

SİSTEM PROGRAMLAMA ve İLERİ C UYGULAMALARI

KAAN ASLAN

Bu notlar C ve Sistem Programcıları Derneği bünyesinde verilen Sistem Programlama ve İleri C Uygulamaları - I dersinde Ahmet Boşca tarafından tutulan notlardır. Notların hazırlanmasında eski notlardan da yararlanılmıştır. Notlar üzerinde düzeltmeler yapılmamıştır. Önerileriniz için mail adresim:

bozcaa@yahoo.com

C ve SİSTEM PROGRAMCILARI DERNEĞİ

Mecidiyeköy – İSTANBUL

www.csystem.org

05.04.2008

İÇİNDEKİLER

Sayfa

1. GİRİŞ.....	7
1.1. Çok Kullanılan İşletim Sistemleri.....	7
1.2. Yazılım Geliştirme Süreci	8
1.3. Standartlardaki Önemli Kavramlar	8
1.4. Standartlara Uyum ve Derleyici Mesajları.....	9
1.5. Okunabilirlik	10
1.6. Macar Notasyonu	12
2. BİÇİMSEL DİLLER ve PROGRAMLAMA DİLLERİ.....	13
2.1. Sentaks Açıklama Notasyonları	14
3. ÇEVİRİCİ PROGRAMLAR ve DERLEYİCİLER.....	14
3.1. Derleme İşlemi	15
4. PROGRAMLAMA DİLLERİNİN STANDARTLARI.....	17
4.1. C Standartları.....	17
4.2. C99 Standartlarına Eklenen Tipik Farklılıklar	18
4.3. Derleyici Eklentileri.....	19
4.4. C++ Standartları	19
5. C# ve JAVA SİSTEMLERİNDEKİ ARAKOD ÇALIŞMASI.....	19
6. BİÇİMSEL DİL.....	21
6.1. BNF Notasyonu.....	22
6.2. Programlama Dillerindeki Sentaks ve Semantik Kısıtlamalar.....	27
6.3. C 'de Operandların Yapılış Sırası.....	28
6.4. Bildirim İşlemi	29
7. FONKSİYON GÖSTERİCİLERİ (POINTER To FUNCTIONS).....	32
7.1. Fonksiyon Göstericisi Yolu ile Fonksiyonların Çağrılması.....	35
7.2. Fonksiyon Göstericilerinin Fonksiyon Parametresi Olarak Kullanılması	36
7.3. Fonksiyonların Fonksiyon Adresine Geri Dönmesi Durumu	37
7.4. Fonksiyonların Göstericilerine İlişkin Diğer Durumlar	39
7.5. Fonksiyon Göstericileri Neden Kullanılır?.....	43
7.6. Türden Bağımsız İşlem Yapan Yeni Amaçlı Fonksiyonlarda Fonksiyon Göstericilerinin Kullanılması	43
8. C 'nin STANDART TÜR İSİMLERİ.....	46
8.1. size_t Türü	46
8.2. ptrdiff_t Türü	46
8.3. time_t Türü	46
9. ÇOK KULLANILAN ÇEŞİTLİ C FONKSİYONLARI.....	47
9.1. remove Fonksiyonu	47
9.2. rename Fonksiyonu	47
9.3. mkdir Fonksiyonu	47
9.4. chdir Fonksiyonu	48
9.5. rmdir Fonksiyonu.....	48
9.6. access Fonksiyonu.....	48
9.7. Geçici Dosya Açan Fonksiyonlar	49
9.7.1. tmpfile Fonksiyonu	49
9.7.2. tmpnam Fonksiyonu	49
10. HANDLE SİSTEMLERİ	52

10.1.	Handle Sistemini Açan Fonksiyonlar	52
10.2.	Handle Sistemini Kullanan Fonksiyonlar	52
10.3.	Handle Sistemini Yok Eden Fonksiyonlar	53
11.	CACHE SİSTEMLERİ	53
11.1.	Sistem Programlama Uygulamalarında Karşılaşılan Tipik Cache Sistemleri	55
11.2.	C nin Standart Dosya Fonksiyonlarının Kullandığı Cache Mekanizması	57
12.	PROSES KAVRAMI	60
12.1.	Proseslerin Bloke Olması	63
12.2.	Preemptive ve Non-Preemptive Çok Prosesli Sistemler	64
12.3.	Modern İşletim Sistemlerindeki Koruma Mekanizması	65
12.4.	Standart Dosya Fonksiyonlarının Tamponlama İşlemlerinin Ayrıntıları	67
12.5.	Proseslerin Adres Alanı	68
13.	STDIN, STDOUT ve STDERR DOSYALARI	69
13.1.	Yönlendirme İşlemi	70
13.2.	Stderr dosyasının kullanılması	72
13.3.	Stdin, stdout ve stderr dosyalarının tamponlama mekanizmaları	73
13.4.	Stdout üzerindeki tamponlamamanın etkisi	73
13.5.	Stdin üzerindeki tamponlamamanın etkisi	74
13.6.	C nin Stdin'den Okuma Yapan Fonksiyonları	75
13.6.1.	getchar Fonksiyonu	76
13.6.2.	gets Fonksiyonu	76
13.6.3.	scanf Fonksiyonu	79
13.7.	Stdin Tamponunun Boşaltılması	82
13.8.	gets Fonksiyonu Yerine fgets Fonksiyonunun Kullanılması	84
14.	ALGORİTMA ANALİZİ	84
14.1.	Algoritmanın Karmaşıklığı	86
15.	TEMEL VERİ YAPILARI	88
15.1.	Kuyruk Veri Yapısı	889
15.1.1.	Kuyruk Veri Yapısının Kullanım Alanları	89
15.1.2.	Kuyruk Veri Yapısının Gerçekleştirilmesi	90
15.1.2.1.	Dizi Kaydırma Yöntemi ile Kuyruk Yapısının Gerçekleştirilmesi	90
15.1.2.2.	Döngüsel Yöntem ile Kuyruk Yapısının Gerçekleştirilmesi	93
15.1.2.3.	Bağlı Liste Yöntemi ile Kuyruk Yapısının Gerçekleştirilmesi	95
15.2.	Veri Yapılarının Türden Bağımsız Hale Getirilmesi	96
15.3.	Bağlı Listeler	98
15.3.1.	Bağlı Listelerin Kullanım Alanları	99
15.3.2.	Bağlı Listeler ile Dizilerin Karşılaştırılması	99
15.3.3.	Tek Bağlı ve Çift Bağlı Listeler	100
15.3.4.	Bağlı Listelerin Gerçekleştirilmesi	101
15.3.5.	Tek Bağlı Listelerin Handle Tekniği ile Gerçekleştirilmesi	102
15.3.6.	Bağlı Listenin Dolaşılması	103
15.3.7.	Bağlı Listenin Sonuna Eleman Eklenmesi	104
15.3.8.	Bağlı Listeye Eleman Insert Edilmesi	105
15.3.9.	Listeden Eleman Silinmesi	106
15.3.10.	Bağlı Listelerin Genelleştirilmesi	108
15.3.11.	Genel Bağlı Liste Oluşturulmasında Diğer Bir Yöntem	110

15.4.	Stack Sistemleri	112
15.4.1.	Stack Sistemlerinin Kullanım Alanları	112
15.4.2.	Stack Sisteminin Gerçekleştirilmesi	113
15.5.	Hash Tabloları	116
15.5.1.	Sıralı Yoklama Yöntemi (Linear Probing).....	117
15.5.2.	Hash Fonksiyonları.....	118
15.5.3.	Hash Tablolarının Kullanım Alanları.....	118
15.5.4.	Hash Tablolarının Gerçekleştirilmesi	119
15.6.	Arama İşlemlerinin Temelleri.....	121
15.7.	Ağaç Yapıları.....	123
15.7.1.	Dengelenmemiş İkili Ağacın Oluşturulması	126
16.	ÖZYİNELEMELİ ALGORİTMALAR VE KENDİ KENDİNİ ÇAĞIRAN FONKSİYONLAR	128
16.1.	Özyinelemeli Fonksiyon Örnekleri	129
17.	PROSESLER ARASINDA ALTLIK ÜSTLÜK İLİŞKİSİ	145
17.1.	Proseslerin Çevre Değişkenleri (Environment Variables)	146
17.2.	Prosesin Default Çalışma Dizini.....	1499
18.	PROSESLER ARASI HABERLEŞME	151
18.1.	Ağ Kavramı ve Tarihsel Gelişimi.....	152
18.2.	Protokol Kavramı ve Protokol Katmanları	153
18.3.	TCP/IP Protokol Ailesi	156
18.3.1.	TCP ve UDP Protokolleri	156
18.4.	Client-Server Çalışma Modeli.....	158
18.5.	Soket Arayüzü.....	159
18.6.	WinSock Arayüzü	160
18.6.1.	WinSock Arayüzü İçin İlk ve Son İşlemler.....	160
18.7.	Server Programının Organizasyonu	162
18.7.1.	socket Fonksiyonu	163
18.7.2.	bind Fonksiyonu	164
18.7.3.	listen fonksiyonu	166
18.7.4.	Accept Fonksiyonu.....	167
18.8.	Client Programın Organizasyonu	168
18.8.1.	Connect Fonksiyonu	169
18.8.2.	Server IP adresini sockaddr_in yapısına yerleştirilmesi	170
18.8.3.	Send ve Receive İşlemleri	172
18.8.4.	Soketin Kapatılması.....	174
18.9.	Client-Server Uygulamalarda Düzenli Bilgi Gönderilip Alınması	175
18.10.	Çok Clientli Uygulamalar	177
19.	THREADLER	179
19.1.	Threadli Çalışmanın Avantajları	180
19.2.	Threadlerin Bellek Alanları	180
19.3.	Threadler Üzerinde İşlemler	181
19.4.	Windows Sistemlerinde Thread İşlemleri.....	181
19.4.1.	CreateThread Fonksiyonu.....	181
19.4.2.	Threadlerin Sonlandırılması	183
19.5.	Threadlerin Senkronizasyonu	184

19.5.1.	Kritik Kodların Oluşturulması	184
19.5.2.	Windows Kernel Senkronizasyon Nesneleri	187
19.5.3.	WaitForSingleObject Fonksiyonu	187
19.6.	Event Nesnelerinin Kullanımı	188
19.7.	Üretici Tüketici Problemi	189
19.7.1.	Üretici Tüketici Probleminin Event Nesneleri İle Çözümü	189
19.7.2.	Semafor Nesneleri	190
19.7.3.	Çok Tamponlu Üretici-Tüketici Problemlerinin Semafor Nesneleri İle Çözümlemesi	192
19.8.	Mutex Nesneleri	193
19.9.	Çok İşlemcili Çalışma	194
19.10.	volatile Nesnelerin Çok Threadli Çalışmalarda Önemi	197
19.11.	Windows Sistemlerinde Thread Çizelgesi	198
20.	DERLEYİCİLERİN KOD OPTİMİZASYONU	199
20.1.	Ölü Kod Eliminasyonu (Death Code Elimination)	200
20.2.	Gereksiz Kodların Eleme Edilmesi	201
20.3.	Ortak Alt İfadelerin Eleme Edilmesi (Common Subexpression Elimination)	202
20.4.	Sabit İfade Yerleştirme (Constant Folding)	202
20.5.	Sabit İfadelerinin Yaydırılması (Const Propagation)	203
20.6.	Ortak Blok (Basic Block)	203
20.7.	Göstericilerin Eliminasyonu	204
20.8.	Inline Fonksiyon Açımı	205
20.9.	Döngü Değişmezleri (Loop Invariants)	206
20.10.	Döngü Açımı (Loop Unrolling)	207
20.11.	Döngü Ayırması (Loop Splitting)	208
20.12.	Kod Üretimi Aşamasında Yapılan Optimizasyonlar	208
20.13.	Yazmaç Tahsisatı (Register Allocation)	209
20.14.	Komut Çizelgesi (Instruction Scheduling)	209
20.15.	Derleyicilerin Optimizasyon Seçenekleri	209
20.15.1.	Microsoft Derleyicilerindeki Optimizasyon Seçenekleri	210
20.15.2.	GCC Derleyicilerindeki Optimizasyon Seçenekleri	211
21.	BELLEK İŞLEMLERİ	212
21.1.	İkincil Bellekler	214
21.2.	Disk İşlemleri	214
21.3.	Disk IO İşlemi	216
21.4.	Sektör Transferi İşleminin Programlama Yolu ile Gerçekleştirilmesi	217
21.5.	Sektörlerin Numaralandırılması	218
21.6.	İşletim Sistemlerinin Dosya İşlemlerine İlişkin Disk Organizasyonları	218
21.7.	Cluster yada Blok Kavramı	219
21.8.	Çeşitli İşletim Sistemlerinin Çeşitli Disk Organizasyonları	222
21.9.	Disk Organizasyonları Fat Türevi Dosya Sistemlerinin Disk Organizasyonları	222
21.10.	Boot Sektör	223
21.11.	Sektör okuma yazma işlemleri	227
21.12.	BPB bölümünün Elde Edilmesi	228
21.13.	Fat Bölümü	229
21.14.	Fat Elemanlarından bir catch sisteminin oluşturulması	231

21.15.	Cluster Okuma Yazma İşlemleri.....	234
21.16.	Rootdir Bölümü ve Dizin Organizasyonu.....	235
21.17.	Alt Dizinlerin Organizasyonu ve Yol İfadesinin Çözülmesi İşlemi.....	237
21.18.	Formatlama İşlemi.....	238
21.19.	Fat Dosya Sistemine İlişkin Disk Organizasyon Bozuklukları.....	239
21.20.	Disk Bölümleme Tablosu ve Disk Bölümlerinin Anlamı.....	240
21.21.	İşletim Sistemimin Yüklenmesi ve Boot İşlemi.....	243
21.22.	I-Node Dosya Sistemlerine İlişkin Disk Organizasyonun Genel Yapısı.....	245
22.	KESME İŞLEMLERİ.....	247
22.1.	IRQ Kaynakları.....	249
22.2.	IRQ Kesme Kodlarının Set Edilmesi.....	254
22.3.	Yardımcı İşlemciler.....	254
22.4.	Paralel Port Kullanımı.....	257
22.5.	Paralel Portta Veri Transferi.....	259
22.6.	Seri Haberleşme.....	261
22.7.	Hataların Belirlenmesi Üzerine Yöntemler.....	264
22.8.	assert Makrosunun Kullanımı ve Projelerin Delay ve Release Versiyonları....	265
22.9.	Değişken Syıda Parametre Alan Fonksiyonlar.....	267

1. GİRİŞ

İyi bir C programcısı en az şu özelliklere sahip olmalıdır:

Hâkimiyet: Programcı, C dilinin kurallarına hâkim olmalıdır.

Uygulama konularında deneyim: İyi bir C programcısı pek çok uygulama konusunda bilgi ve deneyim sahibi olmalıdır.

Analiz yeteneği: İyi bir C programcısı problemi iyi bir biçimde modellemeli, adımlarına ayırmalı ve kodlama sırasında ciddi problemlerle karşılaşmadan kodlamayı bitirmelidir.

Fiziksel ve ruhsal kondisyon: İyi bir C programcısı, uzun süre hatta sonuçlandırana kadar bir projede çalışabilecek kadar kondisyon sahibi olmalıdır. Bazen aynı projede aylarca çalışmak gerekebilir.

1.1. Çok Kullanılan İşletim Sistemleri

Win16 sistemleri, Windows 3.x sistemleridir ve artık neredeyse tamamen kullanımdan kalkmışlardır. Win32 sistemler ise; Windows 95, 98, ME, NT, 2000, XP 'dir. Win32 sistemleri 95 grubu ve NT grubu sistemler olmak üzere 2 gruba ayrılabilirler. Windows 95, 98, ME için 95 grubu, NT, 2000 ve XP için NT grubu işletim sistemleri denilebilir.

UNIX, AT&T Bell laboratuvarların da oluşturulmuştur. Kaynak kodları, bir araştırma projesi olarak oluşturulup serbest olarak dağıtıldığı için, çeşitli kurum ve kuruluşlar tarafından eşzamanlı olarak kullanılmış ve geliştirilmiştir. En önemlileri: Berkeley BSD, Sun Solaris, HP-UNIX sistemleri, SCO-UNIX ve Linux sistemleridir. Bu sistemlerin hepsi, farklılıklar olmasına rağmen, benzer mimariye sahiptir. UNIX sistemleri bir arabirim çerçevesinde standart hale getirilmeye çalışılmıştır. POSIX (Portable Operating System Interface for UNIX) UNIX standartlarına verilen isimdir.

1.2. Yazılım Geliştirme Süreci

Küçük yazılımlarda sezgisel yöntemler kullanılabilir ancak büyük yazılımlarda, geliştirilme sürecinin çok kişi tarafından yapıldığı durumlarda bir sistematik izlenmelidir. Genellikle sezgisel ya da bilinçli olarak izlenen sistematik aşamalardan oluşur. Bu aşamalar şunlardır:

- 1. Sistem Analizi:** Öncelikle yazılımın ne yapması gerektiği, yani yazılımdan istenenler iyi tespit edilmelidir. Bu aşamada genellikle istekler yanlış, eksik ya da belirsiz bir biçimde belirlenir.
- 2. Belirlemelerin Yapılması:** Sistemin analizi yapıldıktan sonra istenen şeylerin spesifikasyon biçiminde resmi ya da yarı resmi olarak belirtilmesi gerekir.
- 3. Tasarım:** Bu aşamada yazılımın tasarımı yapılır yani proje hangi modüllerden oluşacaktır, bu modüller ne iş yapacaktır, hangi programlama dil ve araçları kullanılacaktır gibi sorular cevaplanır.
- 4. Kodlama:** Bu aşamada, tasarımı yapılmış olan yazılımın gerçek kodlaması yapılır.

5. **Test:** Yazılımın aşağı seviyeli ve yüksek seviyeli testleri kodlama aşamasına paralel olarak yürütülür.
6. **Bakım:** Yazılım kullanılmaya başladıktan sonra çeşitli kısmi geliştirmeler, düzeltmeler yapılabilir ve kullanım hizmetleri verilebilir

1.3. Standartlardaki Önemli Kavramlar

Implementation-Defined (Implementation-Dependent) Behavior: Bir durum standartlarda bu şekilde belirtilmişse, bu konu derleyicileri hazırlayanların inisiyatifine bırakılmıştır. Bu durumlar derleyiciyi yazanlar tarafından belgelenmek zorundadır. Örneğin “int” türünün uzunluğu implementation dependent biçimindedir. Minimum limitleri verilmiştir ama gerçek uzunluk derleyiciyi yazanlara bırakılmıştır. Derleyiciyi hazırlayanlar bu seçimi nasıl yaptıklarını belirtmek zorundadırlar.

Undefined Behavior: Yorumlanması derleyiciden derleyiciye değişebilen ve genellikle anlamsız olan durumları belirtir. Bu durumlara karşı derleyicinin davranışı standartlarda undefined behavior olarak belirtilmiştir. Derleyiciyi hazırlayanlar bu tür işlemleri nasıl yorumladıklarını belgelemek zorunda değildirler. Örneğin:

```
result = ++a + ++a;
```

Tipik bir undefined behavior durumudur.

Unspecified Behavior: “Implementation-dependent behavior” a benzer, birkaç olasılık söz konusuysen, derleyicinin hangi olasılığını tercih ettiğinin belirsiz olduğu durumdur. Olasılıklar sınırlıdır, seçeneklerin hepsi mantıklıdır, ancak derleyiciyi hazırlayanlar seçimlerini belgelemek zorunda değildirler.

Diagnostic Message: Standartlara göre belirtilen sentaks ve semantik kurallarına uymayan durumlar derleyici tarafından tespit edilerek bir mesaj olarak gösterilmelidir. Bu mesaja diagnostic message denir. Standartlarda, mesajın ne olması gerektiği, yani uyarı ya error olması gerektiği belirtilmemiştir. Yalnızca bu tür durumlarda bir mesaj vermesi gerektiği belirtilmiştir.

1.4. Standartlara Uyum ve Derleyici Mesajları

C90 ve C99 standartlarında C programlama dilinin kuralları sentaks ve semantik bakımından açıklanmıştır. Bu standartlar, bir programın kurallara uymaması durumunda derleyicinin hangi kuralın ihlal edildiğine dair bir teşhis mesajı vermesi gerektiğini belirtmişlerdir. C90 ve C99’da yanlış yazılmış bir programın derlenmemesi gerektiği veya doğru yazılmış bir programın da derlenmesi gerektiği hakkında bir şey söylenmemiştir. Yalnızca kurallara uyulmaması halinde derleyicinin bir teşhis mesajı verilmesi gerektiği belirtilmiştir. Bu durumda standartlara uygun bir derleyici kurallara uyulmadığı durumda, işlemi uyarı ile geçiştirerek derleme işlemini yapabilir ya da “error” vererek derleme işlemini yapmayabilir. Normalde derleyicinin “warning” ve “error” mesajları geçersiz kodlarda kullanılmaktadır. Örneğin bir göstericiyi doğrudan bir int değişkene atamak geçersiz bir durumdur. Hemen hemen tüm derleyiciler bu durumda warning verseler de aslında derleme işlemini yapmayabilirler. Bir adresin farklı türden bir göstericiye doğrudan atanması standartlarca geçerli olmadığına göre derleyicinin vermesi gereken mesaj uyarı veya “error” olabilir

```
char s[10];  
int *pi = s;
```

Aslında uygulamada bazı mesajlar tipik uyarı olarak değerlendirmektedir.

C++ standartlarında, kurallara uyulmaması durumunda bir mesajın yanı sıra programın da derlenmemesi gerektiği belirtilmiştir. Böylece C de uyarı ile geçiştirilen pek çok durum C++’da “error” durumuna yükseltilmiştir

1.5. Okunabilirlik

Bir programa bakıldığında ne yapılmak istendiğinin kolay biçimde anlaşılmasına okunabilirlik denir. Okunabilirliği sağlamak büyük ölçüde programcının sorumluluğundadır. Okunabilirliği sağlamak için dikkat edilecek temel noktalar şunlardır.

1. Kodun genel yerleşimi (tablamalar, indentler) düzgün ve tutarlı olmalıdır. Tablama konusunda çeşitli yaygın kullanılan biçimler vardır.
2. Programın küçük mantıksal ölümleri satır boşlukları verilerek bloklanabilir.

3. Kaynak kodun gerekli yerlerine açıklamalar yerleştirilmelidir. Açıklamalar kısa cümleler biçiminde olmalı ve kritik noktalara yerleştirilmelidir. Açıklamalar küçük soru cümleleri biçiminde oluşturulabilir veya blokların başlarına yerleştirilebilir.
4. Başlık dosyalarının ve kaynak dosyalarının başına başlık kısmı yerleştirilmelidir. Örnek bir başlık kısmı şu şekilde olabilir:

```
/* **** */

FILE           : general.h
AUTHOR        : Kaan ASLAN
LAST UPDATE   : 22/11/2002
PLATFORM      : Any
TRANSLATOR    : Any

This is the general header file prepared for project groups of the C
and System Programmers Association (CSD)

Copyleft (c) 1993 by C and System Programmers Association (CSD)
All Rights Free

**** */
```

5. Program içerisindeki sabitler anlamlı bir biçimde sembolik sabitler biçiminde ifade edilmelidir. Örneğin:

```
if (n == PERSONELSAYISI) {
    /*****/
}
```

6. Değişkenler anlamlı bir biçimde isimlendirilmelidir. Değişken isimleri hangi konuya ilişkin olduğu hakkında fikir vermelidir. İsimlendirmede tutarlılık olmalıdır. Programın farklı bölümlerinde aynı isimler aynı anlamı ifade edecek biçimde kullanılmalıdır
7. Kullanılan tür isimleri konuya uygun bir biçimde seçilebilir yani tür isimleri de onların hangi konuda kullanıldığını açıklayacak biçimde olabilir.

1.6. Macar Notasyonu

Macar notasyonu, temel olarak bir deęişkenin isimlendirilmesine ilişkin kurallar içeren bir belirlemeler topluluęudur. Charles Simone tarafından oluşturulmuştur. Microsoft API programlama sisteminde ve SDK (software development kit) içerisinde bu notasyonu kullanılmaktadır. Macar notasyonunun en önemli özellięi deęişken isimlerinin, deęişkenin türünü belirten bir önekle başlatılmasıdır. Macar notasyonunun ana noktaları şunlardır.

Fonksiyon isimleri, her sözcüğün ilk harfi büyük olacak şekilde önce bir fiil, sonra nesnesi gelecek şekilde isimlendirilir.

Her deęişken, hangi türden olduğunu belirten öneklerle başlatılır. Önekler küçük harflerden oluşturulur, öneklerden sonra her sözcüğün ilk harfi büyük yazılır.

Önek	Tür	Önek	Tür
c	char	p	pointer
l	long	pc	char *
d	double	pl	long *
s	short	pv	void *
f	float	psz (zero terminated string)	Yazı gösteren char *
u	unsigned int	b	BYTE
b	BOOL	w	WORD
f	flag	dw	DWORD

Örneęin:

```
DWORD dwMaxSector;  
long lResult;  
BOOL bValid;  
PSTR pszName;
```

```
PVOID pvBlock;
```

Maalesef Macar notasyonunda isimler uzama eğilimindedir. int türden değişkenler, dizi isimleri, yapı isimleri genellikle önek almazlar, bunun yerine ilk sözcüğün tamamı küçük harflerle yazılır, varsa sonraki sözcüklerin ilk harfleri büyük yazılır. Yapı türünden nesnelere bazen yapının türüne ilişkin küçük önekler alırlar. Örneğin:

```
RECT rectWindowPos;  
POINT ptLeftTop;
```

Genellikle global değişkenler g_, statik değişkenler s_ ve C++ dilinde sınıfın veri elemanları m_ ile başlayarak isimlendirilirler. Bu eklerden sonra tür belirten bir önek daha getirilir.

Macar notasyonu çok katı uygulandığında bazen uzun ve can sıkıcı hale gelebilir. Programcılar bazı gevşetmeler yapılabilir. Örneğin bazı önemli değişkenlerin önüne tür belirten önekler getirilir. Macar notasyonu ile yazılmış bir örnek program:

```
#include <stdio.h>  
#include <general.h>  
  
PVOID CopyMemory(PVOID pvDest, PCVOID pvSource, DWORD dwSize)  
{  
    PBYTE pDest = (PBYTE) pvDest;  
    PBYTE pSource = (PBYTE) pvSource;  
  
    while (dwSize-- > 0)  
        *pDest++ = *pSource++;  
    return pvDest;  
}  
  
int main(void)  
{  
    char szCity1[50] = "Ankara";  
    char szCity2[50];  
  
    CopyMemory(szCity2, szCity1, strlen(szCity1) + 1);  
    puts(szCity2);  
  
    return 0;  
}
```

2. BİÇİMSEL DİLLER ve PROGRAMLAMA DİLLERİ

Doğal diller ile programlama dilleri birbirine çok benzer. Bir dilin tüm kurallarına gramer denir.

Gramer = Sentaks + Semantik + Morfoloji + Etimoloji

Sentaks, doğru yazılım kurallarıdır. Semantik, doğru yazılmış olan yazıların ne anlama geldiği hakkındaki kurallardır. Sentaks kuralları matematiksel formüller ile açıklanıyor ise bu dillere biçimsel (formal) diller denir. Programlama dilleri biçimsel diller grubuna girer. Doğal diller çok fazla iki anlamlı öğeler içerdiğinden dolayı kesin bir sentaks ifadesi matematiksel bakımdan mümkün değildir. Biçimsel diller üzerinde en önemli çalışmalar Noam Chomsky tarafından yapılmıştır (50’li yıllarda).

BNF à Sentaks ifade etmek için kullanılır.

2.1. Sentaks Açıklama Notasyonları

Programlama dillerinin çoğu Chomsky kategorisine göre bağlam-bağımsız(context-free) dillerdir. Bağlam-bağımsız dillerin sentaks gösterimi için çeşitli yöntemler önerilmiştir. En çok kullanılan BNF (Backus Nover Form) denilen yöntemdir. ISO bunu EBNF(Extended BNF) olarak standardize etmiştir.

Anahtar Notlar:

C derslerinde [] ve < > tekniği kullanıldı. < > içerisinde bulunan öğeler kesinlikle bulunması gereken öğelerdir. [] içerisinde bulunanlar isteğe bağlı öğeleri belirtir.

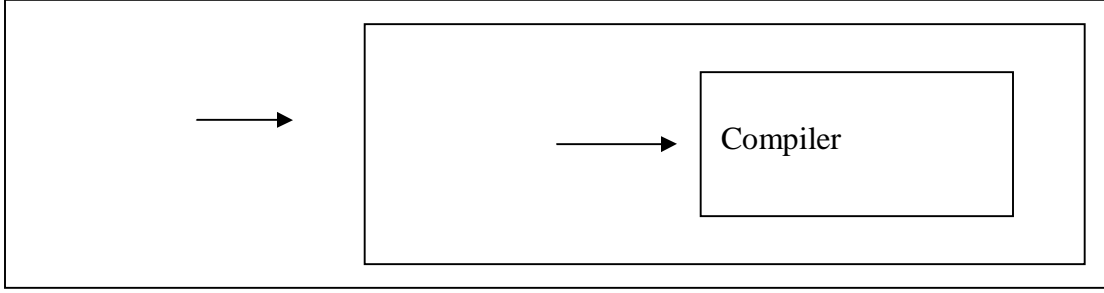
```
if (<ifade>
    <deyim>
[else
    <deyim> ]
```

Anahtar Notlar:

“Formal Language”, “Auto Theory” ismi verilen ana konular biçimsel dillerin teorik temelini konu almaktadır.

3. ÇEVİRİCİ PROGRAMLAR ve DERLEYİCİLER

Bir dilde yazılmış bir programı eşdeğer olarak başka bir dilde yazılmış programa dönüştüren programa çevirici program denir. Bir çevirici programda hedef dil aşağı seviyeli bir dil (saf makine dili, ara kod, sembolik makine dili) ise bu çeşit çeviri programlara derleyici (compiler) denir. Derleyiciler bir çeşit çevirici programlardır. Bir programlama dilinde yazılmış program üzerinde herhangi bir işlem uygulayan programlara dil işlemcileri(language processors) denir.



Başka bir sistem için kod üreten derleyicilere çapraz derleyici (cross compiler) ismi verilir. Örneğin PC 'de kod yazıp daha sonra PIC 'lerde çalıştırdığımız kodları derleyen derleyiciler bu gruba girer.

Bazı dil işlemcileri kaynak kodu okuyup hiç hedef kod üretmeden o anda yorumlayıp çalıştırabilmektedir. Bu tür programlara yorumlayıcı (interpreter) denir. BASIC, Lisp, AWK gibi dillere yönelik işlemciler yorumlayıcı yöntemler ile çalışmaktadır. Yorumlayıcıların derleyicilere göre çok daha kolay yazılması gibi bir avantajı bulunurken, programları daha yavaş çalıştırması gibi bir dezavantajı bulunur. Programların hızlı çalışmasının önemli bir faktör olmadığı durumlarda yorumlayıcılar kullanılır. Yorumlayıcılar birer çevirici program değildir.

Anahtar Notlar:

IDE yazılım geliştirmeyi kolaylaştıran, kendi içerisinde editörü, menüleri ve çeşitli araçları olan yardımcı programlardır. İlk IDE yazan firma Borland tır(Borland 1.0). Microsoft 'un IDE 'si "Visual Studio" isimli üründür. KDevelop Linux 'ta kullanılan bir IDE 'dir. Eclipse, Netbeans, Dev-C++ IDE 'lerini de sık kullanılan IDE 'ler arasında sayabiliriz.

3.1. Derleme İşlemi

Derleyici çeşitli modüllerin birleşiminden oluşan bir program olarak düşünülebilir.

Derleme işlemi sırasında ilk aşama kaynak kodların atomlarına(tokens) ayrılmasıdır(lexical analysis, tokenizing).

Atomsal analiz sonucunda elde edilen atomlar sentaks analizi yapılarak modüle yollanır. Bu modül gelen atomlara bakarak programın geçerli olup olmadığına bakar. Derleyicinin bu kısmına parser (syntax analyzer) denir. Sentaks analizi sırasında program gramere uygun değilse hata mesajları üretilir. Sentaks analizi sırasında derleyici programı işlenebilir veri yapısına dönüştürür. Buna ayrıştırma ağacı(pars tree) denilmektedir.

Sentaks analizinden sonra semantik analiz uygulanır. Doğru yazılmış her program(sentaksa göre) geçerli değildir. Semantik analiz aşamasında çeşitli semantik kurallara uygunluk kontrol edilir. Örneğin aynı isimli değişkenlerin tanımlanmaması, bir fonksiyonun aynı miktarda girdi ile çağrılması, bir göstericiye aynı türden adres atanması semantik kuraldır.

Bu üç modül (lexical analysis, syntax analysis, semantic analysis), derleyicinin ön yüzü(front end) 'dür. Ön yüz genel anlamada analiz işi yapar.

Pek çok derleyicide bir sonraki modülde ara kod üretimi yapılır. Ara kod ayrıştırma ağacının hedef koda daha yakın bir biçimdir. (Ara-kod Üretimi, Intermediate Code Generation)

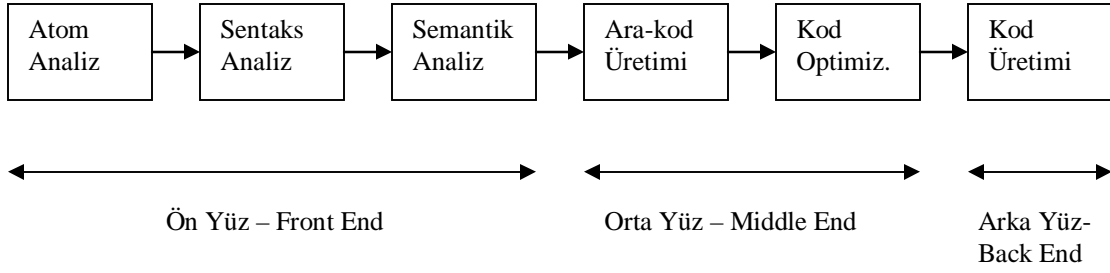
Derleyicinin bir sonraki modülünde programcının yazdığı kod iyileştirilir. Buna kod optimizasyonu (code optimization) denir.

Derleyicinin son modülünde iyileştirilmiş koda bakılarak hedef kod üretilir. (code generation)

Derleyicilerin ön yüzleri analiz arka yüzleri senteze yöneliktir. Derleyicilerin modüllere ayrılmasından önemli faydalar sağlanmıştır. Örneğin M tane dil için N tane işlemciye yönelik derleyici yazılmak istensin. Normalde $M*N$ tane derleyici yazılması gerekirken bu modüllerin kullanımı sayesinde $M+N$ tane derleyici yeterli olur. Her dil için bir ön yüz ve her bir işlemci için bir arka yüz yazmak yeterli olur.

Derleyici piyasasında kod optimizasyonu ve üretimi en önemli unsurdur. Bu yüzden bu modülleri iyi tasarlanan derleyiciler müşteriler tarafından daha çok rağbet görür.

Derleyiciler çok iyi test edilmelidir. Çünkü diğer sistem programları da bunlar kullanılarak oluşturulmaktadır.



Derleme Sürecine İlişkin Anahtar Sözcükler:

Translator, Compiler, Language Processor, Lexical Analysis, Code Optimization, Code Generation, Pars Tree, Parsing, Front End, Back End, Compiler Parting, Cross Compiler, Compiler Development Project, gcc, lcc, tcc

Anahtar Notlar:

Assembler monte eden anlamına gelir. Kaynak kodu sembolik makine dili, hedef kodu saf makine dili olan çevirici programlara sembolik makine dili derleyicisi(assembler) denir.

4. PROGRAMLAMA DİLLERİNİN STANDARTLARI

Standartlar bir dilin gramerini anlatan resmi dokümanlardır. Standartlar standardizasyon kurumları tarafından oluşturulabildiği gibi dili tasarlayan kişiler tarafından ya da şirketler tarafından da oluşturulabilir.

Standartlarda dilin sentaks kuralları genelde BNF ya da türevleri ile ifade edilmektedir. Semantik kurallar hukuk diline benzeyen cümlelerle açıklanır.

4.1. C Standartları

C programlama dili 1970 – 71 yıllarında Dennis Ritchie tarafından, Unix tabanlı işletim sistemleri üzerinde sistem programlama uygulamaları geliştirme aracı olarak kullanılmak üzere oluşturuldu. 1973 yılında UNIX, C ile tekrar yazıldı. C 'nin bir programlama dili olarak yaygınlaşması ise 1980 – 1990 arasında olmuştur. ANSI, 1983'te başladığı C 'yi standartlaştırma çalışmalarını 1989'da tamamladı. Buna paralel, 1990'da ISO tarafından da standartlaştırıldı. Bu standart, ISO/IEC 9899:1990'dır ve kısaca C90 olarak bilinir.

C en son, ISO tarafından 1999 yılında standartlaştırılmıştır. Bu standardın kodu 9899:1999'dur ve kısaca C99 denir. Maalesef, bu son sürüm henüz derleyiciler tarafından tam olarak desteklenmemektedir. C99'a C++'tan özenilerek bazı özellikler eklenmiştir.

Standart C fonksiyonlarının neler olduğu standartlarda belirlenmiştir. Ancak derleyiciler, bu fonksiyonların dışında pek çok özel fonksiyon bulundurabilirler. Bu fonksiyonlardan bazıları derleyicilerin çoğunda bulunmaktadır. (itoa(),strupr(),strlwr()...gibi.)

4.2. C99 Standartlarına Eklenen Tipik Farklılıklar

Yerel diziler değişken uzunlukta olabilir. Yerel değişkenler bloğun herhangi bir yerinde bildirilebilir.

```
int n;  
scanf("%d", &n);  
int a[n];
```

// ile satır sonuna kadar yorumlama yapma standartlaştırılmıştır. Normalde C90 standartlarında yorum satırları için sadece `/**/` kullanılabilir.

long long ve `_Bool` türleri oluşturulmuştur (16 byte).

restrict göstericiler kavramı vardır.

Yapılara ve dizilere ilk değer verirken ayrık elemanlara değer atanabilir.

```
struct SAMPLE s = {.a = 10, .c = 20};
```

Fonksiyonların geri dönüş değeri türü belirtme zorunlu hale getirilmiştir. (implicit int kuralı kalkmıştır.)

Dizi ve yapı türüne tür dönüştürmesi uygulanabilmektedir.

```
struct sample s;  
s = (struct sample) {1, 2, 3};
```

Yeni standart C fonksiyonları eklenmiştir.

Önişlemci komutları iyileştirilmiştir.

Inline fonksiyonlar C99 'a eklenmiştir.

4.3. Derleyici Eklentileri

Bir C derleyicisi standartlara uygun olması yanı sıra ek bir takım özelliklere sahip olabilir. Bu özelliklere derleyicilerin eklentileri (extensions) denilmektedir. Pek çok C derleyicisi ek birtakım kütüphanelere ve anahtar sözcüklere sahiptir. Eğer bu eklentiler geçerli bir programı geçersiz hale getirmiyorsa derleyici standartlara uygundur. Aksi durumda standartlara uygun değildir.

Anahtar Notlar:

C 'de başı _ ile başlayan global faaliyet alanındaki tüm isimler, başı __ ile başlayan yerel ve global faaliyet alanındaki tüm isimler ve başı _ ve sonraki ilk harfi büyük olan tüm isimler programcının kullanımı için yasaklanmıştır. Derleyiciler eklentilerini bu isimlerle seçmektedirler.

Eğer programcı bu isimleri kullanırsa tanımsız davranış (undefined behavior) olur. Pek çok derleyicide eklentiler çeşitli seçeneklerle pasif hale getirilmektedir. Eğer eklentiler aktifken standartlara uygunsuzluk çıkarıyor, fakat pasif iken çıkarmıyorsa bu derleyiciye standartlara uygundur diyebiliriz.

Derleyicilerin eklentilerini kullanmak taşınabilirliği azaltır. Örneğin Linux işletim sistemi gcc derleyicisinin pek çok eklentisini kullanmaktadır. Bu durumdan dolayı Linux gcc dışında bir derleyici ile derlenememektedir.

4.4. C++ Standartları

C++ standartları 1998 yılında oluşturuldu. ISO/IEC 14482:1998 kod ismi ile bilinir. 2003 yılında dil üzerinde bir takım düzeltmeler yapılmış, dile yeni kurallar eklenmemiştir. Bu standartlara ISO/IEC 14882:2003 kod ismi verilmiştir. Son standart budur.

C++ için 2008 yılı içerisinde bitirilmesi hedeflenmiş olan yeni standartlar üzerinde çalışılmaktadır.

5. C# ve JAVA SİSTEMLERİNDEKİ ARAKOD ÇALIŞMASI

C# ve Java çalışma ortamlarındaki derleyiciler doğrudan makine kodları üretmemektedir. Hiçbir işlemcinin makine kodu olmayan ve ismine ara kod denilen sahte makine kodları üretmektedir. Örneğin C# 'ta bir program derlendiğinde belirli bir link aşaması olmadan doğrudan EXE uzantılı bir dosya elde edilir. Fakat bunun içerisinde Intel makine kodları değil sahte makine kodları bulunmaktadır. .NET ve Mono ortamında C# derleyicisinin oluşturduğu ara koda ILASM (Intermediate Language ASM) denilmektedir. Java dünyasında "Java Byte Code" olarak isimlendirilmektedir. Gerek .NET, gerekse Java dünyasında bu ara kodlar doğrudan çalıştırılmaz. Bir çalışma ortamının bu kodu yorumlayarak çalıştırması gerekir. Bu çalışma ortamına .NET dünyasında CLR(Common Language Routine), Java dünyasında JVM(Java Virtual Machine) denir.

Programın yorumlanarak başka bir yazılım tarafından çalıştırılması performans kaybına yol açmaktadır. Bunun yanı sıra ara kod ile çalışmanın önemli faydaları da bulunur.

- Binary Portability: Çalıştırılabilen kodun taşınabilirliği sağlanmış olur. Böylece bir program başka sistemlere götürülerek çalıştırılabilir. Tabi çalışma ortamının o sistemde kurulu olması gerekir.
- Language Interoperability: Ara kodlu çalışma, diller arasında entegrasyona iyi bir biçimde destek vermektedir. Örneğin .NET ortamında biz bir projeye C# ile başlayıp, projeyi VB.net ile devam ettirip, C++.net ile sonlandırabiliriz. Bir programda yazılmış kodların başka bir kodla kullanılması dual code temelinde çok zordur. Bunun için Microsoft Com yöntemini tasarladıysa da bu yöntem etkin olmaktan uzaktır.
- Kodların yorumlanarak çalışması daha güvenli bir çalışma sunmaktadır.

JIT: Çalıştırma ortamının ara kodu yorumlaması nasıl yapılmaktadır?

Ara kod komut komut yorumlanmaz. Fonksiyon fonksiyon yorumlanmaktadır. Bir fonksiyon ilk çağrıldığında fonksiyon doğal koda dönüştürülür ve daha sonra çalıştırılır. Dönüştürülen fonksiyon bir “cash” sisteminde biriktirilir. Bu çalıştırma sürecine JIT(Just In Time Compilation) denir.

6. BİÇİMSEL DİL

Teorik bakımdan biçimsel bir dil, ismine terminal semboller denilen bir kümenin elemanlarından oluşturulan bir kümedir. Terminal semboller Σ sembolü ile ifade edilir. Dil ise genellikle L sembolü ile ifade edilir.

Örneğin:

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

İşte terminal sembollerinin istenildiği kadar birleştirilmesi ile elde edilen kümelere dil denilmektedir.

Örneğin:

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

$$L1 = \{aaaa, bbab, ac\}$$

$$L2 = \{aaaaa, bbbbb, c\}$$

$$L3 = \{a, ab, ba, aba, bab, aaab, baaa\}$$

Bir dildeki terminal semboller sınırlı olmak zorundadır. Fakat dil içerisindeki elemanlar sonsuz sayıda olabilir. Bu durumda elemanları tek tek küme parantezi içerisinde yazmak mümkün olmaz.

Örneğin:

C bir dildir ve sonsuz sayıda C programı yazılabilir. C programı için Σ kümesi, C programında kullandığımız tüm karakterlerdir. C dili de tüm geçerli C programlarının kümesidir. Biz C dilini küme parantezleri içerisinde yazmaya çalışsak, geçerli tüm C programları bu küme parantezi içerisine yerleştirilmesi gerekir.

$$L = \{CP1, CP2, CP3, \dots\}$$

Tüm C programları sonsuz sayıda olduğuna göre C dilini bu şekilde tanımlamak mantıklı değildir.

Basit bir dil küme parantezi ile yazılabilir. Fakat bir programlama dili için başka yöntemler bulunmalıdır. İşte BNF ya da EBNF gibi notasyonlar temel olarak yukarıdaki gibi bir L kümesini tanımlamakta kullanılmaktadır. Biçimsel dillerin bu konu ile uğraşan bölümüne “Generate Grammar” diyoruz. İlk kurumsal çalışmalar Chomsky tarafından yapılmış ve diller 5 gruba ayrılmıştır.

6.1. BNF Notasyonu

BNF notasyonu özellikle bağlam-bağımsız (context-free) dilleri üretebilmek için düşünülmüş olan bir yöntemdir. Bu yöntemde bir kök sembol, bir grup ana sembol ve bir grup terminal sembol kullanılır. Kök sembolden başlanarak çeşitli üretim kuralları ile terminal sembollere kadar inilir. İşte dil kök sembolden başlanarak oluşturulan tüm kümeyi kapsamaktadır.

Ara sembol kuralları “:” ile açıklanır. Veyalar ayrı satırlara yazılabilir ya da aynı satırda “|” sembolü ile ayrıştırılabilir. Terminal semboller genellikle tek tırnak içerisinde yazılır. Fakat birçok programlama dilinin standardı koyu veya italik yazım kullanır. Kök sembol ara semboller kümesi içerisinde bulunan bir semboldür. Dilin tüm elemanları kök sembolden başlanarak ara sembollerden geçilerek bütün ara sembollerden kurtularak elde edilen terminal semboller kümesidir.

Örneğin:

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

$$V = \{X, Y, Z\}$$

$$S = \{X\}$$

```
X: YZ
Y: 'a' | Y'a'
Z: 'b'
```

Y sembolünden hareketle dilin birkaç elemanını üretelim:

```
X → YZ → Y 'a' → Y 'a' 'b' → Y 'a' 'a' 'b' → 'a' 'a' 'a' 'b' → aaab
```

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

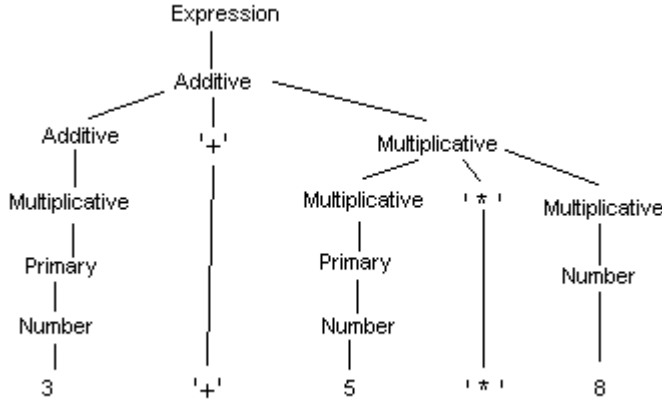
Burada kök sembol number, terminal semboller 0,1,...,9 karakterleridir. Bu durumda 7, 17, 997, 49, 017 bu dilin elemanıdır.

Ödev: Number isimli bir BNF gramerini yazınız. Fakat başı 0 ile başlayan sayılar bu dilin elemanı değildir. (0 bu dilin elemanıdır.)

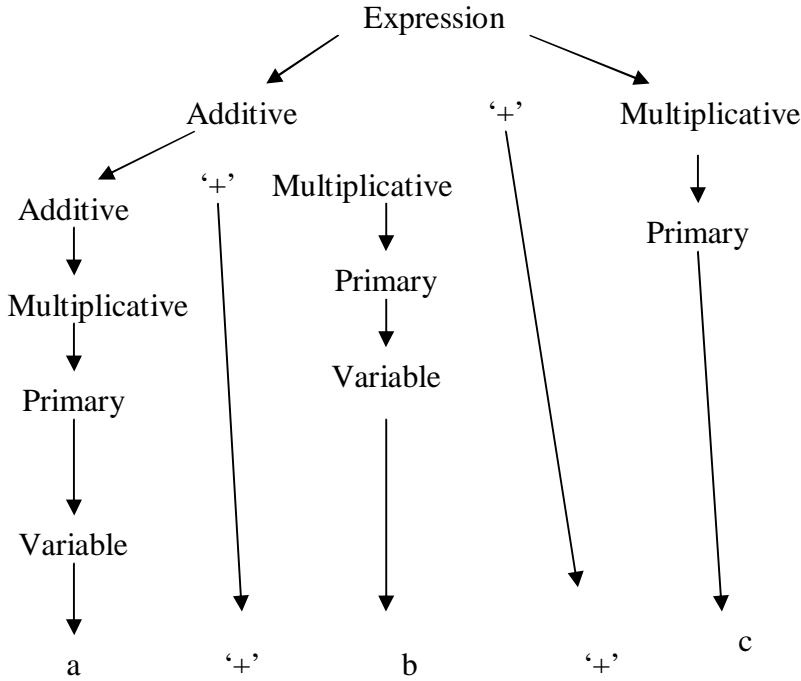
Örnek olarak “expression” isimli bir dili BNF notasyonu ile tanımlamaya çalışalım.

```
Expression:
    Additive - Expression
Additive:
    Additive '+' Multiplicative | Additive '-' Multiplicative |
    Multiplicative
Multiplicative:
    Multiplicative '*' Primary | Multiplicative '/' Primary | Primary
Primary:
    Number | Variable
```

Soru: $3 + 5 * 8$ ifadesi expression dilinin bir elemanı mıdır?



Soru: $a + b + c$ bu dilin bir elemanı mıdır? Yanıt evet



Soru: $a * b * c$ bu dilin bir elemanı mıdır? Yanıt evet.

Görüldüğü gibi verilen dilde '*' ve '/' işleminin + ve - işlemine göre önceliği vardır. Bu öncelik gramerden çıkan doğal bir sonuçtur. Gerçekten de C standartlarında operatör önceliğinden bahsedilmez. Zaten BNF gramerinden çıkan doğal sonuç öncelik tablosunu oluşturmaktadır. Şimdi yukarıda expression dilini paranteze öncelik verecek şekilde geliştirelim:

Expression:
Additive

Additive:

Additive '+' Multiplicative | Additive '-' Multiplicative |
Multiplicative

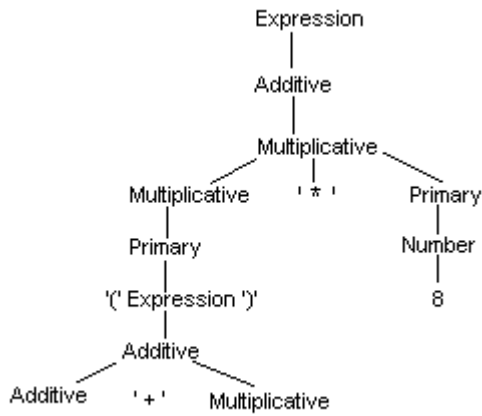
Multiplicative:

Multiplicative '*' Primary | Multiplicative '/' Primary | Primary

Primary:

Number | Variable | '('Expression')'

Soru: $(3 + 5) * 8$ bu dilin bir elemanı mıdır? Evet.



Soru: $(3 + 5) * (8 - 2)$ Expression dilinin bir elemanı mıdır?

Evet elemanıdır. Burada “primary * primary” durumuna gelinebilir. Sonra “primary” kuralı $3 + 5$ ve $8 - 2$ biçiminde açılabilir. Şimdi yukarıda expression dilini ‘!’ operatörüne öncelik verecek şekilde geliştirelim:

Expression:

Additive

Additive:

Additive '+' Multiplicative | Additive '-' Multiplicative |
Multiplicative

Multiplicative:

Multiplicative '*' Primary | Multiplicative '/' Primary | Unary

Unary:

```
'!' Primary | Primary
```

Primary:

```
Number | Variable | '('Expression')
```

Soru: $!(a + b) * c$ “Expression” dilinin bir elemanı mıdır?

Evet elemanıdır. Burada görüldüğü gibi “a + b” ‘nin sonucu ‘!’ operatörüne sokulup “c” ile çarpılmıştır.

Soru: $!!!a$ Expression dilinin bir elemanı mıdır? Hayır değildir.

Expression dilinin bunu açabilmesi için aşağıdaki gibi bir düzeltme yapmamız gerekir:

Unary:

```
'!' Unary | Primary
```

Görüldüğü gibi soldan sağa ya da sağdan solalığa da BNF notasyonundan çıkartılan doğal bir sonuçtur. $a+b+c$ işlemi $a+b$ ile c nin toplanacağı anlamına gelir. Çünkü bunu a ile b nin toplanacağı biçiminde açamayız. Yukarıdaki gramere işaret ‘-’ ve işaret ‘+’ operatörleri de eklenebilir.

Unary:

```
'+' Unary | '-' Unary | '!' Unary | Primary
```

Şimdi bu gramere parametrelili fonksiyon çağırma ekleyelim.

Expression:

```
Additive
```

Additive:

```
Additive '+' Multiplicative | Additive '-' Multiplicative |  
Multiplicative
```

Multiplicative:

```
Multiplicative '*' Primary | Multiplicative '/' Primary | Unary
```

Unary:

```
'+' Unary | '-' Unary | '!' Primary | Primary
```

Primary:

```
Number | Variable | '('Expression')' |
```

```
Variable '(' Arguman_List ')'
```

Argument_List:

Expression, Expression ',', Argument_List

Soru: !Foo(3,5) +Bar(a,b) ifadesi expression dilinin bir elemanı mıdır? Yanıt evet.

Soru: Func(a+5, Bar(x,y)) ifadesi expression dilinin bir elemanı mıdır? Yanıt evet.

Yukarıdaki dil oldukça bilinçli oluşturulmuştur. Hâlbuki bazen yanlışlıkla bir ifade birden fazla yoldan geçilerek açılabilir. Böyle gramerlere iki anlamlı (ambiguous) gramer diyoruz.

Expression:

Additive

Additive:

Additive '+' Additive | Additive '-' Additive | Multiplicative

Multiplicative:

Multiplicative '*' Multiplicative |

Multiplicative '/' Multiplicative | Primary

Primary:

Number | Variable

Bu gramerde “a + b * c” gibi bir ifade “a + b” nin sonucu c ile çarpılacak ya da a ile “b * c” nin sonucu toplanacak şekilde açılabilir. O halde gramer iki anlamlıdır ve bir işe yaramaz.

6.2. Programlama Dillerindeki Sentaks ve Semantik Kısıtlamalar

Bir programlama dillerindeki geçerli tüm programları BNF notasyonu ile açıklamak mümkün olmayabilir. BNF yalnızca sentaks açısından geçerliliği açıklamaya yeterlidir. Sentaks geçerliliğinden sonra semantik kuralları da kontrol etmek gerekir.

Örneğin aşağıdaki C programı standartlarda belirtilen C kurallarına uyar. Fakat semantik kurallara uymamaktadır.

```
int main()
{
    int a, a;

    /*****/

    return 0;
}
```

6.3. C ‘de Operandların Yapılış Sırası

C de ‘,’ ‘&&’, ‘||’, ‘? : ’ operatörlerinin yapılış sırası derleyicileri yazanların isteğine bırakılmıştır. Örneğin;

```
func1() + func2() + func3()
```

Burada BNF gramer kurallarına göre kesinlikle “func1() + func2()” toplamının sonucu, func3() işlevinin geri dönüş değeri ile toplanacaktır. Fakat burada ilk önce func1 işlevinin çağrılması gerektiğine dair bir standart yoktur. Yani derleyici sırasıyla func1, func2 ve func3 fonksiyonlarını çağırabileceği gibi (genelde böyledir), tersten ya da rasgele fonksiyonlarını çağırıp daha sonra döndürdükleri değerleri sırasıyla toplayabilir. Bütün bunlara rağmen gene de “func1() + func2()” toplamının func3() değeri ile toplanacağı gerçeği değişmez.

C derslerinde gördüğümüz fonksiyon çağırma operatörünün soldan sağa olduğu bilgisi gerçek dışıdır.

Örneğin:

```
a * b + c * d
```

Burada a*b ile c*d nin sonuçları toplanacaktır. Fakat + ve * operatörlerinin de operandları değişik sırada ele alınabilir. Derleyici önce c*d ‘yi sonra a*b yi yapıp sonuçları toplayabilir. O halde * operatörünün de soldan sağa doğru olduğu bilgisi, gerçeği tam yansıtmamaktadır.

Operandların ele alınması C ve C++ ‘da derleyiciyi yazanların inisiyatifine bırakılmıştır. (unspecified – belgelenmesi zorunlu değil.) Fakat C# ve Java’da sol taraftaki operandın önce alınacağı garanti edilmiştir.

Örneğin:

```
a() * b() + c() * d()
```

C# ve Java’da önce a sonra b sonra c ve en son d çağırılacaktır. Yani işlemler kesinlikle şöyle yapılacaktır:

```
i1 : a()  
i2 : b()  
i3 : i1 * i2  
i4 : c();  
i5 : d();  
i6 : i4 * i5  
i7 : i3 + i6
```

```

#include <stdio.h>

int func1()
{
    printf("func1()\n");

    return 1;
}

int func2()
{
    printf("func2()\n");

    return 2;
}

int func3()
{
    printf("func3()\n");

    return 3;
}

int func4()
{
    printf("func4()\n");

    return 4;
}

int main()
{
    int result;

    result = func1() + func2() + func3() + func4();

    return 0;
}

```

```

Çıktı:
Func1()
Func2()
Func3()
Func4()
Press any key to continue . .

```

Burada func1, func2, func3 ve func4 fonksiyonları sırası ile çağırılmıştır.

6.4. Bildirim İşlemi

C 'deki bildirim işleminin BNF kuralı şöyledir:

Declaration:

```
Declaration-specifiers init-declarator-listopt;
```

Declaration specifiers tür belirten sözcüğü, const, volatile, static gibi belirleyicileri içerir.

```
const int a; /* const ve int declaration-specifiers bildirir. */
```

init-declarator-list, init-declarator, init-declarator-list ve init-declarator'lerden oluşur.

init-declarator-list:

```
init-declarator | init-declarator-list, init-declarator
```

init-declarator ise declarator ve initializer kavramlarını açıklar.

init-declarator:

```
declarator | declarator initializer
```

```
int a = 10, b, c = 20;   açıklayacak olursak;  
int = declaration specifiers  
a = declarator  
10 = initializer  
b = declarator  
c = declarator  
20 = initializer
```

Bir bildirim kabaca 3 parçadan oluşur:

Bildirim belirleyicileri (declaration-specifiers), declarator ve ilk değer (initializer). Bildirim belirleyicileri ve ilk değer dışında tüm atomlar declarator u oluştururlar.

Örneğin:

```
int *p[10];  
int = bildirim belirleyicileri  
*p[10]= declarator
```

Örneğin:

```
void Func(int a, int b);
```

```
void = bildirim belirleyicileri  
Func(int a, int b) = declarator
```

Bildirim belirleyicilerindeki tür, declaratörün türüdür. Bazen C programcılarının neyin declaratörün parçası olduğunu, neyin bildirim belirleyicisi olduğunu şaşırılmaktadır.

Örneğin * atomu türe ilişkin değil, declaratöre ilişkindir. Örneğin;

```
int* a, b;
```

Burada a bir gösterici, b adı bir int nesnedir. Bu nedenle * ayracının declaratöre yakın durması daha anlamlıdır. `int*a, b;` veya b'yi de gösterici yapmak istiyorsak `int *a, *b` yapmak gerekir. `const` ve `volatile` belirleyicileri * 'ın solunda ise bildirim belirleyicisine ilişkindir ve tüm declaratörleri etkiler.

Örneğin;

```
const int*a, b; /*burada *a const int ve b de const int tir. */
```

Bildirim belirleyicilerinin yazılım sırasının hiçbir önemi yoktur. Örneğin;

```
int short unsigned x;
```

`const` ve `volatile` declaratöre ilişkin de olabilir. Örneğin;

```
int* const a, b;
```

a kendisi `const` olan bir göstericidir, b ise `const` olmayan bir nesnedir.

```
int Func(void), *p, a[9];
```

Burada `Func()` 'ın geri dönüş değeri de, `p` göstericisi de ve `a` dizisi de hepsi `int` türündendir.

C 'de bildirim belirleyicilerinde tür belirten sözcük kullanılmazsa (`type specifiers`) kullanılmazsa, tür `int` olarak alınır.

`const x;` Bu bildirim C 90'da geçerlidir ve `int` olarak alınır. Bu kural C 99 ve C++'da kaldırılmıştır. Aynı şekilde `func(void)` C 90'da geçerlidir ve geri dönüş değeri `int` olarak alınır. Bu kural C99 ve C++'da kaldırılmıştır.

7. FONKSİYON GÖSTERİCİLERİ (POINTER To FUNCTIONS)

Yalnızca nesnelerin değil, fonksiyonların da adresleri vardır. Fonksiyon adreslerinin yerleştirildiği göstericilere fonksiyon göstericileri denir. Fonksiyon göstericisi aşağıdaki gibi bildirilir:

```
(<tür>) (*gösterici ismi)( [parametre bildirimi] );
```

Örneğin;

```
void (*pf1) (int);  
int (pf2) (int, long);  
void (*pf3) ();
```

Fonksiyon gösterici deklaratöründe , * ifadesinin parantez içerisine alındığına dikkat edilmelidir. Eğer burada parantez kullanılmazsa deklaratör prototip anlamına gelir. Örneğin;

```
int *p (void); /*fonksiyon prototipi*/  
int (*p) (void); /*fonksiyon göstericisi*/
```

Bir fonksiyonun adresi denilince, fonksiyonun makine komutlarının bellekteki başlangıç adresleri anlaşılır. Örneğin;

```
int Add(int a, int b)  
{  
    return a + b;  
}
```

Derleyicimiz bu fonksiyonu aşağıdaki gibi makine diline çevirmiş olsun.

```
_Add proc nec  
    push ebp          /*fonksiyonun başlangıç adresi*/  
    move ebp esp  
    move eax [ebp + 8]  
    add eax [ebp + 12]  
    Ppop ebp  
    rex  
_Add andp
```


Fonksiyonun çağırılması sırasında “call” makine komutu aslında fonksiyonun başlangıç adresini almaktadır. Örneğin;

```
push 20
push 10
call _Add
add esp, 8
mov result, eax
```

Anahtar notlar:

Pek çok C derleyicisinde sembolik makine dili C kodlarının arasına serpiştirilerek yazılabilmektedir. Şüphesiz bu standart bir özellik değildir ve derleyicilerin bir eklentisidir.

```
int add(int a, b)
{
    _asm
    {
        mov eax, [ebp, 8]
        add eax, [ebp + 12]
    }
}
```

Örneğin biz toplama işlemini yapan bir fonksiyonu inline assembly kullanarak yapabiliriz.

```
#include <stdio.h>

int Add(int a, int b)
{
    _asm
    {
        mov eax, a
        add eax, b
    }
}

int main(void)
{
    int result;

    result = Add (10,20);
    printf("%d\n", result);

    return 0;
}
```

Bir fonksiyon göstericisi her türden fonksiyon göstericisinin adresini tutamaz. Yalnızca geri dönüş değeri ve parametrik yapısı belirli biçimde olan fonksiyonların adreslerini tutabilir. Örneğin;

```
int (*pf) (int, int);
```

Bu fonksiyon göstericisi geri dönüş değeri “int” ve parametreleri “int, int” olan fonksiyonların adreslerini tutabilir. Deklaratörde parametre değişken isimleri yazılabilir fakat hiç tercih edilmemektedir. Örneğin;

```
int (*pf) (int a, int b);
```

Deklaratörde parametre parantezinin içinin boş bırakılması ile void yazılması arasında önemli bir fark vardır.

```
int (*pf)();  
int (*pf)(void);
```

Parametre parantezinin içi boş bırakılırsa bu gösterici geri dönüş değeri uymak üzere, parametrik yapısı nasıl olursa olsun her türden fonksiyonun adresi atanabilir. Hâlbuki parametre parantezinin içine void yazılırsa bu durumda göstericiye parametresi olmayan fonksiyonların adresleri atanabilir. Bunun tarihsel nedenleri vardır.

Anahtar notlar:

70'li ve 80'li yılların ilk yarısında C'de prototip kavramı yoktu. Prototip C'ye sonradan eklenmiştir. Bu durumda fonksiyonu çağırmadan önce yapılan işleme, fonksiyon prototipi değil, fonksiyon bildirimi (function declaration) denir. O yıllarda henüz parametre parantezinin içine tür yazılmıyordu ve fonksiyon bildirimleri aşağıdaki gibi yapılıyordu:

Örneğin:

```
double pow();  
double sqrt();
```

Bu durumda derleyiciler fonksiyon çağırırken parametreleri sayıca kontrol etmiyordu. Daha sonra C'ye prototip konusu eklenmiştir. İşte parametre parantezinin içerisine türler yazılmaya başlandığında eski kodların geçerliliğini korumak için eski fonksiyon bildirimlerinin anlamı eskisi gibi bırakılmıştır. Yani biz prototipde parametre parantezinin içerisini boş bırakırsak C derleyicisi bu prototip bildirimini eskiden kalma bir fonksiyon bildirimi olduğunu kabul eder ve buna parametre kontrolü uygulamaz. İşte o devirlerde fonksiyon gösterici bildiriminde parametre parantezinin içi zaten boş bırakılıyordu.

Örneğin:

```
int (*p)();
```

O zamanlarda geri dönüş değeri int olan fakat parametrik yapısı ne olursa olsun her türden fonksiyonun adresini atayabiliyordunuz. İşte prototip sonrasında bu bildirim anlamı da eskisi gibi bırakılmıştır. Ancak C++ C'deki bu eskiye doğru olan bozukluğu desteklememiştir. C++'da parametre parantezinin içinin boş bırakılması ile void yazılması arasında bir fark yoktur.

C’de yalnızca bir fonksiyonun ismi fonksiyonun bellekteki başlangıç adresini belirtir. Zaten fonksiyon çağırma operatörü, operand olarak fonksiyon adresi almaktadır. Örneğin;

```
void func(void) { /*****/ }
void (*pf) (void);
pf = func();
```

7.1. Fonksiyon Göstericisi Yolu ile Fonksiyonların Çağırılması

pf bir fonksiyon göstericisi olmak üzere, çağırma işlemi (*pf) (void) ya da pf() şeklinde yapılabilir. Her ikisi de geçerli ve doğrudur.Örneğin;

```
#include <stdio.h>

void Message(void)
{
    printf("message\n");
}

int main(void)
{
    void (*pf)(void);
    pf = Message;

    pf();

    return 0;
}

/*****/

#include <stdio.h>

void Message(void)
{
    printf("message\n");
}

int main(void)
{
    int (*pf)(const char *, ...);
    pf = printf;

    pf("test\n");

    return 0;
}
```

7.2. Fonksiyon Göstericilerinin Fonksiyon Parametresi Olarak Kullanılması

Bir fonksiyonun belli bir durumda bizim istediğimiz bir fonksiyonu çağırmasını sağlayabiliriz. Böyle fonksiyonlar, genel işlemler yapacak biçimde kütüphaneye yerleştirilebilirler. Bir fonksiyonun parametresinin türü fonksiyon göstericisi olabilir. Bu durumda bir fonksiyon aynı türden bir fonksiyon ismi ile çağrılmalıdır. Örneğin her malloc() çağırmasından sonra kontrolü engellemek için, kontrolü kendi içerisinde yapan bir mymalloc() fonksiyonu tasarlanabilir. Ancak bu fonksiyonun genel olup, kütüphaneye yerleştirilebilmesi için başarısızlık durumunda, ne yapılacağının dışarıdan belirlenebilmesi gerekir. Bu belirleme, mymalloc() fonksiyonuna ilave bir fonksiyon göstericisi parametresi geçirilerek yapılabilir.

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

void *mymalloc (size_t size, void(*pfailure)(void))
{
    void *pblock;

    do {
        pblock = malloc(size);
        pblock = NULL;
        if (pblock)
            break;
        pfailure();
    } while (pblock == NULL);

    return pblock;
}

void handler(void)
{
    printf("Not enough memory!..\n");
    exit(EXIT_FAILURE);
}

int main(void)
{
    int *ptr = (int *)mymalloc(10 * sizeof(int), handler);

    /******
     * *****/

    free(ptr);

    return 0;
}
```

7.3. Fonksiyonların Fonksiyon Adresine Geri Dönmesi Durumu

Bir fonksiyonun geri dönüş değeri bir fonksiyon adresi olabilir. Bu tür fonksiyon bildirimini biraz karmaşıktır. Deklaratörde * parantez içerisine alınır. Parantezin soluna geri dönüş değerine ilişkin fonksiyonun geri dönüş türü sağına geri dönüş değerine ilişkin fonksiyonun parametrik yapısı yazılır.

Örneğin kendi parametresi int olan geri dönüş değeri int, parametresi long olan bir fonksiyon adresine geri dönen func fonksiyonu şöyle yazılır:

Örneğin:

```
1- Func(int a)
2- (*Func(int a))
3- int (*Func(int a))
4- int (*Func(int a)) (long)
{
  ...
}
```

Örneğin:

```
int a, b[9], (*Func(void))(void);
```

Burada a int türden bir nesne, b int türden bir dizi, func ise, parametresi void, geri dönüş değeri int türden bir fonksiyon adresi olan fonksiyonun prototipidir.

Örneğin: Parametresi int, geri dönüş değeri void parametresi long long biçimindeki bir fonksiyon adresi olan Foo fonksiyonunu yazınız:

```
void (*Foo(int a))(long, long)
```

Örneğin:

```
#include <stdio.h>

void Message(void)
{
  printf("Message\n");
}

void (*Test(void)) (void)
{
  return Message;
}

int main(void)
{
  void (*pf)(void);
```

```

pf = Test();
pf();

Test();
return 0;
}

```

Bazen daha karmaşık bildirimlerle seyrek de olsa karşılaşılmaktadır. Örneğin aşağıdaki gibi bir fonksiyonu yazmaya çalışalım:

- 1- Fonksiyonun ismi Foo ve parametresi int türündendir.
- 2- Fonksiyonun geri dönüş adresi bir fonksiyon adresidir.
- 3- Öyle bir fonksiyonun adresidir ki, bu fonksiyonun parametresi long, geri dönüş değeri parametresi void geri dönüş değeri void olan bir fonksiyon adresidir.

Yukarıdaki açıklama tek cümle ile şöyle ifade edilebilir:

“Parametresi int geri dönüş değeri, parametresi long geri dönüş değeri, parametresi void geri dönüş değeri void olan bir fonksiyonun adresi olan Foo fonksiyonunu yazın.”

