir. Ancak kursumuzda bu kadar bilgi yeterli görülmüştür. Fakat ne olursa olsun ne kadar çok SQL bilinirse o kadar etkin işlemler yapılabilmektedir.

C'de SQLite İle İşlemler

İşlemlere başlamadan önce SQLite'ın kurulumunu yapmamız gerekir. SQLite yukarıda da bahsedildiği gibi çok küçük (tek bir dinamikm kütüphaneden oluşan) bir VTYS'dir. Dolayısıyla onun kurulması Windows'ta bildiğimiz anlamda bir setup işlemi ile yapılmaz. Tabii bizim C'den SQLite kütüphanesini kullanabilmemiz için ona ilişkin başlık ve kütüphane dosyalarını elde etmemiz gerekir. Windows'ta buun için önce SQLite'ın resmi download sayfasına girilir (https://sqlite.org/download.html) . Sonra aşağıdaki iki zip dosyası indirilir:

1) Precompiled Binaries for Windows (32 bit ya da 64 bit)
2) Source Code (SQLLite Amalgamation)

Birinci indirmede Windows için gereken sqlite3.dll ve sqlite3.def dosyaları elde edilir. Buradaki ".def" dosyasına "module definition file" denilmektedir. Bu dosya "DLL'in import kütüphanesi" gibi link aşamasına dahil edilebilir. Ya da istenirse aşağıdaki komutla bu ".def" dosyasından ".lib" uzantılı "import kütüphanesi de oluşturulabilmektedir:

```
LIB /DEF:sqlite3.def
```

İkinci indirmeden biz SQLite'ın kaynak dosyalarını elde ederiz. Buradaki "sqlite3.h" dosyası SQLite fonksiyonları için başlık dosyası niteliğindedir.

Linux'ta SQLite'ı aynı biçimde ".zip" dosyalarını indirerek kurabiliriz. Ancak paket yönetici programlar bu paketleri otomatik olarak indirip kurabilmektedir. Debian kökenli (Debian, Ubuntu, Mint vs.) apt-get kullanan sistemlerde bu işlem şöyle yapılabilir:

```
sudo apt-get install sqlite3 libsqlite3-dev
```

Mac OS X için kurulum Windows'takine benzemektedir. Yine ilgili ".zip" dosyaları indirilip kurulum yapılabilir.

SQLite C API'lerinin dokümantasyonu https://sqlite.org/docs.html sayfasında ayrıntılı biçimde bulunmaktadır.

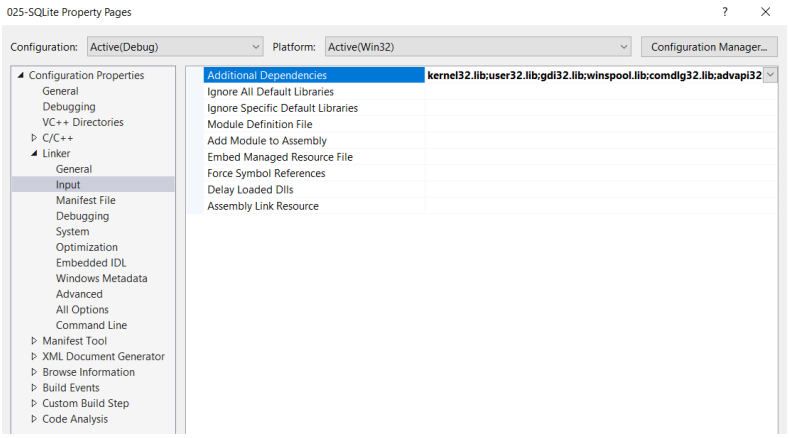
SQLite Test Kodunun Derlenerek Çalıştırılması

SQLite'ın versiyon numarasını yazdıran bir test kodu şöyle oluşturulabilir:

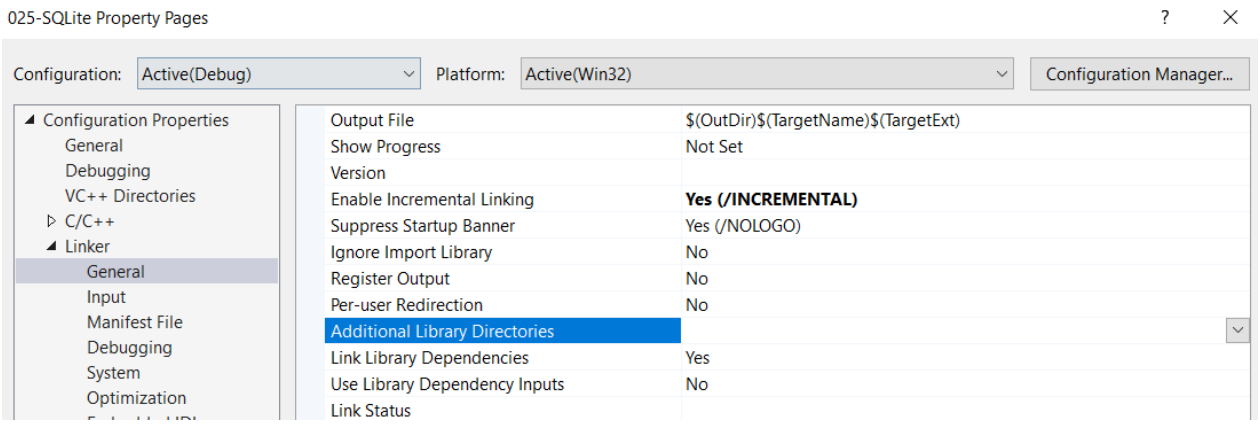
```
#include <stdio.h>
#include "sqlite3.h"

int main(void)
{
    printf("%s\n", sqlite3_libversion());

    return 0;
}
```

Visual Studio'da derleme ve bağlama işlemi için şunlara dikkat edilmelidir:

1) sqlite3.lib import kütüphanesinin link aaşamasına dahil edilmesi gerekir. Bu işlem proje seçeneklerinden "Linker/Input/Additional Dependencies" kısmındna yapılabilir:



Ancak sqlite3.lib eğer proje dizininde değilse onun bulunduğu dizinin de "Linker/General/Additional Library Directoires" ile belirtilmesi gerekir:



Ayrıca program çalışırken “sqlite3.dll” dosyasının DLL arama dizinlerinden birinin içerisinde bulunması gerekir. (Örneğin exe dosyasının bulunduğu yer, prosesin çalışma dizini, Windows, SysWOW64, Windows/System vs. 32 bit DLL’lerin 64 bit Windows sistemlerinde “System32” dizininde değil de “SysWOW64” dizininde bulunması gerektiğini anımsayınız.)

Linux ve Mac OS X sistemlerinde derleme işlemi gcc ile aşağıdaki gibi yapılabilir:

```
gcc -o sample sample.c -l sqlite3
```

Burada -l seçeneği ile “libsqlite3.a” ya da “libsqlite3.so” dosyası bağlama işlemine sokulmaktadır.

SQLite C Fonksiyonlarının Kullanımı

C’de SQLite veritabanı ile işlem yapmak için önce o veritabanının sqlite3_open fonksiyonuyla açılması gerekir. Bu işlemden sqlite3 türünden bir handle elde edilir. sqlite3_open fonksiyonun prototipi şöyledir:

```
int sqlite3_open(
  const char *filename,   /* Database filename (UTF-8) */
  sqlite3 **ppDb          /* OUT: SQLite db handle */
);
```

Fonksiyonun birinci parametresi bizden sqlite dosyasının yol ifadesini alır. İkinci parametresi sqlite3 isimli yapı türünden bir göstricinin adresini almaktadır. Fonksiyon handle alanını oluşturur. Onun adresini bu göstericinin içerisine yerleştirir. Fonksiyonun geri dönüş değeri işlemin başarısını belirtmektedir. Fonksiyon başarılıysa SQLITE_OK değerine geri döner. Fonksiyon başarısız olduğunda yine dosyanın sqlite3_close fonksiyonuyla kapatılması gerekir. Hata nedeni de sqlite3_errmsg fonksiyonuyla yazdırılabilir. Bu durumda sqlite dosyasının açılması tipik olarak şöyle yapılabilir.

```
#include <stdio.h>
#include <stdlib.h>
#include "sqlite3.h"

int main(void)
{
    sqlite3 *db;

    if (sqlite3_open("world.sqlite", &db) != SQLITE_OK) {
        fprintf(stderr, "Cannot open database: %s\n", sqlite3_errmsg(db));
        sqlite3_close(db);
        exit(EXIT_FAILURE);
    }

    printf("Ok\n");

    return 0;
}
```

VTYS’ye sql cümlesi göndermek için sqlite3_exec fonksiyonu kullanılır. Fonksiyonun prototipi şöyledir:

```
int sqlite3_exec(
  sqlite3*,                                  /* An open database */
  const char *sql,                           /* SQL to be evaluated */
  int (*callback)(void*,int,char**,char**),  /* Callback function */
  void *param,                               /* 1st argument to callback */
  char **errmsg                              /* Error msg written here */
);
```

Fonskiyonun birinci parametresi sqlite3_open fonksiyonundan elde edilen handle değeridir. İkinci parametre sql cümlesinin yazısını alır. Üçüncü parametre işlemden sonra çağrılacak “callback” fonksiyonun adresini alır.

Bu parametre NULL geçilebilir. Dördüncü parametre bu "callback" fonksiyona geçirilecek argümanı belirtir. Son parametre char * türünden bir göstericinin adresini almaktadır. Hata drumunda hata mesajının adresi bu göstericiye yerleştirilir. Fonksiyonun geri dönüş değeri işlemin abaşrısını belirtir. Fonksiyon başarılıysa SQLITE_OK değerine geri dönmektedir. Bu durumda biz hata mesajını yazdırdıktan sonra sqlite3_free fonksiyonu ile tahsis edilen alanı serbest bırakabiliriz. Örneğin:

```
#include <stdio.h>
#include <stdlib.h>
#include "sqlite3.h"

int main(void)
{
    sqlite3 *db;
    char *errMsg;
    char *cmd = "INSERT INTO student(student_name, student_no, school_id) VALUES ('Saadettin Teksoy',
7623, 2);";

    if (sqlite3_open("student.sqlite", &db) != SQLITE_OK) {
        fprintf(stderr, "Cannot open database: %s\n", sqlite3_errmsg(db));
        sqlite3_close(db);
        exit(EXIT_FAILURE);
    }

    if (sqlite3_exec(db, cmd, NULL, NULL, &errMsg) != SQLITE_OK) {
        fprintf(stderr, "Cannot open database: %s\n", errMsg);
        sqlite3_free(errMsg);
        sqlite3_close(db);
        exit(EXIT_FAILURE);
    }

    printf("Ok\n");

    return 0;
}
```

Veritabanından kayıtların elde edilmesi biraz daha ayrıntılı bir konudur. İstenilen kayıtların elde edilmesi için iki yol vardır. Birincisinde önce sqlite3_prepare fonksiyonu ile SQL SELECT cümlesi VTYS'ye gönderilir. Sonra her bir kayıt tek tek sqlite3_step fonksiyonu çağrılarak elde edilir. sqlite3_prepare fonksiyonundan elde edilen kayıtların bir liste oluşturduğunu sqlite3_step fonksiyonunun da listede sonraki kayıta geçtiğini düşünebilirsiniz. Yani adeta sqlite3_step fonksiyonu imleci konumlandırıyormuş gibidir. O andaki kayıtın sütun elemanları sqlite3_column_xxx fonksiyonlarıyla elde edilebilir. Burada xxx o sütunun türünü belirtmektedir.

sqlite3_prepare fonksiyonunun prototipi şöyledir:

```
int sqlite3_prepare(
  sqlite3 *db,            /* Database handle */
  const char *zSql,       /* SQL statement, UTF-8 encoded */
  int nByte,              /* Maximum length of zSql in bytes. */
  sqlite3_stmt **ppStmt,  /* OUT: Statement handle */
  const char **pzTail     /* OUT: Pointer to unused portion of zSql */
);
```

Fonksiyonun birinci parametresi sqlite3_open fonksiyonundan elde edilen handle değeridir. İkinci parametre SELECT cümlesini belirtir. Üçüncü parametre ikinci parametredeki SELECT cümlesine ilişkin yazının uzunluğunu belirtir. Bu parametre negatif değer geçilirse bu yazı null karaktere kadar ele alınır. Fonksiyonun dördüncü parametresi sqlite3_stmt türünden bir yapı göstericisinin adresini almaktadır. Bu da bir handle değeri gibidir. Kayotlar elde edilirken bu handle değeri kullanılmaktadır. Son parametre NULL geçirlebilir. Fonksiyon başarı durumunda SQLITE_OK değerine geri dönmektedir. Fonksiyon başarılı olduğunda imleç ilk kayıdın bir gerisini göstermektedir. Yani işleme önce bir kez sqlite3_step çağrısı yaparak başlamak gerekir. sqlite3_step fonksiyonunun prototipi şöyledir:

```
int sqlite3_step(sqlite3_stmt*);
```

Fonksiyonun parametresi sqlite3_prepare fonksiyonundan alınan handle değeridir. Son kayda erişildikten sonra sqlite3_step fonksiyonu SQLITE_DONE değerine geri dönmektedir.

İmleç konumlandırıldıktan sonra sütun değerleri sqlite3_column_xxx fonksiyonlarıyla elde edilmektedir. Bu fonksiyonlardan bazılarının prototipleri aşağıda verilmiştir:

```
const void *sqlite3_column_blob(sqlite3_stmt*, int iCol);
int sqlite3_column_bytes(sqlite3_stmt*, int iCol);
int sqlite3_column_bytes16(sqlite3_stmt*, int iCol);
double sqlite3_column_double(sqlite3_stmt*, int iCol);
int sqlite3_column_int(sqlite3_stmt*, int iCol);
sqlite3_int64 sqlite3_column_int64(sqlite3_stmt*, int iCol);
const unsigned char *sqlite3_column_text(sqlite3_stmt*, int iCol);
const void *sqlite3_column_text16(sqlite3_stmt*, int iCol);
int sqlite3_column_type(sqlite3_stmt*, int iCol);
sqlite3_value *sqlite3_column_value(sqlite3_stmt*, int iCol);
```

Bu fonksiyonların birinci parametreleri handle değerini ikinci parametreleri de sütun bilgisi elde edilecek sütunun indekx numarasını (index numarası sıfır orijinlidir) belirtir. Geri dönül değeri xxx türündendir.

sqlite_prepare fonksiyonuile elde edilen sqlite3_stmt türünden handle alanı sqlite3_finalize fonksiyonuyla serbest bırakılır:

```
int sqlite3_finalize(sqlite3_stmt *pStmt);
```

Örneğin:

```
#include <stdio.h>
#include <stdlib.h>
#include <Windows.h>
#include "sqlite3.h"

int main(void)
{
    sqlite3 *db;
    sqlite3_stmt *stmt;
    unsigned const char *name;
    int no;
    wchar_t wname[64];
    int result;

    SetConsoleOutputCP(65001);

    if (sqlite3_open("student.sqlite", &db) != SQLITE_OK) {
        fprintf(stderr, "Cannot open database: %s\n", sqlite3_errmsg(db));
        sqlite3_close(db);
        exit(EXIT_FAILURE);
    }
    if (sqlite3_prepare(db, "SELECT * FROM student WHERE student_id < 10", -1, &stmt, NULL) !=
SQLITE_OK) {
        fprintf(stderr, "Cannot open database: %s\n", sqlite3_errmsg(db));
        sqlite3_close(db);
        exit(EXIT_FAILURE);
    }

    while (sqlite3_step(stmt) != SQLITE_DONE) {
        name = sqlite3_column_text(stmt, 1);
        no = sqlite3_column_int(stmt, 2);
```

```
        printf("%s, %d\n", name, no);
    }

    sqlite3_finalize(stmt);
    sqlite3_close(db);

    return 0;
}
```

Yazısal sütunaların elde edilmesi sırasında encoding sorunu oluşabilir. Sqlite default olarak UTF8 kodlamasını kullanmaktadır. UTF8 kodlamasının console ekranında doğru gösterilmesi iki yolla yapılabilir:

1) UTF8 kodlamasını normal UNICODE kodlamaya dönüştürüp wprintf fonksiyonuyla ekrana yazdırmak. (Tabii console ekranının fcode page'inin buna göre ayarlanması gerekir.) UTF8 kodlamasını UNICODE'da dönüştürmek için Windows'un MultiByteToWideChar fonksiyonu kullanılabilir.

2) UTF8 kodlamasını normal printf fonksiyonuyla yazdırmaya çalışmak ancak console'un code page'ini UTF8'e ayarlamak. Console'un code page'ini UTF8'e ayarlamak için Windows sistemlerinde aşağıdaki çağrının yapılması gerekir:

```
SetConsoleOutputCP(65001);
```

SELECT cümlesinin uygulanmasında sqlite3_prepare fonksiyonunun dışında ikinci yöntem "callback" fonksiyon kullanmaktır. Bu durumda doğrudan sqlite3_exec fonksiyonuyla SELECT cümlesi verilir. Ancak bu fonksiyondaki "callback" fonksiyon NULL yerine her kayıt bulundukça çağrılacak fonksiyonun adresi biçiminde girilir. sqlite3_exec fonksiyonu her kayıt bulunduğunda bu fonksiyonu çağırmaktadır. Örneğin:

```
#include <stdio.h>
#include <stdlib.h>
#include <Windows.h>
#include "sqlite3.h"

int callback_func(void *notUsed, int argc, char **colContents, char **colNames)
{
    int i;

    for (i = 0; i < argc; ++i) {
        if (i != 0)
            printf(", ");
        printf("%s", colContents[i]);
    }
    printf("\n");

    return 0;
}

int main(void)
{
    sqlite3 *db;
    sqlite3_stmt *stmt;
    unsigned const char *name;
    int no, result;
    char *err_msg;

    SetConsoleOutputCP(65001);

    if (sqlite3_open("student.sqlite", &db) != SQLITE_OK) {
        fprintf(stderr, "Cannot open database: %s\n", sqlite3_errmsg(db));
        sqlite3_close(db);
        exit(EXIT_FAILURE);
    }
```

```
    if (sqlite3_exec(db, "SELECT student_name, student_no  FROM student WHERE student_id < 10",
callback_func,  NULL, &err_msg) != SQLITE_OK) {
        fprintf(stderr, "Cannot open database: %s\n", err_msg);
        sqlite3_free(err_msg);
        sqlite3_close(db);
        exit(EXIT_FAILURE);
    }
    sqlite3_close(db);

    return 0;
}
```

callback fonksiyonun parametreleri nelerdir? Bu yöntemde sqlite3_exec bize tüm sütunların içeriğini bir yazı dizisi olarak vermektedir. İşte callback fonksiyonun üçüncü parametresi bu dizidir. İkinci parametre ise bu dizinin uzunluğunu belirtir. Başka bir deyişle ikinci parametre aslında select edilen sütun sayısıdır. Son parametre veritabanındaki sütunların listesine ilişkin yazı dizisidir.

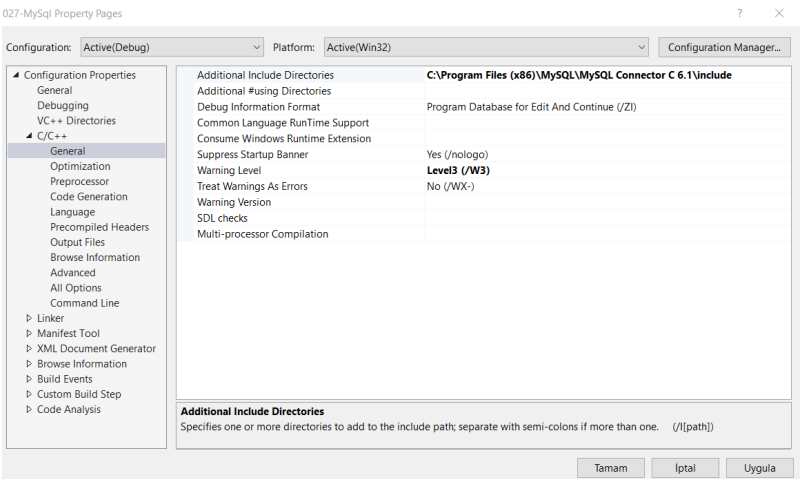
C'de MySql İle İşlemler

C'de MySql VTYS'si ile işlemler için öncelikle yime MySql'in gerekli kütüphanelerinin client tarafta kurulması gerekir. Bunun için Windows'ta, Mac OS X sistemlerinde ve Linux'ta "MySql C Connector () " denilen kurulum yapılabilir. Debian türevi (Ubuntu, Mint vs.) sistemlerde aşağıdaki apt-get komutu bu paketin indirilerek kurulmasını sağlamaktadır:

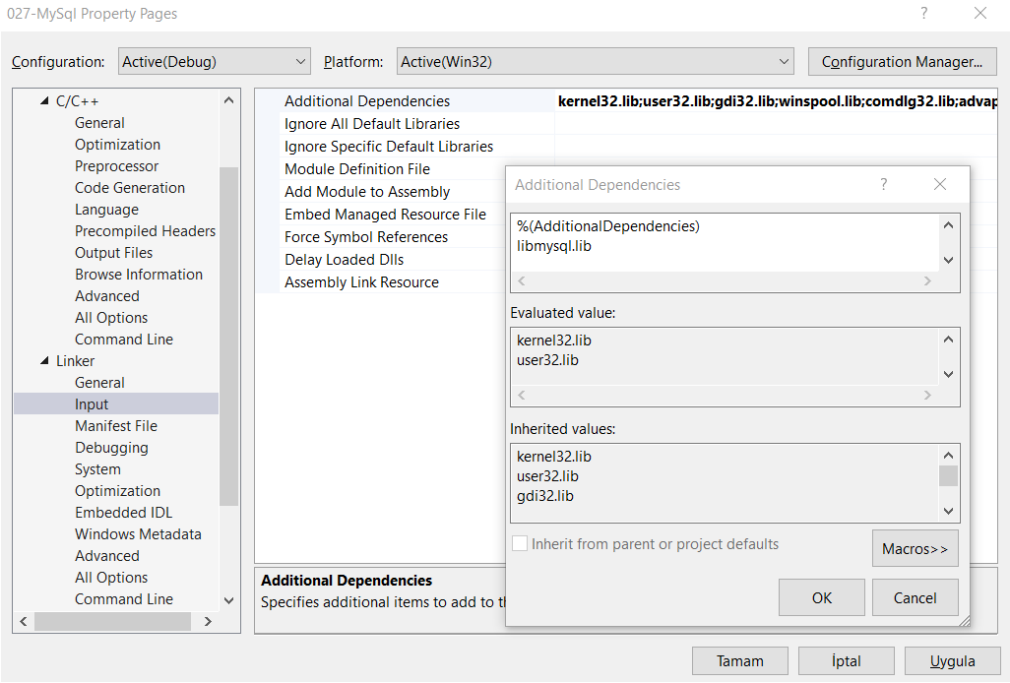
```
sudo apt-get install libmysqlclient-dev
```

Visual Studio IDE'sinde MySql çalışması için şu hazırlıklar yapılmalıdır:

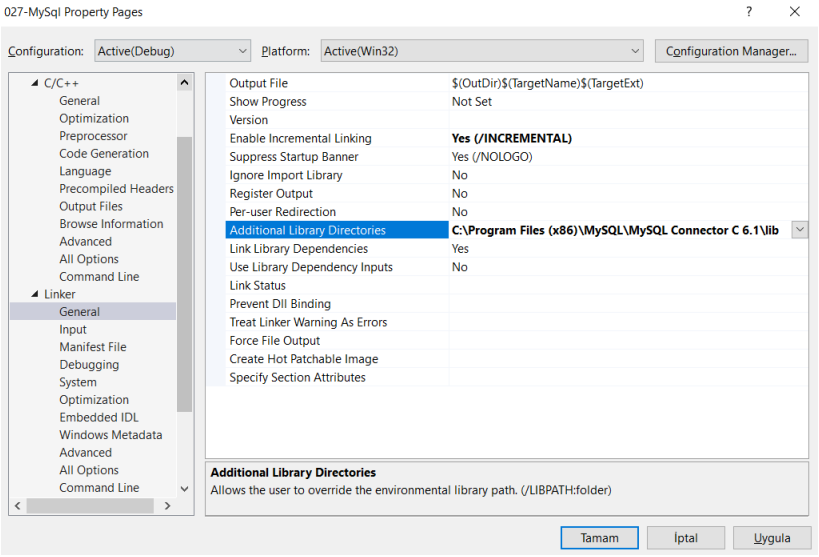
1) Bir C projesi oluşturulur.
2) MySql'in C için hazırlanmış başlık dosyaları "Connector"ün kurulduğu dizinin içerisindeki "Include" dizinindedir. Bu dizine de önişlemci aşamasında bakılmasını sağlamak gerekir. Bunun için Microsoft derleyicilerinde (cl.exe) komut satırından /I ve gcc ve clang derleyicilerinde -I seçeneği bu işlem için kullanılabilmektedir. Ayrıca Visual IDE'sinde bu belirleme görsel yolla da yapılabilir. Bunun için proje seçeneklerine gelinir. "C-C++/General/Additional Include Dirfectories" alanına diizn girilir:



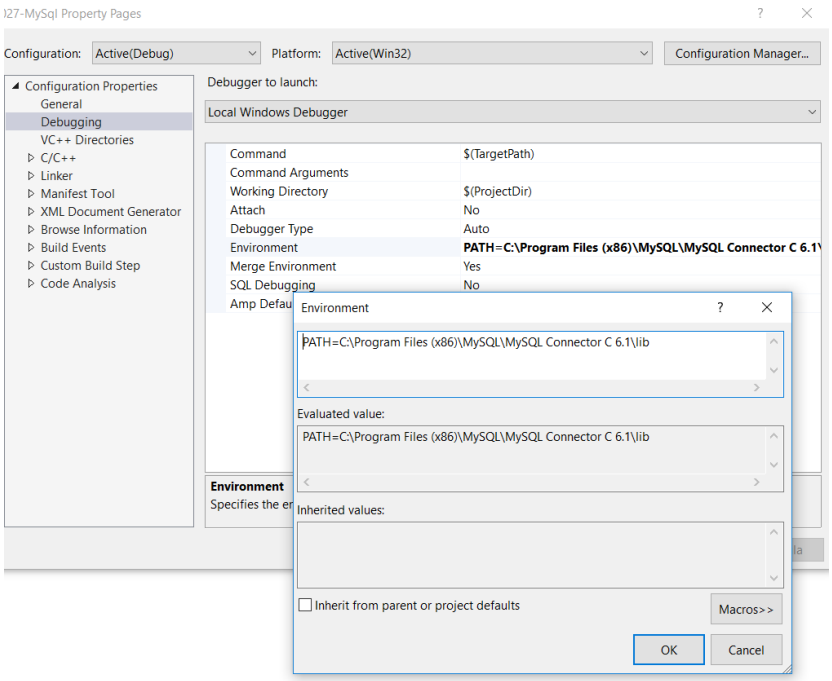
3) Şimdi sıra link aşaması için ayar yapmaya gelmiştir. Mysql C fonksiyonları "libmysql.dll" dosyası içerisindedir. Bu dosyanın import kütüphanesi de "libmysql.lib" biçiminde "connector"ün kurulu olduğu dizinin altındaki LIB dizininde bulunmaktadır. Import kütüphanesi Visual Studio'da proje seçeneklerinden "Linker/Input/Additional Dependencies" kısmına girilir.



Tabii burada yalnızca import kütüphanesinin ismi girilmektedir. Bunun yol ifadesinin "Linker/General/Additional Library Directories" kısmında belirtilmesi gerekir:



Artık projemiz build edilecektir. Tabii programı çalıştırırken "libmysql.dll" isimli dinamikj kütüphane sistem tarafından bulunamayabilir. Çünkü bilindiği gibi Windows dinamik kütüphaneleri belirli yerlerde aaramaktadır ve en sonunda PATH çevre değişkeni ile belirtilen dizinlere de tek tek bakmaktadır. Biz PATH çevre değişkenine bu kütüphanenin bulunduğu dizini de ekleyebiliriz. Ya da Visual Studio'da yalnızca çalıştırılacak proje için bunun yapılmasını sağlayabiliriz. Bu işlem proje seçeneklerinden "Debugging/Environment" kısmından yapılabilir:



Artık Visual Studio için her şey hazır durumdadır. Linux ve Mac OS X sistemlerinde zaten ilgili kütüphaneler sistemin baktığı yerlerde olduğu için herhangi bir sorun çıkmayacaktır.

Programda ilk yapılacak şey mysql_init fonksiyonu çağırarak bir handle elde etmektir:

```
MYSQL *mysql_init(MYSQL *mysql);
```

Bu fonksiyon parametre olarak bizden MYSQL türünden bir nesnenin adresini ister onun içini doldurur. Eğer parametre NULL girilirse fonksiyon bu nesneyi kendisi tahsis edip bize aresini verecektir. Fonksiyon başarısız olabilir. Başarısızlık durumunda NULL adrese geri döner.

```
#include <stdio.h>
#include <stdlib.h>
#include "mysql.h"

int main(void)
{
    MYSQL *db;

    if ((db = mysql_init(NULL)) == NULL) {
        fprintf(stderr, "mysql_init failed\n");
        exit(EXIT_FAILURE);
    }
    printf("Ok\n");

    return 0;
}
```

Şimdi mysql'in server'ına bağlanma aşamasına gelinmiştir. Bu işlem mysql_real_connect fonksiyonuyla yapılmaktadır:

```
MYSQL *mysql_real_connect(MYSQL *mysql, const char *host, const char *user, const char
*passwd, const char *db, unsigned int port, const char *unix_socket, unsigned long
client_flag);
```

Görüldüğü gibi fonksiyonun oldukça fazla parametresi vardır. Birinci parametre mysql_init fonksiyonundan alınan handle değeridir. Bu parametre NULL geçilirse mysql_init fonksiyonunu bu fonksiyon kendisi çağırıp handle değerini de kendisi alıp bize bunu bizevermektedir. Eğer bu paremetreye handle değeri verilirse

fonksiyon bize bizim verdiğimiz değerin aynısını geri verir. Tabii fonksiyon başarısız da olabilir. Bu duurmda NULL adrese geri döner. Fonksiyonun ikinci parametresi server'ın ip adresi ya da host ismidir. IP adresi girilecekse noktalı biçimde olmalıdır. Fonksiyonun üçüncü parametresi kullanıcı ismini, dördüncü parametresi parolayı belirtir. beşinci parametre kullanılacak veritabanın ismidir. Altıncı parametre server'ın port numarasını belirtir. Bu parametre sıfır geçilirse default 3306 numaralı port anlaşılır. Sonraki parametreler de sırasıyla NULL adres ve 0 biçiminde default değerlerle geçilebilir. Örneğin:

```
#include <stdio.h>
#include <stdlib.h>
#include "mysql.h"

int main(void)
{
    MYSQL *db;

    if ((db = mysql_init(NULL)) == NULL) {
        fprintf(stderr, "mysql init failed\n");
        exit(EXIT_FAILURE);
    }

    if (mysql_real_connect(db, "127.0.0.1", "root", "csd1993", "student", 0, NULL, 0) == NULL) {
        fprintf(stderr, "Error: %s\n", mysql_error(db));
        mysql_close(db);
        exit(EXIT_FAILURE);
    }
    printf("Ok\n");

    return 0;
}
```

Hata durumunda hatanın yazısını mysql_error fonksiyonuyla elde edebiliriz:

```
const char *mysql_error(MYSQL *mysql);
```

Başarısızlık durmunda yine mysql_init ile yapılan tahsisatın geri bırakılması için mysql_close fonksiyonun çağrılması gerekir.

SQL cümlesini server'a gönderip işletmek için mysql_query fonksiyonu kullanılmaktadır.

```
int mysql_query(MYSQL *mysql, const char *stmt_str);
```

Fonksiyonun birinci parametresi handle değerini ikinci paarametresi SQL komut yazısını alır. Fonksiyon başarı durumunda sıfır başarısızlık durumunda sıfır dışı bir değere geri döner. Örneğin:

```
#include <stdio.h>
#include <stdlib.h>
#include "mysql.h"

void err_exit(MYSQL *db)
{
    fprintf(stderr, "Error: %s\n", mysql_error(db));
    mysql_close(db);
    exit(EXIT_FAILURE);
}

int main(void)
{
    MYSQL *db;

    if ((db = mysql_init(NULL)) == NULL) {
        fprintf(stderr, "mysql init failed\n");
        exit(EXIT_FAILURE);
    }
```

```
    if (mysql_real_connect(db, "127.0.0.1", "root", "csd1993", "student", 0, NULL, 0) == NULL)
        err_exit(db);

    if (mysql_query(db, "INSERT INTO student_info(student_name, student_no) VALUES('Can Apaydin', 123);"))
        err_exit(db);

    printf("Ok\n");

    return 0;
}
```

Belli koşulu sağlayan kayıtların ele geçirilmesi için başka bir deyişle SELECT cümlesi ile seçilen kayıtların elde edilmesi için birkaç yol vardır. Bunun için önce SELECT cümlesi yine sql_query fonksiyonuyla uygulanır. Sonra select edilen kayıtların elde edilmesi için şu işlemler yaplır:

1) mysql_store_result fonksiyonu çağrılarak bir "result handle değeri" elde edilie:

```
MYSQL_RES *mysql_store_result(MYSQL *mysql);
```

Fonksiyon paarametre olarak bizden mysql_init ile elde edilen handle değerini alır ve bize kayıtları elde etmemiz için gereken MYSQL_RES * türünden bir handle değeri verir.

2) Kayırların tek tek ele geçirilmesi için mysql_fetch_row fonksiyonu bir döngü içerisinde çağrılır. Bu fonksiyonun prototipi şöyledir:

```
MYSQL_ROW mysql_fetch_row(MYSQL_RES *result);
```

Fonksiyon mysql_store_result fonksiyonundan elde edilen handle değerini alır ve MYSQL_ROW türüyle geri döner. Bu tür aslında char ** biçiminde typedef edilmiştir. Yani char türünden göstericileri tutan dizinin adresini belirtir. İşte bu fonksiyon NULL adres döndürene kadar döngü içerisinde ilerlenir. Arık kayıtlara ilişkin sütun bilgilerine MYSQL_ROW türünden göstericiye sütun numarası indeks yapılarak erişilir. Ancak bu türden erişim bize tüm sütunları yazı gibi vermektedir. Örneğin:

```
#include <stdio.h>
#include <stdlib.h>
#include "mysql.h"

void err_exit(MYSQL *db)
{
    fprintf(stderr, "Error: %s\n", mysql_error(db));
    mysql_close(db);
    exit(EXIT_FAILURE);
}

int main(void)
{
    MYSQL *db;
    MYSQL_RES *res;
    MYSQL_ROW row;

    if ((db = mysql_init(NULL)) == NULL) {
        fprintf(stderr, "mysql init failed\n");
        exit(EXIT_FAILURE);
    }

    if (mysql_real_connect(db, "127.0.0.1", "root", "csd1993", "student", 0, NULL, 0) == NULL)
        err_exit(db);

    if (mysql_query(db, "SELECT * FROM student_info"))
        err_exit(db);

    if ((res = mysql_store_result(db)) == NULL)
```

```
        err_exit(db);

    while ((row = mysql_fetch_row(res)) != NULL)
        printf("%s, %s, %s\n", row[0], row[1], row[2]);

    mysql_free_result(res);
    mysql_close(db);

    return 0;
}
```

Select ile edilen sütunlar sayısı mysql_num_fields fonksiyonuyla elde edilebilir:

```
#include <stdio.h>
#include <stdlib.h>
#include "mysql.h"

void err_exit(MYSQL *db)
{
    fprintf(stderr, "Error: %s\n", mysql_error(db));
    mysql_close(db);
    exit(EXIT_FAILURE);
}

int main(void)
{
    MYSQL *db;
    MYSQL_RES *res;
    MYSQL_ROW row;

    if ((db = mysql_init(NULL)) == NULL) {
        fprintf(stderr, "mysql init failed\n");
        exit(EXIT_FAILURE);
    }

    if (mysql_real_connect(db, "127.0.0.1", "root", "csd1993", "student", 0, NULL, 0) == NULL)
        err_exit(db);

    if (mysql_query(db, "SELECT student_name, student_no FROM student_info"))
        err_exit(db);

    if ((res = mysql_store_result(db)) == NULL)
        err_exit(db);


    while ((row = mysql_fetch_row(res)) != NULL) {
        int i;

        for (i = 0; i < mysql_num_fields(res); ++i) {
            if (i != 0)
                printf(", ");
            printf("%s", row[i]);
        }
        printf("\n");
    }

    mysql_free_result(res);
    mysql_close(db);

    return 0;
}
```

mysql_store_result fonksiyonuyla elde edilmiş olan handle alanı mysql_free_result fonksiyonuyla serbest bırakılmaktadır.

C ile MySQL kullanımı konusunun bazı ayrınrıları vardır. Bu ayrıntılar burada ele alınmayacaktır. İlgili dokümanlara başvurulabilir.

Karakter Tabloları ve Karakter Kodlamaları

Bilindiği gibi yazılar karakterlerden oluşmaktadır. Karakterler de gerek birinci belleklerde gerekse ikincil belleklerde ikilik sistemde sayı olarak tutulmaktadır. Hangi sayıların hangi karakterleri temsil ettiği kullanılan karakter tablosuna bağlıdır. İlk zamanlardan bu yana bilgisayar sistemlerinde pek çok katrakter tablosu kullanılmıştır. Karakter kodlamasındaki en önemli sorunlardan biri bir tabloya göre kodlanmış olan karakterin yanlışlıkla başka bir tabloya göre yorumlanmaya çalışılmasıdır.

Karakter kodlaması için kullanulan terminoloji çeşitli kaynaklarda birbirinden farklılık gösterebilmektedir. Burada öncelikle bu terminoloji üzerinde biraz duralım:

Karakter Kümesi ya da Karakter Repertuarı (Caharcted Set / Character Repertoire): Birbirindne farklı karakterlerden oluşturulmuş olan topluluğa karakter kümesi ya da karaketer repertuarı denilmektedir. Her şeyden önce karakter kodlaması için bir karakter kümesinin öncelikle belirlenmiş olması gerekir. Bazı karakter kümeleri az sayıda karakteri içerirken (örğenin Mors alfabesi) bazıları çok fazla karaktere sahip olabilmektedir (Örneğin UNICODE tablosu). Bir karakter kümesinde her karakterin bir ismi olmalıdır. İsmin yanı sıra karakterler için onların nasıl görüntülenceceğine ilişkin semboller de tanımlanır. Bu sembollere İngilizce "glyph" denilmektedir. Tabii bu semboller ("glyph"ler) font konusunuyla da ilişkilidir. Yani bir karakter birbirinden az çok farklı sembollerle ifade edilebilirler.

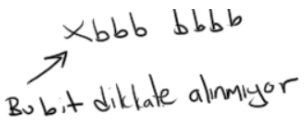
Karakter Kodu (Character Code / Code Point / Code Value): Karakter kümesi (ya da repertuarı) belirlendikten sonra tasarımcı bu kğme içeriisndeki her karaktere birbirinden farklı bir tamsayı karşılık getirir. Buna ilgili karakterin karakter kodu denilmektedir. Karakter kodu terimi İngilizce'de pek çok kaynak tarafından "code point" sözcükleriyle ya da "character code" sözcükleriyle ifade edilmektedir. Küme içeisindeki karakterlere karşı düşürülen tamsayılar sıfır orijinli ve peşi sıra olmak zorunda olmak zorunda değildir. Ancak pek çok tablolama sisteminde karakterlere karakter kodları sıfır orijinli ve ardışıl olarak atanmaktadır. Örneğin 'A' karakterinin ASCII karakter kümesindeki karakter kodu 65'tir. Bazı tablolarda karakter kodları belli bir mantığa göre sıraya dizilmişleridir. Örneğin ASCII tablosunda küçük harflere ve büyük harflere peşi sıra karakter kodu atanmıştır.

Karakter Kodlaması (Character Encoding): Artık sıra karakter kodlarının bellekte byte'larla nasıl temsil edileceğine gelmiştir. İşte buna karakter kodlaması denilmektedir. Karakter tablosundaki bir karakter kodu değişik biçimlerde bilgisayar belleğine yerleştirilebilir. İşte karakter kodlaması hangi karakter kodlarının hangi bitsel dizilimlerle (ya da byte'larla) bilgisayar ortamında saklanacağını belirtmektedir. Yani adeta karakterlerin karakter kodlarının bilgisayar belleğine aktarılması için yapılan bir dönüştürme söz konsusur.

Pekiyi nedne karakter kodları bir dönüştürmeyle ikilik sistemde ifade edilmektedir? Doprudan karakter kodunun kendisin belirttiği sayı bilgisayar belleğinde ikilik sistemde tutulamaz mı? Gerçekten de karakter kodları doğrudan bitlere ve byte'lara dönüştürülerek kodlanabilir. Yani karakter kodlaması hiç devreye sokulmayabilir. Ancak bazı durumlarda karakter kodlarının bazen sıkıştırılarak ifade edilmesi ya da başka amaçlarla değiştirilerek ifade edilmesi gerekebilmektedir. Örneğin UTF32, UTF16 ve UTF8 aslında birer karakter kodlama sistemidir. Burada aslında UNOCIDE tablodaki karakter kodları kodlanmaktadır. Fakat örneğin ASCII karakter tablosunda doğrudan karakterlerin karakter kodları birer byte olarak aynı değerlerle kodlanmaktadır.

ASCII Tablosu

ASCII (American Standard Code for Information Interchange) tablosu bilgisayar sistemleri için ilk kullanılan standart tablodur. Tablo telgraf kodları temel alınarak oluşturulmuştur. İlk çalışmalar 1960'ta başlatılmıştır. İlk versiyonu 1963'te oluşturulmuştur. Bunu 1967'ye kadar çeşitli revizyonlar izlemiştir. ASCII tablosu 128 karakterli bir tablodur. Tablodaki karakterlere 0-127 arasında karakter kodları karşılık düşürülmüştür. Bunlar da bir byte ile karakter kodlaması na sokulmuştur. Karakter kodlamasındaki byte'ın yüksek anlamlı biti dikkate alınmamaktadır:



Yüksek anlamlı bir "parity" amacıyla da kullanılabiliyordu. Tablodaki ilk 32 karakter kodu kontrol karakterlerine ayrılmıştır. Sonra onları SPACE karakteri izler. Tabloda önce büyük harfler sonra küçük harfler dizilmiştir. Bunların aralarında 6 başka karakter vardır.

EBCDIC Tablosu

EBDIC (Extended Binary Coded Decimal Interchange Code). IBM firması tarafından geliştirilmiş ve kullanılmış bir tablodur. 256 karaktere sahiptir. Dolayısıyla ASCII tablosundan daha geniştir. Tabloda önce küçük harfler sonra büyük harfler bulunmaktadır. Üstelik bu harflerin hepsinin karakter kodları tabloda peşi sıra gitmemektedir. Özellikle IBM'in eski main frame'lerinde (örneğin System 360 ve türevlerinde) bu tablo yoğun olarak kullanılmıştır.

Genişletilmiş ASCII Tabloları

ASCII tablosu Amerikalılar tarafından geliştirilmiştir. Dolayısıyla tablodaki karakterler İngilizcedeki karakterleri kapsamaktadır. Halbuki pek çok latin dilinde farklı birtakım karakterler vardır. İşte İngilizce'de olmayan bu farklı karakterlerin ifade edilebilmesi için değişik kurumlar ve şirketler ASCII tablosunu genişletmişlerdir. Böylece ortaya 8 bitlik ilk yarısı standard ASCII karakterlerden oluşan ikinci yarısı ülkelere özel karakterlerden oluşan pek çok tablo ortaya çıkmıştır. Genellikle bu tablolara "Code Page" ismi altında numaralandırılarak beelirtilmektedir. "Code Page" terimi bir tablodaki farklı yerleşimleri ifade etmek için kullanılmaktadır. Bu anlamda ASCII tablosunun genişlitelmiş pek çok "code page"leri vardır. Bunların en önemlilkerini kısaca gözden geçirelim:

ISO 8859 Tabloları: ISO pek çok genişletilmiş ASCII tablosunu 8859 kod numarasıyla standart hale getirmiştir. 8859-1 "code page"i ne Latin-1 de denilmektedir. Burada temel Avrupa dillerindeki önemli karakterler bulunmaktadır. 8859 "code page"lerinin 8859-N biçiminde değişik versiyonları vardır. 8859-9 Türkçe karakterleri içeren versiyonudur.

Microsoft'un Genişletilmiş ASCII Tabloları: Microsoft firması da ASCII tablosunu genişleterek pek çok "code page" oluşturmuştur. Örneğin Microsoft'un 1252 tablosu 8859-1 Latin "code page"ine benzemektedir. Microsoft'un farklı ülkeler için pek çok bu biçimde "code page"i vardır. Türkçe karakterler için de 1254 "code page"i kullanılmaktadır. Microsoft'un 1254 numaralı codepage'i aşağı yukarı ISO'nun 8859-9'u ile aynıdır.

DOS'un Kullandığı Genişletilmiş ASCII Tabloları: Eskiden DOS zamanlarında farklı diller için pek çok "code page" oluşturulmuştu. Bunlara genel olarak "OEM Code Page"leri denilmektedir. Bunlarında çeitli numaraları vardır. Örneğin Türkçe karakterleri içeren versiyonu 853 kod numarasayıla bilinmektedir. Buradaki Türkçe karakterlerin karakter kodlarıyla ISDO 8859-9 ve Microsoft'un 1254'ündeki Türkçe karakter kodları tamamen farklıdır. (Windows'un ayarları Türkçe ise hala eskiden olduğu gibi ALT tuşuna baslarak desimal sistemde OEM 853 kodları ile Türkçe karakterler çıkartılabilmektedir.)

Yukarıdakilerin dışında başka IBM'in EBCIDZ tabanlı ve Apple'ın eski sistemlerinde kullandığı "code page"ler vardır. Konun ayrıntıları başka dokğmanlardan izlenebilir.

Bugün Türkçe için temelde bir byte'lık ASCII tablosunun genişletilmiş versiyonu biçiminde üç seçenek söz konusudur:

ISO 8859-9
Microsoft 1254 (Windows sistemleri kullanıyor)
Microsoft OEM 853 (DOS sistemleri kullanıyordu)

Yukarıda da belirtildiği gibi Microsfot'un 1254'ündeki Türkçe karakterler ISO'nun 8859-9'u ile aynıdır.

UNICODE Tablosu

Tek byte'lık karakter tabloları çok uzun bir süre kullanılmıştır. Bugün hala kullanılmakadır. Ancak yukarıda da ele alındığı gibi tek byte'lık (256'lık) karakter tabloları alan bakımından etkin olsa da bazı sorunlar oluşturmaktadır:

- Tek byte'lık çok fazla tablo (code page) vardır. Böylece bir yazının hangi tabloya göre kodlandığının bilinmesi gerekir. Bu kadar çok tablo (code page) kafa karıştırmaktadır.

- Bir byte'lık tablolarda en fazla 256 karakter tanımlanabilmektedir. (Zaten farklı "code page"lerin ortaya çılartılmasının nedeni de aslında farklı karakter grupları için farklı tablolar oluşturma niyetidir.) Halbuki Japonca, Çince gibi dillerdeki karakterlerin sayısı 256'dan fazladır.

- Bir byte'lık karakter tablolarında Latin dillerinde bile birden çok dilin kullanıldığı yazılar oluşturulamamaktadır. Yani örneğin hem Yunanca hem Türkçe karakterler aynı yazı içerisinde bulunamamaktadır.

İşte bir byte'lık (256'lık) karakter tabloları bu sorunları doğurduğu için birden çok byte'tan oluşan tabloların tasarım süreci başlamıştır. Ne de olsa artık bellekler de bollaşmıştır. Dolayısıyla karakterlerin birden fazla byte ile tutulması artık pek çok sistemde sorun oluşturmamaktadır.

Japonca, Çince, Arapça gibi fazla sayıda farklı karakteri olan diller için geniş karakter repertuarı olan özel tablolar geliştirilmiştir. Örneğin IBM'in 932 numaralı "code page"i Japonca karakterleri barınndırmaktadır. Buradaki karakterler bir byte ya da iki byte ile kodlanmaktadır. Microsoft'un da Japonca için benzer biçimde 943 numaralı "code page"i vardır.

İşte tüm bu karışıklıkları gidermek için tüm dillerin karakterlerini içeren tek bir tablonun tasarım süreci başlamıştır. Bu tabloya UNICODE denilmektedir. UNICODE tablo ISO tarafından da 10646 koduyla (ISO/IEC 10646) standardize edilmiştir. UNICODE konsorsiyumunun standardıyla ISO 10646 uyumludur. Ancak UNIOCDE konsorsiyumunun standarları çok daha geniş birtakım belirlemelere sahiptir. ISO'nun bu standardına kısaca UCS (Universal Coded Character Set) denilmektedir.

Unicode standartları ilk kez 1991 yılında oluşturulmuştur. Sonra bunu çeşitli versiyonları izlemiştir. Bugün için son versiyon UNICODE 9.0'dır. UNICODE ilk çıktığında karakter sayısı 65536 ile sınırlıydı. Dolayıyla karakterlere iki byte içerisine sığabilen karakter kodları (code points) verilebiliyordu. Ancak daha sonra tablodaki karakter sayısı artırılmıştır. Bugün her biri 65536'lık gruptan oluşan 17 tane düzlem (plane) vardır. Böylece tablodaki toplam karakter sayısı 17 * 65536 = 1114112 tanedir. Tablonun ilk düzlemine (yani 0-65536 arasındaki karakterlerine) "Temel Çokdilli Düzlem (Basic Multilingual Plane) denilmektedir. Bu düzlemde tüm doğal dillerin karakterleri ve temel semboller bulunmaktadır. (Zaten UNICODE ilk çıktığında yalnızca bu düzlem vardı.) UNICODE tablodaki ilk 128 karakter ASCII tablosunun aynısıdır. İkinci 128 karakter ise ISO

8859-1 (Latin 1)'deki karakterlerin aynısıdır. Başka bir deyişle UNICODE tablonun ilk 256 karakteri ISO 8859-1 tablosunun aynısıdır.

UNICODE tablonun bazı düzlemlerinin bazı bölgelesi boştur (reserved) bazoı düzlemlerin de tamamı boştur. Ayrıca son iki düzlem tamamen "kullanıcı tanımlı (user defined)" olarak bırakılmıştır. Yani kişiler isterlerse kendi uydurdukları karakterleri bu alana yerleştirebilmektedirler.

UNICODE karakterlerin karakter kodları (code points) yani sayısal değerleri geleneksel olarak U+HHHHHH biçiminde (buradaki HHHHHH hex sistemdeki rakamlardır) belirtilmektedir. Örneğin U+61 karakteri 'a' karakteridir.

UNICODE tablodaki karakterleri byte'larla ifade etmek için üç tür karakter kodlaması (character encoding) bulunmaktadır: UTF8, UTF16 ve UTF32. UTF öneki İngilizce "Unicode Transformation Format" sözcüklerinden kısaltılmıştır. Madem ki UNICODE tabloda toplam 1114112 karakter vardır. O halde onların hepsini temsil edebilecek byte miktarı 3 byte olabilir. Ancak 3 byte ikinin kuvveti olmadığı için iyi bir uzunluk değildir. Bu nedenle her bir UNICODE karakter doğal olarak 4 byte'la ifade edilebilir. İşte bu karakter kodlamasına UTF32 denilmektedir. UTF16 kodlamasında ilk düzlemdeki karakterler (yani 0 ile 65535 arasındaki karakterler ki bunlar zaten en temel çokdilli karakterlerlerdir) 2 byte ile diğerleri 4 byte ile ifade edilmektedir. UTF8'de ise karakterler 1, 2, 3, 4 byte uzunluğunda olabilmektedir. Bu kodlamaya genel olarak "multibyte kodlama" denilmektedir. Şimfi bu formatları yakından inceleyelim.

UTF-32 Kodlaması

Bu kodlama aslında oldukça basittir. 1114112 karakterin her biri 4 byte ile kodlanır. Başka bir deyişle karakterlerin kod numaraları doğurdan 4 byte'lık bir sayı biçiminde ifade edilmektedir. Kodlama "Little Endian" ya da "Big Endian" olarak yapılabilir. Kodlamanın nasıl yapılmış olduğu metin dosyalarında ya da akımlarda (stream) "BOM (Byte Order Mark)" karakterinde belirtilebilmektedir. Ancak BOM karakteri zorunlu değildir. UTF32 her karakterin eşit byte uzunluğuna sahip olduğu (fixed length) bir kodlama biçimidir. Ancak bu kodlama fazla yer kaplama eğilimindedir. Bu nedenle pek fazla tercih edilmemektedir.

UTF-16 Kodlaması

Yukarıda da belirtildiği gibi bu kodlama biçiminde BMP düzlemindeki karakterler iki byte'la diğerleri 4 byte'la kodlanmaktadır. Kodalama biçiminin ayrıntıları şöyledir:

- UNICODE tablonun BMP düzleminde (ilk 64K'lık bölümünde) U+D800'dan U+DFFF'ye kadar olan alan zaten kullanılmamaktadır (reserved). Kodlamada bu bölge karakterin 2 byte mı yoksa 4 byte mı olduğunu belirlemekte kullanılır. UTF-16 kodlamasındaki 2 byte'lar da "Little Endian" ya da "Big Endian" olarak kodlanabilmektedir.

- Eğer karakter [U+0000, U+D7FF] ya da [U+E000, U+FFFF] arasındaysa doğrudan bu karakterin karakter kodu (code point) 2 byte olarak kodlanır. Pek çok dildeki karakterlerin hepsi bu aralıktadır. Örneğin "Ağrı dağı çok yüksek" gibi Türkçe bir yazının tüm karakterleri bu aralıkta olduğu için bu yazı her bir karakteri 2 byte ile "Little Endian" formatta aşağıdaki gibi kodlanabilir:

```
00000000  41 00 1F 01 72 00 31 01  20 00 64 00 61 00 1F 01  A...r.1. .d.a...
00000010  31 01 20 00 E7 00 6F 00  6B 00 20 00 79 00 FC 00  1. ...o.k. .y...
00000020  6B 00 73 00 65 00 6B 00                           k.s.e.k.
```

- Eğer karakter [U+10000, U+10FFFF] alanındaysa (yani ilk 64K'lık alanın dışındaysa önce bu değerden 65536 (0x10000) çıkartılır. Buradan elde edilen 20 bit iki tane 10'luk kısma ayrılır. Yüksek anlamlı 10 bitlik kısma 0xD800 toplanır bu ilk 2 byte'ı oluşturmaktaır. Düşük anlamlı 10 bitlik kısma da 0xDC00 toplanarak bu da ikinci 2 byte'ı oluşturur. Wikipedia.com'daki şu örnek verilebilir: Kodlanacak karakter U+10437 olsun. Bu karakter ilk 64K'lık düzlem içerisinde olmadığı için önce bundan 0x10000 (yani 65536 değeri) çıkartılır.

Burandan 0x00437 (0000 0000 0100 0011 0111) elde edilir. Bu 20 bitlik değer iki ayrı 10 bite ayrılır: 0000 0000 01 : 00 0011 0111. Burada yüksek anlamlı 10 bit 0xD800 ile düşük anlamlı 10 bit ise 0xDC00 ile toplanır ve sırasıyla 0xD801, 0xDC37 2 byte'ları elde edilir. Bu durumda karakterin "Little Endian" kodlaması şöyle gözükecektir: 01 D8 37 DC.

Pekiyi UTF-16 kodlanmış (örneğin "Little Endian") bir karakterin karakter kodunu (code point) nasıl ederiz? Aslında tamamen ters işlem yapılmaktadır. Şöyle ki: Önce 2 byte değer çekilir. Bu değerin [U+D800, U+DFFF] arasında olup olmadığına bakılır. Eğer değer bu aralıkta değilse normal olarak bu değer zaten karakterin UNICODE karakter kodunu belirtir. Eğer değer bu aralıktaysa ters işlem yapılmalıdır. Yani ilk 2 byte'tan 0xD800, ikinci byte'tan 0xDC00 çıkartılır. Yeniden 10'ar bit oluşturulup 20 bit elde edilir. Buna da 0x10000 toplanır.

UTF-8 Kodlaması

Bu kodlama geçmişe doğru uyumlu lacak biçimde birkaç evrim geçirmiştir. Kodlamanın genel biçimi aşağıdaki özet şekille anlaşılabilir:

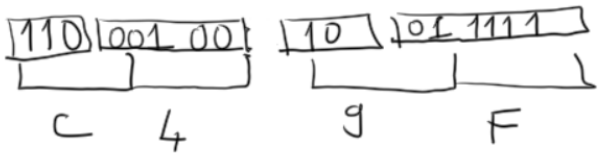
UTF-8 (2003)

Number of bytes	Bits for code point	First code point	Last code point	Byte 1	Byte 2	Byte 3	Byte 4
1	7	U+0000	U+007F	0xxxxxxx			
2	11	U+0080	U+07FF	110xxxxx	10xxxxxx		
3	16	U+0800	U+FFFF	1110xxxx	10xxxxxx	10xxxxxx	
4	21	U+10000	U+10FFFF	11110xxx	10xxxxxx	10xxxxxx	10xxxxxx

Bu tablo ttüm UTF-8 kodlamasını aslında tek başına açıklamaktadır. Şöyle ki:

- Eğer ilgili karakterin numarası [U+0000, U+007F] arasındaysa bu tek byte'la 0xxxxxxx biçiminde kodlanacaktır. Bu durumda bu byte'ın yüksek anlamlı biti 0 ve düşük anlamlı 7 biti karakterin kod numarasından oluşturulacaktır. Böylece standary ASCII karakterlerinin hepsi 1 byte ile kodlanabilmiş olacaktır.

- Eğer ilgili karakterin numarası [U+0080, U+07FF] arasındaysa (yani 11 bit alana sığabiliyorsa) bu durumda bu karakter 2 bye ile kodlanacaktır. Şöyle ki: Bu 2 byte'ın ilk byte'nın ilk byte'ı 110xxxxx ikinci byte'ı da 10xxxxxx biçiminde olacaktır. Buradaki x'ler on bir bitin bitlerini belirmektedir. Örneğin Türkçe 'ğ' karakteri UNICODE U+011F'tir. Buradaki 0x11F söz konusu bu aralığa düşmektedir. Bu 0x11F on bir bit olarak 011 0001 1111 biçiminde kodlanır. Sonra bu aşağıdaki gibi byte'lara dönüştürülür:

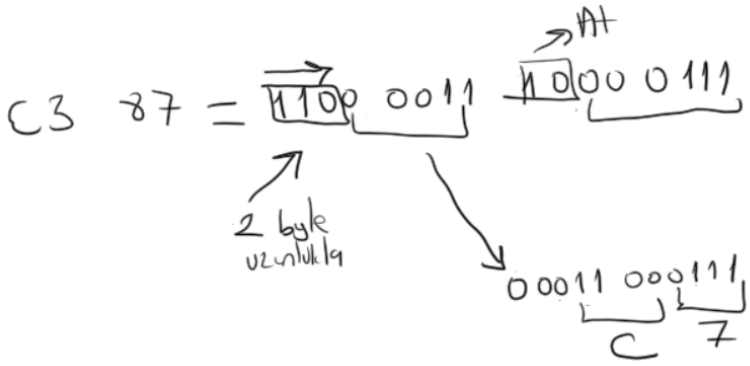


- Eğer ilgili karakterin numarası [U+0800, U+FFFF] arasındaysa (yani 16 bit alana sığıyorsa)bu durumda karakter 3 byte ile 1110xxxx 10xxxxxx 10xxxxxx biçiminde kodlanır.

- Eğer ilgili karakterin numarası [U+10000, U+10FFFF] arasındaysa (yani 21 bit alana sığıyorsa)bu durumda karakter 4 byte ile 11110xxx 10xxxxxx 10xxxxxx 10xxxxxx biçiminde kodlanır.

UTF-8'den ters dönüşüm nasıl yapılmaktadır? Örneğin Türkçe 'Ç' karakteri C3 87 byte'ları biçiminde UTF-8 olarak kodlanmaktadır. Bu karakterin kod numarası nedir? Geri dönüştürme algoritması çok kolaydır. İlk

byte'ın yğksek anlamlı bitlerinden düşük anlamlı bitlerine doğru ilk 0 görülene kadar ilerlenir. Bu ilk 0 dikkate alınmaz. Buradaki 1'lerin sayısı karakterin kaç byte'tan oluştuğunu bize vermektedir. Sonra ters işlem yapılarak karakteri oluşturan bitler elde edilir. Örneğin:



UNICODE UTF Kodlamalarında BOM (Byte Order Mark) Alanı

Elimizde salt metindne oluşan bir yazı olduğunu düşünelim. Biz bunun nasıl kodlandığını nereden anlayabiliriz (Yani hangi karakter tablosunun ve kodlamasının kullanıldığını)? Bunun standart bir yöntemi yoktur. Hiçbir metin dosyası bunu belirten özel bir alana (örneğin header) sahip olmak zorunda değildir. Tabii bazı tahminlerde bulunulabilir. Buna ilişkin sezgisel yöntem kullanan sistemler vardır. Örneğin özellikle bazı dillerde kullanılan karakterler görüldüğü zaman onun bir byte'lık ilgili dile ilişkin bir "code page" olduğu düşünülebilir. Ancak UNICODE kodlamada isteğe bağlı olarak karakter kodlaması dosyanın ya da akımın (stream) ilk karakterinde kodlanabilmektedir. Bu karaktere BOM (Byte Order Mark) denilmektedir. BOM U+FEFF numaralı UNICODE karakterdir. İşte bu karakter nasıl kodlanmışsa dosyanın kodlama biçimi de odur. Şöyle ki:

- U+FEFF karakteri UTF-8 olarak EF BB BF biçiminde kodlanır. O halde dosyanın ilk üç byte'ı bu biçimdeyse dosya içerisindeki yazının UTF-8 olarak kodlandığı anlaşılır.

- U+FEFF karakteri Big Endian UTF-16'da FE FF olarak kodlanır. O halde dosyanın ilk iki byte'ı FE FF ise bunun UTF-16 Big Endian olduğu anlaşılmalıdır.

- U+FEFF karakteri Little Endian olarak UTF-16'da FF FE olarak kodlanır. O halde dosyanın ilk iki byte'ı FF FE ise bunun UTF-16 Little Endian olduğu anlaşılmalıdır.

- U+FEFF karakteri Big Endian UTF32'de 00 00 FE FF biçiminde kodlanır. O halde dosyanın ilk dört byte'ı 00 00 FE FF ise bunun UTF-32 Big Endian olduğu anlaşılmalıdır.

- U+FEFF karakteri Little Endian UTF32'de FF FE 00 00 biçiminde kodlanır. O halde dosyanın ilk dört byte'ı FF FE 00 00 ise bunun UTF-32 Little Endian olduğu anlaşılmalıdır.

Karakter Kodlamasına İlişkin Sorunlar

Karakter kodlamalarına ilişkin sorunların özünü karakterleri kodlayan taraf ile bunu yorumlayan tarafın aynı karakter tablosunu veya aynı karakter kodlamasını kullanmıyor olması oluşturmaktadır. Gerçekten de karakterlerin iletildiği sistemlerde "karakterleri kodlayan" ve "onları yorumlayan" iki ayrı taraf vardır. Bunların arasında bir uyumun olması gerekmektedir.

Örneğin Windows'ta Visual Studio IDE'sini kullanarak bir C programı yazdığımızda Türkçe karakterlerin uygun birr biçimde görüntülenemediğini görebiliriz. Burada yine iki taraf söz konusudur: Kodlayan taraf ve yorumlayan taraf. Kolayan taraf Visual Studio IDE'sinin C editörüdür. Kodlama işleminin C ya da C++ Programlama Dilleri ile bir ilgisi yoktur. Yorumlayan taraf ise stdout dosyasına ilişkin aygıt sürücüsüdür. Visual Studio C Editörünün default kullandığı karakter tablosu 1 byte'lık Windows 1254 Code Page'cidir (Anımsanacağı gibi bu code page ISO 8859-9 ile uyumludur). Halbuki Consle için stdout aygıt sürücüsünün kullandığı karakter tablosu DOS (OEM) 857'dir. İşte bu uyuşmazlık Türkçe karakterlerin düzgün gözükmesini engellemektedir. Pekiyi bu durumda ne yapmak gerekir? Ya Visual Studio C editörünün code page'ini DOS (OEM) 857'ye çekmek ya da stdout augıt sürücüsünün code page'ini Windows 1254'e çekmek.

Windows'ta console ekranına ilişkin stdout aygıt sürücüsünün code page'i SetConsoleOutputCP ile değiştirilip, GetConsoleOutputCP ile alınabilir.

```
BOOL WINAPI SetConsoleOutputCP( _In_ UINT wCodePageID);
UINT WINAPI GetConsoleOutputCP(void);
```

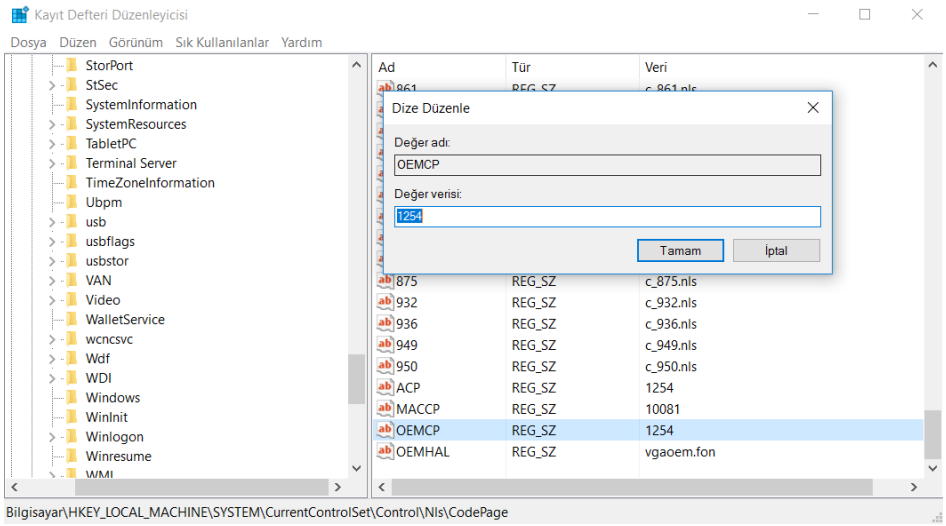
O halde biz örneğin console'a ilişkin stdout aygıt sürücüsünün code page'ini editör ile uyumlu yaparak Türkçe karakterlerin düzgün gözükmesini sağlayabiliriz:

```
#include <stdio.h>
#include <Windows.h>

int main(void)
{
    SetConsoleOutputCP(1254);

    printf("Ağrı dağı çok yüksek\n");

    return 0;
}
```

Ya da bu işlemin tersini yapabiliriz. Yani Visual Studio Editörü ile C dosyasını save ederken 857 code page'ini seçebiliriz. Diğer bir yol da console'un code page'ini değiştirmektir. Komut satırında "chcp" komutu console'un code page'ini alıp set etmekte kullanılabilir. Ancak bu komut proses temelinde bunu yaptığı için bu işlemden yalnızca bu işlemin yapıldığı console ekranı etkilenir. Örneğin programımızı 1254 code page'i ile yazıp derlediğimizi düşünelim:

```
VS2015 x86 Native Tools Command Prompt - Kopya

d:\Dropbox\Kurslar\SysProg2-2016\Src\028-EncodingProblems\Debug>chcp 1254
Active code page: 1254

d:\Dropbox\Kurslar\SysProg2-2016\Src\028-EncodingProblems\Debug>028-EncodingProblems
Ağrı dağı çok yüksek

d:\Dropbox\Kurslar\SysProg2-2016\Src\028-EncodingProblems\Debug>
```

Pekiyi console ekranının code page'i 857 iken kalıcı olarak nasıl windows 1254'e ya da UTF-8'e (65001) çekilebilir? İşte bu bilgi "HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Nls\CodePage\OEMCP" registry ayarlarında tutulmaktadır. "Regedit" programı ile bu anahtardaki değer değiştirilebilir:



Benzer biçimde biz ayarlamaları UNICODE UTF8 için de yapabiliriz.

C Programlama Dilinin standartlarında kaynak programın kodlaması konusunda ne söylenmiştir? C90 standartlarında kaynak kodun hangi karakter tablosu ile kodlanacağı belirtilmemiştir. Ancak temel karakterlerin 0 ile 127 arasında karakter kodlarına sahip olması gerektiği söylenmiştir. Benzer biçimde değişkenler de temel karakterler kullanılarak isimlendirilmek zorundadır. Böylece C90'da farklı karakter tabloları kullanılabilir ve ilk 127 karakterden sonraki karakterler herhangi bir tablo ile kodlanabilir. Şüphesiz dil içerisinde bu tür özel karakterler yalnızca tek tırnak ya da çift tırnak içerisinde geçebilir. C99 ve C11'de ve C++'ta değişken isimlendirmede UNICODE karakterler \uhhhh ya da \Uhhhhhhhh biçiminde kullanılabilir. C/C++ editörleri bunları ilgili karakter görüntüsüyle gösterebilir. Ve ilgili karakterleri de kaynak dosyaya bu biçimde kodlayabilir. Ayrıca C99, C11 ve C++ standartları değişken isimlendirmesinde derleyiciyi yazanların uygun göreceği başka karakterlerin de doğrudan kullanılabileceğini belirtmiştir. Tabii bu tür derleyiciye bağlı karakterlerin kaynak kod içerisinde kullanılması taşınabilirlik sorunlarına yol açabilmektedir.

Pekiyi Windows'ta stdin dosyasına ilişkin aygıt sürücünün default karakter kodlaması nedir? İşte giriş ile çıkış arasında uyumu korumak için Windows console'a ilişkin stdout code page'i ile stdin code page'ini default durumda aynı ayarlamaktadır. Böylece örneğin eğer console'un code page'i 857 iken aşağıdaki programı çalıştırmış olalım:

```
#include <stdio.h>

int main(void)
{
    int ch;

    ch = getchar();
    printf("%d, %c\n", ch, ch);

    return 0;
}
```

Ekranda şunları görürüz:

```
ş
159, ş
```

Console'a ilişkin stdin dosyasının code page'i GetConsoleCP API fonksiyonu ile alınabilir ve SetConsoleCP fonksiyonuyla set edilebilir:

```
UINT WINAPI GetConsoleCP(void);
BOOL WINAPI SetConsoleCP(_In_ UINT wCodePageID);
```

Anahtar Notlar: Komut satırındaki chcp komutu o andaki console ekranının hem stdout dosyasına hem de stdin dosyasına ilişkin code page'i değiştirir.

Ancak Windows'ta eğer stdin dosyasının code page'i UNICODE UTF-8'e (65001) getirilirse bu durumda Türkçe karakterler console'dan okunamamaktadır.

Linux dağıtımlarında temel kodlama biçimi UNICODE UTF-8'dir. Bu sistemlerde console (terminal) ekranlarına ilişkin stdout aygıt sürücüsü default olarak UTF-8 kodlamasını kullanmaktadır. Benzer biçimde console (terminal) ekranlarına ilişkin stdin aygıt sürücüsü de yine UNICODE UTF-8 kodlamasını kullanır. Tabii bu durumda C programlarında Türkçe karakterlerin ekranda gösterilmesinde sorun oluşmaması için kaynak dosyanın editörde UTF-8 kodlamasıyla oluşturulup saklanması gerekir. Linux dağıtımlarındaki bazı konfigürasyon dosyalarında console'un default kodlama biçimi manuel olarak değitirilebilmektedir. Ayrıca pek çok dağıtımda console penceresine geçildiğinde "Terminal" menüsünün altındaki "Set Character Encoding" kullanılarak o anda console'un karakter kodlaması değiştirilebilmektedir.

Linux sistemlerinde console UTF-8 kodlamasını kullanan console ekranlarının stdin dosyasından okuma yaparken ASCII dışı karakterler için okuma sırasında birden fazla byte verilmektedir. Böylece örneğin console UTF-8 kodlamasına sahipse biz tek bir getchar fonksiyonuyla basılan Türkçe karakteri alamayız. Her getchar bize UT-8 kodlamasına sahip karakterin sıradaki byte'ını verir. Ancak gets gibi fgets gibi fonksiyonlar eğer console'un (terminalin) karakter kodlaması UTF-8 ise bize her karakteri farklı uzunlukta olabilen (multi byte) bir dizilim verirler. Örneğin böyle bir durumda aşağıdaki programı derleyerek UTF-8 kodlamasına sahip console ekranında test edelim:

```
#include <stdio.h>

int main(void)
{
    unsigned char s[100];
    int i;

    gets(s);
    for (i = 0; s[i] != '\0'; ++i)
        printf("%02X ", s[i]);
    printf("\n");

    return 0;
}
```

```
csd@csd-VirtualBox ~/Desktop $ ./test
ağrı dağı
61 C4 9F 72 C4 B1 20 64 61 C4 9F C4 B1
```

C ve C++'ta Multibyte ve Wide Karakter Kullanımı

Bilindiği gibi C ve C++'ta char türü "byte" anlamında kullanılmıştır. Bu dillerin standartlarına göre sizeof(char) her zaman 1 vermek zorundadır. Ancak byte'ın (yani char türünün) kaç bitten oluşacağı derleyiciyi yazanların (dolasıyla söz konusu sistemi tasarlayanların) isteğine bırakılmıştır. Böylelikle C ve C++'taki char türü ASCII gibi bir byte'lık tablolama sistemlerindeki karakterleri tutabilirken UNICODE gibi birden fazla byte ile kodlanabilen karakterleri tutamamaktadır.

C standartlarında (C90 ve C99) bir byte'tan uzun olan karakterlerin tutulması ismine wchar_t denilen ayrı bir tür de tanımlanmıştır. Aslında wchar_t (isimlendrmesinden de anlaşılacağı gibi) bir anahtar sözcük değildir. Bir typedef ismidir. Pek çok sistemde bu tür unsigned short olarak (yani 2 byte olarak) bildirilmiştir. wchar_t türünün hangi tablolama sistemine göre karakter tutacağı konusunda standartlarda bir belirleme yapılmamıştır. Ancak derleyiciler bu türü tipik olarak UNICODE karakterler için kullanmaktadırlar. Tabii buradakai kodlama

biçimi yine standartlarda belirtilmemiştir. Tipik olarak endianlık neyse ona uygun biçimde UNICODE karakter kodu bu türden nesnelerin içerisine yerleştirilmektedir. Burada C standartlarında wchar_t türünün UNICODE kod için değil genel bir geniş karakter "wide character" gereksinimi için oluşturulduğunu vurgulayalım. C++'ta ise wchar_t bir anahtar sözcüktür. Her iki dilde de geniş karakter sabitleri tek tırnağın ya da iki tırnağın soluna bitişik olarak L harfi ile temsil edilmektedir. Örneğin:

```
wchar_t name[] = L"Kaan Aslan";
wchar_t *city = L"Eskişehir";
```

C99 ile birlikte resmi olarak C standartlarına geniş karakterler üzerinde işlem yapan prototipleri <wchar.h> içerisinde bulunan standart C fonksiyonları da eklendi. Bu fonksiyonlar isimlendirmelerinde 'w' harfi ve "str" yerine de "wcs" kullanılmaktadır. Bunların bazılarının isimleri aşağıda verilmektedir:

```
fwprintf, fwscanf, swprintf, swscanf, wprintf, wscanf, fgetwc, fgetws, fputwc,
getwchar, putwchar, wcscpy, wcsncpy, wcscat, wcscmp, wcsncmp
```

Pekiyi biz geniş karakterli bir string'i ya da tek bir karakteri nasıl oluştururuz. Aslında burada genellikle şöyle bir yol izlenmektedir: C ya da C++'ta kodu yazdığımız editörde bir ilgili karakterleri bir biçimde kodlarız. Örneğin eğer Türkçe karakterler söz konusu ise bu editörün default code page'ine göre kodlanır. Sonra derleme aşamasına gelindiğinde C ya da C++ derleyicileri bu geniş karakteri gördüğünde buradaki karakterleri programcının hangi code page ile kodladığına göre ona uygun UTF-16 karakter kodlarını kullanırlar. Örneğin Windows'ta 1254 code page'ine uygun olarak aşağıdaki kaynak dosya oluşturulmuş olsun:

```
#include <stdio.h>
#include <wchar.h>

int main(void)
{
    wchar_t *name = L"Ağrı Dağı";
    int i;

    for (i = 0; i < name[i] != '\0'; ++i)
        printf("%04X\n", name[i]);

    return 0;
}
```

Burada aslında Türkçe karakterler kaynak kodda bir byte ile kodlanmıştır. Ancak derleyici default code page'in 1254 olduğunu bildiği için buradaki Türkçe karakterleri UTF16 Little Endian olarak kodlamaktadır. Programın çıktısı şöyle olacaktır:

```
0041
011F
0072
0131
0020
0044
0061
011F
0131
Press any key to continue . . .
```

Burada görüldüğü gibi Türkçe karakterler UTF-16 Little Endian olarak kodlanmıştır.

Burada Window için önemli bir noktayı belirtelim. Bu yazı wprintf ile basıldığında console ekranında düzgün görüntülenmeyecektir.

```c
#include <stdio.h>
#include <wchar.h>

int main(void)
{
    wchar_t *name = L"Ağrı Dağı";

    wprintf(L"%s\n", name);

    return 0;
}
```

```
A?r? Da??
Press any key to continue . . .
```

Console'un default kullandığı karakter tablosunun Windows (OEM) 857 olduğunu anımsayınız. Maalesef Windows'ta SetConsoleOutputCP ile console'u UTF-16'ya doğal kodlu prosesler (native application) geçirememektedir. (.NET "managed" uygulamalarda bu API gereken işlemi yapmaktadır.) Ancak Windows'un GUI alt sistemi çekirdek seviyesinde UNOCODE UTF-16 kodlamasını kullanmaktadır. Böylece bu karakterler GUI fonksiyonlarında sorunsuz görüntülemeye yol açacaklardır. Örneğin:

```c
#include <stdio.h>
#include <wchar.h>
#include <windows.h>

int main(void)
{
    wchar_t *name = L"Ağrı Dağı";

    MessageBoxW(NULL, name, L"Geniş Karakter Testi", MB_OK);

    return 0;
}
```

Özet olarak C ve C++'ta geniş karakter (wide character) kullanımına ilişkin anahtar noktalar şunlardır:

- wchar_t türünün hangi karakter tablosuna ilişkin karakterleri tutacağı standartlarda derleyici yazanların isteğine bırakılmıştır (implementation dependent). Ancak bu tür genel olarak derleyicilerde UNICODE UTF-16 karakterler kodlarını tutar.

- Geniş karakter sabitleri ve string'ler L öneki ile belirtilirler (Örneğin L'A' ya da L"Ankara" gibi).

- C99 ile birlikte C'ye (C++1 ile birlikte de C++'a) geniş karakterler üzerinde işlem yapan yeni fonksiyonlar eklenmiştir.

C11 ile ve C++11 ile birlikte C ve C++'ta artık daha net tanımlanmış UNICODE karakter kavramı getirilmiştir. Buna göre özet olarak şunlar söylenebilir:

- Eğer bir string'in başına ona yapışık olarak u8 getirilirse bu UTF-8 string'i anlamına gelir. Dolayısıyla derleyici bunu UTF-8 multibyte olarak kodlaayacaktır. u8 stringleri olarak char * türündedir. u8 öneki teek tırnağın önüne getirilemez. Örneğin:

```
#include <stdio.h>
#include <windows.h>

int main(void)
{
    char *name = u8"Ağrı Dağı";

    SetConsoleOutputCP(65001);

    printf("%s\n", name);

    return 0;
}
```

Bu örnekte örnekteki kaynak dosya Visual Studio editöründe hazırlanmıştır. Bu editörün de karakter kodlaması Windows 1254'tür. Ancak derleyici bunu temel alarak karakterleri yerleştirirken UTF-8 dönüştürmesi yapmıştır.

- Eğer bir stirngin önüne yapışık olarak "u" getirilirse bu durum UTF-16 kodlaması anlamına gelmektedir. (Yani adeta "L" öneki ile aynı durum olulmaktadır.) Böyle string'ler char16_t türündendir. (Bu da genellikle unsigned short olarak typedef edilmiştir)

- Eğer bir stirngin önüne yapışık olarak "U" getirilirse bu durum UTF-32 kodlaması anlamına gelmektedir. Böyle string'ler char32_t türündendir. (Bu da genellikle unsigned int olarak typedef edilmiştir)

C'de Fonksiyon Çağırma Biçimleri (Calling Convetions)

Fonksiyonların çağrılması ve geri dönüş değerlerinin alınması konusundaki belirlemelere "çağırma biçimi (calling convention)" denilmektedir. Fonksiyon çağırma biçimleri şu konulardaki belirlemeleri içerir:

1) Çağıran fonksiyon ile çağrılan fonksiyon arasında parametre aktarımı nasıl yapılacaktır?

2) Çağrılan fonksiyonun geri dönüş değeri çağıran fonksiyona nasıl aktarılacaktır?

3) Çağrılan fonksiyon hangi yazmaçları bozma hakkına sahiptir, hangi yazmaçları korumak zorundadır?

Çağırma biçimi C standartlarında olan bir konu değildir. Çünkü C standartları böylesi aşağı seviyeli belirlemeleri derleyicilere bırakmıştır. Dolayısıyla çağırma biçimlerini oluşturmak için gereken anahtar sözcükler de derleyicilerde bir eklenti (extension) biçiminde bulunurlar. Çağırma biçimlerine ilişkin anahtar sözcükler genel olarak tür belirten sözcük ile dekleratörün arasına yerleştirilmektedir. Örneğin:

```
void __cdecl foo(int a, int b)
{
    ...
}
```

Microsoft derleyicilerinde çağırma biçimleri yukarıdaki örnekte olduğu gibi iki alt tire ile başlayan anahtar sözcüklerle temsil edilmektedir. gcc derleyicilerinde ise "fonksiyon özellikleri (function attributes)" biçimindeki bir sentksla temsil edilir. Örneğin:

```
void __attribute__((cdecl)) foo(int a, int b)
{
   ...
}
```

Şimdi Microsoft ve gcc derleyicilerindeki çağırma biçimlerini tek tek ele alacağız. Ancak burada bir noktayı vurgulamak istiyoruz: Gerek Microsoft gerekse gcc derleyici ailelerinde 32 bit uygulamalardaki çağırma biçimleriyle 64 bit uygulamalardaki çağırma biçimleri tamamen farklıdır.

Intel Ailesinde 32 Bit Programlarda Kullanılan Çağırma Biçimleri

Intel ailesinin kullanıldığı 32 bit sistemlerdeki çağırma biçimleri pek çok derleyici tarafından desteklenmektedir. Burada bunlar tek tek ele alınacaktır.

cdecl (C Declaration) Çağırma Biçimi

Bu çağırma biçimi Microsoft ve gcc derleyicilerinde C Programlama Dili için default durumdur. (Yani bu derleyicilerde fonksiyon bildirimlerinde çağırma biçimi hiç belirtilmezse sanki bu çağırma biçimi belirtilmiş gibi işlem işlem yapılır.) Her iki derleyici ailesinde de C++'taki global ve static üye fonksiyonlar için de yine default olarak bu çağırma biçimi kullanılmaktadır. Ancak sınıfların static olmayan üye fonksiyonları için Microsoft derleyicilerindeki default çağırma biçimi "thiscall" iken gcc derleyicilerindeki cdecl biçimindedir.

cdecl çağırma biçiminin 32 bit Intel sistemindeki kuralları şunlardır:

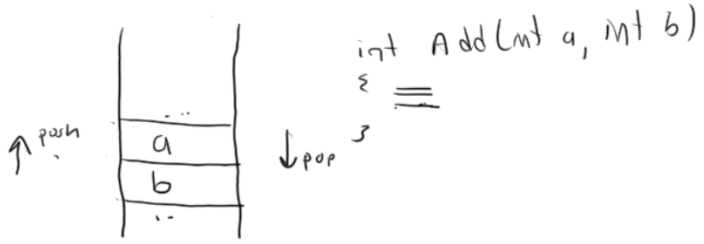
1) Parametre aktarımında stack kullanılır ve parametreler sağdan sola stack'e push edilirler. Parametreler için stack çağıran fonksiyon (caller) tarafından dengelenmektedir.

2) Geri dönüş değerleri yazmaçlar yoluyla aktarılmaktadır. 8 bitlik geri dönüş değerleri AL yazmacı ile, 16 bitlik geri dönüş değerleri AX yazmacı ile (yani EAX'in düşük anlamlı WORD değeri ile), 32 bitlik tamsayı geri dönüş değerleri EAX yazmacı ile ve 64 bitlik tamsayı geri dönüş değerleri de EDX:EAX yazmacı ile aktarılır. float, double ve long double geri dönüş değerlerinin aktarımı için matematik işlemcinin ST(0) yazmacı kullanılmaktadır. Geri dönüş değeri adres türünden olan fonksiyonlarda aktarım için yine EAX yazmacı kullanılır. Geri dönüş değeri yapı türünden olan fonksiyonlarda ise aktarımda önce çağıran fonksiyon geri dönüş değeri için gereken yapı alanını stack'te tahsis eder, onun adresini fonksiyona gönderir, fonksiyon da geri dönüş değerini bu adrese yerleştirir.

3) EAX, ECX ve EDX yazmaçları çağrılan fonksiyon tarafından bozulabilir. Fakat diğer yazmaçlar çağrılan fonksiyon tarafından korunmalıdır.

Burada ikinci ve üçüncü maddenin anlaşılması sembolik makine dili düzeyinde bilgiyi gerektirmektedir. Ancak parametre aktarımında stack'in nasıl kullanıldığı burada ele alınacaktır.

Stack mikroişlemci tarafından kullanılan LIFO prensibiyle çalışan bellekte belirlenmiş olan bir bölgedir. Intel sisteminde stack yukarıya doğru büyür, aşağıya doğru küçülür. cdecl çağırma biçiminde argümanlar sağdan sola stack'e atılıp çağrılan fonksiyon tarafından stack'ten alınmaktadır. Böylece fonksiyon içerisinde ilk parametrenin adresinden hareketle diğer parametrelerin yerleri belirlenebilir.



Parametrelerin sağdan sola stck'e aktarılması değişken sayıda argüman alan fonksiyonların (variadic functions) yazılmasını mümkün hale getirmektedir.

stdcall Çağırma Biçimi

Bu çağırma biçimi Windows sistemlerinde çok yaygın kullanılmaktadır. Windows'un bütün API fonksiyonları ve adresleri bizden alınarak bunlar tarafından çağrılan "callback" fonksiyonlar bu çağırma biçimine sahiptir. Örneğin ExitProcess fonksiyonunun prototipi şöyledir:

```
void WINAPI ExitProcess(UINT uExitCode);
```

API fonksiyonlarının isimlerinin önündeki WINAPI <windows.h> dosyası içerisinde şöyle define edilmiştir:

```
#defineWINAPI __stdcall
```

stdcall çağırma biçimi Linux sistemlerinde seyrek kullanılmaktadır.

stdcall çağırma biçiminin kuralları şöyledir:

1) Parametre aktarımında stack kullanılır ve parametreler sağdan sola stack'e push edilirler. Stack çağrılan fonksiyon (callee) tarafından dengelenir. (cdecl çağırma biçiminde stack'i çağıran fonksiyonun dengelediğini anımsayınız.)

2) Geri dönüş değerlerinin aktarımı tamamen cdecl çağırma biçimindeki gibi yazmaç yoluyla yapılmaktadır.

3) Yine EAX, ECX ve EDX yazmaçları çağrılan fonksiyon tarafından bozulabilir. Fakat diğer yazmaçlar çağrılan fonksiyon tarafından korunmalıdır.

Bu anlatımdan da görüldüğü gibi cdecl çağırma biçimi ile stdcall çağırma biçi arasındaki tek fark stack'i kimin denegeleceği ile ilgilidir. Bu konu sembolik makine dili nilgisi gerektirdiği için burada ele alınmayacaktır. Ancak genel olarak stdcall çağırma biçiminin daha etkin olduğu ancak değişken sayıda argüman alan fonksiyonların yazılmasında güçlük yarattığı söylenebilir. stdcall çağırma biçimi için aşağıdaki fonksiyon prototipileri örnek olarak verilebilir:

```
void __stdcall foo(int a, int b);                 /* Microsoft derleyicileri için prototip */
void __attribute__((stdcall)) foo(int a, int b);  /* gcc derleyicileri için prototip */
```

fastcall Çağırma Biçimi

Bu çağırma biçiminin kuralları Microsoft ve gcc derleyicilerinde yine aynıdır:

1) Parametre aktarımı hem yazmaç hem de stack yoluyla yapılmaktadır. Şöyle ki: Bu çağırma biçiminde ilk iki parametre sırasıyla ECX ve EDX yazmaçlarıyla aktarılır. Eğer parametre sayısı ikiden fazlaysa diğer parametrelerin aktarımı için cdecl çağırma biçimindeki gibi stack kullanılır. (İlk iki parametreden sonraki parametreler yine stack'e sağdan sola push edilir ve stack yine çağrılan fonksiyon tarafından (callee) dengelenir.)

2) Geri dönüş değeri tamamen cdecl'de olduğu gibi yazmaç yoluyla yapılmaktadır.

3) Yine EAX, ECX ve EDX yazmaçları çağrılan fonksiyon tarafından bozulabilir. Fakat diğer yazmaçlar çağrılan fonksiyon tarafından korunmalıdır.

Bu çağırma biçiminde aktarımda yazmaçlar kullanıldığı için çağırma işlemi daha hızlıdır.

fastcall çağırma biçimi için prototip ifadesi şöyle oluşturulabilir:

```
void __fastcall foo(int a, int b);                /* Microsoft derleyicileri için prototip */
void __attribute__((fastcall)) foo(int a, int b); /* gcc derleyicileri için prototip */
```

thiscall Çağırma Biçimi

thiscall çağırma biçimi C++'ta sınıfların static olmayan üye fonksiyonlarının çağrılmasında kullanılmaktadır. Bu çağırma biçiminde static olmayan üye fonksiyonlara this göstericisi ECX yazmacıyla aktarılır. Diğer parametrelerin aktarımı ise tamamen __stdcall çağırma biçimindeki gibidir. Yani parametreler sağdan sola stack'e push edilirler. Parametreler için stack'i çağrılan fonksiyon (callee) dengeler. Geri dönüş değerinin aktarımı da yazmaçlar yoluyla yapılmaktadır. Yine EAX, ECX ve EDX yazmaçlarını çağrılan fonksiyon bozabilir fakat diğerlerini korumak zorundadır.

thiscall çağırma biçiminde de değişken sayıda argüman alan fonksiyonlar etkin bir biçimde yazılamamaktadır.

Intel Ailesinde 64 Bit Programlarda Kullanılan Çağırma Biçimleri

64 bit uygulamalarda çağırma biçimleri de değişmektgedir. Çünkü Intel 64 bit sistemlerde genel amaçlı yazmaçların sayısını artırmıştır.

64 bit Windows sistemlerinde Microsoft'un C derleyicileri 32 bitteki çağırma biçimlerini sentaks olarak kabul etse de onları dikkate almamaktadır. (Yani örneğin biz bir fonksiyonun bildiriminde çağırma biçimi olarak __stdcall ya da __cdecl anahtar sözcüklerini kullansak bile bu durum hataya yol açmaz. Ancak bu anahtar sözcükler semantik olarak dikkate alınmamaktadır.

64 Bit Windows sistemlerinde Microsoft C derleyicileri tarafından uygulanan çağırma biçiminin ana hatları şöyledir:

- İlk 4 parametre eğer tamsayı türündense ya da gösterici türündense bunlar sırasıyla RCX, RDX, R8 ve R9 yazmaçlarıyla aktarılmaktadır. Diğer parametreler sağdan sola stack'e 64 bit olarak push edilir. Dörtten fazla parametre söz konusu olduğundan stack çağıran fonksiyon tarafından (caller) dengelenmektedir. 64 bit Microsoft C derleyicilerinde int ve long türleri 4 byte, long long türü ise 8 byte'tır. 8 byte'tan daha kısa olan parametreler RCX, RDX, R8 ve R9 yazmaçlarının düşük anlamlı 8 bit, 16 bit ve 32 bitlik kısımları yoluyla aktarılır. Bu aktarım sırasında bu yazmaçların yüksek anlamlı byte'ları 0 olmaktadır.

- float ve double türünden parametreler XMM0, XMM1, XMM2 ve XMM3 yazmaçlarıyla aktaraılmaktadır. Ancak parametreler karışık türlerdense yazmaçlar pozisyona göre kullanılmaktadır.

- Fonksiyonun geri dönüş değeri yine 8 byte ve daha küçük tamsayı türlerine ilişkinse RAX yazmacı ile gerçek sayı türlerine ilişkinse XMM0 yazmacı ile aktarılmaktadır. Daha büyük geri dönüş değerleri stack yoluyla aktarılır. Ancak çağıran fonksiyonun stack'te bu yeri ayırmış olması gerekir.

- Microsoft'un 64 bit çağırma biçiminde çağıran fonksiyon CALL işleminden önce stack'te 4 parametre yazmacı için toplam 32 byte'lık bir tampon alan ("register spill" alan) oluşturmalıdır. Fonksiyonun parametreleri daha az olsa bile yine bu 32 byte'lık alanın oluşturulması gerekmektedir.

64 bit çağırma biçimlerinde değişken sayıda argüman alan fonksiyonlar yazılabilirler. Ancak bu fonksiyonların yazımında ilk dört parametrenin yazmaçlardan geri kalanlarının stack'ten alınması gerekmektedir.

64 Bit Linux, MAC OS X ve BSD sistemlerinde kullanılan çağırma biçimi Microsoft'unkine benzemekle birlikte bazı farklılıklar içermektedir. Bu çağırma biçimi "64 bit System 5 ABI (Application Binary Interface)" isimli dokümanda açıklanmıştır.

- İlk 6 parametre 8 byte ya da daha küçük tamsayı türlerine ilişkinse sırasıyla RDI, RSI, RDX, RCX, R8 ve R9 yazmaçlarıyla aktarılır. İlk 6 gerçek sayı türü ise XMM0, XMM1, XMM2, XMM3, XMM4, XMM5 ve XMM6 yazmaçlarıyla aktarılmaktadır. Eğer fonksiyon daha fazla parametreye sahipse bunlar sağdan sola stack'e push edilmektedir. Stack'in temizlenmesi yine çağıran fonksiyon tarafından yapılmaktadır. Ancak bu aktarım Windows'taki gibi pozisyon temelli değildir.

1'inci Parametre: RDI/XMM0
2'inci Parametre: RSI/XMM1
3'üncü Parametre: RDX/XMM2
4'üncü Parametre: RCX/XMM3
5'inci Parametre: R8/XMM4
6'ıncı Parametre: R9/XMM5

Örneğin:

```
void Foo(int a, int b, int c, int d);
```

Burada aktarımda şu yazmaçlar kullanılacaktır:

a -> RDI
b -> RSI
c -> RCX
d -> RDX

Örneğin:

```
void Foo(int a, double b, int c, double d);
```

a -> RDI
b -> XMM0
c -> RSI
d -> XMM1

- Geri dönüş değeri eğer 8 bte ya da daha küçük tamsayı türlerine ilişkinse RAX yazmacı yoluyla, gerçek sayı türleri ise XMM0 yazmacı yoluyla aktarılmaktadır.

C'de Değişken Sayıda Argüman Alan Fonksiyonlar (Variadic Functions)

C'de fonksiyonun değişken sayıda argüman alması durumu ... (ellipsis) ile belirtilmektedir. Örneğin printf fonksiyonunun prototipi şöyledir:

```
int printf(const char *format, ...);
```

C standartlarına göre ... parametresi parametre listesinin sonunda bulunmak zorundadır. Ayrıca değişken sayıda argüman alan fonksiyonların en az bir zorunlu parametresinin bulunması gerekir. Örneğin:

```
void Foo(...);                  /* geçersiz! */
```

```
void Bar(int a, ...);           /* geçerli */
void Tar(int a, int b, ...);    /* geçerli */
void Car(int a, ..., int b);    /* geçersiz! */
```

Değişken sayıda argüman alan fonksiyonlarda bizim bir biçimde fonksiyonun kaç argümanla çağrıldığını bilmemiz gerekir. Bu argüman listesinin sonuna özel bir değer yerleştirilerek belirlenebilir. Ya da birinci parametreden ipucu elde ederek de belirlenebilir. Örneğin printf bunu ilk parametredeki format karakterlerinin sayısı ile belirlemektedir. Ayrıca değişken sayıda argüman alan fonksiyonları yazan programcının fonksiyonun hangi türden argümanlarla çağrıldığını da bilmesi gerekir. Bu konuda baştan bir önkabul yapılabilir. (Örneğin tüm argümanların int olması gerektiği gibi). Örneğin printf fonkisyonu argümanların türlerini format karakterlerine bakarak tespit etmektedir.

Pekiyi değişken sayıda argüman alan bir fonksiyon nasıl yazılabilir? Eğer Intel ailesinde 32 bit sistemde cdecl çağırma biçimi (default çağırma biçimi) kullanılıyorsa ilk parametrenin adresini bilndiğinde diğer parametreler stack'te peşi, sıra geleceği için onların yerleri tespit edilebilir. Örneğin sonu 0 ile biten int türden bir grup sayıyı toplayan Add isimli fonksiyonu aşağıdaki gibi yazabiliriz:

```
#include <stdio.h>

int Add(int a, ...)
{
    int total = 0;
    int paramVal, *pVal;

    pVal = &a;
    while (*pVal) {
        total += *pVal;
        ++pVal;
    }

    return total;
}

int main(void)
{
    int result;

    result = Add(10, 20, 30, 40, 50, 60, 0);
    printf("%d\n", result);

    return 0;
}
```

Fakat bu yazım biçimi platforma ve çağırma biçimine sıkı sıkıya bağlıdır. Örneğin parametre aktarımın yazmaç yoluyla yapıldığı sistemlerde kod çalışmaz. İşte C'de bu işlemi yapacak standart makrolar ya da fonksiyonlar bulundurulmuştur. Bu makro ya da fonksiyonlar standarttır her sistemde o sisteme gçre gerçekleştirilmiştir. Bu makro ya da fonksiyonların prototipleri <stdarg.h> dosyası içerisindedir. Standart makrolar ya da fonksiyonlar kullanılarak değişken sayıda argüman alan fonksiyonlar şöyle yazılırlar:

1) Önce va_list türünden bir tane değişken tanımlanır.
2) Bu değişken va_start makrosu ile ilkdeğerlenir. Bu makronun birinci parametresi va_list türünden değişkeni ikinci parametresi ilk parametreyi (aslında ... argümanının hemen solundaki parametreyi) almaktadır.

3) Sonra sırasıyla her argüman va_arg makrosuyla elde edilir. Bu makronun birinci parametresi va_list türünden değişkeni ikinci parametresi ise çekileek argümanın türünü belirtir.

4) Kullanımın sonunda va_end makrosu ya da fonksiyonu çağrılmalıdır.

Örneğin birinci parametresiyle belirtilen sayıda int değeri toplayan bir fonksiyon yazmak isteyelim:

```c
#include <stdio.h>
#include <stdarg.h>

int Add(int count, ...)
{
    int total = 0;
    int i;
    va_list va;

    va_start(va, count);
    for (i = 0; i < count; ++i)
        total += va_arg(va, int);
    va_end(va);

    return total;
}

int main(void)
{
    int result;

    result = Add(5, 10, 20, 30, 40, 50);
    printf("%d\n", result);

    return 0;
}
```

Bu tür fonksiyonlarda derleyici prototip ya da tanımlamada açıkça tür görmediği için argümanı programcının girdiğine uygun olarak fonksiyona (örneğin stack'e) yollar. Bu durumda programcının fonksiyonun argümanlarının uygun türlerden olamsını sağlaması gerekir. Örneğin yukarıdaki Add fonksiyonunu double değerleri, toplaacak biçime dönüştürmek isteyelim:

```c
#include <stdio.h>
#include <stdarg.h>

double Add(int count, ...)
{
    double total = 0;
    int i;
    va_list va;

    va_start(va, count);
    for (i = 0; i < count; ++i)
        total += va_arg(va, double);
    va_end(va);

    return total;
}

int main(void)
{
    double result;

    result = Add(2, 10., 20.);
    printf("%f\n", result);

    return 0;
}
```

Burada Add fonksiyonunun çağrımına dikkat ediniz:

```c
result = Add(2, 10., 20.);
```

Eğer biz argümanları yanlışlıkla int türden verseydik tamamen yanlış bir değer elde ederdik:

```
result = Add(2, 10, 20);
```

C'de ... (ellipsis) parametresine karşı gelen argümanlar "default argüman dönüştürmesi ile" fonksiyona gönderilmktedir. Default argüman dönüştürmesi şöyledir:

1) int türünden küçük olan türler int türüne yükseltme kuralı uygulanarak fonksiyona yollanır.

2) float türü double türüne yükseltilerek fonksiyona yollanır.

3) diğer türler oldukları gibi hiç dönüştürülmeden fonksiyona yollanırlar.

Bu makrolar ve fonksiyonlar 32 bit sistemlerde cdelc dışındaki diğer çağırma biçimlerinde de çalışmaktadır. Ancak yukarıda da belirtildiği gibi bu işlem cdecl çağırma biçiminin dışında daha zor gerçekleştirilir.

Şimdi printf fonksiyonunu yalnızca %c, %d ve %f format karakterlerini basit yalın olarak destekleyecek biçimde yazmaya çalışalım:

```
#include <stdio.h>
#include <stdarg.h>

double __fastcall Add(int count, ...)
{
    double total = 0;
    int i;
    va_list va;

    va_start(va, count);
    for (i = 0; i < count; ++i)
        total += va_arg(va, double);
    va_end(va);

    return total;
}

int myprintf(const char *format, ...)
{
    int ival;
    double dval;
    int i, count = 0;
    char buf[64];
    va_list va;

    va_start(va, format);

    while (*format != '\0') {
        if (*format == '%') {
            switch (*++format) {
                case 'd':
                    ival = va_arg(va, int);
                    sprintf(buf, "%d", ival);
                    break;
                case 'c':
                    buf[0] = va_arg(va, int);
                    buf[1] = '\0';
                    break;
                case 'f':
                    dval = va_arg(va, double);
                    sprintf(buf, "%f", dval);
                    break;
            }
```

```
            for (i = 0; buf[i] != '\0'; ++i) {
                putchar(buf[i]);
                ++count;
            }
        }
        else {
            putchar(*format);
            ++count;
        }
        ++format;
    }

    va_end(va);

    return count;

}

int main(void)
{
    int a = 10;
    char b = 'x';
    double c = 12.345;
    int result;

    result = myprintf("a = %d, b = %c, c = %f\n", a, b, c);
    myprintf("result = %d\n", result);

    return 0;
}
```

Pekiyi scanf fonksiyonu nasıl yazılmış olabilir? Burada scanf fonksiyonu yazmak biraz zaman alabilir. Ancak nasıl yazıldığını açıklayabiliriz: scanf'te format karakterleri girişin yerleştirileceği adresin türünü belirtir. Yani biz her % karakterini gördüğümüzde bu format karakterine uygun olan adresi va_arg ile çekeriz. scanf fonksiyonunda bizim çekeceğimiz argümanlar aslında birer adrestir biz de bilgiyi o adrese yerleştiririz.

C'de vprintf, vfprintf ve vsprintf fonksiyonları da çok sık kullanılmaktadır.

```
int vprintf(const char *format, va_list argptr);
int vfprintf(FILE *stream, const char *format, va_list argptr);
int vsprintf(char *buffer, const char *format,va_list argptr);
```

vprintf fonksiyonun birinci parametresi printf fonksiyonundaki format yazısının aynısıdır. Fonksiyon printf'teki geri kalan parametreleri ikinci parametresi olan va_list ile ister. Bir fonksiyonun parametresinde va_list gördüğümüz zaman biz adeta bunun ... parametresinin aktarılmasında kullanılacağını anlamalıyız. vfprintf ise fprintf fonksiyonunun v'li biçimidir. vsprintf fonksiyonu da benzer biçimde sprintf fonksiyonun v'li biçimidir. Biz bu fonksiyonlar sayesinde sırasıyla printf, fprintf vs sprintf fonksiyonlarını çağıran fonksiyonlar yazabiliriz. Örneğin:

```
void ErrExit(const char *format, ...)
{
    va_list va;

    va_start(va, format);
    vfprintf(stderr, format, va);
    va_end(va);
    fflush(stderr);
    exit(EXIT_FAILURE);
}
```

Burada ErrExit aslında adeta fprintf fonksiyonunu çağırıyor gibidir. Bu fonksiyon format parametresinden va_list değerini elde etmiş ve bunu vfprintf fonksiyonuna yollamıştır.

Paralel Programlamaya Giriş

Birarada çalışan kodları betimlemek için pek çok terim kullanılmaktadır. Bunların arasında bağlam bakımından bazı farklılıklar bulunmaktadır. Öncelikle bu terimlerin ve kavramların anlamlarını açıklayalım:

Concurrent Computing / Concurrent Programming (Birden Fazla Akışla Programlama): Bu terim en genel olanlardandır. Birden fazla akışın kullanılarak bir işin gerçekleştirilmesini anlatır. Örneğin çok thread'li uygulamalar tipik olarak "concurrent computing" kavramı içerisinde değerlendirilebilir. Benzer biçimde paralel programlama da, dağıtık programlama da bir "concurrent computing" faaliyetidir.

Distributed Computing / Distributed Programming (Dağıtık Programlama): Bu terim ve anlattığı kavram işlerin birden fazla bilgisayara dağıtılarak birlikte gerçekleştirilmesi anlamına gelir. Terimin odaklandığı nokta söz konusu işlerin aynı makinede değil de bir ağdaki makinelerde koordineli olarak yürütülmesi sürecidir. Bugün dağıtık uygulamalar çok yaygınlaşmıştır. Şüphesiz dağıtık uygulamalar birtakım protokolleri kullanrak proseslerarası haberleşme yöntemleriyle koordinasyonu sağlamaktadır. Tipik olarak bunun için IP protokol ailesi tercih edilmektedir. Ancak dağıtık uygulamalar için daha yüksek seviyeli birtakım framework'ler ve kütüphaneler de oluşturulmuştur. (Örneğin .NET'in WCF framework'ü, Java'nın "Remoting" kütüphanesi vs.) Dağıtık uygulamalar ayrı bir kursun konusu olabilecek derece geniş bir konudur. Bugün pek çok dağıtık sistem yaşantımıza girmiş durumdadır. Örneğin bulut uygulamaları, web tabanlı pek çok uygulama (örneğin Facebook, Twitter gibi), veri madenciliği uygulamaları dağıtık sistemleri kullanmaktadır. Dağıtık sistemler hata toleransını düşürebilmekte, ölçeklendirilebilir (scalable) bir sistem sunabilmektedir. Bir sistemin ölçeklendirilebilir (scalable) olması demek onun fazla yük ve talep altında kolayca genişletilip gereksinimi karşılayabilmesi demektir.

Multithreaded Computing/Multithreaded Programming: Bu terim aynı makinede işlerin birliktee birden fazla thread oluşturularak gerçekleştirilmesi anlamına gelir. Şüphesiz bu terim de aslında bir "concurrent programming" şemsiyesi altındadır.

Parallel Computing /Parallel Programming: Bu terim bir işi aynı makinede thread'lere ayırarak ve onları farklı CPU ya da çekirdeklere atayarak aynı anda çalıştırma gayretini anlatmaktadır. Dolayısıyla burada toplamda hızlı bir işlemin yapılması hedeflenir. Şüphesiz tek CPU ya da tek çekirdekli sistemlerde paralel programlama yapılamaz. Öncelikle paralel programlama için söz konusu sistemde birden fazla CPU ve/veya ya da çekirdek bulunması gerekir. Hatta paralel programlama faaliyetinde bir süredir grafik kartlarındaki işlemcilerden de faydalanılabilmektedir.

Pekiyi madem ki paralel programlama işlerin thread'lere bölünüp mümkünse aynı anda farklı işlemcilerde ya da çekirdeklerde çalıştırılması anlamına geliyor biz bunu aşağıy seviyede nasıl sağlayabiliriz? Thread'lerin yaratılmasını ve senkronize edilmesini "Sistem Programlama ve İleri C Uygulamaları I" Kursunda görmüştük. Bir thread'in belli bir CPU ya da çekirdekte çalışmaya zorlanması sürecine "processor affinity" denilmektedir. Windows ve Linux gibi işletim sistemlerinde bu işi yapan API fonksiyonları bulunmaktadır. Bilindiği gibi işletim sistemleri hiçbir müdahalede bulunulmadığı durumda thread'leri toplam performansı yükseltecek biçimde CPU ya da çekirdeklere atamaktadır. Ancak biz bu API fonksiyonları sayesinde yarattığımız thread'lerin belli CPU ya da çekirdeklere de çalışmasını isteyebiliriz.

Windows'ta SetThreadAffinityMask fonksiyonu thread'in hangi CPU ya da çekirdeklere atanabileceğini belirlemekte kullanılır.

```
DWORD_PTR WINAPI SetThreadAffinityMask(
  __in          HANDLE hThread,
  __in          DWORD_PTR dwThreadAffinityMask
);
```

Fonksiyonun birinci parametresi thread'in handle değerini ikinci parametresi thread'in hangi CPU ya da çekirdeklere atanabileceğini belirtir. Thread bu parametrede 1 olan bitlere karşı gelen CPU ya da çekirdeklerde çalıştırılabilir. Fonksiyon başarı durumunda thread'in bir önceki affinity mask değeri ile geri döner. Başarısızlık durumunda fonksiyon 0 değeri ile geri dönmektedir. Örneğin biz n çekirdekli bir sistemde yarattığımız n thread'in farklı çakirdeklerde çalıştırılmasını şöyle sağlayabiliriz:

```
GetSystemInfo(&g_si);
for (i = 0; i < g_si.dwNumberOfProcessors; ++i) {
    if ((hThreads[i] = CreateThread(NULL, 0, ThreadProc, (void *)DoProc, 0, &dwThreadIds[i])) == NULL) {
        fprintf(stderr, "Cannot create thread: %ld\n", GetLastError());
        exit(EXIT_FAILURE);
    }

    SetThreadAffinityMask(hThreads[i],  1 << i);
}
```

Windows'ta prosesin de bir "affinity mask" değeri vardır. Bu değer SetProcecessAffinityMask fonksiyonuyla değiştirilebilir. Bu değer bir ana şalter görevi görmektedir. Biz prosesin thread'ini onun "affinity mask"inde belirtilen CPU ya da çekirdek dışındaki bir CPU ya da çekirdeğe SetThreadAffinityMask fonksiyonuyla atayamayız. Windows default durumda genel olarak hem prosesin hem de thread'in "affinity mask" değeri tüm CPU ya da çekirdeklere açık biçimdedir.

UNIX/Linux sistemlerinde thread'in "affinity mask" değeri pthread-setaffinity_np fonksiyonuyla ayarlanabilir ve pthread_getaffinity_np fonksiyonuyla da alınabilir:

```
#include <pthread.h>

int pthread_setaffinity_np(pthread_t thread, size_t cpusetsize, const cpu_set_t *cpuset);
int pthread_getaffinity_np(pthread_t thread, size_t cpusetsize, cpu_set_t *cpuset);
```

Görüldüğü gibi aslında paralel programlama faaliyeti aşağı seviyede işletim sisteminin sunduğu thread fonksiyonlarıyla ve "affinity mask" işlemiyle sağlanmaktadır. Ancak bu işlemlerin aşağı seviyede programcı tarafından thread'ler yaratılarak yapılması oldukça zahmetlidir. İşte bu nedenle paralel programlamayı kolaytıran çeşitli kütüphaneler ve framework'ler geliştirilmiştir. Bunlar sayesinde biz kodumuzu daha yüksek seviyeli olarak yazıp bütün bu thread işlemlerini bu framework'lere ya da kütüphanelere devrebilmekteyiz.

Paralel Programlama İçin Kullanılan Yüksek Seviyeli Kütüphaneler

Paralel programlama için farklı dillerde ve ortamlarda pek çok farklı kütüphane ve framework'ler bulunmaktadır. Örneğin NET'te Framework 4.0 ile birlikte "Task Parallel Library" ismiyle bir paralel programalam kütüphanesi .NET ortamına dahil edilmiştir. Java'da farklı birçok seçenek bulunmaktadır. "Fork And Join" isimli kütüphane en yaygın kullanılanlardan biridir. C ve C++'ta MPI kütüphanesi ve onun daha spesifik ve açık kaynak kodlu biçimi olan OpenMP ağırlıklı olarak tercih edilmektedir. MPI ve OpenMP Fortan'dan da kullanılabilmektedir. Biz kursumuzda OpenMP'nin kullanımı üzerinde duracağız.

Paralel programlama kütüphaneleri ve ortamları "declarative" ve "impertive" olmak üzere ikiye ayrılmaktadır. "Declarative" kütüphanelerde ya da ortamlarda programcı kodunu normal biçimde yazar. Sonra birtakım direktiflerle onu paralelize eder. Halbuki "imperative" kütüphanelerde ya da ortamlarda programcı tamamen fonksiyon çağrılarıyla kodu paralel hale getirmektedir. Tabii bunların karmaşımı olan hibrit kütüphaneler ve ortamlar da vardır. OpenMP hem "declarative" hem de "imperative" özellikleri olan bir kütüphanedir. Yani bazı işlemler doğrudan direktiflerle bazıları ise fonksiyon çağrılarıyla yapılmaktadır.

OpenMP'ye Giriş

OpenMP'nin ilk uyarlaması ilk kez 1997 yılında gerçekleştirilmiştir. Bu uyarlama yalnızca Fortran'a yönelikti. Daha sonra 2000 yılında 2.0 versiyonuyla OpenMP C ve C++'ı kapsayacak biçimde genişletildi. Bunu 2008

yılında 3.0, 2013 yılında 4.0 versyionları izledi. Kursun yapıldığı zaman dilimindeki en son versiyonu 4.5'tir. Open MP pek çok firmanın sponsorluğunda (örneğin AMD, Intel , IBM, Oracle ve Nvidia gibi) geliştirilmekte olan açık bir yazılımdır.

OpenMP yukarıda da belirtildiği gibi "declarative" ve "imperative" özellikleri olan bir kütüphanedir. Microsoft'un C ve C++ derleyicileri, gcc ve g++ derleyicileri, clang derleyicileri OpenMP'yi desteklemektedir.

OpenMP'nin Kullanıma Hazır Hale Getirilmesi

OpenMP doğrudan derleyiciler tarafından desteklenmektedir. Dolayısıyla kurulumu söz konusu değildir. Bizim ilgili derleyicide OpenMP seçeneğini açmamız gerekir. Bunun için Microsoft derleyicilerinde proje seçeneklerine gelinir. C-"C++/Language/OpenMP supports" seçeği açılır. Aslında bu işlem komut satırı "cl.exe" derleyicisinde /openmp seçeneğinin kullanılmasına yol açmaktadır. Yani "cl.exe" komut satırında OpenMP uygulamalarını derlerken "/openmp" seçeneğini girmemiz gerekir. Bu seçenek girildiğinde OpenMP kütüphanelerini ve başlık dosyalarını kullanabilir duruma geliriz. gcc ve clang derleyicilerinde komut satırında -f openmp seçeneği OpenMP kütüphanesi devreye sokulabilir.

C ve C++'ta OpenMP kütüphanesi için <omp.h> isimli tek bir başlık dosyası kullanılmaktadır. Ayrıca eğer OpenMP etkin duruma getirildiyse _OPENMP isimli makronun da bildirilmiş olduğu varsayılmaktadır. Böylece programcı koşullu derleme işlemini yapabilir:

```
#ifdef _OPENMP
    ...
#endif
```

OpenMP'nin Temel Kullanımı

C ve C++'ta OpenMP'deki "declarative" özellikler #pragma omp direktifiyle belirtilmektedir. Bu pragma direktifini OpenMP'ye özgü bir anahtar sözcük (openMP direktifi) izler, sonları da bazı cümlecikler (clause) izlemektedir. #pragma omp direktifinin genel biçimi şöyledir:

```
#pragma omp directive-name [clause[ [,] clause] ... ] new-line
```

OpenMP pragma direktifleri tek bir değimi etkisi altına almaktadır. Tabii bu deyim basit, bileşik ya da bir kontrol deyimi olabilir. Yani eğer biz bu direktiflerin birden fazla deyimi etkisi aşltına almasını istiyorsak bloklama yapmamız (bileşik deyim) gerekir.

En önemli OpenMP direktiflerinden birisi parallel direktifidir:

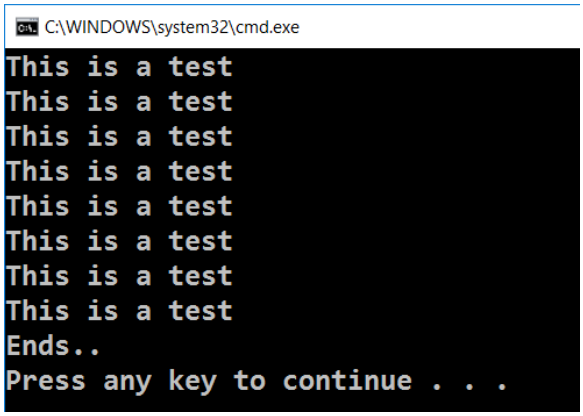
```
#pragma omp parallel
```

direktidir. Bu direktif belirlenen deyimin birden fazla thread tarafından paralel çalıştırılaacağı anlamına gelir. Default durumda toplam CPU ya da çekirdek sayısı kadar thread bu paralel çalışmaya katılır. Örneğin:

```
#include <stdio.h>
#include <omp.h>

void DoParallel(void)
{
#pragma omp parallel
    printf("This is a test\n");
}

int main(void)
{
    DoParallel();
```

```
    printf("Ends...\n");

    return 0;
}
```

Denemenin yapıldığı bilgisayardaki ekran çıktısı şöyledir:



Buradan da tahmin edileceği gibi denemenin yapıldığı makinede toplam 8 çekirdek vardır. OpenMP direktifine giren thread'e OpenMP terminolojisinde "asıl thread (master thread)" denilmektedir. Asıl thread'de bu paralel çalışmaya katılmaktadır. #pragma mp parallel direktifinde ilgili deyimin sonunda OpenMP terminolojisiyle bir engel (barrier) vardır. Tüm thread'ler paralel biçimde belirlenen deyimi çalıştırdıktan sonra bu engelde birbirlerini beklerler (yani hepsinin bu çalışmayı bitirmesini). Tabii bu beklemeye asıl thread (master thread) de dahil olmaktadır. Engelden sonra diğer thread akışları yok edilir ve alışma yine asıl thread tarafından devam ettirilir.

Aşağıdaki OpenMP programını inceleyiniz:

```
#include <stdio.h>
#include <omp.h>

void DoParallel(void)
{
    #pragma omp parallel
    {
        printf("1 ");
        printf("2 ");
        printf("3 ");
    }
}

int main(void)
{
    DoParallel();
    printf("Ends...\n");

    return 0;
}
```

Bu programın bir çalıştırılmasında ekran çıktısı şöyledir:

```
1 2 1 3 1 1 2 3 1 2 1 2 2 3 1 2 2 3 3 3 3 1 2 3 Ends...
Press any key to continue . . .
```

Burada paralel akışların seri hale getirilmediğine ve dolayısıyla bir iç içe geçmenin olduğuna dikkat ediniz.

Paralel çalışmada paralel çalıştırılan bileşik deyimin içerisinde bildirilen değişkenler her thread için farklıdır. Yani thread bunların kendine özgü bir kopyasına kullanır. Örneğin:

```
#include <stdio.h>
#include <omp.h>

void DoParallel(void)
{
    #pragma omp parallel
    {
        int a = 0;
        ++a;
        printf("%d\n", a);
    }
}

int main(void)
{
    DoParallel();
    printf("Ends...\n");

    return 0;
}
```

Ekran çıktısı şöyle olmaktadır:

```
1
1
1
1
1
1
1
1
Ends...
Press any key to continue . . .
```

Ancak yukarıdaki bloğun yerel değişkenleri ve global değişkenler default durumda paralel çalıştırmayı sağlayan thread'ler arasında paylaşılmaktadır. Örneğin:

```
#include <stdio.h>
#include <omp.h>

void DoParallel(void)
{
    int a = 0;

    #pragma omp parallel
    {
        ++a;
        printf("%d\n", a);
    }
}

int main(void)
{
    DoParallel();
    printf("Ends...\n");

    return 0;
}
```

```
1
2
3
4
5
6
7
8
Ends...
Press any key to continue . . .
```

Tabii bu çıktıda sayıların peş peşe gelmesinin bir garantisi yoktur. Ancak üst bloktaki yerel a değişkenin thread tarafından ortak kullanıldığı görülmektedir. Default durumdaki davranış paralel bloğa özgü olarak C ve C++'ta "default(shared)" ya da "default(none)" olarak değiştirilebilir. "default(none)" üst bloktaki yerel değişkenlerin ayrıca private ya da shared cümlesiyle durumunun belirtilmesi gerektiği anlamına gelir.

Paralel blok çalıştırıldığında OpenMP bunun için yaratılan thread'lere 0'dan başlayarak birer numara verir. Bu numara omp_get_thread_num fonksiyonuyla elde edilebilir. Örneğin:

```
#include <stdio.h>
#include <omp.h>

void DoParallel(void)
{

    int i;

    #pragma omp parallel  default(none) shared(i)
    {
        for (i = 0; i < 10; ++i)
            printf("thread No: %d, i = %d\n", omp_get_thread_num(), i);
    }
}

int main(void)
{
    DoParallel();
    printf("Ends...\n");

    return 0;
}
```

```
C:\WINDOWS\system32\cmd.exe
thread No: 1, i = 0
thread No: 1, i = 1
thread No: 6, i = 0
thread No: 4, i = 0
thread No: 0, i = 0
thread No: 3, i = 0
thread No: 7, i = 0
thread No: 6, i = 1
thread No: 6, i = 3
thread No: 6, i = 4
thread No: 5, i = 0
thread No: 5, i = 6
thread No: 5, i = 7
thread No: 2, i = 0
thread No: 2, i = 9
thread No: 1, i = 1
thread No: 7, i = 2
thread No: 4, i = 2
thread No: 6, i = 5
thread No: 0, i = 1
thread No: 5, i = 8
thread No: 3, i = 1
Ends...
Press any key to continue . . .
```

Üst bloklardaki yerel değişkenler private yapıldığında artık onların içerisindeki değerler dikkate alınmaz. Yani thread'ler onlara sanki değer atanmamış gibi davranırlar. Paralel bloğun çıkıldığında private değişkenlerin durumunun ne olacağı belirsizdir. Eğer hem direktifin dışındaki değişkenlerin yaratılan thread'ler için private olması hem de ilkdeğerinin kullanışması (yani ilkdeğerinin dikkate alınması) isteniyorsa private yerine firstprivate cümlesi kullanılır. Örneğin:

```
#include <stdio.h>
#include <omp.h>

int main(void)
{
    int i = 0;

    #pragma omp parallel firstprivate(i)
    {
        #pragma omp critical
        {
            for (; i < 10; ++i)
                printf("Thread Num: %d: %d\n", omp_get_thread_num(), i);
            printf("-------------\n");
        }
    }

    return 0;
}
```

Programcılar genellikle parallel bloğunda fonksiyon çağırırlar. Bu fonksiyon her thread için bir kez çalıştırılmış olur. Örneğin:

```
#include <stdio.h>
#include <omp.h>

void DoParallel(void)
{
    int i;

    for (i = 0; i < 10; ++i)
        printf("thread No: %d, i = %d\n", omp_get_thread_num(), i);
}

int main(void)
{
#pragma omp parallel
    DoParallel();

    printf("Ends...\n");

    return 0;
}
```

#pragma omp parallel içerisinde #pragma omp for direktifi kullanılırsa bu direktif bizden bir for döngüsü ister. Bu for döngüsünün n tane thread tarafından fakat her bir döngü değişkjeni için bir kez olmak koşuluyla hızlı bir biçimde çalıştırılmasını sağlar. Örneğin:

```
int i;

#pragma omp parallel
{
    #pragma omp for
    for (i = 0; i < 32; ++i)
        HeavyProcess();
}
```

Burada toplam 8 çekirdeğin bulunduğunu düşünelim. Döngü toplam 32 kez dönecektir. Ancak bu sekiz thread'in her biri 4 dönüş yapacak biçimde bu döngü thread'lere paylaştırılır. Sonuçta döngü daha çabuk bitirilmiş olur. Tabii bu döngüde döngü değişkeninin her değeri için yine döngü yalnızca bir kez çalıştırılır. #pragma omp for direktifi #pragma omp parallel içerisinde bulunmazsa bu etki söz konusu olmaz. Tabii biz #pragma omp parallel direktifi içerisinde başika şeyler de yapabiliriz. Örneğin:

```
int i;

#pragma omp parallel
{
    #pragma omp for
    for (i = 0; i < 32; ++i)
        HeavyProcess();
    OtherProcess();
}
```

Burada OtherProcess yine birden fazla thread tarafından birden fazla kez çalıştırılacaktır. Ancak for döngüsaü birden fazla thread tarafından toplamda belirtildiği miktarda çalıştırılmış olacaktır.

Eğer #pragma omp parallel direktifinde yalnızca #pragma omp for bulunacaksa bu yalın olarak #pragma omp parallel for
biçiminde belirtilebilir. Örneğin:

```
#pragma omp parallel for
for (i = 0; i < 32; ++i)
    HeavyProcess();
```

#pragma omp for direktifi yalnızca for döngüsü ile kullanılır. Diğer döngülerle ya da deyimlerle kullanılmaz. Yani bu direktifi for döngüsü izlemek zorundadır.

Pekiyi paralel for döngüleri işlemi ne kadar hızlandırabilmektedir? Aşağıda 10 milyar kez dönen boş bir döngü paralel for ile ve seri olarak çalıştırılmaktadır. 8 çekirdekli sistemde aralarındaki farka dikkat ediniz:

```
#include <stdio.h>
#include <Windows.h>
#include <omp.h>

int main(void)
{
    long long int i;
    LARGE_INTEGER li1, li2, freq;
    __int64 result;

    QueryPerformanceFrequency(&freq);

    QueryPerformanceCounter(&li1);

    #pragma omp parallel for
    for (i = 0; i < 10000000000LL; ++i) {
        /* ... */
    }

    QueryPerformanceCounter(&li2);

    result = li2.QuadPart - li1.QuadPart;
    printf("%f\n", (double)result / freq.QuadPart);

    QueryPerformanceCounter(&li1);

    for (i = 0; i < 10000000000LL; ++i) {
        /* ... */
    }
```

```
    QueryPerformanceCounter(&li2);

    result = li2.QuadPart - li1.QuadPart;
    printf("%f\n", (double)result / freq.QuadPart);

    return 0;
}
```

```
4.834711
19.213666
Press any key to continue . . .
```

Paralel for dönüsünde döngü değişkeni sırasıyla değer almak zorunda değildir. Ancak her değer için bir kez işlem yapılacağı garanti edilmiştir. Aşağıdaki örneği inceleyerek sonucu yorumlayınız:

```
#include <stdio.h>
#include <omp.h>

int main(void)
{
    int a[32];
    int i;

    #pragma omp parallel for
    for (i = 0; i < 32; ++i) {
        a[i] = i;
        printf("%d ", a[i]);
    }
    printf("\n");

    for (i = 0; i < 32; ++i)
        printf("%d ", a[i]);
    printf("\n");

    return 0;
}
```

```
0 16 17 18 24 25 28 29 4 5 20 6 21 7 12 13 19 14 1 15 2 3 30 31 8 9 22 10 23 11 26 27
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
Press any key to continue . . .
```

Bu örnekte ayrıca a dizisinin default olarak “shared” olduğuna dikkat ediniz.

#pragma omp single direktifi #pragma omp parallel direktifi içerisinde kullanıldığında ilgili deyimin yalnızca bir kez yapılacağını belirtir. Örneğin:

```
#include <stdio.h>
#include <omp.h>

int main(void)
{
    #pragma omp parallel
    {
        printf("Each per thread\n");
        #pragma omp single
        {
            printf("Only one thread\n");
        }
    }
    printf("Ends...\n");
```

```
    return 0;
}
```

```
Each per thread
Only one thread
Each per thread
Each per thread
Each per thread
Each per thread
Each per thread
Each per thread
Each per thread
Ends...
Press any key to continue . . .
```

Burada #pragma omp single bloğu toplamda yalnızca bir kez çalıştırılmıştır. Halbuki diğerleri her thread için bir kez çalıştırılmıştır. Tabii hangi thread'in #pragma omp single bloğunu çalıştıracağı konusunda bir beelirleme yoktur.

#pragma omp single direktifinin sonunda bir bariyer (barrier) vardır. OpenMP terminolojisinde bariyer tüm akışların birbirlerini beklediği noktalara denilmektedir. (Yani örneğin WaitForMultipleObjects gibi bir nesneyle o noktada beklendiğini düşünebilirisiniz.)

```
#include <stdio.h>
#include <omp.h>

int main(void)
{
    omp_set_num_threads(10);

    #pragma omp parallel
    {
        printf("Each per thread: %d / %d\n", omp_get_thread_num(), omp_get_num_threads());
        #pragma omp single
        {
            printf("Only one thread: %d / %d\n", omp_get_thread_num(), omp_get_num_threads());
        } -----> Bu noktada barrier var.
        printf("Parallel ends...\n");
    }
    printf("Ends...\n");

    return 0;
}
```

```
Each per thread: 0 / 10
Each per thread: 2 / 10
Each per thread: 6 / 10
Each per thread: 7 / 10
Each per thread: 8 / 10
Each per thread: 5 / 10
Only one thread: 0 / 10
Each per thread: 4 / 10
Each per thread: 3 / 10
Each per thread: 1 / 10
Each per thread: 9 / 10
Parallel ends...
Parallel ends...
Parallel ends...
Parallel ends...
Parallel ends...
Parallel ends...
Parallel ends...
Parallel ends...
Parallel ends...
Parallel ends...
Ends...
Press any key to continue . . .
```

Default durumda paralel blokların toplam CPU ya da çekirdek sayısı kadar thread tarafından çalıştırıldığını belirtmiştik. Bu durum istenirse omp_set_num_threads fonksiyonuyla değiştirilebilir. Örneğin:

```
#include <stdio.h>
#include <omp.h>

int main(void)
{
    omp_set_num_threads(12);

    #pragma omp parallel
    {
        printf("Each per thread\n");
        #pragma omp single
        {
            printf("Only one thread\n");
        }
    }
    printf("Ends...\n");

    return 0;
}
```

```
Each per thread
Only one thread
Each per thread
Each per thread
Each per thread
Each per thread
Each per thread
Each per thread
Each per thread
Each per thread
Each per thread
Each per thread
Each per thread
Ends...
Press any key to continue . . .
```

Paralel blokları işleyecek toplam thread sayısı omp_get_num_threads fonksiyonuyla elde edilebilir. Örneğin:

```c
#include <stdio.h>
#include <omp.h>

int main(void)
{
    omp_set_num_threads(12);

    #pragma omp parallel
    {
        printf("Each per thread: %d / %d\n", omp_get_thread_num(), omp_get_num_threads());
        #pragma omp single
        {
            printf("Only one thread: %d / %d\n", omp_get_thread_num(), omp_get_num_threads());
        }
    }
    printf("Ends...\n");

    return 0;
}
```

```
Each per thread: 3 / 12
Each per thread: 7 / 12
Each per thread: 4 / 12
Each per thread: 9 / 12
Each per thread: 10 / 12
Each per thread: 11 / 12
Each per thread: 1 / 12
Each per thread: 5 / 12
Each per thread: 2 / 12
Each per thread: 6 / 12
Only one thread: 3 / 12
Each per thread: 0 / 12
Each per thread: 8 / 12
Ends...
Press any key to continue . . .
```

#pragma omp parallel direktifinde istenirse yalnızca bu direktife özgü olarak thread sayısı ayarlanabilir. Bunun için num_threads cümleciği kullanılır. Örneğin:

```c
#include <stdio.h>
#include <omp.h>

int main(void)
{
    #pragma omp parallel num_threads(5)
    {
        printf("Each per thread: %d / %d\n", omp_get_thread_num(), omp_get_num_threads());
        #pragma omp single
        {
            printf("Only one thread: %d / %d\n", omp_get_thread_num(), omp_get_num_threads());
        }
        printf("Parallel ends...\n");
    }
    printf("-------------------\n");

    #pragma omp parallel num_threads(2)
    {
        printf("Each per thread: %d / %d\n", omp_get_thread_num(), omp_get_num_threads());
    #pragma omp single
        {
            printf("Only one thread: %d / %d\n", omp_get_thread_num(), omp_get_num_threads());
        }
        printf("Parallel ends...\n");
    }
```

```
    printf("Ends...\n");

    return 0;
}
```

```
Each per thread: 0 / 5
Each per thread: 2 / 5
Each per thread: 4 / 5
Only one thread: 0 / 5
Each per thread: 1 / 5
Each per thread: 3 / 5
Parallel ends...
Parallel ends...
Parallel ends...
Parallel ends...
Parallel ends...
------------------
Each per thread: 0 / 2
Only one thread: 0 / 2
Each per thread: 1 / 2
Parallel ends...
Parallel ends...
Ends...
Press any key to continue . . .
```

Burada ilk parallel bloğa beş thread'in sonraki paralel bloğa iki thread'in girdiğine dikkat ediniz. Halbuki omp_sen_num_threads fonksiyonu ana şalter görevi yapmaktadır.

OpenMP'de kritik kod oluşturmak için #pragma omp critical direktifi kullanılmaktadır. Bu direktifte belirtilen deyim iç içe değil başından sonuna kadar tek bir thread tarafından yapılır. Örneğin:

```
#include <stdio.h>
#include <omp.h>

#include <stdio.h>
#include <omp.h>

int main(void)
{
    #pragma omp parallel num_threads(5)
    {
        printf("1) %d/%d\n", omp_get_thread_num(), omp_get_num_threads());
        printf("2) %d/%d\n", omp_get_thread_num(), omp_get_num_threads());
        printf("3) %d/%d\n", omp_get_thread_num(), omp_get_num_threads());
    }

    printf("Ends...\n");

    return 0;
}
```

Burada parallel bloğun içerisindeki üç ayrı printf fonksiyonu thread'ler atarfından aynı zaman diliminde asenkron biçimde çalıştırılacaktır. Bir çalıştırmanın ekran çıktısı aşağıdaki gibidir:

```
1) 0/5
2) 0/5
3) 0/5
1) 3/5
1) 4/5
2) 4/5
1) 1/5
3) 4/5
2) 1/5
3) 1/5
1) 2/5
2) 2/5
2) 3/5
3) 2/5
3) 3/5
Ends...
Press any key to continue . . .
```

Ancak biz bu üç printf fonksiyonunu kritik kod içerisine alabiliriz. Bu durumda bu üç printf fonksiyonu başından sonuna kadar tek bir thread akışı tarafından çalıştırılacaktır. Diğer thread akışları kritik kod bloğunun başında kritik koda girmiş olan thread'in kritik koddan çıkmasını bekleyecektir. Örneğin:

```
1) 0/5
2) 0/5
3) 0/5
1) 2/5
2) 2/5
3) 2/5
1) 1/5
2) 1/5
3) 1/5
1) 3/5
2) 3/5
3) 3/5
1) 4/5
2) 4/5
3) 4/5
Ends...
Press any key to continue . . .
```

Şimdi 10 tane thread'in aynı int türden nesneyi 100'er kere artırdığını düşünelim. Toplamda bu değişkenin son değerinin 1000 olması beklenir. Ancak akışlar iç içe geçtiği için durum böyle olmayacaktır:

```
#include <stdio.h>
#include <omp.h>

int main(void)
{
    int count = 0;

    #pragma omp parallel shared(count), num_threads(10)
    {
        int i;

        for (i = 0; i < 100; ++i)
            ++count;
    }
    printf("%d\n", count);

    return 0;
}
```

```
728
Press any key to continue . . .
```

Şimdi artırım işlemini kritik koda alalım:

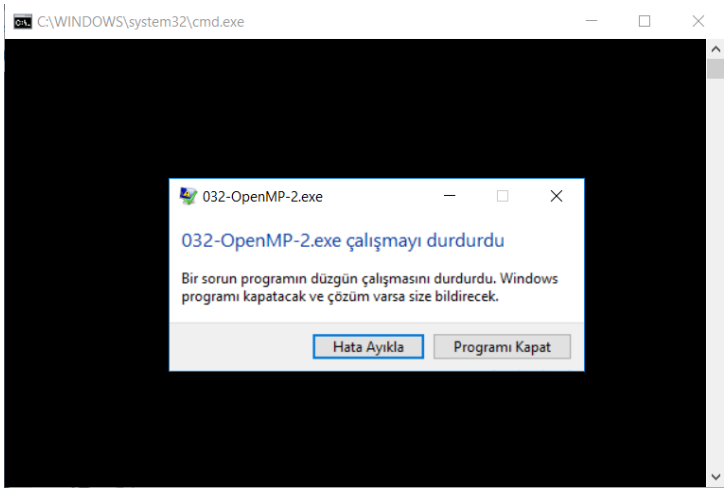
```
#include <stdio.h>
#include <omp.h>

int main(void)
{
    int count = 0;

    #pragma omp parallel shared(count), num_threads(10)
    {
        int i;

        for (i = 0; i < 100; ++i)
            #pragma omp critical
            ++count;
    }
    printf("%d\n", count);

    return 0;
}
```

```
1000
Press any key to continue . . .
```

Örneğin C++'ta aynı vectöre paralel biçimde ekleme yapmak isteyelim:

```
#include <iostream>
#include <vector>
#include <omp.h>

using namespace std;

int main()
{
    vector<int> vect;

    #pragma omp parallel  shared(vect)
    {
        int i = 0;

        for (i = 0; i < 10000; ++i)
            vect.push_back(i);
    }

    cout << vect.size() << endl;

    return 0;
}
```

Burada birden fazla thread vektöre push_back fonksiyonu ile ekleme yapmak istediğinde program muhtemelen çökecektir:



Şimdi kritik kod uygulayalım:

```
#include <iostream>
#include <vector>
#include <omp.h>

using namespace std;

int main()
{
    vector<int> vect;

    #pragma omp parallel shared(vect)
    {
        int i = 0;

        for (i = 0; i < 10000; ++i)
            #pragma omp critical
            vect.push_back(i);
    }

    cout << vect.size() << endl;

    return 0;
}
```

```
80000
Press any key to continue . . .
```

#pragma omp barrier ile paralel blok içerisinde bir bariyer bilinçli olarak oluşturulabilir. Örneğin:

```
#include <stdio.h>
#include <omp.h>

int main(void)
{
    #pragma omp parallel
    {
        int i;

        for (i = 0; i < 100; ++i)
            putchar(i % 26 + 'A');
        #pragma omp barrier

        for (i = 0; i < 100; ++i)
            putchar(i % 10 + '0');
    }
```

```
    return 0;
}
```

Burada kullanılan bariyer nedeniyle paralel blok içerisindeki bütün thread'ler harfleri yazdıktan sonra bekleyecek, sonra sayıları yazmaya başlayacaktır. Tabii burada kritik kod uygulanmadığı için harfler ve sayılar kendi içlerinde thread temelinde iç içe geçebilmektedir:

```
#include <stdio.h>
#include <omp.h>

int main(void)
{
    #pragma omp parallel
    {
        int i;

        for (i = 0; i < 26; ++i)
            putchar(i + 'A');
        #pragma omp barrier

        for (i = 0; i < 10; ++i)
            putchar(i + '0');
    }

    printf("\n");

    return 0;
}
```

```
AAAABBABBCDEFGABACCDEAFGCHBIJCKDEFDGBHCIDJEBFCLDEEFHGHDIKJLKMLMNNFOIPEQFMGCHDEF
GHHIIJRKNLOMINJOGPJHIOJKPLKMQLKMLNOPQQRRSTRUSVTPUQVRWNXSOTPMQSRYSSTTUUVVWWXYZZW
UYJZNOPQRSTUVWKXLYMZNTOPQRVSWTUVUWVXWYXZXYXZYZYZ0010010213245627304152671829891
0343506738945678945361427348565798192345667879 89
Press any key to continue . . .
```

Eğer #pragma omp barrier direktifini kaldırırsak aşağıdakine benzer bir görüntüyle karşılaşırız:

```
AAAAABBCBDAEAFBGHAIBJCKBLBMCNCOBPCQDRCSCTDUDVWXDYZE01D2D3456E7C8E9FGEHEIFJKLDMNO
PQFRESFGGEHFIGJTHHIIFJGHIJKKLMNUOJPFKKLLMMVNQOGPWQRSTLUNVOPQHRXSRTUMVGWHWIIJYKSL
MNNXOYZ0J1KZLTMUNXOYPZQPRQ0RS2TOUVVWWSXTY3ZP0011X2345U6V7WQX2YYZ0124354657687989
89RS3TU451678V923Z4W0X5Y617829Z304152637485967 89
Press any key to continue . . .
```

#pragma omp parallel ve #pragma omp single bloklarının sonlarında default olarak bir bariyer bulunduğunu anımsayınız.

Bir değişkenin artırıldığı, eksiltildiği gibi durumlarda derleyicinin bunu tek bir makine komutuyla yapması zorunlu değildir. Örneğin ++a gibi bir işlem masum gözükse de aslında derleyici tarafından birkaç makine komutuyla yapılıyor olabilir:

```
MOV     reg, a
INC     reg
MOV     reg, a
```

İşte paralel çalışma sırasında bu makine komutları arasında thread'ler arası geçiş söz konusu olabilir. Bu durumda başka bir thread bu nesneyi kullanmaya çalışırsa onun değeri umulduğu gibi artmayabilir. Aslında Intel'de "Sistem Programlama ve İleri Uygulamaları I" kursunda da belirtildiği gibi bazı bdurumlarda bazı makine komutları bile atomik değildir. Bunları atomic yapmak için Intel işlemcilerinde LOCK önekinin kullanıldığını anımsayınız. İşte OpenMP'de #pragma omp atomic direktifi yalın işlemlerin hiç kritik koda

sokulmadan atomic bir biçimde (örneğin LOCK öneki ile) yapılmasını sağlamaktadır. Bu tür durumlarda krtikik kod oluşturmak yerine atomik işlem yapmak performasnı artırmaktadır. Örneğin:

```
#include <stdio.h>
#include <omp.h>

int main(void)
{
    int count = 0;

#pragma omp parallel shared(count), num_threads(10)
    {
        int i;

        for (i = 0; i < 100; ++i)
            #pragma omp atomic
            ++count;

    }
    printf("%d\n", count);

    return 0;
}
```

```
1000
Press any key to continue . . .
```

#pragma omp atomic tek bir ifade için uygulanır. Bu ifadenin yalın operatörlerden oluması gerekir. Ayrıntılı bilgi için OpenMP'nin dokmanlarına başvurrulabilir.

#pragma omp master direktifi ilgili bloğun asıl thread tarafından (yani akışın#pragma omp parallel bloğuna girdiği thread tarafından) gerçekleştirilmesini sağlar. Örneğin:

```
#include <stdio.h>
#include <omp.h>

int main(void)
{
    printf("Master thread: %d\n", omp_get_thread_num());
    #pragma omp parallel
    {
        printf("Common: %d\n", omp_get_thread_num());
        #pragma omp master
        printf("Only master thread: %d\n", omp_get_thread_num());
    }

    return 0;
}
```

```
Master thread: 0
Common: 0
Only master thread: 0
Common: 1
Common: 2
Common: 3
Common: 5
Common: 4
Common: 6
Common: 7
Press any key to continue . . .
```

Daha önceden belirtildiği gibi default olarak #pragma omp parallel ve #pragma omp single direktiflerinin sonunda default bir bariyer vardır. Eğer bu bariyerin bulunması istenmiyorsa bu direktiflere nowait cümlesi eklenir. Örneğin:

```
#include <stdio.h>
#include <omp.h>

int main(void)
{
    #pragma omp parallel
    {
        int i;

        for (i = 0; i < 26; ++i)
            putchar(i + 'A');

        #pragma omp single
        {
            putchar('?');
        }
        /* Burada default bir bariyer var */
        for (i = 0; i < 10; ++i)
            putchar(i + '0');
    }

    printf("\n");

    return 0;
}
```

```
ABABACCDDAEBAABECBDCECFAGHIBCFDEFDGHDIJKLMNOFPQJKELMNEOGPHQRRSFTUVFWGXYZABSCTGHB
IJHKLM?NOPUQCRGSHTIUVWVXDYZIJIKJLKMJWKDLMENLOPQRSTUEVFWMXNOPQFRSGTHUVYWXZYGZHNII
JKLOMNOXPQRJSTUPVKWXYZLMYNQOPRQZRSSTTUUVWXVYWXZYZ000012345678912031405160718192 3
041231456789526789232435263748945567384956786978 9
Press any key to continue . . .
```

Burada #pragma omp single diretifinin onunda default bir bariyer olduğuna dikkat ediniz. Şimdi direktifin sonuna nowait cümleciğini ekleyelim:

```
#include <stdio.h>
#include <omp.h>

int main(void)
{
    #pragma omp parallel
    {
        int i;

        for (i = 0; i < 26; ++i)
            putchar(i + 'A');

        #pragma omp single nowait
        {
            putchar('?');
        }
        /* Artık burada bariyer yok */
        for (i = 0; i < 10; ++i)
            putchar(i + '0');
    }

    printf("\n");

    return 0;
```

```
}
```

```
ABCDEAAFABBGCADBECABABCCDEFGFHIJDKCLMNBOHPGQRSTDUDVEWIXJHKIAJBECFDCEFFGLHKIJEKLD
MNGHLMINFOPYQRSTUVWXYZ?EFGHIJGHIJKLZ01234506G78M9JKLMNOPQRSTUVWXYZM01122K3L45O67
H8934N56O7N89IJPKQLRSTOUMVPWMQPRXSNTONPOQYPZQ0U1234Q5RRSTVUVW6XSYWZR07S8T9123456
U7X89VWYXYTZU0Z12V34W56078X91Y2Z340516234758697 89
Press any key to continue . . .
```

Bariyerin kalktığına dikkat ediniz.

#pragma omp sections direktifi #pragma omp section direktiflerinden oluşur. Bu direktifler ilgili #pragma omp section bloklarının her birinin ayrı thread'ler tarafından çalıştırılacağını belirtir. Yani bu bloklar toplamda yine single bloğunda olduğu gibi yalnızca bir kez çalıştırılmaktadır. Tabii #pragma omp sections direktifinin de yine #prama omp parallel direktifi içerisinde bulunması uygundur. Aksi takdirde çok thread'li çalıştırma zaten yapılmaz. Örneğin:

```
#include <stdio.h>
#include <omp.h>

int main(void)
{
    #pragma omp parallel
    {
        printf("Common: %d/%d\n", omp_get_thread_num(), omp_get_num_threads());

        #pragma omp sections
        {
            #pragma omp section
            {
                printf("Only one thread section 1: %d/%d\n", omp_get_thread_num(), omp_get_num_threads());
            }

            #pragma omp section
            {
                printf("Only one thread section 2: %d/%d\n", omp_get_thread_num(), omp_get_num_threads());
            }

            #pragma omp section
            {
                printf("Only one thread section 3: %d/%d\n", omp_get_thread_num(), omp_get_num_threads());
            }
        }
    }

    return 0;
}
```

```
Common: 1/8
Common: 5/8
Common: 3/8
Only one thread section 3: 3/8
Common: 4/8
Common: 0/8
Only one thread section 1: 1/8
Common: 6/8
Only one thread section 2: 5/8
Common: 7/8
Common: 2/8
Press any key to continue . . .
```

#pragma omp section direktiflerinin sonunda default bir bariyer olmadığına dikkat ediniz. Pekiyi #pragma omp single direktifi ile #pragma omp sections direktifleri arasında ne fark vardır? İki direktifte de söz konusu blok

yalnızca bir kez çalıştırılmıyor mu? İşte #pragma omp sections direktifinde biz OpenMP'ye söz konusu section'ların paralel çalıştırılması yönünde bir ricada bulunuruz. OpenMP bu section'ları ayrı thread tarafından çalıştırabilir ya da bizim ricamızı dikkate almayıp bunların bazılarını ya da hepsini aynı thread tarafından çalıştırabilir. Oysa #pragma omp single direktifinde bizim OpenMP'ye söylediğimiz şey ilgili bloğun yalnızca bir kez çalıştırılmasıdır. Birkaç single bloğunu peş peşe koyduğumuzda OpenMP bunların farklı thread'ler atarfından yapılmasına gayret etmez. Yukarıdaki örneği single blokları ile gerçekleştirelim:

```
#include <stdio.h>
#include <omp.h>

int main(void)
{
    #pragma omp parallel
    {
        printf("Common: %d/%d\n", omp_get_thread_num(), omp_get_num_threads());

        #pragma omp single nowait
        {
            printf("Only one thread section 1: %d/%d\n", omp_get_thread_num(), omp_get_num_threads());
        }

        #pragma omp single nowait
        {
            printf("Only one thread section 2: %d/%d\n", omp_get_thread_num(), omp_get_num_threads());
        }

        #pragma omp single nowait
        {
            printf("Only one thread section 3: %d/%d\n", omp_get_thread_num(), omp_get_num_threads());
        }
    }

    return 0;
}
```

```
Common: 0/8
Only one thread section 1: 0/8
Common: 6/8
Common: 7/8
Common: 4/8
Common: 1/8
Common: 5/8
Common: 2/8
Common: 3/8
Only one thread section 2: 5/8
Only one thread section 3: 5/8
Press any key to continue . . .
```

#pragma omp sections direktifinin sonunda default bir bariyer vardır. Ancak bunun içerisindeki #pragma omp section direktiflerinde default bir bariyer yoktur.

Paralel for döngüleri eğer paralel iki dizi üzerinde işlem yapıyorsa ve işlemci mimarisi de buna uygunsa OpenMP'ye işlemleri SIMD (Single Instruction Multiple Data) komutlarıyla yapması konusunda tavsiyede bulunulabilir. Bu işlem #pragma omp simd direktifiyle yapılmaktadır. Örneğin:

```
#include <stdio.h>
#include <omp.h>
```

```
int main(void)
{
    int i;
    int a[10] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
    int b[10] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
    int c[10];

    #pragma omp simd
    for (i = 0; i < 10; ++i)
        c[i] = a[i] * b[i];

    for (i = 0; i < 10; ++i)
        printf("%d ", c[i]);
    printf("\n");

    return 0;
}
```

Visuals Studio IDE'sinin kullandığı C ve C++ derleyicilerinin OpenMP desteğinin versiyonu 2.0'dır. Bazı OpenMP direktifleri OpenMP'nin sonraki versiyonlarında eklenmiştir. Bu nedenle #pragma omp simd direktifi mevcut Microsoft derleyicilerinde tanınmamaktadır. Bu denemeyi Qt Creator IDE'sinde (MinGW ya da gcc ya da clang) test edebilirsiniz.

Anahtar Notlar: QtCreator IDE'sinde OpenMP seçeneğini açmak için QMake dosyasına (.pro dosyasına) aşağıdaki satırları eklemelisiniz:

```
QMAKE_CFLAGS += -fopenmp
QMAKE_CXXFLAGS += -fopenmp
LIBS += -fopenmp
```

SIMD komutları modern mikroişlemcilerin çoğunda bulunmaktadır. Bunlar tek bir komutla bir grup paaralel işlem yapmayı sağlamaktadır. Örneğin her biri double türünden olan 8'erlik sayıları karşılıklı tek bir SIMD makine komutuyla çarpabiliriz.

Paralel Programlama Hangi Tür Uygulamalarda Tercih Edilmektedir?

Paralel programlama CPU yoğun (CPU bound) ve özellikle çok yoğun ve fazla miktarda işlemlerin yapıldığı uygulamalarda işlem süresin, azaltmak için kullanılan bir tekniktir. Bu tekniğin yersiz kullanılması toplam performansı artıracağına tam tersi azaltabilmektedir. Örneğin az miktarda bir döngüyü paralel yapmanın bir faydası olmayacağı gibi tam tersine zaman bakımdan zarar da oluşturabilir:

```
#include <stdio.h>
#include <Windows.h>
#include <omp.h>

int main(void)
{
    long long int i;
    LARGE_INTEGER li1, li2, freq;
    __int64 result;

    QueryPerformanceFrequency(&freq);

    QueryPerformanceCounter(&li1);

#pragma omp parallel for
    for (i = 0; i < 1000LL; ++i) {
        /* ... */
    }

    QueryPerformanceCounter(&li2);

    result = li2.QuadPart - li1.QuadPart;
```

```
    printf("%f\n", (double)result / freq.QuadPart);

    QueryPerformanceCounter(&li1);

    for (i = 0; i < 1000LL; ++i) {
        /* ... */
    }

    QueryPerformanceCounter(&li2);

    result = li2.QuadPart - li1.QuadPart;
    printf("%f\n", (double)result / freq.QuadPart);

    return 0;
}
```

```
0.001546
0.000002
```

Bunun nedeni ne olabilir? Paralel programlaama için arka planda thread'ler yaratılıp duruma göre yok edilmektedir. Tüm bunların da bir zaman maliyeti vardır. Eğer döngü uzun olsaydı thread'lerin yaratılması ve yok edilmesinin maliyeti buna değebilirdi.