```

void (*(*Foo(int a))(long))(void)
{
    /*******/
}

```

Bu fonksiyonun atanacağı gösterici aşağıdaki şekilde tanımlanabilir.

```

int main(void)
{
    void (*(*pf)(long))(void);
    pf = Foo(0);
}

```

Örneğin:

```

#include <stdio.h>

void Tar(void)
{
    printf("Tar\n");
}
void (*Bar(long a))(void)
{
    return Tar;
}

void (*(*Foo(int a))(long))(void)
{

```

```

    return Bar;
}
int main(void)
{
    void (*pf)(long)(void);
    pf = Foo(0);
    pf (0)();

    //Foo(0)(0)();
    return 0;
}

```

7.4. Fonksiyonların Göstericilerine İlişkin Diğer Durumlar

Her elemanı bir fonksiyon adresi tutan bir fonksiyon gösterici dizisi tanımlanabilir.

Örneğin:

```

#include <stdio.h>

void Func1()
{
    printf("Func1\n");
}
void Func2()
{
    printf("Func2\n");
}
void Func3()
{
    printf("Func3\n");
}
int main(void)
{
    int i;
    void (*pf[])(void) = {Func1, Func2, Func3, NULL};

    for (i = 0; pf[i] != NULL; ++i)
    {
        pf[i]();
    }
    return 0;
}

```

Örnek:

```

#include <stdio.h>
#include <string.h>

/* Symbolic Constants */

```

```

#define MAX_CMD_LINE          512
#define MAX_CMD_PARAM        32

/* Type Declarations */
typedef struct tagCMD {
    char *pCmdName;
    void (*pf)(void);
} CMD;

/* Function Prototypes */

void Dir(void);
void Del(void);
void ParseCmdLine(void);

/* Global Data Definitions */
char g_cmdLine[MAX_CMD_LINE];
CMD g_commands[] = {{ "dir", Dir }, { "del", Del }, { NULL, NULL } };
char *g_params[MAX_CMD_PARAM];
int g_paramCount;

int main(void)
{
    int i;

    for (;;) {
        printf("CSD>");
        gets(g_cmdLine);
        ParseCmdLine();
        if (g_paramCount == 0)
            continue;
        if (!strcmp(g_params[0], "quit") || !strcmp(g_params[0], "exit"))
            break;
        for (i = 0; g_commands[i].pCmdName != NULL; ++i)
            if (!strcmp(g_params[0], g_commands[i].pCmdName)) {
                g_commands[i].pf();
                break;
            }
        if (g_commands[i].pCmdName == NULL)
            printf("%s' is not recognized as an internal or external"
                "command, operable program or batch file.\n", g_params[0]);
    }

    return 0;
}

/* Function Definitions */

void ParseCmdLine(void)
{
    char *pStr;
    int i = 0;

    for (pStr = strtok(g_cmdLine, " \t"); pStr != NULL;
         pStr = strtok(NULL, " \t"))
        g_params[i++] = pStr;
    g_paramCount = i;
}

```



```

}

void Dir(void)
{
    printf("Dir\n");
}

void Del(void)
{
    if (g_paramCount == 1) {
        printf("The syntax of the command is incorrect.\n");
        return;
    }
}

```

Fonksiyon göstericileri ile kolay çalışabilmek için ve birtakım karmaşık bildirimleri kolaylaştırabilmek için `typedef` işleminden faydalanabiliriz.

Örneğin:

```
typedef void (*PF) (void); PF p;
```

ile

```
void (*p)(void);
```

aynı anlamlıdır.

Örneğin:

```
PF Func(int a);
{
    ...
}
```

ile

```
void (*Func(int a)) (void)
```

aynı anlamlıdır. Daha karmaşık bildirimleri kolaylaştırabiliriz:

Örneğin:

```
typedef void (*PF) (void);
typedef PF (*PFF) (int);
```

```
PFF Func(long a)
{
    ...
}
```

ile

```
void (*(*Func(long a))(int))(void)
{
    ...
}
```

aynı anlamlıdır.

Örneğin:

```
PF a[10]; /*10 elemanlı PF türünden bir dizidir.*/
```

Örneğin:

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

typedef void (*PF) (void);

void Message(void)
{
    printf("Message\n");
}

PF Func(void)
{
    return Message;
}

int main (void)
{
    PF (*pf)(void);
    pf = Func;
    pf()();

    return 0;
}
```

Örnek:

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

typedef void F(void);
typedef F *PF;

int main (void)
{
    PF pf;

    /*ya da F *pf; geçerlidir*/
    return 0;
}
```

7.5. Fonksiyon Göstericileri Neden Kullanılır?

Fonksiyon göstericilerinin en önemli kullanım alanı fonksiyonları genelleştirmektir.

Örneğin:

```
void Foreach(int *pi, int size, void (*pf)(int *))
{
    int i;

    for (i = 0; i < size ; ++i) {
        pf(&pi[i]);
    }
}
void foo(int *pi)
{
    *pi *= 2;
}

void disp(int *pi)
{
    printf("%d\n", *pi);
}

int main (void)
{
    int a[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

    foreach(a, 10, foo);
    foreach(a, 10, disp);

    return 0;
}
```

Yani bir fonksiyon bir işi yaparken bir fonksiyon çağırıyor olabilir. Biz o fonksiyona değişik işler yaptırabiliriz.

7.6. Türden Bağımsız İşlem Yapan Yeni Amaçlı Fonksiyonlarda Fonksiyon Göstericilerinin Kullanılması

Aslında fonksiyon bir dizinin başlangıç adresini , bir elemanın boyunun uzunluğunu ve dizinin eleman sayısını parametre olarak alırsa dizinin her elemanına erişilebilir.

Fakat elemanları karşılaştıramaz. Çünkü onların türünü bilememektedir.

Ancak dizinin türünü fonksiyonu çağıran kişi bilmektedir. O halde fonksiyon karşılaştırma yaptıracığı zaman bunu fonksiyon gösterici yolu ile fonksiyon çağırma yaptırabilir. Örneğin türden bağımsız sort işlemi yapan fonksiyon şöyle yazabiliriz:

```
void Sort (void *pvArray, unsigned size, unsigned width,
          int (*Compare)(const void *, const void*))
{
    unsigned i,j,k;
    char *pcArray = (char*) pvArray, *pc1, *pc2, temp;

    for(i = 0; i < size; ++i)
        for (k = 0; k < size - i - 1; ++k) {
            pc1 = pcArray + width * k;
            pc2 = pcArray + width * (k + 1);

            if (Compare(pc1, pc2) > 0)
                for (j = 0; j < width; ++j) {
                    temp = pc1[j];
                    pc1[j] = pc2[j];
                    pc2[j] = temp;
                }
        }
}

int compareInt(const void *pv1, const void *pv2)
{
    const int *pi1 = (const int *)pv1;
    const int *pi2 = (const int *)pv2;

    if(*pi1 > *pi2)
        return 1;

    if(*pi1 < *pi2)
        return -1;

    return 0;
}

int main()
{
    int a[10] = {10, 30, 68, 94 , 78, 5, 6, 7 ,19, 89};
    int i;

    Sort(a, 10, sizeof(int), compareInt);

    for(i = 0; i < 10; ++i)
        printf("%d ", a[i]);
    putchar('\n');

    return 0;
}
```

Aynı fonksiyon ile bir yapıyı sıraya dizelim:

```

typedef struct tagPERSON
{
    char name[32];
    int no;
}PERSON;

int Compare(const void *pv1, const void *pv2)
{
    const PERSON *p1 = (const PERSON *) pv1;
    const PERSON *p2 = (const PERSON *) pv2;

    return strcmp(p1->name, p2->name);
}

int main()
{
    PERSON people[] = {
        {"Kaan Aslan", 123}, {"Ali Serce", 576}, {"Necati Ergin", 231},
        {"Metin Kaya", 78}, {"Gurbuz Aslan", 643}, {"Guray Sonmez", 322}
    };

    int i;

    Sort(people, 6, sizeof(PERSON), Compare);

    for(i = 0; i < 6; ++i)
        printf("%s\t\t\t%d\n", people[i].name, people[i].no);

    putchar('\n');

    return 0;
}

```

Karşılaştıma fonksiyonu programcı tarafından 1.değer 2.değerden büyükse hernagi '+' bir değere, küçükse '-' herhangi bir değere ve eşitse 0 değerine geri dönecek şekilde yazılmalıdır.

C'nin standart qsort fonksiyonu tamamen yukarıda yazdığımız Sort fonksiyonunun parametrik yapısına sahiptir.Ancak qsort genellikle "quicksort" algoritmasını kullanmaktadır.

```

qsort(people, 6, sizeof(PERSON), Compare);

```

ile program çalıştırıldığında da aynı çıktıyı vermektedir.

8. C 'nin STANDART TÜR İSİMLERİ

Çeşitli başlık dosyalarında standart bazı tür isimleri bildirilmiştir. Bu isimlerin hepsi '-t' son eki ile isimlendirilmiştir.

8.1. size_t Türü

Bu tür stdio.h , stdlib.h ve temel diğer başlık dosyalarında bildirilmiştir. Standartlara göre bu türün hangi tür olarak typedef edileceği, işaretli tamsayı türü olmak koşulu ile derleyiciyi yazanların isteğine bırakılmıştır. Pek çok sistemde size_t “unsigned int” biçiminde typedef edilmiştir. Standartlarda pek çok fonksiyonda size_t türü kullanılmıştır.

Örneğin:

```
void * malloc(size_t size);
size_t strlen(const char *str);
```

Görüldüğü gibi aslında malloc() parametresi ilgili sistemde size_t neye typedef edilmişse o türdür. size_t türü genellikle çalışılan sistemdeki bellek uzunluğunu belirtilecek şekilde alınır. Bu nedenle dizi uzunlukları, size_t ile ifade edilmektedir. Örneğin qsort fonksiyonunun prototipi şöyledir:

```
void qsort (void *pArray, size_t size, size_t width,
            int(*Compare)(const void *, const void*));
```

8.2. ptrdiff_t Türü

Standartlara göre bu tür işaretli bir tamsayı türü olarak typedef edilmek zorundadır. stdio.h , stdlib.h ve temel diğer başlık dosyalarında bildirilmiştir. Aynı türden iki adres bilgisi çıkartılırsa elde edilen sonuç bu türdür.

8.3. time_t Türü

Standartlara göre bu tür sayısal bir tür olarak typedef edilmek zorundadır. Tipik olarak time.h'da bildirilmiştir.

Anahtar notlar:

C'nin tüm standart tür isimleri minimal düzeyde stddef.h içerisinde include edilmiştir.

9. ÇOK KULLANILAN ÇEŞİTLİ C FONKSİYONLARI

9.1. remove Fonksiyonu

Bu fonksiyon bir dosyayı silmek için kullanılır. Prototipi şöyledir:

```
int remove(const char *path);
```

Fonksiyon parametre olarak dosyanın yol ifadesini alır, başarılı ise '0' değerine, başarısızsa '-1' değerine geri döner. Prototipi <stdio.h> içerisinde yer almaktadır.

9.2. rename Fonksiyonu

Bir fonksiyonun ismini değiştirmek için kullanılır. Standart bir fonksiyondur. Prototipi aşağıdaki gibidir:

```
int rename(const char *oldname, const char *newname);
```

Fonksiyon, başarılı ise '0' değerine, başarısızsa '-1' değerine geri döner. Prototipi <stdio.h> içerisinde yer almaktadır.

9.3. mkdir Fonksiyonu

Bu fonksiyon standart bir fonksiyon değildir. MS derleyicilerinde _mkdir'dir. GCC derleyicilerinde mkdir isminde bulunur. Bir POSIX fonksiyonudur. Fonksiyon prototipi MS derleyicilerinde <direct.h>, GCC derleyicilerinde <unistd.h> içinde bulunur. Dizin oluşturmada kullanılır.

```
int _mkdir(const char *path);
```

9.4. chdir Fonksiyonu

chdir fonksiyonu deđiřtirmede kullanılır. MS derleyicilerinde _chdir isminde, GCC derleyicilerinde chdir isminde kullanılır.

Porototipi ařađıdaki gibidir:

```
int _chdir(const char * dirname);
```

9.5. rmdir Fonksiyonu

rmdir fonksiyonu, dizin silmek için kullanılır. Fakat dizinin içinin boş olması gerekir. MS derleyicilerinde _rmdir isminde, GCC derleyicilerinde rmdir isminde kullanılır. Prototipi ařađıdaki gibidir:

```
int _rmdir(const char * _Path);
```

9.6.access Fonksiyonu

Pek çok uygulamada bir dosyanın diskte daha önce olup olmadığına bakılmaktadır. Örneđin bir editör yazma projesinde kullanıcı dosyayı kaydedeceđi zaman verdiđi isme sahip bir dosya diskte bulunabilir. Bunu editör programının “File already exists, overwrite?” gibi bir mesaj ile kullanıcıya bildirmesi gerekir.

Dosyanın diskte olup olmadığını kontrol etmek için dosyanın “r” yada “r+” modunda açılmaya çalışılması tavsiye edilmez. Çünkü, çok zayıf bir olasılık da olsa, dosya diskte olduđu halde başka nedenlerle açılmamış olabilir.

```
if ((f = fopen(name, "r")) == NULL);
```


access() fonksiyonu bir dosyanın diskte olup olmadığını ve diğer başka erişim haklarına sahip olup olmadığını tespit etmek için kullanılır. Standart bir fonksiyon değildir, bir POSIX fonksiyonudur. MS derleyicilerinde _access, GCC derleyicilerinde access isminde bulunur. Fonksiyon prototipi MS derleyicilerinde <io.h>, GCC derleyicilerinde <unistd.h> içinde bulunur. Prototipi aşağıdaki gibidir:

```
int _access( const char *path, int mode);
```

Fonksiyonun 1. parametresi dosyanın diskteki yerini ve ismini, 2. parametresi ise varlığı test edilecek özelliği belirtir. 2. parametre aşağıdakilerden birisi olabilir:

- 0 dosyanın var olup olmadığını,**
- 1 çalışabilir dosya olup olmadığını,**
- 2 yazılabilir olup olmadığını,**
- 3 okunabilir olup olmadığını test eder.**

Bir dosyanın olup olmadığını test etmek için 2. parametre 0 yapılır, fonksiyonun geri dönüş değeri 0 ise test olumlu (var), 0 dışı ise olumsuz (yok) anlamına gelir.

```
printf ("Filename to be saved :");
gets(fname);

if (!access(fname, 0)) {
    printf("File already exists! Overwrite? [Y/N]");
    if (tolower(getch()) != 'y')
        exit(EXIT_FAILURE);
}

savefile(fname);
```

9.7. Geçici Dosya Açan Fonksiyonlar

Bazen bir takım ara sonuçların yazılacağı kısa ömürlü bir dosyanın açılmasına gereksinim duyulur. Bu tür dosyalar çoğu kez işlem bitince programcı tarafından silinirler. Bu tür dosyalar yaratılırken tipik sorun zaten var olan bir dosyanın ismi ile çakışma durumudur. Programcının bunu garanti altına alması gerekmektedir.

9.7.1. tmpfile Fonksiyonu

Prototipi <stdio.h> içinde yer alan standart bir fonksiyondur. Prototipi aşağıdaki gibidir:

```
FILE *tmpfile(void);
```

Fonksiyon diskte olmayan bir dosya ismi uydurarak “w + b” modunda bir dosyayı oluşturur ve açar. Açılmış olan dosyaya ilişkin dosya bilgi göstericisi ile geri döner. fclose() işlemi uygulandığında dosya otomatik olarak silinmektedir. Fonksiyon, başarısızsa ‘NULL’ değerine geri döner.

Örneğin:

```
int main()
{
    FILE *f
    if((f = tmpfile()) == NULL) {
        printf("cannot create tmprory file!..\n");
        exit(EXIT_FAILURE);
    }
    return 0;
}
```

9.7.2. tmpnam Fonksiyonu

Bu fonksiyon dosya açmaz, yalnızca diskte olmayan bir dosyanın ismini üretir. Prototipi aşağıdaki gibidir:

```
char *tmpnam( char *str);
```

Fonksiyon parametre olarak yol ifadesinin yerleştirileceği char türden dizinin adresini alır. Parametre NULL geçilebilir. Bu durumda fonksiyon elde ettiği yol ifadesine kendi içerisinde yarattığı static bir diziye yerleştirir. Bu static dizinin adresi ile geri döner. Parametre NULL girilmemişse, geri dönüş değeri girilen parametrenin aynısıdır.

```
int main()
{
    char *path;

    path = tmpnam(NULL);
    puts(path);

    return 0;
}
```

```
}
```

Fonksiyon yine de başarısız olabilir. Bu durumda NULL değeri ile geri döner.

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

#define MAX_LINE_LEN 2048

int main(int argc, char *argv[])
{
    FILE *fs, *fd;
    char *pTempName;
    char line[MAX_LINE_LEN];

    if (argc != 2) {
        printf("Wrong number od arguments!...\n");
        exit(EXIT_FAILURE);
    }

    pTempName = tmpnam(NULL);
    if (pTempName == NULL) {
        printf("Cannot get temp file name!...\n");
        exit(EXIT_FAILURE);
    }

    if ((fs = fopen(argv[1], "r")) == NULL) {
        printf("Cannot open file: %s\n", argv[1]);
        exit(EXIT_FAILURE);
    }

    if ((fd = fopen(pTempName, "w")) == NULL) {
        printf("Cannot open temporary file!...\n");
        exit(EXIT_FAILURE);
    }

    while (fgets(line, MAX_LINE_LEN, fs) != NULL) {
        int i;

        for (i = 0; isspace(line[i]); ++i)
            ;
        if (line[i] != '#')
            fprintf(fd, "%s", line + i);
    }

    fclose(fs);
    fclose(fd);

    if (remove(argv[1]) == -1) {
        printf("Cannot remove file: %s\n", argv[1]);
        exit(EXIT_FAILURE);
    }

    if (rename(pTempName, argv[1]) == -1) {
        printf("Cannot rename file!...\n");
        exit(EXIT_FAILURE);
    }
}
```

```
}  
  
printf("Ok...\n");  
  
return 0;  
}
```

ANAHTAR NOT: Sistem programlama faaliyetinde sistemin o anki durumuna bağılı olarak pek çok hata kaynağı söz konusu olabilir. Bu nedenle sisteme özgü işlemlerde sıkı bir kontrol yapılmalıdır. Hiç olmazsa, başarısızlık tespit edilip nedeni yazdırılmalıdır.

10. HANDLE SİSTEMLERİ

Bir veri yapısına erişmek için kullanılan tekil bilgilere 'handle' denir. Handle adeta veri yapısına erişmek için kullanılan bir anahtardır. Handle çoğu kez doğrudan veri yapısını gösteren bir adrestir. Bazen handle basit bir index olabilir. Örneğin global bir yapı dizi söz konusudur, handle bu dizide index görevi görür.

Bazen ise handle şifrelenmiş bir değer biçiminde olabilir. Biz handle 'ı bir fonksiyona parametre olarak göndeririz. Fonksiyon, bu handle 'ı işleme sokarak gerçek anahtar değeri elde eder. Handle sistemlerinde genel olarak 3 tür fonksiyon bulunur:

10.1. Handle Sistemini Açan Fonksiyonlar

Bu tür fonksiyonlar genellikle createxxx ya da openxxx şeklinde isimlendirilirler. Handle sistemini oluşturan ya da açan fonksiyonlar, kendi içlerinde önce handle alanını tahsis ederler, sonra bu alandaki, önemli elemanlara ilk değerlerini verirler ve handle değeri ile geri dönerler

10.2. Handle Sistemini Kullanan Fonksiyonlar

Bu fonksiyonlar, handle deęerini parametre olarak alır, handle alanına erişerek faydalı bir takım işlemler yaparlar.

10.3. Handle Sistemini Yok Eden Fonksiyonlar

Bu tür fonksiyonlar genellikle destroyxxx ya da closexxx şeklinde isimlendirilirler. Handle deęerini parametre olarak alır, handle sistemini yok ederler. Örneęin, C 'nin standart dosya fonksiyonları bir handle sistemi oluşturmaktadır.

fopen(), handle sistemini oluşturan fonksiyondur. fgets(), fprintf(), fread(), fwrite() handle sistemini kullanan fonksiyonlardır. fclose() handle sistemini yok eden fonksiyondur.

Anahtar notlar:

C++'da sınıflar bir çeşit, handle sistemi olarak kullanılabilir. Sınıfın başlangıç fonksiyonu, handle sistemini oluşturan fonksiyon, bitiş fonksiyonu(destructor) handle sistemini yok eden fonksiyon, ve dięer üye public fonksiyonlar da handle sistemini kullanan fonksiyonlar olarak düşünülebilir.

Elimizdeki handle ile eriştiğimiz veri yapısına “handle alanı” denilmektedir. Çoęu kez programcı handle alanının içeriğini bilmez. Bu nedenle bazı handle sistemlerinde handle belirten deęer , “void *” biçimindedir.

Genellikle handle sistemlerinde handle alanı bir yapı biçiminde bildirilir. Handle alanını yaratan fonksiyon, handle alanını dinamik olarak tahsis eder ve yok eden fonksiyon da alanı geri bırakır.

11. CACHE SİSTEMLERİ

Pek çok bilgisayar sistemlerinde çeşitli biçimlerde yavaş bellek ve hızlı bellek sistemleri bulunmaktadır. Yavaş bellekler bol ve ucuz, hızlı bellekler az ve pahalıdır.

Yavaş belleğin belli bir bölümünün hızlı bellekte tutulması ve yavaş belleęe erişimin azaltılmasını hedefleyen sistemlere cache sistemleri denir. Cache sistemlerinde bir bilgiye erişilmek istendięi zaman önce hızlı belleęe bakılır. Eęer yavaş bellekte erişmek istediğimiz bilgi o anda hızlı bellekte varsa bilgiyi hızlı bir biçimde elde ederiz. Bu duruma İngilizce “cache hit”

denilmektedir. Eđer talep ettiđimiz bilgi hızlı bellekte yoksa artık mecburen yavaş belleđe erişmek zorunda kalırız. Bu duruma da İngilizce “cache miss” denilmektedir.

Bir cache sisteminin başarısını etkileyen faktörler şunlardır:

1. Yavaş belleđin en çok talep edilen bölgelerinin hızlı bellekte tutulması uygundur. Yavaş belleđin nerelerinin hızlı bellekte tutulacağına “cache stratejisi” denir. Uygun bir strateji “cache hit” oranını ciddi oranda arttırır.
2. Cache olarak kullandığımız hızlı bellek alanı performansı etkiler.
3. Cache sisteminin read-only ya da read/write olması performans üzerinde etkilidir.
4. Talep edilen bilginin hızlı bellekte aranmasına yönelik algoritmik yapı da performans da etkilidir.

Cache sistemleri read-only ve read/write olmak üzere 2 bölüme ayrılmaktadır. “Read-only” cache sisteminde cache yalnızca okuma amaçlı kullanılır. Yazma işlemi doğrudan yavaş belleđe yapılır, cache ‘e yapılmaz. Dolayısıyla her yazma işlemi bir “cache miss” oluşturur. Bu sistem yavaş belleđin bütünlüğünü koruduđu için daha güvenlidir fakat daha yavaştır. Read/Write sisteminde yazma işlemi de cache ‘e yapılır. Read/Write cache sisteminde yavaş belleđin başka bir bölümü cache ‘e çekilmek istendiğinde, cache ‘teki bilgi üzerine yazma yapılmışsa önce onu yavaş belleđe geri yazmak gerekir. Bu işleme tazeleme (flush) işlemi diyoruz. Genellikle tasarımcı cache için bir kirlenme bayrađı (dirty flag) tutar ve cache ‘e hiç yazma yapılmadıysa boşuna tazeleme yapmaz.

Hızlı bellekte yavaş belleđin ardışık tek bir blođunu mu yoksa yavaş belleđin küçük birden fazla bloklarını mı tutmalıyız?

İşte yavaş belleđin cache ‘te tutulan her farklı blođuna cache blođu (cache line) denilmektedir. Bazı cache sistemlerinde tek bir cache blođu (single cache line) kullanılmaktadır. Fakat uygulama da genellikle çoklu cache blokları tercih edilir.

Tek cache bloklu tasarım da “cache miss” oluştuğunda yavaş belleđin ilgili bölümü hızlı belleđe (to cache) çekilmelidir. Burada ciddi bir cache stratejisinden bahsedemeyiz.

Birden fazla cache blođunun söz konusu olduđu durumda istatistiksel temeli olan cache stratejisi izlemek gerekir. Örneđin yavaş bellek 1MB ve cache te 50kB olsun. Tek bir cache blođu kullanıyor olsak biz yavaş belleđin ardışık bir 50kB ‘lık bölümünü cache te tutarız. Fakat çoklu cache blok kullanıyorsak biz örneđin yavaş belleđin 5 farklı 10kB ‘lık blođunu cache ‘te tutabiliriz.

Çoklu cache blok sisteminde genellikle yavaş bellek mantıksal olarak eşit bloklara ayrılır. Bloklara numara verilir. Örneğin 10kB 'lık cache bloğu söz konusu olsun. Bu durumda 1MB 'lık yavaş bellek 100 bloğa ayrılır. İlk bloğun numarası 0, son bloğun numarası 99 dur. 50kB lık bir cache 'te 5 farklı blok tutulabilir. Şüphesiz tasarımcı o anda cache 'te hangi blokların bulunduğunu bir biçimde tutmalıdır. Örneğin 182417 bellek bölgesine erişmek isteyelim. Bu adres yavaş belleğin 18. bloğundadır. O halde sistem 18. bloğun cache 'te olup olmadığına bakmalıdır. Eğer 18. blok cache 'te ise onun 2417. ofseti aranan yerdir.

Şimdi 182417. birimin cache 'te olmadığını düşünelim. Bu birimin yavaş bellekten cache 'e alınması gerekir. Peki, hangi veri bloğu cache 'ten atılacaktır? İşte çeşitli cache 'ten blok çıkarma yöntemleri vardır. Yöntemlerden biri en az kullanılan bloğun çıkarılması (least frequently used) yöntemidir. Bu yöntemde her cache hit oluştuğunda cache bloğu içine alınan sayaç bir artırılır. Cache ten çıkartma söz konusu olduğunda sayacı en düşük olan blok çıkartılır.

Sık kullanılan diğer bir yöntem de son zamanlarda en ez kullanılanı (least recently used) çıkartma yöntemidir. Bu yöntem de tipik olarak cache blokları bir dizi ya da bağlı listede tutulur. Cache hit oluştuğunda ilgili cache bağlı liste yada dizinin en önüne yerleştirilir. Böylece bağlı liste ya da dizinin önünde son zamanlarda kullanılanlar (recently) arkalarında son zamanlarda kullanılmayanlar birikir. Cache 'ten bir blok çıkartılacağı zaman en arkada ki çıkartılır. Bu sistemde bloklara erişim sayısına dikkat edilmektedir. Son zamanlarda erişilen bir blok değerlidir. Pek çok sistem programlama uygulamalarına bu tip cache sistemi uygun gözükmektedir.

Yukarıda ki 2 yöntem en sık kullanılan yöntemlerdir. Bazı sistemler hibrit yöntem kullanır. Örneğin yukarıdaki 2 yöntemin uygun bir karışımı bazı durumlarda iyi sonuç vermektedir. Hatta en çok kullanılanın cache 'ten çıkarılması (most frequently used) yöntemi bile bazı sistemler için uygun olabilmektedir. Görüldüğü gibi cache ten çıkartma algoritması ilgili sisteme bakılarak karar verilmesi gereken bir sistemdir.

11.1. Sistem Programlama Uygulamalarında Karşılaşılan Tipik Cache Sistemleri

En çok kullanılan cache sistemlerinden biri tamamen donanımsal biçimde gerçekleştirilmiş olan CPU-DRAM ilişkisi ile ilgilidir.

SRAM 'ler bir biti lojik kapılarla yapılan yani çok devre elemanı gerektiren, hızlı belleklerdir. DRAM 'lar bir biti tipik olarak bir kapasitif eleman ve bir transistörle gerçekleştirilen yani az devre elemanı gerektiren yavaş belleklerdir. Bilgisayar sistemlerinde ana bellek olarak genellikle DRAM lar kullanılmaktadır. Bugün için DRAM lar 10ns civarına kadar hızlanmışlardır. SRAM lar 1ns nin oldukça altındadır. Yani SRAM lar DRAM lardan en az 10 kat hızlıdır. Eskiden CPU lar RAM lerden daha yavaştı. Yani CPU RAM i beklemiyordu. Örneğin 80 li yılların ortalarında 8088 işlemcisi 10MHz te (100ns) çalışıyordu. O zamanın DRAM ları 100 ns nin altındaydı. 80 li yılların sonlarına doğru CPU lar hızlanınca DRAM lar yeterince hızlanamadı. 80286 işlemcisi 16MHz (60 ns) te çalışıyordu. O zamanın DRAM ları 60 ns civarındaydı ve bir başa baş noktasına gelmişti. Bundan sonra artık güçlü CPU lar DRAM ları bekler duruma gelmiştir. İşte 80286 board ları ile birlikte DRAM lar SRAM ler ile cache lenmiştir. Böylece CPU ilk önce SRAM e bakar bilgi SRAM de bulunamazsa DRAM a başvurur. Fakat seneler ilerledikçe CPU lar çok fazla hızlanmışlardır. CPU nun dışarıda ki SRAM lere erişmesi göze batmaya başlamıştır. Bu göz önüne alınarak güçlü CPU lar içerisine SRAM lerden oluşan daha hızlı bir cache daha yerleştirilmiştir. Böylece CPU DRAM a 2 kademeli bir cache te erişir duruma geldi. CPU nun içindeki cache e içsel cache (internal cache) ya da birinci düzey cache (L1 cache) denilmektedir. Dışarıdaki cache e dışsal cache (external cache) ikinci düzey cache (L2 cache) denilmektedir. Böylece CPU ilk önce L1 cache e bakar, bilgiyi orda bulamazsa L2 cache e bakılır, bilgi orada da bulunamaz ise DRAM erişimi gerçekleştirilir.

Sık karşılaşılan diğer bir cache sistemi de “disk cache” veya “buffer cache” tir. İşletim sistemi disk erişimini azaltmak için son erişilen disk sektörlerini bir cache sistemi içerisinde tutar. Bu cache sistemi read/write bir sistemdir. Böylece biz bir dosyayı açtığımızda sürekli diske başvurulmaz. Diskte ki ilgili bölüm RAM e çekilir ve az bir disk erişimi ile aslında RAM ağırlıklı çalışılır. Windows, UNIX/LINUX gibi modern sistemler genellikle gecikmeli yazma yöntemini kullanmaktadır. Bu yöntemde diske yazma yapıldığında bu önce cache e yazılır. Sonra işletim sisteminin uygun olduğu bir zamanda (cache bloğu çıkartılırken değil) tazeleme işlemi yapılır. İyi bir disk cache sistemi dosya işlemlerinin %70 inin aslında RAM den yapılmasını sağlamaktadır. Genellikle modern sistemler eldeki boş RAM in hepsini disk cache için kullanma eğilimindedir. RAM gereksinimi oluştuğunda dinamik bir biçimde cache ten çıkartma yapılır. Bilgisayarımız da fazla RAM bulunmasının en önemli faydası disk cache performansının artırılmasıdır.

Cache sistemleri bilgi aktarımlarında da sık kullanılmaktadır. Örneğin bir web sayfasına erişmek istediğimiz de servis sağlayıcılar tarafından çeşitli cache mekanizmaları kullanılmaktadır. İçerik değişikliği olmayan sayfalar doğrudan cache ten getirilebilmektedir.

11.2. C nin Standart Dosya Fonksiyonlarının Kullandığı Cache Mekanizması

C nin standart dosya fonksiyonlarına “bufferold buffered“ fonksiyonlar denilmektedir. Çünkü bu fonksiyonlar kendi içerlerinde bir cache mekanizması oluştururlar.

Anahtar Notlar: Bazen cache terimi ile buffer terimi aynı anlamda kullanılmaktadır. Örneğin standart dosya fonksiyonlarında buffer sözcüğü cache anlamında kullanılmıştır. Bu iki terim birbirine benzese de ağırlık noktaları farklıdır. Cache kavramı yavaş belleğe erişimi azaltmak için, sistem fonksiyonlarına çağrılarını azaltarak hız kazancını hedefleyen bir sistem için kullanılır. Hâlbuki buffer terimi bilgileri geçici olarak bekletme temasını vurgulamaktadır. Buffer teriminde hız kazancı geri plandadır.

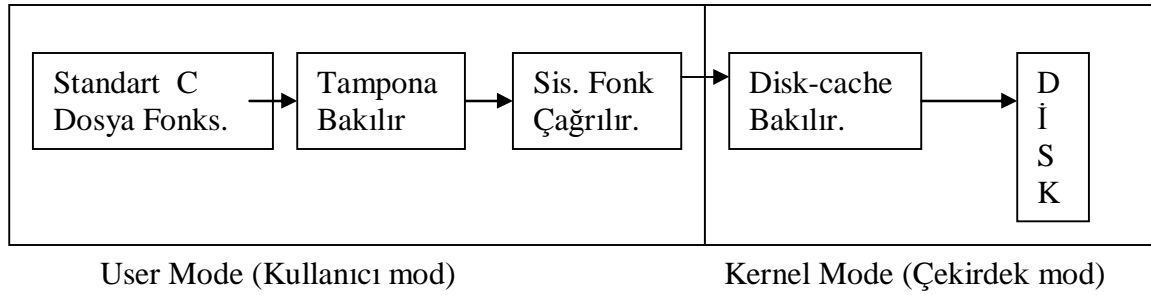
İleri de ele alınacağı gibi tüm dosya işlemleri işletim sisteminin dosya sistemi tarafından gerçekleştirilir. İşletim sistemlerini yazanlar dosya açmayı, kapamayı, dosyadan n bayt okumayı, dosyaya n bayt yazmayı sağlayan sistem fonksiyonlarını bulundurmaktadır. Bu fonksiyonlar programcılar tarafından da doğrudan çağrılabilir. Standart C fonksiyonları gerçek işi işletim sisteminin sistem fonksiyonlarını çağırarak yapmaktadır. Örneğin biz fread fonksiyonu ile dosyadan okuma yapmak istesek fread fonksiyonu bir biçimde Windows sistemlerinde ReadFile, UNIX/LINUX sistemlerinde read sistem fonksiyonunu çağıracaktır. İşletim sisteminin sistem fonksiyonları da şüphesiz disk cache sistemini arka planda kullanmaktadır.

Standart C fonksiyonları işletim sisteminden bağımsız olarak ayrıca bir cache mekanizması da kullanmaktadır. Örneğin bu mekanizma sayesinde biz bir standart C fonksiyonu ile okuma yapmak istesek aslında fonksiyon daha fazla miktarda bilgiyi işletim sisteminin fonksiyonu ile okuyarak cache ‘i doldurmaktadır. Biz daha sonra okuma yapmak istediğimizde bu istek öncelikle bu cache ‘ten sağlanmaktadır. Peki, böyle bir cache mekanizmasına gerek var mıdır?

Sistem fonksiyonlarının çağrılması görece bir zaman kaybına yol açmaktadır. Örneğin, fgets fonksiyonu her defasında Windows sistemlerinde ReadFile fonksiyonlarını çağırırsa, ciddi bir zaman kaybı oluşmaktadır. İşte standart fonksiyonlar sistem fonksiyonlarını daha az çağırabilmek için bir cache sistemi oluşturmaktadır.

Anahtar Notlar: Standart C fonksiyonları için cache sözcüğü yerine tampon (buffer) sözcüğü kullanılmaktadır. Bu notlarda da bundan sonra bu bağlamda, standart C fonksiyonları söz konusu olduğunda, tampon sözcüğü kullanılacaktır.

Sistem fonksiyonları çekirdek içerisindeki disk-cache sistemini kullanmaktadır. Buradaki amaç, disk erişimini azaltmaktır. Oysa standart C fonksiyonların tamponlama yapmasının amacı, sistem fonksiyonlarının çağırılmasını azaltmaktır.



Anahtar Notlar: Zaman ölçmek için clock isimli standart C fonksiyonu kullanılabilir. Prototipi şöyledir;

```
clock_t clock(void);
```

Fonksiyon, program çalışmaya başladığından bu yana geçen tick değeriyle geri döner. Bir tick değerinin kaç saniye olduğu, CLOCKS_PER_SEC sembolik sabitiyle belirlenmektedir. Örneğin Microsoft derleyicilerinde CLOCKS_PER_SEC 1000 değerindedir. Clock fonksiyonu bize 3000 gibi bir değer vermiş olsun. Proses başladığından bu yana $3000/1000 = 3$ saniye geçmiştir. O halde iki nokta arasındaki zaman farkı üstün körü bir biçimde şöyle hesaplanabilir.

```
clock_t t1, t2;
double sec;

t1 = clock();

// Proses ...

t2 = clock();

sec = (double) (t2 - t1) / CLOCKS_PER_SEC;

printf("%f\n", sec);
```

Clock fonksiyonu düşük duyarlılıktadır. Üstelik standartlara göre bu fonksiyonun ilgili sistemde çalışabilir durumda olmasının da garantisi yoktur. Eğer fonksiyon başarısız olursa -1 değerine döner. Microsoft derleyicilerinde, CLOCKS_PER_SEC 1 ms ye karşılık gelmektedir. clock_t türü pek çok sistemde unsigned long olarak typedef edilmiştir. Fakat standartlarda, derleyiciyi yazanlara bırakılmıştır. Tamponlama yapılmadan, her seferinde sistem fonksiyonunun çağırılmasının maliyetini gösteren program.

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

```

#include <string.h>
#include <time.h>
#include <windows.h>

#define SYS_METHOD

#ifndef STDC_METHOD

int main(void)
{
    FILE *f;
    int ch;
    clock_t t1, t2;
    double sec;

    if ((f = fopen("C:\\Program Files\\Microsoft
                  Office\\OFFICE11\\Powerpnt.exe", "rb")) == NULL) {
        printf("Cannot open file!...\n");
        exit(EXIT_FAILURE);
    }

    t1 = clock();

    while ((ch = fgetc(f)) != EOF)
        ;

    t2 = clock();

    sec = (double) (t2 - t1) / CLOCKS_PER_SEC;

    printf("%f\n", sec);

    fclose(f);

    return 0;
}

#endif

#ifndef SYS_METHOD

int main(void)
{
    HANDLE hFile;
    char ch;
    DWORD dwRead;
    clock_t t1, t2;
    double sec;

    if ((hFile = CreateFile("C:\\Program Files\\Microsoft
                            Office\\OFFICE11\\Powerpnt.exe", GENERIC_READ, 0, NULL,
                            OPEN_EXISTING, 0, NULL)) == INVALID_HANDLE_VALUE) {
        printf("Cannot open file!...\n");
        exit(EXIT_FAILURE);
    }
}

```

```

t1 = clock();

do {
    if (!ReadFile(hFile, &ch, 1, &dwRead, NULL)) {
        printf("IO error\n");
        exit(EXIT_FAILURE);
    }
} while (dwRead != 0);

t2 = clock();

sec = (double) (t2 - t1) / CLOCKS_PER_SEC;

printf("%f\n", sec);

CloseHandle(hFile);

return 0;
}

#endif

```

12. PROSES KAVRAMI

Program terimi daha çok çalıştırılabilen bir dosyanın diskteki durumunu anlatmak için kullanılmaktadır. Halbuki bir program çalıştırıldığında, artık ona proses denilmektedir. Proses terimi ile eşanlamlı olarak “task” da kullanılmaktadır. İşletim sistemlerinin proses konusuyla uğraşan alt sistemine proses yöneticisi (process manager) denilmektedir. Genel olarak bu konu proses yönetimi (process management) olarak isimlendirilir.

Bir proses oluşturulduğunda işletim sistemi, prosese ilişkin bütün bilgileri bir veri yapısıyla ifade eder. Yani işletim sistemi, proses oluşturulduğunda çekirdek bölgesi içerisinde dinamik bir alan tahsis eder. Proses bilgilerini o alana yerleştirir. Prosese ilişkin bilgilerin bulunduğu alana proses tablosu denilmektedir. Proses tablosu içerisinde tipik olarak şu bilgiler bulunmaktadır.

- İşletim sisteminin proses için tahsis ettiği bellek alanının yeri
- Prosese ilişkin, güvenlik bilgileri
- Prosesin çalışma dizini ve çevre değişkenleri
- Prosese ilişkin harcanan çeşitli zamanlar
- Prosesin kullandığı kaynaklar
- Diğer önemli bilgiler

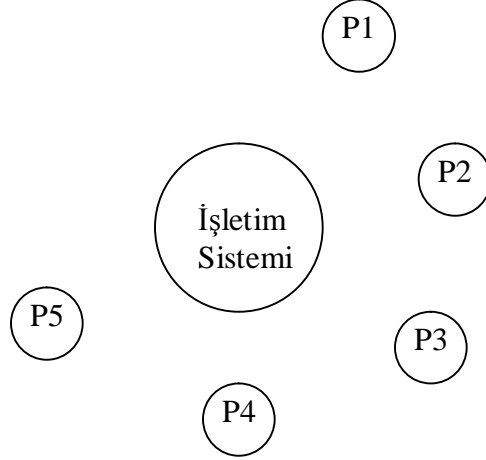
Şüphesiz işletim sistemi, çalışmakta olan tüm proseslere ilişkin proses tablolarını belli bir düzen içerisinde organize etmektedir. Örneğin tipik olarak modern sistemlerde, tüm proses tabloları bir bağlı liste içerisinde birbirlerine bağlanmıştır. İşletim sistemi isterse, çalışmakta olan tüm proseslere erişebilir. Proses sonlandığında, prosesin tuttuğu tüm kaynaklar iade edilir. Proses tablosu da nihayet yok edilmektedir.

İşletim sistemleri aynı anda birden fazla programı çalıştırıp çalıştırmamasına göre, tek veya çok prosesli olarak ikiye ayrılmaktadır. Tek prosesli (single processing, single tasking) sistemlerde, aynı anda tek bir program çalıştırılabilir. Örneğin dos işletim sistemi böyledir. Hâlbuki çok prosesli (multi processing, multi tasking) sistemlerde aynı anda birden fazla program çalışıyor durumda olabilir.

Çok prosesli sistemlerde, prosesler zaman paylaşımli olarak çalıştırılmaktadır. Yani, proses bir miktar çalıştırılır, sonra çalışmasına ara verilip diğer proses bir süre çalıştırılır. Sıra eski prosese geldiğinde kalan yerden çalışmaya devam eder.

Bir prosesin parçalı çalışma süresine kuanta süresi ya da kuantum denir. Örneğin Windows sistemlerin de çeşitli versiyonlarda değişse de tipik 20ms 'lik bir kuantum süresi kullanılmaktadır. Kuantum süresi çok yüksek tutulursa, prosesin kullanıcı etkileşimi ve çevreyle etkileşimi azalır. Kuantum süresi çok düşük tutulursa, vaktin önemli bir bölümü prosesler arası geçişte harcanır. Yani birim zamanda yapılan iş miktarı azalır. Birim zamanda yapılan iş miktarı "through-put" terimi ile ifade edilmektedir. Proses yöneticisinin prosesleri çalıştırıp ara veren bu alt sistemine, proses

çizelgeleyicisi (process scheduler) denilmektedir. Proses çizelgelemesi çeşitli prensiplerle yapılabilmektedir. En çok kullanılan yöntem, döngüsel çizelgeleme (round robin scheduling) dir.



Bu yöntemde her prosese adil olarak eşit zaman ayrılmaktadır. Çizelgeleme için başka algoritmalar da kullanılabilir. Örneğin pek çok işletim sisteminde proseslere öncelik derecesi atanabilmektedir. Bu öncelik derecesine göre proses birim zamanda diğerlerine göre daha fazla CPU zamanı kullanabilir. Fakat bu öncelik derecelendirmesinin de farklı sistemlerde ayrıntıları vardır. Bazı sistemlerde, prosese özgü kuantum süresinin artırılması veya azaltılması ile yapılmaktadır. Örneğin diğer proseslerin kuantum süresi 20ms iken, özel bir proses 60ms lik kuantum süresini kullanabilmektedir. Bazı sistemlerde ise, kuantum süresi değişmemekle birlikte, bir turda bazı proseslere daha fazla sıra gelmektedir.

Anahtar Notlar:

Bilgisayar sistemlerinde genel ve özel konular vardır. Genel anlatım, mevcut olan pek çok şeyin ortak özellikleri üzerine yoğunlaşır. Fakat gerçekteki durumlar çeşitlilik göstermektedir. İşte bu durumda belli bir sistem üzerinde yoğunlaşılabilir. Özel bilgiler edinilmeden önce temel prensipler öğrenilmesi uygundur. Örneğin, belirli bir Windows sisteminin nasıl çizelgeleme uyguladığı özel bir bilgidir. Bu durum Windows'un versiyonları arasında dahi değişebilmektedir. Şüphesiz çizelgeleme konusu bütün mevcut sistemleri kapsayacak şekilde genellenebilir.

Çok işlemcili bilgisayarlarda işletim sistemi, her iki işlemciye de proses atayarak aynı anda iki aynı prosesin çalıştırılmasını sağlayabilmektedir. Pek çok sistemde ortak bir çizelgeleme kuyruğu vardır. CPU lar da bağımsız biçimde çalışmaktadır. Bir CPU da kuantum süresi bittiğinde işletim sistemi kuyruktan ona sıradaki prosesini atar. Yine pek çok bilgisayar sisteminde bir prosesin sonraki turda aynı işlemciye atanması avantaj sağlayan bir durumdur. Bu nedenle işletim sistemi

mümkün olduğunca toplam performansı yükseltmek için prosesi aynı işlemciye atamak istemektedir (Çünkü bir prosesi bir işlemci çalıştırırken o prosese ilişkin bellek alanındaki kimi bilgiler CPU nun cache ine aktarılmaktadır. Cache deki bu bilgilerden yararlanmak avantaj sağlar.).

Anahtar Notlar:

Genel olarak cache sistemleri göz önüne alındığında, CPU nun rasgele bir bellek bölgesine erişmesi yerine, son erişilen yerlere yakın bölgelere erişmesi daha etkin olmaktadır. Bu durum “**locality of reference**” terimiyle ifade edilmektedir.

Çok prosesli sistemlerde bir programın iki noktası arasında geçen zaman o anda sistemde kaç prosesin çalıştığı ile doğrudan ilgilidir. Çünkü proses sayısı arttıkça, proses başına düşen çalışma miktarı azalır.

Zaman içerisinde an ve an izlenmesi gereken başka sistemlerden bağımsız bir çalışması olan sistemlere, gerçek zamanlı sistemler denir. Bu tür sistemler için program yazarken, olay kaçırma durumu oluşabilir. Örneğin bir aygıttan hızlı bir biçimde veri geliyor olsun, bizim sürekli olarak bu aygıtı bakmamız gerekir. Fakat çok prosesli işletim sistemlerinde, programımızın çalışmasında ara verilebilmektedir. İşte prosesimiz çalışmıyor durumda iken bu olayları kaçırabilir.

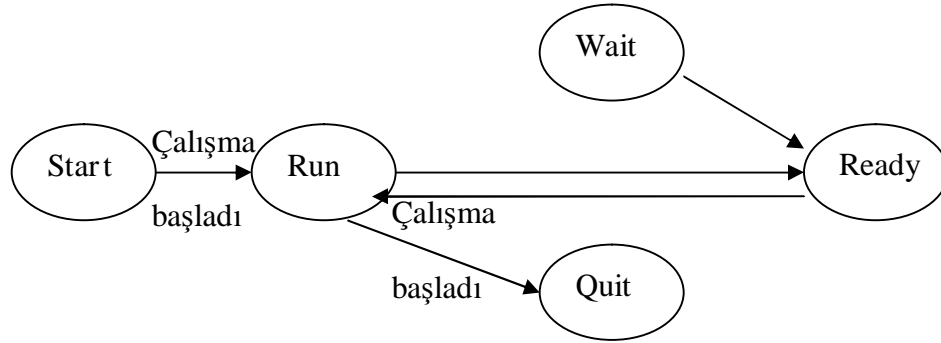
Windows, Linux gibi işletim sistemleri gerçek zamanlı olayları izlemede iyi bir ortam sunmamaktadır. Çünkü bu işletim sistemleri tipik olarak masatüstü uygulamalarda performansı yükseltmek için tasarlanmıştır. Hâlbuki gerçek zamanlı olaylar, tipik değil özel durumlar oluşturmaktadır. Hâlbuki gerçek zamanlı olayları izleyebilmek için özel işletim sistemleri de vardır. Qnix ve Rtos gibi işletim sistemlerine gerçek zamanlı işletim sistemleri denilmektedir.

12.1. Proseslerin Bloke Olması

Dosyadan bilgi okunması, soketten bilgi beklenmesi, klavyeden okuma yapma gibi pek çok olay uzun süreli bekleme yol açmaktadır. İşte bir proses, bir fonksiyon çağırarak böyle bir olayı başlattığında, işletim sistemi prosesi geçici olarak çizelge dışına çıkarır. İlgili olayın gerçekleşip gerçekleşmediğini arka planda kesme tekniklerini de kullanarak izlemektedir. Olay gerçekleştiğinde işletim sistemi prosesi yeniden çizelgeye sokar ve fonksiyon geri döner. Burada fonksiyonu çağırana için ekstra bir bekleme söz konusu değildir. Fakat işletim sistemi, proses boşuna CPU zamanı harcamasın diye onu çizelge dışına çıkartmaktadır. Örneğin işletim

sistemlerinin çoğunda sleep gibi bir isimde bekleme yapan bir fonksiyon vardır. Örneğin sleep(1000). Windows un bu sistem fonksiyonu 1 sn beklemeye yol açacaktır. İşte işletim sistemi bu prosesi geçici olarak çizelge dışına çıkartır ve 1 sn sonra geri yerleştirir. Dışsal bir olayı bekleyen bir prosesin geçici süre, olay gerçekleşene kadar, çizelge dışına çıkartılmasına prosesin bloke edilmesi denir.

Pek çok teorik kitapta bir prosesin yaşam döngüsü şematik olarak gösterilir. Kullanılan tipik şekillerden biri şöyledir.



Burada Run prosesin CPU da çalışmakta olduğunu belirtir. Wait geçici süre çizelge dışına çıkılma durumunu anlatır. Ready, prosesin çizelgede olduğunu ama çalışmak için beklediğini göstermektedir.

12.2. Preemptive ve Non-Preemptive Çok Prosesli Sistemler

Çok prosesli sistemler kendi aralarında preemptive ve non-priemptive olmak üzere ikiye ayrılır. Preemptive sistemlerde kuantum süresi dolduğunda bir donanım kesmesi yoluyla herhangi bir makine komutunda çalışmaya ara verilmektedir. Non-preemptive sistemlerde çalışmakta olan sistemin kendi isteğiyle çalışmaya ara verilir. Unix/Linux ve Win32 sistemleri preemptive sistemlerdir. Windows 3.1i, yani Win16 sistemleri non-priemptive sistemlerdir.

12.3. Modern İşletim Sistemlerindeki Koruma Mekanizması

Çok prosesli sistemlerde aynı anda birden fazla program çalışıyor durumda olduğuna göre bu programlar aynı zamanda RAM de bulunmaktadır. Peki, bir program göstericileri kullanarak başka bir programın bellek alanına erişip orayı bozamaz mı?

Örneğin bir bankanın hesap girişi yapan programı çalışırken, başka bir program bellekteki hesap bilgilerini değiştirirse ne olur?

İşte bu tür durumların bir biçimde engellenmesi gerekir. Bir programın başka bellek alanlarına erişmesini, sistemi çökertecek makine komutlarını kullanmasını engelleyen mekanizmaya koruma mekanizması denilmektedir.

Koruma mekanizmasının oluşturulması birinci elden mikroişlemcinin yardımıyla sağlanmaktadır. İşletim sistemi işlemcinin sağladığı olanaklardan faydalanır.

Intel işlemcileri 80286 ve sonraki modellerinde koruma mekanizmasını sağlamaktadır. Intel 80x86 işlemcilerinin üç tane çalışma modu vardır:

1. Gerçek Mod (Real Mode)
2. Korumalı Mod (Protected Mode)
3. V86 Modu

İşlemci reset edildiğinde gerçek modda çalışmaya başlar. Gerçek mod koruma mekanizmasının olmadığı DOS modudur. Korumalı mod, mikroişlemcinin tüm kapasitesinin kullanıldığı moddur. V86 modu hem korumalı mod programlarının hem de gerçek mod programlarının çalıştırıldığı ara bir moddur.

Intel işlemcileri RAM i sayfa denilen bloklara ayırmaktadır. Her sayfa 4K uzunluktadır ve her sayfaya bir numara verilmiştir.

Intel işlemcilerinde çalışmakta olan kodun bir öncelik derecesi vardır. Toplam 4 öncelik derecesi vardır. Bunlar 0, 1, 2, 3 ile belirtilir. En yüksek öncelik sıfıncı önceliktir. En düşük öncelik 3 numaralı önceliktir. Windows, UNIX/LINUX yalnızca iki öncelik seviyesini kullanmaktadırlar (0, 3). Sıradan programlara “user mode” programlar denir ve bunlar 3 öncelik derecesine sahiptir. İşletim sisteminin kendi kodları ve aygıt sürücüler 0 öncelik derecesine sahiptir ve bunlara “kernel mode” programlar denir.

Intel sisteminde aynı zamanda bellekteki her 4k lık sayfaya 0–3 arası bir öncelik derecesi atanabilmektedir. Bir proses kendisi ile aynı seviyeli veya düşük öncelikli bir sayfaya erişebilir,

daha yüksek öncelikli bir sayfaya erişemez. Eğer erişmeye çalışırsa mikroişlemci bunu tespit eder ve durumu işletim sistemine bildirir (page fault). İşletim sistemi de prosesi cezalandırarak sonlandırır.

İşletim sistemi kendi kodlarını 0 öncelikli sayfalara yerleştirir. Sıradan proseslerin bellek alanlarını 3 öncelikli yapar. Böylece sıradan programlar işletim sisteminin bulunduğu bellek alanlarına erişip oraları bozamazlar. Fakat işletim sistemi her yere erişebilir.

Bir “user mode” proses işletim sisteminin sistem fonksiyonunu çağırdığı zaman ne olur? Sistem fonksiyonu işletim sisteminin bir parçası olduğuna göre işletim sisteminin çeşitli veri yapılarını güncelleyecektir. Peki, bu durum koruma hatasına yol açmaz mı? İşte Intel işlemcileri kapı (gate) denilen bir yöntemle geçici öncelik yükseltmesi sağlayabilmektedir. Şöyle ki; kapılarda sistem fonksiyonlarının başlangıç adresleri bulunur. Kapıya dallanma yapıldığında otomatik olarak prosesin önceliği sıfıra yükseltilir. Böylece sistem fonksiyonu çalışırken proses “kernel mode” a geçmiş olur. Artık işletim sisteminin kodu çalışmaktadır ve bellek koruması ortadan kalkmıştır. Sistem fonksiyonu çağrıldığında oluşan bu geçişe “prosesin kernel moda” geçmesi denilmektedir. Kod geri dönüşte otomatik olarak öncelik derecesi 3 e geri çekilir. Kernel moda geçiş göreceli olarak yavaş bir işlemdir. Çünkü akışın gerçek işi yapan koda geçmesine kadar pek çok makine kodu çalıştırılmaktadır.

Bazı makine komutları doğrudan sistemin çökmesine yol açabilmektedir. Bu makine komutları ancak 0 öncelikli prosesler tarafından kullanılabilir. Örneğin IN, OUT makine komutları böyle komutlardır. Örneğin seri port ve paralel port gibi veya bilgisayara taktığımız kartlar gibi donanım birimleri ile haberleşmek için IN ve OUT komutları gerekmektedir. Peki, biz bu tür programları nasıl yazacağız? İşte bunun için 0 öncelikli kodlama yapmamız gerekir. Yani kodumuzu 0 öncelikli bir biçimde çalıştırmalıyız. Fakat herkes bunu yapsa sistemi çökertebilir. İşte böyle özel programlara aygıt sürücüsü (device driver) programlar denilmektedir. O halde her şeyi yapabilecek bir programın aygıt sürücüsü olarak yazılması gerekir. Aygıt sürücüler öncelikli ve yetkili kullanıcılar tarafından çalıştırılabilen programlardır. Biz bir aygıt sürücüsü yazdığımızı ve makineyi çökertebileceğimizi düşünelim. Bu aygıt sürücüyü sistem yöneticisinin yüklemesi gerekir. Sistem yöneticileri de emin olmadıkları programları yüklemeyiz.

Anahtar Notlar: Sistem Fonksiyonları

İşletim sistemleri normal uygulama programları gibi fonksiyonel bir yapıyla yazılırlar. Sistem içerisinde bazı fonksiyonlar hem sistemin kendi çalışması sırasında sistem tarafından hem de programcı tarafından çağrılabilir. İşletim sisteminin böyle fonksiyonlarına sistem fonksiyonları ya da API (Application

Programming Interface) fonksiyonları denilmektedir. Sistem fonksiyonlarının isimleri ve parametrik yapıları sistemden sisteme değişebilmektedir.

12.4. Standart Dosya Fonksiyonlarının Tamponlama İşlemlerinin Ayrıntıları

“fopen” ile bir dosyayı açtığımızda fopen açılan her dosya için bir tampon oluşturur. Tampon <stdio.h> da bildirilen BUFSIZ sembolik sabitinin belirttiği uzunluktadır. Şüphesiz BUFSIZ değerini değiştirirsek tampon uzunluğu değişmez. BUFSIZ yalnızca bize bu değeri vermek için düşünülmüştür. BUFSIZ değeri pek çok derleyicide bir sektör uzunluğu olan 512 değerindedir. Standartlarda tamponlama mekanizmasının ayrıntılarından bahsedilmemiştir. Yalnızca ana hatlarıyla bir tamponun kullanılması gerektiğinden bahsedilmiştir. Genellikle “single cache line” kullanılmaktadır. Tampon read/write amacı ile kullanılır. Tamponun bellekteki başlangıç adresi dosyanın hangi bölümünün tamponda olduğu gibi bilgiler file yapısının içinde tutulmaktadır.

Bu durumda biz bir dosyayı bayt bayt ele alacaksa uzun bir miktar okuyup kendi tamponumuzu oluşturmamıza gerek yoktur. Bu işi zaten standart dosya fonksiyonları yapmaktadır. Tabi programcı fonksiyon çağrılarında zaman kaybetmek istemeyebilir. İşte bu nedenle fgetc fonksiyonunun getc isimli bir makrosu da bulundurulmuştur. getc bir makrodur ve doğrudan tampondan sıradaki bayt karakterini almaktadır. Standartlara göre getc fonksiyonunun bir makro olması zorunlu değildir. Makro olabileceği belirtilmiştir. Yani getc fonksiyonu fgetc fonksiyonundan daha hızlı çalışma eğilimindedir.

Standartlara göre her dosya için tamponlama stratejisi 3 biçimde olabilir.

1. Tam Tamponlama (Full Buffering): Bu modda tampon tam kapasite ile kullanılır. Yani bilgi tamponda varsa oradan verilir yoksa bir tampon okuma yapılır bilgi oradan verilir. Tampona yazma yapılmışsa tampon boşaltılmadan tampondaki bilgiler geri yazılır.
2. Satır Tamponlaması (Line Buffering): Bu modda tamponda bir satırlık bilgi tutulur. Yani tamponda hiç bilgi olmadığını düşünelim ve fgetc fonksiyonu ile bir karakter okumak isteyelim. Gerçek dosyadan bir tamponluk bilgi değil bir satırlık bilgi okunarak tampona çekilir. Bu modda yazma yapıldığı zaman yine tampona yazılır. Tazeleme işlemi tam tamponlamada tampon dolunca ya da yeni bir bilgi tampona çekileceği zaman yapıldığı halde bu modda \n karakteri yazıldığı zaman yapılmaktadır. Yani bu modda \n karakteri dosyaya yazıldığı anda tazeleme yapılmaktadır.

3. Sıfır Tamponlama (No Buffering): Bu modda tampon hiç kullanılmaz. Her okuma ve yazma işleminde doğrudan sistem fonksiyonları çağrılarak işlem yapılır.

Bir dosya açıldığında default tamponlama stratejisi nedir?

Standartlarda stdin, stdout ve stderr dosyaları için bazı şeyler söylenmiştir. Fakat normal dosyalar için bir şey söylenmemiştir. Yani derleyiciyi yazanlar default durumu istediği gibi alırlar. Hemen hemen her derleyicide default durum tam tamponlamadır.

Bir dosyanın tamponlama stratejisi dosya açıldıktan sonra fakat dosya üzerinde henüz hiçbir işlem yapılmadan değiştirilebilir. Tampon stratejisi setbuf, setvbuf fonksiyonları ile değiştirilmektedir.

```
void setbuf(FILE *stream, char *buf);
```

Bu fonksiyon temel olarak dosyanın kullandığı tampon bölgeyi değiştirmek için düşünülmüştür. İkinci parametre NULL geçilirse 0 tamponlamalı moda geçilir. setvbuf prototipi şöyledir.

```
int setvbuf(FILE *stream, char *buffer, int mode, size_t size);
```

setvbuf fonksiyonu setbuf fonksiyonunu kapsamaktadır ve daha sonra eklenmiştir. Fonksiyonun birinci parametresi dosya bilgi göstericisi, ikinci parametresi değiştirilecek tamponun adresini belirtir. Tampon değişimi yapılmak istenmiyorsa bu parametre NULL geçilmelidir. Üçüncü parametre tamponlama stratejisini belirtir. Şunlardan biri olabilir:

`_IOFBF`

`_IOLBF`

`_IONBF`

Dördüncü parametre tamponun yeni uzunluğunu belirtir. İkinci parametre NULL girilirse, dördüncü parametre ile biz yine tampon büyüklüğünü değiştirebiliriz. Fakat bu durumda tampon fonksiyon tarafından tahsis edilir. Tabi eğer tamponun yerini biz değiştirmek istiyorsak tampon uzunluğunu dördüncü parametre ile belirtilen değere uygun tahsis etmeliyiz. Eğer tamponlama stratejisi sıfır tamponlama yapılırsa bu durumda ikinci ve dördüncü parametreler dikkate alınmaz. Fonksiyon başarılı ise 0 değerine, başarısız ise -1 değerine geri döner.

Standartlara göre setvbuf fonksiyonu başarılı olmak zorunda değildir. Örneğin kütüphaneyi tasarlayanlar normal dosyalara ilişkin satır tamponlaması faydasız olduğu için bu geçişe izin vermeyebilirler. (Örneğin gcc derleyicilerinin yeni versiyonları böyle davranmaktadır.)

12.5. Proseslerin Adres Alanı

Çalışabilen bir program kabaca üç bölümden oluşur:

1. Kod
2. Data
3. Stack

Programdaki tüm fonksiyonların makine fonksiyonları kod bölümünde, statik ömürlü nesnelere (global değişkenler, statik yerel değişkenler, stringler) data bölümünde, yerel ve parametre değişkenleri stack bölümünde bulunurlar. Çalışılan program blok olarak belleğe yüklenir. Data bölümündeki nesnelere program sonlanana kadar bellekte kalırlar. Programın akışı bloğa girdiğinde yerel değişkenler stack üzerinde yaratılırlar ve bloktan çıktığında bellekten silinirler. Fonksiyon iki kez çağrıldığında, fonksiyonun yerel değişkenleri stack in tamamen farklı bir yerinde oluşturulabilir. Bu alanların dışında bir de dinamik tahsisatlarda kullanılan heap denilen başka bir alan daha vardır. Heap bölgesinin nerede olduğu, burada tahsisatların nasıl yapıldığı, büyüklüğü sistemden sisteme değişebilir. Pek çok sistem Heap>Data>Stack biçiminde bir bölüm yapısına sahiptir. Bu durumda örneğin büyük dizilerin, matrislerin önce heap üzerinde sonra data bölgesi üzerinde tahsis edilmesi denenmelidir. Büyük dizilerin yerel olarak tahsis edilmeleri iyi bir teknik değildir. Heap bölgesi sistemlerde prosese özgüdür. Yani bir programda yapılan dinamik tahsisatın o anda çalışmakta olan programa etkisi yoktur. Bir proses sonlandığında heap bölgesi de prosesle birlikte tamamen boşaltılır. Threadli sistemlerde her thread, aynı data ve heap bölümünü ortak kullanır. Ancak her threadin stack i birbirinden ayrılmıştır.

Anahtar Notlar: Thread Kavramı

Proses kavramının dışında bir de proses, içindeki farklı akışları anlatan thread kavramı vardır. Thread bir prosesin sanki farklı bir prosesmiş gibi çizelgelenen bir akışıdır. Prosesler tek bir thread ile çalışmaya başlarlar. Buna prosesin ana threadi denir ve diğer threadler sistem fonksiyonları çağrılarak herhangi bir zaman oluşturulabilirler. Thread oluşturan fonksiyonlarda threadin başlangıç noktası bir fonksiyon adresi olarak verilir. Thread bir fonksiyon değil bir akış belirtir. Farklı iki thread aynı fonksiyon üzerinde ilerleyebilir. Tabii bu threadler yerel değişkenlerin farklı bir kopyasını kullanırlar. Yani bir threadin yerel değişkeni diğer

bir threadin yerel deęişkeniyle karışmaz. Tüm threadler aynı global deęişkenleri görürler. Win32 sistemleri, Linux sistemleri, MacOS sistemleri birden fazla threadle çalışmaya izin veren sistemlerdir. Çok threadli sistemlerde exit() fonksiyonu tüm threadleri sonlandırmaktadır.

13. STDIN, STDOUT ve STDERR DOSYALARI

Dos, Windows, UNIX/LINUX işletim sistemlerinde klavye ve ekran (her ikisine birden terminal denilmektedir) birer dosya gibi ele alınmaktadır. Örneğin ekran sanki bir dosyadır, biz bu dosyaya yazma yaptığımızda ekrana yazmış oluruz. Klavye de bir dosya gibidir, biz bu dosyadan okuma yaptığımızda klavyeden okuma yapmış oluruz.

getchar, gets ve scanf gibi giriş fonksiyonları aslında birer dosya fonksiyonudur ve stdin dosyasından okuma yapmaktadır. printf, puts, putchar gibi fonksiyonlar da birer dosya fonksiyonudur. Default olarak stdout dosyasına yazma yapmaktadırlar. Standartlarda stdin ve stdout dosyalarının hangi aygıtla baęlı olduęu hakkında bir şey söylenmemiştir. Fakat tipik olarak stdin klavye ve stdout da ekranı temsil etmektedir. Fakat bu durum zorunlu deęildir. Yani örneğin printf(...) ile fprintf(stdout, ...) aynı anlamdadır.

stderr hata mesajlarının yazılması için düşünölmüş bir dosyayı temsil eder. Standartlarda stderr dosyasının da başlangıçta hangi dosya ile ilişkin olduęu konusunda bir şey belirtilmemiştir. Mevcut sistemlerde stderr de işin başında ekranı temsil etmektedir.

stdin, stdout ve stderr FILE * türünden stdio.h içerisinde bildirilmiş deęişkenlerdir. Örneğin tipik olarak stdio.h içerisinde şöyle bir görüntü ile karşılaşılmaktadır.

```
extern FILE stdfiles[3]
#define stdin (&stdfiles[0])
#define stdout (&stdfiles[1])
#define stderr (&stdfiles[2])
```

13.1. Yönlendirme İşlemi

stdin, stderr ve stdout dosyaları programı çalıştırırken ya da daha sonra program çalışırken başka bir aygıt ya da dosyaya yönlendirilebilir.

Bir programı çalıştırırken Dos, Windows ve UNIX/LINUX 'ta komut satırında büyüktür işareti ile yönlendirme yapılabilir. Örneğin;

a > b (stdout b dosyasına yönlendirilmiş oldu)

Burada a programı çalıştırılacak fakat stdout b dosyasına yönlendirilecektir. Yani stdout dosyasına yazılanlar artık ekrana değil b dosyasına yazılacaktır. Komut satırından < işareti stdin dosyasını yönlendirmek için kullanılır. Örneğin

a < b

Burada a programı çalıştığı anda artık stdin dosyasında okuma yapılmak istenirse gerçekte b dosyasından okuma yapılacaktır. Büyüktür ve küçüktür işaretleri birlikte kullanılabilir.

A > b < c

Burada stdout b dosyasına stdin c dosyasına yönlendirilmiştir. Dos, Windows 'ta programı çalıştırırken stderr yi yönlendirmenin bir yolu yoktur. Fakat UNIX/LINUX sistemlerinde 2> sembolü ile stderr de yönlendirilebilir.

Biz a programını a > b biçiminde çalıştıralım. Bu durumda stdout b dosyasına yönlendirilir fakat stderr ekran olarak kalır.

UNIX sistemlerinde ve modern sistemlerin çoğunda, pipe denilen bir işlem söz konusudur. Örneğin:

C:\> a | b

Burada çubuk sembolüne pipe denir ve şu işlem yapılır: Sistem a ve b programlarını çalıştırır ama a 'nın stdout dosyasını b 'nin stdin dosyasına yönlendirir. Böylece a 'da ekrana yazılanlar, b 'de klavyeden girilmiş gibi işlem görecektir.

Aslında program çalıştırdıktan sonra yazılım yolu ile de yönlendirme yapılabilir. Standart freopen fonksiyonu yönlendirme yapmakta kullanılabilir.

```
FILE *freopen(const char *path, const char *mode, FILE *stream);
```

Fonksiyonun birinci parametresi hedef dosyanın yol ifadesidir. İkinci parametre açış modunu belirtir. Üçüncü parametre yönlendirilecek dosyayı belirtir. Fonksiyon birinci parametresi ile belirtilen hedef dosyanın bilgi göstericisi ile geri döner. Fonksiyon başarısız olabilir. Bu durumda NULL adresi ile geri dönmektedir.

```
FILE *f;
int i;

if ((f = freopen("test.txt", "w", stdout)) == NULL) {
    printf("cannot redirect file !...\n");
    exit(EXIT_FAILURE);
}

for (i = 0; i < 10; i++)
    printf("%d\n", i);
```

Burada stdout dosyası test.txt dosyasına yönlendirilmiştir. Artık printf fonksiyonu ekrana değil test.txt dosyasına yazacaktır. Örneğimizdeki f dosya bilgi göstericisi kullanılmamıştır fakat açılan test.txt dosyasına ilişkindir.

13.2. Stderr dosyasının kullanılması

Programcı çeşitli hata ve uyarı mesajlarını stdout dosyasına değil stderr dosyasına yazdırmalıdır. Her ne kadar default durumda normal mesajlar ile hata mesajlarının her ikisi de ekrana çıkacak olsa da bunların gerektiği yerde birbirinden ayrılması sağlanabilir.

Program içerisindeki hata mesajlarının printf() fonksiyonu ile stdout dosyasına değil de fprintf() ile stderr dosyasına yazdırılması daha iyi bir tekniktir. Örneğin:

```
if((f = fopen(ptr)) == NULL){
    fprintf(stderr, "....");
    exit(EXIT_FAILURE);
}
```

Böylece tüm hata mesajları başka bir dosyaya yönlendirilebilir.

Anahtar Notlar:

Örneğin pek çok windows ve linux komutu normal mesajları stdout dosyasına, hata mesajlarını stderr dosyasına yazdırmaktadır. Bizde yönlendirme sayesinde bu hata mesajlarını başka bir dosyaya yönlendirerek onların ekrana çıkmasını engelleyebiliriz. Örneğin

```
find / -name "sample.c" 2> /dev/null
```

/dev/null dosyası özel bir dosyadır ve bu dosyaya yazılanlar otomatik olarak silinmektedir. Bu dosya windows ta bulunmamaktadır.

Stdin, stdout ve stderr dosyaları program başladığında açılmış kabul edilir ve programcı bunları kapatmaya çalışmamalıdır.

13.3. Stdin, stdout ve stderr dosyalarının tamponlama mekanizmaları

Stdin, stdout ve stderr dosyaları tıpkı diğer dosyalar gibi tampon (cache) kullanmaktadır. Aslında aygıtların dosya gibi değerlendirilmesi temelde işletim sisteminin sağladığı bir mekanizmadır. Örneğin getchar gibi stdin dosyasından okuma yapan bir fonksiyon çağrıldığında bu fonksiyon işletim sisteminin sistem fonksiyonunu çağırarak okumayı yapmak ister. İşletim sisteminin fonksiyonu klavyeden girilen karakterleri standart C fonksiyonuna verir. Standart C fonksiyonları da bunları tamponlayarak kullanır.

Standartlara göre stdin ve stdout dosyaları için default tamponlama stratejisi işin başında eğer bu dosyalar karşılıklı etkileşimli bir aygıtla yönlendirilmişse (klavye ve ekran böyledir) tam tamponlamalı olamaz. Satır tamponlamalı veya sıfır tamponlamalı olabilir. Karşılıklı etkileşimli bir aygıtla yönlendirilmemişse tam tamponlamalı olmak zorundadır. Daha sonra bu tamponlama stratejileri değiştirilebilir. Standartlara göre stderr dosyası işin başında ister karşılıklı etkileşimli bir aygıtla yönlendirilmiş olsun isterse olmasın kesinlikle tam tamponlamalı olamaz. Sıfır tamponlamalı yada satır tamponlamalı olabilir. Standartlardaki bu anlatımlardan önemli sonuçlar çıkartılabilir.

13.4. Stdout üzerindeki tamponlamanın etkisi

Eğer stdout başlangıçta sıfır tamponlamalı ise putchar, printf gibi fonksiyonlar onları hiç tampona yazmadan doğrudan işletim sisteminin sistem fonksiyonunu kullanarak ekrana yazdıracaktır. Fakat stdout başlangıçta satır tamponlamalı ise printf, putchar gibi fonksiyonların yazdırdıkları şeyler önce tamponda biriktirilecek \n karakteri görüldüğünde tampondan alınarak hepsi ekrana yazdırılacaktır. Bu durum aşağıdaki ilginç sonucu doğurmaktadır.

```
printf("ali");
```

à

Burada akış ok ile gösterilen satıra geldiğinde ali yazısının ekrana çıkması garanti değildir. sıfır tamponlama söz konusuysa çıkar fakat satır tamponlama söz konusu ise çıkmaz. Bunun yerine aşağıdaki gibi yazılsın.

```
printf("ali\n");
```

à

Burada akış ok ile gösterilen bölgeye geldiğinde ali yazısının ekrana çıkması garanti altına alınmıştır. Tabi eğer stdout karşılıklı etkileşimli bir aygıt değil dosyaya yönlendirilmişse her iki durumda da yazının dosyaya yazılmamış olması gerekir.

Anahtar Notlar:

Microsoft derleyicilerinde ve borland derleyicilerinde stdout default olarak sıfır tamponlamalıdır. Halbuki gcc derleyicilerinde default olarak satır tamponlamalıdır.

Aşağıdaki durumda yazı kesinlikle ekranda çıkar.

```
printf("ali");  
fflush(stdout);
```

fflush fonksiyonu o anda tamponda bulunan bilgileri aygıtta aktararak tamponu boşaltmaktadır.

Standartlarda zorunlu tutulmamış olsa da stdin den okuma yapan fonksiyonların önce stdout dosyasına flush edebileceği belirtilmiştir. Gerçekten de stdin den okuma yapan fonksiyonlar genellikle stdout dosyasını flush etmektedir.

Örneğin:

```
printf("sayi giriniz :");  
scanf("%d", &sayi);
```

Her ne kadar burada sayı giriniz yazısının çıkması garanti değilse de scanf fonksiyonu stdout dosyasını flush edecek biçimde yazılmış olabileceğinden dolayı yazı ekranda gözükecektir. Garanti bir durum için yine fflush uygulanmalıdır.

13.5. Stdin üzerindeki tamponlamanın etkisi

Her ne kadar stdin dosyası için başında sıfır tamponlamalı yada satır tamponlamalı olabilirse de pratikte hemen her zaman satır tamponlamalı olarak karşımıza çıkmaktadır. Çünkü işletim sistemleri klavyeden enter tuşuna basılana kadar okuma yapmaktadırlar.

Stdin dosyasından okuma yapan fonksiyonların hepsi önce tampona bakar. Mevcut bilgi orada varsa hemen tampondan alır ve gerçek bir klavye okuması yapmaz. Fakat tamponda bilgi

yoksa işletim sisteminin sistem fonksiyonunu çağırarak klavyeden bir satırlık bilgi okur ve onu tampona yerleştirir, tampondan verir. Örneğin:

```
ch = getchar();
puthcar(ch);
ch = getchar();
putchar(ch);
```

Girişte kullandığımız enter tuşu \n anlamına gelmektedir ve \n karakteri de tampona yerleştirilir. Yukarıdaki örnekte ilk getchar çağrıldığında fonksiyon stdin tamponunda karakter olmadığını görür ve bir satırlık bilgiyi klavyeden okuyarak tampona yerleştirmek ister. Örneğin biz

A

Girişi yapmış olalım. Getchar tampona a\n karakterlerini yerleştirecektir. İlk getchar fonksiyonu bize a yı verece sonraki hiç bekleme yapmadan \n yi verecektir. Getchar, gets, scanf gibi fonksiyonların hepsi aynı stdin dosyasından okuma yaptığına göre ve stdin için ortak bir tampon olduğuna göre bu fonksiyonlar aynı tampondan çalışmaktadır. Örneğin:

```
#include <stdio.h>

int main(void)
{
    int ch;
    char s[30];

    ch = getchar();
    printf("%c\n", ch);
    gets(s);
    puts(s);

    return 0;
}
```

Burada klavyeden ali girilip enter a basılmış olsun. İlk getchar “a” yı alacak, gets ise “li” yi alacaktır.

13.6. C nin Stdin’den Okuma Yapan Fonksiyonları

C programlama dilinde stdin dosyasından okuma yapan getchar, gets, scanf gibi standart fonksiyonları bulunmaktadır.

13.6.1. getchar Fonksiyonu

Getchar fonksiyonu stdin dosyasından bir karakter okur. Prototipi şöyledir:

```
int getchar(void);
```

Fonksiyon başarılı ise okunan değere, dosya sonuna gelinmişse veya IO hatası oluşmuşsa EOF değerine döner. Başarı durumunda yüksek anlamlı baytları sıfır, düşük anlamlı baytı okunan değerle geri döner. Başarısızlık durumunda tüm baytları ff olan -1 değerine geri döner.

Anahtar Notlar:

stdin dosyası klavyeye yönlendirilirse EOF etkisi yaratmak için özel bir tuş kombinasyonu kullanılmaktadır. Bu tuş kombinasyonu dos ve windows sistemlerinde ctrl+z ve UNIX/LINUX sistemlerinde ctrl+d tuşlarıdır. Windows ta ctrl+z nin satır başında verilmesi gerekir. Örneğin

```
int ch;

while ((ch = getchar()) != EOF)
    putchar(ch);

return 0;
```

Eğer stdin dosyası bir dosyaya yönlendirilmiş ise bu döngüden dosyanın sonuna gelindiğinde çıkarılır. Eğer klavyeye yönlendirilmiş ise ctrl+z veya ctrl+d tuşları ile çıkarılır. Bu tuşlara basıldıktan sonra stdin kapatılmaz burada sadece yapay bir etki söz konusudur.

Anahtar Notlar:

Pek çok sistemde stdin sıfır tamponlamalı moda geçememektedir. Bunun nedeni işletim sisteminin enter tuşuna gereksinim duymasındandır.

13.6.2. gets Fonksiyonu

Fonksiyon prototipi şöyledir.

```
char *gets(char *str);
```

Eğer fonksiyon en az bir karakteri yerleştirirse parametresi ile belirtilen adresin aynısına geri döner. Tamponda yalnızca “\n” olsa, gets bu “\n” yi alıp NULL karakter yerleştirir ve parametresi belirtilen adresin aynısına geri döner.

Eğer gets hiçbir yerleştime yapmadan EOF görürse yani o anda dosya sonuna gelinmişse NULL adres ile geri döner.

Örneğin:

```
#include <stdio.h>

int main(void)
{
    char s[1024];

    while (gets(s) != NULL)
        printf("%s\n", s);

    return 0;
}
```

Burada ctrl+z/d yada EOF görülene kadar döngü yinelenir. gets fonksiyonunu getchar kullanarak yazalım.

```
#include <stdio.h>

char *mygets(char *str)
{
    char *ptr = str;
    int ch;

    while ((ch = getchar()) != '\n' && ch != EOF)
        *ptr++ = ch;
    *ptr = '\0';

    if (str == ptr && ch == EOF)
        return NULL;

    return str;
}

char *kaan_gets(char *str)
{
    size_t i;
    int ch;

    for (i = 0; (ch = getchar()) != '\n' && ch != EOF; ++i)
        str[i] = ch;

    if (i == 0 && ch == EOF)
        return NULL;

    str[i] = '\0';
}
```

```

    return str;
}

int main(void)
{
    char s[1024];

    while (mygets(s) != NULL)
        printf("%s\n", s);

    return 0;
}

```

Maalesef gets fonksiyonu kötü tasarlanmış bir fonksiyondur. Çünkü diziyi ne kadar büyük açarsak açalım daha uzun bir giriş yapılabilir. Yani programı her zaman çökertmek mümkün olabilmektedir.

Giriş çok büyük olduğunda tüm karakterlerin alınmasından ziyade programın çökmemesi önemlidir. Bu nedenle bu tür fonksiyonlar artık bir güvenlik parametresi ile tasarlanmaktadır.

Fonksiyonun ikinci parametresi dizinin uzunluğu biçiminde girilmelidir. Fonksiyon en fazla size-1 karakter okur ve NULL karakteride yerleştirerek diziyi taşırılmaz. Diğer davranış gets gibidir. gets_s fonksiyonunu kendimiz yazalım.

Örneğin:

```

char *safe_gets(char *str, size_t size)
{
    size_t i;
    int ch;

    for (i = 0; i < size - 1 && (ch = getchar()) != '\n' && ch != EOF; ++i)
        str[i] = ch;

    if (i == 0 && ch == EOF)
        return NULL;

    str[i] = '\0';

    return str;
}

```

Anahtar Notlar:

GNU ld linker 'ı gets fonksiyonunda uyarı vermektedir. Ayrıca gets gibi klasik yanlış tasarlanmış bazı standart fonksiyonlar için microsoft derleyicileri artık uyarı vermektedir. Yeni microsoft derleyicilerinde bu fonksiyonların güvenli biçimleri de bulunmaktadır. Güvensiz fonksiyonun ismi xxx ise Microsoft 'un bunlar için yazdığı güvenli versiyonlar xxx_s biçiminde isimlendirilmiştir.

Örneğin:

```

#include <stdio.h>

int main(void)
{
    char s[1024];

    while (gets_s(s, 1024) != NULL)
        printf("%s\n", s);

    return 0;
}

```

Maalesef Visual C 8.0 (Visual Studio 2005) bu fonksiyonlar için yanlış bir biçimde “deprecated” terimi kullanılmıştır. Halbuki “deprecated” terimi standartlara özgü resmi bir terimdir ve ileride standartlardan kaldırılması düşünülen diz özelliklerini belirtir. gets gibi fonksiyonlar resmi bir biçimde “depricated” değildirlir. Bu uyarı mesajlarını kaldırmak için iki yol izlenebilir. Programın tepesine:

```
#define _CRT_SECURE_NO_WARNINGS
```

Yazılabilir. Programın yukarısında define etmek yerine önceden tanımlanmış sembolik sabit yöntemi de kullanılabilir. Belirli bir uyarı mesajını pasif hale getirir.

```
#pragma warning
```

Önişlemci komutu kullanılabilir. İlgili waning ‘in numarası 4996 ‘dır. Bu durumda pragma komutu şöyle kullanılabilir.

```
#pragma warning(disable:4996)
```

#pragma standartlarda bahsedilmiştir. Fakat yanna gelenler derleyiciyi yazanlara bırakılmıştır.

Aslında pek çok kütüphanede stdin ‘den okuma yapan diğer fonksiyonlar getch ‘ı kullanmaktadır. Gets stdin dosyasından aldığı karakterleri parametresi ile belirtilen adrese yerleştirir. “\n” karakterini de okur fakat “\n” yerine “\0” karakterini yerleştirir. Görüldüğü gibi önce gets sonra getch kullanılsa sorun oluşmaz fakat önce getch sonra gets kullanılırsa gets tampondaki “\n” yi alarak sonlanır.

13.6.3. scanf Fonksiyonu

Fonksiyon prototipi şöyledir:

```
int scanf(const char *format, ...);
```

scanf fonksiyonu her türden bilgiyi okuma iddiasında olan genel bir fonksiyondur. scanf fonksiyonu baştaki boşluk karakterlerini atarak ilerler. Karakterleri yine tek tek stdin dosyasından

okumaktadır. Giriş formatına uygun olmayan bir karakter gördüğünde onu ungetc fonksiyonu ile tampona geri bırakıp işlevini sonlandırır. scanf fonksiyonu başarılı bir biçimde yerleştirilmiş olan parça sayısı ile geri döner.

scanf fonksiyonu baştaki boşlukları atar fakat sondaki boşlukları atmaz.

Örneğin:

```
#include <stdio.h>

int main(void)
{
    int a, b, result;
    char str[10];

    result = scanf("%d%d", &a, &b);

    printf("result = %d a = %d b = %d\n", result, a, b);

    gets(str);
    puts(str);

    return 0;
}
```

scanf fonksiyonu eğer hiçbir yerleştirme yapamazsa EOF değeri ile geri döner. Örneğin aşağıdaki döngü ile boşluk karakterleri ile ayrılmış tüm sayılar okunabilir.

Örneğin:

```
#include <stdio.h>

int main(void)
{
    int val;

    while (scanf("%d", &val) != EOF)
        printf("%d\n", val);

    return 0;
}
```

Eğer giriş uygun olmazsa, örneğin sayı yerine alfabetik karakterler girilirse sonsuz döngü oluşur. Kullanıcının Ali yazıp Enter 'a bastığını düşünelim. scanf A 'yı alıp beğenmeyecek ve onu tampona geri bırakacaktır. Bu durumda 0 ile geri dönecektir. Döngüden çıkmak içinde bir neden kalmayacaktır. Peki, uygunsuz bir değer girildiğinde de döngüden çıkmak isteyelim.

Örneğin:

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



```

int main(void)
{
    int val, flag;

    while ((flag = scanf("%d", &val)) != EOF && flag)
        printf("%d\n", val);

    return 0;
}

```

scanf ‘de format karakteri olmayan bir karakter girişte bulundurulmak zorundadır.

Örneğin:

```

#include <stdio.h>

int main(void)
{
    int day, month, year;
    int result;

    result = scanf("%d/%d/%d", &day, &month, &year);
    printf("result = %d day = %d month = %d year = %d\n",
           result, day, month, year);

    return 0;
}

```

Burada girişler arasında bölü karakterleri bulundurulmak zorundadır. Format kısmında herhangi bir boşluk karakteri boşluk karakteri görmeyene kadar boşlukları at anlamına gelir. İki format karakteri arasında karakter yoksa sanki boşluk karakteri varmış gibi değerlendirilir. Yani “%d%d” ile “%d %d” ve “%d\n%d” aynı anlamdadır. Boşluk karakterlerinin bir veya birden fazla olmasının hiçbir önemi yoktur.

Örneğin:

```

int main(void)
{
    int day, month, year;
    int result;

    result = scanf("%d / %d / %d", &day, &month, &year);
    printf("result = %d day = %d month = %d year = %d\n", result, day, month,
           year);

    return 0;
}

```

Aşağıdaki işlemde muhtemelen “\n” yanlışlıkla konulmuştur. (\n yerine boşluk (space) de olabilirdi.) scanf bu durumda stdin den sürekli okuma yapar boşluk karakterlerini atar, ilk boşluk olmayan karakteri okur, onu tampona geri bırakıp işlemini sonlandırır.

Örneğin:

```
scanf("%d\n", &val);
```

13.7. Stdin Tamponunun Boşaltılması

Stdin den yeni bir girişin istenmesi için \n görülene kadar (\n dahil) okuma yaparak karakterleri tampondan atmak gerekir. Maalesef bunu yapan standart bir C fonksiyonu yoktur. Bazen bu işlem için fflush(stdin) çağırmasının yapıldığı görülmektedir. Fakat bu çağırma standartlara göre geçerli değildir. Çünkü standartlarda açık bir biçimde stdin dosyasının “r” modunda açılmış olduğu bu modda açılan dosyalara fflush fonksiyonunun uygulanamayacağı belirtilmiştir. Fakat yine de bu çağırma bazı derleyicilerde işe yaramaktadır.

Stdin tamponunu boşaltmak için aşağıdaki işlem uygulanabilir.

```
while (getchar() != '\n')
    ;
```

Ancak burada küçük bir problem vardır (gerçi bu problem ihmal edilebilir). Eğer “\n” den önce EOF görülürse sonsuz döngü oluşur. O halde daha sağlam bir durum şöyle oluşturulur.

```
while ((ch = getchar()) != '\n' && ch != EOF)
    ;
```

Tabi bu kodun ikide bir yazılması yerine işlem fonksiyona yaptırılabilir.

Örneğin:

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

void clr_stdin(void)
{
    int ch;

    while ((ch = getchar()) != '\n' && ch != EOF)
        ;
}
```

Örneğin bir kişiden önce numara, sonra isim isteyelim. Numara scanf ile ismi gets ile alacağımızı düşünelim.

```

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

void clr_stdin(void)
{
    int ch;

    while ((ch = getchar()) != '\n' && ch != EOF)
        ;
}

int main(void)
{
    int no;
    char name[50];

    printf("no: ");
    scanf("%d", &no);

    printf("adi soyadi :");
    gets(name);

    printf("no: %d name = %s\n", no, name);
}

```

Burada tampon boşaltılmalıydı. Aşağıdaki gibi olmalıydı.

```

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

void clr_stdin(void)
{
    int ch;

    while ((ch = getchar()) != '\n' && ch != EOF)
        ;
}

int main(void)
{
    int no;
    char name[50];

    printf("no: ");
    scanf("%d", &no);

    printf("adi soyadi: ");
    clr_stdin();
    gets(name);

    printf("no: %d name = %s\n", no, name);
}

```

13.8. gets Fonksiyonu Yerine fgets Fonksiyonunun Kullanılması

fgets fonksiyonunda bir güvenlik parametresi olduğu için bazı programcılar gets fonksiyonu yerine fgets fonksiyonunu kullanmaktadır. Fakat maalesef fgets \n karakterini de yerleştirmektedir. O halde \n karakterinden de kurtulmak gerekir. Bu işlem klasik olarak şöyle yapılmaktadır.

```
#include <stdio.h>

int main(void)
{
    char s[SIZE];
    char *pStr;

    fgets(s, SIZE, stdin);
    if ((pStr = strchr(s, '\n')) != NULL)
        *pStr = '\0';

    return 0;
}
```

14. ALGORİTMA ANALİZİ

Bir problemi çözümüne götüren adımlar topluluğu için kullanılan terimdir. Algoritma sözcüğü genellikle kesin çözümler için kullanılır. Kesin çözümü garanti etmeyen algoritmik yöntemler için “heuristic” terimi tercih edilmektedir.

Algoritmalar çeşitli bakış açılarına göre sınıflandırılabilirler. Konuya göre sınıflandırma yapıldığında ilk akla gelenler şunlar olabilir:

- Sayılar teorisine ilişkin çeşitli algoritmalar (bölme, çarpma, asal bulma vs..)
- Sıraya dizme algoritmaları
- Arama algoritmaları

- Veri yapılarına özgü algoritmalar
- Optimizasyon algoritmaları
- Graf algoritmaları
- Yazılıma yönelik temel algoritmalar
- Diğerleri...

D. Knuth 'un 3 ciltlik "The Art of the Computing" isimli kitabı temel referans kitabı olarak kabul edilmektedir. Kitabın ciltleri şöyledir:

- 1- Fundamental Algorithms
- 2- Semi Numerical Algorithms
- 3- Sorting and Searching

Günümüzde her tür özel algoritma konusu için ayrı kitaplar bulunmaktadır. Bazı algoritmaların matematiksel ya da istatistiksel temeli vardır. Bazı algoritmalar tamamen temel matematiksel problemlerin sayısal yöntemle çözümüne ilişkindir. Örneğin;

Belirsiz integrale hesap yapma, doğrusal denklem sistemlerinin çözümü,...

Bir algoritma nasıl açıklanabilir? Sözel anlatım belirli bir noktaya kadar etkilidir, fakat ince ayrıntıları açıklamakta yetersiz kalır. Algoritmanın programını yazmak, daha açıklayıcı olsa da bu durumda algoritma az çok dile bağlı hale gelmiş olur. Akış şemaları gibi yöntemler dilden bağımsız gösterim sağlamak için kullanılır. Genellikle bu yöntemlerin hepsine aynı anda başvurulabilir. Fakat algoritma tamamen dilden bağımsız olarak açıklanamaz. Bu nedenle, algoritma kitapları genellikle bir programlama dilini temel alarak hazırlanır.

Knuth, klasik kitabında algoritmaların gerçekleştirilmesini, MIX ismini verdiği, sembolik makine dilinde yapmıştır. Bu yöntem detayların açıklanması için faydalı olsa da can sıkıcı olabilmektedir.

Bir algoritma diğerinden daha iyi midir? Nasıl kıyaslayabiliriz? Kıyaslama için en çok kullanılan iki ölçüt, hız ve kaynak kullanımınıdır. Hız ölçütü baskın bir şekilde tercih edilmektedir. Bu nedenle kıyaslamada hiçbir şeyden bahsedilmemişse, hız bakımından kıyaslama yapılacağı anlaşılmaktadır.

Hız kıyaslaması basit bir konu değildir. Koda bakar bakmaz algoritmanın hızını anlayamayız. Algoritmaların çoğu bir dizi gibi, bilgiler üzerinde işlemler yapar. Bu durumda bu bilgilerin istatistiksel dağılımı, sonuç üzerinde etkili olabilir.

Kıyaslama için simülasyon yöntemi her zaman geçerli bir yöntem olarak kullanılabilir. Bu yöntemde iki algoritmanın da programı yazılarak işlem zamanı belirlenir. Girdi kümesinin etkisini ortalama için, deney bir kez değil, çok fazla kez yinelenmelidir. Elde edilen ortalama değer kıyaslama için kullanılmalıdır. Bu yöntem, uygulaması çok zor bir yöntemdir. Bu nedenle, kıyaslama için sayısal yöntemler tercih edilmektedir.

Bazı problemler (özellikle optimizasyon problemleri) girdi kümesi büyüdükçe aşırı derecede bilgisayar zamanı gerektirmektedir. Maalesef bu algoritmaların, büyük girdi kümeleri için makul bir çözümü bulunmamaktadır. İşte bu tür durumlarda en iyi çözüm yerine kaliteli bir çözüm ile yetinilebilir. Örneğin, gezgin satıcı problemi “traveling salesman problem” NP (non-polynomial) tarzda zor bir problemdir. Bu problemde bir merkezden yola çıkan kişi, tüm şehirlere uğrayarak yine merkeze geri dönmek istemektedir. Hangi rotayı izlemesi gerekir?

Bazı algoritmaların dağıtık biçimleri söz konusu olabilmektedir. Örneğin algoritma bağımsız bir takım parçaları paralel yürütüp sonra sonuçları birleştirebilir. Büyük bir matrisin çarpımı bu biçimde gerçekleştirilebilir. Ancak her türlü dağıtık çözüm hızı arttırmayabilir. Çünkü işlerin makinelere dağıtılıp toplanması da zaman kaybı içermektedir. Algoritmaların bu biçimde dağıtık sürümlerine dağıtık algoritmalar (distributed algorithms) denilmektedir.

14.1. Algoritmanın Karmaşıklığı

Algoritmaları kıyaslayabilmek için kullanılan yöntemlerden diğeri, algoritmanın karmaşıklığı denilen yöntemdir. Bu yöntemde, algoritmayı temsil eden bir ya da birden fazla işlem seçilir, bu işlemin kaç kez yapıldığına bakılır. Muhtemelen bu sayı çeşitli girdilerin bir fonksiyonu olacaktır. Örneğin;

Algoritma, $O(k,l,m)$ gibi üç parametrelili olabilir ve bu işlemin sayısı da bu parametrelerin bir fonksiyonu olarak ifade edilebilir.

Örneğin bir dizi içerisindeki en büyük sayının bulunması algoritmasına bakalım:

```
max = a[0];
```

```

for (i = 1; i < n; ++i)
  if (max < a[i])
    max = a[i];

```

Burada if deyimini kritik işlem olarak seçersek karmaşıklık $O(n) = n-1$ olur. Fakat karmaşıklığın fonksiyonunu çıkarmak o kadar kolay olmayabilir. Örneğin, dizi içerisinde kesinlikle olduğu bilinen bir sayıyı sıralı bir biçimde arayalım:

```

for (i = 0; i < n; ++i)
  if (val == a[i]) {
    /*****/
    return;
  }

```

Burada en iyi olasılıkla bir kez, en kötü olasılıkla n kez kontrol yapılacaktır. Peki, ortalama nedir? İşte karmaşıklığı ortalama olarak ve en kötü olasılıkla bazen de en iyi olasılıkla ayrı ayrı belirlemek faydalı olabilir.

Bu algoritmadaki en kötü durum, $O(n) = n$

Ortalama durum için şöyle bir hesap yapılabilir:

$$(1 + 2 + 3 + \dots + n) / n = n(n-1) / (2n) = (n+1) / 2$$

Algoritmanın kesin karmaşıklığının bulunması çoğu zaman istatistiksel hesaplamalar gerektirmektedir. Girdi kümesinin dağılımı konuyu hemen istatistik ve olasılık tarafına çeker.

Algoritma analizi, genellikle yukarıdaki yöntemle ve matematiksel işlemlerle gerçekleştirilmektedir ve matematiğin bir konusudur. Pratik çıkarımlar için kategori yaklaşımı kullanılır. Kategori yaklaşımında algoritmalar, iyiden kötüye doğru çeşitli sınıflara ayrılır. Aynı sınıfta bulunan algoritmalar için daha iyi analiz yapılması gerekir. İyiden kötüye doğru karmaşıklık sınıfları şunlardır:

- $O(1)$ → sabit karmaşıklık
- $O(n) = \log_n$ → logaritmik karmaşıklık
- $O(n) = n$ → doğrusal karmaşıklık
- $O(n) = n \log_n$ → doğrusal logaritmik karmaşıklık
- $O(n) = n^2$ → karesel karmaşıklık
- $O(n) = n^3$ → küpsel karmaşıklık

- $O(n) = n^k$ â polinomsal karmaşıklık
- $O(n) = k^n$ â üstel karmaşıklık
- $O(n) = n!$ â faktöriyel karmaşıklık
- Ve diğerleri

En iyi karmaşıklık sınıfı sabit karmaşıklıktır. Eğer algoritmada bir döngü yoksa algoritma tekil işlemlerle gerçekleştirilmişse, sabit karmaşıklık söz konusudur. Örneğin üçgenin alanının bulunması böyledir.

Logaritmik karmaşıklıkta bir döngü vardır, fakat döngü logaritmik bir biçimde dönmektedir. Örneğin, sıralı diziler üzerinde ikili arama yönteminde olduğu gibi.

Eğer algoritma iç içe döngü içermiyorsa bir ya da birden fazla tekil döngü içeriyorsa doğrusal karmaşıklık söz konusudur. Örneğin, dizi toplamını bulmak böyle bir karmaşıklık içerir.

Son iki kategori, doğrusal logaritmik karmaşıklıktır. Bu algoritmalarda iç içe iki döngü olmakla beraber, döngülerden biri logaritmik dönmektedir. Özyinelemeli bazı algoritmalarda bu durumla karşılaşmaktadır. Örneğin tipik olarak “quick sort” algoritmasında olduğu gibi.

Eğer algoritmada iç içe iki döngü varsa (birden fazla olabilir ve ayrıca birden fazla tekil döngü de var olabilir), karmaşıklık kareseldir.

İç içe 3 tane, 4 tane ve k tane döngü söz konusu olabilir. Bu durumda tipik olarak küpsel ve k’sal karmaşıklık durumları oluşur.

Polinomun derecesi önemlidir. Örneğin, bir algoritma 5 tane iç içe döngü 3 tane de tekil döngü içeriyor olsun. Diğer bir tane iç içe 6 tane döngü içeriyor olsun. İkinci polinomun derecesi daha yüksektir.

Üstel ve faktöriyel karmaşıklıklar, polinomsal olmayan (non-polynomial) karmaşıklık denir. NP problemler, bilgisayarlarla çözümü adeta mümkün olmayan çözümlerdir. Örneğin, gezgin satıcı problemi, tipik bu gruptandır.

NP problemlerin önemli bir bölümünde, çözüm verildiğinde, bunun sınanması, polinomsal karmaşıklıkta yapılabilir. (Bu nedenle, NP problemlerin aslında polinomsal çözümlerinin olduğu, fakat henüz bulunamadığı biçimde bir umut da sürmektedir.)

Sonuç olarak, iki algoritmayı kıyaslamak için kabaca kategori yaklaşımını kullanabiliriz. Daha ayrıntılı kıyaslama için kesin karmaşıklık değerleri hesaplanabilir. Örneğin, kabarcık (bubble) sıralaması ve seçerek (selection) sıralama algoritmalarının her ikisi de iç içe iki döngü

gerektirmektedir. Bu durumda bu iki algoritma da karesel karmaşıklıktadır ve benzer performanstadır. Biz ikisi arasındaki farkı daha açık ifade etmek için kategori yaklaşımından çıkıp kesin karmaşıklığı hesaplamamız gerekir. Hâlbuki “quick sort” algoritması logaritmik dönen ve normal dönen iç içe iki döngü içerir. Karmaşıklık kategorisi $n \log_n$ a doğrusal logaritmiktir. Bu durumda “quick sort” algoritması diğer ikisinden daha iyi bir algoritmadır.

15. TEMEL VERİ YAPILARI

Aralarında fiziksel ya da mantıksal ilişki bulunan birden fazla nesnenin oluşturduğu topluluğa veri yapısı (data structures) denir. Bazı veri yapıları programlama dili tarafından sentaks olarak doğrudan desteklenmektedir. Örneğin diziler, yapılar ve birlikler böyledir. Diğer veri yapıları dildeki, mevcut olanaklar kullanılarak oluşturulur.

En fazla kullanılan temel veri yapıları şunlardır:

- Kuyruklar
- Stack sistemleri
- Bağlı listeler
- Hash tabloları
- İkili ağaçlar

Bu veri yapılarının dışında görece olarak daha az kullanılan başka veri yapıları da vardır. Örneğin ikili ağaçların dışında değişik ağaç yapıları, graf veri yapısı gibi...

Tipik bir veri yapısı, bir handle sistemi kullanılarak kurulabilir. Tipik bir veri yapısında, nesnelere yerleştirileceği alanlar ve işlem yapan fonksiyonlar bulunur. Nesnelere saklanacağı alan, handle sisteminin içinde organize edilebilir. İşlem yapan fonksiyonlar, handle değerini parametre olarak alır ve işlem yaparlar.

15.1. Kuyruk Veri Yapısı

Kuyruk veri yapısı, FİFO prensibi ile çalışmaktadır. Kuyruğa eleman yerleştirme ve kuyruktan eleman alma gibi iki işlem söz konusudur. Yerleştirme kuyruğun sonuna yapılır, kuyruğun başından eleman alınır. Kuyruğun belli bir kapasitesi söz konusudur. Kuyruk boşken eleman alınamaz, dolu iken eklenemez.

15.1.1. Kuyruk Veri Yapısının Kullanım Alanları

Kuyruk veri yapısı, bilgilerin geçici süre bekletildiği durumların çoğunda kullanılmaktadır. Örneğin tipik olarak, bilgi bir kanaldan gelip, başka bir kanaldan alınmaktadır. Bu işlemler paralel yürütülmektedir. Bilgi geldiğinde, bir kuyruğa yerleştirilir, sonra sırası bozulmadan kuyruktan alınır. Örneğin klavyeden basılan tuşlar kaybolmamakta, bir kuyruğa yerleştirilmektedir. Seri porttan alınan bilgiler işletim sistemi tarafından bir kuyruğa yerleştirilebilir ve porttan okuma yapıldığı sırada büyük olasılıkla kuyruklanmış bilgi alınmaktadır (veri yapısı kuyruk, bekleme işi tamponlama denir). Ya da örneğin, yazıcıya birden fazla print isteği yollanmış olsun. Bunlar bir kuyruk sisteminde saklanır. Yazıcı işini bitirdiğinde kuyruktan yeni iş isteyip yazar.

C 'de kuyruk için özel bir veri yapısı yoktur. C++ da Queue isimli template sınıf, C# ve Java'daki Queue sınıfı temel kuyruk işlemlerini yapmaktadır.

15.1.2. Kuyruk Veri Yapısının Gerçekleştirilmesi

Kuyruk yapısı 3 biçimde gerçekleştirilebilir:

1. Dizi kaydırma yöntemi ile
2. Döngüsel yöntem ile
3. Bağlı liste yöntemi ile

15.1.2.1. Dizi Kaydırma Yöntemi ile Kuyruk Yapısının Gerçekleştirilmesi

Bu yöntemde kuyruğun sonu bir gösterici ile ya da indeks ile tutulur. Eleman sona yerleştirilir, baştan alınarak dizi kaydırılır.

Anahtar Notlar:

Bazı dizisel veri yapılarının başını ve sonunu belirtmek için İngilizce head ve tail sözcükleri tercih edilmektedir.

Kuyruk veri yapısına ilişkin başlık dosyası şu şekilde olabilir:

```
#ifndef _QUEUE_H_
#define _QUEUE_H_

#include <stddef.h>

/* Sybolic Constants */

#define TRUE 1
#define FALSE 0

/* Typedef Declerations */

typedef int BOOL;
typedef int DATATYPE;

typedef struct tagQUEUE {
    size_t size;
    size_t tail;
    DATATYPE *pQueue;
} QUEUE, *HQUEUE;

/* Function Prototypes */

HQUEUE createQueue(size_t size);
void closeQueue(HQUEUE hQueue);
BOOL putQueue(HQUEUE hQueue, DATATYPE val);
BOOL getQueue(HQUEUE hQueue, DATATYPE *pItem);
BOOL isEmptyQueue(HQUEUE hQueue);

#endif
```

Kuyruk veri yapısına ait uygulama dosyası şu şekilde yazılabilir:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "queue.h"
```

```

#define PUBLIC
#define PRIVATE static

/* Function Definitions */

PUBLIC HQUEUE createQueue(size_t size)
{
    HQUEUE hQueue;

    if ((hQueue = (HQUEUE)malloc(sizeof(Queue))) == NULL)
        return NULL;

    if ((hQueue->pQueue = (DATATYPE *)malloc(sizeof(DATATYPE) * size))
        == NULL) {
        free(hQueue);
        return NULL;
    }

    hQueue->size = size;
    hQueue->tail = 0;

    return hQueue;
}

PUBLIC void closeQueue(HQUEUE hQueue)
{
    free(hQueue->pQueue);
    free(hQueue);
}

PUBLIC BOOL putQueue(HQUEUE hQueue, DATATYPE val)
{
    if (hQueue->tail == hQueue->size)
        return FALSE;

    hQueue->pQueue[hQueue->tail++] = val;

    return TRUE;
}

PUBLIC BOOL getQueue(HQUEUE hQueue, DATATYPE *pItem)
{
    size_t k;

    if (hQueue->tail == 0)
        return FALSE;

    *pItem = *hQueue->pQueue;

    /*     for (k = 0; k < hQueue->tail; k++)
        hQueue->pQueue[k] = hQueue->pQueue[k + 1];
    */
    memmove(hQueue->pQueue, hQueue->pQueue + 1,
            sizeof(DATATYPE) * (hQueue->tail - 1));

    --hQueue->tail;
}

```

```

    return TRUE;
}

PUBLIC BOOL isEmptyQueue(HQUEUE hQueue)
{
    return hQueue->tail == 0;
}

```

Şöyle test edilebilir:

```

#include <stdio.h>
#include <stdlib.h>
#include "Queue.h"

int main(void)
{
    HQUEUE hQueue;
    int i, val;

    if ((hQueue = CreateQueue(10)) == NULL) {
        fprintf(stderr, "Cannot create queue!...\n");
        exit(EXIT_FAILURE);
    }
    for (i = 0; i < 10; ++i)
        PutQueue(hQueue, i);

    while (!IsEmptyQueue(hQueue)) {
        GetQueue(hQueue, &val);
        printf("%d\n", val);
    }

    CloseQueue(hQueue);

    return 0;
}

```

15.1.2.2. Döngüsel Yöntem ile Kuyruk Yapısının Gerçekleştirilmesi

Bu yöntemde kuyruğun başını ve sonunu gösteren head ve tale isimli iki gösterici yada index alınır. Ayrıca kuyruktaki eleman sayısı da tutulur. Alma işlemi head göstericisinin gösterdiği yerden yapılır ve head göstericisi 1 ilerletilir. Yerleştirme işlemi tale göstericisinin gösterdiği yere yapılır ve tale göstericisi 1 ilerletilir. Göstericiler dizinin sonuna geldiğinde yeniden başa geçilir.

Head ve tale aynı yeri gösteriyorsa kuyruk ya tam doludur ya da boşdur. Kuyruğun dolu mu boş mu olduğu kuyruk içerisinde bulunan eleman sayısına göre belirlenir.

Queue.h dosyası şu şekilde yazılabilir:

```
#ifndef _QUEUE_H_
#define _QUEUE_H_

#include <stddef.h>

/* Sybolic Constants */

#define TRUE 1
#define FALSE 0

/* Typedef Declerations */

typedef int BOOL;
typedef int DATATYPE;

typedef struct tagQUEUE {
    size_t size;
    size_t head;
    size_t counter;
    size_t tail;
    DATATYPE *pQueue;
} QUEUE, *HQUEUE;

/* Function Prototypes */

HQUEUE createQueue(size_t size);
void closeQueue(HQUEUE hQueue);
BOOL putQueue(HQUEUE hQueue, DATATYPE val);
BOOL getQueue(HQUEUE hQueue, DATATYPE *pItem);
BOOL isEmptyQueue(HQUEUE hQueue);

#endif
```

Queue.c uygulama dosyası şu şekilde yazılabilir:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "queue.h"

#define PUBLIC
#define PRIVATE static

/* Function Definitions */

PUBLIC HQUEUE createQueue(size_t size)
{
    HQUEUE hQueue;
```

```

    if ((hQueue = (HQUEUE)malloc(sizeof(Queue))) == NULL)
        return NULL;
    if((hQueue->pQueue = (DATATYPE *)malloc(sizeof(DATATYPE) * size)) ==
        NULL) {
        free(hQueue);
        return NULL;
    }

    hQueue->size = size;
    hQueue->counter = hQueue->head = hQueue->tail = 0;

    return hQueue;
}

PUBLIC void closeQueue(HQUEUE hQueue)
{
    free(hQueue->pQueue);
    free(hQueue);
}

PUBLIC BOOL putQueue(HQUEUE hQueue, DATATYPE val)
{
    if (hQueue->counter == hQueue->size)
        return FALSE;

    /* hQueue->tail %= hQueue->size; */
    if (hQueue->tail == hQueue->size)
        hQueue->tail = 0;

    hQueue->pQueue[hQueue->tail++] = val;
    ++hQueue->counter++;

    return TRUE;
}

PUBLIC BOOL getQueue(HQUEUE hQueue, DATATYPE *pItem)
{
    if (hQueue->counter == 0)
        return FALSE;

    /* hQueue->head %= hQueue->size; */
    if (hQueue->head == hQueue->size)
        hQueue->head = 0;

    *pItem = hQueue->pQueue[hQueue->head++];
    hQueue->counter--;

    return NULL;
}

PUBLIC BOOL isEmptyQueue(HQUEUE hQueue)
{
    return !hQueue->counter;
}

```

15.1.2.3. Bağlı Liste Yöntemi ile Kuyruk Yapısının Gerçekleştirilmesi

Bağlı liste tekniğinin bir bağlı liste oluşturulur. Bağlı listenin ilk elemanı ve son elemanı handle alanında tutulur. Eleman eklerken listenin sonuna eklenir, eleman alınırken listenin başından eleman alınır.

15.2. Veri Yapılarının Türden Bağımsız Hale Getirilmesi

Veri yapılarının farklı türler ile çalışabilmesi için 2 yöntem düşünülebilir.

Birincisi her tür için fonksiyonların yeniden yazılma yöntemi: Bu yöntemde fonksiyon isimleri her tür için değiştirilmelidir. Şüphesiz hızlı bir yöntemdir. Fakat kod tekrarı bir dezavantaj oluşturmaktadır.

İkinci yöntemde, veri yapısına ilişkin fonksiyonlar türden bağımsız bir biçimde oluşturulur: Bu yöntemde fonksiyonlar veri yapısında tutulacak nesnenin türünü bilmemektedir. Bu nedenle nesnelere hep adres yolu ile aktarılmalıdır ve void göstericilerden faydalanılır. Şüphesiz veri yapısında saklanan elemanların bayt uzunluğu bilinmelidir. Bu tür türden bağımsız uygulamalarda türe bağımlı işlemler genel olarak fonksiyon göstericileri ile fonksiyonları kullanan kişiye yaptırılır. Örneğin türden bağımsız kuyruk sistemi için handle alanı şöyle olabilir:

```
typedef struct tagQUEUE {
    size_t size;
    size_t head;
    size_t count;
    size_t tail;
    size_t width;
    void *pQueue;
} QUEUE, *HQUEUE;
```

Fonksiyonların prototipleri şöyle olabilir:

```
HQUEUE CreateQueue(size_t size, size_t width);
void CloseQueue(HQUEUE hQueue);
BOOL PutQueue(HQUEUE hQueue, const void *pItem);
BOOL GetQueue(HQUEUE hQueue, void *pItem);
BOOL IsEmptyQueue(HQUEUE hQueue);
```


GenericQueue.c başlık dosyası şu şekilde yazılabilir:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "GenericQueue.h"

#define PUBLIC
#define PRIVATE static

/* Function Definitions */

PUBLIC HQUEUE CreateQueue(size_t size, size_t width)
{
    HQUEUE hQueue;

    if ((hQueue = (HQUEUE)malloc(sizeof(Queue))) == NULL)
        return NULL;
    if ((hQueue->pQueue = malloc(width * size)) == NULL) {
        free(hQueue);
        return NULL;
    }

    hQueue->size = size;
    hQueue->width = width;
    hQueue->count = hQueue->head = hQueue->tail = 0;

    return hQueue;
}

PUBLIC void CloseQueue(HQUEUE hQueue)
{
    free(hQueue->pQueue);
    free(hQueue);
}

PUBLIC BOOL PutQueue(HQUEUE hQueue, const void *pItem)
{
    if (hQueue->count == hQueue->size)
        return FALSE;
    if (hQueue->tail == hQueue->size)
        hQueue->tail = 0;

    memcpy((char *)hQueue->pQueue + hQueue->tail * hQueue->width, pItem,
hQueue->width);

    ++hQueue->tail;
    ++hQueue->count;

    return TRUE;
}

PUBLIC BOOL GetQueue(HQUEUE hQueue, void *pItem)
{

```

```

if (hQueue->count == 0)
    return FALSE;
if (hQueue->head == hQueue->size)
    hQueue->head = 0;

/*
    hQueue->head = hQueue->head % hQueue->size;
*/

memcpy(pItem, (char *)hQueue->pQueue + hQueue->head * hQueue->width,
hQueue->width);

hQueue->head++;
--hQueue->count;

return TRUE;
}

PUBLIC BOOL IsEmptyQueue(HQUEUE hQueue)
{
    return hQueue->count == 0;
}

```

Test kodu şu şekilde yazılabilir:

```

#include <stdio.h>
#include <stdlib.h>
#include "GenericQueue.h"

int main(void)
{
    HQUEUE hQueue1, hQueue2;

    if ((hQueue1 = CreateQueue(10, sizeof(int))) == NULL) {
        fprintf(stderr, "Cannot create queue!...\n");
        exit(EXIT_FAILURE);
    }

    if ((hQueue2 = CreateQueue(10, sizeof(double))) == NULL) {
        fprintf(stderr, "Cannot create queue!...\n");
        exit(EXIT_FAILURE);
    }

    {
        int i, val;

        for (i = 0; i < 10; ++i)
            PutQueue(hQueue1, &i);

        while (!IsEmptyQueue(hQueue1)) {
            GetQueue(hQueue1, &val);
            printf("%d\n", val);
        }
    }
}

```

```

{
    double i;
    double val;

    for (i = 0.1; i < 10.1; ++i)
        PutQueue(hQueue2, &i);

    while (!IsEmptyQueue(hQueue2)) {
        GetQueue(hQueue2, &val);
        printf("%f\n", val);
    }
}

CloseQueue(hQueue1);
CloseQueue(hQueue2);

return 0;
}

```

Anahtar Notlar:

C++'ın template tabanlı veri yapıları aslında her tür için ayrı fonksiyon yazma sistemine uymaktadır. C# ve Java 'da herşey object sınıftan türetildiği için veri yapılarının genelleştirilmesi bu sınıf eşliğinde gerçekleştirilebilmektedir. Fakat ayrıca generic konusunun bu dillere eklenmesiyle tıpkı C++ da olduğu gibi template tabanlı veri yapıları da eklenmiştir.

15.3. Bağlı Listeler

Bir dizide tüm elemanlar ardışıl olarak bulunmaktadır. Eğer dizinin elemanları ardışıl olmaz ise biz elemanların yerlerini tutmak zorunda kalırız. Elemanların yerlerini ayrı bir gösterici dizisinde tutmak iyi bir çözüm değildir. O halde ardışıl olmayan bir dizi oluşturacaksa akla gelen en etkin yöntem dizinin her elemanının sonraki elemanın adresini tutmasıdır. Böylece ilk elemanın adresi de bir yerde tutulursa ardışıl olmayan bir dizi elde edilmiş olur. Bu tür veri yapılarına bağlı listeler (linked list) denilmektedir.

15.3.1. Bağlı Listelerin Kullanım Alanları

1. Bölünme (Fragmentation) nedeni ile ardışık bellek sıkıntısının çekildiği durumlarda dinamik büyütülen diziler için, bağlı liste daha etkin bir yöntemdir. Çünkü bölünmüş yerlerde bağlı listenin elemanları oluşturulabilir. Örneğin elimizde 100kB 'lık bölünmüş bir bellek olsun. Ardışık en geniş alanın 10K civarında olduğunu

düşünelim. Fakat toplam boş bellek 50K civarında olsun. Biz ancak 10K uzunluğunda bir dizi oluşturabiliriz. Fakat bağlı listeler sayesinde daha büyük uzunluklarda diziler oluşturabiliriz. Özellikle bir bellek bölgesinin (tipik olarak heap) birden fazla dinamik büyütülen, diziler tarafından paylaşıldığı durumlarda bağlı listeler daha etkin bir çözüm oluşturur.

2. Araya eleman ekleme ve silme gibi işlemlerde normal dizilerde eleman kaydırması ve sıkıştırılması yapılmak zorundadır. Hâlbuki bağlı listelerde eleman ekleme ve silme sabit karmaşıklıkta bir işlem olarak yapılabilir. Hâlbuki bu dizilerde doğrusal karmaşıklıktadır.
3. Bağlı listeler bazı durumlarda zorunlu olarak kullanılmaktadır. Örneğin kullanılan belleğin yapısından kaynaklanan bir ardışıklık söz konusu olmayabilir. Bu tür uygulamalarda mecburi olarak bağlı liste kullanırız. Uzunluğunu bilmediğimiz diziler için genellikle “malloc-realloc” tekniği bölünme konusunda sorunlara yol açar. Bu tür durumlarda bağlı listeler ilk akla gelmesi gereken yöntemdir.

15.3.2. Bağlı Listeler ile Dizilerin Karşılaştırılması

1. Eleman erişimi dizilerde çok hızlı yani sabit karmaşıklıktadır. Hâlbuki bağlı listelerde doğrusal karmaşıklıktadır. Yani bu konuda diziler daha avantajlıdır.
2. Aynı bölgenin dinamik olarak büyütülen birden fazla dizi tarafından kullanılacağı durumda bölünme problemi nedeni ile bağlı listeler daha etkindir.
3. Araya ekleme ve silme işlemlerinin yoğun olduğu durumlarda bağlı listeler tercih edilmelidir.

15.3.3. Tek Bağlı ve Çift Bağlı Listeler

Tek bağlı listelerde her eleman bir sonraki elemanı gösterir. Fakat çift bağlı listelerde (doubly linked lists) her eleman hem bir önceki hem de bir sonraki elemanı göstermektedir. Böylece çift bağlı listelerde geriye doğru gidiş de mümkündür. Uygulamada yoğun olarak çift bağlı listeler

kullanılmaktadır. Çift bağlı listenin en önemli avantajı bir elemanın adresi bilindiğinde onun silinebilmesidir. Hâlbuki tek bağlı listede ancak önceki elemanın adresini bilirsek sonraki elemanı silebiliriz.

15.3.4. Bağlı Listelerin Gerçekleştirilmesi

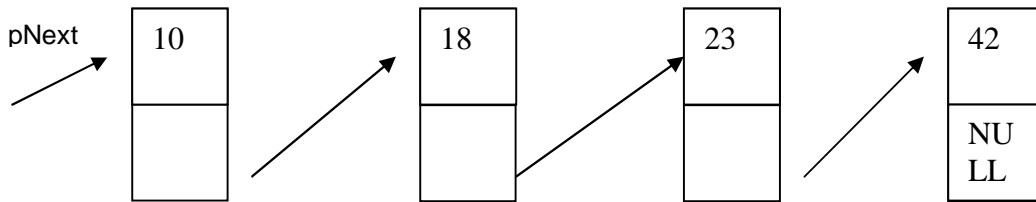
Bağlı listelerdeki her bir elemana düğüm (node) denir ve bir düğüm tipik olarak bir yapı ile temsil edilir. Örneğin tek bağlı liste için düğüm yapısı şöyle olabilir:

```
struct NODE {  
    DATATYPE val;  
    struct NODE *pNext;  
};
```

Çift bağlı liste içinde şöyle olabilir:

```
struct NODE {  
    DATATYPE val;  
    struct NODE *pNext;  
    struct NODE *pPrev;  
};
```

Bağlı listenin ilk elemanı bir gösterici ile tutulur. Genellikle sona ekleme işlemleri için de son elemanın tutulmasında fayda vardır. Ayrıca son elemanın pNext göstericisi NULL yapılması etkin bir yöntemdir.



Anahtar Notlar:

C de henüz bir yapı bildirim yapılmadan yapı türünden nesnel tanımlanamaz fakat göstericiler tanımlanabilir. Bu tür bildirimlere eksik bildirimler (incomplete declarations) denilmektedir. Örneğin:

```
typedef struct tagNODE *PNODE;
```

```

struct tagNODE {
    DATATYPE val;
    PNODE pNext;
}NODE;

```

İşlemi geçerlidir. Bildirim tamamlana kadar bu gösterici ile elemana erişim yapılamaz. Böylece birbirlerini gösteren elemanlara sahip yapılar oluşturulabilir. Örneğin:

```

struct A {
    struct B *pB;
};

struct B {
    struct A *pB;
};

```

Tabi henüz bildirim yapılmamış yapı türünden typedef de yapılabilir. Örneğin:

```

typedef struct tagNODE *PNODE;

```

15.3.5. Tek Bağlı Listelerin Handle Tekniği ile Gerçekleştirilmesi

Bağlı liste list.h ve list.c biçiminde iki dosya halinde organize edilebilir. Handle alanı aşağıdaki gibi olabilir.

```

typedef struct tagNODE {
    DATATYPE val;
    struct tagNODE *pNext;
} NODE;

typedef struct tagLIST {
    NODE *pHead;
    NODE *pTail;
    size_t count;
} LIST, *HLIST;

```

Örneğin CreateList fonksiyonu aşağıdaki gibi yazılır:

```

HLIST CreateList(void)
{
    HLIST hList;

    if ((hList = (HLIST) malloc(sizeof(LIST))) == NULL)
        return NULL;

    hList->pHead = hList->pTail = NULL;
    hList->count = 0;

    return hList;
}

```

```
}
```

Bağlı listenin başına eleman eklemekten önce aşağıdaki gibi bir kontrol yaparız:

```
if (hList->pTail == NULL)
    hList->pTail = pNewNode;
```

Burada pNewNode yeni tahsis edilmiş elemandır. Eleman ekleyen fonksiyonun yeni eklenen düğümün adresine geri dönmesi faydalı sonuçlara yol açar. Bu durumda başa eleman ekleme işlemi şöyle yapılabilir.

```
NODE *AddItemHead(HLIST hList, DATATYPE val)
{
    NODE *pNewNode;

    if ((pNewNode = (NODE *)malloc(sizeof(NODE))) == NULL)
        return NULL;
    pNewNode->val = val;

    pNewNode->pNext = hList->pHead;
    hList->pHead = pNewNode;

    if (hList->pTail == NULL)
        hList->pTail = pNewNode;

    ++hList->count;

    return pNewNode;
}
```

15.3.6. Bağlı Listenin Dolaşılması

Dolaşım işlemi tek bir döngü ile yapılabilir:

```
for (pNode = hList->pHead; pNode != NULL; pNode = pNode->pNext)
```

Dolaşım fonksiyonunun her bulduğu düğümdeki eleman adresi ile bir fonksiyonu çağırması iyi bir tasarımdır. Çağrılan geri dönüş değeri dolaşım sonlandırmak için kullanılabilir.

```
BOOL WalkList(HLIST hList, BOOL (*Proc)(DATATYPE *))
{
    NODE *pNode;
```

```

for (pNode = hList->pHead; pNode != NULL; pNode = pNode->pNext)
    if (!Proc(&pNode->val))
        return FALSE;

return TRUE;
}

```

Anahtar Notlar:

C de parametre değişkeni söz konusu olduğunda göstericiler sanki diziymiş gibi dizi deklaratörü ile belirtilebilir. Örneğin:

```

void func(int p[25]);
void func(int *p);

```

Köşeli parantez içerisine sabit ifadesi de yerleştirilebilir. Yerleştirilen sayısında hiçbir önemi yoktur. Yukarıdaki ikisi eşdeğerdir. Fonksiyon göstericileri içinde alternatif gösterim vardır. Örneğin aşağıdaki ikisi eşdeğerdir:

```

void Func(void Proc(int ,int));
void Func(void *Proc(int, int));

```

Prototiplerde isim yazmak zorunlu olmadığına göre aşağıdaki prototip bildirimleri de geçerlidir.

```

void Foo(void (int, int));
void Bar(int []);

```

15.3.7. Bağlı Listenin Sonuna Eleman Eklenmesi

Bu işlem için aşağıdaki gibi bir kod parçası yeterlidir:

```

NODE *AddItemTail(HLIST hList, DATATYPE val)
{
    NODE *pNewNode;

    if ((pNewNode = (NODE *) malloc(sizeof(NODE))) == NULL)
        return NULL;

    pNewNode->val = val;
    pNewNode->pNext = NULL;

    if (hList->pTail != NULL)
        hList->pTail->pNext = pNewNode;
    else
        hList->pHead = pNewNode;

    hList->pTail = pNewNode;
    ++hList->count;
}

```



```
    return pNewNode;
}
```

15.3.8. Bağlı Listeye Eleman Insert Edilmesi

Araya eleman ekleme işlemi eğer ekleme yapılacak yere ilişkin düğüm adresi biliniyorsa sabit zamanlı bir işlemdir. Eğer bir indekse göre insert işlemi yapılacaksa doğrusal zamanlıdır. Belirli bir indekse göre insert yapan fonksiyon şöyle olabilir:

```
NODE *InsertItemIndex(HLIST hList, size_t index, DATATYPE val)
{
    NODE *pNewNode;
    size_t i;
    NODE *pNode;

    if (index > hList->count)
        return NULL;

    if ((pNewNode = (NODE *) malloc(sizeof(NODE))) == NULL)
        return NULL;
    pNewNode->val = val;

    if (index == 0)
        return AddItemHead(hList, val);
    if (index == hList->count)
        return AddItemTail(hList, val);

    pNode = hList->pHead;
    for (i = 0; i < index - 1; ++i)
        pNode = pNode->pNext;

    pNewNode->pNext = pNode->pNext;
    pNode->pNext = pNewNode;

    ++hList->count;

    return pNewNode;
}
```

Tek bağlı listelerde biz bir düğümün adresini biliyorsak ancak onun önüne insert işlemi yapabiliriz.

```
NODE *InsertItemNode(HLIST hList, NODE *pNode, DATATYPE val)
```

```

{
    NODE *pNewNode;

    if ((pNewNode = (NODE *) malloc(sizeof(NODE))) == NULL)
        return NULL;
    pNewNode->val = val;

    pNewNode->pNext = pNode->pNext;
    pNode->pNext = pNewNode;

    if (hList->pTail == pNode)
        hList->pTail = pNewNode;

    ++hList->count;
}

```

15.3.9. Listeden Eleman Silinmesi

Eleman silme işlemi yine indekse göre yada düğüm adresine göre yapılabilir. Malesef tek bağlı listelerde düğüm adresi bilinen bir elemanı silmeyiz. Ancak onun önündeki elemanı sileriz. indekse göre silme yapan bir fonksiyon şöyle yazılabilir:

```

BOOL DeleteItemIndex(HLIST hList, size_t index)
{
    NODE *pNode, *pDelNode;
    size_t i;

    if (index >= hList->count)
        return FALSE;

    --hList->count;

    if (index == 0) {
        pDelNode = hList->pHead;
        hList->pHead = pDelNode->pNext;
        free(pDelNode);
        if (hList->pHead == NULL)
            hList->pTail = NULL;
        return TRUE;
    }

    pNode = hList->pHead;
    for (i = 0; i < index - 1; ++i)
        pNode = pNode->pNext;

    pDelNode = pNode->pNext;
    pNode->pNext = pDelNode->pNext;

    if (pDelNode == hList->pTail)

```

```

        hList->pTail = pNode;

    free(pDelNode);

    return TRUE;
}

```

Düğüm adresine göre silme işlemi de şöyle yapılabilir:

```

BOOL DeleteItemNode(HLIST hList, NODE *pNode)
{
    NODE *pDelNode;

    pDelNode = pNode->pNext;

    if (pDelNode == NULL)
        return FALSE;

    pNode->pNext = pDelNode->pNext;
    free(pDelNode);

    --hList->count;

    return TRUE;
}

```

Bağlı listede tüm düğümleri silen bir fonksiyona gerek duyulur. Tüm düğümleri silmek için hep sonraki düğümü saklamak gerekir. Fonksiyon şöyle yazılabilir:

```

void ClearList(HLIST hList)
{
    NODE *pNode, *pTempNode;

    pNode = hList->pHead;

    while (pNode != NULL) {
        pTempNode = pNode->pNext;
        free(pNode);
        pNode = pTempNode;
    }

    hList->pHead = hList->pTail = NULL;
    hList->count = 0;
}

```

Nihayet listedeki eleman sayısını veren bir fonksiyon da yazılmalıdır.

```

size_t GetListCount(HLIST hList)

```

```

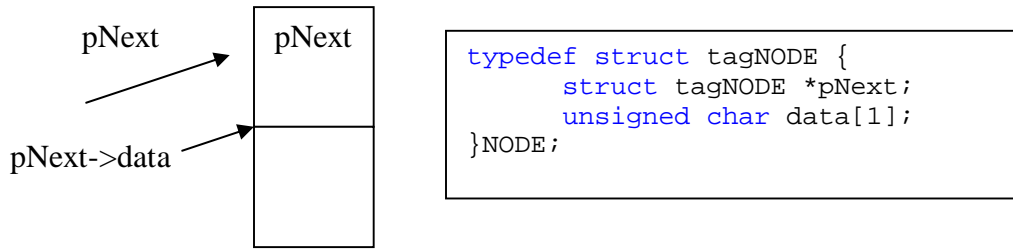
{
    return hList->count;
}

```

15.3.10. Bağlı Listelerin Genelleştirilmesi

Yukarıdaki örneklerde biz birden fazla bağlı liste oluşturabiliriz fakat hepsi DATATYPE türünden olmak zorundadır. Halbuki aynı anda farklı türlerden bağlı listeler oluşturmak isteyebiliriz. Bunun için yine void göstericiler kullanılarak genelleştirme yapılabilir ya da makro tekniği kullanılabilir. Makro tekniği LINUX, BSD, CSD gibi sistemlerde de birbirine benzer biçimlerde kullanılmaktadır.

Bu tür örneklerde düğüm olarak kullanılan yapının bazı elemanları bilinmektedir. Fakat bazı elemanları bilinmemekte yalnızca onların uzunluğu bilinmektedir. Bilinen elemanları yapının başına toplayıp bilinmeyen elemanların yeri belirlenebilir. C90 da sıfır uzunluklu bir dizi olamaz. Fakat C99 da bu tür durumlara olanak sağlamak için yapının son elemanı uzunluğu belirtilmeyen (yani 0 uzunluklu) bir dizi olabilir. Bu durumda NODE yapısı şöyle olabilir.



Şimdi bağlı listede tutacağımız verinin uzunluğunun width kadar olduğunu düşünelim ve bu verinin adresi pData ile temsil edilsin. Yeni bir düğüm tahsis edip bu veriyi o düğüme şöyle yerleştirebiliriz:

```

NODE *pNode;

pNode = (NODE *)malloc(sizeof(NODE *) + width);
memcpy(pNode->data, pNode, width);

```

Türden bağımsız bağlı listenin handle alanı şöyle olabilir:

```
typedef struct tagNODE {
    struct tagNODE *pNext;
    unsigned char data[1];
} NODE;
```

```
typedef struct tagLIST {
    NODE *pHead;
    NODE *pTail;
    size_t width;
    size_t count;
} LIST, *HLIST;
```

Bağlı listeyi yaratan fonksiyon şöyle yazılabilir:

```
HLIST CreateList(size_t width)
{
    HLIST hList;

    if ((hList = (HLIST) malloc(sizeof(LIST))) == NULL)
        return NULL;

    hList->pHead = hList->pTail = NULL;
    hList->width = width;
    hList->count = 0;

    return hList;
}
```

Bağlı listeye eleman ekleyen fonksiyon şöyle yazılabilir:

```
NODE *AddItemHead(HLIST hList, const void *pData)
{
    NODE *pNewNode;

    if ((pNewNode = (NODE *) malloc(sizeof(NODE *) + hList->width)) == NULL)
        return NULL;
    memcpy(pNewNode->data, pData, hList->width);

    pNewNode->pNext = hList->pHead;
    hList->pHead = pNewNode;
    if (hList->pTail == NULL)
        hList->pTail = pNewNode;

    ++hList->count;

    return pNewNode;
}
```

Diziyi dolaşan fonksiyon şöyle yazılabilir:

```
BOOL WalkList(HLIST hList, BOOL (*Proc)(void *))
{
    NODE *pNode;

    for (pNode = hList->pHead; pNode != NULL; pNode = pNode->pNext)
        if (!Proc(pNode->data))
            return FALSE;
}
```

```
    return TRUE;
}
```

15.3.11. Genel Bağlı Liste Oluşturulmasında Diğer Bir Yöntem

Genel bağlı liste için diğer bir yöntem de bir düğüm yapısı oluşturup bağlanacak elemanları soyutlayarak bu düğümleri birbirine bağlamaktır. LINUX ve BSD işletim sistemlerinde bulunan bağlı listeler bu biçimde oluşturulmuştur. Bu yöntem de handle sistemi kullanılmaz. Fakat ilk düğümü tutan nesne bir handle alanı gibi kullanılır.