Paralel programlamanın uygun kullanımına ilişkin bazı örnekler şunlar olabilir:

- Yoğun matematiksel işlemler gereken yerlerde. Örneğin büyük matrislerin çarpımı, nümerik analiz işlemleri vs.

- Optimizasyon algoritmaları ve sezgisel yöntemler. Örneğin NP (Non polynomial) karmaşıklıktaki optimizasyon algortimaları. Gezgiz satıcı problemi, çizelgeleme problemleri, derleyicilerin arka yüz optimizasyonları vs.

- Bazı görüntü işleme faaliyetlerinde.

- Bazı sinyal işleme faaliyetlerinde

- Bazı yapay zeka uygulamalarında. Örneğin satranç programları. Uzman sistemler vs.

- Graf analiz algoritmalarında.

- Bazı gerçek zamanlı uygulamalarda.

- Büyük verilerin (big data) işlenmesinde

- Şifre kırma gibi özel bazı yoğun işlem gerektiren alanlarda.

Çalıştırılabilen Dosya Formatları (Executable File Formats)

Bir program derlenip bağlandıktan sonra çalıştırılabilir hale getirilmiş olur. İşletim sistemi çalıştırılabilir dosyaları diskten alarak onun içerisindeki bilgilerden hareketle onları belleğe yükler ve çalıştırır. Çalıştırılabilen (executable) dosyaların içerisinde yalnızca makine kodları yoktur. Bunların içerisinde static veriler (static data), yükleme için gereken meta veriler de (meta data) bulunmaktadır. Sistem programcısının bu formatları belli düzeyde bilmesi bazı süreçleri yorumlayabilmesi için gerekmektedir. Aynı zamanda bilgisayar virüslerinin, zararlı yazılımların (malware) nasıl devreye girdikleri konusunda da bu bilgilerden faydalanılabilmektedir. Şüphesiz derleyici ve bağlayıcı gibi programların yazımında, kopya koruma sistemlerin gerçekleştirilmesinde çalıştırılabilen dosya formatının iyi bir biçimde bilinmesi gerekmektedir.

Tarihsel olarak pek çok sistemde pek çok çalıştırılabilen dosya formatı kullanılmıştır. Microsoft firmasının DOS işletim sisteminde MZ formatı kullanıyordu (Gerçekten de bu çalıştırılabilen dosyaların ilk iki karakteri MZ harfleriyle başlamaktadır. MZ ismi bu dosya formatını taralayan Mark Zibikovski'den gelmektedir). Microsoft 16 bit Windows sistemlerinde (Windows 3.0, 3.1, 3.11) NE (New Executable) isimli yeni bir çalıştırılabilen dosya formatına geçmiştir. Nihayet 32 bit Windows sistemleriyle Microsoft PE (Portable Executable) formata geçmiştir. PE formatının 32 bit ve 64 bitlik iki ayrı versiyonu vardır.

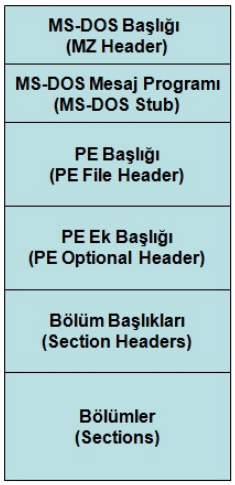
UNIX sistemlerinde ilk kullanılan çalıştırılabilen dosya formatı a.out formatıydı. Sonra bazı sistemler COFF (Common Object File Format) isimli formatı kullandılar. Bu format her ne kadar amaç dosya (object file) formatı olarak biliniyorsa da aynı zamanda çalıştırılabilen de bir formattr. Microsft'un PE formatının temeli buraya dayanmaktadır. Bugün UNIX/Linux sistemleri ağırlıklı olarak ELF (Executable and Linkable Format) denilen formatı kullanmaktadır. Ancak pek çok Unix türevi sistem hale eski formatları da desteklemektedir. ELF formatı ayı zamanda bir amaç dosya (object file) formatıdır. Aslında aynı durum PE formatı için de geçerlidir. Ancak biz kursumuzun bu bölümünde yalnızca çalıştırılabilen dosyalar üzerinde duracağız. Amaç dosyaların formatı için belli düzeyde semblik makine dilinin bilinmesi gerekmektedir.) ELF formatının da 32 bit ve 64 bit versiyonları vardır. Genel olarak hem PE hem de ELF formatı için şunlar söylenebilir: 64 bit işlem sistemleri 32 bit formatları da yükleyerek çalıştırabilmektedir. Ancak 32 bit işletim sistemleri yalnızca 32 bit formatları yükleyerek çalıştırır.

Mac OS sistemleri tarihsel süreç içerisinde çeşitli çalıştırılabilen dosya formatları kullanmıştır. Bugün MAc OS X sistemlerinde kullanılan dosya formatona Mach-O formatı denilmektedir. Bu format bazı bakımlardan a.out formatına bazı bakımlardan da ELF formatına benzemektedir. Mach-O formatı da aslında hem bir amaç kod formatı hem de çalıştırılabilen dosya formatıdır.

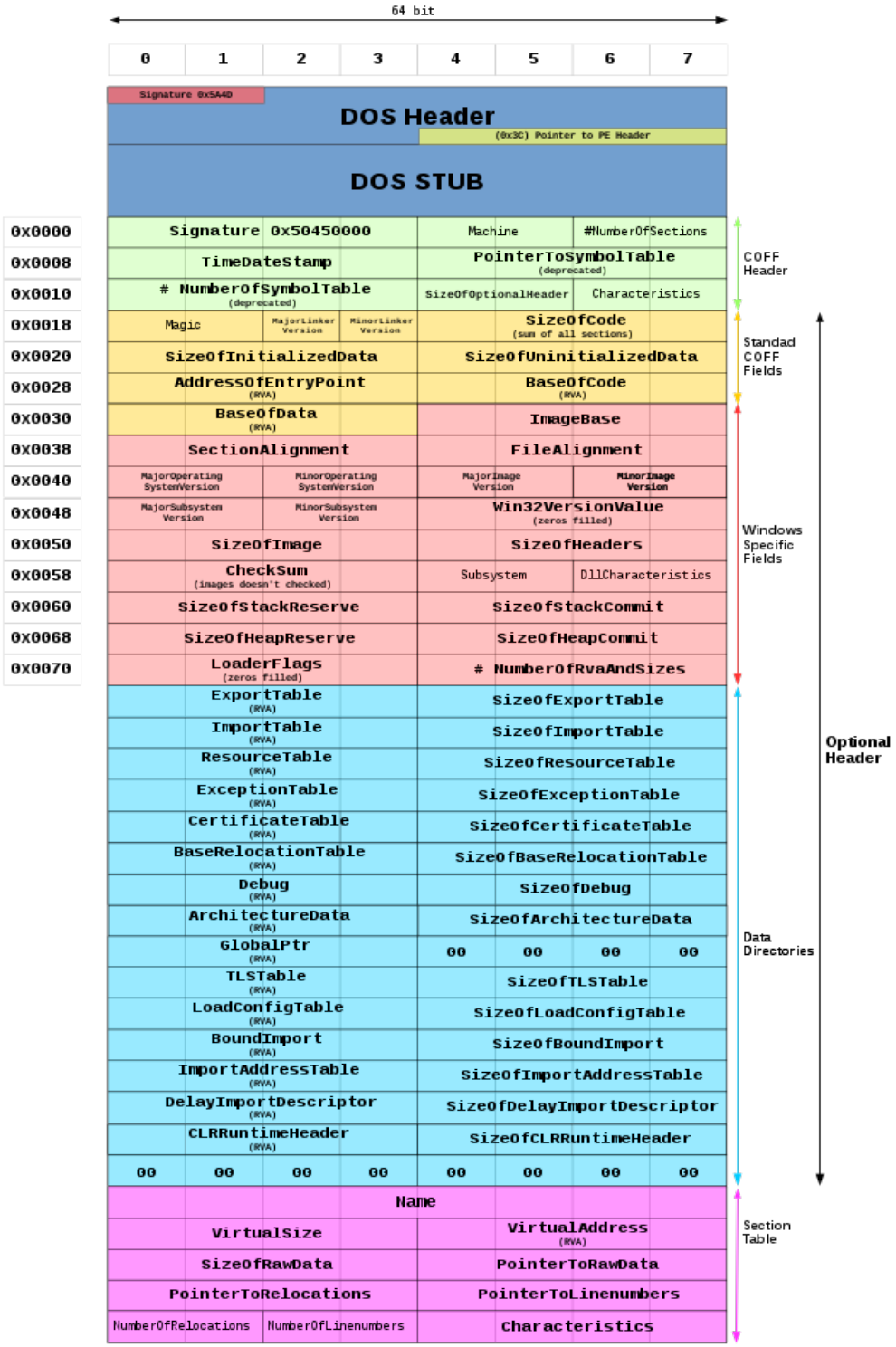
PE (Protable Executable) Dosya Formatı

PE formatı genişletilebilir bir yapıya sahiptir. Format bölümlerden (sections) oluşur. Her bölümde bazı bilgiler bulunmaktadır. (Örneğin fonksiyonların makine kodları .text isimli bir bölümde ilkdeğer verilmiş global nesneler .data isimli bir bölümde ilkdeğer verilmemiş global nesneler ise .bss isimli bir bölümde bulunmaktadır.) PE formatının hemen başında küçük bir DOS programı bulunur. Bu program belki eskiden anlamlıydı. Ancak artık bir öneminin kalmadığı söylenebilir. Eğer PE dosyası yanlışlıkl DOS sisteminde çalıştırılmak istenirse buradaki program "This program cannot be run in DOS" gibi bir mesajı ekraana basmaktadır. Dosya Windows tarafından yüklenirken bu DOS başlığı pas geçilmektedir. Bugün zaten artık Windows'un son versiyonları ancak belirli koşullarda ve yardımcı araçların yüklenmesiyle DOS programlarını çalıştırmaktadır.

PE dosyası iki başlığa sahiptir. Başlık kısmı aşağı yukarı DOS programının bittiği yerden başlar (kesin başlangıç yeri DOS başlığında bulunmaktadır) Bu ilk başlığa "PE File Header (PE Dosya Başlığı)" denilmektedir. İkinci başlık her ne kadar "PE Optional Header (PE İsteğe Bağlı Başlık)" biçiminde isimlendirilmişse de aslında zorunlu olarak bulunmak durumundadır. Buradaki "optional (isteğe bağlı)" sözcüğü ek bilgiler anlamında düşünülmüştür. PE dosya formatının genel yapısı şöyledir:



Daha ayrıntılı şekille PE formatı şöyle ifade edilebilir:



Dosya formatlarını analiz eden programların dosyayı “bellekte tabanlı (memory mapped)” biçi,mde açması uygun olur. Böylelikle dosya üzerinde ileri geri gitme işlemleri vew okuma işlemleri çok kolay paılabilmektedir. “Bellek Tabanlı Dosyalar” Derneğimizde “Sistem Programlama ve İleri C Uygulamaları - I” kursunda ele alınmamakatdır.

PE formatının bölümlerinin içeriği Microsoft C/C++ derleyicileri için <winnt.h> dosyasında bulundurulmaktadır. Örneğin PE dosyasının başındaki MS-DOS Başlığı <winnt.h> içerisinde

IMAGE_DOS_HEADER ismiyle bulunmaktadır. <winnt.h> dosyası zaten <windows.h> dosyası içerisinden include edilmektedir.
Örnekler için PE dosyasını bellek abanlı olarak açan aşağıdaki gibi bir iskelet programdan faydalanabiliriz:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <windows.h>

/* Symbolic Constants */

#define MZ_MAGIC   0x5A4D

/* Function Prototypes */

void ExitSys(LPCSTR lpszMsg, int status);
void ExitUsr(LPCSTR lpszMsg, int status);

/* Global Data Definitions */

IMAGE_DOS_HEADER *g_dosHeader;
void *g_imageAddr;

/* Function Definitions */

int main(int argc, char *argv[])
{
    HANDLE hFile, hFileMapping;
    DWORD fileSize;

    if (argc != 2)
        ExitUsr("wrong number of arguments!..\n", EXIT_FAILURE);

    if ((hFile = CreateFile(argv[1], GENERIC_READ | GENERIC_WRITE,
        FILE_SHARE_READ | FILE_SHARE_WRITE, NULL, OPEN_EXISTING, 0, NULL)) == INVALID_HANDLE_VALUE)
        ExitSys("Cannot open file", EXIT_FAILURE);

    hFileMapping = CreateFileMapping(hFile, NULL, PAGE_READWRITE, 0, 0, NULL);
    if (hFileMapping == NULL)
        ExitSys("CreateFileMapping", EXIT_FAILURE);

    g_imageAddr = MapViewOfFile(hFileMapping, FILE_MAP_READ | FILE_MAP_WRITE, 0, 0, 0);
    if (g_imageAddr == NULL)
        ExitSys("MapViewOfFile", EXIT_FAILURE);

    /* .... */

    UnmapViewOfFile(g_imageAddr);
    CloseHandle(hFileMapping);
    CloseHandle(hFile);

    return 0;
}

void ExitSys(LPCSTR lpszMsg, int status)
{
    DWORD dwLastError = GetLastError();
    LPTSTR lpszErr;

    if (FormatMessage(FORMAT_MESSAGE_ALLOCATE_BUFFER | FORMAT_MESSAGE_FROM_SYSTEM, NULL, dwLastError,
        MAKELANGID(LANG_NEUTRAL, SUBLANG_DEFAULT), (LPTSTR)&lpszErr, 0, NULL)) {
        fprintf(stderr, "%s: %s", lpszMsg, lpszErr);
        LocalFree(lpszErr);
    }
```

```
    exit(status);
}

void ExitUsr(LPCSTR lpszMsg, int status)
{
    fprintf(stderr, "%s\n", lpszMsg);
    exit(status);
}
```

MS-DOS Başlığı

Yukarıda da belirtildiği gibi PE dosyalarının hemen başında küçük bir MS-DOS çalıştırılabilen programı vardır. Dolayısıyla o programında MZ formatında bir başlık kısmı bulunur. MS-DOS başlığı, PE formatı yanlışlıkla DOS ortamında çalıştırılırsa ekrana "This program cannot be run in DOS mode" gibi bir uyarı yazısını ekrana yazdıran basit bir DOS programın başlığını içermektedir. Bu programın bulundurulması gelenekseldir ve artık DOS sistemlerinin büyük ölçüde ortadan kalktığı bu günlerde önemli bir işleve sahip değildir. Ancak MS-DOS başlığı hala PE dosya formatının hemenbaşında bulunmak zorundadır. Microsoft'un <winnt.h> başlık dosyası içerisinde MS-DOS başlığı IMAGE_DOS_HEADER isimli yapıyla ifade edilmiştir:

```
typedef struct _IMAGE_DOS_HEADER {        // DOS .EXE header
    WORD   e_magic;                       // Magic number
    WORD   e_cblp;                        // Bytes on last page of file
    WORD   e_cp;                          // Pages in file
    WORD   e_crlc;                        // Relocations
    WORD   e_cparhdr;                     // Size of header in paragraphs
    WORD   e_minalloc;                    // Minimum extra paragraphs needed
    WORD   e_maxalloc;                    // Maximum extra paragraphs needed
    WORD   e_ss;                          // Initial (relative) SS value
    WORD   e_sp;                          // Initial SP value
    WORD   e_csum;                        // Checksum
    WORD   e_ip;                          // Initial IP value
    WORD   e_cs;                          // Initial (relative) CS value
    WORD   e_lfarlc;                      // File address of relocation table
    WORD   e_ovno;                        // Overlay number
    WORD   e_res[4];                      // Reserved words
    WORD   e_oemid;                       // OEM identifier (for e_oeminfo)
    WORD   e_oeminfo;                     // OEM information; e_oemid specific
    WORD   e_res2[10];                    // Reserved words
    LONG   e_lfanew;                      // File address of new exe header
} IMAGE_DOS_HEADER, *PIMAGE_DOS_HEADER;
```

MS-DOS başlığını hemen sözü edilen küçük DOS programı izler. Bu bölümü de PE dosya formatının başlık kısmı (PE File Header) izlemektedir. Image File Header, PE formatına ilişkin birincil derecede önemli parametrik bilgileri tutmaktadır. Bu başlık <winnt.h> dosyası içerisindeki IMAGE_FILE_HEADER yapısıyla temsil edilmiştir.

PE Dosya Başlığı (PE File Header)

IMAGE_FILE_HEADER başlığı MS-DOS başlığından hemen sonra gelmek zorunda değildir. Bu başlığın yeri MS-DOS başlığının sonundaki e_lfanew elemanında (dosyanın 0x3C offsetinde) belirtilmektedir. (Başka bir deyişle bu başlık dosyada MS-DOS başlığının e_lfanew elemanıyla belirtilen offset'indedir.) Dosya bellek tabanlı olarak açıldığında başlığın yerini belirlemek amacıyla dosyanın yüklenme adresine e_lfanew elemanındaki değeri toplamamız gerekir. Ancak burada önemli bir ayrıntıyı açıklamak istiyoruz: Dosyanın e_lfanew elemanının belirttiği offset'te hemen IMAGE_FILE_HEADER yapısı bulunmamaktadır. Bu yapı bu

offset'ten 4 byte ileridedir. e_lfanew ile belirtilen offsette önce P ve E harflerinin ASCII tablosundaki sayısal karşılıkları (sırasıyla 0x50 ve 0x45 byte'ları) ve sonra da iki tane sıfır byte'ı bulunur. Bu byte'lar dosyanın geçerliliğini sınamak amacıyla kullanılan sihirli bir sayı (magic number) işlevini görürler. Programcı dosyanın PE formatına ilişkin olup olmadığını buradan anlayabilir. Örnek programımızda başlığın yeri şöyle hesaplanmıştır:

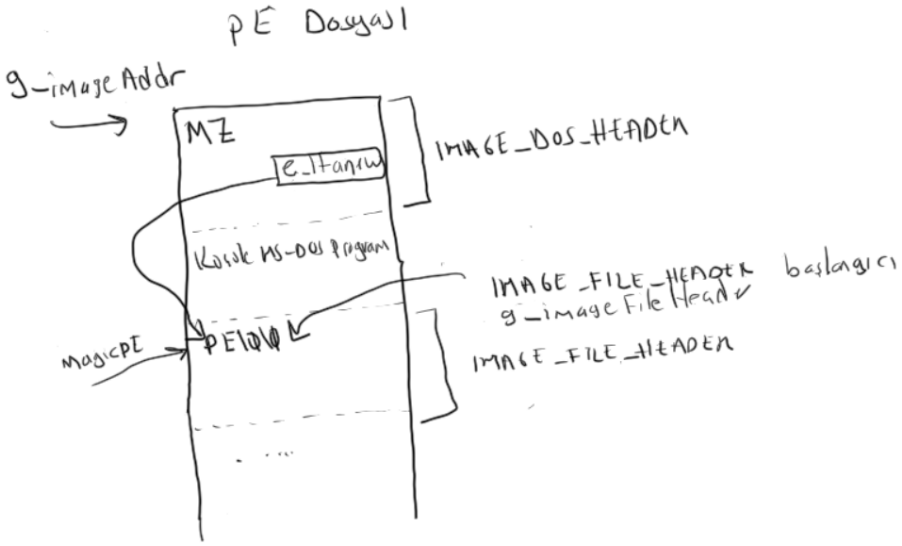
```
DWORD *magicPE;
...
magicPE = (DWORD *)((BYTE *)g_imageAddr + g_dosHeader->e_lfanew);
    if (*magicPE != 0x00004550)       /* PE\0\0 */
        ExitUsr("Invalid PE file!", EXIT_FAILURE);

g_imageFileHeader = (IMAGE_FILE_HEADER *)(magicPE + 1);
printf("Success\n");
```

Özetle şimdiye kadar yapılan tespitler şunlardır:

1) PE dosyasının hemen başında küçük bir MS-DOS programı vardır. Bu program bir MZ başlığına sahiptir. Bu nedenle PE dosyasının ilk byte'ı MZ harfleriyle başlamak zorundadır.

2) PE dosyasının başındaki küçük DOS programından sonra PE'nin asıl başlık kısmı olan "PE File Header (IMAGE_FILE_HEADER) gelmektedir. Ancak bu başlığın nereden başladığı offset olarak MS-DOS başlığındaki (IMAGE_DOS_HEADER) e_lfanew elemanında yazmaktadır.

3) MSDOS başlığındaki e_lfanew elemanı aslında "PE File Header"dan 4 byte önceki sihirli sayının (magic number) yerini gösterir. Yani asıl başlık bu e_lfanew elemanının belirttiği offset'ten 4 byte ileridedir.



"PE File Header (IMAGE_FILE_HEADER(" yapısının elemanları şöyledir:

Yapının elemanları aşağıdaki gibidir:

```
typedef struct _IMAGE_FILE_HEADER {
    WORD    Machine;
    WORD    NumberOfSections;
    DWORD   TimeDateStamp;
    DWORD   PointerToSymbolTable;
    DWORD   NumberOfSymbols;
    WORD    SizeOfOptionalHeader;
    WORD    Characteristics;
```

```
} IMAGE_FILE_HEADER, *PIMAGE_FILE_HEADER;
```

Yapının Machine elemanı programın çalıştırılacağı hedef mimariyi belirtir. Windows sistemleri aslında yalnızca Intel'in x86 ailesinde değil ARM gibi (Windows CE) PowerPC gibi aileleri için de gerçekleştirilmiştir. Burada tipik olarak 0x014C değerini görebiliriz. Bu değer hedef makinenin Intek 366 ve sonrası olacağını beelirtmektedir. Bu makine değerleri <winnt.h> içerisinde (<windows.h> içerisinde) IMAGE_FILE_MACHINE_XXXX biçiminde define edilmiştir. Örneğin 0x014C değeri IMAGE_FILE_MACHINE_I386 olarak define edilmiş durumdadır. 64 Bit uygulamalarda bu değer tipik olarak 0x8664 biçimindedir. Bu da IMAGE_FILE_MACHINE_AMD64 sembolik sabitiyle define edilmiştir. Bu Machine değeri işletim sisteminin yükleyicisi (loader) tarafından daha ilk aşamada kontrol edilmektedir. (Örneğin 32 bit Windows sistemlerin de yükleyici burada IMAGE_FILE_MACHINE_AMD64 değerini görürse programı hiç yüklemez. Ancak tabii 64 bit Windows sistemlerinin hem 32 bit hem de 64 bit programları yükleyerek çalıştırabildiğini anımsayınız.)
Yapının NumberOfSections elemanında PE dosyası içerisinde kaç tane bölümün (sections) olduğu bilgisi vardır. PE dosyasnın bölümleri ileride ele alınacaktır.

Yapının TimeDateStamp elemanı dosyanın yaratımına ilişkin 01.01.1970 tarhinden geçen saniye sayısını vermektedir. Yapının PointerToSymbolTable elemanı amaç dosyalar için anlamlıdır. PE çalıştırılabilen dosyalarında bu alanda 0 değerinin olması gerekir. Benzer biçimde yapının NumberOfSymbols elemanı da PE çalıştırılabilen dosyalarında 0 değerini alır.

Yapının SizeofOptionalHeader elemanı "PE Başlığından (PE File Header)" sonra gelen "PE Ek Başlığı (PE Optional Header)" kısmının byte uzunluğunu belirtmektedir. Çünkü PE Ek Başlığının byte uzunluğu değişebilmektedir.

Yapının Characteristics isimli elemanı bit bit kodlanmış bayraklardan oluşur. Bu eleman PE dosyasının nasıl oranize edildiğini ve içeriğinin ne olduğunu betimlemektedir. Bilindeiği gibi Windows'ta COFF amaç dosya formatı da aslında bir çeşit PE formatıdır. Dinamik yüklenen kütüphaneler (DLL'ler) ve aygıt sürücü dosyaları da aslında PE formatına sahiptir. Buradaki Characteristics bayrakları şunlardan biri olabilir:

Flag	Value	Description
IMAGE_FILE_RELOCS_STRIPPED	0x0001	Image only, Windows CE, and Microsoft Windows NT® and later. This indicates that the file does not contain base relocations and must therefore be loaded at its preferred base address. If the base address is not available, the loader reports an error. The default behavior of the linker is to strip base relocations from executable (EXE) files.
IMAGE_FILE_EXECUTABLE_IMAGE	0x0002	Image only. This indicates that the image file is valid and can be run. If this flag is not set, it indicates a linker error.
IMAGE_FILE_LINE_NUMS_STRIPPED	0x0004	COFF line numbers have been removed. This flag is deprecated and should be zero.
IMAGE_FILE_LOCAL_SYMS_STRIPPED	0x0008	COFF symbol table entries for local symbols have been removed. This flag is deprecated and should be zero.
IMAGE_FILE_AGGRESSIVE_WS_TRIM	0x0010	Obsolete. Aggressively trim working set. This flag is deprecated for Windows 2000 and later and must be zero.
IMAGE_FILE_LARGE_ADDRESS_ AWARE	0x0020	Application can handle > 2-GB addresses.
	0x0040	This flag is reserved for future use.
IMAGE_FILE_BYTES_REVERSED_LO	0x0080	Little endian: the least significant bit (LSB) precedes the most significant bit (MSB) in memory. This flag is deprecated and should be zero.
IMAGE_FILE_32BIT_MACHINE	0x0100	Machine is based on a 32-bit-word architecture.
IMAGE_FILE_DEBUG_STRIPPED	0x0200	Debugging information is removed from the image file.
IMAGE_FILE_REMOVABLE_RUN_ FROM_SWAP	0x0400	If the image is on removable media, fully load it and copy it to the swap file.
IMAGE_FILE_NET_RUN_FROM_SWAP	0x0800	If the image is on network media, fully load it and copy it to the swap file.
IMAGE_FILE_SYSTEM	0x1000	The image file is a system file, not a user program.
IMAGE_FILE_DLL	0x2000	The image file is a dynamic-link library (DLL). Such files are considered executable files for almost all purposes, although they cannot be directly run.
IMAGE_FILE_UP_SYSTEM_ONLY	0x4000	The file should be run only on a uniprocessor machine.
IMAGE_FILE_BYTES_REVERSED_HI	0x8000	Big endian: the MSB precedes the LSB in memory. This flag is deprecated and should be zero.

RVA (Relative Virtual Address) ve VA (Virtual Address) Kavramı

PE formatı bölümlerden (sections) oluşur. Dosya yükleyici tarafından belleğe yüklenirken bu bölümler yeniden sıralanıp belirli değerlerin katlarına hizalanmaktadır. Ayrıca işletim sistemi PE formatını hemen belleğin başından itibaren yüklemek zorunda da değildir. PE formatının yükleme adresi yine formatın kendi içerisinde bulundurulmaktadır. İşte RVA (Relative Virtual Address) PE formatı yüklendikten sonra format içerisindeki belli bir byte'ın yükleme adresinden itibaren göreli uzaklığını belirten bir terimdir. Yani biz belli bir byte'ın RVA'sını biliyorsak bu değere formatın yüklendiği adresi eklediğimizde ilgili byte'ın gerçek doğrusal adresini (linear address) elde etmiş oluruz. Dosya yüklendikten sonra ilgili byte'ın doğrusal adresine VA (Virtual Address) de denilmektedir. Örneğin format içerisinde belli bir bölgenin adresi bize RVA olarak verilmiş olsun. Bunun anlamı şudur: Bu program işletim sistemi tarafından yüklendiğinde ilgili bölge yükleme adresinden itibaren o kadar byte uzaklıkta olacaktır. RVA ile VA arasındaki fark şöyledir: RVA bize ilgili byte'ın yükleme adresinden itibaren uzaklığını verirken VA ise belleğin tepesinden (sıfır adresinden) itibaren uzaklığını verir.

PE formatı içerisinde bazı özel bölümlerin yerleri hep RVA biçiminde verilmiştir. Burada unutulmaması gereken nokta şudur: İlgili byte'ın RVA'sı ile o byte'ın dosya offset'i aynı olmak zorunda değildir. Çünkü yükleme sırasında birtakım ayarlamalar ve hizalamalar yapılmaktadır. Bu nedenle imajı belleğe yüklemeden

doğrudan dosya üzerinde işlem yapacaksak ilgili byte'ın RVA'sını dosya offset'ine dönüştürmemiz gerekir. Pekiyi bir RVA verildiğinde biz onun dosya offset'ini nasıl elde edebiliriz? İşte verilen RVA bir bölüm içerisindedir ve PE formatının kendi içerisinde her bölümün hangi RVA'dan ve dosya offset'inden başladığı belirtilmektedir. O halde bir RVA'nın dosya offset'ine dönüştürülmesi için şunlar yapılabilir:

1) İlgili RVA'nın hangi bölüm içerisinde olduğu bulunur. (Yukarıda da belirtildiği gibi bölümlerin başlangıç RVA'ları ve uzunlukları, ayrıca onların dosyanın hangi offset'inden başladığı bilgisi PE formatının içerisinde yazmaktadır.)

2) İlgili byte'ın RVA'sı onun içinde bulunduğu bölümün başlangıç RVA'sından çıkartılır. Böylece dosya offset'i bulunacak olan byte'ın ilgili bölümün hangi offset'inde olduğu hesaplanır. Nihayet hesaplanan bu değere ilgili bölümün dosya offset'i eklenir.

RVA'yı dosya offsetine dönüştüren fonksiyon ileride ele alınacaktır.

PE Ek Başlığı (Image Optional Header)

Her ne kadar bu başlığın İngilizcesi sanki isteğe bağlı izlenimini veriyorsa da aslında durum böyle değildir. Bu başlık her zaman PE dosya başlığından sonra bulumak zorundadır. Ancak bu başlığın uzunluğu değişebilir. Anımsanacağı gibi başlığın uzunluğu PE Dosya Başlığında (Image File Header) belirtilmekteydi. 32 bit ve 64 bit PE formatlarında bu alandaki elemanların uzunlukları farklı olabilmektedir. 32 bit PE Ek başlığı <winnt.h> dosyası (<windows.h> dosyası) içerisinde IMAGE_OPTIONAL_HEADER32 ismiyle bildirilmiştir. Bu yapı 32 bit sistemlerde aynı zamanda IMAGE_OPTIONAL_HEADER ismiyle de typedef edilmiştir.

```
typedef struct _IMAGE_OPTIONAL_HEADER {
    //
    // Standard fields.
    //

    WORD    Magic;
    BYTE    MajorLinkerVersion;
    BYTE    MinorLinkerVersion;
    DWORD   SizeOfCode;
    DWORD   SizeOfInitializedData;
    DWORD   SizeOfUninitializedData;
    DWORD   AddressOfEntryPoint;
    DWORD   BaseOfCode;
    DWORD   BaseOfData;

    //
    // NT additional fields.
    //

    DWORD   ImageBase;
    DWORD   SectionAlignment;
    DWORD   FileAlignment;
    WORD    MajorOperatingSystemVersion;
    WORD    MinorOperatingSystemVersion;
    WORD    MajorImageVersion;
    WORD    MinorImageVersion;
    WORD    MajorSubsystemVersion;
    WORD    MinorSubsystemVersion;
    DWORD   Win32VersionValue;
    DWORD   SizeOfImage;
    DWORD   SizeOfHeaders;
    DWORD   CheckSum;
    WORD    Subsystem;
    WORD    DllCharacteristics;
    DWORD   SizeOfStackReserve;
    DWORD   SizeOfStackCommit;
    DWORD   SizeOfHeapReserve;
```

```
    DWORD   SizeOfHeapCommit;
    DWORD   LoaderFlags;
    DWORD   NumberOfRvaAndSizes;
    IMAGE_DATA_DIRECTORY DataDirectory[IMAGE_NUMBEROF_DIRECTORY_ENTRIES];
} IMAGE_OPTIONAL_HEADER32, *PIMAGE_OPTIONAL_HEADER32;
```

Yapının Magic elemanında bazı özel değerler bulunmaktadır. Normal 32 bit çalıştırılabilen dosyalar için burada 0x010B, ROM imajları için 0x0107 ve 64 bit çalıştırılabilen dosyalar için ise (PE+) 0x020B değeri bulunmaktadır.

Yapının MajorLinkerVersion ve MinoırLinkerVersion elemanları ilgili dosyanın hangi linker versiyonu oluşturulduğunu belirtmektedir. (Örneğin bu makalenin yazıldığı sistemdeki link.exe isimli Microsoft'un linker programının versiyonu 14.00'dır.)

Yapının SizeofCode elemanı dosya içerisindeki makine kodlarının toplam miktarını belirtir. Kodlar genellikle yalnızca ".text" bulunmaktadır. Ancak birden fazla bölümde de bulunabilmektedir.

Yapının SizeofInitializedData elemanı ilkdeğer verilmiş static nesnelerin toplam uzunluğu SizeofUninitialized elemanı ise ilkdeğer verilmemiş nesnelerin toplam uzunluğunu belirtir.

Yapının AddressOfEntryPoint elemanı akışın başlatılacağı adresi RVA olarak vermektedir. Aslında bir C programında akış main fonksiyonundan başlamaz. Derleyicilerin yerleştirmiş olduğu bir başlangıç kodundan (startup code) başlar. Bu kod main fonksiyonunu çağırmaktadır.

Yapının BaseOfCode elemanı PE dosyasının çalıştırılabilen kısmının (.text) bölümünün başlangıcına ilişkin RVA'sını bize vermektedir. Yapının BaseOfData elemanı ise bize data (.data) bölümünün RVA'sını vermektedir.

Yapının ImageBase elemanı PE formatının default yüklenme adresini belrtmektedir. Yani PE dosyası belleğin neresinden itibaren yüklenecektir. Aslında bu adres linker ayarlarıyla değiştirilebilir. Linker da bu yükleme adresini buraya yazmaktadır. Microsoft'un linker'ları tipik olarak PE çalıştırılabilir dosyası 0x00400000 (4 MB) adrese yüklenecekmiş gibi bu ayarı yapmaktadır. DLL'ler için default yükleme adresi 1 GB olarak ayarlanmaktadır. Ancak tabii DLL'ler yeniden yüklenebilir (relocatable) bir kod içerirler. Dolayısıyla linker bunları maliyetli olarak başka bir yere yükleyebilmektedir.

Yapının SectionAlignment elemanı bölümlerin kaçın katlarına bellekte hizalanacağını belirtir. Default olarak linker'lar bunu sayfa genişliğine (4096 byte) hizalamaktadır.

Yapının FileAlignment elemanı bölümlerin dosyadaki hizalanma miktarını belirtir. Buradaki değer genellikle 2'nin kuvveti olur. Microsoft Linker'ları default olarak bunu 512 almaktadır.

Yapının MajorOperatingSystemVersion ve MinorOperatingSystemVersion elemanları hedef işletim sisteminin minimum versiyon numarasını belirtmektedir.

Yapının MajorImageVersion ve MinorImageVersion elemanları PE dosyasının version numarasını vermektedir. MajorSubsystemVersion ve MinorSubsystemVersion ise dosyanın yüklenebilmesi için işletim isteminin gereksinim duyduğu minimal alt sistem numarasını vermektedir. Yapının Win32VersionValue elemanı "ayrılmış (reserved)" durumdadır. Bu alanda 0 olmalıdır.

Yapının SizeofImage elemanı dosyanın sanal bellekte ne kadar yer kaplayacağını belirtir. Burada belirtilen değere dosya başlıkları, bölümler dahildir. Bölümler arasındaki hizalama bu değeri büyütebilir. Bu değer SectionAlignment değerinin bir katı (tipik olarak sayfa katları) olmak zorundadır. Buradaki değerin çalıştırılabilen dosyanın diskte kapladığı alanla bir ilgisi yoktur. Çünkü diskteki dosya içerisinde bazı bölümlerin yalnızca uzunlukları belirtilir. Halbuki burada dosyanın belleğe yüklendiğinde kaplayacağı alan

belirtilmektedir. İşletim sistemi doğrudan bu bilgiden hareketle dosyanın yüklenmesi için ne kadar sanal bellek gerektiğini anlamaktadır.

Yapının SizeofHeaders elemanı dosyanın başlıklarının toplamını bize verir. Bu değer FileAlignment değerinin katları olmak zorundadır.

Yapının Checksum elemanı DWORD negatif checksum değerini tutmaktadır. Ancak aygıt sürücüler dışında bu değer linker tarafından dosyaya sıfır olarak yazılır. Yükleyici de zaten aygıt sürücü dışındaki dosyaları yüklerken checksum değerini kontrol etmemektedir.

Yapının Subsystem elemanı dosyanın hangi alt sisteme yönelik olduğunu belirtir. (Örneğin dosya bir aygıt sürücü dosyası mıdır? Windows GUI uygulaması mıdır? Yoksa Console uygulaması mıdır? Windows CE uygulaması mıdır?)

Constant	Value	Description
IMAGE_SUBSYSTEM_UNKNOWN	0	An unknown subsystem
IMAGE_SUBSYSTEM_NATIVE	1	Device drivers and native Windows processes
IMAGE_SUBSYSTEM_WINDOWS_GUI	2	The Windows graphical user interface (GUI) subsystem
IMAGE_SUBSYSTEM_WINDOWS_CUI	3	The Windows character subsystem
IMAGE_SUBSYSTEM_OS2_CUI	5	The OS/2 character subsystem
IMAGE_SUBSYSTEM_POSIX_CUI	7	The Posix character subsystem
IMAGE_SUBSYSTEM_NATIVE_WINDOWS	8	Native Win9x driver
IMAGE_SUBSYSTEM_WINDOWS_CE_GUI	9	Windows CE
IMAGE_SUBSYSTEM_EFI_APPLICATION	10	An Extensible Firmware Interface (EFI) application
IMAGE_SUBSYSTEM_EFI_BOOT_ SERVICE_DRIVER	11	An EFI driver with boot services
IMAGE_SUBSYSTEM_EFI_RUNTIME_ DRIVER	12	An EFI driver with run-time services
IMAGE_SUBSYSTEM_EFI_ROM	13	An EFI ROM image
IMAGE_SUBSYSTEM_XBOX	14	XBOX
IMAGE_SUBSYSTEM_WINDOWS_BOOT_APPLICATION	16	Windows boot application.

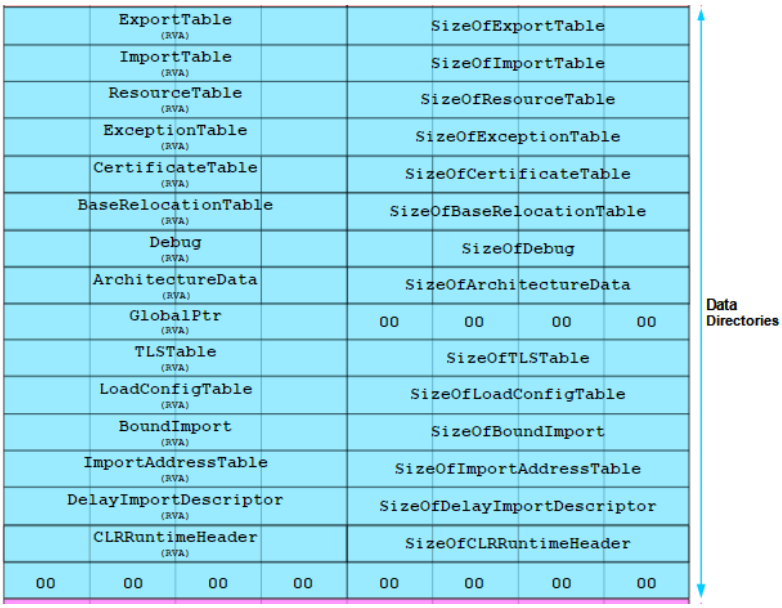
Yapının DllCharacteristics elemanı eğer dosya bir DLL dosyası ise DLL'e ilişkin bazı bilgileri verir. Bu aln bit bit kodlanmış çeşitli değerlerden oluşmaktadır. Dosyanın "relocatable" olduğu bilgisi de burada yer almaktadır.

Yapının SizeOfStackReserve SizeOfStackCommit elemanları thread'lerin (ana thread de dahil olmak üzere) default stack'lerinin hangi büyüklükte olacağını belirtir. Microsoft linker'ları tipik olarak thread'ler için 1MB alan (SizeOfStackReserve) ayırırlar. Commit miktarı başlangıçta ne kadar alanın sanal bellek olarak commit edileceğini belirtir. Stack için ayrılacak default alan linker ayarlarıyla değiştirilebilir.

Yapının SizeOfHeapReserve ve SizeOfHeapCommit elemanları prosesin default heap'i için ayrılacak alanı belirtmektedir. Process'in default heap'i büyüyebilen biçimdedir. Yani buradaki SizeOfHeapReserve değeri toplam heap alanını belirtmez. Başlangıçta sayfa tablolarında ayrılacak olan heap alanı belirtir. Bunun dışında bellek yettikçe heap alanı otomatik büyütülmektedir. Microsft Linker'ları SizeOfHeapReserve alanına 1 MB değerini, SizeOfHeapCommit alanına da 4096 değerini yazmaktadır. 32 bit Windows sistemlerinde heap alanı maksimum "User Alanı (User Space)" kadar olabileceğine göre teorik heap değeri 2 GB'dir)

Yapının LoaderFlags elemanı ayrılmıştır (reserved) ve burada 0 değeri bulunur.

Yapının NumberOfRvaAndSizes elemanı “PE Ek Başlığının (Image Optional Header)” sonunda bulunan “Veri Dizini (Data Directory)” denilen kısmın kaç elemandan oluştuğunu belirtmektedir. Normal olarak veri dizininde 16 eleman bulunur. Ancak duruma göre farklı olabilir. Veri dizini (data directory) PE dosyasının önemli alanlarının yerlerini ve uzunluklarını tutmaktadır. Bu alanlar çeşitli bölümlerin (sections) içerisinde bulunuyor olabilirler. Her alan için önce bir DWORD bir RVA ve sonra DWORD bir uzunluk bilgisi bulunur. “Veri Dizini (Daata directory)” içerisindeki alanlar şunlardır:



Veri Dizini IMAGE_OPTIONAL_HEADER32 yapısının sonındaki IMAGE_DATA_DIRECTORY yapısıyla temsil edilmiştir. Bu yapı da şöyle bildirilmiştir:

```
typedef struct _IMAGE_DATA_DIRECTORY {
    DWORD   VirtualAddress;
    DWORD   Size;
} IMAGE_DATA_DIRECTORY, *PIMAGE_DATA_DIRECTORY;
```

Yapının VirtualAddress elemanı ilgili alanın RVA’sını, Size elemanı ise Uzunluğunu belirtmektedir. PE dosyasının Veri dizininde belirtilen alanlar hakkında bazı bilgiler ileride ele alınacaktır.

Bölüm Tablosu (Section Table)

PE Ek başlığını (Image Optional Header) Bölüm tablosu denilen bir tablo izler. Bu tablo hemen PE ek başlığının bittiği yerden başlamaktadır. Ayrıca anımsanacağı gibi PE Ek Başlığının uzunluğu ve toplam bölüm sayısı PE Başlığında (Image File Header) belirtilmekteydi.

Bölüm Tablosu aslında Bölüm Başlıklarından (Section Header) oluşan bir dizi gibidir. Bir bölüm başlığı IMAGE_SECTION_HEADER yapısıyla temsil edilmektedir.

```
typedef struct _IMAGE_SECTION_HEADER {
    BYTE    Name[IMAGE_SIZEOF_SHORT_NAME];
    union {
            DWORD   PhysicalAddress;
            DWORD   VirtualSize;
    } Misc;
    DWORD   VirtualAddress;
    DWORD   SizeOfRawData;
    DWORD   PointerToRawData;
    DWORD   PointerToRelocations;
    DWORD   PointerToLinenumbers;
    WORD    NumberOfRelocations;
```

```
    WORD    NumberOfLinenumbers;
    DWORD   Characteristics;
} IMAGE_SECTION_HEADER, *PIMAGE_SECTION_HEADER;
```

PE formatı kabaca başlık kısımlarından ve bölümlerden oluşur. Her bölüm çalıştırılabilen dosyanın bazı kısımlarını tutmaktadır. Örneğin tipik olarak programın makine kodları (yani tüm fonksiyonların kodları) ".text" isimli bölümde, ilkdeğer verilmiş static nesneler ".data" isimli bölümde, ilkdeğer verilmemiş static nesnler ise ".bss" bölümünde bulunur. Aslında bazı bölümler Windows sistemlerinde zorunlu bazı isimlere sahiptir. Örneğin program kodlarının ".text" bölümünde olması zorunlu tutulmuştur. Ancak PE dosyasında kaç bölüm olacağı ve diğer bölümlerin isimleri dosyayı oluşturan derleyicilere bağlıdır (PE dosyasını bağlayıcı (linker) oluşturur ancak o da aslında amaç dosyalarını (object files) birleştirmektedir. Bölüm bilgileri aslında amaç dosyadan gelmektedir.)

Anahtar Notlar: Microsoft C Derleyicilerinde programcı isterse PE formatına #pragma section direktifiyle ek bölümler yerleştirebilir. #pragma section direktifinin genel biçimi şöyledir:

```
#pragma section("isim", [özellikler])
```

Özellikler ',' ile ayrılmış şu sözcüklerden oluşabilir: read, write, execute, shared, nopage, nopage, nocache, discard, remove.

Örneğin:

```
#pragma section("mysec", read, write)
```

Bölüm eklendikten sonra o bölümde yer alacak global ve static yerel nesnelerin bildirimlerinin başına __declspec(allocate("bölüm ismi")) bildirim belirleyicisinin getirilmesi gerekir. Örneğin:

```
#pragma section("mysec",read,write)

__declspec(allocate("mysec"))
int g_i = 0x12345678;

__declspec(allocate("mysec"))
char g_s[] = "Kaan Aslan";
```

Burada önce "mysec" isimli bir bölüm oluşturulmuştur. Sonra da onun içerisine g_i ve g_s isimli iki nesne yerleştirilmiştir. Microsft derleyicileri tipik olarak default biçimde şöyle bir yerleşim yapmaktadır:

- Programın içerisindeki tüm fonks,yonların makine kodlarını ".text" bölümüne
- İlkdeğer verilmiş global nesneleri ve static yerel nesneleri ".data" bölümüne
- İlkdeğer verilmemiş global nesneleri ve static yerel nesneleri ".bss" bölümüne
- String ifadeleri ".rdata" bölümüne

IMAGE_SECTION_HEADER yapısındaki Name elemanı bölümün ismi belirtmektedir. VirtualSize elemanı bölüm belleğe yüklendikten sonra onun bellekteki büyüklüğünü belirtir. Yapının VirtualAddress isimli elemanı bölümün başlangıç RVA'sını vermektedir.

Anahtar Notlar: Windows sistemlerinde ".exe" gibi ".dll" gibi yüklenebilen dosyalara "modül" denilmektedir.
Bir modülün yüklendikten sonraki adresi GetModuleHandle fonksiyonuyla elde edilebilir:

```
HMODULE WINAPI GetModuleHandle(
  __in LPCTSTR lpModuleName
);
```

Parametre olarak NULL değer girilirse "executable" dosyanın yükleme adresi elde edilir. Herhangi bir modül için ilgili "dll"in simi ve uzantısı parametre olarak verilmelidir. GetModuleHandle fonksiyonun geri dönüş değeri HANDLE türündendir (void *).

Anahrar Notlar: Anımsanacağı gibi PE Ek başlığında (IMAGE_OPTIONAL_HEADER) ilgili modülün önerilen yükleme adresi bulunmaktadır. Ancak bu önerlien yükleme adresi yükleyici tarafından kullanılmayabilir. Yani yükleyici eğer modülün içerisinde "relocation" bilgisi varsa modülü çeşitli gerekçelerle başka bir adrese yükleyebilir. "Exectuable" dosyalar için Microsoft'un bağlayızıları önerilen yükleme adresini 4MB (0x400000) olarak ayarladığını anımsayınız.

Yapının SizeOfRawData elemanı bölümün dosya içerisinde kaç byte yer kapladığı bilgisini verir. Bölümün dosyada kapladığı alan bölümgn bellekte kapladığı alandan (VirtualSize) daha fazla olabilir. Çünkü dosyada

bölümler belli katlara hizalanmaktadır. Dolayısıyla bir bölümün uzunluğu PE Ek başlığında belirtilen FileAlignment değerinin katları olmalıdır. Yapının PointerToRawData elemanı ilgili bölümün dosyada hangi offsetten başladığını belirtmektedir. RVA değerini dosya offsetine dönüştürebilmek için bu değerden faydalanılmaktadır. Yapının PointerToRelocations elemanı amaç dosyalar için anlamlıdır. Amaç dosyalarda (COFF dosyalarında) burada ilgili bölümün "relocation" bilgisinin bulunduğu dosya offset'i bulunur. Çalıştırılabilen (executable) dosyalarda bu alanda 0 değeri bulunmalıdır. Yapının PointerToLineNumber elemanında amaç dosyalar için debug amaçlı satır numaraları bilgilerinin bulunduğu dosya offseti yer alır. Çalıştırılabilen (executable) dosyalarda bu alanda 0 bulunmalıdır. Yapının NumberOfRelocations elemanında "relocation bilgilerinin sayısı" bulunur. Çalıştırılabilen (executable) dosyalarda bu alanda 0 değeri bulunmaktadır. Yne yapının NUmberOfFileNumbers elemanında amaç dosyalardaki (COFF dosyaalarındaki) debug amaçlı satır numaralarının sayısı bulunur. Çalıştırılabilen dosyalarda bu alanda 0 bulunmalıdır. Yapının Caharcteristics elemanında ilgili bölümün özellikleri bit bit kodlanmıştır. Özellikler şunlardan oluşmaktadır:

Flag	Value	Description
	0x00000000	Reserved for future use.
	0x00000001	Reserved for future use.
	0x00000002	Reserved for future use.
	0x00000004	Reserved for future use.
IMAGE_SCN_TYPE_NO_PAD	0x00000008	The section should not be padded to the next boundary. This flag is obsolete and is replaced by IMAGE_SCN_ALIGN_1BYTES. This is valid only for object files.
	0x00000010	Reserved for future use.
IMAGE_SCN_CNT_CODE	0x00000020	The section contains executable code.
IMAGE_SCN_CNT_INITIALIZED_DATA	0x00000040	The section contains initialized data.
IMAGE_SCN_CNT_UNINITIALIZED_ DATA	0x00000080	The section contains uninitialized data.
IMAGE_SCN_LNK_OTHER	0x00000100	Reserved for future use.
IMAGE_SCN_LNK_INFO	0x00000200	The section contains comments or other information. The **.drectve** section has this type. This is valid for object files only.
	0x00000400	Reserved for future use.
IMAGE_SCN_LNK_REMOVE	0x00000800	The section will not become part of the image. This is valid only for object files.
IMAGE_SCN_LNK_COMDAT	0x00001000	The section contains COMDAT data. For more information, see section 5.5.6, "COMDAT Sections (Object Only)." This is valid only for object files.
IMAGE_SCN_GPREL	0x00008000	The section contains data referenced through the global pointer (GP).
IMAGE_SCN_MEM_PURGEABLE	0x00020000	Reserved for future use.
IMAGE_SCN_MEM_16BIT	0x00020000	Reserved for future use.
IMAGE_SCN_MEM_LOCKED	0x00040000	Reserved for future use.
IMAGE_SCN_MEM_PRELOAD	0x00080000	Reserved for future use.
IMAGE_SCN_ALIGN_1BYTES	0x00100000	Align data on a 1-byte boundary. Valid only for object files.
IMAGE_SCN_ALIGN_2BYTES	0x00200000	Align data on a 2-byte boundary. Valid only for object files.
IMAGE_SCN_ALIGN_4BYTES	0x00300000	Align data on a 4-byte boundary. Valid only for object files.
IMAGE_SCN_ALIGN_8BYTES	0x00400000	Align data on an 8-byte boundary. Valid only for object files.
IMAGE_SCN_ALIGN_16BYTES	0x00500000	Align data on a 16-byte boundary. Valid only for object files.
IMAGE_SCN_ALIGN_32BYTES	0x00600000	Align data on a 32-byte boundary. Valid only for object files.
IMAGE_SCN_ALIGN_64BYTES	0x00700000	Align data on a 64-byte boundary. Valid only for object files.
IMAGE_SCN_ALIGN_128BYTES	0x00800000	Align data on a 128-byte boundary. Valid only for object files.
IMAGE_SCN_ALIGN_256BYTES	0x00900000	Align data on a 256-byte boundary. Valid only for object files.
IMAGE_SCN_ALIGN_512BYTES	0x00A00000	Align data on a 512-byte boundary. Valid only for object files.
IMAGE_SCN_ALIGN_1024BYTES	0x00B00000	Align data on a 1024-byte boundary. Valid only for object files.
IMAGE_SCN_ALIGN_2048BYTES	0x00C00000	Align data on a 2048-byte boundary. Valid only for object files.
IMAGE_SCN_ALIGN_4096BYTES	0x00D00000	Align data on a 4096-byte boundary. Valid only for object files.
IMAGE_SCN_ALIGN_8192BYTES	0x00E00000	Align data on an 8192-byte boundary. Valid only for object files.
IMAGE_SCN_LNK_NRELOC_OVFL	0x01000000	The section contains extended relocations.
IMAGE_SCN_MEM_DISCARDABLE	0x02000000	The section can be discarded as needed.
IMAGE_SCN_MEM_NOT_CACHED	0x04000000	The section cannot be cached.
IMAGE_SCN_MEM_NOT_PAGED	0x08000000	The section is not pageable.
IMAGE_SCN_MEM_SHARED	0x10000000	The section can be shared in memory.

Flag	Value	Description
IMAGE_SCN_MEM_EXECUTE	0x20000000	The section can be executed as code.
IMAGE_SCN_MEM_READ	0x40000000	The section can be read.
IMAGE_SCN_MEM_WRITE	0x80000000	The section can be written to.

RVA'nın Dosya Offset'ine Dönüştürülmesi

Yukarıdan da görüldüğü gibi PE formatının pek çok yerinde ilgili elemanın yeri RVA (Relative Virtual Address) olarak verilmiştir. RVA'nın dosya belleğe yüklendikten sonra dosyanın yüklenme yerinden itibaren bellekte göreli uzaklık belirttiğini anımsayınız. PE dosya formatı yükleyici için oluşturulduğundan dosya offset'i yerine RVA'ların kullanılması çok uygundur. Başlıklar içerisinde dosya offset'i belirten tek yer aslında bölümlerin yerleridir. Bölümlerin yerleri bölüm başlıklarında hem RVA olarak hem de dosya offset'i olarak verilmiştir. O halde belli bir RVA değerini dosya offset'ine dönüştürebilmek için önce o RVA'nın hangi bölüm içerisinde ve o bölümün hangi offset'i içerisinde olduğunu bulmamız gerekir. Bundan sonra o bölümün dosya offset'i belli olduğuna göre o offset değerini dosya offset'ine ekleyerek ilgili RVA'nın dosya içerisindeki yerini bulabiliriz. Örneğin:

```
DWORD RVAToFileOffset(DWORD rva)
{
    for (int i = 0; i < g_imageFileHeader->NumberOfSections; ++i)
        if (rva >= g_sectionHeaders[i].VirtualAddress &&
                        rva < g_sectionHeaders[i].VirtualAddress + g_sectionHeaders[i].Misc.VirtualSize)
            return rva - g_sectionHeaders[i].VirtualAddress + g_sectionHeaders[i].PointerToRawData;

    return 0;
}
```

PE Dosyasını Analiz Etmek İçin Kullanılan Çeşitli Araçlar

PE dosyasını analiz eden programı kendimiz sınıf içerisinde yazdığımız kodlarda olduğu gibi yazabiliriz. Ancak bunun için gerek paralı gerekse açık kaynak kodlu ve/veya bedava mini programlar bulunabilmektedir.

DUMPBIN

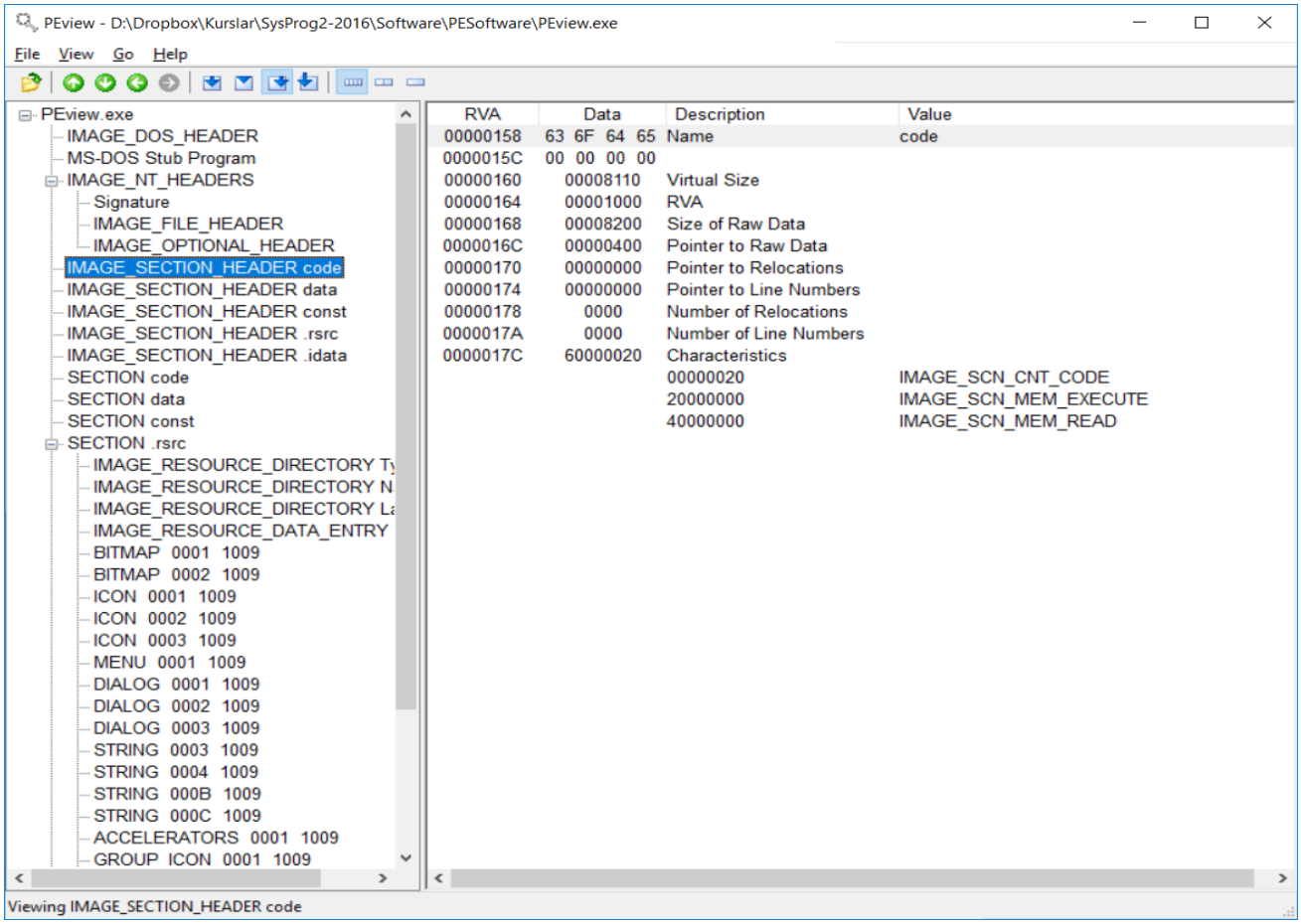
Anahtar Notlar: "Dumpbin" Microsoft'un C/C++ derleyici paketinde bulunan bir utility programdır. Bu program "object module (.obj)" dosyalarını, dinamik kütüphane dosyalarını (dll), statik kütüphane dosyalarını (lib) ve çalıştırılabilen (executable) dosyaları incelemek için kullanılmaktadır. Visual Studio IDE'si kurulduğunda "dumpbin" de kurulmuş olmaktadır. Dumpbin bir console uygulamasıdır. "dumpbin" hiç seçeneksiz çalıştırıldığında yalnızca dosyadaki bölümlerin isimlerini ve uzunluklarını gösterir. Örneğin:

```
D:\Dropbox\Kurslar\80X86-ARM-Assembly\Src\C\Sample\Debug>
D:\Dropbox\Kurslar\80X86-ARM-Assembly\Src\C\Sample\Debug>dumpbin sample.exe
Microsoft (R) COFF/PE Dumper Version 14.00.23506.0
Copyright (C) Microsoft Corporation.  All rights reserved.


Dump of file sample.exe

File Type: EXECUTABLE IMAGE

  Summary

        1000 .00cfg
        1000 .data
        1000 .gfids
        1000 .idata
        2000 .rdata
        1000 .reloc
        1000 .rsrc
        5000 .text
       10000 .textbss
```

“Dumpbin” /HEADERS seçenceği ile çalıştırılırsa PE dosyasının başlık kısımlarını görüntüler. “/SECTION: <isim>” seçeneği ile dumpbin istediğimiz bir bölümü de bize gösterebilmektedir. “/DISASM” seçeneği “.text” bölümünü “assembly” sentaksıyla bize gösterir. /ALL seçeneği PE formatı içerisindeki tüm bilgileri bize vermektedir. Bu seçenekte bölümlerin yalnızca başlık kısımları değil, içerisindeki veriler de görüntülenmektedir. Diğer seçenekler için MSDN yardım sistemine başvurabilirsiniz.

PEView

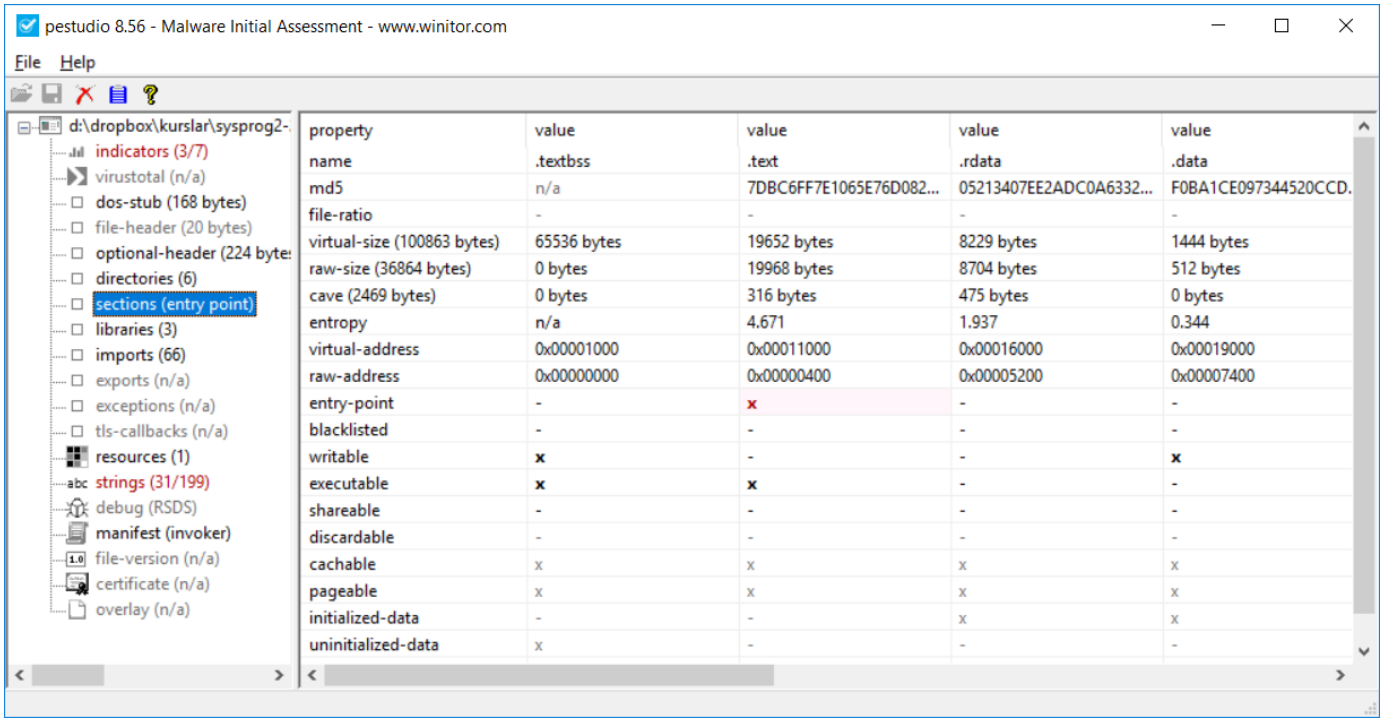
PE dosyasını analiz etmek için kullanılan mini bir programdır. Bu program dosya başlıklarını, bölüm başlıklarını ve bölümlerin içeriklerini GUI uygulaması olarak gösterir.



PEView programı çeşitli bilgileri bize hem dosya offet’i hem de RVA olarak da vermektedir. Ancak RVA’dan dosya offetin’e dönüşüm yapma işlevi yoktur. Ayrıca PEView dosyanın değiştirilmesine izin vermemektedir. Yalnızca görüntülenmesini sağlamaktadır.

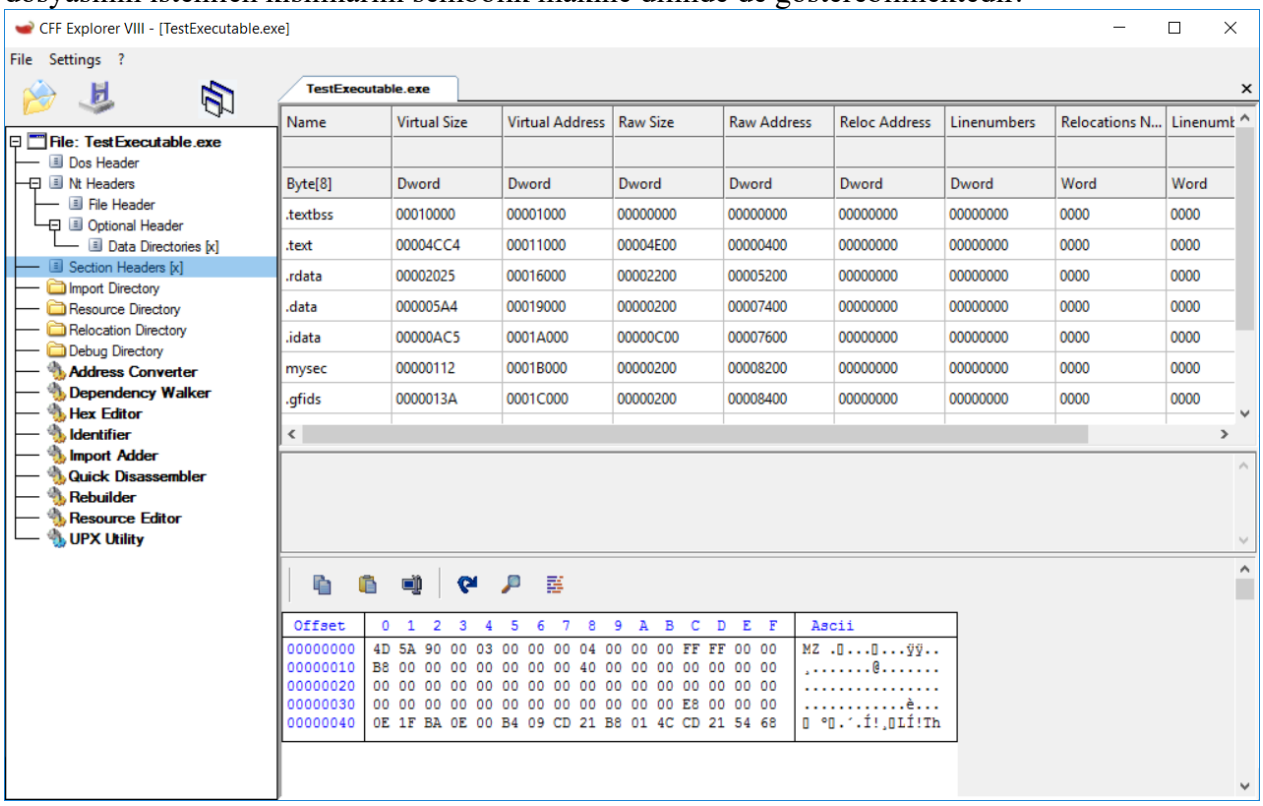
PEStudio

Bu program PE dosyasının pek çok bölümünü ayrıntılı olarak görüntülemektedir. Ancak bir güncelleme işlevi yoktur. Adresleme hep RVA olarak yapılmıştır.



Explorer Suite (CFF)

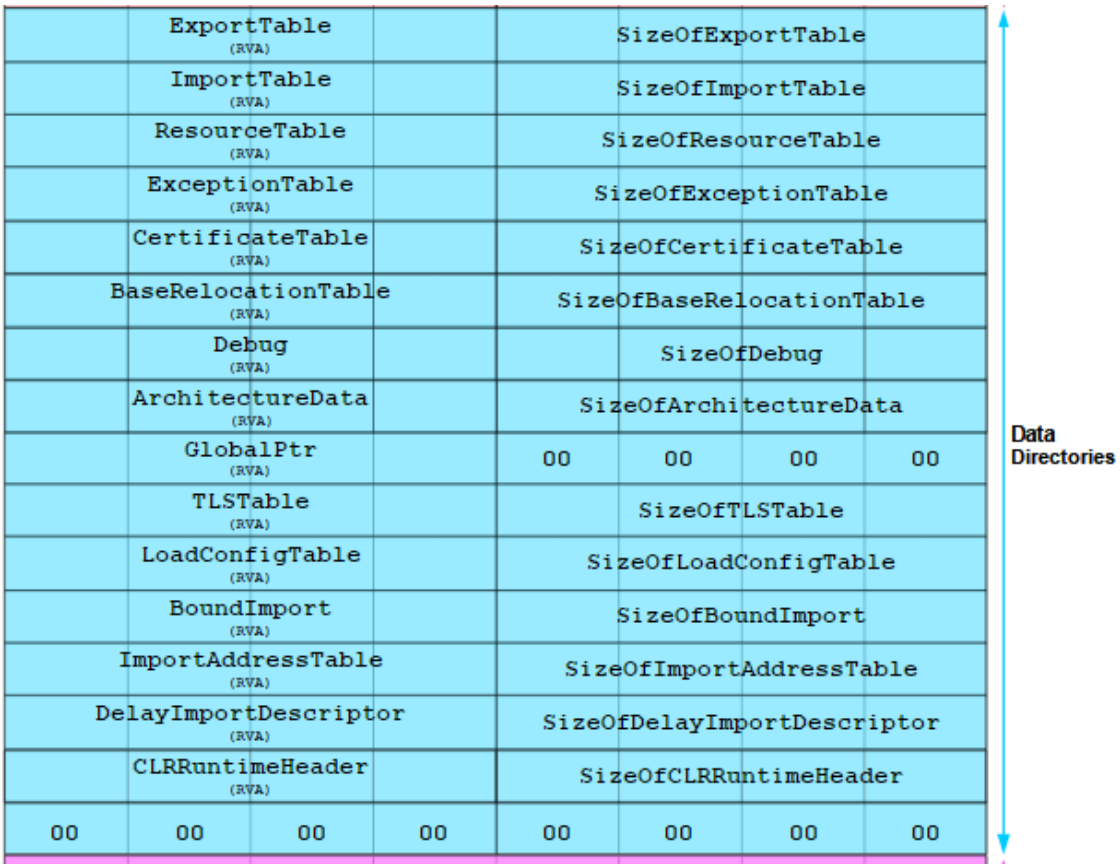
PE dosyasını analiz etmek için ve diğer bazı işlemler için kullanılan küçük programlardan oluşan bir pakettir. Paketin en önemli programı CFF'dir. CFF yukarıdaki programlardan daha yeteneklidir. Bu nedenle pek çok kaynak bu programa referans etmektedir. CFF'de PE başlıkları ve bölümleri görüntülenip üzerinde değişiklikler yapılabilmektedir. AYnı zamanda RVA'dan dosya pffset'ine dönüştürme gibi bir işleve de sahiptir. CFF PE dosyasının istenilen kısımlarını sembolik makine dilinde de gösterebilmektedir.



PE Dosyasının Veri Dizini (Data Directory)

Anımsanacağı gibi PE dosyasının IMAGE_OPTIONAL_HEADER isimli ek başlığının sonunda 16 elemanlık bir "veri dizini (data directory)" bulunmaktaydı. Bu veri dizininde PE dosyası için önemli bazı alanların

(tabloların) adresleri ve uzunlukları tutulmaktadır. Bu alanlar aslında çeşitli bölümlerin (sections) içerisindedir. Veri dizinindeki bu alanlar (ya da tablolar) dosya hakkında önemli metada bilgilerini tutar. Veri dizininin genel formatını yeniden aşağıda veriyoruz:

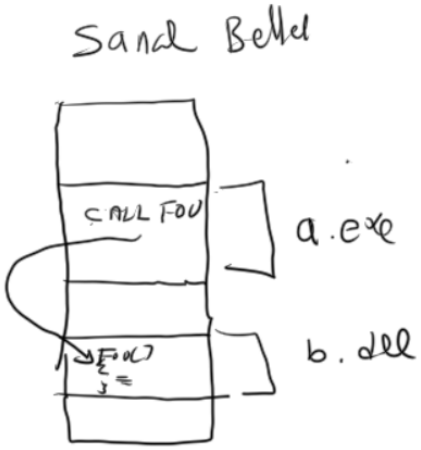


Veri dizinindeki alanların (tabloların) yerleri RVA olarak verilmiştir. Şimdi PE dosyasnın veri dizinindeki bazı tabloları ve bunların işlevlerini inceleyelim. Çünkü bazı işlemler için en azından bu tabloların ne amaçla oluşturulduğunun bilinmesi gerekmektedir.

Export Tablosu (Export Table)

Veri dizinin ilk elemanında Export tablosunun RVA'sı ve uzunluğu bulunmaktadır. Export tablosu genellikle DLL'lerde dolu biçimde çalıştırılabilen (executable) dosyalarda boş biçimdedir. Ancak çalıştırılabilen dosyalarda da export tablosu dolu olabilir. Export tablosu PE dosyasının genellikle ".edata" ya da ".rdata" bölümlerinde bulundurulmaktadır. Export tablosunu iyi anlayabilmek için DLL'lerin nasıl yüklendiğini ve DLL'deki fonksiyonların nasıl çağrıldığını bilmek gerekir.

Bir DLL dosyası işletim sistemi tarafından bellekte belli bir adrese bütünsel olarak (yani bir kısmı değil hepsi) yüklenmektedir. Bir DLL'den çağırma yapan program gerçekten de sanki kendi dosyasındaki bir fonksiyonu çağırıyormş gibi bunu CALL makine komutu ile yapar. Ancak derlenip bağlanmış bir programda DLL'in nereye yükleneceği henüz belli olmadığı için bu CALL makine komutunun operandı (yani hangi adrese CALL yapılacağı) henüz bilinmemektedir. İşte işletim sistemi DLL'li yükledeikten sonra daha ileride ele alınacağı üzere yüklenen çalıştırılabilen dosyada CALL makine makine komutunun operandını (dallanılacak adresi) DLL'in yüklenme adresine ve çağrılan DLL fonksiyonunun DLL içerisindeki yerine göre ayarlar. Örneğin a.exe programı b.dll içerisindeki Foo fonksiyonunu çağırmiş olsun:



İşletim sisteminin yükleyicisi b.dll dosyasını yükleyip eğer b.dll içerisinde Foo fonksiyonun yerini (RVA'sını) biliyorsa CALL makine komutunu düzeltip çağrının doğru çalışmasını sağlayabilir. İşte export tablosu export edilmiş (__declspec(dllexport) ile bildirilmiş) sembollerin isimlerinin ve RVA'larının tutulduğu bir tablodur. Yani bizim bir DLL'deki fonksiyonu çağırabilmemiz için o fonksiyonun adresinin RVA olarak DLL'in export tablosunda kayıtlı olması gerekir. Export tablosu <winnt.h> dosyasında (<windows.h> dosyasında) IMAGE_EXPORT_DIRECTORY yapısıyla temsil edilmiştir:

```
typedef struct _IMAGE_EXPORT_DIRECTORY {
    DWORD   Characteristics;
    DWORD   TimeDateStamp;
    WORD    MajorVersion;
    WORD    MinorVersion;
    DWORD   Name;
    DWORD   Base;
    DWORD   NumberOfFunctions;
    DWORD   NumberOfNames;
    DWORD   AddressOfFunctions;     // RVA from base of image
    DWORD   AddressOfNames;         // RVA from base of image
    DWORD   AddressOfNameOrdinals;  // RVA from base of image
} IMAGE_EXPORT_DIRECTORY, *PIMAGE_EXPORT_DIRECTORY;
```

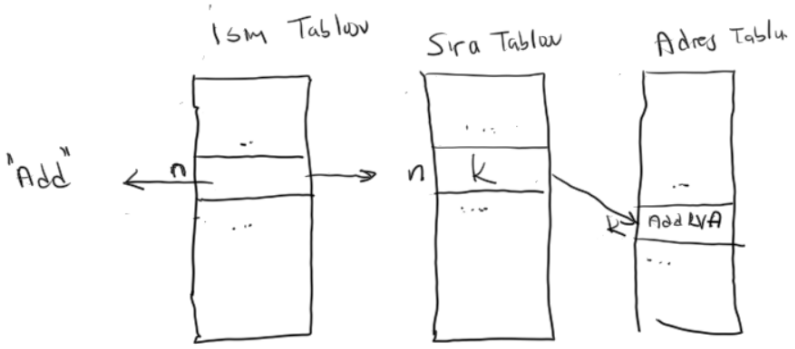
Yapının Characteristics elemanı reserve edilmiştir ve burada 0 bulunmalıdır. Yapının TimeDateStamp elemanı bu export tablosunun oluşturulduğu tarih ve zamanı belirtir. Yapının MajorVersionve MinorVersion elemanları adeta bir yorum alanı gibidir. Linker ya da başka bir kaynak buraya bir versiyon numarası yazabilir. Daha sonra onu okuyarak kendi işlemleri için kullanabilir. Genellikle bu alanda 0 bulunmaktadır. Yapının Name elemanı DLL isminin bulunduğu yerin RVA'sını belirtmektedir. Bu RVA'da DLL ismi sonu '\0' olan ASCII karakterleri biçiminde bulunmaktadır.

Yapının Base elemanı "Sıra Tablosubunun (Ordinal Table)" ilk elemanının başlangıç numarasını vermektedir. Bu numara default olarak 1 durumdadır. Yapının NumberOfFunctions elemanı expoert edilen elemenalrın toplam sayısını vermektedir. Yapının NumberOfNames elemanı İsim Tablosundaki (Name Table) isimlerin sayısını belirtir. Genellikle bu isimlerin sayısı NumberOfFunctions elemanındaki ezport edilen sembollerin sayısı ile aynıdır.

Yapının AddressOfFunctions, AddressOfNames ve AddressOfNameOrdinals elemanları sırasıyla export tablosuna ilişkin "Adres tablosunun (Address Table)", "İsim tablosunun (Name Tables)" ve "Sıra tablosunun (Ordinal Table)" adreslerini RVA olarak tutar.

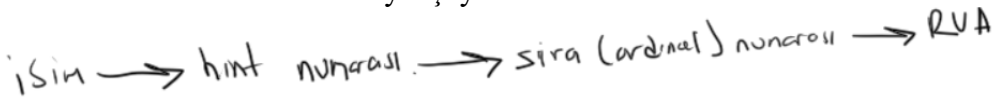
Yukarıdaki IMAGE_EXPORT_DIRECTORY yapısı aslında export tablosu için bir başlık kısmı gibidir. Export tablosunu üç tablo oluşturmaktadır: İsim Tablosu (yeri AddressOfNames elemanında), Sıra Tablosu (yeri AddressOfNameOrdinals elemanında) ve Adres Tablosu (yeri AddressOfFunctions elemanında).

Yükleyicinin export tablosunu incelemekten amacı export edilmiş beli bir sembolün (fonksiyon ya da verinin) RVA olarak adresini bulmaktır. Bu süreç üç tablo yardımıyla aşağıdaki gibi yürütülür:



İsim tablosu ile eleman sıra tablosu paralel iki tablodur. Yani bunların elemanları karşılıklıdır. Adresi bulunacak sembol ismi İsim tablosunda aranır. (İsim atblosunda aslında isimler değil isimlerin başlangıç adreslerinin RVA'ları bulunmaktadır). İsmin tablonun n'inci elemanında bulunduğunu varsayalım. Bundan sonra yükleyici sıra tablosunun n'inci elemanına başvurur. Sıra tablosunun n'inci elemanında sembolün adres tablosundaki indeksi bulunmaktadır. Bu indekse k diyelim. Snra yükleyici nihayet Adres tablosunun k'ıncı elemanından ilgili sembolün RVA'sını elde eder. Arama işlemi isimle başlatıldığında süreç böyle ilerlemektedir. Ancak bazen arama işlemi için isim değil doğurdan sıra tablosundaki indeks (isim tablosundaki indeks ile aynı) de kullanılabilmektedir. Sembolün sıra tablosundaki indeksine "hint" numarası denilmektedir. Örneğimizde Add fonksiyonunun hint numarası n'dir. Eğer yükleyici ilgili sembolün hint numarasını biliyorsa isim araması yapmaz. Doğrudan sıra tablosunun hint ile belirtilen indeksine başvurur. Buradan sembolün adres tablosundaki indeksini elde eder. Arama doğurdan adres tablosundaki indeks verilerek de en hızlı biçimde yapılabilmektedir. Örneğimizde bu k değerine sıra numarası (ordinal number) denilmektedir.

O halde sembolün adresi sırasıyla şöyle elde edilmektedir:



PE32 formatında İsim Tablosunun, Sıra Tablosunun ve Adres Tablosunun girişleri 4 byte uzunluktadır. Yukarıdaki organizasyonda Adres tablosunun neden parelel olmadığı merak edilebilir. Microsoft bazı fonksiyonların hiç belirtilmeden ve İsim tablosunda girişi olmadan sıra numarasıyla (ordinal number) çağrılmasına izin vermiştir. Bu nedenle bir adres tablosunda bulunan bir fonksiyonun isminin bulunması zorunlu değildir.

Biz LoadLibrary ile bir DLL', dinamik olarak yükleyerek GetProcAddress fonksiyonuyla ilgili sembolün adresini alıp onu kullanabiliriz. GetProcAddress fonksiyonunda fonksiyonun ismi yerine onun sıra numarasını (ordinal number) da verebiliriz. Ancak sıra numarasının yüksek anlamlı WORD değeri 0 olan bir adres gibi verilmesi gerekir. Zaten Windows sistemlerinde bir modül 64K'dan daha düşük adrese yüklenememektedir. Örneğin DLL'de sıra numarası 1 olan fonksiyonu şöyle çağırabiliriz:

```
#include <stdio.h>
#include <stdlib.h>
#include <Windows.h>

typedef int(*PF)(int, int);

int main(void)
{
    HMODULE hModule;
    PF pf;

    if ((hModule = LoadLibrary("..\\TestDll\\Debug\\TestDll.dll")) == NULL) {
```

```
        fprintf(stderr, "cannot load library: %lu\n", GetLastError());
        exit(EXIT_FAILURE);
    }

    if ((pf = (PF)GetProcAddress(hModule, MAKEINTRESOURCE(1))) == NULL) {
        fprintf(stderr, "cannot get address: %lu\n", GetLastError());
        exit(EXIT_FAILURE);
    }
    printf("%d\n", pf(10, 20));


    return 0;
}
```

Burada MAKEINTRESOURCE makrosu bir tamsayı değeri yüksek anlamlı WORD'ü 0 olan char * türünden bir adrese dönüştürmektedir.

LoadLibrary ve GetProcAddress fonksiyonlarını biz nasıl yazabiliriz? Aslında LoadLibrary eksik fakat basit bir biçimde bellek tabanlı bir dosya olarak prosesin adres alanına yüklenebilir. Tabii burada dosyanın sayfa özelliklerini uygun biçimde ayarlaamak gerekebilir. GetProcAddres ise dosyanın export yazılabilir.

Import Tablosu (Import Table)

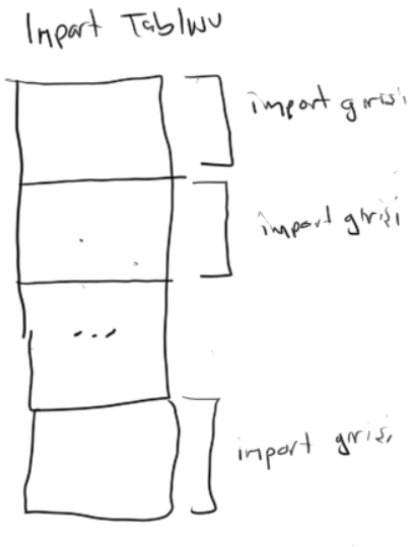
Import tablosu PE formatınının en önemli kısımlarından biridir. Bu tablonunda yeri ve uzunluğu PE Ek Başlığındaki veri dizininde bulundurulmaktadır. Import tablosuna statik yüklenen DLL'lerden çağırma yapılırken erişilir. Anımsanacağı gibi bir DLL iki biçimde yüklenebilmektedir:

1) Statik Olarak: Biz programımızda DLL fonksiyonlarını normal fonksiyon gibi çağırırız. Bağlayıcı PE formatına hangi DLL'lerden hangi fonksiyonların çağrıldığı bilgisini yazar. İşletim sisteminin yükleyicisi de çalıştırılabilen dosyayla birlikte DLL'leri de yükler ve ileride ayrıntıları anlatılaca olan bağlantıları yapar.

2) Dinamik Olarak: Programın çalışma zamanı sırasında LoadLibrary API fonksiyonula DLL yüklenip GetProcAddress fonksiyonuyla ilgili sembolün adresi elde edilebilir. Dinamik yükleme nispeten işletim sistemi için daha kolaydır. Bu süreçte yükleyici DLL'i belleğe yükler Sonra istenne fonksiyonun (sembolün) adresini export tablosundan bulur ve bize verir. Biz de o fonksiyonu çağırırız.

PE formatının import tablosu başka DLL'lerden çağırma yapan programlarda bulunur. Bir DLL başka bir DLL'den çağırma yapabileceğine göre import tablosu hem ".exe" dosyalarda hem de ".dll" dosyalarında bulunabilmektedir. Import tablosu genellikle ".idata" isimli bölümde bulundurulmaktadır. Import tablosunun adresi PE ek başlığındaki dizin tablosunda tutulur.

Import tablosu aslında "import girişlerinden (import directory entry)" oluşan bir dizi biçimindedir. Her giriş bir DLL'den kullanılan sembollere ilişkindir.



Import girişleri <winnt.h> (<windows.h>) dosyasında IMAGE_IMPORT_DESCRIPTOR yapısıyla temsil edilmektedir:

```
typedef struct _IMAGE_IMPORT_DESCRIPTOR {
    union {
        DWORD   Characteristics;            // 0 for terminating null import descriptor
        DWORD   OriginalFirstThunk;         // RVA to original unbound IAT (PIMAGE_THUNK_DATA)
    } DUMMYUNIONNAME;
    DWORD   TimeDateStamp;                  // 0 if not bound,
                                            // -1 if bound, and real date\time stamp
                                            //     in IMAGE_DIRECTORY_ENTRY_BOUND_IMPORT (new BIND)
                                            // O.W. date/time stamp of DLL bound to (Old BIND)

    DWORD   ForwarderChain;                 // -1 if no forwarders
    DWORD   Name;
    DWORD   FirstThunk;                     // RVA to IAT (if bound this IAT has actual addresses)
} IMAGE_IMPORT_DESCRIPTOR;
typedef IMAGE_IMPORT_DESCRIPTOR UNALIGNED *PIMAGE_IMPORT_DESCRIPTOR;
```

Import girişleri kabaca hangi DLL'den hangi fonksiyonların çağrıldığını tutan bir tablodur.

Yapının OriginalFirstThunk elemanı "Import Lookup Tablosu"nun RVA olarak adresini tutmaktadır. TimeDateStamp elemanı modülün oluşturulduğu tarih ve zamanı verir. ForwardChain "ileriye bakmaya" ilişkindir. Burada bu konu ele alınmayacaktır. Yapının Name elemanı modülün ismini vermektedir. Yapının FirstThunk elemanı ise "Import Adres Tablosunun" RVA cinsinden adresin i verir.

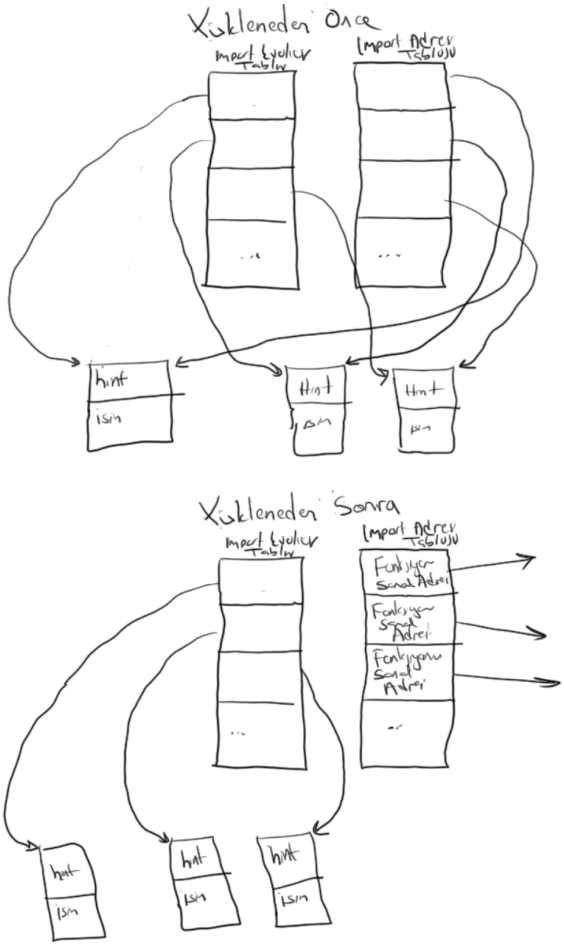
Yukarıdan da görüldüğü gibi Import girişlerinde iki paralel tablo vardır: "Import Lookup Tablosu" ve "Import Adres Tablosu". "Import Lookup Tablosu" DLL içerisinde referans edilmiş fonksiyonların isimlerinin bulunduğu tablodur. "Import Address Tablosu" ise referans edilen DLL fonksiyonlarının RVA'larını (yani on DLL'lerin yüklenme adresinden itibaren uzaklığını) tutmaktadır. "Import Lookup Table" PE32 formatında 4 byte'lık girişlere sahiptir. Her giriş ilgili DLL fonksiyonun isminin bulunduğu RVA'yı belirtir (ayrıntıya bakınız). Ancak burada belirtilen ismin RVA'sının yüksek anlamlı biti 1 ise bu DLL fonksiyonu isimle değil sıra numarasıyla (ordinal number) bulunur. Sıra numarası da buradaki 4 byte'ın düşük anlamlı WORD'ündedir.

Aslında Import Lookup Tablosunun girişleri doğrudan isimlerin RVA'larını tutmaz. Bu girişler aslında "Hint/Name Tablosu" denilen bir yapıyı gösterir. İsim bu yapının içerisindedir. Hint/Name tablosu değişken uzunlukta girişe sahip olduğu <winnt.h> (<windows.h>) içerisinde bir yapıyla temsil edilmemiştir. Genel yapısı şöyledir:

Offset	Size	Field	Description
0	2	Hint	An index into the export name pointer table. A match is attempted first with this value. If it fails, a binary search is performed on the DLL's export name pointer table.
2	variable	Name	An ASCII string that contains the name to import. This is the string that must be matched to the public name in the DLL. This string is case sensitive and terminated by a null byte.
*	0 or 1	Pad	A trailing zero-pad byte that appears after the trailing null byte, if necessary, to align the next entry on an even boundary.

Yapının ilk 2 byte'ı ilgili fonksiyonunun "hint numarası"nı belirtir. İsim ise ASCII olarak kodlanmış değişken uzunlukta bir alandır. Import Lookup tablosunun ve Import Address Tablosununun uzunluğu herhangi bir yerde yazmamaktadır. Bunların son elemanı sıfırdır.

Import Adress Tablosu yükleyici tarafından doldurulmaktadır. İşin başında yani program yüklenmeden önce Import Lookup Tablosu ile Import Address Tablosu tamamen aynı içeriğe sahiptir. Ancak yükleme sırasında yükleyici Import Address Tablosunu uygun biçimde doldurur ancak Import Lookup Tablosuna dokunmaz.



Her ne kadar Import Lookup tablosu ve Import Addrfess Tablosu başlangıçta aynı içeriğe sahipse de yükleyici yükleme işleminden sonra ileride ele alındığı üzere DLL'lerin export tablolarından hareketle Import Address Tablosunu söz konusu fonksiyonların sanal bellekteki başlangıç adresini gösterecek biçimde ayarlamaktadır. Windows'un bazı DLL'lerinden çağırma yapıldığında işin başında Import Lookup Tablosu boş olabilir. Bu durumda PE dosyasını analiz ederken Import Address Tablosunu kullanmak daha uygun olabilir.

Statik Yükleme Sırasında DLL'lerde Fonksiyonların Çağrılması

Bir DLL'den export edilmiş bir fonksiyon çağrıldığında derleyici tipik olarak henüz çağrılan fonksiyonun adresini bilemediği için dolaylı CALL (indirect CALL) komutu uygular. Intel işlemcilerinde dolaylı CALL komutunun sembolik makine dilindeki gösterimi şöyledir:

```
CALL [adres]
```

Halbuki doğurdan CALL komutunun gösterimi şöyledir:

```
CALL    adres
```

Doğrudan CALL komutunda dallanılacak adres (yer değiştime miktarı) doğurdan makine komutunun operandı olmaktadır. Halbuki dolaylı CALL komutunda gerçek dallanılacak adres komutun operandında verilen adresteki yerden elde edilmektedir. Başka bir deyişle:

```
CALL   [adres]
```

komutu "adres ile belirtilen yerden çekilen 4 byte'ta (32 bit sistemde) belirtilen yere dallan" anlamına gelir. Pekiyi derleyici henüz derleme aşamasında fonksiyonun DLL'de olduğunu nereden anlayıp dolaylı CALL komutu uygulamaktadır? Bilindiği gibi normal fonksiyon çağrılarına derleyici aslında doğrudan CALL komutu uygulamaktadır. İşte __declspec(dllimport) bildirimi bu işe yaramaktadır. Örneğin:

```
__declspec(dllimport) void Foo(void);
```

gibi bir prototipi gören derleyici fonksiyonun bir DLL fonksiyonu olduğunu anlar ve dolaylı CALL komutu uygular. Aslında bir DLL fonksiyonu __declspec(dllimport) ile bildirilmeden de çağrılabilmektedir. Örneğin:

```
void Foo(void);
```

Yani aslında DLL'den çağırma yapılırken kullanılan __declspec(dllimport) bildirimi zorunlu değildir. Bu durumda derliyici doğrudan CALL komutu uygular. Ancak bağlayıcı çağrılan fonksiyonun DLL fonksiyonu olduğunu görünce tüm doğrudan CALL komutlarını dolaylı CALL komutlarına dönüştürür. Burada doğurdan CALL komutu ile dolaylı CALL komutunun aynı uzunlukta komutlar olması gerekmektedir. Intel'de böyle olduğu için bu düzeltmeyi bağlayıcı yapabilmektedir. Ancak ne olursa olsun bu iyi bir teknik değildir. Boşuna bağlama aşamasını uzatır. En normal durum DLL'den çağrılan fonksiyonların prototiplerinde __declepc(dllimport) bildiriminin yapılmasıdır.

DLL'deki fonksiyonlar çağrıldığında dolaylı CALL komutu Import Adress Tablosundaki girişe referans etmektedir. Import Address Tablosu yükleyici tarafından fonksiyonun gerçek sanal adreslerini içerecek biçimde doldurulmaktadır. Şöyle ki: Yükleyici DLL'i yükler. Export edilmiş fonksiyonların RVA'larını DLL'in export tablosundan alır. Bunu yükleme adresiyle toplar ve gerçek sanal adresleri elde eder. Bu adresleri de Import Tablosunun Import Address Tablosu dizisine yerleştirir. Zaten derleyicinin ürettiği kod buraya referans etmektedir. Fonksiyon normal olarak çağrılır.

Statik yüklemedeki süreç şöyle özetlenebilir:

1) Programcı daha derleme aşamasında ortada DLL yokken DLL fonksiyonunu prototipte __declspec(dllimport) belirleyicisini kullanarak çağırır. Bu durumda derleyici CALL işleminin hedefini bilmemekle birlikte dolaylı CALL komutu uygular.

2) DLL kullanan program bağlama (linking) aşamasına geldiğinde çağrılan fonksiyonu bağlayıcı (linker) DLL'in import kütüphanesinde bulur. DLL'in import kütüphenesinde tek tek export edilmiş fonksiyonların "isimleri, sıra (ordinal) ve hint numaraları" bulunmaktadır. Bu aşamada bağlayıcı çağrılan fonksiyonun bir DLL fonksiyonu olduğunu anlar. PE formatının importg tablosunu oluşturur. DLL çağrılarındaki dolaylı CALL

komutlarının operandlarını Import Address Tablosundaki uygun slotu referans edecek biçimde düzeltir. Tabii Import Lookup tablosunu ve Import Adress Tablosunu oıluşturur. Artık PE dosyası oluşturulmuştur.

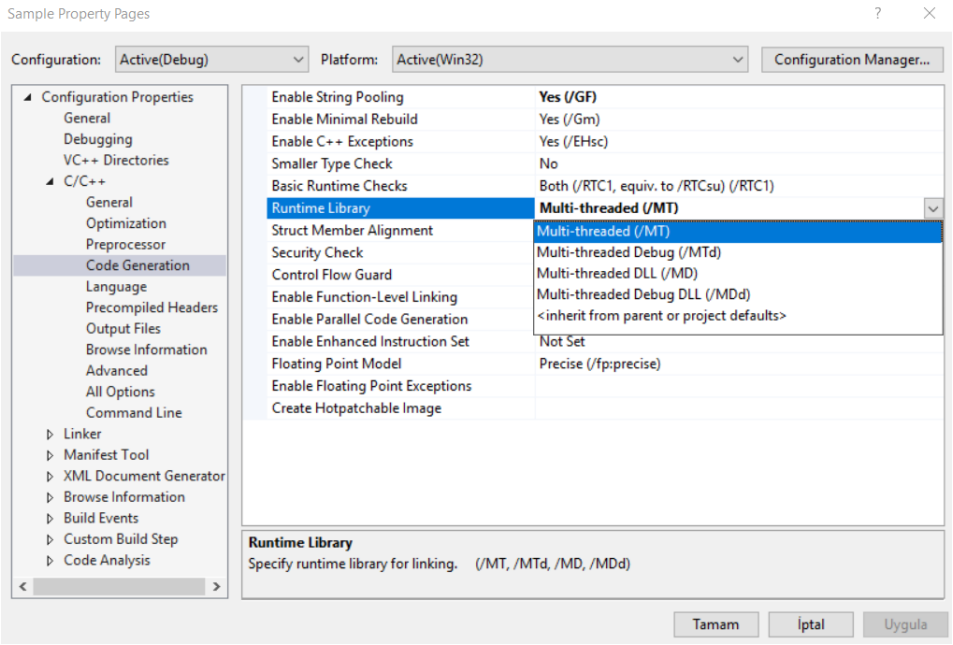
3) Program yüklenirken yükleyici çalıştırılabilen programa ilişkin PE dosyasının import tablosuna bakarak o çalıştırılabşlen dosyanın kullandığı DLL'leri tespit eder. Bu DLL'lleri bazı yerlerde sırasıyla arar. (Import tablosunda DLL'lerin yol ifadelerinin olmadığını yalnızca isimlerinin olduğunu anımsayınız). DLL'leri tek tek bularak onları adres alanına bütünsel olarak yükler. DLL'lerin export tablolarına bakar. Çalıştırılabilen dosyadan çağrılan fonksiyonları export tablosunda bulur. Onların RVA'larını yükleme adresine toplayarak gerçek sanal adresleri elde eder ve çalıştırılabilen dosyanın Import Address Tablosu girişlerini doldurur. DLL'lerin yüklenmesi özyinelemeli bir işlemdir. Çünkü DLL'ler başka DLL'lerden fonskiyonlar da çağırmış olabilirler. Bu nedenle işletim sistemi DLL'li yüklerken o DLL'lerin de import tablolarına bakarak onların gereksinim duyduğu diğer DLL'leri de yüklemektedir. Böylece bir dizi DLL programın adres alanına yüklenmiş olur.

Çok DLL kullanan programların yüklenmesinin biraz gecikeceği söylenebilir. Bunun için Microsoft bazı teknikler düşünmüştür. Ancak programcı eğer programının yüklenme zamanı bu nedenle çok uzuyorsa bazı DLL'leri programın belli aşamalarında onlar kullanılacağı zaman dinamik olarak yükleyebilir.

Aslında Windows işletim sistemi farklı prosesler aynı DLL'i kullandığında onları boşuna fiziksel belleğe yeniden yüklemez. Onların yalnızca bir kopyasını fiziksel belleğe yükleyip proseslerin sayfa tabloları yoluyla onların aynı fiziksel belleğe erişmelerini sağlar. Pekiyi durumda bir proses bir DLL'de değişiklik yaptığında ne olacaktır? İşte işletim sistemi böyle bir değişiklik yapıldığında değişikliğin yapıldığı sayfanın fiziksel kopyasını o anda çıkartmaktadır. Bu tekniğe "copy on write" denilmektedir. Windows sistemlerinde iki proses DLL tekniğiyle haberleşme de yapabilmektedir. Şöyle ki: İki proses aynı DLL'i kullanır. Ancak DLL'de bir bölüm (section) yaratılarak o bölüm default "copy on write" olmaktan çıkartılır. Böylece proseslerden biri DLL'e yazma yaptığında diğeri onu görebilir.

Windows işletim sisteminin tüm API fonksiyonları yine DLL'lerin içerisindedir. Windows'un en temel üç DLL'i "kernel32.dll", "user32.dll" ve "gdi32.dll" isimli DLL'lerdir. Bu DLL'lerin sisimlerinde 32 geçtiğine bakmayınız. Bunlar aslında 64 bit Windows sistemlerinde 64 bit DLL'lerdir. Yalnızca geleneksel isimler kullanılmaya devam edilmiştir. "Kernel32.dll" içerisinde en temel API fonksiyonları bulunur. Örneğin proses yaratan, thread işlemlerini yapan tüm fonksiyonlar bu DLL'in içerisindedir. "User32.DLL"de GUI fonksiyonları bulunmaktadır. Örneğin pencerelerle ilgili işlemler yapan tüm fonksiyonlar bu DLL'lin içerisindedir. "GDI32.dll" Windows GDI denilen çizim fonksiyonlarının bulunduğu kütüphanedir. Bir C programını derlediğimizde derleyici tarafından yerleştirilen "başlangıç kodu (startup code)" pek çok API fonksiyonunu kullandığı için tipik olarak bir C programı en azından "Kernel32.dll" çağırması yapmaktadır. Gerçekten de Visual Studio'da bir proje yaratıldığında en azından yukarıdaki üç DLL'in import kütüphanelerine otomatik referans edilmeketdir. Gerçekten de "Kernel32.dll"yi kullanmayan herhangi bir program görmek çok zayıf olasılıktır. Çünkü prosesi sonlandıran ExitProcess bile bu DLL'in içerisindedir.

C ve C++ derleyicilerindeki standart fonksiyonlar da derleyiciye ve ayarlarına bağlı olarak statik kütüphanelerde ya da dinamik kütüphanelerde bulunabilmektedir. Örneğin Visual Studio IDE'sinin son versiyonlarında default olarak hibrit bir durum söz konusudur. Kütüphane fonksiyonlarının bazıları statik olarak bağlanmakta bunlar bazı dinamik kütüphane fonksiyonlarını çağırmaktadır. Bu nedenle bir konsol uygulamasının bile bir bilgisayardan diğerine taşınırken bazı DLL'lere gereksinim duyabileceği göz önüne alınmalıdır. Eğer istenirse Visual Studio IDE'sinde stabdart C fonksiyonları tamamen statik olarak bağlanabilir. Bunun için proje ayarlarından C-C++/Runtime Library seçeneğinde DLL olmayan (Örneğin Multi-threaded) bir seçenek seçilmelidir:



Ayrıca standart C fonksiyonlarının bulunduğu DLL'lerin de Visuals Studio'nun versiyonundan versiyonuna farklılık gösterebileceğini belirtelim.

PE Dosya Enjeksiyonu (PE File Injection)

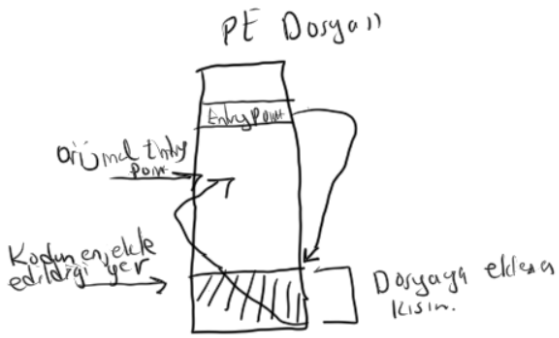
PE dosya enjeksiyonu iki biçimde yapılabilmektedir:

1) Kodun doğrudan PE dosyasının içerisinde yerleştirilmesi yöntemi
2) Çalışmakta olan bir prose başka bir prosesin kod enjekte etmesi yöntemi

Birinci yöntemde çalıştırılabilen dosya değiştirilerek içerisine kod eklenir ve bu kodun çalıştırılması sağlanır. İkinci yöntemde zaten çalışmakta olan bir program vardır. Başka bir program o programın bellek alanına kod enjekte eder ve çalışmakta olan programın belleğe yüklenmiş olan PE metadata bilgilerini değiştirir. Bu yöntemde çalışmakta pek çok programa arzu edilmeyen işlemler yaptırılabilmektedir.

Kodun PE Dosyasına Yerleştirilmesiyle Yapılan Enjeksiyon

Bu enjeksiyonun aslında basit bir fikri vardır: Çalıştırılabilen PE dosyasının bir kısmına bir kod yerleştirilir. Sonra dosyanın PE Ekbaşlığındaki "AddressOfEnrtPoint" alanı yeni kodunun RVA'sını gösterecek biçimde ayarlanır. Tabii bu kod çalıştırıldıktan sonra orijinal AddressOfEntryPoint noktasına geri dallanılır. Böylece araya girilmiş olur. Pekiyi kodun enjekte edileceği bölge neresi olmalıdır? Kod PE dosyasında kullanılmayan bir bölgeye enjekte edilebilir. Tabii oradaki bilginin bozulmaması gerekir. Bunun için ilk akla gelen yer bölümlerin (sections) arasındaki boşluklardır. Anımsanacağı gibi dosyada bir bölüm bittiğinde hemen diğeri başlamaz. Arada hizalamadan dolayı belli bir boşluk bulunabilir. Dosyadaki hizalama "PE Ek Başlığındaki (PE Optional Header)" FileAlignment alanında belirtilmektedir ve tipik olarak 512 byte'tır (1 sektör). Tabii enjekte edilecek kodu içine alabilecek böyle bir bölge hiç bulunmayabilir. Bu nedenle bunun yerine dosyaya ekleme yaparak enjekte edilecek kodu dosyanın sonuna yerleştirme yoluna da gidilmektedir. Dosyanın sonunda nasıl olsa bir bölüm (section) vardır. Bu bölümün sonuna ekleme yapılıp yükleyicinin bu kısmı da yüklemesi sağlanır.



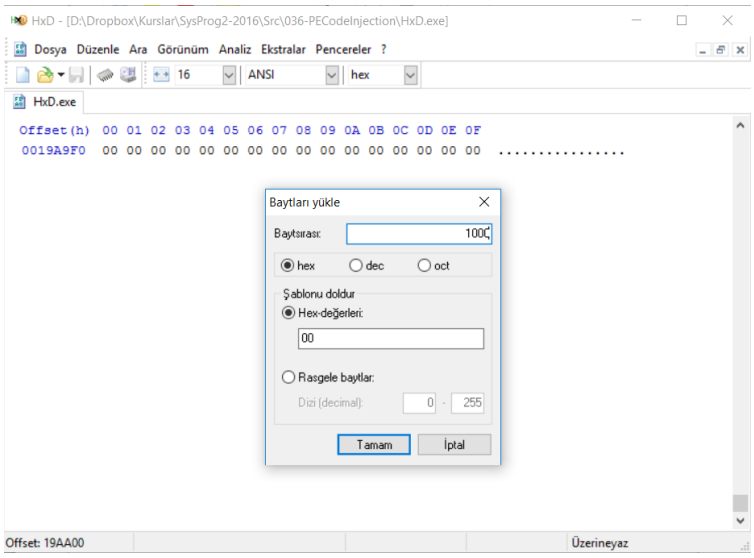
Pekiyi enjekte ettiğimiz kod ne yapacaktır? İşte burada enjeksiyon fikrinin ne olduğuna karar vermek gerekir. Enjeksiyon tamamen virüs ya da zararlı kodları (malware) çalıştırmak fikriyle yapılabileceği gibi, bazı dinamik analiz araçları için iyi niyetle de yapılabilir. Biz buradaki uygulamamızda enjekte edilen kodun bir mesaj penceresi çıkartmasını sağlayacağız.

Manuel kod enjeksiyonu için iki araç yeterlidir.

1) CFF Explorer (PE Dosyasını görüntülemek ve bazı metadatalar üzerinde işlem yapnak için)
2) HxD (Bu bedava bir hex editördür. Bazı bilgileri koda yerleştirmek için kullanılır.

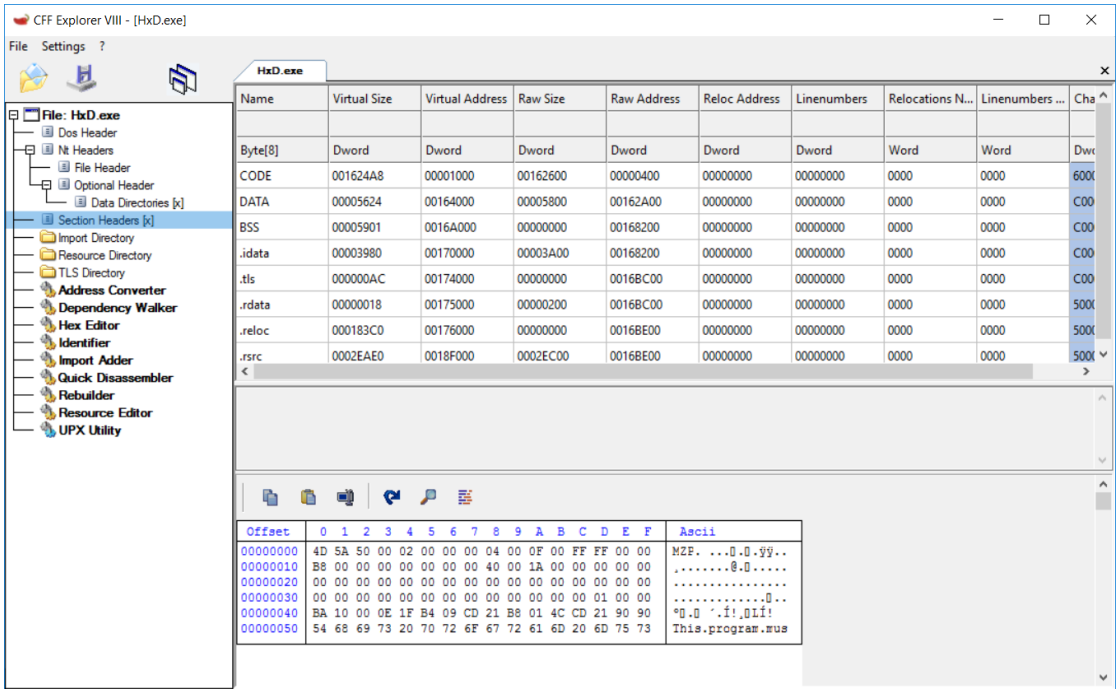
Burada HxD programının bir kopyasını alıp kendisine enjeksiyon uygulayacağız. Bu işlem için sırasıyla şu adımlar uygulanacaktır:

1) PE dosyasının sonuna belli bir boş alan eklenir. Örneğin burada 4096 byte bir alan (0x1000) eklenecektir. Bu işlem bir kodla ya da doğrudan HxD editörünün kendisiyle yapılabilir. Bunun için imleç dosyanın sonuna getirilir ve Edit/Add Bytes (Düzenle / Byte Ekle) seçilir.



Burada eklemenin PE Ek başlığında SectionAlignment değerinin katları olması gerekmektedir. 4K bir sayfa büyüklüğüdür. Eklemeler sayfa büyüklüğünün katları olmalıdır.

Biz aslında bu işlemle birlikte dosyasnın son bölümünü genişletmiş olduk. Dosya bölümlerini görüntüleyelim:



Görüldüğü gibi aslında en son bölüm olan ".rsrc" bölümüne ekleme yapmış olduk. Şimdi bu bölümü section tablosundan büyütelim:

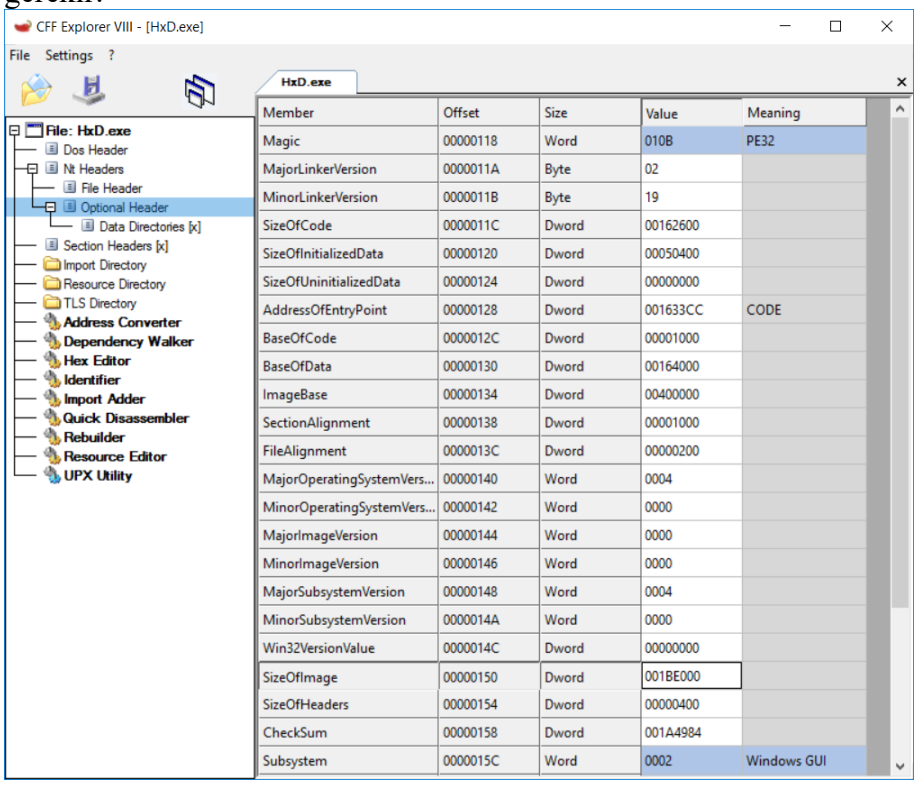
Eski hali:

.rsrc	0002EAE0	0018F000	0002EC00	0016BE00	00000000	00000000	0000	0000	500(

Yeni hali:

.rsrc	0002FAE0	0018F000	0002FC00	0016BE00	00000000	00000000	0000	0000	500(

Tabii image'in toplam uzunluğunu da (SizeOfImage) PE Ek başlığından (PE Optional Header) büyütmemiz gerekir:



SizeOdImage alanının eski hali:

SizeOfImage	00000150	Dword	001BE000

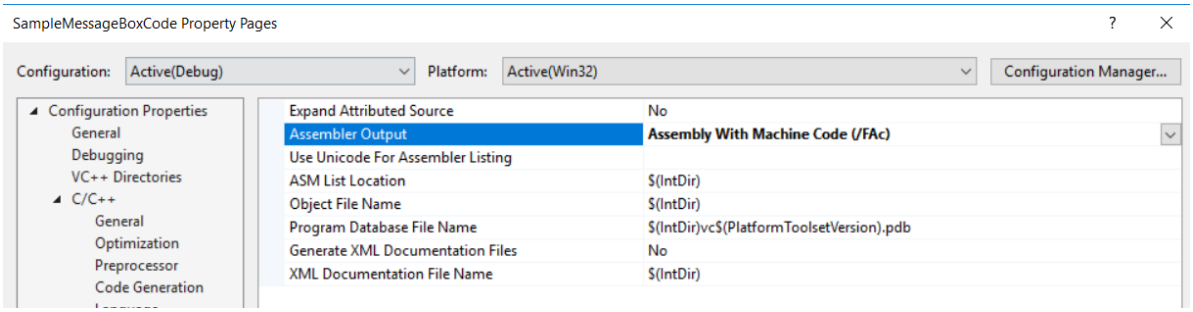
Yeni Hali:

SizeOfImage	00000150	Dword	001BF000

2) Enjekte edilecek kod belirlenir ve eklenen kısmın uygun bir yerine yerleştirilir. Biz test amacıyla ekrana bir MessageBox çıkartan kod enjekte etmek isteyelim. Bu kodun makine dili kodlarını elde etmek gerekir. Bunun önce bir MessageBox çıkartan küçük bir program yazıp oradan makine kodları alınabilir. Ancak burada dikkat edilmesi gereken nokta MessageBox API fonksiyonunun User32.DLL içerisinde olduğudur. Buradaki makine kodu Import Adress Tablosuna göreli bir CALL komutu içerecektir. Bu kodu alıp enjekte edilecek dosyaya gömeceğimiz zaman bu MessageBox göreli adresinin yeni programa göre değiştirilmesi gerekir. Ayrıca MessageBox çağrılırken yazıların başlangıç adreslerinin de enjekte edilen kodda değiştirilmesi gerekmektedir. Böyle bir basit MessageBox çıkartan program aşağıdaki gibi yazılabilir:

```
#include <windows.h>

int main(void)
{
    MessageBoxW(0, L"Test", L"Injection", MB_OK);

    return 0;
}
```

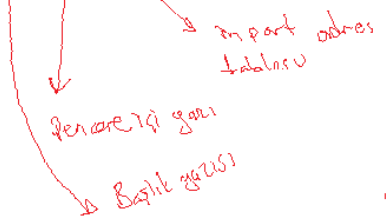
Biz programın MessageBox çıkartan kısmını Dumpbin /DISASM seçeneğiyle elde edbiliriz. Ya da doğrudan derleme sırasında “Assembly With Machine Code (/FAc)” seçeneği ile de elde edebiliriz.



Elde edilen .COD dosyası içerisinde MessageBox çıkartan kısım şöyledir:

```
00020  6a 00              push     0
00022  68 00 00 00 00     push     OFFSET ??_C@_1BE@ELJFDIJD@?$AAI?$AAn?$AAj?$AAe?$AAc?$AAt?$AAi?$AAo?$AAn?$AA?$AA@
00027  68 00 00 00 00     push     OFFSET ??_C@_19FNGKFHMA@?$AAT?$AAe?$AAs?$AAt?$AA?$AA@
0002c  6a 00              push     0
0002e  ff 15 00 00 00 00  call     DWORD PTR __imp__MessageBoxW@16
```

```
00020   6a 00           push    0
00022   68 00 00 00 00  push    OFFSET ??_C@_1BE@ELJFDIJD@?$AAI?$AAn?$AAj?$AAe?$AAc?$AAt?$AAi?$AAo?$AAn?$AA?$AA@
00027   68 00 00 00 00  push    OFFSET ??_C@_19FNGKFHMA@?$AAT?$AAe?$AAs?$AAt?$AA?$AA@
0002c   6a 00           push    0
0002e   ff 15 00 00 00 00 call  DWORD PTR __imp__MessageBoxW@16
```



MessageBox'ın pencere başlığında "Test" yazısı, pencerenin içerisinde de "Injection" yazısı çıkacak olsun. Biz bu iki yazıyı dosyada eklediğimiz alanda bir yere UNICODE olarak yerleştirelim:

```
0019A940  54 00 65 00 73 00 74 00 00 00 00 00 00 00 00 00  T.e.s.t.........
0019A950  49 00 6E 00 6A 00 65 00 63 00 74 00 69 00 6F 00  I.n.j.e.c.t.i.o.
0019A960  6E 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  n...............
```

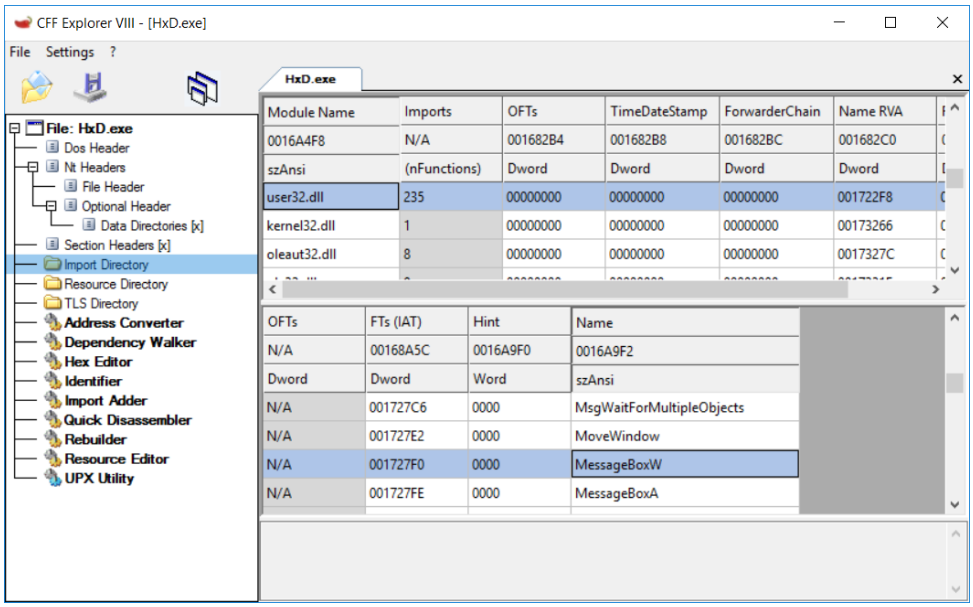
Görüldüğü gibi "Test" yazısının başlangıç dosya offseti 0x19A940'tır. Bunun belleğin tepesinden itibaren sanal bellek adresi CFF Explorer'da aşağıdaki gibi elde edilebilir:

VA	005BDB40
RVA	001BDB40
File Offset	0019A940

Görüldüğü gibi sanal adres 0x5BDB40 biçimindedir. Şimdi de benzer biçimde "Injection yazısının sanal adresini bulalım. Bu yaznın da dosya offseti 0x19A950'dir.

VA	005BDB50
RVA	001BDB50
File Offset	0019A950

Bunun da sanal bellek adresi 0x5BDB50'dir. Artık kodda tek bir kalmıştır. O da MessageBoxW fonksiyonunun import tablosu adresidir. Bu adres tabii enjekte edilecek koddaki adres olmalıdır.



Yukarıdaki şekilde enjeksiyonun yapıldığı HxD.exe dosyası içerisinde MessageBoxW fonksiyonun import adres tablosundaki yeri görülmektedir. Ancak bizim isteğimiz bu import adres tablosu slotunun adresidir. Çünkü enjekte edeceğimiz kod DLL çağırması için buraya dolaylı CALL işlemi uygulayacaktır. CFF.Explorer ilgili slotun adresini (ikinci satır) bize "Import Lookup Tablosu" olarak vermektedir. Halbuki bizim "Import Address Tablosu" slotunun adresine ihiyacımız vardır. Kendi yazdığımız program bunu bize vermektedir:

```
Slot Address: 0x0017085C, Slot Value: 0x001727F0, Hint: 0 (0x0000), MessageBoxW
```

Biz kendi programımızda Slot Adresi RVA olarak yazdırmıştık bunu sanal adrese dönüştürebiliriz (Yükleme adresi olan 0x40000 değerini toplayarak):

VA	0057085C
RVA	0017085C
File Offset	00168A5C

Buradan sanal adresin 0x57085C olduğu görülmektedir.

Kodu yerleştirirken aşağıdaki alanların sanal adres (belleğin tepesinden itibaren) belirtecek biçimde güncellenmesi gerekir. Ayrıca bizim bu kodun sonuna orijinal EntryPoint'e JMP komutu eklememiz de gerekir. Orijinal Entry Point PE Ek başlığında belirtilmektedir:

AddressOfEntryPoint	00000128	Dword	001633CC

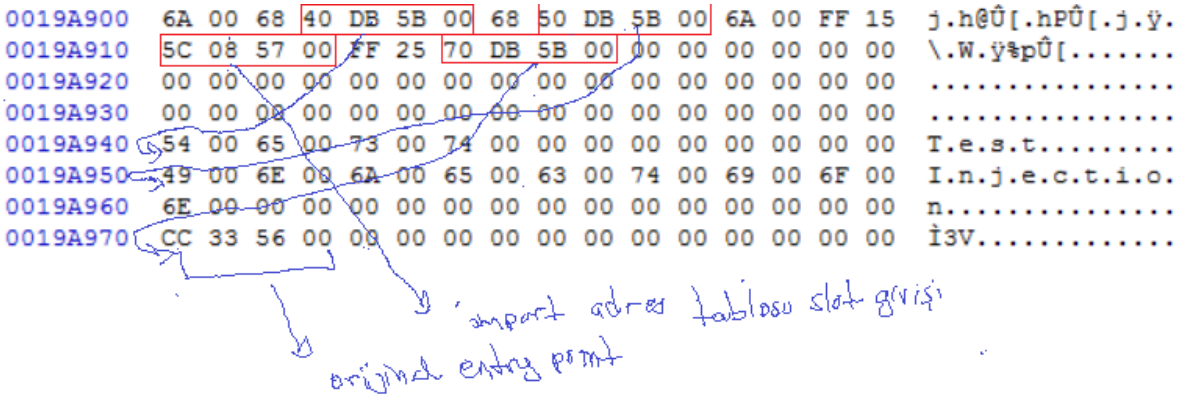
Burada orijinal Entry Point'in 0x1633CC olduğu görülmektedir. Bu değer RVA cinsindendir. Sanal bellekteki yeri 0x40000 ekleyerek bulunabilir:

VA	005633CC
RVA	001633CC
File Offset	001627CC

Burada dallanılacak sanal adres 0x5633CC'dir. Ancak Intel'de JMP makine komutunun operandı göreli uzaklık değeri almaktadır. Bu değerin de hesaplanması zordur. Bunun yerine orijinal Entry Poin'te dolaylı JMP uygulamak daha pratiktir. Bunu sağlamak için yukaıdaki adresi kodda bir yere yazıp o yerin RVA sını kullanmak uygun olur. Intel'de dolaylı JMP komutu FF 25 biçimindedir. Bu komut operand olarak JMP edilecek yerin sanal adresini alır. O halde enjekte edilecek kod aşağıdaki oluşturulur:

```
0019A900  6A 00 68 40 DB 5B 00 68 50 DB 5B 00 6A 00 FF 15
0019A910  5C 08 57 00 FF 25 70 DB 5B 00 00 00 00 00 00 00
0019A920  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0019A930  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0019A940  54 00 65 00 73 00 74 00 00 00 00 00 00 00 00 00
0019A950  49 00 6E 00 6A 00 65 00 63 00 74 00 69 00 6F 00
0019A960  6E 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0019A970  CC 33 56 00 00 00 00 00 00 00 00 00 00 00 00 00
```

Burada makine komutlarında doldurulan yerleri şöyle gösterebiliriz:



Son olarak artık PE formatındaki orijinal entry point değiştirilerek enjekte edilen kodu gösterecek hale getirilmelidir: Kodumuz dosyanın 0x19a900 offsetindedir. PE Ek başlığında entry point RVA olarak verilmiştir. Bizim oraya bunu RVA olarak yazamamız gerekir:

VA	005BDB00
RVA	001BDB00
File Offset	0019A900

Görüldüğü gibi RVA 0x1BDB00'dır. Şimdi de orijinal entry point'i değiştirelim:

AddressOfEntryPoint	00000128	Dword	001BDB00

Burada biz manuel bir enjeksiyon uyguladık. Aslında bu işlem tamamen bir kodla yapılabilir. Aslında bunu programlama yoluyla yapabilecek alt yapıya sahip durumdayız. 034-PortableExecutableFile isimli örnek kodda bunu yapabilecek temel kodlar bulunmaktadır. Programlama yoluyla enjeksiyon şu adımlar izlenerek yapılabilir:

1) PE dosyası açılarak dosyanın sonuna belli bir uzunlukta boş (0'larla dolu) bir bölge eklenir.

2) Bölüm başlıklarından dosyanın sonu hangi bölümdeyse (genellikle son bölümdür ama zorunlu değildir) o bölümün VirtualSize ve RawSize uzunlukları güncellenir.

3) PE Ek başlığındaki SizeofImage alanı güncellenir.

4) Kod dosyanın sonuna yerleştirilir. Eğer kod DLL kullanıyorsa dolaylı CALL komutunun operandı import adres tablosundaki yeri gösterecek biçimde değiştirilir. Ayrıca kodda birtakım başka adres kullanılıyorsa bunların güncellenmesi sağlanır. Yerleştirilecek kodunsonuna orijinal entry point dallanması eklenir.

5) Programın orijial entry point'i yeni yerleştirilen kodu gösterecek biçimde değiştirilir.

64 Bit PE (PE32+) Formatı

64 bit PE formatına PE32+ da denilmektedir. PE32+ formatı 64 bit işletim sistemleri için PE32 formatının 64 bite uyarlanmış halidir. Microsoft PE32 ve PE32+ formatlarını aynı doküman içerisinde birlikte ele almıştır. Aslında PE32+ formatı yapı bakımından 32 bit PE formatından farklı değildir. Her iki formatın da başlık kısımları bölümleri alan olarak aynıdır. PE32 ile PE32+ formatları arasındaki tek fark PE32+ formatında adres belirten bazı alanların 4 byte yerine 8 byte olmasıdır. 64 bit Windows sistemlerinde kullanılan PE formatı aslında sanıldığı gibi tam 64 bit değildir. Zaten Microsoft bu yüzden bu formatı PE32+ biçiminde isimlendirmiştir. Aslında PE32+ formatında RVA alanları (yani yükleme yerinden göreli uzaklık) yine 4 byte'ta bırakılmıştır. Microsoft'un gerek PE32 gerekse PE32+ formatları dosya olarak 2GB'yi aşamamaktadır.

PE32+6 formatında 64 bite yükseltilmiş önemli alanlar şunlardır:

ImageBase: PE ek başlığında PE formatının sanal belleğe yüklenme adresi PE32 formatında 4 byte PE32+ formatında 8 byte yer kaplamaktadır.

SizeOfStackReser ve SizeOfStackCommit: Stack için ayrılacak ve commit edilecek alan PE32 formatında 4 byte PE32+ formatında 8 byte yer kaplamaktadır.

SizeOfHeapReser ve SizeOfHeapCommit: Heap için ayrılacak ve commit edilecek alan PE32 formatında 4 byte PE32+ formatında 8 byte yer kaplamaktadır.

PE formatının PE32 mi yoksa PE32+ mı olduğu PE Ek başlığındaki Magic elemanına bakılarak belirlenir. Bu WORD eleman 0x010B ise format PE32, 0x020B ise format PE32+ biçimindedir.

ELF (Executable and Linkable Format) Formatı

Önceki konularda da belirtildiği gibi bugün UNIX/Linux sistemlerinde ağırlıklı olarak ELF formatı kullanılmaktadır. (Mac OS X sistemleri ELF formatı değil Mach-O formatını kullanmaktadır) Aslında ELF formatı yapı bakımından PE formatına benzerdir. Bu format da bölümlerden oluşur. Bu formatın da başında bir başlık kısmı vardır. Bu başlık kısmında çeşitli meta-data bilgileri bulunur. Burada biz ELF formatını çok detaylı ele almayacağız. Ancak format hakkında orijinal dokümanlar kursumuzun "Doc" klasöründe bulunmaktadır. ELF formatı da 32 bit ve 64 bit olmak üzere iki biçime sahiptir. Bu formatlar farklı dıkümanlarda ayrı ayrı değerlendirilmiştir. Bugün kullandığımız UNIX/Linux sistemleri 64 bit olduğu için ve bu sistemlerdeki gcc derleyicileri default 64 bit derleme yaptığı için (Microsft'un Visuasl Studio IDE'si default 32 bit derleme yapmaktadır) ELF64 formatıyla daha çok karşılaşılmaktadır. Ancak ELF formatının da 32 bit ve 64 bit versiyonları tıpkı PE formatında olduğu gibi yapı bakımından aynıdır. Yalnızca bazı alanlarının genişlikleri farklıdır.

UNIX/Linux sistemlerinde ELF formatını incelemek için "objdump" ve "readelf" isimli iki araç hazır bulunmaktadır. Bu araçlar "binutils" paketi içerisindedir. Bu paket de zaten neredeyse tüm Linux dağıtımlarında default biçimde yüklenmektedir.

ELF formatı üzerinde dosya yine bellek tabanlı olarak açılıp analizler yapılabilir. Ancak ELF için özel hazırlanmış BFD (Binary File Descriptor Library) denilen bir kütüphane vardır. Aslında "binutils" içerisindeki araçlar ("ld" bağlayıcısı da dahil olmak üzere) aşağı seviyeli işlemlerde bu BFD kütüphanesini kullanmaktadır. BFD kütüphanesinin dokümanlarına GNU dokğmanlarından erişilebilir. Kütüphane "bfd.h" isimli bir başlık dosyasını kullanmaktadır. Bağlama aşamasında "libbfd.a" kütüphanesinin "-l bfd" seçeneği ile bağlama işlemine sokulması gerekir.

ELF dosyasının başında bir ELF başlığı vardır. 32 bit ve 64 bit ELF başlıkları aşağıdaki gibidir:

```
#define EI_NIDENT       16

typedef struct {
        unsigned char   e_ident[EI_NIDENT];
        Elf32_Half      e_type;
        Elf32_Half      e_machine;
        Elf32_Word      e_version;
        Elf32_Addr      e_entry;
        Elf32_Off       e_phoff;
        Elf32_Off       e_shoff;
        Elf32_Word      e_flags;
        Elf32_Half      e_ehsize;
        Elf32_Half      e_phentsize;
        Elf32_Half      e_phnum;
        Elf32_Half      e_shentsize;
        Elf32_Half      e_shnum;
        Elf32_Half      e_shstrndx;
} Elf32_Ehdr;
```

```
typedef struct
{
        unsigned char   e_ident[16];    /* ELF identification */
        Elf64_Half      e_type;         /* Object file type */
        Elf64_Half      e_machine;      /* Machine type */
        Elf64_Word      e_version;      /* Object file version */
        Elf64_Addr      e_entry;        /* Entry point address */
        Elf64_Off       e_phoff;        /* Program header offset */
        Elf64_Off       e_shoff;        /* Section header offset */
        Elf64_Word      e_flags;        /* Processor-specific flags */
        Elf64_Half      e_ehsize;       /* ELF header size */
        Elf64_Half      e_phentsize;    /* Size of program header entry */
        Elf64_Half      e_phnum;        /* Number of program header entries */
        Elf64_Half      e_shentsize;    /* Size of section header entry */
        Elf64_Half      e_shnum;        /* Number of section header entries */
        Elf64_Half      e_shstrndx;     /* Section name string table index */
} Elf64_Ehdr;
```

Figure 2. ELF-64 Header

Görüldüğü gibi iki başlığın içeriği de aynıdır ancak alan uzunlukları farklıdır. Buradaki typedef isimlerinin ne anlama geldiği iligi dokümanlarda açıklanmıştır:

Figure 1-2: 32-Bit Data Types

Name	Size	Alignment	Purpose
Elf32_Addr	4	4	Unsigned program address
Elf32_Half	2	2	Unsigned medium integer
Elf32_Off	4	4	Unsigned file offset
Elf32_Sword	4	4	Signed large integer
Elf32_Word	4	4	Unsigned large integer
unsigned char	1	1	Unsigned small integer

Table 1. ELF-64 Data Types

Name	Size	Alignment	Purpose
Elf64_Addr	8	8	Unsigned program address
Elf64_Off	8	8	Unsigned file offset
Elf64_Half	2	2	Unsigned medium integer
Elf64_Word	4	4	Unsigned integer
Elf64_Sword	4	4	Signed integer
Elf64_Xword	8	8	Unsigned long integer
Elf64_Sxword	8	8	Signed long integer
unsigned char	1	1	Unsigned small integer

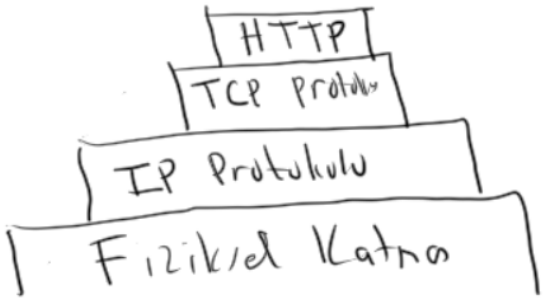
Yapıların e_ident elemanı ELF dosyası hakkında bazı temel bilgileri vermektedir. Yapıların e_type elemanı ELF dosyasının türünü belirtir. (Yani dosya çalıştırılabilir bir dosya mıdır, object modeule dosyası mıdır vs). Yine başlıkta ELF bölümlerinin sayısı ve bunların yerleri bulunmaktadır. Genel organizasyon PE formatına oldukça benzemektedir.

IP Ailesinin Uygulama Katmanına İlişkin Önemli Protokolleri

"Sistem Programlama ve İleri C Uygulamaları I" kursunda IP ailesi genel olarak tanıtılmıştı ve TCP/IP ile UDP/IP uygulamaları temel düzeyde ele alınmıştı. O kursata biz WİnSock ve BSD soket API'lerini incelemiş ve TCP'de client-server uygulamalar yapmıştık. Bu bölümde IP ailesinin uygulama katmanına (yani TCP ve UDP üzerine oturtulmuş) ilişkin prokoller üzerinde duracağız. Bu protokoller büyük ölçüde TCP üzerine oturtulmuştur. Ancak bunlar aslında aşağı seviyeli protokoller hakkında belirlemelerde bulunmazlar. Zaten mesajlaşmanın sağlandığı bir ortam dikkate alınarak kendi kurallarını ortaya koymuşlardır. Yani buradaki protokoller aslında istenirse başka ailelerde (hatta başka bağlantı modellerinde de) benzer biçimde kullanılabilirler. IP ailesinin uygulama katmanı protokolleri denildiğinde akla ilk olarak HTTP, Telnet, SSH, FTP, POP, SMTP" gibi protokoller gelmektedir. Bu bölümde bu protokollerin bir bölümü temel düzeyde ele alınacaktır. Pek çok framework ve kütüphane zaten bu protokolleri işletecek hazır API'lere ya da sınıflara sahiptir.

HTTP Protokü ve World Wide Web

HTTP World Wide Web (WWW) denilen Web sayfalarının client ve server arasında transfer edilmesi amacıyla kullanılan temel bir protokoldür. Burada Web tarayıcıları client program görevindedir. Web sunucuları da (Örneğin Apache, Microsoft IIS Server gibi) server görevini yapmaktadır.



HTTP protokolünün 3 versiyonu vardır: HTTP/1.0, HTTP/1.1 ve HTTP/2.0. Bu versiyonlar birbiriyle uyumludur. En çok kullanılan versiyon HTTP/1.1 olmuştur. Biz kursumuzda zaten GTTP/1.1 ve HTTP/2.0 arasındaki farklılıklar üzerinde durmayacağız. Her iki versiyonda da ortak olan temel unsurlar üzerinde duracağız.

HTTP Protokolü CERN'de Tim Berners-Lee tarafından 1989 yılında geliştirilmeye başlanmıştır. Geliştirilme süreci Internet'in bir krumu olan "Internet Engineering Task Force (IETF)" tarafından koordine edilmiştir.

IETF'nin oluşturduğu dokümanlara geleneksel olarak RFC (Request For Comments) denilmektedir. Her RFC'nin bir numarası ve versiyonu vardır. HTTP'nin ilk kullanımları 90'lı yılların başlarında başlatılmıştır. Böylece dünyanın ilk Web siteleri de 90'lı yılların başlarında deneme amaçlı oluşturulmuştur. Internet'e evlerden modemlerle "dial up" bağlanma yazlnızca Türkiye'de değil tüm dünyada 90 yılların ortalarında başlamıştır. Bu yıllara kadar Internet'e yalnızca Üniversitelerden ve büyük kurumların "main-frame"lerinden girilebiliyordu.

HTTP protokolüne ismini veren "Hyper Text" bir text doküman içerisinde başka bir dokümana referans edilmesi anlamına gelmektedir. Heyper text dokümanların içerisindeki bu linklere "hyperlink" denilmektedir. HTTP ve WWW temelde dokümanlar arasında gezinmeyi sağlayabilen bir özelliğe sahiptir.

HTTP protokolünde eb temel olarak olarak client program server'a bağlanarak ondan bir web sayfasının içeriğini ister. Server'da web sayfasının içeriğini text bir biçimde HTML denilen dekleratif bir dille client'a gönderir. Client aldığı HTML kodlarını yorumlar ve onu GUI biçimde görüntüler. Client programa genellikle "tarayıcı (browser)" denilmektedir.

Tabii burada HTTP'nin basit ve temel kullanımı gösterilmiştir. Oysa protokolün bazı ayrıntıları vardır. Server client'a web sayfasını içeriğini gönderdiğinde o içerikteki bazı öğeler için (örneğin resimler vs. olabilir) client server'dan yeniden istekte bulunabilmektedir. Böylece bazen bir web sayfasının görüntülenmesi birden fazla client-server haberleşmesiyle yapılabilmektedir.

HTML (Hyper Text Markup Language) de çeşitli versiyonlara sahip dekleratif bir dildir. Bugün HTML'in son versiyonu HTML 5'tir. HTML 5 ile birlikte dile pek çok yenilik eklenmiştir. Artık HTML yalnızca sttaik web sayfaları için değil dinamik web sayfaları için de kullanılabilir özelliklere sahip olmuştur.

Web siteleri halk arasında "statik" ve "dinamik" olmak üzere iki kısma ayrılmaktadır. Statik web sitelerinde server istek karşısında client'a daha önce hazırlanmış hazır HTML dosyasının içeriğini gönderir. Böylece her client her durumda hep aynı görüntürü görür. Örneğin kullanıcıya yalnızca bilgi veren basit web siteleri statik web sitelerine örnektir. Dinamik web siteelerinde client istekte bulunduğunda server tarafta çalışan bir yorumlayıcı mekanizma o client'ın o anki durumuna bağlı olarak bir HTML kodu üretip onu gönderebilmektedir. Örneğin kullanıcı bir formu doldurduğunda formu alan server doldurulanlara bağlı olarak bir sayfa içeriği üreterek client' gönderiyor olabilir. Bu dinamik bir web sitesine örnektir. Dinamik web sitelerinde server tarafta HTML kodunun üretilmesini sağlayan bir yorumlayıcı (interpreter) mekanizma vardır. İşte dinamik web siteleri server tarafta kullanılan teknolojiye göre isimlendirilebilmektedir. Dinamik web sitelerinin oluşturulması için en çok kullanılan araçlardan biri PHP denilen bir dildir. PHP yorumlayıcı biçimde çalışır. Server tarafta geeliştirme yapan PHP programcısı client'ın istediği web sayfası içeriğini HTML olarak PHP ile üretir. Yine server tarafta Microsoft'un ASP denilen (ASP .NET, ASP .NET MVC gibi) teknolojisi de yoğun olarak kullanılmaktadır. Bu teknoılojide server'daki HTML kodu C#, VB.NET gibi dillerle oluşturulmaktadır. Yine dinamik web siteleri için Java tabanlı JSP gibi teknolojiler tercih edilebilmektedir. Bunların dışında server tarafta HTML üretmek için kullanılan pek çok programa dili (örneğin Python, Perl gibi) ve araç kullanılmaktadır. Genel bir araç olarak CGI (Commom Gateway Interface) denilen bir teknolojide vardır. CGI'da server tarafta herhangi bir dille (örneğin C, C++) HTML üretici bir program yazılır. Bu program web server tarafından çalıştırılır. Bu programın stdout dosyasına yazdıkları client tarafa server tarafından gönderilir.

Dinamik web siteleri arka planda VTYS'lerini de çok yoğun kullanabilmektedir. Gerçekten de bugün VTYS'lerin en önemli kullanım mekanizması web işlemlerindedir. Örneğin client bir formu doldurdup server'a

gönderdiğinde server bunu veri tabanına yazar. Client veritabından çeşitli kayıtları görüntülemek isteyebilmektedir. VTYS için daha çok bedava MySql, Postgre, H2 gibi VTYS'ler tercih edilmektedir. Microsoft dünyasında ağırlıklı olarak SqlServer kullanılmaktadır.

HTTP protokolünü öğrenirken bir server programın hazır olarak bulunmasında fayda vardır. Bu server bir hosting firmasından dolaylı olarak tedarik edeilebileceği gibi yerel makineye de kurulabilir. Web Server olarak ilk akla gelenler Açık kaynak kodlu Apache, Microsoft'un IIS'i ve NGinX yazılımlarıdır. Apache, Microsoft IIS ve NGinX Microsoft Windows sistemlerine kurulabilmektedir. Ancak Microsoft IIS Linux sistemleri için henüz gerçekleştirilmemiştir. Şu an itibarıyla Linux ve Mac OS X sistemlerinde en yoğun kullanılan server "Apache"dir. Ayrıca Microsoft'un server'ları ücretli yazılımlardır. Dolayısıyla Microsoft hosting bedelleri daha yüksek olma eğilimindedir.

Her ne kadar Microsoft IIS'i kendi Windows Server sistemleri için gerçekleştirdiyse de bu ürün bedava olarak normal Windows sistemlerine de kurulabilmektedir. Ancak bu sistemlerde IIS bazı kısıtlara sahiptir. Örneğin maksimum bağlanılacak client sayısı oldukça az düzeydedir. Client Windows sistemlerinde IIS'in kurulumu için "Denetim Masası/Programlar ve Özellikler" menüsüne gelinir. Buradan "Windows Özelliklerini Aç veya Kapat" seçilir. Buradan da "Internet Information Services" seçilerek kurulum yapılır.

Apache hem Linux, hem Mac OS X hem de Windows sistemlerine çok kolay kurulabilmektedir. Mac OS X sistemlerinde Apache zaten default olarak bulundurulmaktadır. Yani ayrıca bir kurulum yapmaya gerek olmaz. Windows sistemlerinde hem Apache hem de IIS kurulacaksa bunların port numaralarının farklı verilmesi (örneğin birinin 80 diğerinin 8080 gibi) uygun olur.

HTTP protokolü default olarak 80 numaralı portu kullanmaktadır. Ancak HTTP server'ların hepsi başka bir port numarasını kullanacak biçimde konfigüre edilebilmektedir.

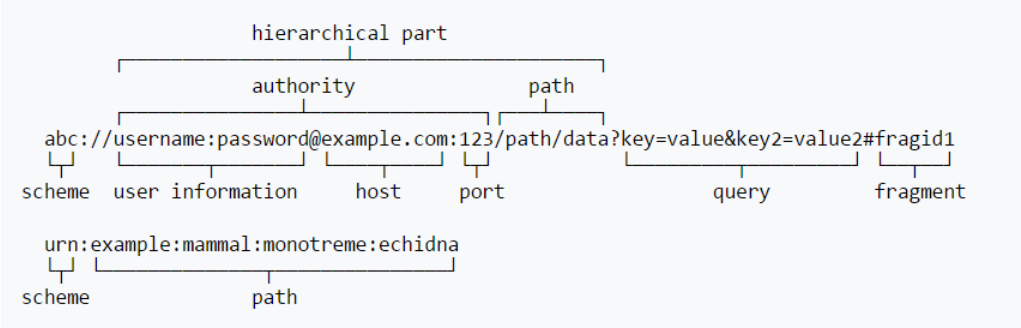
URI ve URL Kavramları

Web'te herhangi bir kaynak için onu betimleyen en genel adrese URI (Uniform Resource Identifier) denilmektedir. Bazı tür URI'ler ise URL (Uniform Resource Locator) biçiminde isimlendirilmektedir. URI kavramı ve URI sentaksı RFC-2396'da dokğmante edilmiştir (Kurs dokümanları içerisinde mevcuttur). Tıpkı dosyaların yol ifadelerinde olduğu gibi URI'ler de mutlak (absolute) ya da göreli (relative) olabilir. Mutlak URI'ler bir "scheme" ile başlatılıp ':' atomuyla sürdürülürler. Mrneğin:

"http://www.csystem.org/Text.txt"

gibi. Halbuki göreli URI'lerde bu "scheme" ve ':' atomu bulunmaz. URI'ler '/' karakterleriyla bölümlere ayrılırlar. Bunlara "yol bölümleri (path segment)" denilmektedir. Bir URI'nin genel biçimi aşağıdaki gibi gösterilebilir (Örnekler Wikipedia.org'den alınmıştır):

```
scheme:[//[user:password@]host[:port]][/]path[?query][#fragment]
```



Mutlak URI'lere şu örnekler verilebilir:

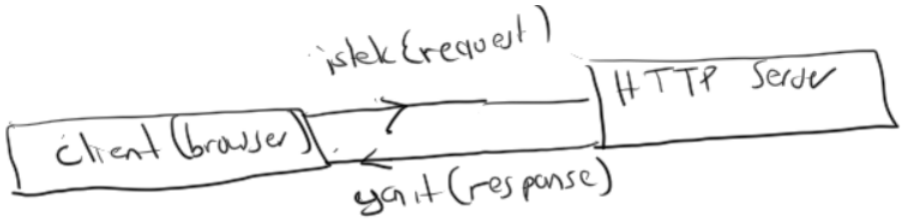
```
https://example.org/absolute/URI/with/absolute/path/to/resource.txt
https://example.org/absolute/URI/with/absolute/path/to/resource
```

Göreli URI'lere de şu örnekler verilebilir:

```
/example.org/scheme-relative/URI/with/absolute/path/to/resource.txt
/example.org/scheme-relative/URI/with/absolute/path/to/resource
/relative/URI/with/absolute/path/to/resource.txt
relative/path/to/resource.txt
../../../resource.txt
./resource.txt#frag01
resource.txt
```

HTTP Protokolünün Temel Yapısı

HTTP tipik olarak "request/response" denilen client-server modeli kullanmaktadır. "Requset/Respobse" modelinde client server'dan istekte bulunur, server da bunun karşılığında client'a yanıt verir. Tabii server isteği karşılayamazsa verdiği yanıt bir hata mesajı olabilmektedir. Bunun dışında server client'a mesaj göndermez. Server yalnızca istek (request) karşısında client'a yanıt verir. Server her istek (request) için yalnızca tek bir yanıt (response) vermektedir.



HTTP'de IP ailesinin diğer uygulama katmanındaki pek çok protokolde olduğu gibi mesajlar metinsel olarak (yani yazı biçiminde) gönderilip alınmaktadır. Yani iletişim text tabanlı yapılmaktadır.

Soket API'leriyle HTTP Server İle Mesajlaşmak

C'de aşağı seviyeli soket API'leriyle HTTP client programı yazmak oldukça kolaydır. Bir TCP soket yaratılır. Sonra HTTP server ile bağlantı kurulur. Sonra "istek (request)" mesajı gönderilir ve sonra da server'dan "yanıt (response)" mesajı alınır. Örnek bir çatı program Windows sistemleri için şöyle yazılabilir:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <WinSock2.h>
#include <Windows.h>

#define PORTNO     80             /* default well known http port */
#define HOSTNAME   "localhost"

void ExitSys(LPCSTR lpszMsg, int status, DWORD dwLastError);
int ReadLineUnbuffered(SOCKET sock, char *buf, size_t len);
int ReadSocket(SOCKET sock, void *buf, size_t len);

int main(void)
{
    WSADATA wsaData;
    struct sockaddr_in sinClient;
    SOCKET clientSock;
    struct hostent *host;
    char line[1024 + 1];
    char *buf;
    char *msgs[] = { "GET /Test.png HTTP/1.1\r\n", "Host: localhost\r\n", "\r\n", NULL };
```

```
    int result, i;
    long contentLength;

    if ((result = WSAStartup(MAKEWORD(2, 2), &wsaData)) != 0)
        ExitSys("WSAStartup", EXIT_FAILURE, result);

    if ((clientSock = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP)) == INVALID_SOCKET)
        ExitSys("socket", EXIT_FAILURE, WSAGetLastError());

    sinClient.sin_family = AF_INET;
    sinClient.sin_port = htons(PORTNO);
    sinClient.sin_addr.s_addr = inet_addr(HOSTNAME);
    if (sinClient.sin_addr.s_addr == INADDR_NONE) {
        if ((host = gethostbyname(HOSTNAME)) == NULL)
            ExitSys("gethostbyname", EXIT_FAILURE, WSAGetLastError());
        memcpy(&sinClient.sin_addr.s_addr, host->h_addr_list[0], host->h_length);
    }

    if (connect(clientSock, (struct sockaddr *) &sinClient, sizeof(sinClient)) == SOCKET_ERROR)
        ExitSys("connect", EXIT_FAILURE, WSAGetLastError());

    for (i = 0; msgs[i] != NULL; ++i)
        if (send(clientSock, msgs[i], strlen(msgs[i]), 0) == SOCKET_ERROR)
            ExitSys("send", EXIT_FAILURE, WSAGetLastError());

    contentLength = -1;
    for (;;) {
        result = ReadLineUnbuffered(clientSock, line, 1024);
        if (result == SOCKET_ERROR)
            ExitSys("recv", EXIT_FAILURE, result);
        if (result == 0)       /* socket is closed by server */
            break;
        if (result == 2)       /* header lines ended */
            break;

        if (strstr(line, "Content-Length: "))
            contentLength = strtol(line + 15, NULL, 10);
        printf(line);
    }

    if (contentLength != -1) {
        if ((buf = (char *)malloc(contentLength)) == NULL) {
            fprintf(stderr, "cannot allocate memory!\n");
            exit(EXIT_FAILURE);
        }
        if (ReadSocket(clientSock, buf, contentLength) == SOCKET_ERROR)
            ExitSys("recv", EXIT_FAILURE, WSAGetLastError());

        for (i = 0; i < contentLength; ++i)
            putchar(buf[i]);
        putchar('\n');
        free(buf);
    }

    shutdown(clientSock, SD_BOTH);
    closesocket(clientSock);

    return 0;
}

int ReadLineUnbuffered(SOCKET sock, char *buf, size_t len)
{
    char ch1 = '\0', ch2;
    size_t i;
    int result;
```

```
    for (i = 0; i < len - 1; ++i) {
        if ((result = recv(sock, &ch2, 1, 0)) == SOCKET_ERROR)
            return SOCKET_ERROR;
        if (result == 0)
            return 0;
        buf[i] = ch2;
        if (ch2 == '\n' && ch1 == '\r') {
            buf[i + 1] = '\0';
            return i + 1;
        }
        else
            ch1 = ch2;
    }
    return i;
}

int ReadSocket(SOCKET sock, void *buf, size_t len)
{
    int result;
    size_t index = 0;
    size_t left = len;
    char *cbuf = (char *)buf;

    while (left > 0) {
        if ((result = recv(sock, cbuf + index, left, 0)) == SOCKET_ERROR)
            return SOCKET_ERROR;
        if (result == 0)
            break;
        left -= result;
        index += result;
    }
    return index;
}

void ExitSys(LPCSTR lpszMsg, int status, DWORD dwLastError)
{
    LPTSTR lpszErr; b

    if (FormatMessage(FORMAT_MESSAGE_ALLOCATE_BUFFER | FORMAT_MESSAGE_FROM_SYSTEM, NULL, dwLastError,
        MAKELANGID(LANG_NEUTRAL, SUBLANG_DEFAULT), (LPTSTR)&lpszErr, 0, NULL)) {
        fprintf(stderr, "%s: %s", lpszMsg, lpszErr);
        LocalFree(lpszErr);
    }

    exit(status);
}
```

Aynı program UNIX/Linux ve Mac OS X sistemleri için de şöyle yazılabilir:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>

#define PORTNO          80              /* default well known http port */
#define HOSTNAME        "localhost"

void err_exit(const char *msg);
int read_line_unbuffered(int sock, char *buf, size_t len);
int read_socket(int sock, void *buf, size_t len);
```

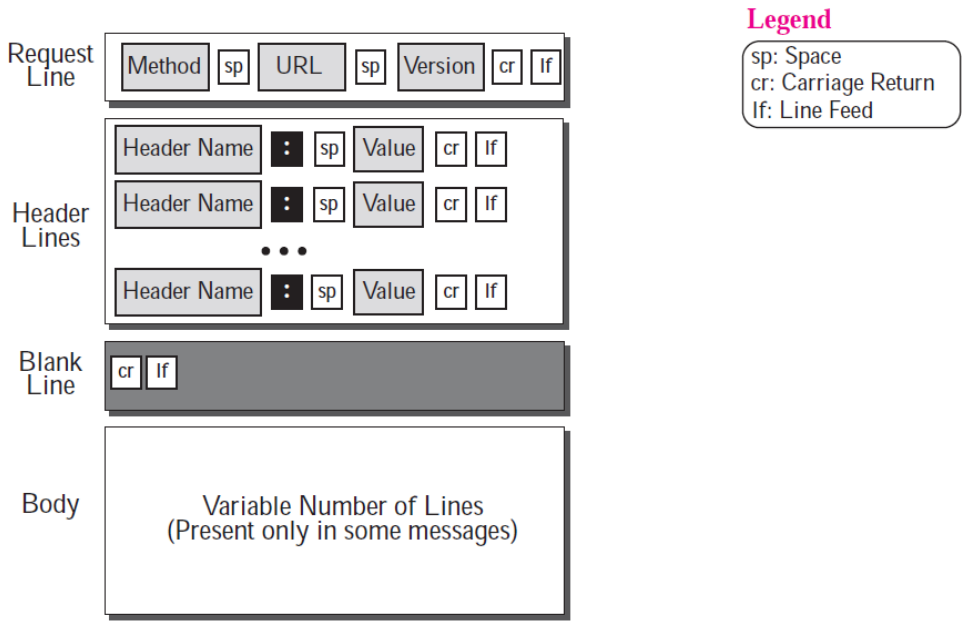
```c
int main(void)
{
    int result;
    struct sockaddr_in sinClient;
    int clientSock;
    struct hostent *host;
    char line[1024];
    char *buf;
    char *msgs[] = { "GET /Test.txt HTTP/1.1\r\n", "Host: localhost\r\n", "\r\n", NULL };
    int i;
    long contentLength;

    if ((clientSock = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP)) == -1)
        err_exit("socket");

    sinClient.sin_family = AF_INET;
    sinClient.sin_port = htons(PORTNO);
    sinClient.sin_addr.s_addr = inet_addr(HOSTNAME);
    if (sinClient.sin_addr.s_addr == INADDR_NONE) {
        if ((host = gethostbyname(HOSTNAME)) == NULL)
            err_exit("gethostbyname");
        memcpy(&sinClient.sin_addr.s_addr, host->h_addr_list[0], host->h_length);
    }
    if (connect(clientSock, (struct sockaddr *) &sinClient, sizeof(sinClient)) == -1)
        err_exit("connect");

    for (i = 0; msgs[i] != NULL; ++i)
        if (send(clientSock, msgs[i], strlen(msgs[i]), 0) == -1)
            err_exit("send");

    contentLength = -1;
    for (;;) {
        result = read_line_unbuffered(clientSock, line, 1024);
        if (result == -1)
            err_exit("read_line_unbuffered");
        if (result == 0)       /* socket is closed by server */
            break;
        if (result == 2)       /* header lines ended */
            break;

        if (strstr(line, "Content-Length: "))
            contentLength = strtol(line + 15, NULL, 10);
        printf(line);
    }

    if (contentLength != -1) {
        if ((buf = (char *)malloc(contentLength)) == NULL) {
            fprintf(stderr, "cannot allocate memory!\n");
            exit(EXIT_FAILURE);
        }
        if (read_socket(clientSock, buf, contentLength) == -1)
            err_exit("read_socket");

        for (i = 0; i < contentLength; ++i)
            putchar(buf[i]);
        putchar('\n');
        free(buf);
    }

    shutdown(clientSock, SHUT_RDWR);
    close(clientSock);

    return 0;
}
```

```
int read_line_unbuffered(int sock, char *buf, size_t len)
{
    char ch1 = '\0', ch2;
    size_t i;
    int result;

    for (i = 0; i < len - 1; ++i) {
        if ((result = recv(sock, &ch2, 1, 0)) == -1)
            return -1;
        if (result == 0)
            return 0;
        buf[i] = ch2;
        if (ch2 == '\n' && ch1 == '\r') {
            buf[i + 1] = '\0';
            return i + 1;
        }
        else
            ch1 = ch2;
    }
    return i;
}

int read_socket(int sock, void *buf, size_t len)
{
    int result;
    size_t index = 0;
    size_t left = len;
    char *cbuf = (char *)buf;

    while (left > 0) {
        if ((result = recv(sock, cbuf + index, left, 0)) == -1)
            return -1;
        if (result == 0)
            break;
        left -= result;
        index += result;
    }
    return index;
}

void err_exit(const char *msg)
{
    perror(msg);

    exit(EXIT_FAILURE);
}
```

Protokolde sonraki bölümde açıklanacağı gibi birtakım bilgiler satır satır gönderilip alınmaktadır. Satırların sonunda da protokole göre CR/LF (\r\n) karakterlerinin bulunması gerekir. Maalesef yalnızca tek bir satırı etkin biçimde okuyan bir soket API fonksiyonu yoktur. Fakat bazı framework ve kütüphanelerde bu fonksiyonlar hazır olarak bulunmaktadır. Örneklerimizde bir satırın okunması byte byte yapılmıştır. Bu yöntem biraz yavaş olmasına karşın basit bir yöntemdir.

HTTP İstek (Request) Mesajları

HTTP client program (tipik olarak tarayıcı) server'a "istek (request)" mesajı gönderir. Server da ona "yanıt (response)" mesajıyla karşılık verir. İstek mesajlarının genel biçimi şöyledir (Şekil "TCP/IP Protocol Suits-Behrouz Forouzan" kitabından alınmıştır):



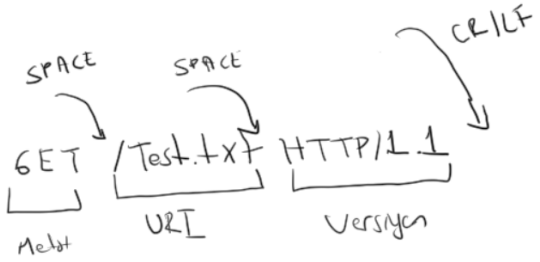
Mesaj bir istek satırıyla (Request Line) başlamktadır. Satırın hemen başında isteğin ne olduğunu anlatan bir "Method" sözcüğü vardır. Metot şunlardan biri olabilir:

Method	*Action*
GET	Requests a document from the server
HEAD	Requests information about a document but not the document itself
POST	Sends some information from the client to the server
PUT	Sends a document from the server to the client
TRACE	Echoes the incoming request
CONNECT	Reserved
DELETE	Remove the Web page
OPTIONs	Enquires about available options

Metot belirten sözcük büyük harflerle yazılmak zorundadır. HTTP mesajlarının büyük harf-küçük harf duyarlılıkları vardır. 8 metot olsa da aslında tarayıcılar bunlardan yalnızca GET ve POST metotlarını kullanmaktadır. Bazı Web sunucuları da yalnızca bu metotları işlemektedir. Diğer metotlar başka bazı teknolojiler tarafından kullanılabilmektedir. Biz de burada GET ve POST metotları üzerinde duracağız.

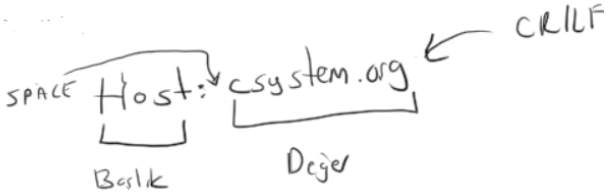
İstek satırında metot belirten sözcüğü yalnızca bir tane SPACE karakteri izler. Bunu isteğe konu olan URI (ya da URL) izlemektedir. Bundan sonra yine bir SPACE karakteri bulunur ve bunu da HTTP versiyonu izler. HTTP'nin 1.1 versiyonu için buradaki yazı "HTTP/1.1", 2.0 versiyonu için "HTTP/2.0" olmalıdır. HTTP'nin 1.1 versiyonu neredeyse tüm tarayıcılar tarafından desteklenmektedir. Ancak 2.0 için tarayıclarda bazı ayarların yapılması gerekebilmektedir. Tüm HTTP satırlarının sonu CR/LF karakter çiftleriyle bitilmelidir. Örnek bir istek satırı şöyle olabilir:

GET /test.txt HTTP/1.1



Metotta sözü edilen URL bir dosya belirtmek zorunda değildir. Kaldı ki dosya belirten URL'lerde de sunucu bu dosyanın içeriğini göndermek zorunda değildir. Örneğin talep edilen dosya bir ASP.NET dosyası ise (.aspx uzantılı) sunucu bu dosyadaki scripti çalıştırıp onun sonucunu istemciye (tarayıcıya) gönderebilir. Ya da örneğin talep edilen dosya bri PHP dosyası (.php uzantılı) dosyası ise sunucu bu dosyayı çalıştırıp sonucu istemciye gönderebilir. Yani biz URL ile bir dosya değil bir sonuç da elde edebiliriz. Tabii hangi dosya uzantılarına karşı sunucunun ne yapacağı sunucu tarafında belirtilmiştir. Bu uzantılar söz konusu değilse sunucu bize talep edilen dosyanın kendi içeriğini gönderebilir. Dosyaların türleri ve içerikleri MIME (Multipurpose Internet Mail Extensions) denilen bir standartla belirlenmiştir.

İstek mesajlarında istek satırını bir grup başlık satırı (header lines) izlemektedir. Başlık satırları bir anahtar sözcükle başlar bunu hemen bir ':' karakteri izler. Sonra bunu bir SPACE karakteri ve yazısal bir değer kısmı izler. Sonra yine başlık satırı CR/LF çiftiyle sonlandırılmaktadır. Örnek bir satır şöyle olabilir:



Başlık satırlarının sonunda boş bir satır (CR/LF) bulunur. Böylece mesajı alan sunucu başlık kısımlarının bittiğini anlar. Mesajın başlık kısımlarını mesaj gövdesi (body) izlemektedir. Gövde boş olabilir ya da herhangi bir bilgi içerebilir. Bazı metotlarda gövde boş olmaktadır, bazı metotlarda gövde bazı bilgileri içerecek biçimde dolu olur. Gövdede kaç bye olduğu "Content-Length" isimli başlık elemanında belirtilmektedir. Örneğin:

```
Content-Length: 1234
```

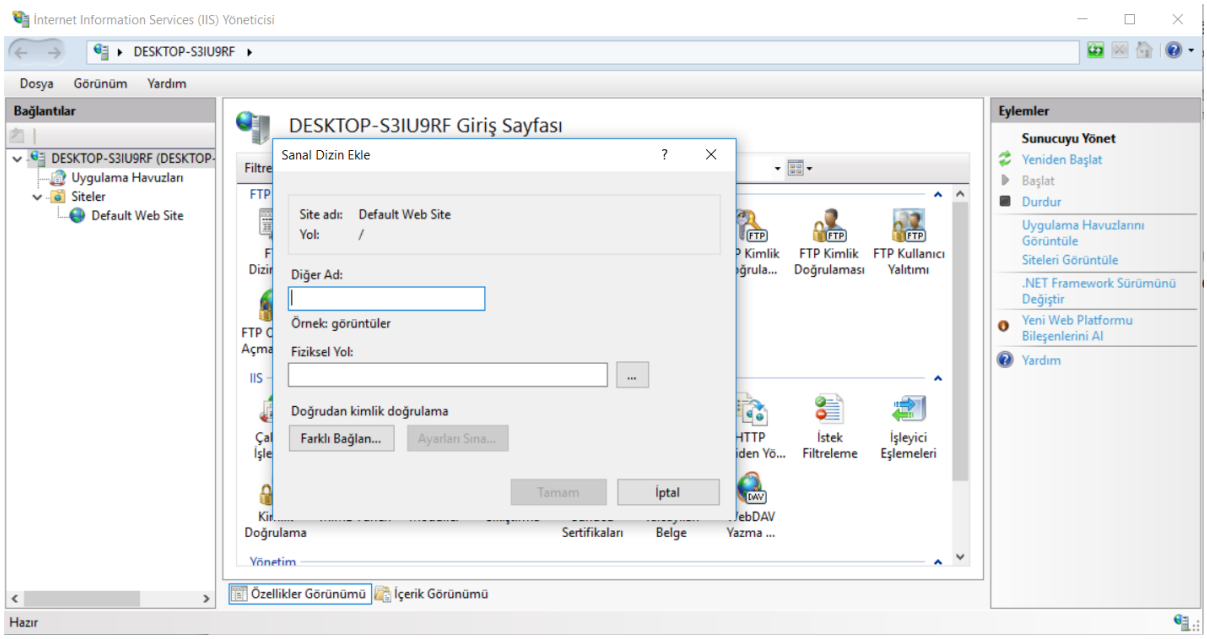
Başlık satırları şunlardan biri olabilmektedir:

Header	Description
User-agent	Identifies the client program
Accept	Shows the media format the client can accept
Accept-charset	Shows the character set the client can handle
Accept-encoding	Shows the encoding scheme the client can handle
Accept-language	Shows the language the client can accept
Authorization	Shows what permissions the client has
Host	Shows the host and port number of the client
Date	Shows the current date
Upgrade	Specifies the preferred communication protocol
Cookie	Returns the cookie to the server
If-Modified-Since	Returns the cookie to the server

HTTP istek mesajlarının başlık kısımları HTTP dokümanlarından incelenebilir. Biz burada bunlar hakkında daha detaylı bilgi veremeyeceğiz.

HTTP GET Metodu

HTTP GET metodu sunucudan istekte bulunmak için kullanılır. Yukarıda da belirtildiği gibi GET metodunda belirtilen URI gerçekte bir dosya belirtmek zorunda değildir. Buradaki URI mutlak ya da göreli olabilir. Göreli URI sunucunun belirlediği kök dizinden itibaren yer belirtir. Sunucu üzerinde sanal dizinler (virtual directory) açılabilmektedir. Örneğin:



Pek çok HTTP sunucusu birden fazla domain ile ilgili işlem yapabilmektedir. Bu nedenle sunucu için hangi domain (host name) ile ilgili istek yapıldığı da önemlidir. Örneğin Derneğimizin hismet aldığı "Dream Host" isimli hosting firması Derneğimize atadığı sunucuda yalnızca bizim Web sitemizi host etmemektedir. Başka bir deyişle www.csystem.org ile IP numarası aynı olan pek çok domain de bulunabilmektedir. İşte bu durumda biz ilgili IP ile HTTP sunucuna bağlandığımızda bizim hangi domain için istekte bulunduğumuzu sunucunun anlaması gerekir. Sunucu istek yapılan domain'i eğer o domain mutlaksa doğrudan URI'den belirler. Örneğin:

GET http://csystem.org/test.txt HTTP/1.1\r\n

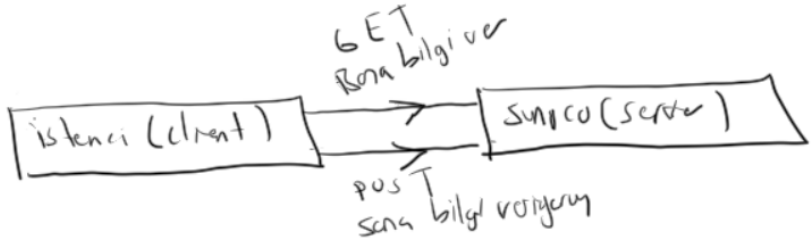
Burada istek yapılan domain www.csystem.orgdir. Ancak domain ismi göreli ise sunucu domain'i anlayamayacağı için Host isimli başlıkla bunun istek mesajında belirtilmesi gerekir:

GET /test.txt HTTP/1.1\r\n ---> İstek satırı (Request Line)
Host: www.csystem.org\r\n ---> Başlık satırı (Header Line)

GET metodunda gövde boş olur. (Bu zorunlu değildir ancak genellikle böyle olur). Gövdeler doluysa gövde kısmında kaç byte bilgi olduğu Content-Length isimli başlık satırında belirtilir.

POST Metodu

POST metodu istemciden sunucuya bilgi göndermek için kullanılmaktadır. Oysa GET metodu genellikle tam tersi amaçla yani sunucudan istemciye bilgi göndermek amacıyla kullanılır.



Aslında hem GET hem de POST metodu her iki amaçla da kullanılabilmektedir. Yani GET metodunda da istemci sunucya bilgi gönderebilir. Ancak bu işlemin POST mesajında yapılması daha güvenlidir. GET ile POST arasındaki önemli farklılıklar şunlardır:

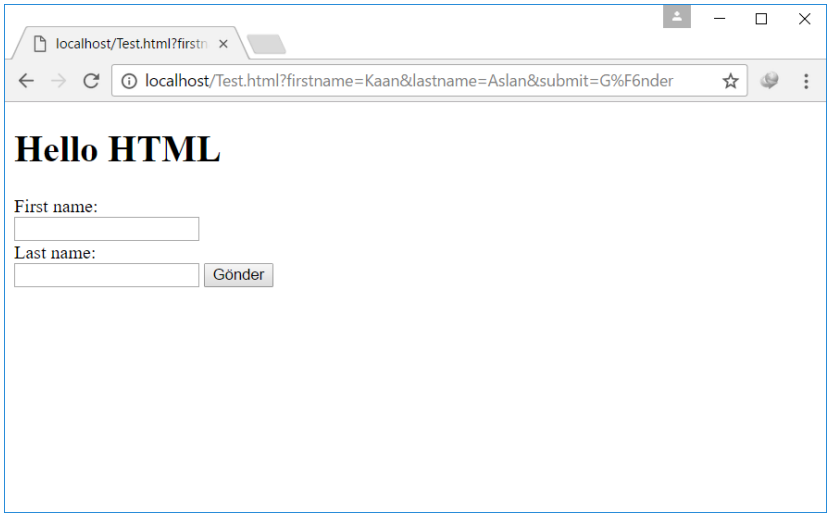
- GET metoduyla alınan bilgiler cache'lenebilir. Ancak POST için genellikle cache'leme yapılmaz.

- GET metoduyla göndeilen mesajlar tarayıcı tarafından "bookmark" yapılabilirler. Fakat POST mesajlarının tarayıcılara "bookmark" yapılmasına izin vermezler.

- GET metoduyla sunucya bilgi aktarımında aktarılacak bilgi URL'nin sorgu kısmıyla gönderilir. Bunun da bir limiti vardır. Halbuki POST metodunda bilgiler istemciden suucuya mesajların gövdelerinde aktarılırlar. Gövdeler için uzunluk sınırı yoktur.

- POST metoduyla ile aynı sayfa yeniden gönderilmeye çalışıldığında genellikle tarayıcılar bir uyarı vermektedir. Halbuki GET metodunda böyle bir uyarı verilmemektedir.

Örneğin bir HTML formunda default bilgi gönderim biçimi GET metoduyla yapılmaktadır. GET metodu bilgiyi mesaj gövdesinde değil URL'nin sorguma parçasıyla ('?' karakterinin sağ tarafı) gönderir. Örneğin:



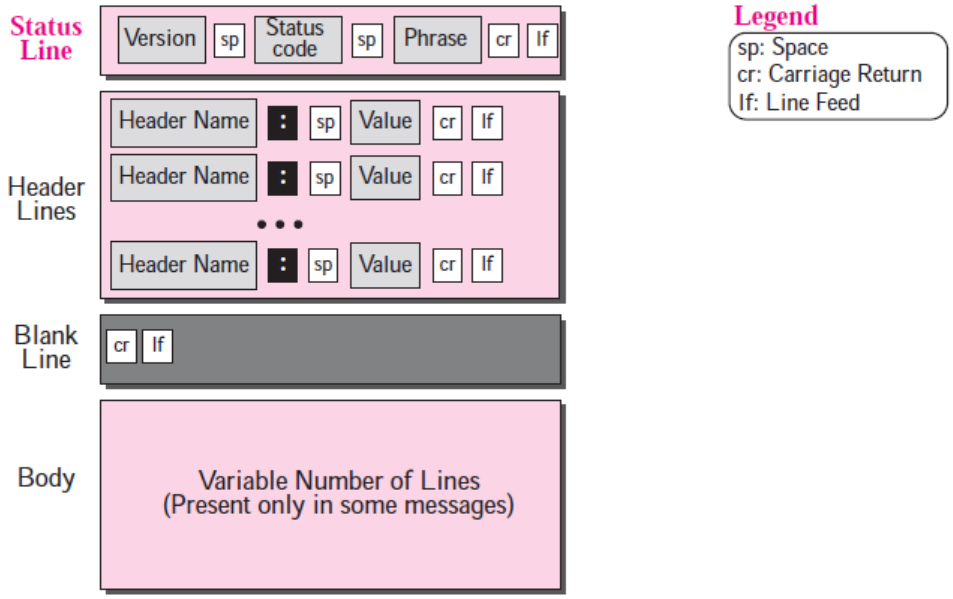

```
<!-- Test.html -->
<!DOCTYPE html>

<html>
   <head>
      <h1>Hello HTML</h1>
   </head>
   <body>
      <form>
         First name:<br>
         <input type="text" name="firstname"><br>
         Last name:<br>
         <input type="text" name="lastname">
         <input type="submit" name="submit">
      </form>
   </body>

</html>
```

Halbuki POST metodunda bilgi URL'nin parçası olarak değil mesaj gövdesinde gönderilmektedir:

HTTP Yanıt (REsponse) Mesajları

HTTP protokolğnde her istek mesajına karşı sunucu bir yanıt mesajı göndermektedir. Yanıt mesajında istenilen işlemin başarısı ve istenilen bilginin içeriği gönderilmektedir. HTTP yanıt mesajının genel biçimi şöyledir:

Figure 22.12 *Format of the response message*



Yanıt mesajı da istek mesajında olduğu gibi saatırlar oluşur. Her satırın sonu yine CR/LF ile bitmektedir. İlk satıra "Durum Satırı (status Line)" denir.

Durum satırı öce versyon numarası ile başlar. Bunu bir SPACE karakteri ve "Durum Kodu (Status Code)" izler. Örneğin 200 kodu "OK", 404 "Not Found" anlamına gelmektedir. Durum kodunu da "Durum Yazısı (Phrase)" izlemektedir. Durum yazısı durum kodunun yazısakl biçimidir. (Örneğin "OK", "Not Found" gibi).

HTTP yanıt mesajının durum kodları (status codes) şöyledir:

Table 22.3 *Status Codes and Status Phrases*

Status Code	*Status Phrase*	*Description*
	Informational	
100	Continue	The initial part of the request received, continue.
101	Switching	The server is complying to switch protocols.
	Success	
200	OK	The request is successful.
201	Created	A new URL is created.
202	Accepted	The request is accepted, but it is not immediately acted upon.
204	No content	There is no content in the body.
	Redirection	
301	Moved permanently	The requested URL is no longer used by the server.
302	Moved temporarily	The requested URL has moved temporarily.
304	Not modified	The document has not modified.
	Client Error	
400	Bad request	There is a syntax error in the request.
401	Unauthorized	The request lacks proper authorization.
403	Forbidden	Service is denied.
404	Not found	The document is not found.
405	Method not allowed	The method is not supported in this URL.
406	Not acceptable	The format requested is not acceptable.
	Server Error	
500	Internal server error	There is an error, such as a crash, at the server site.
501	Not implemented	The action requested cannot be performed.
503	Service unavailable	The service is temporarily unavailable.

Hata kodlarının gruplandığına dikkat ediniz.

Başlık satırları yine istek mesajlarında olduğu gibi bir formata sahiptir. Örneğin sunucu istemciye istediği bilgiyi gövde kısmında gönderirken "Content_Length" başlık satırında bunun uzunluğunu belirtir. Yanıt mesajındaki başlık satırlarından önemli olanları şunlardır:

Table 22.4 *Response Header Names*

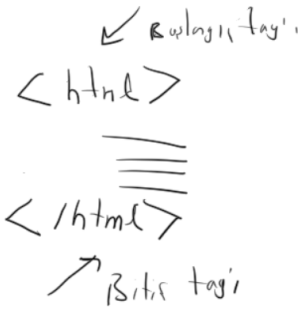
Header	*Description*
Date	Shows the current date
Upgrade	Specifies the preferred communication protocol
Server	Gives information about the server
Set-Cookie	The server asks the client to save a cookie
Content-Encoding	Specifies the encoding scheme
Content-Language	Specifies the language
Content-Length	Shows the length of the document
Content-Type	Specifies the media type
Location	To ask the client to send the request to another site
Accept-Ranges	The server will accept the requested byte-ranges
Last-modified	Gives the date and time of the last change

Buradaki başlık satırlarının anlamları HTTP dokümanlarından öğrenilebilir.

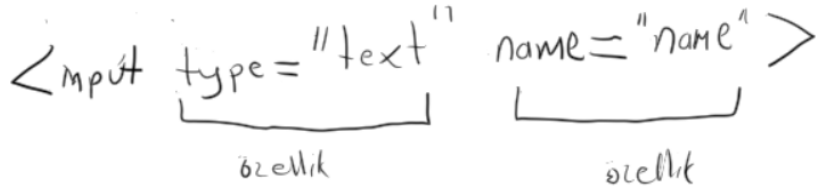
HTML ve CSS Hakkında Kısa Bilgi

Web sayfaları içerik olarak HTML (Hyper Text Markup Language) denilen dekleratif bir dille kodlanmaktadır. HTML web sayfasının görüntüsünü ifade etmek için kullanılmaktadır. Aslında son yıllarda pek çok GUI arayüzü bir biçimde deklaratif dillere doğru kaymaya başlamıştır. HTML oldukça detaylı bir dildir. HTML'in ilk çıktığından bu yana pek çok versiyonu oluşturulmuştur. Son versiyonu HTML5'tir. HTML yalnızca web sayfası içeriğini iletecek biçimde organize edilmiştir. Eskiden içeriğin yanı sıra HTML'de biçim de kodlanabiliyordu. Sonraları gittikçe biçim içerikten ayrılmıştır. Artık sweb sayfasının biçimi (genel görüntü detayı) CSS (Cascading Style Sheet) denilen dilde kodlanmaktadır. Yani artık HTML içerik bilgisini , CSS ise biçim bilgisini letmektedir.

HTML tıpkı XML gibi elemanlardan (tag'lardan) oluşan bir dildir. Bir eleman bir başlangıç tag'i ve bitiş tag'i ile bildirilir. Elemanlar iç içe yuvalanabilirler. Ancak dışarıda tek bir kök vardır. Başlangıç tag'ı "<" ve tag isminden oluşur. Bitiş tag'ı ise "</" ve tag isminden oluşmaktadır. Örneğin:



Başlangıç tag'ından sonra isim="değer" biçiminde özellik (attribute) bildirimi yapılabilir. Her tag'ın hangi özelliklere sahip olduğu HTMl standartlarında belirtilmiştir. Örneğin:



HTML tam olarak XML standartlarına uymamaktadır. Örneğin bazı elemanlar yalnızca başlangıç tag'ına sahip olabilirler. Ayrıca HTML kodları tarayıcılar tarafından çok katı bir sentaks kontrolüne sokulmazlar. Tarayıcılar mümkün olduğunca bozuk kodları yine de göstermeye çalışırlar. HTML ve CSS'in bir web sayfasını tam olarak betimleyebildiğini söyleyemeyiz. Yani farklı tarayıcılar aynı kodları birbirlerinden az çok farklı olarak görüntüleyebilirler. Tasarımda özellikle buna gayret edilmiştir. Tipik bir html "Merhaba Dünya" kodu şöyle oluşturulabilir:

```
<!DOCTYPE html>

<html>
   <head>
      <h1>Merhaba Dünya</h1>
   </head>

   <body>
      Bu bir denemedir
   </body>
</html>
```

Telnet Protokolü

Telnet Protokolü uzak bilgisayara bağlanıp onda login olarak iş yaptırmak için kullanılan temel bir uygulama protokolüdür. Yani Telnet bir çeşit uzaktan (remote) kontrol protokolüdür. Telnet kendi içerisinde bir şifreleme mekanizmasına sahip olmadığı için günümüz koşullarında güvensiz (unreliable/unsecure) kabul edilmektedir. (Gerçekten de telnet oturumu sırasında ağdaki başka bir host'un Wireshark gibi bir programla bile) giririlen kullanıcı adı ve parolayı elde etmesi çok kolaydır. Bu nedenle günümüzde Telnet'ten ziyade onun daha güvenli bir biçimi olan SSH (Secure Shell) protokolü tercih edilmektdir. Telnet ismi (Teletype Network) sözcüklerinden kısaltılarak uydurulmuştur. Telnet Internet'in en eski protokollerindendir. Daha Internet IP ailesine bile geçmeden önce Telnet protokolü kullanılıyordu.

Telnet server yazılımları pek çok işletim sisteminde yüklenmeye hazır olarak bulunmaktadır. Windows server sistemlerinde Telnet protokolü kullanıma hazırdır. Ancak Windows 10 sisteemlerinde Microsoft güvensiz olduğu gerekçesiyle server programı server listesinde çıkarmıştır. Ancak Telnet client programı durmaya devam etmektedir. Bu nedenle Windows 10 sistemlerinde deneme yapılacaksa bedava bir "Telnet Server" yazılımı ayrıca yüklenmelidir. Bunun için iyi bir seçenek "freesshd"programı olabilir.

Ubuntu türevi sistemlerde (örneğin Mint) telnet server yazılımı şu aşamalardan geçilerek kurulup konfigüre edilir:

1) `sudo apt-get install xinetd telnetd`

Komutu ile telnet server yazılımı kurulur

2) Daha sonra sudo nano "/etc/xinetd.d/telnet" dosyası (/etc/xinet.d dizininde olduğuna dikkat ediniz) aşağıdaki komutla yaratılır:

```
sudo nano /etc/xinetd.d/telnet
```

Bu dosyanın içerisine aşağıdaki içerik yerleştirilir:

```
# default: on
# description: The telnet server serves telnet sessions; it uses
# unencrypted username/password pairs for authentication.
service telnet
{
    disable = no
    flags = REUSE
    socket_type = stream
    wait = no
    user = root
    server = /usr/sbin/in.telnetd
    log_on_failure += USERID
}
```

3) Aşağıdaki komutla xinetd servisleri yeniden çalıştırılır:

```
sudo service xinetd restart
```

Mac OS X sistemlerinde Telnet server yazılımı hazır olarak gelmektedir. Ancak onun aktive edilmesi gerekir. Bunun için aşağıdaki komut kullanılabilir:

```
sudo launchctl load -w /System/Library/LaunchDaemons/telnet.plist
```

Telnet client program olarak çok seçenek vardır. Windows 10'da zaten client program program listesinde hazır bulnmaktadır. Bunun yüklenmesi "Windows özelliklerini aç veya kapat" menüsünden yüklenebilir. Linux sistemlerinde ve Mac OS X sistemlerinde komut satırından çalışan client program zaten default kurulum ile hazır bulunmaktadır. Bunun dışında bu sistemlerde üçüncü parti pek çok Telnet ve SSH client programlar vardır. Örneğin Windows'ta "Putty" ve "MobaXTerm" çok kullanılan seçeneklerdendir.

Telnet Protokolünün Esasları

Telnet protokolü default olarak 23 numaralı TCP portunu kullanır. Protokol oldukça yalındır. Client tarafı 23 numaralı TCP portundan server'a bağlanır. Sonra klavyeden basılan karakterleri tek tek ya da satır olarak server'a gönderir. Sunucu da uzak makinede istemci için login işlemi yapar. Oradaki programların stdout dosyasına yazdıkları şeyleri client'a yollar. Bu durumda client yazılım şöyle gerçekleştirilmelidir:

- Client sürekli server'dan gelen bilgileri okuyarak kendi terminal ekranına basmalıdır.
- Client yerel kullanıcının klavyede bastığı karakterleri tek tek ya da satırsal olarak server'a gönderir.

Basit olarak el alırsak Telnet iletişiminde client ile server şöyle işbirliği içerisinde çalışmaktadır: Client klavyeden basılan karakterleri server'a yollar. Server bu karakterleri sanki klavyeden girilmiş gibi uygular. Böylece server uzak makinede birtakım programların çalışmasına yol açar. Bu programların ekrana yazdıklarını da client'a yollar. Böylece adeta client'ın klavyesi sanki server'ın klavyesi gibi, client'ın ekranı da server'ın ekranıymış gibi bir durum oluşturulur. Aslında yalnızca Telnet protokolünde değil her türlü "Remote Kontrol" programlarında (örneğin "Teamviewer", "Windows Remote Desktop", "VNC" gibi) bu ana fikir kullanılmaktadır.

Telnet güvenilir bir protokol değildir. Eskiden veri hırsızlığı ve kırma (hacking) faaliyetleri çok rastlanan şeyler değildi. Bu nedenle Telnet protokolü güvenli kusuru olmasına karşın uzun süre kullanılmıştır. Ancak günümüz koşullarında bu protokol artık güvensiz olduğu gerekçesiyle artık çok az kullanılmaya başlanmıştır. Telnet yerine SSH (Secure Shell) protoklü tercih edilmektedir.

Telnet protolünde client'ın klavyeden girdiği bazı karakterler kontrol karakterleri olabilmektedir. Yani bazı karakterler client'in makinesindeki işletim sistemine özgü anlamlara sahip olabilir. Örneğin client Ctrl+C ile bir prosesi sonlandırmak isteyebilir. Ancak server için bu Ctrl+C karakteri bir anlam ,fade etmeyebilir. İşte

protokolde özel kontrol karakterleri ASCII tablosunun ikinci yarısındaki bazı karakter numaralarıyla temsil edilmiştir. Bu karakterlere NVT (Network Virtual Terminal) karakterleri denlmektedir. Bu karakterlerin listesi şöyledir:

Table 20.1 *Some NVT control characters*

Character	*Decimal*	*Binary*	*Meaning*
EOF	236	11101100	End of file
EOR	239	11101111	End of record
SE	240	11110000	Suboption end
NOP	241	11110001	No operation
DM	242	11110010	Data mark
BRK	243	11110011	Break
IP	244	11110100	Interrupt process
AO	245	11110101	Abort output
AYT	246	11110110	Are you there?
EC	247	11110111	Erase character
EL	248	11111000	Erase line
GA	249	11111001	Go ahead
SB	250	11111010	Suboption begin
WILL	251	11111011	Agreement to enable option
WONT	252	11111100	Refusal to enable option
DO	253	11111101	Approval to option request
DONT	254	11111110	Denial of option request
IAC	255	11111111	Interpret (the next character) as control

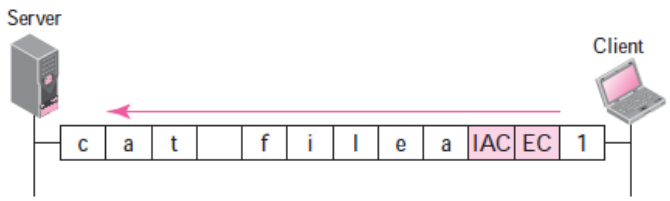
Bu kontrol karakterleri doğrudan değil başına IAC (255 numaralı karakter) karakteri getirilerek gönderilir. Örneğin client server'dan gönderdiği bir karakterin silinmesi istesin (backspace işlemi). Bunun için kullanılan kontrol karakteri EC (Erase Character) karakteridir. Ancak bunu tek başına değil başına IAC karakterini getirerek aşağıdaki gibi yollamalıdır:

```
IAC EC
```

Kontrol karakterleri yazının bir parçası olarak yazının içinde gönderilebilir. Ancak yukarıda da belirtildiği gibi bu durumda kontrol karakterinden önce IAC karakterinin kodlanması gerekir. Örneğin client server'a aşağıdaki karakterleri gönderiyor olsun (desimal biçimde gösteriyoruz):

```
97 108 105 255 247
```

Burada client 'a',' l',' i' karakterlerini server'a göndermiş, daha sonra son karakter olan 'i' karakterini back space etkisi yaratarak silmek istemiştir. Başka bir örnek aşağıdaki gibi verilebilir:



Telnet haberleşmesinde client ile server seçenekler üzerinde anlaşma (option negotiation) sağlamak üzere haberleşebilmektedir. Seçenek anlaşması yine bir IAC karakteri (255 numaralı ASCII karakter) ile başlatılır. Sonra bunu bir istek biçimi ve isteğin ne olduğunu anlatan (option) bir kod izler. Yani istek anlaşmasının kodlanması üç byte ile şöyle yapılmaktadır:

```
IAC <istek biçimi> <isteğin ne olduğu>
```

İstek anlaşması client'tan server'a ya da server'dan client'a gönderilebilir. İstek gönderildiğinde karşı taraf bunu kabul etmek zorunda değildir. Karşı yanıt olarak yine aşağıdaki 3 byte'lık kodlamayı gönderir:

```
IAC <istek yanıtı > <isteğin ne olduğu>
```

İstek biçimi şunlardan biri olabilir:

```
WILL   (251)
DO     (253)
```

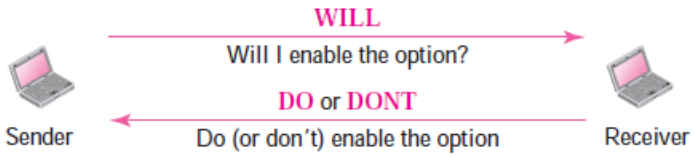
İstek yanıtı da şunlardan biri olabilir:

```
DO     (253)
DONT   (254)
WONT   (252)
```

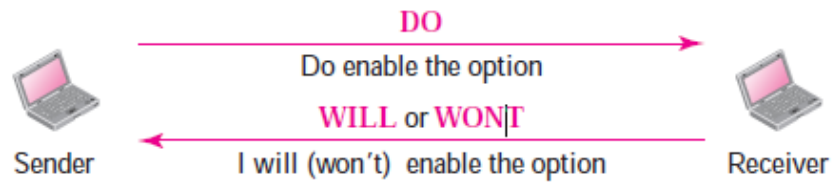
Seçenler de şunlardan biri olabilir:

Code	*Option*	*Meaning*
0	Binary	Interpret as 8-bit binary transmission
1	Echo	Echo the data received on one side to the other
3	Suppress go-ahead	Suppress go-ahead signals after data
5	Status	Request the status of TELNET
6	Timing mark	Define the timing marks
24	Terminal type	Set the terminal type
32	Terminal speed	Set the terminal speed
34	Line mode	Change to line mode

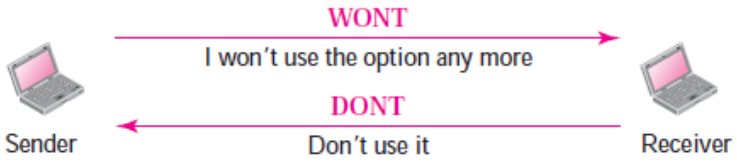
Bir taraf karşı tarafa "ben artık bu seçeneği kullanmak istiyorum, sen dersin?" biçiminde mesajı WILL ile gönderir. KArşı taraf da buna DO ya da DONT ile yanıt verir. DO "tamam kabul ediyorum", DONT "hayır eskisi gibi devam et, ben kabul etmiyorum" anlamına gelir.



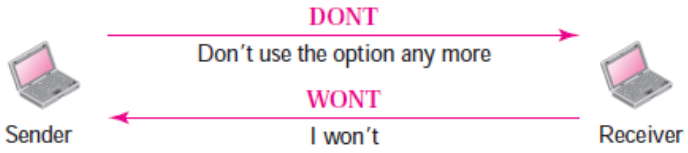
Bir araf karşı tarafa "senin artık bu seçeneği kullanmanı istiyorum, ne dersin?" biçiminde mesajı DO ile gönderir. Karşı taraf da buna WILL ya da WONT ile yanıt verir. WILL isteği kabul ettiği, WONT etmediği anlamına gelir.



Bir taraf karşı tarafa bir seçeneği artık kullanmayacağını bunu "disable" etmek istediğini WONT mesajı ile bildirir. Karşı taraf "disable" mesajlarına olumlu yanıt vermek zorundadır. O da olumlu yanıtı DONT ile verir.



Bir taraf karşı tarafa "senin artık bu özellği disable etmeni istoyum" mesajını DONT ilegönderir. Karşı tarafın bu mesajı onaylaması zorunludur. Bunu da WONT ile bildirir:



Görüldüğü gibi ENABLE mesajları karşı taraf tarafından kabul edilebilir ya da edilmeyebilir. Ancak disable mesajları kabul edilmek zorundadır.

Telnet client program yazılırken client tarafın IAC komutlarını alarak özellikle seçenek anlaşmalarına (option negotiation) düzgün yanıt vermesi gerekir.

Telnet client program yazarken server'ın gönderdiği ekran görüntüsü renklendirme ya da bazı imleç konumlandırması içerebilir. Genel olarak eskiden beri terminaller ya da terminal sürücüleri ANSI komut kümesi denilen bir komut kümesini desteklemektedir. Bu komut kümesine göre terminal sürücüsüne (yani stdout dosyasına) gönderilen karakterler 0x1B (ESC karakteri) ile başlatılıyorsa sonraki karakterler komut anlamına gelir. Default pek çok terminal sürücüsü bu ANSI komut kümesini desteklemektedir. Ancak Windows'un console uygulamaları default olarak bu komut kümesini desteklememektedir. Onu destekler hale getirmek için SetConsoleMode API fonksiyonun çağrılması gerekir.

FTP Protokolü

FTP (File Transfer Protocol) en yaygın kullanılan uygulama katmanı protokollerinden biridir. İsmnden de anlaşılabileceği gibi rrotokolün ana amacı client ve server sistemler arasında dosya transferini sağlamaktır. Gerçekten de UNIX/Linux tabanlı hosting firmalarının sağladığı alanlara dosya transferleri genellikle bu protokolle yapılmaktadır.

FTP çok eski bir protokoldür. Protokolün ilk versiyonu 1971 yılında RFC-114 ile çıkmıştır. Bu ilk versiyon Internet'in eski protokol ailesi olan NCP'de kullanılıyordu. Internet IP ailesine geçince bu protokolde 1980 yılında IP ailesine özgü biçimde değiştirildi. Bu yeni biçim RFC-765 dokümanıyla standardize edilmiştir. Daha sonra 1985 yılında RFC-959 ismiyle protokol üzerinde bazı düzeltmeler ve eklemeler de yapılmıştır. Daha sonra protokole diğer bazı eklemeler başka RFC numaralarıyla eski belirlemeler sabit kalmak koşuluyla yapılmıştır.

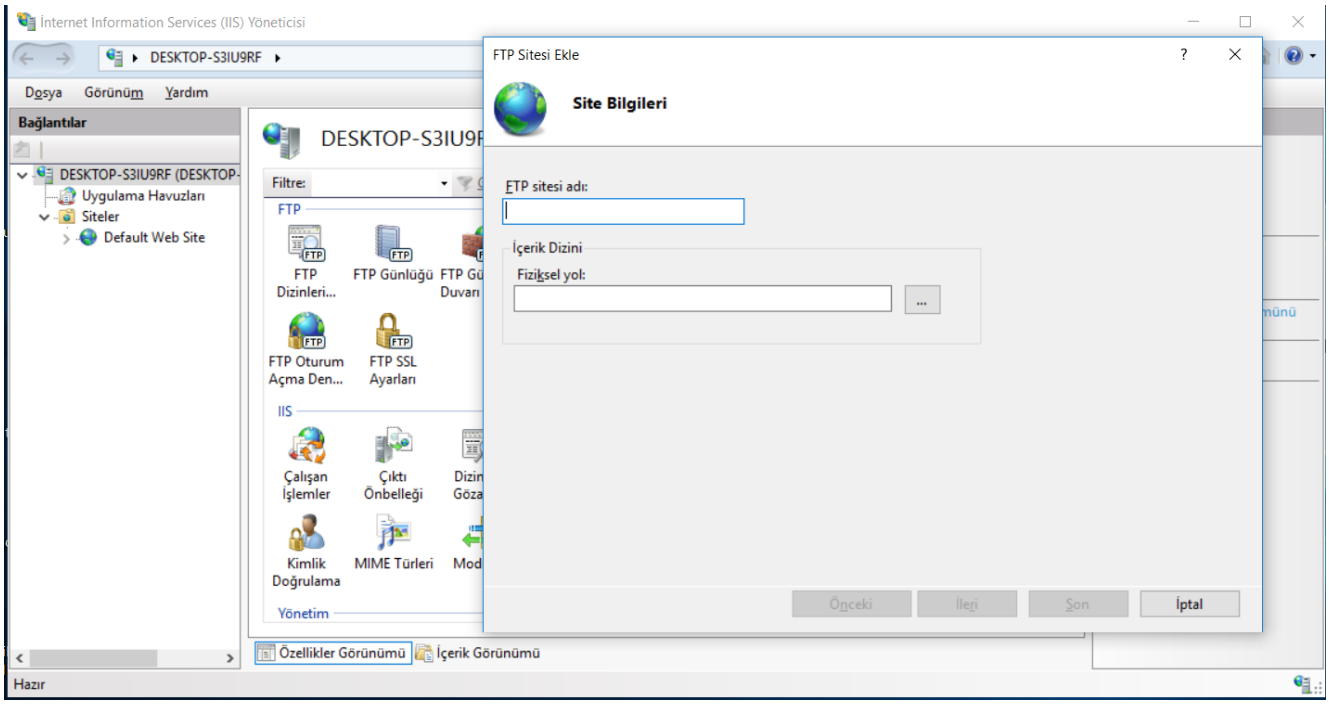
FTP için pek çok bedava ve açık kaynak kodlu client ve server yazılımları vardır. Client program olarak komut satırından "ftp" ile kullanılan klasik bir uygulama vardır. Bu komut satırı uygulaması pek çok Linux sisteminde default biçimde zaten yüklenmektedir. Windows sistemlerinde IIS içerisinde FTP sunucusu yüklendiğinde bu client program da yüklenmiş olur. Komut satırından çalışan ftp client programı oldukça basit bir kullanıma sahiptir. Program server'ın host adresi ya da ismi verilerek aşağıdaki gibi çalıştırılır:

```
ftp <server ismi ya da IP adresi>
```

Sonra bizden sırasıyla kullanıcı adı ve parola istenecek ve FTP sunucusuyla mantıksal bağlantı sağlanacaktır.

Komut satırından çalışan “ftp” client programının yanı sıra Windows sistemlerinde pek çok GUI FTP client programı vardır. Windows’ta en çok tercih edilen bedava GUI tabanlı FTP client programlar “Core FTP”, “Cute FTP” ve “File Zilla” isimli yazılımlardır. “File Zilla”nın Linux GUI versiyonu da vardır (ancak “Core FTP” ve “Cute FTP”nin Linux versiyonları yoktur.)

FTP server için de pek seçenek vardır. Windows’un kendi içerisinde IIS’te zaten Windows’un kendi FTP server’ı bulunmaktadır. Ancak Windows’ta FTP server’ı yükledikten sonra IIS içerisinde onun konfigüre edilmesi gerekmektedir. bunun için IIS’te “Siteler” kısmında bağlam menüsünden “FTP Sitesi Ekle” seçilir.



Linux sistemlerinde pek çok FTP sunucusu olmak birlikte vsftpd en fazla tercih edilendir. Onun kurulumu çok basittir. Ancak konfigüre edilmesi için bazı konfigürasyon dosyalarına bazı şeylerin yazılması gerekmektedir. Kurulum şöyle yapılabilir:

1) Aşağıdaki komu ile server program indirilerek kurulmalıdır:

```
sudo apt-get install vsftpd
```

2) xinetd isimli servis yönetici program kurulu değilse aşağıdaki gibi kurulmalıdır:

```
sudo apt-get install xinetd
```

3) FTP server’ın konfigürasyonu /etc/vsftpd.conf dosyasından ayarlanabilir. Örneğin bu dosyada anonim bağlantıyı açmak için “anonymous_enable” özelliği “YES” yapılmalıdır.

4) Nihayet ftp server programı aşağıdaki komutla yeniden başlatılmalıdır :

```
sudo service xinetd restart
```

Bunun yerine başlatma işlemi xinetd yerine doğrudan da yapılabilir:

```
sudo /etc/init.d/vsftpd restart
```

FTP Prokolünün Temelleri

FTP protokolü Telnet gibi tek port ve bağlantıyla haberleşmemektedir. Bu protokol iki port ve bağlantı kullanmaktadır. Bağlantılardan birine "kontrol (control)" diğerine ise "veri (data)" bağlantısı denilmektedir. Server tarafında kontrol defaulr kontrol portu için 21, veri portu için 20 kullanılmaktadır. Kontrol bağlantısı Telnet bağlantısına benzemektedir. Bu bağlantı client'tan server'a ve server'dan client'a komut gönderip almak için kullanılır. Veri bağlantısı ise yine client'tan serverá ve server'dan client'a içerik göndermek için (örneğin dosya içeriği ya da dizin içeriği gibi) kullanılır.

Protokolde server 21h portunu kontrol bağlantısı için pasif olarak (yani karşı taraf bağlansın diye, dinleme soketi olarak) açar. Client da dolayısıyla aktif biçimde herhangi bir porttan (ephemeral port) server'ın 21 portuna bağlanır. Tüm komutlar "command-response" biçiminde işlenmektedir. Yani tıpkı Telnet'te olduğu gibi bir taraf bir tarafa bir komut gönderdiğinde diğer taraf ona bir yanı vermektedir. Genel olarak server komutun durumu (status) hakkında (yani işlemin başarısı hakkında) bildirimlerde bulunur.

Client server'a ilk kez bağlandığında server durumu onaylamak için client'a kontrol bağlantısından "220 (Service Ready)" yanıtını gönderir. Bundan client server'a yine kontrol portundan "USER <kullanıcı ismi>" ve sonra da "PASS <parola>" komutlarını göndererek mantıksal bağlantıyı sağlar.

Mantıksal bağlantı sağlandıktan sonra transfer için veri bağlantısının yapılması gerekir. Pekiyi veri bağlantısı kaç numaralı porttan yapılmaktadır? Dahası bunun için kim kime bağlanmaktadır? İşte FTP veri bağlantısı bu noktada "aktif" ve "passif" ismiyle ikiye ayrılmaktadır. Default mod "aktif" moddur. Aktif modda veri bağlantısı için client soket açar ve dinleme durumuna geçer, server client'a bağlanır. Fakat "pasif" bağlantıda tam tersi server soket açıp dinleme durumuna geçer ve client ona bağlanır.

Aktif veri bağlantısında client "PORT <port numarası>" komutunu server'a gönderir. Server da bu komutta belirtilen porta bağlanır. Yani server'ın veri aktarımı için client'ın hangi portuna bağlanacağını client server'a bildirmektedir. Pasif bağlantıda ise client server'a "PASV" komutunu gönderir. Bu durumda tam tersi server client'a "227 Entering PAssive Mode (x, y, z, k, m, n)" komutunu gönderecektir. Bu komutta "x, y, z, k" server'ın IP adresi, "m, n" ise port numarasının sırasıyla yüksek ve düşün anlamlı byte değerleridir. Bundan sonra client bu belirlenen porttan server'a bağlanır. "PASV" ve "PORT" komutları tüm transfer komutları için yenieden gönderilmelidir. Örneğin client önce "PASV" komutuyla pasif moda geçip sonra "LIST" komutuyla dizin listesini istemiş olsun. Bundan sonra client yeniden "LIST" ya da "RETR" gibi bir komutu yeniden uygulayacaksa yeniden "PASV" komutuyla pasif moda geçmelidir. Her tansfer sonrasında server veri bağlantı soketini kapatmaktadır.

Pekiyi 20 numaralı portun anlamı nedir? Aktif modda server client'ın istediği porttan ona bağlanırken bind işlemi yaparak bağlanmada kullanılan yerel portunu 20 olarak ayarlamaktadır. Yani server'ın aktif bağlantıda veri için kullanacağı yerel port numarası 20'dir. Pasif modda client'ın server'a veri bağlantısı için hangi porttan bağlandığının bir önemi yoktur.

FTP protokolündeki komutlar genellikle birkaç grup altında toplanarak ele alınmaktadır. Erişim komutları (access commands) bağlantı için kullanılan komutlardır:

Access commands

Command	*Argument(s)*	*Description*
USER	User id	User information
PASS	User password	Password
ACCT	Account to be charged	Account information
REIN		Reinitialize
QUIT		Log out of the system
ABOR		Abort the previous command

Dosya komutları klasik dosya işlemlerini yapan komutlardır:

File management commands

Command	*Argument(s)*	*Description*
CWD	Directory name	Change to another directory
CDUP		Change to parent directory
DELE	File name	Delete a file
LIST	Directory name	List subdirectories or files
NLIST	Directory name	List subdirectories or files without attributes
MKD	Directory name	Create a new directory
PWD		Display name of current directory
RMD	Directory name	Delete a directory
RNFR	File name (old)	Identify a file to be renamed
RNTO	File name (new)	Rename the file
SMNT	File system name	Mount a file system

Veri farmatlama (data formatting) komutları transfer edilecek verinin formatı hakkında belirlemede bulunmaktadır.

Data formatting commands

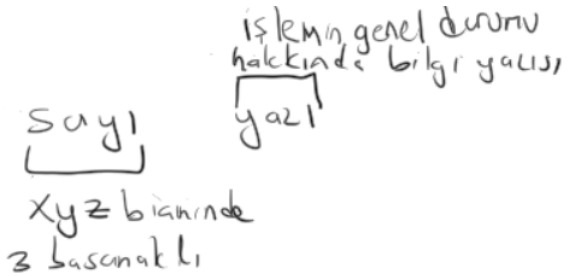
Command	*Argument(s)*	*Description*
TYPE	A (ASCII), E (EBCDIC), I (Image), N (Nonprint), or T (TELNET)	Define file type
STRU	F (File), R (Record), or P (Page)	Define organization of data
MODE	S (Stream), B (Block), or C (Compressed)	Define transmission mode

Bunların yanı sıra diğer bazı komutlar da vardır:

Miscellaneous commands

Command	*Argument(s)*	*Description*
HELP		Ask information about the server
NOOP		Check if server is alive
SITE	Commands	Specify the site-specific commands
SYST		Ask about operating system used by the server

Client server'a kontrol bağlantısından bir komut gönderdiğinde serve ona bir (ya da bazen birden fazla) yanıt gönderir. Komutlar ve yanıtlar sonu CR/LF ile biten satırlar biçimindedir. (Tıpkı Telnet'te olduğu gibi). Bit yanıt iki kısımdan oluşmaktadır: Üç digitlik bir sayı ve bir de yazı.



Yanıtın yazı kısmı standart değildir. Dolayısıyla FTP server buradaki yanıt yazısını istediği gibi oluşturabilir. 3 basamaklı durum kodunun (status code) birinci basamağı 1, 2, 3, 4, 5 değerlerinden biri olabilir. Bu basamaklar şu anlamlara gelmektedir:

1: İşlem başlatılmıştır. Ancak bittiğinde yeniden bildirimde bulunulacaktır.
2: İşlem bitirilmiştir. Artık server başka bir komutu kabul edip işleyebilir.
3: Komut kabul edilmiştir. Ancak komut için başka bilgilere de gereksinim vardır.
4: İşlem yapılamamaktadır. Ancak bu geçici olabilir. Yani şimdi yapılamıyor olması bir süre sonra yapılamayacağı anlamına gelmez.
5: Komut kabul edilmemiştir. Yeniden gönderilirse yine kabul edilmeyecektir.

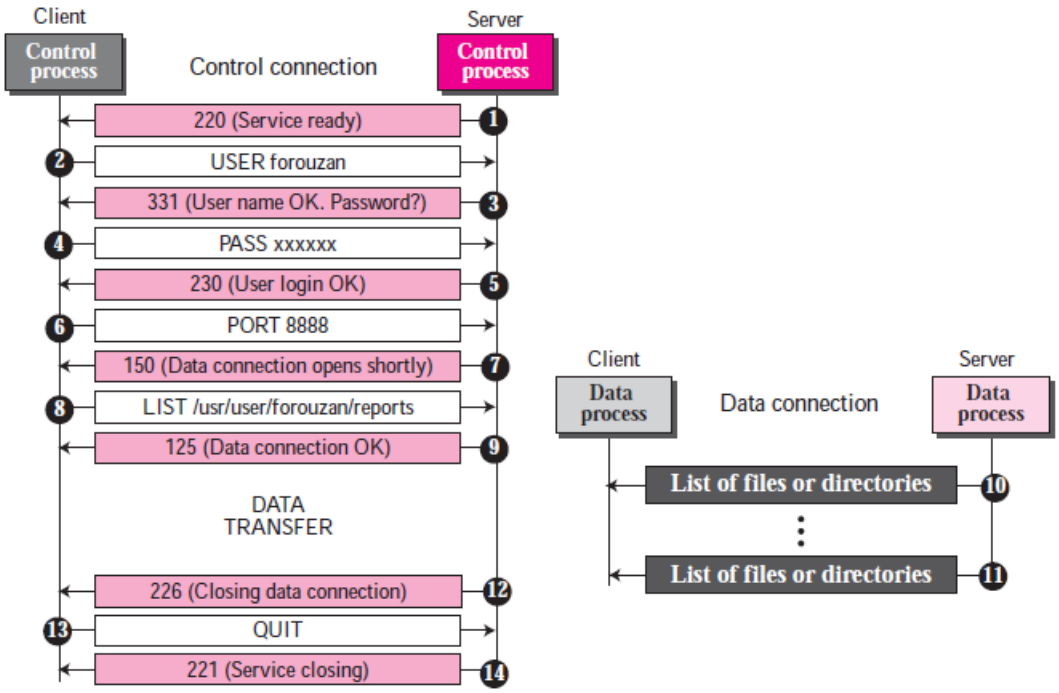
İkinci basamak yine komutun ve işlemin durumu hakkında bilgi verir. İkinci basamak şunlardan biri olabilir:
0: Sorunun nedeni sentaksla ilgilidir.
1: Yanıt bilgi içermektedir.
2: Yanıt bağlantı durumuyla ilgilidir.
3: Sorun yetki derecesi ile ilgilidir.
4: Sorun ya da komut belirsizdir (unspecified)
5: Sorun ya da komut dosya sistemi ile ilgilidir.

Üçüncü basamak ek bilgi içermektedir. FTP'de bazı çok kullanılan yanıtlar ve açıklama yazıları aşağıdaki tabloda verilmiştir:

Responses

Code	*Description*
Positive Preliminary Reply	
120	Service will be ready shortly
125	Data connection open; data transfer will start shortly
150	File status is OK; data connection will be open shortly
Positive Completion Reply	
200	Command OK
211	System status or help reply
212	Directory status
213	File status
214	Help message
215	Naming the system type (operating system)
220	Service ready
221	Service closing
225	Data connection open
226	Closing data connection
227	Entering passive mode; server sends its IP address and port number
230	User login OK
250	Request file action OK
Positive Intermediate Reply	
331	User name OK; password is needed
332	Need account for logging
350	The file action is pending; more information needed

Örneğin FTP protokolünde bir client taraf server'a bağlanıp ondan dizin içeriğini LIST komutuyla isteyecek olsun. Haberleşme akışı şöyle olacaktır:



Burada önce client 21 numaralı porttan server'a bağlanır. Server ona hemen kontrol bağlantısından "220 Service ready" yanıtını gönderir. Bu yanıt her şeyin normal gittiğini belirtmektedir. Bundan sonra client "USER <kullanıcı ismi>" komutunu server'a gönderir. Server da buna "331 User Name OK, Password?" yanıtı ile karşılık verir. Bu yanıttan server'ın kullanıcı ismini kabul ettiğini ve client'tan artık parola istediğini anlıyoruz. Bu durumda client server'a bu kez "PASS <parola>" komutu ile parola gönderir. Bunu da alan server eğer bu bilgiler doğruysa client'a "230 User Login OK" yanıtını gönderir. Bundan sonra aktif bağlantı için client server'a "PORT <por numarası>" komutu ile port numarasını iletir. Server bunu alır ve ilgili porta ilişkin soketi pasif olarak (yani dinlemek amacıyla) açar. Ve client'a bu kez "150 Data connection opens shortly" yanıtını gönderir. Bundan sonra client server'a yine kontrol bağlantısından "LIST <dizin yol ifadesi>" komutunu gönderir. Artık client server'a ilgili veri portundan bağlanır. Server da client bağlantısını kabul eder ve client'a "125 Data connection OK" yanıtını gönderir. Bundan sonra server 'da dizin içeriğini veri bağlantısı yoluyla satır satır yazı biçiminde client'a yollayacaktır. İşlem bittiğinde server açmış olduğu dinleme soketini kapatır ve client'a "226 Closing data connection" nyanıtını yollar. Client işlemini bitirmek için server'a bu kez "QUIT" mesajını yollar. Server clien'tın bağlantıyı koparmak istediğini anlar. Ona "221 Service closing" yanıtını gönderir ve client'ın komut bağlantı soketini kapatır.

TFTP (Trivial FTP) Protokolü

TFTP protokolü adeta FTP protokolünün basit bir bir versiyonu gibidir. TFTP protokolü UDP kullanmaktadır. Server'ın default port numarası 69'dur. Windows sistemlerinde TFTP server olarak pek çok program bulunmaktadır. Örneğin TFTPD32 isimli program bu amaçla kullanılabilir. Windows kendi içerisinde zaten hazır bir TFTP client program da bulundurmaktadır. ("Windows Özelliklerini Aç ve Kapat"tan bunu yükleyebiliriz.) TFTPD32 aynı zamanda zaten bir client program görevini de yapmaktadır. Linux sistemlerinde TFTP server kurulumu şu adımlardan geçilerek yapılabilir:

1) Aşağıdaki komut uygulanarak server program kurulur.

```
sudo apt-get install tftpd
```

2) /etc/xinetd.d/tftp dosyası yaratılarak aşağıdaki içerik yazılır:

```
service tftp
{
    protocol        = udp
```

```
    port            = 69
    socket_type     = dgram
    wait            = yes
    user            = nobody
    server          = /usr/sbin/in.tftpd
    server_args     = /tftpboot
    disable         = no
}
```

3) Aşağıdaki komutlar uygulanır:

```
sudo mkdir /tftpboot
sudo chmod -R 777 /tftpboot
sudo chown -R nobody /tftpboot
```

4) xinet de aşağıdaki komutla yeniden çalıştırılır:

```
sudo service xinetd restart
```

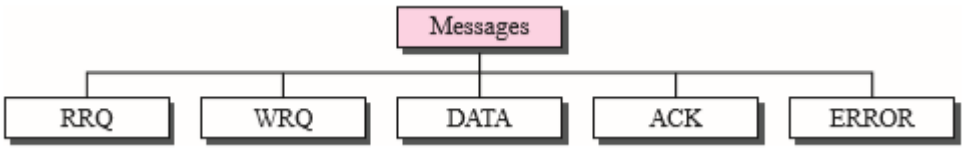
Linux sistemlerinde TFTP client program da aşağıdaki komutla kurulabilmektedir:

```
sudo apt-get install tftp
```

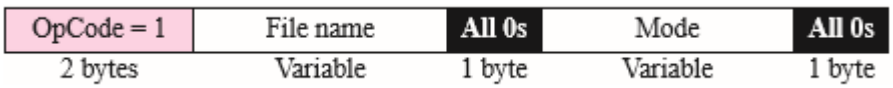
TFTP Protokolünün Betimlemesi

TFTP güvenlik mekanizması olmayan (kullanıcı adı ve parola gibi) oldukça basit bir dosya transfer protokolüdür. IP ailesinin diğer uygulama katmanı protokollerinden farklı olarak TFTP UDP kullanmaktadır. Tipik olarak haberleşme client'ın 69 numaralı UDP portundan server'a paket göndermesiyle başlatılır. Gönderilen paketlerin başında bir başlık kısmı vardır. Bu başlık kısmında client'ın server'dan ne istediği belirtilir.

TFTP'de client'tan server'a ve server'dan client'a gönderilebilecek mesajlar şunlardır:



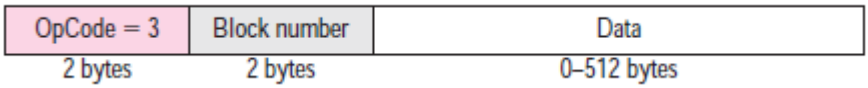
Yukarıda 5 çeşit mesaj olduğunu gördük. Örneğin RRQ mesajı client'tan server'a gönderilen bir mesajdır. Mesajın genel formatı şöyledir:



Görüldüğü gibi başlığın ilk iki byte'ına WORD düzeyde binary olarak 1 değeri yerleştirilmelidir. Bu 1 değeri client'ın server'dan dosya istediğini belirtir. Daha sonra bu iki byte'ı dosya ismi izler. Dosya ismi değişken uzunlukta byte'lardan oluşmaktadır ve yazısal bir alandır. Bu alanın bittiği sonra gelen 0 byte'ından anlaşılmaktadır. Bu 0 byte'ını da transfer modu izler. Eğer transfer edilecek dosya metin tabalıysa buraya "netascii" yazısı, binary tabanlıysa buraya "octet" yazısı gelmektedir. Başlığın sonunda bir tane yine 0 byte'ı bulundurulur.

RRQ mesajına karşılık server eğer söz konusu dosya mevcutsa ve gönderilebilecek durumdaysa buna DATA mesajlarıyla yanıt verir. DATA mesajları her biri (son kısım haricinde) 512 byte data içeren paketler biçimindedir. Yani server dosyayı 512'lik parçalara ayırıp client'ın kullandığı UDP portundan paket paket

(protokol dokümanlarında blok terimi kullanılmaktadır) client'a yollamaktadır. DATA mesajının genel biçimi şöyledir:

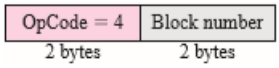


Mesajın ilk iki byte'ı OpCode olarak 3 değerini içerir. Bunu Blok numarasını belirten 2 byte'lık bir alan izler. Sonra da 512 byte'lık dosya parçası mesajda bulunmaktadır.

WRQ möesajı da client'tan server'a gönderilir ve formatı şöyledir:

Görüldüğü gibi mesajın RRQ mesjından tek farkı OpCode alanın 2 olmasıdır.

TFTP protokolünde client ile server arasında akış kontrolü için çift taraflı bir zaman aşımı (timeout) yöntemi kullanılmaktadır. Server bir DATA bloğunu gönderdikten sonra belli bir zaman aşımı süresi kadar bekler. Eğer client bu süre içerisinde ACK mesajını göndermezse server paketin client'a ulaşamadığını düşünür. Aynı DATA bloğunu yeniden gönderir. ACK mesajı 4 byte uzunluktadır ve genel biçimi şöyledir:



Şimdi TFTP protokülündeki haberleşmeyi daha ayrıntılı olarak inceleyelim. Bunun için RRQ mesajını temel alalım.
Haberleşme sırasında client kendi yerel host'undaki boş bir port numarasını (ephemaral port) kullanır. Ancak RRQ komutunu içeren UDP paketini server'ın bulunduğu host'un 69 numaralı portuna yollar. Server client'ın yerel port numarasını elde ederek kendi host'undaki boş bir port numarasını (ephemeral port) kullanarak DATA yanıtını yine UDP paketi biçiminde o porta yollamaktadır. Artık client paketi alındığına yönelik ACK mesajlarını server'ın bu portuna yollar.

Bu işlemi adım adım şöyle örneklendirebiliriz:

1) Client boş bir port numarası (ephemeral port) ile server'ın 69 numaralı portuna RRQ komutunu gönderir. Bu sıradaki client'ın boş port numarasının 5050 olduğunu varsayalım.

2) Server DATA yanıtını client'ın 5050 numaralı portuna yollar. Ancak bu işlemi yaparken kendisi de yerel boş bir portu kullanır. Bu portun da 6060 olduğunu varsalım.

3) Client 5050 numaralı porttan DATA yanıtını alır. Ancak henüz 1 numaralı bloğu almıştır. Diğer bloğu alabilmek için server'ın 6060 numaralı portuna ACK komutunu gönderir. Server'da diğer bloğu yine 5050 numaralı porta gönderir. Client yeniden 6060 portuna ACK göndererek işlemleri böyle devam ettirir.

Yukarıda haberleşme sırasında iki taraflı zaman aşımı kontrolünün yapıldığını belirtmiştik. İşte ikinci taraf client tarafından yapılan zaman aşımı kontrolünü oluşturmaktadır. Server DATA bloğunu gönderdikten sonra belli bir süre client'ın ACK göndermesini bekler. Eğer client ACK mesajını göndermezse server client'ın bloğu almadığından şüphelenip bunu yeniden göndermektedir. (Tabii toplamda bunun da bir sayısı vardır). Pekiyi client'ın gönderdiği ACK mesajı yolda kaybolursa ne olacaktır?Biindiği gibi UDP güvenilir (reliable) bir protokol değildir. Yani bir taraf mesajı gönderir fakat karşı tarafın gerçekten alıp almadığını bilemez. İşte TFTP protokolünde client da ACK mesajını gönderdikten sonra server'dan sonraki DATA bloğunu beklemektedir.

Eğer server sonraki data bloğunu göndermezse bu kez client yeniden ACK gönderir. Ancak bu haberleşmede bazı prürüzler de olabilmektedir.

POP3 (Post Office2) Protokolü

POP3 e-postaları server'dan almak için kullanılan bir protokoldür. POP3 protoklünden önce POP2 ve POP1 vardı. POP1 protokolü 1984 yılında geliştirildi. Bunu 1985 yılında POP2 izledi. POP3 1988 yılında geliştirilmiştir. (RFC-1081). Daha sonra 1996 yılında son kez düzeltilmiştir (RFC-1939). E-Postaları almak için çok kullanılan diğer bir protokol de IMAP (Internet Message Access Protocol) isimli protokoldür. IMAP POP3 protokolünün oldukça gelişmiş bir biçimidir. Bugün her iki protokol de e-posta almak için yoğun olarak kullanılmaktadır.

E-Posta gönderme işlemi alma işleminden oldukça farklı olduğu için ayrı bir protokol olarak tasarlanmıştır. Bu protokole SMTP (Simple Mail Transfer Protocol) denilmektedir. Aslında SMTP e-posta alma konusunda da kullanılabilen bir protokoldür. Ancak bugün uygulamada e-postalar SMTP ile gönderilip POP3 ya da IMAP ile alınmaktadır.

POP3 protokolünde E-Postaları client'a göndermek için POP3 Server yazılımları kullanılmaktadır. Yani POP3 diğer protokollerde olduğu gibi hem server hem de client programları yazılabilen bir protokoldür. POP3 server programı her kullanıcının e-postalarını bir posta kutusunda (maildrop) tutar. POP3 client program da server'a bağlanarak e-postaların listesini ya da belli e-postaların içeriklerini oradan indirebilir. Client isterse server'daki posta kutusundaki e-postaları da silebilmektedir. Servis sağlayıcıları genellikle e-posta göndermek (outgoing mail server) ve almak (incoming mail server) için ayrı server adresleri temin etmektedir. Örneğin Derneğimizin hizmet aldığı hosting şirketi e-posta gönderimi ve alımı için "mail.csystem.org" isimli host'u temin etmiştir.

POP3 protokolü default olarak 110 numaralı TCP portunu kullanmaktadır. İletişim POP3 server'ın 110 numaralı dinlemesiyle başlar. Client bu porttan server'a bağlanır. Bağlanmadan sonra server "+OK Hello there." biçiminde bir mesaj yollar. (Server'ın yanıt metinleri standart değildir. Ancak istek olumsulysa "+OK" karakterleri ile olumsuzsa "-ERR" karakterleri ile başlayan bir mesaj göndermektedir.)

Client'ın server'a gönderdiği mesajlar ve server'ın client'a gönderdiği yanıt mesajları diğer protokollerde olduğu gibi metin tabanlıdır. Komutlar 3 ya da 4 karakter uzunluğundadır ve büyük harf küçük harf duyarlılığı yoktur. Her komutun sonu yine CR/LF çifti ile bitmek zorundadır. Server'ın client'a gönderdiği yanıtlar genellikle tek satırdır. Onlar da CR/LF ile biter. Ancak birden fazla satırlı yanıtların her bir satırı CR/LF ile bitmektedir. Ancak mesajın sonunun anlaşılması için yalnızca '.' karakteri ve CR/LF'den oluşan bir satır gönderilir. Eğer tesadüfen herhangi bir satır '.' karakteri içeriyorsa bu durumda CR/LF '.' CR/LF karakterleri gönderilmektedir.

Client'ın komutları şunlardan oluşmaktadır:

1) USER <kullanıcı ismi>: Burada kullanıcı ismi herhangi bir isim olabilir. Ancak genellikle e-posta adresi bu amaçla kullanılmaktadır.

2) PASS <parola>: Kullanıcı parolasını belirtir. Maalesef buradaki parola şifrelenmemiştir. IMAP protokolü bu bakımdan daha güvenlidir.

3) LIST: Server'daki kullanıcının posta kutusunda bulunan e-postaları numara ve byte uzunluğu olarak birden çok satır biçiminde elde etmek için kullanılır.

4) RETR <e-posta numarası>: Belirtilen numaralı e-postanın içeriğini elde etmek için kullanılır.

5) DELE <e-posta numarası>: Belirtilen numaralı e-postayı kullanıcının posta kutusundan siler.

6) STAT: Toplam kaç tane ve toplam kaç byte'lık e-posta olduğu bilgisini verir.

7) NOOP: Bu komut bir şey yapmaz. Yalnızca başarılı ("+OK") bir yanıtla geri döner.

8) RSET: DELE komutuyla silinmiş olan mesajları geri kurtarır.

9) TOP <mesaj numarası> <satır sayısı>: Bu komut başlık kısımları hariç belli numaralı bir mesajın ilk n satırını elde etmek için kullanılır.

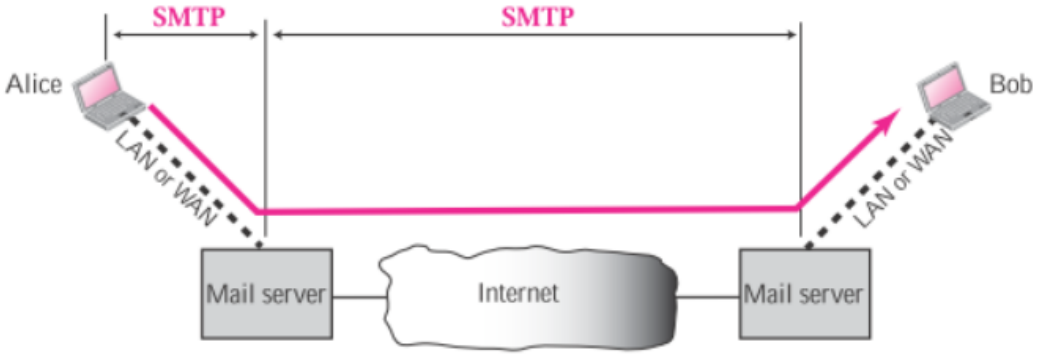
10) QUIT: bağlantıyı sonlandırır ve silinmek üzere işaretlenmiş olan e-postaları tamamen siler.

DELE komutu aslında gerçek bir silme yapmayıp yalnızca işaretleme yapmaktadır. Kullanıcı sildiğine pişman olursa RSET komutu ile silinmek üzere işaretlenmiş olan tüm e-postaları geri kurtarabilmektedir. QUIT işlemi silinmek üzere işaretlenmiş e-postaları gerçekten silmektedir. Ancak POP3 bağlantısında server client'tan komut gelmezse belli bir zaman aşımından sonra soketi kapatmaktadır. Bu durum QUIT uygulandığı anlamına gelmez.

SMTP (Simple Message Transfer Protocol) Protokolü

SMPTP ilk kez 1982 yılında RFC-821 ile standardize edildi. Daha sonra 2008'de RFC-5321 ile güncellendi. Bu protokol e-posta mesajlarının gönderilmesi için kullanılmaktadır. Bilindiği gibi e-posta mesajlarının alınması için POP3 ve IMAP protokolleri kullanılmaktadır. Ancak gönderme işlemi ağırlıklı biçimde SMTP ile yapılmaktadır.

SMTP protokolü diğerlerinde olduğu gibi TCP üzerine otrutulmuştur. SMTP client program 25 ya da 587 numaralı porttan server'a bağlanır. Göndereceği e-postayı aşağıda ele alınacağı üzere server'a iletir. SMTP server program hedef e-posta adresine ilişkin SMTP server programa e-postayı iletir. Bu e-posta hedef server'daki posta kutusuna yerleştirilir. Buradan da POP3 ya da IMAP protokolü yardımıyla client tarafından okunur.



Örneğin biz Derneğimizin bir e-posta adresinden info@kanattim.com adresine bir e-posta göndermek isteyelim. Bunun Bunun için öncelikle servis sağlayıcımızın e-posta sunucusunun (outgoing mail server) adresini biliyor olmamız gerekir. Örneğib "C ve Sistem Programcıları Derneği" "Dreamhost" isimli servis sağlayıcıyı kullanmaktadır. Bunun için SMTP server adresi "mail.csystem.org" biçimindedir. SMTP server'lar herkesten gelen istekleri karşılamak istememektedir. Pek çok servis sağlayıcı ancak kendi hizmet verdiği kullanıcıların e-posta göndermelerine izin vermektedir. Yine pek çok SMTP server başkasının adına e-posta gönderilmesin diye e-postayı gönderirken de kullanıcı adı ve parola istemektedir. Biz bu örnekte mail.csystem.org adresindeki server'a e-postamızı ilettikten sonra bu sunucu karşı atarfın (yani kanattim.com) e-posta sunucusuna bunu yine SMTP ile iletir. İşte karşı tarafın sunucusu e-postayı alarak o kullanıcıya ilişkin posta kutusuna e-postayı yerleştirmektedir.

SMTP client programla server arasındaki haberleşme sırasıyla şu aşamalardan geçilerek yapılır:

1) Client ptogram server'a 25 numaralı ya da 587 numaralı porttan (server port numarası daha farklı da olabilmektedir) server'a bağlanır. Buna karşılık server 220 kod numaralı başarılı bağlanma mesajını gönderir. (Client ve server yine metin tabanlı sonu CR/LF ile biten yazılarla komut ve yanıtlaarı göndermektedir.)

```
C:\Users\csystem\Desktop>telnet mail.csystem.org 587

220 homiemail-a39.g.dreamhost.com ESMTP
```

2) Client HELO komutunu server'a gönderir. Komutun genel biçimi şöyledir:

```
HELO <kaynak domain ismi>
```

Server bu komuta 250 numaralı olumlu mesajla yanıt verir.

```
220 homiemail-a39.g.dreamhost.com ESMTP
HELO csystem.org
250 homiemail-a39.g.dreamhost.com
```

3) Client hangi e-posta adresinden e-postanın gönderileceğini MAIL FROM komutuyla server'a bildirir. MAIL FROM komutunun genel biçimi şöyledir:

```
MAIL FROM: <kaynak e-posta adresi>
```

```
220 homiemail-a39.g.dreamhost.com ESMTP
HELO csystem.org
250 homiemail-a39.g.dreamhost.com
MAIL FROM: dropbox@csystem.org
250 2.1.0 Ok
```

4)Eğer SMTP server bir kimlik doğrulaması (authentication) istiyorsa bu durumda client programın AUTH LOGIN komutu ile bunu vermesi gerekir. Bu mesaj uygulandığında server base64 kodlamasıyla bize sırasıyla kullanıcı ismini ve parolayı sormaktadır. Biz ona kullanıcı ismini ve parolayı base64 kodlamasıyla vermeliyiz. (Bunun için https://www.base64decode.org gibi Online base64 kodlama çeviricilerini kullanabilirsiniz)

```
220 homiemail-a39.g.dreamhost.com ESMTP
HELO csystem.org
250 homiemail-a39.g.dreamhost.com
MAIL FROM: dropbox@csystem.org
250 2.1.0 Ok
AUTH LOGIN
334 VXNlcm5hbWU6
ZHJvcGJveEBjc3lzdGVtLm9yZw==
334 UGFzc3dvcmQ6
Y3N5c3RlbS0xOTkz
235 2.7.0 Authentication successful
```

5) Bu işlemden sonra client server'a RCPT To komutunu göndererek e-postanın hangi adrese gönderileceğini belirtir. RCPT TO komutunun genel biçimi şöyledir:

RCPT TO: <hedef e-posta adresi>

6) Client DATA komutuyla e-posta mesajının içeriğini server'a satır satır gönderir. Gönderme işlemi sırasında her satır CR/LF karakteri ile sonlandırılır. en son satırda da e-postanın sonunu belirtmek için bir tane '.' karakteri kullanılır. DATA komutundan sonra server bir onaylama yanıtı gönderir. Bunun üzerine client aşağıdaki formatta e-postanın içeriğini göndermelidir.

7) E-posta gönderildiğinde haberleşme client tarafın QUIT komutu uygulamasıyla sonlandırılırç Server buna 221 numaralı yanıtla karşılık verir ve soket kaapatılır.

Aşağıda bir e-postanın telnet ile gönderilmesine örnek verilmiştir:

```
220 homiemail-a39.g.dreamhost.com ESMTP
HELO csystem.org
250 homiemail-a39.g.dreamhost.com
MAIL FROM: dropbox@csystem.org
250 2.1.0 Ok
AUTH LOGIN
334 VXNlcm5hbWU6
ZHJvcGJveEBjc3lzdGVtLm9yZw==
334 UGFzc3dvcmQ6
Y3N5c3RlbS0xOTkz
235 2.7.0 Authentication successful
RCPT TO: info@kanattim.com
250 2.1.5 Ok
DATA
354 End data with <CR><LF>.<CR><LF>
From: Kaan Aslan
To: Ali Serçe
Subject: Deneme e-postası

Bu bir denemedir.
.
250 2.0.0 Ok: queued as B6CAC150074
QUIT
221 2.0.0 Bye


Ana bilgisayara bağlantı kayboldu.
```

SMTP protokolüne ilişkin diğer komutlar aşağıda verilmiştir:

Table 23.1 *Commands*

Keyword	*Argument(s)*	*Keyword*	*Argument(s)*
HELO	Sender's host name	NOOP	
MAIL FROM	Sender of the message	TURN	
RCPT TO	Intended recipient	EXPN	Mailing list
DATA	Body of the mail	HELP	Command name
QUIT		SEND FROM	Intended recipient
RSET		SMOL FROM	Intended recipient
VRFY	Name of recipient	SMAL FROM	Intended recipient

Yukarıda ele alınmayan birkaç komuta dikkat ediniz. RSET işlemi tamamen iptal etmek için kullanılmaktadır. VRFY komutu ilgili e-posta kullanıcısının hedef sistemde olup olmadığını sorgulamak için kullanılır. NOOP boş bir komuttur. Karşı taraf buna her zaman olumlu bir yanıt gönderir. Bağlantıyı sorgulamak için kullanılabilmektedir. TURN komutu pek çok SMTP gerçekleştirimi tarafından kullanılmamaktadır. Diğer komutlar için RFC-5321'e başvurulabilir.

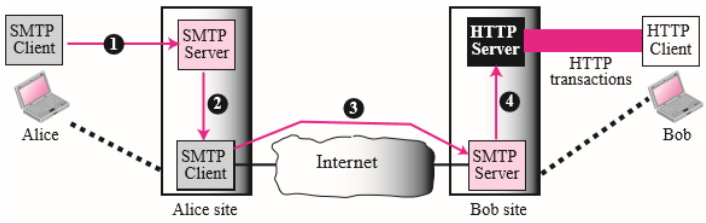
SMTP server programın gönderdiği yanıtlar üç basamaklı bir yanıt koduyla başlar. Yanıt mesajlarının metinleri sistemden sisteme farklılık gösterebilir. Yine yanıt mesajları CR/LF ile biten metinsel ve staırsal biçimdedir. Hata kodlarının listesi de şöyledir:

Code	Description
	Positive Completion Reply
211	System status or help reply
214	Help message
220	Service ready
221	Service closing transmission channel
250	Request command completed
251	User not local; the message will be forwarded
	Positive Intermediate Reply
354	Start mail input
	Transient Negative Completion Reply
421	Service not available
450	Mailbox not available
451	Command aborted: local error
452	Command aborted; insufficient storage
	Permanent Negative Completion Reply
500	Syntax error; unrecognized command
501	Syntax error in parameters or arguments
502	Command not implemented
503	Bad sequence of commands
504	Command temporarily not implemented
550	Command is not executed; mailbox unavailable
551	User not local
552	Requested action aborted; exceeded storage location
553	Requested action not taken; mailbox name not allowed
554	Transaction failed

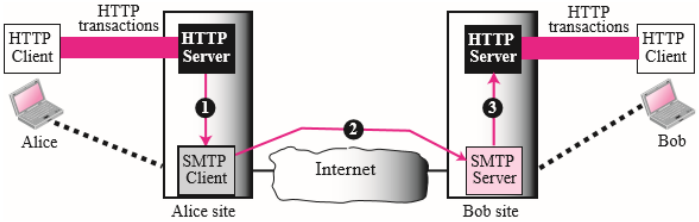
Web Tabanlı E-Posta Kullanımı

Bugün pek çok e-posta kullanıcısı gmail gibi, hotmail gibi, yahoo gibi tarayıcı tarafından erişilebilen e-posta servislerini kullanmaktadır. Genel olarak bu servislere İngilizce "Web Tabanlı E-Posta (Web Based Mail)" ya da kısaca "WebMail" denilmektedir. Web tabanlı E-Posta servisleri aslında arka planda SMTP protokollerini kullanmaktadır. Şöyle ki: Örneğin biz tarayıcıdan gmail.com'a giriş yaptığımızda tarayıcı aslında bize bir arayüz sunmaktadır. Biz HTTP kullanarak göndereceğimiz e-posta'yı bu arayüzde oluştururuz. gmail e-posta server'ı bunu yine SMTP kullanarak karşı taraftaki server'a iletir. Karşı taraf da web tabanlıysa karşı tarafın kullanıcısı yine HTTP kullanarak POP3 ya da IMAP ile alınan mesajları tarayıcıda görür. Yani web tabanlı e-posta hizmetleri aslında arka planda yine standart protokolleri kullanmaktadır.

Aşağıdaki birinci senaryoda e-posta gönderen Alice web tabanlı çalışmamaktadır. Ancak e-posta web tabanlı çalışan bir SMTP server'a gönderilmektedir.



Örneğin bu senaryoda biz Derneğimizin e-posta sunucusunu kullanarak bir gmail kullanıcısına e-posta gönderebiliriz. İkinci senaryoda e-postayı gönderen Alice de web tabanlı çalışmaktadır:



Paket Analiz Kütüphaneleri

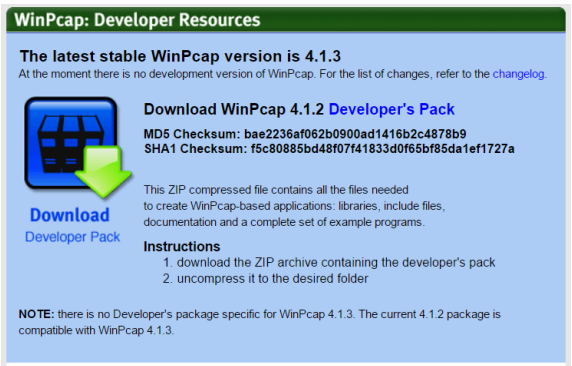
Yerel ağ içerisindeki tüm paket gönderip alımları aslında yardımcı programlarla izlenebilmektedir. Bu tür programlara "ağ yoklayıcıları ya da ağ koklayıcıları (network sniffers)" denilmektedir. Pekiyi ağ yoklayıcıları

nasıl çalışırlar? Aşağı seviyeli ethernet ya da wireless paketlerinin gönderimi ve alımı sırasında aslında paketler yerel ağdaki tüm host'lara iletilmektedir. Bu hostlardan yalnızca biri (MAC adresi uyuşan ethernet kartı) bu paketi sahiplenerek almaktadır. Fakat istenirse bize gelmiş olan fakat bizle alakalı olmayan IP paketlerinin de alınması mümkündür. İşte network yoklayıcıları ethetnet kartına ya da wireless alıcısına gelen tüm paketleri bize gösterebilmektedir. Network yoklayıcıları ile birtakım sorunlar çözülebildiği gibi birtakım bilgiler de ele geçirilebilmektedir. Örneğin Internet kafeler yasanın zorunlu tuttuğu bazı loglama işlemlerini bu biçimde netwok trafiğini izleyerek yapmaktadır.

Bugün pek çok açık kaynak kodlu ve bedava newtwork yoklayıxı programlar vardır. Bunların çok kullanınlanlarından bir bölümü şöyledir: Wireshark, tcpdump, Kismet, ettercap. Bunlar arasından en çok kullanılanı şüphesiz Wireshark'tır (eski ismi Ethereal). Wireshark ve tcpdump aslında arka planda "pcap" denilen bir network yoklayıcı kütüphanesini kullanmaktadır. Bunun Windows versiyonuna "WinPcap" denilmektedir. Yani aslında Wireshark bir "pcap" önyüz (frontend)" programı gibidir. "pcap" kütüphanesi pek çok dilden kullanılabilecek biçimde sarmalanmıştır. Tabii kütüphane orijinal olarak C'de yazılmıştır. Ancak biz bu kütüphaneyi sınıfsal olarak C++, Java ve C# dünyasında da kullanabilmekteyiz.

Wireshark'ı Windows, Linux ve Mac OS X sistemlerinde indirip kurmak oldukça kolaydır. Tabii biz kursumuzda bu programın kullanımdan ziyade bu programın kullandığı "pcap (ya da WinPcap)" kütüphanesini temel düzeyde inceleyeceğiz. WinPcap kütüphanesini kurmak için ilgili web sayfasına girilir. Kurulum programı yalnızca DLL'leri kurmaktadır. Halbuki bizim geliştirici olarak başlık dosyalarına ve import kütüphanelerine de gereksinimimiz olacaktır. İlgili DLL'lerin kurulması için web sayfasından "Download WinPcap for Windows" seçilir:

DLL'lerin import kütüphaneleri ve başlık dosyaları "Developer's Pack" isimli paketin içerisindedir:



Ubuntu türevi Linux sistemlerinde kurulum şöyle yapılabilir:

```
sudo apt-get install pcap*
```

IP Ailesine İlişkin Veri Bağlantı (Data Link) ve Network Katmanı Protokolleri

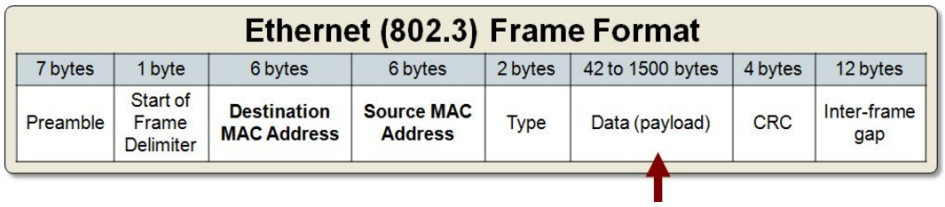
Bu bölümde aşağı seviyeli Ethernet ve IP ailesinin önemli bazı protokolleri ele alınacaktır.

Ethernet Protokolü (IEEE 802.3)

Bugün bilgisayarlarımızda en aşağı seviyeli (OSI modelinin veri bağlantı katmanına ilişkin) ağ haberleşmeleri Network kartlarıyla ya da wireless (WiFi) kartları sayesinde yapılmaktadır. Ethernet kartları tipik olarak kullandığımız telli netword kartlarıdır. Bu kartlar ilk kez 80'li yıllarda tasarlanmıştır ve sonraları git gide geliştirilmiştir. Ethernet kartları bilgileri paketler halinde bir karttan diğerine yollayıp alma potansiyeline sahiptir. Tabii yollanan paketler aslında ağa bağlı tüm birimlere gönderilir. Yani ethernet kartları ortak bir kanala bağlıdır. İçlerinden yalnızca biri (ya da broadcast yapıldığında hepsi) bunu sahiplenir. Bilgilerin kanala bırakılması USB protokolüne benzer biçimdedir. Bir kart bilgiyi göndermeden önce kanalı yoklar. Eğer kanal doluysa kısa bir süre bekler yeniden yoklar. Böylece kanalın boş olduğu bir durum yakalandığında paket gönderilir.

Ethernet kartları IEEE 802.3 protokolünü kullanmaktadır. Her ethernet kartının 6 byte uzunluğunda bir MAC (Media Access Control) adresi vardır. MAC adreslemesi fiziksel bir adreslemedir. Yani donanım birimi tarafından belirlenmiş durumdadır. Kart üreticileri bunları dünya genelinde tek olacak biçimde belirlemektedir. (Tabii günümüzde artık sahte MAC adresleri üretilebilmektedir ve bazı kartların MAC adresleri değiştirilebilmektedir.)

Bir ethernet paketi kendi içerisinde bir başlık kısmından bir de veri (data) alanından oluşmaktadır. Ethernet paketinin veri kısmı en fazla 1500 byte olabilmektedir. Ethernet paket formatı birbirine benzer birkaç biçimden oluşmaktadır. Ethernet protoklü evrimleşirken paket formatında küçük birbirleriyle uyumlu değişiklikler de yaplmıştır. Ethernet paketinin yapısı şöyledir:

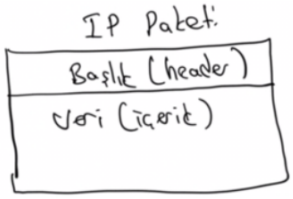


İlk 7 byte senkronizasyon için gerekmektedir. Daha sonraki byte paket bilgisinin başladığını belirtir. Bunu 6'şat byte'lık hedef ve kaynak MAC adresi alanları izlemektedir. 2 byte'lık Type alanı aynı zamanda Uzunluk alanı olarak da kullanılmaktadır. Eğer buradaki değer 1500'den büyükse bu durumda aslında bu alan uzunluğu değil ethernet veri alanındaki bilginin hangi üset düzey protokol bilgisini çerdiğini tutmaktadır. Eğer buradaki değer 1500'den küçükse bu değer doğrudan ethernet veri alanının uzunluğunu tutmaktadır. Data byte'larını hata kontrolü için gereken CRC byte'ları izler. Bu tür protokollerde gönderilen ve alınan toplam bilgiye "çerçeve (frame)" denilmektedir. Çerçevenin sonunda 12 byte'lık bir boşluk da bırakılmaktadır.

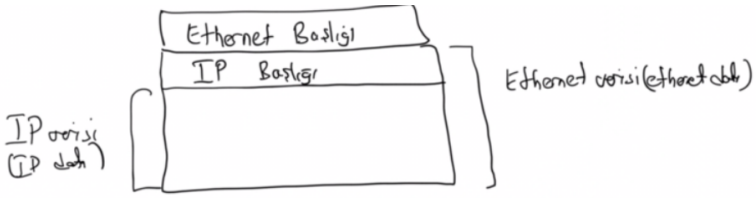
IP Protokolü (Internet Protocol)

IP ailesine ismini veren en önemli protokol IP protokolüdür. IP protokolü OSI belirlemelerinde "network katmanı (network layer)"na karşılık gelmektedir. IP tek başına güvenilir olmayan paket tabanlı bir haberleşme olanağı sunmaktadır. IP paketleri bir kaynaktan bir hedefe rotalanır (routing). Rotalayıcı (router) denilen aygıtlar sayesinde ağlar arasında yolculuk eder ve hedefe ulaştırılır. Tabii bu yolculuk sırasında paket kaybolabilir ya da bir rotalayıcı tarafında atılabilir. IP ailesinde ve genel olarak network katmanı protokollerinde gönderilip alınan paketlere "datagram" da denilmektedir. "Datagram" terimiyle "paket" terimi aynı anlamda kullanılmaktadır.

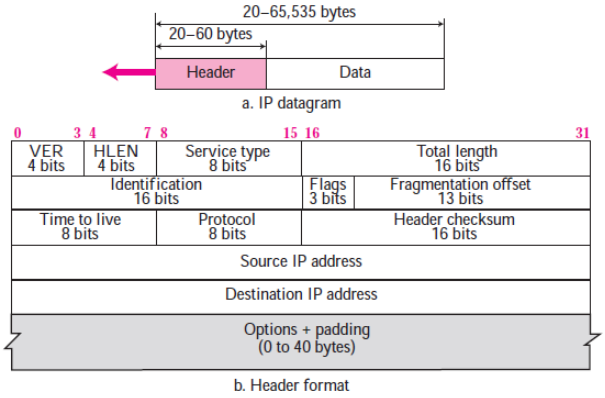
Bir IP paketi bir başlık kısmından bir de veri (data) kısmından oluşur.



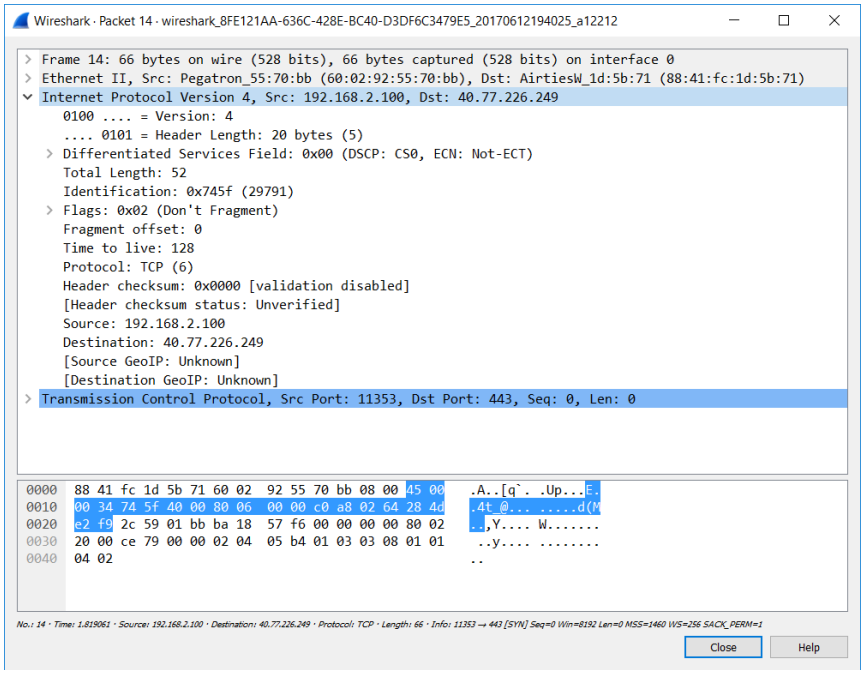
Tabii IP paketi yerel ağdaki bilgisayarımıza ethernet kartı yoluyla gelmişse bunun başında bir de ethert başlığı bulunacaktır:



IP paketinin başındaki başlık kısmı 20 byte ile 60 byte uzunlukta olabilmektedir. Başlığın ilk 20 byte'ı kesinlikle bulunmak zorundadır. Ancak 20 byte'tan sonbraki kısım seçenekler (options) ve tamamlama (padding) amaçlı kullanılmaktadır. IP paketinin ve başlığının genel görüntüsü şöyledir:



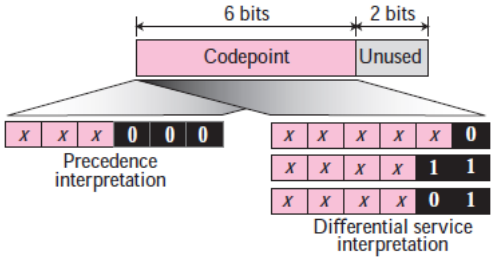
Şimdi IP başlığındaki alanları tek tek ele alalım. Aşağıda Wireshark kullanılarak elde edilen bir IP paketini görüyorsunuz. Bu paket bir ethernet paketi olarak elde edilmiştir. Dolayısıyla ethernet paketinin başında bir ethernet başlığı bulunmaktadır. Bu paket aynı zamanda bir TCP paketidir. Ancak TCP paketi de aslında IP paketinin veri kısmında gönderilmektedir.



- Başlığın ilk byte'ının yüksek anlamlı 4 biti versiyon numarasını belirtir. IPV4 için burada 4 bulunur.

- Başlığın ilk byte'ının düşük anlamlı 4 biti IP başlığının uzunluğunu belirtir. Buradaki değer 4 ile çarpılmalıdır (başka bir deyişle burada başlıkta kaç tane 4 byte olduğu bilgisi vardır.) IP başlığının en küçük uzunluğu 20 olabilir. Bu durumda bu alanda 5 değerini görmeliyiz. Bu alandaki en büyük değer 15 olabileceğine göre IP başlığının maksimum uzunluğu da 60 byte olabilir.

- IP başlığının ikinci byte'ı "servis türü (service type)" bilgisini içerir. Bu alanda zamanla bazı revizyonlar yapılmıştır. Buradaki byte bitlere ayrıştırılarak yorumlanmaktadır. Düşük anlamlı 2 bit kullanılmaz. Geriye kalan yüksek anlamlı 6 bite "code point" denilmektedir. Bu 6 bitin düşük anlamlı 3 biti 0 ise yüksek anlamlı 3 biti IP paketinin öncelik derecesini belirtmektedir. Bazen rotalayıcılar (router) kendi kuyrukları dolduğunda bazı paketleri atmak (discard) etmek zorunda kalabilirler. İşte bu durumda en düşük önelikli paketler atılır (discard edilir). Eğer 6 bitin düşük öncelikli 3 biti sıfır değilse yüksek önelikli bitleri servis türünü belirtmektedir. Bundan burada bahsetmeyeceğiz.



- IP başlığının sonraki WORD değeri IP paketinin başlık kısmı dahil olmak üzere toplam uzunluğunu belirtir. Buradan da görüldüğü gibi IP paketi en fazla 2^{16} - 1 = 65535 byte olabilir. Tabii bunun içerisine başlık kısmı da dahildir. Paketteki veri kısmının uzunluğu buradaki bilgiden başlık uzunluğun çıkartılmasıyla elde edilir.

- Başlığın "Identification", "Flags" ve "Fragmentation Offset" elemanları IP paketinin parçalandığı durumda parçalama hakkında bilgiler vermektedir. Bu konu ileride ele alınacaktır.

- Başlığın "Time To Live" kısmı IP paketinin yolunu kaybetmesi durumunda atılmasını sağlamak için düşünülmüştür. Tipik olarak kaynak rotalayıcı buraya bir değer yazar (diyelim ki 15) sonra her rotalayıcı bunu diğerine göndermeden önce 1 eksiltir. Bu değer 0'a geldiğinde ilgili rotalayıcı artık paketi atar (discard eder).

Tabii normalde rotalama mekanizmasında böyle bir durumla karşılaşılması öngörülmemektedir. Ancak rotalayıcıların rotalama tabloları bozulmuş olabilir. Bu durumda paket yolunu kaybedebilir. Bunun sürekli olarak oradan oraya yollanması gereksiz bir yük oluşturur. İşte bu alan belli bir noktadan sonra paketi atmak için kullanılmaktadır.

- Başlığın "Protocol" alanı bir byte uzunluktadır. Buraya IP paketinin veri kısmında bulunan üst düzey protokol bilgisi yerleştirilir. Örneğin bir TCP paketi yukarıda da belirtildiği gibi aslında IP paketinin veri kısmına yuvalanmaktadır. İşte o halde bu "protocol" alanında TCP protokolünü belirten değer bulunur. Hangi 8 bir değerin hangi protoklü belirttiği önceden belirlenmiştir. Bazı değerler aşağıdaki tabloda görülmektedir:

Value	*Protocol*	*Value*	*Protocol*
1	ICMP	17	UDP
2	IGMP	89	OSPF
6	TCP		

- "Header Checksum" alanı paket bilgilerinin yolda bozulup bozulmadığını anlamak için kullanılmaktadır.

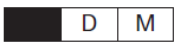
- Başlığın "Source IP" ve "Destination IP" elemanları IP paketinin çıktığı ve hedeflendiği host'u belirtmektedir. Bu alanlar paketin yolculuğu sırasında hiç değişmemektedir. Paket hedefe vardığında paketin hangi host tarafından gönderilmiş olduğu bilinmek zorundadır.

IP Paketlerinin Parçalanması (Fragmentation)

Yukarıda da belirtildiği gibi bir IP paketi 65535 byte kadar uzunlukta olabilmektedir. Halbuki bazı durumlarda paket başka fiziksel katmanlara veriliyor olabilir. Bu fiziksel katmanların desteklediği paket uzunlukları daha küçük olabilir. Örnein tipik olarak hedef LAN'a gelmiş olan paket buradaki rotalayıcı tarafından LAN'a ethernet paketi olarak verilebilmektedir. Maksimum ethernet paketi ise 1500 byte olabilmektedir. İşte bu tür durumlarda IP paketleri parçalara ayrılarak daha düşük kapasiteli fiziksel katmanlara verilebilmektedir. Örneğin 5000 byte'lık bir IP paketi ethernet paketleri olarak tek bir paket biçiminde hedef host'a gönderilemez. Bunun en azından 4 ethernet paketi biçiminde iletilmesi gerekir. İşte bu tür durumlarda rotalayıcılar paketleri parçalara ayırarak bu fiziksel katmanlara uygularlar. Bu parçalar hedefte birleştirilerek yeniden orijinal IP paketi elde edilmektedir. IP paketlerinin parçalanması rotalayıcılar (router) tarafından yapılmaktadır.

IP paketinin parçalanması sırasında her parça yine geçerli IP başlığına sahip olur. Parçalama bilgileri IP başlığındaki "Identification", "Flags" ve "Fragmentation Offset" alanları yoluyla iletilmektedir. "Identification" parçalanan paketi temsil eden 16 bitlik bir değerdir. Bu değere sahip tüm parçalar aynı paketin parçalarını oluşturmaktadır. Bu değer paketi parçalayan host tarafından üretilir. Host bu değeri her parçalayacağı paket için artırıyor olabilir. Başlık alanındaki "Flags" kısmı 3 bitten oluşmaktadır. Bu üç bitin yüksek anlamlı biti kullanılmamaktadır (reserved). Yüksek anlamlı ikinci biti (D) biti paketin parçalanıp parçalanmayacağı bilgisini verir. Bu bit 1 ise IP paketi parçalanmamalıdır. Eğer rotalayıcı bu biti 1 görürse paketi parçalamaz. Ancak fiziksel katman bu uzunluğu kaldıramayacak durumdaysa hata oluşur. Paket atılır.
Flags alanın üçüncü biti (M biti) ise paketin parçalandığını ve bu parçanın son parça olup olmadığını belirtir. Eğer bu bit 1 ise ilgili parça paketin son parçası değildir, 0 ise son parçasıdır.

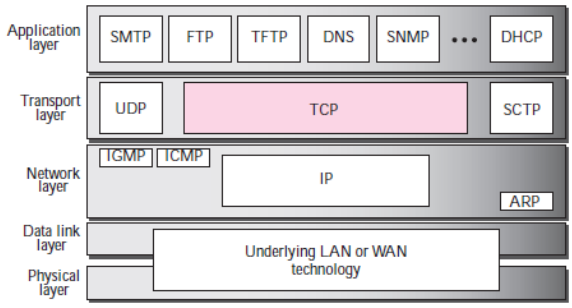
D: Do not fragment
M: More fragments



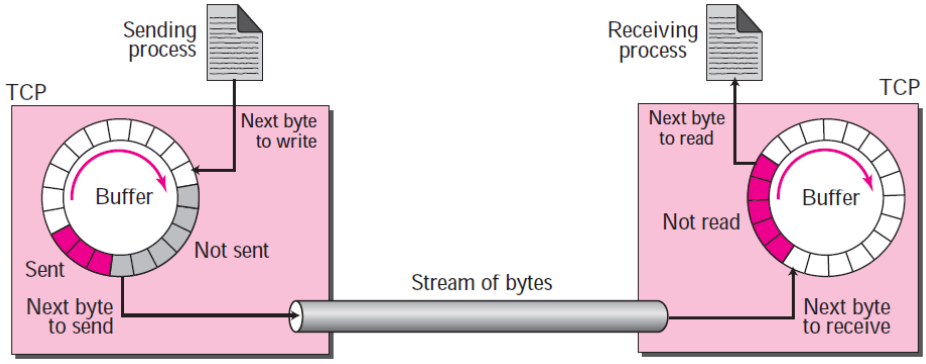
IP başlığındaki "Fragmentation Offset" ilgili parçanın asıl IP paketinin hangi kısmı olduğunu belirtir. Bu 13 bit alanda asıl IP paketindeki parçanın başladığı offset numarası vardır. Her paket parçasının bir IP başlığı olduğuna dikkat ediniz. Yani paketin her parçası hedefe geldiğinde bu paketin hangi parçalarının asıl IP paketinin neresi olduğu anlaşılabilmektedir.

TCP Protokolü (Transmission Control Protocol)

Bilindiği gibi TCP IP ailesinin transport katmanına ilişkin en önemli protokollerinden biridir. IP protokolü üzerine yığılmıştır. Yani TCP verileri aslında IP paketinin veri kısmına kodlanarak IP paketi biçiminde gönderilip alınmaktadır. TCP güvenilir (reliable), stream tabanlı (stream based), bağlantılı (connection oriented) bir protokoldür. Protokolün güvenilir olması paket iletimlerinde sorun olduğunda bunun tespit edilmesi ve sorunun giderilmeye çalışılmasıyla ilgilidir. Stream tabanlı protokoller adeta boru haberleşmesi gibi byte düzeyinde akış sunarlar. Yani gönderen tarafın gönderdiğini alan taraf tek hamlede almak zorunda değildir. Soketten istediği kadar byte'ı istediği zaman okuyabilir. Bağlantılı protokolden kastedilen de iletişim için önce iki tarafın anlaşması gerekliliğidir. Bu da tipik olarak istemci-sunucu (client-server) tarzı bir modeli akla getirmektedir.



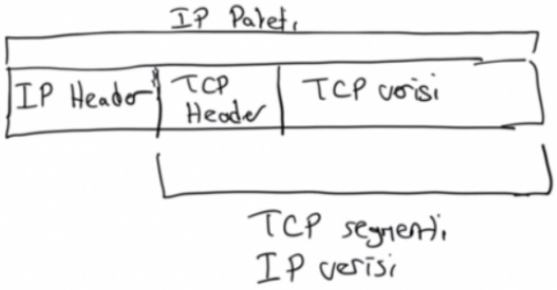
Stream tabanlı haberleşmeyi sağlamak için TCP'de alınmış olan ve gönderilecek olan bilgiler bir kuruk sistemi içerisinde tamponlanmaktadır. Genellikle bu amaçla iki tampon kullanılır: Gönderici tamponu (sending buffer) ve alıcı tamponu (receiving buffer). Bu tamponların büyüklükleri işletim sisteminden işletim sistemine değişebilmektedir. Ancak her bağlantı için bu tamponlardan ayrı ayrı oluşturulmaktadır. Bunların aytıntılı veri yapıları sistemden sisteme değişebilmektedir.



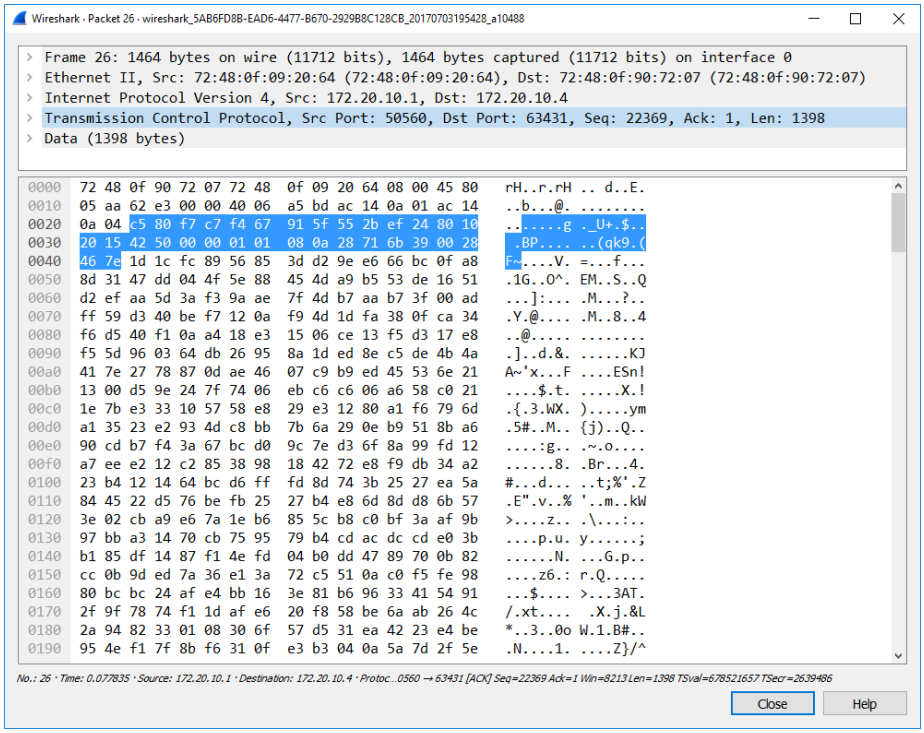
Yukarıdaki şekilde de temsil edildiği gibi gelen bilgiler bir kuyruk sisteminde bakletilir. Proses soketten okuma yaptığında eğer daha önce gelip kuyruklanmış bilgiler varsa bunları okur. Eğer kuyruk tamamen boşsa blokede bekler. Gönderim sırasında da gönderilecek bilgiler önce bir kıyruğa yazılırlar. Uygun bir zamanda bu kuyruktan alınarak grup halinde gönderilirler.

Bilindiği gibi transport katmanına ilişkin TCP ve UDP protokollerinde port numarası kavramı da vardır. Aynı host'a gönderilen TCP paketleri bu port numaralarına göre ayrıştırılarak kuyruklanmaktadır. Böylece işletim sistemi (network alt sistemi) port numaraları temelinde bilgileri organize etmektedir.

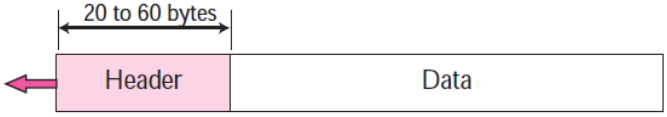
TCP stream tabanlı bir protokol olduğu için gönderilen ve alınan TCP parçacıklarına "TCP segmenti" denilmektedir. TCP terminlojisinde paket yerine segment teriminin kullanıldığına dikkat ediniz. Şüphesiz bir TCP segmenti aslında IP paketinin verisi gibi gönderilip alınmaktadır.



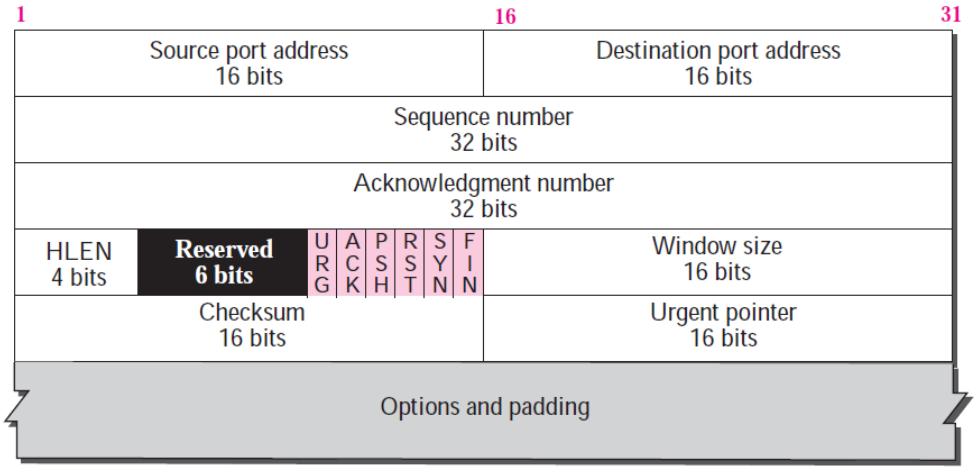
Örneğin, Wireshark'ta yakalanan bir TCP paketi aşağıdaki gibi görüntülenmiştir:



TCP segmentinin başlık kısmı (TCP header) 20 ile 60 byte arasında olabilmektedir:



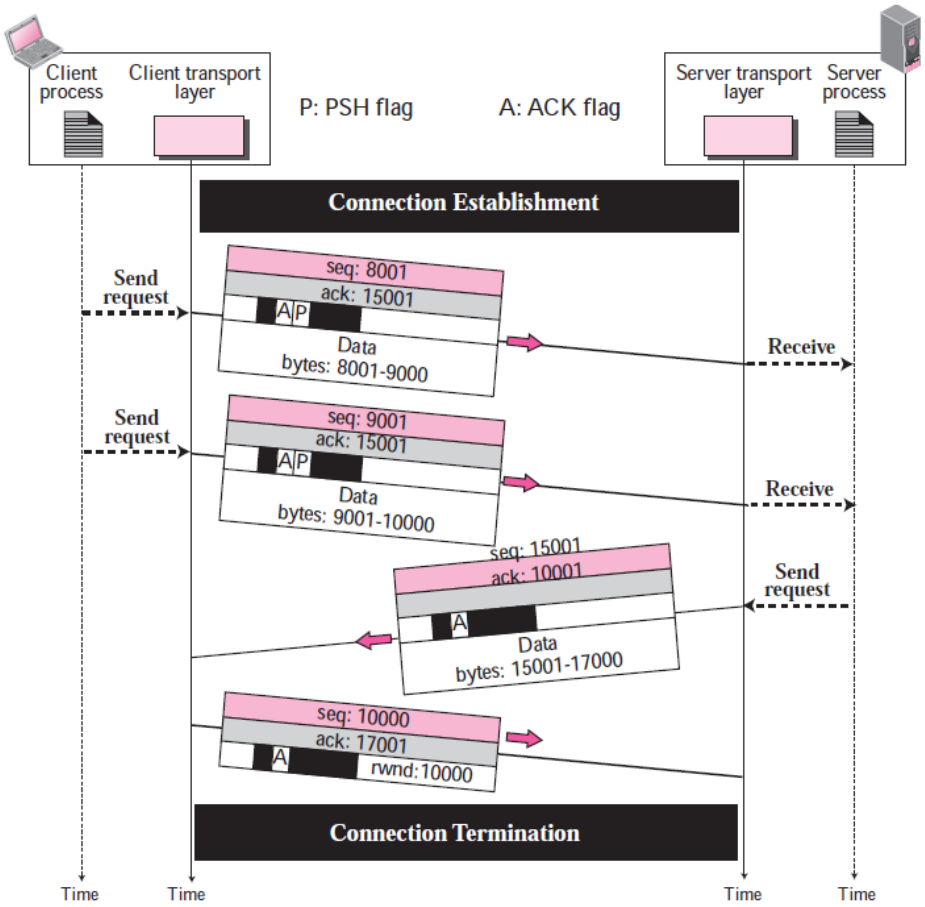
TCP başlığının formatı da şöyledir:



Başlığın ilk iki WORD elemanı sırasıyla kaynak port numarasını ve hedef port numarasını belirtmektedir. (Buradaki değerler big endian formata göre yerleştirilmiştir.) Başlıktaki "sıra numarası (sequence number)" ilgili TCP segmentinin gönderilen bilgilerdeki yerini belirtmektedir. Örneğin biz send fonksiyonuyla 3000 byte'lık bir gönderim yaptığımızda bu 3000 byte tek bir IP paketi içerisinde tek bir TCP segmenti olarak gönderilmek zorunda değildir. Ayrıca gönderen taraf küçük zaman aralıklarıyla gönderimde bulunduğunda bu gönderiler network tamponunda fazlaca bekletilmeyeceği için ayrı IP paketleri olarak gidecektir. Alan taraf açısından düşünürsek alan taraf aynı porta gelmiş birkaç IP paketini gördüğünde bunları nasıl sıraya dizecektir? Şüphesiz alan tarafa gelen paketler gönderen tarafın gönderme sırasına göregelmek zorunda değildir. İşte sıra numarası IP paketlerinin içerisindeki TCP segmentlerinin göreli sırasını belirlemek için bulundurulur. Ancak sıra numarası 1, 2, 3 biçiminde giden sayılar değildir. Aslında sıra numarası segmentteki TCP verisinin

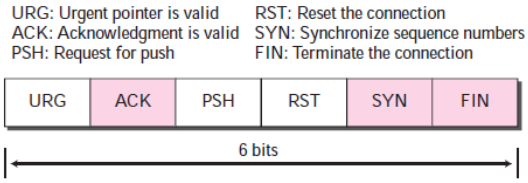
(başlıktan sonra kısmın) offsetini belirtir. Yani adeta TCP verisindeki her byte'ın dosyadaki gibi bir offset numarası var gibidir ve sıra numarası da o segmente gönderilen TCP verisinin hangi offsetten başladığını belirtir. Ancak sıra numarası 0'dan başlamak zorunda değildir. DWORD uzulukta rastgele bir değerden başlatılır. (Wireshark default durumda bu rastgele değeri sanki sıfırmış gibi göstermektedir. Bunu engellemek için "Edit/Prefernces/Protocols/TCP/Relative sequence numbers"ta çarpılama kaldırılmalıdır.) Örneğin sıra numarasının başlatıldığı rastgele değer 150100 olsun. Şimdi tek bir segmentte 100 byte gönderildiğini düşünelim. Bu segmentin başlığındaki sıra numarası 150100 olacaktır. Bu 100 byte'tan sonra 50 byte'lık bilginin başka bir segment'te gönderildiğini düşünelim. Bu TCP segmentindeki sıra numarası da 150150 olacaktır. Sıra numarasının ilk değeri bağlantı kurulurken belirlenmektedir. Bağlantı kurulurken bağlantıyı kurmaya çalışan (connect olmak isteyen) taraf rastgele sıra numarasını üretir vebu sıra numarasıyla SYN segmentini gönderir. Sonra karşıdan SYN+ACK segmenti geldikten sonra buna ACK segmenti ile yanıt verirken bu sayıyı bir artırır. Bundan sonra artık sıra numarası gönderilen veri miktarı kadar artırılacaktır. Benzer biçimde bağlantıyı kabul eden taraf da SYN+ACK segmentini gönderirken rastgele sıra numarasını oluşturmaktadır. Sonra da buna 1 ekleyerek sonraki ilk segmentin sıra numarasını oluşturur. Artık o da gönderdiği veri miktarın uzunluğu kadar sıra numarasını artırır. SYN, SYN+ACK ve ACK bağlantı segmentleri ileride ele alınmaktadır.

TCP başlığında sıra numarasını "alındı numarası (acknowledgement number)" izlemektedir. TCP akış kontrolü olan bir protokoldür. Akış kontrolü demek gönderenin alan tarafın gönderilene aldığına emin olması demektir. Bu tipik olarak gönderme işleminden sonra karşı taraftan "alındı" bilgisinin gönderilmesiyle yapılmaktadır. Eğer gönderen taraf karşı taraftan alındı bilgisi gelmezse bilginin yolda kaybolduğunu düşünür. Onu yeniden göndermek isteyebilir. İşte bu alındı numarası akış kontrolü için kullanılmaktadır. Alındı numarası hangi değerde olmalıdır? İşte bilgiyi alan taraf karşı tarafa alındı gönderirken alındı numarası olarak karşı tarafın bir sonraki gönderimde kullanacağı sıra numarasının bir fazlasını kullanır. Örneğin gönderen taraf en son 150150 sıra numarasıyla 30 byte'lık veriye sahip bir TCP segmenti göndermiş olsun. Alan taraf alındı numarası olarak 150181 gönderir. Aşağıdaki şekilde böyle bir haberleşme örneği verilmiştir:



Buradaki şekilde sol taraftaki host sağ taraftaki host'a üst üste iki TCP segmenti yollamıştır. Her iki gönderim de 1000 byte'lık TCP verisi içermektedir. Sağ taraf bu iki segmenti tek bir segment ile (tek bir ACK ile) onaylamıştır. Gerçekten de alındı onayı her segment için ayrı ayrı verilmek zorunda değildir. Biriktirilip tek hamlede pek çok sıralı segment için tek bir onay verilebilir. Ayrıca alındı onayı TCP verisiyle birlikte de gönderilebilmektedir. Sağ taraf gerçekten de sol tarafa hem 2000 byte TCP verisi gönderirken aynı zamanda aldıklarına onay da vermiştir.

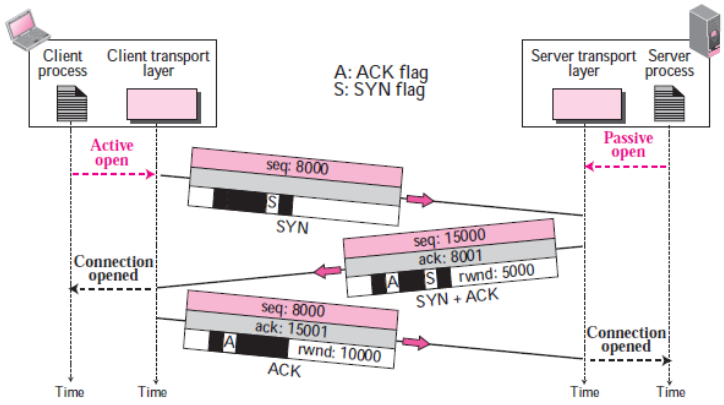
TCP başlığındaki HLEN alanı TCP başlığının uzunluğunu belirtir. Başlıkta HLEN için 4 bit ayrılmıştır. Başlık uzunluğu burada yazan değerin 4 katıdır (başka bir deyişle başlık uzunluğunu elde etmek için buradaki değer 4 ile çarpılmalıdır). O halde en büyük TCP başlığı 15 * 4 = 60 byte olabilir. Her TCP başlığının bir bayrak alanı (kontrol alanı) vardır. Burada bit bit anlamlı 6 bayrak bulunmaktadır:



SYN, FIN ve ACK bayrakları bağlantı sağlanırken ve kapatılırken kullanılmaktadır. PSH ve URG bayrakları "push" ve "urgent" gönderimlerde kullanılır. RST bayrağı bağlantının reset edilmesiyle ilgilidir. TCP başlığındaki Checksum alanı hata kontrolü için bulundurulmuştur. TCP segmentinin yolda bozulması checksum ile kontrol edilmektedir. "Window size" ve "urgent pointer" alanları şimdilik burada dikkate alınmayacaktır.

TCP Bağlantısının Kurulması

Bilindiği gibi soket haberleşmesinde client taraf connect işlemi yaptığında server taraf accept işlemi ile bağlantıyı sağlamaktadır. Bağlantının aşağı seveli olarak sağlanması üç IP paketinin (içi boş TCP segmentlerinin) gönderilmesiyle yapılmaktadır. Buna TCP terminolojisinde "üç yönlü el sıkışma (three-way handshaing)" denilmektedir. Üç yönlü el sıkışmada önce client taraf SYN bayrağını ve başlangıç sıra numarasını (sequence number) set ederek boş bir TCP segmenti yollar. SYN bayrağı yeni bir sıra numarasının üretildiği anlamına gelmektedir. Server taraf buna SYN ve ACK bayrakları set edilmiş bir TCP segmentiyle yanıt verir. Böylece server hem bağlantı isteğinin alındığını (ACK bayrağı) ve aynı zamanda kendisi için yeni bir sıra numarası üretildiğini belirtir. Nihayet bu kez client son olarak ACK bayrağı set edilmiş bir TCP segmenti yollar ve böylece el sıkışma tamamlanmış bağlantı kurulmuş olur. Örneğin:

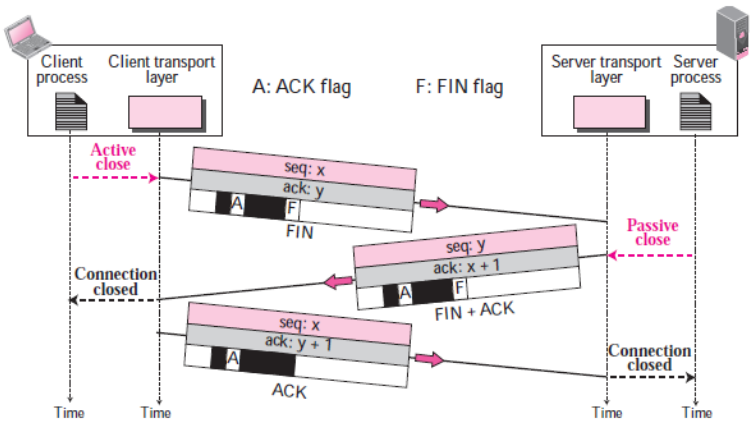


Örneğin 192.168.2.61 numaralı client host'un 192.168.2.124 numaralı server'a (Telnet server) bağlanmasına ilişkin üç TCP segmenti Wireshark ile aşağıdaki gibi görüntülenmektedir:

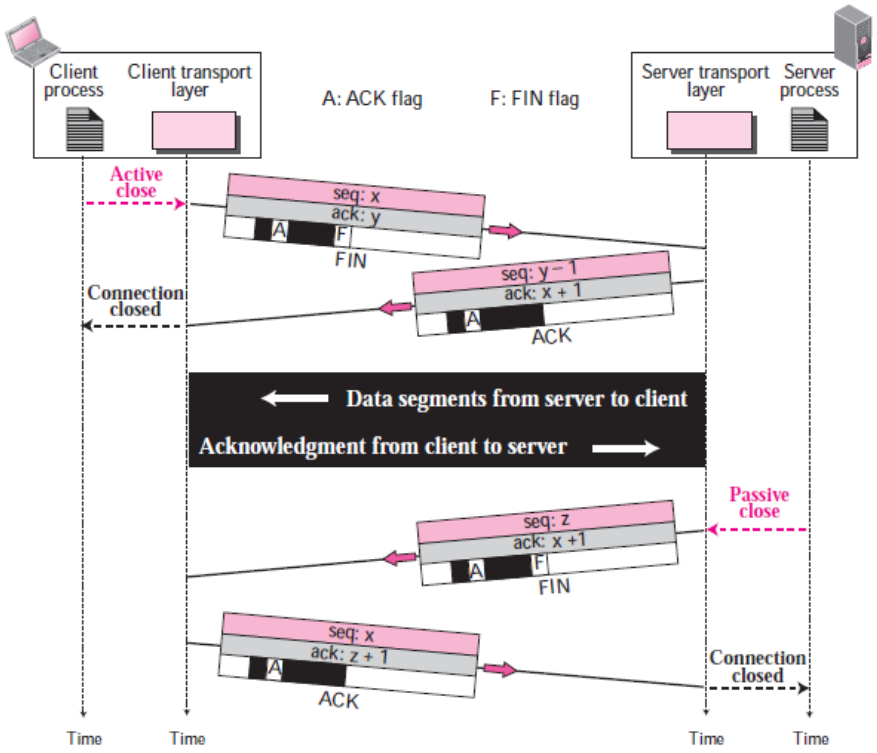
No.	Time	Source	Destination	Protocol	Leng	Info
35	3.778025	192.168.2.61	192.168.2.124	TCP	66	6018 → 23 [SYN] Seq=3332974236 Win=8192 Len=0 MSS=1460 WS=256 SACK…
36	3.778354	192.168.2.124	192.168.2.61	TCP	66	23 → 6018 [SYN, ACK] Seq=2976957298 Ack=3332974237 Win=8192 Len=0 …
37	3.778711	192.168.2.61	192.168.2.124	TCP	60	6018 → 23 [ACK] Seq=3332974237 Ack=2976957299 Win=65536 Len=0

Bağlantının Kapatılması

Bağlantının kapatılması da tam olarak ya da yarım olarak (half close) yapılabilmetedir. Tam kapatmada yine üç yönlü el sıkışma kullanılmaktadır. Client ya da server taraf işlemi başlatabilir. Önce bir taraf diğer tarafa FIN bayrağı set edilmiş boş bir TCP segmenti yollar. Diğer taraf buna FIN + ACK bayrakları set edilmiş boş bir TCP segmentiyle karşılık verir. Nihayet son olarak diğer taraf ACK bayrağı set edilmiş boş bir TCP segmentiyle son onayı vermektedir. Örneğin:



Yarım kapatma işleminde (soket API'lerinde shutdown fonksiyonu ile sağlanır) bir taraf önce yalnızca göndermeyi kesmeyi taahhüt eder. Sonra alım yapabilir. Sonra gerçekten tam olarak iletişimi sonlandırır. Örneğin client taraf server tarafa birtakım bilgileri gönderim yarım kapatma uygulayabilir. Artık client taraf server tarafa bir daha bir göndermede bulunmayacaktır. Ancak server'ın gönderdiği bilgileri okuyabilecektir. Yarım kapatmada bir taraf FIN segmentini yollar. Karşı taraf buna yalnızca ACK yollamaktadır. Böylece FIN yollayan taraf okuma yapabilir ancak gönderme yapamaz. Yarım kapatmadan sonra aynı biçimde karşı taraf da yarım kapatma uygularsa iletişim tam olarak sonlandırılmış olur. Örneğin:



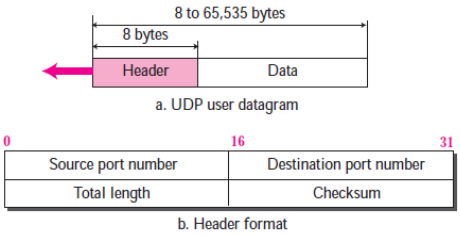
TCP Akış Kontrolünde Zamanlama

TCP akış kontrolünün bazı ayrıntıları vardır. Ancak kursumuzun süresi itibari ile bu ayrıntılar ele alınmayacaktır. Burada akış kontrolündeki temel bir durumu dikkate alacağız. Bir taraf diğer tarafa bir TCP segmenti gönderdiğinde karşı atarftan bir ACK gelmezse ne olur? Normal olarak TCP akış kontrolünde her gönderilen segment ya da segment grubu için onların alındığını belirten bir ACK segmentinin gönderilmesi gerekir. Böylece gönderen kişi segmentin alındığından emin olur. Pekiyi bir segmente karşılık ACK gelmemesinin nedeni ne olabilir? Aslında bunun birkaç nedeni söz konusu olabilmektedir. Birincisi gönderen kişinin TCP segmenti (IP kapeti) karşı tarafa hiç ulaşmamış olabilir. İkincisi karşı tarafın gönderdiği ACK segmenti diğer tarafa ulaşmamış olabilir. Başka diğer sebepler de söz konusu olabilmektedir. Fakat ne olursa olsun karşı tarafından segmentin alındığını bildirmemesi bunun karşı tarafa ulaşamamış olma şüphesini akla getirir. İşte TCP'de gönderen taraf bir zamanlayıcıyı başlatır. Belli bir zaman aşımı dolana kadar karşı taraftan ACK segmentinin gelmesini bekler. Eğer karşı taraftan söz konusu zaman aralığında bu segment gelmemişse gönderen taraf yeniden segmenti gönderir.

UDP Protokolü (User Datagram Protocol)

Bilindiği gibi UDP bağlantısız (connectionless) ve güvenilir olmayan (unreliable) bir protokoldür. UDP protokolü AP protokolünün üzerine oturtulmuştur. Yani UDP paketleri (user datagram) IP paketi biçiminde gönderilip alınır. Tabii UDP transport katmanına ilişkin bir protokol olduğu için port numarasına sahiptir. Pratikte IP protokolü tek başına kullanılmadığı için paket haberleşmesi UDP ile yapılmaktadır.

UDP paketinin başlık kısmı (UDP Header) şöyledir:



Görüldüğü gibi UDP başlığı çok az elemana sahiptir. Zaten paket haberleşmesi için gereken diğer bilgilerin hepsi IP başlığında vardır. Yine başlıkta 16 bit kaynak ve hedef port numaraları vardır. Bunu UDP paketinin toplam uzunluğu izler. Toplam uzunluk UDP başlık kısmı ile UDP veri içeriğinin toplamı kadardır. Aslında anımsanacağı gibi IP başlığında zaten IP paketinin toplam uzunluğu bulunuyordu. Bu uzunluktan IP başlığının uzunluğu çıkartılırsa yine UDP paketinin toplam uzunluğu elde edilir. Yani aslında buradaki toplam uzunluk (total length) alanı hiç olmayabilirdi. Ancak hesaplama gerekmemesi için bu uzunluk yeniden UDP başlığında bulundurulmuştur. UDP başlığının son elemanı "checksum" bilgisidir. "Checksum" bilgisi IP başlığındaki bazı bilgiler, UDP başlığı ve UDP verileri dikkate alınarak hesaplanır. Aslında hesaplama işlemi IP paketindeki checksum hesabı gibidir. Hem IP paketinde hem de UDP paketinde checksum için 16 bit "1'e tümleme yöntemi" kullanılmaktadır. Bu yönteme göre checksum'ı elde edilecek bilgi 16 bitlik (WORD) kısımlara ayrılır. Bu 16 bitlik değerler toplanır. Elde edilen sonuç 16 biti aşarsa taşan kısım oradan kopartılıp değer yeniden toplanır. Örneğin aşağıdaki iki 16 bit değer "1'e tümleme checksum yöntemiyle" toplanacak olsun:

```
1010 1111 1000 1010
1111 0101 1100 0001
```

Bu toplamdan aşağıdaki sonuç elde edilir:

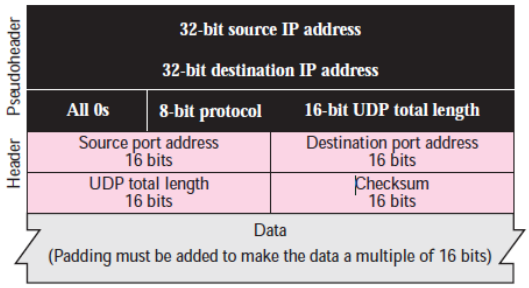
```
1 1010 0101 0100 1011
```

Burada taşan kısım 1'dir. Bu 1 değeri sayıya eklenmektedir:

```
1010 0101 0100 1011
                1
```

```
1010 0101 0100 1100
```

Bu biçimde 16 bir değerler toplana toplana gidilir. En sonunda elde edilen değerin 1'e tümlemesi alınarak (yani 0'lar 1, 1'ler 0 yapılarak) nihai değer hesaplanır. Bu değer de başlığın checksum kısmına yerleştirilir. Karşı taraf aynı işlemleri yaparak bir değer elde eder. Tabii hesaplanan checksum da buna dahil olduğu için toplam elde edilen değer 0 olacaktır.) İşte alan taraf eğer bu checksum hesabından sıfır elde etmezse paketin yolda bozulduğu sonucunu çıkartır ve paket sanki hiç gelmemiş gibi atar. Karşı tarafa da "yeniden gönder" gibi bir bildirimde bulunmaz. UDP checksum bilgisinin hesaplanmasında ve kontrol edilmesinde kullanılan üç alan şunlardır:



ARP (Address Resolution Protocol) Protokolü

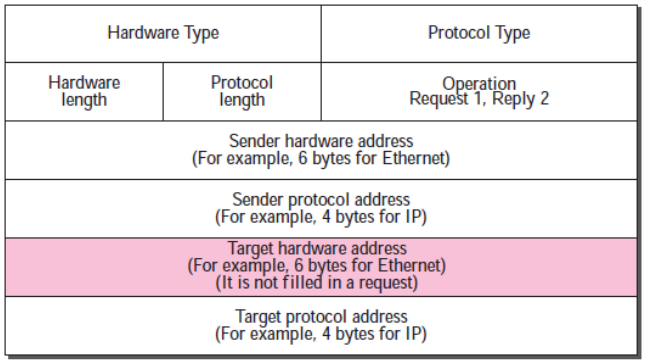
Bilindiği gibi IP protokol ailesinde ister versiyon 4 olsun ister versiyon 6 olsun her host'a tüm ağ genelinde tek olan mantıksal bir IP numarası atanmaktadır. IP numaralarının atanması tipik olarak DHCP protokolü yardımıyla yapılmaktadır. Bu protokol sonraki bölümde ele alınacaktır. Tabii IP protokolünün kendisi bir "ağ katmanı (network layer)" protokolüdür. IP ailesindeki host'ların ilişkisi tamamen mantıksal düzeydedir (yani yazılımsal düzeydedir). Halbuki eninde sonunda paketler havadan telsiz (wireless) haberleşmesi ile ya da kablolardan elektirksel yolla iletilir. Anımsanacağı gibi bu alt katmana "data bağlantı katmanı (data-link layer)" ve "fiziksel katman (physical layer)" denilmektedir. PC'lerde ağırlıklı olarak "ethernet sistemi" kullanılmaktadır. "Ethernet" hem bir veri katmanını hem de fiziksel katmanı belirtir. Ethernet sisteminde hem fiziksel unsurlar (soketler, elektirksel işaretler vs.) hem de mantıksal unsurlar (ethernet protokolü, aktarılan paketlerin yapısı vs.) tanımlanmıştır.

IP haberleşmesinde bir host ip adresini bildiği başka bir host'a paket gönderecekken eninde sonunda onun fiziksel adresini (yani veri bağlantı katmanında kullanılacak adresini) bilmek zorundadır. Örneğin Ethernet sisteminde tipik olarak paket nihayetinde bir hedef MAC adresi belirtilerek ağdaki kanallara bırakılır. Pekiyi A isimli bir host B isimli bir host'a IP paketi gönderecek olsun. A host'unun B'nin IP adresini bildiğini varsyalım. Nihayetinde bu paket fiziksel olarak hangi host hedef alınarak gönderilecektir? Başka bir deyişle IP adresini bildiğimiz host'ın fiziksel adresi (ethernet için konulursak MAC adresi) nedir? İşte ARP protokolü IP adresi bilinen host'un fiziksel adresini elde etmek için kullanılmaktadır. Baişka bir deyişle ARP protokolü mantıksal adresleri (IP adreslerini) fiziksel adreslere (MAC adresine) dönüştürmek için kullanılır. Örneğin yerel ağda A makinesi 192.168.1.20 IP adresine sahip B nmakinesine bir IP paketi gönderecek olsun. Yerel ağda bu paket niayetinde ethernet bağlantısıyla fiziksel olarak gönderilecektir. Bunun için A makinesi bir IP paketi oluşturur. Bu IP paketini bir Ethernet paketi içerisine (onun data kısmına) yerleştirir. Şimdi ethernet hattına bırakmak için hedef makinenin MAC adresini bilmesi gerekmektedir. İşte A makinesi bu işlem için önce B makinesinin MAC adresini elde etmek ister. Bu MAC adresi de ARP protokolü ile elde edilmektedir. ARP protokülü Yalnızca aynı yerel ağdaki iki host arasında değil aşağıdaki dört durumda da kullanılmaktadır:

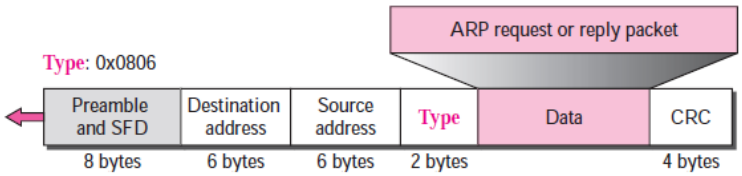
1) Yerel ağdaki bir host'un yerel ağdaki diğer bir host'a bilgi göndermesi için. (Bu durumda bilgi gönderecek host diğerinin fiziksel adresini (MAC adresini) bilmek zorundadır.)
2) Router'ın yerel ağdaki bir host'a bilgi göndermesi için. (Bu durumda router yerel ağdaki host'un fiziksel adresini (MAC adresini) bilmek zorundadır.)

3) Yerel bir ağdaki bir host'un router'a (gateway'e) bilgi göndermesi için. (Bu durumda yerel ağdaki host default gateway IP adresini yani router'ın IP adresini bilir. Ancak onun fiziksel adresini (MAC adresini) elde etmek zorundadır.)
4) Bir router'ın dış fiziksel katmandaki başka bir router'a bilgi göndermesi için. (Bu durumda başka fiziksel katman protokolleri kullanılarak bir router diğerinin o fiziksel protoldeki fiziksel adresini bilmek zorundadır.)

ARP protokolü oldukça basittir. Bilgiyi gönderecek host bir ARP paketi düzenler. Bunu fiziksel katman protokolünün data kısmına (örneğin ethernet paketinin data kısmına) yerleştirir. Bilgiyi fiziksel hatta bağlı tüm host'lara (broadcasting) gönderir. O hosta'lardan biri bu gönderiye yanıt verir ve kendi MAC adresini diğerine bildirir. ARP paketinin yapısı şöyledir:



Bu ARP paketi ethernet sisteminde ethernet data'sı olarak pakete yerleştirilmektedir:



ARP paketindeki elemanlar şöyledir:

Hardware Type: Burada ARP paketinin yerleştirileceği veri bağlantı ve fiziksel katmanın türü kodlanmaktadır. Örneğin Ethernet sistemi için buradaki değer 1'dir.

Protocol Type: Burada kullanılan ağ katmanı protokolüne ilişkin bir numara bulunur. Örneğin IPV4 için buradaki değeri 0x0800 biçimindedir.

Hardware Length: Burada fiziksel adresin byte uzunluğu bulunmaktadır. Örneğin MAC adresleri 6 byte uzunluğundadır.

Protocol Length: Burada ağ katmanı protokolünde kullanılan adresin byte uzunluğu bulunur. IPV4 için bu değer 4, IPV6 için 16'dır.

Operation: Hedeflenen işlemi belirtir. Karşı taraf yanıt olarak da ARP paketi yollamaktadır. Operation kodu gönderen için 1, yanıt veren için 2'dir.

Sender Hardware Address: ARP paketini gönderen host'un fiziksel adresi bulunur. (Yani ARP paketini gönderen host'un MAC adresi)

Sender Protocol Address: Burada ARP paketini gönderen host'un ağ katmanındaki protokol adresi bulunur. (Yani tipik olarak IPV4 ya da IPV6 adresi).

Target Hardware Address: Bu alan hedeflenen MAC adresine ilişkindir. Tabii ARP paketini gönderen taraf zaten bunu öğrenmek istemektedir. Bu nedenle gönderen taraf bu alanı doldurmaz. Bu alanın içerisine sıfır yazar. Yanıt veren taraf ise kendi MAC adresini alana yazmaktadır.

Target Protocol Address: Bu alana da hedef IP adresi yerleştirilmektedir. Yani ARP paketi tüm bağlı host'lara gönderilir. Bu IP adresi uyuşan host buna yanıt verir.

Anahtar Notlar: Hem Windows sistemlerinde hem de UNIZ/Linux sistemlerinde komut satırında kullanılan arp isimli bir komut vardır. Bu komut o andaki adres çözümlemesi yağılam host'ları bize göstermektedir. -a seçeneği tüm hostlar'ın görüntülenmesini sağlar.

Örneğin:

```
csd@csd-virtual-machine ~ $ arp
Address                  HWtype  HWaddress           Flags Mask            Iface
192.168.188.2            ether   00:50:56:fa:d9:f0   C                     ens33
192.168.188.254          ether   00:50:56:fc:df:36   C                     ens33
```

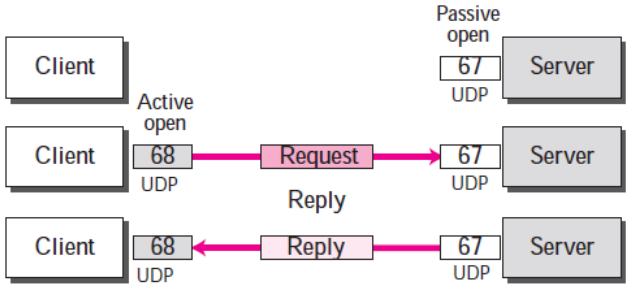
Burada hangi IP'ler için hangi MAC adreslerinin karşılık geldiği görülmektedir. Tabii bu arp komutundan elde edilen liste o andaki host'a ilişkin olan listedir. Her host'un içerisinde ayrı bir ARP alt sistemi vardır. Bu alt sistem IP protokol stack'ine dahildir.

Pekiyi bir host başka bir host'un fiziksel adresini ARP protokolü ile elde ettikten sonra bir daha o host'a bilgi gönderecekken bu adresi yeniden elde etmey çalışır mı? Mademki fiziksel adres elde edilmiştir. O halde bu adres saklanıp sonraki göndermelerde de kullanılabilir. Ancak bu adresin uzun süre saklanması da pek uygun değildir. Çünkü bir biçimde host'lar devre dışı kalabilir. Ya da onların fiziksel adresleri değişebilir. (Bazı protokollerde bu mümkündür). İşte işletim sistemlerinin network alt sistemleri genellikle ARP uygulayarak elde ettikleri fiziksel adres bilgilerini belli bir cache'ler. Bu süre dolduktan sonra o bilgiyi ARP uygulayarak yeniden elde eder.

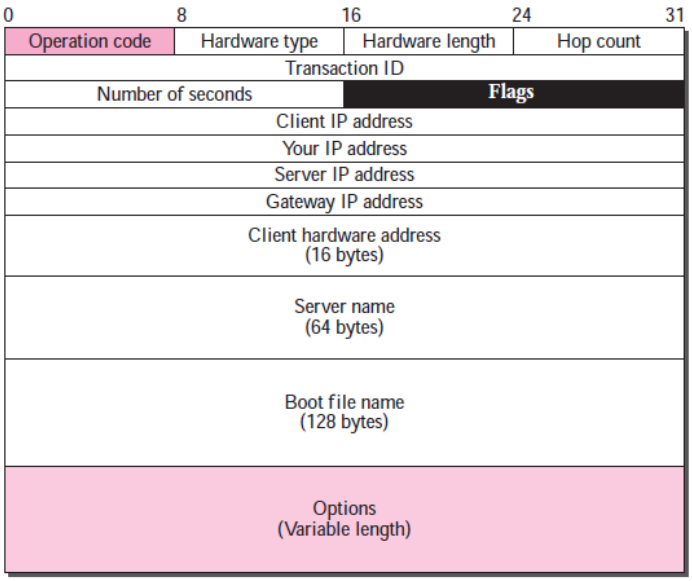
DHCP (Dynamic Host Configuration Protocol) Protokolü

Bir host'un IP adresi nasıl belirlenmektedir? IP adreslerinin belirlenmesi için statik tablolar (önceden hazırlanmış konfigürasyon dosyaları) kullanılıyor olabilir. Ancak pek çok durumda IP adresleri DHCP denilen bir protokol kullanılarak dinamik biçimde elde edilmektedir. DHCP server programı kimlere hangi IP adreslerini verdiğini kendi içerisinde bir listede tutar. Belli bir kurala göre IP adreslerini host'lara dağıtır. Şüphesiz bu sistem yine static IP verme yetisine de sahiptir. Yani DHCP server bize isteğimiz doğrultusunda birtakım koşullar da sağlanıyorsa hep aynı IP adresini verebilir. Server ile client arasındaki DHCP konuşması tipik olarak bilgisayarımız açıldığında ya da bir ağ kartı etkin hale getirildiğinde otomatik gerçekleşir. Tabii bu konuşmanın yalnızca bir kez yapılıyor olması gerekmez. Örneğin kablonun çıkması gibi bağlantı kesilme durumunda ya da istek üzerine yeniden DHCP konuşması yapılarak host yeni IP adresi elde edebilmektedir. DHCP protokolü yalnızca yerel ağda değil router'lar ile ISP'ler arasında da benzer biçimde uygulanmaktadır.

DHCP protokolünde UDP ve TFTP protokolleri de dolaylı olarak kullanılmaktadır. DHCP protokolünde IP almak isteyen tarafa client, bunu dağıtan ve onaylayan tarafa server denilmektedir. Haberleşme client'ın 68 numaralı porttan server'ın 67 numaralı portuna UDP paketi göndermesiyle başlatılır. Tabii client henüz DHCP server'ın IP adresini ve fiziksel adresini (MAC adresini) bilmemektedir. Bu nedenle client bu UDP paketini broadcast olarak gönderir. Anımsanacağı gibi UDP'de "broadcast" paket gönderme yeteneği vardır. Bu durumda paket fiziksel ağa bağlı tüm host'lara gönderilmektedir. Tabii her ne kadar client henüz server hakkında hiçbir şey bilmiyorsa da bu UDP broadcast mesaj ethernet broadcast paketi olarak tüm host'lara gönderilebilmektedir. Başka bir deyişle UDP'de broadcast paket göndermek için zaten herhangi bir IP adresi ve fiziksel adres bilmeye gerek yoktur. Client 67 numaralı porta broadcast UDP paketini gönderdiğinde DHCP server buna yanıt verir. Server yanıtını client'ın 68 numaralı portuna yollar.



DHCP paketinin formatı şöyledir:



DHCP protokolünde tüm bilgiler server'dan client'a UDP paketi olarak gönderilmez. Bir dosya ile tranfer edilir. Şöyle ki: Yukarıdaki pakette de görüldüğü gibi server client'a bir dosya ismi gönderir. Client'ta bu dosyayı TFTP protokolü ile server'dan ister.

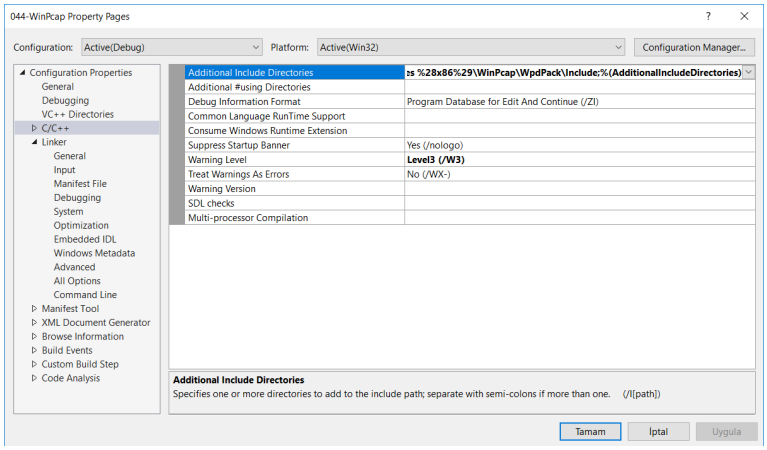
pcap ve WinPcap Kütüphanelerinin Kullanılması

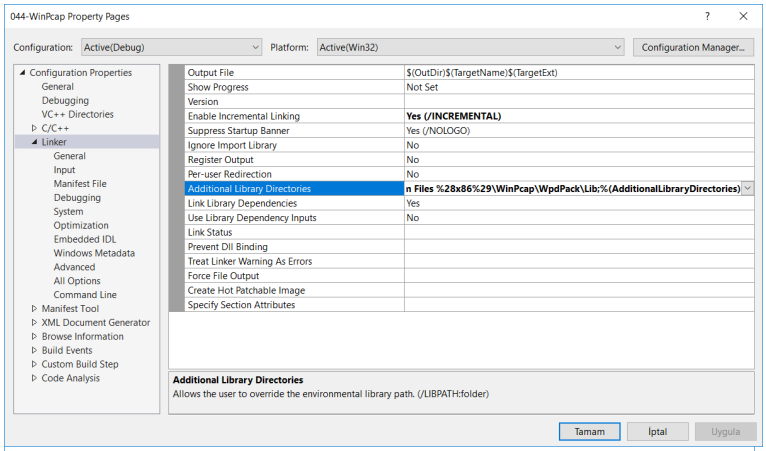
pcap paket yakalama (packet sniffing) için en çok tercih edilen kütüphanedir. Pek çok uygulama arka planda bu kütüphaneyi kullanmaktadır. Örneğin WireShark aslında adeta pcap kütüphanesinin bir önyüz uygulaması gibidir. pcap temelde bir C kütüphanesidir ancak pek çok dilden de dolaylı olarak kullanılabilmektedir. pcap kütüphanesi Linux sistemlerinde, BSD sistemlerinde Mac OS X sistemlerinde doğrudan kullanılabilmektedir. Kütüphanenin Windows versiyonuna WinPcap denilmektedir.

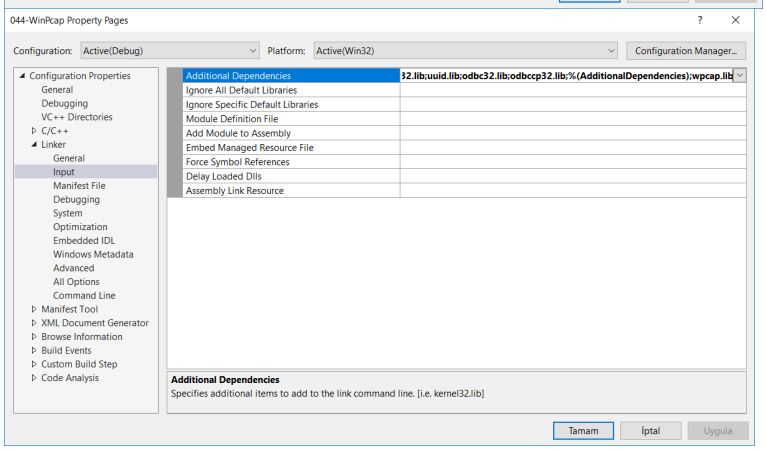
Debian türevi Linux dağıtımlarında (Ubuntu, Mint gibi) pcap kütüphanesi şöyle kurulur:

```
sudo apt-get install libpcap-dev
```

Mac OS X sistemlerinde XCode IDE'si yüklendiğinde zaten pcap kütüphanesi de yüklenmiş olmaktadır. Windows sistemlerinde doğrudan projenin web sitesinden Windows için kurulum dosyası indirilerek kurulum yapılabilir. Kurulum yapıldıktan sonra proje ayarlarından WinPcap kütüphanesinin bulunduğu include dizini ve kütüphane dizini girilmelidir:







Pcap kütüphanesindeki tüm fonksiyonlar pcap_xxx biçiminde isimlendirilmiştir. Fonksiyonların çoğu UNIX/Linux sistemlerinde alıştığımız gibi başarı durumunda 0, başarısızlık durumunda -1 değerine geri dönmektedir. Başarısızlık oluştuğunda fonksiyonlar başarısızlığın nedenini bizden adresini aldıkları yere yazmaktadır.

Pcap kütüphanesi ile işlemler önce ağ aygıtlarının tespit edilmesiyle başlatılabilir. Bunun için int pcap_findalldevs fonksiyonu kullanılabilir. Daha sonra bu fonksiyonun pcap_findalldevs_ex isimli daha geniş bir uyarlaması da oluşturulmuştur. pcap_findalldevs fonksiyonun prototipi şöyledir:

```
int pcap_findalldevs(pcap_if_t **alldevsp, char *errbuf)
```

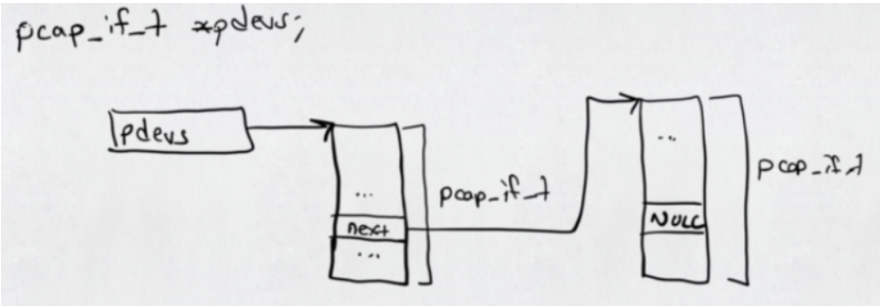
Fonskiyonun birinci parametresi pcap_if_t türünden bir göstericinin adresini almaktadır. Programcı bu göstericiyi tanımlar. Bunun adresini fonksiyona verir. Fonksiyn da bu gösterinin içerisine adres yerleştirir. Fonksiyonun ikinci parametresi hata durumunda hata mesajının yazılacağı dizinin adresini belirtmektedir. Bu dizinin PCAP_ERRBUF_SIZE kadar uzunlukta olması gerekmektedir. Fonksiyon baaşarılıysa 0, başarısızsa -1 değerine geri dönmektedir. Fonksiyon tipik olarak şöyle kullanılabilir:

```
pcap_if_t *pdevs;
char errBuf[PCAP_ERRBUF_SIZE];

if (pcap_findalldevs(&pdevs, errBuf) == -1) {
    fprintf("pcap_findalldevs: %s\n", errBuf);
    exit(EXIT_FAILURE);
}
```

Fonksiyon aygıt bilgilerini bir bağlı listeye yerleştirir. Bu bağlı listenin ilk elemanının adresini de bize verir. pcap_if_t türünün elemanları şunlardır:

struct **pcap_if** *	**next** *if not NULL, a pointer to the next element in the list; NULL for the last element of the list*
char *	**name** *a pointer to a string giving a name for the device to pass to **pcap_open_live()***
char *	**description** *if not NULL, a pointer to a string giving a human-readable description of the device*
struct **pcap_addr** *	**addresses** *a pointer to the first element of a list of addresses for the interface*
u_int	**flags** *PCAP_IF_ interface flags. Currently the only possible flag is **PCAP_IF_LOOPBACK**, that is set if the interface is a loopback interface.*

Yapının name elemanı aygıt ismini desciption elemanı onu betimleyen ifadeyi verir. Yapının next elemanı bağlı listenin sonraki elemanını göstermektedir.



Bu bağlı listenin dolaşılması ve aygıt bilgilerinin yazdırılması şöyle yapılabilir:

```
#include <stdio.h>
#include <stdlib.h>
#include <pcap.h>

int main()
{
    pcap_if_t *pdevs;
    char errBuf[PCAP_ERRBUF_SIZE];

    if (pcap_findalldevs(&pdevs, errBuf) == -1) {
        fprintf("pcap_findalldevs: %s\n", errBuf);
        exit(EXIT_FAILURE);
    }

    while (pdevs != NULL) {
        printf("Name: %s, Description: %s\n", pdevs->name, pdevs->description);
        pdevs = pdevs->next;
    }

    return 0;
}
```

pcap_findalldevs fonksiyonu ile elde edilen bağlı liste pcap_freealldevs fonksiyonuyla serbest bırakılabilir.

```
void pcap_freealldevs(pcap_if_t *alldevsp);
```

Fonksiyon parametre olarak pcap_if_t türünden bağlı listenin başlangıç düğümünün adresini alır.

Tüm aygıt listesini almak yerine işletim sisteminin belirlediği default aygıtın ismini hızlı biçimde ele geçirebiliriz. default aygıt hem ethernet aygıtı hem de wireless aygıtı varsa ethernet aygıtı, yalnızca wireless aygıtı varsa wireless aygıtıdır. Default aygıtın ismi pcap_lookup_dev fonskiyonuyla elde edilmektedir:

```
char *pcap_lookupdev(char *errbuf);
```

Fonksiyonun parametresi yine hata durumunda hata mesajının yerleştirileceği tamponun adresidir. Geri dönüş değeri aygıt ismini bize verir. Fonskiyon başarısızlık durumunda NULL adrese geri döner.

Artık sıra aygıtı açmaya gelmiştir. Aygıtı açmak için birkaç fonksiyon vardır. Canlı izleme amacıyla açma pcap_open_live fonksiyonuyla yapılmaktadır:

```
pcap_t *pcap_open_live(const char *device, int snaplen, int promisc, int to_ms, char *ebuf);
```

Fonskiyonun birinci parametresi açılacak aygıtın ismini alır. İkinci parametre paket elde edilirken onun ne kadarının elde edileceğini belirtir. Üçüncü parametre açış modunu belirtir. Aslında network ya da wireless adaptörler yerel ağdaki tüm paketleri elde edebilmektedir. Ancak bunlar yalnızca gelen paket kendilerine ilişkinse işletim sistemini haberdar ederler. (Bu haberdar etme işlemi tipik olarak kesme ile yapılmaktadır.) Ancak bu adaptörler "promiscuous mode" denilen moda sokulduklarında kendileriyle ilgili olsun ya da olmasın her paket için işletim sistemini haberdar etmektedir. İşte bu üçüncü parametre eğer 0 ise "normal mode", sıfır dışı ise "promiscuos mode" anlaşılır. Paket yakalıcı (packet sniffer) programlar "promiscuous" modu kullanmaktadır. Fonksiyonun dördüncü parametresi zaman aşımı değeridir. Yani paket elde edilirken belli bir süre beklenir. Eğer adaptöre paket gelmemişse fonksiyon başarısız olur. Fakat bu parametre özel olarak 0 girilirse paket gelene kadar blokeye yol açmaktadır. Fonksiyonun son parametresi yine hata mesajının yerleştirileceği tamponun adresini alır. Fonksiyon başarı durumunda pcap_t türünden bir handle değerine başarısızlık durumunda ise NULL değerine geri döner.

Açılmış olan aygıtın kapatılması pcap_close fonksiyonuyla yapılmaktadır:

```
void pcap_close(pcap_t *p);
```

Aygıtı açtıktan sonra artık sıra gelen paketleri almaya gelmiştir. (Ancak bu aşamada filtre de uygulanmak istenebilir.) Paketleri almanın iki yolu vardır. Birinci yolda paketler tek tek pcap_next fonksiyonu çağrılarak alınır. Dolayısıyla eğer birden fazla paketin alınması isteniyorsa bu fonksiyonu da döngü içerisinde çağırmak gerekir. İkinci yöntemde pcap_loop isimli fonksiyona bir fonksiyon adresi argüman olarak verilir. Paket geldikçe bu pcap_loop fonksiyonu adresini veriğimiz fonksiyonu ("callback" fonksiyonu) çağırır. Biz önce pcap_next fonksiyonunu inceleyelim. Fonksiyonun prototipi şöyledir:

```
const u_char *pcap_next(pcap_t *p, struct pcap_pkthdr *h);
```

Fonksiyonun birinci parametresi pcap_open_live fonksiyonundan elde edilen handle değeridir. İkinci parametre elde edilen paket hakkında bazı bilgilerin yerleştirileceği pcap_pkthdr türünden yapı adresidir. Bu yapı aşağıdaki gibi bildirilmiştir:

struct timeval	**ts** *time stamp*
bpf_u_int32	**caplen** *length of portion present*
bpf_u_int32	**len** *length this packet (off wire)*

Yapının ts elemanı paketin elde edildiği zaman bilgisini, caplen elemanı bize verilen paket verisinin miktarını (biz bu miktarın maksimum pcap_open_live fonksiyonunda belirtmiştik) len elemanı da paketin gerçek uzunluğunu belirtir. Fonksiyon başarı durumunda paket verilerinin yerleştirilmiş olduğu alanın adresiyle geri döner. Fonksiyon başarısız olabilir. (Örneğin zaman aşımı dolmuştur) bu durumda NULL adrese geri döner. Fonksiyon aşağıdaki gibi bir programla test edilebilir:

```
#include <stdio.h>
#include <stdlib.h>
#include <pcap.h>

int main()
{
    char errBuf[PCAP_ERRBUF_SIZE];
    char *dev;
    pcap_t *hpcap;
    u_char *pdat;
    struct pcap_pkthdr hdr;
    bpf_u_int32 i;
    pcap_if_t *pdev, *pdevs[1024];
    int devno;

    if (pcap_findalldevs(&pdev, errBuf) == -1) {
        fprintf("pcap_findalldevs: %s\n", errBuf);
        exit(EXIT_FAILURE);
    }

    for (i = 0; pdev != NULL; ++i) {
        printf("%d) Description: %s\n", i + 1, pdev->description);
        pdevs[i] = pdev;
        pdev = pdev->next;
    }
    printf("\nAygıt seciminiz:");
    scanf("%d", &devno);
    dev = pdevs[devno - 1]->name;

    if ((hpcap = pcap_open_live(dev, BUFSIZ, 1, 0, errBuf)) == NULL) {
        fprintf(stderr, "pcap_open_live: %s\n", errBuf);
        exit(EXIT_FAILURE);
    }

    for (;;) {
        if ((pdat = pcap_next(hpcap, &hdr)) == NULL) {
            fprintf(stderr, "pcap_next: cannot capture packet!..\n");
            exit(EXIT_FAILURE);
        }

        for (i = 0; i < hdr.caplen; ++i)
            printf("%02X%c", pdat[i], i % 16 == 15 ? '\n' : ' ');
        printf("\n\n");
    }

    pcap_close(hpcap);
    pcap_freealldevs(pdevs[0]);

    return 0;
}
```

Paketleri yakalamak için diğer bir fonksiyon pcap_loop fonksiyonudur. Bu fonksiyon kendi içerisinde döngü oluşturarak paket geldikçe bizim belirdeğimiz bir fonksiyonu çağırır. pcap_loop fonksiyonunun prototipi şöyledir:

```
int pcap_loop(pcap_t *p, int cnt, pcap_handler callback, u_char *user);
```

Fonksiyonun birinci parametresi pcap_open_live fonksiyonundan elde edilen handle değeridir. İkinci parametre kaç paket alındıktan sonra döngünün sonlandırılacağını belirtir. Burada negatif bir değer "sonsuza kadar" anlamına gelmektedir. Üçüncü parametre çağrılacak fonksiyonun (callback function) adresini alır. pcap_handler türü aşağıdaki gibi typedef edilmiştir:

```
typedef void (*pcap_handler)(u_char *user, const struct pcap_pkthdr *h, const u_char *bytes);
```

pcap_loop fonksiyonun son parametresi çağrılacak fonksiyona aktarılacak kullanıcı tanımlı bilgiyi belirtir. Bu parametre NULL geçilebilir. Fonksiyon toplam belirtilen paket sayısı bittiğinde dolaylı sonlanmışsa 0 değerine, hata durumunda -1 değerine ve pcap_breakloop dolayısıyla sonlanmışsa -2 değerine geri döner.

Çağrılacak fonksiyonun birinci parametresi pcap_loop fonksiyonuna girilen son argümandır. İkinci parametre yine gelen paketin bilgilerini içeren pcap_pkthdr türünden yapının adresini belirtir. Son parametre gerçek paket verileridir. Fonksiyonun örnek kullanımı şöyledir:

```
#include <stdio.h>
#include <stdlib.h>
#include <pcap.h>

void myhandler(u_char *user, const struct pcap_pkthdr *header, const u_char *pdat);

int main()
{
    char errBuf[PCAP_ERRBUF_SIZE];
    char *dev;
    pcap_t *hpcap;
    u_char *pdat;
    struct pcap_pkthdr hdr;
    bpf_u_int32 i;
    pcap_if_t *pdev, *pdevs[1024];
    int devno;

    if (pcap_findalldevs(&pdev, errBuf) == -1) {
        fprintf("pcap_findalldevs: %s\n", errBuf);
        exit(EXIT_FAILURE);
    }

    for (i = 0; pdev != NULL; ++i) {
        printf("%d) Description: %s\n", i + 1, pdev->description);
        pdevs[i] = pdev;
        pdev = pdev->next;
    }
    printf("\nAygıt seciminiz:");
    scanf("%d", &devno);
    dev = pdevs[devno - 1]->name;

    if ((hpcap = pcap_open_live(dev, BUFSIZ, 1, 0, errBuf)) == NULL) {
        fprintf(stderr, "pcap_open_live: %s\n", errBuf);
        exit(EXIT_FAILURE);
    }

    if (pcap_loop(hpcap, 100, myhandler, NULL) == -1) {
        fprintf(stderr, "pcap_loop error!\n");
        exit(EXIT_FAILURE);
    }

    pcap_close(hpcap);
    pcap_freealldevs(pdevs[0]);

    return 0;
}

void myhandler(u_char *user, const struct pcap_pkthdr *header, const u_char *pdat)
```

```
{
    int i;

    for (i = 0; i < header->caplen; ++i)
        printf("%02X%c", pdat[i], i % 16 == 15 ? '\n' : ' ');
    printf("\n\n");
}
```

İstersek pcap_loop fonksiyonunun oluşturduğu döngüyü çağrılan fonksiyondan (callback function) da kırabiliriz. Bunun için pcap_breakloop fonksiyonu kullanılmaktadır:

```
void pcap_breakloop(pcap_t *handle);
```

Örneğin pcap_loop fonksiyonunu sonsuz döngüye sokup içeriden pcap_breakloop ile aşağıdaki gibi çıkabiliriz:

```
#include <stdio.h>
#include <stdlib.h>
#include <pcap.h>

void myhandler(u_char *user, const struct pcap_pkthdr *header, const u_char *pdat);

int main()
{
    char errBuf[PCAP_ERRBUF_SIZE];
    char *dev;
    pcap_t *hpcap;
    u_char *pdat;
    struct pcap_pkthdr hdr;
    bpf_u_int32 i;
    pcap_if_t *pdev, *pdevs[1024];
    int devno;

    if (pcap_findalldevs(&pdev, errBuf) == -1) {
        fprintf("pcap_findalldevs: %s\n", errBuf);
        exit(EXIT_FAILURE);
    }

    for (i = 0; pdev != NULL; ++i) {
        printf("%d) Description: %s\n", i + 1, pdev->description);
        pdevs[i] = pdev;
        pdev = pdev->next;
    }
    printf("\nAygıt seciminiz:");
    scanf("%d", &devno);
    dev = pdevs[devno - 1]->name;

    if ((hpcap = pcap_open_live(dev, BUFSIZ, 1, 0, errBuf)) == NULL) {
        fprintf(stderr, "pcap_open_live: %s\n", errBuf);
        exit(EXIT_FAILURE);
    }

    if (pcap_loop(hpcap, -1, myhandler, (u_char *)hpcap) == -1) {
        fprintf(stderr, "pcap_loop error!\n");
        exit(EXIT_FAILURE);
    }

    pcap_close(hpcap);
    pcap_freealldevs(pdevs[0]);

    return 0;
}

void myhandler(u_char *user, const struct pcap_pkthdr *header, const u_char *pdat)
{
    int i;
```

```
    static int count = 0;
    pcap_t *hpcap = (pcap_t *)user;

    for (i = 0; i < header->caplen; ++i)
        printf("%02X%c", pdat[i], i % 16 == 15 ? '\n' : ' ');
    printf("\n\n");

    ++count;
    if (count == 100)
        pcap_breakloop(hpcap);
}
```

Filtreleme pcap kütüphanesinin en önemli özelliklerinden biridir. WireShark gibi pcap önyüz programları kullanıcıların filre oluşturmasına izin vermektedir. Filtre belli koşulu sağlayan paketlerin elde edilmesini amaçlar. Bunun için şüphesiz paket içerğinin analiz edilmesi gerekir. Aslında pcap kütüphanesi arka planda filtreleme işlemi için Berkeley Packet Filter isimli kütüphaneyi kullanmaktadır. Filtreleme için önce filtre yazısı belirlenip bir filtre derlemesi yapılır. Filtreleme işlemi şu adımlardan geçilerek yapılmaktadır:

1) Önce pcap_lookupnet fonksiyonu kullanılarak network aygıtı hakkında bazı bilgilr (özellikle ip adresi) elde edilmelidir. pcap_lookupdev fonksiyonun prototipi şöyledir:

```
int pcap_lookupnet (const char *device, bpf_u_int32 *netp, bpf_u_int32 *maskp, char *errbuf);
```

Fonksiyonun birinci parametresi aygıtın ismini alır. İkinci ve üçüncü parametreler aygırın ip adresiyle network mask değerinin yerleştirileceği nesnelerin adresini almaktadır. Son parametre yine hata durumunda hata mesajının yerleştirileceği tamponu belirtir. Fonksiyon başarı durumunda 0 değerine başarısızlık durumunda -1 değerine geri döner.

2) Aygıt pcap_open_live fonksiyonuyla açılır ve handle değeri elde edilir.

3) pcap_compile fonksiyonu ile filtre derleme işlemine sokulur. Bu fonksiyon filtrelemeyi işleme sokulacak bir veri yığınına dönüşürür. Fonksiyonun prototipi şöyledir:

```
int pcap_compile(pcap_t *p, struct bpf_program *fp,
           char *filter, int optimize, bpf_u_int32 netmask);
```

Fonksiyonun birinci parametresi bağlantıdan elde edilen handle değeridir. İkinci parametre bpf_program türündne bir yapının adresini alır. Üçüncü parametre filtre yazısını belirtmektedir. Dörüdüncü parametre işlemde optimizasyon yapılıp yapılmayacağını belirtir. Bu parametre 0 ya da 0 dışı bir değer (tipik olarak 1) biçiminde girilebilir. Son parametre pcap_lookupnet fonksiyonundna elde edilen networdk mask değeridir. Fonksiyon başarı durumunda 0, başarısızlık durumunda -1 değerine geri döner.

4) Derlenen filtre pcap_set_filter fonksiyonuyla set edilir. Fonksiyonun prototipi şöyledir.

```
int pcap_setfilter(pcap_t *handle, struct bpf_program *fp);
```

Fonskiyonun birinci parametresi açılmış aygıtın handle değerini ikinci parametresi pcap_compile elde edilen bpf_program yapı nesnesinin adresini alır. Fonksiyhon başarı durumunda 0, başarısızlık durumunda -1 değerine geri döner. Örnek bir filtreleme şöyle olabilir:

```
#include <stdio.h>
#include <stdlib.h>
#include <pcap.h>

void myhandler(u_char *user, const struct pcap_pkthdr *header, const u_char *pdat);

int main()
{
```

```
    char errBuf[PCAP_ERRBUF_SIZE];
    char *dev;
    pcap_t *hpcap;
    u_char *pdat;
    struct pcap_pkthdr hdr;
    bpf_u_int32 i;
    pcap_if_t *pdev, *pdevs[1024];
    int devno;
    bpf_u_int32 netp, maskp;
    struct bpf_program prog;

    if (pcap_findalldevs(&pdev, errBuf) == -1) {
        fprintf("pcap_findalldevs: %s\n", errBuf);
        exit(EXIT_FAILURE);
    }

    for (i = 0; pdev != NULL; ++i) {
        printf("%d) Description: %s\n", i + 1, pdev->description);
        pdevs[i] = pdev;
        pdev = pdev->next;
    }
    printf("\nAygıt seciminiz:");
    scanf("%d", &devno);
    dev = pdevs[devno - 1]->name;

    if (pcap_lookupnet(dev, &netp, &maskp, errBuf) == -1) {
        fprintf(stderr, "pcap_lookupnet: %s\n", errBuf);
        exit(EXIT_FAILURE);
    }

    if ((hpcap = pcap_open_live(dev, BUFSIZ, 1, 0, errBuf)) == NULL) {
        fprintf(stderr, "pcap_open_live: %s\n", errBuf);
        exit(EXIT_FAILURE);
    }

    if (pcap_compile(hpcap, &prog, "port 80", 1, maskp) == -1) {
        fprintf(stderr, "pcap_compile failed!\n");
        exit(EXIT_FAILURE);
    }

    if (pcap_setfilter(hpcap, &prog) == -1) {
        fprintf(stderr, "pcap_setfilter failed!\n");
        exit(EXIT_FAILURE);
    }

    if (pcap_loop(hpcap, 100, myhandler, NULL) == -1) {
        fprintf(stderr, "pcap_loop error!\n");
        exit(EXIT_FAILURE);
    }

    pcap_close(hpcap);
    pcap_freealldevs(pdevs[0]);

    return 0;
}

void myhandler(u_char *user, const struct pcap_pkthdr *header, const u_char *pdat)
{
    bpf_u_int32 i;

    for (i = 0; i < header->caplen; ++i)
        printf("%02X%c", pdat[i], i % 16 == 15 ? '\n' : ' ');
    printf("\n\n");
}
```

Pcap kütüphanesinin başka fonksiyonları da vardır. Bunlar kütüphanenin dokümanları incelenerek öğrenilebilir.

Linux Çekirdeğine Temel Bakış

Diğer işletim sistemlerinde de olduğu gibi Linux işletim sisteminin asıl motor kısmı çekirdeğidir. Aslında Linux bir çekirdek projesidir. Çekirdeğin dışındaki katmanlar ve uygulamalar tamamen başka proje grupları tarafından geliştirilmiş yazılımlardır. Linux dağıtımları aynı çekirdeğin üzerine değişik yazılımları giydirerek bize kolay kullanımlı bir sistem sunmaktadır. Başka bir deyişle Linux dağıtımlarında Linux çekirdeği değişmemekte fakat bunun dışındaki tüm yazılımlar ve paketler değişebilmektedir.

Bugün iki yüzün üzerinde Linux dağıtımı bulunmaktadır. Bazı dağıtımlar çok eskiden başlatılmıştır. Bazı dağıtımlar bazı dağıtımların üzerine kurulmuştur. Yani onlar temel alınarak oluşturulmuştur. En temel dağıtımlar ve türevleri şunlardır:

- Debian Dağıtımı: Debian pek çoklarına göre GNU projesini ve Linux felsefesini en iyi yansıtan dağıtımdır. Debian'ın kendisinin yanı sıra Debian türevi olan pek çok dağıtım oluşturulmuştur: Knoppix, Ubuntu, Mint bunlardan en fazla kullanılanıdır.

- Fedora ve Red Hat Dağıtımları: Red Hat firması tarafından geliştirilen Fedora da çok kullanılan bir temel dağıtımdır. Bu dağıtımı temel alan en önemli dağıtımlar "Red Hat Enterprise Linux (ticari bir dağıtım)" ve Fedora'nın adeta server versiyonu olan CentOS'tur. Bugün CentOS Linux server olarak en fazla tercih edilen sistemdir.

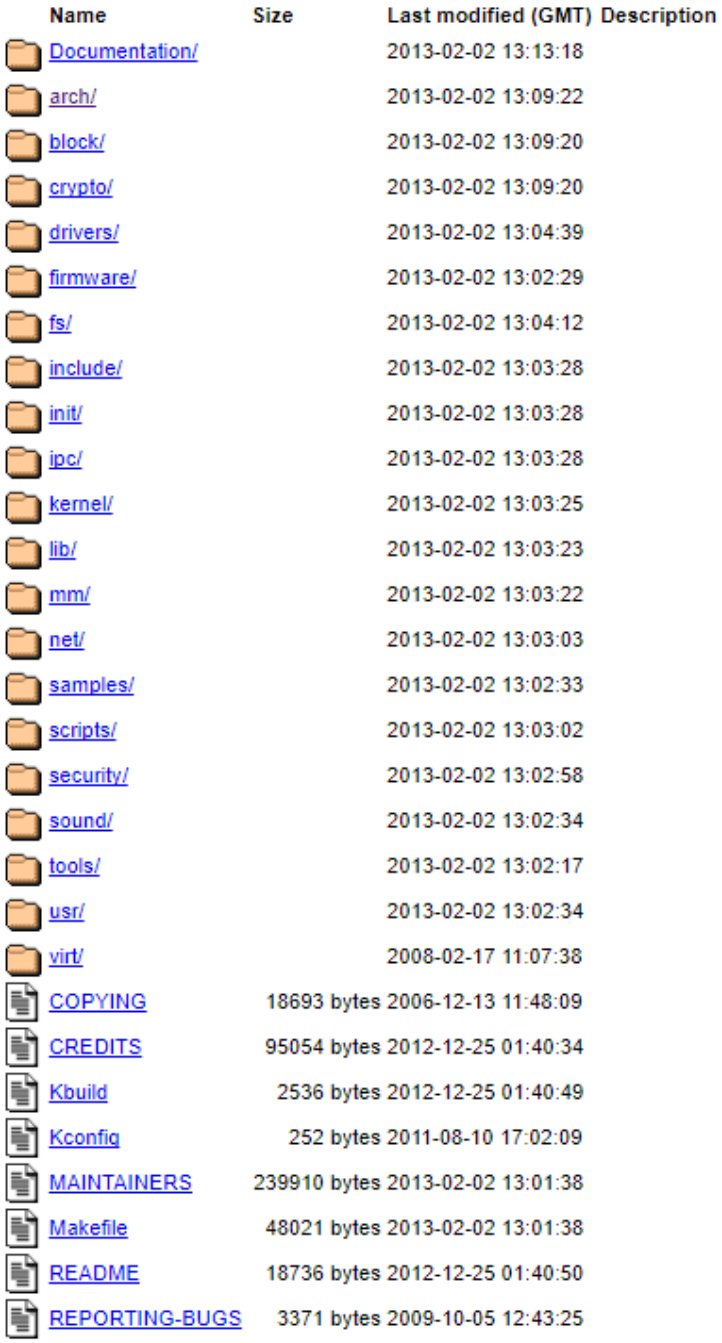
- Suse Dağıtımları: Suse isimli firma tarafından oluştrulmuş olan en eski dağıtımlardandır. Suse ticari bir dağıtımdır. Bu nedenle açık kaynak kod grupları tarafından genellikle dışlanmıştır. Ancak pek çok kullanıcısı vardır. Suse'nin bedava olan versiyonuna "Open Suse" denilmekedir.

Dağıtımlar arasındaki en önemli farklılıklardan ikisi kullanılan paket yöneticileri ve masaüstü pencere sistemidir. Bazı dağıtımlar biren fazla paket yöneticisini ve birden fazla masaüstü pencere sistemini kullanabilmektedir. Bazı dağıtımların yalnızca tek bir versiyonu vardır. Bunlar hem client hem de server olarak kullanılmaktadır. Bazı dağıtımların client ve/veya server versiyonları vardır.

Linux çekirdeklerine diğer yazılımlarda olduğu gibi birer numara verilmiştir. Numaralandırma biçimi belli bir tarihten sonra değiştirilmiştir. Eski sistemde versiyonlar arasında küçük artırımlar yapılıyordu. Gerçekten de önemli atlamaların olduğu versiyonlar şunlardır: Versiyon 2.0, 2.2, 2.4 ve 2.6. Yeni versiyonlamada artırımlar yüksek miktarda yapılmaktadır. Kursun yapıldığı tarihte çekirdeğin en büyük stabil versiyonu 4.13.2'dir. Bu yeni versiyonlar aslında eski sistemde 2.8.X'lere karşılık gelebilecek numaralardır.

Linux çekirdeklerinin resmi dağıtım yeri "kernel.org" sitesidir. Buradan istenilen sürüme ilişkin çekirdeğin kaynak kodları indirilip yerel makineye kopyalanabilir. Bu sitede daha ilk versiyonundan beri bütün çekirdek kodları bulunmaktadır. Linux çekirdeğinde değişiklik yapmak isteyen programcıların öncelikle Linux çekirdeğinin kaynak kodlarını indirmesi gerekir. Bazen programcılar yeni çıkmış bir çekirdeği henüz kullandıkları dağıtım kendini yenilemeden yüklemek isteyebilirler. Bu durumda bu siteden çekirdek kodlarını indirip derlerler.

Linux çekirdek kodlarının tipik organizasyonu aşağıdaki gibidir (2.6 çekirdeği esas alınmıştır):



Name	Size	Last modified (GMT)	Description
Documentation/		2013-02-02 13:13:18	
arch/		2013-02-02 13:09:22	
block/		2013-02-02 13:09:20	
crypto/		2013-02-02 13:09:20	
drivers/		2013-02-02 13:04:39	
firmware/		2013-02-02 13:02:29	
fs/		2013-02-02 13:04:12	
include/		2013-02-02 13:03:28	
init/		2013-02-02 13:03:28	
ipc/		2013-02-02 13:03:28	
kernel/		2013-02-02 13:03:25	
lib/		2013-02-02 13:03:23	
mm/		2013-02-02 13:03:22	
net/		2013-02-02 13:03:03	
samples/		2013-02-02 13:02:33	
scripts/		2013-02-02 13:03:02	
security/		2013-02-02 13:02:58	
sound/		2013-02-02 13:02:34	
tools/		2013-02-02 13:02:17	
usr/		2013-02-02 13:02:34	
virt/		2008-02-17 11:07:38	
COPYING	18693 bytes	2006-12-13 11:48:09	
CREDITS	95054 bytes	2012-12-25 01:40:34	
Kbuild	2536 bytes	2012-12-25 01:40:49	
Kconfig	252 bytes	2011-08-10 17:02:09	
MAINTAINERS	239910 bytes	2013-02-02 13:01:38	
Makefile	48021 bytes	2013-02-02 13:01:38	
README	18736 bytes	2012-12-25 01:40:50	
REPORTING-BUGS	3371 bytes	2009-10-05 12:43:25	

Kök dizindeki “Documentation” dizini önemli bazı dosyalar ve modüller hakkında bilgiler veren text dosyalar içermektedir. Bu dosyalardaki bilgiler pedagojik değildir. Ancak değerlidir.

“kernel” dizini çekirdeğin en aşağı seviyeli işlevlerini yerine getiren kaynak dosyaları bulundurmaktadır. Bu dizini çekirdeğin çekirdeği olarak yorumlayabiliriz. Bu dizinde ağırlıklı olarak proses yönetimine ve çizelgeleyici alt sisteme ilişkin kaynak kodlar bulunmaktadır.

Tüm çekirdek kodlarındaki bütün başlık dosyaları include dizinininin içerisindeki dizinlerdedir. Örneğin çekirdeğin önemli başlık dosyalarının hemen hepsi include/linux dizininin içerisinde bulunmaktadır.

Kökün altındaki fs dizini dosya sistemine ilişkin kodların kaynak dosyalarını tutmaktadır. Bu dizinin köküne dosya sisteminden bağımsız kodlar yerleştirilmiştir. Belli bir dosya sisteminin kodları ise o dosya sistemine ilişkin dizinin içerisinde bulunmaktadır.

Kök dizinin altındaki arch (architecture) dizini işlemciye bağlı sembolik makine dili kodlarının bulunduğu dizindir. Linux işletim sisteminin %97’si C kullanılarak (ancak standart C değil, gcc’deki eklentilerle birlikte)

kalan %3'lük kısmı da ilgili işlemcinin makine kodları kullanılarak yazılmıştır. İşte arch dizininde her işlemci ailesi için bir dizin vardır. Bu dizinin içerisinde o işlemciye ilişkin sembolik makine dili kodları bulunur.

Kök dizinin altındaki mm (memory management) dizininde ana belleği idare eden kodlar bulunmaktadır. Linux işletim sisteminin en karmaşık bölümlerinden biri de bellek yönetimidir.

Kök dizinin altındaki init dizininde çekirdeğin başlangıç kodları bulunur. Yani çekirdek dosyası yüklendikten sonra birtakım veri yapılarının ve alt sistemlerin ilklenmesi için gereken kodlar buradadır. Örneğin çekirdeğin giriş noktası (çekirdeğin main fonksiyonu diyebiliriz) start_kernel fonksiyonu buradaki main.c dosyası içerisindedir.

Kök dizini altındaki net dizini ağ (network) yönetimine ve protokollere ilişkin kodları barındırmaktadır.

Çekirdek Kodları Nasıl Hangi Durumlarda Çalışmaktadır?

Yalnızca Linux'ta değil pek çok işletim sisteminde çekirdek kodları tipik olarak şu durumlarda çalışma fırsatı elde etmektedir:

- Sistem ilk kez boot edilirken. İşletim sistemi boot işlemi sırasında yüklendiğinde çekirdek dosyası (kernel image) diskten alınarak belleğe yüklenir ve oradaki bir başlangıç noktasına atlanır (start_kernel fonksiyonu) burada birtakım ilk işlemler yapılmaktadır.

- Sistem kapatılırken: Sistem kapatılırken benzer biçimde işletim sisteminin kodları çalıştırılarak prosesler ve alt sistemler güvenli biçimde sonlandırılır.

- User moddaki normal prosesler sistem fonksiyonlarını çağırdığında: Biz programlama dillerinde çalışırken onların kütüphane fonksiyonları arka planda sistem fonksiyonlarını çağırabilmektedir. İşte bu işlem sırasında (system call) çekirdek

- Donanım kesmeleri oluştuğunda. Bir donanım kesmesi oluştuğunda akış o anda çalıştırılan koddan koparılarak "kesme kodu (interrupt handler)" denilen koda yönelir. İşte kesme kodları işletim sisteminin çekirdeği içerisinde bulunmaktadır.

- İşletim sisteminin de sanki normal proseslerin thread'leri gibi çizelgelenen yani arka planda çalışan thread'leri vardır. Bunlara "kernel thread" ya da "kernel deamon" denilmektedir. Yani thread'ler arası geçiş (context switch) gerçekleştiğinde yeni geçilen thread işletim sisteminin kendisinin bir thread'i de olabilir. Aygıt sürücüler de kernel thread oluşturabilmektedir.

Görüldüğü gibi işletim sisteminin kodları arka planda çoğu kez çalışmadan beklemektedir. Bir olay olduğunda (sistem fonksiyonu çağrıldığında, kesme oluştuğunda) devreye girmektedir. Ancak bazı kodlar (kernel thread'ler) periyodik olarak normal thread'ler gibi arka plan işlemleri yapmak için devreye de girmektedirler.

Çekirdek Modülleri (Kernel Modules) ve Aygıt Sürücüler (Device Drivers)

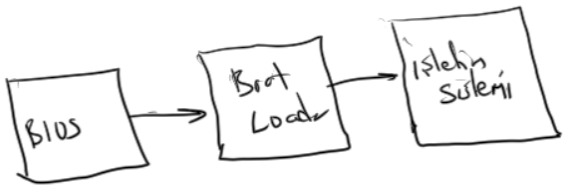
Linux'ta çekirdek modunda çalışan iki tür koddan bahsedilebilir: Çekirdek modülleri ve aygıt sürücüler. Bir çekirdek modülü çekirdek modunda (kernel mode'da) çalışmak üzere çekirdek alanına yerleştirilmiş olan kodlardan oluşur. Nasıl masaüstü bilgisayarımızın kart genişleme yuvasına yeni bir kart taktığımızda artık o kart donanımın bir parçası gibi oluyorsa aynı biçimde çekirdek modülleri de çekirdeğe takıldıklarında çekirdeğin bir parçası gibi, onunla aynı haklarda çalışabilmektedir. IO işlemleri yapan ve kesme kullanan çekirdek modüllerine Linux'ta aygıt sürücü denilmektedir. Her aygıt sürücüsü bir çekirdek modülüdür. Ancak her çekirdek modülü bir aygıt sürücü değildir. Çekirdek modülleri ve aygıt sürücüler çekirdek modunda çalışırlar. Buradaki fonksiyonların bir bölümü istenirse kullanıcı modundan da çağrılabilirler. Çekirdek

modunda çalışması gereken pek çok kod söz konusu olabilmektedir. Örneğin donanım aygıtlarını programlayan kodlar, kesmelere yanıt veren kodlar, sistem faaliyetlerini değiştiren ve izleyen kodlar gibi.

Linux Sisteminin Başlatılması

Bugün masaüstü bilgisayar işletimi sistemini otomatik olarak yükleyecek biçimde tasarlanmıştır. Genel boot süreci "Sistem Programlama ve İleri C Uygulamaları-1" numaralı kursta ele alınmıştı. Burada Linux'a özgü bazı durumlardan bahsedilecektir.

Bilgisayarı açtığımızda çalışma EEPROM içerisindeki koddan başlar. Buraya BIOS (Basic Input Output System) denilmektedir. Buradaki kod aktif boot sürücünün ilk sektörünü belleğe yükleyerek akışı ona devreder. Oradaki kod da işletim sisteminin yüklenmesi işlemini başlatmaktadır. UEFI BIOS'larda boot süreci daha ayrıntılı ve esnek olarak ayarlanmıştır. Fakat temel fikir aynıdır. Ancak makinelerde bireden fazla işletim sisteminin bulunabilmesi durumu nedeniyle işletim sisteminin yüklenmesi "boot loader" denilen programlara da devredilebilmektedir. Böylece aktif sürücünün ilk sektörüne "boot loader" denilen programı yükleyen program yerleştirilir. Bilgisayar açıldığında kontrolü "boot loader" ele almış olur. Boot loader bize hangi işletim sistemi ile bilgisayarımızı başlatacağımızı sorar. İşletim sisteminin yükleycisini bu "boot loader" yükler.



Açık kaynak kodlu ve mülkiyete sahip pek çok "boot loader" bulunmaktadır. Linux dünyasında iki önemli boot loader kullanılmaktadır: Lilo ve GRUB. Lilo eskiden çok yoğun kullanılıyordu. Ancak GRUB Lilo'ya göre çok gelişmiş seçenekler sunmaktadır. GRUB dosya sistemlerini tanıyarak çekirdek imajını (kernel image) diskten herhangi bir dizinden yükleyebilmektedir. Uzun süredir Linux sistemlerinde çekirdek imajı /boot dizininde bulundurulmaktadır. Lilo ve GRUB kendi konfigürasyon dosyalarını okuyarak bilgileri oradan alırlar. Kurulum programları ya da sistem yöneticileri de bu konfigürasyon dosyalarını oluşturmaktadır. Bu durumda çekirdek derlendikten sonra elde edilen çekirdek imajı ilgili dizine yerleştirilmeli ve eğer gerekiyorsa boot loader'ın konfigürasyon dosyalarında değişikilikler de yapılmalıdır. Bu işlemlerin daha zahmetsiz yapılması için bazı dağıtımlar bazı programlar ya da komutlar da bulundurabilmektedir. İşte bootloader konfigürasyon dosyasında belirtilen çekirdek imajını yükleyerek akışı ona devretmektedir.

Linux Çekirdek Modülleri (Kernel Modules)

Yukarıda da belirtildiği gibi çekirdek modülleri çekirdeğe kod eklemek için oluşturulmaktadır. Aygıt sürücüler de birer çekirdek modülüdür. Linux çekirdeği tüm modülleri kendi içerisinde bir veri yapısında toplamaktadır. Yani çekirdek yüklenen her modülün bilgilerini onlara gerektiğinde erişebilmek için kendi içerisinde tutar.

Linux sistemlerinde yüklenmiş olan modüller hakkındaki bilgiler çekirdek tarafından oluşturulmuş olan /proc/modules dosyasında ve /sys/module dizinindeki dosyalardan elde edilebilir.

Linux'ta çekirdek modülleri çekirdeğin bir parçası olarak da daha sonra dinamik olarak da yüklenebilmektedir. Dinamik modül yüklemesi ve boşaltması için insmod ve rmmod programlarından faydalanılmaktadır. Tabii bu komutların kullanılması için prosesin root önceliğinde olması gerekmektedir.

Merhaba Dünya Çekirdek Modülünün Yazımı ve Derlenmesi

Linux'ta çekirdek modüllerinin geliştirilmesi için öncelikle gerekli olan çekirdek kodları ve araçları sisteme yüklenmiş olmalıdır. Genel olarak bunun için yükleme yeri olarak "/lib/$(uname -r)" dizinidir. Çekirdek modüllerinin kullanacağı başlık dosyaları çoğu kez zaten gcc kurulumuyla birlikte (yani Linux'u

kurduğumuzda) default olarak "/usr/include/src/$(uname -r)/include" dizininde bulunmaktadır. Benzer biçimde yine Linux sistemlerini kurduğumuzda gcc ile birlikte çekirdek modüllerini geliştirmek için gereken kütüphaneler ve diğer dosyalar /lib/modules/$(uname -r)" dizininde hazır bulunmaktadır. Yani çekirdek modülü geliştirmek için artık sistemde gcc'nin bulunması yeterlidir. Zaten sözünü ettiğimiz bu dosyalar gcc yüklenirken sisteme yüklenmiş durumdadır.

Kullandığımız Linux çekirdeğinin versiyon numarası (aynı zamanda dizin ismi) "uname -r" komutuyla elde edilmektedir. Örneğin:

```
csd@csd-virtual-machine ~ $ uname -r
4.4.0-53-generic
```

Kabuk üzerinde $(komut) biçiminde bir kalıp kullanıldığında kabuk ilgili komutu çalıştırır. Komutun stdout dosyasına yazdıklarını $(komut) yerine yerleştirir. Böylece biz kabul üzerinde aşağıdaki gibi bir komut vermiş olalım:

```
cd /lib/modules/$(uname -r)
```

Burada /lib/modules dizininin altındaki uname -r komutunun çıktısı olarak verilen dizine geçmiş oluruz.

"Merhaba Dünya" çekirdek modül programı minimal olarak aşağıdaki gibi yazılabilir:

```
/* helloworld.c */

#include <linux/module.h>
#include <linux/kernel.h>

int init_module(void)
{
    printk(KERN_INFO "Hello World!..\n");

    return 0;
}

void cleanup_module(void)
{
    printk(KERN_INFO "Goodbye World!..\n");
}
```

Bu modülü build edebilmek için aşağıdaki gibi Makefile isimli bir Make dosyasının hazırlanması gerekir:

```
/* Makefile */

obj-m += helloworld.o

all:
    make -C /lib/modules/$(shell uname -r)/build M=$(PWD) modules
clean:
    make -C /lib/modules/$(shell uname -r)/build M=$(PWD) clean
```

Bu işlemlerden sonra ilgi dizinde make komutunu uyguladığımızda ürün olarak helloworld.ko isimli bir çekirdek modül dosyası elde ederiz. Daha sonra çekirdek modülü insmod komutuyla yüklenip rmmod komutuyla boşaltılabilir:

```
csd@csd-virtual-machine ~/Study/SysProg2-2016/KernelModule $ sudo insmod helloworld.ko
csd@csd-virtual-machine ~/Study/SysProg2-2016/KernelModule $ sudo rmmod helloworld.ko
```

Öncelikle modülün nasıl build edildiği (yani Makefile üzerinde) üzerinde duralım. Oluşturduğumuz bu make dosyası aslında "/lib/modules/$(shell uname -r)/build" dizinindeki make dosyasını çalıştırmaktadır. Yani

aslında çekirdek modülünü build eden hazır bir Makefile dosyası söz konusu dizinde bulunmaktadır. Oradaki hazır make dosyası bizim hazırladığımız make dosyasının başındaki yönergeyi okur ve hangi dosyaların make işlemine dahil edileceğini buradan alır. Aslında bizim yazdığımız make dosyasının en önemli kısmı şudur:

```
obj-m += helloworld.o
```

Bu kısım çekirdek modülünü oluşturan dosyanın helloworld.c olduğunu bunun derlenip helloworld.ko yapılacağını belirtir. Çekirdek modülü birden fazla kaynak dosya olarak da yazılabilir. Bu durumda yukarıdaki satıra başka satırlar da eklenecektir. O halde yukarıdaki make dosyası çekirdek modülleri için her zaman aynı biçimde bulundurulur. Yalnızca dosyanın başındaki kısım derlenecek kaynak dosyalara göre farklı olacaktır. Pekiyi neden tüm make dosyasını biz yazmıyoruz da zaten yazılmış olan "/lib/modules/$(shell uname -r)/build" dizinindeki make dosyasından faydalanıyoruz? İşte bunun nedeni işlemi basitleştirmektir. Çekirdek modülleri ELF formatına ilişkindir. Ancak formatın bazı bölümleri özel düzenlenmektedir. Ayrıca bağlama derleme ve bağlama işleminde pek çok dosyanın devreye sokulması gerekmektedir.

Aslında biz yukarıdaki Makefile dosyasını parametrik olarak da yazabilirdik:

```
obj-m += $(file).o

all:
    make -C /lib/modules/$(shell uname -r)/build M=$(PWD) modules
clean:
    make -C /lib/modules/$(shell uname -r)/build M=$(PWD) clean
```

Tabii bu durumda bu dosyayı make yaparken file isimli parametreyi de girmemiz gerekir:

```
kaan@kaan-VirtualBox ~/Study/DeviceDrivers $ make file=helloworld
make -C /lib/modules/4.10.0-38-generic/build M=/home/kaan/Study/DeviceDrivers modules
make[1]: Entering directory '/usr/src/linux-headers-4.10.0-38-generic'
  Building modules, stage 2.
  MODPOST 1 modules
make[1]: Leaving directory '/usr/src/linux-headers-4.10.0-38-generic'
```

Örneğimizdeki "helloworld.c" dosyasında iki fonksiyon olduğunu görüyorsunuz. Bunlar init_module ve cleanup_module fonksiyonlarıdır. Bir çekirdek modülü insmod programıyla yüklenirken init_module fonksiyonu çağrılır. Bu fonksiyonu biz bir sınıfın başlangıç fonksiyonu (constructor) gibi düşünebiliriz. Benzer biçimde modül rmmod programıyla boşaltılırken de cleanup_module fonksiyonu çağrılmaktadır. Bu fonksiyon da adeta bir sınıfın bitiş fonksiyonu (destructor) gibi düşünülebilir. Bir çekirdek modülü yüklendiğinde bazı ilk işlemlerin yapılması gerekebilmektedir. İşte bu ilk işlemler init_module fonksiyonunda yapılırlar. Benzer biçimde init_module fonksiyonunda yapılan işlemler cleanup_module fonksiyonunda geri alınabilirler.

Örnek programımızda iki başlık dosyasının include edildiğini görüyorsunuz. "linux/module.h" dosyası pek çok fonksiyonun prototipinin ve önemli makroların ve sembolik sabitlerin bulunduğu en temel başlık dosyasıdır. "linux/kernel.h" dosyası ise bazı durumlarda gerekmektedir. Örneğimizde printk fonksiyonun içerisindeki KERN_INFO makrosu bu başlık dosyasının içerisindedir.

Örnek kodumuzda init_module fonksiyonunda aşağıdaki printk çağrısı yapılmıştır:

```
printk(KERN_INFO "Hello World!..\n");
```

printk fonksiyonu kullanım olarak printf fonksiyonuna benzemekle birlikte aslında alakasız bir fonksiyondur. Biz çekirdek modüllerinde standart C kütüphanesini (özellikle IO kütüphanesini) kullanamayız. Yalnızca çekirdek içerisinde kullanımımıza izin verilmiş olan fonksiyonları kullanabiliriz. printk (print kernel) fonksiyonu çekirdeğin içerisinde bulunan yazdırma yapan bir fonksiyondur. Pekiyi printk yazdırma işlemini nereye yapar? İşte bu fonksiyon başındaki makroya bağlı olarak yazdırma işlemini fiziksel konsola aynı zamanda da bir log dosyasına yapmaktadır. (XWindow sistemlerindeki konsolun fiziksel konsol olmadığını

anımsatalım). printk fonksiyonu eski sistemlerde "/var/log/messages" isimli log dosyasına yazıyordu. Artık bir süredir. "/var/log/syslog" dosyasına yazma yapmaktadır. Bu text dosyalar zamanla büyüdüğü için bunların sonlarına "tail" komutu ile bakabiliriz. Örneğin:

```
csd@csd-virtual-machine ~/Study/SysProg2-2016/KernelModule $ sudo insmod helloworld.ko
[sudo] password for csd:
csd@csd-virtual-machine ~/Study/SysProg2-2016/KernelModule $ sudo rmmod helloworld.ko
csd@csd-virtual-machine ~/Study/SysProg2-2016/KernelModule $ tail /var/log/syslog
Sep 25 21:51:41 csd-virtual-machine anacron[838]: Job `cron.daily' terminated
Sep 25 21:51:41 csd-virtual-machine anacron[838]: Normal exit (1 job run)
Sep 25 21:52:21 csd-virtual-machine systemd[1787]: Time has been changed
Sep 25 21:52:21 csd-virtual-machine systemd[1]: Time has been changed
Sep 25 21:52:21 csd-virtual-machine systemd[1]: apt-daily.timer: Adding 2h 55min 5.07939
1s random time.
Sep 25 21:53:54 csd-virtual-machine kernel: [  444.616305] helloworld: module license 'u
nspecified' taints kernel.
Sep 25 21:53:54 csd-virtual-machine kernel: [  444.616308] Disabling lock debugging due
to kernel taint
Sep 25 21:53:54 csd-virtual-machine kernel: [  444.616335] helloworld: module verificati
on failed: signature and/or required key missing - tainting kernel
Sep 25 21:53:54 csd-virtual-machine kernel: [  444.616764] Hello World!..
Sep 25 21:54:09 csd-virtual-machine kernel: [  458.969175] Goodbye World!..
```

printk fonksiyonun yazdırdığı mesajlar dmesg isimli komut tarafından da görüntülenebilmektedir. Örneğin:

```
kaan@kaan-VirtualBox ~/Study/DeviceDrivers $ sudo insmod generic.ko
kaan@kaan-VirtualBox ~/Study/DeviceDrivers $ sudo rmmod generic.ko
kaan@kaan-VirtualBox ~/Study/DeviceDrivers $ dmesg
[ 4919.110176] Hello World..
[ 4932.878028] Goodbye World!..
```

dmesg ike çekirdeğin ring tamponunu boşaltmak için -C ya da --clear seçeneği kullanılabilir. Ayrıca dmesg başka bir terminalde -w biçiminde gerçek zamanlı olarak da çalıştırılabilmektedir.

Pekiyi kodumuzdaki KERN_INFO makrosu sentaks ve semantik olarak ne anlama gelmektedir. Bu makrolara "log düzeyi (log level)" denilmektedir. Aslında log düzeyi belirten KERN_XXX biçimindeki makrolar yalnızca iki tırnak içerisinde bir sayıyı koda eklemektedir:

```
#define KERN_SOH        "\001"          /* ASCII Start Of Header */
#define KERN_SOH_ASCII  '\001'

#define KERN_EMERG      KERN_SOH ""    /* system is unusable */
#define KERN_ALERT      KERN_SOH "1"    /* action must be taken immediately */
#define KERN_CRIT       KERN_SOH "2"    /* critical conditions */
#define KERN_ERR        KERN_SOH "3"    /* error conditions */
#define KERN_WARNING    KERN_SOH "4"    /* warning conditions */
#define KERN_NOTICE     KERN_SOH "5"    /* normal but significant condition */
#define KERN_INFO       KERN_SOH "6"    /* informational */
#define KERN_DEBUG      KERN_SOH "7"    /* debug-level messages */
```

Bu durumda aslında:

```
printk(KERN_INFO "Hello World!..\n");
```

çağırısı ile:

```
printk("\0016Hello World!..\n");
```

eşdeğerdir. ('\001' karakterinin gülen adam olduğunu anımsayınız).

Aslında printk fonksiyonu başında log düzeyi makrosu olmadan da kullanılabilir. Bu durumda belli bir log düzeyi default olarak kullanılmaktadır. Default log düzeyinin ne olacağı çekirdeğin CONFIG_DEFAULT_MESSAGE_LOGLEVEL konfigürasyon parametresiyle belirlenmektedir. Biz proc

dosya sistemindeki /proc/sys/kernel/printk dosyası bize printk için default log düzeylerini vermektedir. Örneğin:

```
kaan@kaan-VirtualBox ~/Study/DeviceDrivers $ cat /proc/sys/kernel/printk
4       4       1       7
```

Buradaki birinci değer printk için geçerli (current) log düzeyini, ikinci değer ise default log düzeyini belirtmektedir. Ayrıca <linux/kernel.h> dosyası içerisinde her log düzeyi için şu makrolar da tanımlanmıştır:

```
pr_emerg
pr_alert
pr_crit
pr_err
pr_warning
pr_notice
pr_info
pr_debug
```

Yani örneğin:

```
printk(KERN_INFO "Hello World!..\n");
```

çağrısı:

```
pr_info(KERN_INFO "Hello World!..\n");
```

ile eşdeğerdir.

Aslında çekirdek modülü yüklendiğinde ve boşaltıldığında çağrılacak fonksiyonların isimleri değiştirilebilir. Bunun için module_init ve module_exit makroları kullanılır. Bu makrolar kaynak kodun herhangi bir yerine yerleştirilebilir. Tipik olarak çekirdek modül programcıları bu makroları kaynak kodun sonuna yerleştirmektedir. Tabii bu makrolar kullanılmamışsa default fonksiyonlar init_module ve cleanup_module isminde olmak zorundadır. Default durumda bu isimdeki fonksiyonların "dışsal bağlama (external linkage)" özellğine sahip olması (yani static olmaması) gerekir. Ancak biz module_init ve module_exit makrolarını kullanacaksak böyle bir zorunluluk yoktur. Hatta bu durumda fonksiyonun "içsel bağlama (internal linkage)" özelliğine sahip olması (yani static yapılması) daha iyi bir tekniktir. O halde helloworld modülümüze ilişkin dosya şöyle de olabilir:

```
/* helloworld.c */

#include <linux/module.h>
#include <linux/kernel.h>

static int helloworld_init(void)
{
    printk(KERN_INFO "Hello World!..\n");

    return 0;
}

static void helloworld_exit(void)
{
    printk(KERN_INFO "Goodbye World!..\n");
}

module_init(helloworld_init);
module_exit(helloworld_exit);
```

Ayrıca modülün init ve cleanup fonksiyonlarının başına sırasıyla __init ve __exit makrolarının getirilmesi durumuyla sık karşılaşılmaktadır. Bu makrolar eğer modül dinamik olarak yüklenmişse bir etkiye sahip değillerdir. Ancak statik yüklemede (yani çekirdeğin bir parçası olarak başlangıçta yüklemede) bu makrolar "bu fonksiyonlar çağrıldıktan sonra artık koddan atılabilirler" anlamına gelmektedir. Tipik bu makrolar ilgili fonksiyonları bazı bölümlere (section) yerleştirir. Yükleme bitince ve modül boşaltılınca bu bölümler bellekten atılırlar. Makroların tanımlanması şöyle yapılmıştır:

```
42 /* These are for everybody (although not all archs will actually
43    discard it in modules) */
44 #define __init          __section(.init.text) __cold notrace
45 #define __initdata      __section(.init.data)
46 #define __initconst     __constsection(.init.rodata)
47 #define __exitdata      __section(.exit.data)
48 #define __exit_call     __used __section(.exitcall.exit)
```

Buradaki __section makrosu da şöyle tanımlanmıştır:

```
43 #ifndef __KERNEL__
44 #ifndef __section
45 # define __section(S) __attribute__ ((__section__(#S)))
46 #endif
```

Bu makroların kullanımı çalışma için zaten önemli değildir. Ancak bulundurulması toplamda daha anlamlıdır. __init ve __exit makroları <linux/init.h> dosyası içerisinde tanımlanmıştır. O halde helloworld modül programımız şöyle de olabilir:

```
/* helloworld.c */

#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/init.h>

static int __init helloworld_init(void)
{
    printk(KERN_INFO "Hello World!..\n");

    return 0;
}

static void __exit helloworld_exit(void)
{
    printk(KERN_INFO "Goodbye World!..\n");
}

module_init(helloworld_init);
module_exit(helloworld_exit);
```

Bu makrolar tür ifadesinden önceye de yerleştirilebilir.

Çekirdek Modüllerine Argüman Geçirmek

Nasıl normal programlara komut satırı argümanları geçirebiliyorsak çekirdek modüllerine de komut satırı argümanları geçirebiliriz. Komut satırı agümanları insmod işlemi sırasında modül isminin sağına değişken=değer biçiminde geçirilir. Modül argümanları modülün içerisinde module_param isimli makrolar yoluyla elde edilmektedir. Bu makrolar koda (tipik olarak kaynak kodun tepesine) yerleştirildiğinde komut satırı argümanları bu makrolarda belirtilen değişkenlerin içerisine yerleştirilirler. Biz burada module_param makrosunun nasıl yazılmış olduğunu ele almayacağız. module_param makrosu üç parametre almaktadır:

```
module_param(değişken_ismi, tür, erişim_hakları);
```

Makronun birinci parametresi ilgili komut satırı argümanının yerleştirileceği değişkenin ismini alır. İkinci parametresi bu değişkenin türünü belirtmektedir. Türler tipik olarak şunlardan biri olabilir:

```
int
long
short
uint
ulong
ushort
charp
bool
invbool
```

Yazılar (stringler) tipik olarak charp türüyle temsil edilirler. Bu durumda makronun birinci parametresi char türden bir gösterici olmalıdır. Makronun son parametresi ilgili parametreyle ilgili sysfs dosya sisteminde oluşturulacak dosyanın erişim erişim belirtir. Her modül parametresi için /sys/module/<modül ismi>/parameters dizini içerisinde parametre ismi ile bir dosya yaratılmaktadır. Bu parametre tipik olarak S_IRUSR|S_IWUSR|S_RGRP|S_IROTH biçiminde girilebilir. Eğer bu parametre 0 olarak girilirse bu durum böyle bir dosyanın hiç yaratılmayacağı anlamına gelir.

Modül parametreleri dizisel bir biçimde de olabilir. Bu durumda da module_param_array makrosu kullanılmaktadır. Bu makro dört parametre almaktadır:

```
module_param_array(değişken_ismi, tür, adres, erişim_hakları);
```

Makronun birinci parametresi değerlerin yerleştirileceği dizinin ismini, ikinci parametresi dizinin türünü, üçüncü parametresi diziye geçirilen parametrelerin sayısının yerleştirileceği nesnenin adresini, dördüncü parametresi de sysfs dosya sisteminde yaratılacak dosyanın erişim haklarını almaktadır. Dizisel parametre aktarımı <dizi ismi>=değer1,değer2,değer3, ... biçiminde yapılmalıdır. module_param ve module_param_array makroları <linux/moduleparam.h> dosyası içerisindedir. Ancak güncel sürümlerde <module.h> dosyası zaten bu dosyayı da include etmektedir. Örneğin:

```
/* generic.c */

#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/init.h>
#include <linux/moduleparam.h>

static int number = 123;
static int values[5];
static char *msg;
static int count;

module_param(number, int, S_IRUSR|S_IWUSR|S_IRGRP|S_IROTH);
module_param(msg, charp, S_IRUSR|S_IWUSR|S_IRGRP|S_IROTH);
module_param_array(values, int, &count, S_IRUSR|S_IWUSR|S_IRGRP|S_IROTH);

static int __init generic_init(void)
{
    int i;

    printk(KERN_INFO "Kernel number param: %d\n", number);
    printk(KERN_INFO "Kernel msg param: %s\n", msg);
    printk(KERN_INFO "Kernel values param:\n");

    for (i = 0; i < count; ++i)
        printk(KERN_INFO "%d\n", values[i]);
```

```
    return 0;
}

static void __exit generic_exit(void)
{
    printk(KERN_INFO "Goodbye World!..\n");
}

module_init(generic_init);
module_exit(generic_exit);
```

Şimdi biz modülü yüklerken bu number ve values değişkenleri için komut satırı argümanı girebiliriz:

```
kaan@kaan-VirtualBox ~/Study/DeviceDrivers $ sudo insmod generic.ko number=10 msg="\"this is a test\"" values=100,200,300,400,500
kaan@kaan-VirtualBox ~/Study/DeviceDrivers $ sudo rmmod generic.ko
kaan@kaan-VirtualBox ~/Study/DeviceDrivers $ dmesg
[ 1693.968737] Goodbye World!..
[ 1918.221375] Kernel number param: 10
[ 1918.221376] Kernel msg param: this is a test
[ 1918.221376] Kernel values param:
[ 1918.221377] 100
[ 1918.221377] 200
[ 1918.221377] 300
[ 1918.221378] 400
[ 1918.221378] 500
[ 1937.187805] Goodbye World!..
```

Burada dikkat edilmesi gereken nokta insmod kullanılırken “değişken=değer” kısmında değişken isminin modüldeki değişkenle aynı olmasıdır. Aksi takdirde bir hata mesajı oluşmaz ancak argüman ilgili değişkene yerleştirilmez.

Biz modülü yüklerken ilgili değişken için argüman girmezsek o makro o değişkene yerleştirme yapamayacağından dolayı o değişkende ona verilen ilkdeğerin kalacağına dikkat ediniz.

Çekirdek Fonksiyonlarının Hata Kodları

Çekirdek fonksiyonlarının büyük bölümünün geri dönüş değerleri int türdendir. Bu fonksiyonlar başarı durumunda 0 değerine, başarısızlık durumunda negatif hata kodlarına geri dönmektedir. Çekirdek düzeyinde bir errno değişkeni yoktur. errno değişkeni glibc kütüphanesi tarafından tamamen kullanıcı modunda (user mode) oluşturulmaktadır. Dolayısıyla biz çekirdek fonksiyonlarında hatanın nedenini ilgili fonksiyonların geri dönüş değerlerinden elde ederiz. Hata kodlarının pozitif halleri EXXX biçiminde sembolik sabitlerle define edilmiştir:

```
#ifndef _ASM_GENERIC_ERRNO_BASE_H
#define _ASM_GENERIC_ERRNO_BASE_H

#define EPERM            1      /* Operation not permitted */
#define ENOENT           2      /* No such file or directory */
#define ESRCH            3      /* No such process */
#define EINTR            4      /* Interrupted system call */
#define EIO              5      /* I/O error */
#define ENXIO            6      /* No such device or address */
#define E2BIG            7      /* Argument list too long */
#define ENOEXEC          8      /* Exec format error */
#define EBADF            9      /* Bad file number */
#define ECHILD          10      /* No child processes */
#define EAGAIN          11      /* Try again */
#define ENOMEM          12      /* Out of memory */
#define EACCES          13      /* Permission denied */
#define EFAULT          14      /* Bad address */
#define ENOTBLK         15      /* Block device required */
#define EBUSY           16      /* Device or resource busy */
#define EEXIST          17      /* File exists */
#define EXDEV           18      /* Cross-device link */
#define ENODEV          19      /* No such device */
#define ENOTDIR         20      /* Not a directory */
#define EISDIR          21      /* Is a directory */
#define EINVAL          22      /* Invalid argument */
#define ENFILE          23      /* File table overflow */
#define EMFILE          24      /* Too many open files */
#define ENOTTY          25      /* Not a typewriter */
#define ETXTBSY         26      /* Text file busy */
#define EFBIG           27      /* File too large */
#define ENOSPC          28      /* No space left on device */
#define ESPIPE          29      /* Illegal seek */
#define EROFS           30      /* Read-only file system */
#define EMLINK          31      /* Too many links */
#define EPIPE           32      /* Broken pipe */
#define EDOM            33      /* Math argument out of domain of func */
#define ERANGE          34      /* Math result not representable */

#endif
```

Çekirdek kodlarında aşağıdakine benzer pek çok kodla karşılaşabilirsiniz:

```
if (something_wrong)
    return -EBADF;
```

Oysa anımsanacağı gibi POSIX standartlarında genellikle fonksiyon başarılıysa 0 değerine başarısızsa -1 değerine geri dönmektedir. Hata nedeni de errno değişkenin içerisinden pozitif bir değer olarak alınmaktadır.

Aygıt Sürücü Çeşitleri

Bir çekirdek modülünün kullanıcı modundan (user mode) kullanılabilmesi için onun bir aygıt dosyası ile ilişkilendirilmesi gerekir. Böylece çekirdek modülü artık bir aygıt sürücü kimliğine kavuşur. Aygıt sürücüler kullanıcı modundan dosya gibi açılarak kullanılırlar. Yani onlar open POSIX fonksiyonuyla açılırlar, close fonksiyonuyla kapatılırlar. Aygıt sürücü üzerinde read, write ve lseek gibi dosya işlemleri yapılabilmekte ve istendiğinde de ioctl isimli bir POSIX fonksiyonuyla aygıt sürücü içerisindeki belli fonksiyonlar kullanıcı modundan çağrılabilmektedir. Tabii aygıt sürücü üzerindeki read, write, lseek ve ioctl gibi işlemler çekirdek moduna geçilerek gerçekleştirilirler. Aslında aygıt sürücüler üzerinde dosya fonksiyonları çağrıldığında sonraki konuda ele alınacağı gibi aygıt sürücünün belirlenen bazı fonksiyonları çağrılmaktadır. Böylece biz aygıt sürücülere kullanıcı modundan istediğimiz işlemleri yaptırabilmekteyiz.

Aygıt sürücüler "karakter aygıt sürücüleri (character device driver)" ve "blok aygıt sürücüleri (block device device driver)" olmak üzere ikiye ayrılmaktadır. Karakter aygıt sürücüleri byte byte aktarım yapılan sürücülerdir. Blok aygıt sürücüleri ise bloklu aktarıma izin vermektedir. Biz burada önce karakter aygıt sürücülerini göreceğiz.

Aygıt Dosyaları

Aygıt sürücülerin birer dosyaymış gibi kullanıldığını söyledik. Pekiyi bu dosyalar nasıl oluşturulmaktadır ve nerede bulunmaktadır? İşte aygıt sürücüleri açmak için kullanılan dosyalara "aygıt dosyaları (device files)" denilmektedir. Aygıt dosyaları geleneksel olarak kökün altındaki "dev" dizininde bulunurlar. Aygıt dosyaları bir dizin girişine ve bir i-node elemanına sahiptir. Ancak bunlar diskte bir dosyayı işaret etmezler. Bir aygıt sürücüyü (yani çekirdek modülünü) işaret ederler. İşletim sistemi open fonksiyonuyla aygıt sürücü dosyası

açılmak istendiğinde bunun bir disk dosyası olmadığını bir aygıt sürücüye (çekirdek modülüne) ilişkin olduğunu anlar ve bu aygıt sürücünün fonksiyonlarını çalıştırır. Aygıt sürücüler "ls -l" komutunda dosya türü olarak 'c' ve 'b' harfleriyle temsil edilmektedir. Buradaki 'c' harfi karakter aygıt sürücüsünü için 'b' harfi blok aygıt sürücüsünü belirtmektedir. Örneğin:

```
brw-rw----  1 root disk      1,   7 Eyl 25 21:46 ram7
brw-rw----  1 root disk      1,   8 Eyl 25 21:46 ram8
brw-rw----  1 root disk      1,   9 Eyl 25 21:46 ram9
crw-rw-rw-  1 root root      1,   8 Eyl 25 21:46 random
```

Aygıt dosyalarının normal disk dosyalarından farklı olarak birer aygıt numarası vardır. Aygıt numaraları iki parçaya ayrılmaktadır: "Büyük (major)" ve "küçük (minor)" numara. Örneğin Yukarıdaki /dev dizininden alınan görüntüde random aygıt dosyası bir karakter aygıt sürücüsünü belirtmektedir. Bunun büyük numarası 1, küçük numarası 8'dir. Aygıt dosyası olmayan dosyaların böyle büyük ve küçük aygıt numaraları yoktur. Genel olarak büyük numara aygıt sürücüsünü (yani driver'ı), küçük numara ise aygıtı (device) belirtmektedir. Bir aygıt sürücüsü benzer pek çok aygıtı yönetmek üzere yazılmış olabilir. İşte büyük numara aygıt sürücünün kendisini, küçük numara ise aygıt sürücünün yönettiği belirli bir aygıtı temsil etmektedir.

Tipik olarak sistem programcısı aygıt sürücüsünü oluştururken ona bir büyük numara ve bir ya da birden fazla küçük numara atar. Daha sonra kullanıcı modunda onunla ilişki kurmak için de büyük numarası ve küçük numarası onunla uyuşan bir aygıt dosyası yaratır. Artık bu aygıt dosyası yol ifadesi verilerek open fonksiyonula açıldığında ilgili aygıt sürücü ile bağlantı kurulmuş olur. Yani bizim install edeceğimiz her bir aygıt sürücüsü için en az bir aygıt dosyasına gereksinimimiz vardır.

Pekiyi aygıt dosyaları nasıl oluşturulmaktadır? Aygıt dosyaları aslında mknod isimli bir POSIX fonksiyonuyla oluşturulmaktadır. Bu da aslında arka planda bir sistem fonksiyonunu çağırmaktadır:

```
#include <sys/stat.h>

int mknod(const char *path, mode_t mode, dev_t dev);
```

Fonksiyonun birinci parametresi yaratılacak aygıt dosyasının yol ifadesini, ikinci parametresi erişim haklarını, üçüncü parametresi de onun büyük ve küçük aygıt numarasını belirtir. Fonskiyon başarı durumunda sıfır değerine başarısızlık durumunda -1 değerine geri döner. Fonksiyon eğer çağıran prosesin root önceliği yoksa başarısız olur. Ancak aygıt dosyalarını bu fonksiyonla yaratmak yerine mknod isimli kabuk komutuyla da yaratabiliriz. Zaten bu kabuk komutunun yaptığı tek şey mknod isimli POSIX fonksiyonunu çağırmaktır. mknod komutu tipik olarak aşağıdaki gibi kullanılır:

```
sudo mknod generic c 7 10
```

Burada aygıt dosyası generic isminde yaratılacaktır. İsimden sonraki 'c' harfi aygıtın "karakter aygıt sürücüsüne" ilişkin olduğunu belirtir. Eğer aygıtın blok aygıt sürücüsüne ilişkin olması istenseydi burada 'b' harfi kullanılmalıydı. Daha sonraki 7 büyük numarayı, 10 ise küçük numarayı belirtmektedir. İşlem için root hakkının olması gerekir. Komut varsayılan durumda dosyayı "rw- r-- r--" modunda yaratmaktadır. Ancak istenirse -m seçeneği ile erişim hakları istenildiği gibi belirtilebilir:

```
sudo mknod -m=666 generic c 7 10
```

Burada işlemlerin sırasına dikkat ediniz. Programcı tipik olarak büyük ve küçük numarayı aygıt sürücü kodu içerisinde o anda boş olan numaralardan dinamik olarak seçer. Sonra da bu mknod komutuyla bu numaralara ilişkin aygıt dosyasını oluşturur. Tabii programcı eğer bir numaranın zaten boş olduğunu biliyorsa önce dosyayı yaratıp sonra aygıt sürücüsünün bu numarayı kullanmasını da sağlayabilir.

Aygıt dosyalarının büyük ve küçük numaraları dev_t türüyle temsil edilmektedir. Bu dev_t türü güncel Linux çekirdeklerinde 32 bitlik işaretsiz tamsayı türü (unsigned int) biçiminde typedef edilmiştir. dev_t türünün

içerisinden büyük numarayı almak için MAJOR, küçük numarayı almak için MINOR isimli makrolar bulunmaktadır. Büyük ve küçük numaralardan dev_t oluşturmak için de MKDEV makrosu kullanılır. Tüm bu makrolar <linux/kdev_t.h> başlık dosyasında bildirilmiştir:

```
#ifndef _LINUX_KDEV_T_H
#define _LINUX_KDEV_T_H

#include <uapi/linux/kdev_t.h>

#define MINORBITS       20
#define MINORMASK       ((1U << MINORBITS) - 1)

#define MAJOR(dev)      ((unsigned int) ((dev) >> MINORBITS))
#define MINOR(dev)      ((unsigned int) ((dev) & MINORMASK))
#define MKDEV(ma,mi)    (((ma) << MINORBITS) | (mi))
```

Karakter Aygıt Sürücülerinin Yazımı

Karakter aygıt sürücüleri blok aygıt sürücülerine göre daha kolay gerçekleştirilebilmektedir. Karakter aygıt sürücülerinde bloklu okuma yazma olmadığı için bunlara yönelik bir cache sistemi de bulunmamaktadır. Pek çok tipik aygıt sürücüsü karakter aygıt sürücüsü biçiminde gerçekleştirilmiştir. Öte yandan blok aygıt sürücüleri cache sistemi kullanırlar. Örneğin disk aygıt sürücüleri tipik blok aygıt sürücülerini oluşturmaktadır.

Karakter aygıt sürücülerini oluştururken ilk yapılacak şey aygıt sürücüsünün büyük ve küçük numaralarını belirleyip tahsis etmektir. Programcı aygıt numaralarını statik olarak ya da dinamik olarak belirleyebilir. Yani programcı gerçekleştireceği karakter aygıt sürücüsü için kafasında bir aygıt numarasını işin başında statik olarak belirlemiş olabilir. Ya da bu belirlemeyi sürücüyü yüklerken dinamik olarak yapabilir. Aygıt numaralarının statik olarak belirlenmesinin en önemli sakıncası bu numaraların başka sürücüler tarafından zaten kullanılıyor olmasıdır. Oysa dinamik belirlemede programcı aygıt numaraları boş olanlardan o anda seçebilmektedir.

Aygıt numaralarını statik belirlemek için register_chrdev_region isimli fonksiyon kullanılır:

```
int register_chrdev_region(dev_t from, unsigned count, const char *name);
```

Fonksiyonun birinci parametresi aygıt numarasının büyük ve küçük numaralarını belirten dev_t türünden değerdir. İkinci parametre kaç tane minör numaranın tahsis edileceğini belirtir. Belli bir minör numaradan başlayarak belli sayıda adışıl minör numara tahsis edilebilmektedir. Üçüncü parametre ise aygıt sürcüsünün sysfs ve proc dosya sistemlerinde görüntülenecek olan ismini belirtir. Fonksiyon başarı durumunda 0 değerine başarısızlık durumunda negatif hata koduna geri dönemektedir. Tabii eğer tahsis edilmek istenen numaralar zaten kullanılıyorsa fonksiyon başarısızlıkla geri dönecektir. register_chrdev_region fonksiyonuyla tahsis edilen aygıt numarası unregister_chrdev_region fonksiyonuyla bırakılabilir:

```
void unregister_chrdev_region(dev_t from, unsigned count);
```

Örneğin:

```
/* generic-chrdev.c */

#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/init.h>
#include <linux/fs.h>

static dev_t g_dev = MKDEV(23, 0);

static int __init generic_init(void)
{
    int result;
```

```
    if ((result = register_chrdev_region(g_dev, 1, "generic-chrdev")) != 0) {
        printk(KERN_INFO "register_chrdev_region: %d\n", result);
        return result;
    }

    printk(KERN_INFO "success\n");

    return 0;
}

static void __exit generic_exit(void)
{
    unregister_chrdev_region(g_dev, 1);

    printk(KERN_INFO "Goodbye World!..\n");
}

module_init(generic_init);
module_exit(generic_exit);
```

Şüphesiz aygıt sürücüler için numaralar genellikle dinamik biçimde tahsis edilmektedir. Dinamik tahsisat için alloc_chrdev_region fonksiyonu kullanılmaktadır.