Tek bağlı liste için öncelikle aşağıdaki gibi bir yapı oluşturulabilir:

```
typedef struct tagDLIST_HEADER {
    struct tagLIST_NODE *pHead, *pTail;
} DLIST_HEADER;
```

TagDLISTNODE yapısı şöyledir:

```
typedef struct tagDLIST_NODE {
    struct tagDLIST_NODE *pNext, *pPrev;
} DLIST_NODE;
```

Aşağıdaki iki makro tanımlamayı kolaylaştırabilir.

```
#define INIT_DLIST()                {NULL, NULL}
#define DEFINE_DLIST(name)         DLIST_HEADER name = INIT_DLIST()
```

Listenin başına şöyle eleman ekleyebiliriz:

```
void AddNodeHeadDL (DLIST_HEADER *pHeader, DLIST_NODE *pNewNode)
{
    if (pHeader->pHead != NULL) {
        pHeader->pHead->pPrev = pNewNode;
        pNewNode->pNext = pHeader->pHead;
    }
    else {
        pNewNode->pNext = NULL;
        pHeader->pTail = pNewNode;
    }

    pNewNode->pPrev = NULL;
    pHeader->pHead = pNewNode;
}
```

Sona ekleme işlemi şöyle yapılabilir:

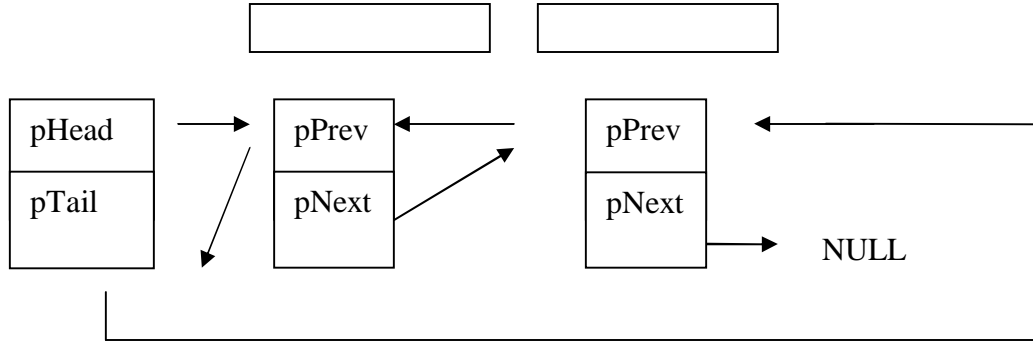
```
void AddNodeTailDL(DLIST_HEADER *pHeader, DLIST_NODE *pNewNode)
{
    if (pHeader->pTail != NULL) {
        pHeader->pTail->pNext = pNewNode;
        pNewNode->pPrev = pHeader->pTail;
    }
    else { /* list contains no item */
        pNewNode->pPrev = NULL;
        pHeader->pHead = pNewNode;
    }
    pNewNode->pNext = NULL;
    pHeader->pTail = pNewNode;
}
```

Listede tutulan veri yapısının başına erişmek için aşağıdaki gibi bir makro yazılabilir:

```
#define EXTRACT_OBJECT(ptr, type, member) \
    ((type *) ((unsigned char *) (ptr) - (unsigned long) (&((type *) 0)->member)))
```

Listeyi dolaşmak için ise aşağıdaki gibi bir makro kullanılabilir:

```
#define WALK_LIST(head, pos) \
    for (pos = (head)->pHead; (pos) != NULL; (pos) = (pos)->pNext)
```



Yukarıdaki fonksiyonlar ve makrolar görünüşe göre yalnızca DLIST_NODE nesnelerini birbirine bağlamaktadır. Aslında listede tutulan gerçek nesnelerin birer DLIST_NODE elemanı olmalıdır. Bu durumda bu fonksiyonlar ve makrolar gerçek nesnelerin düğümlerini birbirine bağlar

Örneğin PERSON yapılarından bir bağlı liste oluşturalım. Şimdi bağlı liste için bir header tanımlayalım.

```
typedef struct tagPERSON {
    char name[32];
    int no;
    DLIST_NODE node;
} PERSON;
```

Burada per PERSON türünden bir nesnenin adresidir. Dolaşım işlemi şöyle yapılabilir.

```
WALK_LIST(&people, pNode) {
    per = EXTRACT_OBJECT(pNode, PERSON, node);

    printf("%s %d\n", per->name, per->no);
}
```

15.4. Stack Sistemleri

Stack sistemleri LIFO prensibi ile çalışan kuyruk sistemidir. Gerek doğada gerekse tipik bilgisayar programlarında stack sistemleri ile karşılaşmaktadır.

Stack e geleneksel olarak eleman ekleme işlemine push işlemi, stack ten eleman alma işlemine pop işlemi denilmektedir.

15.4.1. Stack Sistemlerinin Kullanım Alanları

Bazı sistemler stack çalışmasına benzerler. Bu sistemler ile doğada da sık karşılaşılır. Örneğin tabakların üst üste konulup alınması, asansöre son giren kişinin ilk olarak çıkması stack sistemine uygundur. Bazı iskambil oyunlarında oyuncunun en üstteki kağıdı alabilmesi böyle bir sistem ile sağlanabilir. Parsing işlemlerinde stack sistemleri yoğun olarak kullanılır. Örneğin Reverse Polish Notation olarak bilinen postfix hesaplama yönteminde stack kullanılmaktadır. Bu

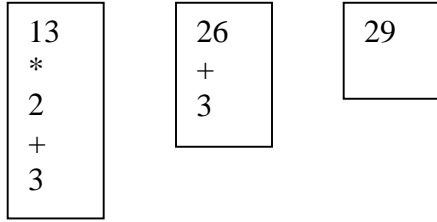
yöntemde infix operatörler bir algoritma ile postfix biçime dönüştürülür sonra stack kullanılarak işlemler yapılır. Örneğin:

$3 + 5 * 4$ à $3 5 4 * +$
Infix biçim postfix biçim
 $1 + 2 * 3 + 4$ à $1 2 3 * + 4 +$

Şekline dönüştürülür. Normal yukarıdan aşağıya pars işleminde de stack sistemi kullanılmaktadır.

$3 + 2 * (5 + 9)$

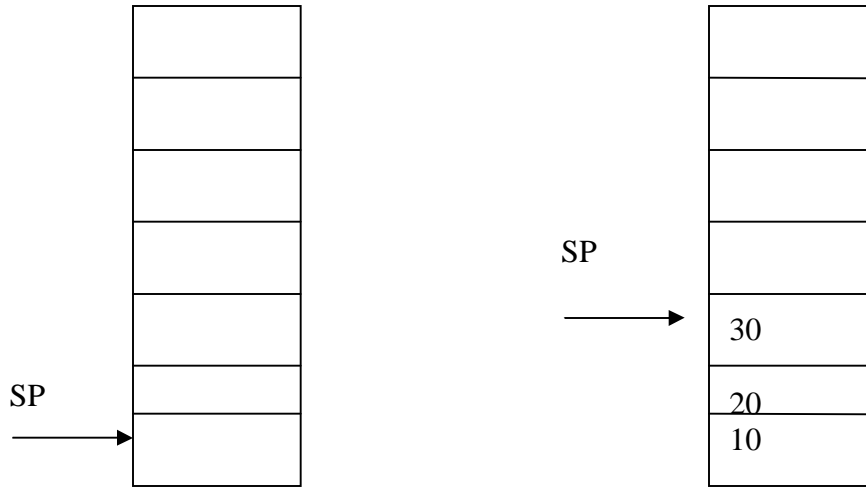
Burada tipik yöntem önce daha yüksek öncelikli operatöre bakma ve önceki operandları stack te tutmaktır. Programcı 3 ü gördüğü zaman onu stack e atar. Sonra + ve 2 yi görünce hemen işlemi yapmaz. Daha yüksek öncelikli bir operatör gördüğü için onu da stack e atarak devam eder. Sonra stack ten çeke çeke işlemleri yapar.



Editörlerdeki undo mekanizması tipik bir stack sistemidir. Her ctrl+z tuşuna basıldığında en son yapılan işlem geri alınmaktadır.

15.4.2. Stack Sisteminin Gerçekleştirilmesi

Stack sistemi klasik olarak şöyle gerçekleştirilir. Stack için bir dizi alınır. İsmine stack pointer denilen bir gösterici bu dizinin sonuna çekilir.



Push işlemi sırasında sp bir azaltılarak bilgi sp nin gösterdiği yere yazılır. Pop işlemi sırasında sp nin gösterdiği yerdeki bilgi alınır ve sp bir artırılır. Çok fazla push yapılırsa stack yukardan taşar (stack over flow), çok fazla pop yapılırsa stack aşağıdan taşar (stack under flow).

Stack sistemi için handle alanı şöyle olabilir:

```
typedef int DATATYPE;

typedef struct tagSTACK {
    DATATYPE *pStack;
    DATATYPE *SP;
    size_t size;
} STACK, *HSTACK;
```

Handle alanını oluşturan fonksiyon şöyle yazılabilir:

```
HSTACK CreateStack(size_t size)
{
    HSTACK hStack;

    if ((hStack = (HSTACK) malloc(sizeof(STACK))) == NULL)
        return NULL;

    if ((hStack->pStack = (DATATYPE *) malloc(sizeof(DATATYPE) * size)) ==
        NULL) {
        free(hStack);
        return NULL;
    }

    hStack->SP = hStack->pStack + size;
    hStack->size = size;

    return hStack;
```

```
}
```

Push işlemi şöyle yapılabilir:

```
BOOL Push(HSTACK hStack, DATATYPE val)
{
    if (hStack->SP == hStack->pStack)
        return FALSE;
    *--hStack->SP = val;

    return TRUE;
}
```

Pop işlemi için bir kontrol yapmayabiliriz:

```
DATATYPE Pop(HSTACK hStack)
{
    return *hStack->SP++;
}
```

Handle alanını kapatan fonksiyon da şöyle yazılabilir:

```
void CloseStack(HSTACK hStack)
{
    free(hStack->pStack);
    free(hStack);
}
```

Stack in boş yada dolu olduğunu anlamak için şöyle bir fonksiyon yazabiliriz:

```
BOOL isEmpty(HSTACK hStack)
{
    return hStack->SP == hStack->pStack + hStack->size;
}
```

Şöyle bir örnek yapılabilir:

```
int main(void)
{
    HSTACK hStack;
    char *name = "kaan aslan";

    if ((hStack = CreateStack(50)) == NULL) {
        fprintf(stderr, "Cannot create stack!...\n");
        exit(EXIT_FAILURE);
    }
}
```

```

while (*name != '\0')
    Push(hStack, *name++);

while (!IsEmpty(hStack))
    putchar(Pop(hStack));

putchar('\n');

CloseStack(hStack);

return 0;
}

```

15.5. Hash Tabloları

Hash tabloları arama amaçlı kullanılan veri yapılarıdır. Bir takım değerler bir anahtarla tabloya yerleştirilir. Sonra anahtar değer verilerek bilgi geri alınır. Eğer tablodaki eleman sayısı önceden kestirilebiliyorsa ve tablo yeterli uzunlukta oluşturulursa hashing yöntemi arama için en etkin yöntemlerden biridir. İkili ağaç gibi yöntemlere göre üstündür. Hash tablosu yönteminin altında yatan fikir anahtar değerden bir dizi indeksi elde edip dizinin o indeksteki elemanından bilgiyi çekmektir. Örneğin kişilerin bilgileri numaraya göre yerleştirilip aranacak olsun. Bir yapı dizisi oluşturulsa kişinin numarası o diziyeye indeks yapıp kişinin bilgileri dizinin o elemanına yerleştirilse bundan hızlı bir yöntem olamaz. Örneğin numarası 1349 olan kişinin bilgileri dizinin 1349. elemanına yerleştirilir ve geri alınırken de numara 1349 olarak girildiğinde dizinin o elemanından alınır. Fakat yukarıdaki yöntemin çeşitli problemleri vardır:

- Numara çok geniş bir aralıkta ve az sayıda kişi olabilir. Bu durum boşu boşuna çok büyük bir dizinin açılmasına neden olur.
- Numara çok geniş bir aralıktadır fakat bu kadar dizi oluşturmaya bellek yetmemektedir.

Yukarıdaki yöntem anlamlı olarak düzeltilirse faydalı bir sisteme dönüştürülebilir. Bu yöntemde yerleştirme işlemi için oluşan diziyeye hash tablosu denir. Anahtar değerden dizi indeksi elde etmeye yarayan fonksiyona hash fonksiyonu denilir. Örneğin hash tablosu 1000 elemanlı olsun. Anahtar değer 10 dijitalik sayılar olsun. İşte hash fonksiyonu bu 10 dijitalik sayıdan sıfır ile 999 arasında bir indeks elde eden bir fonksiyondur. Anahtar değer kişinin ismi gibi bir yazı da olabilir. O halde hash fonksiyonu yazıyı indekse dönüştüren bir fonksiyon olacaktır.

Biz anahtar değeri hash fonksiyonuna sokup tablonun ilgili elemanına yerleştirsek bu kez bir çakışma (collision) problemi ortaya çıkacaktır. Örneğin numarayı indekse dönüştüren hash fonksiyonu 1000 e bölümünden elde edilen kalan fonksiyonu olsun. Bu durumda örneğin sonu 847 ile biten tüm anahtar değerler aynı dizi indeksini elde ederler.

İşte çakışma oluştuğunda izlenecek stratejiye göre hashing yöntemi çeşitli alt yöntemlere ayrılmaktadır. Bunlardan en fazla kullanılan ayrı zincir oluşturma (seperate chaining) yöntemidir. Ayrı zincir oluşturma yönteminde hash tablosu bir bağlı zincir dizisi biçimindedir. Yani örneğin hash tablosu bağlı listelerin head göstericilerinden oluşmuş bir dizi olabilir. Yerleştirilecek değer hash fonksiyonuna sokulur, dizi indeksi elde edilir, bilgi o bağlı listenin başına hemen eklenir. Örneğin, 8176781 numaralı kişinin bilgileri 781. bağlı listeye eklenir. Bu bağlı listede sadece bu kişinin bilgileri değil sonu 781 ile biten tüm kişilerin bilgileri bulunur. Örneğin 17486674 numaralı kişiyi arayacak olalım. Hash fonksiyonu ile 674 değeri elde edilecek ve artık 674 nolu bağlı listede doğrusal arama yapılacaktır.

Knuth a göre bağlı listedeki ortalama eleman sayısı 10 u geçmedikten sonra bu yöntem çok iyi bir yöntemdir. Bu durumda örneğin 10000 elemanın yerleştirileceği tahmin edilen bir tablonun 1000 lik açılması gerekir. Ortalama karşılaştırma sayısı 5 olacaktır.

Ayrı zincir oluşturma yönteminde eğer sisteme çok fazla eleman eklenmek istenirse sistem doğrusal arama yöntemine benzemeye başlar. Sisteme çok az eleman eklenirse sistem rastgele erişimli bir sistem gibi olur. Bu sistemde ortalama olarak sisteme kaç eleman geleceği konusunda bir öngöründe bulunulmalıdır. Hash tablosu daha sonra büyütülebilirse de bunun doğal olarak bir maliyeti olacaktır.

15.5.1. Sıralı Yoklama Yöntemi (Lineer Probing)

Bu yöntemde hash fonksiyonları ile elde edilen indekse değer yerleştirilmeye çalışılır. Hash tablosu bir yapı dizisi biçimindedir. İlgili indeks dolu ise ilk boş indekse yerleştirme yapılır. Örneğin :

Tablo 100 eleman uzunluğunda olsun. Yerleştirilecek kişi 18898 numarasına sahip olsun. Hash fonksiyonu olarak bölümden elde edilen kalan fonksiyonunun kullanıldığını düşünelim. Tablonun 68 nolu elemanı, dolu ise 69, dolu ise 70... elemanlarına bakılır. İlk boş elemanın 79

olduğunu düşünelim. Değer 79 'a yerleştirilir. Sıralamalı yoklama yönteminin en önemli dezavantajı arama sırasında başarısız aramalarda (un-successful search) tablonun sonuna kadar gitme zorunluluğudur. Gerçi bunu önlemek için çeşitli yöntemler kullanılmaktadır. Fakat bu yöntemlerin de maliyeti vardır. Ayrıca bu yöntemde tablo uzunluğundan daha fazla elemanı tabloya yerleştiremeyiz. Tabloya girecek eleman sayısı kestirilebiyorsa ve fazla değilse büyük bir tablo açmak etkin bir aramaya olanak sağlamaktadır.

15.5.2. Hash Fonksiyonları

Bir hash fonksiyonunun iyi olabilmesi için değerleri tabloya iyi bir biçimde yaydırması gerekir. Örneğin bölümünden elde edilen kalan fonksiyonu çok kötü bir hash fonksiyonudur. Gelen değerler yanlı bile olsa iyi bir hash fonksiyonunun onu tabloya iyi bir biçimde dağıtması gerekir. Örneğin çift sayının çift sayıya bölümünden elde edilen kalan hep çifttir. Analizler tablo uzunluğunun asal sayı olması durumunda hash fonksiyonlarının performansının arttığını göstermiştir. Bu nedenle tablo uzunlukları hep asal sayı olarak alınmaktadır (örneğin 100 değil 101, 1000 değil 1001 gibi). İyi bir hash fonksiyonunun kolay hesaplanabilir olması gerekir. Yazıyı sayıya dönüştüren, sayıyı sayıya dönüştüren pek çok kaliteli hash fonksiyonu önerilmektedir.

15.5.3. Hash Tablolarının Kullanım Alanları

Hash tabloları özellikle bellek üzerinde hızlı aramanın istendiği her durumda kullanılabilir. Fakat bu yöntemde tabloya yerleştirilecek eleman sayısının önceden öngörülebilmesi önemli bir faktördür. Örneğin LINUX işletim sisteminde toplam 32767 işlem aynı anda çalışabilmektedir. Bazı fonksiyonlar proses id denilen bir sayıyı alıp proses tablosuna erişmektedir. Proses id verildiğinde proses tablosunun adresini elde etmek için tipik olarak bir hash tablosu kullanılmaktadır. Ya da örneğin LINUX taki buffer cache mekanizmasında bir disk bloğunun cash te olup olmadığını anlamak için hash tabloları kullanılmaktadır. Cache te o anda 100 lerce disk bloğu bulunuyor olabilir. Bir blok okuması yapılmak istendiğinde bloğun cash te olup olmadığına bakılacaktır. İşte cache teki blokların tek tek incelenmesi yerine bir hash tablosu kullanılmaktadır.

15.5.4. Hash Tablolarının Gerçekleştirilmesi

Hash tabloları ayrı zincir oluşturma yöntemi ile kolay bir biçimde gerçekleştirilebilir. Bağlı listelerin çift bağlı olması düğüme göre silme işlemlerini kolaylaştırmaktadır. Burada tek bağlı listeli sistem uygulanacaktır.

Hash tablosu için handle alanı şöyle olabilir:

```
typedef struct tagNODE {
    DATATYPE val;
    struct tagNODE *pNext;
} NODE;

typedef struct tagHASHT {
    size_t tableSize;
    NODE **pHeads;
    size_t count;
} HASHT, *HHASHT;
```

Bu tablo şöyle oluşturulabilir:

```
HHASHT CreateHash(size_t tableSize)
{
    HHASHT hHash;
    size_t i;

    if ((hHash = (HHASHT) malloc(sizeof(HASHT))) == NULL)
        return NULL;

    if ((hHash->pHeads = (NODE **) malloc(sizeof(NODE *) * tableSize))
        == NULL) {
        free(hHash);
        return NULL;
    }

    for (i = 0; i < tableSize; ++i)
        hHash->pHeads[i] = NULL;

    hHash->tableSize = tableSize;
    hHash->count = 0;

    return hHash;
}
```

En basit bir hash fonksiyonunu şöyle oluşturalım.

```
size_t HashFunc(HHASHT hHash, const char *name)
{
    size_t count = 0;
```

```

while (*name != '\0') {
    count += *name;
    ++name;
}

return count % hHash->tableSize;
}

```

Hash tablosuna eleman ekleme işlemi şöyle yapılabilir:

```

BOOL AddHash(HHASHHT hHash, const DATATYPE *pVal)
{
    NODE *pNode;
    size_t index;

    if ((pNode = (NODE *) malloc(sizeof(NODE))) == NULL)
        return FALSE;

    pNode->val = *pVal;
    index = HashFunc(hHash, pVal->name);

    pNode->pNext = hHash->pHeads[index];
    hHash->pHeads[index] = pNode;

    ++hHash->count;

    return TRUE;
}

```

Arama yapan fonksiyon şöyle olabilir:

```

DATATYPE *FindHash(HHASHHT hHash, const char *name)
{
    NODE *pNode;
    size_t index;

    index = HashFunc(hHash, name);

    for (pNode = hHash->pHeads[index]; pNode != NULL; pNode = pNode->pNext)
        if (!strcmp(pNode->val.name, name))
            return &pNode->val;

    return NULL;
}

```

Hash tablosu şöyle kapatılabilir:

```

void CloseHash(HHASHHT hHash)
{

```



```

size_t i;
NODE *pNode, *pTemp;

for (i = 0; i < hHash->tableSize; ++i) {
    pNode = hHash->pHeads[i];
    while (pNode != NULL) {
        pTemp = pNode->pNext;
        free(pNode);
        pNode = pTemp;
    }
}

free(hHash->pHeads);
free(hHash);
}

```

15.6. Arama İşlemlerinin Temelleri

Arama algoritmaları algoritmalar ve veri yapıları konusunun en önemli konularından birisidir. Yerleştirilen bir bilginin daha sonra geri alınması pek çok uygulamada gerekmektedir. Arama işlemi için bir anahtar değer kullanılır. İşlem sonucunda o anahtara ilişkin bilgiler elde edilir.

Arama işlemi başarılı (successful) ya da başarısız (unsuccessful) olabilir. Bazı sistemlerde biz kesinlikle bilginin bulunduğunu biliriz ve aramanın başarısız olma olasılığı yoktur.

Arama konusundaki karmaşıklık için başarılı aramalardaki arama karmaşıklığı, başarılı aramalardaki en kötü olasılıktaki karmaşıklığın, başarısızlık durumundaki karmaşıklığın dikkate alınması gerekir. Örneğin sıralı yoklama yönteminde başarısız aramalarda tüm hash tablosunu gözden geçirmek gerekebilir. Çok fazla başarısız aramanın yapıldığı durumda bu yöntem şüpheli duruma düşmektedir. Bir sistem için uygun veri yapısı ve algoritmanın belirlenmesi için o sistemin genel niteliklerinin analiz edilmesi gerekmektedir.

En basit arama yöntemi sıralı arama (sequential search) yöntemidir. Bu yöntemde elemanlar tek tek gözden geçirilir. Ortalama arama başarı karmaşıklığı, $O(n) = n + 1$, en kötü olasılıkta karmaşıklık $O(n) = n$, başarısız arama karmaşıklığı $O(n) = n$ dir.

Sıralı arama için tipik algoritma şöyledir:

```

int a[SIZE];

for (i = 0; i < SIZE; ++i)
    if (a[i] == key) {
        /*****/
    }

```

```
}
```

Knuth bu yöntemin daha etkin olabileceğini söylemiştir:

```
int a[SIZE + 1];  
  
a[SIZE] = key;  
  
for (i = 0;; ++i)  
    if (a[i] == key)  
        break;  
  
if (i == SIZE) {  
    /**/  
}  
else {  
    /***/  
}
```

En iyi arama yöntemi ne olabilir. Sıralı arama en kötü yöntemdir. Şüphesiz en iyi arama yöntemi sabit karmaşıklıkta bir yöntem olabilir. Yani rastgele erişimli bir yöntem. Eğer hiçbir çakışma söz konusu değilse ve çok fazla elemanda yoksa düz bir hash tablosu ideal yöntem olabilirdi. Fakat bu yöntemde bile eğer anahtar değer bir yazı ise yazının indekse dönüştürülmesi gerekmektedir. Yani ek bir takım işlemler söz konusudur. Eğer aranacak elemanlar çok az sayıda ise en kötü sanılan sıralı arama yöntemi en iyi yöntem olabilir.

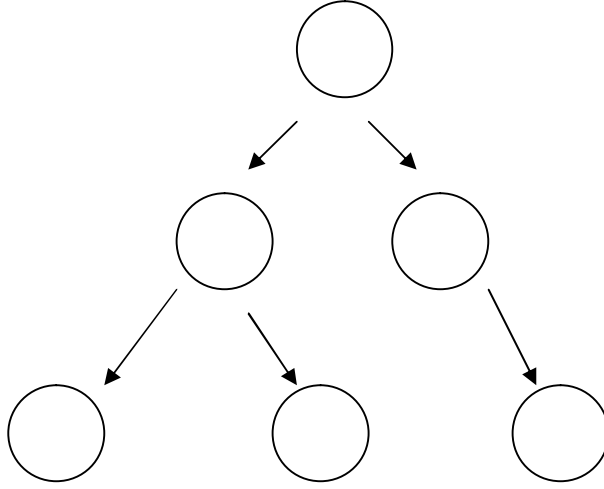
Eğer değerler zaten sıralı bir biçimde bulunuyorsa ikili arama (binary search) yöntemi çok etkin bir yöntemdir. Başarılı en kötü durum yada başarısız aramalardaki karmaşıklık durumu $O(n) = \log_2 n$ durumudur. Örneğin 1 milyon kayıt için yaklaşık olarak 20 karşılaştırma yapılır. Peki ya dizilim sıralı değilse? Önce sıraya dizip sonra ikili arama yapmak çok pahalıya mal olur. Tabii eğer sisteme yeni bir eleman eklenmeyecekse bir kere sort yapıp hep ikili arama uygulayabiliriz. Değerler sıralı ise ikili aramadan daha iyi bir yöntem olabilir mi? Eğer değerlerin dağılımı üzerinde bir bilgi sahibi isek aralık daraltma işlemini ona göre yapabiliriz. Örneğin sayılar küçükten büyüğe dizilmiş olabilir fakat herbiri bir öncekinden 3 kat daha büyük olabilir. Bu durumda ikili aramada ki gibi çubuğu ortaya çekmektense üçte ikilik bir noktaya çekmek daha etkili olabilir. Bu yöntemde üstel arama (exponential search) denilmektedir. Şüphesiz bu yöntem için dağılım hakkında bilgi sahibi olmak gerekir. Eğer bilinmiyorsa bu yöntem yarar yerine zarar getirir.

Algoritmik arama yöntemlerinde daha bilgi yerleştirilirken bir takım notlar alınmaktadır. Sonra o notlardan faydalanılarak hızlı bir arama gerçekleştirilir. Aslında hash tablosu yöntemi de bu tür bir yöntemdir. Hash değeri not edilen bilgidir.

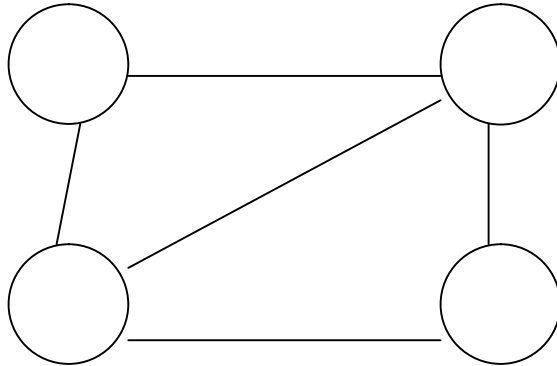
Algoritmik arama yöntemleri ağırlıklı olarak ağaçlar kullanılarak gerçekleştirilir. Disk ve bellek üzerindeki arama algoritmalarında kullanılan yöntem açısından bazı farklılıklar vardır.

15.7. Ağaç Yapıları

Belirli bir düğüme tek bir yerden gelinebilen düğüm topluluğuna ağaç denilir. Örneğin:



Eğer bir düğüme birden fazla şekilde gelinebiliyorsa, bu tür düğüm topluluğuna graf denilmektedir. Örneğin:

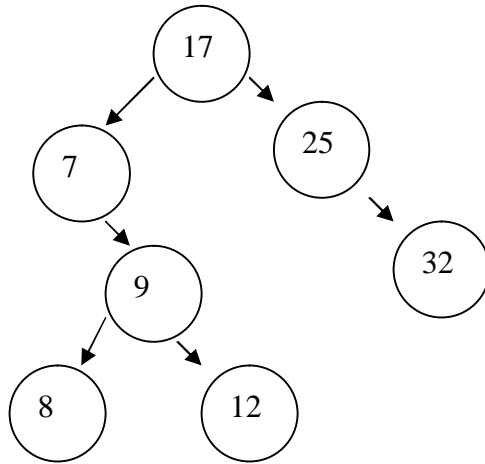


Eğer grafın üzerinde yönlendirme yoksa çift taraflı yönlendirmenin olduğu düşünülmelidir. Bir ağaçta tek bir kök vardır. İki düğümü bağlayan yola İngilizce “arc” denilir. Ayrıca bir ağaçta tek bir kökün bulunması ve o köke hiçbir gidişin olmaması gerekir.

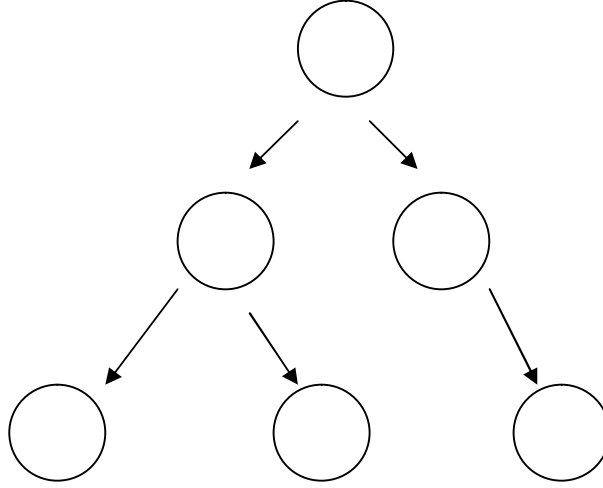
Bir ağaçta her düğümün bir yüksekliği vardır. Buna düğümün derecesi de denilmektedir. Düğümün yüksekliği o düğümüne gidilmek için kaç tane yoldan geçildiği bilgisidir. Bir ağacın yüksekliği en yüksek düğümü ile belirlenir. Ağaçta her bir düğümün tek bir üst düğümü (parent node) bulunmaktadır. Bir düğümün hiç alt düğümü yoksa yaprak (leaf) denir. Kök ve yapraklar dışındaki düğümlere de ara düğüm (internal nodes) denilir.

Bir ağaçta her bir düğümün en fazla m tane alt düğümü olması koşulu varsa bu ağaca m li ağaç (n ary tree) denir. Özel olarak bir düğümün en fazla iki alt düğümü varsa bu tür ağaçlara ikili ağaç (binary tree) denilir.

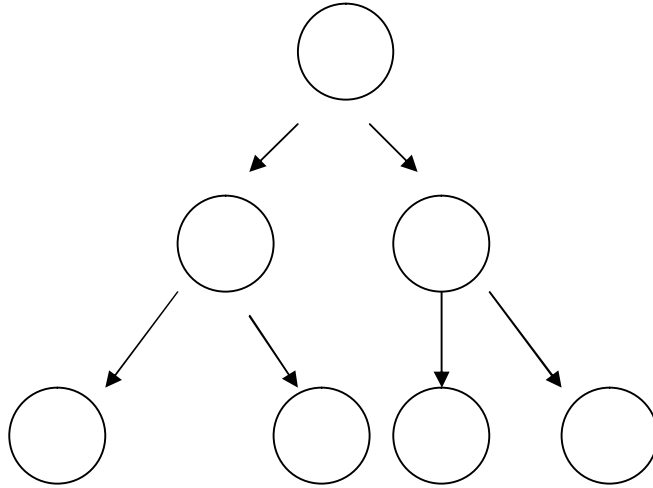
Eğer bir ağaç arama amaçlı kuruluyorsa bu tür ağaçlara arama ağaçları (search trees) denir. En yaygın kullanılan arama ağaçları ikili arama ağaçlarıdır (binary search trees). Bir ikili arama ağacını küçükleri sola büyükleri sola yerleştirerek oluşturabiliriz. Örneğin:



Eğer bir ikili ağaçta tüm yaprakların yüksekliği arasında belirli bir fark varsa (tipik olarak 1) bu tür ağaçlara dengelenmiş ikili ağaç türü denilir.



Eğer bir ikili ağaçta yapraklar dışındaki bütün düğümlerin tam olarak iki alt düğümü varsa bu tür ikili ağaçlara tam ikili ağaçlar (complete binary trees) denilmektedir.



Bir ikili arama ağacı dengeli ise tıpkı ikiye bölme yönteminde olduğu gibi en kötü başarılı arama karmaşıklığı yada başarısız arama karmaşıklığı $O(n) = \log_2 n$ dir.

İkili arama ağaçları dengesiz olarak kullanılabilir. Fakat dengesiz bir ikili ağacın en kötü durumu bağlı liste haline dönüşmesidir. Ağaca yeni düğüm eklendikçe dengenin bozulmasını engelleyen ve ağacı dengede tutan yöntemler vardır. En önemli iki yöntem kırmızı-siyah ağacı (red black tree) ve AVL ağacı (Adelson-Velski-Lendis tree) dir. Kırmızı-siyah ağacında denge faktörü 1 den büyük olabilmektedir. Fakat dengeye getirmek daha kolaydır. AVL ağacı ise daha dengelidir. Fakat dengeleme işlemi biraz daha karmaşıktır.

Bir ikili ağaç düğümler birer yapı olacak biçimde yollarda birer gösterici olacak biçimde bir yapı ile temsil edilebilir. İkili ağacın bir düğümü aşağıdaki gibi bir yapı ile temsil edilebilir:

```
typedef struct tagBNODE {
    DATATYPE val;
    struct tagBNODE *pLeft;
    struct tagBNODE *pRight;
} BNODE;
```

Kök düğümde bir göstericide tutulursa ikili arama ağacı oluşturulabilir.

15.7.1. Dengelenmemiş İkili Ağacın Oluşturulması

İkili ağacın handle alanı aşağıdaki gibi oluşturabilir:

```
typedef struct tagBTNODE {
    DATATYPE val;
    struct tagBTNODE *pLeft;
    struct tagBTNODE *pRight;
} BTNODE;

typedef struct tagBTREE {
    BTNODE *pRoot;
    size_t count;
} BTREE, *HBTREE;
```

Handle alanını oluşturan fonksiyon şöyle olabilir:

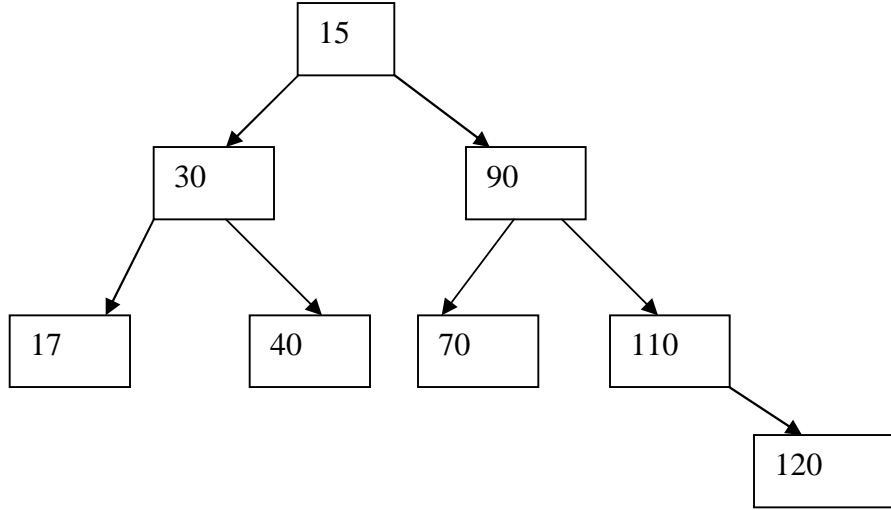
```
HBTREE CreateBTree(void)
{
    HBTREE hBTree;

    if ((hBTree = (HBTREE)malloc(sizeof(BTREE))) == NULL)
        return NULL;

    hBTree->pRoot = NULL;
    hBTree->count = 0;

    return hBTree;
}
```

İkili ağaca eleman eklemek için önce eklenecek yerin bilinmesi gerekir. Örneğin:



Burada örneğin 85 numaralı bir düğümü eklemek isteyelim. Yer bulma işlemi oldukça basittir. Kök düğümden başlanır, büyükse sağa küçükse sola doğru gidilir. NULL görünce durulur.

```
BTNODE *InsertItem(HBTREE hBTree, DATATYPE val)
{
    BTNODE *pNewNode, *pNode, *pPrevNode;

    pNewNode = (BTNODE *) malloc(sizeof(BTNODE));
    if (pNewNode == NULL)
        return NULL;

    pNewNode->pLeft = pNewNode->pRight = NULL;
    pNewNode->val = val;

    if (hBTree->pRoot == NULL) {
        hBTree->pRoot = pNewNode;
        return pNewNode;
    }

    pNode = hBTree->pRoot;
    while (pNode != NULL) {
        pPrevNode = pNode;
        pNode = val < pNode->val ? pNode->pLeft : pNode->pRight;
    }

    if (val < pPrevNode->val)
        pPrevNode->pLeft = pNewNode;
    else
        pPrevNode->pRight = pNewNode;
}
```

```
    return pNewNode;
}
```

İkili arama ağacı özellikle arama için kullanıldığına göre, aramada bir anahtar ve değerinin olması daha anlamlıdır. Bu nedenle aslında düğümün şöyle oluşturulması daha uygun olur.

```
typedef struct DATATYPE {
    KEYTYPE key;
    VALUETYPE val;
}
```

Fakat bu tasarımda VALUETYPE aynı zamanda anahtar değeri de içeriyorsa, bilgi çiftlemesi söz konusu olur. Eğer anahtarda DATATYPE içerisinde ise, bu durumda biz arama sırasında anahtar kısmı doldurulmuş bir DATATYPE verebiliriz. Yukarıda ki InsertItem fonksiyonunda DATATYPE bir yapı olamaz çünkü biz onu karşılaştırma operatörleri ile kullanmışız. Fakat bu örnek algoritmik yapının anlaşılması için verilmiştir.

Eleman arayan fonksiyon şöyle yazılabilir:

```
DATATYPE *FindItem(HBTREE hBTree, DATATYPE val)
{
    BTNODE *pNode;

    pNode = hBTree->pRoot;
    while (pNode != NULL) {
        if (val == pNode->val)
            return &pNode->val;
        pNode = val < pNode->val ? pNode->pLeft : pNode->pRight;
    }

    return NULL;
}
```

16. ÖZYİNELEMELİ ALGORİTMALAR VE KENDİ KENDİNİ ÇAĞIRAN FONKSİYONLAR

Sistem programlamada karşılaşılan bazı tipik algoritmalar özyinelemelidir. Özyinelemeli algoritmalar tipik olarak kendi kendini çağıran fonksiyonlar ile gerçekleştirilir. Aslında özyinelemeli algoritmalar suni bir stack kullanımı ile özyinelemeli olmayan fonksiyonlarla da gerçekleştirilebilir.

Sistem programlama da kullanılan tipik özyinelemeli algoritmalarından bazıları şunlardır:

- Ağaç dolaşma algoritmaları
- Permütasyon oluşturan algoritmalar
- Parsing algoritmaları
- Dizin dolaşma algoritmaları
- Diğerleri

Bir algoritma üzerinde çözüm için ilerlediğiniz zaman eğer başladığımız noktaya çok yakın bir durum ile karşılaşıyorsak bu algoritma özyinelemeli karaktere sahiptir. Tipik olarak kendi kendini çağıran fonksiyonlar ile gerçekleştirilir.

Bir fonksiyonun kendini çağırması ile başka bir fonksiyonu çağırması arasında işlevsel hiçbir farklılık yoktur. Parametre değişkenleri ve yerel değişkenler stack'te oluşturulduğu için her çağırmada onların yeni bir kopyası oluşturulmuş olur. Şüphesiz fonksiyonun hiç geri dönmeden sürekli kendini çağırması stack taşması oluşturarak programın çökmesine yol açar. Normal olarak fonksiyon kendi kendini çağırarak ilerlemeli fakat belirli bir noktadan sonra çıkış sürecine girmelidir.

16.1. Özyinelemeli Fonksiyon Örnekleri

1- Bir sayının faktöriyelini alan bir fonksiyonun özyinelemeli olarak yazılmasına hiç gerek yoktur. Tipik yazım şöyle olabilir:

```
long Factorial(int n)
{
    long i = 1;

    while (n > 1) {
        i *= n;
    }
}
```

```

        n--;
    }
    return i;
}

```

Özyinelemeli fonksiyon aşağıdaki gibi yazılabilir:

```

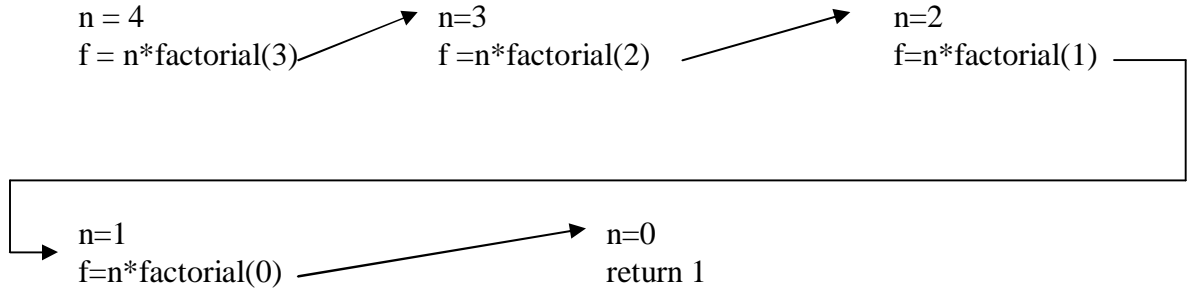
long Factorial(int n)
{
    long f;

    if (n == 0)
        return 1;

    f = n * factorial(n - 1);

    return f;
}

```



2- Özyinelemeli fonksiyonlar bir şeyi tersten yapmak için sıklıkla kullanılmaktadır. Yani fonksiyon kendini çağıra çağıra sona kadar gider, sonra çıkarken işlemini yapar. Örneğin bir yazıyı tersten yazan putsrev isimli fonksiyonun özyinelemeli yazılmasına gerek yoktur. Fakat şöyle yazılabilir:

```

void putsRev(const char *str)
{
    if (*str == '\0')
        return;

    putsRev(str + 1);
    putchar(*str);
}

```

3- Tek bağlı listeyi tersten dolaşmak için önce kendini çağıra çağıra sona kadar gideriz. Sonra çıkışta işlemlerimizi yaparız.

```
BOOL WalkListRev(NODE *pNode)
{
    if (pNode == NULL)
        return;

    WalkListRev(pNode->pNext);
    printf("%d\n", pNode->val);
}
```

4- Bir sayının bitlerini şöyle yazdırabiliriz:

```
void bitprint(unsigned n)
{
    if (n == 0)
        return;
    bitprint(n >> 1);
    putchar((n & 1) + '0');
}
```

5- En büyük elemanın bulunarak ilk elemanla ya da son elemanla yer değiştirilmesi biçiminde sıraya dizme yöntemine seçerek sıralama (selection sort) denilmektedir. Seçerek sıralama işlemi özyinelemeli bir fonksiyon ile de gerçekleştirilebilir:

```
void SSort(int *p, int size)
{
    int i, max, maxIndex;

    if (size == 1)
        return;

    max = p[0];
    maxIndex = 0;
    for (i = 1; i < size; ++i)
        if (max < p[i]) {
            max = p[i];
            maxIndex = i;
        }

    p[maxIndex] = p[size - 1];
    p[size - 1] = max;

    SSort(p, size - 1);
}
```

```
}
```

6- Bilgisayar sistemi yalnızca karakterleri yazmaya izin vermektedir. Sayı yazdıran fonksiyonlar aslında karakter yazdıran fonksiyonlar ile gerçekleştirilir. Yani aslında printf gibi fonksiyonlar putchar kullanılarak yazdırılmıştır. Örneğin 123 gibi bir sayıyı putchar kullanarak yazdırabilmemiz için önce sayıyı basamaklarına ayırmamız gerekir. Klasik yöntem bölmek, bölümünden elde edilen kalanı bulmaktır. Fakat bu durumda sayılar tersten yazılmaktadır.

```
while (n) {
    putchar(n % 10 + '0');
    n /= 10;
};
```

İlk akla gelecek yöntem bu karakterleri henüz bastırmayıp, char türden bir dizinin içerisinde toplamak ve diziyi strrev ile ters-düz etmektir.

```
char s[20];
int i;

while (n) {
    s[i++] = n % 10 + '0';
    n /= 10;
};
s[i] = '\0';
strrev(s);
```

Oysa bu işlem tipik olarak aşağıdaki gibi kendi kendini çağıran bir fonksiyon ile gerçekleştirilebilir:

```
void printd(int n)
{
    if (n < 0) {
        putchar('-');
        n = -n;
    }

    if (n / 10)
        printd(n/10);

    putchar(n % 10 + '0');
}
```

Bu fonksiyonu herhangi bir tabana göre çevirme yapacak şekilde genelleştirebiliriz.

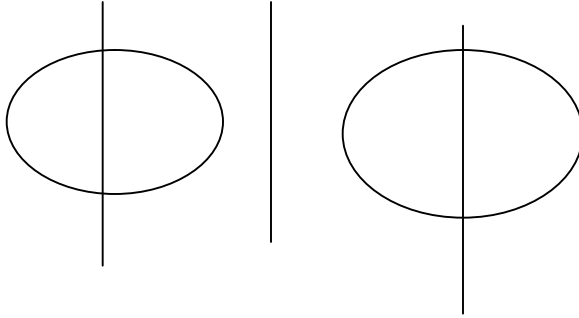
```

void printnum(unsigned n, int base)
{
    if (n / base)
        printnum(n / base, base);

    putchar(n % base > 9 ? n % base - 10 + 'A' : n % base + '0');
}

```

7- En iyi sıralama algoritması olarak bilinen Quick-Sort algoritması $n \log n$ karmaşıklığa sabittir ve özyinelemeli bir karaktere sahiptir.



Algoritmanın dayandığı fikir şöyle özetlenebilir. Bir pivot eleman seçilir (pivot elemanın iyi seçilmesi olumludur fakat dizinin dağılımı bilinmiyorsa rasgele bir elemanı pivot seçebiliriz. Örneğin bu ilk eleman olabilir). Sonra dizi baştan ve sondan hareketle sol taraftan pivottan küçük değerler sağ taraf pivottan büyük değerler olacak biçime getirilir. Sonra pivottan itibaren, pivotun iki tarafı ile aynı işlemler öz yinelemeli olarak yinelenir.

Bir sayıdan küçükleri solda büyükleri sağda toplayan basit bir fonksiyon şöyle yazılabilir:

```

int Partition(int *pi, size_t size, int val)
{
    size_t left = 0, right = size - 1;
    int temp;

    while (left < right) {
        while (pi[left] < val && left < right)
            ++left;

        while (pi[right] >= val && right > left)
            --right;

        temp = pi[left];
        pi[left] = pi[right];
        pi[right] = temp;
    }

    return left;
}

```

```
}
```

Partition işlemi sonucunda bulunan indekste pivot değerinin olması gerekir. Bundan sonra ikiye bölmeler yapılabilir. Partition fonksiyonu bir dizinin başlangıç adresini alıp left ve right ile belirtilen iki çubuk arasındaki elemanları parçalara ayıracak biçimde yazılırsa ve pivot yer değiştirmesini yaptıktan sonra pivot elemanın bulunduğu indekse geri dönerse bu durumda quick sort algoritması aşağıdaki gibi basit bir şekle dönüşür:

```
int Partition(int *a, size_t left, size_t right)
{
    int pivot = a[left];
    int i = left;
    int k = right + 1;
    int temp;

    for (;;) {
        do
            ++i;
        while (i < right && a[i] <= pivot);

        do
            --k;
        while (k > left && a[k] >= pivot);

        if (i >= k)
            break;

        temp = a[i];
        a[i] = a[k];
        a[k] = temp;
    }

    a[left] = a[k];
    a[k] = pivot;

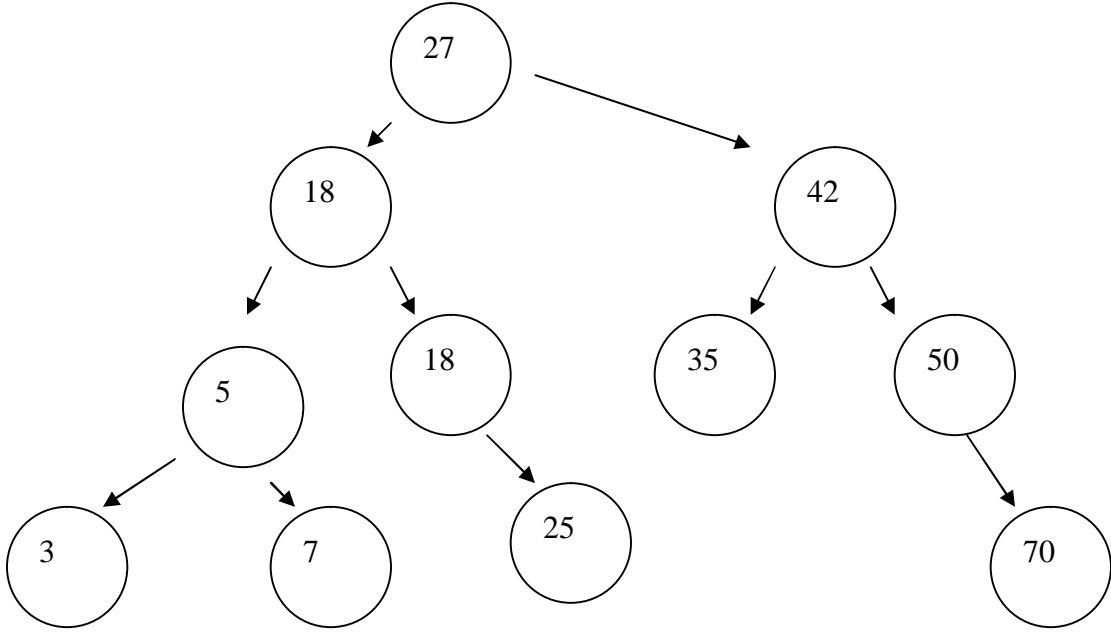
    return k;
}
```

```
void QSort(int *a, size_t left, size_t right)
{
    size_t pivotIndex;

    if (left < right) {
        pivotIndex = Partition(a, left, right);
        QSort(a, left, pivotIndex - 1);
        QSort(a, pivotIndex + 1, right);
    }
}
```

Normal olarak partition işlemi ile bölme beraber yapılabilir. Ayrıca bir sarma fonksiyon ile parametreleri düzeltebiliriz.

8- İkili arama ağacını dolaşmak için tipik olarak özyinelemeli fonksiyon kullanılabilir. Örneğin:



İkili ağaç 3 biçimde dolaşılabilir. Dolaşım ortadaki düğümün sol ve sağ düğümlere göre hangi sırada dolaşılacağına göre isimlendirilmiştir.

1. Inorder Dolaşım: Bu dolaşımında sol, üst ve sağ dolaşımı uygulanır. İkili ağaç böyle dolaşılırsa sıralı olarak dolaşılmış olur.
2. Preorder Dolaşım: Bu dolaşımında önce üst düğüm sonra sol ve sağ düğümler dolaşılır. Bu dolaşım az kullanılmaktadır.
3. Postorder Dolaşım: Bu dolaşımında önce sol ve sağ alt düğümler, sonra üst düğüm dolaşılır.

Inorder dolaşan fonksiyon şöyle yazılabilir:

```
void WalkTree(BTNODE *pNode)
{
    if (pNode == NULL)
        return;
```

```

WalkTree(pNode->pLeft);
printf("%d\n", pNode->val);
WalkTree(pNode->pRight);
}

```

Bu fonksiyon kök düğümün adresi ile çağrılmalıdır.

İkili ağacın tüm düğümlerinin free hale getirilmesi için ağaç postorder biçimde dolaşılmalıdır. Çünkü önce alt düğümlerin sonra üst düğümlerin free hale getirilmesi gerekmektedir. O halde ikili ağacı boşaltmak için şöyle bir fonksiyon yazılabilir:

```

void FreeTree(BTNODE *pNode)
{
    if (pNode == NULL)
        return;

    FreeTree(pNode->pLeft);
    FreeTree(pNode->pRight);

    free(pNode);
}

```

9- Dizin ağacının dolaşılması tipik olarak özyinelemeli bir algoritmadır. Her işletim sisteminin bir dizin içerisindeki dosyaları elde etmek için sunduğu API fonksiyonları vardır. Microsoft Windows derleyicileri bu API fonksiyonlarını sarmalayan `_findfirst`, `_findnext` ve `_findclose` fonksiyonlarını bulundurmaktadır. Bu işlemin çok benzeri UNIX sistemlerinde `opendir`, `readdir` ve `closedir` fonksiyonları ile yapılmaktadır. Bu fonksiyonlar bir dizin içerisindeki dosyaları elde ederken onların normal bir dosya mı yoksa bir dizin dosyası mı olduğunu da vermektedir. (Sistem programlamada dosya kavramı normal dosyalar ve dizinleri de içerecek şekilde kullanılmaktadır.)

`_findfirst` fonksiyonunun prototipi şöyledir.

```
intptr_t _findfirst(const char *file, struct _finddata_t *fileinfo);
```

Fonksiyonun birinci parametresi bir yol ifadesidir. Bu yol ifadesi joker karakteri içerebilir.

Örneğin:

```
“C:\\windows\\*.tmp”
```

Fonksiyonun ikinci parametresi dosya bilgilerinin yerleştirileceği `_finddata_t` türünden bir yapının adresini alır. Fonksiyon koşulu sağlayan ilk dosyanın bilgilerini elde eder ve bir handle değeri ile geri döner. Fonksiyon başarısız ise -1 değerine geri dönmektedir. Bu handle değeri

`_findnext` fonksiyonuna geçirilerek koşulu sağlayan diğer dosyalar bulunur. Nihayet `_findclose` fonksiyonuyla handle alanı kapatılır. `_findnext` fonksiyonunun prototipi şöyledir:

```
int _findnext(intptr_t _FindHandle, struct _finddata_t *_FindData);
```

Fonksiyon başarılı ise 0 değerine başarısız ise -1 değerine geri döner. `_findclose` ise şöyledir:

```
int _findclose(intptr_t _FindHandle);
```

Fonksiyon başarılı ise 0 başarısız ise -1 değerine döner. Tüm bu fonksiyonların prototipi `io.h` içerisinde. `_finddata_t` yapısı ise şöyledir.

```
struct _finddata_t {
    unsigned attrib;
    __time32_t time_create;
    __time32_t time_access;
    __time32_t time_write;
    __fsize32_t size;
    char name[260];
}
```

Bir dizindeki dosyaların listesini almak için şöyle bir kod yazılabilir.

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

int main(void)
{
    intptr_t handle;
    struct _finddata_t f;
    int result;

    handle = _findfirst("c:\\windows\\*.\"", &f);
    result = handle == -1;

    while (!result) {
        printf("%s\n", f.name);
        result = _findnext(handle, &f);
    }

    _findclose(handle);

    return 0;
}
```

Elde edilen dosyanın bir dizin dosyası olup olmadığı `_finddata_t` yapısının `attrib` elemanı ile belirlenebilir. Yapının `attrib` elemanı bit bit kodlanmıştır. Yani her bit bir özelliğin olup olmadığını belirtmektedir.

Arch	dir		system	hidden	readonly
------	-----	--	--------	--------	----------

`Attrib` elemanı, bütün bitleri 0 olan, ilgili biti 1 olan bir sayı ile `and` çekilirse ilgili bitin set edilip edilmediği, dolayısıyla dosyanın o özelliğe sahip olup olmadığı belirlenebilir. Bunun için `io.h` içerisinde aşağıdaki sembolik sabitler bulundurulmuştur.

```
#define _A_NORMAL 0x00
#define _A_RDONLY 0x01
#define _A_HIDDEN 0x02
#define _A_SYSTEM 0x04
#define _A_SUBDIR 0x10
#define _A_ARCH 0x20
```

Bu durumda bir dizindeki alt dizinlerin yanına `dir` biçiminde işaret koymak istersek bunu şöyle yapabiliriz:

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

int main(void)
{
    intptr_t handle;
    struct _finddata_t f;
    int result;

    handle = _findfirst("*.*", &f);
    result = handle == -1;

    while (!result) {
        printf("%-30s %s\n", f.name, (f.attrib & _A_SUBDIR) ? "<DIR>" : "");
        result = _findnext(handle, &f);
    }

    _findclose(handle);

    return 0;
}
```

Sınıf Çalışması:

Bir dizinin dizindeki dosyalarını alınız, dosyaların özelliklerini listenin yanına “Read Only + Archive” biçiminde yazdırınız.

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

void PutAttr(unsigned int attr)
{
    char *attrText[] = {"Read Only", "Hidden", "System", "", "Dir",
"Archive"};
    int i;

    for (i = 0; attr ; ++i) {
        if (attr & 1) {
            printf("%s", attrText[i]);
            if (attr >> 1)
                printf("+");
        }
        attr >>= 1;
    }
    putchar('\n');
}

int main(void)
{
    intptr_t handle;
    struct _finddata_t f;
    int result;

    handle = _findfirst("*. *", &f);
    result = handle == -1;

    while (!result) {
        printf("%-30s", f.name);
        PutAttr(f.attrib);
        result = _findnext(handle, &f);
    }

    _findclose(handle);

    return 0;
}
```

Her alt dizinde ilk iki dosya “.” ve “..” isimli dizin dosyalarıdır.

Bir dizinden başlayarak alt dizinler ile birlikte tüm ağacı dolaşmak için özyinelemeli bir algoritma gerekir. Algoritma oldukça basittir. Dizindeki dosyalar yazdırılarak ilerlenir. Bir dizin ile karşılaşıldığında o dizine geçilerek fonksiyonun kendini çağırması sağlanır.

```

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

#define MAX_PATH          260

void WalkDirTree(const char *dir)
{
    intptr_t handle;
    struct _finddata_t f;
    int result;
    char path[MAX_PATH], subDir[MAX_PATH];

    sprintf(path, "%s\\*.*", dir);

    handle = _findfirst(path, &f);
    result = handle == -1;

    while (!result) {
        if (!strcmp(f.name, ".") || !strcmp(f.name, "..")) {
            result = _findnext(handle, &f);
            continue;
        }
        if (f.attrib & _A_SUBDIR) {
            sprintf(subDir, "%s\\%s", dir, f.name);
            WalkDirTree(subDir);
        }
        printf("%s\n", f.name);

        result = _findnext(handle, &f);
    }

    _findclose(handle);
}

int main(void)
{
    WalkDirTree("e:\\sysprog-cupa");

    return 0;
}

```

Fonksiyonu biraz daha kullanışlı hale getirebiliriz. Bunun için bir sarma fonksiyon kullanabiliriz. Fonksiyonun bize tüm yol ifadesini yada her ikisini birden vermesi de daha uygun olabilir.

```

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

#define MAX_PATH          260

```

```

#define FALSE          0
#define TRUE           1

typedef int BOOL;

void WalkDirTree(const char *dir, void (*Proc)(const char *path, const
struct _finddata_t *file))
{
    intptr_t handle;
    struct _finddata_t f;
    int result;
    char path[MAX_PATH], newFile[MAX_PATH];

    sprintf(path, "%s\\*.*", dir);

    handle = _findfirst(path, &f);
    result = handle == -1;

    while (!result) {
        if (!strcmp(f.name, ".") || !strcmp(f.name, "..")) {
            result = _findnext(handle, &f);
            continue;
        }
        sprintf(newFile, "%s\\%s", dir, f.name);
        Proc(newFile, &f);
        if (f.attrib & _A_SUBDIR)
            WalkDirTree(newFile, Proc);

        result = _findnext(handle, &f);
    }

    _findclose(handle);
}

void Disp(const char *path, const struct _finddata_t *f)
{
    printf("%s\n", path);
}

int main(void)
{
    WalkDirTree("e:\\sysprog-cupa", Disp);

    return 0;
}

```

Bazen bir dosya bulunduğunda artık daha fazla işlemin devam etmesi istenmez. Bu durum aşağıdaki gibi sağlanabilir:

```

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

```

```

#define MAX_PATH          260
#define FALSE             0
#define TRUE              1
typedef int BOOL;

BOOL WalkDirTree(const char *dir, BOOL (*Proc)
                (const char *path, const struct
                _finddata_t *file))
{
    intptr_t handle;
    struct _finddata_t f;
    int result;
    char path[MAX_PATH], newFile[MAX_PATH];

    sprintf(path, "%s\\*.*", dir);

    handle = _findfirst(path, &f);
    result = handle == -1;

    while (!result) {
        if (!strcmp(f.name, ".") || !strcmp(f.name, "..")) {
            result = _findnext(handle, &f);
            continue;
        }
        sprintf(newFile, "%s\\%s", dir, f.name);
        if (!Proc(newFile, &f)) {
            _findclose(handle);
            return FALSE;
        }
        if (f.attrib & _A_SUBDIR)
            if (!WalkDirTree(newFile, Proc)) {
                _findclose(handle);
                return FALSE;
            }

        result = _findnext(handle, &f);
    }

    _findclose(handle);

    return TRUE;
}

BOOL Disp(const char *path, const struct _finddata_t *f)
{
    printf("%s\n", path);

    return TRUE;
}

int main(void)
{
    WalkDirTree("e:\\sysprog-cupa", Disp);

    return 0;
}

```

Sınıf Çalışması:

Bir kök dizinden itibaren dizin ağacını bir ağaç yapısı biçiminde bellekte oluşturunuz. Bir düğüm bir dosyaya ya da bir dizine ilişkin olabilir. Eğer düğüm dizine ilişkinse bu düğümün alt düğümleri bu dizinin içindeki dosyaları belirtmektedir. Yani bir dizin içerisindeki dosyaları bağlı liste biçiminde düşünebiliriz. Bağlı listenin elemanı bir dizin ise buda başka bir bağlı listenin head göstericisini tutmaktadır.

Açıklama: Bu ağaç yapısının bir düğümü şöyle oluşturulabilir:

```
typedef struct tagNODE {
    struct _finddata_t f;
    struct NODE *pNext;
    struct NODE *pHead;
} NODE;
```

Burada normal dosyalar için pHead göstericisinin bulundurulması gereksiz görülebilir. Gerçi uygulamada bu kaybın ciddi bir önemi yoktur. Fakat istenirse iki tür düğüm bulundurulabilir.

```
typedef struct tagFILE_NODE {
    struct _finddata_t f;
    struct tagFILE_NODE *pNext;
} FILE_NODE;

typedef struct tagDIR_NODE {
    struct _finddata_t f;
    FILE_NODE *pNext;
    FILE_NODE *pHead;
} DIR_NODE;
```

Böylece liste elemanları FILE_NODE biçiminde tutulabilir. Duruma göre DIR_NODE * türüne dönüştürülebilir. Aynı işlem C++ 'ta türetme yolu ile yapılabilir.

```
struct FileNode {
    struct _finddata_t m_f;
    FileNode *m_pNext;
};

struct DirNode : public FileNode {
    FileNode *m_pHead;
};
```

Bağlı liste için standart kütüphanede bulunan list sınıfı kullanılacaksa, daha değişik bir düzenlemeye gidilebilir.

```
struct FileNode {
    struct _finddata_t m_f;
};

struct DirNode : public FileNode {
    std::list<FileNode *> m_head;
};
```

Şimdi ağacı ilk biçimde kurduğumuzu düşünelim.

```
typedef struct tagNODE {
    struct _finddata_t f;
    struct NODE *pNext;
    struct NODE *pHead;
} NODE;
```

Böyle bir ağacı dolaşma işlemi yine özyinelemeli olarak yapılmak zorundadır.

```
void WalkListTree(NODE *pDir)
{
    NODE *pNode = pDir;

    while (pNode != NULL) {
        /***/
        if (f.attrib & _A_SUBDIR)
            WalkDirTree(pNode->pHead);
        pNode = pNode->pNext;
    }
}
```

Başka bir örnek: sample.exe dosyasının bulunması

```
BOOL Disp(const char *path, const struct _finddata_t *f)
{
    char *pStr;

    pStr = strrchr(path, '\\');
    if (pStr)
        if (!strcmp(pStr + 1, "sample.exe")) {
            printf("%s\n", path);
            return FALSE;
        }

    return TRUE;
}
```


C# programı ile yukarıdaki yapmak istediğimizi basitçe yapabiliriz:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.IO;

namespace SampleCSharp
{
    class Program
    {
        static void Main(string[] args)
        {
            string[] files;

            files = Directory.GetFiles("e:\\sysprog-cupa", "*.*",
SearchOption.AllDirectories);

            foreach (string s in files)
                Console.WriteLine(s);
        }
    }
}
```

Anahtar Notlar:

Her işletim sisteminde yol ifadesinin maksimum karakter sayısı önceden belirlenmiş durumdadır. Örneğin LINUX sistemlerinde 256, Windows sistemlerinde 260 karakterdir. Bu durumda biz yukarıdaki WalkDirTree fonksiyonuna 260'dan daha büyük bir yol ifadesi geçemeyiz. WalkDirTree fonksiyonu prosesin default çalışma dizinini değiştirecek biçimde yeniden yazılabilir. Böylece bu sınır büyük ölçüde kaldırılmış olur.

17. PROSESLER ARASINDA ALTLIK ÜSTLÜK İLİŞKİSİ

UNIX/LINUX sistemlerinde katı bir biçimde Windows sistemlerinde gevşek bir biçimde prosesler arasında üstlük altlık ilişkisi vardır. Bir proses yani program başka bir procesi yani programı çalıştırdığında çalıştıran proses üst proses, çalıştırılan proses alt proses olur. Örneğin biz komut satırından bir programı çalıştırdığımızda, (komut satırı cmd.exe isimli bir programdır)

çalıştırılan prosesin üst prosesi cmd.exe olacaktır. Solomon tarafından yazılmış olan proses explorer isimli program Windows'ta çalışan prosesler üzerinde çeşitli faydalı bilgileri vermektedir.

Gerek unix sistemlerinde gerekse windows sistemlerinde olsun, üst ve alt prosesler arasında bazı aktarımlar yapılmaktadır. Daha teknik bir anlatımla üst prosesin proses tablosundaki bazı bilgiler yaratılan prosese aktarılmaktadır. Bu tipik bilgiler şunlar olabilir:

- Unix sistemlerinde user id, group id, effective user id, effective group id ve Windows sistemlerinde ki güvenlik bilgileri aktarılır.
- Prosesin default çalışma dizini
- Çevre değişkenleri
- Diğer önemli bilgiler

17.1. Proseslerin Çevre Değişkenleri (Environment Variables)

Pek çok işletim sisteminde çevre değişkeni kavramı vardır. Çevre değişkenleri “değişken = değer” çiftlerinden oluşur ve genellikle alt prosese doğrudan geçirilirler. Çevre değişkenleri UNIX/LINUX sistemlerinde shell üzerinden

```
export değişken = değer
```

işlemi ile oluşturulabilir. Windows komut satırında

```
set değişken = değer
```

işlemi ile oluşturulabilir. Bu biçimde shell üzerinden çevre değişkenleri oluşturulduğunda bu çevre değişkenleri shell'in çevre değişkenleri olur. Dolayısıyla shell'den çalıştırılacak programlara doğrudan aktarılır.

Unix sistemlerinde shell üzerinde bir çevre değişkeni \$ ile kullanılırsa çevre değişkenine karşı gelen değer elde edilir. Örneğin:

```
export fiyat = 1300
```

```
echo $fiyat
```

1300 yazılmasına neden olur.

Windows'un komut satırında çevre değişkeni iki yüzde arasına getirilirse aynı etki oluşturulabilir.

Çevre değişkenlerini set etmek ve kullanmak için iki C fonksiyonu kullanılır.

```
char *getenv(const char *name);
```

Prototipi `stdlib.h` başlık dosyası içinde bulunur. Fonksiyon parametre olarak çevre değişkeninin ismini alır. Geri dönüş değeri olarak çevre değişkeninin değerini verir. Fonksiyon değişkenin değerini statik bir alana yazıp onun adresi ile döner. Eğer ilgili çevre değişkeni yoksa fonksiyon `NULL` değerine geri döner.

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

int main(void)
{
    char *pStr;

    if ((pStr = getenv("fiyat")) == NULL) {
        fprintf(stderr, "Cannot get environment variable\n");
        exit(EXIT_FAILURE);
    }

    puts(pStr);

    return 0;
}
```

Prosesin çevre değişkenleri alt prosese geçirilebildiğine göre komut satırı da bir proses olduğuna göre biz komut satırında bir çevre değişkeni oluşturup komut satırından bir program çalıştırsak çalıştırılan programa o çevre değişkeni geçer.

Windows'ta komut satırında `set` komutu unix/linux sistemlerinde `env` komutu shell in tüm çevre değişkeni listesini dökmetedir.

Çevre değişkenlerini programlama yolu ile oluşturmak için `setenv` fonksiyonu kullanılır. `setenv` fonksiyonunun bir benzeri `putenv` isimli fonksiyondur.

`setenv` fonksiyonu POSIX sistemlerinde bu isimle bulunur. Microsoft sistemlerinde bu fonksiyon `putenv` biçimindedir.

```
int _putenv(const char *envstring);
```

Fonksiyonun parametresi değişken = değer biçimindedir. Fonksiyon başarılı ise 0 değerine, başarısız ise -1 değerine geri döner. Eğer ilgili çevre değişkeni varsa yeni değer üzerine yazılır. Örneğin:

```
int main()
{
    _putenv("fiyat = 1300");
    return 0;
}
```

Şüphesiz alt prosesteki değişiklik üst prosesteki değişikliği etkilememektedir. yani örneğin biz komut satırında

n bir program çalıştırsak bu program ile de bir çevre değişkenini set etsek komut satırı bundan etkilenmez.

Peki, komut satırına yani shell'e otomatik bir biçimde bir çevre değişkenini nasıl ekleyebiliriz?

Bunun için Windows'ta kontrol panel/sistem menüsü kullanılır. Burada belirli bir kullanıcının shell'i için ekleme yapılabileceği gibi tüm sistemi etkileyecek ekleme de yapılabilir.

Benzer biçimde Linux sistemlerinde de login işlemi sırasında bazı dosyalar gözden geçirilmektedir. Örneğin /etc/profile dosyası hangi shell kullanılırsa kullanılsın dikkate alınır. Bu dosyanın içerisinde çevre değişkenlerini oluşturabiliriz. Yada örneğin yaygın kullanılan bash shell i için.

Çevre değişkenleri bazı global belirlemeleri yapmak için sıklıkla kullanılmaktadır. Örneğin bir C derleyicisi include dosyalarının default dizinini elde etmek için INCLUDE isimli çevre değişkenine başvurabilir. yada örneğin biz bir data dosyasını DATAFILE isimli bir çevre değişkeninin belirttiği bir dizinde arayabiliriz. Böylece kullanıcı bu data dosyasını başka bir dizine yerleştirecekse DATAFILE çevre değişkenini ayarlamak durumundadır.

Anahtar Notlar:

Windows sistemlerinde çevre değişkenlerinin büyük harf küçük harf duyarlılığı yoktur fakat Linux sistemlerinde vardır.

PATH isimli çevre değişkeni hem Windows hem de Linux sistemlerinde işletim sisteminin kendisi tarafından da kullanılmaktadır. Windows 'ta PATH çevre değişkeninin değeri noktalı virgüllerle ayrılmış dizinlerden oluşur. Örneğin:

```
set PATH = c:/Windows; c:\test
```

Linux sistemlerinde dizin ayırmak için iki nokta üst üste karakteri kullanılmaktadır. Örneğin:

```
export PATH = /bin: /wr/bin
```

Örneğin Windows komut satırında biz PATH çevre değişkenine aşağıdaki gibi bir komutla ekleme yapabiliriz:

```
c:\cs -cupa > set PATH = %PATH% ; c:\test
```

Peki, PATH çevre değişkeni ne işe yarar?

PATH çevre değişkeni işletim sisteminin program çağıran sistem fonksiyonları tarafından doğrudan bakılmaktadır. Windows'ta CreateProcess API fonksiyonu

CreateProcess API fonksiyonu eğer çalıştırılacak dosya doğrudan dosya ismi biçiminde belirtilmişse PATH çevre değişkenine bakmaktadır. Dosyayı sırasıyla PATH çevre değişkeninde ; ile ayrılmış bölgede arar.

Anahtar Notlar:

Yukarıda da belirtildiği gibi CreateProcess eğer dosya ismi bir dizin içerecek biçimde belirtilmişse PATH çevre değişkenine bakmaz. Yani örneğin biz komut satırından bir programı study/test biçiminde çalıştırmak istersek.

Fakat program yalnızca test denilerek bakılırsa PATH çevre değişkenine bakılmaktadır. CreateProcess her zaman önce bulunulan dizine sonra PATH ile belirtilen dizine bakmaktadır.

Linux sistemlerinde program çalıştırmakta kullanılan exec fonksiyonlarının p'li versiyonları (execvp, execlp) eğer dosya isminde bölü karakteri yoksa PATH çevre değişkenine bakar. Bu fonksiyonlar bu durumda prosesin çalışma dizinine bakmazlar.

Şüphesiz program çalıştıran bu sistem fonksiyonları fonksiyon hangi proses tarafından çalıştırılmışsa o prosesin PATH çevre değişkenine bakmaktadır.

17.2. Prosesin Default Çalışma Dizini

Bir dosyanın hangi dizinde olduğunu belirten yazıya yol ifadesi (path name) denir. Pek çok sistem fonksiyonu dosya ile ilgili işlem yaparken bir yol ifadesi istemektedir. Dizin geçişleri dos ve Windows sistemlerinde \ ile Linux sistemlerinde / ile yapılmaktadır. Ayrıca Linux sistemlerinde sürücü kavramı yoktur.

Linux sistemlerinde mount etme kavramı vardır. Bir dosya sistemi bir dizine mount edilir (monte edilir) böylece o dosya sisteminin kök dizini o dizin olur. Dos ve Windows sistemleri bunun yerine sürücü kullanmaktadır. Her farklı dosya sistemi bir sürücü biçimindedir.

Pek çok işletim sisteminde her prosesin bir default çalışma dizini (current working directory) vardır. Genellikle kullanıcılar kendilerinin sanki bir dizini içerisinde durduklarını sanırlar. Aslında böyle bir durum söz konusu değildir. Komut satırında görülen prompt aslında o komut satırı programının çalışma dizinidir. Bir prosesin başka bir prosesi çalıştırması durumunda prosesin default çalışma dizini alt prosese geçirilmektedir.

Bir prosesin çalışma dizini Windows sistemlerinde `_chdir`, unix/Linux sistemlerinde `chdir` fonksiyonlarıyla değiştirilir.

```
int _chdir(const char *dirname);
```

Fonksiyon başarılı ise 0 değerine, başarısız ise -1 değerine geri döner. Fonksiyonun Windows sistemlerindeki prototipi `direct.h`, unix/Linux sistemlerindeki prototipi `unistd.h` içerisinde yer almaktadır.

Prosesin çalışma dizinini elde etmek için Windows sistemlerinde `_getcwd`, unix/Linux sistemlerinde `getcwd` kullanılır.

```
char * _getcwd(char *buffer, int maxlen);
```

Fonksiyonun birinci parametresi geçerli dizinin yerleştirileceği diziyi ikinci parametresi bu dizinin uzunluğunu belirtir. Fonksiyon birinci parametresine yazılan adresin aynıysa ile geri döner.

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

#define MAX_PATH 260
#define MAX_CMD_LINE 4096

int main(void)
{
    char cwd[MAX_PATH];
    char cmdLine[MAX_CMD_LINE];

    for (;;) {
        _getcwd(cwd, MAX_PATH);
        printf("%s>", cwd);
        gets(cmdLine);
    }
}
```

```
    return 0;  
}
```

Prosesin default çalışma dizini proses tablosunda saklanmaktadır.

Bir PATH ifadesini mutlak (absolute) ve görelî (relative) olmak üzere ikiye ayırabiliriz. Eğer bir yol ifadesindeki sürücü ismi hariç ilk karakter Windows ta \, unix/Linux ta / ise bu tür yol ifadelerine mutlak yol ifadeleri denilir. Örneğin:

```
/root/sslsl  
C:\windows\system32
```

Mutlak yol ifadeleri kök dizinden itibaren yer belirtir. Eğer ilk karakter / veya \ değilse bu tür yol ifadelerine görelî yol ifadeleri denilir. Örneğin:

```
a\b\c.dat  
x/y/z.dat  
test.dat
```

Prosesin default çalışma dizini görelî yol ifadeleri için başlangıç yerini belirtir. Bunun dışında kullanılmaktadır. Örneğin:

```
f = fopen("test.dat", "r");
```

Burada test.dat dosyası görelî bir biçimde verilmiştir. O halde prosesin çalışma dizininde aranacaktır.

Prosesin çalışma dizini Windows sistemlerinde CreateProcess API fonksiyonu ile proses yaratılırken belirlenmektedir. Bu fonksiyonda duruma göre programcı default çalışma dizininin üst prosesden alınmasını isteyebilir yada bunu kendisi set edebilir. Örneğin Visual Studio IDE si ctrl+f5 ile programı çalıştırırken, çalıştırılacak programın çalışma dizinini proje dizini olarak ayarlamaktadır. Unix/Linux sistemlerinde prosesin çalışma dizini fork işlemi sırasında doğrudan üst prosesden alınmaktadır. Tabi programcı exec uygulamadan önce chdir fonksiyonuyla bunu değiştirebilir.

18. PROSESLER ARASI HABERLEŞME

Modern işletim sistemlerinin çoğunda proseslerin bellek alanı izole edilmiştir. Bir proses başka bir prosesin bellek alanına erişemez. Bir prosesin başka bir prosese herhangi bir biçimde

bilgi gönderebilmesi için işletim sisteminde değişebilecek çeşitli yöntemler uygulanmaktadır. Bu konuya genel olarak prosesler arası haberleşme (interprocess communication – IPC) denilmektedir.

Prosesler arası haberleşme aynı makinedeki proseslerin haberleşmesi ve ağdaki herhangi iki makinede çalışan herhangi iki prosesin haberleşmesi biçiminde ikiye ayrılmaktadır. Şüphesiz tek bir makinede bir ağ kabul edilmektedir ve ağ haberleşmeleri tek bir makine üzerinde yapılabilir.

Yerel makine üzerindeki prosesin haberleşmesine yönelik farklı sistemlerde pek çok mekanizma vardır. Bazı mekanizmalar pek çok işletim sisteminde benzer biçimde kullanılır. Bu mekanizmalardan bazıları şunlardır:

- Boru (pipe) haberleşmesi
- Paylaşılan bellek alanları (shared memory)
- Mesaj sistemleri

Bunları dışında işletim sistemlerine özgü yerel makine haberleşmeleri vardır.

Ağ üzerindeki prosesler arası haberleşmesi bir protokol gerektirmektedir. Çünkü bir ağ farklı donanımlara özgü farklı işletim sistemlerinin bulunduğu çeşitli makineler bağlanabilir. Bunların ortak bir dilden konuşabilmesi için ortak bazı belirlemelere gereksinim vardır. Bu belirlemelere protokol denilmektedir. Yine de bazı ağ haberleşmeleri belirli bir sisteme özgü olabilir ve belirli bir firmanın protokollerini kullanıyor olabilir. Örneğin Microsoft firmasının Microsoft network, Novell firmasının IPX/SPX, Apple firmasının AppleTalk diye isimlendirilen ve ağ haberleşmesinde kullanılan protokolleridir. Fakat şüphesiz 1983 yılında internetinde kullanmaya başladığı TCP/IP protokol ailesi, en çok kullanılan protokol ailesidir.

Bu bölümde ağ üzerindeki herhangi iki makinede bulunan proseslerin haberleşmesine yönelik IP ailesinin temel kullanımı ele alınacaktır.

18.1. Ağ Kavramı ve Tarihsel Gelişimi

1960'lı yıllara kadar bilgisayarları birbirine bağlama fikri üzerinde durulmamıştır. Fakat 60'lı yıllarla birlikte ağ oluşturma kavramı üzerinde uygulama faaliyetleri başlamıştır. İlk ciddi ağ

oluřturma faaliyeti Amerikan Savunma Bakanlıđı'na bađlı DARPA diye bilinen alıřma grubu tarafından bařlatılan ARPANET projesidir. Bu proje potansiyel bir nkleer savař durumu iin yrrlđe sokulmuřtur ve ilk ciddi ađ bađlantısı bu proje kapsamında gerekleřtirilmiřtir. 1968 yılındaki bu ilk ciddi denemeden sonra Amerika'da ARPA ađına zamanla niversiteler, devlet kurumları ve ticari kurumlar katılmıřtır. 70'li yıllarda bu ađ Avrupa'ya yayılmıřtır. ARPA ađı ilk zamanlar NCP (network control protocol)denilen bir protokol kullanıyordu. Sonra 1983 yılında IP protokol ailesine geilmiřtir. 90'lı yıllarla birlikte kiřisel kullanıcılarda bu ađa bađlanmaya bařlamıřtır ve bu ađa "The Internet" denilmektedir. 80'li yıllarda kiřisel bilgisayarlarında geliřmesi ile kurumlar ve řirketler kendi kiřisel bilgisayarlarını da birbirine bađlamaya bařlamıřtır. Genellikle bir binanın dıřına ıkmayan bađlantılar ieren bu ađlara yerel ađlar (local area network) denilmektedir.

Yerel ađlar yaygınlařmaya bařlayınca yerel ađları birbirine bađlama teması da gndeme gelmiřtir. Yerel ađlarda kullanılan protokoller farklı olabilir. Fakat yerel ađlar birbirine bađlanacaksa ađların birbirlerine bađlanması konusunda belirlemelere sahip olan protokoller kullanılmalıdır. Internet szcđ ađlardan oluřturulan ađ anlamına gelir ve bu anlamda pek ok řirket, kurum ya da řahıs internet oluřturabilir. Global dzeyde, herkes tarafından bilinen dev ađa da, halk arasında internet denilmektedir. Yani internet bir cins isim olmasına rađmen zel isim olarak da kullanılmaktadır.

IP (Internet Protocol) protokol ailesi hem yerel ađlarda hem de yerel ađların birbirine bađlanması ile elde edilen geniř ađlarda kullanılmak zere tasarlanmış geniř bir ailedir. IP protokol ailesi herkese aık (public) bir protokoldr (ticari deđildir).

Ađ oluřturmanın ilk ve en ařađı seviyeli ařaması bilgisayarları fiziksel olarak birbirine bađlamaktır. Ađ ierisindeki her bir bilgisayardan diđerine fiziksel bir bađlantı kurulabilmesi gerekir. Bunun iin seri portlar kullanılabileceđi gibi ok daha geliřmiř zel network kartları (NIC, Network Interface Card) kullanılabilir. Bugn PC'lerde yaygın olarak Ethernet kartları olarak bilinen zel network kartları kullanılmaktadır.

18.2. Protokol Kavramı ve Protokol Katmanları

Bir haberleřme iin bazı ortak kurallara uyulması gerekmektedir. Bu ortak kurallara protokol denir. Protokollerde tamamen yazılımdaki ktphaneler gibi st ste yıđılmıřtır. Bir protokol

diğerinin üzerinde ise diğerinin zaten var olduğu fikri ile tanımlamalar yapmaktadır. Böylece hem protokol oluşturmada esneklik sağlanmış olur hem de daha bağımsız tanımlamalar yapılabilir. Örneğin TCP protokolü IP protokolünün üzerinde, FTP ise TCP protokolünün üzerindedir. Yani TCP protokolü zaten IP protokol kurallarının var olduğu durumda ek bir takım olanaklar sunmaktadır. FTP protokolünde transfer işlemi TCP olanakları ile gerçekleştirilmektedir.

Katmanlı ağ oluşturmak için çeşitli önerilerde bulunulmuştur. Bunlardan en yaygın bilineni OSI (Open System Interconnection) modelidir. OSI modelinde bir protokol ailesi oluşturmak isteyenlerin tipik olarak hangi katmanları oluşturması gerektiği belirtilmektedir. Bu model yalnızca öneri sunmakta herhangi bir bağlayıcılığı bulunmamaktadır. Örneğin IP protokol ailesi 7 katmanlı OSI modeline uymamaktadır. OSI katmanları şunlardır:

- Application Layer (Uygulama Katmanı)
- Presentation Layer (Sunum Katmanı)
- Session Layer (Oturma Katmanı)
- Transport Layer (İletim Katmanı)
- Network Layer (Ağ Katmanı)
- Data-Link Layer (Veri Bağı Katmanı)
- Physical Layer (Fiziksel Katman)

Fiziksel katman network kartı, haberleşmede kullanılan kablolu sistem ve sinyal karakteristikleri gibi belirlemeler içermektedir. Örneğin Ethernet protokolü yani Ethernet kartları ve genel anlamda bu karakteristikler büyük ölçüde fiziksel katmana ilişkindir. Örneğin biz bilgisayarları seri porttan da birbirine bağlayabiliriz Ethernet karttan da. Fiziksel olarak bu iletişimler farklıdır. Fakat neticede bir bilgisayardaki baytlar diğerine gönderilebilmektedir. Bu en aşağı katmandır ve genellikle IP protokol ailesinde bu katmanı Ethernet kartları ve kablolu ve aktarım standartları oluşturmaktadır.

Data-Link katmanı fiziksel katmanın üzerindeki daha organizasyonel katmandır. Bu katmanda tipik olarak şu belirlemeler yapılmaktadır:

- Makinelerin fiziksel adreslenmesi: Örneğin Ethernet protokolünde her bir Ethernet kartının dünya genelinde tek olan bir MAC adresi vardır. Yani bir makine başka bir makineye bilgi gönderecekse o makinedeki Ethernet kartının MAC adresini bilmek zorundadır. MAC adresi Windows sistemlerinde ipconfig, Unix sistemlerinde ifconfig komutu ile elde edilebilir.

- Akış kontrolü veri bağı katmanı ile ilişkilendirilmiştir.
- Hata (Error) kontrolü: Gönderilen ve alınan bilginin hatalı gönderilip gönderilmediğinin belirlenmesi bu katmana ilişkindir.

Görüldüğü gibi Ethernet protokolü daha çok fiziksel ve veri bağı protokollerinin birleşiminden meydana gelmektedir.

Ağ katmanında artık bilgiler fiziksel olarak gönderilip alınabilmekte ve bazı mantıksal organizasyonlar yapılabilmektedir. Ağ katmanında genellikle mantıksal bir adresleme sistemine ilişkin belirlemeler yapılmaktadır. Örneğin fiziksel bakımdan her makinenin MAC adresi bulunabilir fakat ağların birbirine bağlanması durumunda pek çok bakımdan MAC adresi kullanışsız olmaktadır. Bunun yerine ağdaki her bir elemana başka bir biçimde adresler verilebilir. Örneğin IP protokolü tipik olarak ağ katmanına ilişkindir ve burada mantıksal adresleme olarak IP adresleri kullanılmaktadır. Ayrıca ağ katmanında bilgi paket paket gönderiliyorsa bunların nasıl ve hangi rotadan gönderileceğine ilişkin belirlemeler de bulunmaktadır.

İletim katmanında artık güvenliği artırmak için daha yüksek seviyeli belirlemeler yapılmaktadır. Örneğin IP protokol ailesindeki TCP ve UDP protokolleri iletim katmanına ilişkindir. Bu katmanda adresleme konusunda daha ince belirlemeler yapılır. Örneğin port kavramı bu katmana ilişkin bir tanımlamadır. Yine bu katmanda eğer bir bağlantı kullanılacaksa buna ilişkin belirlemeler ya da datagram biçiminde gönderim alım olacaksa buna ilişkin belirlemelerde yapılmaktadır. Yine akış ve hata kontrolü bu aşamada da uygulanabilir. Haberleşme güvenliğini artırıcı unsurlar bu katmana ilişkindir.

Oturum katmanında iletişimin genel biçimi tanımlanır. Örneğin full-duplex mi half-duplex mi olduğu belirlenir. Senkronizasyon işlemleri de bu katmana ilişkindir.

Sunum katmanında artık haberleşmeye ilişkin pek çok detay halledilmiş durumdadır. Bu katman tipik olarak sayısal ve alfabetik dönüştürmeler, sıkıştırma ve açma işlemleri, şifreleme ve çözme işlemleri gibi belirlemeler ile uğraşır.

Uygulama katmanında nihayet en yüksek seviyeli protokoller bulunmaktadır. Yani artık gerçek işi yapan uygulama yazılımları bu katmana ilişkindir. Örneğin IP protokol ailesinde http, FTP, POP3, SMTP gibi protokoller tipik olarak yüksek seviyeli protokollerdir ve uygulama katmanına ilişkindirler.

Şüphesiz bir protokol ailesinde bu yedi katman için bire bir karşılıklı bir katmanlama yapılmayabilir. Örneğin IP protokol ailesi kabaca 4 katmanlıdır ve OSI modeline uymamaktadır.

OSI modelindeki fiziksel ve veri-bağı katmanları IP ailesinde tek bir katman olarak değerlendirilir. Ağ ve iletim katmanları OSI'de belirtildiği biçimde kullanılmıştır. Oturum ve Sunum katmanları bir anlamda yoktur, bir anlamda ise belli ölçüde diğer katmanların içersindedir. Fakat uygulama katmanı OSI'de belirtildiği gibidir. IP protokol katmanları aşağıdaki gibidir:

- Uygulama Katmanı
- İletim Katmanı
- Ağ Katmanı
- Veri-Bağı ve Fiziksel Katman

18.3. TCP/IP Protokol Ailesi

IP (Internet Protocol) ailesi denildiğinde yalnızca IP protokolü değil bunun yukarısındaki tüm protokoller anlaşılmalıdır. IP protokolü OSI modelindeki ağ katmanına karşı gelmektedir. Paket anahtarlama (packet switching) bir yapıya sahiptir. Bu protokolda ağa bağlı her bir birime host denilmektedir. Host bir bilgisayar olabileceği gibi bir yazıcı ya da kamera olabilir. Bu protokolda her hostun IP numarası denilen mantıksal adresi vardır. IPv4'te IP numarası 4 bayt (32 bit) uzunluğundadır. IPv6'ta 16 bayt (128 bit) uzunluğa ötelenmiştir.

IP protokolünü kullanarak biz bir grup bilgiyi (buna paket denir) farklı yollardan belli bir hosta gönderebiliriz. Bunun için hostun IP numarasını bilmemiz gerekir. IP protokolü ağlar arası bir protokol olduğuna göre ilgili host bizim yerel ağımızda bulunabileceği gibi başka bir ağda da bulunuyor olabilir.

IP protokolü 1983 yılında İnternetin resmi protokolü haline getirilmiştir. Dolayısıyla biz IP protokol ailesini kullanarak hem kendi yerel ağımızda (Intranet Uygulamaları) hem de global İnternet denilen ağda uygulamalarımızı çalıştırabiliriz.

18.3.1. TCP ve UDP Protokolleri

İletim katmanındaki protokoller bağlantılı (connection oriented) ya da bağlantısız (connectionless) olabilir. TCP bağlantılı, UDP bağlantısız bir protokoldür. Bağlantılı protokollerde önce gönderici ile alıcı arasında bir bağlantı kurulur, sonra paketler güvenli bir biçimde aktarılır.

Aktarım sırasında bağlantı korunmaktadır. Paket aktarımının doğru yapılıp yapılmadığı belirlenir ve duruma göre hatayı telafi etmek için yeniden gönderim uygulanır. Hâlbuki bağlantısız protokollerde gönderim sonlanana kadar bir bağlantı oluşturulmaz. Gönderici paketi gönderir, alıcının alıp almadığı ile ilgilenmez. Bağlantılı protokoller daha güvenli fakat yavaştır. Bağlantısız protokoller daha hızlı fakat güvensizdir. Bağlantısız protokoller bir radyo yayınına benzetilebilir. Yayının alıcı tarafından alınıp alınmadığını, yayını yapan bilmemektedir. Bağlantısız protokoller görüntü, ses gibi bilgilerin iletilmesinde tercih edilebilir. Genellikle bu tür uygulamalarda arada bir gönderinin başarısız olmasının ciddi bir önemi yoktur. Fakat IP protokol ailesinin uygulama katmanındaki protokolleri TCP üzerine kurulmuştur. Bu nedenle TCP tabanlı çalışmaya TCP/IP’de denilmektedir. UDP tarzı çalışmaya UDP/IP denilmektedir.

Bir protokol özellikle iletim katmanı tarafından stream tabanlı ya da datagram tabanlı biçimde oluşturulabilir. TCP (Transmission Control Protocol) stream tabanlı bir çalışma sunar. UDP (User Datagram Protocol) ise datagram tabanlı bir çalışma sunar. Datagram gönderilen ve alınan paketlere verilen bir isimdir. Şüphesiz TCP de paket haberleşmeli bir altyapıya sahiptir. Çünkü gerek TCP, gerek UDP IP protokolü üzerine oturtulmuştur. Paketlerin gönderilip alınması IP protokolüne ilişkindir. UDP ‘deki datagram terimi mesaj tabanlı bir çalışmayı anlatmak için kullanılmıştır.

Stream haberleşmesinde bilgiler bir bayt topluluğu olarak alıcıya gelir. Alıcı gelen bilgilerden istediği kadarını elde edebilir. Hepsini okumak zorunda değildir. Gelen bilginin organizasyonunu ancak kendisi anlayabilir. Halbuki mesaj tabanlı sistemde gelen bilgi paketlenerek gelmektedir. Alıcı taraf bunu belirlenmiş bir grup olarak alabilir. Şüphesiz TCP ve UDP’nin her ikisinde de arka planda yine IP protokolü kullanılarak paketli haberleşme yapılmaktadır. Fakat bilgi karşı tarafa ulaştığında artık paketlemenin hiçbir önemi kalmamıştır. Örneğin: Biz TCP protokolü kullanarak bir miktar baytı göndermek isteyelim. Aslında TCP, IP protokolünü kullanarak bunu paketler halinde gönderecektir. Fakat gönderilecek bilgi tek bir IP paketine sığmayabilir. Bu durumda TCP protokolü birden fazla IP paketi oluşturarak bunu gönderir. Her IP paketinin içerisinde bir başlık kısmı vardır. Alıcı taraf, paketler değişik rotalardan gidebildiği için farklı sıralardan alabilir. Alıcı taraf bir takım IP paketlerini almıştır. Fakat TCP protokolünde bir TCP başlığı da vardır. Alıcı taraf TCP protokolünün yerleştirdiği bu bilgilere dayanarak paketleri birleştirir ve nihayet gönderilmiş olan bilgileri bir bayt topluluğuna dönüştürür. İşte stream tabanlı çalışmalara karşımızda nihayet bir bayt topluluğu vardır. Karşı taraf

3000 bayt gönderdiği halde biz istediğimiz kadar bayt okuyabiliriz. Fakat mesaj tabanlı çalışmada genel olarak bilgi karşı tarafta toplandığında bir bayt topluluğu olarak değil yine gruplanmış bir biçimde tutulur. Alıcı taraf bu grubu alabilmektedir.

Ayrıca TCP ve UDP protokollerinde artık bir de port numarası devreye girmiştir. Buradaki port kavramının, seri ve paralel portlarda olduğu gibi fiziksel bir anlamı yoktur. Port numarası hotsa gelen IP paketlerinin uygulama temelinde ayrıştırılması için düşünülmüştür. Bir şirketteki dâhili numaralara benzetilebilir. IP protokolü port numarası kavramını kullanmaz, dolayısı ile bu protokolde belli bir hosta gönderim yapılabilir. O host içerisindeki bir uygulamaya gönderim yapılamaz. Örneğin bir TCP bilgisi hosta ulaşmış olsun. O host üzerindeki pek çok program hosta gelen bilgiyi okumak isteyebilir. Peki, hangi program gelen bilgiyi elde edecektir? İşte daha gönderim sırasında TCP ve UDP protokollerinde gönderici bir port numarası da belirtmek zorundadır. İşte o hostta okuma yapan tüm programlar aslında bir port numarası eşliğinde okuma yaparlar. İşletim sistemi gelen tüm TCP ve UDP paketlerini bir port numarası eşliğinde organize edip programlara dağıtmaktadır. Toplam port numarası 65536 tanedir. İlk 1024 port numarası uygulama katmanındaki yüksek seviyeli protokollere ayrılmıştır. Örneğin HTTP 80 numaralı portu kullanmaktadır. Örneğin bir browser bir web sayfasına bağlanırken, TCP protokolünü ve 80 numaralı portu kullanmaktadır.

18.4. Client-Server Çalışma Modeli

TCP bağlantılı bir protokol olduğu için tipik olarak client-server bir çalışma modeli aklı gelmektedir. Client-server çalışma modelinde bir server program bir ya da birden fazla client programlar vardır. Server program gerçek işi yapan programdır. Client serverdan istekte bulunur, server işi yapıp sonuçları gönderebilir. Bir server genellikle aynı anda birden fazla client'a hizmet verecek şekilde oluşturulmaktadır. Peki, client'lar neden kendi işlerini yapmayıp server'a ihtiyaç duyuyorlar?

- Bazı kaynaklar (veri tabanları, yazıcılar) fiziksel olarak bazı hostlara bağlıdır, yada onların kontrolü altındadır. Herhangi bir hostun o kaynakları kullanabilmesi için client-server modeli bir çalışmaya gereksinim duyulur. Örneğin veri tabanı yönetim sistemleri tipik olarak client-server mimarisi ile tasarlanmıştır. Biz veri tabanı yönetim sistemine bir sql

cümlesi gönderebiliriz. Veri tabanı yönetim sistemi pek çok client ile birlikte bize de hizmet verebilir. Ya da örneğin printer bir hosta bağlıdır. Biz client olarak bir sayfanın yazılmasını ondan isteyebiliriz. Bazen böyle bir mimari tamamen bir zorunluluk haline gelebilir. Örneğin bir ev terminali ya da bir ATM cihazı tüm dataları kendi içerisinde bulunduramaz. O halde ATM cihazları birer client program durumundadır. Server program merkezdedir.

- Bazı durumlarda bir iletişime giren kişinin diğerlerini tanıyabilmesi gerekir. Bu tür durumlarda client-server çalışma zorunlu olarak kullanılır. Chat programları bunlara tipik örneklerdir. Biz sisteme girdiğimizde kimlerin bağlı olduğunu bilemeyiz. Fakat server bunu bilebilir. Örneğin msn programı böyledir. Msn Messenger bir client programdır. Server a bağlanıldığında server client in veri tabanına bakar ve onu contact listesinde kimlerin olduğunu ve onların hangilerin online olduğunu belirler ve client a mesajlar göndererek onların yeşil gözükmesini sağlar. Kişi sistemden çıkarken bunu yine server a söyler. Server kişinin contact listesindeki herkese mesaj göndererek client programların onları offline göstermesini sağlar.
- Server programların çalıştığı hostlar özel ve güçlü donanımlara sahip olabilir. Client bu güçten faydalanmak için işini server'a yaptırabilir. Örneğin bir süper bilgisayara bağlanıp hızlı bir hesabı ona yaptırabiliriz.
- Client-server çalışma modeli dağıtık uygulamalarda da kullanılmaktadır. Server işi dağıtan program durumundadır ve burada adeta ters bir mantık işletilir. Server işi dağıtarak client lara yaptırır.

18.5. Soket Arayüzü

IP protokol ailesi işletim sistemi tarafından, işletim sisteminin bir parçası olarak gerçekleştirilmiş olmalıdır. Yani işletim sisteminin içerisinde Ethernet kartını kontrol eden sürücüler, IP paketini oluşturan, gönderip alan kodlar, TCP konuşmasını gerçekleştiren kodlar hep bulunmaktadır. Fakat bir biçimde programcı devreye girerek bu işleri yaptırması gerekir. Nasıl

işletim sistemlerinin API'leri varsa ağ haberleşmesi içinde buna benzer API'ler bulunmalıdır. İşte ağ haberleşmesi için kullanılan API fonksiyonlarına soket arayüzü denilmektedir.

Soket arayüzü pek çok protokol için ortak fonksiyonları bulunduran genel bir arayüzdür. Yani biz soket fonksiyonlarını kullanarak IP protokol ailesinde ya da örneğin AppleTalk ailesinde aynı fonksiyonlar ile çalışabiliriz. Tabi biz kursumuzda bu fonksiyonları IP ailesi üzerinde kullanacağız.

18.6. WinSock Arayüzü

Soket arayüzü ilk kez BSD/UNIX sistemlerinde gerçekleştirilmiştir. Daha sonra pek çok sistemde benzer şekilde gerçekleştirilmiştir. Microsoft, orijinal BSD soket arayüzünü Windows sistemlerine uyumlu hale getirmiştir. Microsoft'un Windows için yazdığı bu soket arayüzünü WinSock denilmektedir.

Microsoft'un soket arayüzünde iki grup fonksiyon vardır. İlki küçük harflerle isimlendirilmiş UNIX uyumlu fonksiyonlardır. İkincisi WSA ile başlayan Macar notasyonu ile isimlendirilmiş soket fonksiyonlarıdır. WSA fonksiyonları daha geniştir ama taşınabilir değildir. Burada, UNIX uyumlu fonksiyonlar kullanılacaktır.

18.6.1. WinSock Arayüzü İçin İlk ve Son İşlemler

Windows soket sistemini proses temelinde başlatmak için ve işlemi sonlandırmak için başlangıçta ve bitimde iki fonksiyon çağırılmalıdır. Bu çağrılar UNIX grubu sistemlerde yapılması gerekmemektedir. Bu çağrılar windows sistemine özgüdür ve UNIX uyumlu fonksiyonları kullansak bile bu çağrılar yapmak zorundayız.

Anahtar Notlar:

Windows ta soket işlemleri için, Winsock2.h isimli başlık dosyası dahil edilmelidir. Ayrıca, WinSock Kütüphanesi Ws2_32.dll içerisindedir. Bu nedenle, link aşamasında Ws2_32.lib import kütüphanesinin dahil edilmesi gerekir.

Anahtar Notlar:

Tipik bir TCP/IP client-server uygulama için bir client bir de server program yazılmalıdır. Visual Studio IDE sinde client-server programları organize etmek için, boş bir solution açıp içerisine client ve server projeleri

dahil edilmesi iyi bir tekniktir. Bu işlem iki biçimde yapılabilir. Birincisi, doğrudan ilgili import kütüphanesini projeye eklemektir. İkincisi, Linker'dan dependencies bölümüne lib dosyası eklenir.

WinSock sistemini başlatmak için WSStartup fonksiyonu kullanılır.

```
int WSStartup(WORD wVersionRequested, LPWSADATA lpWSADATA);
```

Fonksiyonun birinci parametresi talep edilen WinSock versiyon numarasıdır. Son versiyon 2.2 olduğu için bu talep yapılabilir. WORD türü işaretli 2 baytlık tamsayı türüdür. Bu iki bayt yüksek anlamlı baytı büyük numara (major), düşük anlamlı küçük numara biçiminde girilir. İkinci parametre WSADATA isimli bir yapının adresini almaktadır. Fonksiyon başarı durumunda 0 değerine, başarısızlık durumunda hata değerine geri döner. WSStartup fonksiyonu WSADATA yapısının içeriği değerli bilgilerle doldurur. Fonksiyonun birinci parametresinde yüksek bir WinSock numarası girilirse fonksiyon yine başarılı olur ve sistemde yüklü olan en yüksek WinSock u kullanır. Fonksiyonun birinci parametresi tipik olarak MAKEWORD makrosuyla oluşturulur. O halde fonksiyon tipik olarak şöyle çağrılabilir.

```
WSADATA wsd;
int result;

if ((result = WSStartup(MAKEWORD(2, 2), &wsd)) != 0) {
    fprintf(stderr, "Cannot load winsock library: %d\n", result);
    exit(EXIT_FAILURE);
}
```

WinSock sisteminin program sonunda WSACleanup fonksiyonu ile boşaltılmalıdır. Fonksiyon başarılı ise sıfıra, başarısızsa SOCKET_ERROR değerine geri döner. Geri dönüş değerinin test edilmesine gerek yoktur.

```
int WSACleanup(void);
```

Anahtar Notlar:

Pek çok soket fonksiyonu başarısızlık durumunda SOCKET_ERROR değerine döner. Fakat ayrıca başarısızlığın nedenini elde etmek için Windows sistemlerinde aşağıdaki fonksiyon kullanılmaktadır.

```
int WSAGetLastError(void)
```

Bu fonksiyon, son soket işlemindeki hatanın nedenini anlatan bir sayıya geri döner. Tüm hata değerleri düzenli olarak WSA_XXX biçimindeki sembolik sabitler olarak define edilmiştir. UNIX sistemlerinde klasik olarak errno yöntemi kullanılmaktadır.

18.7. Server Programının Organizasyonu

Server Programının tipik olarak şu aşamalardan geçilerek yazılır.

1. Dinleme soketi oluşturulur.
2. Soket bind edilir.
3. Listen fonksiyonu ile soket dinleme konumuna geçirilir.
4. Accept işlemi ile client bağlantıları kabul edilir.
5. Send ve receive fonksiyonları ile gönderim ve alım yapılır.
6. Shut down ve closeSocket fonksiyonları ile iletişim sonlandırılır.

Windows sistemlerinde bu işlemler için çağrılacak fonksiyonlar şunlardır.

WSAStartUP

socket

bind

listen

accept

Send/recieve

Shut down

Closesocket

WSACleanun

18.7.1. socket Fonksiyonu

Soket arayüzü tipik bir handle çalışması sunmaktadır. Handle sistemini oluşturan fonksiyon socket fonksiyondur. Kapanan closeSocket fonksiyonudur.

```
SOCKET socket(int af, int type, int protocol);
```

Fonksiyonun 1. parametresi kullanarak protokol ailesini belirtmektedir. Bu parametre için AF_INET (PF_INET) kullanılır. İkinci parametre, kullanılacak soketin stream mi datagram mı olduğunu anlatan tür parametresidir. TCP çalışma için (TCP stream tabanlı olduğundan) bu parametre SOCK_STREAM biçiminde girilmelidir. UDP için (mesaj tabanlı) SOCK_DGRAM kullanılır. 3 parametre kullanılacak protokol ailesindeki spesifik protokolü belirtmektedir. Aslında IP ailesi için bu parametrenin girilmesine gerek yoktur. 0 olarak geçilebilir. Bu durumda ikinci parametre SOCK_STREAM ise TCP, SOCK_DGRAM ise UDP çalışma söz konusudur. Fakat

yine de 0 yerine IPPROTO_TCP ya da IPPROTO_UDP geçilebilir. Başarı durumunda SOCKET türü ile temsil edilen handle durumuna geri döner. Fonksiyon başarısızlık durumunda SOCKET_ERROR değerine geri dönmektedir. WSAGetLastError ile hatanın nedeni alınabilir. Bu durumda, tipik olarak TCP/IP çalışma için socket şöyle oluşturulabilir.

```
SOCKET serverSocket;  
/*****  
****  
*****/  
if ((serverSocket = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP)) ==  
INVALID_SOCKET) {  
    fprintf(stderr, "can not create");  
    exit(-1);  
}
```

Server program socketi oluşturduktan sonra bağlamalıdır. Sokeni oluşturduktan sonra hangi network kartından ve hangi porttan gelecek bağlantı isteklerinin kabul edileceğini belirlemek için yapılır.

18.7.2. bind Fonksiyonu

Prototipi şöyledir.

```
int bind(SOCKET s, const struct sockaddr *name, int namelen);
```

Birinci parametre socket fonksiyonundan elde edilen handle'dır. Programcı bir yapının içini doldurarak bunun adresini ikinci parametre olarak geçmelidir. Fonksiyon prototipindeki struct sockaddr, tüm protokoller için ortak olan bir yapıyı belirtmek için kullanılmıştır. Aslında her protokolün ayrı bir yapısı vardır. Örneğin TCP/IP için sockaddr_in isimli yapı kullanılmaktadır. Yani bind fonksiyonunun ikinci parametresi void * olabilir. Şüphesiz biz sockaddr_in yapısının içini doldurduktan sonra adresi dönüştürerek fonksiyona vermeliyiz. Bind fonksiyonu ikinci parametrede aslında hangi yapının adresi verildiğini üçüncü parametreden anlamaktadır. Üçüncü

parametreye ikinci parametreye geçilen yapının bayt uzunluğu girilmelidir. Bind fonksiyonu başarı durumunda sıfır, başarısızlık durumunda SOCKET_ERROR değerine döner ve başarısızlığın nedeni WSAGetLastError fonksiyonu ile elde edilebilir.

Şimdi sockaddr_in yapısının içinin doldurulduğunu varsayalım. bind fonksiyonunu şöyle çağıracağız.

```
struct sockaddr_in sinServer;
/***/
if (bind(serverSocket, (struct sockaddr *) &sinServer, sizeof(sinServer))
    == SOCKET_ERROR) {
    fprintf(stderr, "Cannot bind socket: %d\n", WSAGetLastError());
    exit(EXIT_FAILURE);
}
```

sockaddr_in yapısı aşağıdaki gibi bildirilmiştir.

```
struct sockaddr_in {
    short sin_family;
    unsigned short sin_port;
    struct in_addr sin_addr;
    char sin_zero[8];
};
```

Kullanıcı bu yapıyı doldurmalıdır. sin_family elemanına protokol ailesi girilir. (TCP için) AF_INET girilir. Yapının sin_port elemanına bağlantının kabul edileceği port numarası girilir. Yapının sin_addr elemanı in_addr isimli bir yapı türündendir. Bu elemana, dinlemenin yapılacağı network kartının IP numarası girilmelidir. Fakat, özel olarak INADDR_ANY değeri hosta bağlı tüm network kartlarını belirtmektedir. Sin_zero elemanı kullanılmaktadır. Tipik olarak bu elemanda sıfır değerleri bulunur. Bu eleman boş bırakılabilir. IP ailesinde dikkate alınmamaktadır.

Genel olarak protokoller heterojen makinelerin birbirine bağlanmasında kullanıldığından endianlık durumu sorun oluşturabilir. Örneğin port numarası Intel işlemcilerinde little, alfa işlemcilerinde big endian olarak kodlanırsa sorun oluşur. İşte protokol ailesinde genel olarak bir byte tan uzun olan tüm sayılar için ortak bir endian seçilmelidir. **İşte IP ailesinde big endian notasyon kullanılmaktadır.** Bu nedenle protokol numarası, IP numarası gibi sayıların hepsi big endian formata dönüştürülmelidir. Bu dönüştürmeyi yapan htons ve htonl fonksiyonları vardır. htons (host to network byte order short) short değeri dönüştürmek için, htonl (host to network

byte oridiren long) long değeri dönüştürmek için kullanılır. O halde, tipik olarak sockaddr_in yapısının içi şöyle doldurulmalıdır.

```
struct sockaddr_in sinServer;  
  
sinServer.sin_family = AF_INET;  
sinServer.sin_port = htons(PORTNO);  
sinServer.sin_addr = htonl(INADDR_ANY);
```

bind işleminden sonra, artık socket dinleme durumuna getirilmelidir. Bu işlem listen fonksiyonu ile yapılır.

18.7.3. listen fonksiyonu

Fonksiyon prototipi şöyledir.

```
int listen(SOCKET s, int backlog);
```

Fonksiyonun birinci parametresi socketin handle değeri. İkinci parametresi bekleme kuyruğunun uzunluğunu belirtir. Client lar tarafından yapılan bağlantı istekleri bir kuyruk sisteminde saklanır. Accept fonksiyonu kuyrukta sırada bulunan bağlantı isteğini alarak onu kabul eder. Örneğin, server accept fonksiyonunu çağırılmadan üç farklı client bağlanma isteğinde bulunuyor olsun. Bu bağlantı istekleri bir kuyrukta saklanır. Tabi ki bir zaman aşımı periyodu vardır. Daha sonra accept uygulandığında, kuyruktaki sıraya göre bağlantı alınır. İşte ikinci parametre bu kuyruğun uzunluğunu belirtmektedir. Mesela 8 değeri uygun bir değerdir. Fonksiyon başarılıysa sıfır değerine, başarısızsa SOCKET_ERROR değerine döner. WSAGetLastError ile hatanın nedeni alınabilir. Listen fonksiyonu tipik olarak aşağıdaki gibi çağırabilir.

```
if (listen(serverSocket, 8) == SOCKET_ERROR) {  
    fprintf(stderr, "Cann not listen");  
    exit(-1);  
}
```

Listen fonksiyonu çağrıldığında artık işletim sistemi network kartına gelen paketleri kontrol eder ve bağlantı isteklerini belirleyerek kuyuklar. Yani listen fonksiyonu bu işlemi başlatmaktadır, aktif dinlemeyi işletim sistemi yapmaktadır. Akış listen da beklememektedir.

Artık bağlantıyı kabul etme zamanı gelmiştir. Accept fonksiyonu çağrılarak bağlantılar kabul edilebilir. Şüphesiz bir server birden fazla client bağlantısını kabul edebilir. Yani accept fonksiyonu bir kez çağrılmak zorunda değildir. Her çağrıldığında yeni bir bağlantı kabul edilir. O halde çok clientli uygulamalar için tipik olarak, accept fonksiyonu bir döngü içerisinde birden fazla kez çağrılmalıdır.

18.7.4. Accept Fonksiyonu

Blokeli ve blokesiz modda kullanılabilir. Default durum blokeli moddur. Blokesiz mod işlemleri hakkında genel bilgiler verilecektir. Blokeli modda accept fonksiyonu çağrıldığı zaman, akış accept fonksiyonu içerisinde kalır. Ta ki bir client bağlantısı kabul edilene ya da socket kapatılana kadar. Yani accept çağrıldığında eğer zaten connect isteğinde bulunan bir client varsa, accept bu bağlantıyı kabul ederek hemen geri döner. Yoksa, accept bir client connect uygulayana kadar bekler. Accept fonksiyonu ile bağlantı kabul edildiğinde fonksiyon yeni bir socket geri verir. Bu socket bağlanılan client ile konuşmakta kullanılır. Bağlantı sağlamak için kullanılan socketle, bağlantı kabul edildikten sonra elde edilen socket farklıdır. Accept fonksiyonu başarısızlık durumunda INVALID_SOCKET değeri ile geri döner. Başarısızlığın nedeni WSAGetLastError fonksiyonu ile elde edilebilir. INVALID_SOCKET -1 olarak define edilmiştir. Fonksiyon prototipi şöyledir.

```
SOCKET accept(SOCKET s, struct sockaddr *addr, int *addrlen);
```

Birinci parametre, dinleme socketinin handle değerini alır. İkinci ve üçüncü parametreler bağlantıdan sonra client bilgilerini içerecek biçimde doldurulur. Yani, bağlanılan client e ilişkin sockaddr_in yapısı, buraya doldurulmaktadır. IP ailesinde bu parametre tipik olarak client in IP adresinin elde edilmesi için kullanılır. Üçüncü parametre, IP ailesi için sockaddr_in yapısının sizeof değerini tutan int türden bir nesnenin adresi biçiminde girilmelidir. Son parametre NULL geçilebilir. Bu durumda karşı tarafa ilişkin bir bilgi edinilmez. Fonksiyonun geri dönüş değeri

kabul edilen client a ilişkin, onunla konuşmak için gereken socket değeridir. Fonksiyon tipik olarak şöyle çağrılabilir.

```
int  addrsize;
SOCKET clientSocket;
struct sockaddr_in sinClient;

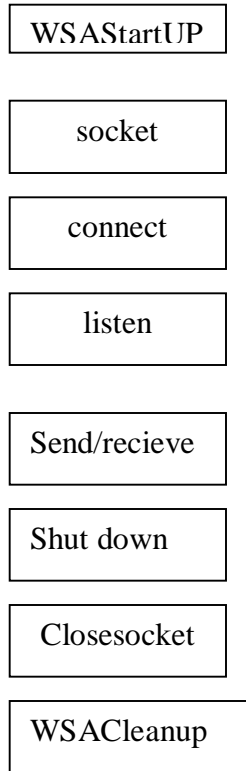
addrSize = sizeof(sinClient);

if (clientSocket = accept(serverSocket, (struct sockaddr *)&sinClient,
    &addrSize)) == INVALID_SOCKET) {
    fprintf(stderr, "Can not accept: %d\n", WSAGetLastError());
    exit(-1);
}
```

Accept fonksiyonu birden fazlaca çağrılarak farklı clientlarla bağlantılar kabul edilebilir. Artık accept işleminden sonra, bağlantı gerçekleşmiştir. Client ile server send ve receive fonksiyonları ile full duplex haberleşme yapabilirler.

18.8. Client Programın Organizasyonu

Client tıpkı telefonda arama işleminde olduğu gibi, servreın IP adresini bilerek ona bağlanmaya çalışır. Bağlantı sonrasında server bağlanılan clientın IP adresini elde edebilmektedir. Client programın sırasıyla şunları yapması gerekir.



Görüldüğü gibi client önce bir soket yaratır. Sonra o soketle connect fonksiyonunu çağırarak server a bağlanmak ister.

18.8.1. Connect Fonksiyonu

connect fonksiyonunun prototipi şöyledir.

```
int connect(SOCKET s, const struct sockaddr *name, int namelen);
```

Birinci parametresi yaratılan soketi alır. İkinci parametre IP ailesi için sockaddr_in türünden bir yapının adresi olarak girilmelidir. Client serverın IP adresini bu yapının içerisine yerleştirir. Üçüncü parametre yine ikinci parametre ile girilen yapının uzunluğudur. Başarı durumunda 0 değerine, başarısızlık durumunda SOCKET_ERROR değerine döner. Başarısızlığın nedeni WSAGetLastError fonksiyonu ile elde edilebilir.

18.8.2. Server IP adresini sockaddr_in yapısına yerleştirilmesi

Client connect fonksiyonunu çağırılmadan önce, sockaddr_in türünden bir yapı nesnesi tanımlayıp bunun içeriğini doldurmalıdır. Bu yapının sin_family ve sin_port elemanları server da olduğu gibi AF_INET ve PORTNO(server port numarası) ile doldurulmalıdır.

```
struct sockaddr_in sinServer;  
  
sinServer.sin_family = AF_INET;  
sinServer.sin_port = PORTNO;
```

Sıra IP adresinin yerleştirilmesine gelmiştir. Anımsanacağı gibi, sockaddr_in yapısının sin_addr elemanı struct in_addr isimli yapı türündendir. In_addr yapısı aşağıdaki biçimdedir.

```
struct in_addr {  
    union {  
        struct {UCHAR s_b1, s_b2, s_b3, s_b4} S_un_b;  
        struct {USHORT s_w1, s_w2} S_un_w;  
        ULONG s_addr;  
    } S_un;  
};
```

Görüldüğü gibi yapının S_un elemanı bir birlik türündendir. O birliğin s_addr elemanı IP adresini bir long değer olarak alır. Birliğin diğer elemanları IP adresini diğer formatlarda almaktadır. O halde eğer IP adresi long bir sayıya dönüştürülürse şöyle girilebilir.

```
sinServer.sin_addr.S_un.S_addr = htonl(<ip adresi>);
```

Aşağıdaki gibi bir makro bulunmaktadır.

```
#define s_addr S_un.S_addr
```

Bu durumda ifade şu şekilde de yazılabilir.

```
sinServer.sin_addr.s_addr = htonl(<ip adresi>);
```

Fakat IP adresinin long bir sayı biçiminde girilmesi pek kullanışlı değildir. Genellikle byte byte noktalı biçimde tutulmaktadır. Örneğin;

```
88.235.101.131
```

İşte `inet_addr` isimli fonksiyon noktalı biçimde verilen IP adresini long biçime dönüştürmektedir.

```
unsigned long inet_addr(const char *cp);
```

Eğer yazının formatı uygun değilse, fonksiyon `INADDR_NONE` değerine geri döner. O halde IP adresini şöyle de yerleştirebiliriz.

```
sinServer.sin_addr.s_addr = inet_addr("88.235.101.131");  
  
if (sinServer.sin_addr.s_addr == INADDR_NONE) {  
    /*****/  
}
```

İsimler daha kullanışlı olduğundan, IP numaralarının isimlerle eşleştirilmesi yoluna gidilmiştir. Bir isme karşılık hangi IP numarasının geldiği, global internet ağında çeşitli serverlar da tutulmaktadır. Programcı, ilgili serverlardan sorarak, isme ilişkin IP adresini elde edebilir. Bu hizmete Domain Name System (DNS) denilmektedir. Benzer biçimde yerel bir ağda çalışıyorsak bu durumda domain isimleri bilgisayar ismi biçimindedir ve yerel ağ tarafından çözülür.

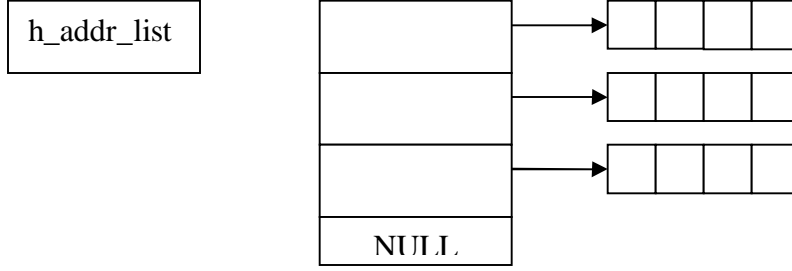
Host ismini IP numarasına dönüştürmek için `gethostbyname` fonksiyonu kullanılır.

```
struct hostent * gethostbyname(const char *name);
```

Fonksiyon host ismini alır ve static düzeyde tahsis edilmiş olan bir `hostent` yapısının adresi ile geri döner. `Hostent` yapısı şöyledir.

```
struct hostent {  
    char *h_name;  
    char **h_aliases;  
    short h_addrtype  
    short h_length;  
    char **h_addr_list;  
};
```

Yapının `hname` elemanı, host ismini içermektedir. `H_aliases` host isminin alternatif isimlerini belirten bir dizidir. Görüldüğü gibi bir hosta bağlı birden fazla isim bulunabilir. `H_addrtype` elde edilen host adresinin türünü belirtir. `H_lengt` host adresinin IP adresinin byte uzunluğunu belirtmektedir. `H_addr_list` host adreslerini gösterir. Bir host ismine karşı birden fazla IP adresi gelebilir (örneğin www.google.com host ismine bağlı pek çok IP adresi vardır)



Tipik uygulamalarda client hem IP adresiyle, hem de host ismiyle bağlantı yapabilecek biçimde tasarlanır. Bunun için klasik kullanılan yöntem şudur.

```
char hostname[] = "www.google.com";

sinServer.sin_addr.s_addr = inet_addr(hostname);

if (sinServer.sin_addr.s_addr == INADDR_NONE) {
    host = gethostbyname(hostname);
    if (host == NULL) {
        fprintf(stderr, "Unable to resolve server: %s\n", HOSTNAME);
        exit(EXIT_FAILURE);
    }
    memcpy(&sinServer.sin_addr.s_addr, host->h_addr_list[0],
        host->h_length);
}
```

18.8.3. Send ve Receive İşlemleri

Send ve receive işlemleri tipik olarak blokeli ve blokesiz modda yapılabilir. Default mod blokeli. Dolayısıyla burada send ve receive fonksiyonların blokeli moddaki davranışı ele alınacaktır.

Modern sistemlerde hosta IP paketi geldiğinde sistem bunu değerlendirerek gelen bilgiyi uygun biçimde tamponlar. Benzer biçimde bilgi gönderilirken de önce tampona alınmakta, sonra uygun zamanda etkin biçimde gönderilmektedir. Böylece okuma için receive fonksiyonunu çağırdığımızda daha önce gelmiş ve network tamponunda bekletilmiş olan bilgiyi okuyabiliriz. Send işlemi de böyle yürütülmektedir. Blokeli modda receive network tamponunda okunacak hiç bir byte yoksa, yeni bir byte gelene kadar blokede bekler. Fakat en az bir byte varsa receive okuyabildiği kadar byte okur. Bloke oluşmaz. Örneğin biz receive fonksiyonu ile 100 byte

okumak isteyelim. Eğer tamponda byte yoksa akış receive içerisinde bekler. Diyelim ki hosta yeni bir 50 byte gelmiş olsun. Receive 100 byte ın hepsini oukyana kadar beklemez. Bu 50 byte okuyarak başarılı biçimde geri döner. Aynı davranış send fonksiyonunda da böyledir. send fonksiyonu önce bilgileri network tamponuna yazar. Belli bir süre sonra bilgi gerçek anlamda paketlenerek gönderilir (IO scheduling). Blokeli modda send fonksiyonu network tamponu dolu olduğu için hiçbir byte tampona yazamazsa blokede bekler. En az bir byte tazıldığında bloke çözülür. Örneğin biz 100 byte lık bir bilgiyi send fonksiyonu ile göndermek isteyelim fakat network tamponu dolu olsun. Akış send fonksiyonu içerisinde bekler. Tamponda 10 bytelık yer açıldığını düşünelim. send 100 byte ı yazana kadar beklemez. 10 byte ı yazıp geri döner.

send ve receive fonksiyonlarının prototipi şöyledir.

```
int send(SOCKET s, const char *buf, int len, int flags);
int recv(SOCKET s, char *buf, int len, int flags);
```

Birinci parametreler haberleşme soketini belirtir. İkinci parametreleri transfer edilecek bilginin bulunduğu bellek adresi. Üçüncü parametre aktarılacak bilginin byte uzunluğudur. Son parametre gönderim ve alım sırasındaki bazı belirlemeler için kullanılır. Özel bir belirleme de bulunulmayacaksa 0 girilmelidir. Fonksiyonların geri dönüş değerleri, gönderilen ve alınan byte miktarıdır. Eğer karşı taraf soketi kapatmışsa receive fonksiyonu 0 değerine döner. Diğer problemlerde receive fonksiyonu SOCKET_ERROR değerine döner. Benzer biçimde send fonksiyonu karşı taraf soketi kapatmışsa ya da hata durumunda SOCKET_ERROR değerine döner.

Bu sistemde, 100 byte okunmak istensin. Receive fonksiyonunu bir kez çağırabiliriz. Bir döngü içerisinde birden fazla kez çağırarak gerekir. Klasik algoritma şöyledir:

```
int read_socket(SOCKET s, void *pBuf, int n)
{
    int result;
    int index = 0;
    int left = n;

    while (left > 0) {
        result = recv(s, (char *) pBuf + index , left, 0);
        if (result == 0)
            return index;
        if (result == SOCKET_ERROR)
            return SOCKET_ERROR;

        index += result;
        left -= result;
    }
}
```

```
    return index;
}
```

Görüldüğü gibi fonksiyon n byte okumadan geri dönmemektedir. Fonksiyonun geri dönüş değeri socket hatası oluştuğunda ya da karşı taraf n byte okuyamadan socketi kapattığında n den farklı olabilir. Send fonksiyonu da benzer biçimde yazılabilir.

```
int WriteSocket(SOCKET s, const void *pBuf, int n)
{
    int result;
    int index = 0;
    int left = n;

    while (left > 0) {
        result = send(s, (const char *) pBuf + index , left, 0);
        if (result == 0)
            return index;
        if (result == SOCKET_ERROR)
            return SOCKET_ERROR;

        index += result;
        left -= result;
    }

    return index;
}
```

18.8.4. Socketsin Kapatılması

Socketsin kapatmak için kullanılan closeSocket fonksiyonu şalterin kapatılması gibi anı bir kapatma etkisi yapmaktadır. Biz closeSocket uyguladıktan sonra artık socketsin kullanamayız. Socketsin kapatıldıktan sonra artık tampon da boşaltılır. Tampona gelmiş olan, fakat henüz okunmamış olan bilgiler okunamaz. Benzer biçimde send fonksiyonu ile tampona yazılan bilgiler closeSocket den sonra artık karşı tarafa gönderilmez. İşte bu durumu dikkate alarak socketsin iyi biçimde (graceful) kapatılması salık verilir. Bunun için shut down fonksiyonu kullanılmaktadır.

Shut down fonksiyonunun prototipi şöyledir:

```
int shutdown(SOCKET s, int how);
```

Fonksiyonun birinci parametresi socketsin handle değeri, ikinci parametresi şunlardan biri olabilir:

```
SD_SEND          SD_RECEIVE          SD_BOTH
```

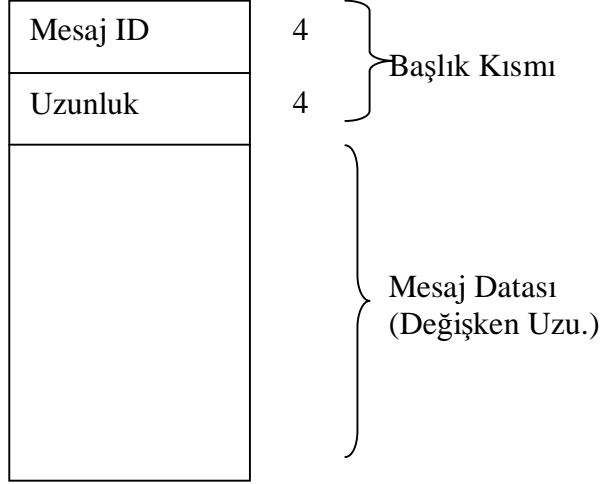
İkinci parametre SD_SEND geçilirse artık gönderme işlemi yapılamaz fakat network tamponuna gelmiş olan bilgiler recv fonksiyonu ile okunabilir ve tampondaki bilgilerde gönderilebilir. SD_RECEIVE kullanılırsa artık alma işlemi yapılamaz fakat gönderme işlevi yapılabilir. SD_BOTH her ikisinin de yapılabileceği anlamındadır. En çok kullanılan değerdir. Bu işlem karşı tarafla bağlantıyı normal olarak sonlandırır. Network tamponundaki bilgilerin gönderilmesine yol açar. Eğer soket kapanmadan önce tüm mesajlaşmalar bitirildiyse bu parametre SD_BOTH biçiminde geçilebilir. Kişi gönderme işlemini bitirdikten sonra tampondakileri okumak isterse bu parametreyi SD_SEND olarak geçebilir. shutdown fonksiyonu IP haberleşmesinin sonlandırılması için gereken bir takım işlemleri düzenli olarak yapmaktadır.

18.9. Client-Server Uygulamalarda Düzenli Bilgi Gönderilip Alınması

Client-Server haberleşmelerde client serverdan pek çok istekte bulunabilir, serverda client a çeşitli bilgiler iletebilir. Örneğin IRC stili bir chat uygulamasında bir client servera bağlandığında servera bir nick eşliğinde bir nick(takma ad) yollar. Server bunu kendi listesi içinde tutar, sonra client bir mesaj yazdığı anda serverdan bu mesajı herkese dağıtmasını ister. Sonra client sistemden çıkmadan önce bu isteği servera iletir. Server kendi listesinden client'ı siler. Serverda client'a mesajlar göndermektedir. Örneğin yeni birisi sisteme girdiğinde server bütün clientlara mesaj göndererek bu kişiyi eklemesini ister. Server bir client dan mesaj geldiğinde, bunu göstermesi için tüm clientlara mesaj gönderir. Benzer biçimde disconnect işleminde serverdan clientlara mesaj gönderilmektedir.

O halde client-server uygulamalarda, client ile server arasındaki tüm haberleşme için hangi bilgilerin gönderilip alınacağı baştan belirlenmelidir. Mesajlar genellikle değişken uzunlukta olma eğilimindedir.

Client dan servera, serverdan clienta gönderilen mesajlar için genel olarak sabit uzunlukta bir başlık kısmı ve değişken uzunlukta bir içerik kısmı bulundurulabilir. Örneğin:



Bu durumda client yada server önce porttan recv fonksiyonu ile başlık kısmı kadar bilgiyi (örneğin 8 bayt) okur. Başlık kısmındaki uzunluğa bakarak porttan o kadar bilgiyi daha okur. Böylece tüm mesajı elde etmiş olur. Mesaj verisinin formatı mesajın ID'sine bağlıdır. Rahat bir çalışma için her mesaja karşılık bir yapının bulundurulması uygundur. Mesaj başlığı gibi bir yapı ile temsil edilebilir:

```
typedef struct tagMSGHEADER {
    DWORD id;
    DWORD length;
} MSGHEADER;
```

Şimdi gönderilen tüm mesajlara birer ID verelim:

```
enum MSGIDS {
    MSG_CTS_CONNECT,
    MSG_CTS_SENDDMESSAGE,
    MSG_CTS_DISCONNECT,
    MSG_STC_CONNECT,
    MSG_STC_DISCONNECT,
    MSG_STC_SENDDMESSAGE,
    ...
};
```

Şimdi her mesaj için ayrı bir yapı oluşturulabilir. Örneğin client servera bağlanacaksa ona bir nick name verebilir. Şüphesiz büyük uygulamalarda çok karmaşık mesajlar söz konusu olabilir.

Şimdi de örnek oluşturmak amacıyla clientın servera gönderdiği mesajların serverda işlenmesine bir örnek verelim:


```

MSGHEADER msgHeader;
int result;

for (;;) {
    result = read_socket(client Socket, &msgHeader, sizeof(msgHeader));

    if (result == 0)
        break;

    switch (msgHeader.id) {
        case MSG_CTS_CONNECT:
            /***/
            break;
        case MSG_CTS_SENDMESSAGE:
            /***/
            break;
        case MSG_CTS_DISCONNECT:
            /***/
            break;
    }
}

```

Mesajın ID'si belirlendiğine göre önce veya sonra mesaj uzunluğu kadar bilgiyi soketten okuyarak işlevimizi yapabiliriz.

Bazen client ve server programları biz yazmayız da zaten mevcut bir sistem için client yazmak isteriz. Bunun için şüphesiz client ile server arasındaki tüm mesaj trafiği ayrıntıları ile bilinmelidir.

18.10. Çok Clientli Uygulamalar

Tipik bir server program birden fazla cliente hizmet verebilecek biçimde tasarlanır. Buradaki en önemli problem bir client ile uğraşırken bloke oluşması ve diğer clientlere hizmet verilememesidir. Örneğin server bağlandığı 10 clientin socket handlelarını bir dizide tutuyor olsun. Bir döngü içerisinde server 10 clientin socketinden de mesaj okumaya çalışırsa recv fonksiyonu blokeli modda beklemeye yol açabilir. Bu durumda bir clientteki bekleme diğer clientleri etkilemektedir. İşte bu problemi çözenin tipik olarak iki yolu vardır:

1. Akış oluşturma yöntemi: Bu yöntemde server blokeli modda çalışmaya devam eder, fakat her bir client bağlantısında ayrı bir akış oluşturarak (tipik olarak bir thread) o clientle o akış içerisinde konuşur. Böylece o akış bloke olsa bile diğer akışlar bu olaydan etkilenmez. Bu yöntem

basit olmasına karşın çok fazla sayıda client için kullanışsızdır. Çünkü her yeni client için bir akış oluşturulduğunda fazla miktarda sistem kaynağı kullanılmaktadır.

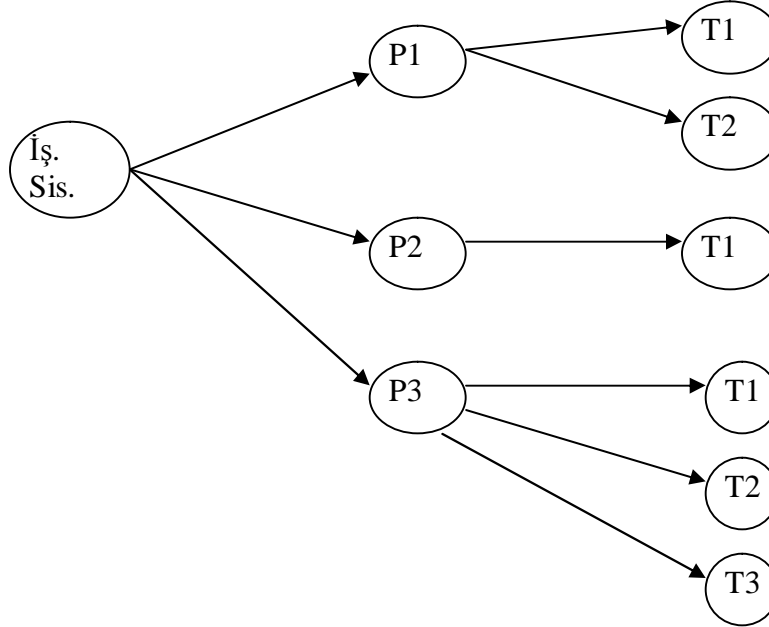
2. Etkin yoklama yöntemi: Bu yöntemde soketler blokeli moddan çıkartılır. Böylece send ve recv fonksiyonları blokeye yol açmazlar. Bu yöntemde server örneğin o döngü içerisinde sürekli 10 sokete recv fonksiyonunu uygular. Her client için ayrı bir tampon oluşturur. Hangi clientten bilgiyi okuduysa onun tamponuna bilgiyi ekler. Bilgi istediği noktaya geldiğinde işleme sokar. Fakat bu yöntem meşgul bir döngü (busy loop) oluşturarak CPU'yu gereksiz bir biçimde meşgul etmektedir. İşte işletim sistemleri bu tür durumlarda uygulanacak bazı yöntemler oluşturmuşlardır. Bu yöntemlerde en çok kullanılanlardan biri select modelidir. Select modelinde işletim sistemi tarafından sağlanan select isimli bir fonksiyon vardır. Select fonksiyonu bir grup soketi parametre olarak alır ve bunları izlemeye alır. Eğer hiçbir sokete bilgi gelmemişse select threadi çizelge dışına çıkartır. Böylece CPU zamanı harcamadan etkin bekleme durumu oluşur. Soketlerden en az birine bilgi geldiğinde işletim sistemi bunu fark eder ve threadi yeniden çizelgeye yerleştirir ve select fonksiyonu sonlanır. Select fonksiyonundan çıkıldığında programcı hangi sokete bilgi geldiğinden dolayı fonksiyondan çıkıldığını sorgular. Hangi sokete bilgi geldiyse o sokete recv fonksiyonu uygulanır ve yeniden select fonksiyonuna girilir.

Windows sistemlerinde select modelinden ziyade asenkron modeller daha çok tercih edilmektedir. Örneğin "IO Completion port" denilen asenkron yöntem çok tercih edilmektedir ve .NET ve Java platformları arka planda modern Windows sistemlerinde bu mekanizmayı kullanmaktadır. Bu yöntemde programcı işlevi başlatır. Hem akış devam eder, hem de arka planda işletim sistemi işlemi yapar. İşlem bitince işletim sistemi programcının belirlediği bir fonksiyonu çağırır.

19. THREADLER

Bir prosesin aynı adres alanındaki farklı akışlarına thread denilmektedir. Threadli sistemler ilk olarak 80'li yıllarda gerçekleştirilmeye başlanmış fakat ciddi anlamda 90'lı yıllarda yaygınlaşmıştır. Bugün Windows, Unix/Linux, MacOSX gibi sistemlerin hepsi thread özelliğine sahiptir. Bu sistemlerde her proses bir veya birden fazla thread akışından oluşur. Proses çalışmaya tek bir akışla yani threadle başlar. Bu ana thread main fonksiyonunun başladığı akıştır. Diğer threadler çalışma zamanı sırasında fonksiyonlarla oluşturulur. Her thread akışı sanki bir bağımsız bir programmış gibi işletim sistemi tarafından diğerlerinden bağımsız çizelgelenmektedir.

Windows ve Linux işletim sistemleri thread temelinde çizelgeler uygulamaktadır. Bu sistemde thread hangi sisteme ilişkin olursa olsun çizelgeleme bakımından sistemde prosesler yoktur, threadler vardır. Örneğin:



Her bir kuantum süresi bittiğinde çalışmakta olan thread'e ara verilir, bekleyen diğer thread'e geçilir. Geçilen thread ya aynı prosesin threadidir ya da farklı bir prosesin threadidir. Genel olarak aynı prosesin threadleri arasındaki geçiş daha hızlıdır.

19.1. Threadli Çalışmanın Avantajları

1. Arka planda yoklama yöntemiyle izlenmesi gereken olayların söz konusu olduğu durumlarda threadler organizasyonu çok kolaylaştırmaktadır. Örneğin hem bir porttan gelen bilgileri hem alıp hem de başka bir şeyleri yapacağımızı düşünelim ya da örneğin programımız ekranın sağ üst köşesine canlı bir saat çıkaracak olsun. İşte threadli sistemlerde bir thread oluşturulur, bu arka plan olaylar threade bırakılır, diğer akış normal bir biçimde devam eder.

2. Threadler, blokedan diğer akışların etkilenmemesi için yoğun olarak kullanılmaktadır. Örneğin on soketten bilgi okumak için threadli model kullanılabilir. Böylece recv fonksiyonu bir sokette bloke olsa bile diğer threadler çalışmaya devam etmektedir.

3. Threadler paralel programlama ya da dağıtık algoritmaların gerçekleştirilmesi amacıyla da kullanılmaktadır. Örneğin biz bir işin bağımsız parçalarını, seri olarak değil de aynı anda paralel olarak yaptırabilirsek daha hızlı bir çalışma söz konusu olur. CPU'dan çektiğimiz toplam zaman artacaktır. Şüphesi tek CPU'lu ya da tek çekirdekli sistemlerde gerçek bir paralel çalışma söz konusu olmaz. Fakat yine de bir hız kazancının sağlanacağı aşikârdır.

19.2. Threadlerin Bellek Alanları

Bir prosesin tüm threadleri aynı veri ve heap alanını kullanmaktadır. Yani örneğin bir thread global bir değişkene bir değer yazsa, diğer thread bu değeri görebilir. Fakat threadlerin stackleri birbirinden ayrılmıştır. Her bir threadin ayrı bir stack'i vardır. Yerel değişkenler ve parametre değişkenleri stack'de oluşturulduğuna göre her thread akışı yerel değişkenlerin farklı bir kopyasını kullanıyordur. Örneğin 2 farklı thread aynı fonksiyon üzerinde ilerlerken değişkenler birbirine karışmaz. Kod aynıdır. Fakat her thread yerel değişkenlerin o threade özgü kopyasını kullanmaktadır.

Şüphesiz statik yerel değişkenlerde data bölümünde oluşturulduğu için threadler arasında tıpkı global değişkenler gibi paylaşılır.

19.3. Threadler Üzerinde İşlemler

Thread mekanizması işletim sisteminin proses yöneticisinin kontrolü altındadır. Modern işletim sistemlerinde thread oluşturmak için ve threadleri yönetebilmek için çeşitli sistem fonksiyonları vardır. Örneğin Windows işletim sisteminde Kernel32.dll içerisinde bulunan bir grup API fonksiyonu ile thread işlemleri yapılmaktadır. Linux sistemlerinde threadler yine klasik fork işlemi ile oluşturulur. Fakat fork işlemi sırasında bellek kopyalaması yapılmaz. Böylece adeta aynı bellek alanında çalışan iki farklı proses biçiminde thread oluşturulmaktadır (lightweight process). Fakat Linux sistemlerinde, Windows'ta olduğu gibi sistem fonksiyonlarını doğrudan çağırarak thread işlemlerini yapmak çok zor olduğu için bu sistemlerde thread işlemleri için çeşitli kütüphaneler kullanılmaktadır. pthread kütüphanesi diye bilinen kütüphane çok yaygındır ve POSIX standartları tarafından desteklenmektedir.

19.4. Windows Sistemlerinde Thread İşlemleri

Windows sistemlerinde thread işlemleri için API fonksiyonları kullanılmaktadır. Windows API fonksiyonlarının büyük bölümünün prototipleri windows.h içerisinde bulunmaktadır.

19.4.1. CreateThread Fonksiyonu

Thread yaratmak için CreateThread API fonksiyonu kullanılmaktadır. Fonksiyonun prototipi içerisinde windows.h ta typedef edilmiş olan bir çok tür ismi bulunmaktadır.

```
HANDLE CreateThread {
    LPSECURITY_ATTRIBUTES lpThreadAttributes,
    SIZE_T dwStackSize,
    LPTHREAD_START_ROUTINE lpStartAddress,
    LPVOID lpParameter,
    DWORD dwCreationFlags,
    LPDWORD lpThreadId
};
```

Fonksiyonun birinci parametresi thread kernell nesnesinin güvenlik bilgilerini belirtir. Bu parametre security attributes denilen bir yapının adresini alır. Bu parametre NULL geçilebilir. Bu

durumda güvenlik uygulanmaz. Fonksiyonun ikinci parametresi threadin stack uzunluğunu belirtmektedir (SIZE_T unsigned int biçiminde typedef edilmiştir.). Bu parametre 0 geçilirse, bu durumda PE (portable executable) formatında belirtilen stack uzunluğu alınır. PE formatını linker oluşturduğu için bu değer linker seçeneklerinden ayarlanabilmektedir. Microsoft linkerları default olarak 1MB değerini yazmaktadır. O halde bu parametreyi 0 geçerse default stack 1MB uzunluğunda olacaktır.

Fonksiyonun üçüncü parametresi thread akışının başlatılacağı fonksiyonun adresini almaktadır. LPTHREAD_START_ROUTINE aşağıdaki gibi bir typedef ismidir.

```
typedef DWORD (__stdcall * LPTHREAD_START_ROUTINE)(void *);
```

Yani thread fonksiyonunun geri dönüş değeri “DWORD”, parametresi “void *” olmak zorundadır.

Anahtar Notlar:

windows.h içerisinde BYTE, WORD, DWORD sırasıyla, 1,2,4 baytlık işaretli tamsayı türlerini belirtir. Türün önüne p yada lp eki getirilirse adres türü anlaşılır. pc ya da lpc gösterdiği yer const olan adrestir. Başlı h ile başlayan typedef isimleri void * anlamına gelmektedir.

Anahtar Notlar:

Fonksiyonun geri dönüş değeri ile ismin arasına getirilen anahtar sözcüklere fonksiyon belirleyicileri (function specifiers) denilmektedir. Standartlarda fonksiyon belirleyicileri diye bir konu yoktur. Fonksiyon belirleyicileri eklenti biçiminde bulunmaktadır. __stdcall, __pascal, __cdecl Intel sisteminde pek çok derleyicinin kullandığı belirleyicilerdir. Bu belirleyicilere fonksiyon çağırma biçimi (call convention) denilmektedir. Microsoft derleyicilerinde default durum __cdecl biçimindedir. Yani hiçbir şey yazmamakla __cdecl yazmak aynı anlamdadır. Thread fonksiyonlarının __stdcall çağırma biçimine sahip olması zorunludur.

CreateThread fonksiyonunun 4. parametresi thread fonksiyonuna gönderilecek parametredir. Fonksiyonun 5. parametresi olan dwCreationFlags yaratılan thread ile ilişkin bazı belirlemelerin yapılması için kullanılır. Örneğin bu parametre CREATE_SUSPENDED biçiminde geçilebilir. Bu durumda thread yaratılır fakat ResumeThread fonksiyonu uygulanana kadar çalışmaz. Bu parametre 0 olarak geçilebilir. Bir threadin handle ve ID değeri vardır. CreateThread threadin handle değeri ile geri döner. İşte fonksiyonun son parametresi DWORD bir nesnenin adresini alır, ID değeri oraya yerleştirir. Bazı fonksiyonlar bizden handle değerini bazı fonksiyonlar ise ID değerini ister.

CreateThread fonksiyonu, başarı durumunda threadin handle değerine, başarısızlık durumunda NULL değerine geri döner.

Anahtar Notlar:

Konsol ekranının herhangi bir yerine cursor den bağımsız olarak yazıyı basan fonksiyon şöyle yazılabilir:

```
void Writes(int row, int col, const char *str)
{
    HANDLE hStdout = GetStdHandle(STD_OUTPUT_HANDLE);
    DWORD dw;
    COORD coord;

    coord.X = col;
    coord.Y = row;

    WriteConsoleOutputCharacter(hStdout, str, strlen(str), coord, &dw);
}
```

19.4.2. Threadlerin Sonlandırılması

Bir threadin sonlandırılması çeşitli biçimlerde gerçekleştirilebilir:

1. Thread fonksiyonu sonlandığında threadte sonlanmış olur. Herhangi bir thread akışı ExitThread API fonksiyonu ile çağrılırsa thread sonlanır.

```
void WINAPI ExitThread(DWORD dwExitCode);
```

Ana threadin diğer threadlerden hiçbir ayrıcalığı yoktur. Yani ana thread sonlandığı halde diğer threadler çalışmaya devam edebilir. Fakat bir prosesdeki son threadte sonlandığında proses otomatik olarak sonlandırılır.

2. Prosesi sonlandıran standart exit fonksiyonu Windows sistemlerinde ExitProcess sistem fonksiyonunu, Unix/Linux sistemlerinde _exit sistem fonksiyonunu çağırarak sonlandırma yapmaktadır. Bir proses sonlandığında tüm threadlerde otomatik olarak sonlandırılmaktadır. Main fonksiyonu bittiğinde exit işlevi uygulandığı için bütün threadlerde sonlanacaktır.

Anahtar Notlar:

Sample.c kaynak dosyasını bir projeye ekleyip build işlemi yapmış olalım. Sample.c dosyası derlenerek sample.obj haline getirilecek ve link işlemi yapılacaktır. Fakat derleyici link aşamasında sample.obj dosyası yanı sıra ismine start up module denilen ve genellikle crt0.obj biçiminde bulunan bir dosyayı da link işlemine sokmaktadır. Gerçek programın başlangıç noktası bu dosyanın içerisinde ve main fonksiyonu da aslında buradan çağrılmaktadır. Yani C programında ilk çalıştırılan kod main değildir. Derleyicinin başlangıç kodudur.

```
/******
```

```
*****  
*****/  
call _main  
call _exit  
/*****  
*****  
*****/
```

Derleyicinin başlangıç kodu gerekli pek çok işlemi yapmaktadır. Örneğin komut satırı argümanları, bu başlangıç kodu tarafından main fonksiyonuna aktarılmaktadır.

Anahtar Notlar:

Main fonksiyonunda hiç return uygulanmazsa main fonksiyonuna özgü olarak 0 ile return edildiği varsayılır. Ayrıca standartlarda main fonksiyonundan elde edilen geri dönüş değeri ile exit fonksiyonunun çağrılacağı söylenmiştir. Yani main fonksiyonunda n değeri ile return etmek ile exit(n) çağırması aynı sonuca yol açmaktadır.

3. Bir thread başka bir threadi TerminateThread fonksiyonu ile sonlandırır.

```
BOOL WINAPI TerminateThread(HANDLE hThread, DWORD dwExitCode);
```

Fonksiyonun birinci parametresi threadin handle değeri, ikinci parametresi çıkış kodudur. Fakat threadlerin bu şekilde sonlandırılması tavsiye edilmez.

19.5. Threadlerin Senkronizasyonu

Thread konusunun önemli bir bölümünü senkronizasyon konusu kaplamaktadır. Çünkü çok threadli uygulamalar yaparken threadlerin birbirlerini beklemesi gerekebilmektedir.

19.5.1. Kritik Kodların Oluşturulması

Başından sonuna kadar tek bir thread akışı tarafından çalıştırılması gereken kodlara kritik kodlar denilmektedir. Bir thread bir kritik koda girdiğinde başka bir thread bu kritik koda ya da bu kritik kodun ilişkin olduğu bir koda girmemelidir. Thread akışı kritik kodayken threadler arası geçiş oluşabilir. Fakat başka bir thread önceki thread çıkana kadar bu koda girmemelidir.

Kritik kodlar ortak kaynaklara erişirken sıklıkla kullanılmaktadır. Örneğin iki thread aynı bağlı listeye ekleme yapıyor olsun. Ekleme sırasında düğümler birbirine bağlanırken threadler arası geçiş oluşabilir. Başka bir threadte ekleme ya da silme gibi işlemleri yapmaya çalışırsa bağlı liste kararsız durumda olduğu için sorunlar oluşabilir. Ekleme yapan thread işlemini bitirmeden

diğer threadler bu bađlı listeyi kullanmaya alıřmamalıdır. Yada rneđin rand gibi bir fonksiyon global bir deđiřkeni kullanmaktadır. İki threadte bu fonksiyonu ađırırrsa yanlıř deđerler oluřabilir. Grldđđđ gibi ortak kaynaklara eriřimde, eriřimlerin seri hale getirilmesi paralel yapılmaması gerekmektedir.

Windows'ta kritik kod oluřturmak iin InitializeCriticalSection, EnterCriticalSection, LeaveCriticalSection, DeleteCriticalSection fonksiyonları kullanılır. Bu fonksiyonların kullanımı řoyledir:

1. CRITICAL_SECTION trnden bir nesne tanımlanır. rneđin:

```
CRITICAL_SECTION g_cs;
```

2. Programın bařında InitializeCriticalSection fonksiyonu ile bu nesneye ilk deđerleri verilir.


```
void InitializeCriticalSection(LPCRITICAL_SECTION lpCriticalSection);
```

3. Fonksiyon parametre olarak CRITICAL_SECTION nesnesinin adresini alır. rneđin:

```
InitializeCriticalSection(&g_cs);
```

4. Kritik kod řoyle oluřturulur:

```
EnterCriticalSection(&g_cs);  
/*****/  
LeaveCriticalSection(&g_cs);
```



Bir thread akıřı EnterCriticalSection fonksiyonuna girdiđi zaman bařka bir akıř fonksiyona girmeye alıřırrsa bloke olarak izelge dıřına ıkartılır ve bekleme durumuna geer. Ta ki kritik koddaki thread LeaveCriticalSection fonksiyonunu ađırana kadar. Birden fazla thread EnterCriticalSection fonksiyonunda bekliyorsa kritik koddaki thread, kritik koddan ıkınca hangisinin gireceđi konusunda resmi bir bilgilendirme yapılmamıřtır.

```
void EnterCriticalSection(LPCRITICAL_SECTION lpCriticalSection);  
void LeaveCriticalSection(LPCRITICAL_SECTION lpCriticalSection);
```

Nihayet tm iřlemlerin sonunda DeleteCriticalSection fonksiyonu ađırılmalıdır.

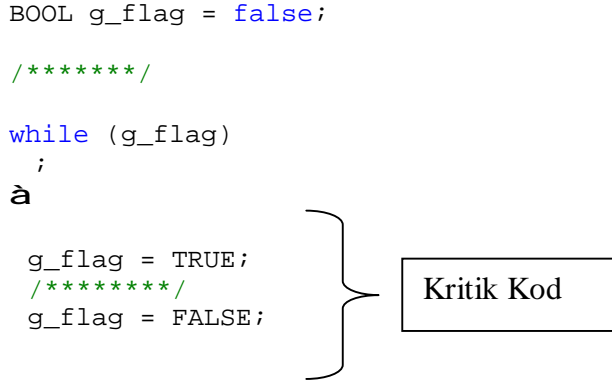
```
void DeleteCriticalSection(LPCRITICAL_SECTION lpCriticalSection);  
DeleteCriticalSection(&g_cs);
```

Kritik kod uygulaması manüel yolla yapılamaz. Örneğin aşağıdaki gibi bir kritik kod oluşturmak isteyelim.

```
BOOL g_flag = false;

/*****/

while (g_flag)
{
    g_flag = TRUE;
    /*****/
    g_flag = FALSE;
}
```



Ok ile belirtilen noktada threadler arası geçiş oluşursa başka bir thread kritik koda girebilir.

Peki, EnterCriticalSection ve LeaveCriticalSection gibi fonksiyonlar bu mekanizmayı nasıl sağlamaktadır? İşte bu tür yöntemler genellikle kernel moda geçilerek yada geçilmeden uygulanabilmektedir. Örneğin biz flag kontrolünü yapmadan önce kesme oluşumunu özel makine komutları ile engellersek flag işlemini güvenli bir biçimde yapabiliriz. Fakat bu özel makine komutlarını kullanabilmek için kernel moda geçmek gerekir. Fakat pek çok modern işlemcide bu tür işlemleri atomik yapabilmek için test ve set işlemlerini birlikte yapıp jump eden özel makine komutları vardır.

Anahtar Notlar:

Preemptive bir işletim sisteminin kernel yapısı, preemptive veya non-pre-emptive olabilir. Pre-emptive kernel demek bir sistem fonksiyonu çağrılıp kernel moda geçildiğinde de threadler arası geçiş oluşabilir demektir. Yani akış kernel modda iken kuantal süresi dolduğunda yine threadler arası geçiş uygulanmaktadır. Fakat non-preemptive sistemlerde, kernellarda akış kernel modda iken çıkana kadar kesilme yapılmamaktadır. Non-preemptive kernel tasarımı daha kolaydır. Çünkü pek çok kernel veri yapısının senkronize edilmesine gerek kalmaz. Örneğin Linux işletim sistemi 2.6'ya kadar non-preemptive bir kernel a sahipti, 2.6 ile büyük ölçüde pre-emptive yapılmıştır.

Anahtar Notlar:

Kernel mimarisi için bir çizginin iki ucunda mikro kernel ve monolitik kernel tasarımları söz konusudur. Tabii bir kernel bu arada bir yerlerde olabilir. Mikro kernel mimaride, kernel minimal düzeyde çok temel işlemleri yapacak biçimde tasarlanır, kernelin diğer işlevleri isteğe bağlı olarak yüklenen modüller ile gerçekleştirilir. Monolitik yapıda kernel tek parça büyük bir dosya halindedir. Boot işlemi sırasında belleğe bütünsel olarak yüklenir.

Kritik kod içine alma işlemi ne kadar küçük bir kodu içerirse o kadar iyidir. Çünkü bu durumda bloke olma olasılığı azalır. Örneğin bir bağlı listeyi biz dışardan veya içerden senkronize edebiliriz. Dışardan senkronizasyonda fonksiyon çağrılarını senkronize edilir. Fakat bu durumda

geniş bir blok kritik kod içerisine alınmış olur. Halbuki bağlı liste fonksiyonlarının içerisinde tam düğüm bağlantıları yapılırken senkronizasyon uygulanabilir. Bu durumda bağlı liste kendiliğinden thread güvenli olur. Fakat tek threadli uygulamalarda gereksiz bir senkronizasyon söz konusu olacaktır. Bu tür durumlarda sıklıkla aynı kütüphanenin tek threadli ve çok threadli versiyonları ayrı ayrı bulundurulmaktadır.

19.5.2. Windows Kernel Senkronizasyon Nesneleri

Windows sistemlerinde kernel moda geçerek çalışan bir grup senkronizasyon nesnesi prosesler arasında kullanılabilir. Bu senkronizasyon nesnelерinin hepsi WaitForxxx fonksiyonlarında beklenebilmektedir.

Kernel senkronizasyon nesneleri diğer işletim sistemlerinde de benzer biçimlerde bulunmaktadır. Örneğin Mutex, Semaphore ve Event nesneleri pek çok sistemde benzer biçimde bulunmaktadır.

19.5.3. WaitForSingleObject Fonksiyonu

WaitForSingleObject fonksiyonu senkronizasyon nesnelерini bekleyen genel bir fonksiyondur.

```
DWORD WaitForSingleObject(HANDLE hHandle, DWORD dwMilliseconds);
```

Fonksiyonun birinci parametresi senkronizasyon nesnesinin handle değeridir. İkinci parametre zaman aşımı (time out) parametresidir. Bir senkronizasyon nesnesinin iki durumu vardır. Nesne açık (signaled) durumda iken WaitForSingleObject fonksiyonu bekleme yapmaz. Nesne kapalı (non-signaled) iken, fonksiyon nesne açık duruma gelen kadar bekler. Açık ve kapalı durumlar beklenen nesneye bağlı olarak tanımlanmaktadır. Fonksiyon eğer zaman aşımından dolayı sonlanmışsa WAIT_TIMEOUT değerine, nesne açık duruma geldiğinden dolayı sonlanmışsa WAIT_OBJECT_0 değerine geri döner. Fonksiyon yanlış handle geçilmesi gibi nedenlerden dolayı başarısız olabilir. Bu durumda WAIT_FAILED değerine geri döner.

19.6. Event Nesnelerinin Kullanımı

Event nesneleri özellikle bir olay gerçekleşene kadar bekleme oluşturmaktadır. Tipik olarak üretici-tüketici problemlerinin çözümünde kullanılabilir. Event nesnesinin kullanımı şöyledir:

1. Event nesnesi CreateEvent fonksiyonu ile oluşturulur. Handle değeri global bir değişkene yerleştirilebilir.

```
HANDLE CreateEvent(  
    LPSECURITY_ATTRIBUTES lpEventAttributes,  
    BOOL bManualReset,  
    BOOL bInitialState,  
    LPCTSTR lpName  
);
```

Fonksiyonun birinci parametresi nesnenin güvenlik bilgilerini belirtir. NULL geçilebilir. İkinci parametre event nesnesinin manuel mi yoksa otomatik mi olduğunu belirtir. Bu parametre FALSE geçilirse otomatik, TRUE geçilirse maneldir. Üçüncü parametre nesnenin başlangıçta açık mı yoksa kapalı mı olacağını belirtir. Son parametre prosesler arası kullanım için gereken ismi belirtir. Eğer aynı prosesin threadleri arasında senkronizasyon uygulanacaksa bu parametre NULL geçilebilir. Fonksiyon başarı durumunda event nesnesinin handle değerine başarısızlık durumunda NULL değerine geri döner.

2. Event nesnesi WaitForSingleObject fonksiyonu ile beklenir.

3. SetEvent fonksiyonu ile event nesnesi açık duruma geçirilebilir.

```
BOOL SetEvent(HANDLE hEvent);
```

Fonksiyon Event nesnesinin handle değerini parametre olarak alıp onu açık duruma getirmektedir.

4. Nihayet işlem bitince Event nesnesi CloseHandle fonksiyonu ile yok edilir. Fonksiyon Event nesnesinin handle değerini parametre olarak almaktadır.

```
BOOL CloseHandle(HANDLE hObject);
```

Event nesnesi otomatik modda ise WaitForSingleObject fonksiyonu ile nesnenin açık duruma gelmesinden dolayı geçiş yapıldığında otomatik olarak nesne yeniden kapalı duruma geçer. Aksi halde nesneyi kapalı duruma geçirmek için ResentEvent fonksiyonu uygulanmalıdır.

19.7. Üretici Tüketici Problemi

Üretici-tüketici problemi (producer-consumer problem) çok threadli uygulamalarda ve prosesler arası haberleşmelerde sık gereksinim duyulan bir durumu belirtmektedir. Bu problemde threadlerin biri bir değer elde eder ve bunu bir ortak kullanılan nesneye yazar. Bu threade üretici denilmektedir. Diğer thread ise yazılan değeri paylaşılan nesneden alarak kullanır. Buna da tüketici thread denilmektedir. Üretici ve tüketici threadler bunu döngü içerisinde birden fazla kez yapmaktadır. Örneğin bir thread bir asal sayı bulup bunu bir global bir değişkene yerleştirebilir, diğeri de bunu alarak kullanabilir ve bu işlem böyle devam edebilir. Bu tür uygulamalarda üreticinin tüketici almadan yeni bir değeri yerleştirmemesi tüketicinin de üretici yeni bir değer koymadan almaya çalışmaması gerekir. Bu çok karşılaşılan klasik bir durumdur.

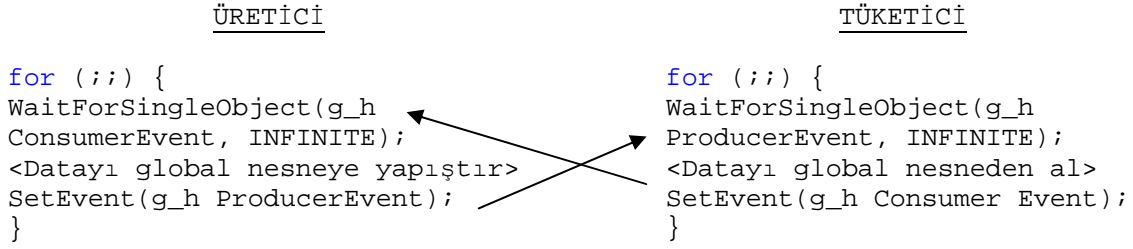
Üretici-tüketici probleminde paylaşılan alan yani tampon bölge bir elemanlık olabilir. Fakat daha genel bir biçimde tampon alan n'lik bir kuyruk sistemi olabilmektedir. Ayrıca bu problemin genel hali n*m 'lik olabilir. Yani yukarıda tipik olarak 1*1 lik ve 1 elemanlık durum açıklanmıştır. Örneğin 3 thread değer oluştururken, 5 thread değer almaya çalışabilir.

19.7.1. Üretici Tüketici Probleminin Event Nesneleri İle Çözümü

1*1 lik ve tek elemanlı üretici tüketici problemi 2 Event nesnesi kullanılarak çözülebilir. Ancak aynı problemin k elemanlık biçimi, yani paylaşılan bölgenin bir kuyruk sistemi olduğu biçimi Event nesneleri ile çözülemez. Tampon bölgenin k elemanlık olduğu genel biçim ancak semaforlar ile çözülebilir.

```
HANDLE g_h ConsumerEvent, g_h ProducerEvent;  
/***/  
g_h ConsumerEvent = CreateEvent(NULL, FALSE, TRUE, NULL);
```

```
g_h ProducerEvent = CreateEvent(NULL, FALSE, FALSE,
NULL);
```



Anahtar Notlar:

Threadlerin ve proseslerin kendisi de bir senkronizasyon nesnesi gibi kullanılabilir. Threadler yada prosesler çalışıyor durumda iken nesne kapalı durumdadır. Çalışma bittiğinde açık duruma geçer. Dolayısıyla bir thread sonlanana kadar şöyle bekleme yapılabilir.

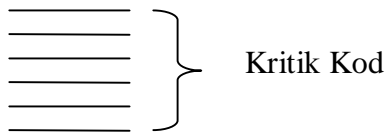
```
WaitForSingleObject(hThread, INFINITE);
```

Anahtar Notlar:

Windows kernel senkronizasyon nesnelerinin hepsi kernel modda çalışmaktadır. Yani kernel moda geçiş yaptığı için bir gecikme söz konusu olabilmektedir.

19.7.2. Semafor Nesneleri

Semafor nesneleri de pek çok işletim sisteminde benzer biçimde bulunmaktadır. Örneğin Linux sistemlerinde de semaforlar vardır. Semafor nesneleri bir kritik koda en fazla n tane akışın girmesini sağlayan sayaçlı nesnelerdir. Örneğin:



Daha önce kritik koda tek bir akışın girmesini sağlamıştık. Semaforlarla kritik koda birden fazla, fakat en fazla n tane akışın girmesi sağlanabilir. Örneğin kritik koda en fazla 3 tane akışın girmesini isteyelim. 1, 2, 3 numaralı akışlar kritik koda girerler. Ama bunlardan biri çıkmadan yeni bir akış kritik koda giremez.

Bir kritik koda en fazla n tane akışın girmesinin anlamı nedir? Kritik kodda n tane kaynaktan birinin tahsis edildiğini düşünelim. Her giren akış için bir kaynak tahsis edilir. Bu durumda yeni bir akışa kaynak kalmamışsa onun beklemesi gerekir.

Win32 sistemlerindeki semafor kullanımını şöyledir.

1. Semafor nesnesi CreateSemaphore fonksiyonu ile oluşturulur. Semafor oluşturulurken kritik koda kaç akışın gireceği belirtilir. Fonksiyon prototipi şöyledir:

```
HANDLE CreateSemaphore (  
    LPSECURITY_ATTRIBUTES lpSemaphoreAttributes,  
    LONG lInitialCount,  
    LONG lMaximumCount,  
    LPCTSTR lpName  
);
```

Fonksiyonun birinci parametresi nesnenin güvenlik bilgileridir. NULL geçilebilir. İkinci parametre kritik koda en fazla kaç akışın gireceğini bildiren sayaç parametresidir. Üçüncü parametre sayacın çıkabileceği maksimum değerdir ve genellikle ikinci parametreyle aynı değerde verilir. Son parametre prosesler arası paylaşım için gerekli olan isimdir. NULL geçilebilir. Fonksiyon başarı durumunda semaforun handle değerine, başarısızlık durumunda NULL değerine geri döner.

2. Kritik kod aşağıdaki gibi oluşturulur:

```
WaitForSingleObject(...)  
;  
/***/  
ReleaseSemaphore(...);
```

} Kritik Kod

Semafor nesnesi, semafor sayacı 0 dışındaysa açık, 0 ise kapalı durumdadır. WaitFor fonksiyonlarından her girişte semafor sayacı bir azaltılır. ReleaseSemaphore ise semafor sayacını artırır. ReleaseSemaphore fonksiyonunun prototipi şöyledir:

```
BOOL ReleaseSemaphore(  
    HANDLE hSemaphore,  
    LONG lReleaseCount,  
    LPLONG lpPreviousCount  
);
```

Fonksiyonun birinci parametresi semaforun handle değeri, ikinci parametresi artırım değeridir (Tipik 1 olarak verilir.). Semafor sayacının değeri hiçbir zaman CreateSemaphore da belirtilen değeri geçemez. Fonksiyonun son parametresi LONG bir nesnenin adresini alır ve sayacın önceki değerini oraya yerleştirir. Bu parametre NULL geçilebilir, bu durumda bir yerleştirme yapılmaz.

3. Kullanım bittikten sonra semafor nesnesi CloseHandle fonksiyonu ile kapatılır.

Anahtar Notlar:

Windows sistem programlama konusunda şu kaynaklara başvurulabilir:

- Jeffrey Richter'in içeriği benzer, fakat her baskısının ismi farklı olan kitapları (bu kitabın klasik versiyonu: Programming Applications for Microsoft Windows) Bu kitabın son versiyonu "Windows via C/C++" isimindedir.
- Matt Pietrek'in Windows 95 Programming Secret kitabıdır. Bu kitapta kernela ilişkin pek çok veri yapısı açıklanmaktadır.
- Windows System Programming (The Addison-Wesley Microsoft Technology Series)
- Solomon'un "Classic Windows Internals".
- Solomon'un eski sitesi sysinternals artık Microsoft tarafından alınmıştır. Burada çok sayıda makale bulunmaktadır. (www.sysinternals.com)
- Reactos Operating System'in sitesi open source Windows yazma projesidir. Dolayısıyla buradaki kaynak kodlar, Windows'un gerçek kodları hakkında iyi bir fikir verebilir. (www.reactos.org)
- Windows'un çalınmış kaynak kodları belirli alt sistemler hakkında bilgi vermektedir.

Anahtar Notlar:

<http://fxr.watson.org/> sitesinde kod inceleme yapılabilir.

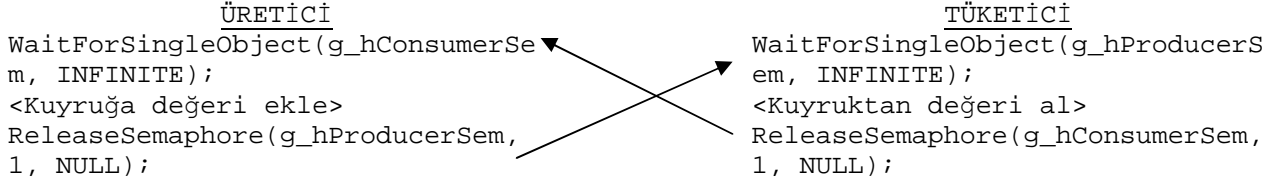
19.7.3. Çok Tamponlu Üretici-Tüketici Problemlerinin Semafor Nesneleri İle Çözülmesi

Bu tür üretici-tüketici problemleri en fazla karşılaşılanlardandır. Burada üreticinin bilgiyi yerleştireceği veri yapısı bir kuyruk biçimindedir ve kuyruğun belirli bir sınırı vardır. Üretici thread gelen bilgiyi kuyruğa yerleştirir, kuyruk dolduğunda blokede bekler. Tüketici threadte kuyruk tamamen boşsa bekler. Bu sistem üretici ve tüketicinin daha az beklemesine yol açtığı için şüphesiz daha verimlidir.

Bu çözümde tipik olarak bir üretici bir de tüketici için iki semafor oluşturulur. Üretici semaforunun başlangıç sayaç değeri n, tüketicinin 0 olacaktır. Üretici her WaitFor fonksiyonundan geçtiğinde kuyruğa yeni bir eleman yerleştirir. Böylece kuyruğa en fazla n tane eleman yerleştirebilir. Üretici ReleaseSemaphore ile tüketicinin, tüketicide üreticinin semafor sayacını artırır. Şüphesiz yine de üretici ve tüketici threadler aynı anda kuyruk sistemi üzerinde işlem yapıyor durumda olacaklardır. Kuyruk sisteminin ayrıca senkronize edilmesi gerekir.

Çok tamponlu yani kuyruk sistemli üretici-tüketici problemi şöyle çözülebilir:

```
HANDLE g_hConsumerSem, g_hProducerSem;  
g_hConsumerSem = CreateSemaphore(NULL, n, n, NULL);  
g_hProducerSem = CreateSemaphore(NULL, 0, n, NULL);
```

19.8. Mutex Nesneleri

Mutex (Mutual Exclusion) neredeyse her işletim sisteminde bulunan bir senkronizasyon nesnesidir. Mutex nesnesinin sahipliği bir thread tarafından alınabilir ve ancak o thread tarafından bırakılabilir. Pek çok sistemde Mutex nesneleri sayaçlıdır (Örneğin Windows sistemlerinde böyledir. POSIX sistemlerinde isteğe bağlıdır.). Yani aynı thread sahipliği birden fazla kez alabilir. Bu durumda birden fazla kez sahipliğin bırakılması gerekir.

Mutex kritik kod oluşturmak için kullanılabilir fakat “critical section” işlemlerinden farkı vardır. Sahiplik thread temelindedir. Bazı uygulamalarda bunun tipik kullanımları vardır.

Mutex aslında temel bir senkronizasyon nesnesidir ve bazı sistemlerde diğer senkronizasyon nesneleri dolaylı biçimde mutex nesneleri kullanılarak gerçekleştirilmektedir.

Mutex kullanımı tipik olarak şöyledir:

1. Mutex nesnesi CreateMutex fonksiyonu ile yaratılır.

```
HANDLE CreateMutex(
    LPSECURITY_ATTRIBUTES lpMutexAttributes,
    BOOL bInitial Owner,
    LPCTSTR lpName
);
```

Fonksiyon birinci parametresi kernel nesnesinin güvenlik bilgilerini belirtir ve NULL geçilebilir. İkinci parametre mutex nesnesinin sahipliğinin işin başında yaratan thread tarafından alınıp alınmayacağını belirtmektedir. Bu parametre TRUE ise alınır, FALSE ise alınmaz. Son parametre prosesler arasındaki paylaşımında kullanılan isimdir. Fonksiyon başarı durumunda mutex nesnesinin handle değerine, başarısızlık durumunda NULL değerine geri döner.

2. Mutex nesnesi eğer sahiplik alınmışsa açık durumda alınmamışsa kapalı durumdadır. Kritik kod şöyle oluşturulur:

```
WaitForSingleObject(...);
/****/
ReleaseMutex(...);
```

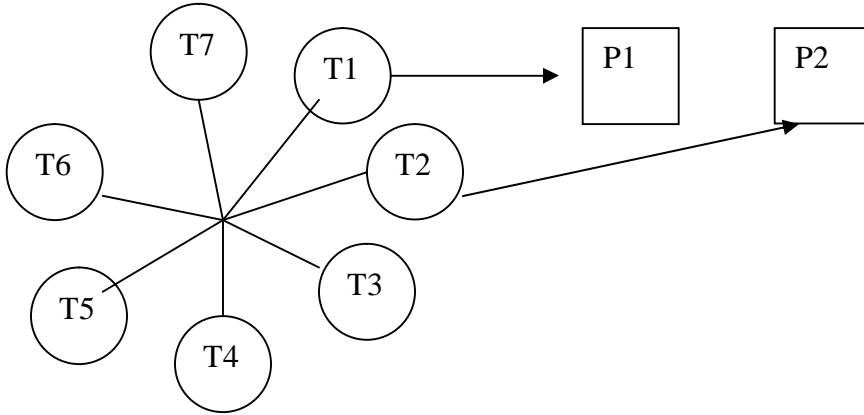
Eğer thread mutex nesnesinin sahipliğini almış durumdaysa WaitFor fonksiyonlarından geçer. ReleaseMutex mutex nesnesinin sahipliğini bırakır. Eğer mutex nesnesinin sahipliğini hiçbir thread almamışsa nesne yine açık durumdadır WaitFor fonksiyonu sahipliğini alarak geçer. Yani eğer mutexin sahipliğini başka bir thread almışsa WaitFor fonksiyonlarından geçiş yapılamaz. Fakat sahiplik boştaysa ya da sahipliği zaten almış durumdaysak geçiş yapılır. ReleaseMutex fonksiyonu ancak sahiplik ilgili thread tarafından alınmışsa sahipliği bırakabilir. Yani biz sahipliğini almadığımız bir mutexin sahipliğini bırakamayız.

```
BOOL ReleaseMutex(HANDLE hMutex);
```

3. CloseHandle fonksiyonu ile kullanımı bitince mutex nesnesinin sahipliği bırakılabilir.

19.9. Çok İşlemcili Çalışma

İşletim sistemleri birden fazla işlemciyi (çok çekirdekli sistemler birden fazla işlemcili sistemler gibi davranır.) thread temelinde çizelgelemektedir. Yani bir işlemciye bir thread atanmaktadır. Genellikle modern sistemler tek bir çizelge oluşturup belli kriterlere de bakarak sıradaki threadi boşalmış olan işlemciye atamaktadır. Örneğin iki çekirdekli bir Intel işlemcisini düşünelim ve sistemde 7 tane thread bulunuyor olsun. İşletim sistemi bir threadi bir işlemciye diğerini diğer işlemciye verir. Kuanta süresi bittiğinde hangi işlemci boşalmışsa sıradaki ona atar.



İşletim sistemi thread temelinde atama yaptığına göre hiçbir zaman bir T anında programımız birden fazla işlemci tarafından çalıştırılıyor durumda olmayacaktır. Tek threadli bir program çok işlemcili sistemlerde yine daha hızlı çalışabilir. Fakat bu birim zamanda toplam çalıştırılan kod miktarının artmasının doğal bir sonucudur. Biz programımızı çok threadli tasarlırsak gerçekten

aynı anda çalışma mümkün olabilir. Bu durumda işletim sistemi threadlerden birini bir işlemciye diğerini diğer işlemciye verebilir. Aslında çok işlemcili sistemlerde bir işlemci boşaldığında hangi threadin işlemciye atanacağı konusunda bazı ince hesaplar yapılmaktadır. Hemen sıradakinin atanması yerine bekleyen başka bir threadin atanması daha uygun olabilir. Örneğin bir thread daha önce işlemcilerin birine atanmışsa sonraki kuantada gene aynı işlemciye atanması içsel cache mekanizması yüzünden toplam performansı artırmak için daha uygun olabilir. Tabi bu faktörlerden yalnızca biridir. Başka faktörlerde vardır. O halde kabaca şunlar söylenebilir:

İşlemciye thread atanacağı zaman çizelgede bekleyen threadler gözden geçirilir ve en uygun olan seçilir. Şüphesiz döngüsel bir çizelgeleme söz konusu yine olabilir ve işletim sistemi mümkün olduğunca gene adil davranabilir. Örneğin kuyrukta çok beklemek seçilmek için bir avantajdır. Yani son ara verilmiş threadin o kadar thread beklerken yeniden ele alınması adil bir durum değildir.

Çok işlemcili sistemlerde bu işlemcilerin hepsi aynı RAM'e erişmektedir. Fakat bir işlemci RAM'e erişirken diğerini durdurmaktadır. Aslında erişimi mikro düzeyde aynı anda yapılmamaktadır. İki işlemci tek bir makine komutuyla aynı bellek bölgesine erişiyor olsun. Örneğin bir thread bir global bir değişkene bir değer yazarken diğeri okuyor olabilir. Erişimler tek bir makine komutuyla yapılıyor olsa bile bir makine komutunun işleyişinde baştan sona kadar diğer işlemcinin erişmesi engellenmemektedir. Makine komutu işletilirken nano saniyeler mertebesinde RAM'i tutup bırakabilmektedir. İşte kötü bir tesadüf ile bu sırada bir iç içe geçme oluşabilir. Bu anlatımlardan çıkan sonuç şudur: Çok işlemcili sistemlerin çoğunda farklı threadler aynı global değişkene erişiyorsa bu işlemler büyük bir tesadüfle farklı işlemciler tarafından aynı anda yapılırsa bozulmalar oluşabilir.

Anahtar Notlar:

Modern güçlü işlemcilerin bir özelliği de birbirinden bağımsız ardışık komutları belirtilen sırada değil de farklı sırada yaparak bundan bir avantaj sağlamaktır (instruction reordering). Fakat bu durum senkronizasyon konusunda çok işlemcili sistemlerde bazı problemlere yol açabilmektedir. Yine modern işlemcilerde "branch prediction" denilen gelişmiş özellikler vardır. Bu sayede komut çalıştırırken sonraki hangi komutlar için hazırlıklar yapılacağı tahmin edilmektedir.

Özetle çok işlemcili sistemlerde garanti program yazmak istiyorsak global değişkenlere erişirken yine bir çeşit senkronizasyon kullanmalıyız. Bu nedenle programda yanlış değerler oluşma olasılığı vardır fakat çok zayıftır. Global değişkenlere erişirken yukarda ele alınmış olan senkronizasyon nesnelere kullanmak işlemleri çok yavaşlatır. İşte bunlar için InterLock fonksiyonlar denilen özel fonksiyonlar bulunmaktadır. Intel işlemcilerinde ve pek çok işlemcide

Lock işlemcili makine komutu (prefix) sonraki makine komutu çalıştırılırken diğer işlemcileri durdurmaktadır. O halde global değişkenlere erişirken Lock gibi işlemleri kullanmak gerekir. İşte InterLock fonksiyonlar erişim sırasında bu gibi makine komutlarını kullanır.

Global bir değişkene erişmek için Windows sistemlerinde InterLockedxxx fonksiyonları kullanılmaktadır. Örneğin bir global değişkene değer atamak için InterlockedExchange fonksiyonu kullanılır.

```
LONG InterlockedExchange(  
    PLONG Target,  
    LONG Value);
```

Fonksiyonun birinci parametresi değer yerleştirileceği long bir nesnenin adresini alır. İkinci parametre yerleştirilecek değeri belirtmektedir. Fonksiyon geri dönüş değeri olarak değişkenin eski değerini vermektedir. Örneğin aşağıdaki gibi global bir değişkenimiz olsun.

```
long g_i;
```

Biz buna şöyle değer atamalıyız:

```
InterlockedExchange(&g_i, 100);
```

Bir değişkenin değerini bir artırmak için InterlockIncrement fonksiyonu kullanılabilir.

```
LONG InterlockIncrement(PLONG Adde);
```

Fonksiyon artırım yapılacak nesnenin adresini alır ve bunun değerini bir artırır.

Bir değişkenin içindeki n artırmak için InterlockedExchangeAdd kullanılmaktadır. Örneğin büyük bir global diziyi iki farklı thread çok işlemcili bir sistemde belli bir değerle doldurmak istesin.

```
long g_a[SIZE];
```

```
long g_i;
```

Biz burada diziyi atama veya artırım sırasında Interlocked fonksiyonlarını kullanmak zorundayız.

```
InterlockedIncrement(&g_i);
```

Ya da örneğin bir thread değişkeni set edip diğer threadten okunduğu durumda da benzer biçimde bir önlem alınmalıdır.

19.10. volatile Nesnelerin Çok Threadli Çalışmalarda Önemi

Global nesnelerin volatile anahtar sözcüğü ile bildirilmesi onların geçici süre yazmaçlarda tutulmasını engellemektedir. Standartlara göre volatile bir nesneye her erişildiğinde gerçek bir bellek erişimi yapılmak zorundadır. Örneğin threadlerden biri bir global değişkene bakarak döngüde işlem yapıyor olsun.

```
while (g_flag) {  
    //.....  
}
```

Biz başka bir threadten g_flag'e 0 atayarak bu döngüyü bitirmek isteyebiliriz. derleyici optimizasyonları tek akışa yönelik yapılmaktadır. yani derleyici programın tek akıştan oluştuğunu varsaymaktadır. Optimizasyon seçenekleri açıldığında derleyiciler bellek erişimini azaltmak için yazmaçları kullanabilirler. Örneğin:

```
for (;;) {  
    //*****  
    g_flag = ifade;  
    //-----  
}
```

```
Foo();
```

Burada derleyici g_flag değişkenini bir yazmaca çekip döngü içerisinde hep o yazmacı kullanabilir. Nihayet Foo fonksiyonu çağrılmadan önce yazmaçtaki değeri yeniden g_flag değişkenine yerleştirerek Foo fonksiyonunu çağırabilir ya da önceki örnekte while içerisinde g_flag değişkenine bakmak yerine, g_flag içerisindeki değeri yazmaca çekip aslında yazmaca bakan kod üretebilir. Çünkü derleyici default olarak threadli bir çalışmayı göz önüne almamaktadır. İşte biz C'de bu tür global değişkenleri her ihtimale karşı volatile bildirmeliyiz.

```
volatile int g_flag;  
  
while (g_flag) {  
  
}  
  
for (;;) {  
    //*****  
    g_flag = ifade;  
    //-----  
}
```

Tabi volatile olmamasından dolayı Derleyicimizi tanıyorsak ya da kod optimizasyonu yapabiliyorsak bir sorun ortaya çıkmaz.

19.11. Windows Sistemlerinde Thread Çizelgesi

Windows sistemlerinde kuanta süresi 20 msn'den 120 msn'ye ye kadar çeşitli etkenlere bağlı olarak değişmektedir. Bu konudaki ayrıntılar Microsoft tarafından dökümanite edilmemiştir. Server grubu sistemlerde kuanta süresi 60 ms yada 120 msn civarındadır. 95–98 grubu sistemlerde 20 msn civarındadır. (Linux sistemlerinde tipik 60 msn biçimindedir.)

Windows sistemleri thread çizelgesi için öncelik sınıfları temelinde döngüsel çizelgeleme yöntemini kullanmaktadır. Windows sistemlerinde her threadin 0–31 arası bir öncelik derecesi vardır.

Anahtar Notlar:

Threadin öncelik derecesi Windows sistemlerinde iki değerin toplamı ile elde edilmektedir. Değerlerden biri threadin içinde bulunduğu prosesin öncelik sınıfıdır. Proseslerin öncelik sınıfı bir taban değer belirtir. Bu taban değere threadin görelî öncelik derecesi eklenmektedir. Prosesin öncelik sınıfı SetPriorityClass API fonksiyonu ile değiştirilebilir. Threadin görelî öncelik derecesi ise SetThreadPriority Fonksiyonu ile belirlenmektedir. Bu çizelgeleme yöntemine göre her farklı thread önceliği için bir öncelik grubu oluşturulur ve sistem sanki diğer threadler yokmuş gibi en yüksek öncelikli grubu döngüsel çizelgeler. Bu gruptaki tüm threadler bitirse bir sonraki öncelik grubuna geçilir. Örneğin çizelgede aşağıdaki önceliklere ilişkin öncelikler bulunsun:

18 18 18 17 16 16 9 8 8 8 8 4 0

Burada önce döngüsel öncelikteki 3 thread döngüsel çizelgelenir. Bu threadlerin hepsi bitir veya bloke olursa bu kez 17 öncelikli thread çizelgede tek başına yer alır. 17 de bloke olursa bu kez 16'lı grup çizelgelenir. Bu biçimdeki çalışma adaletsizmiş gibi gelebilir. İlk anda düşük öncelikli threadlerin hiç çalışmayacağını sanabiliriz. Fakat ne olursa olsun blokelere dolayı düşük öncelikli threadlerde çalışma şansı bulabilmektedir. Windows sistemlerinde 0 öncelikli bir thread işletim sisteminin Idle threadidir ve hiçbir işlem yapılmıyorsa devreye girer. Windows'ta default öncelik derecesi 8'dir. Yüksek öncelikli threadler bloke olduğunda düşük öncelikli threadler çalışma şansı bulabilir. Fakat bu durumda yüksek öncelikli bir threadin blokesi çözüldüğünde hemen çalışma yüksek öncelikli thread'e verilir. Bir thread arada sırada birşeyler yaptığı halde yüksek bir öncelik derecesine sahip olabilir. Böylece thread etkileşime çabuk geçer ve yaptığı işi de hızlı yapar.

Ayrıca Windows sistemlerinde “priority boosting” denilen bir mekanizma da kullanılmaktadır. Bu mekanizma ana hatlarıyla açıklanmıştır fakat ayrıntılar dokümanite edilmemiştir. Bu mekanizmaya göre bir pencere aktif hale getirildiğinde bu tür prosesler ve threadler belirli bir kuanta için öncelik yükseltmesine sokulur. Sonra önceliği programcının belirlediği duruma getirilir. Böylece kullanıcının etkileştiği pencereler ilişkin threadler daha etkin olur.

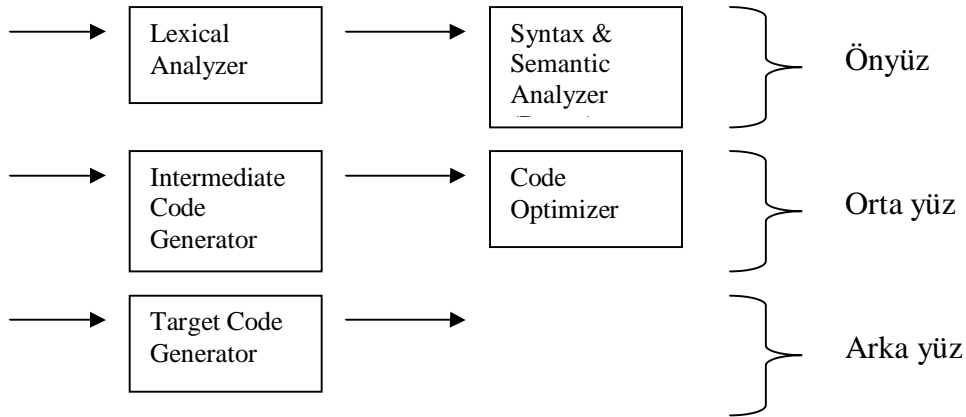
Anahtar Notlar:

İşletim sistemine ilişkin bir kod kaç biçimde çalışma fırsatı bulabilir? Tipik senaryolar şunlardır.

1. Çeşitli donanım kesmeleri oluştuğunda işletim sisteminin kodları çalıştırılmaktadır.
2. İşletim sisteminin de tıpkı bizim sıradan proseslerimizin threadleri gibi hiç bitmeyen kernel threadleri vardır. Kernel threadler çizelgeleme bakımından sıradan threadlerle aynı biçimde çalıştırılır. Çoğu kez arada bir devreye girer ve bazı kritik işlemleri yapar.
3. Sistem fonksiyonları ya da aygıt sürücüler yoluyla işletim sistemlerinin kodları çalıştırılabilir.
4. İşletim sistem boot edilirken zaten işletim sistemi kodlar çalıştırılarak sistem açılmaktadır. Çizelgeleme konusunda işletim sistemini temsil eden ortadaki yuvarlakçık timer kesmesi ile devreye giren işletim sisteminin çizelgelemesini gerçekleştiren kernel kodlarını belirtmektedir.

20.DERLEYİCİLERİN KOD OPTİMİZASYONU

Modern derleyiciler derleme sürecinde çeşitli iyileştirmeler yaparak kodun daha iyi çalışmasını sağlamaktadır. Bu tür derleyicilere kod optimizasyonu yapan derleyiciler (optimizing compilers) denilmektedir. Kod optimizasyonu çeşitli aşamalarda gerçekleştirilebilir. Tipik olarak ara kod üretiminden sonra önemli optimizasyonlar ara kod üzerinde gerçekleştirilir. Ayrıca kod üretimi aşamasında da hedef mimariye ilişkin optimizasyonlar yapılmaktadır.



Kod optimizasyonu her ne kadar ara kod üretiminden sonra tek bir modül biçiminde gösterilmişse de sentaks analizinden sonra pars tree üzerinde ve hedef kod üretimi sırasında da ciddi optimizasyonlar yapılmaktadır. Fakat gerçekten de klasik optimizasyonların büyük bölümü ara kod üretiminden sonra gerçekleştirilir.

Optimizasyon kodun eşdeğerliği bozulmadan yapılan bir faaliyettir. Yani yazdığımız programın anlamı değişmez. Kod daha etkin çalışacak biçimde üretilir.

Kod optimizasyonu çeşitli tipik kategorilere ayrılarak ele alınabilir. Döngüler en önemli optimizasyon kaynağıdır. Yani döngülere yoğunlaşılması önemlidir çünkü program yaşamının büyük bir çoğunluğu döngülerde geçer.

Optimizasyon işlemi hız ve kaynak kullanımı ölçütlerine bağlı olarak yapılabilir. Yani kod daha hızlı çalışacak biçimde ya da daha az yer kaplayacak biçimde optimize edilebilir. Ancak hız optimizasyonu çok daha baskın olarak tercih edilmektedir. Önemli optimizasyon temaları aşağıda açıklanmaktadır.

20.1. Ölü Kod Eliminasyonu (Death Code Elimination)

Bu optimizasyon genellikle kodu küçültme hedefindedir. Derleyiciler erişilemeyen kodları elimine ederek bunlar içinde gereksiz biçimde kod üretmezler. Örneğin:

```
void Foo(void)
{
    //*****

    return;
    a = 10;    à Ölü kod
    b = 20;    à Ölü kod
}
```

Burada akış return deyiminden ileri geçmeyeceğine göre bu atamaların bulunmasına hiç gerek yoktur. Pek çok C derleyicisi bu durumlarda “unreachable code” biçiminde uyarı da vermektedir.

Bir değişkene bir değer atandığını fakat o değer hiç kullanılmadığını düşünelim. Bu da ölü kod statüsündedir. Örneğin:

```
{
    int a = 10;

    //*****
}
```

Burada derleyici eğer a hiç kullanılmadıysa a için yer ayırmayabilir ya da a'ya atama yapmayabilir. C derleyicilerinin çoğu yine bu durumda uyarı vermektedirler (a declared not used).

Bazen ölü kodlar deyimlerin yanlış kullanılmasıyla da oluşabilir. Örneğin:

```
if (0)
    ifade;
```


Burada ifade asla çalıştırılmayacaktır. Elimine edilebilir. Ya da örneğin:

```
char ch = 0xFC;

if (ch == 0xFC) {
    /*****/
}
```

Derleyici “if always false” gibi bir uyarı verebilir. Eğer char türü default işaretliyse bu ifade hiçbir zaman doğru olmaz. Ya da örneğin:

```
short a;

if (a > 150000) {
    /*****/
}
```

short hiçbir zaman 150000’i geçemeyecekse bu kodda anlamsız ve ölü koddur.

Gereksiz kodlamalar da bu biçimde ele alınabilir.

20.2. Gereksiz Kodların Elemine Edilmesi

Derleyici akışsal analiz yaparak gereksiz kodları elemine etmeye çalışır. Örneğin:

```
if (ifade) {
    /****
    ****/
    ifade1;
}
else {
    /****
    ****/
    ifade1;
}
```

Burada ifade1’in if dışına yerleştirilmesi eşdeğerliği bozmaz. Fakat gereksiz bir kod elimine edilmiş olur. Örneğin:

```
if (a)
    return 1;
else
    return 0;
```

Burada yapılmak istenen şudur.

```
return a != 0;
```

20.3. Ortak Alt İfadelerin Elemine Edilmesi (Common Subexpression Elimination)

Peşi sıra ifadeler içerisindeki ortak alt ifadeler bir geçici değişkene yada yazmaca alınarak hız kazancı sağlanabilir. Örneğin:

```
a = x + y + z;  
b = x + y + k;  
c = x + y + l;
```

Burada $x + y$ ortak alt ifadedir. Dolayısıyla aslında aşağıdaki kod daha hızlı çalışır.

```
temp = x + y;  
a = temp + z;  
b = temp + k;  
c = temp + l;
```

Burada gereksiz bir biçimde $x + y$ tekrar tekrar yapılmayabilir. Şüphesiz burada temp bir yazmaç olabilir. Böyle değerlendirme yapıldığında tek başına değişkenlerinde bir alt ifade biçiminde yorumlanabileceği görülür. Örneğin:

```
a = x;  
b = x;  
c = x;
```

Burada her defasında bellekteki x 'i yeniden çekmektense bir kere yazmaca çekip yazmaçtan atama yapmak daha hızlı çalışmaya yol açar.

Anahtar Notlar:

Kritik kodlar dışında programcı için okunabilirlik çok daha önemli olmalıdır. Yani derleyicilerin zaten yapabildiği optimizasyonları kod içerisinde yapmak kodun okunabilirliğini azaltabilir. Yukarıdaki örnekte $x + y$ alt ifadesinin temp'e programcı tarafından yerleştirilmesi kötü bir tekniktir. Güçlü derleyicilerin hemen hepsi bu tür eliminasyonları yapabilmektedir.

20.4. Sabit İfadesi Yerleştirme (Constant Folding)

Sabit ifadelerinin değerleri çalışma zamanına bırakılmadan derleme zamanında hesaplanabilir. Örneğin:

```
x = 10 + y + 2;
```

Derleyici bu kodu $x = 12 + y;$ biçimine dönüştürebilir.

20.5. Sabit İfadelerinin Yaydırılması (Const Propagation)

Bu optimizasyon temasında derleyici değişmeyeceğini bildiği ifadelerin değerlerini ya baştan hesaplar ya da değişken yerine sabit kullanır. Sabit kullanımı da pek çok mimaride hız kazancı sağlamaktadır. Örneğin:

```
int x = 10 + 2;
int y;

y = x + 5;
```

Burada derleyici x içerisindeki ifadenin değişmediğini görerek $y = x + 5$; yerine $y = 17$; yerleştirebilir. Tabi burada x volatile yapılırsa derleyici böyle bir optimizasyonu yapmaz.

20.6. Ortak Blok (Basic Block)

Bu tema kodu küçültmek için uygulanmaktadır. Eğer kod içerisinde tekrarlanan bloklar varsa bu bloklara tek yerden girilip tek yerden çıkılıyorsa derleyici bu ortak kodlardan bir tane yerleştirip oraya jump işlemi yapabilir. Örneğin:

```
if (test1) {
    ifade1;
    ifade2;
    ifade3;
}

/*****
*****/

if (test2) {
    ifade1;
    ifade2;
    ifade3;
}
```

Burada if deyiminin doğruysa kısmı ortak bloktur. Dolayısıyla tıpkı bir fonksiyon gibi ele alınabilir.

```
if (test1)
    call REPEAT;
/****/
if (test2)
    call REPEAT;
/*****/
REPEAT:
    ifade1;
    ifade2;
    ifade3;
    ret;
```

20.7. Göstericilerin Eliminasyonu

Göstericilerin gösterdiği yerlere erişim, yani * operatörünün kendisi pek çok sistemde en az iki makine komutuna yol açmaktadır. Örneğin 32 bit Intel mimarisinde *p= val gibi bir işlem

```
mov eax, p
mov [eax], val
```

gibi bir makine koduyla gerçekleştirilir. Bir döngü içerisinde sürekli *p ifadesinin kullanıldığını düşünelim.

```
for (;;) {
    /***/
    a[i + 2] = *p;
}
```

Eğer derleyici p'nin göstericiği yerdeki bilginin değişmeyeceğinden emin olsa *p'yi bir geçici değişkene ya da yazmaca alır ve onu kullanabilir. Fakat buna emin olabilir mi? Optimizasyon işlemi tek threadli sistemde çalışıldığı varsayılarak yapılmaktadır. Dolayısıyla başka bir threadin *p'yi değiştirmesi gibi bir analizi derleyici yapmak zorunda değildir (Programcı *p global bir nesneyse göstericiyi volatile yaparak threadler arası kullanımda bu optimizasyonu engellemek isteyebilir.) . Derleyici *p'nin gösterdiği yerin değişmeyeceğinden emin olmalıdır. Örneğin p göstericisi bir nesneyi gösteriyor olsun o nesnede döngü içerisinde değişiyor olsun. Bu durumda derleyici bu optimizasyonu yapamaz. Örneğin:

```
for (;;) {
    x = i;
    /***/
    a[i + 2] = *p;
}
```

Burada örneğin eğer p x'i gösteriyorsa, derleyici *p'nin değerini yazmaya yerleştirerek hep o yazmacı kullanamaz. İşte derleyicinin buna emin olması gerekir. Ya da örneğin:

```
for (;;) {
    /***/
    a[i + x] = *p;
    func();
}
```

Ya p global değişkeni gösteriyorsa ve func fonksiyonu da onu değiştiriyorsa? Derleyicinin p'nin global bir nesneyi gösterdiğini anlayabildiğini düşünelim. Eğer func başka bir modüle

tanımlanmışsa, derleyici onun kodunu göremeyeceği için artık *p işlemini optimize edemez. Eğer func aynı kaynak dosya içerisinde ise derleyici inceleyerek buna karar verebilir. Derleyicilerin bu analizi yapması vakit alıcı bir işlemdir.

20.8. Inline Fonksiyon Açımı

Inline fonksiyonlar C90'da yoktur. C99 ve C++'ta vardır. Inline fonksiyonlar optimizasyon konusudur çünkü her ne kadar bu tür fonksiyonlar inline anahtar sözcüğüyle programcının isteği doğrultusunda oluşturuluyorsa da pek çok derleyicide kod optimizasyonu sırasında değerlendirilmektedir. Bir fonksiyon inline anahtar sözcüğüyle bildirilirse derleyici bu fonksiyon çağrıldığında fonksiyon çağrısı değil fonksiyon iç kodunu koda yerleştirir. Örneğin:

```
inline int Kare(int a)
{
    return a * a;
}
```

Şimdi biz bu fonksiyonu şöyle çağırmış olalım:

```
a = Kare(b);
```

Derleyici muhtemelen şöyle bir kod açacaktır:

```
a = b * b;
```

Görüldüğü gibi inline fonksiyonlar makrolara oldukça benzemektedir. Fakat çok daha kolay ve güvenli yazılabilmektedir. Ancak inline anahtar sözcüğü bir emir değil tavsiye niteliğindedir. Derleyiciler uzun kodları ya da çok zaman alan döngüler içeren kodları bir fayda sağlanamayacağı gerekçesiyle açmayabilirler.

Her ne kadar inline fonksiyonlar C90'da yoksa da pek çok C derleyicisinde uzun yıllardır bir eklenti biçiminde bulunmaktadır. Inline fonksiyonların uygunsuz kullanımları kodu büyütür. C99 ile C++'taki inline fonksiyonlar arasında bazı küçük semantik farklılıklar vardır. Inline fonksiyonların açılımı yapılması için kodlarının derleyiciler tarafından görülmesi gerekir. Bunun için bu fonksiyonların başlık dosyalarına yerleştirilmesi anlamlıdır. Fakat bu fonksiyonlar yine "external linkage" ye sahiptir. Yani başka bir modülden normal bir fonksiyon gibi de çağrılabilir. Inline fonksiyonlarının başka ayrıntıları da vardır. Fakat burada optimizasyon konusu üzerinde durulmuştur.

20.9. Döngü Değişmezleri (Loop Invariants)

Döngü değişmezlerinin döngünün dışına çıkartılması en klasik döngü optimizasyonlarından biridir. Örneğin:

```
for ( ; ; ) {  
    x = 10;  
    /****/  
}
```

Döngü içerisindeki fonksiyon çağrılarının döngü değişmezi olarak değerlendirilmesi çoğu kez mümkün değildir. Örneğin:

```
for ( ; ; ) {  
    x = Func();  
    /****/  
}
```

Çünkü fonksiyon başka yan etkilere yol açabilir. Örneğin bir global değişkeni değiştiriyor veya bir dosyaya bir şeyler yazıyor olabilir. Programcı bu yan etkinin birden fazla kez oluşmasını sağlamak için çağrıyı döngü içerisine almış olabilir. Tabi derleyici fonksiyonu tanımlamasını görürse ve bunun bir yan etkiye yol açmadığını belirlerse bu durumda çağrıyı döngü dışına çıkartabilir. Örneğin bir karakter dizisinin aşağıdaki gibi dolaşılması kötü bir fikirdir:

```
for (i = 0; i < strlen(s); i++) {  
    /*****/  
}
```

Derleyiciler standart C fonksiyonlarının ne yaptığını bilmek zorunda değildir. Hatta derleyiciler standart C fonksiyonlarının farkında da değildirler. Dolayısıyla derleyici strlen fonksiyonunun bir yan etkiye sahip olabileceğini düşündüğü için her defasında bunu yeniden çağırarak zorunda kalır. O halde bu kodun programcı tarafından şöyle düzeltilmesi gerekir:

```
length = strlen(s);  
for (i = 0; i < length; i++) {  
    /*****/  
}
```

Fakat pek çok iddialı derleyicide “intrinsic function” denilen fonksiyon kavramı da vardır. Örneğin Microsoft C derleyicilerinde pek çok standart C fonksiyonu “intrinsic function”dır. Bu tür fonksiyonların derleyici ne yaptığını bilir. Hatta bunlar için prototip de istemez. Dolayısıyla bu tür fonksiyonlar döngü değişmezi olarak değerlendirilebilir. Başka ilave ek optimizasyonlara yol açabilir. Örneğin:

```
x = strlen(s);  
y = strlen(s + 1);
```

Burada eğer `strlen` “intrinsic function” ise derleyici bu fonksiyonun yazı uzunluğunu bulduğunu bilir dolayısıyla ikinci çağrıyı hiç yapmadan $y = x - 1$ gibi bir kod üretebilir.

20.10. Döngü Açımı (Loop Unrolling)

Bu optimizasyon temasında bir döngünün dönme miktarı azaltılır. Bunun için derleyici döngüyü genişletir, döngü içerisindeki ifadeleri birden fazla kez yazar, böylece de döngünün az dönmelerini, dolayısıyla daha az döngü kontrolü yapılmasını sağlar. Örneğin:

```
for (i = 0; i < 10; i++)  
    ifade;  
  
for (i = 0; i < 5; i++) {  
    ifade;  
    ifade;  
}
```

Aşağıdaki kod daha fazla yer kaplamasına karşın daha hızlı çalışır. Hatta abartılı olarak tamamen döngüyü kaldırıp peşi sıra 10 tane ifadeyi yazabilirdik. Tabii bunun bir sınırı vardır. Yani milyon kez dönen bir döngünün tam açılması düşünülemez. Fakat 4 kat ya da 8 kat açılım yapılabilir.

Bazen döngü açimleri daha karmaşık bir biçimde yapılabilir. Buna “loop unwinding” denir.

Örneğin:

```
for (i = 0; i < 100; i++)  
    Func(i);
```

Yerine aşağıdaki kod yazılabilir:

```
for (i = 0; i < 100; i += 5) {  
    Func(i);  
    Func(i + 1);  
    Func(i + 2);  
    Func(i + 3);  
    Func(i + 4);  
}
```

20.11. Döngü Ayırması (Loop Splitting)

Bazen bir döngünün içerisinde if gibi deyimler varsa bu if deyiminden kurtulmak için derleyici döngüleri ayırabilir. Örneğin:

```
for (i = 0; i < 100; i++)  
  if (i % 2 == 0)  
    ifade1;  
  else  
    ifade2;
```

Yerine aşağıdaki açılım yapılabilir:

```
for (i = 0; i < 100; i += 2)  
  ifade1;  
for (i = 1; i < 100; i += 2)  
  ifade2;
```

Tabi döngüyü bölmek gerçekten bir avantaj sağlıyorsa yapılabilir.

20.12. Kod Üretimi Aşamasında Yapılan Optimizasyonlar

Klasik optimizasyonlar ara kod üzerinde ya da pars tree üzerinde yapılmaktadır. Bu optimizasyonlardan geçildikten sonra optimize edilmiş olan kod için üretim yapılır. İşte bazı optimizasyonlar kod üretimine yöneliktir. Bunların en önemlileri şunlardır.

- Komut seçimi (instruction selection)
-

Tipik olarak CISC tabanlı işlemcilerde komutların cycle süreleri birbirinden farklı olma eğilimindedir. Yani çok sayıda komut vardır ve bunların çalışma süreleri farklıdır. Halbuki RISC tabanlı işlemcilerde komutların çalışma süreleri eşit olma eğilimindedir. Bu durumda girişi yapabilmek için farklı makine komutları seçilebilir. İşte derleyicinin hız optimizasyonunda en hızlı bir biçimde çalışacak komutları seçmesi gerekir. Benzer biçimde büyüklük optimizasyonunda da derleyici en az yer kaplayacak makine komutlarıyla programcının istediği işleri yapmaya çalışır. Genel olarak CISC tabanlı işlemcilerde komutlar farklı uzunluklarda olma eğilimindedir. Halbuki RISC işlemcilerinde eşit uzunluktadır.

Komut seçimi en önemli problemlerden biridir. Bu aşama RISC işlemcilerinde daha kolay, CISC işlemcilerinde daha zor gerçekleştirilir. Çünkü RISC işlemcilerinde komut sayısı azdır ve aralarında ciddi bir performans farkı yoktur.

20.13. Yazmaç Tahsisatı (Register Allocation)

Derleyiciler prensip olarak mümkün olduğu kadar çok değişkeni, mümkün olduğu kadar uzun süre yazmaçlarda tutmak isterler. Genel olarak RISC tabanlı işlemcilerde fazla sayıda yazmaç bulunma eğilimindedir. ALFA gibi çoğu RISC işlemcisinde 32 tane genel amaçlı yazmaç, 32 tane de gerçek sayı yazmacı bulunmaktadır. Fakat Intel, Pentium gibi CISC işlemcilerinde yazmaç sayısı çok azdır. İşte derleyici yazmaç sayısını n olmak üzere ne zaman ve hangi n tane değişkeni yazmaçta tutacağına karar vermelidir. Bu NP tarzda bir problemdir ve çeşitli hevristic (iyi çözümü veren) çözümleri vardır(ACM de bu konuya ilişkin birçok makale bulunabilir.).

20.14. Komut Çizelgelemesi (Instruction Scheduling)

Pek çok programda bazı ifadelerin ya da deyimlerin yerlerinin değiştirilmesi eşdeğerliği bozmaz. Örneğin:

```
x = a;  
y = b;  
z = c;  
k = a;
```

Burada biz $x = a$ ile $k = a$ 'yı peş peşe getirsek de değişen bir şey olmayacaktır. Fakat bu sıralama performansı etkileyebilir. İşte komut çizelgelemesi komutların yerlerini değiştirerek bundan bir çıkar sağlamaya yönelik işlemlerdir. Bazı komutların bazı komutlardan önce yada sonra gelmesi çeşitli nedenlerden dolayı performansı etkileyebilir.

20.15. Derleyicilerin Optimizasyon Seçenekleri

Optimizasyon mekanizmasına sahip derleyicilerin çoğunda her bir optimizasyon seçeneği Duruma göre açılıp kapatılabilir. Açılıp kapatma işlemi #pragma komutlarıyla kod içinde yapılabileceği gibi komut satırında derleyici seçenekleri ilede yapılabilmektedir. Ayrıca IDE li

sistemlerde ayarlama işlemi menüden yapılabilir. Böylece derleyici komut satırında bu belirlemeleri kullanabilir.

Microsoft ve Borland derleyicilerindeki optimizasyon seçenekleri gcc ve intel derleyicilerine göre daha azdır. Seçenekleri tek tek açıp kapatmak zahmetli olduğu için bu derleyicilerde katagori olarak açıp kapatmak seçenekleri de vardır.

20.15.1. Microsoft Derleyicilerindeki Optimizasyon Seçenekleri

Microsoft derleyicilerinde /Od seçeneği derleyici optimizasyonlarını kapatır. Bu durum default durumdur. Çünkü debug işlemleri optimizasyon kapatıldığında daha sağlıklı yürütülmektedir. **(komut satırından c: cl /O1 örnek program)/O1** seçeneği Sizeof optimizasyonu yapmak için kullanılır. Yani bu optimizasyon da derleyici hızlı çalışmadan ziyade daha az yer kaplayacak kod üretmeye çalışır. /O1 seçeneği aslında daha ayrıntılı olan şu seçeneklerin hepsinin açılmasıyla eşdeğerdir: /O2 seçeneği hızlı kod üretmek için kullanılan seçenektir. Burada derleyici yukarıda açıklanan optimizasyon temalarını kullanarak kod u hızlı çalışacak hale getirmeye çalışmaktadır. O2 seçeneği relaese mod için default durumdur.

Anahtar Notlar:

Derleyicinin çeşitli ayarlarından bir konfügrasyon oluşturulabilir. Microsoft ide lerinde proje oluşturulduğunda 2 temel konfügraasyon yaratılmış durumdadır. Bunlarda biri Debug diğeri relaese knfügrasyonlarıdır. Fakat programcı isterse başka konfügrasyonlarda oluşturabilir. Debug konfügrasyonu proje yaratıldığında default aktif olan konfügrasyondur. Konfügrasyon değıştirmesi Araç çubuğundan ya da buid/Confügrasyon manager menüsünden yapılabilir. Debug konfügrasyondaki default optimizasyon seçeneği /Od Relaese konfügraasyondaki ise /O2 dir.

/Od seçenekleri /Ob0 /Ob1 ya da /Ob2 biçiminde kullanılır. Bu seçenek Inline fonksiyon açimleri için kullanılmaktadır. /Ob0 Inline işlemini kapatır ve default durumdur. /Ob1 inline anahtar sözcüğü ile belirtilmiş fonksiyonları açmaya çalışır. /Ob2 inline belirlemesi yapılmamış olsa bile derleyicinin uygun fonksiyonları inline açabilmesini sağlar. /Ob seçeneklerinin kullanılabilmesi için mutlaka /O1 ya da /O2 ya da /Og seçeneklerinin kullanılmış olması gerekir.

Anahtar Notlar:

Özellikle C++ da inline fonksiyonlar yoğun kullanılır. Çünkü sınıfın private elemanlarına erişmekte kullanılan get ve set fonksiyonları yoğundur. Programcının derleyicinin inline açımını yapabilmesi için en azından /O1 ya da /O2 seçenekleri içinde olması gerekir. Mademki /Od proje yaratıldığında default durumdur. O halde bu durumda microsoft derleyicileri kesinlikle açım yapmamaktadır.

Microsoft derleyicilerindeki tüm optimizasyon seçenekleri /Oxxx biçimindedir. Yani bu seçenekler /O karakteriyle başlamaktadır. Yukarıda da belirtildiği gibi /O1 ve /O2 aslında bir seçenek grubudur.

Optimizasyon konusunda açıklanan alt ifadelerin elimine edilmesi gibi çok kullanılan değişkenlerin yazmaçta tutulması gibi döngü de değişmeyen ifadelerin döngü dışına çıkarılması gibi optimizasyon temalarının çoğu /Og seçeneği ile aktif hale getirilmektedir. Kategori olarak /O1 ve /O2 seçenekleri açıldığında zaten /Og de otomatik olarak açılır. Fakat biz /O1 ya da /O2 yi kullanmadan /Og seçeneğini kullanabiliriz. Diğer optimizasyon seçenekleri derleyici dökümanlarından izlenebilir.

20.15.2. GCC Derleyicilerindeki Optimizasyon Seçenekleri

GCC derleyicilerinde -fxxx biçiminde yani -f harfleriyle başlayan pek çok optimizasyon seçeneği vardır. Neredeyse her bir optimizasyon teması için ayrı bir seçenek bulunmaktadır. Fakat bu kadar seçeneğin açılıp kapatılması çok zahmetlidir. Bunun yerine gcc programcıları yine bazı kategorileri kullanırlar. Önemli kategoriler şunlardır.

-o0: Bu durum default durumdur.(gcc -O0 program adı) Bu seçenek optimizasyonu kapatmaktadır. Microsoft un /Od seçeneğine karşılık gelmektedir.

-o1: Bu seçenek temel bazı alt optimizasyon seçeneklerini aktive etmektedir. Hangi alt optimizasyon seçeneklerinin aktive edildiği burada ele alınmayacaktır. Fakat -o1 seçeneği temel optimizasyonları yerine getiren bir seçenektir. Bu seçenekte daha fazla optimizasyon seçeneği açılmış durumdadır. Fakat bu seçenekte açılan alt seçenekler kod u aşırı biçimde büyütmemektedir.

-o2: -o1 seçeneğinden farklı olmak üzere derleme sırası daha uzar pek çok açık yazılım (Örneğin Linux işletim sistemi) bu seçenekte derlenmektedir.

-o3: Agresif optimizasyon seçeneğidir. Döngü açımı gibi optimizasyonlarda bu seçeneğin içindedir. Kod daha hızlı çalışacak hale getirilir. Fakat bu seçenek kod büyümesine de yol açmaktadır.

-os: Bu seçenek microsoft un /O1 seçeneğine benzer Size optimizasyonu yapmaktadır.

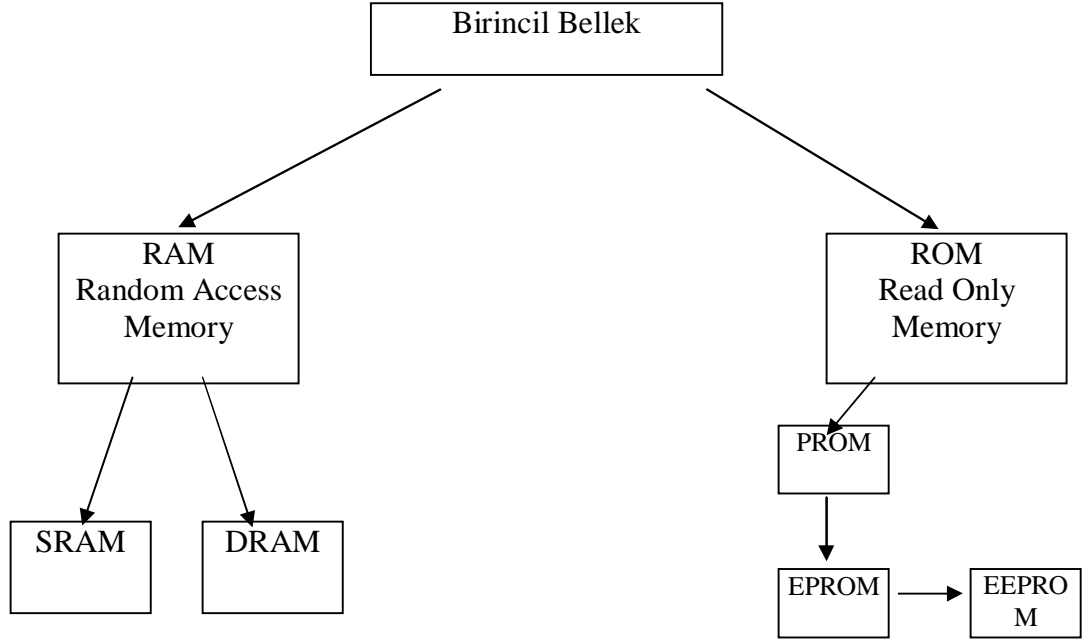
[Ek - 1] Options That Control Optimization

21. BELLEK İŞLEMLERİ

Bilgisayar ortamında bilgilerin daha sonra işlenmesi için saklandığı birimlere bellek denir. Bellekler birincil(primary) ve ikinci(secondary) olmak üzere iki bölüme ayrılmaktadır.

Birincil belleklere ana bellek(main memory) denilmektedir.

CPU ile elektriksel olarak doğrudan bağlantılı belleklere 1.cil bellekler denir. Bu bellekler tarihsel gelişim dikkate alındığında RAM ve ROM bellekler olmak üzere ikiye ayrılır.



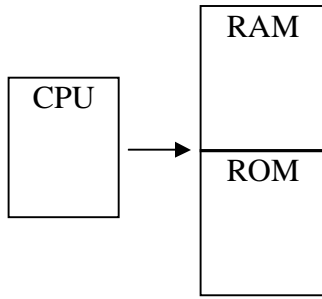
RAM ler hem okunabilen hem yazılabilen yarı ,letken belleklerdir. Kendi aralarında statik ve dinamik olmak üzere iki bölüme ayrılmaktadır.

SRAM ler çok transistörle Flip – Flop biçiminde tasarlanmışlardır. Örneğin bir bit için bir SR tarzda tutucu devre 2 logic kapı içermektedir. Bu logic kapılar birden fazla taransistörle gerçekleştirilir. SRAM ler çok hızlıdır. (Tipik 1 nano saniyenin altında) Fakat bu RAM ler görel olarak daha büyük yer kaplar daha pahalıdır. Tüm bilgisayar belleğini SRAM olarak oluşturmak yerine Cache sistemi kurarak Cache sistemini SRAM olarak oluşturmak daha ekonomiktir.

DRAM belleklerin 1 biti tipik olarak bir tansistör bir kapasitör olarak tasarlanmıştır. Kapasitif elemanın belirli periyotlarda şarj edilmesi gerekir ve buna tazeleme denilmektedir ve DRAM lerin en önemli problemidir. Pek çok CPU nun DRAM ları tazelemek için özel uçları bulunmaktadır. Bugün kullandığımız DDR tarzı RAM ler de DRAM grubundadır. DRAM ların hızları 10 nano saniye çivarına kadar düşürülmüştür. Daha ucuzdur ve daha az yer kaplamaktadır.

ROM lar güç kaynağını kesince bilgiyi tutmaya devam eden yarı iletken birincil belleklerdir. Tarihsel süreç içinde evrim geçirmişlerdir.

Anahtar Notlar:



Her bilgisayar sisteminde bir ROM bölgesinin olması gerekir. Micro işlemciler ve Micro Denetleyiciler Reset edildiğinde çalışma belli bir adresten başlar. Buna Reset Vektörü denilmektedir. Genellikle reset vektörleri ya belleğin başında ya da sonunda bulunur. Bilgisayarın güç kaynağını açtığımızda RAM bölgesi sıfırlandığına göre ve CPU reset vektöründen çalışmaya başladığına göre reset vektöründe hazır bir programın bulunması gerekir. İşte reset vektörünün bulunduğu bölgenin Rom bölümü içinde bulunuyor olması gerekir. Böylece bilgisayarı kapattığımızda RAM bölgesi silinir fakat ROM bölgesindeki bu programlar kalmaya devam eder.

(POST – Power on self test)ROM da çalıştırılan ilk kod POST denilen self test programıdır. Masaüstü sistemlerde daha sonra sıra işletim sisteminin yüklenmesine gelir. Yani işletim sisteminin yüklenmesi tamamen ROM dan başlatılan bir eylemdir. Pek çok micro denetleyici sisteminde bir işletim sistemi olmadan çalışılır. Bu tür sistemlerde programcı yazdığı programı micro denetleyicinin içindeki ROM alanına aktarır. Böylece Micro denetleyici reset edildiğinde çalışma ROM a yerleştirilmiş programdan başlamış olur.

Tarihsel olarak ilk ROM türü belleklere PROM (Programmable ROM) denilmektedir. Bu belleklere genellikle üretici firma tarafından yalnızca 1 kez bilgi yerleştirilir. Daha sonra uzun yıllar EPROM lar(Erasable Programmable ROM) üretilmiştir. EPROM lar özel EPROM silicilerle silinerek yeniden programlanabilmektedir. Nihayet son teknoloji olarak EEPROM (ElectricallyErasable Programmable ROM) denilen ROM türü bellekler tasarlanmıştır. Bu tür bellekler RAM gibi ROM dur. Yani bunlara hiç silmeden program yerleştirilebilir. Bu işlem için özel bir devreye gereksinim duyulmaz ve güç kaynağı kesilince de bilgiyi tutarlar. Benzer bir teknoloji de Flash EPROM teknolojisidir. EEPROM ve Flash EPROM çok benzer yapıdadır. Bugün kullandığımız MP3 vçalar gibi ev bilgisayarları gibi taşınabilir bellekler gibi birimler hep

bu biçimde üretilmektedir. Böylece bu cihazlara firma tarafından yerleştirilmiş programlar(firmware) güncellenebilmektedir. Bugünkü masaüstü bilgisayarlarda ve taşınabilir bilgisayarlardaki ROM tarzı bellekler Flash EPROM belleklerdir. Örneğin bilgisayar Reset edildiğinde çalışan programı ve kesme kodlarını BIOS update yaparak yenisiyle yer değiştirebilmekteyiz.

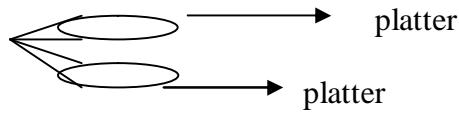
21.1. İkincil Bellekler

Bu bellekler CPU ile doğrudan bağlantılı olmayan bilgisayar sistemi kapatıldığında bilgilerin kalıcı olmasını sağlayan belleklerdir. Tipik olarak diskler floppy disketler, CD ve DVD ler, flash EPROM teknolojisi ile oluşturulmuş memory stikler, teyp bantları

Diskler bugün için en önemli ikincil belleklerdir. Belki gelecekte disklerin yerini EEPROM veya Flash EPROM lar alabilirler.

21.2. Disk İşlemleri

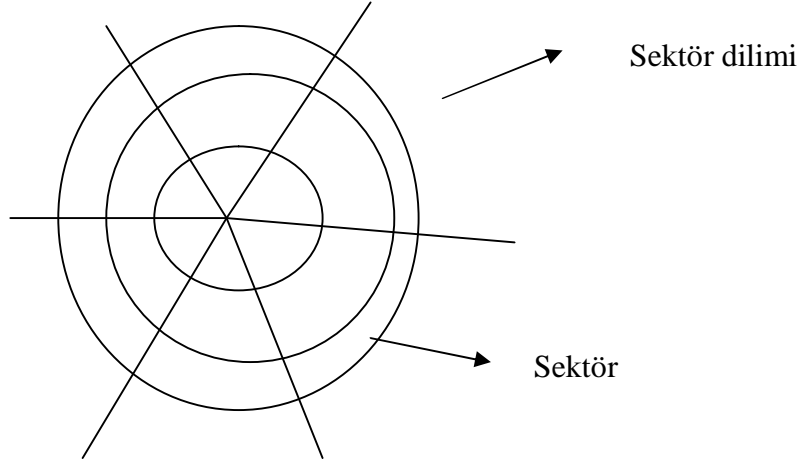
Disk terimi floppy disketleri de içine alan manyetik tabanlı medyalar için kullanılmaktadır. Diskin her yüzeyi track denilen yuvarlar yollardan oluşmaktadır. Disk kafası track hizasına getirilir, kafa yüzeye değmez ama çok yaklaşır, okuma ve yazma işlemleri disk dönerken kafa tarafından yapılmaktadır. Disk birden fazla yüzey içeriyorsa her yüzey ayrı bir kafa tarafından okunur ve yazılır. Kafalar aynı eksene monte edilmiştir, birlikte hareket eder. Hard diskler bir çubuk içerisine geçirilmiş birden fazla platterdan oluşabilmektedir.



Örneğin 2 platterlı bir hard disk 4 yüzeyden oluşmaktadır. Bu dört yüzün her biri kafa tarafından okunur. Yüz sayısı ile kafa sayısı aynı anlamdadır.

Diskin dönüş hızı dakikadaki devir sayısı ile belirtilir. Örneğin floppy disketler 360 RPM civarında dönmektedir. Hard disklerdeki dönüş hızları, hard diske göre değişebilmektedir. Örneğin tipik olarak 5400, 6000, 7200, 9000 gibi değerler söz konusudur.

Aynı zamanda diskin her yüzeyi mantıksal olarak pasta dilimlerine ayrılmıştır. Bu dilimlere sektör dilimleri denir.



Tracklerin sektör dilimleri içerisinde kalan her bir parçasına sektör denilmektedir. Sektör bir diskten okunabilecek ya da diske yazılabilecek en küçük birimdir. Sektörlerin yay uzunlukları farklı olsa da her sektörde eşit uzunlukta bilgi vardır.

Anahtar Notlar:

Son 10 senedir hard disk teknolojisinde bazı değişiklikler yapılmıştır. Örneğin her sektöre eşit sayıda bayt yerleştirmek yerine dış sektörler daha fazla bilgi yerleştirmek yoluna gidilmektedir. Burada klasik sektör tanımının dışına çıkılsa da yine sektör kavramı muhafaza edilmiştir. Yani disk bize, bizde diske yine sektör numarası vererek erişiriz. Anlatımımızda önceki teknoloji dikkate alınmakla birlikte aslında iki sistem arasında soyutlama bakımından önemli bir fark yoktur.

Bir diskin hızı sektör transferi ile belirtilmektedir. Bir sektörün ortalama olarak kaç msn'de yazılıp okunduğu bir hız göstergesidir. Bugün en iyi diskler bu hız 6 msn civarındadır. Ortalama bir diskte 12 msn civarındadır. Şüphesiz bir sektörün okunması ya da yazılması kafanın o andaki durumu ile ilgilidir. Ortalama hızlar simülasyon yoluyla elde edilmiş değerlerdir.

Bir sektörün transferi ile ilgili transfer zamanını etkileyen 3 bileşen vardır.

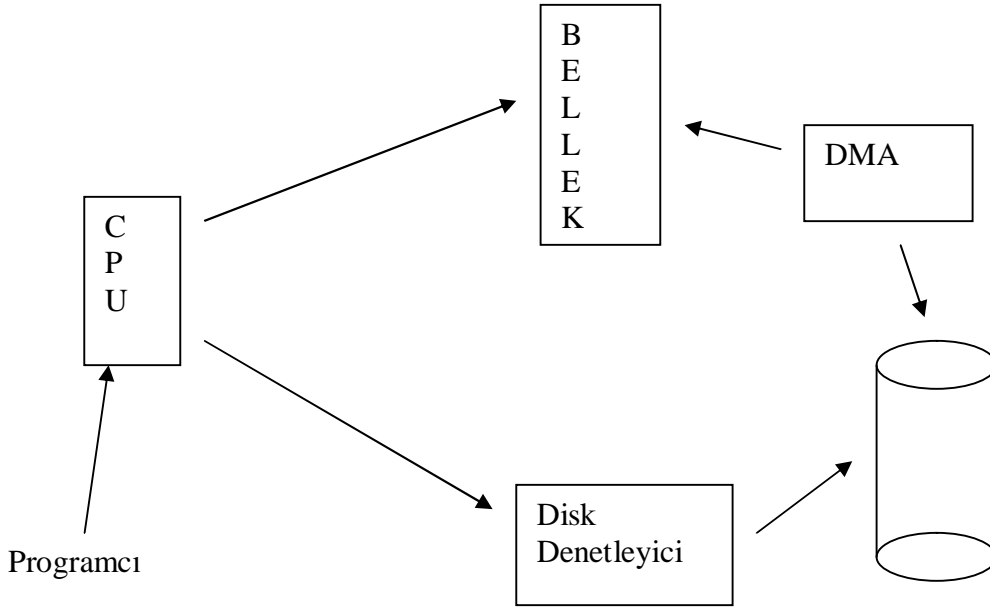
1. **Kafanın Konumlanma Süresi (Seek Time):** Bu süre kafanın bulunduğu durumdan transfer edilecek sektörün bulunduğu track hizasına konumlandırılması için gereken zamanı belirtir. Bu süre hız açısından etkili olan en önemli bileşendir.
2. **Dönme Zamanı (Rotational Delay):** Transfer edilecek sektörün bulunduğu dilimin dönerek kafa hizasına gelmesi arasında geçen süredir. Hız konusundaki ikinci önemli bileşeni

oluşturur. Şüphesiz bu süre diskin dönüş hızıyla ilgilidir. O halde hızlı dönen disklerin görece olarak biraz daha hızlı olması beklenir.

3. Transfer Zamanı: Bu zaman kafanın sektörü okuyarak dış dünyaya ilettiği zamandır. Görece olarak en kısa zaman budur.

21.3. Disk IO İşlemi

Tipik bir bilgisayar sisteminde anakart üzerinde diskleri yönetmek için bir disk denetleyicisi denilen işlemci bulunmaktadır. Sistem programcısı bu işlemciyi programlayarak sektör transfer isteğini sisteme bildirir. Disk denetleyicisi hard disk üzerindeki işlemcileri programlayarak transfer işlemini başlatır. Sistemlerin çoğunda CPU anakart üzerindeki işlemciyi programladıktan sonra artık boşta kalır. Sektörün belleğe aktarımı DMA(direct memory Access) denilen yardımcı bir işlemcinin kontrolünde yürütülmektedir. Disk denetleyicisi işlem bittikten sonra bir kesme yoluyla bunu CPU'ya bildirir. Bu aktarım şekilsel olarak şöyle gösterilebilir:



Bu transfer işlemindeki önemli noktalar şunlardır:

- CPU yani bizim programımız aslında yalnızca disk denetleyicisini ve DMA'yı programlamaktadır. Geri kalan işlemlere CPU karışmaz.
- Diskten belleğe aktarım DMA denilen işlemci yardımıyla otomatik gerçekleştirilir.
- CPU yani programız işlemi başlatır. Fakat işlem bitene kadar bekler. Çok işlemlili sistem (multi threading) sistemlerde tipik olarak ilgili proses yada thread işlem bitene kadar çizelge dışına çıkartılarak bloke bekletilir.
- Transfer bittiğinde donanım kesmesi yoluyla CPU durumdan haberdar edilir böylece işletim sistemi devreye girerek prosesi yada threadi yeniden çizelgeye sokar.

Anahtar Notlar:

Tipik olarak çok işlemlili işletim sistemlerinde çizelgede prosesler ya da threadler bir kuyruk sistemi ile ifade edilir. Bloke olduğunda ilgili proses ya da thread bu kuyruk sisteminden çıkartılıp bir bekleme kuyruğuna (wait queue) alınır. Genellikle bekleme kuyruğu bir tane değildir. Değişik olaylar için değişik bekleme kuyrukları vardır.

21.4. Sektör Transferi İşleminin Programlama Yolu ile Gerçekleştirilmesi

Tipik olarak PC sisteminde daha işletim sistemi bile yüklenmeden yukarıda açıklandığı gibi disk denetleyicisini ve DMA'yı programlayarak transferi gerçekleştiren makine dilinde yazılmış ROM bellekte bulunan bir kod vardır. 13H kesmesi diye bilinen bu kod silinmeyecek biçimde her zaman hazırda bulunmaktadır. Böylece işletim sistemi yüklenirken bu kodlardan faydalanılmaktadır. Fakat işletim sistemi yüklendikten sonra artık transfer işlemi için işletim sisteminin kendi kodları kullanılır. Windows sisteminde tipik olarak bu işlerden sorumlu bir aygıt sürücüsü vardır. Sistem programı ise aygıt sürücüsünden istekte bulunur ve böylece kernel moda geçilir, transfer yapılır. Linux sistemlerinde aynı biçimde aygıt sürücüler yoluyla transfer yapılmaktadır. /dev dizininde bulunan hda, hdb gibi hard diskleri temsil eden aygıt sürücüler open fonksiyonu ile açılır ve sektör okuma yazma işlemleri read/write fonksiyonlarıyla dosya işlemi gibi gerçekleştirilir.

Kursumuzda Windows sistemleri altında sektör transfer edebilmek için aygıt sürücülerini kullanan ReadSektor ve WriteSektor gibi iki fonksiyon yazılmıştır ve bunlar kullanılacaktır. Şüphesiz bu fonksiyonlar kendi içlerinde aygıt sürücülerini kullanarak transfer işlemi yapmaktadır.

Aygıt sürücüde yukarda anlatıldığı gibi disk denetleyicisini ve DMA'yı programlayarak işlemini yapmaktadır.

21.5. Sektörlerin Numaralandırılması

Transfer işlemi için sektörlerin numaralandırılmış olması gerekmektedir. Bunun için iki koordinat sistemi kullanılmaktadır. Bunlar fiziksel ve mantıksal sistemlerdir. Fiziksel koordinat sisteminde sektörün yeri "kafa: thread: sektör dilimi" biçiminde 3 bileşenle belirlenir. Kafa ile yüzey aynı anlamdadır. Yüzeyle sıfırdan başlayarak numara verilmiştir. Trackler de sıfırdan başlayarak numaralandırılmıştır. En dıştaki track 0 numaralı tracktir. Son bileşen sektör dilimidir. Her sektör diliminin de bir numarası vardır. Sektör dilimleri birden başlayarak numaralandırılmıştır. Örneğin 3:28:5; 3. yüzey, 28. thread, 5. sektör dilimi ile belirlenen sektörü gösterir.

Mantıksal koordinat sisteminde sektörün yeri tek değerle belirtilir. Her sektöre bir numara verilmiştir. Numaralandırma şöyle yapılmıştır. 0h:0t:1s numaralı sektörü 0 olmak üzere önce sektör dilimleri, sonra kafalar, sonra trackler dolaşarak artan sayıda numaralandırma söz konusudur.

```
for (track)
  for (head)
    for (sektör dilimi)
```

Eğer sektör dilimlerinin sayısı ve kafa sayısı bilinirse mantıksal sektörü fiziksel sektöre, fiziksel sektörü de mantıksal sektöre dönüştüren formüller yazılabilir.

Anahtar Notlar:

Disk ile ilgili bir kavram da silindir kavramıdır. Silindir her yüzeydeki yani kafadaki n numaralı tracklerden oluşan kümedir. Örneğin 4 yüzeyli yani kafalı bir disk düşünelim. 5. silindir demek 0. yüzdeki 5. track, 1. yüzdeki 5. track, 2. yüzdeki 5. track ve 3. yüzdeki 5. trackin oluşturduğu kümedir.

21.6. İşletim Sistemlerinin Dosya İşlemlerine İlişkin Disk Organizasyonları

Programcıya göre bir dosya ardışık bayt topluluğundan oluşan bir kavramdır. Fakat aslında dosya mantıksal bir kavramdır ve işletim sistemi tarafından organize edilmektedir. Aslında diskte

tek gerçek sektör kavramıdır. Dosya yüksek seviyeli ve uydurma bir kavramdır. Aslında bir dosyanın parçaları disk üzerinde çeşitli sektörlerde yayılmıştır. İşletim sistemi dosyanın hangi parçalarının hangi sektör üzerinde olduğunu bir biçimde bilmektedir. İşletim sistemi diskte sektörlerde yayılmış olan parçaları bize sanki ardışık bir bilgiymiş gibi göstermektedir.

İşletim sisteminin dosya alt sisteminin iki önemli bölümü vardır. Bunlardan biri bellekte yapılan organizasyonlar ile ilgilidir. Diğeri diskte yapılan organizasyonlar ile ilgilidir. Bu nedenle birine bellek tarafı diğeri disk tarafı denir. Bir dosya açıldığında bellekte oluşturulan veri yapıları disk cache sistemi bellek tarafına ilişkin konulardır. Dosyaların hangi sektörde saklanacağı, dosyaların hangi parçalarının hangi sektörlerde bulunacağı disk tarafına ilişkindir.

21.7. Cluster ya da Blok Kavramı

Dosyanın parçalarının disk üzerinde rasgele sektörlerde bulunması dosyaya erişmek için gereken kafa hareketlerini fazlalattırır dolayısıyla performansı düşürür. O halde ilk akla gelecek organizasyon dosyanın parçalarını ardışık sektörlerde yerleştirmektir. Fakat dosyanın ardışık sektörlerde yerleştirilmesi bölünme (fragmentation) denilen probleme yol açmaktadır. Bölünme bellek konusuna ilişkin önemli kavramlardan biridir.

Disk

////////////////////////////////////

////////////////////////////////////

////////////////////////////////////

////////////////////////////////////

////////////////////////////////////

////////////////////////////////////

Bir dosyanın parçaları diskte ardışıl yerleştirilmek zorunda değildir. Eğer böyle bir zorunluluk olsaydı, dosyaların tahsis edilip silinmesi sürecinde bölünme(fragmentation) durumu oluşurdu.

Bölünme çok sayıda küçük fakat ardışıl olmayan boş bölgelerin oluşması anlamına gelir. Bölünme kavramı yalnızca disk için değil her türlü bellek sistemi için söz konusudur. Örneğin heap bölgesinde zamanla bölünmeye maruz kalmaktadır. Yani heap bölgesinde de çok sayıda küçük alanlar oluşabilmekte ve orta büyüklükte tahsisatlar bile yapılamayabilmektedir.

Bölünme sorunu ardışıl yerleştirme zorunluluğunun ortadan kaldırılmasıyla çözümlenebilir. Böylece küçük parçalarda büyük bir birimin parçaları olarak kullanılabilir.

Bölünmenin ortadan kaldırılması için tipik olarak ilgili bellek eşit uzunlukta bloklara ayrılır. Yerleştirme işlemi blok temelinde gerçekleştirilir. Şüpesiz yerleştirilen bilginin hangi parçalarının hangi bloklarda olduğu bir biçimde tutulmalıdır.

Bir dosyanın parçası olabilecek en küçük birime cluster yada blok denilmektedir. İşletim sistemi diski eşit uzunlukta cluster yada bloklara ayırır, dosyanın parçalarını da cluster yada bloklara yerleştirir. Şüpesiz her dosyanın hangi numaralı parçasının hangi cluster yada blokta bulunduğu bir biçimde tutulmalıdır. Bu sistemin anahtar noktaları şunlardır.

- Disk yada volume eşit uzunlukta cluster yada bloklara ayrılmıştır.

0	1	2	3	4		
5	6	7	8	9		
10						

Örneğin diskimiz 100k uzunlukta olsun. Bu durumda diskimizde toplam 100 cluster yada blok vardır. Bunlar 0 dan 99 a kadar numaralandırılmış olacaktır.

- İşletim sistem dosyaları parçalara ayırır ve boş olan cluster yada bloklara yerleştirir, ve bunu bir biçimde bir yerde tutar. Örneğin yukarıdaki diski dikkate alalım, 6k lık bir dosya işletim sistemi tarafından 6 farklı cluster yada bloğa yerleştirilecektir. Bunların 8 – 9 13 – 14 25 - 26 numaralı bloklar olduğunu farz edelim. İşletim sistemin bu dosyaların bu parçalarının bu yerlerde olduğunu bir yerde tutması gerekir.

- Cluster yada blok kavramı işletim sisteminin belirlediği mantıksal büyüklüktür. Diskteki gerçek fiziksel büyüklük sektör kavramıdır. Bir cluster yada blok ardışıl 2 üzeri n sektör olabilir. Örneğin 1, 2, 4, 8, 16, 32... gibi.

- Bir cluster yada blok kaç sektörden oluşmalıdır ? Eğer bir cluster yada blok az sektörden oluşursa, dosyanın parçaları diske çok yayılır, bu da diskin kapı hareketlerini fazlalaştırır ve performansı düşürür. Üstelikte dosyanın hangi parçalarının hangi cluster yada bloklarda tutulması daha fazla yer gerektirir. Bir cluster yada blok çok sayıda sektörden oluşursa bu sefer içsel bölünme (internal fragmentation) artar. İçsel bölünme her dosyanın son parçasının yerleştirildiği cluster yada blokta boş alanların oluşması durumudur. Örneğin bir cluster yada bloğun 1k olduğu bir sistemde 1500 byte uzunluktaki bir dosya 2 cluster yada blok yer kaplayacaktır. Bu durumda (2048 – 1500) byte lık bir alan boşa harcanmış olacaktır. İçsel bölünme sıfırlanamaz fakat bir cluster yada blok ne kadar az sektörden oluşursa ona bağlı olarak azaltılabilir. O halde bir cluster yada bloğun kaç sektörden oluşması gerektiği için şunlar söylenebilir : Cluster yada blok ne kadar az sektörden oluşursa, içsel bölünme miktarı o kadar azaltılır, fakat disk performansı o oranda kötüleşir. Bu nedenle iyi bir noktanın bulunması gerekir. Eğer disk küçükse alan daha önemlidir. Bu durumda bir cluster yada bloğun daha az sektörden oluşması anlamlıdır. Örneğin floppy disklerde bir cluster yada blok 1 sektördür. Fakat medya büyüdükçe alan çok olduğuna göre hız daha önemli duruma gelir. Örneğin büyük disklerde 1 cluster yada blok 16 yada 32 sektör kadar olabilir. Bir cluster yada bloğun kaç sektörden oluşacağına formatlama sırasında karar verilmektedir. İşletim sistemi dosyanın parçalarını şüpesiz mümkün olduğunca ardışıl cluster yada bloklara yerleştirir. Fakat zamanla dosyaların parçaları disk de yayılabilir. Defrag gibi programlar dosyaların cluster yada bloklarını birbirine yakınlaştırmaktadır.

Şüpesiz her türlü dosya sisteminde cluster yada blok kavramı kullanılmak zorunda değildir. Örneğin read-only medyalardaki dosya sistemlerinde dosyaların parçalı yerleştirilmesinin bir anlamı yoktur. Çünkü dosyalar silinip yerine başka dosyalar yaratılmayacaksa pek ala dosyalar ardışıl bir biçimde depolanabilir. Cd lerdeki ve teyip bantlarındaki durum böyledir.

21.8. Çeşitli İşletim Sistemlerinin Çeşitli Disk Organizasyonları

Çeşitli işletim sistemleri diski kendilerine göre organize edip düzenlemeleri oluşturmuştur. Microsoftun dos işletim sistemin dosya sistemine fat sistemi denilir. Daha sonra Microsoft bu dosya sistemini biraz genişletmiştir ve Windows sistemlerinde kullanılan bu genişletilmiş FAT sistemine VFAT denilmektedir. Daha sonra Microsoft NTFS diye isimlendirilen çok ayrıntılı ve geniş dosya sistemini oluşturmuştur. Bugünkü Windows sistemleri tüm bu dosya sistemleri tarafından organize edilmiş olan diskler üzerinde işlem yapabilmektedir.

Unix türevi sistemlerin dosya sistemleri birbirinden farklı olsada birbirlerine benzemektedir. Bütün bu dosya sistemlerine kısaca inode sistemler denilmektedir. İlk unix sistemlerinde UFS diye isimlendirebileceğimiz klasik bir dosya sistemi kullanılıyordu. BSD türevi sistemler FFS (Fast File System) ismi ile bu dosya sistemine eklemeler yapmıştır. Linux sistemlerinde ext, ext2, ext3, reiser fs biçiminde isimlendirilen fakat bu sistemlere çok benzeyen dosya sistemleri kullanılmaktadır. Bunlar dışında irili ufaklı pek çok dosya sistemide vardır. Örneğin OS/2, HPFS IBM tarafından uzun süre kullanılmıştır. Cd lerde ve DVD lerde kullanılan ISO 9660 ve türevleri hemen her işletim sistemi tarafından desteklenmektedir.

21.9. Disk Organizasyonları Fat Türevi Dosya Sistemlerinin Disk Organizasyonları

Bugün hala Microsoft tarafından desteklenen Microsoft un fat sistemi basit ve temel bir disk organizasyonudur. VFAT bunun üzerine bazı eklemeler yapmıştır. Fakat temel mantık aynıdır. NTFS yine FAT tablosu kullanmakla birlikte oldukça karışıktır ve hala tam olarak Microsoft tarafından dökümente edilmemiştir. O halde bir disk organizasyonun ayrıntıları için incelenecek en temel sistem FAT dosya sistemidir.

Bir diski FAT dosya sistemi ile formatladığımızda diskte 4 bölüm oluşur.

- 1) Boot Sektör : 512 byte uzunluğundadır (1 sektör uzunluğunda) ve diskin tüm parametrik bilgileri bu sektörde tutulur.
- 2) FAT : (File Allocation Table) hangi dosyaların hangi parçalarının hangi clusterlarda olduğu bilgisini tutan bilgidir. FAT dosya sistemi bu tablo sisteminin yapısına göre FAT12, FAT16 ve FAT32 olarak 3 biçime ayrılır.

- 3) Root Dir : Bu bölüm kök dizindeki dosyaları tutan bölümdür.
- 4) Data : Dosya parçalarının depolandığı ana bölümdür. Data bölümündeki cluster lar yukarıda açıklandığı gibi numaralandırılmıştır.

Anahtar Notlar

Bir diski manuel olarak incelemek için disk editörleri denilen özel programlar kullanılır. Bu programlar sayesinde disk sektör sektör incelenebilir ve dosya sistemine bağlı olan pek çok disk bölümü incelenebilir. Disk editör programlarının en ünlülerinden birisi Norton isimli programdır fakat uzun süredir güncellenmemiştir. Yine WinHex isimli program Disk Editör olarak kullanılabilir.

Anahtar Notlar

Bir hard disk e tamamen birbirlerinden bağımsız birden fazla işletim sistemi ve dolayısıyla dosya sistemi kurulabilir. Bir hard disk in birbirinden bağımsız sanki ayrı bir diskmiş gibi kullanılan her bir bölümüne bir disk bölümü (partitian) denilmektedir. Örneğin biz bir disk bölümüne window su diğerine Linux u kurabiliriz. Bu iki sistem birbirleri ile karışmaz. Bir işletim sistemi diğerinin disk alanını bozmaz. Hard disk in ilk sektöründe disk bölünme tablosu (partitian table) denilen bir tablo vardır. Bu tablo da tüm disk bölümlerinin nereden başladığı ve ne kadar uzunlukta olduğu bilgisi bulunmaktadır. Her disk bölümüne farklı bir işletim sistemi kurulacağı gibi hiç işletim sistemi kurmadan sadece organizasyon yapılabilir. Yani örneğin biz hard diskimizi iki disk bölümüne ayırıp ikisini NTFS olarak organize edebiliriz. Windows sisteminde birinci partitian a yükleyebiliriz.

21.10. Boot Sektör

Boot Sektör fat dosya sisteminin ilk sektörüdür. Fat dosya sistemi ile formatlanmış her disk bölümünün ayrı bir boot sektörü vardır. Boot Sektör ilgili volmun ilk sektörüdür. Şüpesiz flopy disketlerde bölümlendirme kavramı olmadığı için Boot Sektör flopy sektörün ilk sektörü durumundadır.

Boot sektör kendi içersinde 2 bölümden oluşur.

BPB (Bios Parameter Block)

Yükleyici Program

BPB FAT dosya sisteminin kalbidir. Burada bir tablo vardır ve bu tablo dosya sisteminin bütün parametrik bilgilerini tutar. (Örneğin biz bir disk editörle 40-50 byte uzunluğunda olan BPB tablosunu bozarsak artık işletim sistemi bu disk bölümüne erişemez.). Boot sektör BPB bloğunun formatı şöyledir :

Offset Hex(decimal)	Uzunluk	Anlam
0(0)	3 BYTE	Imp Code
3(3)	8 BYTE	OEM yorum
B(11)	WORD	Sektördeki byte sayısı
D(13)	BYTE	Clusterdaki sektör sayısı
0E(14)	WORD	Ayrılmış sektörlerin sayısı
10(16)	BYTE	FAT kopyalama sayısı
11(17)	WORD	Root girişlerinin sayısı
13(19)	WORD	Toplam sektör sayısı
15(21)	BYTE	Ortam belirleyicisi
16(22)	WORD	FAT in bir kopyasının sektör uzunluğu
18(24)	WORD	Sektör dilimlerinin sayısı
1A(26)	WORD	Kafa sayısı

1C(28)	DWORD	Saklı sektör sayısı
20(32)	DWORD	Yeni toplam sektör sayısı
27(39)	8 BYTE	Volume serial number
2B(43)	11 BYTE	Volume label

Jmp Code

BPB

Yükleyici program

Jump code : Burada BPB bölümünü atlayarak yükleyici programa dallanan intel 80x86 işlemcilerine ilişkin bir jump komutu bulunur.

OEM yorum : Buradaki 8 byte dikkate alınmamaktadır format programları genellikle bu alana işletim sisteminin versiyon numarasını yazmaktadır.

Sektördeki byte sayısı : Burada disk sektörünün kaç byte dan oluştuğu bilgisi yer almaktadır. Tipik olarak burada 512 değeri bulunmaktadır.

Cluster daki sektör sayısı : Burada bir cluster ın kaç sektörden oluştuğu bilgisi yer almaktadır.

Ayrılmış sektörlerin sayısı : Burada boot sektörden sonra, boot sektörde dahil olmak üzere FAT sektörüne kadar kaç boş sektörün bulunduğu bilgisi vardır. Başka bir deyişle burada FAT bölümünün başlangıç mantıksal sektör numarası bulunur. Örneğin burada 1 değerinin bulunması boot sektörden hemen sonra FAT bölümünün geldiğini belirtir. Yani arada boşluk yoktur. FAT bölümünün bazen sonraki track den başlatılması tercih edilebilir.

FAT kopyalama sayısı : FAT bölümü önemli olduğu için onun birden fazla özdeş kopyasının tutulmasının faydalı olacağı sanılmıştır. Fakat birden fazla FAT dosyasının ciddi bir

faydası görülmemiştir. Microsoft eskiden beri FAT in 2 kopyasını bulundurmaktadır. Dolayısıyla bu alanda 2 sayısı görülecektir.

Root girişlerinin sayısı : Rootdir bölümünde kök dizindeki dosya bilgileri yer alır. Her dosya bilgisi 32 byte lık girişler ile ifade edilmektedir. Bu bölgede aslında dolaylı olarak rootdir bölümünün sektör uzunluğu belirtilmektedir. Burada yazılan sayı 32 ile çarpılıp 512 ye bölünürse rootdir bölümünün sektör uzunluğu elde edilir.

Toplam sektör sayısı : Burada eskiden volum ün kaç sektörden oluştuğu bilgisi yer almaktaydı. Maalesef Microsoft 1981 yılında bu formatı (BPB) tasarladığında disk lerin bu kadar büyüyeceğini düşünmemiş. Bu nedenle toplam sektör sayısı için 2 byte lık yer ayırmıştır. Gerçektende 1986/87 ye kadar, yani dos 4.01 versiyonuna kadar sırf bu yüzden volume deki sektör sayısı 65535 ile sınırlanmıştır. 40Mb ve 60Mb lık disklerin çıkmış olmasına karşın dos 3.30 da ve daha öncesinde bir disk bölümü 33Mb ı geçemiyordu. Çünkü $2^9 \times 2^{16} = 2^{25} \cong 33\text{Mb}$. Fakat daha sonra Microsoft dos 4.01 ile toplam sektör sayısı alanını 4 byte a yükseltmiştir. Geçmişe doğru uyumu korumak için BPB nin 20h ofsetin de yeni bir 4 byte lık toplam sektör sayısı alanı oluşturmuştur. Bugün işletim sistemleri önce 13h ofsetine bakmaktadır, eğer buradaki değer 0 (sıfır) ise 20h ofsetindeki 4 byte ı dikkate almaktadır.

Ortam belirleyicisi : Burada ki 1 byte lık sayı volum ün nasıl bir medya içerisinde olduğunu belirtmektedir. Örneğin F8 hard disk anlamına gelmektedir.

FAT in bir kopyasının sektör uzunluğu : FAT in bir kopyasının sektör uzunluğu rootdir alanın başlangıcının belirlenmesi içinde gerekmektedir. FAT bölümü ileride ele alınacaktır.

Sektör dilimlerinin sayısı : Burada toplam sektör dilimlerinin sayısı yer almaktadır.

Kafa sayısı : Burada disk in kaç kafaya sahip olduğu yani kaç kafaya sahip olduğu bilgisi yer almaktadır.

Saklı sektör sayısı : Saklı sektör kavramı ciddi anlamda kullanılmamaktadır. Saklı sektör aslında ilgili volume ün boot sektörünün yerini belirlemekte kullanılmaktadır. Yani boot sektörden önce diskte ki sektör sayısı burada yer alıyor olsada bu alanın ciddi bir kullanımı yoktur.

Yeni toplam sektör sayısı : Burada eskiden 2 byte ile ifade edilen toplam sektör sayısının 4 byte lık yeni biçimi bulunmaktadır.

Volume serial number : Disk formatlandığında rastgele üretilmiş olan 8 byte lık bir değer boot sektöre yazılmaktadır. Şüpheşiz bu değer başka bilgisayarlardaki disklerde ki değerlerle

çakışabilir. Genellikle disketlerde disket deęiřtirmeler sırasında olası hataları engellemek için kullanılmıřtır.

Volume label : Volume label hem kök dizinde bir dosya ismi olarak hemde boot sektörde giriř olarak bulunmaktadır.

21.11. Sektör okuma yazma işlemleri

Makine ilk açıldığında henüz işletim sistemi bile yüklü deęilken, sektör okuyup yazma işlemleri için, makine dilinde yazılmış olan BIOS fonksiyonları kullanılabilir. İşletim sistemi de bu fonksiyonları kullanarak okuma yazma yapmaktadır. İşletim sistemi yüklendikten sonra kendisi de sektör okuyup yazmaya yönelik mekanizmalar sunmaktadır. Kod korumalı modda çalışan işletim sistemlerinde, BIOS fonksiyonları doğrudan kullanıcı tarafından kullanılamaz. Artık bu sistemlerde işletim sisteminin sağladığı olanaklarla bu işlemler gerçekleştirilir.

Windows sistemlerinde aygıt sürücü yoluyla sektör okuma yazma işlemleri gerçekleştirilmektedir. Fakat 95 grubu sistemlerle, NT grubu sistemler farklı aygıt sürücüler kullanılmaktadır.

Unix/LINUX sistemlerinde tıpkı Windows sistemlerinde olduğu gibi dosya sistemleri de birer aygıt sürücü biçiminde sisteme dahil edilmektedir. Dolayısıyla NT grubu sistemlerde olduğu gibi ilgili aygıt sürücü dosyası açılıp, read-write işlemleri yapıldığında, zaten sektör okuyup yazma işlemleri gerçekleştirilmiş olur.

Kursumuzda Windows sistemleri altında aygıt sürücü çağırması yapılarak sektör okuma yazma işlemleri c++ da bir sınıf biçiminde oluşturulmuştur. Ayrıca bu sınıf kullanılarak C den çağrılacak sarma fonksiyonlar da oluşturulmuştur. Kursumuzda yazılmış olan fonksiyonları kullanabilmek için öncelikle OpenDisk fonksiyonunun çağırılması gerekir. Prototipi şöyledir.

```
HWIN32DISK OpenDisk(int volume);
```

Fonksiyonun parametresi açılacak volume ü belirtir. A = 0; B = 1; ... gibi. Fonksiyon başarı durumunda handle değerine, başarısızlık durumunda NULL adrese geri döner. Volume ü kapatmak için CloseDisk fonksiyonu kullanılır.

```
void CloseDisk(HWIN32DISK hw32Disk);
```

Parametre olarak handle değerini almaktadır. Sektör okuma için ReadSector fonksiyonu kullanılır.

```
BOOL ReadSector(HWIN32DISK hw32Disk, DWORD dwStart, WORD  
wNumberOfSectors, PVOID pBuf);
```

Fonksiyonun birinci parametresi HANDLE değeri, ikinci parametresi başlangıç mantıksal sektör numarasını belirtir. Üçüncü parametre okunacak sektör sayısını belirtir. Son parametre tranfer edilecek yerin adresidir.

Write sector fonksiyonu da aynı parametrik yapıya sahiptir.

```
BOOL WriteSector(HWIN32DISK hw32Disk, DWORD dwStart, WORD  
wNumberOfSectors, LPCVOID pBuf);
```

Kullanım şöyle olabilir.

```
HWIN32DISK hw32Disk;  
unsigned char boot[512];  
  
if ((hw32Disk = OpenDisk(5)) == NULL) {  
    fprintf(stderr, "Cannot open file!...\n");  
    exit(EXIT_FAILURE);  
}  
  
if (!ReadSector(hw32Disk, 0, 1, boot)) {  
    fprintf(stderr, "Cannot read disk!...\n");  
    exit(EXIT_FAILURE);  
}  
  
CloseDisk(hw32Disk);
```

21.12. BPB bölümünün Elde Edilmesi

FAT dosya sisteminde BOOT sektör-BPB bölümüm kritik önemlidir. Gerçekten işletim sistemi de her disk erişiminde buradaki bilgilerden faydalanır. Tabii her defasında BPB yi okumak yerine eğer harddisk sözkonusu ise açılış sırasında bir kez okumak daha anlamlıdır. BPB bilgilerini ele geçiren aşağıdaki gibi bir fonksiyon yazılabilir.

```
BOOL GetBPB(HWIN32DISK hw32Disk, BPB *pBPB);
```

Fonksiyonun birinci parametresi BPB bloğu elde edilecek olan diske ilişkin handle değerini belirtir. İkinci parametre BPB bilgilerinin yerleştirileceği yapının adresini alır. BPB yapısı ve GetBPB fonksiyonu eklerde verilmiştir.

Anahtar Notlar: p, BYTE cinsinden bir adres olmak üzere, p adresinden n byte ilerden bir word çekme işlemi şöyle yapılabilir.

$*(\text{WORD }*)(p + n)$

21.13. FAT Bölümü

FAT (File Allocation Table), bir dosyanın hangi parçalarının hangi clusterlarda olduğunu tutan, dosya sisteminin en önemli bölümlerinden biridir. FAT bölümü 12bit, 16bit ya da 32bit olabilmektedir. Volume deki toplam sektör sayısı belirli bir değeri aşıyorsa, 12bit yerine 16bit FAT kullanılır. Yine belirli bir değerün üstünde 32bit FAT kullanılmaktadır. FAT bölümünün hangi sektörden başladığı BPB bölümündeki, saklı sektörlerin sayısı alanı ile belirlenmektedir. Yine BPB'de FAT in kaç kopya olduğu ve bir kopyasının kaç sektörden oluştuğu BPB alanında yazmaktadır. getBPB fonksiyonu bütün bu değerleri düzenli olarak verecek biçimde yazılmıştır. Böylelikle programcı tüm FAT alanını readSector fonksiyonu ile okuyabilir.

Disk analiz programları ve İ.S. Nin kendisi FAT alanının tamamını belleğe okuyamayabilir. Bu durumda bir cache sisteminin oluşturulması gerekir. Örneğin Linux FAT dosya sistemini gerçekleştirirken FAT elemanlarından bir cache oluşturmakta, böylece gereksiz FAT okumalarını mümkün olduğunca engellemektedir. İlgili FAT elemanına erişmek için bir HASH tablosu kullanılmaktadır. Yeni Linux da bu cache sistemi bir hash tablosu biçiminde organize edilmiştir.

FAT tablosu FAT elemanlarından oluşmaktadır. 12 bit FAT de bir FAT elemanı 12bit (1.5 byte), 16bit FAT de 16bit(2 Btye), 32bit fat de 32 bit (4 byte) dir. FAT tablosundaki her fat elemanına bir numara verilmiştir.

FAT tablosu her dosya için ayrı bir bağlı liste içeren bir tablodur. Dosyanın hangi clusterlarda olduğu şöyle tespit edilir:

Dosyanın ilk cluster numarası dizin girişinden elde edilir. örneğin burada 3 yazsın. Sonra 3 numaralı FAT elemanına bakılır. Dosyanın sonraki cluster numarası burada yazar. 6 olsun. 6

numaralı fat elemanına bakılır. Burada 7 yazsın.7 numaralı FAT elemanında özel bir değer vardır. Bu son fat elemanın belirtir. Bu durumda bu dosyanın cluster zinciri 3-6-7 den oluşur.

FAT Tablosu

6
7
EOF

görüldüğü gibi FAT tablosundan birbirinden bağımsız bağlı liste biçiminde dosyas sayısı kadar zincir vardır. Şüphesiz bu sistemde dosyanın cluster zincirini elde etmek için onun başlangıç cluster numarasının biliniyor olması gerekir. Başlangıç cluster numarasının eldesi daha sonra anlatılacaktır. FAT elemanlarının sayısı, data bölümündeki cluster sayısı kadardır. Dolayısıyla formatlama sırasında işletim sistemi volume de kaç sektör olduğunu hesaplar. Buradan data bölümünde kaç cluster olduğunu elde eder ve FAT bölümünün sektör uzunluğuna bu biçimde karar verir.

Hangi clusterların boş olduğuna yine FAT tan bakılarak karar verilmektedir. Bir FAT elemanında 0 değeri varsa, bu durum o fat elemanının dolayısıyla o cluster ın boş olduğu anlamına gelir. Dolayısıyla, diskteki boş alan miktarı boş FAT elemanlarının sayısı ile hesaplanabilmektedir.

Bir FAT elemanındaki değerler ne ifade eder ? FAT16 da fat elemanındaki değerler şu anlama gelmektedir.

Değer	Anlamı
0000	Boş cluster
0001	Kullanılmıyor
0002 – FFEF	Sonraki cluster
FFF0 – FFF6	Kullanılmıyor
FFF7	Bozuk cluster
FFF8 – FFFF	Son cluster (EOF)

12bit fat de ise durum şöyledir.

Değer	Anlamı
000	Boş cluster
001	Kullanılmıyor
002 – FEF	Sonraki cluster
FF0 – FF6	Kullanılmıyor
FF7	Bozuk cluster
FF8 – FFF	Son cluster (EOF)

32bit fat burada ele alınmayacaktır.

0 ve 1 numaralı clusterlar kullanılmadığı için DATA bölümünün ilk cluster ı 2 numaradır. Bu nedenle ilk 2 FAT elemanı dikkate alınmamalıdır. 16 bit fat de ilk 2 fat elemanında <med des> FF FF FF

12 bit FAT de

<med des> FF FF

Değerleri bulunur. Burdak, med des boot sektörde de bulunan ortam belirleyici byte dir.

21.14. Fat Elemanlarından bir Catch sisteminin oluşturulması

Bir cache sistemi tipik olarak bir hatch tablosu biçiminde oluşturulabilir. Böylece bu catch sistemine dayanarak bir Fat elemanından sonraki elaman okunabilir. Bu modül test amaçlı fatcatch.h ve fatcatch.c dosyaları biçiminde oluşturulacaktır. Hash tablosundaki her bir düğüm aşağıdaki gibi organize edilebilir.

```
typedef struct tagNODE {
    WORD clu;
    WORD nextClu;
    struct tagNODE *pNext;
}NODE;
```

Bu durumda hatch tablosu aşağıdaki gibi kurulabilir.

```
Node *hashTable[TABLE_SIZE]
```

Bizim bu catch sisteminde bir cluster verildiğinde sonraki clusterın numarasını veren bir fonksiyona gereksinimimiz olacaktır.

```
WORD GetNextClu(const BPB *pBPB, WORD clu);
```

Burada fonksiyon BPB yapısının adresini ve sonraki cluster ı parametre olarak almıştır. Fonksiyon sonraki cluster numarasına geri dönmektedir. Buradadisk için gereken HWIN32DISK handle BPB yapısının içine yerleştirilebilir. GetNextClu fonksiyonu şöyle yazılabilir:

```
WORD GetNextClu(const BPB *pBPB, WORD clu)
{
    int index;
    NODE *pNode;
    BYTE sector[512];
    int fatSector, sectorOffset;
    WORD nextClu;
    NODE *pNewNode;

    /*HASH TABLE LOOKUP*/

    index = HashFunc(clu);
    pNode = g_hasTable[index];

    while (pNode != NULL) {
        if (pNode -> clu == clu)
            return pNode -> nextClu;
        pNode = pNode -> pNext;
    }

    /*Catch miss! read from fat*/

    fatSector = clu * 2 / 512;
    sectorOffset = clu % 256 * 2;

    if (!ReadSector(pBPB -> hw32Disk, Pbpb -> fatOrigin +fatSector, 1,
sector))
        return 0;

    nextClu = *(WORD *) (sector + sectorOffset);
```



```

/*Insert hash table*/

if ((pNewNode = (NODE *) malloc (sizeof(NODE))) == NULL)
    return 0;

pNewNode -> clu = clu;
pNewNode -> nextClu = nextClu ;
pNewNode -> pNext = g_hashTable[index];
g_hashTable[index] = pNewNode;

return nextClu;
}

```

Buradaki catch sisteminin bazı tipik problemleri vardır :

- 1) Hash tablosu ve catch sistemi sınırsız olarak büyümektedir. Halbu ki bu durum kesinlikle istenmemektedir. Gerçek uygulamalarda tablodaki eleman sayısı belirli miktara eriştiğinde tablodan eleman çıkartılması gerekir. Peki tablodan hangi elemanlar çıkartılacaktır? Genellikle bu tür durumlarda LRU (Least Recently Used) algoritması kullanılır. Bu algoritmada bir bağlı liste alınır, eleman kullanıldıkça bağlı listenin önüne alınır. Böylece bağlı listenin sonunda son zamanlarda en az kullanılan elamenlar kalmış olur. Cache den eleman çıkarılacağı zaman bağlı listenin sonundan çıkartılmış olur.
- 2) Yukarıdaki örnekte düğüm yapısı kolaylık olsun diye tek bağlı liste biçiminde alınmıştır. Halbu ki cash ten eleman silinebileceğine göre, düğümlerin çift bağlı liste olması silme sırasında avantaj sağlar.

Bu durumda aslında Node yapısının aşağıdaki gibi olması daha gerçekçidir:

```

typedef struct tagNODE {
    WORD clu;
    WORD nextClu;
    struct tagNODE *pNext;
    struct tagNODE *pPrev;
    struct tagNODE *pLRUNext;
    struct tagNODE *pLRUPrev;
}NODE;

```

Görüldüğü gibi bir düğüm hem hash tablosunda hemde LRU listesinden referans edilmektedir. Burada eleştirilebilecek bir nokta tek bir cluster numarası için büyük bir düğümün tahsis edilmesidir. Pratikte dosyanın cluster numaraları büyük oranda peşi sıra gittiğine göre bir düğümde birden fazlada cluster numarasıda tutulabilir. Yani bir düğüm tek bir clusterı tutmayabilirden birden fazla cluster'ı tutabilir. Bu durumda arama yine benzer biçimde yapılabilir.

Tabi aynı düğümdeki cluster ların aynı hash değerlerini vermesi gerekebilir. Fakat buda basit olarak sağlanabilir.

Şüpesiz yukarıdaki GetNextClu fonksiyonu FAT16 için mevcuttur. Fat12 için farklı olacaktır. Dosyanın cluster zincirinin elde edilebilmesi için aşağıdaki gibi bir fonksiyon yazılabilir:

```
WORD *GetCluChain(const BPB *pBPB, WORD clu, int *pCount);
```

Fonksiyon 1.parametresi BPB yapısını, 2.parametresi dosyanın ilk cluster numarasını almaktadır. 3.parametre cluster zincirindeki eleman sayısını belirtmektedir. Fonksiyon cluster zincirinin adresi ile geri dönmektedir. Fonksiyon geri verdiği adres içerde malloc ile tahsis edilmiştir. Fonksiyonu çağırın kişi free etmelidir.

```
#define CHAIN_BLOCK_SIZE          32
WORD *GetCluChain(const BPB *pBPB, WORD clu, int *pCount)
{
    WORD *pChain = NULL;
    int i;

    for (i = 0; clu < 0xFFFF8; ++i)
        if (i % CHAIN_BLOCK_SIZE == 0) {
            pChain = (WORD *) realloc(pChain, ((i + CHAIN_BLOCK_SIZE) *
sizeof(WORD)));
            if (pChain == NULL)
                return NULL;
        }
        pChain[i] = clu;
        clu = GetNextClu(pBPB, clu);
    }
    *pCount = i;
    return pChain;
}
```

21.15. Cluster Okuma Yazma İşlemleri

İşletim sistemi düzeyinde artık sektör okuyup yazmaktan ziyade cluster düzeyinde okuma yazma işlemleri daha yaygındır. O halde cluster okuyup yazan fonksiyonlara gereksinimimiz vardır. Cluster okuyup yazan fonksiyonlar şöyle olabilir:

```
BOOL ReadCluster (const BPB *pBPB, DWORD cluNo, void *pClu);
BOOL WriteCluster (const BPB *pBPB, DWORD cluNo, const void *pClu);
```

1. parametresi BPB yapısını, 2.parametresi cluster numarasını almaktadır. 3.parametre transfer adresidir. ReadCluster ve WriteCluster fonksiyonları şöyle yazılabilir:

```
BOOL ReadCluster (const BPB *pBPB, DWORD cluNo, void *pClu)
```

```

{
    DWORD sectNo = pBPB->dataOrigin + (cluNo - 2) * pBPB->spc;
    return ReadSector(pBPB->hW32Disk, sectNo, pBPB->spc, pClu);
}

BOOL WriteCluster (const BPB *pBPB, DWORD cluNo, const void *pClu)
{
    DWORD sectNo = pBPB->dataOrigin + (cluNo - 2) * pBPB->spc;
    return WriteSector(pBPB->hW32Disk, sectNo, pBPB->spc, pClu);
}

```

21.16. Rootdir Bölümü ve Dizin Organizasyonu

Fat12 ve Fat16 sistemlerinde kök dizindeki tüm dosyaların her biri 32 byte lık kayıtlar biçiminde rootdir bölümünde tutulmaktadır. Yani rootdir bölümünün organizasyonu şöyledir:

Dizin giriş	32 byte
-----	32 byte
Dizin giriş	32 byte
-----	32 byte
Dizin giriş	32 byte

Dizin giriş	

.....	

Rootdir bölümünde yalnızca kök dizindeki bilgileri vardır. Dizin girişinde bir dosyanın ismi,uzantısı, yaratım tarihi, uzunluğu ve dosyanın ilk clusterının numarası bilgileri bulunmaktadır. İşletim sistemi bir yol ifadesi verildiğinde dosyanın ilk clusterını dizin girişinden elde etmektedir. Dizin girişinin formatı şöyledir.

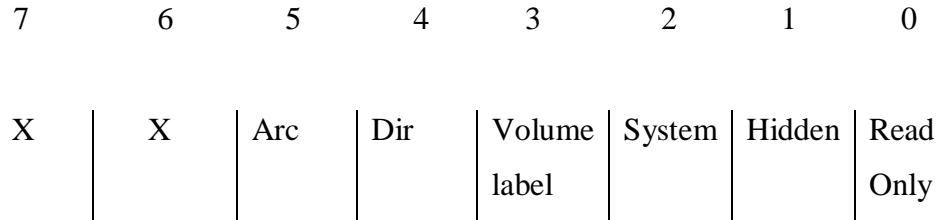
Offset hex(decimal)	Uzunluk	Anlamı
0(0)	8 BYTE	Dosya ismi
8(8)	3 BYTE	Uzantı
B(11)	1 BYTE	Özellik

C(12)	2 BYTE	Rezerved
E(14)	WORD	Yaratım zamanı
10(16)	WORD	Yaratım tarihi
12(18)	8 BYTE	Rezerved
1A(26)	WORD	İlk cluster no
1C(28)	DWORD	Dosya uzunluğu

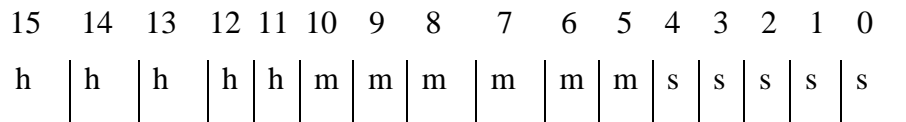
Dosya ismi : Dosta klasik olarak dosya isimleri 8 karakterdir. Bu limit Windows sitemleri ile 260 karaktere uzatılmıştır.

Uzantı : Dosya uzantısı klasik olarak dosta 3 byte dır. Uzantı kavramı kaldırılmış, dosya ismi birden fazla nokta içerebilir.

Özellik : Daha öncede _findfirst ve _findnext fonksiyonlarında görüldüğü gibi dosya özelliği bir byte içersinde bitlerden oluşmuş durumdadır. Bitlerin anlamı şöyledir:



Yaratma zamanı : Dosya zamanı bit olarak aşağıdaki gibi kodlanmıştır:



Görüldüğü gibi saniye için alan yetmediği için 5 bit verilmiştir. İşletim sistemi buraya saniye değerinin yarısını yazmaktadır. Yani örneğin saniye değeri 50 ise, işletim sistemi buraya 25 yazar fakat 2 ile çarpıp gösterir. Yani tek saniyeler tutulmamaktadır.

Yaratım Tarihi : Burada 2 byte içerisinde bit-bit kodlanmış bir biçimde yaratım tarihi bulunur.

Yıl için epoch 1981. Dolayısıyla burada yazılan değere 1980 eklenmelidir.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
y	y	y	y	y	y	y	m	m	m	m	s	s	s	s	s

Anahtar Notlar

Dos işletim sistemi yani Fat dosya işlemi yalnızca dosyanın yaratım tarihi ve zamanını tutmaktadır. Halbuki NTFS ve Unix/Linux sistemleri son değiştirme tarih ve zamanlarını, son erişim tarih ve zamanını da tutmaktadır.

İlk cluster no : Dosyanın ilk cluster numarası dizin girişinde yazmaktadır. Diğer cluster lar Fat zincirinden elde edilir. Yani işletim sistemi önce dosyaya ilişkin dizin girişini elde etmektedir.

Dosya uzunluğu : Burada 4 byte lık bir alanda dosya uzunluğu bulunur.

21.17. Alt Dizinlerin Organizasyonu ve Yol İfadesinin Çözülmesi İşlemi

Root-dir bölümünde yalnızca kök dizindeki dosyalara ilişkin bilgiler bulunmaktadır. Aslında sıradan bir dosya ile sıradan bir dizin arasında hiçbir farklılık yoktur. Her ikisinde bir fat zincirine sahip olan dosyalardır. Bir girişin sıradan bir dosyayı yoksa dznmi olduğu 32 byte dizin girişindeki özellik byte ından anlaşılmaktadır.

Bir yol ifadesi verildiğinde işletim sistemi o yol ifadesinin sonundaki dosyanın 32 byte lık dizin girişine erişmek ister. Buna yol ifadesinin çözülmesine(pathname resolution) denir. Örneğin \a\b\c.dat dosyasında c.dat dosyasının dizin girişini bulmak isteyelim. Bu mutlak yol ifadesi şöyle çözülmelidir.

- 1) İşletim sistemi önce kök dizinde yani rootdir bölümünde 32 byte lık girişlerde a ismini arar. a dosya ismini bulursa, onun bir dizin olduğunu doğrular sonra onun cluster zincirinde 32 byte lık girişlerde b yi arar.
- 2) b ismini bulursa onun cluster numarasından faydalanarak onun cluster zincirindeki 32 byte lık girişlerde c.dat dosyasını arar.

Modern işletim sistemleri yol ifadesinin çözülmesi sırasında dizin girişlerinden bir cache sistemi oluşturmaktadır. Böylece aynı dizinin yada dosyanın yeniden aranacağı zaman gereksiz disk okumaları yapılmamış olur.

21.18. Formatlama İşlemi

Formatlama bir volume ün kullanıma hazır hale getirilmesidir. Formatlama iki aşamada yürütür.

- 1) Aşağı seviyeli formatlama
- 2) Yukarı seviyeli formatlama

Aşağı seviyeli formatlama sırasında donanımın diski kullanabilmesi için çeşitli işlemler yapılır. Aşağı seviyeli formatlama işletim sistemine bağlı değildir ve hangi işletim sistemi kullanılıyor olursa olsun kesinlikle yapılmak zorundadır. Harddisk lerde aşağı seviyeli formatlama üretici firma tarafından işin başında yapılmış durumdadır. Kullanıcıların aşağı seviyeli formatlama yapması tavsiye edilmemektedir. Dolayısıyla örneğin Windows da harddisk i formatlamak istediğinizde söz konusu olan formatlama aşağı seviyeli formatlama değildir, yüksek seviyeli formatlamadır. Ancak floppy disketlerde aşağı seviyeli formatlama yapılabilir.

Yüksek seviyeli formatlamada yalnızca diskin organizasyonel bölümleri uygun değerlerle doldurulur. Yüksek seviyeli formatlama işletim sisteminden işletim sistemine göre (dosya sisteminden dosya sistemine) değişmektedir. Örneğin Fat dosya sisteminde yüksek seviyeli formatlama sırasında boot sektör bilgileri oluşturulur, Fat bölümü sıfırlanır rootdir bölümüde tam olarak sıfırlanır. Aslında data bölümünün ayrıca sıfırlanmasına gerek yoktur. Windows yüksek seviyeli formatlamayı hızlı(quick) ve yavaş olmak üzere ikiye ayırmıştır. Hızlı formatlamada başka ilave bir işlem yapılmaz. Fakat yavaş formatlamada tek tek sektörler verify işlemine sokulmaktadır. Verify işlemi sektörlerin tek tek muayine edilmesi anlamına gelmektedir. Böylece bozuk sektörlerin ilişkin olduğu cluster lar “bad cluster” olarak işaretlenir. Hızlı formatlama işleminde data bölümüne dokunulmaz ve verify işlemi yapılmaz. Ancak yavaş formatlama işleminde tüm data bölümüde formatlamaktadır.

21.19. Fat Dosya Sistemine İlişkin Disk Organizasyon Bozuklukları

Bir disk bozukluğu fiziksel yada organizasyonel olabilir. Örneğin disk kafalarının bozulması, diskin çizilmesi gibi sorunlar ciddi sorunlardır ve genellikle tedavisi mümkün değildir. Fakat disk bozukluklarının çoğu organizasyoneldir. Diskteki bilgilerin kaybedilmiş olduğu kabul edilirse, disk yeniden formatlanarak kullanılabilir.

Fat dosya sisteminde çok sık karşılaşılan önemli oragnizasyon problemleri şunlardır.(Buradaki bozukluğu anlatan ifade check-disk ve scan-disk programlarının raporlarına göre belirtilmiştir.)

1) General error reading/writing : Microsoft'un disk onarıcı programları boot sektör BPB bloğu içerik bakımından bozuk ise disk üzerinde bir işlem yapmaz. Norton'un klasik NDD programı yada Final Data programı bu durumda bozuk BPB yi onarmaktadır. Bu hata BPB bloğu içerik bakımından bozuk olduğunda ortaya çıkar.

2) Bad Fat : Eskiden Microsoft Fat bölümünün ilk iki elemanını Fat'in sihirli numarası olarak kontrol ediyordu. Eğer Fat'in ilk iki elemanında yukarıda açıklanan özel değerler yoksa Fat üzerinde işlemler yapmıyordu. Fakat Windows un yeni versiyonları buna hiç bakmamaktadır.

3) Lost Cluster : Bu bozuklukta Fat de bir zincir vardır fakat bu zinciri gösteren volume de hiçbir dizin girişi yoktur. Genellikle elektrik kesintileri sırasında oluşmaktadır. Onarıcı programlar genellikle bu zinciri gösteren yeni bir dizin girişi yaratırlar.

4) Coss Linked Files : Bu bozuklukta birden fazla Fat zinciri belirli bir noktadan sonra birleşerek tek bir zincir olarak devam etmektedir. Dosyalardan biri silindiğinde diğeri de sorunlu hale gelir. Bu bozuklukta elektrik kesilmeleri gibi nedenlerle oluşmaktadır. Cross Linked Files durumu genellikle Lost Cluster durumu ile birlikte görülmektedir. Onarıcı programlar genellikle zincirin çaprazlaştığı yerden itibaren ortak kısmın dosyaya özgü yeni kopyasını oluşturmaktadır.

5) Allocation Error : Bu bozuklukta dosyaya ilişkin cluster zincirinin belirttiği uzunluk ile dizin girişinde ki uzunluk örtüşmemektedir. Örneğin bir clusterın 2 sektör olduğu sistemde dosyanın cluster zincirinin 5 cluster olduğunu düşünelim. Bu dosya 9 * 512 ile 10 * 512 arasında bir uzunlukta olmalıdır. Fakat bu dosyanın uzunluğu için dizin girişinde örneğin 100 byte lkık gibi bir değer yazıyorsa bir tuhaflık söz konusudur. Onarıcı programlar genellikle dosyanın cluster uzunluğuna güvenip dizin girişini güncellemektedir.

21.20. Disk Bölümleme Tablosu ve Disk Bölümlerinin Anlamı

Harddiskler birden fazla dosya sistemi veya işletim sistemi yüklenebilsin diye birbirinden bağımsız bölümlere ayrılmışlardır. Her disk bölümü bağımsız bir harddisk bölümü gibi işletim sistemleri tarafından değerlendirilirler. Böylece biz bir disk bölümüne Windows u , bir disk bölümüne Linux u kurarak, istediğimiz işletim sistemi ile bilgisayarımızı açabiliriz.. Her disk bölümü ardışıl sektörlerden oluşur. Her disk bölümünün hangi sektörden başlayıp kaç sektör uzunlukta olduğu bir tabloda tutulur(bu tabloya partitian table, yani disk bölümleme tablosu denir.). Disk bölümleme tablosunun ilk bölümüne MBR(Master Boot Record) denilmektedir. Bu sektör fiziksel olarak sıfır numaralı sektördür. Disk bölümleme tablosu MBR nin sonundadır. MBR nin son 2 byte ı 55A sihirli sayılardır. İşte disk bölümleme tablosu bu sihirli sayının ncesindeki 64 byte dır. MBR nin başında yükleyici program vardır.

Yükleyici program

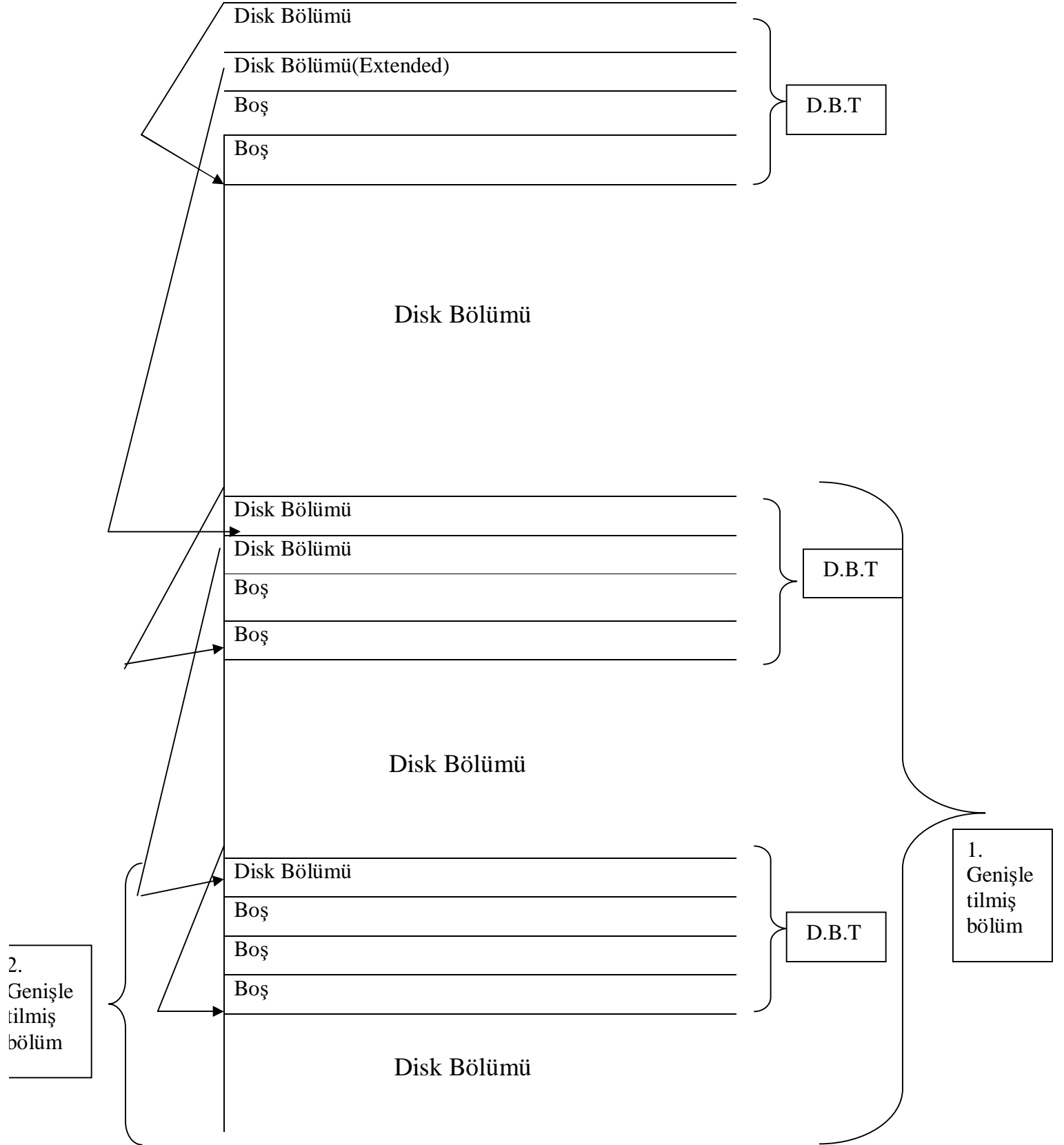
Disk bölümleme tablosu	
(64byte)	
	55A

Disk bölümleme tablosu 16 şar byte lık dört partition dan oluşmaktadır. . Her 16 byte lık kayıt bir disk bölümünün nereden başladığını, nerede bittiğini ve o disk bölümünde hangi dosyanın olduğunu bildirir. Her dosya sisteminin bir ID numarası vardır. Böylece disk bölümleme tablosuna bakan bir kişi orada hangi dosya sisteminin bulunduğunu anlayabilir.

Disk bölümleme tablosu 4 girişten oluştuğuna göre acaba dört den fazla disk bölümü oluşturulamaz mı ? İşte genişletilmiş(extended) disk bölümü kavramı ile bu durum mümkün hale getirilmiştir. Genişletilmiş disk bölümü bir disk bölümüdür fakat sanki ayrı bir harddisk gibi

değerlendirilmektedir. Genişletilmiş disk bölümünün ilk sektöründe de genişletilmiş disk bölümlene tablosu vardır. Örneğin;

DISK



Şuayrıntılar önemlidir ;

- Ana girişte en fazla 4 tane disk bölümü oluşturulabilir, fakat diskte 4 ten fazla disk bölümü oluşturmak istiyorsak, ana girişlerden bir tanesini genişletilmiş disk bölümü olarak yaratmalıyız.
- Bir disk bölümlene tablosunda birden fazla genişletilmiş disk bölümü bulunamaz.
- Genişletilmiş disk bölümündeki disk bölümlene tablosu 0, 1 yada 2 eleman içerebilir. Daha fazla eleman içeremez.
- Microsoft terminolojisine göre MBR deki disk bölümlene tablosunun ana girişlerine ilişkin disk bölümleri birincil disk bölümleri(primary partition) denilmektedir.
- Microsoft terminolojine göre genişletilmiş disk bölümlerindeki disk bölümlerine mantıksal olarak, mantıksal sürücüler(logical drives) denilmektedir.

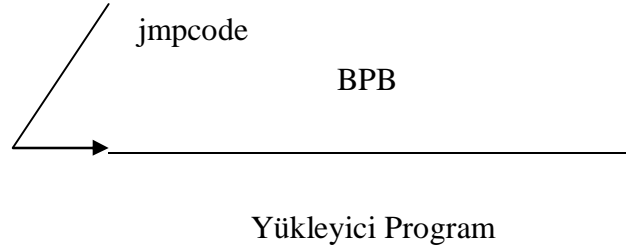
Örneğin biz 100Gb lik bir harddiskimizi Windows da 20 şer Gb lik 5 sürücüye ayırmak isteyelim. Bu işlem tipik olarak şöyle yapılmalıdır: Önce ana girişte 20Gb lik bir primary partiton yaratılır. Daha sonra geri kalan 80Gb lik alan extended partition olarak oluşturulmalıdır. Sonra bu extended partition içersinde mantıksal disk bölümleri oluşturulur. Her disk bölümü oluşturulduğunda, ilgili program disk bölümü içersinde yeni bir disk bölümü oluşturarak yukarıda belirtilen zinciri oluşturur.

21.21. İşletim Sistemimin Yüklenmesi ve Boot İşlemi

Bilgisayarımız açığımızda işletim sistemimizin yüklenmesi otomatik bir biçimde yapılır. İşletim sisteminin yüklenmesi bir dizi olaylar sonucunda gerçekleşir.

Bilgisayar açıldığında çalışma EPROM içersindeki kod dan başlatılır. Bu kod POST(Power On Self Test) işlemini yaptıktan sonra setup ta belirtilen sıraya göre(burada önce a sonra c olduğunu farz edelim) ilgili sürücünün ilk sektörünü, bellekte 7C00 adresine okur ve oraya jump eder. Görüldüğü gibi bilgisayarı açığımızda boot sektör içersindeki program belleğe yüklenerek çalıştırılmaktadır. Floppy disketlerde MBR diye bir sektör yoktur. Eğer söz konusu medya hard disk ise ilk sektör MBR dir, böylece MBR deki program çalıştırılmış olur. MBR deki

program disk bölümlenme tablosunu inceler, oradaki aktif disk bölümünü belirler. Aktif disk bölümünün ilk bölümünü belleğe yükleyerek bu kez oraya jump eder. Artık işletim sisteminin yüklenmesi aktif disk bölümünün ilk sektöründeki programa bırakılmıştır. Örneğin ilgili disk bölümünün ilk sektörü Fat sistemi içeriyorsa, Fat sisteminin boot sektöründeki program çalıştırılır. Anımsanacağı gibi boot sektörünün başında bir jump kod vardır. Bu jump BPB bölümünün üzerinden atlayarak yükleyici programa geçer.



Boot sektör içersindeki yükleyici program işletim sistemini yüklemektedir. Şüpesiz kücücük bir programın koskoca bir işletim sistemini yüklemesi beklenemez. Tipik olarak yükleme birkaç aşamada yapılmaktadır. Örneğin boot sektör içersindeki yükleyici program Linux ta asıl yükleyici programı belleğe yükleyerek oraya jump eder. Linux ta setup programı denilen bu yükleyici program kernal dosyalarını belleğe yükler ve oraya jump eder.

Bir disk bölümünden boot işlemi sağlayabilmek için o disk bölümünün aktif hale getirilmesi gerekmektedir. Çünkü MBR deki default program böyle çalışmaktadır. İşte bu işlemi görsel bir hale getirebilmek için boot-loader programları kullanılmaktadır. Boot-loader programlar birkaç biçimde organize edilebilmektedir.

- 1) Boot-loader program doğrudan MBR ye yüklenebilir. Bu durumda orijinal MBR programı bozulacaktır.(Windows da fdisk/MBR yazarsan orjinali yükler). Şüpesiz boot-loader program bir sektöre sığmayacağına göre, MBR den sonraki sektörleride kullanmaktadır. Zaten genellikle ilk disk bölümü MBR den hemen sonra değil birinci track in başına hizalanmıştır.
- 2) Orijinal MBR programı bozulmaz(bu yöntem tercih edilebilir). Boot-loader program aktif disk bölümünün boot sektörüne yerleştirilir.

Boot-loader nereye yerleştirilmiş olursa olsun, bir menü eşliğinde boot edilecek disk bölümünü sorar ve o disk bölümünün boot sektörünü yükleyerek akışı oraya devreder.

21.22. I-Node Dosya Sistemlerine İlişkin Disk Organizasyonun Genel Yapısı

Linux/Unix sistemlerindeki dosya sistemlerinin disk organizasyonu birbirine çok benzerdir. Bu disk organizasyonları Fat sisteminden oldukça farklıdır. Bu organizasyonlardan bugün için en fazla kullanılan ext2(second extended file system) organizasyonudur. Tipik bir I-node grubu sistemin disk organizasyonunda disk bölümü 3 bölüme ayrılır.

Disk Bölümü

Super Block

i-node block

Data Block

Super Block : Super Block Fat dosya sistemindeki BPB bölümüne benzemektedir. Burada disk bölümlerinin nereden başladığı, kaç sektör uzunlukta olduğu, bir bloğun kaç sektörden oluştuğu gibi temel bilgiler vardır.

I-node block : i-node block i-node elemanlarından oluşur. Her i-node elemanın ilk eleman sıfır olmak üzere bir numarası vardır. Dosyanın bütün bilgileri i-node elemanında saklanır. Her dosyanın bir i-node numarası vardır. Dosyanın i-node numarası o dosyanın bilgilerinin kaç numaralı i-node elemanında olduğunu belirtir. I-node dosya sistemlerinde dosyanın hangi

blocklarda(yada cluster larda) bulunduğu yine i-node elemanında belirtilmektedir. Yani ayrıca Fat gibi bir durum yoktur. Bir i-node elemanında kabaca şu bilgiler vardır:

- Dosyanın uzunluğu
- Erişim bilgileri
- Dosyanın userid ve grupid değerleri
- Dosyanın tarih ve zaman bilgileri
- Dosyanın türü
- Dosyanın parçalarının hangi blocklarda olduğu bilgisi

Data block : Data block Fat dosya sistemindeki data bölümü ile aynı anlamdadır. Bu dosya sisteminde de dizin ile dosya arasında bir fark yoktur. Bu dosya sisteminde de dizinler dizin girişlerinden oluşur. Bir dizin girişi dosyanın ismi ve i-node numarasından oluşmaktadır.

Dizin Dosyası

İsim	i-node numarası
İsim	i-node numarası
İsim	i-node numarası
....

Sıfır numaralı i-node elemanı kök dizine ilişkindir. Bu sistemlerde yol ifadesinin çözümlenmesi şöyle yapılmaktadır. Örneğin /a/b/c.dat dosyasına ilişkin i-node elemanı bulunmak istensin. Burada önce sıfır(0) numaralı i-node elemanı elde edilir, kök dizinin data blocklarının nerede olduğu belirlenir. Orada a ismi aranır. a'nın i-node numarası elde edilir ve bu kez a dizinin data blocklarında b aranır. b'nin i-node numarası elde edilir ve sonra benzer biçimde c.dat dosyasının i-node numarası elde edilir.

Bu dosya sisteminde dosyanın ismi dışındaki bütün bilgileri i-node elemanında saklanmıştır. Bu sistemde i-node blocktaki i-node elemanlarının sayısından daha fazla dosya yaratılamaz.

22. KESME İŞLEMLERİ

Kesme(interrupt) micro işlemcinin çalıştığı kod a ara verip başka bir kod u çalıştırması durumudur. Ortaya çıkış mekanizmasına göre kesmeler üç bölüme ayrılmaktadır.

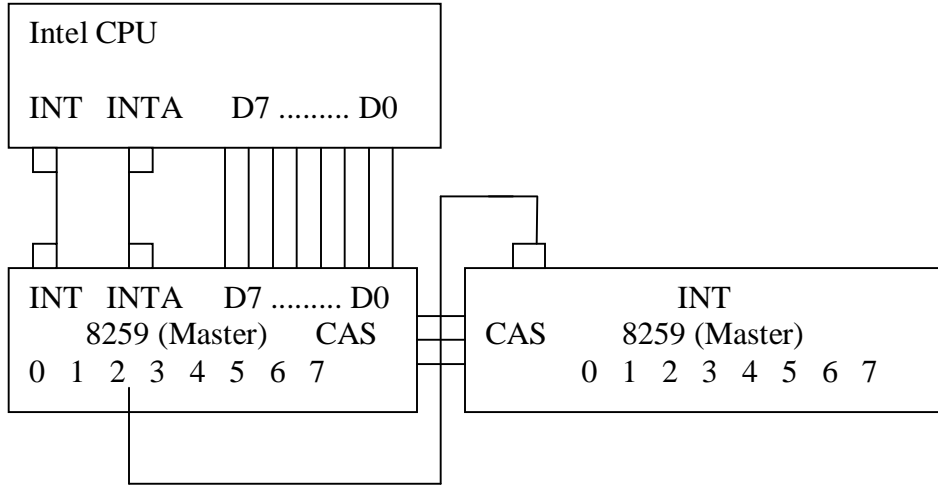
Yazılım kesmeleri(Software interrupt): Yazılım kesmeleri programcı tarafından Makine komutuyla oluşturulan kesmelerdir. Örneğin intel işlemcilerinde int Makine komutu yazılım kesmesi oluşturmak için kullanılmaktadır.

Donanım Kesmeleri(Hardware Interrupt): Kesme denildiğinde akla gelen kesmeler donanım kesmeleridir. Bu kesmeler CPU nun kesme ucunun(iny ucu) elektriksel olarak uyarılmasıyla asenkron biçimde oluşur. Her mikro işlemcide ve microdenetleyicide bir kesme ucu(int ucu) vardır. Bu kesme ucu 5 volt ya da sıfır volt kenara tetiklemeli ya da sabit uyarmalı olarak aktive edilebilir. Bu uç aktive edildiğinde micro işlemci çalıştırmakta olduğu koda ara vererek önceden belirlenmiş bir kesme kodunu çalıştırır. Micro işlemciler bir Makine kodu çalışırken çalışmaya ara veremezler. Birmak,na komutunun çalışması bittikten sonra kesme koduna dallanabilirler. Yani tek bir Makine kodu atomiktir.

İçsel Kesmeler(Internal Interrupt): İçsel kesmeler microişlemcinin bir Makine kodunu çalıştırırken kendisinin oluşturduğu kesmelerdir. Örneğin intel gibi büyük micro işlemcilerin çok sayıda içsel kesmesi vardır.

Kesme oluştuğunda Ne Olur? Kesmenin oluşum biçimi nasıl olursa olsun kesme oluştuğunda bazı tipik işlemler micro işlemci tarafından yapılamamaktadır. Örneğin işlemci geri dönüşe olarak sağlamak için sonraki komutun adresini stack e push eder. Yine bazı işlemciler bayrak yazmacındaki bazı bitleri otomatik olarak resetlemektedir. Yine micro işlemciler kesme oluştuğunda bir kod a dallanırlar. Hangi kod a dallanacaklarını kesme vektörü denilen bir yerden elde ederler. Pek çok küçük micro işlemci ve micro denetleyicide kesme vektörü tek elemanlıdır. Kesme vektörüne çalıştırılacak kod un adresi yazılır. Sistem programcısı bu sstemlerde kesme kodunu bellekte oluşturur. Onun adresini kesme vektörüne yazar. Intel gibi bazı yüksek kapasiteli micro işlemcilerde vektör birden fazla elemandan oluşturulur. Örneğin intel de toplam 256 kesme

söz konusudur. Her kesmenin bir numarası vardır. Kesme bir numarayla birlikte oluşur. Yani bir kesme oluşması değil 5 numaralı kesmenin oluşması sözkonusudur. Intel işlemcisinin int ucu doğrudan değil 8259 kesme denetleyicisi yolu ile dışsal aygıtlara bağlanmıştır. 8259 kesme denetleyicisi 70 li yıllarda tasarlanmış eski bir denetleyicidir. Fakat hala bu uyum kullanılmaktadır. Ayrıca yeni board lar da APIC(Advanced Programmable Interrupt Controller) denilen modern bir kesme denetleyicisi de kullanılmaktadır. 8259 kesme denetleyicisinin int ucunu nun int ucuna bağlanmıştır ve 8 giriş ucu vardır. Pc ler ilk çıktığında tek bir kesme denetleyicisi vardı, AT ler ile birlikte 2 kesme denetleyicisi kullanılmaya başlanmıştır. Kesme denetleyicisinin her bir giriş ucuna IRQ(Interrupt Request) denilir ve her bir uç dışsal bir birime bağlanmıştır. Ayrıca 1. kesme denetleyicisinin 2. numaralı giriş ucu, 2. kesme denetleyicisinin int ucuna bağlanmıştır. (2 tane 8259 denetleyicisini bağlamak için 1. denetleyicinin herhangi bir ucunun 2.denetleyicinin int ucuna bağlanması gerekir, PC mimarisinde bunun için 2 numaralı uç seçilmiştir). Kesme denetleyicisi ile işlemci arasındaki bağlantı daha gerçekçi olarak aşağıdaki gibidir.



Dışsal bir kesme şöyle oluşmaktadır.

1) Kesme denetleyicisinin bir IRQ ucunu dışsal bir birim aktive eder.

- 2) Kesme denetleyicisi CPU nun int ucunu aktive eder. CPU IF bayrağının durumuna bakarak, IF yi ya kabul eder yada etmez. Eğer kesmeyi kabul ederse, inta ucunu aktive eder.
- 3) Kesme denetleyicisi kesmenin kabul edildiğini anladığında D0-D7 uçlarından kesmenin numarasını CPU ya bildirir.
- 4) CPU kesme koduna geçerek kesme kodunu çalıştırır.

22.1. IRQ Kaynakları

Kesme denetleyicisinin her bir ucuna değişik kaynaklar bağlanmıştır. Bu bağlantıların çoğuna 1980 yılında ilk Pc ler zamanında karar verilmiştir. Bugün halen aynı mimari devam ettirilmektedir.

IRQ0 : Ana kesme denetleyicisinin 0 numaralı ucuna Intel'in 8254 zamanlayıcı devresi bağlıdır. Intel 8254 PIT(Programmable Interval Time) belli bir periodda darbe üreten osilatör devresidir. Bilgisayar açıldığında BIOS saniyede 18.2 darbe üretecek şekilde bu zamanlayıcıyı ayarlar. Örneğin Linux 2.6 da readin önceliği değiştirilmesse default olarak 100 milisaniyelik bir quanta değeri kullanılmaktadır.

Anahtar Notlar

Dos işletim sistemi 18.2 değerini değiştirmemektedir. 18.2 değeri aslında 8254 işlemcisine bağlanan kristal frekansı ile ilgilidir. Kullanılan kristal frekansı 1193182 Hertz dir. 18.2 değeri bu değer 65535 e bölünmesinden elde edilmiştir. O halde default darbe frekansı 18.20679 biçindedir.

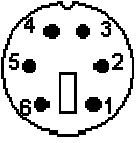
IRQ0 windows, Linux/Unix sistemlerinde treadler arası geçiş mekanizması için kullanılır. Örneğin Linux işletim sistemi boot edildiğinde BIOS un oluşturduğu default 18.2 değerini 10 mili saniyeye ve yeni kernal larda da 1 mili saniyeye çekmektedir. Fakat her timer kesmesi oluştuğunda treadler arası geçiş yapılmamaktadır. Bu durumda örneğin hızlı bir Pc de son Linux versiyonlarında timer 1 mili saniye ye kurulmakta ve kabaca her 100 timer kesmesinde treadler arası geçiş oluşturulmaktadır. Yine tipik olarak Windows sistemleri 60 mili saniye civarında bir quanta kullanmaktadır.

IRQ1 : IRQ1 hattı 8042 klavye denetleyicisine bağlıdır. Bu konuyu anlamadan önce klavye alt sisteminin çalışması incelenecektir.

Klavyenin içinde kabaca yatay ve düşey hatların kesim noktasına tuşlar yerleştirilmiştir. Bu hatların klavye içersindeki klavye işlemcisine verilmiştir. Böylece bir tuşa basılıp çekildiğinde, klavye içersindeki işlemci bu tuşun kaç numaralı tuş olduğunu anlayabilir. Klavye içindeki işlemci klasik olarak Intel 8048 yada türevi mikrodnetleyiciler ile gerçekleştirilmektedir. Fakat uyumlu pek çok klavye işlemcisi farklı firmalar tarafından üretilmiştir.(Örneğin Holtek firmasının HT82K68A Keyboard Encode işlemcisi tipik olarak klavye üreticileri tarafından kullanılmaktadır.). Klavye işlemcisinin uçları doğrudan connector'e verilmiştir. Klavye connector'ünün tipik uçları şunlardır.



- 1) KPD Clock
- 2)KPD Data
- 3) N/C
- 4) GND
- 5) +5V(VCC)



- 1) KPD Clock
- 2) GND
- 3) KPD Data
- 4) N/C
- 5) +5V(VCC)
- 6) N/C

Klavyeden bir tuşa bastığımızda klavye işlemcisi bu tuşun hangi tuş olduğunu belirler ve tuşun numarasını belirten ismine klavye tarama kodu(keyboard scen code) denilen data/clock

uçları ile seri bir biçimde dışarıya gönderir. Tipik olarak clock ucunun yükselen kenarında data ucu örneklenir. Görüldüğü gibi aktarım seri bir biçimde data ucundan yapılmaktadır. Aktarım ayrıntıları için klavye işlemcilerinin darta sheetlerine bakılmalıdır. Klavye üzerindeki yazıların klavye çalışması ile hiçbir ilgisi yoktur. Her tuşun bir numarası vardır, dış dünyaya yalnız tarama kodu iletilir. Klavye işlemcisi yalnızca tuşa bastığımızda değil elimizi çektiğimizde de kod göndermektedir. Bastığımızda gönderilen koda make kod, çektiğimizde gönderilen koda break kod denilmektedir. Pc klavyeleri break kod olarak önce bir F0 byte ı sonra make kodun aynısını göndermektedir. Örneğin bastığımız tuş için gönderilen 1 byte lık make kod hex 18 olsun. Elimizi tuştan çektiğimizde F018 biçiminde 2 byte gönderilir. O halde biz bu klavyeyi yalnızca Pc lerde kullanmak zorunda değiliz, tamamen başka aygıtlarda da kullanabiliriz.

Klavye connector'unu Pc ye taktığımızda bu uçlar doğrudan klavye denetleyicisi diye bilinen(tipik olarak Intel 8042 gibi işlemciler) bir denetleyiciye bağlıdır.

Klavyeden gelen klavye tarama kodu klavye denetleyicisine ulaştığında klavye denetleyicisi bu kodu sistem tarama kodu(systems scan code) denilen başka bir tarama koduna dönüştürür, kendi içerisindeki bir yazmaçta saklar ve IRQ1 hattını aktive eder. IRQ1 oluştuğunda klavye denetleyicisinin içinde 1 byte lık sistem tarama kodu bulunmaktadır. Bu değer 60h portu okunarak elde edilebilir.

İşletim sistemlerinde IRQ1 kesme kodu klavye denetleyicisinden basılmış yada çekilmiş tuşu olarak klavye tamponuna yerleştirir.

Klavye tarama kodu ile sistem tarama kodu arasındaki farklılıklar şunlardır:

- Make kodlar farklıdır.
- Break kod 2 byte değil tek byte a indirgenmiştir. Make kodun yüksek anlamlı biti 1 yapılarak break kod elde edilir.

İşletim sistemi basılan ve çekilen tuşları olarak bir kuyruğa yerleştirir, programlar bu kuyruktan tuşları almaktadır.

Shift gibi alt gibi, caps lock gibi tuşların hiçbir özelliği yoktur. Örneğin biz önce shift sonra a tuşuna basmış olalım ve elimi önce a sonra shift tuşunda çekmiş olalım. Önce shift tuşu için make kod sonra A tuşu için make kod sonra A tuşu için break kod en son da shift için break kod oluşturulur. Şimdi elimizi önce shift tuşuna sonra A ve B tuşlarına basmış olalım. Sonra elimizi bu tuşlardan çekmiş olalım. Sırasıyla şu kodlar için kesmeler oluşacaktır:

shift(make), A(make), B(make), A(break), B(break), shift(break). İşletim sistemi basılan tuşları kuyruğa yerleştirdikten sonra özel tuşlar için bayraklar tutar ve kullanıcının hangi tuşlara bastığını belirler. Örneğin bir programcı programlama dilinde bir karakter okumak isterse, işletim sistemi shift(make) ve A(make) tuşlarını tampondan atar ve shift bayrağını set eder, ve programcıya (büyük harf) A tuşunun ascii karşılığını verir. Sonra programcı bir karakter daha okumak isterse ona (büyük harf) B yi verir, çünkü hala shift bayrağı set edilmiş durumdadır. Makine açıldığında BIOS ta basılan tuşlardan bir kuyruk sistemi oluşturmaktadır. BIOS un oluşturduğu bu kuyruk sistemi dos tarafından da aynı biçimde kullanılır. BIOS un ve dos un klavye işlemleri şöyle organize edilmektedir:

- 1) 15 karakterlik bir klavye tamponu tutulur.
- 2) Shift, control, alt gibi tuşlara basıldığında tampona bir şey yazılmaz, yalnızca bayraklar set edilir.
- 3) Ascii tuşlarına basıldığında bayrakların durumuna bakılarak ilgili tuşlar kuyruğa yazılmaktadır.
- 4) Ctrl+alt+del, pause gibi özel tuşlar IRQ1 kesme kod tarafından o an değerlendirilmektedir.

Klavye denetleyicisi ile kalvyenin içersindeki işlemci ile data bağlantısı tek taraflı değil çift taraflıdır. Yani klavye denetleyiciside klavye içersindeki işlemciyi programlamaktadır.

Klavyede elimizi bir tuşa bastığımızda önce o tuş için bir make kod gönderilir, elimizi tutmaya devam ettiğimizde aynı tuş için make kod gönderilmeye devam eder. Bu mekanizmaya typematik denilmektedir. Typematik klavye içersindeki işlemci tarafından kontrol edilir, tuşa basıldığı andan typematik e kadar geçen süre ve typematik periodu ayarlanabilmektedir.

Klavye üzerindeki bazı tuşlara ilişkin ışıkların yakılması klavye tarafından yapılan özel bir işlem değildir. Örneğin biz caps tuşuna basmış olalım. Işık klavye devresi tarafından yakılmaz. Klavye caps tuşunun tarama kodunu klavye denetleyicisine gönderir. Sonra IRQ1 oluşur, IRQ1 kesme kodu klavye işlemcisini programlayarak ışığın yanmasını sağlar. Yni örneğin biz IRQ1 i disable etsek caps lock tuşuna bastığımızda ışık yanmaz.

IRQ2 : Birinci kesme denetleyicisinin 2 numaralı ucu ikinci kesme denetleyicisine bağlı olduğu için böyle bir uç yoktur.

IRQ3 : Bu hatta 2 ve 4 numaralı comport lara baęlı olan UART iřlemcisi baęlıdır. Pc ler normal olarak 4 seri porta kadar desteklenmektedir. UART(Universial Asynchronous Receiver Transmitter) eřitli durumlarda IRQ oluřturabilmektedir.

IRQ4 : Bu hatta com1 ve com3 seri portlarına baęlı UART iřlemcisi baęlıdır.

IRQ5 : Bu IRQ hattı genellikle ses kartları tarafından kullanılmaktadır. Geniřleme yuvasına takılan kartlarda IRQ oluřturabilmektedir. rneęin ses kartları tipik byle kartlardır. 5 numaralı bu u klasik olarak LPT2 paralel port iřlemcisine baęlıdır fakat artık bilgisayarlarda 2 paralel port bulunmadıęı iin bu IRQ ses kartları kullanabilmektedir.

IRQ6 : Bu IRQ hattına Intel 8272 flopy denetleyicisi baęlıdır.

IRQ7 : Bu IRQ hattına LPT1 paralel port iřlemcisi baęlıdır. Paralel portlarda IRQ oluřturabilmektedir.

IRQ8 : İkinci kesme denetleyicisinin 0 numaralı ucuna Real Time Clock iřlemcisi baęlıdır. Pc ler ilk ıktıęında bir saat devresi iermiyordu. Dolayısıyla bilgisayarı kapattıęımızda tarife zaman bilgisi kayboluyordu. Daha sonra AT lerle birlikte bir saat iřlemcisi anakarta eklenmiřtir.

Bilgisayarımızı kapattıęımızda board üzerindeki bu saat iřlemcisi alıřmaya devam etmektedir. Bu iřlemci board üzerindeki kk pilden beslenmektedir. RTC iřlemcisi aynı zamanda CMOS setup bilgilerinide tutmaktadır. Yani bilgisayarı aarken F2 yada del tuřu ile girdięimiz setup bu RTC iřlemcisinin iindedir. Tipik olarak bilgisayarı atıęımızda alıřma EPROM ierisindeki blgeden bařlar, buradaki kodbazı test iřlemlerini yaptıktan sonra klavyeden F2 yada del tuřuna basılıp basılmadıęına bakar. Bu tuřlara basılmıřsa RTC ierisindeki CMOS blgesine bařvurur, bu blgeyi bir menu eřlięinde edit eder. BIOS aynı zamanda aılıř sırasında bu CMOS bilgilerine bakarak evre birimlerinide ona gre programlamaktadır. rneęin biz buradan bir hard disk i devre dıřı bırakabiliriz. Bu durumda BIOS IDE denetleyicisine bilgi gndererekaygıtı disable eder. Yani CMOS setup BIOS taki statik kodu biraz dinamik hale getirmek iin dřnlmřtir. (CMOS password bilgiside CMOS un ierisinde tutulmaktadır, dolayısıyla bu pili takıp ıkarttıęımızda yada kısa devre yaptıęımızda password de disable olur.).

IRQ9-10-11 : Bu hatlar geniřleme yuvasına takılan hatlar iin serbest bırakılmıřtır.

IRQ12 : Bu IRQ hattına PS2 fairesi baęlıdır. Yani kk connector l farelerimiz bu IRQ hattını kullanmaktadır.

IRQ13 : Bu IRQ hattına matematik iřlemci baęlıdır. Eskiden noktalı sayılar üzerinde iřlem yapan matematik iřlemci ayrı bir iřlemci olarak boarda takılıyordu. Fakat 80486DX modeli ile birlikte

matematik işlemcide aynı işlemci paketinin içine yerleştirilmiştir. Bugünkü pentiumlarda yine matematik işlemci ile normal işlemci aynı entegrenin içerisinde birbirine bağlı bir biçimdedir. Matematik işlemci bir problem oluştuğunda ana işlemciye IRQ yolu ile durumu bildirmektedir.

IRQ14 : Bu IRQ hattına birincil(primary) IDE denetleyicisi bağlıdır. Örneğin hard disk te bir sektörün okunması bittikten sonra hard disk durumu IDE denetleyicisine bildiri, IDE denetleyiciside IRQ durumu oluşturur.

IRQ15 : Bu IRQ hattına ikincil IDE denetleyicisi bağlıdır.

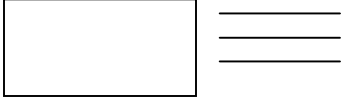
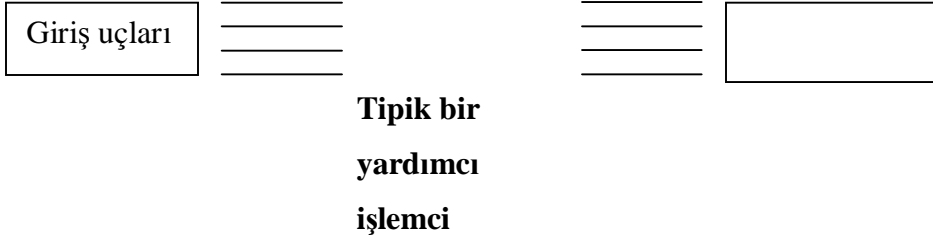
22.2 IRQ Kesme Kodlarının Set Edilmesi

Bir IRQ oluştuğunda çağrılacak kod programcı tarafından belirlenebilir. Intel işlemciler her kesme numarası için bir adres içeren kesme vektörüne sahiptir. Bu kesme vektörüne adres yerleştirebilmek için Windows ve Unix/Linux gibi sistemlerde kernel düzeyinde programlama gerekir.

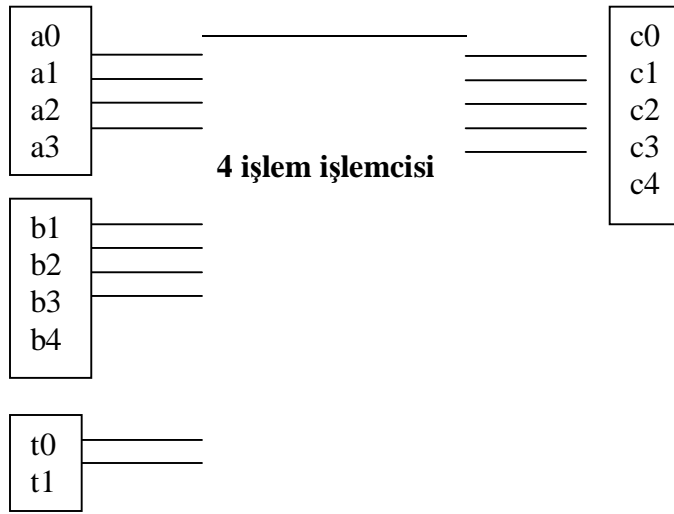
22.3 Yardımcı İşlemciler

Bir bilgisayar sisteminde bir yada birden fazla ana işlemci ve çok sayıda yardımcı işlemci vardır. Ana işlemci ile yardımcı işlemciler elektriksel olarak bağlantı halindedir. CPU yani merkezi işlemci programcının yazdığı komutları çalıştırır. Yardımcı işlemciler CPU programlamaktadır. O halde biz programımızda yardımcı işlemciye gönderilecek komutları belirleriz., CPU bizim isteğimiz doğrultusunda onları programlar. Yani sonuçta yardımcı işlemcileri yine biz programlamış olmaktadır.

Tipik bir yardımcı işlemci 3 grup uç topluluğundan oluşur: Giriş uçları, Çıkış uçları ve Kontrol uçları. İşlemcinin uçları ikilik sistemde birer bit olarak kullanılmaktadır. Örneğin tipik olarak 5V civarında bir gerilim lojik 1, 0V civarında bir gerilim lojik 0 olarak değerlendirilir. O halde CPU yardımcı işlemciye aslında ikilik sistemde sayılar göndermekte ve onlardan ikilik sistemde cevaplar almaktadır. Bu ikilik sistemdeki bilgiler elektriksel düzeyde gerçekleştirilmektedir.



Örneğin 4 bitlik iki sayı üzerinde dört işlem uygulayan tipik bir işlemcin uçları şu şekilde olabilir:



T0	T1	Anlamı
0	0	Toplam
0	1	Çıkarma
1	0	Çarpma
1	1	Bölme

Bir işlemciye komut göndermek demek ona ikilik sistemde sayılar göndermek demektir. CPU bütün işlemcilere bağlıdır, fakat belirli bir işlemciyi seçebilmesi için port numarası denilen bir numaraya gereksinim duyar. Böylece komutu diğerlerine değil yalnızca ona gönderebilir.

Bazı işlemcilerin tek bir port numarası vardır, bazı işlemcilerin birden fazla port numarası bulunabilir. Intel işlemcilerinde yardımcı işlemciye bilgi göndermek için out makine kodu, yardımcı işlemciden bilgi almak için in makine kodu kullanılır. **Örneğin; out 21, FF.** Burada biz 21h portuna FF değeri gönderilmiştir. **Örneğin; in al, 21.** Burada ise 21h portundan değer alınmış ve al yazmaçına yerleştirilmiştir.

Bir işlemcinin bir portun hangi sayıları gönderdiğimizde bunun ne olarak ele alınacağı ve yorumlanacağı işlemciye göre değişmektedir. Her yardımcı işlemcinin programlanması ayrı bir biçimde o işlemci öğrenilerek yapılır.

Yardımcı işlemcilere komut gönderme Windows ve Linux/Unix sistemlerinde ancak aygıt sürücülerle mümkündür. Fakat komut gönderme işini yapan basit aygıt sürücüler yazılmış olarak bulunmaktadır. Örneğin Windows da inpout32 isimli aygıt sürücü 2 fonksiyon sunmakta, bu fonksiyonlarda in ve out komutlarını kernal moda uygulamaktadır. Linux sistemlerinde /dev dizinin altındaki port isimli aygıt sürücü zaten bu işi yapacak şekilde düzenlenmiştir.

Örneğin klavye denetleyicisi 60h ve 61h portlarını kullanmaktadır. Anımsanacağı gibi son basılan yada çekilen tuşun klavye tarama kodu dönüştürülerek sistem tarama kodu haline getirilmekteydi. Klavye denetleyicisi bu değeri 60h portuna yerleştirir, sistem programcısı son basılan yada çekilen tuşu anlamak için 60h portundan okuma yapmaktadır. Gerçektende IRQ1 kesme kodu önce 60h portunu okumakta, bu bilgiyi almakta ve basılan tuşa karar vermektedir. Yine klavyedeki özel tuşların ışığına yakmak için typematik ayarlarını yapmak için 61h portunu kullanmak gerekir.

22.4. Paralel Port Kullanımı

Paralel port bilgisayar yazıcı haberleşme için düşünülmüş olan genel amaçlı bir porttur. 25 adet pine sahiptir. Her pine bilgisayar yazıcı haberleşmesi dikkate alınarak bir isim verilmiştir. Aslında paralel port bağımsız amaçlardaki kullanılabilir. Pin isimleri şöyledir:

Pin No(DB25)	Signal name
1	nStrobe
2	Data0
3	Data1
4	Data2
5	Data3
6	Data4
7	Data5
8	Data6
9	Data7
10	nAck
11	Busy
12	Paper-out
13	Select
14	Linefeed
15	nError
16	nInitialize
17	nSelect-Printer
18-25	Ground

Paralel portlar ilk çıktıklarından bu yana evrim geçirmiştir. Bugünkü paralel portlar geçmişteki modlarıda destekler durumdadır. İlk paralel portlar standart mod ile bugün kullanılabilir. Standart moda yalnızca 4 bit gönderip alma yapılabilir. Daha sonra

EPP(Enhanced Paralel Port) modu tasarlanmış ve port biraz geliştirilmiştir. Bu moda 8 bitlik transfer yapılabilir. Nihayet paralel portlar biraz daha geliştirilerek ECP(Extended Capability Port) haline getirilmiştir. Bugün bu 3 modda kullanılabilir. Aralarında önemli farklılıklar yoktur ve burada standart mod açıklanacaktır.

USB portların gelişmesi ile ve yaygınlaşması ile paralel portun cazibesi büyük oranda kaybolmaktadır. Fakat basit kullanımı nedeniyle yine de varlığını sürdürmektedir.

Paralel portun 3 port adresi vardır: Taban port adresi değişebilmekle birlikte çok büyük olasılıkla LPT1 için 378h, LPT2 için 278h dir.

Anahtar Notlar

Bir işlemci birden fazla port kullanıyorsa bu portlar ardışıdır ve ilk port numarasına taban port denilmektedir.

Paralel port paralel port işlemcisi tarafından yürütülür ve bu işlemcinin 3 adet yazmacı vardır. Bu yazmaçlar portlara bağlanmıştır. Taban+0 numaralı porta data yazmacı, taban+1 numaralı porta durum yazmacı, taban+2 numaralı porta komut yazmacı bağlıdır. Bu yazmaçlar 8 bit uzunluğundadır. Paralel portun data yazmacı paralel portun 2 ile 9 numaralı pinlerini temsil etmektedir. Bu yazmaç standart moda okunabilir yada yazılabilir. Ancak standart moda paralel portun 2-9 numaralı pinleri yalnızca out amaçlı kullanılabilir. Bu nedenle biz bu yazmacı okuduğumuzda ilgili pinlerin durumunu değil son yazdığımız değeri okuruz. Programcı taban port numarasını kullanarak out işlemi yaptığında değerler bu yazmacı yapılır, paralel port işlemcisi 1 olan bitleri 5V a 0 olan bitleri 0V a çeker. O halde paralel porttan bir ledi yakıp söndürmek için aşağıdaki gibi bir kod yazılabilir.

```
while (!kbbit()) {
    Out32(0x378, 1);
    Sleep(1000);
    Out32(0x378, 0);
    Sleep(1000);
}
```

7	6	5	4	3	2	1	0
9	8	7	6	5	4	3	2

Paralel portun taban+1 numaralı portuna durum yazmacı bağlıdır. Durum yazmacı da bir grup pin ile ilişkilendirilmiştir. Fakat bu pinler yazma amaçlı değil okuma amaçlıdır.

7	6	5	4	3	2	1	0
~11	10	12	13	15	x	x	1

Durum Yazmacı

Biz bu yazmacı okursak dışarıdan o pinlere uygulanan işaretleri elde ederiz. Bu yazmaç read only dir ve yazmaya izin yoktur. 7. bit(yani 11. pin terstir(toggle)). Yani 11. pin dışarıdan sifıra çekildiğinde biz onu bir diye, bire çekildiğinde sıfır diye okuruz. 1 ve 2 numaralı bitler kullanılmamaktadır. Görüldüğü gibi standart moda paralel port 8 bit out yaparken yalnızca 5 bit in yapabilmektedir. Böylece eski paralel portlar 4 bit-4bit haberleşme yapıyorlardı.

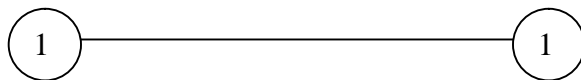
Komut yazmacı paralel portu programlamakta kullanılır, aynı zamanda bu yazmaç yine bazı out uçları ile ilişkilendirilmiştir.

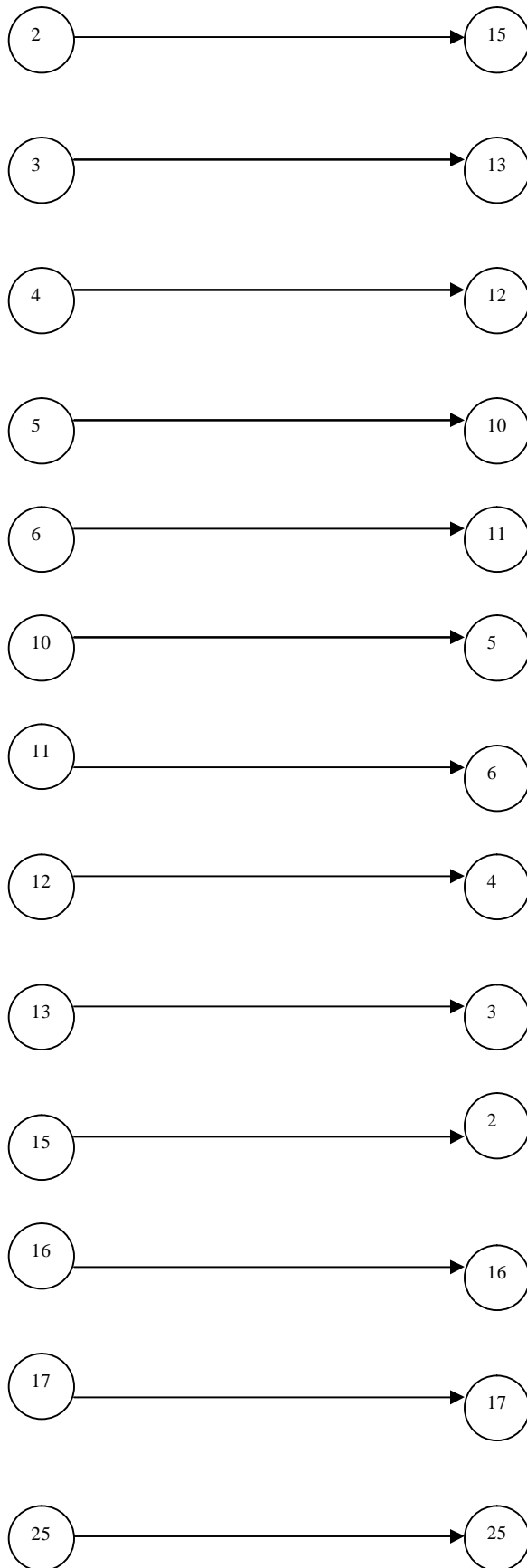
7	6	5	4	3	2	1	0
x	x	x	x	~17	16	~14	~1

Komut Yazmacı

22.5. Paralel Portta Veri Transferi

Eskiden ağ haberleşmesi yaygın değil iken ve Ethernet kartları kullanılmıyor iken bilgisayardan bilgisayara veri transferi çok zordu.(90 yılların ortalarına kadar). O günlerde paralel portlar iki bilgisayar arasında dosya aktarımı içinde kullanılıyordu. İsmine lablink denilen özel bir kablo, bir portun out uçlarını diğerinin in uçlarına bağlayarak elde ediyordu. Sonra özel yazılımlar 1byte lık bilgiyi 4 bitlik 2 parçaya ayırıp, bu kabloodan gönderip alabiliyordu.





22.6. Seri Haberleşme

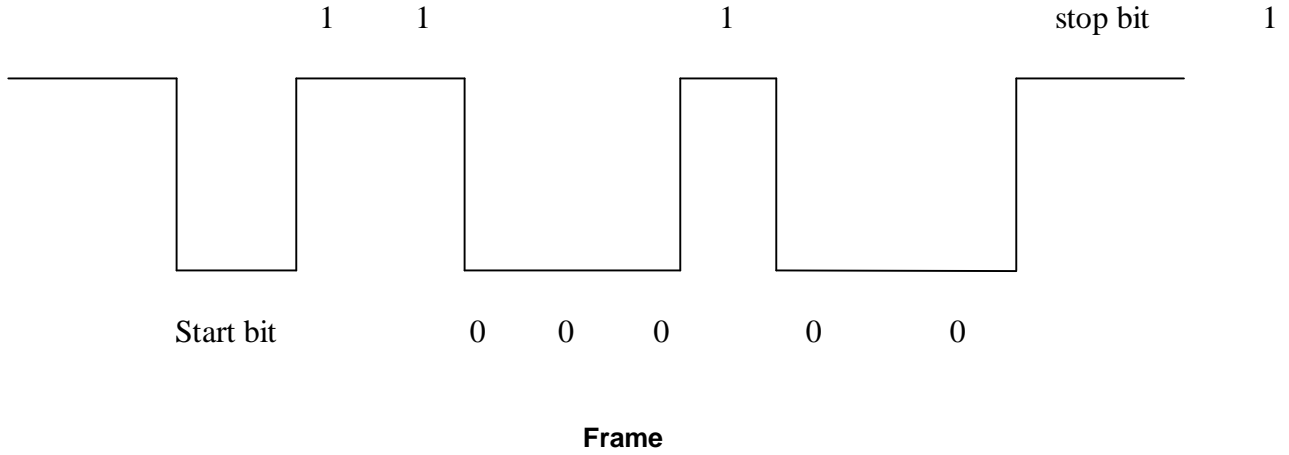
Seri haberleşme temel olarak zaman aralıklarıyla bilginin bit-bit karşı tarafa aktarılması ile meydana gelir. Teorik olarak yalnızca 2 hat ile seri bilgi akıratımı yapılabilir. Gönderici taraf byte 1 bitlerine ayırır, 1 olan bit için hattı örneğin 5V a, 0 olan bit için 0V a(bu seviyeler değişebilir) çeker. Karşı tarafta aynı hızda hatta bakar. Böylece byte aktarımı yapılır. Ancak 2 hat ile aynı anda yalnızca bir taraf diğerine bilgi gönderir. Fakat aynı anda her iki tarafta birbirine bilgi gönderip alıcak ise en az 3 hat gerekir.

Seri haberleşme gönderip alma durumuna göre 3'e ayrılır.

- 1) **Simplex Haberleşme** : Yalnızca bir taraf bilgi gönderir, ters yönde gönderim olmaz. Bunun için 2 adet hat yeterlidir.
- 2) **Half-Duplex Haberleşme** : Burada 2 taraf ta birbirine bilgi gönderip alabilir, fakat bunuları aynı anda yapamaz. Bunun içinde 2 adet hat yeterlidir.
- 3) **Full-Duplex Haberleşme** : İki tarafta aynı anda birbirlerine bilgi gönderip alabilir. Bu haberleşme için en az 3 hat gerekmektedir.

Seri haberleşmenin en önemli problemi iki tarafın aynı hızda gönderip alma yapmasıdır. Bugünkü elektronik devreler bunu sağlayabilmektedir. Diğer önemli bir problem alıcı tarafın hatta baktığında o anda bilgilerin gönderilip gönderilmediğini belirlemesi zorunluluğudur.

Seri haberleşme kendi içersinde asenkron ve senkron olmak üzere iki ye ayrılır. Asenkron haberleşme doğada en çok karşılaşılan durumdur. Burada byte ın bitleri eşit hızda gönderilip alınır, fakat byte lar arasında rastgele bekleme süreleri söz konusudur. Halbuki senkron haberleşmede hem bytler ın bitleri hemde byte lar aynı hızda gönderilip alınır. Asenkron seri haberleşmede klasik yöntem start bit stop bit yöntemidir. Bilgi gönderilmiyorken hat lojik 1 seviyesinde bekletilir, gönderici tarafbilgiyi göndermeden önce hattı 1 birim sıfıra çeker. Böylece alıcı taraf bilginin gönderilmeye başlandığını anlar. Artık iki tarafta belirlenen hızda bilgileri gönderip alırlar. Nihayet işlem bitince hat yeniden eski 1 durumuna çekilir. Bunada stop bit denilir.

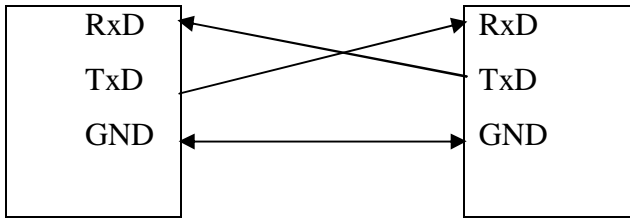
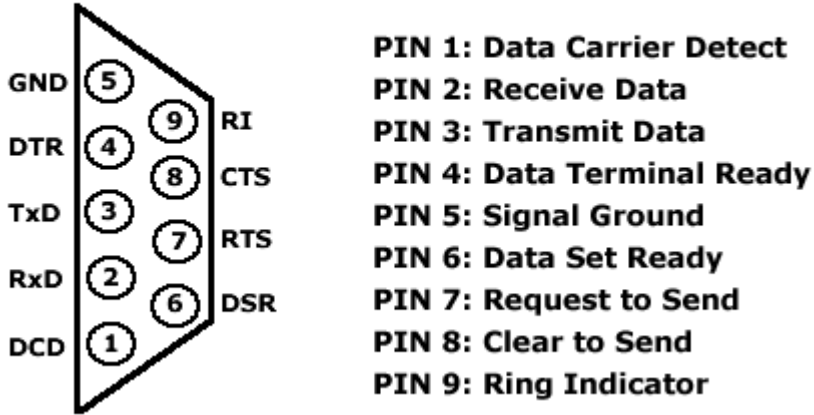


Pc lerde seri haberleşme için UART (Universal Asynchronous Receiver and Transmitter) denilen yardımcı işlemci kullanılmaktadır. UART Intel firmasının 8250, National firmasının 16550 işlemcileridir. UART işlemcisi belirli bir frekansta programlanabilmektedir. Programcı UART işlemcisine bilgiyi byte olatak verir, işlemci onu bitlerine ayırıp karşı tarafa belirlenen hızda gönderir. UART aynı zamanda full-dublex olarak alma işleminide yapabilmektedir. İki UART işlemcisi birbirine bağlanabilir. UART işlemcisinin en önemli 3 ucu TxD(Transmit-data), RxD(Receive-data, GND(Ground) dır. UART işlemcisi seri porta bağlı olarak kullanılır. Biz UART a 1 bitlik bir bilgi verdiğimizde, UART bunu bitlerine ayırarak TxD uçlarından bit-bit kodlar. Bilgisayar tarafı 9PIN MALE connector biçimindedir. UART ve seri port bilgisayar modem haberleşmesi için düşünülmüştür ve pinlere isimleri buna göre verilmiştir. UART işlemcisinin ucu doğrudan değil rs232 dönüştürücü ucu ile seri porta verilmiştir. Normalde UART işlemcisinin uçları TTL düzeyindedir.(Tipik +5V sıfır). Halbuki seri portun uçları rs232 sinyal seviyelerine göre ayarlanmıştır. Bu sistem tipik olarak +-15V civarındadır. UART işlemcisinin uçları ile seri port un uçları aynıdır. En önemli 3 uç yukarıda da belirtildiği gibi RxD, TxD ve GND uçlarıdır.

Seri port işlemleri çeşitli düzeylerde yapılabilir. En aşağı düzeyde UART işlemcisinin portlarına bilgi gönderilip alınarak işlemler gerçekleştirilebilir. Fakat bu işlemleri yapan başkaları tarafından yazılmış yada C kütüphanesine eklenmiş, eklenti niteliğindeki fonksiyonlarda kullanılabilir. Windows ve Unix/Linux sistemlerinde UART ın haberleşme portlarına doğrudan

erişmemiz engellenmiştir. Bu işletim sistemlerinde seri port işlemlerini gerçekleştiren özel sistem fonksiyonları vardır.

RS-232 DB-9 Male Pinout



UART işlemcisi 7 haberleşme portu kullanır. Pc lerde Com1 için taban port adresi 3F8H, Com2 için 2F8H dir.

UART içerisinde çeşitli yazmaçlar vardır. Bu yazmaçlar çeşitli haberleşme portlarına bağlıdır.

İki UART ın haberleşebilmesi için öncelikle bunların aynı değerlerle set edilmesi gerekir. Set etme sırasında şu işlemler yapılır:

- UART ın gönderme ve alma hızı belirlenir.
- Data bitlerinin sayısı belirlenir. Start bitten sonra data bitlerinin sayısı 5, 6, 7 yada 8 olabilir. Yani 8 olması zorunlu değildir.

- Stop bitler 1 yada 2 olabilir. Bazen haberleşmeyi yavaşlatmak için ve hattan kaynaklanan problemleri tolere etmek için 2 stop bit kullanma yoluna gidilmektedir. Fakat son yıllarda böyle bir gereksinim kalmamıştır.
- UART bilgi gönderir yada alırken parity ayarı ve kontrolü yapabilmektedir.

Yukarıdaki ayarların her iki bilgisayarda da aynı biçimde yapılması gerekmektedir. Aksi halde hemen frame hatası oluşur.

22.7. Hataların Belirlenmesi Üzerine Yöntemler

Bir bilgiyi iletirken yada saptarken bilgi ile birlikte bazı anahtar değerlerde iletilir yada saklanırsa, daha sonra bu değerlere bakılarak bilginin bozulup bozulmadığı anlaşılabilir. Bunun için çeşitli yöntemler kullanılmaktadır.

- 1) **Parity Yöntemi** : Bu yöntem özellikle seri haberleşmede yaygın kullanılmaktadır. Örneğin UART işlemcileri parity hesabı yapmaktadır. Parity yönteminde her byte için yada bit grubu için ayrıca birde parity biti gönderilir. Parity tek (odd) yada, çift (even) biçiminde olabilir. Tek parity demek data içerisindeki 1 olan bitlerin sayısını teke tamamlamak için gönderilen bittir. Çift parity ise çifte tamamlamak için gönderilen parity dir. **Örneğin ;**

10101110 tek parity = 0, çift parity = 1

□

data

Parity yöntemi bozulmalara çok duyarlı bir yöntem değildir.

- 2) **Checksum Yöntemi** : Bu yöntem dosyalarda yani dosyaların bozulmasını belirlemek için tercih edilen bir yöntemdir. Checksum basit bir yöntemdir fakat gücü sınırlıdır. Byte, word, dword versiyonları vardır. Bu yöntemde dosyanın tüm byteleri kümülatif olarak taşanlar atılacak biçimde toplanır. Daha sonra tekrar aynı hesap yapıldığında önceki değer ile aynı değer bulunursa bilgi bozulmamıştır, bulunmazsa bilgi bozulmuştur. Bazen checksum

değeri negatif checksum biçiminde de hesaplanır. Negatif checksum ismi üzerinde elde edilen checksum ın negatif değeridir. Eğer checksum bilgiside dosyanın içerisinde bir yerde bulunacak ise bu durumda dosyanın checksum ı hesaplandığında sıfır (0) elde edilir. Bozulma kontrolü için CRC(Cyclic Redundancy Check) kullanılır. CRC yöntemini hesaplaması daha zordur fakat değişimlere karşı duyarlılığı çok daha fazladır. Örneğin checksum yönteminde dosya 2 byte yer değiştirmişse checksum değeri değişmez. CRC yöntemi modemlerde, disklerde, ve sıkıştırma programlarında yaygın kullanılan yöntemdir.

22.8. assert Makrosunun Kullanımı ve Projelerin Delay ve Release Versiyonları

Programdaki yanlış çalışmaya yol açan gizli hatalara böcek denilmektedir. Yapılan araştırmalar, programcının yaptığı hataların 5% ile % 8 arasının nihayi ürüne yansıdığını göstermektedir. Ayrıca böceklerin ürün satıldıktan sonra düzeltilmesi çok maliyetli bir işlem olarak bilinmektedir. Böcek oluşumunun en başta programcı tarafından engellenmesi gerekir. Programcı kodunu yazarken aynı zamanda da bir böcek oluşuyormu diye kontrol etmesi gerekir. Bu işlem assert makrosu ile yapılır. assert makrosu assert.h ile bildirilmiştir ve kullanımı şöyledir : assert(ifade).

Pek çok derleyicide assert makrosu aşağıdaki koda benzer şekilde yapılır.

```
#ifndef NDEBUG
#define assert(exp)
if (!(exp)) {
    fprintf(stderr, "Assertion failed : %s-%d\n", --FILE--, --LINE--);
    abort();
}
#else
#define assert(exp)
#endif
```

Görüldüğü gibi ndebug sembolik sabiti define edilmemişse default olarak debug mod söz konusudur. Bu durumda assert makroları yerine kontrol kodları açılır. Eğer ndebug sembolik sabiti define edilmişse tüm assert makroları yerine boşluk atanır. Programcı assert makroları kodun çeşitli yerlerine yerleştirerek anormal durumların oluşmasını gözler. Eğer assert makrosuna takılırsa bir ipucu elde edilmiş olunur. Bu ipucu değerlendirilerek böceğin bulunduğu yere gelinir.

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

    str = strchr("Ankara", 'x');
    myputs(str);
    return 0;
}
```

```
void myputs(const char *str)
{
    assert (str != NULL);
    while (*str != NULL) {
        putchar(*str);
        ++str;
    }
    putchar ('\n');
}
```

Burada fonksiyon null adresle çağırılması normal bir durum değildir. İşte assert makrosu bunu yakalar ve bize hatanın yerini bulmamız için bir ipucu verir. Örneğin

```
void CalcSomething(int val)
{
    assert(val >= 0);
    ...
}
```

Burada val değerinin parametresinin sıfırdan büyük olması gerekir. Normal olan budur.

Programcı gerekli her yere assert makrolarını yerleştirerek kontrol sağlar. Fakat daha sonra kodunda hiçbir böcek olmadığından emin olduğunda assert makrolarının yol açtığı gereksiz kontrollerden kurtulmak isteyecektir. Bunun için tek yapması gereken şey ndebug makrosunu define etmektir. Bir projenin 2 versiyonu vardır. Bunlar debug ve release dir. Debug versiyon assert makroların koda eklendiği, gereksiz kontrollerin yapıldığı versiyondur. Programcı kodunu debug versiyonda geliştirir, satmadan önce release derlemesi yapar. Release derlemesi tüm assert makrolarını koddan çıkartmaktadır. Pek çok IDE de de projenin debug ve release versiyonları menu seçenekleri ile ayarlanmaktadır. Visual studio da default durum debug versiyondur. Bu versiyonda ndebug sembolik sabiti define edilmemiştir ve ayrıca kütüphanelerin debug versiyonları devreye sokulur. Kütüphanelerin debug versiyonlarında yine assert ler uygulanmıştır.

Projenin release versiyonunda ise `ndebug` define edilmiştir ve kütüphanenin release versiyonları devreye sokulur.

22.9. Değişken Sayıda Parametre Alan Fonksiyonlar

C de prototipte yada tanımlama sırasında ... atomu fonksiyonun istenildiği kadar argüman ile çağırılabilceğini belirtir. Örneğin;

```
void Func (int a, ...);
```

Burada fonksiyon en az bir parametre almak zorundadır. ... için sıfır tane yada daha fazla argüman girilebilir. Standartlara göre ... parametresi parametre listesinin sonunda bulunmak zorundadır ve ayrıca ... nın solunda en az bir parametre bulunmak zorundadır. Örneğin `printf` fonksiyonunun tanımı şöyledir;

```
int printf(const char *str, ...);
```

Intel işlemcilerinde parametre aktarımı stack yolu ile ve sağdan sola yapılmaktadır. Yani ilk parametre stack te düşük adreste bulunur, diğer parametrelere bu paramatrenin yerini biliyorsak erişebiliriz.

Parametrelerin stack e atılış sırası ve stack organizasyonu, mimariden mimariye değişebildiği için stack ten paramatre çekme işlemi makrolara yaptırılmaktadır. Bu makrolar `stdarg.h` içerisinde bulunmaktadır. `va_list` türü genellikle bir adres türüdür. Önce bu türden bir nesne tanımlanmalıdır. Daha sonra `va_start` makrosu ile bu `va_list` ve ilk paramatre işleme sokulur. Örneğin;

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <assert.h>
#include <stdarg.h>

void Func(int a, ...)
{
    va_list arg;
    va_start(arg, a);
    val = va_arg(arg, int)
        ...
}
```

```
int main(int argc, char *argv[])
{
    Func(10, 20, 30, 40, 50, 60, 70, 0);
    return 0;
}
```

Burada arg aslında bir göstericidir. va_start makrosuda ilk parametreden sonraki paramatrenin adresini alarak buna yazmaktadır.

va_arg makrosu va_list türünden bir nesneyi ve bir tür bilgisini parametre olarak alır, stackte sıradaki argümanı çeker.

Son olarak va_end makrosu çağırılmaktadır. Bu makronun yaptığı bir şey genellikle yoktur. Aşağıdaki fonksiyon 0 bulana kadar argümanları stackten çekip toplamaktadır;

```
int Add(int a, ...)
{
    va_list arg;
    int total = 0, val;

    va_start(arg, a);
    total += a;
    while ((val = va_arg(arg, int)) != 0)
        total += val;
    va_end(arg);
    return total;
}

int main(int argc, char *argv[])
{
    printf("%d\n", Add(10, 20, 30, 40, 0));
    return 0;
}
```

Tabi bu örnekte biz diğer tüm parametrelerin int türden olduğunu varsaydık. Aslında programcı bir biçimde bu türleri anlamak zorundadır. Zaten fonksiyon 1.parametresi bunun anlaşılması için bir ipucu vermektedir. Örneğin;

```
printf("a = %d b = %lf\n", a, b);
```

Burada printf %d ve %lf formatlarına bakarak ilk paramatrenin int değerinin double türden olduğunu anlar.

C derleyicisi ... parametresine karşı gelen int türünden küçük argümanları int türüne, float türünü ise double türüne dönüştürerek stack e atar.