```
int alloc_chrdev_region(dev_t *dev, unsigned baseminor, unsigned count, const char *name);
```

Fonksiyonun birinci parametresi boşta olan aygıt numarasının yerleştirileceği dev_t türünden nesnenin adresini alır. İkinci parametre küçük numaranın başlangıcını belirtir. Programcı küçük numarayı kendisi belirleyebilmektedir. Üçüncü parametre yine küçük numaraların sayısını belirtir. Son parametre de aygıtın ismidir. Fonksiyon başarı durumunda sıfır değerine başarısızlık durumunda negatif hata değerine geri döner. Yine alloc_chrdev_region fonksiyonuyla tahsis edilmiş olan aygıt numarası unregister_chrdev_region fonksiyonuyla serbest bırakılabilir. Örneğin:

```
/* generic-chrdev.c */

#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/init.h>
#include <linux/fs.h>

static dev_t g_dev;

static int __init generic_init(void)
{
    int result;

    if ((result = alloc_chrdev_region(&g_dev, 0, 1, "generic-chrdev")) != 0) {
        printk(KERN_INFO "register_chrdev_region: %d\n", result);
        return result;
    }

    printk(KERN_INFO "success! Major: %d, Minor: %d\n", MAJOR(g_dev), MINOR(g_dev));

    return 0;
}

static void __exit generic_exit(void)
{
    unregister_chrdev_region(g_dev, 1);

    printk(KERN_INFO "Goodbye World!..\n");
```

```
}

module_init(generic_init);
module_exit(generic_exit);
```

Aygıt sürücü için make dosyası şöyle olacaktır:

```
/* Makefile */

obj-m += $(file).o

all:
    make -C /lib/modules/$(shell uname -r)/build M=$(PWD) modules
clean:
    make -C /lib/modules/$(shell uname -r)/build M=$(PWD) clean
```

Burada make dosyasının komut satırı argümanı aldığına dikkat ediniz. Çalıştırma şöyle yapılmalıdır:

```
make file=generic-chrdev
```

Aygıt numaralarını belirledikten sonra programcının o numaralara ilişkin aygıt dosyalarını yaratması gerekir. Çünkü yukarıda da belirtildiği gibi aygıt sürücüler aygıt dosyaları yoluyla open POSIX fonksiyonuyla açılmaktadır. Aygıt numaraları statik belirlenmişse programcı bunu mknod komutunu kulalnarak zaten manuel biçimde oluşturabilir. Ancak aygıt numaraları dinamik olarak belirlenmişse programcının dinamik belirlenmiş numarayı bilmesi gerekir. İşte aygıt sürücüsü yüklendiğinde /proc/devices dosyasında register_chrdev_region ya da alloc_chrdev_region fonksiyonunda belirtilen isimle bir satır oluşturulmaktadır. Programcı o satırdan aygıt numarasını elde edip mknod komutunu uygulayabilir. Aşağıda chrdev-generic.ko sürücüsü yüklendikten sonra /proc/devices dosyasının içeriğinin bir bölümünü görüyorsunuz:

```
 10 misc
 13 input
 14 sound/midi
 14 sound/dmmidi
 21 sg
 23 chrdev
 29 fb
 89 i2c
 99 ppdev
108 ppp
116 alsa
128 ptm
136 pts
180 usb
189 usb_device
226 drm
244 chrdev-generic
245 chrdev
246 chrdev
247 hidraw
248 aux
249 bsg
250 watchdog
```

Aygıt sürücüsü dinamik olarak yüklendikten sonra bu numarayı alarak mknod komutuyla aygıt dosyasını oluşturan "load" isimli basit bir bash script şöyle yazılabilir:

```
#!/bin/bash

module=$1
mode="666"

rm -f /dev/$module
```

```
/sbin/insmod ./$module.ko ${@:2} || exit 1
major=$(awk "\$2==\"$module\" {print \$1}" /proc/devices)

mknod /dev/$module c $major 0
chmod $mode /dev/$module
```

Buradaki "load" isimli script dosyasının kabuk üzerinde çalıştırılabilmesi için ona "x" özelliğinin atanması gerekir. Bu işlem şöyle yapılabilir:

```
chmod +x load
```

Görüldüğü gibi script dosyası bir komut satırı argümanı almaktadır. Bu komut satırı argümanı yüklenecek aygıt sürücü dosyasını (yalnızca ismini) belirtir. Bu script ile yükleme şöyle yapılabilir:

```
sudo ./load generic-chrdev
```

Aygıt sürücünün boşaltılması kabuk üzerinden yine rmmod komutuyla yapılabilir:

```
sudo rmmod generic-chrdev.ko
```

Modül Makroları

Çekirdek modüllerinde modülü betimlemek için bazı makrolar kullanılmaktadır. Bu makrolar modüle ilişkin çeşitli bilgileri ELF dosyasının bazı bölümlerine yazarlar. Özellikle modülün lisansını belirten MODULE_LICENSE makrosu önemlidir. Bu makroyu görmeyen gcc bağlayıcıları uyarı verebilmektedir. MODULE_LICENSE makrosu modülün lisansını belirtir. Normal olarak bu lisans “GPL” olmalıdır. Ancak başka açık kaynak kodlu lisanslar da buraya girilebilir. Örneğin:

```
MODULE_LICENSE("GPL");
```

Diğer bir modül makrosu MODULE_DESCRIPTION isimli makrodur. Bu makro modül hakkında bir özet bilgi verecek biçimde oluşturulur. Örneğin

```
MODULE_DESCRIPTION("Generic Char Device");
```

MODULE_AUTHOR isimli makro modülü yazan kişilerin isimlerini belirtmek için kullanılmaktadır. Örneğin:

```
MODULE_AUTHOR("Kaan Aslan");
```

Bir çekirdek modülünde yukarıdaki üç makronun da bulundurulması iyi bir tekniktir. Bu makrolar global herhangi bir bölgeye yerleştirilebilir. Örneğin:

```
/* generic-chrdev.c  */

#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/init.h>
#include <linux/fs.h>

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Kaan Aslan");
MODULE_DESCRIPTION("Generic Character Device Driver");

static dev_t g_dev;

static int __init generic_init(void)
{
    int result;
```

```
    if ((result = alloc_chrdev_region(&g_dev, 0, 1, "generic-chrdev")) != 0) {
        printk(KERN_INFO "register_chrdev_region: %d\n", result);
        return result;
    }

    printk(KERN_INFO "success! Major: %d, Minor: %d\n", MAJOR(g_dev), MINOR(g_dev));

    return 0;
}

static void __exit generic_exit(void)
{
    unregister_chrdev_region(g_dev, 1);

    printk(KERN_INFO "Goodbye World!..\n");
}

module_init(generic_init);
module_exit(generic_exit);
```

Karakter Aygıt Sürücülerinin Yaratılması ve Yerleştirilmesi

Yukarıdaki örneklerde biz yalnızca çekirdek modüllerine birer aygıt numarası atadık. Şu ana kadar bu modüllerin bir karakter aygıt sürücüsüyle doğrudan bir ilgileri yoktu. İşte bir karakter aygıt sürücüsünün yaratılması için cdev isimli bir yapı nesnesinin (buna cdev nesnesi de diyebiliriz) ilkdeğerlenmesi ve sonra onun cdev_add fonksiyonuyla sisteme yerleştirilmesi gerekmektedir. cdev yapısı bir karakter aygıt sürücüsünü temsil etmektedir. cdev nesnesi programcı tarafından statik düzeyde yaratılabilir ya da cdev_alloc fonksiyonuyla çekirdeğin heap alanında tahsis edilebilir. Eğer cdev nesnesini programcı statik düzeyde kendisi yaratacaksa bunun cdev_init fonksiyonuyla ilkdeğerlenmesi gerekir. Ancak cdev nesnesi cdev_alloc fonksiyonyla tahsis edilecekse ilkdeğerleme için programcının bir şey yapmasına gerek yoktur. Çünkü zaten ilkdeğerleme cdev_alloc fonksiyonunun kendisi tarafından yapılmaktadır. cdev nesnesi ister programcı tarafından statik düzeyde tahsis edilmiş olsun isterse cdev_alloc fonksiyonuyla dinamik biçimde tahsis edilmiş olsun her iki durumda da onun cdev_add fonksiyonuyla sisteme yerleştirilmesi gerekir. cdev_init fonksiyonun prototipi şöyledir:

```
#include <linux/cdev.h>

void cdev_init(struct cdev *cdev, const struct file_operations *fops);
```

Fonksiyonun birinci parametresi ilkdeğerlenecek cdev nesnesinin adresini alır. İkinci parametresi ise izleyen bölümde ele alınacak olan file_operations yapısı nesnesinin adresini almaktadır. Aşağıda örnek bir file_operations nesnesi tanımlanarak ilkdeğer verilmiştir:

```
struct file_operations generic_fops = {
    .owner = THIS_MODULE,
    .llseek = generic_llseek,
    .read = generic_read,
    .write = generic_write,
    .ioctl = generic_ioctl,
    .open = generic_open,
    .release = generic_release,
};
```

file_operatons yapısı çeşitli fonksiyon adreslerini tutan elemanlara sahiptir. Buradaki generic_llseek, generic_read, generic_write, generic_ioctl, generic_open ve generic_release fonksiyon adresleridir. Normal olarak bu fonksiyonlar modülün içerisine modülü oluşturan programcı tarafından tanımlanmış olmalıdır. Tabii aslında bu fonksiyonların hepsinin bulunması zorunlu değildir. Zaten bu konu izleyen bölümde ele alınacaktır.

Yukarıdaki yapıya ilkdeğer verme sentaksı C99'la birlikte resmiyet kazanmıştır. Ancak uzun süredir zaten gcc derleyicileri bu sentaksı bir eklenti (extension) olarak destekliyordu. Bu sentaksta yapı elemanlarına ".elemman_ismi = ilkdeğer" biçiminde karışık sırada değer atanabilmektedir. Şimdi bu ön bilgiyi de kullanarak bir karakter aygıt sürücüsünün statik düzeyde nasıl tahsis edildiğini ve ilkdeğerlendiğini görelim:

```
struct cdev g_generic_dev;

struct file_operations g_generic_fops = {
    .owner = THIS_MODULE,
    .llseek = generic_llseek,
    .read = generic_read,
    .write = generic_write,
    .ioctl = generic_ioctl,
    .open = generic_open,
    .release = generic_release,
};
...
cdev_init(&g_generic_dev,  &g_generic_fops);
g_generic_dev.owner = THIS_MODULE;
```

Eğer cdev nesnesi cdev_alloc fonksiyonuyla dinamik düzeyde tahsis edilecekse bu durumda cdev_init uygulamaya gerek yoktur. Zaten cdev_alloc kendi içerisinde nesneyi çekirdeğin heap alanında tahsis ettikten sonra ona ilkdeğerlerini verir. cdev_alloc fonksiyonun prototipi de şöyledir:

```
#include <linux/cdev.h>

struct cdev *cdev_alloc(void);
```

Fonksiyonun parametre almaz. Geri dönüş değeri tahsis edilen cdev nesnesinin adresidir. Tabii fonksiyon başarısız olabilir. Bu durumda NULL adrese geri döner. cdev_alloc ile cdev nesnesi dinamik tahsis edilecekse file_operations yapısı programcı tarafından bu cdev nesnesinin ops elemanına manuel olarak atanmalıdır. Örneğin:

```
struct cdev *g_generic_dev;

struct file_operations g_generic_fops = {
    .owner = THIS_MODULE,
    .llseek = generic_llseek,
    .read = generic_read,
    .write = generic_write,
    .ioctl = generic_ioctl,
    .open = generic_open,
    .release = generic_release,
};
...
g_generic_dev = cdev_alloc();
g_generic_dev->ops = &g_generic_fops;
g_generic_dev->owner = THIS_MODULE;
```

file_operations yapısının tüm eelemanları aşağıdaki gibidir:

```
struct file_operations {
    struct module *owner;
    loff_t(*llseek) (struct file *, loff_t, int);
    ssize_t(*read) (struct file *, char __user *, size_t, loff_t *);
    ssize_t(*write) (struct file *, const char __user *, size_t, loff_t *);
    ssize_t(*aio_read) (struct kiocb *, const struct iovec *, unsigned long, loff_t);
    ssize_t(*aio_write) (struct kiocb *, const struct iovec *, unsigned long, loff_t);
    int(*readdir) (struct file *, void *, filldir_t);
    unsigned int(*poll) (struct file *, struct poll_table_struct *);
    long(*unlocked_ioctl) (struct file *, unsigned int, unsigned long);
```

```
    long(*compat_ioctl) (struct file *, unsigned int, unsigned long);
    int(*mmap) (struct file *, struct vm_area_struct *);
    int(*open) (struct inode *, struct file *);
    int(*flush) (struct file *, fl_owner_t id);
    int(*release) (struct inode *, struct file *);
    int(*fsync) (struct file *, loff_t, loff_t, int datasync);
    int(*aio_fsync) (struct kiocb *, int datasync);
    int(*fasync) (int, struct file *, int);
    int(*lock) (struct file *, int, struct file_lock *);
    ssize_t(*sendpage) (struct file *, struct page *, int, size_t, loff_t *, int);
    unsigned long(*get_unmapped_area)(struct file *, unsigned long, unsigned long, unsigned long, unsigned
long);
    int(*check_flags)(int);
    int(*flock) (struct file *, int, struct file_lock *);
    ssize_t(*splice_write)(struct pipe_inode_info *, struct file *, loff_t *, size_t, unsigned int);
    ssize_t(*splice_read)(struct file *, loff_t *, struct pipe_inode_info *, size_t, unsigned int);
    int(*setlease)(struct file *, long, struct file_lock **);
    long(*fallocate)(struct file *file, int mode, loff_t offset, loff_t len);
    int(*show_fdinfo)(struct seq_file *m, struct file *f);
};
```

Tabii aslında bu yapının tüm elemanlarının girilmesine duruma göre gerek yoktur. Değer atanmayan elemanlara NULL adres atanmış olacaktır.

cdev nesnesi yukarıdaki iki biçimden biri yoluyla tahsis edilip ilkdeğerlendikten sonra artık cdev_add fonksiyonuyla yerleştirilmelidir. cdev_add fonksiyonunun prototipi şöyledir:

```
int cdev_add(struct cdev *pcdev, dev_t dev, unsigned count);
```

Fonksiyonun birinci parametresi yerleştirilecek cdev nesnesinin adresini alır. İkinci parametre bu cdev nesnesinin ilişkin olduğu aygıt numarasını belirtir. Normal olarak yukarıdaki örnek kodlarda elde edilen aygıt numarası bu parametreye girilmelidir. Son parametre aygıtın minör numara miktarını belirtmektedir. Çoğu durumda bu parametre için 1 değeri girilmektedir.

Karakter aygıt sürücüsü nasıl tahsis edilmiş olursa olsun cdev_add ile yerleştirildikten sonra aygıt sürücü kaldırılırken cdev_del fonksiyonuyla yerleştirildiği yerden çıkarılmalıdır. cdev_del fonksiyonunun prototipi şöyledir:

```
#include <linux/cdev.h>

void cdev_del(struct cdev *pcdev);
```

Fonksiyon başarı durumunda sıfır değerine başarısızlık durumunda negatif hata koduna geri dönmektedir. Aşağıda statik düzeyde cdev nesnesinin tahsis edilip yerleştirilmesine ilişkin çekirdek modülü örneğini görüyorsunuz:

```
/* generic-chrdev.c  */

#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/init.h>
#include <linux/fs.h>
#include <linux/cdev.h>

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Kaan Aslan");
MODULE_DESCRIPTION("Generic Character Device Driver");

static int generic_open(struct inode *inodep, struct file *filp);
static int generic_release(struct inode *inodep, struct file *filp);
```

```
static dev_t g_dev;
static struct cdev g_mycdev;

struct file_operations g_generic_fops = {
    .owner = THIS_MODULE,
    .open = generic_open,
    .release = generic_release,
};

static int __init generic_init(void)
{
    int result;

    if ((result = alloc_chrdev_region(&g_dev, 0, 1, "generic-chrdev")) != 0) {
        printk(KERN_INFO "register_chrdev_region: %d\n", result);
        return result;
    }

    printk(KERN_INFO "success! Major: %d, Minor: %d\n", MAJOR(g_dev), MINOR(g_dev));

    cdev_init(&g_mycdev, &g_generic_fops);
    g_mycdev.owner = THIS_MODULE;

    if ((result = cdev_add(&g_mycdev, g_dev, 1)) != 0) {
        printk(KERN_ALERT "cannot add character device!..\n");
        unregister_chrdev_region(g_dev, 1);
        return result;
    }

    return 0;
}

static int generic_open(struct inode *inodep, struct file *filp)
{
    return 0;
}

static int generic_release(struct inode *inodep, struct file *filp)
{
    return 0;
}

static void __exit generic_exit(void)
{
    cdev_del(&g_mycdev);
    unregister_chrdev_region(g_dev, 1);

    printk(KERN_INFO "Goodbye World!..\n");
}

module_init(generic_init);
module_exit(generic_exit);
```

Şimdi de cdev nesnesinin dinamik düzeyde tahsis edilmesine örnek verelim:

```
/* generic-chrdev.c  */

#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/init.h>
#include <linux/fs.h>
#include <linux/cdev.h>
```

```
MODULE_LICENSE("GPL");
MODULE_AUTHOR("Kaan Aslan");
MODULE_DESCRIPTION("Generic Character Device Driver");

static int generic_open(struct inode *inodep, struct file *filp);
static int generic_release(struct inode *inodep, struct file *filp);

static dev_t g_dev;
static struct cdev *g_mycdev;

struct file_operations g_generic_fops = {
    .owner = THIS_MODULE,
    .open = generic_open,
    .release = generic_release,
};

static int __init generic_init(void)
{
    int result;

    if ((result = alloc_chrdev_region(&g_dev, 0, 1, "generic-chrdev")) != 0) {
        printk(KERN_INFO "register_chrdev_region: %d\n", result);
        return result;
    }

    printk(KERN_INFO "success! Major: %d, Minor: %d\n", MAJOR(g_dev), MINOR(g_dev));

    if ((g_mycdev = cdev_alloc()) == NULL) {
        printk(KERN_ALERT "cannot allocate character device!..\n");
        unregister_chrdev_region(g_dev, 1);
        return -ENOMEM;
    }
    g_mycdev->owner = THIS_MODULE;
    g_mycdev->ops = &g_generic_fops;

    if ((result = cdev_add(g_mycdev, g_dev, 1)) != 0) {
        printk(KERN_ALERT "cannot add character device!..\n");
        unregister_chrdev_region(g_dev, 1);
        return result;
    }

    return 0;
}

static int generic_open(struct inode *inodep, struct file *filp)
{
    return 0;
}

static int generic_release(struct inode *inodep, struct file *filp)
{
    return 0;
}

static void __exit generic_exit(void)
{
    cdev_del(g_mycdev);
    unregister_chrdev_region(g_dev, 1);

    printk(KERN_INFO "Goodbye World!..\n");
}

module_init(generic_init);
```

```
module_exit(generic_exit);
```

Karakter Aygıt Sürücülerin Kullanıcı Modunda Çalışan Programlar Tarafından Açılması ve Kapatılması

Aygıt sürücüler kullanıcı modunda open POSIX fonksiyonuyla aygıt dosyası belirtilerek açılırlar ve close POSIX fonksiyonuyla kapatılırlar. Linux'ta open POSIX fonksiyonu sys_open isimli sistem fonksiyonunu, close POSIX fonksiyonu da sys_close isimli sistem fonksiyonunu çağırmaktadır. Bir aygıt sürücü open fonksiyonu ile açıldığında aygıt sürücü içerisindeki file_operations yapısının open elemanında belirtilen fonksiyon çağrılır. Benzer biçimde aygıt sürücü close fonksiyonuyla kapatıldığında da file_operations yapısının close fonksiyonu çağrılmaktadır.

```
/* generic-chrdev.c  */

#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/init.h>
#include <linux/fs.h>
#include <linux/cdev.h>

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Kaan Aslan");
MODULE_DESCRIPTION("Generic Character Device Driver");

static int generic_open(struct inode *inodep, struct file *filp);
static int generic_release(struct inode *inodep, struct file *filp);

static dev_t g_dev;
struct cdev g_mycdev;

struct file_operations g_generic_fops = {
    .owner = THIS_MODULE,
    .open = generic_open,
    .release = generic_release,
};

static int __init generic_init(void)
{
    int result;

    if ((result = alloc_chrdev_region(&g_dev, 0, 1, "generic-chrdev")) != 0) {
        printk(KERN_INFO "register_chrdev_region: %d\n", result);
        return result;
    }

    printk(KERN_INFO "success! Major: %d, Minor: %d\n", MAJOR(g_dev), MINOR(g_dev));

    cdev_init(&g_mycdev, &g_generic_fops);
    g_mycdev.owner = THIS_MODULE;

    if ((result = cdev_add(&g_mycdev, g_dev, 1)) != 0) {
        printk(KERN_ALERT "cannot add character device!..\n");
        unregister_chrdev_region(g_dev, 1);
        return result;
    }

    return 0;
}

static int generic_open(struct inode *inodep, struct file *filp)
{
```

```
    printk(KERN_INFO "device opened\n");

    return 0;
}

static int generic_release(struct inode *inodep, struct file *filp)
{
    printk(KERN_INFO "device closed\n");

    return 0;
}

static void __exit generic_exit(void)
{
    cdev_del(&g_mycdev);
    unregister_chrdev_region(g_dev, 1);

    printk(KERN_INFO "Goodbye World!..\n");
}

module_init(generic_init);
module_exit(generic_exit);

/* testapp.c */

#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <unistd.h>

void exit_sys(const char *msg);

int main(int argc, char *argv[])
{
    int fd;

    if ((fd = open("/dev/generic-chrdev", O_RDONLY)) == -1)
        exit_sys("open");

    close(fd);

    return 0;
}

void exit_sys(const char *msg)
{
    perror(msg);
    exit(EXIT_FAILURE);
}
```

Aşağıda aygıt sürücü yüklenip test programı çalıştırıldıktan sonra oluşan mesajları görüyorsunuz:

```
kaan@kaan-VirtualBox ~/Study/DeviceDrivers $ make file=generic-chrdev
make -C /lib/modules/4.10.0-38-generic/build M=/home/kaan/Study/DeviceDrivers modules
make[1]: Entering directory '/usr/src/linux-headers-4.10.0-38-generic'
  CC [M]  /home/kaan/Study/DeviceDrivers/generic-chrdev.o
  Building modules, stage 2.
  MODPOST 1 modules
  LD [M]  /home/kaan/Study/DeviceDrivers/generic-chrdev.ko
make[1]: Leaving directory '/usr/src/linux-headers-4.10.0-38-generic'
kaan@kaan-VirtualBox ~/Study/DeviceDrivers $ sudo ./load generic-chrdev
kaan@kaan-VirtualBox ~/Study/DeviceDrivers $ gcc -o testapp testapp.c
kaan@kaan-VirtualBox ~/Study/DeviceDrivers $ ./testapp
kaan@kaan-VirtualBox ~/Study/DeviceDrivers $ sudo rmmod generic-chrdev.ko
kaan@kaan-VirtualBox ~/Study/DeviceDrivers $ dmesg
[ 4510.154398] success! Major: 245, Minor: 0
[ 4527.819011] device opened
[ 4527.819014] device closed
[ 4534.225500] Goodbye World!..
kaan@kaan-VirtualBox ~/Study/DeviceDrivers $
```

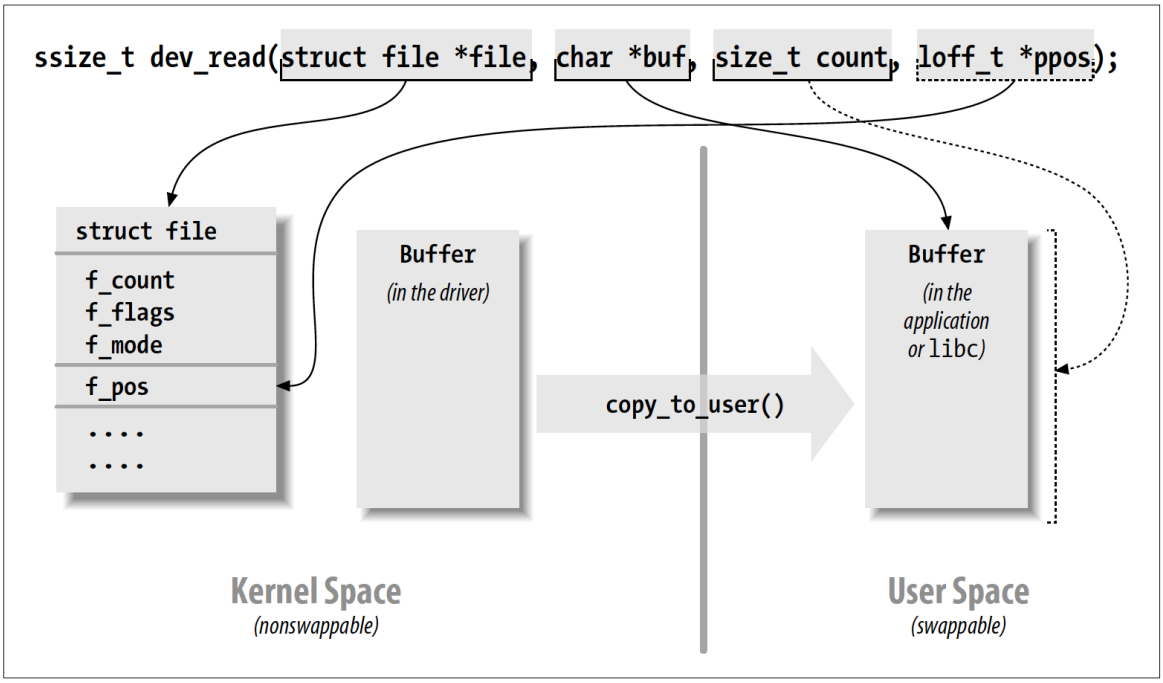
Karakter Aygıt Sürücülerinde read ve write İşlemleri

Bir aygıt sürücüsü open fonksiyonuyla açıldıktan sonra read POSIX fonksiyonuyla aygıt sürücüden okuma yapılmak istendiğinde aygıt sürücünün file_operations yapısının read elemanında belirtilen fonksiyonu çağrılır. Benzer biçimde kullanıcı modundan write POSIX fonksiyonuyla aygıt sürücüye yazma yapılmak istendiğinde de aygıt sürücünün file_operations yapısının write elemanında belirtilen fonksiyonu çağrılacaktır. Biz aygıt sürücüde read fonksiyonu için çağrılan fonksiyona "aygıt sürücünün read fonksiyonu", write fonksiyonu için çağırılan fonksiyona da "aygıt sürücünün write fonksiyonu" diyeceğiz.

Aygıt sürücünün read ve write fonksiyonlarına istediğimiz isimleri verebiliriz. Ancak bu fonksiyonların parametrik yapıları şöyle olmalıdır:

```
ssize_t dev_read(struct file *filp, char __user *buf, size_t size, loff_t *offset);
ssize_t dev_write(struct file *filp, const char __user *buf, size_t size, loff_t *offset);
```

Fonksiyonların birinci parametrelerine çekirdek dosya nesnesinin adresi geçirilmektedir. İkinci parametreler veriler için aktarım adresini belirtmektedir. Yani read fonksiyonunda aygıt sürücüden okunmak istenen bilgileri aygıt sürücü bu adrese yerleştirmeli, write fonksiyonunda da aygıt sürücüye yazılmak istenen bilgileri aygıt sürücü bu adresten almalıdır. Fonksiyonların üçüncü parametreleri okunacak ya da yazılacak byte miktarını belirtmektedir. Fonksiyonların son parametreleri ise file yapısının dosya göstericisinin konumunu belirten f_pos elemanının adresini belirtmektedir. Yani bu fonksiyon çağrıldığında bu göstericinin gösterdiği yerde dosya göstericisinin (dosya göstericisi terimi okuma ve yazma işlemlerinin nereden yapılacağına ilişkin offset değeri belirtmektedir.) Aygıt sürücünün okuma ve yazma miktarı kadar bu göstericinin gösterdiği yerdeki değeri artırması beklenmektedir. Fonksiyonlar başarı durumunda okuyabildikleri ya da yazabildikleri byte sayısına başarısızlık durumunda ise negatif hata değerine geri dönerler. Aşağıdaki şekilde read fonksiyonunun ne yaptığı şekilsel olarak açıklanmaktadır.



Fonksiyonların ikinci parametrelerindeki __user makrosu okunabilirlik için bulundurulmaktadır. Bu makro önişlemci tarafından boşluk ile değiştirilmektedir.

Şimdi okuma işlemi yapılabilen bir karakter aygıt sürücüsü örneği vermek istiyoruz. Aşağıdaki örnekte aygıt sürücünün içerisinde static bir char türden dizi yaratılıp onun içerisine bir yazı yerleştirilmiştir. Bu dizi sanki bir dosya gibi okunabilmektedir. Dosya göstericisi dizide kalınan offset numarasını belirtmektedir. Her okuma işleminden sonra dosya göstericisi okuma yapılan miktar kadar artılmıştır. Yazının sonuna gelindiğinde fonksiyon 0 değeriyle başarısızlık durumunda negatif hata koduyla geri dönmektedir.

```
/* generic-chrdev.c  */

#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/init.h>
#include <linux/fs.h>
#include <linux/cdev.h>
#include <asm/uaccess.h>
#include <linux/string.h>

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Kaan Aslan");
MODULE_DESCRIPTION("Generic Character Device Driver");

static int generic_open(struct inode *inodep, struct file *filp);
static int generic_release(struct inode *inodep, struct file *filp);
static ssize_t generic_read(struct file *filp, char __user *buf, size_t size, loff_t *offset);

static dev_t g_dev;
static struct cdev g_mycdev;

static char g_buf[] = "0123456789abcdefg";

struct file_operations g_generic_fops = {
    .owner = THIS_MODULE,
    .open = generic_open,
    .release = generic_release,
    .read = generic_read,
};
```

```
static int __init generic_init(void)
{
    int result;

    if ((result = alloc_chrdev_region(&g_dev, 0, 1, "generic-chrdev")) != 0) {
        printk(KERN_INFO "register_chrdev_region: %d\n", result);
        return result;
    }

    printk(KERN_INFO "success! Major: %d, Minor: %d\n", MAJOR(g_dev), MINOR(g_dev));

    cdev_init(&g_mycdev, &g_generic_fops);
    g_mycdev.owner = THIS_MODULE;

    if ((result = cdev_add(&g_mycdev, g_dev, 1)) != 0) {
        printk(KERN_ALERT "cannot add character device!..\n");
        unregister_chrdev_region(g_dev, 1);
        return result;
    }

    return 0;
}

static int generic_open(struct inode *inodep, struct file *filp)
{
    printk(KERN_INFO "device opened\n");

    return 0;
}

static int generic_release(struct inode *inodep, struct file *filp)
{
    printk(KERN_INFO "device closed\n");

    return 0;
}

static ssize_t generic_read(struct file *filp, char __user *buf, size_t size, loff_t *offset)
{
    unsigned long left, n;

    left = strlen(g_buf) - *offset;
    n = left < size ? left : size;

    if (n) {
        if (copy_to_user(buf, g_buf + *offset, n))
            return -EFAULT;

        *offset += n;
        printk(KERN_INFO "%lu bytes read at offset %llu\n", n, *offset);
    }

    return n;
}

static void __exit generic_exit(void)
{
    cdev_del(&g_mycdev);
    unregister_chrdev_region(g_dev, 1);

    printk(KERN_INFO "Goodbye World!..\n");
}
```

```
module_init(generic_init);
module_exit(generic_exit);

/* testapp.c */

#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <unistd.h>

void exit_sys(const char *msg);

int main(int argc, char *argv[])
{
    int fd;
    char buf[4];
    ssize_t result;

    if ((fd = open("/dev/generic-chrdev", O_RDONLY)) == -1)
        exit_sys("open");

    while ((result = read(fd, buf, 3)) > 0) {
        buf[result] = '\0';
        puts(buf);
    }
    if (result == -1)
        exit_sys("read");

    close(fd);

    return 0;
}

void exit_sys(const char *msg)
{
    perror(msg);
    exit(EXIT_FAILURE);
}
```

Test işlemi için şunlar yapılmıştır:

```
kaan@kaan-VirtualBox ~/Study/DeviceDrivers $ sudo dmesg --clear
kaan@kaan-VirtualBox ~/Study/DeviceDrivers $ make file=generic-chrdev
make -C /lib/modules/4.10.0-38-generic/build M=/home/kaan/Study/DeviceDrivers modules
make[1]: Entering directory '/usr/src/linux-headers-4.10.0-38-generic'
  CC [M]  /home/kaan/Study/DeviceDrivers/generic-chrdev.o
  Building modules, stage 2.
  MODPOST 1 modules
  CC      /home/kaan/Study/DeviceDrivers/generic-chrdev.mod.o
  LD [M]  /home/kaan/Study/DeviceDrivers/generic-chrdev.ko
make[1]: Leaving directory '/usr/src/linux-headers-4.10.0-38-generic'
kaan@kaan-VirtualBox ~/Study/DeviceDrivers $ sudo ./load generic-chrdev
kaan@kaan-VirtualBox ~/Study/DeviceDrivers $ ./testapp
012
345
678
9ab
cde
fg
kaan@kaan-VirtualBox ~/Study/DeviceDrivers $ sudo rmmod generic-chrdev.ko
kaan@kaan-VirtualBox ~/Study/DeviceDrivers $ dmesg
[ 1112.372502] success! Major: 245, Minor: 0
[ 1120.826843] device opened
[ 1120.826846] 3 bytes read at offset 0
[ 1120.826899] 3 bytes read at offset 3
[ 1120.826903] 3 bytes read at offset 6
[ 1120.826906] 3 bytes read at offset 9
[ 1120.826910] 3 bytes read at offset 12
[ 1120.826913] 2 bytes read at offset 15
[ 1120.826916] 0 bytes read at offset 17
[ 1120.826917] device closed
[ 1140.441911] Goodbye World!..
```

Şimdi de write işlemi için yukarıdaki örneği biraz geliştirelim. write işleminde yazılacaklar dosya göstericisinin gösterdiği yerden itibaren aynı diziye aktarılacak olsun. Test kodunda da önce bu dizye yazma yapalım sonra yazdıklarımızı okumaya çalışalım.

```
/* generic-chrdev.c  */

#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/init.h>
#include <linux/fs.h>
#include <linux/cdev.h>
#include <asm/uaccess.h>
#include <linux/string.h>

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Kaan Aslan");
MODULE_DESCRIPTION("Generic Character Device Driver");

static int generic_open(struct inode *inodep, struct file *filp);
static int generic_release(struct inode *inodep, struct file *filp);
static ssize_t generic_read(struct file *filp, char __user *buf, size_t size, loff_t *offset);
static ssize_t generic_write(struct file *filp, const char __user *buf, size_t size, loff_t
*offset);

static dev_t g_dev;
static struct cdev g_mycdev;

static char g_buf[] = "0123456789abcdefg";

struct file_operations g_generic_fops = {
    .owner = THIS_MODULE,
    .open = generic_open,
    .release = generic_release,
    .read = generic_read,
    .write = generic_write,
};
```

```
static int __init generic_init(void)
{
    int result;

    if ((result = alloc_chrdev_region(&g_dev, 0, 1, "generic-chrdev")) != 0) {
        printk(KERN_INFO "register_chrdev_region: %d\n", result);
        return result;
    }

    printk(KERN_INFO "success! Major: %d, Minor: %d\n", MAJOR(g_dev), MINOR(g_dev));

    cdev_init(&g_mycdev, &g_generic_fops);
    if ((result = cdev_add(&g_mycdev, g_dev, 1)) != 0) {
        printk(KERN_ALERT "cannot add character device!..\n");
        unregister_chrdev_region(g_dev, 1);
        return result;
    }

    return 0;
}

static int generic_open(struct inode *inodep, struct file *filp)
{
    printk(KERN_INFO "device opened\n");

    return 0;
}

static int generic_release(struct inode *inodep, struct file *filp)
{
    printk(KERN_INFO "device closed\n");

    return 0;
}

static ssize_t generic_read(struct file *filp, char __user *buf, size_t size, loff_t *offset)
{
    unsigned long left, n;

    left = strlen(g_buf) - *offset;
    n = left < size ? left : size;

    if (n) {
        if (copy_to_user(buf, g_buf + *offset, n))
            return -EFAULT;

        *offset += n;
        printk(KERN_INFO "%lu bytes read at offset %llu\n", n, *offset);
    }

    return n;
}

static ssize_t generic_write(struct file *filp, const char __user *buf, size_t size, loff_t
*offset)
{
    unsigned long left, n;

    left = strlen(g_buf) - *offset;
    n = left < size ? left : size;

    if (n) {
```

```
        if (copy_from_user(g_buf + *offset, buf, n))
            return -EFAULT;

        *offset += n;
        printk(KERN_INFO "%lu bytes written at offset %llu\n", n, *offset);
    }

    return n;
}

static void __exit generic_exit(void)
{
    cdev_del(&g_mycdev);
    unregister_chrdev_region(g_dev, 1);

    printk(KERN_INFO "Goodbye World!..\n");
}

module_init(generic_init);
module_exit(generic_exit);

/* testapp.c */

#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <unistd.h>

void exit_sys(const char *msg);

int main(int argc, char *argv[])
{
    int fd;
    char buf[100];
    ssize_t result;

    if ((fd = open("/dev/generic-chrdev", O_WRONLY)) == -1)
        exit_sys("open");

    if ((result = write(fd, "ankara", 6)) == -1)
        exit_sys("write");

    close(fd);

    if ((fd = open("/dev/generic-chrdev", O_RDONLY)) == -1)
        exit_sys("open");

    if ((result = read(fd, buf, 100)) == -1)
         exit_sys("read");

    buf[result] = '\0';
    puts(buf);

    close(fd);

    return 0;
}

void exit_sys(const char *msg)
{
    perror(msg);
    exit(EXIT_FAILURE);
}
```

Test kodumuzda henüz dosya göstericisini konumlandıramadığımızdan dosya göstericisini başa almak için dosyayı yeniden açtık. Test kodunun çalıştırılması işlemi de şöyle yapılmıştır:

```
kaan@kaan-VirtualBox ~/Study/DeviceDrivers $ sudo dmesg --clear
kaan@kaan-VirtualBox ~/Study/DeviceDrivers $ make file=generic-chrdev
make -C /lib/modules/4.10.0-38-generic/build M=/home/kaan/Study/DeviceDrivers modules
make[1]: Entering directory '/usr/src/linux-headers-4.10.0-38-generic'
  Building modules, stage 2.
  MODPOST 1 modules
make[1]: Leaving directory '/usr/src/linux-headers-4.10.0-38-generic'
kaan@kaan-VirtualBox ~/Study/DeviceDrivers $ gcc -o testapp testapp.c
kaan@kaan-VirtualBox ~/Study/DeviceDrivers $ sudo ./load generic-chrdev
kaan@kaan-VirtualBox ~/Study/DeviceDrivers $ ./testapp
ankara6789abcdefg
kaan@kaan-VirtualBox ~/Study/DeviceDrivers $ sudo rmmod generic-chrdev.ko
kaan@kaan-VirtualBox ~/Study/DeviceDrivers $ dmesg
[ 4369.756501] success! Major: 245, Minor: 0
[ 4376.467925] device opened
[ 4376.467929] 6 bytes written at offset 6
[ 4376.467930] device closed
[ 4376.467933] device opened
[ 4376.467934] 17 bytes read at offset 17
[ 4376.467984] device closed
[ 4389.403081] Goodbye World!..
```

Aygıt Sürücülerin llseek Fonksiyonları

Aygıt sürücüler dosyalarına ilişkin betimleyicilerle lseek POSIX fonksiyonu çağrıldığında aygıt sürücülerin file_operations dizisinde llseek elemanıyla belirtilen fonksiyonları çağrılmaktadır. llseek elemanına yerleştirilecek fonksiyonun parametrik yapısı şöyle olmalıdır:

```
loff_t dev_llseek(struct file *filp, loff_t off, int whence);
```

Fonksiyonun birinci parametresi dosya nesnesine ilişkin file yapısını, ikinci parametresi konumlandırma offsetîni üçüncü parametresi ise konumlandırma orijinini belirtmektedir. Fonksiyon başarı durumunda konumlandırılmış yeni offset değerine başarısızlık durumunda negatif hata koduna (tipik olarak -EINVAL) değerine geri döner. Bu fonksiyon içerisinde tipik olarak konumlandırma orijini switch içerisine alınarak işlenir. Örneğin:

```
loff_t dev_llseek(struct file *filp, loff_t off, int whence)
{
    loff_t newpos;

    switch(whence) {
      case 0:       /* SEEK_SET */
        /* ... */
        break;

      case 1:       /* SEEK_CUR */
        /* ... */
        break;

      case 2:       /* SEEK_END */
        /* ... */
        break;

      default:      /* can't happen */
        return -EINVAL;
    }
```

```
    if (newpos < 0)
        return -EINVAL;

    filp->f_pos = newpos;

    return newpos;
}
```

file yapısının f_pos elemanının güncellenmesinin fonksiyon tarafından yapıldığına dikkat ediniz. Şimdi yukarıdaki örneği llseek fonksiyonunu kapsayacak biçimde değiştirelim:

```
/* generic-chrdev.c  */

#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/init.h>
#include <linux/fs.h>
#include <linux/cdev.h>
#include <asm/uaccess.h>
#include <linux/string.h>

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Kaan Aslan");
MODULE_DESCRIPTION("Generic Character Device Driver");

static int generic_open(struct inode *inodep, struct file *filp);
static int generic_release(struct inode *inodep, struct file *filp);
static ssize_t generic_read(struct file *filp, char __user *buf, size_t size, loff_t *offset);
static ssize_t generic_write(struct file *filp, const char __user *buf, size_t size, loff_t
*offset);
static loff_t generic_llseek(struct file *filp, loff_t off, int whence);

static dev_t g_dev;
static struct cdev g_mycdev;

static char g_buf[] = "0123456789abcdefg";

struct file_operations g_generic_fops = {
    .owner = THIS_MODULE,
    .open = generic_open,
    .release = generic_release,
    .read = generic_read,
    .write = generic_write,
    .llseek = generic_llseek,
};

static int __init generic_init(void)
{
    int result;

    if ((result = alloc_chrdev_region(&g_dev, 0, 1, "generic-chrdev")) != 0) {
        printk(KERN_INFO "register_chrdev_region: %d\n", result);
        return result;
    }

    printk(KERN_INFO "success! Major: %d, Minor: %d\n", MAJOR(g_dev), MINOR(g_dev));

    cdev_init(&g_mycdev, &g_generic_fops);
    if ((result = cdev_add(&g_mycdev, g_dev, 1)) != 0) {
        printk(KERN_ALERT "cannot add character device!..\n");
        unregister_chrdev_region(g_dev, 1);
        return result;
```

```
    }

    return 0;
}

static int generic_open(struct inode *inodep, struct file *filp)
{
    printk(KERN_INFO "device opened\n");

    return 0;
}

static int generic_release(struct inode *inodep, struct file *filp)
{
    printk(KERN_INFO "device closed\n");

    return 0;
}

static ssize_t generic_read(struct file *filp, char __user *buf, size_t size, loff_t *offset)
{
    unsigned long left, n;

    left = strlen(g_buf) - *offset;
    n = left < size ? left : size;

    if (n) {
        if (copy_to_user(buf, g_buf + *offset, n))
            return -EFAULT;

        *offset += n;
        printk(KERN_INFO "%lu bytes read at offset %llu\n", n, *offset);
    }

    return n;
}

static ssize_t generic_write(struct file *filp, const char __user *buf, size_t size, loff_t
*offset)
{
    unsigned long left, n;

    left = strlen(g_buf) - *offset;
    n = left < size ? left : size;

    if (n) {
        if (copy_from_user(g_buf + *offset, buf, n))
            return -EFAULT;

        *offset += n;
        printk(KERN_INFO "%lu bytes written at offset %llu\n", n, *offset);
    }

    return n;
}

static loff_t generic_llseek(struct file *filp, loff_t off, int whence)
{
    loff_t newpos;

    switch(whence) {
      case 0:       /* SEEK_SET */
        newpos = off;
```

```
        break;
      case 1:       /* SEEK_CUR */
        newpos = filp->f_pos + off;
        break;
      case 2:       /* SEEK_END */
        newpos = strlen(g_buf) + off;
        break;
      default:      /* can't happen */
        return -EINVAL;
    }
    if (newpos < 0 || newpos > strlen(g_buf))
        return -EINVAL;

    filp->f_pos = newpos;

    return newpos;
}

static void __exit generic_exit(void)
{
    cdev_del(&g_mycdev);
    unregister_chrdev_region(g_dev, 1);

    printk(KERN_INFO "Goodbye World!..\n");
}

module_init(generic_init);
module_exit(generic_exit);

/* testapp.c */

#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <unistd.h>

void exit_sys(const char *msg);

int main(int argc, char *argv[])
{
    int fd;
    char buf[100];
    ssize_t result;

    if ((fd = open("/dev/generic-chrdev", O_WRONLY)) == -1)
        exit_sys("open");

    if ((result = write(fd, "ankara", 6)) == -1)
        exit_sys("write");

    close(fd);

    if ((fd = open("/dev/generic-chrdev", O_RDONLY)) == -1)
        exit_sys("open");

    if ((result = read(fd, buf, 100)) == -1)
         exit_sys("read");

    buf[result] = '\0';
    puts(buf);

    lseek(fd, 5, 0);
```

```
    if ((result = read(fd, buf, 100)) == -1)
         exit_sys("read");

    buf[result] = '\0';
    puts(buf);

    close(fd);

    return 0;
}

void exit_sys(const char *msg)
{
    perror(msg);
    exit(EXIT_FAILURE);
}
```

Aygıt sürücü yüklenip test kodu çalıştırıldıktan sonra şöyle bir çıktı elde edilmiştir:

```
kaan@kaan-VirtualBox ~/Study/DeviceDrivers $ make file=generic-chrdev
make -C /lib/modules/4.10.0-38-generic/build M=/home/kaan/Study/DeviceDrivers modules
make[1]: Entering directory '/usr/src/linux-headers-4.10.0-38-generic'
  Building modules, stage 2.
  MODPOST 1 modules
make[1]: Leaving directory '/usr/src/linux-headers-4.10.0-38-generic'
kaan@kaan-VirtualBox ~/Study/DeviceDrivers $ sudo ./load generic-chrdev
kaan@kaan-VirtualBox ~/Study/DeviceDrivers $ gcc -o testapp testapp.c
kaan@kaan-VirtualBox ~/Study/DeviceDrivers $ ./testapp
ankara6789abcdefg
a6789abcdefg
kaan@kaan-VirtualBox ~/Study/DeviceDrivers $ sudo rmmod generic-chrdev.ko
```

Çoklu Açılabilen Karakter Aygıt Sürücüleri

Bir karakter aygıt sürücüsü open fonksiyonuyla farklı minör numaralara karşılık gelen aygıt dosyalarıyla açılabilir. Bu durumda aygıt sürücünün her aygıt için (yani minör numarası farklı olan her aygıt dosyası için) aynı veri yapılarını sıfırdan oluşturması gerekir. Aygıt sürücü dosyaların yazımına geçmeden önce birden fazla minör numaraya ilişkin aygıt dosyalarını yaratıp yüklemek için daha önce yazmış olduğumuz load script'inde değişiklikler yapmamız gerekiyor. load script'inin birden fazla minör numara için yükleme yapan loadmulti isimli versiyonu şöyle yazılabilir:

```
#!/bin/bash

module=$1
mode="666"

rm -f /dev/${module}*

/sbin/insmod ./$module.ko  ${@:3} || exit 1
major=$(awk "\$2==\"$module\" {print \$1}" /proc/devices)

for ((i = 0; i < $2; ++i))
do
    mknod /dev/${module}$i c $major $i
    chmod $mode /dev/${module}$i
done
```

Bu scrip'ti çalıştırırken iki komut satırı argüman vereceğiz. Birinci argüman yüklenecek aygıt sürücü dosyasının ismini, ikinci argüman ise yaratılacak minör numaraların sayısını belirtecektir. Aygıt dosyaları yine dev dizinin altında oluşturulacaktır. Script'in örnek kullanımı şöyle olabilir:

```
kaan@kaan-VirtualBox ~/Study/DeviceDrivers $ sudo ./loadmulti multigeneric-chrdev 5
kaan@kaan-VirtualBox ~/Study/DeviceDrivers $ ls -l /dev/multigeneric-chrdev*
crw-rw-rw- 1 root root 245, 0 Tem 21 11:58 /dev/multigeneric-chrdev0
crw-rw-rw- 1 root root 245, 1 Tem 21 11:58 /dev/multigeneric-chrdev1
crw-rw-rw- 1 root root 245, 2 Tem 21 11:58 /dev/multigeneric-chrdev2
crw-rw-rw- 1 root root 245, 3 Tem 21 11:58 /dev/multigeneric-chrdev3
crw-rw-rw- 1 root root 245, 4 Tem 21 11:58 /dev/multigeneric-chrdev4
```

Birden fazla aygıt ile (minör numara ile) işlem yapmanın tipik adımları şöyledir:

1) Aygıtı temsil eden bir yapı bildirilip cdev nesnesi bu yapının bir elemanı yapılır. Örneğin:

```
struct generic_device {
    /* other members */
    struct cdev cdev;
};
```

2) Her aygıt için (yani minör numara için) bu yapı türünden bir nesne static olarak ya da aygıt sürücünün init fonksiyonda dinamik olarak yaratılır. Sonra minör numaralara ilişin aygıtlar cdev_add fonksiyonuyla yapı içerisindeki cdev denesi kullanılarak yerleştirilirr. Örneğin aygıt sürücüdeki aygıtların NR_DEVICES kadar sayıda olduğunu varsayalım. Static yaratım şöyle olabilir:

```
#define NR_DEVICES      5

struct generic_device g_devices[NR_DEVICES];

static int __init generic_init(void)
{
    int i, k;
    int result;

    if ((result = alloc_chrdev_region(&g_dev, 0, NR_DEVICES, "multigeneric-chrdev")) != 0) {
        printk(KERN_INFO "register_chrdev_region: %d\n", result);
        return result;
    }

    printk(KERN_INFO "success! Major: %d, Minor: %d + %d\n", MAJOR(g_dev), MINOR(g_dev),
NR_DEVICES);

    for (i = 0; i < NR_DEVICES; ++i) {
        cdev_init(&g_devices[i].cdev, &g_generic_fops);
        g_devices[i].cdev.owner = THIS_MODULE;
        if ((result = cdev_add(&g_devices[i].cdev, g_dev, 1)) != 0) {
            printk(KERN_ALERT "cannot add character device!..\n");
            for (k = 0; k < i; ++k)
                cdev_del(&g_devices[k].cdev);
            unregister_chrdev_region(g_dev, NR_DEVICES);
            return result;
        }
    }

    return 0;
}

static void __exit generic_exit(void)
{
    int i;

    for (i = 0; i < NR_DEVICES; ++i)
        cdev_del(&g_devices[i].cdev);

    unregister_chrdev_region(g_dev, NR_DEVICES);
```

```
    printk(KERN_INFO "Goodbye World!..\n");
}
```

Aygıtlara ilişkin yapı nesneleri kmalloc fonksiyonu ile dinamik olarak da yaratılabilir. Tabii her nesne için ayrı kmalloc kullanmak yerine nesnelerin toplamı kadar alanı tek bir kmalloc çağrısı ile de tahsis edebiliriz. Örneğin:

```
#define NR_DEVICES      5

struct generic_device *g_devices;

static int __init generic_init(void)
{
    int i, k;
    int result;

    if ((result = alloc_chrdev_region(&g_dev, 0, NR_DEVICES, "multigeneric-chrdev")) != 0) {
        printk(KERN_INFO "register_chrdev_region: %d\n", result);
        return result;
    }

    printk(KERN_INFO "success! Major: %d, Minor: %d + %d\n", MAJOR(g_dev), MINOR(g_dev),
NR_DEVICES);

    if ((g_devices = kmalloc(NR_DEVICES * sizeof(struct generic_device), GFP_KERNEL)) == NULL) {
        unregister_chrdev_region(g_dev, NR_DEVICES);
        printk(KERN_ALERT "cannot aallocate memory!..\n");
        return -ENOMEM;
    }
    memset(g_devices, 0, NR_DEVICES * sizeof(struct generic_device));

    for (i = 0; i < NR_DEVICES; ++i) {
        cdev_init(&g_devices[i].cdev, &g_generic_fops);
        g_devices[i].cdev.owner = THIS_MODULE;
        if ((result = cdev_add(&g_devices[i].cdev, g_dev, 1)) != 0) {
            printk(KERN_ALERT "cannot add character device!..\n");
            for (k = 0; k < i; ++k)
                cdev_del(&g_devices[k].cdev);
            unregister_chrdev_region(g_dev, NR_DEVICES);
            return result;
        }
    }

    return 0;
}

static void __exit generic_exit(void)
{
    int i;

    for (i = 0; i < NR_DEVICES; ++i)
        cdev_del(&g_devices[i].cdev);

    unregister_chrdev_region(g_dev, NR_DEVICES);
    kfree(g_devices);

    printk(KERN_INFO "Goodbye World!..\n");
}
```

3) Aygıt sürücünün open fonksiyonunda inode elemanının i_cdev elemanından yerleştirilen aygıt nesnesinin adresi elde edilir. Bu adresten container_of makrosuyla yeteri kadar yukarı çıkılarak tanımlanan asıl aygıt yapısına erişilir. Artık open fonksiyonundan elde edilen bu aygıt adresi read ve write fonskiyonlarının tarafından erişilebilsin diye file yapısının private_data elemanına yerleştirilir. file yapısının private_data elemanı tamamen bu tür amaçlarla bulundurulmuştur. Örneğin:

```
static int generic_open(struct inode *inodep, struct file *filp)
{
    struct generic_device *dev;

    dev = container_of(inodep->i_cdev, struct generic_device, cdev);
    filp->private_data = dev;

    /* other codes */

    return 0;
}
```

4) Artık aygıt sürücünün read ve write (ve release) fonksiyonları oluşturulan aygıt nesnesine doğrudan erişebilirler. Örneğin:

```
static int generic_open(struct inode *inodep, struct file *filp)
{
    struct generic_device *dev;

    dev = container_of(inodep->i_cdev, struct generic_device, cdev);
    filp->private_data = dev;

    /* other codes */

    return 0;
}

static ssize_t generic_read(struct file *filp, char __user *buf, size_t size, loff_t *offset)
{
    struct generic_device *dev = (struct generic_device *)filp->private_data;

    /* other codes */

    return 0;
}

static ssize_t generic_write(struct file *filp, const char __user *buf, size_t size, loff_t
*offset)
{
    struct generic_device *dev = (struct generic_device *)filp->private_data;

    /* other codes */

    return 0;
}

static int generic_release(struct inode *inodep, struct file *filp)
{
    struct generic_device *dev = (struct generic_device *)filp->private_data;

    /* other codes */

    return 0;
}
```

Şimdi birden fazla minör numaraya sahip aygıt sürücülere bir örnek verelim:

```
/* multigeneric-chrdev.c  */

#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/init.h>
#include <linux/fs.h>
#include <linux/cdev.h>
#include <asm/uaccess.h>
#include <linux/string.h>
#include <linux/dcache.h>
#include <linux/slab.h>

#define BUFSIZE     50

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Kaan Aslan");
MODULE_DESCRIPTION("Generic Character Device Driver");

struct generic_device {
    char buf[BUFSIZE];
    struct cdev cdev;
};

static int generic_open(struct inode *inodep, struct file *filp);
static int generic_release(struct inode *inodep, struct file *filp);
static ssize_t generic_read(struct file *filp, char __user *buf, size_t size, loff_t *offset);
static ssize_t generic_write(struct file *filp, const char __user *buf, size_t size, loff_t
*offset);
static loff_t generic_llseek(struct file *filp, loff_t off, int whence);

static dev_t g_dev;

struct file_operations g_generic_fops = {
    .owner = THIS_MODULE,
    .open = generic_open,
    .release = generic_release,
    .read = generic_read,
    .write = generic_write,
    .llseek = generic_llseek,
};

#define NR_DEVICES      5

struct generic_device *g_devices;

static int __init generic_init(void)
{
    int i, k;
    int result;

    if ((result = alloc_chrdev_region(&g_dev, 0, NR_DEVICES, "multigeneric-chrdev")) != 0) {
        printk(KERN_INFO "register_chrdev_region: %d\n", result);
        return result;
    }

    printk(KERN_INFO "success! Major: %d, Minor: %d + %d\n", MAJOR(g_dev), MINOR(g_dev),
NR_DEVICES);

    if ((g_devices = kmalloc(NR_DEVICES * sizeof(struct generic_device), GFP_KERNEL)) == NULL) {
        unregister_chrdev_region(g_dev, NR_DEVICES);
        printk(KERN_ALERT "cannot allocate memory!..\n");
```

```
        return -ENOMEM;
    }
    memset(g_devices, 0, NR_DEVICES * sizeof(struct generic_device));

    for (i = 0; i < NR_DEVICES; ++i) {
        cdev_init(&g_devices[i].cdev, &g_generic_fops);
        g_devices[i].cdev.owner = THIS_MODULE;
        memset(g_devices[i].buf, 'a' + i, BUFSIZE);
        if ((result = cdev_add(&g_devices[i].cdev, MKDEV(MAJOR(g_dev), MINOR(g_dev) + i), 1)) !=
0) {
            printk(KERN_ALERT "cannot add character device!..\n");
            for (k = 0; k < i; ++k)
                cdev_del(&g_devices[k].cdev);
            unregister_chrdev_region(g_dev, NR_DEVICES);
            return result;
        }
    }

    return 0;
}

static int generic_open(struct inode *inodep, struct file *filp)
{
    struct generic_device *dev;

    dev = container_of(inodep->i_cdev, struct generic_device, cdev);
    filp->private_data = dev;

    /* other codes */

    return 0;
}

static ssize_t generic_read(struct file *filp, char __user *buf, size_t size, loff_t *offset)
{
    struct generic_device *dev = (struct generic_device *)filp->private_data;
    unsigned long left, n;

    left = BUFSIZE - *offset;
    n = left < size ? left : size;

    if (n) {
        if (copy_to_user(buf, dev->buf + *offset, n))
            return -EFAULT;

        *offset += n;
        printk(KERN_INFO "%lu bytes read at offset %llu\n", n, *offset);
    }

    return n;
}

static ssize_t generic_write(struct file *filp, const char __user *buf, size_t size, loff_t
*offset)
{
    struct generic_device *dev = (struct generic_device *)filp->private_data;
    unsigned long left, n;

    left = BUFSIZE - *offset;
    n = left < size ? left : size;

    if (n) {
        if (copy_from_user(dev->buf + *offset, buf, n))
```

```
            return -EFAULT;

        *offset += n;
        printk(KERN_INFO "%lu bytes written at offset %llu\n", n, *offset);
    }

    return n;
}

static loff_t generic_llseek(struct file *filp, loff_t off, int whence)
{
    loff_t newpos;

    switch(whence) {
      case 0:       /* SEEK_SET */
        newpos = off;
        break;

      case 1:       /* SEEK_CUR */
        newpos = filp->f_pos + off;
        break;

      case 2:       /* SEEK_END */
        newpos = BUFSIZE + off;
        break;

      default:      /* can't happen */
        return -EINVAL;
    }
    if (newpos < 0 || newpos > BUFSIZE)
        return -EINVAL;

    filp->f_pos = newpos;

    return newpos;
}

static int generic_release(struct inode *inodep, struct file *filp)
{
    struct generic_device *dev = (struct generic_device *)filp->private_data;

    /* other codes */

    return 0;
}

static void __exit generic_exit(void)
{
    int i;

    for (i = 0; i < NR_DEVICES; ++i)
        cdev_del(&g_devices[i].cdev);

    unregister_chrdev_region(g_dev, NR_DEVICES);
    kfree(g_devices);

    printk(KERN_INFO "Goodbye World!..\n");
}

module_init(generic_init);
module_exit(generic_exit);

/* testapp.c */
```

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>
#include <unistd.h>

void exit_sys(const char *msg);

#define NR_DEVICES      5

int main(int argc, char *argv[])
{
    int fds[NR_DEVICES];
    char path[] = "/dev/multigeneric-chrdev?";
    char buf[100 + 1];
    ssize_t result;
    int i;


    for (i = 0; i < NR_DEVICES; ++i) {
        path[strlen(path)-1] = '0' + i;
        if ((fds[i] = open(path, O_RDONLY)) == -1)
            exit_sys("open");
    }

    for (i = 0; i < NR_DEVICES; ++i) {
        if ((result = read(fds[i], buf, 100)) == -1)
            exit_sys("read");
        buf[result] = '\0';
        puts(buf);
    }

    for (i = 0; i < NR_DEVICES; ++i)
        lseek(fds[i], i, 0);

    for (i = 0; i < NR_DEVICES; ++i) {
        if ((result = read(fds[i], buf, 100)) == -1)
            exit_sys("read");
        buf[result] = '\0';
        puts(buf);
    }

    for (i = 0; i < NR_DEVICES; ++i)
        close(fds[i]);

    return 0;
}

void exit_sys(const char *msg)
{
    perror(msg);
    exit(EXIT_FAILURE);
}
```

Aygıt sürücünün yüklenmesi ve test kodunun çalıştırılması ile elde edilen çıktı şöyledir:

```
kaan@kaan-VirtualBox ~/Study/DeviceDrivers $ make file=multigeneric-chrdev
make -C /lib/modules/4.10.0-38-generic/build M=/home/kaan/Study/DeviceDrivers modules
make[1]: Entering directory '/usr/src/linux-headers-4.10.0-38-generic'
  Building modules, stage 2.
  MODPOST 1 modules
make[1]: Leaving directory '/usr/src/linux-headers-4.10.0-38-generic'
kaan@kaan-VirtualBox ~/Study/DeviceDrivers $ sudo ./loadmulti multigeneric-chrdev 5
kaan@kaan-VirtualBox ~/Study/DeviceDrivers $ gcc -o testapp testapp.c
kaan@kaan-VirtualBox ~/Study/DeviceDrivers $ ./testapp
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
bbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb
ccccccccccccccccccccccccccccccccccccccccccccccccccc
ddddddddddddddddddddddddddddddddddddddddddddddddddd
eeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeee
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
bbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb
ccccccccccccccccccccccccccccccccccccccccccccccccccc
ddddddddddddddddddddddddddddddddddddddddddddddddddd
eeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeee
kaan@kaan-VirtualBox ~/Study/DeviceDrivers $ sudo rmmod multigeneric-chrdev
```

IOCTL İşlemleri

Şimdiye kadar aygıt sürücümüzü açtık, onun üzerinde okuma yazma ve konumlandırma yaptık. Fakat aslında aygıt sürücüler üzerinde yalnızca okuma, yazma ve konumlandırmanın dışında onların belirli fonksiyonları da user mode'tan çağrılabilmektedir. Yani biz aygıt sürücümüz içerisine bazı fonksiyonlar yerleştirerek onların dışarıdan çağrılabilmesini sağlayabiliriz. Böylelikle aygıt sürücülerimiz belli işlemleri dışarıdan çok daha kolay bir biçimde yerine getirebilir.

Bir aygıt sürücünün bir fonksiyonunu çekirdek moduna geçerek çağırabilmek için ioctl isimli POSIX fonksiyonu kullanılmaktadır. ioctl fonksiyonunun prototipi şöyledir:

```
#include <sys/ioctl.h>

int ioctl(int fd, unsigned long request, ...);
```

ioctl fonksiyonunun birinci parametresi aygıt dosyasına ilişkin dosya betimleyicisini belirtir. İkinci parametre aygıt sürücü içerisindeki fonksiyonu belirten numaradır. ioctl fonksiyonunun değişken sayıda argüman alabildiğini görüyorsunuz. Anck Linux sistemlerinde ioctl fonksiyonu 2 ya da 3 argümanla çağrılabilmektedir. Eğer 3'üncü argüman girilecekse bu unsigned long int türünün uzunluğu kadar bir değer olmalıdır. 32 bit ve 64 bit Linux sistemlerinde adres bilgilerinin unsigned long uzunluğunda olduğunu anımsayınız. Fonksiyon başarı durumunda 0 değerine başarısızlık durumunda -1 değerine geri dönmektedir.

Aygıt sürücü üzerinde user mode'ta ioctl fonksiyonu çağrıldığında aygıt sürücülerin file_operations yapısı içerisindeki ioctl elemanına yerleştirilen fonksiyonları çağrılacaktır. Yeni Linux çekirdeklerinde bu elemanın ismi unlocked_ioctl yapılmıştır. ioctl ya da unlocked_ioctl elemanlarına yerleştirilecek ioctl fonksiyonunun parametrik yapısı şöyle olmalıdır:

```
long dev_unlocked_ioctl(struct file *filp, unsigned int cmd, unsigned long arg);
```

Fonksiyonun birinci parametresi aygıt sürücüsüne ilişkin file nesnesinin adresini ikinci parametresi ioctl fonksiyonunun numarasını üçüncü parametre ise aktarılan argümanı (yani ioctl POSIX fonksiyonun üçüncü argümanını) belirtmektedir. Fonksiyon başarı durumunda pozitif bir değere (genellikle 0 değerine) başarısızlık durumunda negatif hata koduna geri döner. Bu fonksiyonda en çok karşılaşılan hata kodları -EFAULT ve -ENOTTY'dir.

IOCTL fonksiyon numaraları genellikle 4 değer kombine edilerek oluşturulmaktadır. Programcılar fonksiyon numaralarının başka aygıtların numaraları ile çalışmaması için kendilerine özgü bir "magic number" kullanılar. IOCTL fonksiyon numaralarını oluşturan 4 değer şunlardır:

type: Bu 8 bitlik bir "magic number" belirtir. Genellikle programcılar bu değeri bir karakterin ASCII kodu biçiminde girerler.

number: IOCTL fonksiyonun sıra numarasıdır. 0'dan başlanarak numaralandırma yapılabilir.

direction: Bu alan yapılacak veri transferinin uygulama programına göre yönünü belirtir. Bu değer _IOC_NONE, _IOC_READ, _IOCWRITE ya da _IOC_READ|_IOCWRITE biçiminde oluşturulabilir.

size: Bu alan transfer edilecek verinin byte cinsinden uzunluğunu belirtmektedir. Bu numaraların oluşturulmasında <linux/ioctl.h> içerisinde kulçeşitli makrolar bulundurulmuştur:

```
65 #define _IOC(dir,type,nr,size) \
66         (((dir)  << _IOC_DIRSHIFT) | \
67          ((type) << _IOC_TYPESHIFT) | \
68          ((nr)   << _IOC_NRSHIFT) | \
69          ((size) << _IOC_SIZESHIFT))
70
71 #ifndef __KERNEL__
72 #define _IOC_TYPECHECK(t) (sizeof(t))
73 #endif
74
75 /* used to create numbers */
76 #define _IO(type,nr)            _IOC(_IOC_NONE,(type),(nr),0)
77 #define _IOR(type,nr,size)      _IOC(_IOC_READ,(type),(nr),(_IOC_TYPECHECK(size)))
78 #define _IOW(type,nr,size)      _IOC(_IOC_WRITE,(type),(nr),(_IOC_TYPECHECK(size)))
79 #define _IOWR(type,nr,size)     _IOC(_IOC_READ|_IOC_WRITE,(type),(nr),(_IOC_TYPECHECK(size)))
80 #define _IOR_BAD(type,nr,size)  _IOC(_IOC_READ,(type),(nr),sizeof(size))
81 #define _IOW_BAD(type,nr,size)  _IOC(_IOC_WRITE,(type),(nr),sizeof(size))
82 #define _IOWR_BAD(type,nr,size) _IOC(_IOC_READ|_IOC_WRITE,(type),(nr),sizeof(size))
83
84 /* used to decode ioctl numbers.. */
85 #define _IOC_DIR(nr)            (((nr) >> _IOC_DIRSHIFT) & _IOC_DIRMASK)
86 #define _IOC_TYPE(nr)           (((nr) >> _IOC_TYPESHIFT) & _IOC_TYPEMASK)
87 #define _IOC_NR(nr)             (((nr) >> _IOC_NRSHIFT) & _IOC_NRMASK)
88 #define _IOC_SIZE(nr)           (((nr) >> _IOC_SIZESHIFT) & _IOC_SIZEMASK)
```

Aşağıda bir IOCTL örneği görüyorsunuz: