**SİSTEM PROGRAMLAMA VE İLERİ C UYGULAMALARI -Mayıs-2015**

**(Eğitmen: Kaan ASLAN)**

**Sistem Programlama Nedir?**

Bilgisayar donanımıyla arayüz oluşturan uygulama programlarına çeşitli bakımlardan hizmet veren programlara "sistem programları" denir. Bu alana da "sistem programlama" denilmektedir. Sistem programları aşağı seviyeli olma eğilimindedir. Bunları yazmak için belli miktar teori ve mühendislik bilgi gereklidir. Sistem programlama yazılımın ağır sanayisi niteliğindedir. Tipik sistem programları şunlardır:

- İşletim Sistemleri

- Derleyiciler ve yorumlayıcılar

- Editörler

- Debug Programları

- Virüs ve Antivirüs yazılımları

- Haberleşme programları

- Gömülü sistem programları

- Aygıtların programlanması, aygıt sürücüler

- Veritabanı motorları

- Sanallaştırma yazılımları

- Oyun motorları

- ...

Sistem programlama faaliyetleri için en çok kullanılan diller C, C++ ve Sembolik Makina Dilleridir. Tabi bazı sistem programları C#, Java, hatta Python gibi dillerle de yazılabilmektedir. Fakat C/C++ dillerinin asıl uzmanlık alanı sistem programlamadır.

**UNIX Türevi İşletim Sistemlerinin Tarihsel Gelişimi**

UNIX İşletim sistemi AT&T Bell Lab'ta 1969-1970 yılları arasında geliştirilmiştir. Proje ekibinin lideri Ken Thompson'du. Çalışma ekibinde Dennis Ritchie, Brian Kernighan da vardı. Ekip daha önce Multics işletim sistemi üzerinde başka ekiplerle birlikte çalışıyordu. Bu prjeden çekilerek kendi işletim sistemlerini yazmak istediler. Zaten Unix ismi Multics isminden kelime oyunu yapılarak uydurulmuştur. C Programlama Dili bu projenin bir yan ürünü olarak Dennis Ritchie tarafından geliştirilmiştir.

AT&T UNIX'in kodlarına telif uygulamadı fakat UNIX ismini bir marka ismi olarak tescil ettirdi. UNIX'in kaynak kodları pek çok araştırma kurumuna dağıtıldı. Şirketler bunları elde ettiler ve zamanla pek çok kurum şirket bu kodları değiştirerek UNIX türevi işletim sistemi yazamaya çalıştı. Bunların ilki ve en önemlilerinden biri Berkeley California Üniversitesi tarafından geliştirilmiş olan BSD sistemleridir. Daha sonra pek çok firma UNIX türevi sistemlerini çıkarttı. IBM firması AIX, HP firması HP-UX, Sun firması Solaris, SCO firması SCO-UNIX gibi UNIX türevi sistemler geliştirdiler. Ayrıca Microsoft tarafından SCO için XENIX sistemlerinde yazılmıştır.

UNIX'in C'de yazılması bir devrim niteliğindeydi. Çünkü o zamana kadar yüksek seviyeli dillerde bir işletim sistemi yazılmamıştı. UNIX'in C de yazılması bu sistemin port edilmesini çok kolaylaştırmıştır ve işletim sistemi yazma bilincinin gelişmesine yol açmıştır.

Apple firmasının Machintosh makinalarında kullanılan işletim sistemleri bunların 10. versiyonuyla UNIX türevi haline gelmiştir. Bugünkü Mac OS X sistemleri Darwin denilen hibrit bir çekirdeğe sahiptir. Fakat yüksek oranda BSD kodları içermektedir. Bu nedenle Mac OS X sistemlerinin belli bir POSIX uyumu da vardır.

80'li yılların ortalarında pek çok UNIX türevi işletim sistemi oluştulmuş durumdaydı. Bunlar farklı sistemler olsa da AT&T UNIX'e çok benziyorlardı. Zaten onun kaynak kodları bunlarda kullanılmıştı. İşte IEEE bu UNIX türevi sistemlere bir standart getirmeye çalıştı ve POSIX standardizasyon komitesini kurdu. Böylece POSIX standartları oluşturuldu. POSIX (Portable Operating Systems for UNIX) UNIX türevi sistemlerin bulundurması gereken C fonksiyonlarını ve shell komutlarını tanımlamaktadır. Bugün pek çok UNIX türevi sistem POSIX standartlarına uymaktadır.

80'li yılların ortalarında AT&T de artık UNIX'e telif uygulamaya başladı ve böylece bedava bir UNIX türevi sistem kalmadı. Hollandalı profesör Tanenbaum derslerinde kullanmak için Minix isimli Mini bir UNIX sistemi geliştirdi. Bu sistem ticari dünyada pek kullanılmamış olsa da geniş kesimleri motive etti.

80'li yılların ortalarında Richard Stallman FSF (Free Software Foundation) isimli kurumu kurdu ve GNU (GNU Not UNIX) projesini başlattı. Bu projenin amacı bedava ve özgür yazılım akımıyla bir işletim sistemi ve yardımcı programlarını oluşturmaktı. Örneğin bugün UNIX türevi sistemlerde kullanılan gcc derleyicisi, ld bağlayıcısı ve pek çok utility program bu proje kapsamında geliştirilmiştir. (gcc ve temel bazı programlar bizzat Richard Stallman ve yardımcıları tarafından yazılmıştır.)

Özgür yazılım akımının temel prensipleri şunlardır:

- Bu akımla üretilen yazılımların kaynak kodları açılır. Herhangi birisi bunlara ekleme yapabilir, bunları değiştirebilir fakat onlar da açmak zorundadır.

- Programın çalıştırılabilir biçimi istenildiği gibi kopyalanıp dağıtılabilir.

- Özgür yazılım bedava olmak zorunda değildir. Kişiler başka faaliyetlerden para kazanma yoluna gidebilirler.

Özgür yazılımın dışında ona çok benzeyen ancak belli konularda farklılıklar içeren "Açık Kaynak Kod (Open Source)" akımı da vardır. Hem özgür yazılım akımı hem de açık kayna kod akımı benzer fikirlere sahiptir. Bunların hepsine "açık kaynak kod akımları" denebilir. Her akım kendisine uygun bir kalıp lisans sözleşmesi de oluşturmuştur. Özgür yazılımım akımının temel lisansı GPL (GNU Public Liscence)'dir. Bunun yumuşatılmış LGPL (Lesser GPL) biçiminde bir versiyonu da oluşturulmuştur. Ayrıca Apache, MIT, BSD gibi açık kaynak kodlu başka lisanslar da vardır. Şüphesiz bunların aralarında birtakım farklılıklar bulun maktadır.

90'lı yılların başlarında Linus Torwalds Helsinki Üniv.'de öğrenciyken işletim sistemi yazmaya karar vermiştir. UNSENET grububda örgütlenerek bunu başarmıştır. Linux ilk versiyonları 90'ların ilk yarısında çıktı. Sonra gelişme gösterdi. Bugün Linux sistemleri en yaygın kullanılan UNIX türevi POSIX uyumlu sistemlerdir. Linux'un 200'ün üstünde dağıtımı vardır. Linux aslında bir çekirdek geliştirme projesidir. Linux dağıtımlarıyla binlerce açık kaynak kodlu yazılımlar da bilgisayarımıza yüklenmektedir.

Bugün Internete bağlanan tüm makinalar arasında en çok kullanılan işletim sistemi Windows'tur (%50'den fazla), sonra UNIX türevi sistemler gelmektedir (çoğunluk Android'ten), sonra Apple aygıtları (IOS, MAC OS X) geliyor.

**Anahtar Notlar:** Linux'un kaynak kodlarını dolaşarak incelemek için lxr.linux.no sitesi, Darwin, FreeBSD, Minix ve OpenSolaris için fxr.watson.org sitesi kullanılabilir. (Bu sitelerde lxr projesi kullanılmışyıt.)

Anahtar Notlar: Kaynak kodları incelemek ve dolaşmak için en çok kullanılan yazılımlar şunlardır:

- Understand (Proprietary)

- Source Insight (Proprietary)

- SourceNav (Open Source)

**Masaüstü ve Mobil İşletim Sistemleri**

Neredeyse her yaygın masaüstü işletim sisteminin bir mobil versiyonu da oluşturulmuştur. Windows'un mobil versiyonuna genel olarak Windows CE denilmektedir. Windows CE'nin akıllı telefonlar ve tabletler için özelleştirilmiş biçimine Windows Mobile denilmektedir. Apple firmasının (yani Mac OS X'lerin) mobil işletim sistemi IOS (Iphone Operating System). Android bir çeşit mobil Linux sistemidir. Android projesinde Linux alınmış, biraz özelleştirilmiş, bazı parçaları atılmış, buna bir arayüz giydirilmiş ve akıllı telefonlara uygun hale getirilmiştir. Nokia eskiden Symbian sistemlerinde büyük bir pazar payına sahipti. Akıllı telefon geçisişini çok iyi yönetemedi. MeeGo ve Maemo sistemlerini denedi. Sonra büyük bölümünü Microsoft'a satarak batmaktan kurtuldu. Bugün Nokia artık akıllı telefon olarak Windows Mobile sistemlerini üretmektedir.

Bugün için en yaygın kullanılan mobile işeletim sistemi Android'tir (%50'den fazla). Bunu IOS izlemektedir (%30 civarı). Sonra da Windows Mobile gelmektedir (%2 civarı).

Mobil işletim sistemlerinin doğal programlama ortamları sistemden sisteme değişebilmektedir. Gerçi bu sistemlerin hepsinde C/C++ ile programlama yapılabilir. Fakat Android doğal ortamı Java, IOS'un Objective-C ve Swift, Windows Mobile'in C#'tır.

**Orijinal Kod Temeline Sahip İşletim Sistemleri**

Bazı işletim sistemleri bazı işletim sistemlerinin kodları alınıp değiştirilerek oluşturulmuştur (örneğin Android gibi). Bazı işletim sistemlerinin belli bir mimariye uysa da kodları sıfırdan yazılmıştır. Kodları sıfırdan yazılan yani orijinal kod temeline dayanan işletim sistemleri şunlardır:

- AT&T UNIX

- DOS

- Windows

- Linux

- BSD (belli bir yıldan sonra)

- Solaris

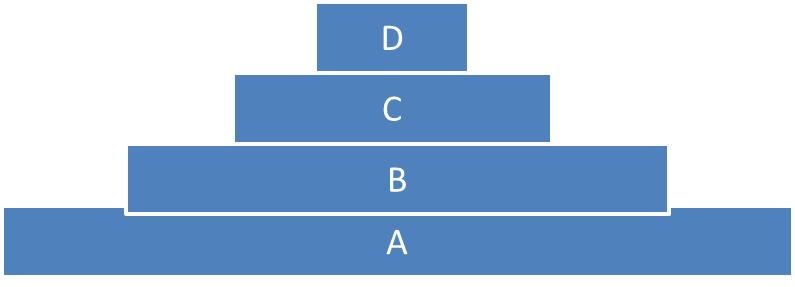
- XENIX

Mac OS X'in çekirdeği olan Darwin Mach çekirdeği ile BSD çekirdeğinin kodlarını hala kullanmaktadır. Bu anlmda orijinal kod temeline sahip değildir. Android de IOS'ta benzerdir.

Peki "bir işletim sistemi yazdık" demek için ne gerekir? Yanıt: Yapılan katkının çekirdek düzeyinde olması ve belli bir yoğunlukta olması gerekir. Bu tanıma göre Ubuntu gibi, Pardus dağıtımlar bir işletim sistemi olarak isimlendirilmemelidir.

**Programlamadaki Katmanlı Yapılar**

Yazılımdaki genel olarak kod tekrarı istenmez. Bu nedenle yazılım sistemleri katmanlı bir yapıya sahip olur. Örneğin B kütüphanesi A kütüphanesinin fonksiyonlarını kullanarak yazılmış olabilir. C'de B'yi kullanarak yazılabilir. D de C'yi kullanabilir:



Kod tekrarının iki önemli dezavantajı vardır: Gereksiz kod büyümesi oluşur ve test işlemlerini zora sokar.

**API (Aplication Programming Interface) Kavramı**

Bir yazılım sisteminde uygulama programcılarının doğrudan çağırabileceği, o sistem ile uygulama programcısı arasında iskele oluşturan fonksiyon ya da sınıf kümesine API denilmektedir. API lastik bir terimdir. Hangi fonksiyonlara API denilebileceği tartışılabilir. Fakat genel olarak uygulama programcısının çağırması bulundurulmuş fonksiyonlardır. Örneğin Java API'leri denildiğinde Java sınıflarını, Windows API'leri denildiğinde Windows'un temel fonksiyonlarını anlarız.

**Library ve Framework Kavramları**

Library ve Framework kavramlarının sınırları tam belli değildir. Değişik kaynaklar bu sınırları değişik biçimde çizebilmektedir. Fakat bir sistemin framework olarak tanımlanabilmesi için şu özelliklerin bulunması gerekir:

- Karmaşıklığın kullanıcıya daha basit gösterilmesi ve bazı hammaliye işlemlerin kullanıcının üzerinden alınması

- Kod akışının ele geçirilmesi ve duruma göre programcıya belli zamanlarda verilmesi

Halbuki kütüphanlerde bir akış ele geçirmek ve arka planda birtakım işlemleri bizim için yapmak gibi bir amaç yoktur. Programın akışı bizdedir. Biz istersek kütüphane fonksiyonlarını çağırırız. Onlar da faydalı işlemler yaparlar. Şüphesiz pek çok framework aynı zamanda birtakım kütüphanelere de (API'lere de) sahiptir.

Bazı ara durumlarda o şeyin framework mü yoksa kütüphane mi olarak adlandırılacağı konusunda tereddütler olabilir.

**Bazı Çok Bilinmeyen Standart C Fonksiyonları**

Bu bölümde kursta kullanılabilecek bazı yaygın C fonksiyonları ele alınacaktır.

**strtok Fonksiyonu**

strtok fonksiyonu bir yazıyı bazı karakterlere göre parse etmekte kullanılan standart bir C fonksiyonudur. Örneğin:

"ankara,adana, izmir, kars"

burada yazı ',' karakterlerinden ayrıştırılmak istenebilir. Ve "ankara", "adana", "izmir", "kars" yazıları elde edilmek istenebilir. Ya da örneğin:

"10/12/2007"

burada tarihin gün, ay, yıl bileşenleri elde edilmek istenebilir.

strtok fonksiyonunun prototipi şöyledir:

#include <string.h>

char \*strtok(char \*str, const char \*delim);

Fonksiyon birinci parametresiyle belirtilen yazı içerisinde ikinci parametresiyle belirtilen karakterleri arar. Bunlardan birini bulursa oraya '\0' yerleştirip, o kısmın adresiyle geri döner. Eğer birinci parametre NULL geçilirse fonksiyon kaldığı yerden devam eder. Eğer bulunacak hiçbir atom kalmamışsa fonksiyon NULL adresle geri döner. Tipik kullanım şöyledir:

#include<stdio.h>

#include<string.h>

int main(void)

{

char text[] = "ankara,izmir,adana,eskisehir";

char \*str;

str = strtok(text, ",");

while (str != NULL) {

puts(str);

str = strtok(NULL, ",");

}

return 0;

}

Tabi while yerine for döngüsü de kullanılabilirdi:

#include<stdio.h>

#include<string.h>

int main(void)

{

char text[] = "ankara,izmir,adana,eskisehir";

char \*str;

for (str = strtok(text, ","); str != NULL; str = strtok(NULL, ","))

puts(str);

return 0;

}

strtok fonksiyonu ile karmaşık parse işlemleri yapılamaz. Örneğin biz fonksiyonla bir C programını atomlarına ayıramayız. strtok ile ancak basit parse işlemleri yapılabilir. strtok fonksiyonu aşağıdaki gibi yazılabilir:

#include<stdio.h>

#include<string.h>

#include<ctype.h>

char \*mystrtok(char \*str, constchar \*delim)

{

staticchar \*pos;

char \*beg;

if (str != NULL)

pos = str;

while (\*pos != '\0'&& strchr(delim, \*pos) != NULL)

++pos;

if (\*pos == '\0')

returnNULL;

beg = pos;

while (\*pos != '\0'&& strchr(delim, \*pos) == NULL)

++pos;

if (\*pos != '\0')

\*pos++ = '\0';

return beg;

}

int main(void)

{

char text[] = "ankara, izmir";

char \*str;

for (str = mystrtok(text, " ,"); str != NULL; str = mystrtok(NULL, " ,"))

puts(str);

return 0;

}

Örneğin bir dosya strtok ile şöyle parse edilebilir:

#include<stdio.h>

#include<stdlib.h>

#include<string.h>

#defineMAX\_LINE 1024

int main(void)

{

FILE \*f;

char line[MAX\_LINE];

char \*str;

if ((f = fopen("test.txt", "r")) == NULL) {

fprintf(stderr, "cannot open file!..\n");

exit(EXIT\_FAILURE);

}

while (fgets(line, MAX\_LINE, f) != NULL) {

for (str = strtok(line, "\t\n"); str != NULL; str = strtok(NULL, "\t\n"))

puts(str);

printf("---------------\n");

}

fclose(f);

return 0;

}

**remove Fonksiyonu**

remove bir dosyayı silmek için kullanılan standart bir C fonksiyonudur. Prototipi şöyledir:

#include <stdio.h>

int remove(const char \*path);

Fonksiyon parametre olarak silinecek dosyanın yol ifadesi (path name) alır. Başarı durumunda sıfır, baqşarıszlık durumunda -1 değerine geri döner.

Örneğin:

#include<stdio.h>

#include<stdlib.h>

int main(void)

{

if (remove("test.txt") == -1) {

fprintf(stderr, "cannot delete file!..\n");

exit(EXIT\_FAILURE);

}

printf("Ok\n");

return 0;

}

**rename Fonksiyonu**

rename dosyanın ismini değiştirmek için kullanılan standart bir C fonksiyonudur. Prototipi şöyledir:

#include <stdio.h>

int rename(const char \*old, const char \*new);

Fonksiyon birinci parametresi dosyanın eski yol ifadesi, ikinci parametresi yeni yol ifadesidir. Fonskiyon başarı durumunda sıfır değerin, başarıszlık durumunda -1 değerine geri döner. Örneğin:

#include<stdio.h>

#include<stdlib.h>

int main(void)

{

if (rename("test.txt", "x.txt") == -1) {

fprintf(stderr, "cannot rename file!..\n");

exit(EXIT\_FAILURE);

}

printf("Ok\n");

return 0;

}

**system Fonksiyonu**

system fonksiyonu kabuk programını interaktif olmayan modda çalıştırarak parametresiyle belirtilen kabuk komutunun kabuk tarafından çalıştırılmasını sağlar. Fonksiyonun protipi şöyledir:

#include <stdlib.h>

int system(const char \*string);

Fonksiyon parametre olarak kabuk komutunu yazı olarak alır. Fonksiyonun geri dönüş değeri sistemden sisteme değişebilmektedir. (implementation dependent). Pek çok sistemde fonksiyon başarı durumunda sıfıt, başarısızlık durumunda -1 değerine geri döner.

Örneğin:

#include<stdio.h>

#include<stdlib.h>

int main(void)

{

if (system("ren x.txt a.txt") != 0) {

fprintf(stderr, "system failed!..\n");

exit(EXIT\_FAILURE);

}

printf("Ok\n");

return 0;

}

**Anahtar Notlar:** Linux aslında bir çekirdek projesidir. Bir Linux dağıtımıyla bilgisaayara yüklediğimiz tüm programlar farklı proje grupları tarafından oluşturulmuştur. Bunlar açık kaynak kodlu olduğu için gerekirse kaynak kodları indirilerek incelenebilir. Örneğin bash, gnome, kde, tarayıcılar, editörler vs. hepsi farklı projelerdir.

**Geçici Dosya Kavramı ve Geçici Dosya Oluşturan Fonksiyonlar**

Bazen çeşitli nedenlerden dolayı bir dosya yaratıp bazı işlemleri o dosya üzerinden yapıp sonra da o dosyayı silmek isteyebiliriz. Bu amaçla kullanılan dosyalara geçici dosyalar (temporary files) denilmektedir. Örneğin tipik olarak C derleyicilerinde önişlemci kaynak kodu okur, onu #'li ifadelerden arındırır ve bir geçici dosyaya yazar. Derleme modülü de onu alarak derler. Tabi bu geçici dosya işlem sonucunda silindiği için biz onu göremeyiz. Fakat bazen elektrik kesilmesi gibi nedenlerle bu dosyalar açıkça görünür hale gelir. Çoğu kez bir dosya üzerinde insert gibi işlemler için geçici dosyalar kullanılmaktadır.

Geçici dosyaların oluşturulmasındaki en önemli sorunlardan biri isim çakışmasıdır. Ya geçici olarak açmak dosya ile aynı isimli dosya zaten varsa?..

**tmpfile Fonksiyonu**

Fonksiyonun prototipi şöyledir:

#include <stdio.h>

FILE \*tmpfile(void);

Fonksiyon "w+b" modunda yeni bir dosyayı isim çakışması olmadan yaratır bize dosya bilgi göstericisi ile geri döner. Biz dosyayı kullandıktan sonra onu fclose ile kapatırız. Bu durumda dosya otomatik silinmektedir. Fonksiyon başarısızlık durumunda NULL adrese geri döner.

Örneğin:

#include<stdio.h>

#include<stdlib.h>

int main(void)

{

FILE \*f;

int i, val;

if ((f = tmpfile()) == NULL) {

fprintf(stderr, "cannot create temporary file!..\n");

exit(EXIT\_FAILURE);

}

for (i = 0; i < 100; ++i)

if (fwrite(&i, sizeof(int), 1, f) != 1) {

fprintf(stderr, "cannot write file!..\n");

exit(EXIT\_FAILURE);

}

fseek(f, 0, SEEK\_SET);

while (fread(&val, sizeof(int), 1, f) == 1)

printf("%d ", val);

if (ferror(f)) {

fprintf(stderr, "cannot read file!...\n");

exit(EXIT\_FAILURE);

}

printf("\n");

fclose(f);

return 0;

}

**tmpnam Fonksiyonu**

Bu fonksiyon geçici dosyayı kendisi açmaz. Bize çakışmayan bir geçici dosya ismi verir. Prototipi şöyledir:

#include <stdio.h>

char \*tmpnam(char \*s);

Fonksiyon parametre olarak bizden geçici dosya isminin yerleştirileceği dizinin adresini ister. Fakat parametre NULL adres olarak da girilebilir. Bu durumda fonksiyon kendi içerisindeki static bir diziye dosya ismini yerleştirir ve onun adresiyle geri döner. Fonksiyon başarısızlık durumunda NULL adrese geri dönmektedir. (Böyle bir başarısızlığın oluşma olasılığı yok denecek kadar zayıftır.)

Aşağıdaki örnekte bir dosyadaki #'li staırlar silinmektedir:

#include<stdio.h>

#include<stdlib.h>

#include<ctype.h>

#defineLINE\_LEN 4096

int issharp(constchar \*str)

{

while (isspace(\*str))

++str;

return \*str == '#';

}

int main(intargc, char \*argv[])

{

FILE \*f, \*ftemp;

char \*ftempnam;

char buf[LINE\_LEN];

if (argc != 2) {

fprintf(stderr, "wrong number of arguments!..\n");

exit(EXIT\_FAILURE);

}

if ((f = fopen(argv[1], "r")) == NULL) {

fprintf(stderr, "cannot open file!..\n");

exit(EXIT\_FAILURE);

}

if ((ftempnam = tmpnam(NULL)) == NULL) {

fprintf(stderr, "cannot get temporary file name!..\n");

exit(EXIT\_FAILURE);

}

if ((ftemp = fopen(ftempnam, "w")) == NULL) {

fprintf(stderr, "cannot create temporary file!..\n");;

exit(EXIT\_FAILURE);

}

while ((fgets(buf, LINE\_LEN, f)) != NULL) {

if (!issharp(buf) && fputs(buf, ftemp) == EOF) {

fprintf(stderr, "cannot write file!..\n");

exit(EXIT\_FAILURE);

}

}

fclose(f);

fclose(ftemp);

if (remove(argv[1]) == -1) {

fprintf(stderr, "cannot deleta file!..\n");;

exit(EXIT\_FAILURE);

}

if (rename(ftempnam, argv[1]) == -1) {

fprintf(stderr, "cannot rename file!..\n");;

exit(EXIT\_FAILURE);

}

return 0;

}

**UNIX/Linux Sistemlerinde C Programlarının Derlenerek Çalıştırılması**

UNIX/Linux sistemlerinde ağırlıklı olarak (neredeyse her zaman) gcc derleycisi kullanılmaktadır. Program bir text editörde yazılır (vi, kate, emacs, Microsoft code, blufish gibi). Derleme işlemi şöyle yapılır:

gcc -o <çalıştırılabilen dosya ismi><kaynak dosya ismi>

Örneğin:

gcc -o sample sample.c

Eğer -o seçeneği kullanılmamışsa çalıştırılabilen dosya ismi a.out olur.

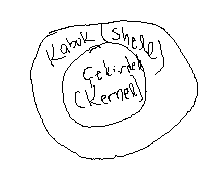
Programı çalıştırabilmek için yalnızca ismi yazılmaz. Onun dizini de belirtilmelidir. Tipik çalıştırma şöyle yapılır:

./sample

gcc derleyicisi pek çok sisteme port edilmiştir. gcc'nin Windows port'una mingw denilmektedir.

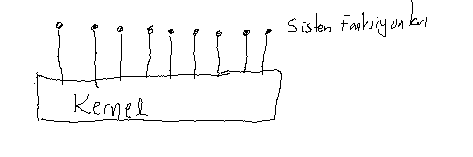
**İşletim Sisteminin Sistem Fonksiyonları, POSIX Fonksiyonları ve Standart C Fonksiyonları**

İşletim sistemleri hali hazırda C Programlama dilinde prosedürel teknikle yazılmış programlardır. İşletim sistemleri kabaca çekirdek ve kabuk kısımlarında oluşur.



Çekirdek asıl kontrol yazılımlarının bulunduğu kısımdır. Kabuk ise kullanıcıyla arayüz oluşturan kısımdır. Örneğin Windows'ta masaüstü kabuk kısımdır. UNIX/Linux sistemlerinde bash bir kabuk programdır.

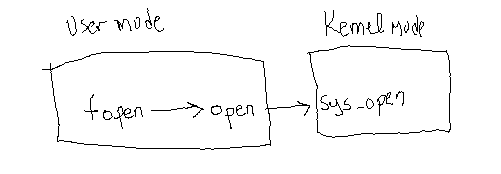
İşletim sistemlerinin çekirdeklerinde binlerce fonksiyon bulunur. Bunların küçük bir kısmı dışarıdan da önemli bazı işleri yapmak için çağrılabilmektedir. Bunlara sistem fonksiyonları (system call) denilmektedir. Her işletim sisteminin sistem fonksiyonlarının isimleri, parametrik yapıları farklıdır. Biz C Programcısı olarak bu sistem fonksiyonlarını doğrudan çağırabiliriz.



Her sistemin sistem fonksiyonları farklı olduğu için sistem fonksiyonları taşınabilir değildir. Bazı faaliyetler tamamen işletim sisteminin kontrolündedir. O faaliyetleri gerçekleştirmek isteyen herkes eninde sonunda işletim sisteminin ilgili sistem fonksiyonunu çağırmak zorundadır. Örneğin bir dosyayı silmek için bir sistem fonksiyonu vardır. Kullandığımız dil ne olursa olsun, eninde sonunda dosya bu fonksiyonla silinmek zorundadır. Çünkü bunun başka yolu yoktur.

POSIX fonksiyonları UNIX türevi sistemlerdeki ortak fonksiyonlardır. POSIX standartları temelde hem kabuk komutlarını hem de C'den çağrılacak ortak fonksiyonları belirlemektedir. POSIX fonksiyonları Linux gibi, BSD gibi, Solaris gibi hatta MAC OS X gibi sistemlerde aynı biçimde kullanılabilmektedir. Bazı POSIX fonksiyonları doğrudan o sistemdeki bir sistem fonksiyonu çağırır. Bazı POSIX fonksiyonları hiçbir bir sistem fonksiyonu çağırmaz. Bazıları da birden fazla sistem fonksiyonunu çağırabilmektedir. Örneğin dosya açmak için open isimli bir POSIX fonksiyonu kullanılmaktadır. Bu fonksiyon Linux sistemlerinde sys\_open isimli sistem fonksiyonunu çağırmaktadır.

Standart C fonksiyonları ise tüm C derleyicilerinde bulunan fonksiyonlardır. En geniş taşınabilirliğe sahip olan fonksiyonlar bunlardır. Örneğin Linux sistemlerde fopen fonksiyonu open POSIX fonksiyonunu çağırır, open POSIX fonksiyonu da sys\_open isimli sistem fonksiyonu çağırır.



Peki bazen işletim sisteminin sistem fonksiyonunu doğrudan çağırmak gerekebilir mi? Taşınabilirlik sağlamak için ortak özelliklere hitap etmek gerekmektedir. Yani örneğin Linux'ta olan fakat BSD'de olmayan bir özellik POSIX fonksiyonun konusu olamaz. Çünkü POSIX fonksiyonları tüm UNIX türevi sistemler için düşünülmüştür. İşte biz bazen spesifik bir sisteme özgü işlemler yapmak isteyebiliriz. Bu durumda doğrudan sistem fonksiyonlarını çağırmak isteyebiliriz. Soru: Linux için fopen mı, open mı, yoksa sys\_open mı daha geniş olnaklara sahiptir. İşte Linux'un sys\_open fonksiyonu Linux'a özgü yazılmıştır. Halbuki fopen fonksiyonu tüm sistemlerde olabilecek özelliklere göre tanımlanmıştır.

**Proses Kavramı**

Program kaynak kod için ya da çalıştırılabilen dosya için kullanılan bir terimdir. Bir program çalıştırıldığında artık proses (process) olur. Yani proses çalışmakta olan programlara denilmektedir. Bir proses yaratıldığında (yani bir program çalıştırıldığında) işletim sistemi onu izlemek için çekirdek alanında bir veri yapısı oluşturur. Bu veri yapısına kavramsal olarak "Proses Kontrol Bloğu (Process Control Bloack)" denilmektedir. Örneğin Linux sistemlerinde Process Kontrol Bloğu task\_struct isimli yapıdır. Pekiyi Proces Kontrol Bloğunda hangi bilgiler saklanmaktadır? İşte tipik bazı bilgiler şunlardır:

- Prosesin erişim hakları

- Prosesin bellek alanı ile ilgili bilgiler

- Prosesin çizelgelemeyle ilgili bilgileri

- Prosesin çeşitli istatistiksel bilgileri

- Prosesin açmış olduğu dosyaların kaydı

- Prosesin çalışma dizini (current working directory)

- Prosesin o anki durumu

- ...

Proses terimi ile task terimi aynı anlamda kullanılmaktadır. (Fakat bazı sistemlerde araya birtakım farklılıklar konulabilmektedir. Fakat genel olarak bu iki terim eşdeğer kabul edilir.) Proses Kontrol Bloğu çekirdek alanı içerisinde tutulur. Böylece user moddan sıradan proseslerin buraya erişerek burayı bozması engellenmiş olur. Şüphesiz işletim sistemi tüm Proses Kontrol Bloklarını birbirlerine bağlı listelerle bağlamıştır. Böylece işletim sistemi istediği zaman bu listeyi dolaşabilir. Bir prosesi yok edebilir.

Proses Kontrol Bloğu içerisinde göstericilerin de bulunduğu ağaç gibi bir yapıdır. Bir bilginin Proses Kontrol Bloğunda olması demek doğudan ya da dolaylı olarak Proses Kontrol Bloğu Yoluyla ona erişilmesi demektir.

Prosesler arasında Altlık-Üstlük (Parent-Child) ilişkisi de vardır. Bir proses işletim sisteminin sistem fonksiyonuyla yaratılır. (Örneğin sys\_fork, Windows'ta CreateProcess). Prosesi oluşturan prosese üst-proses (parent process), yeni oluşturulan prosese alt-proses (child process) denilmektedir. Bir proses yaratıldığında üst prosesin proses kontrol bloğundaki pek çok b ilgi alt prosesin proses kontrol bloğuna aktarılır. Böylece alt proses de aynı haklara sahip bir biçimde çalışır. Örneğin bizim yazdığımız sample.exe prosesi notepad.exe programını çalıştırıp bir proses oluşturabilir. Bu durumda sample.exe prosesi üst proses, notepad.exe prosesi de alt proses olur.

**C'de Tanımsız Davranış (Undefined Behavior), Derleyiciye Bağlı Davranış (Implementation Dependent Behavior) ve Belirsiz Davranış (Unspecified Behavior)**

C'de bazı durumlarda ne olacağı tam olarak tanımlanmamıştır. Bu tür durumlardxan kaçınmak gerekir. Bunlara tanımsız davranışa yol açan kodlar denilmektedir. Tanımsız davranışa yol açan kodlar derleme aşamasında herhangi bir problem oluşturmazlar. Bu kodlar dilin sentaks yapısına tamamen uygundur. Fakat program çalışırken artık her şey olabilir. Program çökebilir, yanlış çalışabilir ya da hiçbirşey olmayabilir. Bu tür kodlardan kaçınmak gerekir. Örneğin nereyei gösterdiği belli olmayan bir göstericinin gösterdiği yere \* ya da [] operatörleriyle erişmek tanımsız davranışa yol açar. Bir ifadede bir nesne ++ ya da -- operatörüyle kullanılmışsa o ifadede artık o nesne gözükmemelidir. Aksi halde tamnımsız davranış oluşur.

Bazı durumlarda standartlar belirlemeyi dokümante etmek koşuluyla derleyicileri yazanlara bırakmıştır. Bu tür kodlara derleyiciye bağlı kodlar denir. Örneğin C'de işaretli bir tamsayının sağa öteelenmesinde işaret bitinin korunup korunmayacağı böyledir. Ya da int türünün kaç byte uzunlukta lduğu böyledir.

Bazı durumlarda ise birkaç seçenek vardır. Bu seçeneklerden herhangi birini derleyiciyi yazanlar seçmiş olabilir. Fakat bunu dokümante etmek zorunda değillerdir. Böyle kodlara "belirsiz davranışa yol açan kodlar" denilmektedir. Örneğinargümanlardan parametre değişkenlerine aktarım sırası soldan-sağa ya da sağdan-sola olabilir. Bunun nasıl olacağı derleyiciden derleyiciye değişebilir. Derleyicileri yazanlar bunu dokümante etmek zorunda değillerdir.

**Windows API Fonksiyonlarında Kullanılan typedef Türleri**

Windows'un API fonksiyonlarının büyük çoğunluğunun prototipleri <windows.h> başlık dosyası içerisindedir. API fonksiyonlarında taşınabilirliği artırmak için çeşitli typedef isimleri kullanılmıştır. Bunların typedef bildirimleri <windows.h> içerisinde yapılmış durumdadır.

API fonksiyonları Pascal stili isimlendirilmiştir. (Yani her sözcüğün ilk harfi büyüktür. Örneğin CreateFile, FindFirstFile gibi.) Ve genellikle önce eylem sonra onun nesnesi gelir. (Örneğin CreateWindow gibi.) Windows API fonksiyonlarında Macar Notasyonu (Hungarian Notation) kullanılmıştır. Macar notasyonunun anahtar özellikleri şöyledir:

- Değişken isimleri onların türelerini belirten küçük harfli öneklerle başlatılmıştır. Tipik kullanıan önekler şunlardır:

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

- Yapı isimleri yine yapıyı temsil eden öneklerle başlatılır. Örneğin:

RECT rectWindow;

POINT ptRef;

- Değişken harflendirmesinde Macar Notasyonu kullanılmaktadır.

- tytpedef tür isimleri büyük harflerle isimlendirilmektedir.

Windows sistemlerindeki çok kullanılan typedef tür isimleri şöyledir:

|  |  |
| --- | --- |
| BYTE | Bir byte'lık işaretsiz tamsayı türü (unsigned char) |
| WORD | İki byte'lık işaretsiz tamsayı türü (unsigned short int) |
| DWORD | Dört byte'lık işaretsiz tamsayı türü (unsigned long int ya da unsigned int) |
| HANDLE | Handle türü (void \*) |
| PXXX, LPXXX | XXX türünden adres türü (örneğin LPVOID, PVOID, LPDWORD) |
| PCXXX, LPCXXX | XXX türünden gösterdiği yer const olan adres (Örneğin LPCVOID demek const void \* demektir) |
| LPSTR | Yazıyı gösteren adres (char \*) |
| LPTSTR | Yazıyı gösteren UNICODE destekli adres (char \* ya da wchar\_t \*) |
| BOOL | int türünü belirtir. Fakat anlam olarak başarı ve başarısızlık dünülmelidir. Geri dönüş değeri BOOL olan API fonksiyonları başarı durumunda sıfır dışı değere, başarızılık durumunda sıfır değerine geri dönmektedir. |

Fonksiyon prototiplerinde parametre değişkenlerinin önündeki \_\_in, \_\_out ve \_\_in\_out sözcükleri okunabilirliği artırmak için düşünülmüştür. Aslında bunlar aşağıdaki gibi define edilmiş makrolardır:

#define \_\_in

#define \_\_out

#define \_\_in\_out

Yani bu makrolar önişlemci tarafından silinmektedir. \_\_in makrosu fonksiyonun parametre değişkenindeki bilgiyi kullanacağı fakat ona bir değer yerleştirmeyeceği anlamına gelir. \_\_out tam tersine fonksiyonun parametre değişkenindeki değeri değiştireceği anlamına gelmektedir. \_\_in\_out ise fonksiyonun hem parametre değişkenindeki değeri kullanacağı hem de ona yeni bir değer yerleştireceği anlamına gelmektedir. Fakat bu makroların gerekliliği tartışmalıdır. Zaten gösterici olmayan parametre değişkenleri \_\_in olmak zorundadır. Gösterici parametre değişkenlerinde \_\_in ya da \_out durumu göstericinin const olup olmamasıyla zaten anlaşılmaktadır. O halde bunun tek faydası \_\_in\_out durumu için olabilir.

**Windows API Fonksiyonlarının Başarısızlık Nedenlerinin Elde Edilmesi**

Windows sistemlerinde bir API fonksiyonu başarız olduğunda onun hangi nedenden dolayı başarısız olduğu bize doğrudan verilmez. Her başarısızlık için DWORD türden bir değer tanımlanmıştır. Son çağrılan API fonksiyonunun başarısızlık nedeni GetLastError API fonksiyonuyla elde edilir:

DWORD GetLastError(void);

Hangi hata kodlarının hangi başarızlıkları belirttiği MSDN yardım dokümanlarında belirtilmektedir. Tabi programlarda hatanın sayısal kodu kullanıcı için pek bir anlam ifade etmemektedir. İşte bir hata koduna karşı onun hata yazısını veren FormatMessage isimli bir API fonksiyonu da vardır. Bu fonksiyonun kullanımık biraz detaylıdır. Burada bu detay ele alınmayacaktır. Biz kursumuzda API fonksiyonu başarısız olduğunda hatayı GetLastError ile alan ve bunu FormatMessage fonksiyonuna vererek hata yazısını elde eden bunu da stderr dosyasına yazdırıp programı sonlandıran ExitSys isimli bir fonksiyon kullanacağız:

void ExitSys(LPCSTRlpszMsg, intstatus)

{

DWORD dwLastError = GetLastError();

LPTSTR lpszErr;

if (FormatMessage(FORMAT\_MESSAGE\_ALLOCATE\_BUFFER | FORMAT\_MESSAGE\_FROM\_SYSTEM, NULL, dwLastError,

MAKELANGID(LANG\_NEUTRAL, SUBLANG\_DEFAULT), (LPTSTR)&lpszErr, 0, NULL)) {

fprintf(stderr, "%s: %s", lpszMsg, lpszErr);

LocalFree(lpszErr);

}

exit(status);

}

ExitSys bir API donksiyonu değildir. Bizim tarafımızdan yazılmış yardımcı bir fonksiyondur. Örneğin bir API fonksiyonu başarısız olduğunda hata mesajı şöyle yazdırılabilir:

HANDLE hFileFind;

WIN32\_FIND\_DATA finfo;

...

if ((hFileFind = FindFirstFile("test.c", &finfo)) == INVALID\_HANDLE\_VALUE)

ExitSys("FindFirstFile", EXIT\_FAILURE);

**UNIX/Linux Sistemlerindeki POSIX Fonksiyonlarının Başarısızlık Nedenlerinin Elde Edilmesi**

POSIX fonksiyonlarının çok büyük çoğunluğunun geri dönüş değeri int türdendir. Fonskiyon başarı durumunda sıfır (dikkat ediniz), başarısızlık durumunda -1 değerine geri döner. Böylece programcılar genellikle başarısızlığı şöyle tespit ederler:

if (some\_posix\_function(...) == -1) {

...

}

Fakat bazı programcılar kontrolü aşağıdaki gibi de yapabilmektedir:

if (some\_posix\_function(...) < 0) {

...

}

Bu biçimde kontrolün mikro mertebede daha etkin olduğu söylenebilir. Fakat bunun bir önemi yoktur.

Adrese geri dönen POSIX fonksiyonları başarısızlık durumunda NULL adrese geri dönmektedir.

Peki fonksiyonun neden başarısız olduğunu nasıl anlayabiliriz? UNIX/Linux sistemlerinde bir POSIX fonksiyonu başarız olduğunda errno isimli int türden bir global değişkene (tıpkı Windows sistemlerinde olduğu gibi) hata kodunu yazmaktadır. Biz de doğrudan bu errno değişkenin içerisinde değere bakabiliriz. errno kütüphanede tanmlanmıştır, bunun extern bildirimi <errno.h> dosyası içerisindedir. Bu durumda hata nedeni şöyle yazdırılabilir:

if (some\_posix\_function(...) == -1) {

fprintf(stderr, "error: %d\n", errno);

exit(EXIT\_FAILURE);

}

errno değişkenin alabileceği tüm hata değerleri <errno.h> dosyası içerisinde EXXX biçiminde sembolik sabitlerle define edilmiştir. Böylece programcı isterse aşağıdaki gibi kod yazabilir:

if (errno == EACCESS) {

...

}

POSIX standartlarında errno'nun sayısal değeri standart olarak belirlenmemiştir. Hata kodlarının sembolik sabitleri standart olarak belirlenmiştir. Biz de EXXX sembolik sabitlerini kullanmalıyız.

UNIX/Linux sistemlerinde bir POSIX fonksiyonu başarısız olursa errno değişkeninde hangi değerlerin bulunabileceği kesin olarak listelenmiştir. Bu listeye fonksiyonun dokümanlarından ulaşılabilir. Halbuki Windows sistemlerinde bir API fonksiyonu başarısız olduğunda başarısızlığın tüm nedenleri dokümante edilmemiştir.

Bu sistemlerde de hata kodunu yazıya dönüştüren strerror fonksiyonu vardır. Ancak perror isimli fonksiyon zaten errno değişkenine bakarak hata kodunu strerror fonksiyonuyla elde edip onu stderr dosyasına yazdırmaktadır:

#include <stdio.h>

void perror(const char \*s);

Fonksiyon önce parametresiyle belirtilen yazıyı yazdırır, sonra ':' karakterini basar, ondan sonra da errno'ya bakarak onun yazısını yazdırır. Bu durumda tipik hata tespiti şöyle yapılmalıdır:

if ((fd = open(...)) == -1) {

perror("open");

exit(EXIT\_FAILURE);

}

**Dizin İçerisindeki Dosyaların Elde Edilmesi**

Bir dizin içerisindeki dosyaların elde edilmesi çok karşılaşılan bir durumdur. Bu işlem için işletim sistemlerinin sistem fonksiyonları vardır. UNIX/Linux sistemlerinde bu işi yapan POSIX fonksiyonları bu sistem fonksiyonlarını çağırmaktadır. Benzer biçimde Windows'ta da bu işi yapan API fonksiyonları bulunmaktadır. (Windows'taki API fonksiyonları UNIX/Linux sistemlerindeki POSIX fonksiyonlarına benzetilebilir.) .NET ve Java gibi ortamlarda bu işlemler tek bir metoda indirgenmiştir. Fakat aslında tabi onlar da eninde sonunda POSIX fonksiyonlarını ya da API fonksiyonlarını çağırmaktadır.

**Windows Sistemlerinde Dizin İçerisindeki Dosyaların Elde Edilmesi**

Bu işlem Windows sistemlerinde şöyle yapılmaktadır:

1) Öncelikle FindFirstFile isimli API fonksiyonuyla koşulu sağlayan ilk dosyanın bilgileri elde edilir. FindFirstFile fonksiyonunun prototipi şöyledir:

HANDLE WINAPI FindFirstFile(

\_\_in LPCTSTR lpFileName,

\_\_out LPWIN32\_FIND\_DATA lpFindFileData

);

Fonksiyonun birinci parametresi dizin listesi elde edilecek dosyaların yol ifadesidir. Bu parametrede \* ve ? joker karakterleri kullanılabilir. Örneğin "C:\\Windows\\\*.\*" gibi, "C:\\Windows\\\*.exe" gibi. İkinci pareametre koşulu sağlayan ilk dosyanın bilgilerinin yerleştirileceği WIN32\_FIND\_DATA isimli bir yapı nesnesinin adresini alır. Fonksiyon başarı durumunda bir HANDLE değerine (burada void \*) geri döner. HANDLE konusu sonraki başlıkta ele alınmaktadır. Fonksiyon başarısızlık durumunda INVALID\_HANDLE\_VALUE isimli define edilmiş bir değere geri döner. WIN32\_FIND\_DATA yapısı şöyledir:

typedef struct \_WIN32\_FIND\_DATA {

DWORD dwFileAttributes;

FILETIME ftCreationTime;

FILETIME ftLastAccessTime;

FILETIME ftLastWriteTime;

DWORD nFileSizeHigh;

DWORD nFileSizeLow;

DWORD dwReserved0;

DWORD dwReserved1;

TCHAR cFileName[MAX\_PATH];

TCHAR cAlternateFileName[14];

} WIN32\_FIND\_DATA, \*PWIN32\_FIND\_DATA, \*LPWIN32\_FIND\_DATA;

Yapının dwFileAttributes elemanı dosyanın özellik bilgisini belirtir. Bu eleman bit bit anlamlıdır. Her bit bir özelliğin olup olmadığını belirtir. Bitleri maskeelemek için FILE\_ATTRIBUTE\_XXX biçiminde makrolar bulundurulmuştur. Örneğin biz bu elemanı FILE\_ATTRIBUTE\_ARCHIVE makrısuyla & işlemine sokarsak ve bundan sıfır dışı bir değer elde edersek o özelliğin varolduğunu anlamalıyız. Windows dosya sistemine de bağlı olarak her dosya için ilk yaratılma zamanını, son erişim zamanını ve son yazma zamanını turmaktadır. WIN32\_FIND\_DATA yapısında bu tarih ve zamanlar FILETIME olarak belirtilmiştir. FILETIME kullanışlı bir tür değildir. Bu 01/01/1601'den geçen 100 nanosaniyelerin sayısıdır. Buradaki tarih zamanlar UTC biçimindedir. İstenirse bu tarih ve zamanlar FileTimeToLocalFileTime fonksiyonuyla yerel saate dönüştürülebilir. FILETIME değerini normal tarih ve zaman bilgisine dönüştürmek için FileTimeToSystemTime fonksiyonu kullanılır. Örneğin:

#include<stdio.h>

#include<stdlib.h>

#include<Windows.h>

void ExitSys(LPCSTR lpszMsg, int status);

int main(void)

{

HANDLE hFileFind;

WIN32\_FIND\_DATA finfo;

SYSTEMTIME st;

if ((hFileFind = FindFirstFile("test.c", &finfo)) == INVALID\_HANDLE\_VALUE)

ExitSys("FindFirstFile", EXIT\_FAILURE);

printf("Fine Name: %s\n", finfo.cFileName);

printf("File Size: %lu\n", finfo.nFileSizeLow);

FileTimeToLocalFileTime(&finfo.ftLastWriteTime, &finfo.ftLastWriteTime);

FileTimeToSystemTime(&finfo.ftLastWriteTime, &st);

printf("Last Write Time: %02d/%02d/%04d %02d:%02d:%02d\n",

st.wDay, st.wMonth, st.wYear, st.wHour, st.wMinute, st.wSecond);

return 0;

}

void ExitSys(LPCSTRlpszMsg, intstatus)

{

DWORD dwLastError = GetLastError();

LPTSTR lpszErr;

if (FormatMessage(FORMAT\_MESSAGE\_ALLOCATE\_BUFFER | FORMAT\_MESSAGE\_FROM\_SYSTEM, NULL, dwLastError,

MAKELANGID(LANG\_NEUTRAL, SUBLANG\_DEFAULT), (LPTSTR)&lpszErr, 0, NULL)) {

fprintf(stderr, "%s: %s", lpszMsg, lpszErr);

LocalFree(lpszErr);

}

exit(status);

}

Dosyanın olası maksimum uzunluğu 264 byte olabilir. Fakat bu yapı 32 bit uyumlu olduğu için ve 32 bit sistemlerde 64 bitlik tamsayı türü doğal olmadığı için dosya uzunluğu da iki ayrı 32 bit değerle verilmiştir (nFileSizeLow, nFileSizeHigh).

2) Koşulu sağlayan diğer dosyaların bilgileri bir döngü içerisinde FindNextFile fonksiyonları çağrılarak elde edilir.

BOOL WINAPI FindNextFile(

\_\_in HANDLE hFindFile,

\_\_out LPWIN32\_FIND\_DATA lpFindFileData

);

Fonksiyonun birinci parametresi FindFirstFile fonksiyonundan elde edilen HANDLE değeridir. İkinci parametre bulunan dosyayanın bilgilerinin yerleştirileceği WIN32\_FIND\_DATA türünden nesnenin adresini alır. Fonksiyon yeni bir dosyayı bulamamışsa sıfır değerine bulmuşsa sıfır dışı bir değere geri döner.

Örneğin:

#include<stdio.h>

#include<stdlib.h>

#include<Windows.h>

void ExitSys(LPCSTR lpszMsg, int status);

int main(void)

{

HANDLE hFileFind;

WIN32\_FIND\_DATA finfo;

SYSTEMTIME st;

if ((hFileFind = FindFirstFile("\*.\*", &finfo)) == INVALID\_HANDLE\_VALUE)

ExitSys("FindFirstFile", EXIT\_FAILURE);

do {

printf("--------------------------\n");

printf("Fine Name: %s\n", finfo.cFileName);

printf("File Size: %lu\n", finfo.nFileSizeLow);

FileTimeToLocalFileTime(&finfo.ftLastWriteTime, &finfo.ftLastWriteTime);

FileTimeToSystemTime(&finfo.ftLastWriteTime, &st);

printf("Last Write Time: %02d/%02d/%04d %02d:%02d:%02d\n",

st.wDay, st.wMonth, st.wYear, st.wHour, st.wMinute, st.wSecond);

} while (FindNextFile(hFileFind, &finfo));

return 0;

}

void ExitSys(LPCSTRlpszMsg, intstatus)

{

DWORD dwLastError = GetLastError();

LPTSTR lpszErr;

if (FormatMessage(FORMAT\_MESSAGE\_ALLOCATE\_BUFFER | FORMAT\_MESSAGE\_FROM\_SYSTEM, NULL, dwLastError,

MAKELANGID(LANG\_NEUTRAL, SUBLANG\_DEFAULT), (LPTSTR)&lpszErr, 0, NULL)) {

fprintf(stderr, "%s: %s", lpszMsg, lpszErr);

LocalFree(lpszErr);

}

exit(status);

}

Dizin olan dosyaların yanına <DIR> biçiminde yazdırmak isteyelim. Bu durumda dizin olan dosyaları yapının dwFileAttributes elemanına bakarak tespit ederiz. Örneğin:

#include<stdio.h>

#include<stdlib.h>

#include<Windows.h>

void ExitSys(LPCSTR lpszMsg, int status);

int main(void)

{

HANDLE hFileFind;

WIN32\_FIND\_DATA finfo;

if ((hFileFind = FindFirstFile("c:\\windows\\\*.\*", &finfo)) == INVALID\_HANDLE\_VALUE)

ExitSys("FindFirstFile", EXIT\_FAILURE);

do {

printf("%-40s%s\n", finfo.cFileName, finfo.dwFileAttributes &FILE\_ATTRIBUTE\_DIRECTORY ? "<DIR>" : "");

} while (FindNextFile(hFileFind, &finfo));

return 0;

}

void ExitSys(LPCSTRlpszMsg, intstatus)

{

DWORD dwLastError = GetLastError();

LPTSTR lpszErr;

if (FormatMessage(FORMAT\_MESSAGE\_ALLOCATE\_BUFFER | FORMAT\_MESSAGE\_FROM\_SYSTEM, NULL, dwLastError,

MAKELANGID(LANG\_NEUTRAL, SUBLANG\_DEFAULT), (LPTSTR)&lpszErr, 0, NULL)) {

fprintf(stderr, "%s: %s", lpszMsg, lpszErr);

LocalFree(lpszErr);

}

exit(status);

}

3) En sonunda FindFirstFile fonksiyonuyla elde edilen handle FindClose fonksiyonuyla kapatılmalıdır. Örneğin:

int main(void)

{

HANDLE hFileFind;

WIN32\_FIND\_DATA finfo;

SYSTEMTIME st;

if ((hFileFind = FindFirstFile("c:\\windows\\\*.\*", &finfo)) == INVALID\_HANDLE\_VALUE)

ExitSys("FindFirstFile", EXIT\_FAILURE);

do {

printf("%-40s%s\n", finfo.cFileName, finfo.dwFileAttributes &FILE\_ATTRIBUTE\_DIRECTORY ? "<DIR>" : "");

} while (FindNextFile(hFileFind, &finfo));

FindClose(hFileFind);

return 0;

}

**UNIX/Linux Sistemlerinde Dizin İçerisindeki Dosyaların Elde Edilmesi**

UNIX/Linux sistemlerinde de işlemler aşağıdaki biçimde yapılmaktadır:

1) Öncelikle dizin opendir isimli bir POSIX fonksiyonuyla açılır:

#include <dirent.h>

DIR \*opendir(const char \*name);

Fonksiyon parametre olarak listesi elde edilecek dizinin yol ifadesini alır. Geri dönüş değeri DIR isimli bir yapı türünden adrestir. Programcının bu yapaıyı bilmesi gerekmez. (Tıpkı fopen fonksiyonunun geri döndürdüğü FILE gibi). Bu bir handle olarak kullanılmaktadır. Fonksiyon başarısızsa NULL adrese geri döner.

2) Dizindeki dosyalar bir döngü içerisinde readdir fonksiyonu çağrılarak elde edilir:

#include <dirent.h>

struct dirent \*readdir(DIR \*dirp);

Fonksiyon opendir fonksiyonundan elde edilen DIR türünden adresi parametre olarak alır ve struct dirent türünden static ömütlü bir yapı nesnesinin adresiyle geri döner. Dizin listesinin sonuna geinmişse fonksiyon NULL adrese geri dönmektedir. dirent yapısı şöyle bildirilmiştir:

struct dirent {

ino\_t d\_ino;

char d\_name[256];

};

Çeşitli UNIX türevi sistemler bu yapıya eklenti biçiminde eleman ekleyebilmektedir. Ancak POSIX standartlarında yapı böyle bildirilmiştir.

Görüldüğü gibi readdir bize dosyanın isminden ve i-node numarasından başka bir bilgi vermemektedir. i-node numarasının ne anlam ifade ettiği başka bir bölümde ele alınacaktır. i-node numarası dosyayı betimleyen sistem genelinde tek olan (unique) bir numaradır.

readdir fonksiyonu bize hem normal dosyaları hem de dizinleri vermektedir.

3) İşlem bitince closedir fonksiyonuyla açılan dizin kapatılır. Tabi dosyalarda olduğu gibi biz bunu kapatmamışsak proses sonlandığında dizin sorunsuz olarak işletim sistemi tarafından kapatılmaktadır.

#include <dirent.h>

int closedir(DIR \*dirp);

Fonksiyon başarı durumunda 0, başarısızlık durumunda -1 değerine geri döner. Başarının kontrol edilmesine gerek yoktur.

Örneğin:

#include<stdio.h>

#include<stdlib.h>

#include<dirent.h>

int main(void)

{

DIR \*dir;

struct dirent \*dire;

if ((dir = opendir("/usr/include")) == NULL) {

perror("opendir");

exit(EXIT\_FAILURE);

}

while ((dire = readdir(dir)) != NULL)

printf("%s\n", dire->d\_name);

closedir(dir);

return 0;

}

**stat POSIX Fonksiyonu**

Bir dosyanın yol ifadesi biliniyorsa onun metadata bilgilerini elde etmek stat fonksiyonu kullanılmaktadır. stat fonksiyonunun fstat ve lstat isimli kardeşleri de vardır (fsata ve lstat burada ele alınmayacaktır). stat fonksiyonu UNIX türevi sistemler için çok önemli bir fonksiyondur:

#include <sys/stat.h>

int stat(const char \*path, struct stat \*buf);

Fonksiyonun birinci parametresi bilgisi elde edilecek dosyanın yol ifadesini alır. İkinci parametre dosya bilgilerinin yerleştirileceği struct stat türünden yapı nesnesinin adresini almaktadır. Fonksiyon başarı durumunda 0, başarısızlık durumunda -1 değerine geri döner. stat yapısı <sys/stat.h> içerisinde aşağıdaki gibi bildirilmiştir:

struct stat {

dev\_t st\_dev; /\* ID of device containing file \*/

ino\_t st\_ino; /\* inode number \*/

mode\_t st\_mode; /\* protection \*/

nlink\_t st\_nlink; /\* number of hard links \*/

uid\_t st\_uid; /\* user ID of owner \*/

gid\_t st\_gid; /\* group ID of owner \*/

dev\_t st\_rdev; /\* device ID (if special file) \*/

off\_t st\_size; /\* total size, in bytes \*/

blksize\_t st\_blksize; /\* blocksize for filesystem I/O \*/

blkcnt\_t st\_blocks; /\* number of 512B blocks allocated \*/

time\_t st\_atime; /\* time of last access \*/

time\_t st\_mtime; /\* time of last modification \*/

time\_t st\_ctime; /\* time of last status change \*/

};

Yapının elemanları önemli bilgiler vermektedir. Ancak bu elemanların hepsi kursun bu noktasında el alınmayacaktır. Yapının st\_size elemanı dosyanın uzunluğunu, st\_atime, st\_mtime ve st\_ctime elemanları dosyanın erişim zamanlarını bize verir. off\_t işaretli bir tamsayı türündendir. time\_t ise 01/01/1970 tarihinden geçen saniye sayısını belirten artimetik bir türdür.

Örneğin:

#include<stdio.h>

#include<stdlib.h>

#include<time.h>

#include<sys/stat.h>

int main(void)

{

struct stat finfo;

if (stat("/usr/include/stdio.h", &finfo) < 0) {

perror("stat");

exit(EXIT\_FAILURE);

}

printf("%ld\n", (long)finfo.st\_size);

printf("%s\n", ctime(&finfo.st\_mtime));

return 0;

}

Dizin listesi elde edilirken her bulunan fonksiyon stat fonksiyonuna sokularak onun bilgileri elde edilebilir. Aslında ls programı da zaten böyhle yapmaktadır. Örneğin:

#include<stdio.h>

#include<stdlib.h>

#include<time.h>

#include<dirent.h>

#include<sys/stat.h>

#define MAX\_PATH 2048

int main(int argc, char \*argv[])

{

DIR \*dir;

struct dirent \*dire;

struct stat finfo;

char path[MAX\_PATH];

if (argc != 2) {

fprintf(stderr, "wrong number of arguments!..\n");

exit(EXIT\_FAILURE);

}

if ((dir = opendir(argv[1])) == NULL) {

perror("opendir");

exit(EXIT\_FAILURE);

}

while ((dire = readdir(dir)) != NULL) {

sprintf(path, "%s/%s", argv[1], dire->d\_name);

if (stat(path, &finfo) < 0) {

perror("stat");

exit(EXIT\_FAILURE);

}

printf("---------------------\n");

printf("%s\n", dire->d\_name);

printf("%ld\n", (long)finfo.st\_size);

printf("%s\n", ctime(&finfo.st\_mtime));

}

closedir(dir);

return 0;

}

**Fonksiyonlar İçin Hata Kontrolleri**

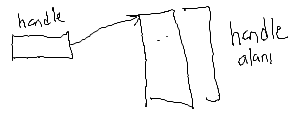
Hata kontrolü bakımından fonksiyonları 2 gruba ayırabiliriz:

1) Her zaman hata kontrolünün yapılması gereken fonksiyonlar: Bunlar sistemin o anki durumuyla ilgili biçimde başarısz olabilecek fonksiyonalrdır. Bu tür fonksiyonlar çağrılırken kesinlikle hata kontrolü yapılmalıdır. (Örneğin fopen, malloc gibi fonksiyonlar…)

2) Eğer programcı her şeyi düzgün yapmışsa, başarısız olma olasılığı olmayan fonksiyonlar için hata kontrolü yapılmayabilir. Örneğin dosya fopen ile düzgün açılmışsa fclose ile kapatılmamasının bir nedeni olamaz. Zaten böyle birşey olsa bile bizim yapabileceğimiz birşey de yoktur. (Bu tür fonksiyonlarda hata kontrolleri programın DEBUG versiyonunda yapılabilir.)

**Handle Sistemleri**

Handle bir veri yapısına erişmekte kullanılan tekil bir anahtar değerdir. Handle bir tamsayı biçiminde olabilir. Bu durumda muhtemelen handle global bir dizide bir indeks belirtmektedir. Handle bir adres biçiminde olabilir. Bu durumda doğrudan bir veri yapısını gösterir. Bazen handle void bir adres olarak karşımıza çıkabilir. (Örneğin Windows API fonksiyonlarında sıkça karşılaştığımız HANDLE ismi void \* olarak typedef edilmiştir.) Bu durumda aslında o adresin gösterdiği yerde bir veri yapısı vardır. Fakat sistemi tasarlayan kişi bunu açıklamak istememiştir. Bazen handle bozulmuş bir biçimde bize verilir. Sistem onu düzelterek veri yapısına erişir. Böylece kullan kişi oraya erişememiş olur. Handle ile erişilen veri yapısına handle alanı denilmektedir:



Bir handle sisteminde üç grup fonksiyon bulunur.

1) Handle sistemlerini yaratan ya da açan fonksiyonlar: Bunlar genellikle Windows sistemlerinde CreateXXX ya da OpenXXX biçiminde isimlendirilmektedir. Bu fonksiyonlar veri yapısını tahsis eder. Onun elemanlarına çeşitli ilkdeğerleri verirler ve handle ile geri dönerler.

2) Handle sistemini kullanan fonksiyonlar: Bunlar handle değerini bizden alıp veri yapısına erişip oradaki bilgileri kullanarak faydalı işlem yaparlar.

3) Handle sistemini kapatan fonksiyonlar: Bunlar handle alanını boşaltıp birtakım son işlemleri yaparlar. Windows sistemlerinde genellikle bu fonksiyonlar CloseXXX ya da DestroyXXX biçiminde isimlendirilmektedir.

Örneğin fopen handle sistemini yaratan bir fonksiyondur. Bize handle değerini FILE \* biçiminde verir. fgetc, fread gibi fonksiyonlar handle sistemini kullanan fonksiyonlardır. fclose da handle sistemini kapatan fonksiyodur. Ya da örneğin FindFirstFile handle alanını yaratır. Bize o alanın adresini void \* olarak verir. FindNextFile handle sistemini kullanan bir fonksiyondur. FindClose ise handle sistemini kapatan fonksiyondur.

Bir matris istemi bir handle sistemi biçiminde aşağıdaki gibi oluşturulabilir.

#include<stdio.h>

#include<stdlib.h>

typedefintDATATYPE;

/\* Type Declarations \*/

typedefstructtagMATRIX {

size\_t rowSize;

size\_t colSize;

DATATYPE \*pMatrix;

} MATRIX, \*HMATRIX;

/\* Function Prototypes \*/

HMATRIX CreateMatrix(size\_t rowSize, size\_t colSize);

void SetMatrix(HMATRIX hMatrix, DATATYPE \*vals);

DATATYPE GetElem(HMATRIX hMatrix, size\_t row, size\_t col);

void PutElem(HMATRIX hMatrix, size\_t row, size\_t col, DATATYPE val);

void DispMatrix(HMATRIX hMatrix);

void CloseMatrix(HMATRIX hMatrix);

/\* Function Definitions \*/

HMATRIX CreateMatrix(size\_trowSize, size\_tcolSize)

{

HMATRIX hMatrix;

if ((hMatrix = (HMATRIX)malloc(sizeof(MATRIX))) == NULL)

returnNULL;

hMatrix->rowSize = rowSize;

hMatrix->colSize = colSize;

if ((hMatrix->pMatrix = (DATATYPE \*)malloc(rowSize \* colSize \* sizeof(DATATYPE))) == NULL) {

free(hMatrix);

returnNULL;

}

return hMatrix;

}

void SetMatrix(HMATRIXhMatrix, DATATYPE \*vals)

{

size\_t i;

for (i = 0; i <hMatrix->rowSize \* hMatrix->colSize; ++i)

hMatrix->pMatrix[i] = vals[i];

}

DATATYPE GetElem(HMATRIXhMatrix, size\_trow, size\_tcol)

{

returnhMatrix->pMatrix[row \* hMatrix->colSize + col];

}

void PutElem(HMATRIXhMatrix, size\_trow, size\_tcol, DATATYPEval)

{

hMatrix->pMatrix[row \* hMatrix->colSize + col] = val;

}

void DispMatrix(HMATRIXhMatrix)

{

size\_t i;

for (i = 0; i <hMatrix->rowSize \* hMatrix->colSize; ++i)

printf("%-3d%s", hMatrix->pMatrix[i], i % hMatrix->colSize == hMatrix->colSize - 1 ? "\n" : "");

}

void CloseMatrix(HMATRIXhMatrix)

{

free(hMatrix->pMatrix);

free(hMatrix);

}

int main(void)

{

HMATRIX hMatrix;

int a[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12 };

if ((hMatrix = CreateMatrix(3, 4)) == NULL) {

fprintf(stderr, "cannot create matrix!..\n");

exit(EXIT\_FAILURE);

}

SetMatrix(hMatrix, a);

DispMatrix(hMatrix);

CloseMatrix(hMatrix);

return 0;

}

Handle alanı istenirse gizlenebilir. Fonksiyonlar kütüphaneye yerleştirilir. Başlık dosyasına protoipler yazılır fakat HMATRIX yapısı burada belirtilmez.

/\* Matrix.h \*/

#ifndef MATRIX\_H\_

#defineMATRIX\_H\_

typedefintDATATYPE;

typedefvoid \*HMATRIX;

/\* Function Prototypes \*/

HMATRIX CreateMatrix(size\_t rowSize, size\_t colSize);

void SetMatrix(HMATRIX hMatrix, DATATYPE \*vals);

DATATYPE GetElem(HMATRIX hMatrix, size\_t row, size\_t col);

void PutElem(HMATRIX hMatrix, size\_t row, size\_t col, DATATYPE val);

void DispMatrix(HMATRIX hMatrix);

void CloseMatrix(HMATRIX hMatrix);

#endif

/\* Matrix.c \*/

#include<stdio.h>

#include<stdlib.h>

#include"Matrix.h"

/\* Type Declarations \*/

typedefstructtagMATRIX {

size\_t rowSize;

size\_t colSize;

DATATYPE \*pMatrix;

} MATRIX;

/\* Function Definitions \*/

HMATRIX CreateMatrix(size\_trowSize, size\_tcolSize)

{

MATRIX \*matrix;

if ((matrix = (HMATRIX)malloc(sizeof(MATRIX))) == NULL)

returnNULL;

matrix->rowSize = rowSize;

matrix->colSize = colSize;

if ((matrix->pMatrix = (DATATYPE \*)malloc(rowSize \* colSize \* sizeof(DATATYPE))) == NULL) {

free(matrix);

returnNULL;

}

return matrix;

}

void SetMatrix(HMATRIXhMatrix, DATATYPE \*vals)

{

MATRIX \*matrix = (MATRIX \*)hMatrix;

size\_t i;

for (i = 0; i < matrix->rowSize \* matrix->colSize; ++i)

matrix->pMatrix[i] = vals[i];

}

DATATYPE GetElem(HMATRIXhMatrix, size\_trow, size\_tcol)

{

MATRIX \*matrix = (MATRIX \*)hMatrix;

return matrix->pMatrix[row \* matrix->colSize + col];

}

void PutElem(HMATRIXhMatrix, size\_trow, size\_tcol, DATATYPEval)

{

MATRIX \*matrix = (MATRIX \*)hMatrix;

matrix->pMatrix[row \* matrix->colSize + col] = val;

}

void DispMatrix(HMATRIXhMatrix)

{

MATRIX \*matrix = (MATRIX \*)hMatrix;

size\_t i;

for (i = 0; i < matrix->rowSize \* matrix->colSize; ++i)

printf("%-3d%s", matrix->pMatrix[i], i % matrix->colSize == matrix->colSize - 1 ? "\n" : "");

}

void CloseMatrix(HMATRIXhMatrix)

{

MATRIX \*matrix = (MATRIX \*)hMatrix;

free(matrix->pMatrix);

free(matrix);

}

#if 0

int main(void)

{

HMATRIX hMatrix;

int a[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12 };

if ((hMatrix = CreateMatrix(3, 4)) == NULL) {

fprintf(stderr, "cannot create matrix!..\n");

exit(EXIT\_FAILURE);

}

SetMatrix(hMatrix, a);

DispMatrix(hMatrix);

CloseMatrix(hMatrix);

return 0;

}

#endif

Handle sistemleri nesne yönelimli programlama tekniğindeki sınıflara oldukça benzemektedir. Başka bir deyişle biz nesne yönelimli teknikte bir handle sistemini bir sınıf olarak tasarlayabiliriz. Şöyle ki: Handle alanı sınıfın private veri elemanları olarak bildirilir. Sınıfın üye fonksiyonları (metotları) bunları ortak kullanmaktadır. (Aslında zaten static olmayan üye fonksiyonlara bu veri yapısının adresi this göstericisi olarak geçirilmektedir.)

**Özyineleme Kavramı ve Özyinelemeli Algoritmalar**

Özyineleme (recursion) bir olgunun kendisini ya da kendisine benzeyen bir olguyu içermesi durumudur. Özyineleme olgusu hem doğada rastladığımız bir oldgudur hem de bilgisayar bilimlerinde sıklıkla karşılaştığımız bir olgudur. Örneğin bir dizin içerisinde dosyalar ve başka dizinler bulunur. Dizin içerisindeki dizine geçtiğimizde orada da benzer yapı karşımıza çıkar. O halde dizin yapısı özyineleme içermektedir.

Özyineleme içeren algortmalara özyinelemeli algortimalar (recursive algorithms) denilmektedir. Örneğin bir dizin ağacını listelemek isteyelim. Dizinden girelim dosya buldukça yazdıralım. Peki karşımıza bir dizin çıkarsa ne yapacağız? Onun da içine geçelim aynı şeyleri onun için de yapalım.

Bir algoritmanın özyinelemeli olup olmadığı nasıl anlaşılır? Eğer algoritmada ilerleyip bir noktaya geldiğimizde geldiğimiz nokta başladığımız noktaya çok benzer bir durum içeriyorsa muhtemelen bu algortima özeyinelemeli bir algoritmadır.

Özyinelemeli algoritmalat tipik olarak kendi kendini çağıran fonksiyonlarla gerçekleştirilir. Tabi özyinemeli algortimaların yapay olarak bu mekanizmanın oluşturulduğu başka biçimlerde de çözümleri olabilse de tipik gerçekleştirimi kendi kendini çağıran fonksiyonlar yoluyladır.

Yazılımda karşımıza çıkan tipik özyinelemeli algortimaalardan bazıları şunlardır:

- Dizi ağacının dolaşılması

- İkili ağaç gibi algortimik ağaçların dolaşılması

- Grafların dolaşılması ve graflarda arama

- Parsing algoritmaları

- Çeşitli arama işlemleri ve problemler (Örneğin 8 vezir problemi)

- Bazı Sort işlemleri (quick sort, merger sort, heap sort) vs.

- Matemetiksel bazı algortimalar

- Bazı optimizasyon algortimaları

Algorimik problemler özyineleme bakımında üç gruba ayrılabilir:

1) Hem normal hem de özyinelemeli olarak gerçekleştirilebilecek problemler

2) Yalnızca özyinelemeli olarak gerçekleştirilebilecek problemler

3) Normal olarak gerçekleştirilebilecek problemler

**Özyinemeli Fonksiyonlar**

Bir fonksiyonun kendini çağırmasıyla başka bir fonksiyonu çağırması arasında aslında hiçbir farklılık yoktur. Örneğin:

void bar(void)

{

int a;

...

}

void foo(void)

{

int a;

...

bar();

...

}

Burada bar çağrıldığında bar'ın içerisinde yeni bir a stack'te oluşturulacaktır ve bar sona erdiğinde akış çağrılan yerden devam edecektir. Fonksiyonun kendi kendini çağırması da tamamen bu biçimde gerçekleşir. Örneğin:

void foo(void)

{

int a;

...

foo();

...

}

Burada foo kendini her çağırdığında yeni bir a yaratılır. foo'nun çalışması sona erdiğinde bir önceki çağırmadan devam eder ve artık a bir önceki çağırmada yaratılan a olur.

Tabii fonksiyonun sürekli kendini çağırması sonsuz döngü oluşmasına yol açar. O halde bir noktaya kadar fonksiyon kendini çağırmalı artık bundan vaz geçmelidir. Örneğin aşağıda fonksiyon sonsuza kadar kendini çağırmamaktadır:

#include<stdio.h>

void foo(intn)

{

printf("Giris:%d\n", n);

if (n == 0)

return;

foo(n - 1);

printf("Cikis:%d\n", n);

}

int main(void)

{

foo(3);

return 0;

}

**Özyinelemeli Fonksiyonlara Örnekler**

Burada verilecek örneklerin büyük bölümü için özyineleme gerekmemektedir. Bu örnekler mekanizmanın anlaşılması için verilmektedir.

**1) Özyinelemeli Faktöriyel Hesabı**

Faktöryel hesabının normal yapılması gereken gerçekleştirimi şöyledir:

#include<stdio.h>

unsignedlong factorial(unsignedn)

{

unsignedlong f = 1;

for (; n> 1; --n)

f \*= n;

return f;

}

int main(void)

{

long result;

result = factorial(10);

printf("%lu\n", result);

return 0;

}

Özyinelemeli versiyonu şöyle yazılabilir:

#include<stdio.h>

unsignedlong factorial(unsignedn)

{

unsignedlong result;

if (n == 0)

return 1;

result = n \* factorial(n - 1);

return result;

}

int main(void)

{

long result;

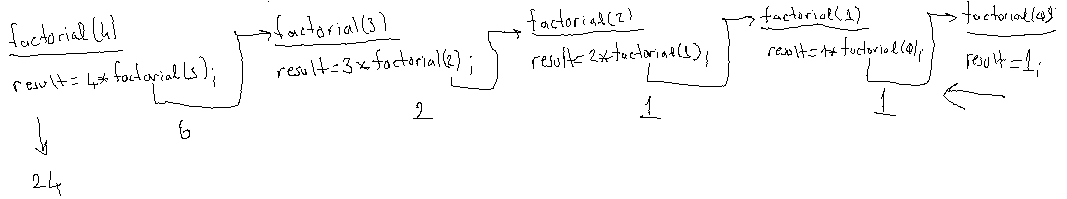
result = factorial(4);

printf("%lu\n", result);

return 0;

}

Bu fonksiyon şöyle çalışmaktadır:



Fonksiyon daha basit şöyle de yazılabilirdi:

unsignedlong factorial(unsignedn)

{

if (n == 0)

return 1;

returnn \* factorial(n - 1);

}

**2) Yazının Tersten Yazdırılması**

Özyineleme düz olan bir şeyi ters çevirmek için sık kullanılmaktadır. Bu problemde de aslında özyinelemeye hiç gerek yoktur. Problemin klasik çözümü şöyledir:

#include<stdio.h>

void putsrev(constchar \*str)

{

int i;

for (i = 0; str[i] != '\0'; ++i)

;

for (--i; i >= 0; --i)

putchar(str[i]);

}

int main(void)

{

putsrev("ankara");

putchar('\n');

return 0;

}

Fonksiyon şöyle de yazılabilirdi:

void putsrev(constchar \*str)

{

constchar \*temp = str;

while (\*str != '\0')

++str;

do {

--str;

putchar(\*str);

} while (str != temp);

}

Fonksiyonun özyinelemeli gerçekleştirimi de şöyle olabilir:

#include<stdio.h>

void putsrev(constchar \*str)

{

if (\*str == '\0')

return;

putsrev(str + 1);

putchar(\*str);

}

int main(void)

{

putsrev("ankara");

putchar('\n');

return 0;

}

**3) Bir Yazıyı Ters Yüz etme (strrev)**

Fonksiyonun klasik özyinelemeli olmayan biçimi şöyle yazılabilir:

#include<stdio.h>

void mystrrev(char \*str)

{

size\_t n, i;

char temp;

for (n = 0; str[n] != '\0'; ++n)

;

for (i = 0; i < n / 2; ++i) {

temp = str[i];

str[i] = str[n - i - 1];

str[n - i - 1] = temp;

}

}

int main(void)

{

char s[] = "ankara";

mystrrev(s);

puts(s);

return 0;

}

Fonksiyonun özyinelemeli versiyonu da şöyle oluşturulabilir:

#include<stdio.h>

void mystrrev\_recur(char \*str, intleft, intright)

{

char temp;

if (left>= right)

return;

temp = str[left];

str[left] = str[right];

str[right] = temp;

++left;

--right;

mystrrev\_recur(str, left, right);

}

void mystrrev(char \*str)

{

mystrrev\_recur(str, 0, strlen(str) - 1);

}

int main(void)

{

char s[] = "ankara";

mystrrev(s);

puts(s);

return 0;

}

Burada mystrrev bir sarma fonksiyondur (wrapper function). Asıl işi yapan fonksiyonu çağıran fonksiyonlara sarma fonksiyonlar denilmektedir.

**4) Bir sayıyı İkilik Sistemde Yazdırma**

Bunun özyineleme içermeyen klasik bir şöyle olabilir:

#include<stdio.h>

void putsbits(unsignedn)

{

int i;

for (i = 15; i >= 0; --i)

putchar(((n>> i) & 1) + '0');

putchar('\n');

}

int main(void)

{

putsbits(100);

return 0;

}

Fonksiyonun çalışması şöyle özetlenebilir (n = 10 olsun):



**5) Sayıların Yalnızca putchar ile Yazdırılması**

Aslında bilgisayar sistemlerinde ekrana sayı yazdırmak diye birşey yoktur. Yalnızca ekrana karakterler yazdırılabilir. Bu durumda aslında printf gibi bir fonksiyon int bir sayıyı yazdırırken printf sayıyı basamaklarına ayrıştırıp onlara karşı gelen karakterleri bastırmaktadır. Yani herşey aslında putchar gibi bir fonksiyonla yapılır. O halde yalnızca putchar kullanarak bir sayının yazdırılması sistem programlama için önemlidir. Bu işlem tipik olarak özyinelemeli yazılır. Bu problemin özyinelemeli olmayan çözümü özyinelemeli çözümünden daha kötüdür. Fonksiyonun özyinelemeli olmayan biçimi şöyle yazılabilir:

,

#include<stdio.h>

void putnum(intn)

{

char s[16];

int i, sign;

if (n< 0) {

sign = -1;

n = -n;

}

else

sign = 1;

for (i = 0; n; ++i) {

s[i] = n % 10 + '0';

n /= 10;

}

if (sign < 0)

s[i++] = '-';

s[i] = '\0';

for (--i; i >= 0; --i)

putchar(s[i]);

putchar('\n');

}

int main(void)

{

int n = -1235678;

putnum(n);

return 0;

}

Tipik özyinelemeli çözüm şöyledir:

#include<stdio.h>

void putnum(intn)

{

if (n< 0) {

putchar('-');

n = -n;

}

if (n / 10)

putnum(n / 10);

putchar(n % 10 + '0');

}

int main(void)

{

int n = -1235678;

putnum(n);

putchar('\n');

return 0;

}

**6) Kapalı Şeklin İçinin Boyanması**

Bu algoritmaya su basması (flood fill) denilmektedir. Tipik olarak kapalı şeklin içerisinde bir nokta alınır. O nokta boyanır. Sonra fonksiyon 4 yönde kendini çağırarak ilerler. Tabi kapalı bölgenin sınırına geldiği zaman ya da daha önce boyanan bir yere geldiği zaman durur. Örneğin g\_bitmap isimli char türden bit matriste '#' karakteri ile oluşturulmuş kapalı bir resim olsun. Resmin içinin boyanması aşağıdaki gibi özyinelemeli bir fonksiyonla yapılabilir:

void flood\_fill(introw, intcol, charch)

{

if (g\_bitmap[row][col] == ch || g\_bitmap[row][col] == '#')

return;

g\_bitmap[row][col] = ch;

flood\_fill(row + 1, col, ch);

flood\_fill(row, col + 1, ch);

flood\_fill(row - 1, col, ch);

flood\_fill(row, col - 1, ch);

}

**7) Seçerek Sıralama Yönteminin Özyinelemeli Uygulanması**

Dizinin en büyük elemanı bulunup dizinin sonuna atılır. Sonra fonksiyon kendisini 1 eksik uzunlukla çağırır. Örneğin:

#include<stdio.h>

void ssort(int \*pi, size\_t size);

int main(void)

{

int a[10] = { 45, 23, -7, 67, 12, 43, 75, 21, 55, 32 };

int i;

ssort(a, 10);

for (i = 0; i < 10; ++i)

printf("%d ", a[i]);

printf("\n");

return 0;

}

void ssort(int \*pi, size\_tsize)

{

int maxIndex;

size\_t i;

int temp;

if (size == 1)

return;

maxIndex = 0;

for (i = 1; i <size; ++i)

if (pi[i] >pi[maxIndex])

maxIndex = i;

temp = pi[size - 1];

pi[size - 1] = pi[maxIndex];

pi[maxIndex] = temp;

ssort(pi, size - 1);

}

**8) Satranç Tahtasına Birbirini Yemeyen 8 Vezir Yerleştiriniz**

Çeşitli optimizasyonlar yapılabilmekle birlikte çözüm şöyle gerçekleştirilebilir: Fonksiyonda bir tahta alınır. Tahtanın ilk boş yerine Vezir yerleştirilip, fonksiyon kendini çağırır. Böylece fonksiyon her kendini çağırdıkça ilk boşa yere yine vezir yerleştirecektir. Böyle böyle 8 vezire gelindiğinde tahta print edilir. Buradaki sorunlardan biri artık vezir yerleştiremeyince tahtanın durumunun ne olacağıdır. Tahta stack'te yerel ya da parametre değişkeni olarak alınırsa zaten fonksiyon çıkışında eski durumuna gelecektir. Tahta global olarak da alınabilir. Bu durumda özyinelemeli fonksiyon sonlandığında tahtanın yeniden bir önceki durumuna çekilmesi gerekir. Örnek bir çözüm şöyle olabilir:

#include<stdio.h>

#defineSIZE 8

int g\_qcount;

int g\_count;

int g\_board[SIZE][SIZE];

void init\_board(void)

{

int r, c;

for (r = 0; r <SIZE; ++r)

for (c = 0; c <SIZE; ++c)

g\_board[r][c] = '.';

}

void print\_board(void)

{

int r, c;

printf("%d\n", g\_count);

for (r = 0; r <SIZE; ++r) {

for (c = 0; c <SIZE; ++c)

printf("%c", g\_board[r][c]);

printf("\n");

}

printf("\n");

}

void locate\_queen(introw, intcol)

{

int r, c;

g\_board[row][col] = 'V';

r = row;

for (c = col + 1; c <SIZE; ++c)

g\_board[r][c] = 'o';

for (c = col - 1; c >= 0; --c)

g\_board[r][c] = 'o';

c = col;

for (r = row - 1; r >= 0; --r)

g\_board[r][c] = 'o';

for (r = row + 1; r <SIZE; ++r)

g\_board[r][c] = 'o';

for (r = row - 1, c = col - 1; r >= 0 && c >= 0; --r, --c)

g\_board[r][c] = 'o';

for (r = row - 1, c = col + 1; r >= 0 && c <SIZE; --r, ++c)

g\_board[r][c] = 'o';

for (r = row + 1, c = col - 1; r <SIZE&& c >= 0; ++r, --c)

g\_board[r][c] = 'o';

for (r = row + 1, c = col + 1; r <SIZE&& c <SIZE; ++r, ++c)

g\_board[r][c] = 'o';

}

void queen8(introw, intcol)

{

char board[SIZE][SIZE];

int r, c;

for (; row<SIZE; ++row) {

for (; col<SIZE; ++col) {

if (g\_board[row][col] == '.') {

for (r = 0; r <SIZE; ++r)

for (c = 0; c <SIZE; ++c)

board[r][c] = g\_board[r][c];

++g\_qcount;

locate\_queen(row, col);

if (g\_qcount == SIZE) {

++g\_count;

print\_board();

}

queen8(row, col);

--g\_qcount;

for (r = 0; r <SIZE; ++r)

for (c = 0; c <SIZE; ++c)

g\_board[r][c] = board[r][c];

}

}

col = 0;

}

}

int main(void)

{

init\_board();

queen8(0, 0);

return 0;

}

Diğer bazı önemli özyinelemeli fonksiyonlar zaten çeşitli konuların içerisinde ele alınacaktır.

**Fonksiyon Göstericileri (Pointer to Functions)**

Fonksiyonlar da aslında ardışıl makina komutlarından oluşmaktadır. Bir fonksiyonun çağrılması için makina dillerinde CALL isimli makina komutları kullanılır. CALL makina komutu çağrılacak fonksiyonun başlangıç adresini operand alır.i Yani biz bir fonksiyonun başlangıç adresini bilirsek onu çağırabiliriz. İşte fonksiyonların başlangıç adreslerini tutabilen göstericilere fonksiyon göstericileri denilmektedir.

Fonksiyon gösterici bildirimlerinin genel biçimi şöyledir:

[geri dönüş değerinin türü]<(\*<gösterici ismi>)([parametre bildirimi]);

Örneğin:

void (\*pf1)(int);

int (\*pf2)(long, long);

void (\*pf3)(void);

Bir fonksiyon göstericisine her türden fonksiyonun adresini yerleştiremeyiz. Ancak geri dönüş değerinin türü ve parametre türleri belirli biçimde olan fonksiyonların adreslerini yerleştirebiliriz. Yukarıdaki örnekte pf1 göstericisine geri dönüş değeri void, parametresi int olan herhangi bir fonksiyonun adresi yerleştirilebilir. pf2'ye geri dönüş değeri int, parametreleri long, long olan fonksiyonların başlangıç adresleri yerleştirilebilir.

Bildirimdeki parantezler önemlidir. eğer bu parantezler olmasa bu bildirimler fonksiyon prototip bildirimleri haline gelir. Aşağıdaki farka dikkat ediniz:

void (\*pf1)(int); /\* fonksiyon göstericisi tanımlanmış \*/

void \*pf2(int); /\* fonksiyon prototip bildirimi yapılmış \*/

Fonksiyon gösterici bildiriminde parametre parantezi içerisine parametre değişken iismleri yazılabilir. Fakat programcılar genel olarak bunu tercih etmemektedir. Örneğin:

void (\*pf2)(long a, long b); /\* geçerli fakat tercih edilmiyor \*/

C'de bir fonksiyonun yalnızca ismi (yani parantezler olmadan yalnızca ismi) onun başlangıç adresi anlamına gelir. Örneğin:

#include<stdio.h>

void foo(void)

{

printf("foo\n");

}

...

void (\*pf)(void);

pf = foo;

pf bir fonksiyon göstericisi olmak üzere, bu göstericinin gösterdiği yerdeki fonksiyonu çağırmak için iki eşdeğer ifade kullanılabilmektedir: pf(...) ya da (\*pf)(...). Örneğin:

#include<stdio.h>

void foo(void)

{

printf("foo\n");

}

int main(void)

{

void (\*pf)(void);

pf = foo;

pf(); /\* foo çağrılır \*/

(\*pf)(); /\* foo çağrılır \*/

return 0;

}

Fonksiyon gösterici bildiriminde parametre parantezinin içinin boş bıraklımasıyla oraya void yazılması arasında farklılık vardır. Eğer parametre parantezinin içi boş bırakılırsa bu durum o göstericinin parametrik yapısı herhangi bir biçimde olan fakat geri dönüş değeri belli türde olan fonksiyonların adreslerini tutabileceği anlamına gelir. Örneğin:

void (\*pf1)(void);

void (\*pf2)();

Burada pf1'e geri dönüş değeri void, parametresi olmayan bir fonksiyonun adresini yerleştirebiliriz. Halbuki pf2'ye geri dönüş değeri olmayan fakat parametrik yapısı herhangi bir biçimde olan fonksiyonların adreslerini yerleştirebiliriz.

Bir fonksiyonun parametresi bir fonksiyon göstericisi olabilir. Örneğin:

void foo(void (\*pf)(void))

{

...

}

Burada foo fonksiyonu geri dönüş değeri void, parametresi olmayan bir fonksiyonun adresini almaktadır. Örneğin:

#include<stdio.h>

void foo(void (\*pf)(void))

{

pf();

}

void bar(void)

{

printf("bar\n");

}

void tar(void)

{

printf("tar\n");

}

int main(void)

{

foo(bar);

foo(tar);

return 0;

}

Bir fonksiyon gösterici dizisi söz konusu olabilir. Örneğin:

int (\*a[3])(void);

Burada a geri dönüş değeri int, parametresi void olan fonksiyonların adreslerini tutan 3 elemanlı bir dizidir.

Bir fonksiyon göstericisine de ilkdeğer verilebilir. Örneğin:

void (\*pf)(void) = foo;

Örneğin:

int (\*a[3])(void) = {foo, bar, tar};

Örneğin:

#include<stdio.h>

void foo(void)

{

printf("foo\n");

}

void bar(void)

{

printf("bar\n");

}

void tar(void)

{

printf("tar\n");

}

int main(void)

{

void(\*a[3])(void) = { foo, bar, tar };

int i;

for (i = 0; i < 3; ++i)

a[i]();

return 0;

}

Peki bir fonksiyonun geri dönüş değeri bir fonksiyon adresi olabilir mi? Evet! Bu durumda fonksiyon isminin soluna '\*' atomu koyup parantezlemek gerekir. Sonra bu parantezlerin soluna geri dönüş değerine ilişkin fonksiyonun geri dönüş değeri, sağında da geri dönüş değerine ilişkin fonksiyonun parametre bildirimi yerleştirilir. Örneğin:

long (\*foo(int a))(double)

{

...

}

Burada foo'nun kendi parametresi int türdendir. foo öyle bir fonksiyon adresine geri dönmektedir ki, onun geri dönüş değeri long, parametresi double türdendir. Bu bildirimin aşama aşama nasıl oluşturulduğunu gösterelim:

foo(int a)

(\*foo(int a))

long (\*foo(int a))

long (\*foo(int a))(double)

Örneğin:

#include<stdio.h>

void foo(inta)

{

printf("foo: %d\n", a);

}

void (\*bar(void))(int)

{

return foo;

}

int main(void)

{

void(\*pf)(int);

pf = bar();

pf(100);

return 0;

}

Bildirimler daha karmaşık olabilir. Örneğin, "öyle bir fonksiyon yazalım ki fonksiyonumuzun kendi parametresi int türden olsun fakat geri dönüş değeri bir fonksiyon adresi olsun. Ama öyle bir fonksiyonun adresi olsun ki, onun parametresi long, onun geri dönüş değeri de geri dönüş değeri double parametresi double olan bir fonksiyon adresi olsun".

foo(int a)

(\*foo(int a))

(\*foo(int a))(long)

(\*(\*foo(int a))(long))

(\*(\*foo(int a))(long))

(\*(\*foo(int a))(long))(double)

double (\*(\*foo(int a))(long))(double)

Bu fonksiyonu kısaca şöyle ifade edebiliriz: "foo parametresi int, geri dönüş değeri, parametresi long, geri dönüş değeri parametresi double, geri dönüş değeri double türden olan bir fonksiyon adresidir".

Örneğin:

#include<stdio.h>

double foo(doublea)

{

printf("foo: %d\n", a);

return 0;

}

double (\*bar(longa))(double)

{

return foo;

}

double(\*(\*tar(inta))(long))(double)

{

return bar;

}

int main(void)

{

double(\*(\*pf1)(long))(double);

double(\*pf2)(double);

pf1 = tar(0);

pf2 = pf1(0);

pf2(0);

return 0;

}

main'deki fonksiyon çağrısı şöyle de yapılabilirdi:

int main(void)

{

tar(0)(0)(0);

return 0;

}

Bildirimler daha karışık da olabilir. Fakat uygulamada böylesi karmaşık bildirimlerle karşılaşılmamaktadır.

**Fonksiyon Göstericilerine Neden Gereksinim Duyulmaktadır?**

Fonksiyon göstericileri çok çeşitli gerekçelerle kullanılabilmektedir. Ancak en yaygın kullanım biçimi "genelleştirme ve kodu devretme" amaçlı kullnımdır. Örneğin bir fonksiyon bizim için birşeyleri buluyor olabilir. Fakat onu bulunca ne yapacağını bize bırakmak isteyebilir. İşte fonksiyon bizden bir fonksiyonun adresini alır, onu bulunca o fonksiyonu çağırır. O fonksiyonun içini de biz yazacağımız için istediğimiz şeyi yapmış oluruz. Bir olay gerçekleştiğinde bizden alınan bir fonksiyonun çağrılması durumunda bu tür fonksiyonlara İngilizce "Callback Function" denilmektedir. Örneğin:

#include<stdio.h>

void for\_each(int \*pi, intsize, void (\*pf)(int \*))

{

int i;

for (i = 0; i <size; ++i)

pf(&pi[i]);

}

void foo(int \*pi)

{

\*pi = \*pi \* \*pi;

}

void bar(int \*pi)

{

printf("%d\n", \*pi);

}

int main(void)

{

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

for\_each(a, 10, foo);

for\_each(a, 10, bar);

return 0;

}

Buradaki for\_each fonksiyonu diziyi dolaşmakta ve her eleman için onun adresini vererek bir fonksiyonu çağırmaktadır.

Örneğin:

#include<stdio.h>

#include<Windows.h>

void do\_periodically(intperiod, void(\*pf)(void))

{

for (;;) {

Sleep(period);

pf();

}

}

void foo(void)

{

putchar('.');

}

int main(void)

{

do\_periodically(1000, foo);

return 0;

}

Örneğin bir komut yorumlayıcının çatısını fonksiyon göstericilerini kullanarak kolay bir biçimde yazabiliriz:

#include<stdio.h>

#include<string.h>

/\* Symbolic Constants \*/

#defineMAX\_CMD\_LINE 1024

#defineMAX\_PARAMS 32

/\* Type Declarations \*/

typedefstructtagCMD{

constchar \*cmdText;

void(\*proc)(void);

} CMD;

/\* Function Prototypes \*/

void parse\_cmdline(void);

void proc\_dir(void);

void proc\_del(void);

/\* Global Object Definitions \*/

char g\_cmdLine[MAX\_CMD\_LINE];

CMD g\_cmds[] = {

{"dir", proc\_dir},

{ "del", proc\_del },

{NULL, NULL}

};

char \*g\_params[MAX\_PARAMS];

int g\_nparams;

/\* Function Definitions \*/

int main(void)

{

int i;

for (;;) {

printf("CSD>");

gets(g\_cmdLine);

parse\_cmdline();

if (g\_nparams == 0)

continue;

for (i = 0; g\_cmds[i].cmdText != NULL; ++i)

if (!strcmp(g\_params[0], g\_cmds[i].cmdText)) {

g\_cmds[i].proc();

break;

}

if (g\_cmds[i].cmdText == NULL)

{

printf("command not found!..\n");

}

}

return 0;

}

void parse\_cmdline(void)

{

char \*str;

g\_nparams = 0;

for (str = strtok(g\_cmdLine, " \t"); str != NULL; str = strtok(NULL, " \t"))

g\_params[g\_nparams++] = str;

}

void proc\_dir(void)

{

printf("dir command...\n");

}

void proc\_del(void)

{

if (g\_nparams != 2) {

printf("the syntax of the command is incorrect.\n");

return;

}

printf("file deleted...\n");

}

**Fonksiyon Göstericileri ve NULL Adres**

NULL adres bir fonksiyon göstericisine atanabilir. Bu durumda fonksiyon göstericisinin içerisinde NULL adres bulunduğu söylenir. Örneğin:

void (\*pf)(void) = NULL;

...

if (pf == NULL) {

...

}

**Fonksiyon Adresleri ve void Göstericiler**

C'de (ve tabi C++'ta) void göstericiler data göstericisi olarak düşünülmüştür. Biz void bir göstericiye herhangi bir türden nesnenin adresini atayabiliriz fakat bir fonksiyonun adresini atayamayız. Örneğin:

void foo(void)

{

...

}

...

void \*pv;

pv = foo; /\* geçerli değil \*/

Tür dönüştürmesi yapılsa bile bu durum geçerli olmaz.

**Fonksiyon Göstericilerine Tür Dönüştürmesi Yapmak**

Bir fonksiyon göstericisine uyumlu olmayan bir fonksiyonun adresi doğrudan atanamaz. Fakat tür dönüştürmesi ile atanabilir. Örneğin:

int foo(int a)

{

...

}

...

void (\*pf)(void);

pf = foo; /\* geçersiz! Tür uyuşmazlığı var \*/

pf = (void (\*)(void)) foo; /\* geçerli \*/

Burada fonksiyon adreslerinin sembolik gösteriminde parantez içerisinde yalnızca \* bulunduğuna dikkat ediniz. Örneğin C'de int (\*)(long) türü, geri dönüş değeri int parametresi long olan bir fonksiyon adres türünü temsil eder. Yukarıdaki dönüştürme typedef bildirimi ile sadeleştirilebilir. Örneğin:

int foo(int a)

{

...

}

...

typedef void (\*PF)(void);

PF pf;

pf = (PF) foo; /\* geçerli \*/

C'de void fonksiyon göstericisi olmadığına göre bir fonksiyonun adresini geçici süre bir göstericide saklayacaksak bu herhangi bir tür fonksiyon göstericisi olabilir. Örneğin:

int foo(int a)

{

...

}

...

typedef void (\*PF)(void);

PF pfv;

pfv = (PF) foo;

...

int (\*pfi)(int);

pfi = (int (\*)(int))pfv;

**typedef Bildirimleri**

C'de typedef anahtar sözcüğü "storage class specifier" olarak gramere yerleştirilmiştir. Her türlü bildirime typedef belirleyicisi getirilebilir. Bir bildirime typedef belirleyicisini eklersek artık o bildirimdeki değişken ismi, o değişkenin türüne ilişkin tür ismi haline gelir. Örneğin:

int I;

I burada int türden bir değişken ismidir. Fakat:

typedef int I;

Burada I artık int türünü temsil eden bir isim haline gelmiştir. typedef bildirimleri de yerel ya da global düzeyde yapılabilir. Örneğin:

int A, B, C;

Burada A, B, C int türden değişkenlerdir. Fakat:

typedef int A, B, C;

Burada hem A, hem B, hem C int türünü temsil etmektedir. Örneğin:

int \*PI;

Burada PI int \* türündendir. Yani int türden bir adres bilgisini tutan değişkendir. Fakat:

typedef int \*PI;

Burada PI int \* türünü temsil etmektedir. Yani:

int \*pi;

ile

PI pi;

aynı anlamdadır.

Mademki C'de belirleyiciler bildirimde herhangi bir sırada yazılabiliyor o halde typedef anahtar sözcüğünün başa getirilmesi de zorunlu değildir. Örneğin:

int typedef I;

Örneğin:

int ARY[10];

Burada ARY int[10] (10 elemanlı int türden dizi türünden) türündendir. Fakat:

typedef int ARY[10];

Burada ARY int[10] türü anlamına gelmektedir. Yani:

int a[10];

ile,

ARY a;

aynı anlamdadır. Örneğin:

struct tagCMD {

...

} CMD;

Burada CMD struct tagCMD türündendir. Fakat:

typedef struct tagCMD {

...

} CMD;

Burada CMD struct tagCMD türünü temsil etmektedir.

typedef özellikle karmaşık bildirimleri daha sade ifade etmek için kullanılabilmektedir. Örneğin:

void (\*PF)(void);

PF burada geri dönüş değeri void parametresi void olan bir fonksiyon gösetricisidir. Fonksiyon göstericilerinin türü sembolik olarak belirtilirken isim kullanılmaz. Örneğin burada PF, void (\*)(void) türündendir. Fakat:

typedef void (\*PF)(void);

Burada PF void (\*)(void) türünü temsil eder. Yani:

void (\*pf)(void);

ile

PF pf;

aynı anlamdadır. Örneğin:

double (\*foo(int a))(long)

{

...

}

Bu bildirimi typedef ile daha sade yazabiliriz:

typedef double (\*PF)(long);

PF foo(int a)

{

...

}

Örneğin:

typedef double (\*PF)(long);

typedef PF (\*PFF)(int);

PFF foo(int a)

{

...

}

Burada foo'yu typedef'isz olarak bildirmek isteyelim:

double (\*(\*foo(int a))(int))(long)

{

...

}

Fonksiyon prototipine typedef uygulanabilir. Örneğin:

typedef void F(double);

Burada F geri dönüş değeri void, parametresi double olan bir fonksiyon prototipini temsil etmektedir.

F foo, bar;

Bu bildirimin eşdeğeri:

void foo(double);

void bar(double);

**Çok Boyutlu Diziler ve Dizi Göstericileri**

C'de bir dizinin ismi dizinin taamamını belirten bir nesnedir. Örneğin:

int a[8];

Burada a bu dizinin tamamnını temsil etmektedir. Ancak C standartlarına göre biz bir dizi ismini bir ifadede kullandığımızda dizi isimleri otomatik olarak o dizinin başlangıç adresine dönüştürülmektedir. Yani dizilerin isimleri aslında dizinin tamamını belirtmekle birlikte işleme soktuğumuzda onların başlangıç adresini belirtiyor durumda olur.

C'de biz çok boyutlu bir dizinin adresini hangi türden bir göstericiye yerleştirebiliriz? Örneğin:

int a[3][6];

Burada a'nın adresini nasıl göstericiye yerleştirebiliriz. İşte bu matrisin adresi aşağıdaki gibi bildirilen bir göstericiye yerleştirilebilir:

int (\*p)[6];

Bu tür göstericilere dizi göstericileri (pointer to array) denilmektedir. Böyle göstericilerde dizi kaç boyutluysa ilk boyut dışındakilerin hepsinin köşeli parantezlerle uzunluğunun belirtilmesi gerekir. Köşeli parantezlerin içi boş bırakılamaz. Örneğin:

int a[2][3][4][5];

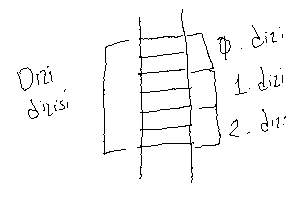
Böyle bir çok boyutlu dizinin başlangıç adresini tutabilecek gösterici şöyle tanımlanabilir:

int (\*p)[3][4][5];

Matrisler aslında dizi dizileridir. Yani dizilerden oluşmuş dizilerdir. Örneğin:

int a[3][2];

Burada aslında her biri 2 elemandan oluşan 3 elemanlı bir dizi dizisi söz konusudur:



Bir dizi göstericisine \* operatörüyle erişirsek o nesne belirtmez, dizinin tamamını belirtir. (Tıpkı dizi isimlerinde olduğu gibi). Örneğin:

int (\*p)[2];

Burada p göstericisinin gösterdiği yerde int bir bilgi yoktur, int türden bir dizi vardır. p göstericisinin türü int[2] biçimindedir. \*p bir nesne belirtmez. \*p ifadesini sanki bir dizi ismini kullanıyormuşuz gibi düşünebiliriz.

C'de bir dizinin de adresi alınabilir. Dizi adresleri dizi göstericilerine atanır. Örneğin:

int a[2];

int \*pi;

int (\*pa)[2];

pi = a; /\* geçerli \*/

pi = &a; /\* geçerli değil! \*/

pa = &a; /\* geçerli \*/

\*pa = 10; /\* geçersiz, sanki a = 10 gibi \*/

\*pa + 1 /\* bu ifade geçerlidir, \*pa ile belirtilen dizinin sonraki elemanının adresi elde edilir \*/

\*\*pa = 10; /\* geçerli, pa göstericisinin gösterdiği yerdeki dizinin ilk elemanına atanıyor \*/

(\*p)[2] = 10; /\* geçerli \*/

Örneğin:

int a[3][2];

Burada mademki bir dizinin ismini işleme soktuğumuzda artık o, o dizinin ilk elemanının adresi anlamına geliyor, o halde a ifadesi int[2] türünden bir dizinin adresi anlamına gelir. Bu da dizi göstericisine yerleştirilebilir:

int a[3][2];

int (\*pa)[2];

pa = a; /\* geçerli \*/

Bir dizi göstericisini bir artırdığımızda içindeki adres gösterdiği dizinin uzunluğu kadar artar. Örneğin:

int a[3][2];

int (\*pa)[2];

pa = a; /\* Burada pa dizi dizisinin 0'ınci indeksli dizisini göstermektedir \*/

++pa; /\* Burada pa dizi dizisinin 1'inci indeksli dizisini göstermektedir \*/

Örneğin:

#include<stdio.h>

int main(void)

{

int a[4][3] = {

{ 1, 2, 3 }, { 4, 5, 6 }, { 7, 8, 9 }, {10, 11, 12}

};

int (\*pa)[3];

int val;

int \*pi;

pa = a;

val = \*\*pa;

printf("%d\n", val); /\* 1 \*/

val = (\*pa)[1];

printf("%d\n", val); /\* 2 \*/

val = pa[1][1]; /\* eşdeğeri (\*(pa + 2))[1] \*/

printf("%d\n", val); /\* 5 \*/

pi = pa[1];

val = pi[2];

printf("%d\n", val); /\* 6 \*/

pa += 2;

val = (\*pa)[1];

printf("%d\n", val); /\* 8 \*/

return 0;

}

Bir matrisin elemanlarına neden iki köşeli parantezle erişildiği kolaylıkla anlaşılabilir. Örneğin:

int a[3][2];

int val;

val = a[2][1];

Burada a[2] ifadesi a dizisinin 2'inci indisli elemanı anlamına gelir. O da bir dizidir. O halde a[2] aslında bir dizi ismi gibidir ve 2'inci indisli dizinin başlangıç adresini belirtir. Biz oradan 1 ileriye gidersek toplamda matrisin 2'inci indisli satırının 1'inci indisli elemanına erişmiş oluruz.

Örneğin:

#include<stdio.h>

void foo(int(\*pa)[3])

{

int i;

for (i = 0; i < 3; ++i)

printf("%d ", (\*pa)[i]);

printf("\n");

}

int main(void)

{

int a[3] = { 1, 2, 3 };

foo(&a); /\* geçerli \*/

return 0;

}

**Proseslerin Çalışma Dizinleri (Current Working Directory)**

Bir dosyanın yerini belirten yazısal ifadeye yol ifadesi (path name) denilmektedir. Yol ifadeleri mutlak (absolute) ya da göreli (relative) olabilir. Mutlak yol ifadesi kök dizinden itibaren yer belirtirken, göreli yol ifadesi prosesin çalışma dizininden itibaren yer belirtir. Her prosesin bir çalışma dizini vardır. Proseslerin çalışma dizinleri Proses Kontrol Bloğunda saklanmaktadır.

Bir yol ifadesinin ilk karakteri Windows'ta '\', UNIX/Linux sistemlerinde '/' ise böyle ifadeleri mutlaktır, değilse görelidir. Örneğin:

"a.dat" /\* Windows, göreli \*/

"\a.dat" /\* Windows, mutlak \*/

"a\b\c.dat" /\* Windows, göreli \*/

"\a\b\c.dat" /\* Windows, mutlak \*/

"a.dat" /\* UNIX/Linux, göreli \*/

"/a.dat" /\* UNIX/Linux, mutlak \*/

"a/b/c.dat" /\* UNIX/Linux, göreli \*/

"/a/b/c.dat" /\* UNIX/Linux, mutlak \*/

Windows UNIX/Linux uyumunu korumak için '/' karakterini de geçişlerde kabul etmektedir.

Windows sistemlerinde ayrıca sürücü (drive) kavramı da vardır. Her sürücünün asyrık kökü bulunur. UNIX/Linux sistemlerinde sürücü yoktur. Dolayısıyla tek bir kök vardır.

Peki Windows'ta mutlak ya da göreli yol ifadesi hangi sürücüye ilişkindir?

Windows'ta sürücü de içeren yol ifadelerine tam yol ifadesi (full path name) denilmektedir. Sürücü bir harf ve ':' karakterinden oluşmaktadır. Örneğin:

"c:\a\b\c.dat" /\* tam yol ifadesi \*/

"e:\a\b\c.dat" /\* tam yol ifadesi \*/

Windows'ta prosesin çalışma dizini Proses Kontrol Bloğunda tam yol ifadesi biçiminde tutulur. İşte eğer biz mutlak bir yol ifadesinde sürücü kullanmamışsak, işletim sistemi prosesin çalışma dizini hangi sürücüdeyse o mutlak yol ifadesinin o sürücüye ilişkin olduğunu düşünmektedir. Örneğin, prosesimizin çalışma dizini "e:\temp" olsun. Biz de "\a\b\c.dat" biçiminde bir mutlak yol ifadesi vermiş olalım. Buada kök e sürücüsünün köküdür.

Windows'ta ilginç bir durum daha vardır. Göreli bir yol ifadesi bir sürücü içerebilir. Örneğin prosesimizin çalışma dizini "d:\temp" olsun:

"c:a\b\c.dat"

"e:c.dat"

Buradaki göreli yol ifadeleri nereden itibaren yer belirtmektedir? İşte Windows burada bazı çevre değişkenlerine başvurur. Bu çevre değişkenleri yoksa Windows yine bu sürücülerin kök dizinlerinden itibaren yolu belirlemektedir. Yani bu çevre değişkenleri tanımlanmamışsa (pek çok sistemde tanımnlanmamıştır) yukarıdaki yol ifadeleri aşağıdakilerle eşdeğer olur:

"c:\a\b\c.dat"

"e:\c.dat"

Proseslerin çalışma dizinleri Windows'ta GetCurrentDirectory isimli API fonksiyonuyla elde edilebilir:

DWORD GetCurrentDirectory(

DWORD nBufferLength,

LPTSTR lpBuffer

);

Fonksiyonun birinci parametresi çalışma dizininin yerleştirileceği dizinin uzunluğunu, ikinci parametre bunun adresini alır. Fonksiyon prosesin çalışma dizinini bu diziye yerleştirp sonuna null karakter koyar. Fonksiyon normal olarak diziye yerleştirdiği karakter sayısıyla (null karakter dahil değil) geri dönmektedir. Eğer uzunluk yetersizse fonksiyon kırparak onu yerleştirir. Bu durumda fonksiyon yol ifadesinin yerleştirileceği dizinin uzunluğunu (null karakterle birlikte) bize geri dönüş değeri olarak verir. Örneğin:

#include<stdio.h>

#include<Windows.h>

int main(void)

{

char cwd[1024];

GetCurrentDirectory(1024, cwd);

printf("%s\n", cwd);

return 0;

}

UNIX/Linux sistemlerinde prosesin çalışma dizini getcwd isimli POSIX fonksiyonuyla elde edilir:

#include <unistd.h>

char \*getcwd(char \*buf, size\_t size);

Fonksiyonun birinci parametresi yol ifadesinin yerleştirileceği dizinin adresini, ikinci parametresi bunun uzunluğunu alır. Fonksiyon birinci parametreyle verilen adresin aynısına geri döner. Eğer ikinci parametreyle belirtilen uzunluk null karakter dahil yol ifadesinin uzunluğundan küçükse fonksiyon başarısız olur ve NULL adresine geri döner. Örneğin:

#include<stdio.h>

#include<stdlib.h>

#include<unistd.h>

int main(void)

{

char cwd[1024];

if (getcwd(cwd, 1024) == NULL) {

perror("getcwd");

exit(EXIT\_FAILURE);

}

puts(cwd);

return 0;

}

Windows sistemlerinde ayrıca \_getcwd isimli POSIX fonksiyonuyla aynı biçimde çalışan bir kütüphane fonksiyonu vardır. Tabi fonksiyon da kendi içerisinde GetCurrentDirectory API fonksiyonunu çağırmaktadır.

Prosesin çalışma dizini Windows'ta SetCurrentDirectory API fonksiyonuyla değiştirilebilir:

BOOL SetCurrentDirectory(LPCTSTR lpPathName);

Fonksiyon yeni çalışma dizininin bulunduğu dizinin adresini parametre olarak alır. Geri dönüş değeri başarı durumunu belirtmektedir. Örneğin:

#include<stdio.h>

#include<Windows.h>

void ExitSys(LPCSTR lpszMsg, int status);

int main(void)

{

char cwd[1024];

if (!SetCurrentDirectory("c:\\windows"))

ExitSys("SetCurrentDirectory", EXIT\_FAILURE);

GetCurrentDirectory(1024, cwd);

printf("%s\n", cwd);

return 0;

}

void ExitSys(LPCSTRlpszMsg, intstatus)

{

DWORD dwLastError = GetLastError();

LPTSTR lpszErr;

if (FormatMessage(FORMAT\_MESSAGE\_ALLOCATE\_BUFFER | FORMAT\_MESSAGE\_FROM\_SYSTEM, NULL, dwLastError,

MAKELANGID(LANG\_NEUTRAL, SUBLANG\_DEFAULT), (LPTSTR)&lpszErr, 0, NULL)) {

fprintf(stderr, "%s: %s", lpszMsg, lpszErr);

LocalFree(lpszErr);

}

exit(status);

}

UNIX/Linux sistemlerinde prosesin çalışma dizini chdir isimli POSIX fonksiyonuyla değiştirilir:

#include <unistd.h>

int chdir(const char \*path);

Fonksiyonun parametresi çalışma dizininin yerleştirildiği dizinin adresini alır. Fonksiyon başarı durumunda sıfır, başarısızlık durumunda -1 değerine geri döner.

#include<stdio.h>

#include<stdlib.h>

#include<unistd.h>

int main(void)

{

char cwd[1024];

if (chdir("/usr/include") == -1) {

perror("chdir");

exit(EXIT\_FAILURE);

}

if (getcwd(cwd, 1024) == NULL) {

perror("getcwd");

exit(EXIT\_FAILURE);

}

puts(cwd);

return 0;

}

Komut satırı programları (bash gibi, cmd.exe gibi) aslında prosesin çalışma dizinini elde edip prompt'a yazdırmaktadır. Dolayısıyla bunlar da birer prosestir ve aslında bir dizine oturmak gibi bir kavram yoktur. Kullanıcı komutu yazdığında komut satırı programları doğrudan kullanıcının yazdığı yol ifadesini kullanır. Bu yol ifadeleri de mutlak ya da göreli olabilir. Eğer bu yol ifadeleri göreliyse prompt'ta belirtilen çalışma dizininden itibaren yer belirtir. Örneğin:

#include<stdio.h>

#include<string.h>

#include<Windows.h>

/\* Symbolic Constants \*/

#defineMAX\_CMD\_LINE 1024

#defineMAX\_PARAMS 32

/\* Type Declarations \*/

typedefstructtagCMD{

constchar \*cmdText;

void(\*proc)(void);

} CMD;

/\* Function Prototypes \*/

void print\_syserr(constchar \*msg);

void parse\_cmdline(void);

void proc\_cls(void);

void proc\_exit(void);

void proc\_cd(void);

/\* Global Object Definitions \*/

char g\_cmdLine[MAX\_CMD\_LINE];

CMD g\_cmds[] = {

{ "cls", proc\_cls },

{ "exit", proc\_exit },

{ "cd", proc\_cd },

{ NULL, NULL }

};

char g\_cwd[MAX\_PATH];

char \*g\_params[MAX\_PARAMS];

int g\_nparams;

/\* Function Definitions \*/

int main(void)

{

int i;

for (;;) {

GetCurrentDirectory(MAX\_PATH, g\_cwd);

printf("%s>", g\_cwd);

gets(g\_cmdLine);

parse\_cmdline();

if (g\_nparams == 0)

continue;

for (i = 0; g\_cmds[i].cmdText != NULL; ++i)

if (!strcmp(g\_params[0], g\_cmds[i].cmdText)) {

g\_cmds[i].proc();

break;

}

if (g\_cmds[i].cmdText == NULL)

printf("command not found!..\n");

}

return 0;

}

void parse\_cmdline(void)

{

char \*str;

g\_nparams = 0;

for (str = strtok(g\_cmdLine, " \t"); str != NULL; str = strtok(NULL, " \t"))

g\_params[g\_nparams++] = str;

}

void proc\_cd(void)

{

if (g\_nparams == 1) {

puts(g\_cwd);

return;

}

if (g\_nparams > 2) {

printf("too many argumnets!..\n");

return;

}

if (!SetCurrentDirectory(g\_params[1]))

print\_syserr("cd");

}

void proc\_cls(void)

{

system("cls");

}

void proc\_exit(void)

{

exit(EXIT\_SUCCESS);

}

void print\_syserr(constchar \*msg)

{

DWORD dwLastError = GetLastError();

LPTSTR lpszErr;

if (FormatMessage(FORMAT\_MESSAGE\_ALLOCATE\_BUFFER | FORMAT\_MESSAGE\_FROM\_SYSTEM, NULL, dwLastError,

MAKELANGID(LANG\_NEUTRAL, SUBLANG\_DEFAULT), (LPTSTR)&lpszErr, 0, NULL)) {

fprintf(stderr, "%s: %s", msg, lpszErr);

LocalFree(lpszErr);

}

}

**Nokta ve Nokta Nokta Dizinleri**

Hem Windows'ta hem de UNIX/Linux sistemlerinde özel iki dizin vardır. Bunlar "." ve ".." dizinleridir. Nokta dizini mutlak yol ifadesi olarak prosesin çalışma dizinini, nokta nokta yol ifadesi çalışma dizininin üst dizinini belirtir. Örneğin "sample.dat" yol ifadesiyle ".\sample.dat" yol ifadesi aynı anlamdadır. Örneğin, "..\..\sample.dat" yol ifadesi bulunulan dizinin iki üset dizinindeki "sample.dat" dosyası anlamına gelir.

**Anahtar Notlar:** Windows'ta ve UNIX/Linux sistemlerinde dizin geçişlerinde birden fazla '\' ya '/' karakteri kullanılabilir. Yani örneğin "c:\temp\ad.at" yol ifadesi ile "c:\temp\\\\\\\a.dat" yol ifadesi eşdeğerdir.

**Proses Çalışmaya Başladığında (Deafulrt) Çalışma Dizini Neresidir?**

UNIX/Linux sistemlerinde prosesin çalışma dizini üst prosesten aktarılmaktadır. Yani bu sistemlerde bir programı başka bir program çalıştırır. Çalıştıran programa üst proses (parent process), çalıştırılan programa alt proses (child process) denilmektedir. Örneğin Linux'ta komut satırından bir program çalıştırdığımızda onun çalışma dizini komut satırı programıın (tipik olarak bash) çalışma dizini olacaktır. Tabi komut satırı programının çalışma dizinini de biz cd komutuyla değiştirebilmekteyiz.

Windows sistemlerinde durum benzerdir. Prosesi çalıştırmak için kullanılan CreateProcess API fonksiyonunun bir parametresinde yaratılacak çalışma dizininin yol ifadesi verilir. Eğer bu verilmezse (yani NULL geçilirse) bu durumda üst prosesten bu bilgi alınır. Örneğin Visual Studio'da CTRL+F5 yaptığınızda programı Visual Studio çalıştırır. Visual Studio C/C++ projelerinde çalıştırdığı prıgramın çalışma dizinini CreateProcess sırasında proje dizini olarak ayarlamaktadır. Yani biz Visual Studio'da programı çalıştırırken programımız default çalışma dizini proje dizini olmaktadır.

**Dizin Ağacının Dolaşılması**

Dizin dolaşımı özyinelemeli bir fonksiyonla yapılabilir. Genel tasarım şöyledir: Fonksiyonun bir parametresi vardır o da içi listelenecek dizindir. Fonksiyon o dizinin içerisine geçer (yani çalışma dizinini o dizin olarak ayarlar) ve dosyaları listelemeye başlar. Bir dizin gördüğünde onu argüman yaparak fonksiyon kendisini çağırır tabi çıkarken de bir üst dizine geri dönmelidir.

**Anahtar Notlar**: Windows "Windows Explorer"da dosya ve dizinleri görüntülerken doğal sırada görüntülememektedir. Yani FindFirstFile ve FindNextFile fonksiyonlarıyla elde ettiğimiz sıra ile Windows'un bize "Windows Explorer"da gösterdiği sıra aynı değildir.

Windows'ta dizin dolaşma işlemi şöyle yapılabilir:

#include<stdio.h>

#include<string.h>

#include<Windows.h>

void WalkDir(constchar \*path)

{

HANDLE hFF;

WIN32\_FIND\_DATA finfo;

if (!SetCurrentDirectory(path))

return;

hFF = FindFirstFile("\*.\* ", &finfo);

if (hFF == INVALID\_HANDLE\_VALUE)

return;

do {

if (!strcmp(finfo.cFileName, ".") || !strcmp(finfo.cFileName, ".."))

continue;

printf("%s\n", finfo.cFileName);

if (finfo.dwFileAttributes &FILE\_ATTRIBUTE\_DIRECTORY) {

WalkDir(finfo.cFileName);

SetCurrentDirectory("..");

}

} while (FindNextFile(hFF, &finfo));

FindClose(hFF);

}

int main(void)

{

WalkDir("c:\\");

return 0;

}

Windows'ta bizim bazı dizinlere yetki sorunu yüzünden erişme hakkımız yoktur. Yani o dizinlere geçemeyiz ve onun listesini alamayız. Yukarıdaki program bu durumlarda ekrana hata mesajı çıkartmamaktadır. Eğer istenirse bu mesaj aşağıdaki gibi yazdırılabilir:

#include<stdio.h>

#include<string.h>

#include<Windows.h>

void PutSysError(constchar \*msg);

void WalkDir(constchar \*path)

{

HANDLE hFF;

WIN32\_FIND\_DATA finfo;

if (!SetCurrentDirectory(path)) {

PutSysError("SetCurrentDirectory");

return;

}

hFF = FindFirstFile("\*.\*", &finfo);

if (hFF == INVALID\_HANDLE\_VALUE)

return;

do {

if (!strcmp(finfo.cFileName, ".") || !strcmp(finfo.cFileName, ".."))

continue;

printf("%s\n", finfo.cFileName);

if (finfo.dwFileAttributes &FILE\_ATTRIBUTE\_DIRECTORY) {

WalkDir(finfo.cFileName);

SetCurrentDirectory("..");

}

} while (FindNextFile(hFF, &finfo));

FindClose(hFF);

}

void PutSysError(constchar \*msg)

{

DWORD dwLastError = GetLastError();

LPTSTR lpszErr;

if (FormatMessage(FORMAT\_MESSAGE\_ALLOCATE\_BUFFER | FORMAT\_MESSAGE\_FROM\_SYSTEM, NULL, dwLastError,

MAKELANGID(LANG\_NEUTRAL, SUBLANG\_DEFAULT), (LPTSTR)&lpszErr, 0, NULL)) {

fprintf(stderr, "%s: %s", msg, lpszErr);

LocalFree(lpszErr);

}

}

int main(void)

{

WalkDir("c:\\");

return 0;

}

Yukarıdaki programların küçük bir sorunu vardır. WalkDir burada prosesin çalışma dizinini değiştirip onu öyle bırakmaktadır. Bunu engellemek için bir sarma fonksiyon yazılabilir:

#include<stdio.h>

#include<string.h>

#include<Windows.h>

void WalkDirRecur(constchar \*path)

{

HANDLE hFF;

WIN32\_FIND\_DATA finfo;

if (!SetCurrentDirectory(path))

return;

hFF = FindFirstFile("\*.\*", &finfo);

if (hFF == INVALID\_HANDLE\_VALUE)

return;

do {

if (!strcmp(finfo.cFileName, ".") || !strcmp(finfo.cFileName, ".."))

continue;

printf("%s\n", finfo.cFileName);

if (finfo.dwFileAttributes &FILE\_ATTRIBUTE\_DIRECTORY) {

WalkDirRecur(finfo.cFileName);

SetCurrentDirectory("..");

}

} while (FindNextFile(hFF, &finfo));

FindClose(hFF);

}

void WalkDir(constchar \*path)

{

char cwd[MAX\_PATH];

GetCurrentDirectory(MAX\_PATH, cwd);

WalkDirRecur(path);

if (!SetCurrentDirectory(cwd)) {

fprintf(stderr, "Cannot set directory!..\n");

exit(EXIT\_FAILURE);

}

}

int main(void)

{

WalkDir("c:\\");

return 0;

}

Anahtar Notlar: printf fonksiyonunda alan belirten değer de argüman olarak girilebilir. Bunun için %\* sentaksı kullanılır. Örneğin "%\*d" format karakterleri için iki argüman girilmelidir. Birincisi \* için uzunluk belirten argüman, ikincisi d için değerin bizzat kendisi. Örneğin:

char buf[] ="ankara";

int n;

scanf("%d", &n);

printf("%-\*sxxx\n", n, buf);

Dizin ağacını tab'larla kademeli olarak da yazdırabiliriz. Örneğin:

#include<stdio.h>

#include<string.h>

#include<Windows.h>

#defineTABSIZE 4

void PutSpace(int count);

void WalkDirRecur(constchar \*path, inttabCount)

{

HANDLE hFF;

WIN32\_FIND\_DATA finfo;

if (!SetCurrentDirectory(path))

return;

hFF = FindFirstFile("\*.\*", &finfo);

if (hFF == INVALID\_HANDLE\_VALUE)

return;

do {

if (!strcmp(finfo.cFileName, ".") || !strcmp(finfo.cFileName, ".."))

continue;

PutSpace(tabCount \* TABSIZE);

if (finfo.dwFileAttributes &FILE\_ATTRIBUTE\_DIRECTORY) {

printf("%s <DIR>\n", finfo.cFileName);

WalkDirRecur(finfo.cFileName, tabCount + 1);

SetCurrentDirectory("..");

}

else

printf("%s\n", finfo.cFileName);

} while (FindNextFile(hFF, &finfo));

FindClose(hFF);

}

void WalkDir(constchar \*path)

{

char cwd[MAX\_PATH];

GetCurrentDirectory(MAX\_PATH, cwd);

WalkDirRecur(path, 0);

if (!SetCurrentDirectory(cwd)) {

fprintf(stderr, "Cannot set directory!..\n");

exit(EXIT\_FAILURE);

}

}

void PutSpace(intcount)

{

while (count-- > 0)

putchar(' ');

}

int main(void)

{

WalkDir("e:\\DropBox\\Kurslar\\SysProg-2015");

return 0;

}

Dizin ağacını dolaşan fonksiyon fonksiyon göstericileriyle genelleştirilebilir. Şöyle ki: WalkDir fonksiyonu bizden ayrıca bir sonksiyon adresi alır, dizin ağacında her dosyayı buldukça o fonksiyonu çağırır. Böylece biz o fonksiyonun içerisinde ister dosyayı yazdırır istersek başka birşey yaparız. Böyle bir fonksiyon şöyle yazılabilir:

#include<stdio.h>

#include<string.h>

#include<Windows.h>

BOOL WalkDirRecur(constchar \*path, BOOL(\*Proc)(constWIN32\_FIND\_DATA \*, int), intlevel)

{

HANDLE hFF;

WIN32\_FIND\_DATA finfo;

BOOL bResult;

if (!SetCurrentDirectory(path))

return;

hFF = FindFirstFile("\*.\*", &finfo);

if (hFF == INVALID\_HANDLE\_VALUE)

returnTRUE;

bResult = TRUE;

do {

if (!strcmp(finfo.cFileName, ".") || !strcmp(finfo.cFileName, ".."))

continue;

if (!Proc(&finfo, level)) {

bResult = FALSE;

break;

}

if (finfo.dwFileAttributes &FILE\_ATTRIBUTE\_DIRECTORY) {

if (!WalkDirRecur(finfo.cFileName, Proc, level + 1)) {

bResult = FALSE;

break;

}

SetCurrentDirectory("..");

}

} while (FindNextFile(hFF, &finfo));

FindClose(hFF);

return bResult;

}

void WalkDir(constchar \*path, BOOL(\*Proc)(constWIN32\_FIND\_DATA \*, int))

{

char cwd[MAX\_PATH];

GetCurrentDirectory(MAX\_PATH, cwd);

WalkDirRecur(path, Proc, 0);

if (!SetCurrentDirectory(cwd)) {

fprintf(stderr, "Cannot set directory!..\n");

exit(EXIT\_FAILURE);

}

}

BOOL DispTree(constWIN32\_FIND\_DATA \*fd, intlevel)

{

if (!stricmp(fd->cFileName, "sample.pdb"))

returnFALSE;

printf("%\*s\n", level \* 4 + strlen(fd->cFileName), fd->cFileName);

returnTRUE;

}

int main(void)

{

WalkDir("e:\\dropbox\\kurslar\\sysprog-2015", DispTree);

return 0;

}

Buradaki call-back fonksiyonun geri dönüş değerinin BOOL türden olduğuna dikkat ediniz. Bu fonksiyon 0 ile geri döndüğünde artık özyineleme sonlandırılmaktadır. Özyinelemenin sonlandırılma biçimini inceleyiniz.

UNIX/Linux sistemlerinde dizin ağacını dolaşan fonksiyon şöyle yazılabilir:

#include<stdio.h>

#include<string.h>

#include<sys/stat.h>

#include<unistd.h>

#include<dirent.h>

void walkdir(constchar \*path)

{

DIR \*dir;

structdirent \*dire;

structstat finfo;

if ((dir = opendir(path)) == NULL)

return;

if (chdir(path) < 0)

return;

while ((dire = readdir(dir)) != NULL) {

if (!strcmp(dire->d\_name, ".") || !strcmp(dire->d\_name, ".."))

continue;

printf("%s\n", dire->d\_name);

if (lstat(dire->d\_name, &finfo) < 0)

break;

if (S\_ISDIR(finfo.st\_mode)) {

walkdir(dire->d\_name);

chdir("..");

}

}

closedir(dir);

}

int main(void)

{

walkdir("/");

return 0;

}

UNIX/Linux sistemleri için de aynı biçimde fonksiyonun fonksiyon göstericisi alarak işlem yapan biçimini yazabiliriz:

#include<stdio.h>

#include<string.h>

#include<sys/stat.h>

#include<unistd.h>

#include<dirent.h>

#defineFALSE 0

#defineTRUE 1

typedefintbool\_t;

bool\_t walkdir\_recur(constchar \*path, bool\_t(\*proc)(constchar \*, structstat \*, int), intlevel)

{

DIR \*dir;

structdirent \*dire;

structstat finfo;

bool\_t result;

if ((dir = opendir(path)) == NULL)

returnFALSE;

if (chdir(path) < 0) {

closedir(dir);

returnFALSE;

}

result = TRUE;

while ((dire = readdir(dir)) != NULL) {

if (!strcmp(dire->d\_name, ".") || !strcmp(dire->d\_name, ".."))

continue;

if (lstat(dire->d\_name, &finfo) < 0) {

result = FALSE;

break;

}

if (!proc(dire->d\_name, &finfo, level)) {

result = FALSE;

break;

}

if (S\_ISDIR(finfo.st\_mode)) {

if (!walkdir\_recur(dire->d\_name, proc, level + 1)) {

result = FALSE;

break;

}

if (chdir("..") < 0) {

result = FALSE;

break;

}

}

}

closedir(dir);

return result;

}

bool\_t walkdir(constchar \*path, bool\_t(\*proc)(constchar \*, structstat \*, int))

{

char cwd[4096];

bool\_t result;

if (getcwd(cwd, 4096) == NULL)

returnFALSE;

result = walkdir\_recur(path, proc, 0);

if (chdir(cwd) < 0)

returnFALSE;

return result;

}

bool\_t disp(constchar \*path, structstat \*finfo, intlevel)

{

printf("%\*s\n", level \* 4 + (int)strlen(path), path);

returnTRUE;

}

int main(void)

{

walkdir("/home/csd", disp);

return 0;

}

**Türden Bağımsız Her Diziyi Sıraya Dizebilen Fonksiyon Nasıl Yazılabilir?**

Normal olarak bir sort fonksiyonu yalnızca belli bir türdeki dizileri sıraya dizebilir. Örneğin:

void sort(int \*pi, size\_t size);

Buradaki sort fonksiyonu yalnızca int türden bir diziyi sıraya dizebilir. Eğer biz long türden bir diziyi sıraya dizmek istiyorsak başka bir fonksiyon yazmalıyız:

void sort\_long(long \*pi, size\_t size);

Böyle her tür için içi aynı olan fakat türleri farklı olan fonksiyonları defalarca yazmak gerekebilir. Bu zahmeti ortadan kaldırmak için C++'ta template, Java ve C#'ta generic fonksiyonlar bulunmaktadır. Fakat template ya da generic fonksiyonlar özü değiştirmemektedir. Yalnızca zahmeti ortadan kaldırmaktadır. Peki her türden diziyi sıraya dizebilen bir sort fonksiyonu yazılamaz mı? Böyle bir sort fonksiyonunun parametresi void bir gösterici olmalıdır. Ayrıca dizinin bir elemanının uzunluğu ve dizi uzunluğu fonksiyona parametre olarak geçirilmelidir. Algoritma ne olursa olsun her sort fonksiyonunda dizinin iki elemanının karşılaştırılıp yer değiştirmesi teması vardır. İşte karşılaştırma işlemi fonksiyonu çağırana bırakılabilir. Böylece fonksiyon her türden diziyi sıraya dizebilir. Örneğin:

#include<stdio.h>

void swap\_elem(char \*pc1, char \*pc2, size\_twidth)

{

size\_t i;

char temp;

for (i = 0; i <width; ++i) {

temp = pc1[i];

pc1[i] = pc2[i];

pc2[i] = temp;

}

}

void sort(void \*base, size\_tcount, size\_twidth, int (\*cmp)(constvoid \*, constvoid \*))

{

size\_t i, k;

char \*cbase = (char \*)base;

char \*pc1, \*pc2;

for (i = 0; i <count - 1; ++i)

for (k = 0; k <count - 1 - i; ++k) {

pc1 = cbase + k \* width;

pc2 = cbase + (k + 1) \* width;

if (cmp(pc1, pc2) > 0)

swap\_elem(pc1, pc2, width);

}

}

int comparer1(constvoid \*pv1, constvoid \*pv2)

{

constint \*pi1 = (constint \*)pv1;

constint \*pi2 = (constint \*)pv2;

if (\*pi1 > \*pi2)

return 1;

if (\*pi1 < \*pi2)

return -1;

return 0;

}

typedefstructtagPERSON {

char name[32];

int no;

} PERSON;

int comparer2(constvoid \*pv1, constvoid \*pv2)

{

constPERSON \*p1 = (constPERSON \*)pv1;

constPERSON \*p2 = (constPERSON \*)pv2;

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

}

int comparer3(constvoid \*pv1, constvoid \*pv2)

{

constPERSON \*p1 = (constPERSON \*)pv1;

constPERSON \*p2 = (constPERSON \*)pv2;

return p1->no - p2->no;

}

int main(void)

{

{

int a[10] = { 34, 23, 45, 11, 78, 43, 34, 87, 33, 21 };

int i;

sort(a, 10, sizeof(int), comparer1);

for (i = 0; i < 10; ++i)

printf("%d ", a[i]);

printf("\n");

printf("--------------------\n");

}

{

int i;

PERSON persons[] = {

{ "Ali Serce", 123 }, { "Kaan Aslan", 456 }, { "Necati Ergin", 321 },

{ "John Lennon", 54 }, { "Abidin Yarata", 115 }

};

sort(persons, 5, sizeof(PERSON), comparer2);

for (i = 0; i < 5; ++i)

printf("%s, %d\n", persons[i].name, persons[i].no);

printf("--------------------\n");

sort(persons, 5, sizeof(PERSON), comparer3);

for (i = 0; i < 5; ++i)

printf("%s, %d\n", persons[i].name, persons[i].no);

return 0;

}

}

C'nin standart qsort fonksiyonu da aynı parametrik yapılara sahiptir ve genel kullanımı yukarıda yazmış olduğumuz sort fonksiyonu ile aynı biçimdedir:

void qsort(void \*base, size\_tcount, size\_twidth, int (\*cmp)(constvoid \*, constvoid \*));

Örneğin:

#include<stdio.h>

#include<stdlib.h>

int comparer1(constvoid \*pv1, constvoid \*pv2)

{

constint \*pi1 = (constint \*)pv1;

constint \*pi2 = (constint \*)pv2;

if (\*pi1 > \*pi2)

return 1;

if (\*pi1 < \*pi2)

return -1;

return 0;

}

typedefstructtagPERSON {

char name[32];

int no;

} PERSON;

int comparer2(constvoid \*pv1, constvoid \*pv2)

{

constPERSON \*p1 = (constPERSON \*)pv1;

constPERSON \*p2 = (constPERSON \*)pv2;

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

}

int comparer3(constvoid \*pv1, constvoid \*pv2)

{

constPERSON \*p1 = (constPERSON \*)pv1;

constPERSON \*p2 = (constPERSON \*)pv2;

return p1->no - p2->no;

}

int main(void)

{

{

int a[10] = { 34, 23, 45, 11, 78, 43, 34, 87, 33, 21 };

int i;

qsort(a, 10, sizeof(int), comparer1);

for (i = 0; i < 10; ++i)

printf("%d ", a[i]);

printf("\n");

printf("--------------------\n");

}

{

int i;

PERSON persons[] = {

{ "Ali Serce", 123 }, { "Kaan Aslan", 456 }, { "Necati Ergin", 321 },

{ "John Lennon", 54 }, { "Abidin Yarata", 115 }

};

qsort(persons, 5, sizeof(PERSON), comparer2);

for (i = 0; i < 5; ++i)

printf("%s, %d\n", persons[i].name, persons[i].no);

printf("--------------------\n");

qsort(persons, 5, sizeof(PERSON), comparer3);

for (i = 0; i < 5; ++i)

printf("%s, %d\n", persons[i].name, persons[i].no);

return 0;

}

}

**Sınıf Çalışması:** Belli bir dizindeki dosyaları uzunluklarına göre listeleyen programı yazınız.

**Çözüm:**

#include<stdio.h>

#include<stdlib.h>

#include<string.h>

#include<Windows.h>

#defineBLOCK\_SIZE 10

void ExitSys(LPCSTR lpszMsg, int status);

void Disp(constWIN32\_FIND\_DATA \*files, size\_t size);

int CompareBySize(constvoid \*pv1, constvoid \*pv2);

int CompareByName(constvoid \*pv1, constvoid \*pv2);

int main(void)

{

HANDLE hFileFind;

WIN32\_FIND\_DATA finfo;

WIN32\_FIND\_DATA \*files;

int count;

if ((hFileFind = FindFirstFile("c:\\windows\\\*.\*", &finfo)) == INVALID\_HANDLE\_VALUE)

ExitSys("FindFirstFile", EXIT\_FAILURE);

count = 0;

files = NULL;

do {

if (!(finfo.dwFileAttributes &FILE\_ATTRIBUTE\_DIRECTORY)) {

if (count % BLOCK\_SIZE == 0) {

if ((files = (WIN32\_FIND\_DATA \*)realloc(files, (count + BLOCK\_SIZE) \* sizeof(WIN32\_FIND\_DATA))) == NULL) {

fprintf(stderr, "cannot allocate memory!..\n");

exit(EXIT\_FAILURE);

}

}

files[count] = finfo;

++count;

}

} while (FindNextFile(hFileFind, &finfo));

qsort(files, count, sizeof(WIN32\_FIND\_DATA), CompareBySize);

Disp(files, count);

printf("------------------------------\n");

qsort(files, count, sizeof(WIN32\_FIND\_DATA), CompareByName);

Disp(files, count);

free(files);

return 0;

}

int CompareBySize(constvoid \*pv1, constvoid \*pv2)

{

constWIN32\_FIND\_DATA \*f1 = (constWIN32\_FIND\_DATA \*)pv1;

constWIN32\_FIND\_DATA \*f2 = (constWIN32\_FIND\_DATA \*)pv2;

if (f1->nFileSizeLow > f2->nFileSizeLow)

return 1;

if (f1->nFileSizeLow < f2->nFileSizeLow)

return -1;

return 0;

}

int CompareByName(constvoid \*pv1, constvoid \*pv2)

{

constWIN32\_FIND\_DATA \*f1 = (constWIN32\_FIND\_DATA \*)pv1;

constWIN32\_FIND\_DATA \*f2 = (constWIN32\_FIND\_DATA \*)pv2;

return stricmp(f1->cFileName, f2->cFileName);

}

void Disp(constWIN32\_FIND\_DATA \*files, size\_tsize)

{

size\_t i;

for (i = 0; i <size; ++i)

printf("%-40s%lu\n", files[i].cFileName, (unsignedlongint) files[i].nFileSizeLow);

}

void ExitSys(LPCSTRlpszMsg, intstatus)

{

DWORD dwLastError = GetLastError();

LPTSTR lpszErr;

if (FormatMessage(FORMAT\_MESSAGE\_ALLOCATE\_BUFFER | FORMAT\_MESSAGE\_FROM\_SYSTEM, NULL, dwLastError,

MAKELANGID(LANG\_NEUTRAL, SUBLANG\_DEFAULT), (LPTSTR)&lpszErr, 0, NULL)) {

fprintf(stderr, "%s: %s", lpszMsg, lpszErr);

LocalFree(lpszErr);

}

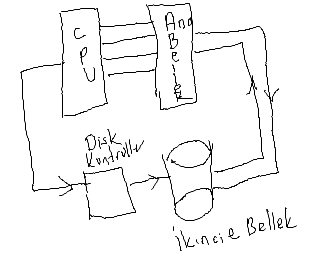
exit(status);

}

**Bellek Sistemleri**

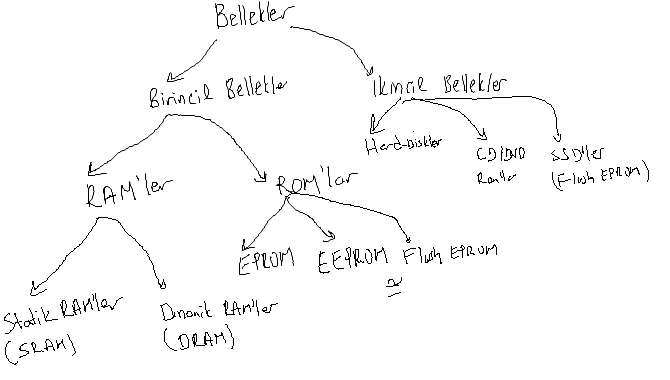
Bilgileri tutmakta kullanılan birimlere bellek (memory) denir. Bilgisayar sistemlerindeki bellekler "Birincil Bellekler (Primary Memory)" ve "İkincil Bellekler "Secondary Memory" olmak üzere ikiye ayrılmaktadır. Birincil belleklere "ana bellekler" ya da halk arasında "RAM" de denilmektedir. Birincil Bellekler ya da Ana Bellekler CPU ile elektriksel olarak bağlantı halindedir. Bunlar bilgisayarın güç kaynağı kesildiğinde bilgileri tutamazlar. İkincil bellekler güç kaynağı kesildiğinde bilgileri tutabilen belleklerdir. Tipik bir çalışmada bilgisayar kapatılmadan önce bilgiler İkincil Belleklerde saklanır. İkincil bellekler genellikle dizinler ve dosyalar biçiminde organize edilmiştir. İkincil bellekler tipik olarak hard-diskler, SSD'ler, flash EPROM'lar, CD/DVD ROM'lar biçiminde organize edilmektedir.

Tipik olarak bir bilgisayar sisteminin bellek mimarisi şöyle özetlenebilir:



CPU çalışırken sürekli program komutlarını ve nesneleri Ana Bellekten okur ve oraya yazar. İkincil Belleklerle Ana Bellek arasında aktarım yolu vardır. Aktarım şöyle yapılır: CPU İkincil Belleği kontrol eden birime (Disk Contrller) komutlar yollayarak ondan transfer yapmasını ister. Bundan sonra artık olayı izlemez. Kontrol birimi disk üzerindeki işlemcileri programlar, diskin kafalarını hareket ettirir. Bilgileri Birincil Bellekte belirlenen adresten itibaren transfer eder. İşlem bitince CPU'yu bir donanım kesmesiyle haberdar eder.

Bellekler tipik olarak aşağıdaki gibi sınıflandırılmaktadır:



Birincil Bellekler RAM'ler ve ROM'lar biçiminde ikiye ayrılmaktadır. RAM sözcüğü "Random Access Memory" sözcüklerinden kısaltılmıştır. Buradakai "Randrom" sözcüğü hız belirtmektedir. RAM'ler read/write belleklerdir. Bunlar teknolojik olarak Statik RAM ve Dinamik RAM olmak üzere iki biçimde üretilmektedir. Statik RAM'ler daha hızlıdır (tipik olarak 1 nanosaniyenin altında), Dinamik RAM'lar daha yavaştır (tipik olarak 10 nanosaniye civarında). SRAM'ler daha büyük yer kaplar, DRAM'lar daha küçük yer kaplarlar. SRAM'ler flip-flop devresiyle gerçekleştirilmektedir. Dolayısıyla bunların 1 biti genellikle 4 transistörle yapılır. Halbuki DRAM'ların 1 biti bir transistör ve bir kapasitif elemanla yapılmaktadır. DRAM'larda tazeleme problemi vardır. Kapasitif elemandaki yük zamanla düşer. Bunun düşmemesi için tazleme yapılmaktadır. Tazeleme işlemini de bizzat CPU'unun kendisi yapmaktadır. SRAM'ların tazeleme problemleri yoktur. Toplamda SRAM'ler DRAM'lardan daha pahalıdır. Bu yüzden ana bellekler DRAM'larla yapılmaktadır. SRAM'ler CPU içerisindeki cache sistemlerinde tercih edilmektedir.

ROM sözcüğü "Read Only Memory" sözcüklerinden kısaltılmıştır. Bu sözcüğün artık bugün için teknolojik bir anlamı kalmamıştır. Eskiden ROM'lar yalnızca okunabiliyordu fakat bilgisayarın güç kaynağı kesildiğinde bilgileri yine tutabiliyordu. EPROM'lar artık teknoloji dışı kalmaya başlamıştır. EPROM'lar (Erasable Programmable ROM) EPROM silici denilen aygıtla silnebiliyor ve EPROM programlayıcıyla içine yeni bilgiler aktarılabiliyordu. Yeni teknolojide artık CPU tarafından yazılabilen EEPROM (Electrically Erasable ROM) ve Flash EPROM'lar kullanılmaktadır. Bunlar yine güç kaynağı çekildiğinde bilgiyi tutmaya devam ederler. Ancak hiç bilgisayardan çıkartmadan yeniden programlanabilirler.

Bir CPU reset edildiğinde çalışma belli bir adresten başlar. Buna CPU'nun reset vektörü denilmektedir. Orada bir kodun hazır olması gerekir. İşte bu kod eskiden EPROM'larda tutuluyordu. Artık bunlar için EEPROM bellekler kullanılmaktadır. Kullandığımız PC'lerde bu bellek alanına BIOS (Basic Input Output System) denilmektedir.

İkincil bellekler için hala en yoğun kullanılan teknoloji Disk teknolojisidir. Diskler ElektroMekanik aygıtlardır. Bunların fiyatları çok makuldür. Fakat güç harcamaları yüksektir ve çok hızlı değillerdir. Ayrıca hard diskler sarsıntıya karşı çok korunaklı değildir. Bugün artık yavaş yavaş hard disklerin yerlerini EEPROM ve Flash EPROM tarzı bellekler almaktadır. EEPROM tarzı belleklerden okuma çok hızlıdır (nano saniye mertebesinde) fakat bunlara yazma göreli olarak yavaştır (mili saniyeler mertebesinde). Ayrıca bu teknolojide bir RAM bloğuna belli kez yazma yapılabilmektedir. Gerçi bu sayı gitgide yükseltiliyor olsa da hala handikaplardan biridir.

**Cache Sistemleri**

Bilgisayar sistemlerinde genellikle hızlı ve yavaş bellekler söz konusudur. Yavaş bellek bol ve ucuzdur, hızlı bellek de az ve pahalıdır. İşte yavaş belleğe erişim oranını azaltmak için Cache sistemleri kullanılmaktadır. Bir cache sisteminde yavaş belleğin belli bir bölümü hızlı bellekte tutulur. Böylece bilgiye erişilmek istendiğinde önce bilgi hızlı bellekte aranır. Bulunursa hemen oradan alınır. Bulunamazsa yavaş belleğe başvurulur. Bu sistemde yavaş belleğin bir bölümünü saklamakta kullanılan hızlı belleğe "cache" denilmektedir. Eğer talep edilen bilgi hızlı bellekte bulunursa bu duruma "cache hit" denilmektedir. Eğer bilgi hızlı bellekte bulunamazsa bu durumda yavaş belleğe başvurulur. Buna da "cache miss" denilmektedir. Bir cache sisteminin performansı "cache hit oranı (cache hit ratio)" ile ölçülür. Cache hit oranı n tane erişimin kaçının cache'ten karşılandığını belirtir. Örneğin cache hit oranı %70 demek, 100 erişimin 70'inin hızlı bellekten karşılanması demektir.

Çok karşılatığımız önemli cache sistemlerinin bazıları şunlardır:

- Modern işletim sistemleri diske erişimi azaltmak için son erişen disk bloklarını RAM'de tutarlar. Bu cache sistemine "disk cache sistemi" ya da "buffer cace" denilmektedir.

- Mikroişlemcilerin içerisinde statik RAM'lerle yapılan cache bellekler vardır. İşlemci DRAM göreli olarak yavaş olduğu için onun belli bölümlerini kendi içerisinde bir cache'e çeker. RAM'e erişeceği zaman önce oraya başvurur. Orada bilgiyi bulursa hızlı bir biçimde elde etmiş olur, bulamazsa DRAM erişimi yapar.

- Web tarayıcıları erişilen web sayfalarının içeriğini yerel makinada diskte tutuyor olabilir. Böyulece aynı sayfaya talep edildiğinde bunu hızlı bir biçimde getirebilir.

- İşlemci sayfat tablosundaki Sayfa girişlerini kendi içerisinde küçük bir tampon alanda tutar. Böylece sayfa tablousuna erişimi azaltır.

- İşletim sistemleri son gezilen dizin girişlerini bir cache sisteminde toplamaktadır. Buna "Directory Entry Cache" denilmektedir.

- Pek çok dilde dosya işlemi yapan fonksiyonlar ve sınıflar "user modda" dosyanın son okunulan bölgelerini bir cache sisteminde tutmaktadır.

**Anahtar Notlar:** Buffer (tampon) sözcüğüyle cache (önbellek) sözcüğü bazen birbirlerine karıştırılmaktadır. Buffer bir meşguliyet yüzünden gelen bilgilerin bekletildiği bölgelere denilmektedir. Buffer sisteminin amacı bilgilerin uygun zamanda işlenmek üzere bekletilmesidir. Halbuki cache sisteminin amacı hız kazancı sağlamaktır. Örneğin Yani buffer sisteminin ana amacı bilgilerin kaybedilmemesi, uygun zamanda işlenmek üzere bekletilmesidir.

**Cache Terminolojisi**

Bir cache sistemi read-only olabilir ya da read-write olabilir. Eğer cache'e yalnızca okuma yaparken erişiliyorsa böyle cache sistemlerine read-only cache sistemleri denir. Eğer cache'e hem okuma hem de yazma amaçlı erişiliyorsa böyle cache sistemlerine read-write cache sistemleri denir. Read-only cache sistemlerinde okuma için önce cache'e başvurulur. Fakat yazma her zaman yavaş belleğe yapılır. Halbuki read-write cache sistemlerinde hem okuma hem de yazma sırasında cache kullanılmaktadır. Read-write cache sistemleri genel olarak daha hızlıdır. Ancak bazı durumlarda elektirik kesilmesi gibi anormalliklerde bilginin bütünlüğü bozulabilir.

Bazı cache sistemlerinde yavaş belleğin tek bir ardışıl bloğu cache'te tutulmaktadır. Bazı sistemlerde ise yavaş belleğin birden fazla küçük blokları cache'te tutulabilmektedir. Yavaş belleğin cache'teki kısımlarını turuan cache bölgelerine "cache line" deilmektedir. Tabi böyle bir sistemde ilgili bilgi aranırken etkin bir biçimde onun cache'te olup olmadığının belirlenmesi gerekir.

Cache için ayrılan cache line'ların tıka basa doldu olduğunu düşünelim. Yeni bir bilgi cache'e alınacak olduğunda ne olacaktır? Bu durumda hangi cache line'ı cache'ten çıkartmak gerekir? Burada kullanılan algoritmalara cache yer değiştirme politikaları (cache replacement policy) denilmektedir. Tipik olarak üç tür yer değiştirme politikası vardır:

1) Least Frequently Used (LFU): Bu algortimada her cache line için bir sayaç tutulur. O cache line'a erişildikçe sayaç artırılır. Sonra cache'te bir line çıkartılacağı zaman sayacı en az olan çıkartılır. Burada o zamana kadar az kullanılmış bir cache bloğunun ileride de az kullanılacağı varsayımı yapılmaktadır.

2) Least Recently Used(LRU): Burada son zamanlarda en az erişilen cache line'lar cache'ten çıkartılma yoluna gidilir. Yani bu algoritmada son zamanlarda erişilmiş olma değerli bir durumdur. Örneğin işletim sistemlerinde cache sistemlerinin çoğunda bu model tercih edilmektedir. Bu sistemin tipik gerçekleştirimi şöyle yapılmaktadır: Bir bağlı listede cache line'lar tutulur (numaraları da tutulabilir). Cache line kullnıldıkça bağlı listenin başına alınır. Böylşece kullanılmayanlar zaten sonda kalır. Cach'ten line çıkartılacağı zaman listede sonda olan çıkartılır.

3) Most Frequenly Use (MFU): Bazı sistemlerde line'lara toplam erişim sabit olabilmektedir. Örneğin her bir bloğa toplamda 1000 kere erişileceği biliniyor olabilir. Böyle bir sistemde tam tersine çok erişilmiş olan cache line'ların cache'ten atılması daha makuldür. Fakat bu algoritmanın uygun olabileceği sistemler çok azdır.

Uygulamada en fazla karşılaşılan cache yer değiştirme politakısı LRU'dur.

**Cache Sistemleri Nasıl Oluşturulur?**

Cache sistemleri donanımsal ya da yazılımsal olarak oluşturulabilmektedir. Donanımsaldan kastededilen tamamen elektrik devreleriyle cache'in oluşturulmasıdır. Örneğin Microişlemcinin içerisindeki cache bellek ile DRAM arasındaki sistem tamamen donanımsal olarak oluşturulmuştur. Yazılımsal oluşturmada cache sistemi bir program tarafından oluşturulur. Örneğin işletim sisteminin disk cache sistemi işletim sisnteminin kernel kodları tarafından oluşturulmaktadır. Şüphesiz biz bu kursta daha çok yazılımsal olarak gerçekleştirilen cache sistemleri üzerinde duracağız.

Cache sistemleri yazılımsal olarak oluşturulurken sistemi kontrol etmek için bir veri yapısına gereksinim duyulur. (Genellikle cache üzerinde arama yapmak için cache blokları bir hash tablosu biçiminde organize edilmektedir.) Aşağıda bir dosyanın "cache line"lar kullanılarak cache'lenmesine ilişkin örnek bir arayüz verilmektedir. Bu örnek yalnızca fikir vermek için oluşturulmuş bir "pseudo code"dur:

typedef struct tagCACHE\_LINE {

char buf[LINE\_SIZE];

size\_t offset

....

} CACHE\_LINE;

typedef struct tagCACHE\_FILE {

FILE \*f;

size\_t cacheSize;

CACHE\_LINE cacheLines[NCACHE\_LINES];

....

} CACHE\_FILE, \*HCACHE\_FILE;

HCACHE\_FILE OpenCacheFile(const char \*path, const char \*mode);

size\_t ReadCacheFile(HCACHE\_FILE hFile, size\_t count, void \*buf);

size\_t WriteCacheFile(HCACHE\_FILE hFile, size\_t count, const void \*buf);

long LocateFilePointer(HACACHE\_FILE hFile, long offset);

**Cache sistemlerinin Performansını Ne Etkiler?**

Bir cache sisteminin performanısını etkileyen unsurlar şunlardır:

1) Kullanılan cache algoritması: Yani yavaş belleğin neresinin cache'te tutulacağını, cache'ten gerektiğinde hangi bloğun çıkartılacağını belirleyen algoritmalar. Bu algoritmalara ilgili sistem analiz edilerek karar verilir.

2) Cache miktarı: Şüphesiz cache ne kadar büyütülürse performan o kadar artar.

3) Cache belleğin hızı: Hızlı cache kullanmak şüphesiz performanı artırır.

Yukarıdaki 3 performans unsurundan en önemlisi "cache algoritması"dır. Büyük bir cache yanlış bir algoritma ile kullanılırsa performans umulduğu gibi artmaz. Cache belleğin hızı konusunda genellikle tasarımcı çok belirleyici olamamaktadır. Cache miktarı da zaten belli sınırlarda olur.

**İşlemcilerin Koruma Mekanizması (Protection mechanisms)**

Çok prosesli işletim sistemlerinde çalışmakta olan programlar aynı RAM üzerinde bulunurlar. İşletim sisteminin kendisi de yine RAM'de bulunmaktadır. Bu sistemlerde bir programın bilerek ya da yanlışlıkla başka programların bellek alanlarına erişmesi istenmez. Çünkü oradaki bilgiler değerli olabilir, oradaki bilgilerdeki bozulma o programın belki de tüm sistemin çökmesine yol açabilir.



Öte yandan bazı makina komutları da sistemin tümden çökmesine yol açabilmektedir. Örneğin CLI gibi, OUT gibi makina komutları sistem güvenliği bakımından tehlikelidir. İşte modern sistemler bu tür olumsuzluklaran başkalarının ve tüm sistemin etkilenmesini engellemek için koruma mekanizmasına sahiptir.

Koruma mekanizmasının iki yönü vardır: Bellek koruması ve komut koruması. Bir program kendi alanın dışına erişim yapamamlıdır. Buna bellek koruması denir. Bir programın sistemi çökertecek makina komutlarını kullanamaması gerekir. Buna da komut koruması denir. Tabi bazı programların (işletim sisteminin ve aygıt sürücülerin) bunlardan muhaf olması da gerekir.

İşte modern büyük kapasiteli işlemciler koruma mekanizmasına sahip olarak tasarlanırlar (Örneğin Intel 80X86, ARM modelleri, MIPS, Itanium, PowerPC vs.) Bu sistemlerde bellek koruması ya da komut koruması ihlal edildiğinde bunu birinci elden işlemci tespit eder ve işletim sistemine bildirir. İşletim sistemi de o prosesi cezalandırarak (genellikle) sonlandırır.

Koruma mekanizmasına sahip işlemcilerde çalışmakta olan kodun bir modu vardır: (İntel 4 mod kullanmasına karşın yalnızca iki mod işletim sistemi yazanlar tarafında kullanılmıştır. Diğer işlemcilerin çoğu iki moda sahiptir) Kernel mode User Mode. Kernel kodları kernel modda çalışır. Kernel moddaki kodlar koruma mekanizmasına takılmazlar. Yani bu kodlar belleğin her yerine erişebilirler ve tüm makina komutlarını kullanabilirler. User mod kodlar ise koruma mekanizması tarafından denetlenirler. Bizim Windows'ta Linux'ya yazmış olduğumuz normal programların hepsi user modda çalıştırılır.

Peki madem işletim sisteminin kodları kernel alanı içerisindeki data'lara erişiyor ve özel komutları kullanbiliyor, biz bir sistem fonksiyonunu çağırdığımızda ne olacaktır? Eğer bizim user moddaki akışımız o sistem fonksiyonunu çağırmışsa oradaki kodlar koruma engeline takılmaz mı? İşte bu tür sistemlerde ismine kapı (gate) denilen bir mekanizmayla bu soruna çözüm getirilmiştir. User mod bir program işletim sisteminin bir sistem fonksiyonunu çağırdığında otomatik olarak kapı mekznizması sayesinde kernel moda geçiş yapar. İşletim sisteminin sistem fonksiyonu kernel modda çalıştırılmış olur. Sistem fonksiyonunun çalışması bittiğinde akış yeniden user moda döner. Buna prosesin "user moddan kernel moda geçmesi (mode switch)" denilmektedir. Intel işlemcilerinde bu mekanizmaya kapı (gate) mekanizması denilmektedir. Kapılar fonksiyonlara yerleştirilebilir. Böylece yalnızca o fonksiyonlar çalıştırıldığında kernel moda geçiş yapılır. Kapı yerleştirmek ancak işletim sisteminin yapabileceği bir faaliyettir. O halde bir proses tüm ömrünü user modda geçirmez. Arada kernel moda da geçer.

Kernel moda geçmenin bir zaman maliyeti vardır. Çünkü geçiş sırasında binlerce makina komutu çalışabilmektedir. Örneğin geçiş sırasında user moddaki stack'teki bilgiler daha korunaklı kernel mod stack'e taşınmaktadır (stack switch).

Peki biz kernel modda çalışacak bir program yazamaz mıyız? Yanıt: Yazabiliriz. Bu tür programlara kernel modülleri ve aygıt sürücüler denilmektedir. aygıt sürücü olarak yazılırlar. Aygıt sürücüler kernel alanına yüklenerek sabki kernel'ın bir parçasıymış gibi çalışırlar. Aygıt sürücü içerisindeki kodlar user mod programlar tarafından çağrılabilmektedir. Bu durumda yine kapı mekanizması yoluyla kernel moda geçiş yapılır. Her işletim sisteminin bir aygıt sürücü mimarisi vardır. Aygıt sürücüler o işletim sistemine özgü (hatta o versiyona özgü) bir biçimde yazılırlar. Çünkü aygıt sürücüler yalnızca kernel'daki fonksiyonları kullanırlar.

**C'nin Standart Dosya Fonksiyonlarının Kullandığı Cache Mekanizması**

C'nin prototipleri <stdio.h> içerisinde olan dosya fonksiyonları işletim sisteminin sistem fonksiyonlarını daha az çağırmak için kendi içerisinde bir cache sistemi kullanmaktadır. Bu nedenle C'nin dosya fonksiyonlarına "buffered IO" fonksiyon ları da denir.

C'nin fopen fonksiyonu işletim sisteminin sistem fonksiyonlarıyla (Linux'ta open POSIX fonksiyonu sys\_open sistem fonksiyonunu çağırmaktadır, Windows'ta CreateFile API fonksiyonu da bir sistem fonksiyonu gibidir) dosyayı açar. Sonra o dosya için bir cache oluşturur. Böylece okuma yazma işlemlerinde bu cache kullanılır. Yani örneğin biz işin başında fgetc fonksiyonu ile dosyadan bir byte okumak istediğimizde, fgetc bir byte değil işletim sisteminin sistem fonksiyonuyla (Linux'ta read, Windows'ta ReadFile) daha fazla bilgiyi okuyarak cache'e çeker. Sonraki okumalarda bize bilgiyi cache'ten verir. Burada cache sistemi tamamen bizim user mod prosesimizin bir parçası biçimindedir. Kernel ile bir ilgisi yoktur. Standart stdio.h fonksiyonlarının kullandığı bu cache sistemi read/write bir sistemdir.

**Anahtar Notlar:** Standart C fonksiyonları için bu bağlamda cache yerine "tampon"sözcüğü kullanılmaktadır. Aslında tampon sözcüğü buradaki durumu iyi yansıtmamaktadır. Çünkü tampon bilgilerin daha sonra işlenmek üzere saklandığı bellek bölgelerini belirtir. Halbuki cache hızlandırma amacıyla kullanılan bir sistemi betimlemektedir. Fakat C'de bu kavram hep tampon (buffer) biçiminde ifade edildiği için biz de aşağıdaki anlatımlarda bazen cache terimini bazen de tampon terimini kullanacağız.

C'nin standart dosya fonksiyonlarının kullandığı cache sisteminin şu özellikleri vardır:

- Cache read/write biçimdedir.

- Hemen her zaman tek cache line kullanılır. Yani cache'te dosyanın tek bir ardışık bölgesi tutulut.

- Her dosyaynın ayrı bir cache'i vardır

Standart dosya fonksiyonlarının kullandığı cache 3 modda çalışabilmektedir: Tam tamponlamalı mod (full buffered mode), satır tamponlamalı mod (line buffered) ve sıfır tamponlamalı mod (no buffered).

Tam tamponlamalı modda tampon tam kapasiteyle kullanılır. Yani okuma ve yazma tampondan yapılır. Yazılan bilgiler tampona yazılır. Tampon dolunca ya da tampona dosyanın başka bir kısmı çekileceği zaman tampondaki bilgi dosyaya yazılır. Aynı zamanda fflush fonksiyonu da tampondaki bilgileri o anda dosyaya aktarmak için kullanılır. Şüphesiz dosya fclose ile kapatıldığında da fflush işlemi yapılmaktadır.

Satır tamponlamalı modda, tamponda (yani cache'te) yalnızca bir satırlık bilgi (yani '\n' görene kadar ('\n' de dahil) bilgi tamponda tutulur. Benzer biçimde tampondaki bilgi '\n' görüldüğünde dosyaya aktarılır.

Sıfır tamponlamalı modda tampon (yani cache) devre dışı bırakılır. Her yazma ve okuma işleminde doğrudan işletim sisteminin sistem fonksiyonları çağrılır.

C standartlarında stdin, stdout ve stderr dosyalarının default tamponlaama modları hakkında bazı şeyler söylenmiştir. (Bu konu ileride ele alınacak). Fakat normal dosyaların default tamponlama modlarının ne olacağı konusunda birşey söylenmemiştir. Bu durum bunların default durumda herhangi bir tamponlama stratejisine sahip olabileceğini belirtir. Ancak tabi sistemlerin hepsinde normal dosyaların default taponlama modları "tam tamponlamalalı mod"dur.

Bir dosyanın tamponlama modu setbuf ve setvbuf fonksiyonlarıyla değiştirilebilmektedir. Aslında setvbuf fonksiyonu zaten setbuf fonksiyonunu işlevsel olarak kapsar. setbuf yetersiz olduğu için setvbuf standartlara dâhil edilmiştir. Ancak dosyanın tamponlama modu değiştirilecekse bu işlemin fopen fonksiyonundan hemen sonra (yani o dosya için başka bir dosya fonksiyonu henüz çağrılmadan) yapılması gerekir. Eğer bu yapılmazsa tanımsız davranış (undefined behavior) söz konusu olur.

setbuf fonksiyonunun prototipi şöyledir:

#include <stdio.h>

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

Fonksiyonun birinci parametresi tamponlama ile ilgili işlem yapılacak dosya bilgi göstericisini, ikinci parametresi yeni tamponun adresini belirtir. İkinci parametre NULL geçilirse sıfır tamponlama söz konusu olabilir. Buradaki yeni tamponun uzunluğu <stdio.h> içerisindeki BUFSIZ kadar olmalıdır. Örneğin:

#include<stdio.h>

#include<stdlib.h>

int main(void)

{

FILE \*f;

char buf[BUFSIZ];

if ((f = fopen("test.txt", "r+")) == NULL) {

fprintf(stderr, "cannot open file!..\n");

exit(EXIT\_FAILURE);

}

setbuf(f, buf); /\* artık tampon olarak buf kullanılacak \*/

/\* ...\*/

fclose(f);

return 0;

}

setvbuf fonksiyonunun parametrik yapısı da şöyledir:

int setvbuf(FILE \*stream, char \*buf, int mode, size\_t size);

Fonksiyonun birinci parametresi tamponlama stratejisi değiştirilecek dosyanın dosya bilgi göstericisini belirtir. İkinci parametre dosyanın yeni tamponunu belirtmektedir. Bu parametre NULL geçilebilir. Bu durumda tamponun fonksiyonun kendisi taarafından tahsis edileceği anlamına gelir. Üçüncü parametre yeni tamponlama modunun ne olacağını belirtir. Bu parametre aşağıdaki sembolik sabitlerden biri olarak girilmelidir:

\_IOFBF (tam tamponlama için)

\_IOLBF (satır tamponlaması için)

\_IONBF (sıfır tamponlama için)

Son parametre yeni tamponun uzunluğunu belirtir. Tabi fonksiyonun üçüncü parametresi \_IONBF olarak girilirse ikinci ve dördüncü parametrenin ne girildiğinin bir önemi yoktur. Fonksiyon baaşarı durumunda sıfır, başarısızlık durumunda sıfır dışı bir değerle geri döner. Örneğin:

#include<stdio.h>

#include<stdlib.h>

int main(void)

{

FILE \*f;

char buf[BUFSIZ];

if ((f = fopen("test.txt", "r+")) == NULL) {

fprintf(stderr, "cannot open file!..\n");

exit(EXIT\_FAILURE);

}

if (setvbuf(f, buf, \_IOLBF, BUFSIZ) != 0) {

fprintf(stderr, "setvbuf failed!..\n");

exit(EXIT\_FAILURE);

}

/\* ...\*/

fclose(f);

return 0;

}

**fgetc mi getc mi?**

C standartlarına göre fgetc ile getc arasındaki tek fark getc'in bir makro olarak yazılmış olabileceğidir. Genellikle getc bir makro olarak yazılır. Böylece bir dosyadan byte byte okuma yapılmak istendiğinde daha hızlı olsun diye getc tercih edilebilmektedir. getc fonksiyonu fonksiyon çağrısına yol açmadığı için daha daha hızlı olma eğilimindedir.

**Aygıt Sürücüler Nasıl Kullanılır?**

Genel olarak işletim sistemlerinde aygıt sürücüler sanki birer dosyaymış gibi kullanılmaktadır. Yani bir aygıt Windows'ta dosya açan CreateFile fonksiyonuyla, UNIX/Linux sistemlerinde open POSIX fonksiyonuyla açılır. Onlardan sanki bir dosyaymış gibi read ve write yapılır. Ve en sonunda yine bir dosyaymış gibi onlar kapatılır. Bir aygıt sürrücüden read yaptığımızda o aygıt sürücünün bizim tarafımızdan yazılmış olan read fonksiyonu çağrılır. write yaptığımızda da aygıt sürücünün bizim tarafımızdan yazılmış olan write fonksiyonu çağrılır. Ayrıca aygıt sürücülerin istenilen bir fonksiyonu Windows'ta DeviceIOControl, UNIX/Linux sistemlerinde ioctl fonksiyonlarıyla çağrılabilmektedir. Tabi şüphesiz aygıt sürücü fonksiyonları kernel modda işlem görürler.

**stdin, stdout, stderr Dosyaları**

C'de stdin, stdout ve stderr dosyaları program başladığında açıldığı kabul edilen aygıt sürücü dosyalarıdır. stdin ve stdout terminal aygıt sürücüsüne yönlendirilmişlerdir. stdin "read-only" stdout "write-only" modda açılmıştır. stdin dosyasından okuma yapılmak istendiğinde terminal aygıt sürücüsü klavyeden alınanları verir. stdout dosyasına da yazma yapılmak istendiğinde terminal aygıt sürücüsü yazdırılmak istenen bilgileri ekrana yazdırır. stderr programın error mesajlarının yazdırılacağı hedefi belirtir. Default olarak sistemlerde stderr dosyası da "write-only" açılmıştır ve terminal aygıt sürücüsüne yönlendirilmiştir. Bu standart dosyalar programcı tarafından açılmazlar. Zaten açık olarak program başlatılır. Yani bunların açılması derleyicilerin başlangıç kodlarında (startup code) yapılmaktadır.

Örneğin biz C'de stdout dosyasına aşağıdaki fprintf fonksiyonuyla birşey yazmak isteyelim:

fprintf(stdout, "this is a test\n");

İleri ele alınacağı üzere bu bilgi önce diğer dosyalar gibi kütüphanenin oluşturduğu tampona yazılır. Buradan hedefe aktarılırken işletim sisteminin sistem fonksiyonlarıyla (Windows'ta WriteFile, UNIX/Linux sistemlerinde write) hedefe yazdırılır. satdout terminal aygıt sürücüsü olduğu için yazılan bilgiler aygıt sürücüye verilir. Aygıt sürücü içerisindeki write fonksiyonu da bunları ekrana yazdırcak biçim oluşturulmuştur.) C'de stdin, stdout ve stderr FILE \* türünden birer dosya bilgi gösterici belirtmektedir.

**C'nin Gizli Dosya Fonksiyonları**

C'nin printf, scanf, getchar, putchar gibi fonksiyonları da aslında gizli birer dosya fonksiyonudur. printf fonksiyonu aslında fprintf fonksiyonunun default olarak stdout dosyasına yazan biçimidir. Benzer biçimde getchar aslında fgetc fonksiyonunun stdin dosyasından okuma yapan biçimidir.

**Dosyaların Yönlendirimesi (IO Redirection)**

Dosyaların hedefleri yönlendirilebilmektedir. Yani örneğin açılmış bir dosyanın hedefi "x.txt" olsun. Bu hedef "y.txt" gibi bir dosyaya yönlendirilebilir. Bu durumda dosyaya yazıldığında "x.txt" dosyasına değil "y.txt" dosyasına yazım yapılacaktır. Yönlendirme okuma amaçlı ya da yazma amaçlı yapılabilmektedir.

**Anahtar Notlar:** Yönlendirmenin mekanizması işletim sistemine ilişkin bazı detayları barındırdığından bu kursun konusu içerisinde değildir. UNIX/Linux sistemlerinde yönlendirme, Windows sistemlerindeki yönlendirme mekanizması o işletim sistemlerine ilişkin sistem programlama kurslarınıun konusu içerisindedir.

Hem Windows'ta hem UNIX/Linux sistemlerinde Komut satırında '>' işareti stdout dosyasının, '<' işareti stdin dosyasının yönlendirileceği anlamına gelir. Örneğin:

./sample > x.txt

Burada UNIX/Linux sistemlerinde sample programı çalıştırılacak, programın stdout dosyası aygıt sürücü yerine "x.txt" yönlendirilecektir. Bu durumda programda ekrana yazılan her şey dosyaya yazılmış olacaktır. Örneğin:

Benzer biçimde stdin dosyası da şöyle yönlendirilebilir:

./sample < x.txt

Bireden fazla yönlendirme berber de yapılabilir. Örneğin:

./sample > x.txt < y.txt

Burada sample programının stdout dosyası "x.txt"'ye stdin dosyası da "y.txt"ye yönlendirilmektedir.

**Anahtar Notlar:** IDE'lerde genel olarak yönlendime işlemi IDE'lerin menüleriyle de yapılabilmektedir. Örneğin Visual Studio'da proje ayarlarında "Debugging" kısmında "Command Argumentes" seçeneğinde yönlendirme ifadelerini girebiliriz.

Şüphesiz programın aynı zamanda komut satırı argümanları da girilebilir:

./sample ali veli selami > x.txt < y.txt

Burada "ali", "veli", "selami" programın komut satırı argümanlarıdır.

**stderr Dosyasının Anlamı**

stderr dosyası programın hata mesajlarının yazdırılması için kullanılan dosyadır. Fakat önceden de belirtildiği gibi dosya da terminale yönlendirilmiştir. Ancak istenirse yönlendirme sayesinde bunlar birbirlerinden ayrılabilir. Windows ve UNIX/Linux sistemlerinde '2>' sembolü stderr dosyasının yönlendirileceği anlamına gelir. Yani:

./sample

Böyle bir çalışmada hem programın normal mesajları hem de hata mesajları ekranda görünür. Fakat:

./sample 2> x.txt

Burada yalnızca programın normal mesajları ekranda görünür, hata mesajları "x.txt" dosyasına yazdırılmaktadır. UNIX/Linux sistemlerinde /dev/null isimli dosya bir aygıt sürücü dosyasıdır. Bu aygıt sürücünün write fonksiyonunun içi boştur. Böylece biz istersek aşağıdaki gibi programın hata mesajlarının kafa karıştırmasını önleyebiliriz:

./sample 2> /dev/null

Bu durumda bizim programın hata mesajlarını stderr dosyasına yazmamız iyi bir tekniktir. Çünkü bu durumda programı çalıştıracak kişiler programın hata mesajlarını yönlendirme şansına sahip olacaklardır.

**Komut Satırında Boru (Pipe) İşlemleri**

Windows ve UNIX/Linux Komut satırında '|' boru işlemi anlamına gelir. '|' karakterinin solunda ve sağında çalıştırılabilen dosya yol ifadeleri bulunur. Örneğin:

a | b

Burada a ve b birer programdır. Shell bir boru oluşturduktan sonra a ve b'yi çalıştırır. Ancak a'nın stdout dosyasını, b'nin ise stdin dosyasını boruya yönlendirir. Böylece a programının ekrana yazdıkları boruya yazılır. b'nin de klavyeden okuduğunu zannettiği şeyler borudan okunur. Böylece "sanki a'nın ekrana yuazdıklarını b klavyeden okuyormuş gibi" bir durum oluşur. Örneğin:

ls -l | wc

wc (word count) bir POSIX shell komutudur (bu sistemlerde neredeyse her komut bir programdır). wc bir dosyası komut satırı argümanı olarak alır. O dosya içerisinde kaç satır, kaç sözcük ve kaç byte olduğunu bize verir. Eğer wc'de dosya ismi verilmezse wc stdin dosyasından okuma yapar. Böylece yukarıdaki örnekte ls -l'nin ekrana yazdıklarını wc sanki stdin dosyasından okuyormuş gibi bir durum oluşur. Eğer böyle bir meknizma olmasaydı biz bunu aaşağıdaki gibi üç aşamada yapmak zorunda kalırdık:

ls -l > temp.txt

wc temp.txt

rm temp.txt

Genel olarak özellikle UNIX/Linux sistemlerinde programlar komut satırı argümanı almamışlarsa okumayı hep stdin dosyasından yaparlar. Bu onların borularla kullanılmasını mümkün hale getirmek için yapılmıştır. Örneğin:

ps -e | grep "sample"

Burada ps -e ile tüm proses listesi satır satır elde edilmek istenmiştir. grep belli bir kalıbı (regex kalıbını) dosya içerisinde bulan ve bulduğu satırın tamamını yazdıran standart bir POSIX komutudur. Böylece yukarıdaki örnekte proses listesinde "sample" geçen satırlar bulunmak istenmiştir.

Biz de programlarımızı boruyla kullanmak istiyorsak komut satırı argümanı verilmemişse stdin dosyasından okuyacak biçimde yazmalıyız. Örneğin:

/\* lcount.c \*/

#include<stdio.h>

#include<stdlib.h>

int main(intargc, char \*argv[])

{

FILE \*f;

int ch;

int count = 0;

if (argc == 1)

f = stdin;

else

if ((f = fopen(argv[1], "r")) == NULL) {

fprintf(stderr, "cannot open file!..\n");

exit(EXIT\_FAILURE);

}

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

if (ch == '\n')

++count;

printf("%d\n", count);

return 0;

}

Burada lcount.c programı hiçbir komut satırı argümanı girilmemişse okumayı stdin dosyasındna yapmaktadır. Şimdi biz bunu aşağıdaki gibi kullanabiliriz:

ps -e | ./lcount

Borular kombine edilebilir. Yani aşağıdaki işlem geçerlidir:

a | b | c

Burada a'nın stdout dosyasına yazdığını b stdin'den okur. b'nin stdout dosyasına yazdıklarını ise c stdin'den okur.

**stdin Dosyalarında EOF Etkisinin Yaratılması**

Yukarıda da açıklandığı gibi aslında biz stdin dosyasından okuma yapmak istediğimizde bu dosya bir aygıt sürücüye yönlendirildiği için terminal aygıt sürücüsünden okuma yapmak isteriz. Bu aygıt sürücü de klavyeden alınanları bize verir. Peki klavye gerçek bir dosya olmadığına göre EOF etkisi nasıl oluşmaktadır. Yani örneğin aşağıdaki döngüde f dosyası stdin'e yönlendirilmişse biz döngüden nasıl çıkarız:

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

...

}

İşte stdin dosyasında EOF etkisi yaratmak için özel tuş kombinasyonları kullanılmaktadır. UNIX/Linux sistemlerinde Ctrl + D, Windows sistemlerinde Ctrl + Z + ENTER tuşları EOF etkisi yaratır. Tabi bu tuş kombinasyonları aslında aygıt sürücüyü kapatmamaktadır. Sadece EOF etkisi yaratır. Biz birden fazla kez bu tuş kombinasyonlarına basabiliriz. (Başka bir deyişle bu tuş kombinasyonlarına bastıktan sonra stdin'den okumaya devam edebiliriz.)

**stdin, stdout ve stderr Dosyalarının Tamponlaması**

stdin, stdout ve stderr dosyaları da normal diğer stdio dosyaları gibi tamponlama mekanizmasına dahildir. Örneğin biz printf fonksiyonuyla stdout dosyasına yazma yapmak istediğimizde printf bunu önce tampona yazabilir. Tampon dolunca aygıt sürücüye aktarabilir. Benzer biçimde biz getchar ile bir karakter bile okuyacak olsak fonksiyon aygıt sürücüden birden fazla karakteri okuyup bir tampona yerleştirir ve bize o tampondakiler bit ene kadar oradan verir. Peki bu standart dosyaların default tamponlama modu nasıldır? Standartlar bu konuda şunları söylemektedir: Eğer stdin ve stdout karşılıklı etkileşimli bir aygıta yönlendirilmişse (terminal karşılıklı bir aygıttır) kesinlikle tam taponlamalı olamaz. Fakat satır ya da sıfır tamponlamalı olabilir. Eğer stdout ve stdin karşılıklı etkileşimli bir aygıta yönlendirilmemişse kesinlikle tam tamponlamalı moddadır. stderr ise başlangıçta hiçbir durumda tam tamponlamalı olamaz. Fakat satır ya da sıfır tamponlamalı olabilir. Tabi bu başlangıçtaki default durumdur. Yoksa daha sonra bu dosyaların tamponlama modları değiştirilebilir (tabi daha önce bir işlem yapılmadan). Bu anlatımdan şu sonuçlar çıkar:

1) stdin ve stdout terminale yönlendirilmişse (tabi bu normal durumdur) başlangıçta default oalrak tam ponlamalı modda olamaz. Fakat sıfır ya da satır tamponlamalı modda olabilir. Bu tamamen derleyiciyi yazanların isteğine kalmıştır. Örneğin Linux libc kütüphanesinde stdout dosyasının default tamponlama modu satır tamponlamalıdır, fakat Windows'taki Microsoft derleyicilerinde sıfır tamponlamalıdır. Örneğin aşağıdaki kodu çalıştıracak olalım:

#include<stdio.h>

int main(void)

{

printf("ankara");

for (;;)

;

return 0;

}

Burada Windows'ta "anakara" yazısı görülecektir fakat Linux'ta görülmeyecektir. Tabi program aşağıdaki gibi olsaydı her iki sistemde de yazı görülecekti.

#include<stdio.h>

int main(void)

{

printf("ankara\n");

for (;;)

;

return 0;

}

Tabi imleç aşağı satırın başına da geçecektir. Fakat bu istenmiyorsa yazının gözükmesini garanti etmek için fflush fonksiyonu da kullanılabilir:

#include<stdio.h>

int main(void)

{

printf("ankara");

fflush(stdout);

for (;;)

;

return 0;

}

Ya da stdout dosyasının tamponlama modunu sıfır tamponlamalı moda çekebiliriz:

#include<stdio.h>

int main(void)

{

setvbuf(stdout, NULL, \_IONBF, 0);

printf("ankara");

for (;;)

;

return 0;

}

2) Biz stdin ve stdout dosyalarını normal disk dosyasına yönlendirirsek kesinlikle tam tamponlamalı mod kullanılır. Örneğin programı aşağıdaki çalıştırmış olalım:

./sample > x.txt

Burada '\n' karakterini görünce değil tampon doulunca dosyaya yazma yapılır.

3) stderr dosyası ister karşılıklı etkileşimli aygıta yönlendirilmiş olsun isterse olmasın hiçbir zaman işin başında tam tamponlamalı modda olamaz. Yani biz stderr dosyasına yazma yaparken yazının sonuna '\n' karakterini eklersek her zaman yazı aygıta transfer edilecektir.

Peki Windows ve Linux'ta stdin dosyasının default tamponlama modu nasıldır? sistemlerin hemen hepsinde eğer stdin terminale yönlendirilmişse (yani kalvyeyi temsil ediyorsa) bu dosyanın default tamponlaması satır tamponlamalı moddur. Yani biz getchat bir fonksiyonla stdin'den bir karakter okumak istediğimizde getchar aygıt sürücüden bir satır talep eder. Biz de ENTER tuşuna basana kadar pek çok karakter girebiliriz. Bastığımız ENTER '\n' anlamına gelir. Bu '\n' karakteriyle birlikte klavyeden girilen tüm karakterler stdin dosyasının tamponuna çekilir. getchar bize onun ilkini verir. Sonra getchar çağırmaya devam edersek fonksiyon bize tampondakileri sırasıyla verecektir. En son getchar bize tampondaki '\n' karakterini verir. Artık tampon boşalmıştır. Bir daha getchar çağırırsak o yine tamponu bir satırla doldurmak ister. Örneğin aşağıdaki programda yalnızca 'a' tuşuna ve sonra ENTER tuşuna basmış olalım. Ne olacaktır?

#include<stdio.h>

int main(void)

{

int ch1, ch2;

ch1 = getchar();

ch2 = getchar();

printf("ch1 = %d, ch2 = %d\n", ch1, ch2);

return 0;

}

Burada tamponda "a\n" karakterleri bulunur. Birinci getchar tampondan a'yı alır. İkinci getchar tamponda bilgi olduğundan klavyeden giriş beklemez. O da '\n' yi alır. Bir daha getchar kullansaydık yeniden giriş istenecekti. Peki aşağıdaki örnekte program nasıl çalışır?

#include<stdio.h>

int main(void)

{

int ch;

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

putchar(ch);

return 0;

}

Burada birinci getchar dolayısıyla bizden bir satır istenir. Arık diğer getchar'lar girdiğimiz yazının karakterlerini alarak yazdırır. Tabi tamponun sonundaki '\n' de yazdırılır. Bu sırada imleç aşağıdaki satırın başına geçmiş olur. Sonraki getchar yine bizden bir satır isteyecektir. Döngüden çıkmak için EOF tuş kombinasyonları kullanılır.

**stdin Dosyasını Kullanan Standart C Fonksiyonlarının Davranışı**

Yukarıda zaten getchar fonksiyonu açıklandı. Burada diğer fonksiyonlar üzerinde duracağız.

**gets Fonksiyonu:** Fonksiyonun prototipi şöyledir:

char \*gets(char \*str);

Fonksiyon '\n' görene kadar ('\n' dahil olmak üzere) ya da EOF görene kadar stdin dosyasından karakterleri okur ve parametresiyle aldığı diziye yerleştirir. Fakat '\n' için '\0' karakterini diziye yerleştirir. gets normal durumda parametresiyle aldığı adresin aynısına geri döner. Fakat stdin dosyasından hiç okuma yapamadan EOF görürse NULL adrese geri döner. Örneğin:

#include<stdio.h>

int main(void)

{

int ch;

char s[100];

ch = getchar();

gets(s);

printf("%c-%s\n", ch, s);

return 0;

}

Burada getchar ile biz "anakara" yazısını girip ENTER tuşuna basmış olalım. Girdiğimiz yazının ilk karakterini getchar alır, diğerlerini gets alır. gets tamponun sonundaki '\n' yi de okur fakat onun yerine diziye '\0' yerleştirir. Peki çağrılar ters sırada olsaydı ne olurdu?

#include<stdio.h>

int main(void)

{

int ch;

char s[100];

gets(s);

ch = getchar();

printf("%c-%s\n", ch, s);

return 0;

}

Burada gets tüm tamponun bir satırla dolmasına yol açar fakat onların hepsini okur. Dolayısıyla getchar tamponu boş göreceğinden bizden yeniden giriş istenecektir. Aşağıdaki programı stdin dosyasını bir dosyaya yönlendirerek test ediniz:

#include<stdio.h>

int main(void)

{

char buf[4096];

while (gets(buf) != NULL)

puts(buf);

return 0;

}

Burada dosyanın sonuna gelindiğinde artok gets hiçbir karakter okuyamadan EOF ile karşılaşır ve NULL adrese geri döner. Böylece döngüden çıkılmış olur.

**scanf Fonksiyonu**

scanf fonksiyonu formatlı okuma yapmak için kullanılmaktadır. Format karakterleri girdinin biçimini ve türünü belirtir. Fonksiyonun prototipi şöyledir:

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

Fonksiyon yerleştirme yapılan nesne sayısına geri döner. Örneğin:

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

scanf önce boşluk karakterlerini atar sonra okumayı yaparak nesneye yerleştirir. Yani scanf baştaki boşluk karakterlerini (leading space) okur fakat sondakileri (trailing space) okumaz, sondakiler stdin tamponunda kalır. scanf stdin dosyasından bilgileri karakter karakter okur, okuduğu karakterin format karakterine uygun olmadığını gördüğünde onu stdin tamponuna geri bırakır (ungetc standard C fonksiyonuna bakınız) ve işlemini sonlandırır. Örneğin:

#include<stdio.h>

int main(void)

{

int a;

int result;

char s[100];

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

gets(s);

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

return 0;

}

Burada scanf için " ali " girişini yapmış olalım. Aşağıdaki gibi bir sonuç elde edilmiştir

a = -858993460, result = 0, s = "ali "

a'ya yerleştirilme yapılmadığı için çöp değer vardır. scanf sıfır ile geri dönmüştür. gets ise tampondan geri kalanların hepsini okumuştur. Örneğin:

#include<stdio.h>

int main(void)

{

int a, b;

int result;

char s[100];

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

gets(s);

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

return 0;

}

Burada klavyeden " 100 ali " girmiş olalım. Şöyle bir sonuç elde edilmiştir:

a = 100, b = -858993460, result = 1, s = "ali "

Görüldüğü gibi a için başarılı yerleştirme yapılmıştır. b için scanf işlemi başlattığında önce ali'nin solundaki boşluk karakterlerini atmıştır. Fakat a'yı beğenmemiş veonu yeniden stdin tamponuna bırakmıştır. gets de geri kalanları almıştır.

scanf fonksiyonunda format karakterleri arasında başka bir karakter konulursa scanf okuma sırasında bu karakterlerin o pozisyonda bulunmasını ister. Örneğin:

#include<stdio.h>

int main(void)

{

int a, b;

int result;

char s[100];

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

gets(s);

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

return 0;

}

Girişin "100 200" biçiminde yapıldığını düşünelim. scanf 100'den sonra ',' karakteri beklemektedir. Bu gelmediği için oradan okuduğu boşluk karakterini tampona geri bırakır ve işlemini sonlandırır. Şöyle bir çıkış elde edilmiştir:

a = 100, b = -858993460, result = 1, s = " 200"

Format karakterleri arasında hiç boşluk kujllanılmazsa bu durum "önceki boşluk karakterlerinin (leading space) atılacağı anlamına" gelir. Fakat araya bir başka bir karakter getirilirse artık boluklar atılmaz kesinlikle o karakter beklenir. Aşağıdaki iki çağrı arasındaki farkı inceleyiniz:

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

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

Örneğin:

#include<stdio.h>

int main(void)

{

int day, month, year;

scanf("%d/%d/%d", &day, &month, &year);

printf("%d/%d/%d\n", day, month, year);

return 0;

}

Format kısmında tek boşluk "bir ya da birden fazla boşluğu at" anlamına gelir. Örneğin:

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

Burada ilk girişten sonraki boşluk karakterlerinin hepsi atılacaktır. Bunun boşluksuz durumdan işlevsel farklılığı yoktur. Aşağıdaki bir çağrıda ilginç bir durum oluşujr:

scanf("%d%d\n", &a, &b);

Burada formak karakterlerinden sonra bir boşluk karkateri (white space) görüldüğü için scanf bu noktada tüm boşluk karakterlerini atmak ister. Böylece biz boşluk girdiğimiz sürece bizden karkter istemeye devam eder. Yani boşluk karakteri girmeyene kadar scanf bizden karakter isteyecektir.

scanf boşluk karakterlerini ayıraç kullandığı için %s ile okuma yaparken ilk boşluk karkaterini gördüğünde okumayı tamamlar. Yani biz %s ile boşluklu bir yazı okuyamayız.

scanf eğer boşluk karakterlerini attıktan sonra (ya da atmadan önce) dosya sonuyla karşılaşırsa EOF değerine (-1) geri döner.

Aşağıdaki gibi bir a.txt dosyası olsun. Dosyanın içerisinde boşluklarla ayrılmış pek çok sayı vardır:

a.txt

------

100 200

300

400

500 600

700 800

900

100

Şimdi biz bu değerleri "IO redirection" ile okumak isteyelim. Komut satırında programı şöyle çalıştırırız:

./sample < a.txt

Aşağıdaki program bunu yapabilir:

#include<stdio.h>

int main(void)

{

int val;

while (scanf("%d", &val) != EOF)

printf("%d\n", val);

return 0;

}

**Anahtar Notlar:** C'nin stdin'den okuma yapan fonksiyonlarının hepsi yalnızca getchar (yani fgetc(stdin)) ve ungetc ile yazılabilir. Benzer biçimde dosyadan formatlı okujma yapan fonksiyonlar da yine fgetc ve ungetc kullanılarak yazılabilir. Örneğin gets fonksiyonunu şöyle yazabiliriz:

char \*mygets(char \*s)

{

int ch;

size\_t i;

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

s[i] = ch;

if (i == 0)

returnNULL;

s[i] = '\0';

returns;

}

**Anahtar Notlar:** gets fonksiyonunun tasarımında bir kusur vardır. Bu fonksiyonda her zaman kullanıcı daha fazla giriş yaparsa dizi taşması söz konusu olabilir. gets fonksiyonu C99-TC3'te "deprecated" yapılmıştır ve C11'de tamamen kaldırılmıştır. Bunun yerine gets\_s getirilşmiştir. Bu fonksiyon dizinin uzunluğunu da parametre olarak alır dolayısıyla dizi taşması engellenmiş olur. gets\_s şöyle yazılabilir:

char \*mygets\_s(char \*s, size\_tn)

{

int ch;

size\_t i;

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

s[i] = ch;

if (i == 0)

returnNULL;

s[i] = '\0';

returns;

}

Eskiden gets\_s yaygın olarak bir eklenti biçiminde bulunmuyordu. O günlerde gets yerine daha güvenlikli fgets kullanılabiliyordu. (Ne de olsa fgets'in uzunluk parametresi vardır.) Ancak fgets kullanılırken bu fonksiyonun '\n' karakterini de okuduğun unutulmamalıdır. Dizideki '\n' karakterinin silinmesi gerekir. Örneğin:

#include<stdio.h>

#include<string.h>

int main(void)

{

char s[20];

char \*str;

fgets(s, 20, stdin);

if ((str = strchr(s, '\n')) != NULL)

\*str = '\0';

puts(s);

return 0;

}

**stdin Tamponunun Boşaltılması**

Bazen tamponda kalanları değil kullanıcının yeni girişini okumak isteyebiliriz. Öneğin iki getcharf fonksiyonunu peş peeşe kullanacak olalım. İkinci getchar ile yeni bir giriş isteyebiliriz. Ya da gets ile scanf birlikte kullanılırken de benzer bir istek oluşabilir. Örneğin:

#include<stdio.h>

int main(void)

{

int no;

char name[64];

printf("No:");

scanf("%d", &no);

printf("Name:");

gets(name);

printf("No = %d, Name = %s\n", no, name);

return 0;

}

Maalesef C'de standart olarak stdin tamponunu boşaltan bir fonksiyon yoktur. Bazı derleyicilerde fflush(stdin) bunu yapıyor olmakla birlikte, bu kullanım uygıunsuzdur. Çünkü her şeyden önce fflush read-only dosyalarda kullanılamaz ve stdin read only bir dosyadır. Tamponu boşaltmak için küçük bir fonksiyon ya da makro yazılabilir:

#include<stdio.h>

void empty\_stdin(void)

{

int ch;

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

;

}

int main(void)

{

int no;

char name[64];

printf("No:");

scanf("%d", &no);

empty\_stdin();

printf("Name:");

gets(name);

printf("No = %d, Name = %s\n", no, name);

return 0;

}

Tamponu boşaltırken EOF kontrolü de yapılmıştır. Aksi taktirde yönlendirme durumunda sonsuz döngü oluşabilir. Aynı fonksiyonu makro olarak dayazabiliriz:

#defineempty\_stdin() \

do { \

int ch; \

\

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

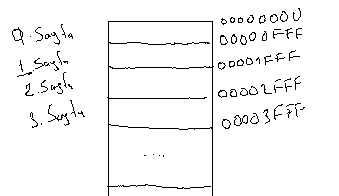
; \

} \

while (0)

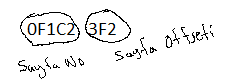
**İşlemcilerin Sayfalama Mekanizmaları (Paging Mechanisms)**

Modern ve güçlü masaüstü işlemcileri sayfalama (paging) denilen bir mekanizmaya sahiptir. Sayfalama mekanizmasında bellek sayfa (page) denilen ardışıl byte bloklarına ayrılmıştır. Sayfaların büyüklükleri işlemciye bağlıdır. Pek çok işlemci 4K uzunluğunda sayfa kullanmaktadır. Bellekte her bir sayfaya 0'dan başlayark bir numara verilmiştir. Örneğin 4K'lın sayfa kullanan sistemde fiziksel belleğin haritalanması şöyledir:

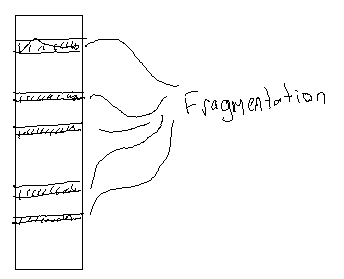


32 bit bir sistemde adresler 4 byte yer kaplar (8 hex digit). Her fiziksel adres aslında bir sayfa içerisindedir ve o sayfada belli bir offset'tedir. Adresin hangi sayfada olduğunu bulmak için onu sayfa uzunluğuna bölmek gerekir. (Tabi 2'lik sistemde bir sayıyı 4096'ya bölmek demek 12 kez sağa kaydırmak, başka bir deyişle sağdaki 12 biti atmak demektir.) Adresin o sayfanın hangi offset'inde olduğunu bulmak için sayfa uzunluğuna bölümünden elde edilen kalanı bulmak gerekir (örneğin sayının 4096'ya bölümünden elde edilen kalan düşük anlamlı 12 bitidir.) Örneğin:

0F1C23F2 adresi hangi sayfa ve offset'tedir?



**Anahtar Notlar**: Bellek sistemlerinde en önemli olgulardan biri bölünme (fragmentation) durumudur. Bölünme arşılık yerleştirilme zorunluluğunun yol açtığı bir sonuçtur. Eğer söz konusu belleğe (disk de olabilir RAM de olabilir) birtakım öğeler ardışıl yerleştiriliyorsa zamanla bunların yaratılıp yok edilmesi sonucunda küçük fakat çok sayıda alan oluşur. Bu alanlar küçük olduğu için pek bir işe yaramazlar. Fakat bellek kullanım oranını ciddi biçimde düşürürler. Bu olguya dışsal bölünme (external fragmentation) denilmektedir. Örneğin dışsal bölünmüş bir bellek şöyle görünümde olabilir (taranmış alanlar boş alanlar):



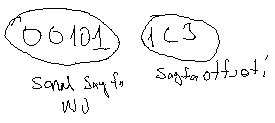
Peki bir program fiziksel belleğe ardışıl yerleştirilmiyorsa nasıl çalışmaktadır? İşte program içerisindeki adreslere "doğrusal adres (linear address), mantıksal adres (logical address) ya da sanal adres (virtual address)" denilmektedir. Biz burada "sanal adres" terimini kullanacağız. İşlemci bu sanal adresleri "sayfa tablosu (page table)" denilen bir tabloya bakarak fiziksel adreslere dönüştürmektedir. İşlemci sayfa tablosunu belli bir yazmacın (register) gösterdiği yerde arar. (Örneğin Intel işlemcilerinde CR3 yazmacı sayfa tablosunun yerini belirtmektedir.) Prosesin sayfa tablosunu işletim sistemi oluşturur ve bu yazmaca onun adresini verir. Böylece işlemci işletim sistemi tarafından oluşturulmuş olan sayfa tablosunu kullanır hale gelmektedir. Peki sayfa tablosu nasıldır? Bazı sistemlerde iki kademeli hatta üç kademeli sayfa tabloları kullanılmaktadır. Örneğin Intel'de iki kademeli sayfa tablosu kullanılır. Ancak biz burada kavramsal karmaşıklık oluşmasın diye tek kademeli bir sayfa tablosunu örnek vereceğiz. Sayfa tablosunun tipik biçimi şöyledir:

|  |  |  |
| --- | --- | --- |
| Sanal Sayfa No  (Aslında Tabloda Yok) | Fiziksel Sayfa No | Sayfa Özellikleri |
| ... | ... | ... |
| 00100 | 01FC3 | RW, P, User, D |
| 00101 | 2FC40 | RW, P, User, D |
| 00102 | 1C167 | RW, P, User, D |
| ... | ... | .... |

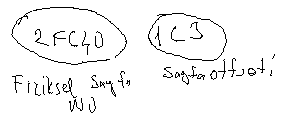
Tablonun ilk sütunu aslında bir indeks numarasıdır. Yani tabloda yoktur. Şöyle ki aslında 00100 sanal sayfasına ilişkin giriş tabloda tablonun başından itibaren bir sayfa girişinin uzunlu ile 0x100'ün çarpımı kadar uzaklıktadır. Tablodaki her bir satıra "sayfa tablosu girişi (page table entry)" denilmektedir. İşlemci bir sanal adresle karşılaştığında önce o sanal adresi sanal sayfa numarasına ve sayfa offsetine ayrıştırır. Sonra sanal sayfa numarasını sayfa tablousna indeks yapar ve oradan fiziksel sayfa numarasını elde eder. Ona da sayfa offset'ini toplar ve nihai fiziksel adresi elde eder. Örneğin aşağıdakiş gibi bir kod olsun:

MOV EAX, [001011C3]

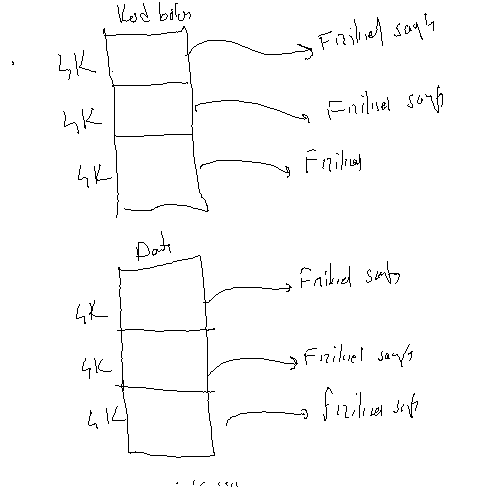
İşlemci bunu aşağıdaki gibi sanal syafa numarasına ve sayfa offset'ine ayırır:



bundan sonra işlemci sayfa tablosuna bakarak 0x101 numaralı sanal sayfa numarasının hangi fiziksel sayfa ile eşleştirildiğine bakar. Yukarıdaki örnekte 0x101 numaralı sanal sayfa 2FC40 fiziksel sayfasıyla eşleştirilmiştir. Bu adresin 4096 ile çarpımına 1C3 eklenir ve şu değer bulunur:



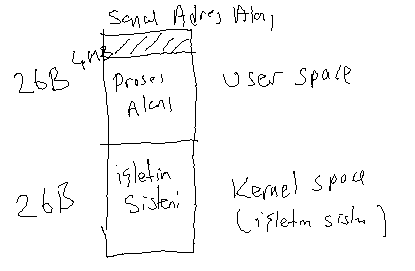
Böylece erişimi fiziksel bellekteki 2FC401C3 adresinden yapar. Böylece programın hepsi aslında 4k'lık fiziksel sayfalara atanır:



Derlenmiş bir kodda derleyicinin üretiiği adreslerin hepsi sanal adreslerdir. Yani derleyici 32 bit bir sistemde sanki program 4GB'lik fiziksel belleğe tek başına yüklenecekmiş gibi kod üretimi yapar. Sanki bütün programlar aynı adresten itibaren fiziksel bellekte tek başlarına çalışacakmış gibi bir koda sahiptir. İşletim sistemi programı yükleyeceği zaman onu sayfalara dağıtır ve sayfa tablosunu uygun biçimde düzenler. Böylece proseslerin sanal adres alanları aynı olmasına karşın onlar farklı fiziksel adreslerde bulunurlar. Başka bir deyişle örneğin Windows'ta ve Linux'ta farklı proseslerdeki aynı sanal adresler aslında fiziksel bellekte aynı yere tekabül etmezler. Aslında bu tür sistemlerde fiziksel belleğin bir önemi yoktur.

Her prosesin ayrı bir sayfa tablosu olduğunu vurgulayalım. İşletim sistemi her prosesler arası geçiş oluştuğunda otomatik olarak işlemcinin kullandığı sayfa tablosunu da değiştirir.

Sayfalama mekanizmasını kullanan işletim sistemlerinde proseslerin bellek alanları tam olarak izole edilmiştir. İşletim sistemi her prosesin sayfa tablosunda onların sanal sayfalarını farklı fiziksel sayfalara yönlendirdiği için istense de bir proses başka bir prosesin alanına erişemez. Tabi işletim sistemi her zaman bellektedir. Örneğin 32 Windows sistemlerinde tipik sanal bellek alanının kullanımı şöyledir:



Tipik olarak Windows sistemlerinde derleyicinin ürettiği kod sanki program fiziksel belleğin 4MB'tan başlayan kısmına yüklenecekmiş gibi oluşturulur. Bu adres .exe dosyanın içerisine yazılır. Ve istenirse Linker ayarlarıyla değiştirilebilmektedir. Yani işletim sistemi bütün proseslerin sayfa tablosunda aynı yerde bulunmaktadır.

Sayfa tablosu girişlerindeki sayfa özellikleri ilgili sayfaya erişim haklarını ve başka birtakım bilgileri içerir. Örneğin bir sayfa read-only ise işlemci oraya yazmaya çalıştığında exception oluşur. İşletim sistemi devreye girer ve prosesi cezalandırarak sonlandırır. Örneğn C'de Microsoft derleyicileri string'leri read-only sayfalara yerleştirmektedir. Böylece bir string update edilmek istendiğinde exception oluşur.

#include<stdio.h>

int main(void)

{

char \*s = "ankara";

\*s = 'x'; /\* read-only sayfaya erişim! \*/

return 0;

}

Pekiyi bu sistemlerde proseslerin bellek alanları izole edilmiştir. Peki bir proses işletim sisteminin alanına erişemez mi? İşte işletim sistemi kendini başka bir mekanizmayla korumaktadır. Her fiziksel sayfa "kernel" ya da "user" biçiminde önceliklendirilmiştir. User sayfalarına ancak "user mod" prosesler erişebilir. "Kernel mod" prosesler her türlü sayfaya erişebilir. Bu durumda biz kendi programımızda işletim sisteminin sayfalarına erişirsek bunlar mod sayfalar olduğu için yine exception oluşur ve prosesimiz sonlandırılır. Sayfanın "user mod" sayfa mı "kernel mod" sayfa mı olduğu yine sayfa tablousu girişinde "ayfa özellikleri" kısmında belirtilmektdir.

Sayfalama mekanizmasını kullana işletim sistemleri bellek yönetiminde şu izlemeleri yapabilmeli ve kararkları verebilmelidir:

- Fiziksel bellekteki sayfaların boş mu dolu mu olduğunun takibi

- Sayfa tablolarının oluşturulması ve programın fiziksel sayfalara atanması

**Sanal Bellek Mekanizması (Virtual Memory)**

Sanal bellek bir programın tamamanın değil bir kısmının fiziksel belleğe yüklenerek disk (ikincil bellekle) ile fiziksel bellek arasında yer değiştirmeli olarak çalıştırılmasına denir. Sanal bellek mekanizmasında bir programın yalnızca belirli kısmı fiziksel belleğe yüklenerek program çalıştırılır. Sonra program çalışırken kod ya da data bakımından fiziksel bellekte olmayan bir kısma erişildiğinde işlemci bunu anlar ve bir içsel kesme oluşturur. Bu içsel kesmeye Intel terminolojisinde "page fault" denilmektedir. Böylece işletim sistemi devreye girer. Prosesin bellekte olmayan sayfasını diskten bulur, onu fiziksel boş bir sayfaya yükler, prosesin sayfa tablosunu günceller ve kesmeden çıkar. Intel terminolojisinde "fault" kesmelerin çılıkdığında çalışma aynı makina komutuyla devam etmektedir. Böylece işletim sistemi işini bitirdiğinde yine akış "fault"a yol açan komutla devam edecektir. Artık ilgili sanal sayfaya sayfa tablosunda bir fiziksel sayfa karşılık getirildiği için fault oluşmaz. Bir sanal sayfanın o anda fiziksel bellekte olup olmadığı sayfa özelliklerindeki P (Present) biti ile anlaşılmaktadır.

Sanal bellek sayesinde bir programın küçük bir kısmı fiziksel belleğe yüklenerek program çalışır. Sanal bellek gerçekleştirimi adım adım şöyle yapılmaktadır:

1) Program kodunda bir sanal adrese erişilmiştir. İşlemci sayfa tablosuna bakar ve bunu fiziksel adrese dönüştürmeye çalışır. eğer sayfanın P biri 0 ise işlemci bunun fiziksel bellekte olmadığına karar verir ve "page fault" oluşturu.

2) İşletim sistemi devreye girer ve prosesin erişilmek istenen sanal sayfasının diskte nerede olduğunu bulur. Onu fiziksel RAM'e yüklemek ister. Tabi bunun için boş bir fiziksel sayfa bulmaya çalışır. Eğer bulursa disktenm o sayfayı o fiziksel sayfaya yükler . Bu sürece İnglizce "Swap-In" denilmektedir. Peki ya fiziksel bellek tıka basa doluysa? Bu durumda işletim sistem, fiziksel bellekten başka bir prosese ait olan bir sayfayı çıkarmak ister. Onu o haliyle yeniden diske yazazacaktır. Bu işleme ise İngilizce "Swap-Out" denilmektedir. Tabi burada işletim sistemi gelecekte en az kullanılacak bir sayfayı fiziksel RAM'den çıkarmak isteyecektir. Tıka basa dolu bir RAM'de her Swap-In işleminde bir Swap-Out yapılması sistemi yavaşlatmaktadır. Bu probleme İngilizce "thrashing" denilmektedir.

3) İşletim sistemi Swap-In yaptıktan sonra prosesin sayfa tablosunu düzeltir ve fault'tan çıkar. Böylece akış fault'a yol açan makina komutuyla devam edecektir. Tabi artık fault oluşmayacaktır.

Yukarıdaki işlemin bazı ayrıntıları vardır. Örneğin "Swap-In" yapıcacağı zaman programın ilgili kısmı diskte nereden alınacaktır? Bu konuda işletims sitemlerinin değişik stratejileri vardır. Bazı işletim sistemleri her executable dosyanın kendisini hem de bir swap dosyasını bu amaçla kullanır (örneğin Windows). Bazı sistemlerde executable dosya hiç kullanılmaz ve birden fazla swap dosyası kullanılır. Bazı sistemler swap dosyası yerine ayrı bir disk bölümünü (partition) kullanılmaktadır. Peki neden executable dosya bunun için yeterli olmamaktadır? Eğer bir fiziksel sayfa üzerinde değişiklik yapılmışsa onu executable dosyaya yazmak mümkün değildir. Fakat read-only sayfaları her zaman oradan alabiliriz. Swap-out yapılacağı zaman sayfanın diske geri yazılması gerekir mi? İşte sayfa tablo girişlerinde bir D biti de bulunmaktadır. İşlemci ne zaman bir sayfaya yazma yapsa bu D bitini set eder. İşletim sistemi de Swap-out yapacağı zaman bu D bitine bakar. Bu D biti 0 ise boşuna sayfayı swap dosyasına yazmaz.

Peki proses hiç tahsis etmediği bir alana erişmek isterse ne olur? Çünkü alanın diskte de bir karşılığı yoktur. İşte işletim sistemi page fault oluştuğunda önce fault'a yol açan sanal adresin gerçekten legal bir adres olup olmadığına bakmaktadır. Yani erişlmek istenen adres tahsis edilmiş bir adres değilse prosesi hemen sonlandırır.

Peki aynı program ikinci kez yüklendiğinde yeniden onun bazı sayfaları fiziksel belleğe yüklenir mi? İşte işletim sistemi değişmemiş ve değişmeyecek sayfaları yeniden fiziksel belleğe yüklememektedir. Yani örneğin biz iki kez notepad.exe programını çalıştırsak programın kod bölümü zaten hiç update edilmeyeceği için iki farklı notepad prosesinin sayfa tablosunda bazı girişler aynı fiziksel adrese yönlendirilmektedir. Böylece aynı programı biz iki çalıştırdığımızda aslında tekrar her şey fiziksel belleği yüklenmez.

İşletim sistemi programın ne kadarını fiziksel belleğe yüklemektedir? Prosesin fiziksel bellekteki kısmına İnglizce "Resident Set" ya da "Working Set" denilmektedir. Bu da işletim sistemlerini yazanların kullandıkları algoritmaya bağlı olarak değişmektedir. İşletims sitemlerinin en zor gerçekleştirilen alt sistemlerinden biri "bellek yöneticisi (memory management)" bölümüdür.

**Proseslerin Yaratılması**

Bir prosesi yaratmak için işletim sistemlerinde sistem fonksiyonları bulunur. Windows sistemlerinde CreateProcess isimli API fonksiyonu, UNIX/Linux sistemlerinde fork fonksiyonu proses yaratmak için kullanılmaktadır.

Windows'ta CreateProcess API fonksiyonunun prototipi şöyledir:

BOOL WINAPI CreateProcess(

\_\_in LPCTSTR lpApplicationName,

\_\_in\_out LPTSTR lpCommandLine,

\_\_in LPSECURITY\_ATTRIBUTES lpProcessAttributes,

\_\_in LPSECURITY\_ATTRIBUTES lpThreadAttributes,

\_\_in BOOL bInheritHandles,

\_\_in DWORD dwCreationFlags,

\_\_in LPVOID lpEnvironment,

\_\_in LPCTSTR lpCurrentDirectory,

\_\_in LPSTARTUPINFO lpStartupInfo,

\_\_out LPPROCESS\_INFORMATION lpProcessInformation

);

Fonksiyonun birinci parametresi çalıştırılacak program dosyasının yol ifadesini alır. Burada yol ifadesi mutlak ya da göreli olarak verilebilir. Fonskiyonun ikinci parametresi programın komut satırı argümanlarını belirtir. Komut satırı argümanları tek bir yazı olarak fonksiyona verilmektedir. Sonra C derleyicilerinin başlangıç kodları (start up code) bu yazıyı boşluklarda parse ederek argv dizisini oluşturmaktadır. İlk komut satırı argümanının programın yol ifadesi olması C'de zorunlu tutulmuştur. Fakat işletim sistemi genelinde böyle bir zorunluluk yoktur. Kolaylık olsun diye şöyle bir seçenek de sunulmuştur: Eğer birinci parametre NULL geçilirse ikinci parametredeki ilk boşluksuz kısım sanki birinci parametredeki çalıştırılabilen dosyanın yol ifadesi gibi ele alınmaktadır. Dosya ismini bu yolla vermenin diğerinde bir farkı daha vardır. Bu yöntemde eğer dosya ismi hiç '\' karakteri içermiyorsa Windows onu sırasıyla bazı dizinlerde arar. eğer buradaki dosya ismi '\' karaketeri içeriyorsa Windows onu yol ifadesi ile belirtilen yerde arar fakat başka bir dizine bakmaz. Yine fonksiyonun birinci parametresi NULL geçilmezse Windows programı başka yerde aramamaktadır. Burada ikinci parametrenin const olmayan bir adres olduğuna dikkat ediniz. (\_\_in\_\_out). Biz ikinci parametreye bir string ifadesi vermemeliyiz. Çünkü CreateProcess bu adresteki bilgiyi saklayıp burayı tampon olarak kullanarak fonksiyon çıkışında yeniden orijinal yazıyı burada bırakır. Ama bu diziyi güncellemektedir. Halbuki string ifadeleri güncellenemez. Eğer birinci parametre NULL geçeilirse ve ikinci parametredeki dosya ismi '\' içermiyorsa Windows dosyayı sırasıyla şu dizinlerde arar:

1) CreateProcess fonksiyonunu uygulayan programın program dosyasının bulunduğu dizin

2) CreateProcess uygulayan prossesin o andaki çalışma dizini

3) 32 Bit Windows System dizini (tipik olarak c:\windows\system32 dizini)

4) 16 bit Windows dizini (tipik olarak c:\windows\system)

5) Windows'un kendi dizini (tipik olarak c:\windows)

6) CreateProcess fonksiyonunu uygulayan prossesin PATH çevre değişkeni ile belirtilen dizinleri

Fonkısiyonun üçüncü ve dördüncü parametreleri prossese ve prossesin ana thread'ine ilişkin güvenlik parametreleridir. Bu parametreler NULL olarak geçilebilir. Bu durumda default güvenlik durumu anlaşılır. Fonksiyonun beşinci parametresi kernel nesnelerinin alt prosseslere geçirilebilmesine ilişkin ana şalter görevindedir. Bu parametre FALSE olarak geçilebilir. Altıncı parametre yaratılacak prossese ilişkin çeşitli belirlemeleri içermektedir. Bu parametre bazı bayrakların bit OR işlemine sokulmasıyla oluşturulur. Fakat istenirse bu parametre sıfır geçilebilir. Fonksiyonun yedinci parametresi yaratılacak prossesin çevre değişken listesini belirtir. Bu paramere NULL geçilirse yaratılacak prossesin çevre değişkenleri üst prossesten alınır. Fonksiyonun sekizinci parametresi yaratrılacak pross esin çalışma dizinini belirtir. Eğer bu parametre NULL geçilirse prosesin çalışma dizini üst prosesten alınır. Fonksiyonun dokuzuncu parametresi yaratılacak prosese ilişkin bazı ayrıntıların belirlenmesini sağlar. Bu paramtereye STARTUPINFO türünden bir yapının adresi geçirilmelidir. Bu yapının ilk elemanına yapının sizeof'u yazılmalıdır. Diğer elemanlar boş bırakılabilir. Çünkü yapının elemanları sıfır ise bu default durum anlamına gelir. Fonksiyonun son parametresi PROCESS\_INFORMATION türünden bir yapının adresini alır. Fonksiyon bu yapının içini bizim için doldurur. Bu yaıya fonksiyon yaratılacak proses nesnesinin HANDLE ve Id değerlerini, yaratılacak ana thread nesnesinin HANDLE ve id değerlerini yerleştirir. Prosesin HANDLE değeri Proses Kontrol Bloğuna erişmek için sistem tarafından kullanılmaktadır.

Fonksiyon başarı durumunda sıfır dışı herhangi bir değere, başarısızlık durumunda 0 değerine geri döner.

Aşağıda maksimum default değerler geçilerek bir prosesin yaratılmasına örnek verilmiştir:

#include<stdio.h>

#include<stdlib.h>

#include<Windows.h>

void ExitSys(LPCSTR lpszMsg, int status);

int main(void)

{

char cmdLine[] = "c:\\windows\\system32\\notepad.exe";

STARTUPINFO si = { sizeof(STARTUPINFO) };

PROCESS\_INFORMATION pi;

if (!CreateProcess(NULL, cmdLine, NULL, NULL, FALSE, 0, NULL, NULL, &si, &pi))

ExitSys("CreateProcess", EXIT\_FAILURE);

return 0;

}

void ExitSys(LPCSTRlpszMsg, intstatus)

{

DWORD dwLastError = GetLastError();

LPTSTR lpszErr;

if (FormatMessage(FORMAT\_MESSAGE\_ALLOCATE\_BUFFER | FORMAT\_MESSAGE\_FROM\_SYSTEM, NULL, dwLastError,

MAKELANGID(LANG\_NEUTRAL, SUBLANG\_DEFAULT), (LPTSTR)&lpszErr, 0, NULL)) {

fprintf(stderr, "%s: %s", lpszMsg, lpszErr);

LocalFree(lpszErr);

}

exit(status);

}

**Anahtar Notlar:** Windows API fonksiyonları hem ASCII hem de UNICODE karakter tablosuyla çalışabilmektedir. Visual Studio'da default durumda karakter tablosu UNICODE biçimdedir. Yukarıdaki örnekte bunun ASCII yapılması gerekir. Bunun için proje seçeneklerine gelinmeli "Caharacter Set" seçeneği "Not Set" yapılmalıdır.

**Sınıf Çalışması**: Basit bir C programı yazınız. Programı derlemek için cl.exe'yi aşağıdaki gibi çalıştırınız:

cl.exe test.c

Visual Studio 2013 için cl.exe şuarada bulunmaktadır:

C:\Program Files (x86)\Microsoft Visual Studio 12.0\VC\bin\cl.exe

**Çözüm:**

#include<stdio.h>

#include<stdlib.h>

#include<Windows.h>

#include<winapifamily.h>

void ExitSys(LPCSTR lpszMsg, int status);

int main(void)

{

char cmdLine[] = "\"C:\\Program Files (x86)\\Microsoft Visual Studio 12.0\\VC\\bin\\cl.exe\" "

/\*

"-I \"C:\\Program Files (x86)\\Microsoft Visual Studio 12.0\\VC\\include\" "

"-I \"c:\\Program Files (x86)\\Windows Kits\\8.1\\Include\\um\" "

"-I \"c:\\Program Files (x86)\\Windows Kits\\8.1\\Include\\shared\" "

\*/

" sample.c";

STARTUPINFO si = { sizeof(STARTUPINFO) };

PROCESS\_INFORMATION pi;

if (!CreateProcess(NULL, cmdLine, NULL, NULL, FALSE, 0, NULL, NULL, &si, &pi))

ExitSys("CreateProcess", EXIT\_FAILURE);

return 0;

}

void ExitSys(LPCSTRlpszMsg, intstatus)

{

DWORD dwLastError = GetLastError();

LPTSTR lpszErr;

if (FormatMessage(FORMAT\_MESSAGE\_ALLOCATE\_BUFFER | FORMAT\_MESSAGE\_FROM\_SYSTEM, NULL, dwLastError,

MAKELANGID(LANG\_NEUTRAL, SUBLANG\_DEFAULT), (LPTSTR)&lpszErr, 0, NULL)) {

fprintf(stderr, "%s: %s", lpszMsg, lpszErr);

LocalFree(lpszErr);

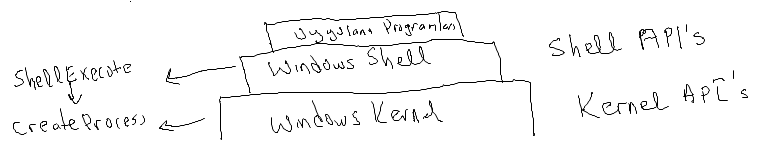
}

exit(status);

}

**ShellExecute Fonksiyonu**

ShellExecute bir sistem fonksiyonu değildir. Zaten kendi içerisinde CrateProcess isimli sistem fonksiyonunu çağırır. ShellExecute bir Shell fonksiyonudur. Windows'un kabuğuna "Windows Explorer" denilmektedir ve proses olarak "explorer.exe" programıdır. Aslında Windows'ta shell tamamen devre dışı bırakılabilir. Tabi bu durumda bu shell fonksiyonlarını kullanamayız. Fakat shell fonksiyonları da Windows'un bir parçası durumundadır.



ShellExecute fonksiyonuna biz çalıştırılmayan bir dosya verebiliriz. ShellExecute "Registry" kayıtlarıne erişerek bu uzantılı dosyanın hangi çalıştırılabilen dosyayla ilişkilendirilmiş olduğunu belirler ve CreateProcess ile o çalıştırılabilen dosyayı çalıştırır. Sonra da bizim verdiğimiz dosyayı ona komut satırı argümanı olarak geçirir.

**Anahtar Notlar:** Bazen Micreosoft bazı API fonksiyonlarını çeşitli nedenlerle geliştirmek istemektedir. Tabi eski fonksiyonların da muhafaza edilmesi gerekir. İşte Microsoft bu tür durumlarda fonksiyonun ilacve yeni bir XXXExli versiyonunu oluşturur (Örneğin CreateWindow, CreateWindowEx gibi.)

Fonksiyonun prototipi şöyledir:

#include <shellapi.h>

HINSTANCE ShellExecute( HWND hwnd,

LPCTSTR lpOperation,

LPCTSTR lpFile,

LPCTSTR lpParameters,

LPCTSTR lpDirectory,

INT nShowCmd

);

Fonksiyonun birinci paraametresi bir GUI penceresinin HANDLE değerini alır. Fonksiyon başarısızlık durumunda bir MessageBox çıkartabildiği için böyle parent pencereye gereksinim duymaktadır. Bu parametre NULL geçilirse masaüstü penceresi anlaşılır. İkinci parametre yapılmak istenen eylemi belirtir. eğer bir program çalıştırılacaksa eylem "open" olmalıdır. Örnek bir ShellWxecute çağrısı şöyle olabilir:

#include<stdio.h>

#include<stdlib.h>

#include<Windows.h>

void ExitSys(LPCSTR lpszMsg, int status);

int main(void)

{

HINSTANCE hInstance;

hInstance = ShellExecute(NULL, "open", "x.txt", NULL, NULL, SW\_SHOWNORMAL);

if ((int)hInstance < 32)

ExitSys("ShellExecute", EXIT\_FAILURE);

return 0;

}

void ExitSys(LPCSTRlpszMsg, intstatus)

{

DWORD dwLastError = GetLastError();

LPTSTR lpszErr;

if (FormatMessage(FORMAT\_MESSAGE\_ALLOCATE\_BUFFER | FORMAT\_MESSAGE\_FROM\_SYSTEM, NULL, dwLastError,

MAKELANGID(LANG\_NEUTRAL, SUBLANG\_DEFAULT), (LPTSTR)&lpszErr, 0, NULL)) {

fprintf(stderr, "%s: %s", lpszMsg, lpszErr);

LocalFree(lpszErr);

}

exit(status);

}

**UNIX/Linux Sistemlerinde Proseslerin Yaratılması**

UNIX/Linux sistemlerinde prosesler fork isimli bir POSIX fonksiyonla yaratılır. fork POSIX fonksiyonu Linux sistemlerinde doğrudan sys\_fork isimli sistem fonksiyonunu çağırmaktadır. Bu sistemlerde proses yaratmanın başka bir yolu da yoktur. (fork fonksiyonunun vfork isimli bir versiyonu da vardır fakat bunun kullanım gerekçesi hepten ortadan kalkmıştır.) Fonksiyonun prototipi şöyledir:

#include <unistd.h>

pid\_t fork(void);

fork bir prosesin özdeş bir kopyasından oluşturur. Yani yaratılan alt proses (child process) için yeni bir kontrol blok yaratılır. Üst prosesin kontrol bloğundaki bilgiler alt prosese kopyalanır. Alt prosesin sanal bellek alanı tamamen üst prosesten (parent process) kopyalanır. Böylece aynı koda ve data sahip özdeş fakat iki farklı iki proses oluşmuş olur.

UNIX/Linux sistemlerinde her prosesin bir proses id ideğeri vardır. Prosesin id değeri pross kontrol bloğuna erişmekte kullanılan bir handle gibi işlem görür. Prosesin id değeri tamsayısal bir değerdir ve pid\_t türü ile temsil edilir.

fork fonksiyonuna bir proses girer (yani üst proses), alt proses fork2un içerisinde yaratılır. Böylece fork'tan iki proses çıkar. Tabi ikisinin de kodu aynıdır. Fakat bağımsız proseslerdir.

fork fonksiyonundan üst proses alt prosesin id değeri ile, alt proses ise sıfır değeri ile çıkar. Böylece kodda üst proses ile alt proses birbirlerinden yarılabilir. fork başarısız da olabilir. Bu durumda -1 değerine geri döner. O halde tipik olarak fork uygulama kalıbı şöyledir:

pid\_t pid;

if ((pid = fork()) == -1) {

perror("fork");

exit(EXIT\_FAILURE);

}

if (pid != 0) { /\* parent process \*/

}

else { /\* child process \*/

}

Örneğin:

#include<stdio.h>

#include<stdlib.h>

#include<unistd.h>

int main(void)

{

pid\_t pid;

if ((pid = fork()) == -1) {

perror("fork");

exit(EXIT\_FAILURE);

}

if (pid != 0) { /\* parent process \*/

printf("parent\n");

}

else { /\* child process \*/

printf("child\n");

}

printf("ends...\n");

return 0;

}

Peki aşağıdaki programda ekrana kaç tane "ends" yazısı çıkar?

#include<stdio.h>

#include<unistd.h>

int main(void)

{

int i;

for (i = 0; i < 3; ++i)

fork();

printf("ends\n");

return 0;

}

Yanıt: 8 tane. Programda ne kadar proses yaratılmışsa o kadar "ends" yazısı çıkar. Burada toplam 8 prose yaratılmaktadır. Yukarıdaki programın eşdeğeri aslında aşağıdaki gibidir:

#include<stdio.h>

#include<unistd.h>

int main(void)

{

int i;

fork();

fork();

fork();

printf("ends\n");

return 0;

}

fork ile biz başka bir progrtamı çalıştıramayız. fork ancak bir prosesin özdeş kopyasını çalıştırır. Peki başka bir program dosyasını nasıl çalıştırırız? İşte bu exec fonksiyonlarıyla yapılmaktadır.

**exec Fonksiyonları**

exec POSIX fonksiyonları bir prosesin başka bir kodla çalışmasına devam etmesini sağlar. exec işlemiyle mevcut prosesin çalıştırdığı kod bellekten atılır. Onun yerine exec fonksiyonunda belirtilen dosya belleğe yüklenir ve o dosyadaki kod çalıştırılır. exec işlemleriyle prosesin kontrol bloğu değişmez. Yani prosesin id'si, yetkileri, çalışma dizini, açtığı dosyalar vs. hep öyle kalır. Proses hayatına başka bir program koduyla devam eder.

Aslında exec isimli bir fonksiyon yoktur. exec bir ailesnin ismidir. Bu ailede yoplam 7 fonksiyon vardır. Bu fonksiyonların yaptıkları şey aynı olmasına karşın yalnızca parametrik yapı yani arayüz farklıdır. Bu 7 fonksiyon şöyledir:

#include <unistd.h>

int execl(const char \*path, const char \*arg, ...);

int execlp(const char \*file, const char \*arg, ...);

int execle(const char \*path, const char \*arg, ..., char \* const envp[]);

int execv(const char \*path, char \*const argv[]);

int execvp(const char \*file, char \*const argv[]);

int execvpe(const char \*file, char \*const argv[], char \*const envp[]);

int execve(const char \*filename, char \*const argv[], char \*const envp[]);

Aslında asıl taban fonksyion execve fonksiyonudur. Yani yalnızca execve bir sistem fonksiyonudur. Diğer fonksiyonlar execve fonksiyonunu çağırmaktadır.

En çok kullanılan exec fonksiyonlarından biri execl fonksiyonudur. (Buradaki l list'ten geliyor.)

#include <unistd.h>

int execl(const char \*path, const char \*arg, ...);

Fonksiyonun birinci parametresi çalıştırılabilir (executable) dosyanın yol ifadesini alır. Diğer parametreler programın komut satırı argümanlarını belirtir. Listenin sonunun NULL adresle bitirilmesi gerekir. execl başarılıysa geri dönmez. Başarısızsa -1 değerine geri döner.

#include<stdio.h>

#include<stdlib.h>

#include<unistd.h>

int main(void)

{

printf("begins...\n");

if (execl("/bin/ls", "/bin/ls", "-l", (char \*)NULL) < 0) {

perror("execl");

exit(EXIT\_FAILURE);

}

printf("ends...\n"); /\* unreachable code \*/

return 0;

}

Fonksiyonun son parametresi NULL girilirken tür dönüştürmesi yapılmalıdır. Yani aşağıdaki çağırma biçimi sorunludur:

if (execl("/bin/ls", "/bin/ls", "-l", NULL) < 0) {

perror("execl");

exit(EXIT\_FAILURE);

}

Çünkü "..." parametresine karşılık NULL argümanı girilirse bu NULL makrosunun nasıl define edilmiş olduğuna bağlı olarak sorunlar çıkabilir. Bilindiği gibi NULL makrosu iki biçimde define edilmiş olabilir:

#define NULL 0

ya da,

#define NULL ((void \*)0)

İşte eğer NULL birinci biçimde olduğu gibi define edilmişse buna karşı gelen parametre gösterici olmadığı için derleyici argüman stack'e 4 byte gönderir. Oysa 64 bit sistemlerde göstericiler 8 byte olduğu için sorun çıkar. Fakat biz bu son parametreyi (char \*)NULL biçiminde girersek bu durumda her halukarda stack'e NULL adres atılacaktır.

Diğer bir çok kullanılan exec fonksiyonu da execv fonksiyonudur (Buradaki 'v' "vector" sözcüğünden kısaltmadır.) execv fonksiyonunda komut satırı argümanları bir gösterici dizisine yerleştirilip geçirilir. Fonksiyonun prototipi şöyledir:

#include <unistd.h>

int execv(const char \*path, char \*const argv[]);

Örneğin:

#include<stdio.h>

#include<stdlib.h>

#include<unistd.h>

int main(void)

{

char \*args[] = { "/bin/ls","-l", NULL };

if (execv("/bin/ls", args) < 0) {

perror("execv");

exit(EXIT\_FAILURE);

}

return 0;

}

Söz konusu gösterici dizisinin NULL adresle sonlandırılmış olması gerekir. (Burada NULL makrosununun dönüştürme yapılması gerekmemektedir.)

Örneğin bir program kendi komut satırı argümanlarıyla aldığı başka programı çalıştırsın. Örneğin programamız sample olsun o da ls'yi çalıştırsın:

./sample /bin/ls -l

execv fonksiyonu bunun çok uygundur:

#include<stdio.h>

#include<stdlib.h>

#include<unistd.h>

int main(intargc, char \*argv[])

{

char \*args[] = { "/bin/ls", "-l", NULL };

if (execv(argv[1], &argv[1]) < 0) {

perror("execv");

exit(EXIT\_FAILURE);

}

return 0;

}

exec fonksiyonunun p'li versiyonları çalıştırılacak dosyayı exec uygulayan prosesin PATH çevre değişkeni ile belirtilen dizinlerde arar. PATH çevre değişkeni ':' karakterleriyle ayrılan dizinlerden oluşan bir yazı biçimindedir. Örneğin:

echo $PATH

/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:/usr/local/games

Ancak exec fonksiyonunun p'li vers,yonlarının PATH çevre değişkenine bakması için yol ifadesinde hiç bir '/' karakterinin bulunmaması gerekir. Aksi halde bu fonksiyonlar PATH çevre değişkenine bakmazlar ve tamamen p'siz versiyonlar gibi yol ifadesinin mutlak ya da göreli olması durumuna göre aramalarını yaparlar. Örneğin:

execlp("a/sample", "a/sample", (char \*)NULL);

Burada execlp PATH çevre değişkenine bakmamaktadır. sample dosyasını prosesin çalışma dizininin altındaki a dizininin içerisinde arar. Ancak yol ifadesinde hiç '/ ' karakteri yoksa bu p'li versiyonlar yalnızca PATH çevre değişkeni ile belirtilen dizinlerde arama yaparlar. Ayrıca çalışma dizinine bakmazlar. Örneğin:

execlp("sample", "a/sample", (char \*)NULL);

Burada sample programı bulunulan dizinde olsa bile oraya bakılmamaktadır. Yalnızca PATH çevre değişkeni ile belirtilen dizinlere bakılmaktadır.

shell programları exec fonksiyonlarının p'li versiyonlarını kullanarak programları çalıştırmaktadır. İşte biz de bu yüzden programları çalıştırırken "./sample" biçiminde isimleri belirtiriz. Eğer böyle bir program "sample" biçiminde çalıştırılmaya çalışılırsa exec fonksiyonlarının p'li versiyonları onları yalnızca PATH çevre değişkeni belirtilen dizinlerde arayacak, dolayısıyla bulamayacaktır. "./sample" yol ifadesi aslında "sample" ile aynı anlama geliyor olsa bile işin içerisinde bir '/' karakteri karıştırılmıştır.

Ayrıca exec fonksiyonlarının bir de e'li versiyonları vardır. (Burada 'e' "environment" sözcüğünden gelmektedir.) Bu versiyonlar program çalıştırılırken çevre eğişken takımının değiştirilmesine yol açmaktadır.

**fork ve exec Fonksiyonlarının Bir Arada Kullanılması**

Bilindiği yalnızca fork fonksiyonu bit prosesin özdeş yeni bir kopyasını oluşturur. Yeni prosesin çalıştırdığı kod aynı kod olacaktır. Yalnız başına exec fonksiyonları ise proses yaratmayıp mevcut prosesin başka bir kodla çalışmasına devam etmesini sağlar. Peki hem bizim programımız çalışmaya devam ederken hem de başka bir programı naısl çalıştırabiliriz? İşte bunun için fork ve exec birlijte kullanılmalıaıdır. Önce bir kez fork yapılır. alt proseste exec uygulanır. Tipik kalıp şöyledir:

if ((pid = fork()) < 0) {

perror("fork");

exit(EXIT\_FAILURE);

}

if (pid == 0)

if (execl(...) < 0) {

perror("execl");

exit(EXIT\_FAILURE);

}

/\* parent devam eder, else'e gerek yok \*/

Ya da daha kompakt yazım uygulanabilir:

if ((pid = fork()) < 0) {

perror("fork");

exit(EXIT\_FAILURE);

}

if (pid == 0 && execl(...) < 0) {

perror("execl");

exit(EXIT\_FAILURE);

}

Örneğin:

#include<stdlib.h>

#include<unistd.h>

int main(void)

{

pid\_t pid;

printf("parent begins\n");

if ((pid = fork()) < 0) {

perror("fork");

exit(EXIT\_FAILURE);

}

if (pid == 0 && execlp("ls", "ls", "-l", (char \*)NULL) < 0) {

perror("execlp");

exit(EXIT\_FAILURE);

}

printf("parent ends..\n");

return 0;

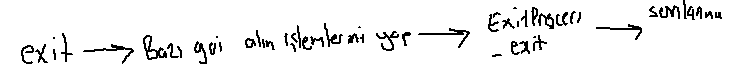
}

Windows sistemleriyle UNIX/Linux sistemlerinin proses yaratma bakımından farklı olduğuna dikkat ediniz. Windows'ta fork ve exec benzeri bir mekanizma yoktur. Orada CreateProcess adeta buradaki fork ve exec'in birliktge kullanılmasına benzemektedir. Yani Windows sisyemlerinde zaten yeni bir proses bir kodla başlatılır.

**Proseslerin Sonlandırılması ve Exit Kodları**

Prosesler işletim sisteminin sistem fonksiyonlarıyla sonlandırılır. Windows sistemlerine ExitProcess API fonksiyonu, POSIX sistemlerinde \_exit (Linux'ta sys\_exit'i çağırır) fonksiyonu prosesi sonlandırmakta kullanılır. C'nin standart exit fonksiyonu ise dolaylı olarak bu fonksiyonları çağırmaktadır.

C'nin standart exit fonksiyonu işletim sisteminin sistem fonksiyonlarını çağırmadan önce standart kütüphaneye ilişkin çeşitli sonlandırma işlemlerini yapmaktadır. Dolayısıyla C'de çalışıyorsak programı işletim sisteminin API fonksiyonları ile değil exit standart C fonkisyonuyla sonlandırmamız daha uygun olur. Örneğin biz bir dosya açıp içine birşeyler yazmış olalım. Bu durumda prosesi işletimn sistem fonksiyonlarıyla sonlandırmamız uygun olmaz. Çünkü stdio fonksiyonlarının oluşturduğu tampon fclose işlemi sırasında fklush edilmektedir. Oysa işletim sisteminin sistem fonksiyonlarının bu tampondan haberi yoktur. Onlar dosyayı işletim sistemi düzeyinde kapatırlar. Fakat tabi eğer biz standart C kütüphanesi ile ilgili önemli işlemler yapmamışsak prosesi doğrudan işletim sisteminin sistem fonksiyonlarıyla da sonlandırabiliriz.



Bir proses sonlandığında işletim sistemine "exit kodu" denilen bir kod iletilir. Prosesin exit kodu C'de exit fonksiyonuna verilen argümandır. main fonksiyonunda return uygulanırsa bu da aynı anlama gelmektedir. C standartlarına göre main fonksiyonun geri dönüş değeri (eğer geri dönerse) exit fonksiyonuna argüman yapılmaktadır. Yani C standartlarına göre bir C programı şöyle çalıştırılır:

exit(main(...));

Ayrıca C'de (main fonksiyonu için istisna olarak) eğer main'de return uygulanmazsa sanki ana bloğun sonunda 0 ile geri dönülmüş gibi işlem uygulanır. Yani:

int main(void)

{

...

}

ile,

int main(void)

{

...

return 0;

}

aynı anlamdadır.

Peki exit kodu ne işe yarar? İşletim sistemi için exit kodunun kaç olduğunun bir önemi yoktur. İşletim sistemi bu kodu alır, saklar. Eğer üst proeses isterse ona verir. Böylece üst proses alt prosesin nasıl sonlandığını bilmiş olur. Duruma göre birşeyler yapabilir. Geleneksel olarak başarılı sonlanmalarda sıfır değeri, başarız sonlanmalarda sıfır dışı değerler tercih edilmektedir.

**Anahtar Notlar:** Shell programları hem bir komut satırı sunarlar hem de bunlar bir yorumlayıcı (interpreter) da içermektedir. Yani her shell ortamının bir "shell script" dili vardır. Bugün için en çok kullanılan shell "bash (bourne again shell)" isimli shell'dir. bash'in de bir script dili vardır. Bu dilin öğrenilmesi zor değildir. Bir shell script bir kaynak dosyadır ve genellikle uzantısı .sh biçimindedir. Shell script dosyaları shell programları tarafından yorumlanarak çalıştırılırlar. Onları çalıştırmanın iki yolu vardır. Birincisi çalıştırılacak script dosyasını shell programına komut satırı argümanı olarak vermektir. Bu durumda shell interaktif modda çalışmayacak, ilgili dosyayı çalıştırıp sonlanacaktır. Örneğin:

/bin/bash sample.sh

İkinci yöntemde script dosyasına sanki çalıştırılabilir bir dosya gibi 'x' hakkı vermektir:

chmod +x sample.sh

Bu durumda dosyanın ilk satırının yorumlayıcı programın yol ifadesini içermesi gerekmektedir.

/\* sample.sh Dosyası \*/

#! /bin/bash

...

exec fonksiyonları böyle bir dosyayı yüklerken onun gerçek bir çalıştırılabilir dosya olmadığını anlar (örneğin bunların ELF başlığı yoktur). Onun birinci satırına bakar ve orada belirtilen programı çalıştırır. Bu programa söz konusu dosyayı komut satırı argümanı olarak geçrir. Yani yine çalışma aşağıdaki gibi olmuç olacaktır:

/bin/bash sample.sh

Tabi biz bu yöntemle aslında her türlü dosyayı çalıştırabiliriz. Örneğin perl, python dosyalarını vs.

Proseslerin exit kodları nasıl alınabilir? Tabi bunun için önce prosesin sonlanması gerekir. Windows'ta GetExitCodeProcess fonksiyonu prosesin exit kodunu almakta kullanılır:

BOOL WINAPI GetExitCodeProcess(

\_\_in HANDLE hProcess,

\_\_out LPDWORD lpExitCode

);

Fonksiyonun birinci parametresi exit kodunun alınacağı prosesin handle değeridir. İkinci parametresi exit kodunun yerleştirileceği DWORD türden nesnenin adresini alır. Eğer söz konusu proses henüz sonlanmamışsa exit kodu olarak 259 (STILL\_ACTIVE) elde edilir.

#include<stdio.h>

#include<stdlib.h>

#include<Windows.h>

void ExitSys(LPCSTR lpszMsg, int status);

int main(void)

{

char cmdLine[] = "..\\TestApp\\Debug\\TestApp.exe";

STARTUPINFO si = { sizeof(STARTUPINFO) };

PROCESS\_INFORMATION pi;

DWORD dwExitCode;

if (!CreateProcess(NULL, cmdLine, NULL, NULL, FALSE, 0, NULL, NULL, &si, &pi))

ExitSys("CreateProcess", EXIT\_FAILURE);

WaitForSingleObject(pi.hProcess, INFINITE);

if (!GetExitCodeProcess(pi.hProcess, &dwExitCode))

ExitSys("GetExitCodeProcess", EXIT\_FAILURE);

if (dwExitCode == STILL\_ACTIVE) {

printf("Program is still running...\n");

exit(EXIT\_SUCCESS);

}

printf("%lu\n", dwExitCode);

return 0;

}

void ExitSys(LPCSTRlpszMsg, intstatus)

{

DWORD dwLastError = GetLastError();

LPTSTR lpszErr;

if (FormatMessage(FORMAT\_MESSAGE\_ALLOCATE\_BUFFER | FORMAT\_MESSAGE\_FROM\_SYSTEM, NULL, dwLastError,

MAKELANGID(LANG\_NEUTRAL, SUBLANG\_DEFAULT), (LPTSTR)&lpszErr, 0, NULL)) {

fprintf(stderr, "%s: %s", lpszMsg, lpszErr);

LocalFree(lpszErr);

}

exit(status);

}

UNIX/Linux sistemlerinde prosesin exit kodu waitxxx isimli fonksiyonlarla elde edilmektedir. wait fonksiyonu yetersiz kaldığı için waitpid isimli fonksiyon da daha sonra POSIX sistemlerine dahil edilmiştir. wait fonksiyonun prototipi şöyledir:

#include <sys/wait.h>

pid\_t wait(int \*status);

wait fonksiyonu hem prosesin sonlanmasını bekler hem de sonlandığında onun exit kodunu alır.

wait ilk sonlanan alt prosesin exit kodunu alır ve sonlanan alt prosesin proses id değeri ile geri döner. (Tabi zaten sonlanmış bir alt proses varsa hemen onun exit kodunu alır, bekleme yapmaz.) Fonksiyon sonlanan prosesin exit kod bilgisini aparametresiyle aldığı int türden nesneye yerleştirir. Başarısızlık durumunda ise -1 değerine geri döner.

Aslında fonksiyonun parametreye yerleştirdiği değer yalnızca exit kodu değildir. Bu int nesnesnin bazı bitleri prosesin neden sonlandığına yönelik bilgi de içermektedir. İşte hangi bitlerin hangi amaçla kullanıldığı sistemden sisteme değişebildiği için standart birkaç makro bulundurulmuştur. WIFEXITED(status) makrosu eğer proses normal bir biçimde sonlanmışsa sıfır dışı değer verir. WEXITSTATUS(statıs) makrosu ise bize exit kodunu verir. Eğer fonksiyonun parametresi NULL geçilirse exit kod elde edilmez. Fakat bekleme yapılır. (Yani wait fonksiyonu Windows'taki WaitForSingleObject ve GetExitCodeProcess fonksiyonlarının birleşimi gibidir.) Örneğin:

#include<stdio.h>

#include<stdlib.h>

#include<unistd.h>

#include<sys/wait.h>

int main(void)

{

pid\_t pid;

int status;

printf("parent begins\n");

if ((pid = fork()) < 0) {

perror("fork");

exit(EXIT\_FAILURE);

}

if (pid == 0 && execl("app", "app", (char \*)NULL) < 0) {

perror("execl");

exit(EXIT\_FAILURE);

}

if (wait(&status) < 0) {

perror("wait");

exit(EXIT\_FAILURE);

}

if (WIFEXITED(status))

printf("%d\n", WEXITSTATUS(status));

else

printf("Process doesn't finish normally\n");

return 0;

}

**UNIX/Linux Sistemlerinde Hortlak (Zombie) Proses Kavramı**

UNIX/Linux sistemlerinde prosesin exit kodu Proses Kontrol Bloğuna yazılır ve buradan alınır. Bu nedenle bir proses bittiği halde henüz onun exit kodunu wait fonksiyonlarıyla üst proses almamışsa prosesin kontrol bloğu işletim sistemi tarafından serbest bırakılmaz. İşte sonlandığı halde exit kodu alınmamasından dolayı kontrol bloğu boşaltılmamış proseslere hortlak prosesler denilmektedir. wait fonksiyonlarıyla exit kod alındığında hortlaklık ortadan kalkar.

UNIX/Linux sistemlerinde eğer üst proses alt prosesten daha önce sonlanmışsa bu durumdaki alt proseslere öksüz (orphan) prosesler denilmektedir. Sistem öksüz duruma düşen prosese init ptosesini (id'si 1 olan proses) üst proses olarak atar. init de başarılı bir biçimde alt proses sonlandığında onun exit kodunu alarak onu zombi olmaktan kurtarır. O halde zombi durumu yalnızca üst proses yaşarken fakat alt proses sonlanmışsa oluşur. eğer alt proses sonlandoktan sonra üst proses de sonlanırsa yine init alt prosesin exit kodunu almaktadır. Peki hortlaklığın önemi nedir? Bazı programlar çok uzun süre (örneğin bir seneden fazla) çalışabilmektedir. Bunlar sürekli hortlak proses üretirse sistem kaynaklarını tüketebilir. Ayrıca hortlak prosesin id değeri de başka bir prosese verilememektedir. wait fonksiyonlarının dışında otomatik hortlaklığı engellemenin başka yolları da vardır.

**Proseslerin Çevre Değişkenleri (Environment Variables)**

Modern işletim sistemlerinin hemen hepsi çevre değişkeni kavramını içermektedir. Çevre değişkeni aslında bir anahtara karşı bir değerin getirildiği bir çifti belirtir. Anahtar da değer de birer yazıdır. Çevre değişkeni denildiğinde genellikle bu anahtardan bahsedilir. Örneğin anahtar (yani çevre değişkeni) "City" olsun. Buna karşı gelen değer "New York" olabilir. Ya da anahtar "Count" olsun. Buna karşı gelen değer "123" olabilir. Böylece her prosesin bir çevre değişken takımı vardır. Her ne kadar böyle bir veri yapısı tamamen manuel olarak oluşturulabilse de çevre değişkenlerinin organizasyonu ve kullanımı çekirdek düzeyinde yapılmaktadır ve taban bir kavramdır. Yani prosesin hiçbir şeyi yokken çevre değişkenlerini kullanabilir. Ayrıca bazı çevre değişkenleri bizzat çekirdek tarafından da kullanılmaktadır (PATH gibi).

Programcı çevre değişkenleriyle ilgili dört şeyi yapabilir:

1) Yeni çevre değişkenini değerle birlikte prosesin çevre değişken listesine ekleyebilir.

2) Bir çevre değişkenini prosesin çevre değişken listesinden silebilir.

3) Bir çevre değişkenine karşılık gelen değeri elde edebilir

4) Prosesin tüm çevre değişken listesini elde edebilir.

Eçvre değişkenleri prosese özgüdür. Yani her prosesin ayrı bir çevre değişken listesi vardır. Genel olarak çevre değişkenleri pek çok işletim sisteminde üst prosesten alt prosese aktarılmaktadır. Yani üst proses bir proses yarattığında üst prosesin çevre değişken listesi alt prosese aktarılmaktadır (tabi bu bir kopyalama gibidir. Bu kopyalamadn asonra üst ve alt proseslerin çevre değişken listesi bağımsız olur.)

Peki prosesin çevre değişken listesi nerede tutulmaktadır? Bazı sistemlerde çevre dğeişkenleri Proses Kontrol Bloğunda tutulurken, modern sistemlerin çoğunda (örneğin Windows ve Linux sistemlerinde) prosesin çevre değişkenleri prosesin bellek alanında tutulmaktadır.

Prosesin eçvre değişkeni verildiğinde onun değerini bize veren getenv isimli satndar bir C fonksiyonu vardır:

#include <stdlib.h>

char \*getenv(const char \*name);

Fonksiyon parametre olarak çevre değişkeninin ismini alır, geri dönüş değeri olarak onun değerinin bulunduğu dizinin adresini verir. Bu adres statik olarak tahsis edilmiştir. Fonksiyon başarısız olabilir. Bu durumda NULL adrese geri döner. Örneğin:

#include<stdio.h>

#include<stdlib.h>

int main(intargc, char \*argv[])

{

char \*value;

if (argc != 2) {

fprintf(stderr, "wrong number of arguments!..\n");

exit(EXIT\_FAILURE);

}

if ((value = getenv(argv[1])) == NULL) {

fprintf(stderr, "environment variable not found!\n");

exit(EXIT\_FAILURE);

}

puts(value);

return 0;

}

Ayrıca getenv standart C fonksiyonunun dışında Windows sistemlerinde GetEnvironmentVariable isimli API fonksyionu aynı işlemi yapmaktadır (zaten Windows altında getenv standart C fonksiyonuı bunu çağırıyor):

DWORD WINAPI GetEnvironmentVariable(

\_\_in LPCTSTR lpName,

\_\_out LPTSTR lpBuffer,

\_\_in DWORD nSize

);

Prosesin çevre değişken listesine yeni bir ekeleme yapmak için kullanılabilecek bir standart C fonksiyonu yoktur. Bunun için UNIX/Linux sistemlerinde setenv ve putenv POSIX fonksyonları Windows sistemlerinde SetEnvironmentVariable API fonksiyonu kullanılmaktadır. setenv fonksiyonunun prototipi şöyledir:

#include <stdlib.h>

int setenv(const char \*name, const char \*value, int overwrite);

Fonksiyonun birinci parametresi çevre değişkenini, ikinci parametresi ise bunun değerini belirtir. Üçüncü parametre sıfır ya da sıfır dışı bir değer olarak girilir. Eğer sıfır girilirse bu eçvre değişkeni zaten varsa fonksiyon balarısız olur. Eğer sıfır dışı olarak girilirse eski değer silinir. Yerine yenisi gelir. Fonksiyon başarı durumunda sıfır, başarısızlık durumunda -1 değerine geri döner. Örneğin:

#include<stdio.h>

#include<stdlib.h>

int main(void)

{

char \*value;

if (setenv("CITY", "Istanbul", 1) < 0) {

perror("setenv");

exit(EXIT\_FAILURE);

}

if ((value = getenv("CITY")) == NULL) {

perror("getenv");

exit(EXIT\_FAILURE);

}

puts(value);

return 0;

}

putenv fonksiyonunun prototipi de şöyledir:

#include <stdlib.h>

int putenv(char \*string)

Fonksiyonun parametresi "anahtar=değer" biçiminde tek bir yazı alır. İlgili çevre değişkeniş varsa üstüne yazar. Başarı durumunda sıfır, başarısızlık durumunda -1 değerine geri döner.

#include<stdio.h>

#include<stdlib.h>

int main(void)

{

char \*value;

if (putenv("CITY=Istanbul") < 0) {

perror("setenv");

exit(EXIT\_FAILURE);

}

if ((value = getenv("CITY")) == NULL) {

perror("getenv");

exit(EXIT\_FAILURE);

}

puts(value);

return 0;

}

Microsoft da putenv POSIX fonksiyonunu \_putenv ismiyle desteklemektedir. (Tabi bu da aslında SetEnvironmentVariable API fonksiyonu çağırıyor).

Windows'ta SetEnvironmentVariable API fonksiyonunun prototipi de şöyledir:

BOOL WINAPI SetEnvironmentVariable(

\_\_in LPCTSTR lpName,

\_\_in LPCTSTR lpValue

);

Fonksiyonun birinci parametresi çevre değişkenini, ikinci parametresi bunun değerini belirtir. Çevre zaten varsa silinerek yeni değer yerleştirilir.

Bir çevre değişkenini silmek bazı sistemlerde mümkünken bazı sistemlerde mümkün değildir.

Bir prosesin tüm çevre değişkenleri nasıl elde edilir? Windows sistemlerinde GetEnvironmentString isimli API fonksiyonu bize prosesin tüm çevre değişkenlerini ve onların değerlerini verir:

LPTCH WINAPI GetEnvironmentStrings(void);

Fonksiyonun geri dönüş değeri charf türden bir göstericidir. Verilen yazı şu biçimdedir:

K1=V1\0K2=V2\0K3=V3\0K4=V4\0\0

Örneğin:

#include<stdio.h>

#include<windows.h>

int main(void)

{

char \*env;

env = GetEnvironmentStrings();

while (\*env != '\0') {

puts(env);

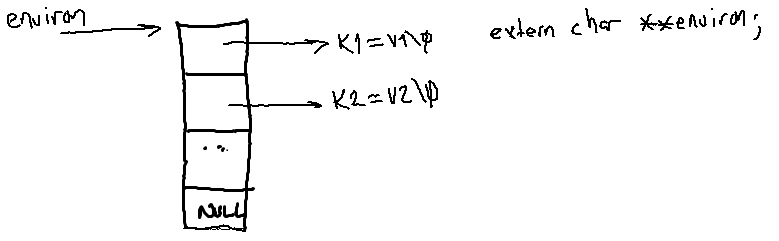
env += strlen(env) + 1;

}

return 0;

}

UNIX/Linux sistemlerinde kütüphane içerisinde tanımlanmış char \*\* türünden environ isimli bir global göstericiyi gösteren gösterici vardır. Bunun extern bildirimi maalesef hiçbir başlık dosyasında yoktur. Programcının extern bildirimini kendisinin yapması gerekmektedir. Bu göstericiyi gösteren gösterici anahtra değer çiftlerini tutan gösterici dizisinin başlangıç adresini gösterir. Dizinin sonunda NULL adres vardır:



Örneğin:

#include<stdio.h>

externchar \*\*environ;

int main(void)

{

int i;

for (i = 0; environ[i] != NULL; ++i)

puts(environ[i]);

return 0;

}

Windows sistemlerinde komut satırında "set" komutu, UNIX/Linux sistemlerinde ise "env" komutu shell prosesinin (cmd.exe ya da /bin/bash) o andaki çevre değişken listesini görüntüler.

Peki çevre değişkenleri nasıl oluşturulmaktadır? UNIX/Linux sistemlerinde fork işlemi sırasında üst prosesin çevre değişkenleri alt prosese aktarılmaktadır. exec sırasında programcı isterse çalıştıracağı programın yeni bir çevre değişken listesi ile çalıştırılmasını sağlayabilir. Yani exec sırasında prosesin çevre değişken listesi atılarak yerine yenisi oluşturulabilmektedir. Bunun için exec fonksiyonlarının e'li versiyonları kullanılır.

#include <unistd.h>

int execle(const char \*path, const char \*arg, ..., char \* const envp[]);

int execvpe(const char \*file, char \*const argv[], char \*const envp[]);

int execve(const char \*filename, char \*const argv[], char \*const envp[]);

Örneğin execle fonksiyonunu şöyle kullanabiliriz:

/\* sample.c \*/

#include<stdio.h>

#include<stdlib.h>

#include<unistd.h>

int main(void)

{

pid\_t pid;

char \*env[] = { "City=Istanbul", "Name=Ali" };

if ((pid = fork()) < 0) {

perror("fork");

exit(EXIT\_FAILURE);

}

if (pid == 0 && execle("app", "app", (char \*)NULL, env) < 0) {

perror("execle");

exit(EXIT\_FAILURE);

}

wait(NULL);

return 0;

}

/\* app.c \*/

#include<stdio.h>

externchar \*\*environ;

int main(void)

{

int i;

for (i = 0; environ[i] != NULL; ++i)

puts(environ[i]);

return 0;

}

O halde biz UNIX/Linux sistemlerinde shell üzerinden bir program çalıştırdığımızda shell'n çevre değişkenleri bizim programamıza aktarlır. Peki shell'in çevre değişkenleri nasıl oluşturulmuştur? Bu konu ileride ele alınacaktır.

Windows'ta CreateProcess sırasında CreateProcess fonksiyonunun 7'nci parametresi NULL geçilirse yaratılan prosese üst prosesin çevre değişkenleri aktarılır. Değilse, bu 7'nci parametre verilen liste aktarılır:

BOOL WINAPI CreateProcess(

\_\_in LPCTSTR *lpApplicationName*,

\_\_in\_out LPTSTR *lpCommandLine*,

\_\_in LPSECURITY\_ATTRIBUTES *lpProcessAttributes*,

\_\_in LPSECURITY\_ATTRIBUTES *lpThreadAttributes*,

\_\_in BOOL *bInheritHandles*,

\_\_in DWORD *dwCreationFlags*,

\_\_in LPVOID *lpEnvironment*,

\_\_in LPCTSTR *lpCurrentDirectory*,

\_\_in LPSTARTUPINFO *lpStartupInfo*,

\_\_out LPPROCESS\_INFORMATION *lpProcessInformation*

);

UNIX/Linux sistemlerinde shell üzerinde,

export K=V

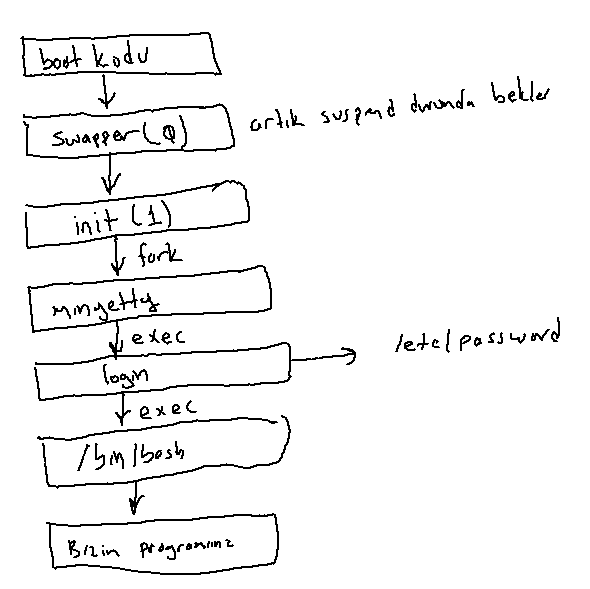
yazılır ENTER tuşuna basılırsa K çevre değişkeni yaratılır ve buna V değeri atanır. (Eğer export komutu belirtilmezse bu değişken shell dilinin değişkeni olur. Fakat shell bunu prosesin çevre değişken listesine eklemez). Shell üzerinden $<çevre değişken ismi> yazılırsa shell bize onun değerini verir. Örneğin:

export Name=Ali

echo $Name

O halde biz shell programına bir çevre değişkeni ekleyip kendi programımızı çalıştırırsak o değişken bizim programımıza da geçecektir.

Peki shell'deki çevre değişkenleri nereden gelmektedir? Tipik bir UNIX/Linux sistemi boot işleminden sonra aşağıdaki gibi proses yaratmaktadır:



Bu hikayedeki çeşitli prosesler birtakım script dosyalarına ve konfigürasyon dosyalarına bakarak birtakım şeyleri ayarlamaktadır. Sistem admin de bu dosyalara uygun şeyleri yazarak açılış sırasında sistemi konfigüre edebilir. Bu bilgiler kernel'a dahil değildir. (Yani örneğin init programı ya da login programı kernel kodları içerisinde yoktur. Bunlar başka proje grupları tarafından yazılmışlardır.)

Peki sistem boıot edilirken hangi proses hangi scrip dosyalarına bakmaktadır? İşte bu da çeşitli dağıtımlara ve dağıtımlarda kullanılan paketlere göre değişebilmektedir.

Linux sistemlerinde pek çok paketin alternatif gerçekleştirimleri vardır. Örneğin init programı için poüler iki paket kullanılmaktadır: sysvinit ve upstart. Örneğin Suse gibi sistemler klasik sysvinit paketini kullanırken, Ubuntu ve türevleri (örneğin Mint gibi) upstart kullanmaktadır. sysvinit paketindeki init programı /etc/inittab dosyasına, upstart paketindeki init rogramı /etc/init/init.conf dosyasına bakmaktadır. Bu dosyaların formatlarına ilişkin bilgi man sayfalarından elde edilebilir.

login prosesi user name ve password sorgulamasını yapar. Password'ü /etc/passwd dosyasına bakarak doğrular ve burada belirtilen programı çalıştırır. /etc/passwd dosyası sistemdeki tüm kullanıcıların kayıtlarının bulunduğu bir text dosyadır. Bu dosyada her bir satır ':' lerle ayrılmış alanlardan oluşur. Örneğin:

csd:x:1000:1000:CSD,,,:/home/csd:/bin/bash

student:x:1001:1000:CSD,,,:/home/student:/bin/bash

usermetrics:x:115:124:User Metrics:/var/lib/usermetrics:/bin/false

clickpkg:x:116:125::/nonexistent:/bin/false

Görüldüğü gibi her kullanıcı için 7 alan vardır. İlk alanda kullanıcının ismi bulunur. İkinci alanda password bulunmaktadır. Burada 'x' karakteri varsa bu, şifrenin /etc/shadow dosyasında olduğunu gösterir. Sonra user id, sonra da grup id değerleri gelmektedir. Son alanda login başarılıysa çalıştırılacak program bulunur. Sondan bir öncekinde o program çalıştırıldığında başlangıç çalıma dizini bulunmaktadır.

login prosesi bazı çevre değişkenlerini prosese dahil etmektedir. Örneğin USER, HOME, PATH, SHELL gibi.

login prgramı shell programını çalıştırır (tipik olarak /bin/bash). Shell tarafından da pek çok çevre değişkeni listeye eklenmektedir. Örneğin: PWD, LINES, COLUMNS, LANG gibi...

Peki biz bir çevre değişkeninin kalıcılığını nasıl sağlayabiliriz? İşletim sistemi başlatılırken çalıştırılan çeşitli script dosyalarının içerisine çevre değişken yaratım komıtlarını eklersek sistem açıldığında otomatik olarak o çevre değişkenleri proses ağacına dahil edilmiş olur. Örneğin çevre değişkenleri için tipik olarak /bin/bash shell'inin çalıştırdığı script dosyaları tercih edilmektedir. Tipik olarak bash iki script dosyasını çalıştırmaktadır. Bunlar: /etc/profile ve ~/.bash\_profile dosyalarıdır. /etc/profile dosyası her kullanıcıyı ilgilendirecek bir dosyadır ve her shell çalıştırıldığında çalıştırılır. Home dizinindeki .bash\_profile dosyası yalnızca ilgili kullanıcı shell'i çalıştırdığında çalıştırılır. Örneğin ~/.bash\_profile dosyasına aşağıdaki satırı yerleştirerek PATH çevre değişkenine ekleme yapabiliriz:

export PATH=$PATH:/home/csd/Study

Windows sistemlerinde de genel mantık aynıdır. Windows'ta masaüstünü "EXPLORER.EXE" isimli proses temsil eder. Windows'ta da bir loin prosesi vardır. Bu proses login işlemini yapar. Explorer.exe prosesi bu proses tarafından çalıştırılır. Biz "Denetim Masası/Sistem/Gelişmiş Sistem Ayarları/Ortam Değişkenleri" menüsünden yeni çevre değişkenleri ekleyip, mevcut olanları değiştirebiliriz. Tabi bu değişiklik o anda çalışmakta olan proseslere yansımaz.

Windows'ta komut satırında (cmd.exe) çevre değişkeni girmek için SET komutu kullanılır. Örneğin:

SET City=Ankara

Yine komut satırında çevre değişkenlerinin anahtarı verildiğinde değeri elde etmek için %Key% ifadesi kullanılır. Örneğin:

SET PATH=%PATH%;c:\Study

**Anahtar Notlar:** Windows'ta çevre değişkenlerinin büyük harf küçük harf duyarlılıkları yoktur. Fakat UNIX/Linux sistemlerinde vardır.

**Çevre Değişkenlerine Neden Gereksinim Duyulmaktadır?**

Çevre değişklenlerine işletim sistemi düzeyindeki global değişkenler gibi bakılabilir. Programlar birtakım dosyaları vs. bazı çevre değikenlerinin belirttiği yerde arayabilirler. Örneğin bir veritabanı dosyası DATABASE isimli bir çevre değişkeninin belirttiği dizinde aranabilir. Bu durumda programcı önce getenv fonksiyonuyla bu çevre dğeiikeninin değerini alır ve dosyayı o dizinde araybilir. Veritabanı dosyası başka dizine yerleştirmek istenirsetek yapılacak şey bu çevre değişkeninin değerini değiştirmektir. Örneğin pek çok C derleyicisi <...> biçiminde include edilmiş dosyaları INCLUDE isimli bir çevre değişkenin belirttiği dizinde aramaktadır. Eğer bu eçvre değişkeni set edilmemişse default bir dizin kullanılmaktadır. Ya da örneğin Java'da çeşitli jar dosyaları ve derlenmiş dosyalar CLASSPATH isimli bir çevre değişkenine bağlı olarak aranmaktadır. Bazı çevre değişkenlerini doğrudan işletim sistemi kullanmaktadır. (Örneğin PATH gibi, LD\_LIBRARY\_PATH gibi).

Bazı çevre değişkenleri işletim sisteminin içinde bulunduğu durum hakkında bilgi veriyor olabilir. Örneğin Windows hangi dizine yüklenmiştir? O anda kullanılan yerel ayarlar nelerdir? Kullanıcının ismi nedir? Home dizini hangisidir vs.

Çevre değişkenleri proseslerarası haberleşmede de bazen kullanılabilmektedir. Örneğin, üst proses belli bir çevre değişkenini set ederek alt prosesi çalıştırır. Alt proses böylece o çevre değişkeninden bilgi alabilir. Böylesi basit bilgi aktarımı için herhangi bir proseslerarası haberleşme yönteminin kullanılmasına gerek yoktur.

**Anahtar Notlar:** UNIX/Linux sistemlerinde (Windows'ta da böyle) shell programları temelde iki modda çalıştırılmaktadır: Interaktif mod ve Interaktif olmayan (yani tek komutluk) mod. İnteraktif modda bir prompt çıkar. Kullanıcı komut girer, o komut çalışır yeniden prompt'a düşülür. Taki exit ya da logout komutu uygulanana kadar. Halbuki interaktif olmayan modda tek bir komut çalıştırılıp shell'den çıkılır. Örneğin system isimli standart C fonksiyonu shell'i böyle çalıştırmaktadır. Tek komutluk çalıştırma için /bin/bash'te (diğer shell'lerde de aynı) -c seçeneği kullanılmaktadır. Örneğin:

/bin/bash -c ls

Bıurada bash ls komutunu çalıştırıp çıkacaktır.

**Proseslerarası Haberleşme (Interprocess Communication - IPC)**

Yukarıda açıklandığı gibi modern sistemlerde sayfa tabloları yoluyla prosesler arasında tam bir izaolasyon sağlanmıştır. Her proses sanki fiziksel belleğe tek başına yüklenip çalışıyormuş gibi bir illüzyon oluşturulur. Bir prosesten diğerine n byte gönderip alma sürecine "Proseslerarası Haberleşme" denilmektedir. Proseslerarası haberleşme kabaca ikiye ayrılır:



Aynı makinanın prosesleri arasındaki haberleşmelerde kullanılan tipik teknikler şunlardır:

- Borular (Pipes)

- Paylaşılan Bellek Alanları (Shared Memory)

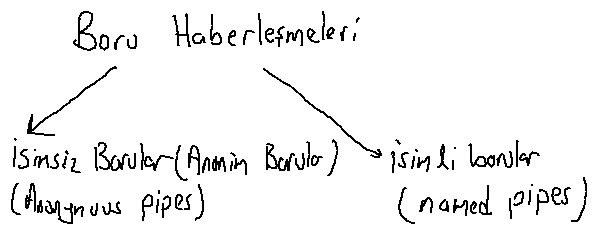
- Mesaj Kuyrukları (Message Queue)

Ağ altında farklı makinaların prosesleri arasında haberleşmede bir protokol de devreye girer. Bu amaçla oluşturulmuş değişik protokoller vardır. Bugün en yaygın kullanılan protokol Internet'in de kullandığı IP protokol ailesidir. Kursumuzda IP protokol ailesi ile haberleşme ele alınacaktır.

**Anahtar Notlar:** Raspberry Pi'ın 2 numaralı modeli oldukça hızlanfırılmıştır (4 core). Microsoft da Windows 10'un Raspberry Pi versiyonunu oluşturmuştur. Bu tür çok sayıda embedded Linux kitleri bulunmaktadır. Ancak Raspberry Pi oldukça standartlaşmış durumdadır ve önemli ölçüde desteğe sahiptir. Dokümantasyonu da oldukça geniştir. Raspberry Pi üzerine çok çeşitli işletim sistemi kurulabiliyorsa da şu an için en çok tercih edilen "Raspbian" isimli Linux dağıtımıdır. Raspbian üzerine IDE'ler, editörler, server programlar kurulabilir. Böylece Raspberry Pi ucuz bir PC olarak kullanılabilmektedir. Bu tür embedded Linux kitleri özellikle IO amaçlı endüstriyel ortamlarda kullanılmaktadır. Yani bu kitlere kameralar, sensörler, step motorlar vs. bağlanarak bunlar başka amaçları gerçekleştirmek için kullanılabilmektedir.

**Boru Haberleşmeleri**

Boru haberleşmeleri hem UNIX/Linux sistemlerinde hem de Windows sistemlerinde kullanılan yaygın bir tekniktir. Bu haberleşme modelinde senkronizasyon da otomatik sağlanmaktadır. Bu bakımdan kullanılması kolay bir yöntemdir. Boru hanerleşmeleri her iki sistemde de "İsimli" ve "İsimsiz (anonim de denir)" olmak üzere ikiye ayrılmaktadır.



Borular UNIX/Linux sistemlerinde tek kanallıdır. Yani bir taraf yazar, diğer taraf okur (half duplex). Fakat tek bir boruyla aynı anda iki tarafın yazıp okuması mümkün değildir. Windows sistemlerinde borular çift kanallıdır. Dolayısıyla bir boru yaratıldığında bir tarafın yazdığını diğer taraf okurken, diğer tarafın yazdığını da karşı taraf okuyabilir. Aynı durumu UNIX/Linux sistemlerinde sağlamak için iki boru yaratmak gerekir.

İsimsiz borular üst ve alt prosesler arasında haberleşme kullanılmaktadır. İsimli borular ise aynı makina içerisindeki herhangi iki proses arasında kullanılabilir.

Tipik bir boru haberleşmesi şöyle yapılır:

1) Önce boru yaratılır.

2) Proseslerden biri boruya bilgiyi yazar. Genellikle borular işletim sistemlerinde sanki birer dosyaymış gibi ele alınırlar. Bu nedenle yazma ve okuma işlemleri dosya fonksiyonlarıyla yapılır. (UNIX/Linux sistemlerinde read ve write POSIX fonksiyonları, Windows sistemlerinde ReadFile ve WriteFile API fonksiyonları).

3) Diğer proses bilgiyi okur. Borular birer FIFO kuyruk sistemi gibidir. Ve borularda stream taabanlı bir haberleşme söz konusudur.

4) Borunun belli bir uzunluğu vardır. Eğer yazan taraf yazar, okuyan taaraf bunları almazsa boru dolar. Boru dolduğu zaman yazan taraf yazmaya çalıştığında bloke olur. Yani donar kalır. Ta ki karşı taraf okuyup boruda yer açılana kadar. Benzer biçimde okuyan taraf borudakilerin hepsini okumuşsa ve yeniden okumak isterse yine bloke olur. Yani donarak bekler. Ta ki diğer taraf boruya birşey yazana kadar.

5) Boru haberleşmesi yazan tarafın boruyu kapatmasıyla sonlandırılmalıdır. Bu durumda okuyan taraf önce boruda kalanları okur. Sonra artık okuma fonksiyonu sıfır ile geri döner. Okuyan taraf da bölece karşı tarafın boruyu kapattığını anlar. O da boruyu kapatır. Önce okuyan tarafın boruyu kapatması normal bir durum değildir. Bu genellikle sistemlerde çökmeye yol açar.

**UNIX/Linux Sistemlerinde İsimsiz Boru Haberleşmeleri**

Blokeli modda UNIX/Linux sistemlerinde isimsiz boru haberleşmesi şöyle yapılır:

1) Üst proses önce pipe isimli fonksiyonla boruyu yaratır.

#include <unistd.h>

int pipe(int pipefd[2]);

**Anahtar Notlar:** C'de fonksiyonun gösterici parametresi dizi formunda belirtilebilir ve köşeli parantezlerin içerisine sabit ifadeleri yazılabilir. Bunun okunabilirlik dışında göstericiden hiçbir farkı yoktur. Dolayısıyla aşağıdaki prototiplerin hepsi eşdeğerdir:

void foo(int \*a);

void foo(int a[]);

void foo(int a[2]);

void foo(int a[100]);

Dolayısıyla pipe fonksiyonunun prototipi şöyle de yazılabilirdi:

int pipe(int \*pipefd);

pipe fonksiyonu başarı durumunda 0, başarısızlık durumunda -1 değerine geri döner. Fonksiyona iki elemanlı bir int dizinin adresi geçirilmelidir. Örneğin:

int pipefds[2];

...

if (pipe(pipefds) < 0) {

perror("pipe");

exit(EXIT\_FAILURE);

}

Fonksiyon bize iki betimleyici (yani iki handle değeri) verir. Birinci betimleyici borudan okuma yapmak için kullanılan betimleyicidir. İkincisi ise boruya yazma yapmak için kullanılacak betimleyicidir.

2) Artık üst proses fork fonksiyonuyla alt prosesi oluşturur. Böylece üst prosesin boru betimleyicileri alt prosese aktarılmış olur. Bu durumda borudan okuma potansiyeli olan ve boruya yazma potansiyeli olan ikişer betimleyici vardır.

if ((pid = fork()) < 0) {

perror("fork1");

exit(EXIT\_FAILURE);

}

3) Şimdi hangi prosesin boruya yazma yapacağı, hangisinin okuma yapacağının belirlenmiş olması gerekir. Yazan taraf okuma betimleyicisini, okuyan taraf da yazma betimleyicisini close fonksiyonuyla kapatmalıdır. Yani boruya yazma ve okuma potansiyelinde olan tek bir betimleyici kalmalıdır.

if ((pid = fork()) < 0) {

perror("fork1");

exit(EXIT\_FAILURE);

}

if (pid != 0) { /\* üst proses boruya yazacak \*/

close(pipefds[0]);

...

}

else { /\* alt proses borudan okuyacak \*/

close(pipefds[1]);

...

}

4) Artık yazan taraf write fonksiyonuyla boruya yazma yapar. Okuyan taraf da read fonksiyonuyla borudan okuma yapacaktır:

#include <unistd.h>

ssize\_t read(int fd, void \*buf, size\_t count);

ssize\_t write(int fd, const void \*buf, size\_t count);

Burada ssize\_t türü POSIX sistemlerinde işaretli bir tamsayı türü olarak typedef edilmek zorunda bir tür ismidir.

Fonksiyonların birinci paarametreleri borunun (dosyanın) handle değeridir. (Dosya handle değerlerine UNIX/Linux sistemlerinde "dosya betimleyicisi (file descriptor) denilmektedir) İkinci parametreler bellekteki okuma ya da yazma için gereken transfer adresidir. Son parametreler okunacak ya da yazılacak byte miktarını belirtir. Fonksiyonlar başarılıysa okunan ya da yazılan byte sayısına, başarısızsa -1 değerine geri dönerler. read fonksiyonu 0 ile geri dönerse bu durum karşı tarafın boruyu kapatmış olduğu ve artık boruda okunacak hiçbirşeyin kalmadığı anlamına gelir. read boru boşsa, write ise boru doluysa zaten geri dönmeden beklerler. Blokeli modda çalışmanın önemli birkaç noktası vardır.

- Borudan n byte okunmak istendiğinde boruda n'den daha az byte varsa read fonksiyonu n byte'ın hepsini okuyana kadar blokede kalır.

- Birden fazla kaynak boruya yazma yaparken iç içe geçme olmaz. Ancak borunun bir uzunluğu vardır (PIPE\_LEN). Bu uzunlktan daha fazla yazmak böylesi bir durumda iç içe geçme oluşturabilir. Yazma sırasında da tüm byte'lar yazılana kadar bloke çözülmez.

Örnek bir program şöyle yazılabilir:

#include<stdio.h>

#include<stdlib.h>

#include<unistd.h>

void pipe\_write\_proc(int fd);

void pipe\_read\_proc(int fd);

int main(void)

{

pid\_t pid;

int pipefds[2];

if (pipe(pipefds) < 0) {

perror("pipe");

exit(EXIT\_FAILURE);

}

if ((pid = fork()) < 0) {

perror("fork");

exit(EXIT\_FAILURE);

}

if (pid != 0) { /\* parent yazacak \*/

close(pipefds[0]);

pipe\_write\_proc(pipefds[1]);

close(pipefds[1]);

wait(NULL);

}

else { /\* child okuyacak \*/

close(pipefds[1]);

pipe\_read\_proc(pipefds[0]);

close(pipefds[0]);

}

return 0;

}

void pipe\_write\_proc(intfd)

{

int i;

for (i = 0; i < 100000; ++i)

if (write(fd, &i, sizeof(int)) < 0) {

perror("write");

exit(EXIT\_FAILURE);

}

}

void pipe\_read\_proc(intfd)

{

int result, val;

while ((result = read(fd, &val, sizeof(int))) > 0)

printf("%d ", val), fflush(stdout);

if (result < 0) {

perror("read");

exit(EXIT\_FAILURE);

}

}

**Shell Pipe İşlemini Nasıl Yapıyor?**

Önceki konularda da görüldüğü gibi shell üzerinden,

a | b

gibi bir işlemde a'nın stdout dosyasına yazdıklarını b stdin'den okumaktadır. Peki bu nasıl yapılmaktadır? İşte shell bunu kabaca şöyle yapar:

1) Önce pipe'ı yaratır

2) a için fork uygular ve a'nın stdout dosyasını boruya yönlendirir. Sonra a için exec uygular. Böylece a stdout dosyasına yazdığını sanırken aslında boruya yazar.

3) b için fork uygular ve b'nin stdin dosyasını boruya yönlendirir. Sonra b için exec uygular. Böylece b stdin dosyasındna okuduğunu sanırken aslında borudan okur. Tabi aslında boruya a'nın yazdıklarını okur.

4) Shell iki prosesin de bitmesini wait ile bekler.

Kursumuzda IO yönlendirmesinin aşağı seviyeli mekanizması üzerinde durulmamıştır. Bu nedenle bu örnek burada yapılmayacaktır. "UNIX/Linux Sistem Programlama" kursunda bu tür örnekler yapılmaktadır.

**Windows Sistemlerinde İsimsiz Boru Haberleşmesi**

Windows'ta isimsiz boru haberleşmesi de çok benzer yapılmaktadır. İsimsiz boru Windows'ta CreatePipe API fonksiyonuyla yaratılır:

BOOL WINAPI CreatePipe(

\_\_out PHANDLE *hReadPipe*,

\_\_out PHANDLE *hWritePipe*,

\_\_in LPSECURITY\_ATTRIBUTES*lpPipeAttributes*,

\_\_in DWORD *nSize*

);

Fonksiyonun birinci ve ikinci parametreleri borudan okumak ve boruya yazmak için gereken handele nesnelerinin adreslerini alır. Fonksiyon buraya okuma ve yazma için gereken handle değerlerini yerleştirir. Üçüncü parametre kernel nesnelerinin güvenlik parametresidir. NULL geçilebilir. Son parametre ise yaratılacak borunun byte cinsinden uzunluğudur. Fonksiyon başarı durumunda sıfır dışı bir değere, başarı sızlık durumunda sıfır değerine geri döner.

Boru yaratıldıktan sonra handle değerlerinin alt prosese geçirilebilmesi için CreateProcess fonksiyonunun 5. parametresi TRUE geçirilmelidir. Bu konu "Windows Sistem Programlama" kursunda ele alınmaktadır. Alt prosese diğer handle'ın (örneğin üst proses yazma yapıyorsa bu yazma hande'ının) geçirilmesine gerek yoktur. CreateProcess'teki 5. parametre "master switch" biçimindedir. Default kernel nesnelerinin handle değerleri alt prosese aktarılmaz. Fakat biz istersek alt prosese belirli handle'ların aktarılmısını isteyebiliriz. Bu akratımın sağlanması yaratıcı fonksiyonların SECURITY\_ATTRIBUTES parametresiyle yapılabileceği gibi SetHandleInformation API fonksiyonuyla da daha sonra yapılabilir.

İsimsiz boru yaratıldıktan sonra yine CreateProcess fonksiyonuyla alt proses oluşturulur. Sonra yine UNIX/Linux sistemlerinde olduğu gibi okuyan taraf yazma borusunu, yazan taraf da okuma borusunu kaparı. Ondan sonra ReadFile ve WriteFile dosya fonksiyonlarıyla okuma yazma işlemleri gerçekleştirilir. Yine önce boruya yazma yapan tarafın boruyu kapatması gerekir. ReadFile borudan sıfır byte okuduğu zamana borunun kapatılmış olduğunu anlar o da boruyu kapatır.

Windows'taki uygulamanın UNIX/Linux sistemlerinden küçük bir farkı daha vardır. Bu sistemlerde CreateProcess adeta UNIX/Linux sistemlerindeki fork/exec işlemine karşılık gelir. Bu duurmda alt proses üst prosesten aktarılan boru handle değerini bilemez. O halde bu handle değerinin de alt prosese aktarılması gerekir. Bu aktarım komut satırı argümanları yoluyla ya da çevre değişkenleri yoluyla yapılabilir.

İsimsiz Boru Haberleşmesi yapan örnek bir Windows programı şöyle yazılabilir:

/\* Parent.c \*/

#include<stdio.h>

#include<stdlib.h>

#include<Windows.h>

void ExitSys(LPCSTR lpszMsg, int status);

void WritePipeProc(HANDLE hPipeWrite);

int main(void)

{

HANDLE hPipeRead, hPipeWrite;

char cmdLine[1024];

STARTUPINFO sa = { sizeof(STARTUPINFO) };

PROCESS\_INFORMATION pi;

if (!CreatePipe(&hPipeRead, &hPipeWrite, NULL, 1024))

ExitSys("CreatePipe", EXIT\_FAILURE);

SetHandleInformation(hPipeRead, HANDLE\_FLAG\_INHERIT, TRUE);

sprintf(cmdLine, "Child.exe %d", hPipeRead);

if (!CreateProcess(NULL, cmdLine, NULL, NULL, TRUE, 0, NULL, NULL, &sa, &pi))

ExitSys("CreateProcess", EXIT\_FAILURE);

CloseHandle(hPipeRead);

WritePipeProc(hPipeWrite);

CloseHandle(hPipeWrite);

return 0;

}

void WritePipeProc(HANDLEhPipeWrite)

{

int i;

DWORD dwByteWritten;

for (i = 0; i < 100000; ++i)

if (!WriteFile(hPipeWrite, &i, sizeof(int), &dwByteWritten, NULL))

ExitSys("WriteFile", EXIT\_FAILURE);

}

void ExitSys(LPCSTRlpszMsg, intstatus)

{

DWORD dwLastError = GetLastError();

LPTSTR lpszErr;

if (FormatMessage(FORMAT\_MESSAGE\_ALLOCATE\_BUFFER | FORMAT\_MESSAGE\_FROM\_SYSTEM, NULL, dwLastError,

MAKELANGID(LANG\_NEUTRAL, SUBLANG\_DEFAULT), (LPTSTR)&lpszErr, 0, NULL)) {

fprintf(stderr, "%s: %s", lpszMsg, lpszErr);

LocalFree(lpszErr);

}

exit(status);

}

/\* Child.c \*/

#include<stdio.h>

#include<stdlib.h>

#include<Windows.h>

void ReadPipeProc(HANDLE hPipeRead);

int main(intargc, char \*argv[])

{

HANDLE hPipeRead;

if (argc != 2) {

fprintf(stderr, "wrong number of arguments!..\n");

exit(EXIT\_FAILURE);

}

sscanf(argv[1], "%d", &hPipeRead);

ReadPipeProc(hPipeRead);

return 0;

}

void ReadPipeProc(HANDLEhPipeRead)

{

DWORD byteRead;

int val;

BOOL result;

while (ReadFile(hPipeRead, &val, sizeof(int), &byteRead, NULL) && byteRead != 0)

printf("%d ", val), fflush(stdout);

}

**İsimli Boru Haberleşmeleri**

İsimsiz boru haberleşmesi yalnızca üst proses ile alt proses arasındaki haberleşmelerde kullanılabilmektedir. Halbuki aralarında üstlük-altlık ilişkisi olmayan herhangi iki prosesin haberleşmesi için isimli boru haberleşmesi kullanılır.

**UNIX/Linux Sistemlerinde İsimli Boru Haberleşmesi**

UNIX/Linux sistemlerinde isimli borular sanki birer dosyaymış gibi ele alınmaktadır. Onlar dosya sisteminde dizin girişlerinde 'p' dosya özelliğiyle görüntülenmektedir. Boru dosyaları gerçek anlamda diskte yer kaplamazlar. Onların yalnızca bir dizin girişleri vardır. Onların gerçek yerleri ana bellektedir. İsimli borular sırasıyla şu adımlarla oluşturulur:

1) UNIX/Linux sistemlerinde isimli borularla çalışmak için önce bir boru dosyasının oluşturulması gerekir. Boru dosyasını oluşturabilmek için mkfifo isimli POSIX fonksiyonu kullanılabilir. Ya da komut satırından mkfifo isimli komut zaten bu fonksiyonu çağırarak isimli boru dosyası oluşturmaktadır. mkfifo fonksiyonunun prototipi şöyledir:

#include <sys/stat.h>

int mkfifo(const char \*pathname, mode\_t mode);

Fonksiyonun birinci parametresi boru dosyasının yol ifadesini, ikinci parametresi boru dosyasının erişim haklarını belirtmektedir. İkinci parametren S\_IRUSR|S\_IWUSR|S\_IRGRP|S\_IROTH biçiminde girilebilir. Bu parametrenin anlamı ileride open fonksiyonunda anlatılacaktır. Fonksiyon başarı durumunda sıfır değerine, başarıszlık durumunda -1 değerine geri döner.

mkfifo komutu da aşağıdaki gibi kullanılabilir:

mkfifo mypipe

2) Boru dosyası open isimli dosya açan PSOIX fonksiyonuyla açılır. Ya da istenirse bunun için fopen fonksiyonu da kullanılabilir. Ancak fopen kullanılacaksa dosya tamponlama stratejisinin "sıfır tamponlamalı moda" çekilmesi uygun olur. open fonksiyonu ileride ele alınacaktır. Ancak burada üstün körü bir biçimde hedefe yönelik kullanımdan bahsedebiliriz. Açış işlemini bir proses okuma modunda diğeri de yazma modunda yapmalıdır:

if ((fd = open("mypipe", O\_RDONLY)) < 0) {

perror("open");

exit(EXIT\_FAILURE);

}

...

if ((fd = open("mypipe", O\_WRONLY)) < 0) {

perror("open");

exit(EXIT\_FAILURE);

}

...

Blokeli modda proseslerden biri boruyu okuma modunda açmışsa, diğeri yazma modunda açana kadar open blokede kalır. Benzer biçinde bir proses bpruyu yazma modunda açmışsa diğeri okuma modunda açana kadar blokede kalır.

3) Artık yazan taraf write fonksiyonuyla boruya yazma yapar. Okuyan taraf da read fonksiyonuyla okumaya yapar. Tabi yine boru dolduğunda yazan taraf, boru boş olduğunda da okuyan taraf blokede kalır.

4) Yine önce yazan tarafın boruyu kapatması gerekir. Daha sonra karşı taraf read fonksiyonuyla sıfır byte okuduğunda o da boruyu kapatır.

Örnek bir program şöyle yazılabilir:

/\* pipeproc1.c \*/

#include<stdio.h>

#include<stdlib.h>

#include<unistd.h>

#include<fcntl.h>

#include<sys/stat.h>

void pipe\_write\_proc(int fd);

int main(void)

{

int fd;

if ((fd = open("mypipe", O\_WRONLY)) < 0) {

perror("open");

exit(EXIT\_FAILURE);

}

pipe\_write\_proc(fd);

close(fd);

return 0;

}

void pipe\_write\_proc(intfd)

{

int i;

for (i = 0; i < 100000; ++i)

if (write(fd, &i, sizeof(int)) < 0) {

perror("write");

exit(EXIT\_FAILURE);

}

}

/\* pipeproc.2 \*/

#include<stdio.h>

#include<stdlib.h>

#include<unistd.h>

#include<fcntl.h>

void pipe\_read\_proc(int fd);

int main(void)

{

int fd;

if ((fd = open("mypipe", O\_RDONLY)) < 0) {

perror("open");

exit(EXIT\_FAILURE);

}

pipe\_read\_proc(fd);

close(fd);

return 0;

}

void pipe\_read\_proc(intfd)

{

int result, val;

while ((result = read(fd, &val, sizeof(int))) > 0)

printf("%d ", val), fflush(stdout);

if (result < 0) {

perror("read");

exit(EXIT\_FAILURE);

}

}

İsimli boru dosyası standart C fonksiyonlarıyla da açılıp, okuma yazma yapılabilir. Tabi yukarıda da belirtildiği gibi bu durumda tamponlama mekanizmasını kaldırmak uygun olur:

/\* pipeproc1.c \*/

#include<stdio.h>

#include<stdlib.h>

#include<unistd.h>

#include<fcntl.h>

#include<sys/stat.h>

void pipe\_write\_proc(FILE \*f);

int main(void)

{

FILE \*f;

if ((f = fopen("mypipe", "w")) == NULL) {

fprintf(stderr, "cannot open file!..\n");

exit(EXIT\_FAILURE);

}

setvbuf(f, NULL, 0, \_IONBF);

pipe\_write\_proc(f);

fclose(f);

return 0;

}

void pipe\_write\_proc(FILE \*f)

{

int i;

for (i = 0; i < 100000; ++i)

if (fwrite(&i, sizeof(int), 1, f) != 1)

break;

if (ferror(f)) {

fprintf(stderr, "cannot write pipe!\n");

exit(EXIT\_FAILURE);

}

}

/\* pipeproc2.c \*/

#include<stdio.h>

#include<stdlib.h>

#include<unistd.h>

#include<fcntl.h>

void pipe\_read\_proc(FILE \*f);

int main(void)

{

FILE \*f;

if ((f = fopen("mypipe", "r")) == NULL) {

fprintf(stderr, "cannot open file!..\n");

exit(EXIT\_FAILURE);

}

setvbuf(f, NULL, 0, \_IONBF);

pipe\_read\_proc(f);

fclose(f);

return 0;

}

void pipe\_read\_proc(FILE \*f)

{

int result, val;

while (fread(&val, sizeof(int), 1, f) > 0)

printf("%d ", val), fflush(stdout);

if (ferror(f)) {

perror("cannot read pipe!\n");

exit(EXIT\_FAILURE);

}

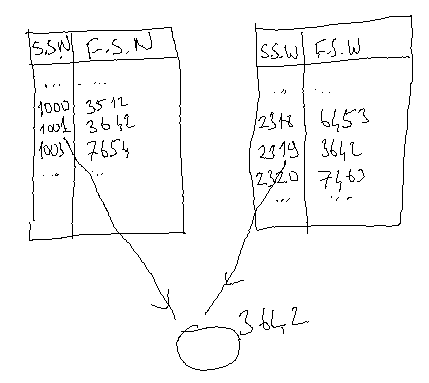
}

**Windows Sistemlerinde İsimli Boru Haberleşmesi**

Bu kursumuzda Windows sistemlerindeki isimli boru haberleşmeleri üzerinde durmayacağız. Çünkü Windows sistemlerinde isimli borular adeta client-server tarzda kullanılmaktadır. Kavramsal karmaşıklığı fazladır. Bu nedenle isimli borular Derneğimizde "Windows Sistem Programlama" kursunun konusu içerisindedir.

**Paylaşılan Bellek Alanları (Shared Memory) Yöntemi İle Proseslerarası Haberleşme**

Anımsanacağı ağı gibi modern sistemlerde işlemcinin sağladığı bir sayafalama ve buna bağlı olarak bir sanal bellek mekanizması vardır. Bu mekanizma proseslerin bellek alanlarını tamamen birbirlerinden izole eder. Paylaşılan bellek alanları yönteminde işletim sistemi iki ya da daha fazla prosesin farklı sanal sayfalarını aynı fiziksel sayfaya yönlendirir. Örneğin:



Burada soldaki prosesin 1001 numaralı sanal sayfası ile sağdaki prosesin 2319 numaralı sanal sayfası aynı fiziksel sayfaya yönlendirilmiştir.

Paylaşılan bellek alanları yöntemi çok hızlı bir yöntemdir. Prosesler hiç kernel moda geçmeden haberleşebilirler. Ancak bu yöntem kendi içerisinde borularda olduğu gibi bir senkronizasyon içermez. Yani proseslerden biri paylaşılan bellek alanına yazdığında diğeri onun yazdığını ne zaman alacaktır? Diğeri yeni birşey yazarsa eskisi ezilmez mi? İşte senkronizasyon için üretici-tüketici problemi (producer-consumer problem) gibi bir yöntemin kullanılması gerekir.

Paylaşılan bellek alanları hem Windows hem de UNIX/Linux sistemlerinde var olan bir yöntemdir. Windows'taki ve UNIX/Linux sistemlerindeki bellek tabanlı dosyalar (memory mapped files) paylaşılan bellek alanları konusuyla birleştirilmiştir.

**Windows Sistemlerinde Paylaşılan Bellek Alanlarının Oluşturulması**

Windows sistemlerinde paylaşılan bellek alanları şu adımlardan geçilerek oluşturulur:

1) Öncelikle bir "file mapping" nesnesinin yaratılması gerekir. Bu işlem CreateFileMapping fonksiyonuyla yapılır:

HANDLE WINAPI CreateFileMapping(

\_\_in HANDLE *hFile*,

\_\_in LPSECURITY\_ATTRIBUTES *lpAttributes*,

\_\_in DWORD *flProtect*,

\_\_in DWORD *dwMaximumSizeHigh*,

\_\_in DWORD *dwMaximumSizeLow*,

\_\_in LPCTSTR *lpName*

);

Fonksiyonun birinci parametresi eğer "bellek tabanlı dosya (memory mapped file)" kullanılacaksa o dosyanın handle değerini alır. eğer yalnızca paylaşılan bellek alanı oluşturulacaksa bu parametreye INVALID\_HANDLE\_VALUE özel değeri geçilmelidir. Fonksiyonun ikinci parametresi file mapping nesnesinin güvenlik parametresidir. NULL geçilebilir. Üçüncü parametre oluşturaln bellek alanının koruma özelliğini belirtir. Bu parametre PAGE\_READWRITE girilirse bu bölgeye hem okuma hem de yazma yapılabilir. Fonksiyonun dördüncü ve beşinci parametreleri paylaşılacak bellek alanın uzunluğunu belirtir. Aslında bu iki parametre 8 byte'lık bir tamsayının düşük ve yüksek anlamlı 4'er byte'ıdır. Bu değerin sayfa katlarında olması anlamlıdır. Son parametre proseslerarası paylaşım için gereken isimdir. Fonksiyon başarı durumunda mapping nesnesine ilişkin bir handle değeri ile, baaşarısızlık durumunda NULL adresle geri döner.

Bu fonksiyon iki proseste de çağrılır. İlk çağıran proses nesneyi yaratmış olur. Artık diğer çağırandaki parametreler etkili olmaz. İkinci çağıran proses yalnızca nesneyi açmış olur. Dolayısıyla fonksiyonu sonradan çağıran kişi için üçüncü, dördüncü, beşinci parametrelerin bir önemi yoktur. Tabi iki proseste de aynı değerler kullanılırsa hangisinin önce çalıştığının bir önemi kalmaz. Eğer istenirse proseslerden biri CreateFileMapping fonksiyonuyla nesneyi yaratırken diğeri OpenFileMapping fonksiyonuyla açabilir. OpenFileMapping yalnızca olanı açma özelliğine sahiptir. Yaratma özelliğine sahip değildir.

2) Bundan sonra MapViewOfFile isimli API fonksiyonuyla gerçek bellek adresi elde edilir. Yani işletim sistemi bu fonksiyonla paylaşılan fiziksel bellek alanını gören bir sanal sayfa numarasını sayfa tablosunda oluşturup bize oranın sanal adresini verecektir:

LPVOID WINAPI MapViewOfFile(

\_\_in HANDLE *hFileMappingObject*,

\_\_in DWORD *dwDesiredAccess*,

\_\_in DWORD *dwFileOffsetHigh*,

\_\_in DWORD *dwFileOffsetLow*,

\_\_in SIZE\_T *dwNumberOfBytesToMap*

);

Fonksiyonun birinci parametresi mapping nesnesinden elde edilen handle değeridir. İkinci parametre sayafay erişim özelliğini belirtir. Tabi buaradaki parametre mapping nesnesi yaratılırken verilen parametreden daha geniş haklara sahip olamaz. Tipik olarak bu parametreye FILE\_MAP\_READ|FILE\_MAP\_WRITE değeri girilmektedir. Fonksiyonun üçüncü parametreleri map edilecek kısmın offetini alır. Tabi bu parametreler bellek tabanlı dosyalar için anlamlıdır. eğer biz yalnızca paylaşılan bellek alanı oluşturuyorsak bu parametreye sıfır geçmemeiz anlamlı olur. Son parametre paylaşılacak bellek alan ın büyüklüğüdür. Mapping nesnesinde belirtilen büyüklük maksimum büyüklüktür. Buradaki değer ondan küçük olabilir ya da ona eşit olabilir. Ancak ondan büyük olamaz. Fonksiyon başarı durumunda paylaşılan bellek alanına erişmekte kullanılan adrese, başarısızlık durumda NULL adrese geri döner.

3) Kullanım bitince paylaşılan bellek alanının sisteme iade edilmesi gerekir. Bu UnmapViewOfFile fonksiyonuyla yapılır:

BOOL WINAPI UnmapViewOfFile(

\_\_in LPCVOID *lpBaseAddress*

);

Fonksiyon paylaşılan bellek alanının başlangıç adresini parametre olarak alır ve o alanı serbest bırakır.

4) Nihayet proses mapping nesnesini de CloseHandle fonksiyonuyla serbest bırakmalıdır. Tabi bittiğinde bu son iki aşama otomatik yapılır.

BOOL WINAPI CloseHandle(

\_\_in HANDLE *hObject*

);

Örneğin:

/\* Process1.c \*/

#include<stdio.h>

#include<stdlib.h>

#include<Windows.h>

void ExitSys(LPCSTR lpszMsg, int status);

int main(void)

{

HANDLE hFileMapping;

char \*memstr;

hFileMapping = CreateFileMapping(INVALID\_HANDLE\_VALUE, NULL, PAGE\_READWRITE, 0, 4096, "ThisIsATest");

if (hFileMapping == NULL)

ExitSys("CreateFileMapping", EXIT\_FAILURE);

memstr = (char \*)MapViewOfFile(hFileMapping, FILE\_MAP\_READ | FILE\_MAP\_WRITE, 0, 0, 4096);

if (memstr == NULL)

ExitSys("MapViewOfFile", EXIT\_FAILURE);

printf("Process-1: Press ENTER to write into shared memory\n");

getchar();

strcpy(memstr, "this is a test");

printf("Process-1: Press ENTER to EXIT");

getchar();

UnmapViewOfFile(memstr);

CloseHandle(hFileMapping);

return 0;

}

void ExitSys(LPCSTRlpszMsg, intstatus)

{

DWORD dwLastError = GetLastError();

LPTSTR lpszErr;

if (FormatMessage(FORMAT\_MESSAGE\_ALLOCATE\_BUFFER | FORMAT\_MESSAGE\_FROM\_SYSTEM, NULL, dwLastError,

MAKELANGID(LANG\_NEUTRAL, SUBLANG\_DEFAULT), (LPTSTR)&lpszErr, 0, NULL)) {

fprintf(stderr, "%s: %s", lpszMsg, lpszErr);

LocalFree(lpszErr);

}

exit(status);

}

/\* Process2.c \*/

#include<stdio.h>

#include<stdlib.h>

#include<string.h>

#include<Windows.h>

void ExitSys(LPCSTR lpszMsg, int status);

int main(void)

{

HANDLE hFileMapping;

char \*memstr;

hFileMapping = CreateFileMapping(INVALID\_HANDLE\_VALUE, NULL, PAGE\_READWRITE, 0, 4096, "ThisIsATest");

if (hFileMapping == NULL)

ExitSys("CreateFileMapping", EXIT\_FAILURE);

memstr = (char \*)MapViewOfFile(hFileMapping, FILE\_MAP\_READ | FILE\_MAP\_WRITE, 0, 0, 4096);

if (memstr == NULL)

ExitSys("MapViewOfFile", EXIT\_FAILURE);

printf("Process-2: Press ENTER to read from shared memory\n");

getchar();

puts(memstr);

printf("Process-2: Press ENTER to exit\n");

getchar();

UnmapViewOfFile(memstr);

CloseHandle(hFileMapping);

return 0;

}

void ExitSys(LPCSTRlpszMsg, intstatus)

{

DWORD dwLastError = GetLastError();

LPTSTR lpszErr;

if (FormatMessage(FORMAT\_MESSAGE\_ALLOCATE\_BUFFER | FORMAT\_MESSAGE\_FROM\_SYSTEM, NULL, dwLastError,

MAKELANGID(LANG\_NEUTRAL, SUBLANG\_DEFAULT), (LPTSTR)&lpszErr, 0, NULL)) {

fprintf(stderr, "%s: %s", lpszMsg, lpszErr);

LocalFree(lpszErr);

}

exit(status);

}

**UNIX/Linux Sistemlerinde Paylaşılan Bellek Alanlarının Oluşturulması**

UNIX/Linux sistemlerinde paylaşılan bellek alanlarını oluşturmak için iki grup fonksiyon vardır. Bunlardan biri en eskiden beri var olan "System V" fonksiyonları denilen gruptur. İkincisi ise 90'lı yıllarda UNIX türevi sistemlere sokulmuş olan modern fonksiyonlardır. Bunlara "POSIX shared memory" fonksiyonları denilmektedir. Aslında her iki fonksiyon grubu da POSIX standartlarında vardır. Biz burada daha sonra çıkmış olan modern paylaşılan bellek alanı fonksiyonlarını göreceğiz. Diğer fonksiyonlar "UNIX/Linux Sistem Programlama" kursunda ele alınmaktadır.

UNIX/Linux sistemlerinde paylaşılan bellek alanı oluşturma işlemi şu adımlardan geçilerek yapılır:

1) Paylaşılan bellek alanı nesnesi shm\_open fonksiyonuyla yaratılır. shm\_open hem nesneyi ilk kez yaratmak için hem de yaratılmış olanı açmak için kullanılır:

#include <sys/mman.h>

#include <sys/stat.h> /\* For mode constants \*/

#include <fcntl.h> /\* For O\_\* constants \*/

int shm\_open(const char \*name, int oflag, mode\_t mode);

Fonksiyonun birinci parametresi paylaşılan bellek alanın proseslerarası kullanımı için gereken isimdir. Bu ismin kök dizinde sanki bir dosya ismiymiş gibi verilmesi gerekmektedir. (Tabi aslında böyle bir dosya yoktur. Dizin girişinde de görülmeyecektir.) İkinci parametre açış modunu belirtir. Açış modeu open fonksiyonunda olduğu gibidir. O\_CREAT "yoksa yarat, varsa olanı aç" anlamına gelir. Bununla birlikte O\_RDONLY ya da O\_RDWR baurakları OR'lanmalıdır. O\_EXCL "nesne varsa açma, başarısızlıkla sonlan" anlamına gelmektedir. Üçüncü parametre nesnenin erişim haklarını belirtir. Bu konu ileride ele alınacaktır. Fakat tipik olarak S\_IRUSR|S\_IWUSR|S\_IRGRP|S\_IROTH biçiminde girilir. Bu üçüncü parametre nesne yaratılırken kullanılır. Nesne yaratılmışsa, yaratılmış olan açılacaksa bu argüman dikkate alınmaz. Fonksiyon başarı durumunda sıfır değerine, başarıszlık durumunda -1 değerine geri döner.

2) Paylaşılan bellek alanına ilişkin dosya için ftuncate fonksiyonuyla bir alanın ayrılması gerekir:

#include <unistd.h>

int ftruncate(int fd, off\_t length);

Fonskiyonun birinci parametresi paylaşılan bellek alanının betimleyicisi, ikinci parametresi alanın uzunluğudur. Fonksiyon başarı durumunda sıfır değerine, başarısızlık durumunda -1 değerine geri döner.

3) Paylaşılan bellek alanı nesnesi için mmap fonksiyonuyla bir sanal adres oluşturulur. İşletim sistemi bu sanal adrese ilişkin sayfayı ortak bir fiziksel sayfaya yönlendirecektir:

#include <sys/mman.h>

void \*mmap(void \*addr, size\_t length, int prot, int flags, int fd, off\_t offset);

Fonksiyonun birinci parametresi önerilen adrestir. Bu parametre NULL geçilirse sistem bize istdeği sanal adresi verir. İkinci parametre paylaşılan bellek alanının uzunluğunu belirtir. Üçüncü parametre erişim bilgilerini belirtmektedir. Bu parametre aşağıdaki bir ya da birden fazla bayrağın OR'lanmasıyla oluşturulabilir:

**PROT\_EXEC** Pages may be executed.

**PROT\_READ** Pages may be read.

**PROT\_WRITE** Pages may be written.

**PROT\_NONE**

Dördüncü parametre ya MAP\_SHARED ya da MAP\_PRIVATE geçilmelidir. Paylaşılan bellek alanı için MAP\_SHARED kullanılmalıdır. Beşinci parametre shm\_open fonksiyonundan elde edilen dosya betimleyicisidir. Son parametrede ayrılan alanın neresinden itibaren map edileceğini belirtir. Bu parametre paylaşılan alan için sıfır geçilmelidir. Fonksiyon başarı durumunda paylaşılan bellek alanın sanal adresiyle, başarısızlık durumunda MA\_FAILED değerine geri döner.

**Anahtar Notlar:** Paylaşılan bellek alanının kullanan fonksiyonlar libc kütüphanesi içerisinde değildir. Dolayısıyla linker bu kütüphaneye default olarak bakmamaktadır. Bu nedenle derleme yapılırken -lrt seçeneği ile librt kütüphanesi devreye sokulmalıdır.

4) Paylaşılan bellek alanı munmap fonksiyonuyla geri bırakılır:

#include <sys/mman.h>

int munmap(void \*addr, size\_t length);

Fonksiyonun birinci parametresi paylaşılan alanın sanal bellek adresi ikinci parametresi ise ne kadar alanın serbesy bırakılacağını belirtir. Fonksiyon başarı durumunda sıfır değerine, başarıszlık durumunda -1 değerine geri döner.

5) Paylaşılan bellek alanı nesnesi close fonksiyonuyla yok edilir.

int close(int fd);

Örnek bir paylaşılan bellek alanı uygulaması şöyle yapılabilir:

/\* shmprocess1.c \*/

#include<stdio.h>

#include<stdlib.h>

#include<string.h>

#include<unistd.h>

#include<sys/mman.h>

#include<sys/stat.h>

#include<fcntl.h>

int main(void)

{

int shmfd;

char \*memstr;

if ((shmfd = shm\_open("/this\_is\_a\_test", O\_CREAT | O\_RDWR, S\_IRUSR | S\_IWUSR | S\_IRGRP | S\_IROTH)) < 0) {

perror("shmfd");

exit(EXIT\_FAILURE);

}

if (ftruncate(shmfd, 4096) < 0) {

perror("ftruncate");

exit(EXIT\_FAILURE);

}

if ((memstr = (char \*)mmap(NULL, 4096, PROT\_READ | PROT\_WRITE, MAP\_SHARED, shmfd, 0)) == MAP\_FAILED) {

perror("mmap");

exit(EXIT\_FAILURE);

}

printf("Process-1: Press ENTER to write into shared memory\n");

getchar();

strcpy(memstr, "this is a test");

printf("Process-1: Press ENTER to EXIT");

getchar();

munmap(memstr, 4096);

close(shmfd);

return 0;

}

/\* shmprocess2.c \*/

#include<stdio.h>

#include<stdlib.h>

#include<string.h>

#include<unistd.h>

#include<sys/mman.h>

#include<sys/stat.h>

#include<fcntl.h>

int main(void)

{

int shmfd;

char \*memstr;

if ((shmfd = shm\_open("/this\_is\_a\_test", O\_CREAT | O\_RDWR, S\_IRUSR | S\_IWUSR | S\_IRGRP | S\_IROTH)) < 0) {

perror("shmfd");

exit(EXIT\_FAILURE);

}

if (ftruncate(shmfd, 4096) < 0) {

perror("ftruncate");

exit(EXIT\_FAILURE);

}

if ((memstr = (char \*)mmap(NULL, 4096, PROT\_READ | PROT\_WRITE, MAP\_SHARED, shmfd, 0)) == MAP\_FAILED) {

perror("mmap");

exit(EXIT\_FAILURE);

}

printf("Process-2: Press ENTER to read from shared memory\n");

getchar();

puts(memstr);

printf("Process-2: Press ENTER to exit\n");

getchar();

munmap(memstr, 4096);

close(shmfd);

return 0;

}

**Anahtar Notlar:** Özellikle gömülü ortamlarda bazı fiziksel bellek bölgeleri çeşitli IO uçlarına yönlendirilmiştir. Yani biz adeta bellekte o yere veri aktardığımızda aslında o IO uçlarına 5V göndermiş oluruz. Benzer biçimde belleğin o bölgesini okuduğumuzda o uçlardaki gerilim değerini okumuş oluruz. Bu tekniğe "bellek tabanlı IO tekniği (mempry mapped IO)" denilmektedir. Örneğin Raspberry PI bu tekniği kullanmaktadır.ş Raspberry PI'da fiziksel belleğim 512 megabyte'ından itibaren IO bölgesi başlar.

**Mesaj Kuyrukları Yöntemi**

Bu yöntem kursumuzda ele alınmayacaktır. "UNIX/Linux Sistem Programlama" kursunda ele alınmaktadır. Bu yöntemde ismine mesaj kuyruğu (message queue) denilen bir kuruk sistemi yaratılır. Bir proses bu kuyruğa yazma yaparken diğeri okuma yapar. Biçim bakımından boru haberleşmesine benzemektedir. Ancak mesaj kuyruklarında paket tarzı bir aktarım vardır. Bir taraf mesaj adı altında bir paket bilgi gönderir. Diğeri bunu tamamen alır. Bir kısmını almak sonra kalanını almak gibi bir durum söz konusu değildir. Boru haberleşmesi "stream" tabanlı bir haberleşme sunduğu halde mesaj kuyrukları paket tabanlı (datagram) tarzı bir haberleşme sunmaktadır.

**Aşağı Seviyeli Fonksiyonlarla Dosya İşlemleri**

Bilindiği gibi dosya sistemi tamamen işletim sisteminin kontrolü altındadır. Biz hangi dille ya da hangi kütüphaneyle çalışıyor olursak olalım eninde sınıfında dosya işlemleri işletim sisteminni sistem fonksiyonlarıyla kernel moda geçerek gerçekleştirilmektedir.

Peki neden standart C fonksiyonları varken işletim sisteminin aşağı seviyeli sistem fonksiyonlarıyla dosya işlemi yapmak isteriz? Standat C fonksiyonları her sistemde olabilecek düşük bir parametrik yapı sunmaktadır. Oysa sistemlerin kendine özgü özellikleri vardır. Örneğin Windows'ta birisi dosyayı açmışsa başka birisinin dosyayı açmasını istemeyebiliriz. Bu işlem fopen ile yapılamaz. Aygıt sürücülerle çalışırken mecburen fopen'da olmayan bazı açış modlarını kullanmak zorunda kalabiliriz. Aynı zamanda işletim sisteminin aşağı seviyeli dosya fonksiyonları bize işletim sisteminin dosya sisteminin tasarımı hakkında da daha iyi bilgiler vermektedir.

**Windows Sistemlerinde Aşağı Seviyeli Dosya İşlemleri**

Windows sistemlerinde temel dosya işlemlerini yapmak için 5 API fonksiyonu kullanılır:

CreateFile

ReadFile (ReadFileEx)

WriteFile (WriteFileEx)

CloseHandle

SetFilePointer (SetFilePointerEx)

ReadFile, WriteFile ve SetFilePointer fonksiyonlarının daha sonra Ex'li (IO Completion port özelliği ile) daha yeetenekli versiyonları oluşturulmuştur. CreateFile fonksiyonun prototipi şöyledir:

HANDLE WINAPI CreateFile(

\_\_in LPCTSTR *lpFileName*,

\_\_in DWORD *dwDesiredAccess*,

\_\_in DWORD *dwShareMode*,

\_\_in LPSECURITY\_ATTRIBUTES *lpSecurityAttributes*,

\_\_in DWORD *dwCreationDisposition*,

\_\_in DWORD *dwFlagsAndAttributes*,

\_\_in HANDLE *hTemplateFile*

);

Fonksiyonun birinci parametresi açılacak dosyanın yol ifadesidir. İkinci parametre açış modunu belirtir. Bu parametre en azından GENERIC\_READ, GENERIC\_WRITE bayraklarından yalnızca birini ya da her ikisini içermelidir. Diğer bayraklar için MSDN dokümanlarına başvurulabilir. Üçüncü parametre dosyanın paylaşım modunu belirtir. Bu parametre FILE\_SHAE\_READ, FILE\_SHARE\_WRITE ve FILE\_SHARE\_DELETE bayraklarıyla oluşturulur. Örneğin FILE\_SHARE\_READ bayrağı "başkası bu dosyayı açsın ama read modda açsın" anlamına gelir. Eğer bu parametre sıfır girilirse başkası dosyayı açamaz. Örneğin fopen fonksiyonu dosyayı her zaman FILE\_SHAREREAD|FILE\_SHARE\_WRITE|FILE\_SHARE\_DELETE modunda açmaktadır. Fonksiyonun dördüncü parametresi kernel nesnesinin güvenlik durumunu belirtir. NULL geçilebilir. Beşinci parametre yaratım modunu belirtir. Örneğin bu parametrede OPEN\_EXISTING bayrağı "dosya varsa aç, yoksa başarısz ol" anlamına gelir. Bu bayrak fopen'daki "r" moduna benzetilebilir. CREATE\_ALWAYS "dosya varsa sıfırla ve aç, dosya yoksa yarat ve aç" anlamına gelir. Bu fopen'daki "w" modu gibidir. CREATE\_NEW "dosya yoksa yarat ve aç, dosya varsa başarısız ol" anlamına gelir. TRUNCATE\_EXISTING ise "dosya yoksa başarıs ol, varsa sıfırla ve aç" anlamına gelir. Fonksiyonun altıncı parametresi yaratılacak dosyanın dosya özelliklerini belirtir. Bu parametre FILE\_ATTRIBUTE\_NORMAL geçilebilir. Bu durumda tipik bazı dosya özellikleri (ayrıntılar için MSDN'e bakınız) verilmiş olur. Eğer olan dosya açılacaksa bu parametre sıfır olarak da geçilebilir. Son parametre zaten açık bir dosyanın handle'ı elimnizde varsa onun özelliklerine sahip bir dosya yaratmak için kullanılır. Bu parametre NULL geçilebilir. Fonksiyon başarı durumunda dosyanın handle değerine başarıszlık durumunda INVALID\_HANDLE\_VALUE değerine geri döner.

Dosyadan okuma yapmak için ReadFile API fonksiyonu kullanılır:

BOOL WINAPI ReadFile(

\_\_in HANDLE *hFile*,

\_\_out LPVOID *lpBuffer*,

\_\_in DWORD *nNumberOfBytesToRead*,

\_\_out LPDWORD *lpNumberOfBytesRead*,

\_\_in LPOVERLAPPED *lpOverlapped*

);

Fonksiyonun birinci parametresi okunanacak dosyanın handle değerini, ikinci parametresi okunan bilgilerinin yerleştirileceği adresi, üçüncü parametresi okunacak byte miktarını, dördüncü parametresi okunmuş olan byte miktarını belirtir. Son parametre "overllaped io" için gereken yapıdır. Bu parametre NULL geçilebilir.

Dosyaya yazmak için WriteFile API fonksiyonu kullanılır. Bu fonksiyonun parametrik yapısı ReadFile fonksiyonundaki gibidir:

BOOL WINAPI WriteFile(

\_\_in HANDLE *hFile*,

\_\_in LPCVOID *lpBuffer*,

\_\_in DWORD *nNumberOfBytesToWrite*,

\_\_out LPDWORD *lpNumberOfBytesWritten*,

\_\_in LPOVERLAPPED *lpOverlapped*

);

Fonksiyon tam ters olarak bizim veridiğimiz adresten itibaren belirttiğimiz miktarda byte'ı dosya göstericisinin gösterdiği yerden itibaren yazar.

Dosya nihayet CloseHandle API fonksiyonuyla kapatılır. CloseHandle yalnızca dosyaları kapatmak için değil ismine "kernel nesnesi" denilen tüm handle alanlarını kapatmak için ortak kullanılan bir fonksiyondur.

Dosya göstericini konumlandırmakl için SetFilePointer API fonksiyonu kullanılmaktadır:

DWORD WINAPI SetFilePointer(

\_\_in HANDLE *hFile*,

\_\_in LONG *lDistanceToMove*,

\_\_in\_out\_opt PLONG *lpDistanceToMoveHigh*,

\_\_in DWORD *dwMoveMethod*

);

Fonksiyonun birinci parametresi dosyanın handle değerini alır. İkinci ve üçüncü parametreler konumlandırma offset'ini belirtmektedir. Üçüncü parametre NULL geçilebilir. NULL geçilmezse buraya konumlandırma offset'inin yüksek anlamlı 4 byte'ı yazılmaktadır. Son parametre konumlandırma orijinini belirtmektyedir. Bu aparemtre FILE\_BEGIN, FILE\_CURRENT ya da FILE\_END biçiminde girilmelidir. Fonksiyon başarı durumunda dosya göstericisinin yewni konumunun düşük anlamlı 4 byte'ına geri döner. Başarısızlık INVALID\_SET\_FILE\_POINTER değerine geri döner.

Aşağı seviyeli dosya kullanımına bir örnek şöyle verilebilir:

#include<stdio.h>

#include<stdlib.h>

#include<Windows.h>

void ExitSys(LPCSTR lpszMsg, int status);

int main(void)

{

HANDLE hFile;

DWORD dwBytresRead, dwBytesWritten;

char rbuf[10 + 1];

char wbuf[] = "xxxxx";

if ((hFile = CreateFile("test.txt", GENERIC\_READ|GENERIC\_WRITE, FILE\_SHARE\_READ, NULL, OPEN\_EXISTING, FILE\_ATTRIBUTE\_NORMAL, NULL)) == INVALID\_HANDLE\_VALUE)

ExitSys("CreateFile", EXIT\_FAILURE);

if (!ReadFile(hFile, rbuf, 10, &dwBytresRead, NULL))

ExitSys("ReadFile", EXIT\_FAILURE);

rbuf[dwBytresRead] = '\0';

puts(rbuf);

if (!WriteFile(hFile, wbuf, strlen(wbuf), &dwBytesWritten, NULL))

ExitSys("ReadFile", EXIT\_FAILURE);

CloseHandle(hFile);

return 0;

}

void ExitSys(LPCSTRlpszMsg, intstatus)

{

DWORD dwLastError = GetLastError();

LPTSTR lpszErr;

if (FormatMessage(FORMAT\_MESSAGE\_ALLOCATE\_BUFFER | FORMAT\_MESSAGE\_FROM\_SYSTEM, NULL, dwLastError,

MAKELANGID(LANG\_NEUTRAL, SUBLANG\_DEFAULT), (LPTSTR)&lpszErr, 0, NULL)) {

fprintf(stderr, "%s: %s", lpszMsg, lpszErr);

LocalFree(lpszErr);

}

exit(status);

}

**UNIX/Linux Sistemlerinde Aşağı Seviyeli Dosya İşlemleri**

UNIX/Linux sistemlerinde aşağı seviyeli dosya işlemleri için birkaç POSIX fonksiyonları kullanılmaktadır. Bu fonksiyonlar doğrudan işletim sisteminin sistem fonksiyonlarını çağırırlar. UNIX/Linux sistemlerindeki temel POSIX dosya fonksiyonları şunlardır:

open

read

write

close

lseek

open fonksiyonun prototipi şöyledir:

#include <sys/stat.h>

#include <fcntl.h>

int open(const char \*pathname, int flags, ...);

Fonksiyon iki ya da üç parametreli kullanılır. Fonksiyonun birinci parametresi açılacak dosyanın yol ifadesini belirtir. İkinci parametre açış modunu belirtmektedir. Açış modu en azından şunlardan yalnızca birini içermek zorudadır:

O\_RDONLY

O\_WRONLY

O\_RDWR

Buna ilaveten aşağıdaki bayraklar da OR'lanarak açış modunda kullanılabilir:

O\_CREAT: Bu modda dosya yoksa yaratılır ve açılır, varsa olan dosya açılır.

O\_TRUNC: Bu modda dosya açılırken varsa aynı zamanda sıfırlanmaktadır. Örneğin O\_CREAT|O\_TRUNC modu "dosya yoksa yarat ve aç, varsa sıfırla ve aç" anlamına gelir. Yani fopen'daki "w" modudur.

O\_APPEND: Bu modda dosyadan okuma yapılabilir. Ancak her yazma işlemi önce dosya göstericisinin dosyanın sonuna çekilmesiyle yapılmaktadır. Yani her write işlemi sona eklemeye yol açar.

O\_EXCL: Bu modda yaratım sırasında dosya varsa open başarısız olur. Yani olmayan bir dosyayı yaratmanın garanti altına alınması için bu mod kullanılmaktadır. O\_EXCL tek başına kullanılamaz. Mutlaka O\_CREAT ile birlikte kullanılmalıdır.

Diğer modlar için ilgili dokümanlara başvurabilirsiniz.

Fonksiyon fonksiyonun son parametresi yaratma riski varsa kullnılmalıdır. Yani ikinci parametrede O\_CREAT belirtilmişse üçüncü parametre girilmelidir. Üçüncü parametre dosyanın erişim haklarını belirtir. Erişim hakları ls -l komutunda aşağıdaki gibi görüntülenmektedir:

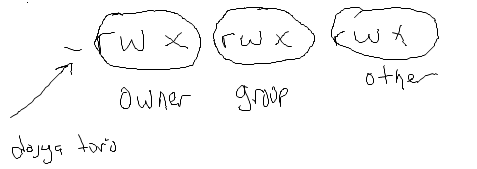
-rwxrwxrwx

Erişim hakları için toplam 9 karakter kullanılır. En soldaki karakter dosyanın türünü belirtir. '-' normal dosya (regular file) anlamına gelir. 'd' dizin anlamına gelir. Diğer erişim hakları 3'lük 3 grup oluşturmaktadır:

UNIX/Linux sistemlerinde her prosesin bir "user id" ve "group id" değeri vardır. Bu değerler sayısaldır ve prosesin kontrol bloğunda saklanır. Üst prosesten alt prosese fork işlemi sırasında aktarılır ve exec işlemi sırasında değişmez. Bu id değerleri login programı tarafından password başarılı girilmişse /etc/passwd ve /etc/group dosyalarına bakılarak set edilir. Ayrıca sistem "user id"leri "user name"lerle, "group id"leri de "group name"lerle ilişkilendirmiştir. Bu ilişkilendirme yine /etc/passwd ve /etc/gropu dosyalarında belirtilmektedir. Böylece konuşurken ya da görüntülerken "user id"ler yerine "user name"ler, "group id"ler yerine "group name"ler söz konusu edilir. Tabi işletim sisteminin çekirdeği "user id" ve "group id" sayısal değerleriyle çalışmaktadır.

UNIX/Linux sistemlerinde her dosyanın ayrıca bir "user id" ve "group id" si vardır.

Bir dosyanın erişim haklarının ilk üç karakterine "owner", sonraki üç karakterine "group" ve sonraki üç karakterine "other" denilmektedir:



İlgili erişim hakkı varsa r, w, x sembolleri yoksa '-' sembolü bulunmaktadır. Örneğin:

- rw- r-- ---

Burada owner'a "rw", gruba yalnızca "r" ve other'a hiçbir hak verilmemiştir.

Erişim hakları open fonksiyonu atrafından kontrol edilmektedir. Ve algoritması şöyledir (maddeleri else-if olarak değerlendiriniz):

1) Önce open kendisini çağıran prosesin user id'si 0 mı diye bakar. Sıfır numaralı user id'ye sahip prosese "super user" ya da "priviled user" ya da "root user" denilmektedir. Eğer dosyaya erişmek isteyen prosesin user id'si 0 ise erişim kabul edilir. 0 numaralı user id'ye sahip proses hiçbir engele takılmaz.

2) open fonksiyonu dosyaya erişmek isteyen prosesin user id'si dosyanın user id'si ile aynı mı diye bakar. (Yani erişim yapmak isteyen proses dosyanın sahibi midir?). Eğer böyleyse ilk üçlük (owner) erişim hakkını dikkate alır.

3) open fonksiyonu dosyaya erişmek isteyen prosesin group id'si dosyanın group id'si ile aynı mı diye bakar. (Yani erişim yapmak isteyen proses dosyanın grubuyla aynı gruptan mıdır?). Eğer böyleyse ikinci üçlük (group) erişim hakkını dikkate alır.

4) Bu durumda open other erişim hakkını dikkate alır.

Örneğin aşağıdaki haklara sahip "x.txt" dosyasını open ile açmaya çalışalım:

- rw- r-- r--

if ((fd = open("x.txt", O\_RDWR)) < 0) {

perror("open");

exit(EXIT\_FAILURE);

}

Eğer bu çağrıyı dosyanın sahibi yaparsa open başarılı olur. Aynı gruptan kişi ya da herhangi bir yaparsa başarısız olur.

Peki bir dosyanın user id'sinin group id'sinin ve erişim haklarının nasıl olacağına kim nasıl karar vermektedir? İşte aslında bunlara dosyayı open fonksiyonuyla yaratn proses karar vermektedir. Şöyle ki:

1) Yeni yaratılan dosyaının user id'si her zaman onu yaratan prosesin user id'si olarak alınır. (Yani dosyayı hangi user yaratmışsa dosyanın user id'si de o olacaktır.)

2) Yeni yaratılan dosyanın group id'si POSIX standartlarında iki seçenekten biri olarak verilebilir:

- Dosyayı yaratan prosesin group id'si olarak

- Dosyanın içinde bulunduğu dizinin group id'si olarak

Maalesef bu durum POSIX standartları oluşturulduğu sırada izlenen iki farklı stratejiden kaynaklanmaktadır. Linux'ta default olarak yeni yaratılan dosyanın group id'si onu yaratan prosesin group id'si olarak alınmaktadır.

3) Dosyanın erişim hakları dosya yaratılırken open fonksiyonun üçüncü parametresinde verilir. Bu üçüncü parametre aşağıdaki sembolik sabitlerin OR'lanmasıyla oluşturulur

S\_IRUSR

S\_IWUSR

S\_IXUSR

S\_IRGRP

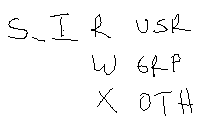
S\_IWGRP

S\_IXGRP

S\_IROTH

S\_IWOTH

S\_IXOTH



Örneğin:

if ((fd = open("x.txt", O\_RDWR|O\_CREAT, S\_IRUSR|S\_WUSR|S\_RGRP)) < 0) {

perror("open");

exit(EXIT\_FAILURE);

}

Open fonksiyonu başarı durumunda dosya betimleyicisi (file descriptor) denien handle değerine başarısızlık durununda -1 değerine geri döner.

**Anahtar Notlar:** Bilgisayar için iki önemli online sözlük kullanılabilir:

1) FOLDOC: Burada bilgisayara ilişkin terimler ve anlamları bulunmaktadır (http://foldoc.org/)

2) DADS: Bu veri yapıları ve algoritnmalara ilişkin sözlğktür (http://xlinux.nist.gov/dads/)

UNIX/Linux sistemlerinde dosyadan okuma yapmak için read isimli POSIX fonksiyonu, dosyaya yazma yapmak için write isimli POSIX fonksiyonu kullanılmaktadır.

#include <[unistd.h](http://linux.die.net/include/unistd.h)>

ssize\_t read(int fd, void \*buf, size\_t count);

ssize\_t write(int*fd*, const void \**buf*, size\_t*count*);

Fonksiyonların birinci parametreleri dosya betimleyicisini alır. İkinci parametreler bellekteki transfer adresini son parametreler ise yazılacak ya da okunacak byte sayısını belirtir. read fonksiyonu okuyabildiği byte sayısına geri döner. Dosyada kalandan daha fazla byte okunmak istenebilir. Bu durumda read okuyabildiği kadarını okur. write fonksiyonu da yazabildiği byte sayısıyla geri döner. Her iki fonksiyon da başarısızlık durumunda -1 değerine geri dönmektedir.

UNIX/Linux sistemlerinde dosya göstericisini konumlandırmak için lseek fonksiyonu kullanılmaktadır.

#include <unistd.h>

off\_t lseek(int fd, off\_t offset, int whence);

Fonksiyonun birinci parametresi dosya betimleyicisini, ikinci parametresi konumlandırmaoffsetini, üçüncü parametresi ise konumlandırma orijinini belirtir. (Genel kullanımı fseek gibidir.). Fonksiyon başarı durumunda konumlanma offset'ine başarısızlık durumunda -1 değerine geri döner. lseek ile dosya göstericisi dosyanın uzunluğunun ötesine konumlandırılabilir. Bu durumda ilk yazma yapıldığında dosya deliği (file hole) oluşur.

Nihayet dosya close fonksiyonuyla kapatılır:

#include <[unistd.h](http://linux.die.net/include/unistd.h)>

int close(int fd);

#include<stdio.h>

#include<stdlib.h>

#include<fcntl.h>

#include<sys/types.h>

#include<unistd.h>

int main(void)

{

int fd;

char str[] = "This is a test\n";

if ((fd = open("test.txt", O\_RDWR | O\_CREAT, S\_IRUSR | S\_IWUSR | S\_IRGRP)) < 0) {

perror("open");

exit(EXIT\_FAILURE);

}

lseek(fd, 0, SEEK\_END);

if (write(fd, str, sizeof(str)) < 0) {

perror("write");

exit(EXIT\_FAILURE);

}

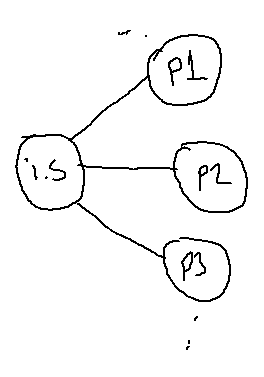
close(fd);

return 0;

}

**İşletim Sistemlerinde Zaman Paylaşımlı Çalışma**

Eskiden thread'ler yokken her prosesin tek bir akışı vardı. İşte örneğin C programlama dilinde tipik olarak proses akışı main fonksiyonundan başlatılmaktadır. Pekiyi çokprosesli işletim sistemlerinde tek bir CPU olduğu durumda birden fazla program nasıl aynı anda çalışabilmektedir? Aslında programlar tek CPU'lu bir sistemde aynı anda çalışmamaktadır. Bir proses CPU'ya atanır. Bir süre çalıştırılır. Sonra çalışmasına ara verilir. Diğeri yine bir süre çalıştırılır. Çalışma bu biçimde devam ettirilir. Sonra prosesler kaldıkları yerden hep böyle çalıştırılırlar. Dışarıdan bakıldığında sanki, bunlar aynı anda çalışıyormuş gibi bir izlenim edinilmektedir. Fakat aslında "biraz ondan biraz bundan" biçiminde zaman paylaşımlı bir çalışma söz konusudur.



Bir prosesin parçalı çalışma süresine quanta süresi ya da quantum denilmektedir. Örneğin Windows sistemlerinde tipik quanta süresi 20 ms. civarındadır. UNIX/Linux sistemlerinde 60 ms. gibi quanta süreleri tercih edilmektedir. Quanta süreleri işletim sisteminden işletim sistemine hatta versiyondan versiyona değişebilir. Sistem yöneticisini genel olarak bunu ayarlama durumunda değildir.

Bir prosesin quanta süresi dolduğunda çalışmasına ara verilip diğer prosese geçilmesi sürecine "proseslerarası geçiş (process switch/task switch)" ya da daha genel olarak "context switch" denilmektedir. Şüphesiz proseslerarası geçiş sırasında bir zaman kaybı oluşmaktadır.

Pekiyi quanta süresi çok uzun seçilirse bunun etkileri ne olur? Quanta süresi çok uzun seçilirse interaktivite azalır. Fakat birim zamanda yapılan gerçek iş miktarı (throughput) yükselir. Quanta süresi çok kısa olursa bu durumda tersine interaktivite çok yüksek olur fakat birim zamanda yapılan iş miktarı (throughput) düşer. Çünkü proseslerarası geçiş işleminin de belli bir maaliyeti vardır.

Windows ve UNIX/Linux sistemleri "preemptive" sistemlerdir. Bir prosesin quata süresi dolduğunda akışın o anda nerede bulunulursa bulunulsun kopartılarak proseslerarası geçiş yapıldığı sistemlere "preemptive" sistemler denilmektedir. Fakat preemptive olmayan çokprosesli sistemler de vardır. Bu tür sistemlere İngilizce "non-preemptive" sistemler ya da "cooperative multitask" sistemler denilmektedir. Örneğin Windows 3.1, PalmOS gibi sistemler böyleydi. Bu sistemlerde proses kendi isteğiyle akışı bırakır ve proseslerarası geçiş oluşur. Bu tür sistemlerin gerçekleştirilmesi daha kolaydır fakat bunlarda bir proses sistemi tekeli altına alabilmektedir.

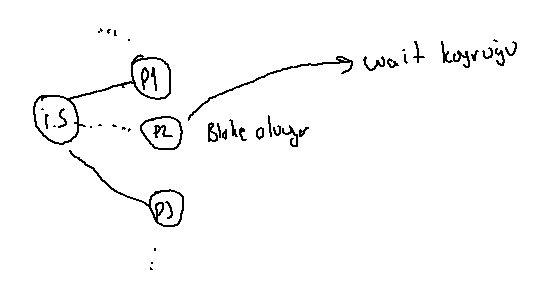
Peroseslerarası geçiş donanım kesmeleriyle yapılmaktadır. Örneğin PC mimarisinde tipik olarak IRQ0 timer devresi kurulur. Bu devre zaman dolduğunda CPU'ya ileti gönderir. Böylece CPU kesme konumuna geçer ve proseslerarası geçişi yapar.

Proseslerin çizelgelenmesi birden fazla CPU ya da çekirdek olduğu durumda çok benzer yapılmaktadır. Bu durum birden fazla gişenin bulunduğu servis sistemlerine benzetilebilir. İşletim sistemleri birden fazla CPU ya da çekirdeğin bulunduğu durumlarda işlemciler ya da çekirdekler için ayrı kuyruklar oluştururlar. Yine her işlemci ya da çekirdek zaman paylaşımlı biçimde çalışmaktadır. Tabi bu durum toplam performansı yükseltir.

**Anahtar Notlar:** CPU'lar belli bir yıla kadar sürekli hızlandı (Moore yasası). Fakat belli bir düzeyden sonra artık fiziksel sorunlardan dolayı CPU'lar hızlandırılması çok zorlaştı. Bu durumda hızlanmayı sağlamak için çok çekirdekli işlemciler yapılmaya başlandı. İki çekirdekli bir CPU aslında iki ayrı CPU'nun tek entegre devre halinde üretilmesidir. Zaten eskiden birden fazla CPU'nun takılabildiği board'lar vardı. Çok çekirdekli sistemler önce "hyper threading" teknolojisi ile başlatılmıştır. Hyper threding'li bir CPU gerçek anlamda iki CPU değildir. Bunun bazı parçalarından iki tane vardır. Fakat bazı parçaları bir tanedir. İşletim sistemleri "hyper threading"li CPU'ları iki CPU gibi görmektedir. Ancak bu CPU'lar gerçek iki CPU (yani gerçek iki çekirdek) kadar performanslı değildir. Şimdilerde Intel hem birden fazla çekirdek hem de her çekirdekte hyper threading uygulayabilmektedir. Örneğin bu yazının yazıldıuğı bilgisayarda Interl'in I7-3770 numaralı CPU'su kullanılmaktadır. Bu entegre devre 4 gerçek çekirdeğe (yani farklı CPU'ya) sahiptir ve her çekirdek ayrıca "hyper threading" yapılmıştır. Böylece Windows ve Linux bu entegre devreyi sanki 8 işlemcili bir sistem gibi görmektedir.

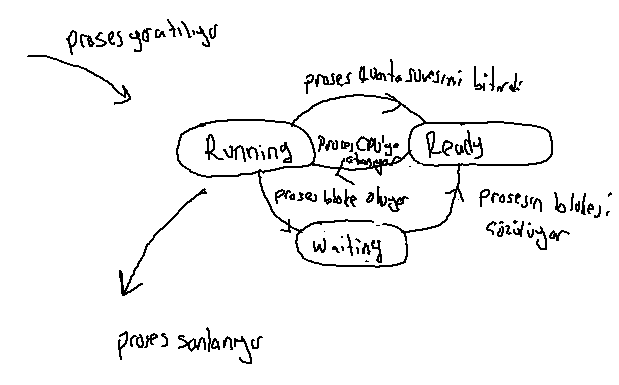
**Bloke Kavramı**

Bir proses çalışırken dışsal bir olayı başlattığında (örneğin disk işlemi, klavye okuması, soket okuması vs. gibi) işletim sistemi prosesi CPU zamanı harcanmasın diye geçici olarak çizelge dışına çıkartır ve olayı kendisi arka planda kesme (interrupt) tekniğiyle izler. Bu sırada sanki proses hiç çizelgede değilmiş gibi bekletilir. İşlem bittiğinde işletim sistemi yeniden prosesi çizelgeye yerleştirir. Sonuçta proses yine olay bitene kadar beklemiş olur fakat boşuna CPU zamanı harcanmamıştır. İşte bir prosesin bir işlem bitene kadar ya da gerçekleşene kadar çizelge dışına çıkartılarak bekletilmesine prosesin bloke olması (blocking) denilmektedir. Örneğin Sleep gibi fonksiyonlar da aslında meşgul bir döngüde sürekli bekleme yapmazlar. Bunlar da bloke edilerek bekletilirler. Böylece bir sistemde yüzlerce proses olabilir fakat bunların çoğu belli bir olayı bekler durumdadır. Yani çok az proses aktif olarak belli bir anda CPU'yu kullanma eğilimindedir.



Aslında prosesin bir quanta süresinin tamamını harcaması çok nadirdir. Örneğin quanta süresi 20 ms. olsun. Pek çok proses daha birkaç milisaniye içerisinde bir IO olayına girer ve uzun süre bekler.

Bir prosesin yaşam döngüsü tipik olarak şöyledir:



Burada Running prosesin o anda CPU'ya atanmış olduğunu gösteriyor. Proses quanta süresini normal olarak doldurduğunda çizelgede bekletilir. Bu durum şekilde "Ready" ile temsil edilmektedir. Proses çalışırken bloke olursa çizelgeden çıkarılır. Bu durum da şekilde "Waiting" ile belirtilmiştir.

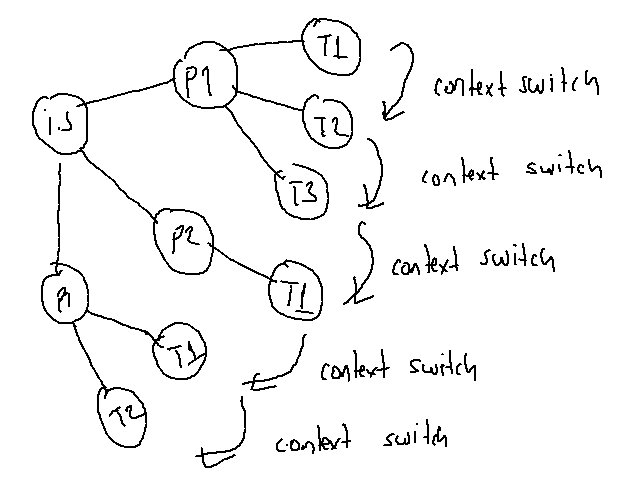
Bir proses bloke olduğunda işletim sistemi onu ismine "wait kuyruğu (wait queue)" denilen bir kuyrukta bekletir. Sonra olay gerçekleşince oradan onu alarak yeniden çizelge kuyruğuna koyar. Genellikle işletim sistemleri her olay için ayrı birer wait kuruğu oluşturmaktadır.

Pekiyi bir proses ne kadar zaman CPU harcayıp ne kadar zaman wait kuyruğunda bekler? Tabi bu prosesinden prosesine değişir. Genel olarak çok CPU kullanan fakat az IO yapan proseslere "CPU yoğun (CPU bound)" prosesler, çok IO yapıp az CPU kullanan proseslere de "IO yoğun (IO bound)" prosesler denir. Genellikle prosesler IO yoğun olma eğilimindedir. Örneğin matematiksel hesaplamalar yapan bir proses CPU yoğun, veritabanı işlemi yapan bir proses IO yoğundur. O halde bir sistemde yüzlerce proses olsa da aslında bunların çoğu uykuda yani wait kuyruğunda bekliyor durumdadır.

**Thread'lerle İşlemler**

Thread sözcüğü etimolojik olarak akışların ipliğe benzetilmesinden hareketle "iplik" sözcüğünden gelmektedir. Thread'ler bir prosesin bağımsız çizelgelenen akışlarını belirtir. Proses çalışmakta olan programın tamamını kavramsal olarak anlatmaktadır. Thread ise yalnızca bir akış belirtir. Dolayısıyla thread'ler proses kavramının içerisinde yer alır. Thread'lerin ilk ciddi denemeleri 80'li yıllarda yapılmıştır. Fakat 90'lı yıllarda işletim sistemlerine gerçek anlamda sokulmuştur. Örneğin DOS'ta thread yoktu. Windows 3.1 sistemleri de thread'li sistemler değildir. Microsoft'un ilk thread'li sistemi Windows NT (1993) ve sonra Windows 95 (1995)'tir. Linux'un ilk versiynlarında thread'ler yoktu. 2.0'dan itibaren thread'li çalışma Linux sistemlerine sokulmuştur.

Çokthread'li işletim sistemlerinde çizelgelenen elemanlar artık prosesler değil thread'lerdir. Yani her quanta süresi dolduğunda bir thread'in çalışmasına ara verilir diğer bir thread çalıştırılır. Bu sistemlerde thread'ler bloke olur. Yani bir prosesin bir thread'i bloke olmuşken diğerleri çalışmaya devam eder.



Artık çizelgeleyici için prosesler değil, thread'ler çizelgelenmektedir. Bu durumda thread'siz sistemler tek thread'li sistemlere benzetilebilir.

Çokthread'li işletim sistemlerinde proses çalışmaya bir thread'le başlar. Yani proses yaratıldığında bir thread de yaratılmış durumdadır. Buna prosesin ana thread'i (main thread) denir. Diğer thread'ler işletim sisteminin sistem fonksiyonlarıyla (yani Windows'ta API fonksiyonlarıyla, UNIX/Linux sistemlerinde POSIX fonksiyonlarıyla) yaratılırlar.

Thread'lere neden gereksinim duyulmaktadır? Bu gereksinim birkaç maddeyle özetlenebilir:

1) Thread'ler arka plan olayları izlemek için iyi bir araç oluşturmaktadır. Örneğin hem bir işi yaparken hem de ekranın sağ üst köşesine saati basmak isteyelim. Saati ne zaman basacağız. Her işlemin arasında basmamız gerekir. Peki klavye ya da disk okumalarında ne olacak? Ya da hem bir işi yaparken hem de arka planda dışsal bir olayı (örneğin seri portu, ya da bir termometreyi) izleyecek olalım. Eskiden bu tür işlemleri yapmak için tüm program organizasyonunu değiştirmek gerekiyordu. Yani bu tür işlemler çok zor yapılıyordu. Halbuki thread'li sistemlerde bir thread yaratıp bu arka plan olayı bu thread' havale edebiliriz. Böylece diğer thread normal işlemini yapabilir.

2) Thread'ler bir programı hızlandırmak için kullanılabilir. Yani biz programımızda çok thread kullanırsak toplamda daha fazla CPU zamanı çekeriz.

3) Thread'ler blokeli IO işlemlerinde yoğun olarak kullanılmaktadır. Yani bir IO işlemi başlattığımızda (örneğin boru ya da soket gibi) belli bir süre bloke oluruz. Bu durumda gerekli olan başka şeyleri yapamayız. İşte IO işlemleri thread'lere yaptırılırsa blokeden yalnızca o thread etkilenir.

4) Thread'ler paralel programlama için mecburen kullanılmaktadır. Paralel programları bir işi parçalara ayırarak onu aynı anda birden fazla işlemci ya da çekirdeğe atayarak gerçekleştirme sürecine denilmektedir.

5) Thread'ler GUI programlama modelinde bazen mecburen kullanılmak zorundadır. Örneğin bir mesaj geldiğinde bir işi uzatırsak kuyrukta sıradaki mesajları işleyemeyiz. İşte uzun sürebilecek işlemler thread'lere havale edilebilir.

Windows'ta thread işlemleri API fonksiyonlarıyla, UNIX/Linux sistemlerinde POSIX fonksiyonlarıyla yapılmaktadır. Tabi bu fonksiyonlar da sistem çağrısı yaparak kernel moda geçmektedir. POSIX'in standart thread fonksiyonlarının hepsi pthread\_xxx biçiminde isimlendirilmiştir. POSIX thread kütüphanesine pthread kütüphanesi de denilmektedir.

**Therad'lerin Yaratılması**

Yukarıda da belirtildiği gibi proses çalışmasına tek bir thread'le başlar. Buna prosesin ana thread'i (main thread) denilmektedir. Daha sonra thread2ler işletim sisteminin sistem fonksiyonlarıyla ya da bunlara çağıran kütüphane fonksiyonlarıyla yaratılır.

Windows sistemlerinde thread'ler CreateThread API fonksiyonuyla yataılmaktadır:

HANDLE WINAPI CreateThread(

\_\_in LPSECURITY\_ATTRIBUTES *lpThreadAttributes*,

\_\_in SIZE\_T *dwStackSize*,

\_\_in LPTHREAD\_START\_ROUTINE *lpStartAddress*,

\_\_in LPVOID *lpParameter*,

\_\_in DWORD *dwCreationFlags*,

\_\_out LPDWORD *lpThreadId*

);

Fonksiyonun birinci parametresi thread'in güvenlik bilgilerine ilişkindir. Bu parametre NULL geçilebilir. İkinci parametre thread'in stack uzunluğunu belirtmektedir. (Thread'lerin stackleri sonraki başlıkta ele alınacaktır.) Bu parametre 0 geçilirse bu durumda PE (Portable Executable Format) formatında belirtilen değer olarak olarak alınır. Microsoft linker'ları bu değeri default 1 MB olarak almaktadır. Üçüncü parametre thread akışının başlatılacağı fonksiyonun adresini alır. LPTHREAD\_START\_ROUTINE türü aşağıdaki gibi typedef edilmiştir:

typedefDWORD (WINAPI \*LPTHREAD\_START\_ROUTINE)(LPVOID lpThreadParameter);

Görüldüğü gibi thread fonksiyonunun geri dönüş değeri DWORD parametresi LPVOID (yani void \*) olmak zorundadır. Ayrıca thread fonksiyonları \_\_stdcall çağırma biçimine sahip olmak zorundadır. WINAPI aşağıdaki gibi define edilmiştir:

#define WINAPI \_\_stdcall

Yani örneğin aşağıdaki fonksiyon bir thread fonksiyonu olabilir:

DWORD \_\_stdcall ThreadProc(void \*param)

{

...

}

Fonksiyonun dördüncü parametresi thread yaratıldığında thread'e geçirilecek argümanı belirtir. sonraki parametre yaratıma ilişkin bazı özellikleri belirtir. Bu parametre sıfır geçilebilir. Ya da CREATE\_SUSPENDED geçilebilir. Bu durumda thread yaratıldığında henüz çalışmıyor durumdadır. Çalıştırmak için ResumeThread API fonksiyonunun çağrılması gerekir. Fakat bu parametre sıfır geçilirse thread yaratılır yaratılmaz çalışır. Fonksiyonun son parametresi thread id'sinin yerleştirileceği DWORD nesnenin adresini alır. Thread'lerin id'leri ve handle değerleri vardır. Fonksiyon başarı durumunda thread'in handle değerine, başarısızlık durumunda NULL adrese geri döner.

Windows sistemlerinde örnek bir thread yaratımı şöyle yapılabilir:

#include<stdio.h>

#include<stdlib.h>

#include<Windows.h>

void ExitSys(LPCSTR lpszMsg, int status);

DWORD\_\_stdcall ThreadProc(void \*param);

int main(void)

{

DWORD dwThreadId;

HANDLE hThread;

if ((hThread = CreateThread(NULL, 0, ThreadProc, NULL, 0, &dwThreadId)) == NULL)

ExitSys("CreateThread", EXIT\_FAILURE);

for (int i = 0; i < 10; ++i) {

printf("Main Thread: %d\n", i);

Sleep(1000);

}

}

DWORD\_\_stdcall ThreadProc(void \*param)

{

for (int i = 0; i < 10; ++i) {

printf("New Thread: %d\n", i);

Sleep(1000);

}

return 0;

}

void ExitSys(LPCSTRlpszMsg, intstatus)

{

DWORD dwLastError = GetLastError();

LPTSTR lpszErr;

if (FormatMessage(FORMAT\_MESSAGE\_ALLOCATE\_BUFFER | FORMAT\_MESSAGE\_FROM\_SYSTEM, NULL, dwLastError,

MAKELANGID(LANG\_NEUTRAL, SUBLANG\_DEFAULT), (LPTSTR)&lpszErr, 0, NULL)) {

fprintf(stderr, "%s: %s", lpszMsg, lpszErr);

LocalFree(lpszErr);

}

exit(status);

}

UNIX/Linux sistemlerinde thread'ler pthread\_create isimli POSIX fonksiyonuyla yaratılmaktadır:

#include <pthread.h>

int pthread\_create(

pthread\_t \*thread,

const pthread\_attr\_t \*attr,

void \*(\*start\_routine) (void \*),

void \*arg

);

Fonksiyonun birinci parametresi thread'in id değerinin yerleştirileceği pthread\_t nesnesinin adresini alır. İkinci parametre thread özelliklerinin bulunduğu yapı nesnesinin adresini almaktadır. Bu parametre NULL geçilirse thread default özelliklerle yaratılır. Üçüncü parametre thread akışının başlatılacağı fonksiyonu belirtmektedir. Therad fonksiyonunun geri dönüş değeri ve paarametresi void \* olmak zorundadır. Fonksiyonun son parametresi thread fonksiyonuna geçirilecek argümanı belirtir. Fonksiyon başarı durumunda sıffır değerine başarıszlık durumunda bizzat hata kodunun kendisine geri döner. POSIX'te thread fonksiyonları errno değerini set etmemektedir. Bu nedenle hata kodunu alıp onu strerror fonksiyonuna sokarak hata yazısının elde edilmesi uygun olur. Örneğin:

pthread\_t tid;

int result;

if ((result = pthread\_create(&tid, NULL, thread\_proc, NULL)) != 0) {

printf("pthread\_create: %s\n", strerror(result));

exit(EXIT\_FAILURE);

}

UNIX/Linux sistemlerinde thread yaratan örnek program şöyle olabilir:

#include<stdio.h>

#include<stdlib.h>

#include<string.h>

#include<unistd.h>

#include<pthread.h>

void \*thread\_proc(void \*param);

int main(void)

{

pthread\_t tid;

int result;

int i;

if ((result = pthread\_create(&tid, NULL, thread\_proc, NULL)) != 0) {

printf("pthread\_create: %s\n", strerror(result));

exit(EXIT\_FAILURE);

}

for (i = 0; i < 10; ++i) {

printf("Main thread: %d\n", i);

sleep(1);

}

return 0;

}

void \*thread\_proc(void \*param)

{

int i;

for (i = 0; i < 10; ++i) {

printf("New thread: %d\n", i);

sleep(1);

}

returnNULL;

}

UNIX/Linux sistemlerinde thread fonksiyonları libc kütüphanesinde değildir, pthread isimli ayrı bir kütüphanededir. Linker yalnızca libcs kütüphanesine baktığı için thread fonksiyonlarını bulamaz. Bu nedenle derlemeyi yaparken linker için linker'ın phread kütüphanesine bakmasını sağlamak gerekir. Bu işlem -lpthread seçeneiğyle sağlanır. Yani thread kullanan bir program tipik olarak şöyle derlenerek link edilmelidir:

gcc -o sample sample.c -lpthread

**Thread'lerin Sonlanması**

Bir thread çeşitli biçimlerd sonlanabilir. En normal sonlanma thread fonksiyonunun sonlanmasıyla sonlanmadır. Thread fonksiyonu sonlandığında thread'in çalışması da biter. Thread'ler Windows sistemlerinde ExitThread API fonksiyonuyla UNIX/Linux sistemlerinde pthread\_exit fonksiyonuyla sonlandırılabilir. Bu fonksiyonlar o andaki thread akışını sonlandırmaktadır. Örneğin:

void Foo(void)

{

for (int i = 0; i < 10; ++i) {

printf("New Thread: %d\n", i);

if (i == 5)

ExitThread(0);

Sleep(1000);

}

}

DWORD\_\_stdcall ThreadProc(void \*param)

{

Foo();

return 0;

}

ExitThread fonksiyonun prototipi şöyledir:

VOID WINAPI ExitThread(

\_in DWORD *dwExitCode*

);

Fonksiyonun parametresi thread'in exit kodudur. Thread'lerin de tıpkı prosesler gibi exit kodları vardır. Thread fonksiyonunun geri dönüş değeri ya da bu sonlandırıcı fonksiyonlar bu exit kodunu belirtirler. pthread\_exit fonksiyonunun prototipi de şöyledir:

#include <pthread.h>

void pthread\_exit(void \*retval);

Fonksiyon yine parametre olarak thread'in exit kodunu almaktadır. UNIX/Linux sistemlerinde threadler'in exit kodları void \* türündendir. Halbuki Windows sistemlerinde DWORD türdendir.

Bir thread başka bir thread tarafından zorla sonlandırılabilir. Bu işlem Windows'ta TerminateThread API fonksiyonuyla, UNIX/Linux sistemlerinde pthread\_cancel fonksiyonuyla yapılır. TermşnateThread thread'i o anda sonlandırır. Ancak pthread\_cancel sonlandırmayı hemen yapmaz. Sonlandırma thread akışı "cancellation point" denilen bazı POSIX fonksiyonlarına girdiğinde oralarda yapılır. POSIX standartlarında hangi fonksiyonların cancellation point olduğu belirtilmiştir.

Nihayet bir proses sonlandığında prosesin bütün thread'leri de sonlanır. Yani main bittiğinde ya da exit fonksiyonu çağrıldığında o and açalışmakta olan bütün thread'ler de sonlandırılır. Örneğin sık yapılan bir hata şudur: Programcı bir grup thread yaratmıştır fakat ana thread'i bekletmemiştir. Böylece ana thread main'i bitirir. Proses biter, diğer thread'ler de sonlanmış olur.

Ayrıca hem Windows'ta hem de UNIX/Linux sistemlerinde ana thread'in diğer thread'lerden hiçbir farkı yoktur. Ana thread sonlandığı halde diğer thread'ler devam edebilir. Pekiyi bu durumda proses ne zaman sonlanır. İşte işletim sistemi eğer exit çağrılmamışsa son thread de sonlandığında prosesi sonlandırır. Örneğin:

#include<stdio.h>

#include<stdlib.h>

#include<Windows.h>

void ExitSys(LPCSTR lpszMsg, int status);

DWORD\_\_stdcall ThreadProc(void \*param);

int main(void)

{

DWORD dwThreadId;

HANDLE hThread;

if ((hThread = CreateThread(NULL, 0, ThreadProc, NULL, 0, &dwThreadId)) == NULL)

ExitSys("CreateThread", EXIT\_FAILURE);

ExitThread(0); /\* main thread sonlandiriliyor \*/

return 0;

}

DWORD\_\_stdcall ThreadProc(void \*param)

{

for (int i = 0; i < 10; ++i) {

printf("New Thread: %d\n", i);

Sleep(1000);

}

return 0;

}

void ExitSys(LPCSTRlpszMsg, intstatus)

{

DWORD dwLastError = GetLastError();

LPTSTR lpszErr;

if (FormatMessage(FORMAT\_MESSAGE\_ALLOCATE\_BUFFER | FORMAT\_MESSAGE\_FROM\_SYSTEM, NULL, dwLastError,

MAKELANGID(LANG\_NEUTRAL, SUBLANG\_DEFAULT), (LPTSTR)&lpszErr, 0, NULL)) {

fprintf(stderr, "%s: %s", lpszMsg, lpszErr);

LocalFree(lpszErr);

}

exit(status);

}

Burada sorun şudur: Ana thread sonlanmış fakat exit fonksiyonu çağrılmamıştır (main bitince exit'in çağrıldığını anımsayınız). Yaratılan thread de sonlanmıştır (Bir thread fonksiyonu sonlandığında arka planda exit çağrılmaz ExitThread ya da pthread\_exit çağrılmaktadır). Bu durumda ortada hiçbir akış kalmadığı halde proses yaşayacak mıdır? İşte işletim eğer exit çağrılmamışsa son thread sonlandığında prosesi otomatik olarak sonlandırır.

**Thread'lerin Birbirleriyle Haberleşmesi**

Thread'ler aynı prosesin parçaları olduğuna göre bunların birbirleriyle haberleşmesi global nesneler ya da heap yoluyla yapılabilir. Global nesneler Prosesin Data veya BSS denilen bölgelerinde yaratılmaktadır. Buraya da tüm thread'ler erişebilir. Heap de thread'ler arasında ortak kullanılan tgekil bir alandır. Yani iki thread heap'te tahsis edilmiş nesnelere erişebilirler. Ancak thread'lerin stack'leri birbirlerinden ayrılmıştır.

Thread'lerin haberleşmesi için özel bir yönteme (boru falan, paylaşılan bellek alanı gibi) gerek yoktur. Zaten Data, BSS ve Heap alanları ortak alanlardır. Yani bu alanlar zaten adeta paylaşılan bellek alanı gibidirler. C'de ilkdeğer verilmiş global değişkenler, static yerel değişkenler ve string ifadeleri Data alanında, ilkdeğer verilmemiş global değişkenler ve static yerel değişkenler BSS alanında, parametre değişkenleri ve yerel değişkenler de stack alanında yaratılmaktadır.

**Thread'lerin Stack'leri**

Thread'lerin stack'leri birbirlerinden ayrılmıştır. Yerel değişkenler ve parametre değişkenleri stack'te yaratıldığı için her thread bunların sanki farklı bir kopyasını kullanıyor gibidir. Örneğin iki farklı thread aşağıdaki aynı foo fonksiyonunda ilerliyor olsun:

void foo(void)

{

int a = 10;

...

++a;

...

++a;

...

++a;

...

}

Burada a yerel bir değişkendir ve stack'te yaratılmaktadır. Fakat iki thread'in de kendine özgü farklı stack'leri vardır. Bu nedenle her iki thread de a'yı kendi stack'lerinde farklı kopya olarak yaratırlar. Yani her thread a'nın aslında kendine özgü bir kopyasını kullanmaktadır. Eğer buradaki a global ya da static yerel olsaydı her thread aynı nesneyi artırıyor olurdu.

**Thread'lerin Sonlanmasının Beklenmesi**

Bazen bir thread bitene kadar diğer bir thread'in bloke edilerek bekletilmesi gerekebilir. İşte Windows sistemlerinde WaitForSingleObject isimli API fonksiyonu, UNIX/Linux sistemlerinde pthread\_join isimli POSIX fonksiyonu kullanılmaktadır.

WaitForSingleObject aslında genel bir fonksiyondur ve zaten ilerinde açıklanacaktır. Fonksiyonun prototipi şöyledir:

DWORD WINAPI WaitForSingleObject(

\_\_in HANDLE *hHandle*,

\_\_in DWORD *dwMilliseconds*

);

Fonksiyonun birinci parametresi thread'in handle değerini alır. İkinci parametre milisaniye cinsinden zaman aşımını belirtmektedir. INFINITE özel değeri zaman aşımını ortadan kaldırır. Fonksiyonun geri dönüş değeri ileride daha detaylı olarak değerlendirilecektir. Ancak fonksiyon başarısızlık durumunda WAIT\_FAILED özel değerine geri döner. Örneğin:

#include<stdio.h>

#include<stdlib.h>

#include<Windows.h>

void ExitSys(LPCSTR lpszMsg, int status);

DWORD\_\_stdcall ThreadProc(void \*param);

int main(void)

{

DWORD dwThreadId;

HANDLE hThread;

if ((hThread = CreateThread(NULL, 0, ThreadProc, NULL, 0, &dwThreadId)) == NULL)

ExitSys("CreateThread", EXIT\_FAILURE);

WaitForSingleObject(hThread, INFINITE);

printf("Ok\n");

return 0;

}

DWORD\_\_stdcall ThreadProc(void \*param)

{

for (int i = 0; i < 10; ++i) {

printf("New Thread: %d\n", i);

Sleep(1000);

}

return 0;

}

void ExitSys(LPCSTRlpszMsg, intstatus)

{

DWORD dwLastError = GetLastError();

LPTSTR lpszErr;

if (FormatMessage(FORMAT\_MESSAGE\_ALLOCATE\_BUFFER | FORMAT\_MESSAGE\_FROM\_SYSTEM, NULL, dwLastError,

MAKELANGID(LANG\_NEUTRAL, SUBLANG\_DEFAULT), (LPTSTR)&lpszErr, 0, NULL)) {

fprintf(stderr, "%s: %s", lpszMsg, lpszErr);

LocalFree(lpszErr);

}

exit(status);

}

pthread\_join fonksiyonun prototipi de şöyledir:

int pthread\_join(pthread\_t thread, void \*\*retval);

Fonksiyonun birinci parametresi thread'in id değerini, ikinci parametresi thread'in exit kodunun yerleştirileceği void \* türünden nesnenin adresini alır. Bu parametre NULL geçilebilir. Fonksiyon başarı durumunda sıfır, başarıszlık durumunda hata kodunun kendisine geri döner. pthread\_join fonksiyonunda bir zaman aşımı parametresi yoktur.

#include<stdio.h>

#include<stdlib.h>

#include<string.h>

#include<unistd.h>

#include<pthread.h>

void \*thread\_proc(void \*param);

int main(void)

{

pthread\_t tid;

int result;

int i;

if ((result = pthread\_create(&tid, NULL, thread\_proc, NULL)) != 0) {

printf("pthread\_create: %s\n", strerror(result));

exit(EXIT\_FAILURE);

}

pthread\_join(tid, NULL);

printf("Ok\n");

return 0;

}

void \*thread\_proc(void \*param)

{

int i;

for (i = 0; i < 10; ++i) {

printf("New thread: %d\n", i);

sleep(1);

}

returnNULL;

}

**Thread'lerin Senkronizasyonu**

Thread senkronizasyonu thread'ler konusunun en çok zaman alan bölümünü oluşturmaktadır. Çünkü thread'ler ortak bir amacı gerçekleştirmek için çalışırken kimi zaman birbirlerini beklemesi gerekebilmektedir. Thread'ler nadir olarak bağımsız iş yaparlar. Genellikle işbirliği ile birlikte bir işi yapacak biçimde organize edilirler. Genel olarak thread'lerin birbirlerini beklemesi ile ilgili konuya "thread senkronizasyonu" denilmektedir.

**Kritk Kod Kavramı**

Başından sonuna kadar tek bir akış tarafından çalıştırılması gereken kod parçalarına kritik kod (critical section) denilmektedir. Pek çok durumda thread'ler ortak bir kaynak üzerinde birarada çalışma yapıyor olabilir. Bu ortak kaynak bir veri yapısı olabileceği gibi dış dünyadaki donanımsal bir aygıt da olabilir. İşte bir thread ortak bir kaynak üzerinde ilerlerken o sırada thread'ler arası geçiş olursa o kaynak kararsız bir durumda kalır. Diğer bir thread kaynağı kullanmak istediğinde sorun çıkar. Örneğin bir thread'in ortak kullanılan bir bağlı listeye ekleme yaptığını diğer thread'in de burada arama yaptığını düşünelim. Ekleme yapan thread tam eklemenin ortasında kesilirse ve diğer thread arama yapmaya çalışırsa program çökebilir. Bu tür ortak kaynak kullanan thread'lerin işin tamamı bitene kadar birbirlerini beklemesi gerekmektedir. Yani örneğimizde ekleme yapan thread arada kesilse bile diğer thread ekleme işlemi bitene kadar onun bu işi bitirmesini beklemelidir. Tabi beklemenin meşgul bir döngü yoluyla değil bloke yoluyla yani CPU zamanı harcamadan yapılması arzu edilir.

Örneği aşağıdaki kodda iki thread aynı global değişkeni birer milyon kere artırmıştır. Acaba global değişken iki milyon değerine ulaşır mı?

#include<stdio.h>

#include<stdlib.h>

#include<Windows.h>

void ExitSys(LPCSTR lpszMsg, int status);

DWORD\_\_stdcall ThreadProc1(void \*param);

DWORD\_\_stdcall ThreadProc2(void \*param);

int g\_i;

int main(void)

{

DWORD dwThreadId1, dwThreadId2;

HANDLE hThread1, hThread2;

if ((hThread1 = CreateThread(NULL, 0, ThreadProc1, NULL, 0, &dwThreadId1)) == NULL)

ExitSys("CreateThread", EXIT\_FAILURE);

if ((hThread2 = CreateThread(NULL, 0, ThreadProc2, NULL, 0, &dwThreadId2)) == NULL)

ExitSys("CreateThread", EXIT\_FAILURE);

WaitForSingleObject(hThread1, INFINITE);

WaitForSingleObject(hThread2, INFINITE);

printf("%d\n", g\_i);

return 0;

}

DWORD\_\_stdcall ThreadProc1(void \*param)

{

for (int i = 0; i < 1000000; ++i)

++g\_i;

return 0;

}

DWORD\_\_stdcall ThreadProc2(void \*param)

{

for (int i = 0; i < 1000000; ++i)

++g\_i;

return 0;

}

void ExitSys(LPCSTRlpszMsg, intstatus)

{

DWORD dwLastError = GetLastError();

LPTSTR lpszErr;

if (FormatMessage(FORMAT\_MESSAGE\_ALLOCATE\_BUFFER | FORMAT\_MESSAGE\_FROM\_SYSTEM, NULL, dwLastError,

MAKELANGID(LANG\_NEUTRAL, SUBLANG\_DEFAULT), (LPTSTR)&lpszErr, 0, NULL)) {

fprintf(stderr, "%s: %s", lpszMsg, lpszErr);

LocalFree(lpszErr);

}

exit(status);

}

Burada bizim işlemi tek bir ++ operatörüyle yapmış olmamız bunun atomik olacağı anlamına gelmez. NMakina komutları atomiktir. Yani bir makina komutu çalıştırılırken zaten thread arası geçiş oluşamaz. Thread'ler arası geçiş iki makina komutu arasında olabilir. İşte derleyiciler ++g\_i ifadesini tek bir makina komutuyla yapmak zorunda değildirler. Örneğin bu işlemi aşağıdaki gibi üç makina komutuyla yapabilirler:

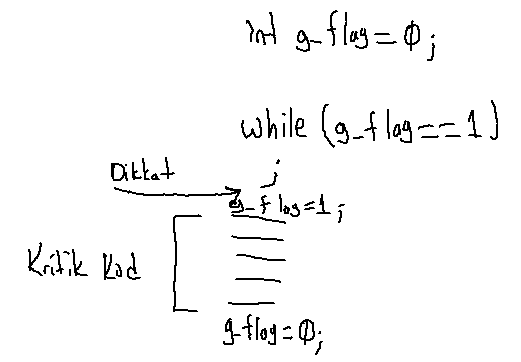
MOV reg, g\_i

INC reg

MOV g\_i, reg

Gerçekten de yukarıdaki program çalışırsa muhtemelen iki milyon değeri görülmeyecektir. Bizim bu artırımı kritk kod altında yapmamız gerekir.

Krtitik kodlar manuel olarak oluşturulamazlar. Örneğin aşağıdaki gibi bir kodla krtitik kod oluşturamayız:



Bu kodun iki dezavantajı vardır: Birincisi bekleyen thread meşgul bir döngüde (busy loop) bekleme yapar. İkincisi burada ok ile gösterilen noktada thread'ler arası geçiş oluşursa birden fazla thread kritik koda girebilir.

Pekiyi bu işlem güvenli bir biçimde nasıl yapılabilir? Aslında yine bir bayrak tutulabilir. Ancak kontrol ve set işlemi yapılırken threadler arası geçiş kapatılabilir. Böylece bayrak güvenli olarak set edilebilir. Fakat thread'ler arası geçişin kapatılabilmesi için gereken makina komutlarını ancak kernel mod programlar kullanabilmektedir. Örneğin işletim sisteminin kernel modda çalışan sistem fonksiyonları bunu yapabilir. Eğer bayrak set edilmişse thread'i uykuya geçirebilir. Bu yöntem kullanılarak senkronizasyon yapan kodlara"kernel mod senkronizasyon nesneleri" denilmektedir. Son 20 yıldır makine dillerine karşılaştırma ve jump işlemini birlikte yapan özel komutlar eklenmiştir. Bu komutlar bayrakları güvenli set etmeyi sağlamaktadır. Böylece hiç olmazsa kontrol yapılırken kernel moda geçiş engellenebilmektedir. Fakat ne olursa olsun bu işi yapan fonksiyonların özel bir biçimde yazılması gerekir. İşte işletim sistemleri bunları yapan hazır fonksiyonlar bulundurmaktadır.

**Windows'ta Kritik Kodların Oluşturulması**

Windows'ta kritik kod oluşturmak için CriticalSection nesnesi ya da mutex nesnelerinden faydalanılmaktadır. CriticalSection nesnesi işlemlerini user modda yapmaktadır. CriticalSection nesnesinin kullanımı şöyledir:

1) CRITICAL\_SECTION isimli yapı türünden global bir nesne yaratılır.

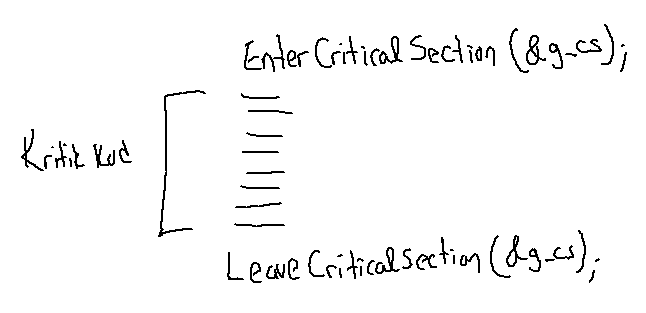
2) Bu nesne InitializeCriticalSection isimli API fonksiyonuyla ilkdeğerlenir.

void WINAPI InitializeCriticalSection(

\_\_out LPCRITICAL\_SECTION *lpCriticalSection*

);

3) Kritik kod aşağıdaki gibi oluşturulur:



Bir thread EnterCriticalSection fonksiyonundan girdiğinde artık başka bir thread oardan girmeye çalışırsa bloke olur ve bekler. Taki diğeri LeaveCriticalSection ile kritik kod kilidini bırakana kadar. Böylece EnterCriticalSection ve LeaveCriticalSection arasındaki kodu tek bir thread akışı başından sonuna kadar çalıştırmış olur. Fonksiyonların prototipleri şöyledir:

void WINAPI EnterCriticalSection(

\_\_in\_out LPCRITICAL\_SECTION *lpCriticalSection*

);

void WINAPI LeaveCriticalSection(

\_\_in\_out LPCRITICAL\_SECTION *lpCriticalSection*

);

Birden fazla thread EnterCriticalSection noktasında blokede beklerse kilit açıldığında hangisi kritik koda girecektir? Şüphesiz adil bir sistemin uygulanması istenir. Ancak Windows bunun bir garantisini bize vermemektedir. Tabi elinden geldiğince adil bir davranış sergileyeceğini söylemektedir.

4) İşimiz bitince DeleteCriticalSection fonksiyonuyla yapılan işlemler geri alınmalıdır:

void WINAPI DeleteCriticalSection(

\_\_in\_out LPCRITICAL\_SECTION *lpCriticalSection*

);

Örneğin:

#include<stdio.h>

#include<stdlib.h>

#include<Windows.h>

void ExitSys(LPCSTR lpszMsg, int status);

DWORD\_\_stdcall ThreadProc1(void \*param);

DWORD\_\_stdcall ThreadProc2(void \*param);

CRITICAL\_SECTION g\_cs;

int g\_i;

int main(void)

{

DWORD dwThreadId1, dwThreadId2;

HANDLE hThread1, hThread2;

InitializeCriticalSection(&g\_cs);

if ((hThread1 = CreateThread(NULL, 0, ThreadProc1, NULL, 0, &dwThreadId1)) == NULL)

ExitSys("CreateThread", EXIT\_FAILURE);

if ((hThread2 = CreateThread(NULL, 0, ThreadProc2, NULL, 0, &dwThreadId2)) == NULL)

ExitSys("CreateThread", EXIT\_FAILURE);

WaitForSingleObject(hThread1, INFINITE);

WaitForSingleObject(hThread2, INFINITE);

printf("%d\n", g\_i);

DeleteCriticalSection(&g\_cs);

return 0;

}

DWORD\_\_stdcall ThreadProc1(void \*param)

{

for (int i = 0; i < 1000000; ++i) {

EnterCriticalSection(&g\_cs);

++g\_i;

LeaveCriticalSection(&g\_cs);

}

return 0;

}

DWORD\_\_stdcall ThreadProc2(void \*param)

{

for (int i = 0; i < 1000000; ++i) {

EnterCriticalSection(&g\_cs);

++g\_i;

LeaveCriticalSection(&g\_cs);

}

return 0;

}

void ExitSys(LPCSTRlpszMsg, intstatus)

{

DWORD dwLastError = GetLastError();

LPTSTR lpszErr;

if (FormatMessage(FORMAT\_MESSAGE\_ALLOCATE\_BUFFER | FORMAT\_MESSAGE\_FROM\_SYSTEM, NULL, dwLastError,

MAKELANGID(LANG\_NEUTRAL, SUBLANG\_DEFAULT), (LPTSTR)&lpszErr, 0, NULL)) {

fprintf(stderr, "%s: %s", lpszMsg, lpszErr);

LocalFree(lpszErr);

}

exit(status);

}

EnterCrticalSection fonksiyonuna verilen parametre kilidi temsil eder. Örneğin programın iki farklı yerinde biz EnterCrticalSection fonksiyonuna aynı argümanı geçersek bunlar aynı kilide ilişkin olurlar. Başka bir örnek şöyle olabilir:

#include<stdio.h>

#include<stdlib.h>

#include<Windows.h>

void ExitSys(LPCSTR lpszMsg, int status);

DWORD\_\_stdcall ThreadProc1(void \*param);

DWORD\_\_stdcall ThreadProc2(void \*param);

CRITICAL\_SECTION g\_cs;

int main(void)

{

DWORD dwThreadId1, dwThreadId2;

HANDLE hThread1, hThread2;

InitializeCriticalSection(&g\_cs);

if ((hThread1 = CreateThread(NULL, 0, ThreadProc1, NULL, 0, &dwThreadId1)) == NULL)

ExitSys("CreateThread", EXIT\_FAILURE);

if ((hThread2 = CreateThread(NULL, 0, ThreadProc2, NULL, 0, &dwThreadId2)) == NULL)

ExitSys("CreateThread", EXIT\_FAILURE);

WaitForSingleObject(hThread1, INFINITE);

WaitForSingleObject(hThread2, INFINITE);

DeleteCriticalSection(&g\_cs);

return 0;

}

void CommonProc(constchar \*str)

{

EnterCriticalSection(&g\_cs);

printf("%s: 1. Step\n", str);

Sleep(rand() % 100);

printf("%s: 2. Step\n", str);

Sleep(rand() % 100);

printf("%s: 3. Step\n", str);

Sleep(rand() % 100);

printf("%s: 4. Step\n", str);

Sleep(rand() % 100);

printf("%s: 5. Step\n", str);

Sleep(rand() % 100);

printf("----------------------\n");

LeaveCriticalSection(&g\_cs);

}

DWORD\_\_stdcall ThreadProc1(void \*param)

{

int i;

for (i = 0; i < 10; ++i)

CommonProc("ThreadProc1");

return 0;

}

DWORD\_\_stdcall ThreadProc2(void \*param)

{

int i;

for (i = 0; i < 10; ++i)

CommonProc("ThreadProc2");

return 0;

}

void ExitSys(LPCSTRlpszMsg, intstatus)

{

DWORD dwLastError = GetLastError();

LPTSTR lpszErr;

if (FormatMessage(FORMAT\_MESSAGE\_ALLOCATE\_BUFFER | FORMAT\_MESSAGE\_FROM\_SYSTEM, NULL, dwLastError,

MAKELANGID(LANG\_NEUTRAL, SUBLANG\_DEFAULT), (LPTSTR)&lpszErr, 0, NULL)) {

fprintf(stderr, "%s: %s", lpszMsg, lpszErr);

LocalFree(lpszErr);

}

exit(status);

}

Örneğin C++'ta iki thread dinamik büyüyen bir diziye (vector) ekleme yapacak olsun. Ekleme işleminin yarıda kesilmemesi gerekir. Bunun için bu işlem kritik kod kontrolüyle yapılmalıdır. Örneğin eğer aşağıdaki koddan senkronizasyon kısmı kaldırılırsa program çökebilir:

#include<stdio.h>

#include<stdlib.h>

#include<Windows.h>

#include<vector>

usingnamespace std;

void ExitSys(LPCSTR lpszMsg, int status);

DWORD\_\_stdcall ThreadProc1(void \*param);

DWORD\_\_stdcall ThreadProc2(void \*param);

CRITICAL\_SECTION g\_cs;

vector<int> g\_v;

int main(void)

{

DWORD dwThreadId1, dwThreadId2;

HANDLE hThread1, hThread2;

InitializeCriticalSection(&g\_cs);

if ((hThread1 = CreateThread(NULL, 0, ThreadProc1, NULL, 0, &dwThreadId1)) == NULL)

ExitSys("CreateThread", EXIT\_FAILURE);

if ((hThread2 = CreateThread(NULL, 0, ThreadProc2, NULL, 0, &dwThreadId2)) == NULL)

ExitSys("CreateThread", EXIT\_FAILURE);

WaitForSingleObject(hThread1, INFINITE);

WaitForSingleObject(hThread2, INFINITE);

DeleteCriticalSection(&g\_cs);

return 0;

}

DWORD\_\_stdcall ThreadProc1(void \*param)

{

for (int i = 0; i < 1000000; ++i) {

EnterCriticalSection(&g\_cs);

g\_v.push\_back(i);

LeaveCriticalSection(&g\_cs);

}

return 0;

}

DWORD\_\_stdcall ThreadProc2(void \*param)

{

for (int i = 0; i < 1000000; ++i) {

EnterCriticalSection(&g\_cs);

g\_v.push\_back(i);

LeaveCriticalSection(&g\_cs);

}

return 0;

}

void ExitSys(LPCSTRlpszMsg, intstatus)

{

DWORD dwLastError = GetLastError();

LPTSTR lpszErr;

if (FormatMessage(FORMAT\_MESSAGE\_ALLOCATE\_BUFFER | FORMAT\_MESSAGE\_FROM\_SYSTEM, NULL, dwLastError,

MAKELANGID(LANG\_NEUTRAL, SUBLANG\_DEFAULT), (LPTSTR)&lpszErr, 0, NULL)) {

fprintf(stderr, "%s: %s", lpszMsg, lpszErr);

LocalFree(lpszErr);

}

exit(status);

}

**Windows'ta WaitForSingleObject ve WaitForMultipleObjects API Fonksiyonları**

Windows'ta WaitForSingleObject ve WaitForMultipleObjects isimli API fonksiyonları senkronizasyon için kullanılan genel fonksiyonlardır. Bir kernel nesnesinin açık (signaled) ve kapalı (nonsignaled) biçiminde iki durumu vardır. WaitForSingleObject eğer nesne kapalıysa bekler. Ta ki açılana kadar. Her nesnenin hangi durumda kapalı hangi durumda açık olduğu belirlenmiştir. Örneğin aslında thread'lerin kendisi de bir senkronizasyon nesnesi olarak kullanılabilir. Thread çalışıyorsa nesne kapalı, thread sonlanmışsa nesne açık durumdadır. WaitForSingleObject fonksiyonunun prototipi şöyledir:

DWORD WINAPI WaitForSingleObject(

\_\_in HANDLE *hHandle*,

\_\_in DWORD *dwMilliseconds*

);

Fonksiyonun birinci parametresi senkronizasyon nesnesinin (duruma göre thread, mutex, semaphore vs.) handle değerini alır. İkinci parametre zaman aşımını belirtir. Eğer nesne açılmazsa en kötü olasılıkla bu kadar milisaniye sonra bekleme sonlandırılır. Ancak INFINITE özel değeri zaman aşımını kaldırmaktadır. Fonksiyon zaman aşamından dolayı sonlanmışsa WAIT\_TIMEOUT özel değerine geri döner. Mutex'lerde mutex'e sahip thread sonlanmışsa fonksiyon WAIT\_ABONDONED özel değerine geri dönmektedir. Nihayet eğer nesne açık duruma geldiğinde dolayı fonksiyon geri dönmüşse WAIT\_OBJECT\_0 değeri elde edilir. Fonksiyon başarısız olabilir. Başarısızlık durumunda WAIT\_FAILED değeri elde edilir.

WaitForMultipleObjects birden fazla nesneyi beklemek kullanılır. Bunların hepsi açık duruma geçene kadar ya da en az biri açık duruma geçene kadar bekleme yapılabilir. Fonksiyonun prototipi şöyledir:

DWORD WINAPI WaitForMultipleObjects(

\_\_in DWORD *nCount*,

\_\_in const HANDLE\* *lpHandles*,

\_\_in BOOL *bWaitAll*,

\_\_in DWORD *dwMilliseconds*

);

Fonksiyonun birinci parametresi dizideki nesnelerin sayısını belirtir. İkinci parametre HANDLE değerlerinin bulunduğu dizinin adresidir. Üçüncü parametre eğer TRUE ise tüm nesneler açık duruma geçene kadar bekleme yapılır, FALSE ise yalnızca bunlardan herhangi biri açık duruma geçene kadar bekleme yapılır. Son parametre zaman aşımını belirtmektedir. Fonksiyon eğer zaman aşımından dolayı sonlanmışsa yine WAIT\_FAILED özel değerine geri döner. Eğer üçüncü parametre FALSE ise geri dönüş değeri WAIT\_OBJECT\_0 + n biçimindedir. Bu n değeri açık duruma geçmiş nesnenin dizideki indeksini belirtir. Benzer biçimde eğer fonksiyon mutex'e sahip bir thread'in sonlanmasıyla sonlanmışsa bu durumda geri dönüş değeri WAIT\_ABONDENT\_0 + n biçimindedir. Buradaki n yine dizide indeks belirtir. Fonksiyon başarısz olursa yine yine WAIT\_FAILED değerine geri döner.

**Windows Sistemlerinde Kritik Kodların Mutex Nesneleriyle Oluşturulması**

Windows'ta mutex nesneleri kernel moda geçiş yaparak çalışırlar. Dolayısıyla eğer proseslerarası bir kullanım söz konusu değilse programcı kritik kodları Windows'ta mümkün olduğunda tukarıda açıklanan CriticalSection nesnesiyle oluşturmalıdır.

Mutex ismi İngilizce "Mutual Exclusion" sözcüklerinden türetilmiştir. Mutex pek çok işletim sisteminde kullanılan temel bir senkronizasyon nesnesidir. Mutex nesnesinin sahipliği (ownership) denilen bir kavram vardır. Bir mutex nesnesinin sahipliği bir thread tarafından alınır. Sahiplik ancak onu alan thread tarafından bırakılabilir. Mutex nesnelerinin eğer sahipliği bir thread tarafından alınmışsa nesne kapalı durumdadır. Eğer bunların sahipliği bir thread tarafından alınmamışsa nesne açık durumdadır.

Mutex nesneleri şöyle kullanılmaktadır:

1) Mutex nesnesi CreateMutex API fonksiyonuyla yaratılır. Proses içi kullanımda mutex'in handle değeri global bir değişkene atanmalıdır:

HANDLE WINAPI CreateMutex(

\_\_in LPSECURITY\_ATTRIBUTES *lpMutexAttributes*,

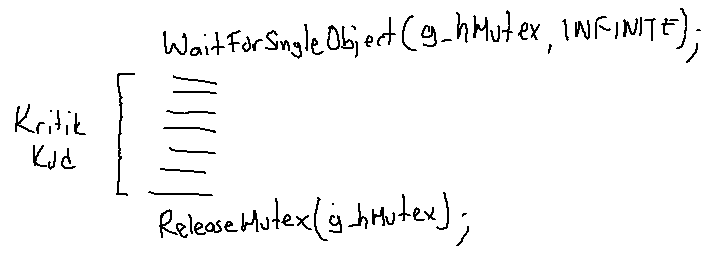
\_\_in BOOL *bInitialOwner*,

\_\_in LPCTSTR *lpName*

);

Fonksiyonun birinci parametresi kernel nesnesinin güvenlik bilgilerini belirtir. Bu parametre NULL geçilebilir. İkinci parametre mutex'in başlangıçtaki sahipliğini belirlemekte kullanılır. eğer bu parametre TRUE geçilirse CreateMutex'i çağıran thread nesnenin sahipliğini işin başında almış olur. FALSE geçilirse nesne sahipliği alınmadan yaratılır. Son parametre proseslerarası kullanımdaki ismi belirtir. Aynı proses içindeki kullanımda buj parametre NULL geçilebilir. Fonksiyon başarı durumunda mutex'in handle değerine, başarısızlık durumunda NULL değerine geri döner.

2) Kritik kod şöyle oluşturulur:



WaitForSingleObject mutex nesnesinin sahipliği bir thread tarafından alınmışsa blokede bekler. Nesnenin sahipliği o thread tarafından bırakıldığında WaitForSingleObject nesne açık duruma geçtiği için beklemeyi sonlandırır ve aynı zamanda nesnin sahipliğini de alır. ReleaseMutex ise nesnenin sahipliğini bırakmaktadır:

BOOL WINAPI ReleaseMutex(

\_\_in HANDLE *hMutex*

);

3) İşlem bittiğinde mutex nesnesi CloseHandle fonksiyonuyla yok edilir.

Pekiyi mutex'in sahipliğini almış bir therad sahipliğii bırakamadan sonlanırsa ne olur? Çünkü onu bekleyen başka thread'ler olabilir. Onların durumu ne olacaktır? İşte bu durumda mutex'lere terkedilmiş mutex'ler (abondened mutex) denilmektedir. Eğer mutex'in sahipliğini almış olan bir thread onu bırakmadan sonlanırsa onu WaitForXXX fonksiyonlarıyla bekleyen thrttead'ler başarıszlıkla geri dönerler. Böylece onu sonsuza kadar bekleme durumu (dedlock) oluşmaz.

Pekiyi bir thread'in bizzat kendisi mutex'in sahipliğini yeniden alabilir mi? Örneğin Foo içerisinde mutex'le oluştuırulmuş bir kritik kod olsun bu kritik kod içerisinde Bar'ı çağıralım. Bar başka yerden de çağrılabileceği için onun içinde de aynı mutex'e ilişkin kritik kod olsun. Bu durumda thread kendi kendisini kilitler mi? Yanıt Windows sistemlerinde hayır. Ancak Windows istemlerinde aynı thread ne kadar WaitForSingleObject'ten geçmişse o kadar ReleaseMutex uygulamalıdır.

Örneğin:

#include<stdio.h>

#include<stdlib.h>

#include<Windows.h>

void ExitSys(LPCSTR lpszMsg, int status);

DWORD\_\_stdcall ThreadProc1(void \*param);

DWORD\_\_stdcall ThreadProc2(void \*param);

HANDLE g\_hMutex;

int main(void)

{

DWORD dwThreadId1, dwThreadId2;

HANDLE hThread1, hThread2;

if ((g\_hMutex = CreateMutex(NULL, FALSE, NULL)) == NULL)

ExitSys("CreateMutex", EXIT\_FAILURE);

if ((hThread1 = CreateThread(NULL, 0, ThreadProc1, NULL, 0, &dwThreadId1)) == NULL)

ExitSys("CreateThread", EXIT\_FAILURE);

if ((hThread2 = CreateThread(NULL, 0, ThreadProc2, NULL, 0, &dwThreadId2)) == NULL)

ExitSys("CreateThread", EXIT\_FAILURE);

WaitForSingleObject(hThread1, INFINITE);

WaitForSingleObject(hThread2, INFINITE);

CloseHandle(g\_hMutex);

return 0;

}

void CommonProc(constchar \*str)

{

WaitForSingleObject(g\_hMutex, INFINITE);

printf("%s: 1. Step\n", str);

Sleep(rand() % 100);

printf("%s: 2. Step\n", str);

Sleep(rand() % 100);

printf("%s: 3. Step\n", str);

Sleep(rand() % 100);

printf("%s: 4. Step\n", str);

Sleep(rand() % 100);

printf("%s: 5. Step\n", str);

Sleep(rand() % 100);

printf("----------------------\n");

ReleaseMutex(g\_hMutex);

}

DWORD\_\_stdcall ThreadProc1(void \*param)

{

int i;

for (i = 0; i < 10; ++i)

CommonProc("ThreadProc1");

return 0;

}

DWORD\_\_stdcall ThreadProc2(void \*param)

{

int i;

for (i = 0; i < 10; ++i)

CommonProc("ThreadProc2");

return 0;

}

void ExitSys(LPCSTRlpszMsg, intstatus)

{

DWORD dwLastError = GetLastError();

LPTSTR lpszErr;

if (FormatMessage(FORMAT\_MESSAGE\_ALLOCATE\_BUFFER | FORMAT\_MESSAGE\_FROM\_SYSTEM, NULL, dwLastError,

MAKELANGID(LANG\_NEUTRAL, SUBLANG\_DEFAULT), (LPTSTR)&lpszErr, 0, NULL)) {

fprintf(stderr, "%s: %s", lpszMsg, lpszErr);

LocalFree(lpszErr);

}

exit(status);

}

**UNIX/Linux Sistemlerinde Kritik Kodların Oluşturulması**

UNIX/Linux sistemlerinde Windows sistemlerindeki gibi bir CriticalSection nesnesi yoktur. Bu sistemlerde kritik kodlar mutex nesneleriyle oluşturulmaktadır. Fakat UNIX/Linux sistemlerineki mutex nesneleri proseslerarası kullanılmayacaksa zaten kernel moda geçmez.

UNIX/Linux sistemlerinde mutex nesneleri şöyle kullanılır:

1) Mutex'i temsil eden pthread\_mutex\_t türünden global bir nesne tanımlanır. Örneğin:

pthread\_mutex\_t g\_mutex;

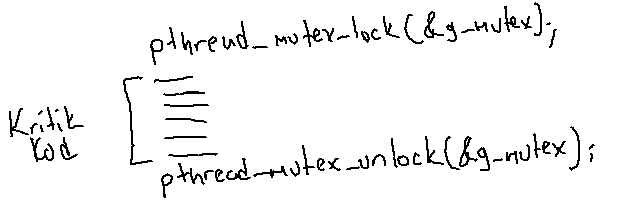
2) Bu mutex nesnesi pthread\_mutex\_init fonksiyonula ilkdeğerlenir.

#include <[pthread.h](http://pubs.opengroup.org/onlinepubs/007908799/xsh/pthread.h.html)>

int pthread\_mutex\_init(pthread\_mutex\_t \**mutex*, const pthread\_mutexattr\_t \**attr*);

Fonksiyonun birinci parametresi global nesnenin adresini, ikinci parametresi mutex özelliklerini belirten yapı nesnesinin adresini alır. İkinci parametre NULL geçilebilir. Bu durumda mutex default özelliklerle yaratılır. Fonksiyon başarı durumunda sıfır, başarıslık durumunda hata kodunun kendisine geri döner.

3) Kritik kod aşağıdaki gibi oluşturulur:



pthread\_mutex\_lock eğer mutex nesnesinin sahipliği başka bir thread tarafından alınmışsa sahiplik bırakılana kadar blokede bekletir. eğer mutex'in sahipliği başka bir thread tarafından alınmamışsa mutex'in sahipliği alınarak kritik koda girilir.

#include <[pthread.h](http://pubs.opengroup.org/onlinepubs/007908799/xsh/pthread.h.html)>

int pthread\_mutex\_lock(pthread\_mutex\_t \**mutex***);**

int pthread\_mutex\_unlock(pthread\_mutex\_t \**mutex***);**

Fonksiyonlar mutex nesnesinin adresini alır, başarı drumunda sıfır, başarısızlık durumunda hata kodunun kendisine geri dönerler.

Örneğin:

#include<stdio.h>

#include<stdlib.h>

#include<string.h>

#include<unistd.h>

#include<pthread.h>

void \*thread\_proc1(void \*param);

void \*thread\_proc2(void \*param);

void common\_proc(constchar \*str);

pthread\_mutex\_t g\_mutex;

int main(void)

{

pthread\_t tid1, tid2;

int result;

if ((result = pthread\_mutex\_init(&g\_mutex, NULL)) != 0) {

printf("pthread\_mutex\_init: %s\n", strerror(result));

exit(EXIT\_FAILURE);

}

if ((result = pthread\_create(&tid1, NULL, thread\_proc1, NULL)) != 0) {

printf("pthread\_create: %s\n", strerror(result));

exit(EXIT\_FAILURE);

}

if ((result = pthread\_create(&tid2, NULL, thread\_proc2, NULL)) != 0) {

printf("pthread\_create: %s\n", strerror(result));

exit(EXIT\_FAILURE);

}

pthread\_join(tid1, NULL);

pthread\_join(tid2, NULL);

return 0;

}

void common\_proc(constchar \*str)

{

pthread\_mutex\_lock(&g\_mutex);

printf("%s: 1. Step\n", str);

usleep(rand() % 300000);

printf("%s: 2. Step\n", str);

usleep(rand() % 300000);

printf("%s: 3. Step\n", str);

usleep(rand() % 300000);

printf("%s: 4. Step\n", str);

usleep(rand() % 300000);

printf("%s: 5. Step\n", str);

usleep(rand() % 300000);

printf("----------------------\n");

pthread\_mutex\_unlock(&g\_mutex);

}

void \*thread\_proc1(void \*param)

{

int i;

for (i = 0; i < 10; ++i)

common\_proc("ThreadProc1");

returnNULL;

}

void \*thread\_proc2(void \*param)

{

int i;

for (i = 0; i < 10; ++i)

common\_proc("ThreadProc2");

returnNULL;

}

**Semaphore Nesneleri**

Trenlerdeki dur geç lambalarına semaphore denilmektedir. Semapore'lar sayaçlı senkronizasyon nesneleridir. Bir kritik koda en fazla n tane akışın girebilmesi için kullanılırlar. Semaforun bir sayacı vardır. Bu sayaç sıfırdan büyükse nesne açık durumdadır. Yani semafor kilitli değildir. Her geçiş yapıldığında sayaç bir eksiltilir. Sayaç sıfıra geldiğinde semafor nesnesi kapalı duruma geçer. Yani kilitlenir. Artık geçiş yapılmak istendiğinde ilgili thread bloke edilir. Kritik koddan çıkıldığında sayaç 1 artırılır. Böylece kritik koda en fazla n tane thread'in girmesi sağlanır.

Pekiyi kritik koda neden n tane akış girsin ki? İşte eğer n tane kaynak varsa ama bunu elde etmek için rekabet eden çok thread bulunuyorsa bazı thread'lerin beklemesi gerekir. Böylece kritik koda giren her thread'e bir kaynak tahsis edilir. semaphore sayacı sıfıra geldiğinde artık tahsis edilecek kaynak kalmamıştır. Bir thread işini bitirdiğinde kaynağı bırakır. Kritik koddan çıkar. Sayaç böylece bir artırılır. Yeni bir thread artık kritik koda girebilir.

Semafor sayacı başlangıçta 1 ise böyle semaphore'lara ikili (binary) semaphore denilmektedir. İkili semaphore'lar mutex nesnelerine çok benzer. Ancak yine de mutex'le ikili semaphore arasında önemli bir fark da vardır. Bir mutex nesnesinin sahipliği ancak onu almış thread tarafından bırakılabilir. Ancak bir semaphore'un sayacı başka thread'ler atarfından artırılabilir.

Semaphore nesneleri özellikle üretici-tüketici (producer-consumer) probleminin çözümü için kullanılmaktadır.

**Üretici-Tüketici Problemi (Producer-Consumer Problem)**

Üretici-Tüketici problemi uygulamalarda en fazla karşılaşılan senkronizasyon problemlerinden bir tanesidir. Bu problemde işlemleri hızlandırmak için programcı iki thread kullanmaktadır. Birinci thread bir bilgiyi elde eder fakat işlemez. Bunu işlemesi için diğer thread'e verir. Bu verme işlemini o bilgiyi ortak paylaşılan alana yazarak yapar. Diğer thread'de bilgiyi oradan alarak işler. İki thread de bunları bir döngü içerisinde çok defalar yapmaktadır. Böyle bir sistem hızlandırma sağlar. Çünkü eğer tek bir thread hem bilgiyi elde edip işlerse toplamda birim zamanda daha az bilgi işlenmiş olur. Halbuki biri bilgiyi elde ederken diğeri işlerse aynı anda çalışma fırsatından dolayı birim zamanda işlenen bilgi miktarı artar. Bu problemde bilgiyi elde eden thread'e üretici thread (producer thread), onu işleyen thread'e de tüketici thread (consumer thread) denilmektedir.

Üretici-Tüketici probleminde sorun şudur: Üretici thread henüz tüketici eski bilgiyi almadan yeni bilgiyi paylaşılan alana yerleştirmemelidir. Yerleştirirse eski bilgi ezilir. Benzer biçimde tüketici thread de henüz üretici thread yeni bilgiyi yerleştirmeden eski bilgiyi paylaşılan alandan ikinci kez almamalıdır.

İşte üretici-tüketici problemleri tipik olarak semaphore nesneleri kullanılarak gerçekleştirilir.

**Windows Sistemlerinde Semaphore Nesnelerinin Kullanılması**

Windows sistemlerinde semaphore nesneleri şöyle kullanılır:

1) CreateSemaphore fonksiyonuyla semaphore yaratılır ve handle değeri global bir değişkenin içerisine yerleştirilir:

HANDLE WINAPI CreateSemaphore(

\_\_in LPSECURITY\_ATTRIBUTES *lpSemaphoreAttributes*,

\_\_in LONG *lInitialCount*,

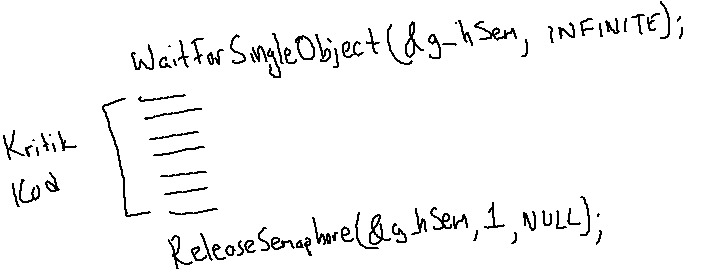
\_\_in LONG *lMaximumCount*,

\_\_in LPCTSTR *lpName*

);

Fonksiyonun birinci parametresi semaphore nesnesinin güvenlik bilgilerini belirtir. Bu parametre NULL geçilebilir. İkinci parametre başlangıçtaki semaphore sayacının değeridir. Üçüncü parametre semaphore sayacının çıkacağı maksimum değeri belirtir. Son parametre proseslerarası kullanım için gereken ismi belirtir. Bu parametre proses içi kullanımda NULL geçilebilir. Fonksiyon başarı durumunda handle değerine, başarıszlık durumunda NULL adrese geri döner.

2) Kirtik kod şöyle oluşturulur:



ReleaseSemaphore fonksiyonu semaphore sayacını artırmaktadır. Bir semaphore nesnesinden sayaç sıfırdan büyükse nesne açık durumda, sıfır ise nesne kapalı durumdadır. WaitForSingleObject fonksiyonundan her geçişte semaphore sayacı otomatik olarak 1 eksiltilir. Sayaç sıfırsa blokede beklenir. ReleaseSemaphore fonksiyonun prototipi şöyledir:

BOOL WINAPI ReleaseSemaphore(

\_\_in HANDLE *hSemaphore*,

\_\_in LONG *lReleaseCount*,

\_\_out LPLONG *lpPreviousCount*

);

Fonksiyonun birinci parametresi semaphore'un handle değerini, ikinci parametresi sayacın artırılacak değerini belirtir. Üçüncü parametre sayacın önceki değerinin yerleştirileceği nesnenin adresini alır. Bu parametre NULL geçilebilir.

3) İşlem bitince semaphore nesnesi CloseHandle fonksiyonuyla yok edilir.

**UNIX/Linux Sistemlerinde Semaphore Nesnelerinin Kullanımı**

UNIX/Linux sistemlerinde semaphore işlemleri iki grup POSIX fonksiyıonuyla yapılabilmektedir. Bunlardan biri eskiden beri kullanılan klasik System 5 semaphore nesneleridir. Bunların parametrik yapıları karşıktır ve kullanımları oldukça zordur. 90'lı yıllarda thread'lerle birlikte yeni modern semaphore fonksiyonları da oluşturulmuştur. Bunlara POSIX semaphore'ları denilmektedir. Halbuki her iki fonksiyon grubu da POSIX standartlarında yer almaktadır. Biz burada modern POSIX semaphore nesnelerinin kullanımını göreceğiz1:

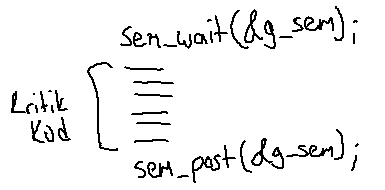
1) Önce sem\_t türünden global bir nesne tanımlanır. Sonra sem\_init fonksiyonuyla bu semaphore nesnesi initialize edilir.

#include <semaphore.h>

int sem\_init(sem\_t \*sem, int pshared, unsigned int value);

Fonksiyonun birinci parametresi semaphore nesnesinin adresini alır. İkinci parametre proseslerarası paylaşım ile ilgilidir. Nesne prosesler arasında paylaşılacaksa bu parametre sıfır dışı bir değer olarak, paylaşılmayacaksa sıfır olarak geçilir. Üçüncü parametre başlangıçtahi semaphore sayacını belirtir.

2) Kritik kod şöyle oluşturulur:



sem\_wait fonksiyonu semaphore sayacı sıfırdan büyükse geçiş yapar. Ancak geçiş yaparken semaphore sayacını 1 eksiltir. sem\_post ise semaphore sayacını 1 artırmaktadır.

#include <semaphore.h>

int sem\_wait(sem\_t \*sem);

int sem\_post(sem\_t \*sem);

sem\_wait ve sem\_post fonksiyonları başarı durumunda 0, başarısızlık durumunda -1 değerine geri dönmektedir. Bunlar ernno değişkenini set ederler.

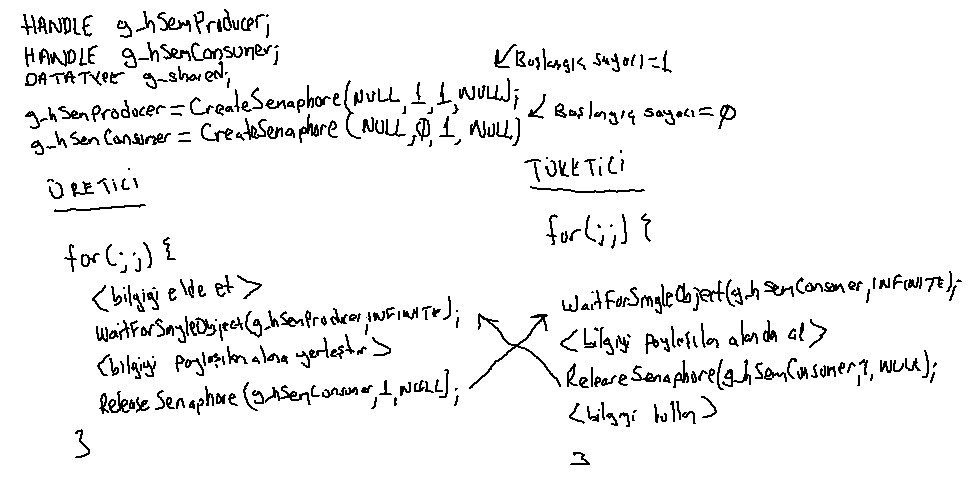
3) İşlemler bitince semaphore nesnesi sem\_destroy fonksiyonuyla boşaltılabilir.

#include <semaphore.h>

int sem\_destroy(sem\_t \*sem);

**Üretici Tüketici Probleminin Semaphore Nesneleriyle Çözümü**

Semaphore'lar tipik olarak üretici-tüketici problemlerini çözmek için kullanılabilmektedir. Bunun için Mutex nesnesi kullanılmaz. Çünkü mutex'in sahipliğini ancak onu alan thread bırakabilmektedir.



Burada başlangıçta üretici semaphore'unun sayacı 1'dir. Bu nedenle işin başında üretici semaphore kilidinden geçer ve sayaç sıfıra düşer. Bu sırada tüketici beklemektedir. Üretici bilgiyi baplaşılan alana yerleştirdikten sonra tüketicinin semaphore sayacını bir artırır. Böylece tüketici blokeden kurtulur ve bilgiyi alır. Bu sorada üretici yine başa dönül beklemektedir. Çünkü artık onun semaphore sayacı sıfıra düşmüştür. Tüketici bilgiyi alınca bu sefer üreticinin semaphore sayacını 1 artırır.

Üretici Tüketici problemi Windows'ta aşağıdaki gibi kodlanabilir:

/\*ProducerConsumer.c \*/

#include<stdio.h>

#include<stdlib.h>

#include<time.h>

#include<Windows.h>

void ExitSys(LPCSTR lpszMsg, int status);

DWORD\_\_stdcall ThreadProcProducer(void \*param);

DWORD\_\_stdcall ThreadProcConsumer(void \*param);

HANDLE g\_hSemProducer;

HANDLE g\_hSemConsumer;

int g\_share;

int main(void)

{

DWORD dwThreadIdProducer, dwThreadIdConsumer;

HANDLE hThreadProducer, hThreadConsumer;

srand(time(NULL));

if ((g\_hSemProducer = CreateSemaphore(NULL, 1, 1, NULL)) == NULL)

ExitSys("CreateSemaphore", EXIT\_FAILURE);

if ((g\_hSemConsumer = CreateSemaphore(NULL, 0, 1, NULL)) == NULL)

ExitSys("CreateSemaphore", EXIT\_FAILURE);

if ((hThreadProducer = CreateThread(NULL, 0, ThreadProcProducer, NULL, 0, &dwThreadIdProducer)) == NULL)

ExitSys("CreateThread", EXIT\_FAILURE);

if ((hThreadConsumer = CreateThread(NULL, 0, ThreadProcConsumer, NULL, 0, &dwThreadIdConsumer)) == NULL)

ExitSys("CreateThread", EXIT\_FAILURE);

WaitForSingleObject(hThreadProducer, INFINITE);

WaitForSingleObject(hThreadConsumer, INFINITE);

CloseHandle(g\_hSemProducer);

CloseHandle(g\_hSemConsumer);

return 0;

}

DWORD\_\_stdcall ThreadProcProducer(void \*param)

{

int i = 0;

for (;;) {

Sleep(rand() % 300);

WaitForSingleObject(g\_hSemProducer, INFINITE);

g\_share = i;

ReleaseSemaphore(g\_hSemConsumer, 1, NULL);

if (i == 99)

break;

++i;

}

return 0;

}

DWORD\_\_stdcall ThreadProcConsumer(void \*param)

{

int val;

for (;;) {

WaitForSingleObject(g\_hSemConsumer, INFINITE);

val = g\_share;

ReleaseSemaphore(g\_hSemProducer, 1, NULL);

printf("%d ", val);

Sleep(rand() % 300);

if (val == 99)

break;

}

printf("\n");

return 0;

}

void ExitSys(LPCSTRlpszMsg, intstatus)

{

DWORD dwLastError = GetLastError();

LPTSTR lpszErr;

if (FormatMessage(FORMAT\_MESSAGE\_ALLOCATE\_BUFFER | FORMAT\_MESSAGE\_FROM\_SYSTEM, NULL, dwLastError,

MAKELANGID(LANG\_NEUTRAL, SUBLANG\_DEFAULT), (LPTSTR)&lpszErr, 0, NULL)) {

fprintf(stderr, "%s: %s", lpszMsg, lpszErr);

LocalFree(lpszErr);

}

exit(status);

}

Bu programı WaitForSingleObject ve ReleaseSemaphore kısımlarını kapatarak da test ediniz. Semaphore kontrolünün olmadığı durumda pek çok bilginin kaçtığı, bazılarının ise birden fazla kez işlendiği görülecektir. Yuklarıdaki program UNIX/Linux Sistemlerinde şöyle yazılabilir:

#include<stdio.h>

#include<stdlib.h>

#include<string.h>

#include<unistd.h>

#include<pthread.h>

#include<semaphore.h>

void \*thread\_proc1(void \*param);

void \*thread\_proc2(void \*param);

sem\_t g\_semProducer;

sem\_t g\_semConsumer;

int g\_share;

int main(void)

{

pthread\_t tid1, tid2;

int result;

if (sem\_init(&g\_semProducer, 0, 1) < 0){

perror("sem\_init");

exit(EXIT\_FAILURE);

}

if (sem\_init(&g\_semConsumer, 0, 0) < 0){

perror("sem\_init");

exit(EXIT\_FAILURE);

}

if ((result = pthread\_create(&tid1, NULL, thread\_proc1, NULL)) != 0) {

printf("pthread\_create: %s\n", strerror(result));

exit(EXIT\_FAILURE);

}

if ((result = pthread\_create(&tid2, NULL, thread\_proc2, NULL)) != 0) {

printf("pthread\_create: %s\n", strerror(result));

exit(EXIT\_FAILURE);

}

pthread\_join(tid1, NULL);

pthread\_join(tid2, NULL);

sem\_destroy(&g\_semProducer);

sem\_destroy(&g\_semConsumer);

return 0;

}

void \*thread\_proc1(void \*param)

{

int i = 0;

for (;;) {

usleep(rand() % 300000);

sem\_wait(&g\_semProducer);

g\_share = i;

sem\_post(&g\_semConsumer);

if (i == 99)

break;

++i;

}

returnNULL;

}

void \*thread\_proc2(void \*param)

{

int val;

for (;;) {

sem\_wait(&g\_semConsumer);

val = g\_share;

sem\_post(&g\_semProducer);

usleep(rand() % 300000);

printf("%d ", val);

fflush(stdout);

if (val == 99)

break;

}

printf("\n");

returnNULL;

}

**Üretici Tüketici Probleminin Tamponlu Verrsiyonu**

Üretici Tüketici probleminde ortadaki paylaşılan alanın tek bir elemandan oluşması yerine onun birden fazla elemandan oluşan bir tampon bölge olması sistemi hızlandırır. Çünkü bu durumda bekleme olasılığı azalacaktır. Şöyle ki: Bu durumda üretici tampon tamamen doluyken, tüketici de tampon tamamen boşken bekleme yapar. Bu ortadaki paylaşınan tampon kuyruk veri yapısıyla gerçekleştirilmelidir. Bu çözümde semaphore sayaçları kuyruk uzunluğu kadar olmalıdır. Kuyruklar sonraki bölümde ele alınacaktır. Ancak burada sanki ele alınmış gibi kodlama yapacağız.

Tamponlu örnek çözüm şöyle olabilir:

#include<stdio.h>

#include<stdlib.h>

#include<time.h>

#include<Windows.h>

#defineQUEUE\_SIZE 10

void ExitSys(LPCSTR lpszMsg, int status);

DWORD\_\_stdcall ThreadProcProducer(void \*param);

DWORD\_\_stdcall ThreadProcConsumer(void \*param);

HANDLE g\_hSemProducer;

HANDLE g\_hSemConsumer;

int g\_share[QUEUE\_SIZE];

int g\_head, g\_tail;

int g\_count;

int main(void)

{

DWORD dwThreadIdProducer, dwThreadIdConsumer;

HANDLE hThreadProducer, hThreadConsumer;

srand(time(NULL));

if ((g\_hSemProducer = CreateSemaphore(NULL, QUEUE\_SIZE, QUEUE\_SIZE, NULL)) == NULL)

ExitSys("CreateSemaphore", EXIT\_FAILURE);

if ((g\_hSemConsumer = CreateSemaphore(NULL, 0, QUEUE\_SIZE, NULL)) == NULL)

ExitSys("CreateSemaphore", EXIT\_FAILURE);

if ((hThreadProducer = CreateThread(NULL, 0, ThreadProcProducer, NULL, 0, &dwThreadIdProducer)) == NULL)

ExitSys("CreateThread", EXIT\_FAILURE);

if ((hThreadConsumer = CreateThread(NULL, 0, ThreadProcConsumer, NULL, 0, &dwThreadIdConsumer)) == NULL)

ExitSys("CreateThread", EXIT\_FAILURE);

WaitForSingleObject(hThreadProducer, INFINITE);

WaitForSingleObject(hThreadConsumer, INFINITE);

CloseHandle(g\_hSemProducer);

CloseHandle(g\_hSemConsumer);

return 0;

}

DWORD\_\_stdcall ThreadProcProducer(void \*param)

{

int i = 0;

for (;;) {

Sleep(rand() % 300);

WaitForSingleObject(g\_hSemProducer, INFINITE);

g\_share[g\_head++] = i;

g\_head %= QUEUE\_SIZE;

ReleaseSemaphore(g\_hSemConsumer, 1, NULL);

if (i == 99)

break;

++i;

}

return 0;

}

DWORD\_\_stdcall ThreadProcConsumer(void \*param)

{

int val;

for (;;) {

WaitForSingleObject(g\_hSemConsumer, INFINITE);

val = g\_share[g\_tail++];

g\_tail %= QUEUE\_SIZE;

ReleaseSemaphore(g\_hSemProducer, 1, NULL);

printf("%d ", val);

Sleep(rand() % 300);

if (val == 99)

break;

}

printf("\n");

return 0;

}

void ExitSys(LPCSTRlpszMsg, intstatus)

{

DWORD dwLastError = GetLastError();

LPTSTR lpszErr;

if (FormatMessage(FORMAT\_MESSAGE\_ALLOCATE\_BUFFER | FORMAT\_MESSAGE\_FROM\_SYSTEM, NULL, dwLastError,

MAKELANGID(LANG\_NEUTRAL, SUBLANG\_DEFAULT), (LPTSTR)&lpszErr, 0, NULL)) {

fprintf(stderr, "%s: %s", lpszMsg, lpszErr);

LocalFree(lpszErr);

}

exit(status);

}

**Üretici Tüketici Probleminin Diğer Biçimleri**

Üretici tüketici probleminde birden fazla üretici ve/veya birden fazla tüketici olabilir. Birden fazla üreticinin olduğu durumda aslında fazlaca bir sorun yoktur. Üreticiler aynı semaphore'u kullanırlar. Ancak iletim sonlandığında bunların bazıları blokede bekler durumda kalabilir. Bunların o durumdan kurtarılması gerekir. Bu değişik biçimlerde yapılabilir. Birden fazla tüketici de yine aynı semaphore'u kullanabilir. Aynı blokede kalma durumu yine tüketiciler için de söz konusu olabilmektedir.

**Windows'ta Event Senkronizasyon Nesneleri**

Windows'ta event isminde bir senkronizasyon nesnesi vardır. Bu nesne belli bir koşul sağlanana kadar blokeye yol açar. Bunun benzeri UNIX/Linux sistemlerinde "koşul değişkeni (conditional variable)" ismiyle bulunmaktadır. Event nesnelerinin kullanımı şöyledir:

1) Nesne CreateEvent fonksiyonuyla yaratılır:

HANDLE WINAPI CreateEvent(

\_\_in LPSECURITY\_ATTRIBUTES *lpEventAttributes*,

\_\_in BOOL *bManualReset*,

\_\_in BOOL *bInitialState*,

\_\_in LPCTSTR *lpName*

);

Fonksiyonun birinci parametresi nesnenin güvenlik bilgilerini içerir. Bu parametre NULL geçilebilir. İkinci parametre event nesnesinin "otomatik" mi yoksa "manuel" mi olacağını belirtir. Üçüncü parametre event nesnesinin başlangıçta "açık" mı "yoksa "kapalı" mı olacağını belirtir. Son parametre proseslerarası kullanım için gereken isimdir.

2) Olay gerçekleşene kadar bekleme işlemi WaitForSingleObject fonksiyonıula yapılır.

3) Olay gerçekleştiğinde nesneyi açık duruma geçirmek için SetEvent fonksiyonu kullanılır:

BOOL WINAPI SetEvent(

\_\_in HANDLE *hEvent*

);

Event nesnesi otomatikse WaitForXXX fonksiyonlarından geçildiğinde otomatik olarak yeniden kapalı duruma geçer. Eğer nesne manuelse kapalı duruma geçirmek için ResetEvent fonksiyonu kullanılmalıdır:

BOOL WINAPI ResetEvent(

\_\_in HANDLE *hEvent*

);

4) Kullanım bittikten sonra nesne CloseHandle fonksiyonuyla yok edilir.

Örnek bir event kullanım şöyle olabilir:

#include<stdio.h>

#include<stdlib.h>

#include<time.h>

#include<Windows.h>

void ExitSys(LPCSTR lpszMsg, int status);

DWORD\_\_stdcall ThreadProc1(void \*param);

DWORD\_\_stdcall ThreadProc2(void \*param);

HANDLE g\_hEvent;

int main(void)

{

DWORD dwThreadIdProc1, dwThreadIdProc2;

HANDLE hThreadProc1, hThreadProc2;

if ((g\_hEvent = CreateEvent(NULL, FALSE, FALSE, NULL)) == NULL)

ExitSys("CreateEvent", EXIT\_FAILURE);

if ((hThreadProc1 = CreateThread(NULL, 0, ThreadProc1, NULL, 0, &dwThreadIdProc1)) == NULL)

ExitSys("CreateThread", EXIT\_FAILURE);

if ((hThreadProc2 = CreateThread(NULL, 0, ThreadProc2, NULL, 0, &dwThreadIdProc2)) == NULL)

ExitSys("CreateThread", EXIT\_FAILURE);

WaitForSingleObject(hThreadProc1, INFINITE);

WaitForSingleObject(hThreadProc2, INFINITE);

return 0;

}

DWORD\_\_stdcall ThreadProc1(void \*param)

{

printf("ThreadProc1 is waiting...\n");

WaitForSingleObject(g\_hEvent, INFINITE);

printf("Ok, thanks...\n");

return 0;

}

DWORD\_\_stdcall ThreadProc2(void \*param)

{

printf("Press ENTER to continue...\n");

getchar();

SetEvent(g\_hEvent);

return 0;

}

void ExitSys(LPCSTRlpszMsg, intstatus)

{

DWORD dwLastError = GetLastError();

LPTSTR lpszErr;

if (FormatMessage(FORMAT\_MESSAGE\_ALLOCATE\_BUFFER | FORMAT\_MESSAGE\_FROM\_SYSTEM, NULL, dwLastError,

MAKELANGID(LANG\_NEUTRAL, SUBLANG\_DEFAULT), (LPTSTR)&lpszErr, 0, NULL)) {

fprintf(stderr, "%s: %s", lpszMsg, lpszErr);

LocalFree(lpszErr);

}

exit(status);

}

Üretici tüketici probleminin tekli versiyonu event nesneleriyle de çözülebilir. (Fakat tamponlu versiyonu çözülemez):

/\*ProducerConsumer.c \*/

#include<stdio.h>

#include<stdlib.h>

#include<time.h>

#include<Windows.h>

void ExitSys(LPCSTR lpszMsg, int status);

DWORD\_\_stdcall ThreadProcProducer(void \*param);

DWORD\_\_stdcall ThreadProcConsumer(void \*param);

HANDLE g\_hEventProducer;

HANDLE g\_hEventConsumer;

int g\_share;

int main(void)

{

DWORD dwThreadIdProducer, dwThreadIdConsumer;

HANDLE hThreadProducer, hThreadConsumer;

srand(time(NULL));

if ((g\_hEventProducer = CreateEvent(NULL, FALSE,TRUE, NULL)) == NULL)

ExitSys("CreateSemaphore", EXIT\_FAILURE);

if ((g\_hEventConsumer = CreateEvent(NULL, FALSE, FALSE, NULL)) == NULL)

ExitSys("CreateSemaphore", EXIT\_FAILURE);

if ((hThreadProducer = CreateThread(NULL, 0, ThreadProcProducer, NULL, 0, &dwThreadIdProducer)) == NULL)

ExitSys("CreateThread", EXIT\_FAILURE);

if ((hThreadConsumer = CreateThread(NULL, 0, ThreadProcConsumer, NULL, 0, &dwThreadIdConsumer)) == NULL)

ExitSys("CreateThread", EXIT\_FAILURE);

WaitForSingleObject(hThreadProducer, INFINITE);

WaitForSingleObject(hThreadConsumer, INFINITE);

CloseHandle(g\_hEventProducer);

CloseHandle(g\_hEventConsumer);

return 0;

}

DWORD\_\_stdcall ThreadProcProducer(void \*param)

{

int i = 0;

for (;;) {

Sleep(rand() % 300);

WaitForSingleObject(g\_hEventProducer, INFINITE);

g\_share = i;

SetEvent(g\_hEventConsumer);

if (i == 99)

break;

++i;

}

return 0;

}

DWORD\_\_stdcall ThreadProcConsumer(void \*param)

{

int val;

for (;;) {

WaitForSingleObject(g\_hEventConsumer, INFINITE);

val = g\_share;

SetEvent(g\_hEventProducer);

printf("%d ", val);

Sleep(rand() % 300);

if (val == 99)

break;

}

printf("\n");

return 0;

}

void ExitSys(LPCSTRlpszMsg, intstatus)

{

DWORD dwLastError = GetLastError();

LPTSTR lpszErr;

if (FormatMessage(FORMAT\_MESSAGE\_ALLOCATE\_BUFFER | FORMAT\_MESSAGE\_FROM\_SYSTEM, NULL, dwLastError,

MAKELANGID(LANG\_NEUTRAL, SUBLANG\_DEFAULT), (LPTSTR)&lpszErr, 0, NULL)) {

fprintf(stderr, "%s: %s", lpszMsg, lpszErr);

LocalFree(lpszErr);

}

exit(status);

}

**Okuma Yazma Kilitleri (Read/Write Locks)**

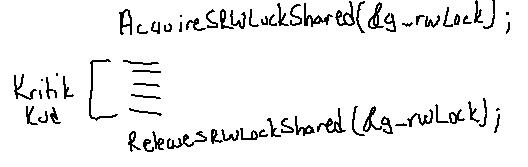
Bir veri yapısına birden fazla akışın okuma ve yazma yapma amacıyla eriştiğini düşünelim. Burada şüphesiz kritik kod oluşturmak gerekir. Ancak veri yapısından birden fazla thread'in okuma yapmasında sakınca yoktur. Ancak bir thread okuma yaparken diğer thread yazma yapmamalıdır. Bir thread yazma yapmak için veri yapısına erişmişse diğer thread'ler ne okuma ne de yazma amaçlı aynı veri yapısına erişmelidir. İşte bunu sağlayan senkronizasyon nesnesine okuma yazma kilitleri denilmektedir. Okuma yazma kilitleri hem Windows'ta hem de UNIX/Linux sistemlerinde benzer biçimde bulunmaktadır.

**Windows'ta Okuma Yazma Kilitleri**

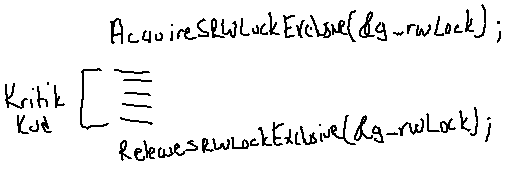
Windows'ta okuma yazma kilitleri şöyle kullanılmaktadır:

1) Global düzeyde RWLOCK isimli yapı türünden bir nesne tanımlanır. Ve bu nesne InitializeSRWLokc fonksiyonuyla ilklenir.

2) Kaynağa okuma amaçlı erişiliyorsa kritik kod şöyle oluşturulmalıdır:



3) Kaynağa yazma amaçlı erişiliyorsa kritik kod şöyle oluşturulmalıdır:



Söz konusu fonksiyonların prototipleri şöyledir:

VOID WINAPI InitializeSRWLock(

\_\_out PSRWLOCK *SRWLock*

);

VOID WINAPI AcquireSRWLockExclusive(

\_\_in\_out PSRWLOCK *SRWLock*

);

VOID WINAPI AcquireSRWLockShared(

\_\_in\_out PSRWLOCK *SRWLock*

);

VOID WINAPI ReleaseSRWLockExclusive(

\_\_in\_out PSRWLOCK *SRWLock*

);

VOID WINAPI ReleaseSRWLockShared(

\_\_in\_out PSRWLOCK *SRWLock*

);

**UNIX/Linux Sistemlerinde Okuma Yazma Kilitleri**

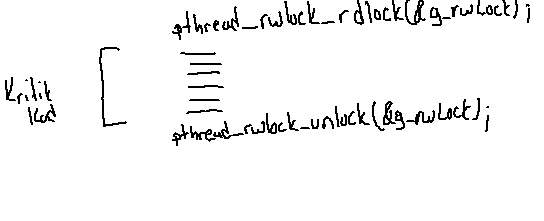
Bu sistemlerde de okuma yazma kilitleri şöyle kullanılır:

1) pthread\_rwlock\_t türünden global bir nesne tanımlanır ve bu nesne pthread\_rwlock\_init fonksiyonuyla ilklenir.

int pthread\_rwlock\_init(pthread\_rwlock\_t \**rwlock***,** const pthread\_rwlockattr\_t \**attr***);**

Fonksiyonun ikinci parametre nesnenin diğer özelliklerini belirtir. NULL geçilebilir.

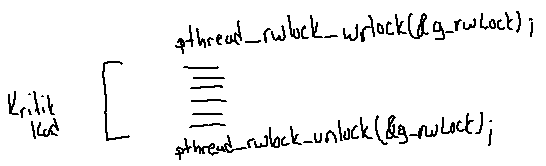
2) Kaynağa okuma amaçlı erişiliyorsa kritik kod şöyle oluşturulmalıdır:



int pthread\_rwlock\_rdlock(pthread\_rwlock\_t *\*rwlock***);**

int pthread\_rwlock\_unlock(pthread\_rwlock\_t *\*rwlock***);**

3) Kaynağa yazma amaçlı erişiliyorsa kritik kod şöyle oluşturulmalıdır:



4) İşlemler bitince rwlock nesnesi pthread\_rwlock\_destroy fonksiyonuyla yok edilir:

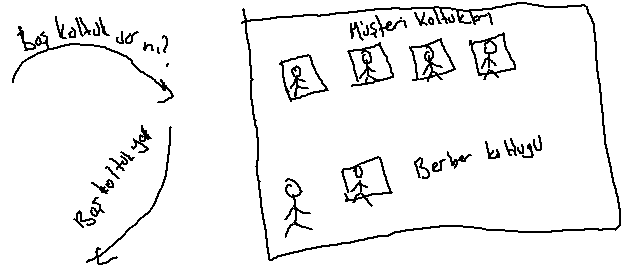
int pthread\_rwlock\_destroy(pthread\_rwlock\_t \*rwlock**);**

**Literatürde Çok Geçen Fakat Uygulamada Çok Önemi Olmayan Birkaç Senkronizasyon Problemi**

Üretici-Tüketici problemi uygulamada en fazla karşılaşılan senkronizasyon problemidir. Ancak literatürde geçen birkaç. senkronizasyon problemi daha vardır. Bunların önemli bir uygulama alanı olduğu söylenemz.

**Uyuyan Berber Problemi (Sleeping Barber Problem)**

Bu problemde bir berber dükkanı vardır. Dükkanda belirli sayıda bekleme koltuğu tabi bir berber koltuğu ve berber bulunur. Berber bekleme koltuklarında hiç müşteri yoksa uyur. (Yani bloke olur. CPU zamanı harcamaz.) Müşteri içeri girmek istediğinde koltuklara bakar. Koltuklar doluysa beklemez. Bekleme koltukları boşsa orada bekler. Beklerken de uyur (bloke olur). Berber koltuğundaki kişinin tıraşı bittiğinde bekleyen kişilerden sıradaki uyandırılarak kultuğa oturtulur.



Problemin çözümünde bekleme koltuklerı için bir semafor alınır. Bu semaforun başlangıçtaki sayacı sıfırdır. Berber bu semafora bakarak işlem yapar. Demek ki başlangıçta berber hiç müşteri olmadığı için uyuyacaktır. Bir müşteri geldiğinde semafor sayacı 1 artar. berber böylece uyanır. Onu koltuğa alır. Semafor sayacını bir eksiltir. Dışarıdan gelen müşterinin koltukların dolu olduğunu anlaması gerekir. Bunun için bir syaç tutulur. Tabi sayaç da kritik kod bloğu ile azaltılıp yükseltilir. Pekiyi müşterinin koltukta uyuması ve berberin de müşteriyi uyandırması nasıl olacaktır? İşte bunun için de ayrı bir senkronizasyon nesnesine ihtiyaç vardır. Binary bir semafor işi görebilir. Mutex olmaz. Çünkübaşka bir thread mutex'in kilidini açamaz. Windows'taki Event nesnesi de kullanılabilir. Benzer biçimde UNIX/Linux sistemlerindeki "koşullu değişkenler (conditional variables)" de bu amaçla kullanılabilirler.

Uyuan Berber Probleminin Çözümü Windows sistemlerinde şöyle yapılabilir:

#include<stdio.h>

#include<stdlib.h>

#include<time.h>

#include<Windows.h>

#defineNUMBER\_OF\_SEATS 3

#defineNUMBER\_OF\_CUSTOMERS 20

void ExitSys(LPCSTR lpszMsg, int status);

DWORD\_\_stdcall ThreadProcBarber(void \*param);

DWORD\_\_stdcall ThreadProcCustomer(void \*param);

HANDLE CreateBarber(constchar \*name);

HANDLE CreateCustomer(constchar \*name);

HANDLE g\_hSemaphoreBarber;

HANDLE g\_hMutexSeat;

HANDLE g\_hEventCustomer;

HANDLE g\_hEventShaving;

int g\_emptySeatCount;

int main(void)

{

HANDLE allHandles[10];

char \*name;

int i;

g\_emptySeatCount = NUMBER\_OF\_SEATS;

if ((g\_hSemaphoreBarber = CreateSemaphore(NULL, 0, NUMBER\_OF\_SEATS, NULL)) == NULL)

ExitSys("CreateSemaphore", EXIT\_FAILURE);

if ((g\_hMutexSeat = CreateMutex(NULL, FALSE, NULL)) == NULL)

ExitSys("CreateMutex", EXIT\_FAILURE);

if ((g\_hEventCustomer = CreateEvent(NULL, FALSE, FALSE, NULL)) == NULL)

ExitSys("CreateSemaphore", EXIT\_FAILURE);

if ((g\_hEventShaving = CreateEvent(NULL, FALSE, FALSE, NULL)) == NULL)

ExitSys("CreateSemaphore", EXIT\_FAILURE);

allHandles[0] = CreateBarber("Barber");

Sleep(500);

for (i = 1; i <= NUMBER\_OF\_CUSTOMERS; ++i) {

if ((name = (char \*)malloc(10)) == NULL) {

fprintf(stderr, "cannot allocate memory!..\n");

exit(EXIT\_FAILURE);

}

sprintf(name, "Customer-%d", i);

allHandles[i] = CreateCustomer(name);

Sleep(rand() % 300);

}

WaitForMultipleObjects(NUMBER\_OF\_CUSTOMERS, allHandles, TRUE, INFINITE);

return 0;

}

HANDLE CreateBarber(constchar \*name)

{

HANDLE hThreadBarber;

DWORD dwThreadIdBarber;

if ((hThreadBarber = CreateThread(NULL, 0, ThreadProcBarber, (void \*)name, 0, &dwThreadIdBarber)) == NULL)

ExitSys("CreateThread", EXIT\_FAILURE);

return hThreadBarber;

}

HANDLE CreateCustomer(constchar \*name)

{

HANDLE hThreadCustomer;

DWORD dwThreadIdCustomer;

if ((hThreadCustomer = CreateThread(NULL, 0, ThreadProcCustomer, (void \*)name, 0, &dwThreadIdCustomer)) == NULL)

ExitSys("CreateThread", EXIT\_FAILURE);

return hThreadCustomer;

}

DWORD\_\_stdcall ThreadProcBarber(void \*param)

{

constchar \*name = (constchar \*)param;

for (;;) {

Sleep(200);

printf("(%s) Berber uyuyor\n", name);

WaitForSingleObject(g\_hSemaphoreBarber, INFINITE);

printf("(%s) Berber uyandi musteriyi koltuga davet etti\n", name);

SetEvent(g\_hEventCustomer);

printf("Berber musteriyi tiras ediyor\n");

Sleep(rand() % 300);

printf("Berber tirasi bitirdi\n");

SetEvent(g\_hEventShaving);

WaitForSingleObject(g\_hMutexSeat, INFINITE);

++g\_emptySeatCount;

ReleaseMutex(g\_hMutexSeat);

}

return 0;

}

DWORD\_\_stdcall ThreadProcCustomer(void \*param)

{

constchar \*name = (constchar \*)param;

printf("(%s) musteri geldi ve dukkana bakiyor\n", name);

Sleep(rand() % 300);

WaitForSingleObject(g\_hMutexSeat, INFINITE);

if (g\_emptySeatCount > 0) {

printf("(%s) bos koltuga oturdu ve uyudu\n", name);

--g\_emptySeatCount;

ReleaseMutex(g\_hMutexSeat);

ReleaseSemaphore(g\_hSemaphoreBarber, 1, NULL);

WaitForSingleObject(g\_hEventCustomer, INFINITE);

printf("(%s) musteri uyandi berber koltuguna oturdu\n", name);

WaitForSingleObject(g\_hEventShaving, INFINITE);

printf("(%s) musteri tirasini oldu, dukkandan cikiyor\n", name);

}

else {

printf("(%s) musteri dukkana giremedn ayriliyor\n", name);

ReleaseMutex(g\_hMutexSeat);

}

return 0;

}

void ExitSys(LPCSTRlpszMsg, intstatus)

{

DWORD dwLastError = GetLastError();

LPTSTR lpszErr;

if (FormatMessage(FORMAT\_MESSAGE\_ALLOCATE\_BUFFER | FORMAT\_MESSAGE\_FROM\_SYSTEM, NULL, dwLastError,

MAKELANGID(LANG\_NEUTRAL, SUBLANG\_DEFAULT), (LPTSTR)&lpszErr, 0, NULL)) {

fprintf(stderr, "%s: %s", lpszMsg, lpszErr);

LocalFree(lpszErr);

}

exit(status);

}

Burada birden fazla müşteri koltukta bekliyorsa obnlardan bir tanesinin uyandırılması Event nesneleriyle yapılmıştır. Bu tür nesnelerin hepsi (hem Windows'ta hem de UNIX/Linux sistemlerinde) bekleyen thread'lerden hangisinin uyandırılacağı konusunda bir garanti vermemektedir. Bu garantini verilmesi için bir kuyruk sistemi kullanılabilir. Kuyruğun içerisinde koltukta bekleyen müşterilere ilişkin event nesneleri olur.

#include<stdio.h>

#include<stdlib.h>

#include<time.h>

#include<Windows.h>

#defineNUMBER\_OF\_SEATS 3

#defineNUMBER\_OF\_CUSTOMERS 20

void ExitSys(LPCSTR lpszMsg, int status);

DWORD\_\_stdcall ThreadProcBarber(void \*param);

DWORD\_\_stdcall ThreadProcCustomer(void \*param);

HANDLE CreateBarber(constchar \*name);

HANDLE CreateCustomer(constchar \*name);

HANDLE g\_hSemaphoreBarber;

HANDLE g\_hMutexSeat;

HANDLE g\_hEventShaving;

int g\_emptySeatCount;

HANDLE g\_handleQueue[NUMBER\_OF\_SEATS];

int g\_head, g\_tail;

int main(void)

{

HANDLE allHandles[10];

char \*name;

int i;

g\_emptySeatCount = NUMBER\_OF\_SEATS;

if ((g\_hSemaphoreBarber = CreateSemaphore(NULL, 0, NUMBER\_OF\_SEATS, NULL)) == NULL)

ExitSys("CreateSemaphore", EXIT\_FAILURE);

if ((g\_hMutexSeat = CreateMutex(NULL, FALSE, NULL)) == NULL)

ExitSys("CreateMutex", EXIT\_FAILURE);

if ((g\_hEventShaving = CreateEvent(NULL, FALSE, FALSE, NULL)) == NULL)

ExitSys("CreateSemaphore", EXIT\_FAILURE);

allHandles[0] = CreateBarber("Barber");

Sleep(500);

for (i = 1; i <= NUMBER\_OF\_CUSTOMERS; ++i) {

if ((name = (char \*)malloc(10)) == NULL) {

fprintf(stderr, "cannot allocate memory!..\n");

exit(EXIT\_FAILURE);

}

sprintf(name, "Customer-%d", i);

allHandles[i] = CreateCustomer(name);

Sleep(rand() % 300);

}

WaitForMultipleObjects(NUMBER\_OF\_CUSTOMERS, allHandles, TRUE, INFINITE);

return 0;

}

HANDLE CreateBarber(constchar \*name)

{

HANDLE hThreadBarber;

DWORD dwThreadIdBarber;

if ((hThreadBarber = CreateThread(NULL, 0, ThreadProcBarber, (void \*)name, 0, &dwThreadIdBarber)) == NULL)

ExitSys("CreateThread", EXIT\_FAILURE);

return hThreadBarber;

}

HANDLE CreateCustomer(constchar \*name)

{

HANDLE hThreadCustomer;

DWORD dwThreadIdCustomer;

if ((hThreadCustomer = CreateThread(NULL, 0, ThreadProcCustomer, (void \*)name, 0, &dwThreadIdCustomer)) == NULL)

ExitSys("CreateThread", EXIT\_FAILURE);

return hThreadCustomer;

}

DWORD\_\_stdcall ThreadProcBarber(void \*param)

{

constchar \*name = (constchar \*)param;

HANDLE hEventCustomer;

for (;;) {

Sleep(200);

printf("Berber uyuyor\n");

WaitForSingleObject(g\_hSemaphoreBarber, INFINITE);

printf("Berber uyandi musteriyi koltuga davet etti\n");

hEventCustomer = g\_handleQueue[g\_head++];

g\_head %= NUMBER\_OF\_SEATS;

SetEvent(hEventCustomer);

printf("Berber musteriyi tiras ediyor\n");

Sleep(rand() % 300);

printf("Berber tirasi bitirdi\n");

SetEvent(g\_hEventShaving);

WaitForSingleObject(g\_hMutexSeat, INFINITE);

++g\_emptySeatCount;

ReleaseMutex(g\_hMutexSeat);

}

return 0;

}

DWORD\_\_stdcall ThreadProcCustomer(void \*param)

{

constchar \*name = (constchar \*)param;

HANDLE hEventCustomer;

printf("(%s) musteri geldi ve dukkana bakiyor\n", name);

Sleep(rand() % 300);

WaitForSingleObject(g\_hMutexSeat, INFINITE);

if (g\_emptySeatCount > 0) {

printf("(%s) bos koltuga oturdu ve uyudu\n", name);

--g\_emptySeatCount;

if ((hEventCustomer = CreateEvent(NULL, FALSE, FALSE, NULL)) == NULL)

ExitSys("CreateSemaphore", EXIT\_FAILURE);

g\_handleQueue[g\_tail++] = hEventCustomer;

g\_tail %= NUMBER\_OF\_SEATS;

ReleaseMutex(g\_hMutexSeat);

ReleaseSemaphore(g\_hSemaphoreBarber, 1, NULL);

WaitForSingleObject(hEventCustomer, INFINITE);

printf("(%s) musteri uyandi berber koltuguna oturdu\n", name);

WaitForSingleObject(g\_hEventShaving, INFINITE);

printf("(%s) musteri tirasini oldu, dukkandan cikiyor\n", name);

CloseHandle(hEventCustomer);

}

else {

printf("(%s) musteri dukkana giremedn ayriliyor\n", name);

ReleaseMutex(g\_hMutexSeat);

}

return 0;

}

void ExitSys(LPCSTRlpszMsg, intstatus)

{

DWORD dwLastError = GetLastError();

LPTSTR lpszErr;

if (FormatMessage(FORMAT\_MESSAGE\_ALLOCATE\_BUFFER | FORMAT\_MESSAGE\_FROM\_SYSTEM, NULL, dwLastError,

MAKELANGID(LANG\_NEUTRAL, SUBLANG\_DEFAULT), (LPTSTR)&lpszErr, 0, NULL)) {

fprintf(stderr, "%s: %s", lpszMsg, lpszErr);

LocalFree(lpszErr);

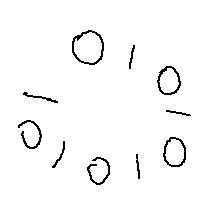
}

exit(status);

}

**Yemek Yiyen Filozoflar Problemi (Dining Philosophers)**

Bu problemde bir grup kişi yuvarlak yemek masasında oturmaktadır. Onların sol ve sağında birer çatal vardır. Fakat yemeği yiyebilmek için iki çatala da ihtiyaç duyarlar. Yemek yeme süreci bir döngü içerisinde iki çatalı alıp biraz yiyip bırakmak biçiminde olmaktadır.



Problem 1965 yılında Edsger Dijkastra tarafından tanımlanmıştır. Burada amaç kişilerin aç kalmasını (starvation) ve kilitlenmeyi (dedlock) engellemektir. Kilitlenme bir thread diğerini beklerken diğerinin de başka bir şeyi beklemesi ve böylece sonsuz bir beklemenin oluşması durumudur. Örneğin her kişi önce solundaki ya da sağındaki çatalı alırsa bir kilitlenme oluşur. Her kişi iki çatalı birden almaya çalışırsa da alamayabilir. Yine kilitlenme oluşma riski vardır. Ayrıca bazı kişilere iki çatalı alma sırası hiç gelemeyebilir. Yani bazılarının aç kalma riski de vardır.

Problemin çözümü burada yapılmayacaktır. Ancak klasik çözümde kişiler iki senkronizasyon nesnesiyle (örneğin semaphore) önce soldaki, sonra sağdaki çatalları almaya çalışırlar. (Çözüm için Tanenbaum'un "Operating System Design And Implementation" kitabına bakılabilir.)

**Sigara İçen Kişiler Problemi (Cigarette Smokers Problem)**

Bu problemde sihgara içmek isteyen üç kişi vardır. Bunların sigara içmesi için tütün, kağıt ve kibriti elde etmesi gerekir. Problemde bir de yürütücü (agent) vardır. Yürütücü rastgele iki malzemeyi masaya koyar. İhtiyacı olanlar alıp sigara içmeye çalışır. Ancak sigara içmek isteyen kişilerde bu malzemelerden yalnızca biri zaten sınırsız ölçüde vardır. Yani birinde tütün, birinde kağıt, birinde de kibrit sınırsız ölçüde zaten bulunmaktadır. O halde bu kişiler diğer iki malzemeyi tedarik etmeye çalışırlar. Örneğin, yürütücü masaya kibrit ve kağıt koysun. Bunun ikisini de tütüne sahip kişi alırsa sigarayı içer. Fakat farklı kişiler alırsa yürütücünün diğer iki malzemeyi bırakmasını bekler. Problemde hedef en az bekleme oluşturacak biçimde tiryakilerin maksimum ölçğde sigara içmesini sağlamaktır.

Çözüm için yine her malzemeyi elde bulunduranlar diğer ikisi için iki ayrı semaphore ile beklerler. Ancak çözüm burada verilmeyecektir.

**Atomik İşlemler**

Normal olarak bir makina komutu çalışırken kesilme olmaz. Bir işlemin atomik olması demek araya kimse girmeden başından sonuna kadar tek başına yapılması demektir. Yani makina komutları atomiktir. Ancak biz C'de bir işlemi tek bir ifadeye yapmamız derleyicinin onu tek bir makina komutuyla yapacağı anlamına gelmez. Örneğin:

++g\_val;

Bizim bunu tek bir operatörle yapmış olmamız derleyicinin bunu tek bir makina komutuyla yapacağı anlamına gelmez. Örneğin Intel'de bu işlem tek bir komutuyla yapılabilir:

INC g\_val

Fakat bu işlem birkaç makina komutuyla da yapılabilir:

MOV EAX, g\_val

INC EAX

MOV g\_val, EAX

Şimdi aynı global değişkeni iki thread de belli miktar artırsın. Acaba sonuçta global değişkenin değişkenin değeri olması gereken değerde olacak mıdır?

#include<stdio.h>

#include<stdlib.h>

#include<Windows.h>

#defineN 1000000

void ExitSys(LPCSTR lpszMsg, int status);

DWORD\_\_stdcall ThreadProc1(void \*param);

DWORD\_\_stdcall ThreadProc2(void \*param);

int g\_val;

int main(void)

{

DWORD dwThreadId1, dwThreadId2;

HANDLE hThread1, hThread2;

if ((hThread1 = CreateThread(NULL, 0, ThreadProc1, NULL, 0, &dwThreadId1)) == NULL)

ExitSys("CreateThread", EXIT\_FAILURE);

if ((hThread2 = CreateThread(NULL, 0, ThreadProc2, NULL, 0, &dwThreadId2)) == NULL)

ExitSys("CreateThread", EXIT\_FAILURE);

WaitForSingleObject(hThread1, INFINITE);

WaitForSingleObject(hThread2, INFINITE);

printf("%d\n", g\_val);

return 0;

}

DWORD\_\_stdcall ThreadProc1(void \*param)

{

int i;

for (i = 0; i <N; ++i)

++g\_val;

return 0;

}

DWORD\_\_stdcall ThreadProc2(void \*param)

{

int i;

for (i = 0; i <N; ++i)

++g\_val;

return 0;

}

void ExitSys(LPCSTRlpszMsg, intstatus)

{

DWORD dwLastError = GetLastError();

LPTSTR lpszErr;

if (FormatMessage(FORMAT\_MESSAGE\_ALLOCATE\_BUFFER | FORMAT\_MESSAGE\_FROM\_SYSTEM, NULL, dwLastError,

MAKELANGID(LANG\_NEUTRAL, SUBLANG\_DEFAULT), (LPTSTR)&lpszErr, 0, NULL)) {

fprintf(stderr, "%s: %s", lpszMsg, lpszErr);

LocalFree(lpszErr);

}

exit(status);

}

Yukarıdaki programda sayacın değerinin 2000000 olması gerekir. Halbuki böyle çıkmamaktadır. Bu tür basit işlemlerin bile kritik kod içerisinde yapılması gerekir. Örneğin:

#include<stdio.h>

#include<stdlib.h>

#include<Windows.h>

#defineN 1000000

void ExitSys(LPCSTR lpszMsg, int status);

DWORD\_\_stdcall ThreadProc1(void \*param);

DWORD\_\_stdcall ThreadProc2(void \*param);

int g\_val;

CRITICAL\_SECTION g\_cs;

int main(void)

{

DWORD dwThreadId1, dwThreadId2;

HANDLE hThread1, hThread2;

InitializeCriticalSection(&g\_cs);

if ((hThread1 = CreateThread(NULL, 0, ThreadProc1, NULL, 0, &dwThreadId1)) == NULL)

ExitSys("CreateThread", EXIT\_FAILURE);

if ((hThread2 = CreateThread(NULL, 0, ThreadProc2, NULL, 0, &dwThreadId2)) == NULL)

ExitSys("CreateThread", EXIT\_FAILURE);

WaitForSingleObject(hThread1, INFINITE);

WaitForSingleObject(hThread2, INFINITE);

printf("%d\n", g\_val);

DeleteCriticalSection(&g\_cs);

return 0;

}

DWORD\_\_stdcall ThreadProc1(void \*param)

{

int i;

for (i = 0; i <N; ++i) {

EnterCriticalSection(&g\_cs);

++g\_val;

LeaveCriticalSection(&g\_cs);

}

return 0;

}

DWORD\_\_stdcall ThreadProc2(void \*param)

{

int i;

for (i = 0; i <N; ++i) {

EnterCriticalSection(&g\_cs);

++g\_val;

LeaveCriticalSection(&g\_cs);

}

return 0;

}

void ExitSys(LPCSTRlpszMsg, intstatus)

{

DWORD dwLastError = GetLastError();

LPTSTR lpszErr;

if (FormatMessage(FORMAT\_MESSAGE\_ALLOCATE\_BUFFER | FORMAT\_MESSAGE\_FROM\_SYSTEM, NULL, dwLastError,

MAKELANGID(LANG\_NEUTRAL, SUBLANG\_DEFAULT), (LPTSTR)&lpszErr, 0, NULL)) {

fprintf(stderr, "%s: %s", lpszMsg, lpszErr);

LocalFree(lpszErr);

}

exit(status);

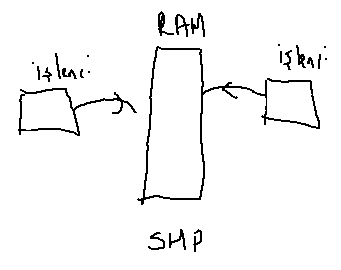
}

Bu tür basit işlemler için bile kritik kod oluşturulması biraz sıkıcı olabilmektedir.

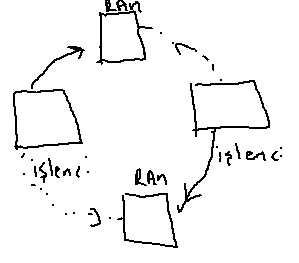
Ayrıca çok işlemcili ya da çok çekirdekli sistemlerde aynı tek bir makina komutunun atomikliği de tartışılır hale gelebilmektedir. Örneğin iki çekirdek de aşağıdaki makina komutunu tesadüfen aynı anda çalıştırmış olsun:

INC g\_val

Burada bazı sistemlerde yine bozulma olabilir. Bugün çok işlemcili ya da çok çekirdekli sistemlerde iki mimnari kullanılmaktadır: SMP ve NUMA. SMP (Symetric Multi Processor) mimarisinde işlemciler ya da çekirdekler aynı RAM'e bağlıdır. Dolayısıyla bus çatışması olmaması için biri RAM'e erişirken diğerini durdurur. (Yani bu süreç yüzünden 2 çekirdekli bir sistem iki katlı hızlı değildir. Olsa olsa %20-25 daha hızlıdır. 4 çekirdekliş sistem de 4 kat hızlı değildir.)

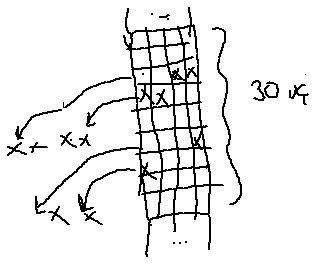


NUMA'da (Non Unified Memory Access) ise her işlemcinin ya da çekirdeğin RAM'i ayrıdır. Fakat bunlar birbirlerinin RAM'lerine (bank'lerine) de erişebilmektedir. Tabi bu durumda erişim daha yavaş olur (Zaten Non Unified Memory Access ismi buaradan gelmektedir.) Böylece bunlarda birbirini durdurma daha az rastlanır.



Bizim kullandığımız ana kartların çok büyük çoğunluğu SMP mimarisine uygundur. (Ancak örneğin AMD'nin Opteron işlemcilerinin kullanıldığı ana kartlarda NUMA mimarisi kullanılmaktadır.) Her iki mimarinin de bazı avantajları ve dezavantajları vardır.

Bazı işlemciler belleğe erişen makina komutlarında bus'ı başından sonuna kadar tutmazlar. Biraz tutup bırakıp, sonra yeniden tutup bırakabilirler. İşte durumda diğer işlemci ya da çekirdek aynı yeri yere erişirse bozuk bir değer oluşabilmektedir. Örneğin Intel işlemcilerinde bazı makina komutlarında bu durum vardır. Ayrıca hizalanmamış bilgilere erişen makina komutlarının çoğunda da bu problem vardır.



Yani Intel işlemcileri için şunları söyleyebiliriz:

1) Belleğe erişen bazı makina komutları adres hizalanmış olsun ya da olmasın çok işlemcili erişimde atomik değildir.

2) Hizalanmamış adresler çok işlemcili erişimde atomik değildir.

Fakat Intel işlemcilerinde (diğer işlemcilerde de benzer mekanizma vardır) bir komutun başına LOCK prefix'i getirilirse o komut çok işlemcili sistemlerde de atomik çalışmaktadır. Yani komutun başından sonuna kadar bus tutulur. Böylece sistem programcısı tek bir makina komutunun çok işlemcili sistemde atomik olmayacağı durumda komutun başına LOCK prefix'i getirmelidir. Örneğin:

LOCK ADD [EBX], EAX

Başka işlemcilerde bu durum göz önüne alınmalıdır. Özetle tek bir makina komutu bile çokişlemcili ya da çok çekirdekli sistemlerde atomik olmayabilmektedir. Bu konunun o sistem için araştırılması gerekir.

Tek işlemcili ya da çekirdekli sistemlerde her zaman makina komutları atomiktir. Çünkü zaten başka bir akış aynı yere erişlemez. Makina komutu çalışırken de thread'ler arası geçiş (context switch) gerçekleşmez.

İşte tek makina komutuyla yapılabilecek artırma, eksiltme gibi basit işlem için kritik kod oluşturmaya gerek kalmasın diye Microsoft bir grup InterlockedXXX API fonksiyonu bulundurmuştur. Bu fonksiyonlar işlemlerini tek bir makina komutuyla yaparlar üstelik de komutun başına LOCK prefix'i getirirler. Böylece kritik kod oluşturmanın maliyetinden kaçınılmış olur. Bu fonksiyonların tam listesine MSDN dokümanlarından bakılabilir. Örneğin IntelockedIncrement lock'lu tek bir makina komutuyla bir nesneyi artırmaktadır:

LONG \_\_cdecl InterlockedIncrement(

\_\_in\_out LONG volatile\* *Addend*

);

InterlocedAdd ise atomik toplama yapmaktadır:

LONG \_\_cdecl InterlockedAdd(

\_\_in\_out LONG volatile\* *Addend*,

\_\_in LONG *Value*

);

Şimdi hiç kritik kod oluşturmadan yukarıdaki global değişkenin artırılması örneğini InterlockedIncrement ile yapalım:

#include<stdio.h>

#include<stdlib.h>

#include<Windows.h>

#defineN 1000000

void ExitSys(LPCSTR lpszMsg, int status);

DWORD\_\_stdcall ThreadProc1(void \*param);

DWORD\_\_stdcall ThreadProc2(void \*param);

int g\_val;

int main(void)

{

DWORD dwThreadId1, dwThreadId2;

HANDLE hThread1, hThread2;

if ((hThread1 = CreateThread(NULL, 0, ThreadProc1, NULL, 0, &dwThreadId1)) == NULL)

ExitSys("CreateThread", EXIT\_FAILURE);

if ((hThread2 = CreateThread(NULL, 0, ThreadProc2, NULL, 0, &dwThreadId2)) == NULL)

ExitSys("CreateThread", EXIT\_FAILURE);

WaitForSingleObject(hThread1, INFINITE);

WaitForSingleObject(hThread2, INFINITE);

printf("%d\n", g\_val);

return 0;

}

DWORD\_\_stdcall ThreadProc1(void \*param)

{

int i;

for (i = 0; i <N; ++i)

InterlockedIncrement(&g\_val);

return 0;

}

DWORD\_\_stdcall ThreadProc2(void \*param)

{

int i;

for (i = 0; i <N; ++i)

InterlockedIncrement(&g\_val);

return 0;

}

void ExitSys(LPCSTRlpszMsg, intstatus)

{

DWORD dwLastError = GetLastError();

LPTSTR lpszErr;

if (FormatMessage(FORMAT\_MESSAGE\_ALLOCATE\_BUFFER | FORMAT\_MESSAGE\_FROM\_SYSTEM, NULL, dwLastError,

MAKELANGID(LANG\_NEUTRAL, SUBLANG\_DEFAULT), (LPTSTR)&lpszErr, 0, NULL)) {

fprintf(stderr, "%s: %s", lpszMsg, lpszErr);

LocalFree(lpszErr);

}

exit(status);

}

UNIX/Linux sistemlerinde Windows'taki InterlockedXXX fonksiyonlarının bir eşdeğeri var mıdır? Yanit: POSIX standartlarında ve glibc kütüphanesinde yoktur. Ancak <atomic.h> dosyası içerisinde bu fonksiyonlar bulunmaktadır. <atomic.h> dosyası aslında kernel'a ilişkin kaynak kodların içerisindeki bir dosyadır. Fakat bu dosyadaki fonksiyonlar inline assembly olarak yazıldıkları için doğrudan include edilerek kullanılabilirler. Eğer kernel kaynak kodları varsa bu dosyayı doğrudan oradan alabiliriz. Yoksa Internet'ten de indirebiliriz. Tabi bizim bulunduğumuz işlemciye ilişkin <atomic.h> dosyasını kullanmamız gerekir.

**Fonksiyonların Thread Güvenliliği (Thread Safety)**

Bir fonksiyonun thread güvenli (thread safe) olması demek o fonksiyonun çok thread'li ortamda birden fazla thread tarafından çağrıldığında soruna yol açmaması demektir. Yalnızca parametre ve yerel değişken kullanan fonksiyonlar thread güvenlidir. Fakat global ya da static yerel nesnelere, ortak kaynaklara kilitsiz erişen fonksiyonlar thread güvenli değillerdir. Thread güvenli olmayan fonksiyonların çok thread'li uygulamalarda kritik kod içerisinde çağrılması gerekir. Çok thread'li sistemlerde thread fonksiyonların thread güvenli olup olmadığı sorgulanmalıdır. Genellikle kütüphanelerin dokümantasyonlarında bunlar belirtilir. Örneğin Windows'un API fonksiyonlarının hepsi thread güvenlidir.

Pekiyi standart C fonksiyonları thread güvenli midir? İşte bazı fonksiyonlar static data kullandığı için thread güvenli olmama eğilimindedir. Örneğin:

localtime

asctime

rand

tmpnam

Fonksiyonları gibi... C standartlarında C11'e kadar hiç thread lafı edilmemiştir. C11'de thread'ler isteğe bağlı öğeler olarak dile dahil edilmiştir. Standartlarda thread lafının edilmemesi derleyici yazanların bu fonksiyonları thread güvenli ya da thread güvensiz yazabileceği anlamına gelir. Pekiyi örneğin biz localtime fonksiyonunu iki thread'ten de kullanıyorsak senkronize etmeli miyiz? Yanıt: Derleyiciyi tanımamız localtime fonksiyonunun o derleyici de thread güvenli olup olmadığını bilmemiz gerekir.

İşte Microsoft Visual Studio 2005'e kadar standart C kütüphanesinin "single threaded" ve "multi threaded" olmak üzere iki versiyonundan bulunduruyordu. Programcı da hangi kütüpheneyle programının link edileceğini proje ayarlarından seçiyordu. Yani Microsoft'ta eskiden standart C kütüphanesinin iki ayrı versiyonu vardı. Fakat Visual Studio 2005'ten itibaren artık Microsoft yalnızca "multi threaded" C kütüphanesi bulundurmaktadır. Single thread uygulamalarda biraz zaman kaybı söz konusu olsa da bu kayıp önemli olmayabilir.

gcc derleyicilerinde sorunlu C fonksiyonlarının ayrı isimlerle iki versiyonundan bulundurulmuştur. Sonu \_r ile biten fonksiyonlar (r harfi "reentrant" sözcüğünden geliyor) thread güvenlidir. Normal isimdekiler thread güvenli değildir. Yani örneğin gcc derleyicilerinde localtime fonksiyonu thread güvenli değildir. Halbuki localtime\_r fonksiyonu thread güvenlidir.

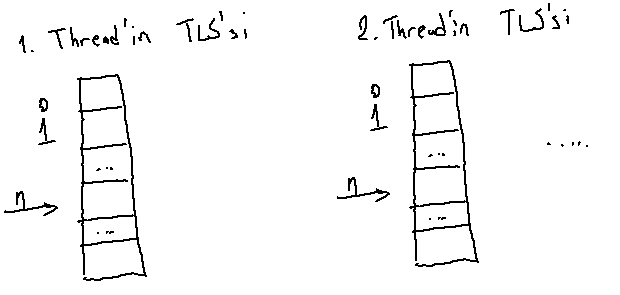
**Thread'e Özgü Alanlar**

Bazen global değişken gibi davranan thread'e özgü nesnelere gereksinim duyulmaktadır. Bilindiği gibi normal olarak bir global değişken tüm thread'lerde ortak biçimde kullanılmaktadır. Thread'e özgü global değişkenler olsaydı, nasıl bir etki oluşurdu? Örneğin g\_x thread'e özgü bir global değişken olsun. Bu durumda foo gibi bir fonksiyon bunu kullandığında akış thread'e ilişkinse o thread'in g\_x global değişkenini kullanıyor olurdu. Benzer biçimde bar fonksiyonu için de durum aynı olurdu. Thread'e özgü alanlara Windows dünyasında "Thread Local Storage (TLS)", UNIX/Linux dünyasında "Thread Specific Data (TSD)" denilmektedir. Thread'e özgü alanlar adreta thread'e özgü global değişken etkisi yaratmaktadır.

**Windows'ta TLS Kullanımı**

Windows'ta TLS kullanımı şöyledir:

1) Önce (tipik olarak thread'ler yaratılmadan önce) TlsAlloc API fonksiyonu ile bir slot oluşturulur ve o slotun indeksi alınır. İşletim sistemi her thread içinm ayrı bir TLS alanı oluşturmaktadır. Thread'lerin TLS alanları slotlardan oluşan bir dizi gibidir. Aynı index aynı değişkene karşılık gelmez. Her thread'in TLS'sinin ilgili indeksi aslında farklı bir global değişken gibi davranmaktadır.



TlsAlloc fonksiyonunun prototipi şöyledir:

DWORD WINAPI TlsAlloc(void);

Fonksiyon başarı durumunda TGLS slotunun numarasına, başarıszlık durumunda TLS\_OUT\_OF\_INDEXES değerine geri döner. Slot numarasının global bir değişkende saklanması uygun olur.

2) TLS slotuna değer yerleştirmek için TlsSetValue, değer almak içinm TlsGetValue API fonksiyonları kullanılır.

BOOL WINAPI TlsSetValue(

\_\_in DWORD *dwTlsIndex*,

\_\_in LPVOID *lpTlsValue*

);

LPVOID WINAPI TlsGetValue(

\_\_in DWORD *dwTlsIndex*

);

Slotlarda saklanan bilgi bir göstericidir. Aslında bir gösterici her şeyi tutabilmeye adaydır. Eğer biz birden fazla bilgi tutacaksak bunu bir yapı olarak oluştururuz, sonra malloc fonksiyonu ile heap'te tahsisat yaparız. Onun adresini slota yerleştiririz.

3) Kullanım bitince TlsAlloc ile elde edilen slot TlsFree API fonksiyonuyla serbest bırakılır:

BOOL WINAPI TlsFree(

\_\_in DWORD *dwTlsIndex*

);

Windows sistemleri için örnek bir TLS kullanımı şöyle olabilir:

#include<stdio.h>

#include<stdlib.h>

#include<string.h>

#include<Windows.h>

typedefstructtagPERSON {

char name[32];

int no;

} PERSON;

void ExitSys(LPCSTR lpszMsg, int status);

DWORD\_\_stdcall ThreadProc1(void \*param);

DWORD\_\_stdcall ThreadProc2(void \*param);

void Foo(void);

DWORD g\_tlsVarIndex;

int main(void)

{

DWORD dwThreadId1, dwThreadId2;

HANDLE hThread1, hThread2;

if ((g\_tlsVarIndex = TlsAlloc()) == TLS\_OUT\_OF\_INDEXES)

ExitSys("AlsAlloc", EXIT\_FAILURE);

if ((hThread1 = CreateThread(NULL, 0, ThreadProc1, NULL, 0, &dwThreadId1)) == NULL)

ExitSys("CreateThread", EXIT\_FAILURE);

if ((hThread2 = CreateThread(NULL, 0, ThreadProc2, NULL, 0, &dwThreadId2)) == NULL)

ExitSys("CreateThread", EXIT\_FAILURE);

WaitForSingleObject(hThread1, INFINITE);

WaitForSingleObject(hThread2, INFINITE);

return 0;

}

DWORD\_\_stdcall ThreadProc1(void \*param)

{

PERSON \*per;

if ((per = (PERSON \*)malloc(sizeof(PERSON))) == NULL) {

fprintf(stderr, "cannot allocate memory!..\n");

exit(EXIT\_FAILURE);

}

strcpy(per->name, "Kaan Aslan");

per->no = 123;

TlsSetValue(g\_tlsVarIndex, per);

Foo();

TlsFree(g\_tlsVarIndex);

return 0;

}

DWORD\_\_stdcall ThreadProc2(void \*param)

{

PERSON \*per;

if ((per = (PERSON \*)malloc(sizeof(PERSON))) == NULL) {

fprintf(stderr, "cannot allocate memory!..\n");

exit(EXIT\_FAILURE);

}

strcpy(per->name, "Ali Serce");

per->no = 674;

TlsSetValue(g\_tlsVarIndex, per);

Foo();

return 0;

}

void Foo(void)

{

PERSON \*per;

per = (PERSON \*)TlsGetValue(g\_tlsVarIndex);

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

}

void ExitSys(LPCSTRlpszMsg, intstatus)

{

DWORD dwLastError = GetLastError();

LPTSTR lpszErr;

if (FormatMessage(FORMAT\_MESSAGE\_ALLOCATE\_BUFFER | FORMAT\_MESSAGE\_FROM\_SYSTEM, NULL, dwLastError,

MAKELANGID(LANG\_NEUTRAL, SUBLANG\_DEFAULT), (LPTSTR)&lpszErr, 0, NULL)) {

fprintf(stderr, "%s: %s", lpszMsg, lpszErr);

LocalFree(lpszErr);

}

exit(status);

}

**UNIX/Linux Sistemlerinde TSD Kullanımı**

Thread'e özgü alanların UNIX/Linux sistemlerindeki kullanımı Windows sistemlerindeki kullanımına çok benzemektedir. İşlemler sırasıyla şu adımlardan geçilerek gerçekleştirilir:

1) Önce pthread\_key\_create fonksiyonu ile bir slot yaratılır. (Bu fonksiyon Windows'taki TlsAlloc fonksiyonuna benzemektedir.)

#include <pthread.h>

int pthread\_key\_create(pthread\_key\_t \*key, void (\*destructor)(void\*));

Fonksiyonun birinci parametresi slotun numarsının yerleştirileceği pthread\_key\_t türünden nesnenin adresini alır. İkinci parametre slot yok edilirken çağrılacak fonksiyonu belirtilir. Bu parametre NULL geçilebilir.

2) Slota değer pthread\_setspecific fonksiyonuyla yerleştirilip, pthread\_getspecific fonksiyonuyla geri alınabilir:

#include <pthread.h>

void \*pthread\_getspecific(pthread\_key\_t key);

int pthread\_setspecific(pthread\_key\_t key, const void \*value);

Burada da slota bir gösterici yerleştirilip geri alınmaktadır.

3) İşlem bitince slot pthread\_key\_delete fonksiyonuyla yok edilir:

#include <pthread.h>

int pthread\_key\_delete(pthread\_key\_t key);

Örnek bir kullanım şöyle olabilir:

#include<stdio.h>

#include<stdlib.h>

#include<string.h>

#include<unistd.h>

#include<pthread.h>

typedefstructtagPERSON {

char name[32];

int no;

} PERSON;

void \*thread\_proc1(void \*param);

void \*thread\_proc2(void \*param);

void foo(void);

pthread\_key\_t g\_slotKey;

int main(void)

{

pthread\_t tid1, tid2;

int result;

if (pthread\_key\_create(&g\_slotKey, NULL) != 0) {

fprintf(stderr, "pthread\_create: %s\n", strerror(result));

exit(EXIT\_FAILURE);

}

if ((result = pthread\_create(&tid1, NULL, thread\_proc1, NULL)) != 0) {

fprintf(stderr, "pthread\_create: %s\n", strerror(result));

exit(EXIT\_FAILURE);

}

if ((result = pthread\_create(&tid2, NULL, thread\_proc2, NULL)) != 0) {

fprintf(stderr, "pthread\_create: %s\n", strerror(result));

exit(EXIT\_FAILURE);

}

pthread\_join(tid1, NULL);

pthread\_join(tid2, NULL);

pthread\_key\_delete(g\_slotKey);

return 0;

}

void \*thread\_proc1(void \*param)

{

PERSON \*per;

if ((per = (PERSON \*)malloc(sizeof(PERSON))) == NULL) {

fprintf(stderr, "cannot allocate memory!..\n");

exit(EXIT\_FAILURE);

}

strcpy(per->name, "Kaan Aslan");

per->no = 123;

pthread\_setspecific(g\_slotKey, per);

foo();

returnNULL;

}

void \*thread\_proc2(void \*param)

{

PERSON \*per;

if ((per = (PERSON \*)malloc(sizeof(PERSON))) == NULL) {

fprintf(stderr, "cannot allocate memory!..\n");

exit(EXIT\_FAILURE);

}

strcpy(per->name, "Ali Serce");

per->no = 674;

pthread\_setspecific(g\_slotKey, per);

foo();

returnNULL;

}

void foo(void)

{

PERSON \*per;

per = (PERSON \*)pthread\_getspecific(g\_slotKey);

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

}

**Thread Havuzları (Thrad Pools)**

IO yoğun uygulamalarda (örneğin tipik olarak server soket uygulamalarında) çok sayıda kısa ömürlü thread'in yaratılması gerekebilmektedir. Örneğin server programa bir client'tan istek gelir. Server bir thread açarak o thread'in bunu yapmasını sağlar. Çünkü o sırada server başka client'ların isteklerini de bekletmek istemez. Thread'lerin yaratılma ve yok edilme zamanları göreli olarak maliyetlidir. İşte bu tür uygulamalarda thread'lerin yaratılması ve yok edilmesini hızlandırmak için thread havuzları düşünülmüştür.

Thread havuzları aslında thread mekanizmasını kullanan bir organizasyondur. Thread havuzlarında belli bir miktar thread yaratılmış ama çalışmıyor biçimde (suspend biçimde) bekletilir. Programcı bir thread istediği zaman havuzda zaten yaratılmış bir thread ona verilir. Programcının thread ile işi bittiğinde thread yok edilmez, yeniden havuzda bekletilir. Tabi havuz yetersiz kalırsa ihtiyaca göre büyütülebilmektedir. Havuzda çok fazla sayıda thread atıl biçimde bekliyorsa yine havuz dinamik olarak küçültülebilmektedir.

Thread huvuzları Windows sistemlerinde API fonksiyonları tarafından desteklenmektedir. Yani thread havuzundan thread alan vs. işlemleri yapan API fonksiyonları vardır. Tabi bu API fonksiyonları aslında taban fonksiyonlar değildir. Bnlar thread API fonksiyonlarını kendi içerisinde çağırmaktadır. UNIX/Linux sistemlerinde thread havuzu işlemlerini yapan standart POSIX fonksiyonları yoktur. Dolayısıyla thread havuz organizasyonunu programcıların yapması beklenmektedir. Qt gibi, Java ve .NET, Mono gibi framework'lerde thread havuzu işlemlerini yapan sınıflar bulunmaktadır. C'de UNIX/Linux sistemlerinde thread havuzu işlemlerini yapan başkaları tarafından yazılmış kütüphaneler de bulunmaktadır (Google'dan "thread pool implelemantation in C" aramasını yapnızı.)

Thread havuzları yukarıda da belirtildiği gibi kısa ömürlü çok sayıda thread'in gereksinim duyulduğu durumlarda kullanılmalıdır. Az sayıda uzun süre çalışan thread'ler için havuz organizasyonunun bir faydası yoktur.

**Thread Öncelikleri ve Thread'lerin Çizelgelenmesi (Thread Scheduling)**

Thread çizelgelemesinin nasıl yapıldığı işletim sisteminden işletim sistemine hatta aynı işletim sisteminde versiyondan versiyona değişebilmektedir. Bu konun ayrıntıları "Windows Sistem Programlama" ve "UNIX/Linux Sistem Programlama" kurslarında ele alınmaktadır. Burada bir fikir oluşsun diye Windows'taki thread çizelgelemesinin dayandığı fikir açıklanacaktır.

Windows'ta uygulanan thread çizelgeleme algoritmasına "öncelik sınıfları ile döngüsel çizelgeleme (priority class based round robin scheduling" denilmektedir.

Windows'ta her thread'in [0-31] arasında bir öncelik derecesi vardır. Sistemde aynı öncelik derecesine sahip thread'ler bir sınıf oluşturur. Örneğin sistemde şu önceliklere sahip thread'ler bulunuyor olsun:

T1: 8

T2: 8

T3: 8

T4: 12

T5: 12

T6: 18

Burada önce yalnızca T6 thread2i tek başına çalıştırılır. O thread bloke olduğunda ya da çalışması bittiğinde T4 ve T5 döngüsel çizelgelenir. Bunlar da bloke olduğunda ya da bunların da çalışması bittiğinde bu sefer T1, T2 ve T3 thread2leri çalıştırılır.

Windows'un uyguladığı bu çizelgeleme algoritmasında en dikkat çekici nokta yüskek öncelikli thread'lerin CPU'yu tekeline alabilme durumudur. Çok çekirdekli ya da çok işlemcili sistemlerde diğer çekirdekler ya da işlemciler sıradaki düşük öncelikli thread'lere atanmaktadır. Bu nedenle bu durumun oluşma olasılığı biraz daha azaltılmış olmaktadır. Windows sistemlerinde en normal olan durum IO yoğun thread'lerin önceliğini yükseltmektir. IO yoğun thread'ler çok büyük zamanlarını uykuda geçirirler. Bir IO olayı gerçekleştiğinde uyanarak ona yanıt verirler. Bazı uygulamalarda hızlı yanıt için bu tür thread'lerin önceliklerinin yükseltilmesi tercih edilebilmektedir. Yüksek bir thread uykudayken IO olayı gerçekleşirse hemen düşük öncelikli thread'in çalışmasına ara verilir ve o thread CPU'ya atanır. Tabi CPU yoğun bir thread'in önceliğini yükseltirsek gerçekten bu thread CPU'yu tekeline alabilir.

Pekiyi Windows'ta bir thread'in önceliği nasıl değiştirilmektedir? 8 öncelik normal bir önceliktir. Bizim çalıştırdığımız programlardaki thread'lerin önceliği 8'dir. Thread'in önceliği iki sayının toplamı biçiminde oluşturulur:

1) Prosesin öncelik sınıfı

2) Thread'in göreli önceliği

Proseisn öncelik sınıfı bir taban değer oluşturur. Bu taban değere thread'in göreli önceliği (+ ya da - olarak) toplanır. Prosesin öncelik sınıfı üst prosesten alt prosese aktarılmaktadır. Prosesin öncelik sınıfı GetPriorityClass fonksiyonuyla alınıp SetPriorityClass fonksiyonuyla set edilebilmektedir. Thread'in göreli önceliği ise GetThreadPriority fonksiyonuyla alınıp SetThreadPriority fonksiyonuyla set edilebilir. MSDN yardım sisteminde hangi thread öncelikleri için prosesin öncelik sınıfının ve thread'in göreli önceliğinin nasıl ayarlanacağı belirtilmiştir.

Linux sistemlerindeki çizelgeleme algoritmaları POSIX standartlarının belirlediği birtakım kurallara uymakla birlikte kendine özgü niteliklere de sahiptir.

**Algoritmalar ve Veri Yapıları (Algorithms and Data Structures)**

BU bölümde temel algoritma analizi ve veri yapıları konusu ele alınacaktır.

**Algoritma Nedir?**

Bir problemi çözüme götüren adımlar topluluğuna algoritma (algorithm) denilmektedir. Sözcük etimolojik bakımdan cebirin kurucusu El-Harizmi'nin isminden türetilmiştir. Algoritma problemi kesin çözüme götürür. Ancak bazı problemlerin kesin çözümü çok uzun zaman alabilmektedir. Bunlar için kısa sürede iyi bir çözüm ile yetinilebilir. İşte problemin kesin çözümünü bulmayan ama iyi bir çözüm vaat eden adımlar topluluğuna "sezgisel yöntemler (heuristic)" denilmektedir.

Algoritmalar nasıl ifade edilirler? Bunun için çok formel bir yöntem yoktur. Bazen akış diyagramları, sahte kodlar (pseudo codes) kullanılabilir. Hemen her zaman sözel bir anlatım buna eşlik eder. Algoritma anlatmak için özel diller tasarlanmış olsa da bunlar yaygın bir kullanım bulamamıştır. Algoritmayı anlatmak için en çok kullanılan yöntemlerden biri popüler dille algoritmayı gerçekleştirmektir. Eskiden bu amaçla C yaygın kullanılırdı. Şimdi C'nin yanı sıra C++, Java, C# gibi diller de bu amaçla kullanılmaktadır.

Pekiyi algortma dilden bağımsız mıdır? Algoritmaların çoğu dilden bağımsız olarak ifade edilebilir. Ancak ince noktalara gelindiğinde algoritmalar da dile bağımlı hale gelebilmektedir. Örneğin bir dilde belli işi daha kolay yapan deyimler olabilir. Bu durumda algoritma o deyimlerle gerçekleştirilebilir. Ya da örneğin fonksiyonel dillerde algoritmalar özyineleme içerecek biçimde tasarlanmaktadır.

Bazı problemlerin çözümü için tek bir yol bulunuyor olabilir. Fakat bazıları için birden fazla algoritma söz konusu olabilir. Bu durumda bu algoritmaları kıyaslamak gerekir. Kıyaslamak için ise bir ölçüt gerekir. Algoritmaları kıyaslama için iki önemli ölçüt vardır: Hız ve kaynak kullanımı. Ancak baskın ölçüt hızdır. Bu nedenle algoritmaları karşılaştırmak denildiğinde default olarak hız ölçütüne göre karşılaştırma anlaşılmalıdır. Algoritmaların masaya yatırılıp karşılaştırılması sürecine "algoritma analizi (analyisis of algorithms)" denilmektedir.

**Algoritma Analizi**

İki algoritmanın hızı nasıl ölçülebilir? İlk akla gelen yöntem simülasyon yöntemidir. Bu yöntemde algoritmalar çalıştırılıp zaman ölçülür. Tabi algoritmaların çoğu birtakım girdiler üzerinde işlem yapmaktadır (örneğin bir dizi üzerinde). Bu durumda o girdilerin dağılımı da önemlidir. Eğer biz simülasyonu bir kez yaparsak yanlış sonuçlar elde edebiliriz. Rastgele girdiler üzerinde çok sayıda denemeinin bir ortalaması ancak bir fikir verebilir.

Simülasyon yöntemi matematiksel bir yöntem değildir. Bu nedenle bize önemli bir bilgi vermez. Algoritmaları cebirsel olarak ölçmek daha sağlam bir zemin oluşturabilmektedir. Algoritmaları analiz ederken kullanılabilecek sağlam yöntemlerden biri algoritmadaki çözüm için toplamda kaç işlemin gerektiğine bakmaktır. (Tabi tüm işlemlerin makina zamanları aynı değildir fakat bu durum ihmal edilebilir.) Örneğin iki sort algoritmasını karşılaştıracak olalım. İkisinde de sort işlemi için kaç işlem gerektiğini hesaplamaya çalışabiliriz. Tabi algoritmaların girdi parametreleri vardır. Sort işleminde dizinin uzunluğu olan N bir girdi parametresidir. Algoritma için gereken işlemlerin sayısı bu N ile ilgili olacaktır.

Algoritmadaki işlemlerin sayısı bulunabilir mi? Ya algoritmada karışık if deyimleri varsa ne olacaktır? Biz akışın nasıl olacağını giridiyi bilmeden anlayamayız. Pekiyi bu durumda işlemlerin sayısı nasıl hesaplanacaktır. Örneğin aşağıdaki en büyük sayıyı bulma algorimasına bakalım:

int a[N] = {...};

int max, i;

max = a[0];

for (i = 1; i < N; +i)

if (max < a[i])

max = a[i];

Burada toplam işlemin sayısı max = a[i] işlemine bağlıdır. Pekiyi bu işlemden kaç tane olacağını diziyi görmeden bilebilir miyiz? Yanit tabii ki hayır. İşte algoritma analizinde işlemlerin sayısını hesaplarken üç durum dikkate alınabilmektedir:

1) Ortalama durum (average case condition)

2) En kötü durum (worst case condition)

3) En iyi durum (best case conditin)

En kötü durum olabileceklerin en kötüsüdür. Yukarıdaki örnekte en kötü durumda max = a[i] işlemi N - 1 kez yapılır. Ortalama durum tüm olasılıkların ortalamasını temsil eder. Yukarıdaki örnekte ortalama olarak max = a[i] işlemi N - 1 / 2 kez yapılmaktadır. En iyi durum da olabileceklerin en iyisini temsil eder. Yukarı örnekte en iyi durumda max = a[i] işlemi hiç yapılmaz.

Tabi analizde en iyi durumun çok yararı yoktur. Çünkü aşırı iyimserlik programlamada çok değerli değildir. En kötü durum senaryosu önemlidir. Çünkü bazı durumlarda en kötü duruma bile hazırlıklı olmak gerekebilir. Şüphesiz en önemli durum ortalama durumdur. Bu algoritmanın karakterini en iyi yansıtan durumdur. O halde bu üç durumun önemi şöyle sıralanabilir: Ortalama durum, en kötü durum, en iyi durum. Genellikle algoritmalar ele alınırken hem ortalama durum hem de en kötü durum analizi yapılır.

Algoritmaların işlem sayısı analizi de sanıldığı kadar kolay değildir. Çünkü özellikle ortalama durumu hesaplamak bazen çok karmaşık olabilmektedir. Algoritmaları pratik bakımdan kıyaslamak için asimtotik notasyonlardan faydalanılmaktadır. Bunların en yaygın kullanılanı Big O notasyonudur.

**Big O Notasyonu**

Big O notasyonu algoritmaları karmaşıklıklarına göre kategorilere ayırır. Kategoriler arasında da iyilik kötülük ilişkisi vardır. Big O notasyonu kaba bir kıyaslama biçimidir. Eğer algoritmalar ayrıntılı bir biçimde kıyaslanacaksa yukarıda belirtilen işlem sayıları hesaplanmalıdır. Big O notasyonunda belli karakterdeki algoritmalar aynı kategoridedir. Adresta bunlar arasında fark yokmuş gibi davranılır. Eğer algoritma birden fazla kategoriye giriyorsa en kötü kategori onun gerçek kategorisini belirtir. İyiden kötüye doğru Algoritmik Big O katregorileri şöyledir:

**O(1):** Bu kategoriye "sabit karmaşıklık" da denilmektedir. Eğer algoritmada hiç döngü yoksa her şey tekil işlemlerle yapılıyorsa algoritma O(1) karmaşıklıktadır. (Örneğin üçgenin alanının bulunması gibi). Tekil işlemlerin sayısı önemli değildir.

**O(log (N)):** Bu karmaşıklıkta N uzunluğunda bir girdi kümesi (örneğin bir dizi) vardır. Algoritmada açık ya da özyinelemeli olarak bir döngü bulunur. Ancak bu döngü N kadar değil log N kadar dönmektedir. (log default olarak 2 tabanına ilişkin düşünülmelidir.). Örneğin "ikili arama (binary search)" algoritması O(log N) karmaşıklıktadır. (Tabi yukarıda da belirtildiği gibi başka tekil işlemler de algoritmada bulunabilir.) Bu karmaşıklık kategorisine "logaritmik karmaşıklık" da denilmektedir.

**O(N):** Bu kategoriye "doğrusal karmaşıklık" da denilmektedir. Bu tür algoritmalarda iç içe olmayan tekil birden fazla döngü bulunabilir. Girdi uzunluğu N ise bu döngüler N ile bağlantılı olarak (tipik olarak N defa ya da N / 2 defa ya da N - 5 defa vs.) dönerler. Fakat dönüş logaritmik değildir. N ile oransal olabilir. Örneğin en büyük sayıyı bulma algoritması, doğrusal arama işlemi doğrusal karmaşıklıktadır.

**O(N log N):** Bu tür algoritmalarda iç içe iki döngü vardır. Ancak döngülerden biri N ile orantılı biçimde dönerken diğeri log N kadar döner. Örneğin "quick sort" algoritması O(N log N) karmaşıklıktadır.

**O(N2):** Bu kategoriye "karesel karmaşıklık" da denilmektedir. Bu tür algoritmalarda iç içe iki döngü vardır (tabi başka tekil döngüler de olabilir.) Döngülerin ikisi de N ile orantılı biçimde dönerler. Örneğin "boubble sort", "selection sort" algoritmaları karesel karmaşıklıktadır.

**O(N3):** Bu kategoriye "küpsel karmaşıklık" da denilmektedir. Burada iç içe N ile orantılı 3 döngü vardır. Örneğin matris çarpımı böyle bir döngüsel yapı gerektirmektedir.

**O(NK):** Bu kategoriye K'sal karmaşıklık denir. İç içe k taneN ile orantılı döngü bulunmaktadır.

Yukarıdaki tüm kategorilere "polinomsal karmaşıklık" da denilmektedir. Polinomsal karmaşıklıktaki problemlerin bugünkü bilgisayarlarla makul zamanda çzöülmesi mümkündür.

**O(KN):** Bu kategoriye üstel karmaşıklık (exponential complexity) de denilmektedir. Bu kategorideki algoritmalardaki işlem sayısı N'e göre çok hızlı artmaktadır. Bu nedenle bugünkü bilgisayarlarla bile bunların kesin çözümleri çok fazla zaman alabilir. Bu nedenle bunların bilgisayar çözümleri mümkün olmayabilir. Bu tür algoritmalarda sezgisel yöntemler (heuristic) önem kazanmaktadır. Örneğin N elemanlı bir kümenin alt kümelerinin sayısı 2N tanedir. Bir kümenin tüm alt kümelerine bakarak işlem yapan algoritöala (böyle çok algoritma vardır) üstel karmaşıklıktadır.

**O(N!):** Bu karmaşıklığa faktöryel karmaşıklığı denilmektedir. Bu en kötü karmaşıklık grubudur. Örneğin gezgin satıcı probleminde satıcı bir merkezden (düğümden) çıkarak tüm şehirleri (düğümleri) dolaşıp tekrar başlangıç yerine geri gelir. Ama. en kısa turu atacak çözümün bulunmasıdır. Bu problemde N tane düğüm varsa (N - 1)! / 2 tane rotayı hesaplamak gerekir. O halde gezgin satıcı problemi faktöryel karmaşıklıkta bir algoritmadır.

Üstel ve faktöryel karmaşıklıklara "polinomsal olmayan (Non-Polynomial ya da kısaca NP)" karmaşıklıkta problem denilmektedir. İşte bu tür problemler hala üzerinde en çok çalışılan ve özel yöntem bulunmaya çalışılan problemlerdir.

**Veri Yapısı (Data Structures) Nedir?**

Aralarında mantıksal ya da fiziksel ilişki bulunan nesnelerin oluşturduğu toluluğa "veri yapısı (data structure)" denilmektedir. Bazı veri yapıları dilin sentaksı tarafından zaten doğrudan desteklenmektedir. Bazılarını ise biz mevcut olan dil olanaklarıyla fonksiyonlar ya da sınıflar biçiminde oluştururuz. C'de derleyici tarafından desteklenen (built-in) veri yapıları şunlardır:

- Diziler (arrays)

- Yapılar (structures)

- Birlikler (unions)

Bazı dillerin standart kütüphanelerinde diğer veri yapıları bulunabilmektedir. Özellikle nesne yönelimli programlama dilelrinin kütüphanelerinde pek çok veri yapısı sınıfsal biçimde gerçekleştirilmiştir. Örneğin Java'nın, .NET'in, C++'ın standart kütüphanelerinde pek çok veri yapısı hazır biçimde zaten bulunmaktadır. O ortamlarda çalışan programcıların bunları yeniden yazması gerekmez. Bazı framework'ler de yine çeşitli veri yapılarını barındırmaktadır. (Örneğin Qt'de MFC'de Cocoa'da temel veri yapıları yine sınıflar biçiminde bulunmaktadır.) Ancak C'nin standart kütüphanesinde genel olarak veri yapılarına ilişkin fonksiyonlar bulunmamaktadır.

Veri yapıları konusunda "Soyut Veri Türü (Abstract Data Type (ADT)) isimli bir kavram da çok sık kullanılmaktadır. Soyut Veri Türü denildiğinde bir veri yapısı üzerinde işlem yapan fonksiyonlar anlaşılmaktadır. Soyut veri yapısında veri yapısını idare için gereken birtakım veriler ve onları yöneten fonksiyonlar bulunur. Aslında soyut veri yapısı adeta veri yapısını oluşturan sınıf gibi düşünülebilir. Örneğin "Stack ADT" başlığını gördüğümüzde stack veri yapısını organize eden ve bunun üzerinde işlemler yapan fonksiyonlardan oluşan kodlar aklımıza gelir. Soyut Veri Yapısı özellikle nesne yönelimli teknikle birlikte kullanılmaya başlanmış bir terimdir.

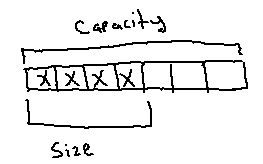
**Temel Veri Yapıları**

Dinamik diziler, kuyruk sistemler, stack sistemleri, bağlı listeler ve hash tabloları temel veri yapılarıdır. Bu bölümde bunların nasıl oluşturulacağı ve algoritmik özellikleri üzerinde durulacaktır.

**Dinamik Diziler (Dynamic Arrays)**

Pek çok uygulamada gereksinim duyulan dizinin uzunluğu zamanla değişebilmektedir. Örneğin bir çizm programında çizilen çekildeki noktaların bri dizide tutulduğunu düşünelim. Başlangıçta bu dizinin hangi uzunlukta olacağını bilemeyiz. Bu dizi duruma göre büyüyebilir. Ya da bir dizin içerisindeki dosyaları bir diziye yerleştirmek isteyelim. Dizindeki dosyaları bulurken önceden bunların kaç tane olduğunu bilemeyiz. İşte bu tür durumlarda bir dizi oluşturarak duruma göre bunu büyütmek gerekir. Bu işlemi yapan veri yapısı ve fonksiyonlara "Dinamik Dizi" dizi denilmektedir.

Dinamik dizi gerçekleştiriminde dizi için önce küçük bir alan tahsis edilir. Bu alan dolduğunda yeniden tahsisat yapılarak daha büyük bir alana geçilecektir. Tabi eski alandaki bilgiler yeni alana kopyalanacak ve eski alan da serbest bırakılacaktır. (C'de bu işlem zaten realloc fonksiyonuyla yapılmaktadır.) Geleneksel olarak dizi için tahsis edilmiş olan toplam alana Capacity, dolu olan eleman sayısına ise Size ya da Count denilmektedir.



Dinamik diziye eleman eklenirken eleman Size ile belirtilen indekse eklenir, Size bir artırılır. Size değeri Capacity değerine geldiğinde Capacity değeri artırılır. Pekiyi Capacity değeri ne kadar artırılmalıdır? Capacity değeri eskisinin katı olarak artırılırsa (yani geometrik olarak artırılırsa) sona eleman ekleme işlemi sabit karmaşıklıktga olur. Eğer Capacity değeri eskisinden N fazla olacak biçimde artırılırsa bu durumda sona eleman ekleme işlemi doğrusal karmaşıklığa yaklaşır. Geometrik artırımdaki sona eleman ekleme karmaşıklığıma "ek maliyetli sabit zamanlı karmaşıklık (amortized constant time complexity)" denilmektedir. "Ek maliyetli sabit zamanlı karmaşıklık" demek, işlemin çoğu kez sabit karmaşıklıkta ancak logaritmik aralıklarla doğrusal karmaşıklıkta yapılması demektir. Tipik olarak dinamik dizilerde Capacity artırımı eskisinin iki katı olacak biçimde yapılır.

Dinamik dizilerde elemana erişim normal dizilerde olduğu gibi sabit karmaşıklıkta (yani O(1) karmaşıklıkta) yapılmaktadır. Araya eleman ekleme ve aradan eleman silme doğrusal karmaşıklıkta yapılmaktadır.

Dinamik dizileri handle sistemi kullanrak aşağıdaki gibi oluşturabiliriz:

/\* DynamicArray.h \*/

#ifndef DYNAMICARRAY\_H\_

#defineDYNAMICARRAY\_H\_

/\* Symbolic Constants \*/

#defineDEF\_CAPACITY 2

/\* Type declarations \*/

typedefintDATATYPE;

typedefstructtagDARRAY {

DATATYPE \*pArray;

int capacity;

int size;

} DARRAY, \*HDARRAY;

/\* Function Prototypes \*/

HDARRAY CreateDArray(void);

HDARRAY CreateDArrayWithSize(int capacity);

int SetCapacity(HDARRAY hDArray, int newCapacity);

int AddItem(HDARRAY hDArray, DATATYPE val);

int InsertItem(HDARRAY hDArray, int index, DATATYPE val);

int DeleteItem(HDARRAY hDArray, int index);

void TrimToSize(HDARRAY hDArray);

/\* Macros \*/

#defineGetSize(hDArray) ((hDArray)->size)

#defineGetCapacity(hDArray) ((hDArray)->capacity)

#defineGetItem(hDArray, index) ((hDArray)->pArray[index])

#endif

/\* DynamicArray.c \*/

#include<stdio.h>

#include<stdlib.h>

#include<string.h>

#include"DynamicArray.h"

/\* Function Definitions \*/

HDARRAY CreateDArrayWithSize(intcapacity)

{

HDARRAY hDArray;

if ((hDArray = (HDARRAY)malloc(sizeof(DARRAY))) == NULL)

returnNULL;

if ((hDArray->pArray = (DATATYPE \*)malloc(sizeof(DATATYPE) \* capacity)) == NULL) {

free(hDArray);

returnNULL;

}

hDArray->size = 0;

hDArray->capacity = capacity;

return hDArray;

}

HDARRAY CreateDArray(void)

{

return CreateDArrayWithSize(DEF\_CAPACITY);

}

int SetCapacity(HDARRAYhDArray, intnewCapacity)

{

DATATYPE \*pNewArray;

if (newCapacity<hDArray->size)

return -1;

if (newCapacity == hDArray->capacity)

returnnewCapacity;

if ((pNewArray = (DATATYPE \*)realloc(hDArray->pArray, newCapacity \* sizeof(DATATYPE))) == NULL)

return -1;

hDArray->pArray = pNewArray;

hDArray->capacity = newCapacity;

returnnewCapacity;

}

int AddItem(HDARRAYhDArray, DATATYPEval)

{

if (hDArray->size == hDArray->capacity && SetCapacity(hDArray, hDArray->capacity \* 2) == -1)

return -1;

hDArray->pArray[hDArray->size++] = val;

returnhDArray->size - 1;

}

int InsertItem(HDARRAYhDArray, intindex, DATATYPEval)

{

if (index>hDArray->size)

return -1;

if (hDArray->size == hDArray->capacity && SetCapacity(hDArray, hDArray->capacity \* 2) == -1)

return -1;

memmove(&hDArray->pArray[index + 1], &hDArray->pArray[index], (hDArray->size - index) \* sizeof(DATATYPE));

hDArray->pArray[index] = val;

++hDArray->size;

returnindex;

}

int DeleteItem(HDARRAYhDArray, intindex)

{

if (index>= hDArray->size)

return -1;

memmove(&hDArray->pArray[index], &hDArray->pArray[index + 1], (hDArray->size - index - 1) \* sizeof(DATATYPE));

--hDArray->size;

returnindex;

}

void TrimToSize(HDARRAYhDArray)

{

SetCapacity(hDArray, hDArray->size);

}

/\* Test.c \*/

#include<stdio.h>

#include<stdlib.h>

#include"DynamicArray.h"

int main(void)

{

HDARRAY hDArray;

int i;

if ((hDArray = CreateDArray()) == NULL) {

fprintf(stderr, "cannot allocate memory!..\n");

exit(EXIT\_FAILURE);

}

for (i = 0; i < 10; ++i)

AddItem(hDArray, i);

for (i = 0; i <GetSize(hDArray); ++i)

printf("%d ", GetItem(hDArray, i));

printf("\n");

DeleteItem(hDArray, 10);

for (i = 0; i <GetSize(hDArray); ++i)

printf("%d ", GetItem(hDArray, i));

printf("\n");

TrimToSize(hDArray);

printf("Size = %d, Capacity = %d\n", GetSize(hDArray), GetCapacity(hDArray));

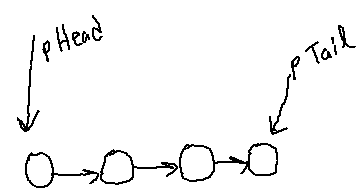
return 0;

}

**Bağlı Listeler**

Öncelik sonralık ilişkisinin olduğu veri yapılarına liste (list) denir. Bu tanıma göre diziler de birer listedir. Önceki elemanın sonraki elemanı gösterdiği listelere bağlı listeler (linked lists) denilmektedir. Bağlı listeler elemanların ardışıl olma zorunluluğunun kaldırıldığı diziler gibidir. Başka bir deyişle elemanları aynı türden olan fakat bellekte ardışıl biçimde bulunmak zorunda olmayan dizilere bağlı liste denilmektedir.

Bağlı listelerdeki her elemana düğüm (node) denilmektedir. Bağlı listenin ilk düğümünün yeri tutulur. Çoğu kez son düğümünün yeri de tutulmaktadır. Böylece bağlı listenin elemanlarına bir döngü ile erişilebilmektedir. Örneğin:



Bağlı listelerin elemanları (düğümleri) C'de bir yapıyla temsil edilir. Örneğin:

struct NODE {

DATATYPE val;

struct NODE \*pNext;

};

Eğer bağlı listelerde sonraki eleman da aynı zamanda önceki elemanı gösteriyorsa bu tür bağlı listelere çift bağlı listeler (doubly linked lists), eğer yalnızca önceki eleman sonraki elemanı gösteriyorsa böyle listelere tek bağlı listeler (single linked lists) denilmektedir. Çift bağlı listelerin bir düğümü de aşağıdaki gibi bir yapıyla temsil edilebilir:

struct NODE {

DATATYPE val;

struct NODE \*pPrev;

struct NODE \*pNext;

};

Bağlı listelerde belirli indeksteki elemana erişmek doğrusal karmaşıklıktadır, çünkü bir döngü gerekir. Fakat bağlı listelerde bir düğümün yerini biliyorsak oraya eleman insert etmek ve oradan eleman silmek sabit karmaşıklıktadır. Bağlı listelerin başına ve sonuna eleman eklemek ise yine sabit karmaşıklıkta yani çok hızlı yapılabilmektedir.

**Bağlı Listelere Neden Gereksinim Duyulmaktadır?**

1) Bağlı listelerde elemanların bellekte ardışıl bulunması gerekmez. Böylece ardışıl bellek sorununun olduğu durumlarda bağlı listeler tercih edilebilir. Ardışıllık bölünmeye (fragmentation) yol açar. Bölünme de bellek verimini düşürme eğilimindedir. Özellikle çok sayıda aynı bölgeyi (örneğin hepa'i) kullanan dinamik dizinin söz konusu olduğu durumlarda bağlı listeler belleğin daha verimli kullanılmasını sağlayabilir.

2) Çok sayıda insert delete işleminin yapıldığı durumlarda bağlı listeler tercih edilebilir. Örneğin bir işletim sisteminde prosesler için proses kontrol blokları oluşturulur. Bunlar bağlı liste halinde birbirlerine bağlanırlar. Proses bittiğinde bunlar yok edilirler. Bu durumda bağlı listeler dinamik dizilere göre çok daha etkin kullanıma yol açmaktadır.

**Bağlı Listelerle (Dinamik) Dizilerin Karşılaştırılması**

1) Belli bir indeksteki elemana erişim dizilerde sabit zamanlıdır fakat bağlı listelerde doğrusal karmaşıklıktadır.

2) Düğümü bilinen bir elemanın önüne insert işlemi balı listelerde sabit zamanlıdır ancak dizilerde doğrusal karmaşıklıktadır.

3) Düğümü bilinen bir elemanın silinmesi bağlı listelerde sabit karmaşıklıktadır ancak dizilerde doğrusal karmaşıklıktadır. (Tabi tek bağlı listelerde silinecek düğümün değil ondan önceki düğümün adresi bilinmelidir.)

4) Başa eleman ekleme dizilerde doğrusal karmaşıklıktadır ancak bağlı listelerde sabit karmaşıklıktadır. Sona eleman ekleme her iki veri yapısında da sabit karmaşıklıktadır.

5) Dizilerin ardışıl alana gereksinim duymaları bir dezavantajdır. Bağlı listeler ardışıl alana gereksinim duymazlar.

6) Bağlı listelerin bellekte toplam kapladığı alan dizilerden fazladır. (Ancak bölünme bundan çok daha büyük belleğin kullnım dışı kalmasına yol açan bir etkendir.)

**Bağlı Listelerin Gerçekleştirilmesi**

Uygulamada çift bağlı listeler tek bağlı listelerden çok daha fazla kullanılmaktadır. Çünkü adresi bilinen bir düğümün silinmesi uygulamalarda çok gereksinim duyulan bir durumdur. Bu nedenle biz burada çift bağlı liste gerçekleştirimi üzerinde duracağız.

Çift bağlı listelerde her düğüm aşağıdaki gibi bir yapıyla teslim edilebilir:

typedefstructtagNODE {

DATATYPE val;

structtagNODE \*pPrev;

structtagNODE \*pNext;

} NODE;

Bağlı listeyi kontrol eden handle alanı şöyle olabilir:

typedefstructtagLLIST {

NODE \*pHead;

NODE \*pTail;

size\_t count;

} LLIST, \*HLLIST;

Çift bağlı listenin örnek gerçekleştirimi şöyle olabilir:

/\* DoublyLinkedList.h \*/

#ifndef DOUBLYLINKEDLIST\_H\_

#defineDOUBLYLINKEDLIST\_H\_

#include<stddef.h>

/\* Symbolic Constants \*/

#defineFALSE 0

#defineTRUE 1

/\* Type declarations \*/

typedefintDATATYPE;

typedefintBOOL;

typedefstructtagNODE {

DATATYPE val;

structtagNODE \*pPrev;

structtagNODE \*pNext;

} NODE;

typedefstructtagLLIST {

NODE \*pHead;

NODE \*pTail;

size\_t count;

} LLIST, \*HLLIST;

/\* Function Prototypes \*/

HLLIST CreateLList(void);

NODE \*AddItemTail(HLLIST hLList, DATATYPE val);

NODE \*AddItemHead(HLLIST hLList, DATATYPE val);

NODE \*InsertItemAfterNode(HLLIST hLList, NODE \*pNode, DATATYPE val);

NODE \*InsertItemIndex(HLLIST hLList, size\_t index, DATATYPE val);

void DeleteItemNode(HLLIST hLList, NODE \*pDelNode);

BOOL DeleteItemIndex(HLLIST hLList, size\_t index);

NODE \*FindItem(HLLIST hLList, DATATYPE val);

BOOL WalkLList(HLLIST hLList, BOOL(\*Proc)(DATATYPE \*val));

BOOL WalkLListRev(HLLIST hLList, BOOL(\*Proc)(DATATYPE \*val));

void ClearLList(HLLISThLList);

void CloseLList(HLLIST hLList);

#endif

/\* DoublyLinkedList.c \*/

#include<stdio.h>

#include<stdlib.h>

#include"DoublyLinkedList.h"

/\* static Function Prototypes \*/

staticNODE \*createNode(DATATYPE val);

/\* Function Definitions \*/

HLLIST CreateLList(void)

{

HLLIST hLList;

if ((hLList = (HLLIST)malloc(sizeof(LLIST))) == NULL)

returnNULL;

hLList->pHead = NULL;

hLList->pTail = NULL;

hLList->count = 0;

return hLList;

}

NODE \*AddItemTail(HLLISThLList, DATATYPEval)

{

NODE \*pNewNode;

if ((pNewNode = createNode(val)) == NULL)

returnNULL;

pNewNode->pNext = NULL;

if (hLList->pHead == NULL) {

hLList->pHead = pNewNode;

pNewNode->pPrev = NULL;

}

else {

hLList->pTail->pNext = pNewNode;

pNewNode->pPrev = hLList->pTail;

}

hLList->pTail = pNewNode;

++hLList->count;

return pNewNode;

}

NODE \*AddItemHead(HLLISThLList, DATATYPEval)

{

NODE \*pNewNode;

if ((pNewNode = createNode(val)) == NULL)

returnNULL;

pNewNode->pPrev = NULL;

if (hLList->pHead == NULL)

hLList->pTail = pNewNode;

else

hLList->pHead->pPrev = pNewNode;

pNewNode->pNext = hLList->pHead;

hLList->pHead = pNewNode;

++hLList->count;

return pNewNode;

}

NODE \*InsertItemAfterNode(HLLISThLList, NODE \*pNode, DATATYPEval)

{

NODE \*pNewNode;

if ((pNewNode = createNode(val)) == NULL)

returnNULL;

pNewNode->pPrev = pNode;

if (pNode->pNext != NULL)

pNode->pNext->pPrev = pNewNode;

else

hLList->pTail = pNewNode;

pNewNode->pNext = pNode->pNext;

pNode->pNext = pNewNode;

++hLList->count;

return pNewNode;

}

NODE \*InsertItemIndex(HLLISThLList, size\_tindex, DATATYPEval)

{

size\_t i;

NODE \*pNode;

if (index>hLList->count)

returnNULL;

if (index == 0)

return AddItemHead(hLList, val);

pNode = hLList->pHead;

for (i = 0; i <index - 1; ++i)

pNode = pNode->pNext;

return InsertItemAfterNode(hLList, pNode, val);

}

void DeleteItemNode(HLLISThLList, NODE \*pDelNode)

{

if (hLList->pHead != pDelNode)

pDelNode->pPrev->pNext = pDelNode->pNext;

else

hLList->pHead = pDelNode->pNext;

if (hLList->pTail != pDelNode)

pDelNode->pNext->pPrev = pDelNode->pPrev;

else

hLList->pTail = pDelNode->pPrev;

--hLList->count;

free(pDelNode);

}

BOOL DeleteItemIndex(HLLISThLList, size\_tindex)

{

NODE \*pNode;

size\_t i;

if (index>= hLList->count)

returnFALSE;

pNode = hLList->pHead;

for (i = 0; i <index; ++i)

pNode = pNode->pNext;

DeleteItemNode(hLList, pNode);

returnTRUE;

}

NODE \*FindItem(HLLISThLList, DATATYPEval)

{

NODE \*pNode;

pNode = hLList->pHead;

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

if (pNode->val == val)

return pNode;

returnNULL;

}

BOOL WalkLList(HLLISThLList, BOOL(\*Proc)(DATATYPE \*val))

{

NODE \*pNode;

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

if (!Proc(&pNode->val))

returnFALSE;

returnTRUE;

}

BOOL WalkLListRev(HLLISThLList, BOOL(\*Proc)(DATATYPE \*val))

{

NODE \*pNode;

for (pNode = hLList->pTail; pNode != NULL; pNode = pNode->pPrev)

if (!Proc(&pNode->val))

returnFALSE;

returnTRUE;

}

void ClearLList(HLLISThLList)

{

NODE \*pNode, \*pNextNode;

pNode = hLList->pHead;

while (pNode != NULL) {

pNextNode = pNode->pNext;

free(pNode);

pNode = pNextNode;

}

hLList->pHead = hLList->pTail = NULL;

hLList->count = 0;

}

void CloseLList(HLLISThLList)

{

ClearLList(hLList);

free(hLList);

}

staticNODE \*createNode(DATATYPEval)

{

NODE \*pNode;

if ((pNode = (NODE \*)malloc(sizeof(NODE))) == NULL)

returnNULL;

pNode->val = val;

return pNode;

}

/\* Test.c \*/

#include<stdio.h>

#include<stdlib.h>

#include"DoublyLinkedList.h"

BOOL Disp(DATATYPE \*val)

{

printf("%d ", \*val);

fflush(stdout);

returnTRUE;

}

int main(void)

{

HLLIST hLList;

int i;

if ((hLList = CreateLList()) == NULL) {

fprintf(stderr, "cannot create linked list!..\n");

exit(EXIT\_FAILURE);

}

for (i = 0; i < 10; ++i) {

if (AddItemTail(hLList, i) == NULL) {

fprintf(stderr, "cannot add item!..\n");

exit(EXIT\_FAILURE);

}

}

DeleteItemIndex(hLList, 99);

WalkLList(hLList, Disp);

printf("\n");

CloseLList(hLList);

return 0;

}

Çift bağlı liste gerçekleştiriminde kontrol yapısında (yani handle alanında) ilk ve son düğümlerin adresleri değil düğümün kendisi tutulursa bazı özel durumlar özel olmaktan çıkabilir. Dolayısıyla tasarım daha sadeleşebilir. Örneğin:

typedefstructtagLLIST {

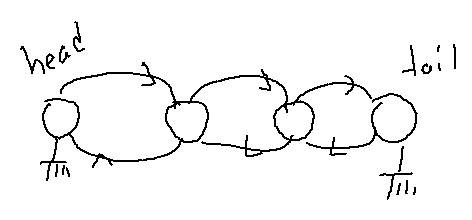
NODE head;

NODE tail;

size\_t count;

} LLIST, \*HLLIST;

Burada son düğümün pNext göstericisi NULL'değerini değil HLIST içerisindeki tail'i gösterecektir. Yani bu tasarımda head ve tail düğümleri sanki baştan listeye ekliymiş gibi ele alınmaktadır:



Bu tasarım baş ve sonla ilgili özel durumları ortadan kaldırır ve bağlı listenin daha kolay gerçekleştirilmesini sağlar. Örnek bir gerçekleştirim şöyle olabilir:

/\* DoublyLinkesList2.h \*/

#ifndef DOUBLYLINKEDLIST2\_H\_

#defineDOUBLYLINKEDLIST2\_H\_

#include<stddef.h>

/\* Symbolic Constants \*/

#defineFALSE 0

#defineTRUE 1

/\* Type declarations \*/

typedefintDATATYPE;

typedefintBOOL;

typedefstructtagNODE {

DATATYPE val;

structtagNODE \*pPrev;

structtagNODE \*pNext;

} NODE;

typedefstructtagLLIST {

NODE head;

NODE tail;

size\_t count;

} LLIST, \*HLLIST;

/\* Function Prototypes \*/

HLLIST CreateLList(void);

NODE \*AddItemTail(HLLIST hLList, DATATYPE val);

NODE \*AddItemHead(HLLIST hLList, DATATYPE val);

NODE \*InsertItemAfterNode(HLLIST hLList, NODE \*pNode, DATATYPE val);

NODE \*InsertItemIndex(HLLIST hLList, size\_t index, DATATYPE val);

void DeleteItemNode(HLLIST hLList, NODE \*pDelNode);

BOOL DeleteItemIndex(HLLIST hLList, size\_t index);

NODE \*FindItem(HLLIST hLList, DATATYPE val);

BOOL WalkLList(HLLIST hLList, BOOL(\*Proc)(DATATYPE \*val));

BOOL WalkLListRev(HLLIST hLList, BOOL(\*Proc)(DATATYPE \*val));

void ClearLList(HLLIST hLList);

void CloseLList(HLLIST hLList);

#endif

/\* DoublyLinkedList2.c \*/

#include<stdio.h>

#include<stdlib.h>

#include"DoublyLinkedList2.h"

/\* static Function Prototypes \*/

staticNODE \*createNode(DATATYPE val);

/\* Function Definitions \*/

HLLIST CreateLList(void)

{

HLLIST hLList;

if ((hLList = (HLLIST)malloc(sizeof(LLIST))) == NULL)

returnNULL;

hLList->head.pNext = &hLList->tail;

hLList->head.pPrev = &hLList->tail;

hLList->tail.pNext = &hLList->head;

hLList->tail.pPrev = &hLList->head;

return hLList;

}

NODE \*AddItemTail(HLLISThLList, DATATYPEval)

{

NODE \*pNewNode;

if ((pNewNode = createNode(val)) == NULL)

returnNULL;

hLList->tail.pPrev->pNext = pNewNode;

pNewNode->pNext = &hLList->tail;

pNewNode->pPrev = hLList->tail.pPrev;

hLList->tail.pPrev = pNewNode;

++hLList->count;

return pNewNode;

}

NODE \*AddItemHead(HLLISThLList, DATATYPEval)

{

NODE \*pNewNode;

if ((pNewNode = createNode(val)) == NULL)

returnNULL;

pNewNode->pPrev = NULL;

pNewNode->pPrev = &hLList->head;

pNewNode->pNext = hLList->head.pNext;

hLList->head.pNext->pPrev = pNewNode;

hLList->head.pNext = pNewNode;

return pNewNode;

}

NODE \*InsertItemAfterNode(HLLISThLList, NODE \*pNode, DATATYPEval)

{

NODE \*pNewNode;

if ((pNewNode = createNode(val)) == NULL)

returnNULL;

pNewNode->pPrev = pNode;

pNewNode->pNext = pNode->pNext;

pNode->pNext->pPrev = pNewNode;

pNode->pNext = pNewNode;

++hLList->count;

return pNewNode;

}

NODE \*InsertItemIndex(HLLISThLList, size\_tindex, DATATYPEval)

{

size\_t i;

NODE \*pNode;

if (index>hLList->count)

returnNULL;

if (index == 0)

return AddItemHead(hLList, val);

pNode = hLList->head.pNext;

for (i = 0; i <index - 1; ++i)

pNode = pNode->pNext;

return InsertItemAfterNode(hLList, pNode, val);

}

void DeleteItemNode(HLLISThLList, NODE \*pDelNode)

{

pDelNode->pPrev->pNext = pDelNode->pNext;

pDelNode->pNext->pPrev = pDelNode->pPrev;

--hLList->count;

free(pDelNode);

}

BOOL DeleteItemIndex(HLLISThLList, size\_tindex)

{

NODE \*pNode;

size\_t i;

if (index>= hLList->count)

returnFALSE;

pNode = hLList->head.pNext;

for (i = 0; i <index; ++i)

pNode = pNode->pNext;

DeleteItemNode(hLList, pNode);

returnTRUE;

}

NODE \*FindItem(HLLISThLList, DATATYPEval)

{

NODE \*pNode;

for (pNode = hLList->head.pNext; pNode != &hLList->tail; pNode = pNode->pNext)

if (pNode->val == val)

return pNode;

returnNULL;

}

BOOL WalkLList(HLLISThLList, BOOL(\*Proc)(DATATYPE \*val))

{

NODE \*pNode;

for (pNode = hLList->head.pNext; pNode != &hLList->tail; pNode = pNode->pNext)

if (!Proc(&pNode->val))

returnFALSE;

returnTRUE;

}

BOOL WalkLListRev(HLLISThLList, BOOL(\*Proc)(DATATYPE \*val))

{

NODE \*pNode;

for (pNode = hLList->tail.pPrev; pNode != &hLList->head; pNode = pNode->pPrev)

if (!Proc(&pNode->val))

returnFALSE;

returnTRUE;

}

void ClearLList(HLLISThLList)

{

NODE \*pNode, \*pNextNode;

pNode = hLList->head.pNext;

while (pNode != &hLList->tail) {

pNextNode = pNode->pNext;

free(pNode);

pNode = pNextNode;

}

hLList->count = 0;

}

void CloseLList(HLLISThLList)

{

ClearLList(hLList);

free(hLList);

}

staticNODE \*createNode(DATATYPEval)

{

NODE \*pNode;

if ((pNode = (NODE \*)malloc(sizeof(NODE))) == NULL)

returnNULL;

pNode->val = val;

return pNode;

}

/\* Test.c \*/

#include<stdio.h>

#include<stdlib.h>

#include"DoublyLinkedList2.h"

BOOL Disp(DATATYPE \*val)

{

printf("%d ", \*val);

fflush(stdout);

returnTRUE;

}

int main(void)

{

HLLIST hLList;

int i;

if ((hLList = CreateLList()) == NULL) {

fprintf(stderr, "cannot create linked list!..\n");

exit(EXIT\_FAILURE);

}

for (i = 0; i < 10; ++i) {

if (AddItemTail(hLList, i) == NULL) {

fprintf(stderr, "cannot add item!..\n");

exit(EXIT\_FAILURE);

}

}

DeleteItemIndex(hLList, 3);

WalkLList(hLList, Disp);

printf("\n");

CloseLList(hLList);

return 0;

}

**Kuyruk (Queue) Veri Yapısı**

Kuyruk FIFO prensibiyle çalışan bir veri yapısıdır. Kuyrukla ilgili iki temel işlem söz konusudur: "Kuyruğa leman ekleme" ve "kuyruktan eleman alma". Kuyruğa insert işlemi ya da kuyruktaki belli bir elemanı silme işlemi bu veri yapısı için anlamlı değildir.

**Kuyruk Veri Yapısına Neden Gereksinim Duyulmaktadır?**

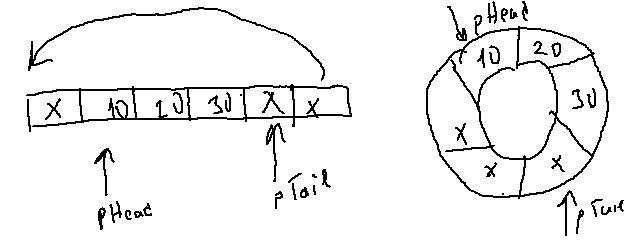
Kuyruklar gemnellikle bilgilerin geçici olarak sıra bozulmadan bekletilmesi için kullanılırlar. (Yani tipik olarak kuyruklar tampon alanların gerçekleştiriminde kullanılmaktadır.) Kuyruk veri yapısıyla pek çok yerde karşılaşılabilmektedir. Örneğin arabalı vapura binme ve inme kuyruğu, yemel kuyrukları gerçek yaşamda karşılaştığımız kuyruklardır. Klavyeden basılan tuşlar işletim sistemi tarafından bir klavye tamponuna yerleştirilir. Bu da bir kuyruk sistemidir. Yazıcıya birden fazla işi iş gönderirse yazıcı bunları kuyruklar ve print işlemini sırasıyla gerçekleştirir. Ya da örneğin işletim sisteminin blokedeki prosesleri ya da thread'leri beklettiği veri yapısı da bir kuyruk sistemi gibidir. Kuyruğa eleman yerleştirme işlemi için geleneksel olarak İngilizce "put" ya da "enqueue" sözcükleri kullanılmaktadır. Eleman almak için de "get" ya da "dequeue" sözcükleri tercih edilir.

**Kuyruk Veri Yapısının Gerçekleştirimi**

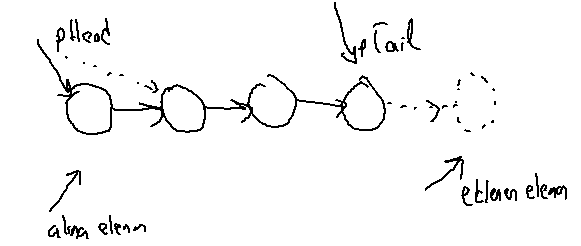
Kuyruklar üç biçimde gerçekleştirilebilirler.

En basit yöntem dizi yöntemidir. Bu yöntemde bir dizi açılır. Eleman kuyruğa yerleştirilmek istendiğinde dizinin sonuna yerleştirilir. Alınmak istendiğinde dizinin başından alınır ve dizi bir kaydırılır. Bu yöntem eleman alma işlemini doğrusal karmaşıklığa yükseltmektedir. Bu nedenle uygulamada tercih edilmez.

İkinci yöntemde (buna döngüsel kuyruk gerçekleştirimi de denilmektedir) bir dizi alınır. İki de gösterici ya da indeks tutulur. Bunlardan birine head göstericisi, diğerine tail göstericisi denilmektedir. Eleman tail göstericisinin gösterdiği yere yerleştirilir ve tail göstericisi bir artırılır. Eleman head göstericisinin gösterdiği yerden alınır ve head göstwricisi bir artırılır. Tabii dizinin sonuna gelindiğinde yeniden başa dönülmelidir.



Üçüncü yöntem bağlı listeyle kuyruk oluşturma yöntemidir. Buna "Linked List Queue" da denilmektedir. Bu yöntemde eleman bağlı listenin sonuna eklenir, başından alınır. Örneğin:



Bu yöntemde kuyruk dinamik olarak büyütülebilmektedir. Tabi dinamik tahsisatların da belli bir zamansal maliyeti vardır. Döngüsel yöntemle kuyruk aşağıdaki gibi oluşturulabilir:

/\* Queue.h \*/

#ifndef QUEUE\_H\_

#defineQUEUE\_H\_

#include<stddef.h>

/\* Symbolic Constants \*/

#defineFALSE 0

#defineTRUE 1

/\* Type declarations \*/

typedefintDATATYPE;

typedefintBOOL;

typedefstructtagQUEUE {

DATATYPE \*pQueue;

size\_t head;

size\_t tail;

size\_t size;

size\_t count;

} QUEUE, \*HQUEUE;

/\* Function Prototypes \*/

HQUEUE CreateQueue(size\_t size);

BOOL PutQueue(HQUEUE hQueue, DATATYPE val);

BOOL GetQueue(HQUEUE hQueue, DATATYPE \*val);

void ClearQueue(HQUEUE hQueue);

void CloseQueue(HQUEUE hQueue);

/\* Macros \*/

#defineIsEmptyQueue(hQueue) ((hQueue)->count == 0)

#defineGetSizeQueue(hQueue) ((hQueue)->size)

#defineGetCountQueue(hQueue) ((hQueue)->count)

#endif

/\* Queue.c \*/

#include<stdio.h>

#include<stdlib.h>

#include"Queue.h"

/\* Function Definitions \*/

HQUEUE CreateQueue(size\_tsize)

{

HQUEUE hQueue;

if ((hQueue = (HQUEUE)malloc(sizeof(QUEUE))) == NULL)

returnNULL;

if ((hQueue->pQueue = (DATATYPE \*)malloc(sizeof(DATATYPE)\* size)) == NULL) {

free(hQueue);

returnNULL;

}

hQueue->size = size;

hQueue->count = 0;

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

return hQueue;

}

BOOL PutQueue(HQUEUEhQueue, DATATYPEval)

{

if (hQueue->count >= hQueue->size)

returnFALSE;

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

hQueue->tail %= hQueue->size;

++hQueue->count;

returnTRUE;

}

BOOL GetQueue(HQUEUEhQueue, DATATYPE \*val)

{

if (hQueue->count == 0)

returnFALSE;

\*val = hQueue->pQueue[hQueue->head++];

hQueue->head %= hQueue->size;

--hQueue->count;

returnTRUE;

}

void ClearQueue(HQUEUEhQueue)

{

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

}

void CloseQueue(HQUEUEhQueue)

{

free(hQueue->pQueue);

free(hQueue);

}

/\* Test.c \*/

#include<stdio.h>

#include<stdlib.h>

#include"Queue.h"

int main(void)

{

HQUEUE hQueue;

DATATYPE val;

int i;

if ((hQueue = CreateQueue(10)) == NULL)

returnNULL;

for (i = 0; i < 100; ++i)

PutQueue(hQueue, i);

while (GetQueue(hQueue, &val))

printf("%d ", val), fflush(stdout);

printf("\n");

CloseQueue(hQueue);

return 0;

}

Kuyruk veri yapısının bağlı liste gerçekleştirimi de şöyle olabilir:

/\* LinkedListQueue.h \*/

#ifndef LINKEDLISTQUEUE\_H\_

#defineLINKEDLISTQUEUE\_H\_

#include<stddef.h>

/\* Symbolic Constants \*/

#defineFALSE 0

#defineTRUE 1

/\* Type declarations \*/

typedefintDATATYPE;

typedefintBOOL;

typedefstructtagNODE {

DATATYPE val;

structtagNODE \*pNext;

} NODE;

typedefstructtagQUEUE {

NODE \*pHead;

NODE \*pTail;

size\_t count;

} QUEUE, \*HQUEUE;

/\* Function Prototypes \*/

HQUEUE CreateQueue(void);

BOOL PutQueue(HQUEUE hQueue, DATATYPE val);

BOOL GetQueue(HQUEUE hQueue, DATATYPE \*val);

void ClearQueue(HQUEUE hQueue);

void CloseQueue(HQUEUE hQueue);

/\* Macros \*/

#defineIsEmptyQueue(hQueue) ((hQueue)->count == 0)

#defineGetCountQueue(hQueue) ((hQueue)->count)

#endif

/\* LinkedListQueue.c \*/

#include<stdio.h>

#include<stdlib.h>

#include"LinkedListQueue.h"

/\* Function Definitions \*/

HQUEUE CreateQueue(void)

{

HQUEUE hQueue;

if ((hQueue = (HQUEUE)malloc(sizeof(QUEUE))) == NULL)

returnNULL;

hQueue->count = 0;

hQueue->pHead = hQueue->pTail = NULL;

return hQueue;

}

BOOL PutQueue(HQUEUEhQueue, DATATYPEval)

{

NODE \*pNewNode;

if ((pNewNode = (NODE \*)malloc(sizeof(NODE))) == NULL)

returnFALSE;

pNewNode->val = val;

pNewNode->pNext = NULL;

if (hQueue->pHead == NULL)

hQueue->pHead = pNewNode;

else

hQueue->pTail->pNext = pNewNode;

hQueue->pTail = pNewNode;

++hQueue->count;

returnTRUE;

}

BOOL GetQueue(HQUEUEhQueue, DATATYPE \*val)

{

NODE \*pNode;

if (hQueue->pHead == NULL)

returnFALSE;

pNode = hQueue->pHead;

\*val = pNode->val;

hQueue->pHead = pNode->pNext;

if (hQueue->pHead == NULL)

hQueue->pTail = NULL;

--hQueue->count;

free(pNode);

returnTRUE;

}

void ClearQueue(HQUEUEhQueue)

{

NODE \*pNode, \*pTempNode;

pNode = hQueue->pHead;

while (pNode != NULL) {

pTempNode = pNode->pNext;

free(pNode);

pNode = pTempNode;

}

hQueue->pHead = hQueue->pTail = NULL;

hQueue->count = 0;

}

void CloseQueue(HQUEUEhQueue)

{

ClearQueue(hQueue);

free(hQueue);

}

/\* Test.c \*/

#include<stdio.h>

#include<stdlib.h>

#include"LinkedListQueue.h"

int main(void)

{

HQUEUE hQueue;

DATATYPE val;

int i;

if ((hQueue = CreateQueue()) == NULL)

returnNULL;

for (i = 0; i < 100; ++i)

PutQueue(hQueue, i);

while (GetQueue(hQueue, &val))

printf("%d ", val), fflush(stdout);

printf("\n");

CloseQueue(hQueue);

return 0;

}

**Stack Veri Yapısı**

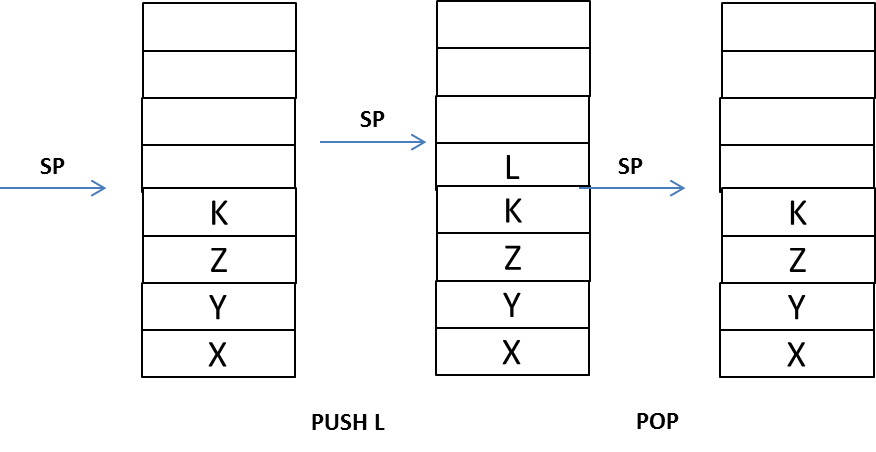
Stack LIFO pensibiyle çalışan kuyruk sistemidir. Yani stack'e son yerleştirilen bilgi ilk olarak alınır. Geleneksel olarak stack'e eleman ekleyen fonksiyonlar İngilizce "push", stack'ten eleman alan fonksiyonlar "pop" biçiminde isimlendirilmektedir. Push ve pop işlemleri O(1) karmaşıklıkta gerçekleştirilmektedir.

**Stack Veri Yapısına Neden Gereksinim Duyulmaktadır?**

Stack sistemlerine gerçek hayatta da karşılaşılmaktadır. Örneğin asansöre son binenler ilk inebilirler. Tabakları üst üsete koyduğumuzda en üsttekini önce alırız. Alış veriş arabalarındaki sistem benzerdir. Programlama da da stack sistemleriyle karşılaşılmaktadır. Örneğin undo mekanizması bir stack sistemini kullanır. (Son yapılan değişikliği ilk geri alırız). Ya da örneğin bir pencere aktifken onu kapattığımızda en üstteki aktif yapılmaktadır. Bu da stack sistemini çağrıştırır. Bazı algoritmalar stack sistemini kullanmaktadır. Örneğin aslında özyineleme içeren algoritmalar özyineleme olmadan yapay bir stack'le de gerçekleştirilebilir. Parsing algoritmaları stack kullanan algoritmalara tipik örnektir. Stack sistemleri mikroişlemcilerin çalışma mekanizmasında mutlak bulunması gereken bir mekanizmadır.

**Stack Veri Yapısının Gerçekleştirilmesi**

Stack sistemleri de iki biçimde gerçekleştirilebilir. Dizi yönteminde stack bir dizi olarak oluşturulur. Stack'in aktif noktası bir göstericiyle gösterilir. Bu göstericiye geleneksel olarak "stack gösterici (stack pointer)" denilmektedir. Push işlemi sırasında stack göstericisi bir azaltılır ve eleman stack göstericisinin gösterdiği yere yerleştirilir. Pop işleminde de stack göstericisinin gösterdiği yerdeki eleman alınır ve stack göstericisi bir artırılır. İşin başında stack göstericisi dizinin sonundadır.



Şüphesiz psuh ve pop işlemleri aslında tam ters olarak da yapılabilir. Yani örneğin iin başında stack göstericisi dizinin başını gösterebilir. Push işlemi sırasında eleman stack göstericisinin gösterdiği yere eklenip stack göstericisi bir artırılabilir. Pop işlemi sırasında da önce stack göstericisi bir azaltılıp oradaki eleman alınabilir. Fakat geleneksel durum. Stack göstericisinin başlangıçta dizinin sonunu göstermesi durumudur.

Bağlı liste tekniğinde (list stack) push işlemi sırasında bağlı listenin önüne eleman eklenir. Pop işleminde yine önünden eleman alınır.

Eğer stack'e çok fazla elemen yerleştirilirse bu durumda stack yukarıdan taşar (stack overflow). Stack'ten çok fazla eleman alınmaya çalışılırsa bu kez stack aşağıdan taşar (stack underflow).

Dizi kullanılarak stack gerçekleştirimi şöyle yapılabilir:

/\* Stack.h \*/

#ifndef STACK\_H\_

#defineSTACK\_H\_

#include<stddef.h>

/\* Symbolic Constants \*/

#defineFALSE 0

#defineTRUE 1

/\* Type declarations \*/

typedefintDATATYPE;

typedefintBOOL;

typedefstructtagSTACK {

DATATYPE \*pStack;

size\_t size;

size\_t count;

DATATYPE \*sp;

} STACK, \*HSTACK;

/\* Function Prototypes \*/

HSTACK CreateStack(size\_t size);

BOOL Push(HSTACK hStack, DATATYPE val);

DATATYPE Pop(HSTACK hStack);

BOOL PopSecure(HSTACK hStack, DATATYPE \*val);

void ClearStack(HSTACK hStack);

void CloseStack(HSTACK hStack);

/\* Macros \*/

#defineIsStackEmpty(hStack) (!(hStack)->count)

#defineGetStackCount(hStack) ((hStack)->count)

#defineGetStackSize(hStack) ((hStack)->size)

#endif

/\* Stack.c \*/

#include<stdio.h>

#include<stdlib.h>

#include"Stack.h"

/\* Function Definitions \*/

HSTACK CreateStack(size\_tsize)

{

HSTACK hStack;

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

returnNULL;

if ((hStack->pStack = (DATATYPE \*)malloc(size \* sizeof(DATATYPE))) == NULL) {

free(hStack);

returnNULL;

}

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

hStack->size = size;

hStack->count = 0;

return hStack;

}

BOOL Push(HSTACKhStack, DATATYPEval)

{

if (hStack->sp == hStack->pStack)

returnFALSE;

\*--hStack->sp = val;

++hStack->count;

returnTRUE;

}

DATATYPE Pop(HSTACKhStack)

{

--hStack->count;

return \*hStack->sp++;

}

BOOL PopSecure(HSTACKhStack, DATATYPE \*val)

{

if (hStack->sp == hStack->pStack + hStack->size)

returnFALSE;

\*val = \*hStack->sp++;

--hStack->count;

returnTRUE;

}

void ClearStack(HSTACKhStack)

{

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

hStack->count = 0;

}

void CloseStack(HSTACKhStack)

{

free(hStack->pStack);

free(hStack);

}

/\* Test.c \*/

#include<stdio.h>

#include<stdlib.h>

#include"Stack.h"

/\* Test code \*/

int main(void)

{

HSTACK hStack;

int i;

if ((hStack = CreateStack(4)) == NULL) {

fprintf(stderr, "cannot create stack!..\n");

exit(EXIT\_FAILURE);

}

for (i = 0; "ankara"[i] != '\0'; ++i)

Push(hStack, "ankara"[i]);

while (!IsStackEmpty(hStack))

putchar(Pop(hStack));

printf("\n");

return 0;

}

Bağlı liste kullanılarak stack geçekleştirimi de şöyle yapılabilir:

/\* StackList.h \*/

#ifndef STACKLIST\_H\_

#defineSTACKLIST\_H\_

#include<stddef.h>

/\* Symbolic Constants \*/

#defineFALSE 0

#defineTRUE 1

/\* Type declarations \*/

typedefintDATATYPE;

typedefintBOOL;

typedefstructtagNODE {

DATATYPE val;

structtagNODE \*pNext;

} NODE;

typedefstructtagSTACK {

NODE \*pHead;

size\_t count;

} STACK, \*HSTACK;

/\* Function Prototypes \*/

HSTACK CreateStack(void);

BOOL Push(HSTACK hStack, DATATYPE val);

BOOL Pop(HSTACK hStack, DATATYPE \*val);

void ClearStack(HSTACK hStack);

void CloseStack(HSTACK hStack);

/\* Macros \*/

#defineIsStackEmpty(hStack) (!(hStack)->count)

#defineGetStackCount(hStack) ((hStack)->count)

#endif

/\* StackList.c \*/

#include<stdio.h>

#include<stdlib.h>

#include"StackList.h"

/\* Function Definitions \*/

HSTACK CreateStack(void)

{

HSTACK hStack;

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

returnNULL;

hStack->pHead = NULL;

hStack->count = 0;

return hStack;

}

BOOL Push(HSTACKhStack, DATATYPEval)

{

NODE \*pNewNode;

if ((pNewNode = (NODE \*)malloc(sizeof(NODE))) == NULL)

returnFALSE;

pNewNode->val = val;

pNewNode->pNext = hStack->pHead;

hStack->pHead = pNewNode;

++hStack->count;

returnTRUE;

}

BOOL Pop(HSTACKhStack, DATATYPE \*val)

{

NODE \*pNode;

if (hStack->pHead == NULL)

returnFALSE;

pNode = hStack->pHead;

hStack->pHead = pNode->pNext;

\*val = pNode->val;

--hStack->count;

free(pNode);

returnTRUE;

}

void ClearStack(HSTACKhStack)

{

NODE \*pNode, \*pDelNode;

pNode = hStack->pHead;

while (pNode != NULL) {

pDelNode = pNode;

pNode = pNode->pNext;

free(pDelNode);

}

hStack->pHead = NULL;

hStack->count = 0;

}

void CloseStack(HSTACKhStack)

{

ClearStack(hStack);

free(hStack);

}

/\* Test.c \*/

#include<stdio.h>

#include<stdlib.h>

#include"StackList.h"

/\* Test code \*/

int main(void)

{

HSTACK hStack;

int i;

DATATYPE val;

if ((hStack = CreateStack()) == NULL) {

fprintf(stderr, "cannot create stack!..\n");

exit(EXIT\_FAILURE);

}

for (i = 0; "ankara"[i] != '\0'; ++i)

Push(hStack, "ankara"[i]);

while (!IsStackEmpty(hStack)) {

Pop(hStack, &val);

printf("%c", val);

}

printf("\n");

return 0;

}

**Çift Yönlü Dinamik Diziler (Double Ended Queue (Deque))**

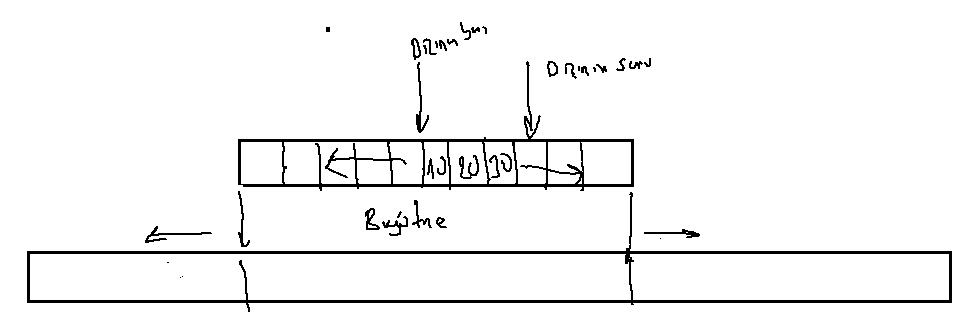
Çift yönlü dinamik dizilerin normal dinamik dizilerden tek farkı başa ve sona eleman eklemenin ek maliyetli sabit zamanlı olmasıdır. Anımsanacağı gibi normal dinamik dizilerde sona eleman ekleme ek maliyetli sabit karmaşıklıktayken başa (ya da araya) eleman eklemek doğrusal karmaşıklıktadır. Çift yönlü dinamik dizilerde elemana erişmek ek maliyetli sabit zamanlı bir işlemdir.

**Çift Yönlü Dinamik Dizilere Neden Gereksinim Duyulmaktadır?**

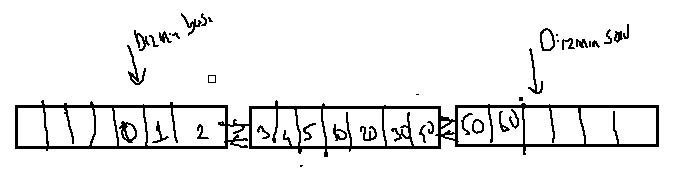
Bazı sistemlerde hem başa hem de sona eleman ekleme mümkün olabilmektedir. Örneğin gelen tek sayıları dizinin başına çift sayıları sonuna yerleştirmek istediğimizi düşünelim. Bunu normal dinamik diziyle yapmak istersek eleman ekleme ancak işlemi doğrusal karmaşıkta yapılabilir. Fakat çift yönlü dinamik dizilerde sabit karmaşıklıkta yapılabilmektedir. Çift yönlü dinamik diziler genel bir veri yapısı olarak da kullanılabilir. Yani örneğin bunlar hem stack hem kuyruk gerçekleştiriminde kullanılabilmektedir.

**Çift Yönlü Dinamik Dizilerin Gerçekleştirilmesi**

Çift yönlü dinamik dizilerin gerçekleştirimi tipik olarak iki biçimde yapılabilmektedir. Birinci biçimde bir dizi alnır. Fakat dizinin indeks olarak başı ve sonu ayrı bir göstericiyle tutturulur. Böylece başa ve sona ekleme sabit zamanlı yapılabilir. Örneğin:



İkinci yöntemde bloklar bir bağlı listeyle birbirine bağlanır.



Bu yöntemde dizinin başında ve sonunda daha az boş yer kalır. Ayrıca araya ekleme ve aradan eleman silme durumunda bundan tüm bloklar etkilenmez. Fakat bu gerçekleştirimde elemana erişmek daha zordur. Blok sayısı logaritmik olarak artırılmıyorsa elemana erişim doğrusal zamana yakınsar. İki gerçekleştiriminde iyi ve kötü tarafları vardır.

**Arama İşlemleri**

Bilgisayar bilimlerinde bir anahtar ile aynı değere sahip elemanın bulunması sürecine arama (searching) denilmeketedir. Arama işlemleri içsel (internal) ve dışsal (external) olmak üzere ikiye ayrılır. Eğer arama birincil bellekte saklanan bilgilerin üzerinde yapılıyorsa buna içsel arama, diskteki dosyaların içerisinde yapılıyorsa buna da dışsal arama denilmektedir. Ancak arama denildiğinde default olarak içsel arama anlaşılmaktadır.

Arama işlemleri nasıl hızlandırılabilir? Örneğin bir dizi içerisinde bir değeri aramak isteyelim, hangi yöntemleri kullanabiliriz? Öncelikle ideal bir arama nasıl olabilir bunun üzerinde düşünelim. Şüphesiz ideal durumda aramanın hızlı bir biçimde adeta rastgele erişimli olarak yapılması istenir. İndeksli arama (indexed search) denilen yöntemde bir dizi açılır. Anahtar diziye indeks yapılarak dizinin o indeksine eleman yerleştirilir. Sonra arama yapılırken doğrudan o indekse bakılır. Örneğin 100 kişiye birer numara verilmiş olsun. Biz de numaraya göre aramak yapacak olalım. Herkesi numarasına uygun indekse yerleştiririz. Araken de o indekse bakarız. Böyle bir sistem O(1) karmaşıklıkta ideal aramaya olanak sağlar. Ancak bu sistem pratikte pek çok durum için uygun değildir. Eğer dizilimdeki eleman sayısı dinamik olarak sürekli artabiliyorsa, ya da anahtarın indeks genişliği çok büyükse yöntem kullanılabilir olmaktan çıkar. (Örneğin 100 kişiyi TC numaralarına göre bu yöntemle diziye yerleştirmek isteyelim. Bizim çok çok büyük bir dizi açmamız gerekir. Bu dizinin çok az kısmı dolu olacaktır.) Fakat yine de indeskli arama buna uygun sistemlerde kullanılabilir.

Eğer bir dizilimde elemanlar arasında hiçbir kural yoksa ilk akla gelen yöntem sıralı aramadır (sequential search). Sıralı aramada dizilimdeki eleman sayısı N ise ortalama karmaşıklık N / 2'dir. Big O notasyonuna göre arama doğrusal karmaşıklıkta yapılmaktadır. (Örneğin 1000000 elemanlı dizi için ortalama olarak 500000 karşılaştırma yapmak gerekir.) Sıralı arama 20'den küçük sayıda diziler için belki de en iyi arama yöntemidir. Çünkü ileride sözünü edeceğimiz algoritmik arama yöntemlerinin kurulumu için belli bir zaman harcanmaktadır. Oysa sıralı aramada böyle bir hazırlığın yapılmasına gerek olmaz. Bu nedenle çok az sayıda eleman için özel bir algoritmik yöntemin düşünülmesine hiç gerek yoktur.

Eğer dizilimdeki elemanlar sıraya dizilmişse ikili arama yöntemi (binary search) en iyi yöntemdir. İkili aramada iki çubuk (yani indeks) tutulur. Ortadaki elemana bakılır. Sol ya da sağ çubuk oraya çekilir. Yani her defasında dizilim yarısı kadar küçültülmektedir. İkili aramanın en kötü durum karmaşıklılığı log2 N'dir. (Örneğin 1024 eleman için 10, 1000000 için yaklaşık 20 karşılaştırma). İkili arama algoritması tipik olarak şöyle gerçekleştirilmektedir.

#include<stdio.h>

#include<stdlib.h>

#include<string.h>

#include<Windows.h>

typedefintDATATYPE;

DATATYPE \*BinarySearch(constDATATYPE \*array, intsize, DATATYPEval)

{

int left = 0, right = size - 1, mid;

DATATYPE \*pVal = NULL;

while (left <= right) {

mid = (left + right) / 2;

if (array[mid] <val)

left = mid + 1;

elseif (array[mid] >val)

right = mid - 1;

else {

pVal = (DATATYPE \*)&array[mid];

break;

}

}

return pVal;

}

int main(void)

{

DATATYPE array[10] = { 1, 3, 5, 7, 9, 11, 13, 15, 17, 19 };

DATATYPE \*pVal;

if ((pVal = BinarySearch(array, 10, 0)) == NULL)

printf("Bulunamadi!..\n");

else

printf("Bulundu: %d\n", \*pVal);

return 0;

}

Pekiyi hep önce diziyi sıraya dizip sonra arama yapsak iyiolmaz mı? Diziyi sıraya dizmenin en iyi karmaşıklığı N log N'dir. Bu durumda sıraya dizmenin maliyeti zaten sıralı aramanın maliyetinden yüksektir. Tabi dizilime az seyrek olarak eleman ekleniyorsa fakat çok arama yapılıyorsa bu yöntem uygun olabilir. Fakat çoğu kez durum böyle değildir.

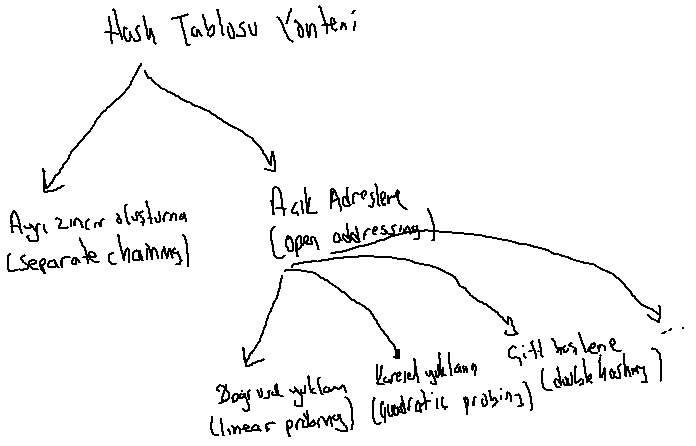
Pekiyi sıralı diziler için ikili aramadan daha iyi bir yöntem olabilir mi? Eğer biz dizilimin ortasına değil de daha uygun bir yerine bakabilirsek sol ya da sağ çubukları daha etkin konumlandırabiliriz. Başka bir deyişle her defasında diziyi ortadan değil de daha uygun bir oranda daraltırsak daha hızlı yakınsama sağlanabilir. (Örneğin her defasında çubukları yarıdan değil, 1/3'ten hizalamak gibi). Tabi böyle bir çıkarımda bulunabilmek için dizilimin dağılımı hakkında bilgi sahibi olmak gerekir. Ancak dizilim hakkında bir bilgi yoksa böyle bir algoritma daha fazla zaman kaybına da yol açabilir. Bu yöntgeme üstel arama (exponential search) denilmektedir. Dağılımı bilinmeyen sıralı dizilerde ikili arama en etkin yöntemdir.

Pekiyi dizinin sıralı olmadığını düşünelim ve sistemin indeksli aramaya da izin vermediğini varsayalım. Daha etkin bir arama nasıl yapılabilir? İşte algoritmik aramalar için daha eleman yerleştirilirken onun aranabileceğini dikkate almak gerekir. Bunun için eleman bir veri yapısına yerleştirilmelidir. Algoritmik arama yöntemleri tipik olarak "arama ağaçları"" ve "hash tabloları" biçiminde iki gruba ayrılabilir. Bunların dışında başka algortimik arama yöntemleri olsa da uygulamada bu iki yöntem grubu en sık kullanılanlardır.

**Hash Tabloları İle Arama (Hashing)**

Hash tabloları ile arama "indeksli arama" ile "sıralı arama"nın bir karmaşımı gibidir. Bu yöntemde bir dizi açılır. Anahtar bir fonksiyona sokularak dizi indeksine dönüştürülür ve eleman o indekse yerleştirilmek istenir. Örneğin kişileri TC kimlik numaralarına göre bir hash tablosuna yerleştirmek isteyelim. Bunun için 100 elemanlı bir dizi açmış olalım. TC kimlik numarası 11 basamaklı büyük bir numaradır. Biz bu numaradan [0, 99] arasında bir indeks elde etmek isteyelim. Bu indeksi elde eden fonksiyona "hash fonksiyonu" denilmektedir. Örneğin 100'e bölümünden elde edilen kalan basit hash fonksiyonu olarak kullanılabilir. Bu durumda 41106234567 TC numaralı kişi bu dizinin 67'inci indeksine yerleştirilmek istenecektir. Eleman aranırken de aynı biçimde anahtardan aynı hash fonksiyonuyla dizi indeksi elde edilir ve o indekse bakılmak istenir.

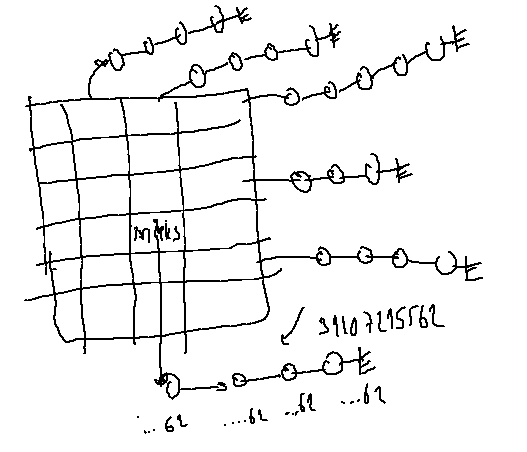
Şüphesiz yukarıda açıklanan yöntemde her zaman bir çakışma (collision) olasılığı vardır. Örneğin 23445623267 tc numaralı kişi için de aynı indeks ele edilecektir. İşte hash tablosu yöntemi çakışma durumundaki stratejiye göre iki temel alt gruba ayrılmaktadır:



Açık adresleme yöntemi de kendi aralarında pek çok alt yönteme ayrılmaktadır. En çok kullanılan çakışma çözüm yöntemi ( collision resolution methods) "ayrı zincir oluşturma" yöntemidir.

**Ayrı Zincir Oluşturma Yöntemi**

Bu yöntemde N elemanlı bir dizi açılır. Fakat dizide elemanlar tutulmaz. Bu bir bağlı liste dizisidir. Yani dizinin her elemanı bir bağlı listenin head göstericisini tutar. Eleman yerleştirileceği zaman hash fonksiyonuna sokulur. Buradan bir dizi indeksi elde edilir. Sonra eleman bu indeksteki bağlı listeye eklenir. Arama yapılırken de benzer biçimde önce anahtar hash fonksiyonuna sokulur. Oradan elde edilen indeksteki bağlı listede doğrusal arama yapılır.



Bu yöntemde eleman eklemek sabit karmaşıklığa sahip bir işlemdir. Arama ise doğrusal karmaşıklıktga gözükmekle birlikte az sayıda eleman sahip dizi içerisinde yapılacağı için çok etkindir. Gerçekten de Knuth bağlı listelerdeki eleman sayısı ortalama 10'u geçmedikten sonra bu yöntemi süper bir yöntem olarak nitelndirmektedir. eğer bağlı listedeki elemanların sayısı çok fazla olursa (yani tablo küçük kalırrsa) yöntem doğusal aramaya yaklaşmaya başlar. O halde tablo uzunluğunu baştan iyi öngörmek gerekir. Örneğin ilgili sisteme ortalama 10000 eleman yerleştirilecekse bizim tabloyu 1000 civarında tutmamız uygun olur.

Hash tabloları özellikle işletim sistemlerinin çekirdek kodlamalarında çok sık karşımıza çıkmaktadır. Örneğin Linux'ta cache sistemlerinde arama yapılırken hep ayrı zincirli hash tabloları kullanılmaktadır. Örneğin Linux'un buffer cache (disk cache) mekanizmasını düşünelim. Burada dosya fonksiyonları okunacak yerin disk blok adresini hesapladıktan sonra onun buffer cache içerisinde olup olmadığına bakmak ister. Blok adresleri birer tamsayıyla belirtilir. Örneğin read fonksiyonu okunacak dosya parçasınının diskin 181317'inci bloğunda olduğunu hesaplasın. Bu blok acaba cache'te midir? İşte Linux bu değeri anahtar yaparak bir hash tablosunda arama yapar. Eğer bulursa doğrudan hiç disk okuması yapmadan bilgiyi oradan alarak verir. Bnezer biçimde dizin girişleri (dentry cache), inode elemanları (inode cache) hep bu biçimde cache sistemleri içerisinde saklanmaktadır.

Ayrı zincir oluşturma yönteminin örnek bir gerçekleştirimi şöyle olabilir:

/\* HashTable.h \*/

#ifndef HASHTABLE\_H\_

#defineHASHTABLE\_H\_

#include<stddef.h>

/\* Symbolic Constants \*/

#defineFALSE 0

#defineTRUE 1

/\* Type declarations \*/

typedefstructtagPERSON {

int no;

char name[64];

} PERSON;

typedefPERSONDATATYPE;

typedefintBOOL;

typedefintKEY;

typedefstructtagNODE {

DATATYPE val;

structtagNODE \*pNext;

} NODE;

typedefstructtagHTABLE {

NODE \*\*pHeads;

size\_t size;

size\_t count;

} HTABLE, \*HHTABLE;

/\* Function Prototypes \*/

HHTABLE CreateHashTable(size\_t size);

BOOL InsertItem(HHTABLE hHtable, constDATATYPE \*pVal);

DATATYPE \*FindItem(HHTABLE hHTable, KEY key);

BOOL DeleteItem(HHTABLE hHTable, KEY key);

void ClearHashTable(HHTABLE hHTable);

void CloseHashTable(HHTABLE hHtable);

/\* Macros \*/

#defineGetCount(hHTable) ((hHTable)->count)

#defineGetSize(hHTable) ((hHTable)->size)

#endif

/\* HashTable.c \*/

#include<stdio.h>

#include<stdlib.h>

#include"Hashtable.h"

/\* static Function Prototypes \*/

staticsize\_t getHash(size\_t size, KEY key);

/\* Function Definitions \*/

HHTABLE CreateHashTable(size\_tsize)

{

HHTABLE hHTable;

size\_t i;

if ((hHTable = (HHTABLE)malloc(sizeof(HTABLE))) == NULL)

returnNULL;

if ((hHTable->pHeads = (NODE \*\*)malloc(sizeof(NODE \*)\* size)) == NULL) {

free(hHTable);

returnNULL;

}

for (i = 0; i <size; ++i)

hHTable->pHeads[i] = 0;

hHTable->count = 0;

hHTable->size = size;

return hHTable;

}

BOOL InsertItem(HHTABLEhHTable, constDATATYPE \*pVal)

{

size\_t index;

NODE \*pNewNode;

index = getHash(hHTable->size, pVal->no);

if ((pNewNode = (NODE \*)malloc(sizeof(NODE))) == NULL)

returnFALSE;

pNewNode->val = \*pVal;

pNewNode->pNext = hHTable->pHeads[index];

hHTable->pHeads[index] = pNewNode;

++hHTable->count;

returnTRUE;

}

DATATYPE \*FindItem(HHTABLEhHTable, KEYkey)

{

size\_t index;

NODE \*pNode;

DATATYPE \*pVal = NULL;

index = getHash(hHTable->size, key);

for (pNode = hHTable->pHeads[index]; pNode != NULL; pNode = pNode->pNext)

if (pNode->val.no == key) {

pVal = &pNode->val;

break;

}

return pVal;

}

BOOL DeleteItem(HHTABLEhHTable, KEYkey)

{

size\_t index;

NODE \*pNode, \*pPrevNode;

BOOL result = FALSE;

index = getHash(hHTable->size, key);

for (pPrevNode = pNode = hHTable->pHeads[index]; pNode != NULL; pNode = pNode->pNext) {

if (pNode->val.no == key) {

if (pNode == pPrevNode)

hHTable->pHeads[index] = pNode->pNext;

else

pPrevNode->pNext = pNode->pNext;

free(pNode);

--hHTable->count;

result = TRUE;

break;

}

pPrevNode = pNode;

}

return result;

}

void ClearHashTable(HHTABLEhHTable)

{

size\_t i;

NODE \*pNode, \*pTempNode;

for (i = 0; i <hHTable->size; ++i) {

pNode = hHTable->pHeads[i];

while (pNode != NULL) {

pTempNode = pNode->pNext;

free(pNode);

pNode = pTempNode;

}

}

for (i = 0; i <hHTable->size; ++i)

hHTable->pHeads[i] = NULL;

hHTable->count = 0;

}

void CloseHashTable(HHTABLEhHTable)

{

ClearHashTable(hHTable);

free(hHTable->pHeads);

free(hHTable);

}

staticsize\_t getHash(size\_tsize, KEYkey)

{

returnkey % size;

}

/\* Test.c \*/

#include<stdio.h>

#include<stdlib.h>

#include<time.h>

#include"HashTable.h"

void getRandomPerson(PERSON \*per)

{

int i;

per->no = rand() % 100000;

for (i = 0; i < 31; ++i)

per->name[i] = 'A' + rand() % 26;

per->name[i] = '\0';

}

int main(void)

{

int i;

PERSON per, perSpecific = { 1234567, "Kaan Aslan"};

HHTABLE hHTable;

DATATYPE \*pPer;

srand(time(NULL));

if ((hHTable = CreateHashTable(101)) == NULL) {

fprintf(stderr, "cannot create hash table!..\n");

exit(EXIT\_FAILURE);

}

for (i = 0; i < 1000; ++i) {

getRandomPerson(&per);

InsertItem(hHTable, i == 500 ? &perSpecific : &per);

}

if (!DeleteItem(hHTable, 1234567)) {

fprintf(stderr, "cannot delete item!..\n");

exit(EXIT\_FAILURE);

}

if ((pPer = FindItem(hHTable, 1234567)) != NULL)

printf("Buldu: %s, %d\n", pPer->name, pPer->no);

else

printf("Bulamadi!\n");

CloseHashTable(hHTable);

return 0;

}

**Hash Fonksiyonu Nasıl Olmalıdır?**

Bilindiği gibi hash fonksiyonu anahtarı dizi indeksine dönüştürmektedir. İyi bir hash fonksiyonunun şu özelliklere sahip olması beklenir:

- Tabloya yaydırmayı iyi yapmalıdır. Yani örneğin anahtar değerleri yanlı olsa bile hash fonksiyonun tabloya yansız bir biçimde eşit miktarda dağıtım yapabilecek yetenekte olması beklenir.

- Hash fonksiyonunun hızlı olması istenir. Çünkü eleman ekleme ve arama işlemlerinde devreye girmektedir.

Sayıdan sayı elde eden, yazıdan sayı elde eden kaliteli hash fonksiyonları için Internet'te araştırma yapılabilir. Örneğin yazıyı sayıya dönüştüren bir hash fonksiyonu şöyle olabilir:

size\_t hash(size\_t size, const char \*str)

{

unsignedlong hash = 5381;

while (\*str != '\0')

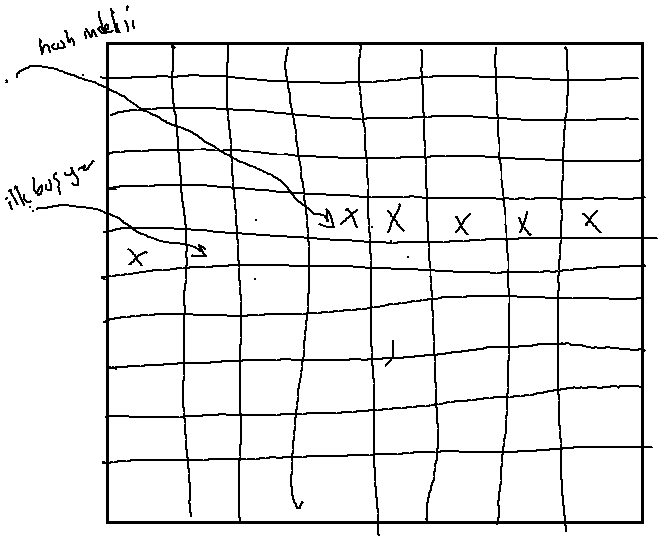
hash = ((hash << 5) + hash) + \*str++;

return hash % size;

}

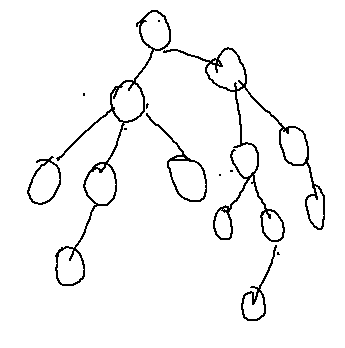
**Hash Tablosu Yönteminde Açık Adresleme (Open Addressing)**

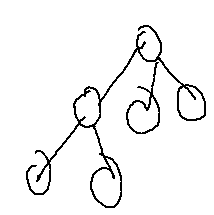
Yukarıda da belirtildiği gibi açık adresleme yöntemi pek çok alt yönteme ayrılmaktadır. En çok kullanılan açık adresleme yöntemi "doğrusal yoklama (linear probing)" denilen yöntemdir. Bu yöntemde anahtardan hash fonksiyonuyla bir tablo indeksi elde edilir. O indeksteki eleman (bucket) boşsa yerleştirme oraya yapılır. Doluysa sırasıyla elemanlar üzerinde ilerlenerek ilk boş yer bulunur. Eleman ilk boş yere eklenir. Arama yapılırken ayhnı biçimde dizi indeksi elde edilir ve elemanlar sırasıyla gözden geçirilir. Tabi ilk boş eleman görüldüğünde artık durulabilir. Bu yöntemde tablonun geniş açılması önemlidir. Eğer tablo küçük kalırsa bu durumda tabloda boş elemanlar azalır. Bu da hem eleman eklerken hem de ararken zaman kaybettirir.

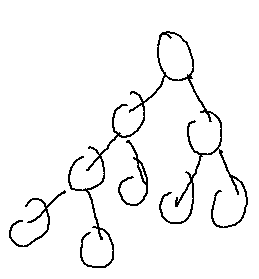


**Ağaç Veri Yapıları (Tree Data Structure)**

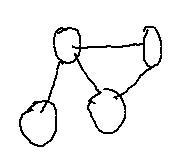
Bir köke sahip sahip olan ve bir düğüme tek bir yoldan ulaşılabilen veri yapılarına ağaç (tree) denir. Ağacın köküne giden bir yol yoktur. Her ağaçta tek bir kök bulunmak zorundadır. Ağaç bir tür graf olarak da değerlendirilebilir. Genel olarak bir düğüme birden fazla yoldan ulaşılabiliyorsa bu tür veri yapılarına graf (graph) denilmektedir. Aşağıda birkaç örnek ağaö görülmektedir:







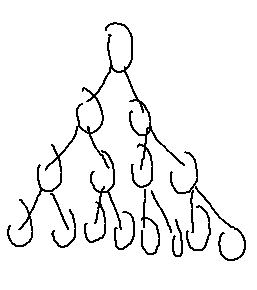
Ancak aşağıdaki veri yapısı bir ağaç değildir, graf belirtir:



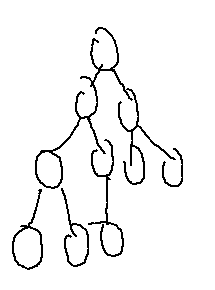
**Ağaç Terminolojisi**

Ağaçlarda kendisine hiçbir yerden gelinemeyen özel düğüme kök (root) denir. Bir düğümden gidilebilen ilk düğümlere o düğümün alt düğümleri (child nodes) denilmektedir. O düğüme gelinen ilk düğüme ise o düğümün üst düğümü (parent node) denilmektedir. Ağaçlarda kök dışındaki her düğümün bir ve yalnızca bir tane üst düğümü vardır. Fakat bir düğümün birden fazla alt düğümü olabilir. Üst düğümleri aynı olan düğümlere kardeş düğümler (sibling nodes) denilmektedir.

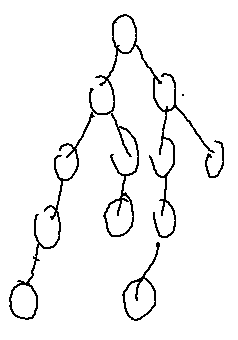
Eğer bir ağaçta her düğümün en fazla n tane alt düğümü olacağı yönünde bir koşul varsa böyle ağaçlara n'li ağaçlar (n'ary tree) denilmektedir. Pratikte en çok kullanılan n'li ağaçlar ikili ağaçlardır. İkili ağaçlarda her düğümün sıfır, bir ya da en fazla iki alt düğümü olabilir. Ağaçlarda hiç alt düğümü olmayan düğümlere yaprak düğüm (leaf node) denilmektedir. Düğümleri birbirine bağlayan çizgiler kenar (edge) olarak isimlendirilir. Bir düğümden yaprağa giden en uzun yolun kenar sayısına o düğümün yüksekliği denilmektedir. Bir ağaçta kökten en azaktaki yaprağa giden yüksekliğe ağacın yüksekliği denir. Bir n'li ağaçta son kademe yapraklar dışındaki tüm düğümlerin tam olarak n tane alt düğümü varsa bu ağaça tam n'li ağaç (full n'ary tree) denilmektedir. Örneğin aşağıda tam bir ikili ağaç görülmektedir:



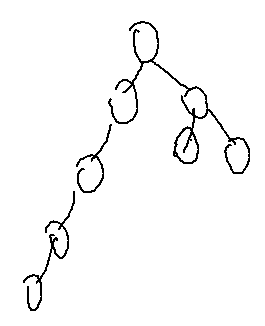
Eğer bir ağaçta sondan b ir önceki kademe dışındaki tüm düğümlerin tam olarak n tane alt düğümü varsa böyle ağaçlara tam n'li ağaçlar (complete n'ary tree) denilmektedir. Örneğin aşağıda tam bir ikili ağaç görülmektedir:



Eğer kökten itibaren her yaprağa giden yolun yüksekliği arasındaki fark en fazla belli sınırlarda kalıyorsa bu tür ağaçlara dengelenmiş ağaçlar (balanced tree) denir. Örneğin:



Burada her yaprağa giden yolun yükseklikleri arasında en fazla bir fark vardır. Böyle bir ağaç dengelidir. Bu bir koşulu altında aşağıdaki ağaç dengeli değildir:

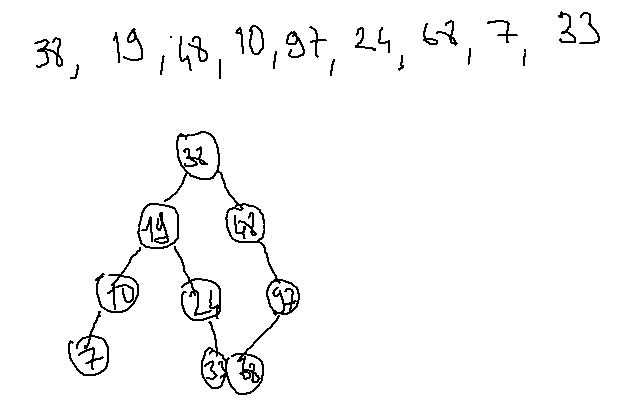


Uygulamada dengeleme için en fazla iki bir yükseklik farkı kullanılmaktadır. Yukarıdaki ağaç bir fark için dengeli değildir. Görüldüğü gibi dengelenmiş (balanced) terimindeki denge ağacın her tarafındaki yüksekliğin belli bir sınır içerisinde olmasını gerektirmektedir. Dengelemedeki koşullar ayrıca tanımlanabilir. Ancak kavram olarak denge ağacın her yerindeki yüksekliğinin benzer olması anlamına gelir.

**Arama Ağaçları (Sarch Trees)**

Ağaçlar pek çok şeyi modellemek için oluşturulabilir. Örneğin bir dizin yapısı da aslında bir ağaçla temsil edebilir. Bir işletmedeki hiyerarşik yapı da bir ağaçla temsil edilebilir. Ancak ağaçlar arama işlemlerinde de çok sık kullanılmaktadır. Arama amaçlı oluşturulmuş ağaçlara arama ağaçları (search tress) denilmektedir. En çok kullanılan arama ağaçları ikili ağaçlardır.

Bir arama ağacında düğümler kayıtları temsil eder ve her düğümde bir anahtar bulunmaktadır. (Örneğin her düğümde bir kişinin bilgileri saklanabilir. Anahtar da kişinin TC kimlik nunarası olabilir.) Arama ağaçları şekilsel olarak betimlendiğinde genellikle düğüm üzerinde yalnızca anahtar gösterilir. Ağaçlarda küçük olan anahtarlar düzenli olarak sol ya da sağ taraf alt düğümünde bulundurulursa bu tür ağaçlar arama ağacı haline gelir. Tipik olarak sol alt düğüme küçük anahtarlar sağ alt düğüme büyük anahtarlar yerleştirilir. Ancak bunun tersi de söz konusu olabilir. Örneğin aşağıdaki sayılar sırasıyla sisteme gelmiş olsun ve biz bundan ikili arama ağacı oluşturmak isteyelim:



**İkili Arama Ağaçları**

İkili arama ağaçları (binary search tree) en çok kullanılan arama ağaçlarıdır. Burada her düğümün en fazla iki alt düğümü olabilir. Küçük değerler sol alt düğüme büyük değerler sağ alt düğüme yerleştirilir. Arama yapılırken kökten girilir. Aranacak değer o andaki düğümün değerinden küçükse sola, büyükse sağa sapılarak ilerlenir. Eleman eklenirken de sola, sağa gidilerek uygun yaprak pozisyonuna gelinir ekleme araya yapılır.

İkili arama ağaçlarında arama yapılırken kök düğümden girilir. Duruma göre sola ve sağa sapılarak eleman bulunmaya çalışılır. Örnek bir arama işlemi şöyle yapılabilir:

DATATYPE \*FindItem(HBINARYTREEhBinaryTree, KEYkey)

{

NODE \*pNode;

pNode = hBinaryTree->pRoot;

while (pNode != NULL) {

if (key> pNode->val.no)

pNode = pNode->pRight;

elseif (key< pNode->val.no)

pNode = pNode->pLeft;

else

break;

}

return pNode ? &pNode->val : NULL;

}

Eleman eklerken önce ekleme yerinin bulunması gerekir. Bunun için yine kök düğümden girilir, duruma göre sola ve sağa gidilerek eklenecek düğüm tespit edilir ve ekleme yapılır. Örneğin:

BOOL InsertItem(HBINARYTREEhBinaryTree, constDATATYPE \*pVal)

{

NODE \*pNewNode, \*pNode, \*pParentNode;

if ((pNewNode = (NODE \*)malloc(sizeof(NODE))) == NULL)

returnFALSE;

pNewNode->val = \*pVal;

pNewNode->pLeft = pNewNode->pRight = NULL;

if (hBinaryTree->pRoot == NULL) {

hBinaryTree->pRoot = pNewNode;

++hBinaryTree->count;

returnTRUE;

}

pParentNode = pNode = hBinaryTree->pRoot;

while (pNode != NULL) {

pParentNode = pNode;

if (pVal->no > pNode->val.no)

pNode = pNode->pRight;

elseif (pVal->no < pNode->val.no)

pNode = pNode->pLeft;

else

returnFALSE;

}

if (pVal->no > pParentNode->val.no)

pParentNode->pRight = pNewNode;

else

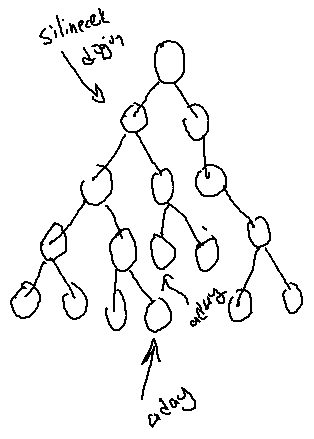
pParentNode->pLeft = pNewNode;

returnTRUE;

}

Ekleme işlemi özyinelemeli olarak da yapılabilir. Bunun için kök düğümden girilir. Eklenecek sol ya da sağ yön tespit edilir. Eğer orada NULL gösterici varsa oraya ekleme yapılır. Yoksa o düğümle fonksiyon kendisini çağırır.

İkili ağaçtan eleman silmek biraz daha zahmetlidir. Çünkü silinecek elemanın yerine ağaçtaki başka elemanın taşınması gerekir. Basit bir fikir şöyle olabilir: Silinecek düğüm onun solundakilerden büyük sağındakilerden küçük olduğuna göre onun yerine geçecek düğüm onun solundakilerin en büyüğü ya da sağındakilerin en küçüğü olabilir. Örneğin:



Silme işleminde 4 özel durum söz konusudur:

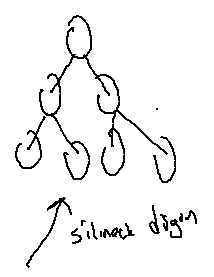
1) Silinecek düğümün hiç alt düğümü yoktur.

2) Silienecek düğümün yalnızca bir alt düğümü vardır.

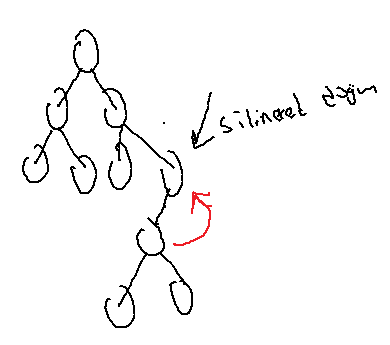
3) Silinecek düğümün iki alt düğümü de vardır.

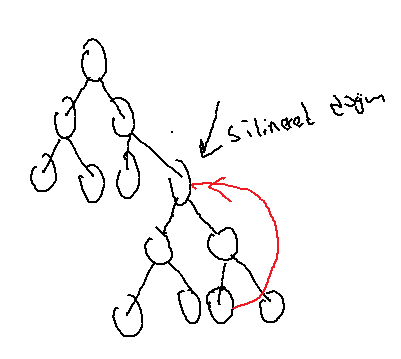
4) Silinecek düğüm köktür.

Silinecek düğümün hiç alt düğümü yoksa düğüm doğrudan silinir tabi onun üst düğümünün uygun linki NULL yapılır.



Silinecek düğümün yalnızca bir alt düğümü varsa diğer al düğüm silinecek düümün yerine geçer.



Eğer silinecek düğümün iki alt düğümü de varsa bu durumda onun sağındakinin en küçüğü ya da solundakinin en büyüğü onun yerine geçer.

Eğer silinecek düğüm kökse handle alanında kökü güncellemek gerekir. Bu özel bir durum oluşturabilmektedir. Tabi handle alanında biz kök düğümün adresini değil onu bir düğüm olarak (dummy düğüm olarak) bu özel durumdan kurtulabiliriz.

BOOL DeleteItem(HBINARYTREE hBinaryTree, KEY key)

{

NODE \*pRemoveNode, \*pRemoveParentNode, \*pReplaceNode;

pRemoveNode = hBinaryTree->pRoot;

while (pRemoveNode != NULL) {

if (pRemoveNode->val.no == key)

break;

pRemoveParentNode = pRemoveNode;

if (pRemoveNode->val.no > key)

pRemoveNode = pRemoveNode->pLeft;

elseif (pRemoveNode->val.no < key)

pRemoveNode = pRemoveNode->pRight;

}

if (pRemoveNode == NULL)

return FALSE;

pReplaceNode = findReplaceNode(pRemoveNode);

if (pReplaceNode != NULL) {

pReplaceNode->pLeft = pRemoveNode->pLeft;

pReplaceNode->pRight = pRemoveNode->pRight;

}

if (pRemoveNode == hBinaryTree->pRoot)

hBinaryTree->pRoot = pReplaceNode;

elseif (pRemoveParentNode->pLeft == pRemoveNode)

pRemoveParentNode->pLeft = pReplaceNode;

else

pRemoveParentNode->pRight = pReplaceNode;

free(pRemoveNode);

--hBinaryTree->count;

return TRUE;

}

static NODE \*findReplaceNode(NODE \*pRemoveNode)

{

NODE \*pNode, \*pParentNode;

if (pRemoveNode->pRight == NULL)

return pRemoveNode->pLeft;

pParentNode = pRemoveNode;

pNode = pRemoveNode->pRight;

while (pNode->pLeft != NULL) {

pParentNode = pNode;

pNode = pNode->pLeft;

}

if (pParentNode->pLeft == pNode)

pParentNode->pLeft = pNode->pRight;

else

pParentNode->pRight = pNode->pRight;

return pNode;

}

İkili ağacın dolaşılması (traverse / walk) üç farklı yolla yapılabilir: Inorder, Preorder ve Postorder dolaşımlar. Bu isimler üst düğümün alt düğümlere göre hangi sırada ziyaret edileceğini belirtir. Inorder dolaşımda önce alt düğümler sonra üst düğüm dolaşılır (önce soldaki alt düğüm ya da sağdaki alt düğüm ziyaret edilebilir). Inorder dolaşımda önce alt düğümlerden birisi sonra üst düğüm sonra da diğer alt düğüm ziyaret edilir. Postorder dolaşımda önce alt düğümler sonra üst düğüm ziyaret edilir. Bu dolaşımların her biri bazı tipik durumlarda tercih edilmektedir. Örneğin ikili ağacı sıralı dolaşmak istersek inorder dolaşım yapmalıyız. Inorder dolaşımda sol-üst-sağ küçükten büyüğe, sağ-üst-sol büyükten küçüğe dolaşıma yol açmaktadır. Şüphesiz bu üç durum da özyinelemeli bir fonksiyonla gerçekleştirilebilir. Örneğin:

BOOL WalkInOrder(HBINARYTREEhBinaryTree, BOOL(\*Proc)(constDATATYPE \*))

{

return walkInOrder(hBinaryTree->pRoot, Proc);

}

staticBOOL walkInOrder(NODE \*pNode, BOOL(\*Proc)(constDATATYPE \*))

{

if (pNode->pLeft != NULL&& !walkInOrder(pNode->pLeft, Proc))

returnFALSE;

if (!Proc(&pNode->val))

returnFALSE;

if (pNode->pRight!= NULL&& !walkInOrder(pNode->pRight, Proc))

returnFALSE;

returnTRUE;

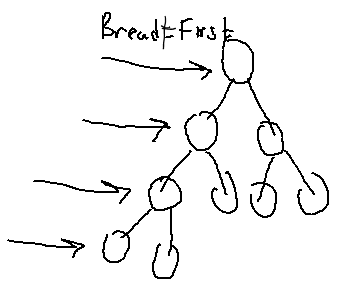
}

Preorder dolaşım ağacı ağaç gibi görüntülemekte kullanılabilir. Posr order dolaşım tipik olarak ağacı silmek için uygulanır. (Yani önce alt düğümleri silip çıkarken üst düğümü silmek gerekir.)

Kod eklenecek

**İkili Ağaçlarda Depth-First ve Breadth-First Dolaşımlar**

Ağaçlarda arama yaparken eğer ağaç bir arama ağacı değilse her düğüme bakmak gerekir. Aslında bu durum genel olarak graflar için de söylenebilir. Böylece ağaçları dolaşırken iki strateji izlenebilmektedir. Depth-First dolaşımda her zaman dibe dalınarak özyinelemeli bir gidiş vardır. Yukarıda inceledeğimiz Inorder, postorder ve preorder dolaşımlar kategori olarak depth-first dolaşım türündendir. Halbuki bread-first dolaşımda ağaç (ya da genel olarak graf) kademeli dolaşılır.

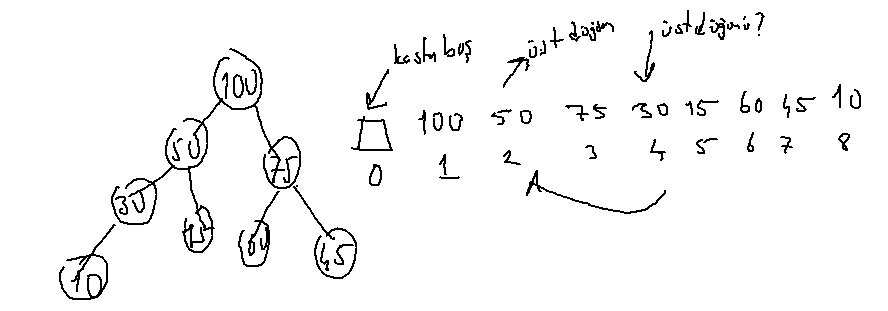


Breadth-First dolaşım için bir kuyruk sistemi alınır. Kök düğüm kuyruğa atılarak işe başlanır. Sonra kuyruktan çekilirken onun alt düğümleri kuyruğa atılır. Bu biçimde ilerlenerek gidilir. Bazı durumlarda bread-first dolaşım istenebilir. Örneğin hiyerarşik bir organizasyonda üst yönetimden alt tarafa doğru kişileri dolaşmak isteyebiliriz.

**Heap Veri Yapısı (Ağaçları)**

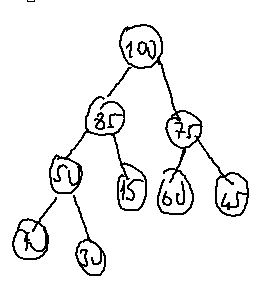
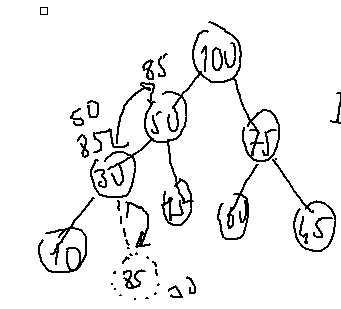
Heap özel bir ağaç türüdür. Öyle ki heap ağacında her zaman üst düğümün anahtarı alt düğümlerinin anahtarlarından ya daha büyüktür (max heap) ya da daha küçüktür (min heap). Ve ağaçları tam (complete) ağaçlardır. Bunun dışında heap ağacının başka bir koşulu yoktur. Heap ağaçları da n'li olabilir. Ancak en çok kullanılan heap ağaçları ikili heap ağaçlarıdır (binary heap).

Heap ağaçları aslında ağaç olarak gerçekleştirmek için tasarlanmamıştır dizi olarak gerçekleştirilmek için tasarlanmıştır. Şöyle ki biz bir heap ağacını bir diziye çevirebiliriz (buna terminoloji de heapify denilmektedir). Heap ağacını dizi olarak oluşturduğumuzda bir düğümün üst düğümü onun indeks olarak tam yarısında bulunur. Örneğin:



Uygulamada dizinin ilk elemanı hesaplamalar kolay yapılsın diye boş bırakılır (boş bırakılmasa da olur ancak hesaplamaların daha kolay yapılması için bu tercih edilmektedir.) Görüldüğü gibi heap ağacı diziye dönüştürüldüğünde bir elemanın üst düğümü onun indeksinin yarısı indeksinde olur.

Pekiyi kuralı bozmadan heap ağacını temsil eden diziye nasıl ekleme yapılabilir? Eleman önce her zaman sona yerleştirilir. Sonra bir döngü içerisinde üst düğümlere erişilerek onunla üst düğüm arasındaki ilişkiye bakılır. Kuralın sağlandığı noktada durulur, kural sağlanmıyorsa onunla üst düğümü yer değiştirilir. Örneğin yukarıdaki ağaca biz 85 ekleyelim:



Heap ağacından eleman silmek için bunun tam tersi yapılır. Ağacın tepesindeki eleman alınır. Onun yerine geçecek eleman onun alt düğümlerinden biridir. Bunlardan hangisi büyükse o yukarı alınır. Böyle devam edilir. Ta ki ağacın yapraklarına gelene kadar.

Heap ağacı her zaman tam (complete) bir ağaç görünümündedir. Zaten o nedenle dizi olarak temsil edilmesi uygundur. Tabi buradaki dizinin dinamik bir dizi gibi olması daha uygundur.

**Heap Ağacına Neden Gereksinim Duylmaktadır?**

Heap ağacının en önemli uygulama alanı öncelik kuyruklarıdır (priority queues). Öncelik kuyruklarında kuyrukta birtakım elemanlar vardır. Kuyruktan eleman alınacağı zaman en yüksek önceliğe sahip eleman alınır. Böyle bir kuyruk sistemi düz bir diziyle gerçekleştirilmeye çalışılırsa her defasında en büyük elemanın bulunması için dizinin dolaşılması gerekir. İşte bunun için en uygun veri yapısı ikili heap ağaçlarıdır. Eleman dizinin sonuna eklenir. Dizi heap kuralını bozmayacak biçimde düzenlenir. Eleman alınacağı zaman baştan alınır yine dizi kuralı bozmayacak biçimde düzenlenir.

Heap ağaçları sıraya dizme için de kullanılmaktadır. Buna heap sort denilmektedir. Heap sort da O(N Log N) karmaşıklıkta kaliteli bir sort algoritmesıdır.

**Heap Veri Yapısının Gerçekleştirilmesi**

Heap veri yapısı için bir dizi kullanılır. Tabi dizinin dinamik bir dizi olması tercih edilir. Heap'a eleman insert edilirken insert edilecek yer en aşağıdan hareketle yukarıya doğru çıkılarak yer değiştirmelerle belirlenir. Heap'in kökündeki elemanın alınması benzer biçimde yapılır. Kök elemanın yerine geçecek eleman aşağıya doğru inilerek belirlenir. Örnek bir heap gerçekleştirimi şöyle olabilir:

/\* HeapTree.h \*/

#ifndef HEAP\_H\_

#defineHEAP\_H\_

#include<stddef.h>

/\* Symbolic Constants \*/

#defineFALSE 0

#defineTRUE 1

#defineDEF\_CAPACITY 8

/\* Type declarations \*/

typedefintDATATYPE;

typedefintBOOL;

typedefstructtagHEAP {

DATATYPE \*pArray;

size\_t capacity;

size\_t lastIndex;

} HEAP, \*HHEAP;

/\* Function Prototypes \*/

HHEAP CreateHeap(void);

BOOL PutHeap(HHEAP hHeap, DATATYPE val);

BOOL GetHeap(HHEAP hHeap, DATATYPE \*val);

void ClearHeap(HHEAP hHeap);

void CloseHeap(HHEAP hHeap);

/\* Macros \*/

#defineIsEmptyHeap(hHeap) ((hHeap)->lastIndex == 1)

#defineGetSizeHeap(hHeap) ((hHeap)->size)

#defineGetCountHeap(hHeap) ((hHeap)->lastIndex - 1)

#defineGetHeapArray(hHeap) ((hHeap)->pArray + 1)

#endif

/\* HeapTree.c \*/

#include<stdlib.h>

#include"HeapTree.h"

HHEAP CreateHeap(void)

{

HHEAP hHeap;

if ((hHeap = (HHEAP)malloc(sizeof(HEAP))) == NULL)

returnNULL;

if ((hHeap->pArray = (DATATYPE \*)malloc(sizeof(DATATYPE)\* DEF\_CAPACITY)) == NULL) {

free(hHeap);

returnNULL;

}

hHeap->capacity = DEF\_CAPACITY;

hHeap->lastIndex = 1;

return hHeap;

}

BOOL PutHeap(HHEAPhHeap, DATATYPEval)

{

DATATYPE \*pTemp;

size\_t i;

if (hHeap->lastIndex == hHeap->capacity) {

if ((pTemp = (DATATYPE \*)realloc(hHeap->pArray, sizeof(DATATYPE) \* hHeap->capacity \* 2)) == NULL)

returnFALSE;

hHeap->pArray = pTemp;

hHeap->capacity \*= 2;

}

i = hHeap->lastIndex;

if (i != 1)

while (val>hHeap->pArray[i / 2] && i > 1) {

hHeap->pArray[i] = hHeap->pArray[i / 2];

i /= 2;

}

hHeap->pArray[i] = val;

++hHeap->lastIndex;

returnTRUE;

}

BOOL GetHeap(HHEAPhHeap, DATATYPE \*val)

{

size\_t i, ci;

DATATYPE lastVal;

if (hHeap->lastIndex == 1)

returnFALSE;

\*val = hHeap->pArray[1];

lastVal = hHeap->pArray[hHeap->lastIndex - 1];

--hHeap->lastIndex;

i = 1;

ci = 2;

while (ci <hHeap->lastIndex) {

if (ci + 1 <hHeap->lastIndex &&hHeap->pArray[ci] <hHeap->pArray[ci + 1])

++ci;

if (lastVal >hHeap->pArray[ci])

break;

hHeap->pArray[i] = hHeap->pArray[ci];

i = ci;

ci \*= 2;

}

hHeap->pArray[i] = lastVal;

returnTRUE;

}

void ClearHeap(HHEAPhHeap)

{

hHeap->lastIndex = 1;

}

void CloseHeap(HHEAPhHeap)

{

free(hHeap->pArray);

free(hHeap);

}

/\* Test.c \*/

#include<stdio.h>

#include<stdlib.h>

#include"HeapTree.h"

int main(void)

{

HHEAP hHeap;

size\_t i;

DATATYPE \*pArray;

DATATYPE val;

if ((hHeap = CreateHeap()) == NULL) {

fprintf(stderr, "cannot create heap!..\n");

exit(EXIT\_FAILURE);

}

PutHeap(hHeap, 100);

PutHeap(hHeap, 50);

PutHeap(hHeap, 75);

PutHeap(hHeap, 30);

PutHeap(hHeap, 15);

PutHeap(hHeap, 60);

PutHeap(hHeap, 45);

PutHeap(hHeap, 10);

pArray = GetHeapArray(hHeap);

for (i = 0; i <GetCountHeap(hHeap); ++i)

printf("%d ", pArray[i]);

printf("\n-------------------------------\n");

while (!IsEmptyHeap(hHeap)) {

if (!GetHeap(hHeap, &val)) {

fprintf(stderr, "heap is empty!\n");

exit(EXIT\_FAILURE);

}

printf("Value: %d\n", val);

pArray = GetHeapArray(hHeap);

for (i = 0; i <GetCountHeap(hHeap); ++i)

printf("%d ", pArray[i]);

printf("\n-------------------------------\n");

}

CloseHeap(hHeap);

return 0;

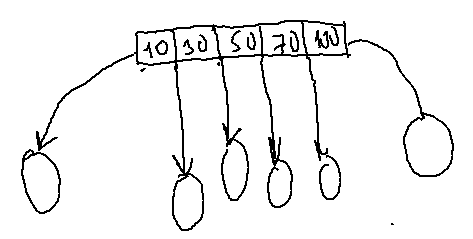
}

**Dışsal Aramalar (External Search)**

Daha önceden de belirtildiği gibi dışsal arama (external search) ikincil belleklerde (yani dosyalarda) yapılan aramaları belirtmektedir. Aslında dışsal aramalar için de benzer algoritmalar söz konusudur. Örneğin ikili arama ağacını dosya üzerindeki kayıtlar için oluşturabilir miyiz? Tabiki evet. Şöyle ki: Dosyanın başına bir başlık kısmı (file header) yerleştiririz. Sonra eklenecek kayıtları hep dosyanın sonuna ekleriz. İkili ağaç düğümlerindeki left ve right göstericiler burada offset'lere karşılık gelir. Böylece kök düğümden gireriz. fseek fonksiyonuyla konumlanarak okumaları yapabiliriz.

Ancak ikili ağaçlar dengelenmiş olsa bile disk aramaları için çok uygun değildir. Çünkü ağacın yüksekliği disk okumalarının sayısını doğrudan artırmaktadır. Her düğüm geçişinde bir disk okuması yapılır ki, disk okumaları göreli olarak yavaştır. Bu durumda biz ağacın yüksekliğini düşürmek isteriz. İşte B Tree denilen ağaç yapısı bunu hedeflemektedir.

B Tree Rudolf Bayer tarafından 1971 yılında bulunmuştur. İsminin neden “B Tree” olduğuna yönelik kesin bir bilgi yoktur. B Tree aslında n'li bir arama ağacıdır Şöyle ki her düğümün iki alt düğümü değil daha fazla düğümü vardır. Örneğin B Tree'nin 6'lı arama ağacı olarak oluşturulduğunu varsayalım. Burada her düğümde tgek bir anahtar değil 5 anahtar bulunacaktır. Tabi bu 5 anahtar bir dizi içerisinde tutulur:



10'dan küçük olanlar, 10 ile 30 arasında olanlar, 30 ile 50 arasında olanlar, 50 ile 70 arasında olanlar ve 100'den büyük olanlar ayrı düğümlere yönlendirilmiştir. Böylece arama yapılırken daha az düzey geçişi oluşur. Bu da disk okulamalarının daha az olması anlamına gelir. Çünkü her düğümden düğüme geçiş bir disk okumasıyla yapılmaktadır. B Tree dengeli bir ağaç yapısıdır. B Tree oluşturulması ve dengelenmesi biraz karmaşıktır.

Pekiyi birincil belleklerde B Tree mi yoksa Binary Tree mi daha etkindir? Aslında birincil bellekler için Binary Tree daha etkindir. Çünkü B Tree'de düğümler arasındaki geçiş sırasında anahtar karşılaştırmaları daha fazladır. Ve genel olarak yapılan işlemler daha yoğundur. Oysa Binary Tree'de hızlı biçimde düğümden düğüme geçiş sağlanır. Disk söz konusu olduğunda asıl yavaşlık düğümler arası geçişte oluştuğu için B Tree çok daha uygun bir yöntem olmaktadır. B Tree'ni biraz revize edilmiş biçimine B+ Tree denilmektedir.

Bugün kullandığımız Sql Server gibi, MySql gibi, Orackle gibi Veritabanı Yönetim Sistemlerinin çekirdek arama algoritmaları hep B Tree ya da B+ Tree'ye dayanmaktadır. Şimdilik bunun ötesinde disk için daha verimli algoritmalar bulunmamıştır.

**Veri Yapılarının Genelleştirilmesi**

Yukarıdaki örneklerde biz veriyapılarının içerisinde tutulan türü DATATYPE ile temsil ettik. Bunun birkaç sakıncası vardır. Birincisi bu yolla aynı programda DATATYPE değiştirilemez. Yani örneğin DATATYPE int ise biz yalnızca int türden bağlı listeler oluşturabiliriz. C++ gibi, Java, C# gibi dillerdeki template (ya da generic) özelliği bu sorunu bir ölçüde çözmektedir. İkinci sorun fonksiyonlara parametre aktarım işleminin değer yoluyla mı adres yoluyla mı yapılacağıdır. Örneğin DATATYPE int ise biz onu fonksiyonlara doğrudan geçirmeyi tercih ederiz. Ancak DATATYPE bir yapı ise biz onu adres yoluyla geçirmeyi tercih ederiz.

Genelleştirme için üç yol izlenebilir:

1) Her tür için o veri yapısını yeniden kodlamak. Örneğin IntLinkedList, PersonLinkedList gibi. Şüphesiz bu yöntem kodu büyütme eğilimindedir. Ancak böyle oluşturulmuş veri yapılarının hem kullanımı kolay hem de diğerlerine göre göreli olarak daha hızlıdır. Zaten template ya da generic mekanizma aslında arka planda böyle bir işleme yol açar. Örneğin:

list<int> a;

list<Person> b;

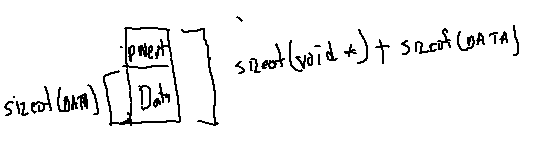
Burada list template bir sınıftır. Kendisi şablon bir bildirimdir. Derleyici her tür için onu yeniden yazar.

2) Bu yöntemde tek bir fonksiyon grubu tüm türler için çalışabilir biçimde oluşturulur. Bu yöntemde kod tekrarı olmadığı için toplamda kodlar daha az yer kaplar. Ancak bu yöntemde fonksiyonların parametrik yapıları dahakarmaşıktır ve genel kullanım daha zordur.

3) Bu üçüncü yöntemde veri yapısı türden bağımsız olarak yalnızca metadata kısımlarla ilişkili biçimde oluşturulur. Örneğin Linux'ta BSD'de CSD'de genelleştirme için bu yöntem kullanılmıştır.i Bu yöntemin de kullanımı zordur. Ancak kod etkinliği yüksektir.

**Veri Yapılarının Türden Bağımsız Olarak Genel Fonksiyonlarla Genelleştirilmesi (2. Yöntem)**

Teorik olarak yukarıda sözünü ettiğimiz veri yapılarını oluşturmak için aslında biz DATATYPE türünün ne olduğunu derleme zamanında bilmek zorunda değiliz. O türün byte uzunluğunu bilsek bu bize yeter. Şöyle ki: Örneğin bağlı liste düğümlerinde saklanan elemanların türlerini bilmiyor olalım. Bu durumda bir düğümün biçimi biz göre şyle olacaktıır:



Tabi bazı durumlarda bir karşılaştırma işlemi de gerekir. Örneğin ikili ağacı bu biçimde oluşturmak istediğimizde eleman eklerken neye göre karşılaştırma yapacağız? Bu tür durumlarda fonksiyon göztericileriyle karşılaştırma fonksiyonu yoluyla karşılaştırmayı aslında veri yapısını kullanan kişiye bırakabiliriz. Şüphesiz böyle bir sistemde artık fonksiyonların elemana ilişkin parametreleri void \* türünden olacaktır. Eleman atamaları da atama operatörüyle değil memcpy gibi bir fonksiyonla yapılacaktır.

Türden bağımsız örnek bir ikili arama ağacı şöyle oluşturulabilir.

/\* BinaryTree.h \*/

#ifndef BINARYTREE\_H\_

#defineBINARYTREE\_H\_

#include<stddef.h>

/\* Symbolic Constants \*/

#defineFALSE 0

#defineTRUE 1

/\* Type declarations \*/

typedefintKEY;

typedefstructtagPERSON {

KEY no;

char name[32];

} PERSON;

typedefintBOOL;

typedefint (\*Compare)(constvoid \*, constvoid \*);

typedefstructtagNODE {

structtagNODE \*pLeft;

structtagNODE \*pRight;

unsignedchar data[1];

} NODE;

typedefstructtagBINARYTREE {

NODE \*pRoot;

size\_t count;

size\_t dataSize;

Compare compare;

} BINARYTREE, \*HBINARYTREE;

/\* Function Prototypes \*/

HBINARYTREE CreateBinaryTree(size\_t dataSize, Compare compare);

BOOL InsertItem(HBINARYTREE hBinaryTree, constvoid \*val);

BOOL WalkInOrder(HBINARYTREE hBinaryTree, BOOL(\*Proc)(constvoid \*));

constvoid \*FindItem(HBINARYTREE hBinaryTree, constvoid \*key, int(\*Compare)(constvoid \*, constvoid \*));

#endif

/\* BinaryTree.c \*/

#include<stdio.h>

#include<stdlib.h>

#include<string.h>

#include"BinaryTree.h"

/\* static Function Prototypes \*/

staticBOOL walkInOrder(NODE \*pNode, BOOL(\*Proc)(constvoid \*));

staticNODE \*createNode(HBINARYTREE hBinaryTree, constvoid \*val);

staticNODE \*deleteNode(NODE \*pRemoveNode);

/\* Function Definitions \*/

HBINARYTREE CreateBinaryTree(size\_tdataSize, Comparecompare)

{

HBINARYTREE hBinaryTree;

if ((hBinaryTree = (HBINARYTREE)malloc(sizeof(BINARYTREE))) == NULL)

returnNULL;

hBinaryTree->pRoot = NULL;

hBinaryTree->count = 0;

hBinaryTree->dataSize = dataSize;

hBinaryTree->compare = compare;

return hBinaryTree;

}

staticNODE \*createNode(HBINARYTREEhBinaryTree, constvoid \*val)

{

NODE \*pNewNode;

if ((pNewNode = (NODE \*)malloc(sizeof(NODE \*) \* 2 + hBinaryTree->dataSize)) == NULL)

returnNULL;

memcpy(pNewNode->data, val, hBinaryTree->dataSize);

pNewNode->pLeft = pNewNode->pRight = NULL;

return pNewNode;

}

BOOL InsertItem(HBINARYTREEhBinaryTree, constvoid \*val)

{

NODE \*pNewNode, \*pNode, \*pParentNode;

int compResult;

if ((pNewNode = createNode(hBinaryTree, val)) == NULL)

returnFALSE;

if (hBinaryTree->pRoot == NULL) {

hBinaryTree->pRoot = pNewNode;

++hBinaryTree->count;

returnTRUE;

}

pParentNode = pNode = hBinaryTree->pRoot;

while (pNode != NULL) {

pParentNode = pNode;

compResult = hBinaryTree->compare(val, pNode->data);

if (compResult > 0)

pNode = pNode->pRight;

elseif (compResult < 0)

pNode = pNode->pLeft;

else

returnFALSE;

}

if (compResult > 0)

pParentNode->pRight = pNewNode;

else

pParentNode->pLeft = pNewNode;

returnTRUE;

}

BOOL WalkInOrder(HBINARYTREEhBinaryTree, BOOL(\*Proc)(constvoid \*))

{

return walkInOrder(hBinaryTree->pRoot, Proc);

}

staticBOOL walkInOrder(NODE \*pNode, BOOL(\*Proc)(constvoid \*))

{

if (pNode->pLeft != NULL&& !walkInOrder(pNode->pLeft, Proc))

returnFALSE;

if (!Proc(pNode->data))

returnFALSE;

if (pNode->pRight!= NULL&& !walkInOrder(pNode->pRight, Proc))

returnFALSE;

returnTRUE;

}

constvoid \*FindItem(HBINARYTREEhBinaryTree, constvoid \*key, int (\*Compare)(constvoid \*, constvoid \*))

{

NODE \*pNode;

int compResult;

pNode = hBinaryTree->pRoot;

while (pNode != NULL) {

compResult = Compare(key, pNode->data);

if (compResult > 0)

pNode = pNode->pRight;

elseif (compResult < 0)

pNode = pNode->pLeft;

else

break;

}

return pNode ? pNode->data : NULL;

}

/\* Test.c \*/

#include<stdio.h>

#include<stdlib.h>

#include<time.h>

#include"BinaryTree.h"

BOOL Disp(constvoid \*val)

{

constPERSON \*per = (constPERSON \*)val;

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

returnTRUE;

}

void GetRandomPerson(PERSON \*per)

{

int i;

for (i = 0; i < 31; ++i)

per->name[i] = 'A' + rand() % 26;

per->name[i] = '\0';

per->no = rand() % 1000;

}

int ComparePerson(constvoid \*pv1, constvoid \*pv2)

{

constPERSON \*p1 = (constPERSON \*)pv1;

constPERSON \*p2 = (constPERSON \*)pv2;

if (p1->no > p2->no)

return 1;

if (p1->no < p2->no)

return -1;

return 0;

}

int ComparePersonKey(constvoid \*pv1, constvoid \*pv2)

{

constint \*p1 = (constint \*)pv1;

constint \*p2 = (constint \*)pv2;

if (\*p1 > \*p2)

return 1;

if (\*p1 < \*p2)

return -1;

return 0;

}

int main(void)

{

HBINARYTREE hBinaryTree;

PERSON persons[] = {

{ 15, "Kaan Aslan"}, {10, "Ali Serce"}, {30, "Sacit Hicyilmaz"}, {8, "Ali Sen"},

{12, "Fatih Terim"}, {20, "Necati Ergin"}, {52, "Guray Sonmez"}, {13, "Sami Ercan"}

};

int i;

int findKey = 20;

PERSON \*per;

srand((unsigned)time(NULL));

if ((hBinaryTree = CreateBinaryTree(sizeof(PERSON), ComparePerson)) == NULL) {

fprintf(stderr, "cannot create binary tree!\n");

exit(EXIT\_FAILURE);

}

for (i = 0; i <sizeof(persons) / sizeof(\*persons); ++i)

if (!InsertItem(hBinaryTree, &persons[i])) {

fprintf(stderr, "cannot insert item!..\n");

exit(EXIT\_FAILURE);

}

WalkInOrder(hBinaryTree, Disp);

printf("------------------\n");

if ((per = FindItem(hBinaryTree, &findKey, ComparePersonKey)))

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

else

printf("cannot find record!\n");

return 0;

}

Türden bağımsız veri yapısı oluşturmanın diğer bir yolu da veri yapısının metadata kısımlarını asıl veri yapısının içerisine gömmektir. Örneğin Linux, BSD, CSD kernel kodlarındaki veri yapıları genel olarak bu biçimde oluşturulmuştur.

Bu yöntemle örneğin bir çift bağlı liste oluşturacak olalım. Bu çift bağlı listenin pNext ve pPrev göstericileri sonraki düğümün başlangıç adresini göstermezler. Sonraki düğümdeki pNext ve pPrev'in bulunduğu metadata alanının başlangıç adresini gösterirler. Örneğin biz bu çift bağlı listenin iki göstericisini aşağıdaki gibi bir yapıyla temsil etmiş olalım:

typedef struct tagNODE {

struct tagNODE \*pNext;

struct tagNODE \*pPrev;

} NODE;

Şimdi biz PERSON yapısından bir bağlı liste oluşturmak isteyelim:

typedef struct tagPERSON {

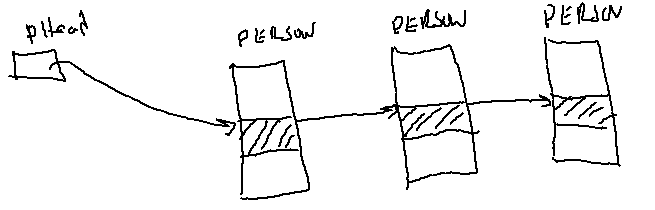
char name[64];

NODE node; /\* yapının herhangi bir yerinde olabilir \*/

int no;

} PERSON;

Bu durum şekilsel olarak şöyle gösterilebilir:



Tabii burada elemanın ismini bildiğimiz bir yapı nesnesinin başlangıç adresinin bulunabilmesi gerekir. Bu oldukça basittir. O elemandan gerektiği kadar geriye giderek yapı nesnesinin adresi elde edilebilir. Zaten C'de bu işi yapan standart offsetof makrosu vardır. Bu makro aşağıdaki gibi yazılabilir:

#define offsetof(st, m) ((size\_t)(&((st \*)0)->m))

#define offsetof(st, m) \_\_builtin\_offsetof(st, m)

#define offsetof(st, m) \_\_builtin\_offsetof(st, m)

#define offsetof(st, m) ((size\_t)(&((st \*)0)->m))

Bu makronun birinci parametresi yapının ismini (yani tür ifadesini), ikinci parametresi elemanın ismini alır. Örneğin:

size\_t offset = offsetof(PERSON, node)

Yukarıdaki makro C standartlarına göre tamamen geçerlidir. Standartlara göre p göstericisi tahsis edilmemiş bir alanı gösteriyor olsa bile (0 adresi de dahil olmak üzere) onun bir elemanın adresini &p->a biçiminde elde etmek oraya gerçek bir erişimi gerektirmediği için geçerlidir.

Tabi yukarıdaki bağlı liste örneğinde bizim node adresinden offsetof kadar eksiltme yapıp sonucu ilgili türe dönüştürmemiz gerekir. Bu da bir makro ile yapılabilir. Örneğin:

#define container\_of(ptr, type, member) ((type \*)( (char \*)(ptr) – offsetof(type, member))

Örneğin PERSON yapılarından bir bağlı liste oluşturmuş olalım. Bu bağlı listenin dolaşılması şöyle yapılabilir:

NODE pHead;

PERSON \*per;

NODE \*pNode = pHead;

while (pNode != NULL) {

per = container\_of(pNode, PERSON, node)

/\* ... \*/

pNode = pNode->pNext;

}

Örnek olarak şu şekilde gerçekleştirilebilir:

/\* GenericLinkedList.h \*/

#ifndef GENERICLINKEDLIST\_H\_

#defineGENERICLINKEDLIST\_H\_

#include<stddef.h>

/\* Symbolic Constants \*/

#defineFALSE 0

#defineTRUE 1

/\* Type declarations \*/

typedefintBOOL;

typedefstructtagNODE {

structtagNODE \*pNext;

structtagNODE \*pPrev;

} NODE;

typedefstructtagPERSON {

int no;

char name[32];

NODE node;

} PERSON;

/\* Function Prototypes \*/

NODE \*AddItemHead(NODE \*pHead, NODE \*pNewNode);

NODE \*AddItemTail(NODE \*pHead, NODE \*pNewNode);

BOOL WalkList(NODE \*pHead, BOOL(\*Proc)(NODE \*));

/\* Macros \*/

//#define container\_of(ptr, type, member) ( (type \*)( (char \*)(ptr) – offsetof(type, member)))

#endif

/\* GenericLinkedList.c \*/

#include<stdio.h>

#include"GenericLinkedList.h"

NODE \*AddItemHead(NODE \*pHead, NODE \*pNewNode)

{

pNewNode->pPrev = pHead;

pNewNode->pNext = pHead->pNext;

pHead->pNext = pNewNode;

returnpNewNode;

}

NODE \*AddItemTail(NODE \*pHead, NODE \*pNewNode)

{

pNewNode->pNext = pHead;

pNewNode->pPrev = pHead->pPrev;

pHead->pPrev = pNewNode;

returnpNewNode;

}

BOOL WalkList(NODE \*pHead, BOOL(\*Proc)(NODE \*))

{

NODE \*pNode = pHead->pNext;

while (pNode != pHead) {

if (!Proc(pNode))

returnFALSE;

pNode = pNode->pNext;

}

returnTRUE;

}

/\* Test.c \*/

#include<stdio.h>

#include"GenericLinkedList.h"

BOOL WalkProc(NODE \*node);

NODE g\_head = { &g\_head, &g\_head };

int main(void)

{

PERSON persons[] = {

{ 15, "Kaan Aslan" }, { 10, "Ali Serce" }, { 30, "Sacit Hicyilmaz" }, { 8, "Ali Sen" },

{ 12, "Fatih Terim" }, { 20, "Necati Ergin" }, { 52, "Guray Sonmez" }, { 13, "Sami Ercan" }

};

AddItemHead(&g\_head, &persons[0].node);

AddItemHead(&g\_head, &persons[1].node);

AddItemHead(&g\_head, &persons[2].node);

AddItemHead(&g\_head, &persons[3].node);

WalkList(&g\_head, WalkProc);

}

BOOL WalkProc(NODE \*pNode)

{

PERSON \*pPerson;

pPerson = (PERSON \*)((char \*)pNode - offsetof(PERSON, node));

printf("%d, %s\n", pPerson->no, pPerson->name);

returnTRUE;

}

Ya da handle sistemi kullanılarak aynı gerçekleştirim benzer biçimde şöyle de yapılabilir:

/\* GenericLinkedList.h \*/

#ifndef GENERICLINKEDLIST\_H\_

#defineGENERICLINKEDLIST\_H\_

#include<stddef.h>

/\* Symbolic Constants \*/

#defineFALSE 0

#defineTRUE 1

/\* Type declarations \*/

typedefintBOOL;

typedefstructtagNODE {

structtagNODE \*pNext;

structtagNODE \*pPrev;

} NODE;

typedefstructtagLLIST {

NODE head;

size\_t count;

} LLIST, \*HLLIST;

typedefstructtagPERSON {

int no;

char name[32];

NODE node;

} PERSON;

/\* Function Prototypes \*/

HLLIST CreateLList(void);

NODE \*InsertItemAfter(HLLIST hLList, NODE \*pInsertNode, NODE \*pNewNode);

NODE \*AddItemHead(HLLIST hLList, NODE \*pNewNode);

NODE \*AddItemTail(HLLIST hLList, NODE \*pNewNode);

BOOL WalkList(HLLIST hLList, BOOL(\*Proc)(NODE \*));

void CloseList(HLLIST hLList);

/\* Macros \*/

//#define container\_of(ptr, type, member) ( (type \*)( (char \*)(ptr) – offsetof(type, member)))

#endif

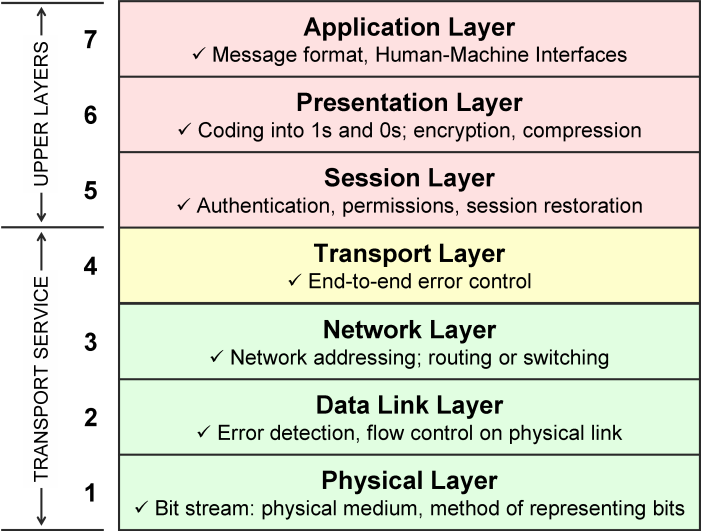
Burada görüldüğü gibi InsertItemAfter fonksiyonu yazılmıştır. AddItemHead ve AddItemTail fonksiyonlarından bu fonksiyon çağrılmıştır.

**Farklı Makinaların Prosesleri Arasında Haberleşme**

Farklı makinalar birbirlerine ağ içerisinde bağlanmış olabilir. Biz bir makinada çalışan bir programın ağa bağlı başka bir makinadaki prosese bilgi göndermesini ve almasını isteyebiliriz. Böyle bir haberleşmede artık işletim sisteminin dışında başka birtakım aktörler de devreye girecektir. Örneğin kablolama sisteminden kullanılan hub'a kadar bazı donanım birimleri işin içine karışmaktadır. Üstelik bu tür haberleşmelerde işletim sistemleri bile birbirlerinden farklı olabilmektedir. İşte hetorejen böyle ortamlarda haberleşmenin sağlıklı yürütülmesi için önceden belirlenmiş birtakım kuralların bulunması gerekir. Örneğin kablo standartları ve konnektörler nelerdir? Network kartının özellikleri nasıl olacaktır? Bilgiler nasıl paketlere ayrılıp gönderilecektir? Makinalar nasıl birbirlerinden ayrılacaktır vs. gibi... İşte tüm bu belirlemelere protokol denilmektedir.

Tıpkı fonksiyonların birbirlerini çağırarak daha yüksek seviyeli işlemleri yapar hale gelmesi gibi protokoller de üst üste yığılarak ayrı ayrı oluşturulmaktadır. Her üst protokol aşağının zaten hazır olduğu fikriyle yalnızca kendi gereksinimlerini tanımlamaktadır. Böyle katmanlı tasarımın pek çok faydası vardır. Örneğin bu sayede üst seviye protokoller detay barındırmazlar ve aşağı düzeydeki protokollerin değişmesinden fazlaca etkilenmezler. İşte farklı makinaların haberleşmesi için bu biçimde oluşturulmuş pek çok protokol ailesi vardır. Örneğin AppleTalk, NETBIOS vs. gibi...

Network altında bilgisayar haberleşmesi için protokol katmanlarının nasıl oluşturulması gerektiğine yönelik IEEE ismine OSI (Open System Interconnection) denilen bir doküman yayınlamıştır. Buna OSI model denilmektedir. OSI model bir protol ailesi değildir. Protokol ailesi oluşturacaklar için bir kılavuz niteliğindedir. OSI'nin toplam 7 katmanı vardır:



OSI'nin en aşağı katmanına “Fiziksel Katman (Physical Layer)” deilmektedir. Fiziksel katmanda iletişimin yapılacağı ortam tanımlanmaktadır. Örneğin kullanılacak kablolar, konnektörler, gerilim seviyeleri gibi. Bunun üzerinde “Veri Bağlantı Katmanı (Data Link Layer)” bulunmaktadır. Bu katmanda network kartlarına ilişkin belirlemeler, fiziksel adresleme belirlemeleri vs. bulunmaktadır. Örneğin Ethernet kartlarının protokolü olan Ethernet Protokolü bir Veri Bağlantı Katmanı Protokolüdür. Network katmanı (Network Layer) mantıksal adreslemenin tanımlandığı, bilginin nasıl paketlere ayrılıp gönderileceğinin tanımlandığı en önemlş katmanlardan biridir. Örneğin IP protokol ailesinin IP Protokolü (Internet Protocol) OSI'ye göre Network katmanına ilişkindir. Network katmanında ayrıca “internetworking” için rotalama belirlemeleri de bulunmaktadır. Network üzerinde “İleti Katmanı (Transport Layer)” bulunmaktadır. Burada paketlerin numaralandırılması, mantıksal port adreslerinin tanımlanması, hata durumunda bunun telafi edilmesi gibi belirlemeler bulundurulmaktadır. Örneğin IP protokol ailesindeki TCP ve UDP protokolleri İleti Katmanına ilişkin protokollerdir. “Otgurum Katmanı (Session Layer)” pek çok ailede bulunmamaktadır. Burada haberleşme için gereken oturum açmaya yönelik belirlemeler bulunur. Örneğin izinler, kimlik doğulama gibi. Bunun yukarısında da “Sunum Katmanı (Presentation Layer)” bulunur. Sunum katmanında gönderilip alnına bilgilerin sıkıştırılmasına, açılmasına, şifrelenmesine vs. yönelik belirlemeler bulunmaktadır. IP protokol ailesi Sunum Katmanına da sahip değildir. Nihayet en tepede “Uygulama Katmanı (Application Layer)” bulunmaktadır. Bu katman artık belli bir amacı gerçekleştirmek için oluşturlan yazılımların kullanacağı belirlemeleri içerir. Örneğin eposta için kullanılan POP3, dosya transferi için kullanılan FTP birer Uygulama Katmanı Protolüdür.

**Internetin Kısa Tarihi**

Bilgisayarları birbirlerine bağlamak ilk kez 60'yıllarda insanların aklına gelmiştir. Soğuk savaş yıllarında Amerika Savunma Bakanlığına bağlı olan DARPA (Defense Advanced Research Project Agency) kurumu birkaç üniversite ile birlikte 1969 yılında ARPANET isimli bir proje başlattı. ARPANET ilk kez 1969 yılında uzak mesafeden dört üniversitenin birbirlerine bağlanmasıyla hayata geçirilmiş oldu. ARPANET'te daha sonra bazı devlet kurumları ve üniversiteeler katılmaya başlamıştır. 70'lı yılların sonlarına doğru ARPANET Amerika'da gelişmeye başlamıştır. 1983 yılında ARPANET NCP (Network Control Protocol) protokolünü bırakarak IP ailesine ailesine geçmiştir. Ve arık ağ Internet ismiyle yayılmaya devam etmiştir. Internet 80 yıllarda Avrupa'ya ve Türkiye'ye de geldi. Ancak tabi kişsel bilgisayarlar daha yeniydi ve Internet'e ancak Üniversitelerden ve bazı devlet kurumlarından, özel sektörden bağlanılabiliyordu. 1990-91 yıllarında HTTP protokü tasarlandı ve ilk Web sayfaları oluşturulmaya başlandı. 90'lı yılların ortalarına doğru tüm dünyada kişisel bilgisayarlarla servis sağlayıcılar sayesinde Internet'e girmek mümkün hale gelmiştir. Daha sonraları modern modem/router'larla yüksek hızlı evden erişimler sağlanmıştır.

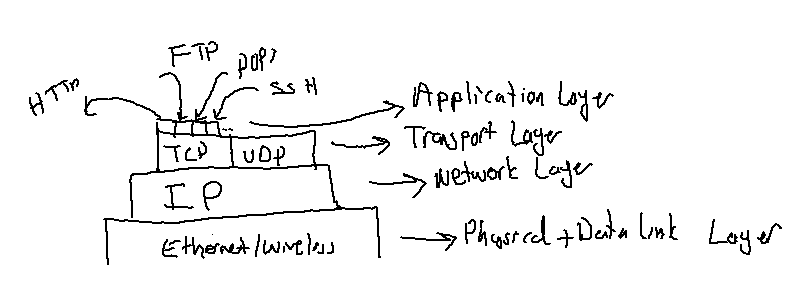
Internet ismi “internetworking” sözcüğünden gelmektedir. Internetworking “yerel ağların birbirlerine router isimli cihazlarla bağlanmalarıyla oluşturulmaktadır. Internetworking temel bir terimdir ve IP protokol ailesinin ismi buradan gelmektedir. Bugün Internet denildiğinde herkesin bağlandığı ARPANET'ten evrimleşen dev ağ aklıma gelir. (Internet yazarken I'yı büyük yazarsak bu ağ anlaşılır.) Şüphesiz mevcut protokoller sayesinde herkes kendi internetini kurabilir. Örneğin biz de birkaç arkadaşınızla ayrı bir Internet dünyası oluşturabiliriz. Hatta bazı ülkelerin bu biçimde kendilerine özgü Internet'leri vardır.

**IP Protokol Ailesi**

IP açık bir protokol ailesidir. Burada açık demekle hiçbir şirketin malının olmadığık bağımsız konsorsiyumlar tarafından yönetildiği anlamına gelmektedir. Ayrıca dokümanlar herkes tarafından paylaşılmakta ve isteyen kişiler önerilerde bulunabilmektedir.

IP protokolü Vint Cerf ve Bob Kahn tarafından 1974 yılında önce TCP sonra IP biçiminde tasarlanmıştır. Sonra aileye diğer üyeler katılmıştır. İlk ciddi gerçekleştirimi BSD sistemlerinde yapılmıştır. 1983 yılında ARPANET'in IP ailesine geçmesiyle popülaritesi çok artmıştır.

IP protokol ailesinin temel protokolleri dört katmandan oluşmaktadır.

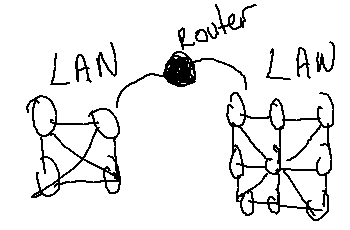


IP protokol ailesi aslında geniş bir ailedir. Ailede pek çok yardımcı protokol vardır. Yukarıdaki şekil yalnızca kursumuzda söz konusu edilen konuları kapsayacak biçimde oluşturulmuştur.

Ailenin en önemli taban protokolü IP (Internetworking Protocol) protokolüdür. Zaten aileye ismini bu protokol vermiştir. IP protokolü paket anahtarlamalı (packet switching) bir protokoldür. Yani bilgiler paket denilen öbeklere ayrılarak gönderilip alınır. IP protokolğnde adresleme artık fiziksel değil mantıksaldır. IP protokol ailesinde ağa bağlı her birime “host” denilmektedir. IP protokolünde her host'un ismine IP adresi denilen mantıksal bir adresi vardır. Mantıksal adres bunun donanımsal olarak belirlenmediği yazılımsal olarak atandığı anlamına gelmektedir. Fakat örneğin Ethernet protokolünün kullandığı MAC adresi fiziksel bir adrestir. Fiziksel adres bunun donanımsal olarak kartın üzerine çakılı olduığu ya da donanımın kendisinin bunu tespit edip işlem yaptığı adres demektir. Dolayısıyla mantıksal adresler dinamiktir, fiziksel adresler statiktir. Mantıksal adresler biz ağa dahil olduğumuzda bize atanmaktadır. Tabi biz de istediğimiz adresin atanması konusunda ısrarcı olabiliriz.

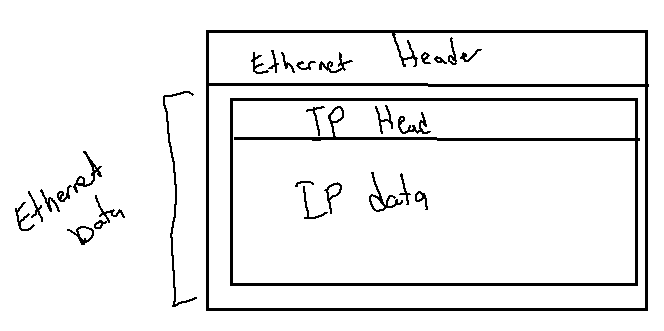
IP protokolünün de versiyonları vardır. Şu anda hala ağırlıklı kullanılan versiyon IPV4'tür. Ancak IPV6 yavaş yavaş daha yaygın kullanılır hale gelmiştir. IPV4te IP adresleri 4 byte uzunluktadır. Ancak IPV6'da IP adresleri 16 byte'tır. 4 byte'lık IP adresleri şu an için artık çok yetersiz kalmaktadır.

Bugün bilgisayarlarımızda fiziksel ve data link katmanı olarak Ethernet ve Wireless Protokolleri kullanılmaktadır. Ethernet protokolü ethernet kartına gereksinim duyar. Bu kart fiziksel olarak bilgileri bilgisayarımızdan dışarı gönderip almakta kullanılır. Ethernet protokolü de paket anahtarlamalı bir protokoldür. Yani bilgiler paket paket gönderilip alınır. Paket anahtarlama hattın etkin kullanımını sağlar. Biz Ethernet kartlarını bir hub'la biribirine bağlayarak yerel bir ağ (local area network) oluşturabiliriz. Bugün evlerimizdeki ağ da yerel bir ağdır. Yerel ağları birbirlerine bağlamak için “router” denilen aygıtlar kullanılır. Ethernet kartı (yani network kartı) aynı ağdaki bir bilgisayardan diğerine paket haberleşmesi için kullanılmaktadır. Ancak router farklı ağlar arasında paket haberleşmesi için kullanılır. Bugün evlerimizdeki ADSL modemler aynı zamanda birer router görevindedir.



Bizim evimizdeki yerel ağ Internet isimli dev ağa router aracılığıyla tek bir host gibi bağlanmaktadır. Dolayısıyla bizim Internet için dışarıdan kullanılacak tek bir IP adresimiz vardır (Tabi tek bir router ve hattımızın bulunduğunu varsayıyoruz). Bizim evimizdeki yerel ağ ayrı bir IP ağıdır. Yani ayrı bir dünyadır. Biz istersek hiç Internet'te çıkmadan kendi yerel ağımızda tüm Internet uygulamalarını (Yani IP protokol uygulamalarını) çalıştırabiliriz. Buna genellikle “Intranet” denilmektedir. O halde bizim evimizdeki bir bilgisayarın bir yerel IP adresi vardır bir de router'ımızın Internet'ten görülen bir IP adresi vardır. Router dış dünyadan gelen paketleri yerel ağda uygun bilgisayara dağıtmaktadır. Yerel ağdaki paketleri de dış dünyaya ilişkinse dış dünyaya yollamaktadır. Biz yerel ağımızdaki bir host'tan diğerine bilgi gönderirken router devreye girmez.

Ip protokolünde gönerilen bir paketin başında “IP header” isimli bir başlık kısmı vardır. Burada pakete ilişkin metadata bilgileri bulunur. Örneğin paket hangi IP adresine gönderilmektedir? Checksum bilgisi nedir? Hangi IP versiyonu kullanılmaktadır? vs. Aslında tabi (böylek olmak zorunda değil ama) bilgiler neticede ethernet kartı ile gönderilip alındığı için IP paketi aslında Ethernet protokolünün ethernet paketinin data bölümünde kodlanır. Ethernet protokolünün de ayrı bir header bölümü vardır. Örneğin:



Ethernet protokolü IEEE 802.3 numaralı standardıyla belirlenmiştir. Wireless protokolü de aynı ailedendir. O da IEEE 802.11 numaralı standarttır.

Ip protokü ile birden fazla paketten oluşan bilgi gönderilebilir mi? Evet fakat bunun için paketlere numara vererek bizim de adeta ayrı bir protokol oluşrumamız gerekir. Zaten TCP protokolü buna benzer bir protokoldür.

TCP protokolü güvenli (reliable) bir protokoldür. Burada güvenlik demek alış verişin yolda bozulmasının teleafi edilmesi ve paketlerin düzgün aktarılması anlamına gelir. Çünkü TCP'de bir akış kontrolü (flow control) vardır. Gönderen tarafla alan taraf karşılıklı konuşarak hatalı giden paketlerin telafisini sağlayabilmektedir. TCP stream tabanlı bir protokoldür. Stream tabanlı demekle byte byte okumaya kaldığı yerden devam edebilmek anlaşılır. TCP ile biz daha büyük bilgileri gönderip alabiliriz. TCP bu durumda bu bilgiyi IP paketlerine böler. Onlara numara verir ve onların karşı tarafa güvenli ulaşmasını denetler. Karşı taraf gelen bilgiyi sanki borudan okuma yapıyormuş gibi byte byte elde edebilir.

UDP (User Datagram Protocol) güvenli olmayan paket tabanlı (datagram) bir haberleşme sunar. Yani UDP'de bilgiler IP'deki gibi bağımsız paketler halinde gönderilip alınır. UDP'de bir paket ya alınır ya alınmaz. Byte byte okuma mümkün değildir. Paketin alındığına dair bir geri bildirim yapılmaz. Tabi bu özelliğinden dolayı UDP daha hızlıdır. UDP özellikle periyodik data gönderimlerinde, televizyon yayını gibi işlemlerde tercih edilmektedir.

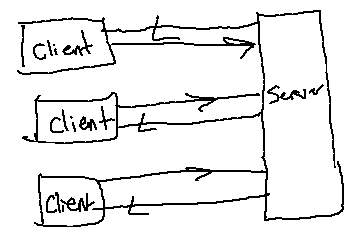
TCP bağlantılı (connection oriented) bir protokoldür, UDP bağlantısızdır (connectionless). Bağlantılı protokol demek iki taraf haberleşmeden önce birbirlerine bağlanıp karşılıklı konuşma için birbirlerini tanımaları demektir. TCP tipik olarak client-server tarzda bir çalışmayı akla getirmektedir. Client-server haberleşmede bir taraf client bir taraf server olur. Client taraf server tarafa bağlanır, haberleşme bundan sonra yapılır.

|  |  |
| --- | --- |
| **TCP** | **UDP** |
| Bağlantılı | Bağlantısız |
| Stream Tabanlı | Datagram Tabanlı |
| Güvenilir | Güvenilir Değil |
| Yavaş | Hızlı |

TCP ve UDP'de işin içine port numarası kavramı da girmektedir. Port numarası aynı host'taki uygulamaları birbirlerinden ayırmak için düşünülmüştür. Adeta şirketlerdeki içsel (internal) telefon numaralarına benzetilebilir. TCP ve UDP protokollerinde bilgi göndermek için yalnızca gönderilecek host'un IP'sinin bilinmesi yeterli değildir. Aynı zamanda oradaki uygulamanının hangi port ile ilgilendiğinin de bilinmesi gerekir. Genellikle gösterimde ip adresi ve port numarası aralarına ':' karakteri getirilerek “ip:port” biçiminde belirtilmektedir. IPV4'te toplam 65536 port numarası vardır (yani port numarası için iki byte yer ayrılır). IPV6'da ise port numaraları 4 byte uzunluğundadır. IPV4'te ilk 1024 port numarası Internet'in kendi uygulama protokelleri için ayrılmıştır. Bunlara “well known” portlar da denilmektedir. Örneğin FTP 21, SSH 22, Telnet 23, HTTP 80 numaralı portları kullanmaktadır. Biz kendi uygulamalarımız için port numarası belirleyeceksek ilk 1024 portu kullanmamalıyız.

**Client-Server Çalışma Modeli**

Yukarıda da belirtildiği gibi TCP tipik olarak client-server bir çalışmayı akla getirmektedir. Client-Server modelde ismine client ve server denilen iki ayrı program vardır. Asıl işi server program yapar. Client yalnızca istekte bulunur. Server işi yapar sonuçları client'a gönderir. Bir server birden fazla client'a hizmet verebilmektedir. >



Client-Server modelde önce client server'a bağlanır. Haberleşme ondan sonra başlar. Client-Server uygulamalar her ne kadar TCP'yi çağrıştırıyorsa da aslında bu bir haberleşme mimarisidir. Yani aslında client-server çalışma için IP ailesinin kullanılması gerekmez. Bu çalışma örneğin aynı makinadaki prosesler arasında borularla mesaj kuyruklarıyla da sağlanabilir.

Client-Server çalışmanın şu avantajları vardır:

1) Server programın çalıştığı makine güçlü olabilir. Biz de onun gücünden yararlanmak istiyor olabiliriz. Örneğin uzun zaman alan bir işlemi el terminalinden yapmak yerine el terminalini client olarak kullanıp asıl işi server'a yaptırmak uygun olabilir.

2) Server program kaynak paylaşımı sağlayabilir. Örneğin yazıcı telk bir bilgisayara bağlıdır. Başka bilgisayardaki print programları client gibi çalışarak yazıcının bağlı makinadaki server programa isteği iletir. Server da print işlemini client için yapar. Ya da örneğin server'a bir veritabanı bağlıdır. Client ondan istekte bulunur. Örneğin banka ATM'lerinde veritabanı ATM makinasının içerisinde değildir. ATM'deki program client program gibi davranmaktadır.

3) Server program client'lar arasında işbirliği sağlayabilir. Onlar arasındaki iletişime aracılık edebilir. Örneğin bir char programında client'lar birbirini tanımamaktadır. Herkes yalnızca server'ı tanır. Her client server'a bağlanır. Server client arasında haberleşmeye aracılık eder. Ağ üzerinde çalışan oyun programları bu biçimde bir server'ın işbirliği ile gerçekleştirilmektedir.

4) Client-Server çalışma dağıtık uygulamalarda da karşımıza çıkabilmektedir. Yani bir işin belirli parçalarını başka bilgisayarlarda yapıp sonra onu birleştirmek isteyebiliriz.

**Soket (Socket) Arayüzü**

Soket arayüzü ağ haberleşmesi için kullanılan bir kütüphanedir. Soket kütüphanesi ilk kez 1983 yılında BSD sistemlerinde gerçekleştirilmiştir. Daha sonra başka sistemlere uygulanmıştır. Microsoft'un soket arayüzü BSD soketlerinden alınmadır. Buna Winsock kütüphanesi denilmektedir. Windows'ta iki grup soket API'si vardır. Bunlardan birincisi tamamen BSD uyumlu API'lerdir. (Burada fonksiyon isimleri BSD'deki ile aynuıdır.) İkinci olarak başı WSA ile başlayan Windows'a özgü soket API'leridir. Biz Windows'ta da BSD uyumlu soket fonksiyonlarını kullanırsak UNIX/Linux uyumunu da sağlamış oluruz.

Soket arayüzü yalnızca IP ailesi için düşünülmüş bir arayüz değildir. Diğer protokolleri de kapsayan genel bir aryüzdür. Bu nedenle fonksiyonların parametrik yapıları biraz daha karmaşık olma eğilimindedir.

**TCP/IP İskelet Client-Server Programların Yazımı**

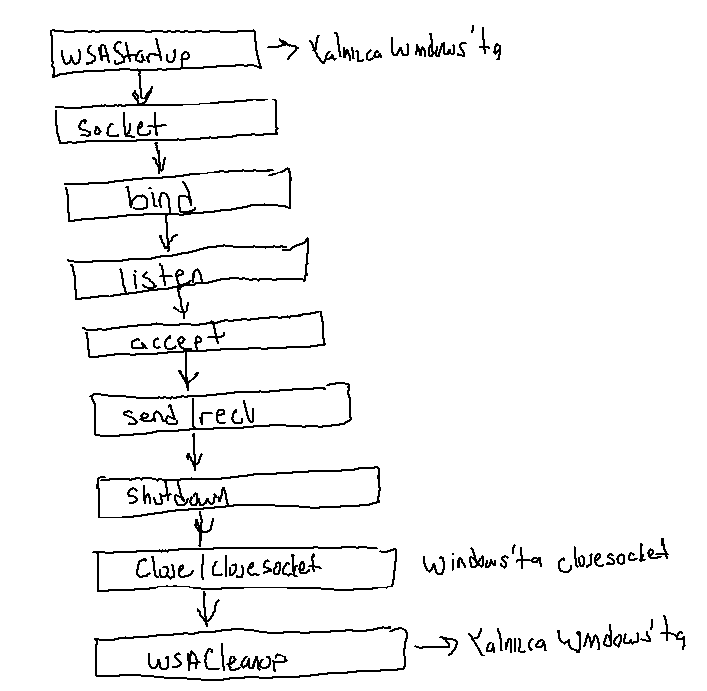
Burada iskelet bir TCP/Ip client-server programın yazımı ele alınacaktır. Uygulamada klasik BSD soket fonksiyonları kullanılacaktır. Windows'a özgü farklılıklara konu içerisinde değinilmektedir.

Anahtar Notlar: Windows'ta soket kütüphanesini ayrıca projeye dahil etmek gerekmektedir. Buunun için proje ayarlarından Linker/Input/Additional Dependencies sekmesinden Ws2\_32.lib import kütüphanesi eklenmelidir.

Windows'ta soket fonksiyonlarında UNIX/Linux sistemlerinde olmayan çeşitli typedef'ler ve sembolik sabitler de kullanılmaktadır. Windows soket uygulamalarını UNIX/Linux sistemlerine taşırken kodun bu kısımlarının düzeltilmesi gerekir. Ayrıca Windows'ta son soket fonksiyonu hatayla geri dönmüşse hata kodu GetLastError fonksiyonuyla değil, WSAGetLastError fonksiyonuyla elde edilmeltedir.

**Server Programın Yazımı**

Tipik bir TCP/Ip server programda sırasıyla şunlar yapılmalıdır:



Windows'ta (Fakat UNIX/Linux sistemlerinde değil) soket sistemini işin başında aktive etmek gerekmektedir. Bu işlem soket kullanan her proseste bir kez yapılmak zorundadır. WSAStartup fonksiyonun prototipi şöyledir:

int WSAStartup(

\_In\_ WORD wVersionRequested,

\_Out\_ LPWSADATA lpWSAData

);

Fonksiyonun birinci parametresi Winsock kütüphanesinin versiyon numarasını belirtir. Yüksek bir numara verilirse hata o luşmaz, en yüksek soket versiyonu işleme sokulur. Bu parametre MAKEWORD makrosuyla oluşturulabilir. Halen Winsock kütüphanesinin son versiyonu 2.2'dir. Fonksiyonun ikinci parametresi WSADATA isimli bir yapının adresini almaktadır. Fonksiyon bu yapının içini faydalı bilgilerle doldurur. Fonksiyon başarı durumunda sıfır değerine başarıszlık durumunda hata kodunun kendisine geri döner.

Soketi yaratmak için socket isimli fonksiyon kullanılır. Fonksiyonun prototipi şöyledir:

SOCKET WSAAPI socket(

\_In\_ int af,

\_In\_ int type,

\_In\_ int protocol

);

Fonksiyonun UNIX/Linux sistemlerindeki geri dönüş değeri int türdendir. Burada Windows'ta bu tür SOCKET isimli typedef ile temsil edilmektedir. (Yani SOCKET typedef ismi UNIX/Linux sistemlerinde yoktur).

Fonksiyonun birinci parametresi kullanılacak protokol ailesini belirtir. IPV4 protokol ailesi için bu parametre AF\_INET biçiminde girilmelidir. İkinci aparemtre kullanılacak protokolün türünü belirtir. (Yani stream tabanlı mı, datagram mı gibi.) TCP için bu parametre SOCK\_STREAM, UDP için SOCK\_DGRAM biçiminde girilmelidir. Üçüncü parametre kullanılacak üst protokolü belirtir. TCP için IPPROTO\_TCP, UDP için IPPROTO\_UDP girilebilir. Fakat IP ailesi için eğer ikinci parametre SOCK\_STREAM girilmişse ya da SOCK\_DGRAM girilmişse bu üçüncü parametre sıfır geçilebilir. Bu durumda SOCK\_STREAM için TCP, SOCK\_DGRAM için UDP anlaşılır. Fonksiyon başarı durumunda soketin handle değerine başarısızlık durumunda INVALID\_SOCKET değerine geri döner. UNIX/Linux sistemlerinde INVALID\_SOCKET isimli bir sembolik sabit yoktur. Bu sistemlerde fonksiyon başarısızlık duurumunda -1 değerine geri dönmektedir (Zaten Windows sistemlerinde de INVALID\_SOCKET -1 olarak typedef edilmiştir.)

Server program soketi yarattıktan sonra bağlamalıdır (binding). Soketin bağlanması (bind edilmesi) sırasında aslında şu belirlemeler yapılmaktadır:

1) Server hangi porttan gelen bağlantı isteklerine yanıt verecektir? Başka bir deyişle server hangi portu kullanacaktır?

2) Server hangi network arayüzünden (network kartından) gelen baplantı isteklerini dikkate alacaktır?

Bind fonksiyonun prototipi şöyledir:

int bind(

\_In\_ SOCKET s,

\_In\_ const struct sockaddr \*name,

\_In\_ int namelen

);

Fonksiyonun birinci parametresi bind edilecek soketin handle değerini almaktadır. İkinci parametre IP ailesi için sockaddr\_in isimli bir yapının adresini alır. Fonksiyonun ikjinci parametresi genel bir tür olarak struct sockaddr \* türündendir. (Eskiden void göstericiler yoktu bu parametre bu nedenle genel bir türü belirtmek için bu biçimde alınmıştır.) Her ne kadar fonksiyonun ikinci parametresi struct sockaddr \* türündense de aslında protokole bağlı olarak bir yapı almaktadır. (Örneğin IP ailesinde struct sockadd\_in \*, UNIX domain soketlerde struct sockaddr\_un \* gibi) Fonksiyon üçüncü parametresi ikinci parametrede girilen yapının byte uzunluğunu (yani sizeof'unu) almaktadır. Fonksiyon başarı durumunda sıfır, başarısızlık durumunda SOCKET\_ERROR (UNUX/LINUX sistemlerinde -1) değerine geri döner.

sockaddr\_in yapısı şöyledir:

struct sockaddr\_in {

short sin\_family;

u\_short sin\_port;

struct in\_addr sin\_addr;

char sin\_zero[8];

};

Yapının sin\_family elemanı kullanılan protokol ailesini belirtir. IP ailesi için bu elemana AF\_INET girilmelidir. Yapının sin\_port elemanına server'ın dinlemek istediği port numarası girilmelidir. Yapının sin\_addr elemanına da server'ın dinlemek istediği network arayüzünün (kartının) IP numarası girilmelidir. Ancak bu parametreye INADDR\_ANY özel değeri girilirse bu durumda server tüm network arayüzünden gelen bağlantıları kabul eder.

IP ailesinde ortak bir belirleme olarak bir byte'tan uzun olan bilgilerin “big endian” formata göre depolanmasına ve iletilmesine karar verilmiştir. Bu durumda biz Intel işlemcilerinde olduğu gibi “little endian” formatta çalışıyorsak bir byte'tan uzun değerlerin big endian'a dönüşütürülmesi gerekir. İşte bunu yapan iki fonksiyon vardır: htons (host to network byte ordering short) ve htonl (host to byte ordering long) fonksiyonları. Bu fonksiyonlar eğer zaten big endian sistemde çalışılıyorsa hiçbir şey yapmadan aynı değerle geri dönmektedir.

struct sockaddr\_in sinServer;

...

sinServer.sin\_family = AF\_INET;

sinServer.sin\_port = htons(PORTNO);

sinServer.sin\_addr.s\_addr = htonl(INADDR\_ANY);

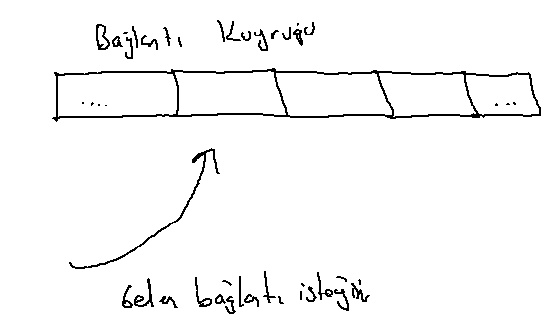
if (bind(serverSock, (structsockaddr \*) &sinServer, sizeof(sinServer)) == SOCKET\_ERROR) {

fprintf(stderr, "bind failed: %lu\n", WSAGetLastError());

exit(EXIT\_FAILURE);

}

Soket bind edildikten sonra artrık server aktif dineme konumuna geçmelidir. Bu işlem listen fonksiyonuyla yapılır. Listen fonksiyonu blokeye yol açmaz. Dinleme işlemini işletim sisteminin kendisi arka planda yapmaktadır. İşletim sistemi prosese ilişkin bir bağlantı isteyi geldiğinde o isteği bir bağlantı kuyruğuna ekler. Artık server program da bağlantı isteklerini kuyruktan alacaktır.



Listen fonksiyonunun bağlantıyı kurmadığına yalnızca gelen bağlantı isteklerini bize ilettiğine dikkat ediniz. Listen fonksiyonunun prototipi şöyledir:

int listen(

\_In\_ SOCKET s,

\_In\_ int backlog

);

Fonksiyonun birici parametresi dinleme soketinin (server soketin) handle değerini alır. İkinci parametre dinleme kuyruğunun eleman uzunluğunu belirtir. Yoğun olmayan server'lar için 8 gibi bir değer uygun olabilmektedir. İşletim sistemi gelen bağlantı isteğini kuyruğa yerleştirir. Eğer kuyruk dolarsa artık yeni bağlantı istekleri kuyruğa yerleştirilemez doğrudan reddedilir. Fonksiyon başarı durumunda sıfır değerine, başarıszlık durumunda SOKET\_ERROR değerine geri döner. Örneğin:

if (listen(serverSock, 8) == SOCKET\_ERROR) {

fprintf(stderr, "bind failed: %lu\n", WSAGetLastError());

exit(EXIT\_FAILURE);

}

Şimdi sıra artık accept işlemine gelmiştir. Accept fonksiyonu kuyrukta sıradaki bağlantı isteğini alır ve bağlantıyı sağlar. Eğer kuyukta hiçbir bağlantı yoksa blokeli modda bağlantı isteyi gelene kadar thread'i blokede bekletmektedir. Blokesiz modda accept kuyrukta bağlantı isteği yoksa başarısızlıkla sonuçlanmaktadır. Default mod blokeli moddur. Fonksiyonun prototipi şöyledir:

SOCKET accept(

\_In\_ SOCKET s,

\_Out\_ struct sockaddr \*addr,

\_Inout\_ int \*addrlen

);

Fonksiyonun birinci parametresi dinleme soketinin handle değerini alır. İkinci parametre bağlanılşan client'ın bilgilerinin yerleştirileceği sockaddr\_in yapısının adresini almaktadır. Üçüncü parametreye ikinci parametredeki yapının byte uzunluğunun yerleştirildiği nesnenin adresi girilmelidir. Fonksiyon duruma göre bunu güncelleyebilir. Fonksiyon başarı durumunda client ile konuşmakta kullanılacak soketin handle değerine geri döner. Başarısızlık durumunda da INVALID\_SOCKET (UNUX/Linux sistemlerinde -1) değerine geri dönmektedir.

Görüldüğü gibi her accept işlemi bize bağlanılan client ile konuşmakta kullanılabilecek yeni bir soket vermektedir. Yani server'ın bir tane dinleme soketi vardır. Bununla listen ve accept yapar. Ancak her accept işleminde yeni bir soket elde edilir. Örneğin:

printf("waiting for connection...\n");

addrLenClient = sizeof(sinClient);

if ((clientSock = accept(serverSock, (struct sockaddr \*) &sinClient, &addrLenClient)) == INVALID\_SOCKET) {

fprintf(stderr, "accept failed: %lu\n", WSAGetLastError());

exit(EXIT\_FAILURE);

}

Server bağlantıyı sağladığında artık bağlandığı client'ın IP numarasını ve port numarasını sockaddr\_in yapısından alabilir. Örneğin:

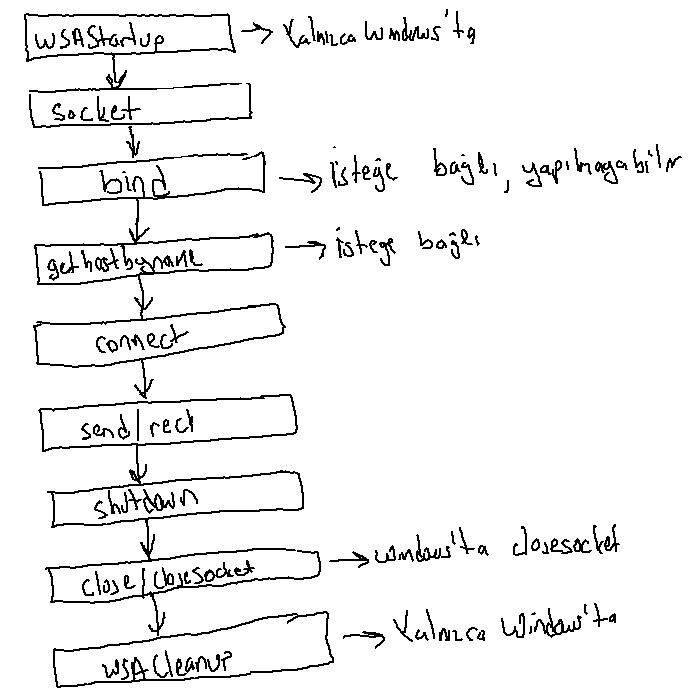
**printf**("Client connected: %s:%d\n", inet\_ntoa(sinClient.sin\_addr), ntohs(sinClient.sin\_port));

Burada inet\_ntoa, IP adresini noktalı formatta yazıya dönüştürmektedir. ntohs ise htons fonksiyonunun ters işlemini yapar.

Bağlantı sağlandıktan sonra artık send/recv fonksiyonlarıyla karşılıklı konuşma yapılır. Bu fonksiyonlar client tarafında da kullanıldığı için client tarafı anlatıldıktan sonra ortak biçimde ele alınacaktır.

**Client Programın Yazımı**

Client program sırasıyla şu aşamalardan geçilerek yazılır:



Görüldüğü gibi client da işin başında bir soket yaratır. Soket'in bind edilmesi server'da zorunludur. Ancak client soketi bind etmeyebilir. Bu durumda işletim sistemi kaynak port olarak rastgele bir port numarası atar. Bağlantıda kaynak soketin de hedef soketin de birer port numarası vardır. Biz kaynak port numarasını belirlemek istiyorsak ya da bağlantı için belli bir network kartından çıkış istiyorsak client tarafta da bind işlemi yapmalıyız.



Örneğin biz 192.168.1.21 IP numaralı host'a 5050 numaralı port'tan bağlanmak isteyelim. Burada hedef port 5050 dir. Ancak client ona herhangi bir porttan bağlanabilir. Yani bizim server'ın 5050 numaralı portuna bağlanmamız için kendi portumuzun 5050 olması gerekmez. İşte client da bind yapılmazsa işletim sistemi ona rastgele (aslında tam olarak rastgele değil, bir kuralı var) bir kaynak port atamaktadır.

Bazı server'lar ancak bazı portlardan gelen bağlantı isteklerini kabul etmektedir. Hatta router'lar buna göre ayarlanabilmektedirler. Örneğin biz evimizde “router'a kaynak port 6300 değilse dış dünyadan gelen bağlantı isteğini hiç içeriye bildirme, doğrudan reddet” diyebiliriz. Fakat kaynak portun bu biçimde sınırlandırılması çok nadir bir uygulamadır.

Client taraf bağlantı için server'ın IP adresini ve port numarasını bilmek durumundadır. Ancak IP adreslerinin akılda tutulması zor olduğu için onlara isimler de karşı düşürülmüştür. Bu isimlere “host ismi (host name)” denilmektedir. Internet'te host isimleri ile IP adreslerini eşleştiren veritabanları bulunmaktadır. Bunlara “Doman Name Server” denir. Bu server'lara ismine DNS denilen özel bir protokolle erişilmektedir. İşte eğer client server'ın host ismini biliyorsa bu host ismini DNS işlemiyle IP adresine dönüştürmesi gerekir. Bunu gethostbyname isimli soket API fonksiyonu yapar. Bu fonksiyonun ters işlemini yapan gethostbyaddress isimli bir fonksiyon da vardır:

struct hostent \*gethostbyname(

\_In\_ const char \*name

);

struct hostent \*gethostbyaddr(

\_In\_ const char \*addr,

\_In\_ int len,

\_In\_ int type

);

Bu iki fonksiyon da bize static olarak tahsis edilmiş hostent isimli bir yapının adresine geri döner. Hostent yapısı şöyledir:

typedef struct hostent {

char \*h\_name;

char \*\*h\_aliases;

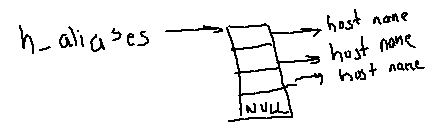
short h\_addrtype;

short h\_length;

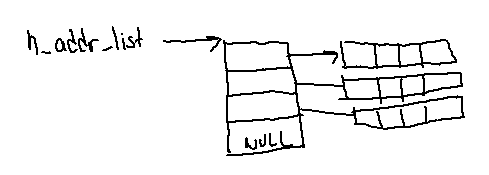
char \*\*h\_addr\_list;

} HOSTENT, \*PHOSTENT, \*LPHOSTENT;

Yapının h\_name elemanı domain host ismini vermektedir. h\_aliases elemanı diğer host isimlerini verir.



Yapının h\_addr\_list elemanı ip numaralarını vermektedir. IPV4'te bu göstericiyi gösteren gösterici 4'er elemanlık char türden dizileri gösterir:



Fakat bazen kullanıcılar host ismini vermek yerine onun IP adresini noktalı biçimde yazı olarak da verebilirler. İşte bu biçimde verilmiş yazıyı dört byte'lık sayıya dönüştürmek için inet\_addr isimli bir fonksiyon kullanılmaktadır.

unsigned long inet\_addr(

\_In\_ const char \*cp

);

Eğer girilen yazı uygun formatta değilse fonksiyon INADDR\_NONE değerine geri döner. İşte verilen yazıyı önce bu fonksiyona sokup eğer yazı noktalı biçimde IP belirtmiyorsa gethostnyname fonksiyonuna sokmak iyi bir tekniktir.

Artık bağlanılacak host'un ip adresi de belirlendiğine göre sıra connect işlemine gelmiştir. Conenct fonksiyonun prototipi şöyledir:

int connect(

\_In\_ SOCKET s,

\_In\_ const struct sockaddr \*name,

\_In\_ int namelen

);

Fonksiyonun birinci parametresi soketin handle değerini alır. İkinci parametre bağlanılacal server'ın bilgilerini alan sockaddr\_in türünden yapısının adresini almaktadır. Yani programcı server'ın ip numarasını ve port numarasını sockaddr\_in türünden bir yapının içerisine yerleştirip bu yapıyı da conenct fonksiyonuna verir. sockaddr\_in yapısı üzerinde server programda biraz durmuştuk. Yapıyı tekrar hatırlayalım:

struct sockaddr\_in {

short sin\_family;

u\_short sin\_port;

struct in\_addr sin\_addr;

char sin\_zero[8];

};

Görüldüğü gibi IP adresi yapının sin\_addr elemanındadır. Ancak sin\_addr in\_addr türünden bir yapı belirtmektedir:

typedef struct in\_addr {

union {

struct {

u\_char s\_b1,s\_b2,s\_b3,s\_b4;

} S\_un\_b;

struct {

u\_short s\_w1,s\_w2;

} S\_un\_w;

u\_long S\_addr;

} S\_un;

} IN\_ADDR, \*PIN\_ADDR, FAR \*LPIN\_ADDR;

Görüldüğü gibi bu yapının içerisinde bir birlik vardır. Birliğin de S\_addr elemanı long türünden ip adresini belirtmektedir. O halde biz sin\_addr.S\_un.S\_addr ifadesi ile long türden ip adresine erişebiliriz. İşte bu ifade uzun olduğu için aşağıdaki gibi bir makroyla ifade kısaltılmıştır:

#define s\_addr S\_un.S\_addr

Bu durumda yukarıdaki ifade sin\_addr.s\_addr biçiminde kısaltılabilir. sockaddr\_in yapısı UNIX/Linux sistemlerinde biraz daha farklı organize edilmiş olabilir. Ancak yapının elemanları aynı isimdedir. Örneğin bu sistemlerde in\_addr içerisinde doğrudan s\_addr bulunuyor olabilir. Bu durumda connect işlemi şöyle yapılır:

struct sockaddr\_in sinClient;

...

sinClient.sin\_family = AF\_INET;

sinClient.sin\_port = htons(PORTNO);

sinClient.sin\_addr.s\_addr = inet\_addr(HOSTNAME);

if (sinClient.sin\_addr.s\_addr == INADDR\_NONE) {

if ((host = gethostbyname(HOSTNAME)) == NULL) {

fprintf(stderr, "gethosybyname failed: %lu\n", WSAGetLastError());

exit(EXIT\_FAILURE);

}

memcpy(&sinClient.sin\_addr.s\_addr, host->h\_addr\_list[0], host->h\_length);

}

if (connect(clientSock, (struct sockaddr \*) &sinClient, sizeof(sinClient)) == SOCKET\_ERROR) {

fprintf(stderr, "connect failed: %lu\n", WSAGetLastError());

exit(EXIT\_FAILURE);

}

connect fonksiyonu uygulandığında server program accept'te beklemiyorsa belli bir zaman aşımı süresi kadar beklenir sonra connect başarısız olur.

Şüphesiz çok client'lı uygulamalarda server bir kez değil döngü içerisinde her client için accept uygulamalıdır. Her accept server'a o client'la konuşmak için ayrı soket verir.

**Client İle Server Arasında Bilgi Alış Verişi send ve recv Fonksiyonları**

Bağlantı sağlandıktan sonra artık client server'a server da client'a send ve recv fonksiyonlarıyla bilgi gönderip alabilir. Send fonksiyonun prototipi şöyledir:

int send(

\_In\_ SOCKET s,

\_In\_ const char \*buf,

\_In\_ int len,

\_In\_ int flags

);

Fonksiyonun birinci parametresi soket handle değerini alır. İkinci parametre gönderilecek bilginin yerleştirileceği dizinin adresini belirtir. Üçüncü parametre gönderilecek byte sayısını belirtmektedir. Son parametre bazı gönderim özelliklerini belirlemekte kullanılır. Bu parametre sıfır geçilebilir. Fonksiyon blokeli modda network tamponuna aktarılan byte sayısı ile geri döner. Network tamponu doluysa fonksiyon tampona yazabildiği kadar bilgiyi yazar ve bloke olmadan geri döner. Send başarısızlık durumunda SOCKET\_ERROR (UNIX/Linux sistemlerinde -1) değerine geri dönmektedir.

send fonksiyonu bilgi karşı tarafa ulaşana kadar beklemez. send bilgiyi network tamponuna yazar hemen geri döner. Network taponundaki bu bilgi paketlenerek karşı tarafa yollanacaktır. Yani send'ten başarılı olarak geri dönülmesi bilgiyi karşı tarafın aldığı anlamına gelmez. Ayrıca network tamponu doluysa send tüm bilginin tampona yazılmasını beklemeyebilir. Yazabildiği kadarını yazıp o değere geri dönebilir. (Bazı sistemlerde send tüm bilgi yazılana kadar blokede kalabilmektedir. O sistemlerde bile çok büyük bilgiler tam olarak tek seferde tampona yazılamayabilir.) Fakat send eğer netowork tamponu tıka basa doluysa en az bir byte yazana kadar blokede bekler. Bu nedenle send'in geri dönüş değeri yine acaba tüm bilgiler tampona aktarıldı mı diye kontrol edilmelidir.

recv fonksiyonu da çok benzerdir:

int recv(

\_In\_ SOCKET s,

\_Out\_ char \*buf,

\_In\_ int len,

\_In\_ int flags

);

Fonksiyonun birinci parametresi soketin handle değerini alır. İkinci parametre bilginin yerleştirileceği char türden dizinin adresini almaktadır. Üçüncü parametre kaç byte okunmak istendiğini belirtir. Son parametre ise okuma işleminin biçimini belirtmektedir. Bu parametre sıfır geçilebilir. recv fonksiyonu talep edilen byte'ın hepsinin okunmasını beklemez. O anda sokete ne kadar bilgi gelmişse talep edilenen o kadarını okur ve okuyabildiği byte sayısı ile geri döner. Yani biz örneğin recv ile 100 byte okumak isteyelim. Bu yüz byte'ı tek bir recv ile okuyamayabiliriz. recv o anda gelmiş olan 40 byte bilgi varsa bize hemen onu verir. Bizi bekletmez. Recv başarısz olduğunda SOCKET\_ERROR değeri ile geri dönmektedir. Ayrıca TCP/IP soket sisteminde karşık tarafın tek bir send gönderdiğini diğer tarafın tek bir recv ile alması garanti değildir. recv fonksiyonu blokeli modda (default durumda) en az bir byte okuyana kadar blokede bekler. Örneğin recv ile biz 1024 byte okumak istemiş olalım. Ancak netowork tamponunda hazır hiçbir byte olmasın. recv blokede bekler. Sonra örneğin tampona 5 byte gelmiş olsun. recv bu 5 byte'ı alıp hemen geri döner.

**shutdown ve close İşlemleri**

Soket bir handle sistemidir. Dolayısıyla tüm handle sistemlerinde olduğu gibi işimiz bitince soketleri de kapatmalıyız. Tabii biz soketi kapatmamışsak prosesin bitmesiyle işletim sistemi dosyalarda olduğu gibi soketleri de kapatmaktadır. Soketin kapatılması UNIX/Linux sistemlerinde close fonksiyonuyla (anımsanacağı gibi close aynı zamanda dosyaları da kapatmaktadır. Zaten soketler de bu sistemlerde birer dosya gibi ele alınmaktadır) Windows sistemlerinde closesocket fonksiyonuyla yapılır. Soket kapatıldıktan sonra artık o soketten okuma ve yazma yapamayız. Ancak soketi kapatmadan önce shutdown fonksiyonunu çağırmak tavsiye edilir. Önce shutdown sonra close işlemine “graceful close” denilmektedir. shutdown fonksiyonu ile biz soketin kapatılması için hazırlık yaparız. Böylece TCP düzeyinde iletişimin kesilmesi için bir el sıkışma gerçekleşir. shutdown fonksiyonunun prototipi şöyledir:

int shutdown(

\_\_in SOCKET s,

\_\_in int how

);

Fonksiyonun birinci parametresi soketin handle değerini alır. İkinci parametre şunlardan biri olabilir:

SD\_RECEIVE

SD\_SEND

SD\_BOTH

SD\_SEND “ben artık bilgi göndermeyeceğim” anlamına gelir. Böylece karşı taraf recv fonksiyonu ile okuma yaptığında sanki soketi kapatılmış gibi görür. Fakat bu sırada SD\_SEND uygulamış taraf soketten okuma yapabilir. SD\_RECEIVE ise “ben artık soketten okuma yapmayacağım, fakat gönderme yapabilirim” anlamına gelir. SD\_BOTH her iki durumu da kapsamaktadır. İşte soket önce el sıkışmalı biçimde shutdown ile sonra da close uygulanarak kapatılmalıdır. Server'ın dinleme soketi için shutdown uygulamasına gerek yoktur.

Windows sistemlerinde iskelet bir TCP/IP Client server program şöyle yazılabilir:

/\* Server.c \*/

#include<stdio.h>

#include<stdlib.h>

#include<WinSock2.h>

#include<Windows.h>

#definePORTNO 5050

void ExitSys(LPCSTR lpszMsg, int status, DWORD dwLastError);

int main(void)

{

DWORD result;

WSADATA wsaData;

SOCKET serverSock, clientSock;

structsockaddr\_in sinServer, sinClient;

int addrLenClient;

char buf[1024];

if ((result = WSAStartup(MAKEWORD(2, 2), &wsaData)) != 0)

ExitSys("WSAStartup", EXIT\_FAILURE, result);

if ((serverSock = socket(AF\_INET, SOCK\_STREAM, IPPROTO\_TCP)) == INVALID\_SOCKET)

ExitSys("socket", EXIT\_FAILURE, WSAGetLastError());

sinServer.sin\_family = AF\_INET;

sinServer.sin\_port = htons(PORTNO);

sinServer.sin\_addr.s\_addr = htonl(INADDR\_ANY);

if (bind(serverSock, (structsockaddr \*) &sinServer, sizeof(sinServer)) == SOCKET\_ERROR)

ExitSys("bind", EXIT\_FAILURE, WSAGetLastError());

if (listen(serverSock, 8) == SOCKET\_ERROR)

ExitSys("listen", EXIT\_FAILURE, WSAGetLastError());

printf("waiting for connection...\n");

addrLenClient = sizeof(sinClient);

if ((clientSock = accept(serverSock, (structsockaddr \*) &sinClient, &addrLenClient)) == INVALID\_SOCKET)

ExitSys("accept", EXIT\_FAILURE, WSAGetLastError());

printf("Client connected: %s:%d\n", inet\_ntoa(sinClient.sin\_addr), ntohs(sinClient.sin\_port));

for (;;) {

if ((result = recv(clientSock, buf, 1024, 0)) == SOCKET\_ERROR)

ExitSys("send", EXIT\_FAILURE, WSAGetLastError());

buf[result] = '\0';

puts(buf);

if (!strcmp(buf, "quit"))

break;

strrev(buf);

if (send(clientSock, buf, strlen(buf), 0) == SOCKET\_ERROR)

ExitSys("send", EXIT\_FAILURE, WSAGetLastError());

}

shutdown(clientSock, SD\_BOTH);

closesocket(clientSock);

closesocket(serverSock);

return 0;

}

void ExitSys(LPCSTRlpszMsg, intstatus, DWORDdwLastError)

{

LPTSTR lpszErr;

if (FormatMessage(FORMAT\_MESSAGE\_ALLOCATE\_BUFFER | FORMAT\_MESSAGE\_FROM\_SYSTEM, NULL, dwLastError,

MAKELANGID(LANG\_NEUTRAL, SUBLANG\_DEFAULT), (LPTSTR)&lpszErr, 0, NULL)) {

fprintf(stderr, "%s: %s", lpszMsg, lpszErr);

LocalFree(lpszErr);

}

exit(status);

}

/\* Client.c \*/

#include<stdio.h>

#include<stdlib.h>

#include<string.h>

#include<WinSock2.h>

#include<Windows.h>

#definePORTNO 5050

#defineHOSTNAME "127.0.0.1"

void ExitSys(LPCSTR lpszMsg, int status, DWORD dwLastError);

int main(void)

{

WSADATA wsaData;

int result;

structsockaddr\_in sinClient;

SOCKET clientSock;

structhostent \*host;

char buf[1024];

char msg[] = "This is a test";

if ((result = WSAStartup(MAKEWORD(2, 2), &wsaData)) != 0)

ExitSys("WSAStartup", EXIT\_FAILURE, result);

if ((clientSock = socket(AF\_INET, SOCK\_STREAM, IPPROTO\_TCP)) == INVALID\_SOCKET)

ExitSys("socket", EXIT\_FAILURE, WSAGetLastError());

/\*

sinClient.sin\_family = AF\_INET;

sinClient.sin\_port = htons(5060);

sinClient.sin\_addr.s\_addr = htonl(INADDR\_ANY);

if (bind(clientSock, (struct sockaddr \*) &sinClient, sizeof(sinClient)) == SOCKET\_ERROR)

ExitSys("bind", EXIT\_FAILURE, WSAGetLastError());

\*/

sinClient.sin\_family = AF\_INET;

sinClient.sin\_port = htons(PORTNO);

sinClient.sin\_addr.s\_addr = inet\_addr(HOSTNAME);

if (sinClient.sin\_addr.s\_addr == INADDR\_NONE) {

if ((host = gethostbyname(HOSTNAME)) == NULL)

ExitSys("gethostbyname", EXIT\_FAILURE, WSAGetLastError());

memcpy(&sinClient.sin\_addr.s\_addr, host->h\_addr\_list[0], host->h\_length);

}

if (connect(clientSock, (structsockaddr \*) &sinClient, sizeof(sinClient)) == SOCKET\_ERROR)

ExitSys("connect", EXIT\_FAILURE, WSAGetLastError());

printf("connected\n");

for (;;) {

printf("Text:");

gets(buf);

if (send(clientSock, buf, strlen(buf), 0) == SOCKET\_ERROR)

ExitSys("send", EXIT\_FAILURE, WSAGetLastError());

if (!strcmp(buf, "quit"))

break;

if ((result = recv(clientSock, buf, 1024, 0)) == SOCKET\_ERROR)

ExitSys("send", EXIT\_FAILURE, WSAGetLastError());

buf[result] = '\0';

puts(buf);

}

shutdown(clientSock, SD\_BOTH);

closesocket(clientSock);

return 0;

}

void ExitSys(LPCSTRlpszMsg, intstatus, DWORDdwLastError)

{

LPTSTR lpszErr;

if (FormatMessage(FORMAT\_MESSAGE\_ALLOCATE\_BUFFER | FORMAT\_MESSAGE\_FROM\_SYSTEM, NULL, dwLastError,

MAKELANGID(LANG\_NEUTRAL, SUBLANG\_DEFAULT), (LPTSTR)&lpszErr, 0, NULL)) {

fprintf(stderr, "%s: %s", lpszMsg, lpszErr);

LocalFree(lpszErr);

}

exit(status);

}

Yukarıdaki client-server programın Linux sistemlerinde eşdeğeri de şöyle yazılabilir:

**Çok Client'lı Uygulamalar**

Normal olarak server'lar aynı anda birden fazla clien'ta hizmet verebilecek biçimde tasarlanırlar. Bu durum server programların yazılmasını biraz karmaşık hale getirmektedir. Öncelikle çok client'lı server programlar yazılırken yalnızca bir tane accept uygulanmaması gerekir. Her accept sıradaki client ile bağlantı kuracağına göre böyle server'larda accept fonksiyonları da bir döngü içerisinde uygulanmalıdır.

Tabii çok client'lı server programlarının asıl zorluğu birden fazla accept uygulamak değildir. Server her bağlandığı client için yeni bir soket elde eder ve o client'la o soketi kullanarak konuşur. İşte birden fazla client ile aynı anda konuşma sırasında server bir client'ta blokede kalabilmektedir. Bu durumda diğer client'larla server'ın konuşması mümkün olmaz. Server ile client'lar arasındaki konuşma aynı anda ve birbirlerini etkilemeden gerçekleştirilmelidir. (Örneğin server client'lardan birine recv uygulamış olsun. Fakat o client henüz server'a birşey göndermemiş olsun. Server akışı blokede kalacağından dolayı server artık diğer client'larla konuşamaz hale gelir.) Pekiyi server'ın birden fazla client ile bloke olmadan aynı anda konuşması nasıl sağlanmaktadır? İşte bunun için birkaç yöntem uygulanmaktadır: Thread oluşturma yöntemi, select ya da poll yöntemi, IO Completion port yöntemi. Şimdi bunları sırasıyla ele alalım:

**Thread Yöntemi**

Bu yöntemde server her bir client ile bağlantı sağladığında bir thread yaratır ve o client ile o thread yoluyla konuşur. Böylece bir thread bloke olsa bile diğerleri çalışmaya devam edeceği için sorun oluşmaz. Örneğin:

for (;;) {

if ((clientSock = accept(serverSock, (structsockaddr \*) &sinClient, &addrLenClient)) == INVALID\_SOCKET)

ExitSys("accept", EXIT\_FAILURE, WSAGetLastError());

printf("Client connected: %s:%d\n", inet\_ntoa(sinClient.sin\_addr), ntohs(sinClient.sin\_port));

if ((hThread = CreateThread(NULL, 0, ClientThreadProc, clientSock, 0, &dwThreadId)) == NULL)

ExitSys("CreatrThread", EXIT\_FAILURE, GetLastError());

}

}

Görüldüğü ibi Thread fonksiyonuna soket parametre olarak geçirilmiştir. Bu durumda her client içinm aynı fonksiyon çalıştırılır fakat bunlara geçirilen parametre farklı olduğu için farklı işlemler gerçekleşecektir. Thread modelinin en önemli dezavantajı fazlaca sistem kaynağı kullanma eğiliminde olmasıdır. Her ne kadar IO olaylarına yanıt veren thread’ler zamanının büyük çoğunluğunu uykuda geçiriyorsa da thread’ler yine de önemli ölçüde sistem kaynağına mal olmaktadır. Örneğin her client için bir thread’in yaratıldığı durumda thread’lerin stack’leri bile ciddi bir bellek harcamasına yol açabilir. O halde thread yöntemi çok fazla client’ın söz konusu olduğı server sistemlerinde uygun değildir. (Buradaki “çok fazla” nitelemesi o andaki sisteme bağlı olarak değişebilir. Örneğin bugün kullandığımız güçlü bir donanımda birkaç yüz thread ciddi sistem kaynağı harcar.) Ancak bazı çok client’lı uygulamalardaki client sayısı makul düzeyde olabilir. Örneğin bir kağıt oyununda client’lar az sayıdadır. Bu tür durumlarda thread modeli basitliğinden dolayı ilk tercih edilecek modellerdendir.

**Select / Poll Modeli**

Bu model özellikle UNIX/Linux sistemlerinde tercih edilen bir modeldir. Fakat Windows sistemlerinde select fonksiyonu da kullanılabilmektedir. select ve poll fonksiyonları aslında aynı işlemi farklı parametrik yapılarla yapan iki POSIX fonksiyonudur. select System 5 grubu sistemlerde poll ise BSD türevi sistemlerde ilk kez tanımlanmıştır. Fakat bugün her iki fonksiyon da POSIX standartlarında standart fonksiyon olarak bulunmaktadır. UNIX/Linux sistemlerinde select ve poll yalnızca soket işlemleri için kullanılan fonksiyonlar değildir. Genel olarak asenkron IO işlemlerinde kullanılmaktadır. Örneğin bu fonksiyonlar normal dosyalarla, borularla da çalışabilmektedir. Tabii Windows sistemlerinde select fonksiyonu bu kadar genel bir fonksiyon değildir. Yalnızca soket işlemlerinde kullanılmaktadır.

select (ve poll) kullanımı şöyledir: Biz soketleri bir dizide toplayıp select fonksiyonuna veririz. select bu soketleri kendisi izler. Eğer onların hiçbirinde bir IO olayı yoksa bizi blokede bekletir. Eğer soketlerin bir ya da birden fazlasında bir IO olayı gerçeklemişse select blokeyi çözer. Biz de hangi soketlerde hangi IO olayının olduğunu sorgularız. Yalnızca onlar üzerinde blokeye yol açmadan işlem yaparız. Örneğin biz select’e 100 client’ın soketini vermiş olalım. O sırada bir client’tan bilgi gelmiş olsun. select blokeyi çözekcektir. Biz de o soketten okuma yaparsak hiç bloke oluşmayacaktır. Tabii select bir kez çağrılacak bir fonksiyon değildir. Genellikle döngü içerisinde bu işlemler hep yinelenir.

Akış select fonksiyonundan çıktığında biz recv yaptığımızda karşı tarafın gönderdiği bilginin yalnızca bir kısmını elde etmiş olabiliriz. Bu nedenle okunan bilgilerin bir tamponda toplanması ve bilginin tamamı geldiğinde işleme sokulması gerekmektedir. Biz select fonksiyonuna hangi IO olayları ile ilgilendiğimizi veririz. select de yalnızca o olayları bizim için izler.

**IO Completion Port ve Asenkron IO Modelleri**

Windows’ta en çok tavsiye edilen model “IO Completion Port” modelidir. Bunun UNIX/Linux sistemlerindeki en yakın karşılığı “Asenkron IO” modelidir. IO completion Port arka planda pek çok framework’ün kullandığı modeldir ve Windows sistemlerinde Microsoft tarafından çeşitli zamanlarda optimize edilmiştir. IO Completion Port modeli de aslında genel bir IO modelidir. Yani yalnızca soketlerle değil diğer IO olaylarıyla da kullanılmaktadır. IO Completion Port modelinde biz soketleri sisteme veririz. Bu soketlerde olay gerçekleştiğinde sistem bizim belirlediğimiz fonksiyonları çağırır. Tabi çağırma sistemin kendisinin yarattığı bir akış tarafından asenkron yapılmaktadır. Biz de bu callback fonksiyon içerisinde okuma yazma işlemlerini ve diğer işlemleri yaparız. UNIX/Linux sistemlerindeki aio\_xxx fonksiyonları da benzer bir mantığa sahiptir.

**Client İle Server Arasındaki Mesajlaşmalar**

Client ile server arasındaki mesajlaşmalar aslında bir uygulama protokolü oluşturmaktadır. Temel olarak mesajlaşma text tabanlı olarak ya da binary düzeyde yapılabilir. Text tabanlı mesajlaşmanın gerçekleştirilmesi daha kolaydır. Fakat binary mesajlaşma daha hızlı olma eğilimindedir. IP ailesinin üst seviye protokollerinin çoğu text düzeyde mesajlaşma yapmaktadır.

Text düzeydeki mesajlaşmalarda client ve serve birbirlerine istekleri ve yanıtları birer yazı olarak gönderir alır. Örneğin dört işlem yapan ve sonucu client’a gönderen bir server söz konusu olsun. Client’tan server’a gönderilen komutların genel yapısı şöyle olabilir:,

“ADD op1 op2”

“SUB op1 op2”

“MUL op1 op2”

“DIV op1 op2”

Server’dan client’a gönderilen yanıtın formatı da şöyle olabilir:

“val”

Server aldığı yazıyı parse eder. Sonucu hesaplar ve onu client’a gönderir. Örneğin IRC stili bir chat programı yazacak olalım. Client’tan server’a gönderilecek mesajlar şunlar olabilir:

“LOGIN user\_name password”

“CHATMSG message”

“LOGOUT”

“GETUSERS”

Server’dan client’a gönderilkecek mesajlar da şöyle olabilir:

“LOGIN\_ACCEPTED”

“NEWMSG message”

“USERLIST”

Örneğin dizin işlemleri yapan bir TCP uygulamasında client’tan server’a gönderilen mesajlar şunlar olabilir:

“LOGIN username password”

“GETDIR”

“CHDIR path”

“LISTDIR”

“GETFILE name”

“SENDFILE name”

“LOGOUT”

Server’dan client’a gönderilen mesajlar da şöyle olabilir:

“LOGIN\_ACCEPTED”

“CURRENT\_DIR”

“OK”

“DIRCONTENTS file list”

“FILECONTENTfile\_data”

“ERROR”

Görüldüğü gibi bu tür uygulamalarda client taraf server’a soketten bağlandıktan sonra bir de ayrıca kendisini server’a kabul ettirmeye çalışmaktadır. Bu işleme login işlemi diyebiliriz. Yani client’ın server’a fiziksel olarak bir porttan bağlanmış olması server’ın onu kabul edeceği anlamına gelmemektedir. Bazen bu iki kavram “fiziksel bağlantı” ve “mantıksal bağlantı” biçiminde de ifade edilebilmektedir.

Komutlar ve yanıtlar yazısal olarak gönderilip alınırken alan taraf yazının sonunu nasıl bilecektir? İşte bunun için iki yöntem kullanılabilir. Birinci yöntemde yazı özel bir karakterle (örneğin ‘\n’ ya da ‘\0’ gibi) sonlandırılır. Karşı taraf da o karakteri görene kadar portu byte byte okur. Tabii portun böyle byte byte okunması çok verimli olmadığı için okuyan tarafın daha büyük bloğu okuyup bir tampona yerleştirmesi ve o tamponda sonlandırıcı karakteri araması daha uygun olur. İkinci yöntemde her yazının uzunluğunun yazıdan önce gönderilmesidir. Yazı uzunluğu binary olarak gönderilebilir.

Belli bir miktarda bilginin kesin olarak gönderilip alınabilmesi için aşağıdaki iki fonksiyon kullanılabilir:

int ReadSocket(SOCKETsock, constvoid \*buf, intcount)

{

int result;

int left = count, index = 0;

while (left > 0) {

if ((result = recv(sock, (char \*)buf + index, left, 0)) == SOCKET\_ERROR)

returnSOCKET\_ERROR;

if (result == 0)

break;

index += result;

left -= result;

}

return index;

}

int WriteSocket(SOCKETsock, constvoid \*buf, intcount)

{

int result;

int left = count, index = 0;

while (left > 0) {

if ((result = send(sock, (char \*)buf + index, left, 0)) == SOCKET\_ERROR)

returnSOCKET\_ERROR;

if (result == 0)

break;

index += result;

left -= result;

}

return index;

}

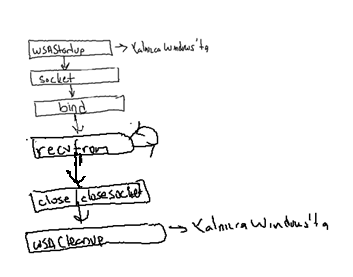
**UDP/IP Haberleşme**

Daha önceden de belirtildiği gibi UDP bağlantısız ve datagram haberleşme sunmaktadır. UDP’de bir bağlantı olmadığı için TCP’deki gibi bir akış kontrolü yoktur. Gönderen taraf bilgiyi bir paket olarak gönderir. Alan taraf bunu byte byte almaz. Paketi tümden alır. Gönderen taraf bilginin alıcı tarafından alındığını bilmez. UDP özellikle periyodik birtakım mesajların iletilmesi için tercih edilmektedir. Örneğin dağıtık bir sistemde server’lar belli periyotlarla proxy’ye “ben çalışıyorum, sağlam durumdayım” mesajı iletmek isteyebilirler. Ya da bir oyun programı periyodik olarak arabanın konumunu ve hızını ış dünyaya iletmek isteyebilir. Bu tür durumlarda UDP hızından ve kullanım kolaylığından dolayı tercih edilmektedir. Televizyon yayını gibi “broadcast” işlemlerde de UDP kullanılmaktadır.

UDP’de TCP’d eolduğu gibi client ve server kavramları çok belirgin değildir. Ancak yine de UDP’de geleneksel olarak bilgiyi alan tarafa server, bilgiyi gönderen tarafa client denilmektedir.

**UDP Server Programın Organizasyonu**

Tipik bir UDP server program şöyle oluşturulmaktadır:



Server hangi port’tan gelen bilgileri alacağını yine bind işlemi ile belirler. recvfrom her türlü client’tın gönderdiği bilgileri okumaktadır. Server gönderilen paketin kimin tarafından gönderildiğini recvfrom fonksiyonunun parametresinden elde eder. recvfrom fonksiyonunun parametrik yapısı şöyledir:

int recvfrom(

\_\_in SOCKET *s*,

\_\_out char\* *buf*,

\_\_in int *len*,

\_\_in int *flags*,

\_\_out struct sockaddr\* *from*,

\_\_in\_out int\* *fromlen*

);

Fonksiyonun birinci parametresi soketin handle değerini, ikinci parametresi UDP paketinin yerleştirileceği dizinin adresini alır. Üçüncü parametre bu dizinin uzunluğunu belirtir. Eğer paket daha büyükse böylece dizi taşmaz, kırpılarak diziye yerleştirilir. Dördüncü parametre 0 geçilebilir. Fonksiyonun beşinci parametresine sockaddr\_in yapısının adresi girilir. Böylece paketin kimden geldiği anlaşılmaktadır. Fonksiyonun son parametresi bizim ve karşı tarafın sockaddr yapısının byte uzunluğunu almaktadır. Biz fonksiyonu çağırmadan önce buraya kendi sockadd\_in yapımızın sizeof değerini girmemiz gerekir. Fonksiyon da bu parametrei güncelleyerek karşı tarafın sockaddr yapısının uzunluğunu elde eder. recvfrom fonksiyonu başarısızlık durumunda SOCKET\_ERROR değerine başarı durumunda da alınan byte sayısına geri döner.

UDP soketinin yaratılırken protokolün UDP olarak belirlenmiş olması gerekir. Örneğin:

if ((serverSock = socket(AF\_INET, SOCK\_DGRAM, IPPROTO\_UDP)) == INVALID\_SOCKET) {

fprintf(stderr, "socket failed: %lu\n", WSAGetLastError());

exit(EXIT\_FAILURE);

}

Tipik bir iskelet UDP server programı şöyle oluşturulabilir:

#include<stdio.h>

#include<string.h>

#include<stdlib.h>

#include<WinSock2.h>

#include<Windows.h>

#definePORTNO 5050

int main(void)

{

WSADATA wsaData;

int result;

SOCKET serverSock;

structsockaddr\_in sinServer;

structsockaddr\_in sinClient;

int addrLenClient;

char buf[8192];

if ((result = WSAStartup(MAKEWORD(2, 2), &wsaData)) != 0) {

fprintf(stderr, "WSAStartup failed: %d\n", result);

exit(EXIT\_FAILURE);

}

if ((serverSock = socket(AF\_INET, SOCK\_DGRAM, IPPROTO\_UDP)) == INVALID\_SOCKET) {

fprintf(stderr, "socket failed: %lu\n", WSAGetLastError());

exit(EXIT\_FAILURE);

}

sinServer.sin\_family = AF\_INET;

sinServer.sin\_port = htons(PORTNO);

sinServer.sin\_addr.s\_addr = htonl(INADDR\_ANY);

if (bind(serverSock, (structsockaddr \*) &sinServer, sizeof(sinServer)) == SOCKET\_ERROR) {

fprintf(stderr, "bind failed: %lu\n", WSAGetLastError());

exit(EXIT\_FAILURE);

}

for (;;) {

addrLenClient = sizeof(sinClient);

result = recvfrom(serverSock, buf, 8192, 0, (structsockaddr \*) &sinClient, &addrLenClient);

printf("%d byte received from %s:%d\n", result, inet\_ntoa(sinClient.sin\_addr), sinClient.sin\_port);

}

closesocket(serverSock);

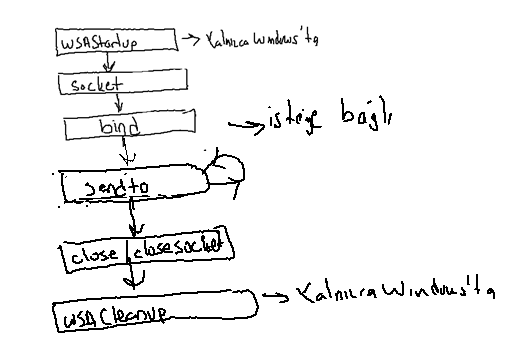
WSACleanup();

return 0;

}

**UDP Client Programın Organizasyonu**

UDP/IP haberleşmede geleneksel olarak paketi gönderen tarafa client denilmektedir. İskelet bir client program şöyle oluşturulur:



sendto fonksiyonun prototipi şöyledir:

int sendto(

\_\_in SOCKET s,

\_\_in const char\* buf,

\_\_in int len,

\_\_in int flags,

\_\_in const struct sockaddr\* to,

\_\_in int tolen

);

Fonksiyonun birinci parametresi soketin handle değerini ikinci parametresi gönderilecek paket içeriğinin bulunduğu dizinin adresini alır. Üçüncü parametre paketin byte sayısıdır. Dördüncü parametre yine 0 geçilebilir. Beşinci parametre gönderilecek server’ın ip adresi ve port numarsının bulunduğu sockaadr\_in yapısının adresini alır. Son parametre yine bu sockaddr\_in yapısının bye uzunluğunu almaktadır. sendto fonksiyonu başarısızlık durumunda SOCKET\_ERROR değerine başarı durumunda gönderilen byte sayısına geri döner.

Tipik bir UPD client program şöyle yazılabilir:

#include<stdio.h>

#include<string.h>

#include<stdlib.h>

#include<WinSock2.h>

#include<Windows.h>

#definePORTNO 5050

#defineHOSTNAME "127.0.0.1"

int main(void)

{

WSADATA wsaData;

int result;

structsockaddr\_in sinClient;

int addrLenClient;

SOCKET clientSock;

structhostent \*host;

char buf[1024];

if ((result = WSAStartup(MAKEWORD(2, 2), &wsaData)) != 0) {

fprintf(stderr, "WSAStartup failed: %d\n", result);

exit(EXIT\_FAILURE);

}

if ((clientSock = socket(AF\_INET, SOCK\_DGRAM, IPPROTO\_UDP)) == INVALID\_SOCKET) {

fprintf(stderr, "socket failed: %lu\n", WSAGetLastError());

exit(EXIT\_FAILURE);

}

sinClient.sin\_family = AF\_INET;

sinClient.sin\_port = htons(PORTNO);

sinClient.sin\_addr.s\_addr = inet\_addr(HOSTNAME);

if (sinClient.sin\_addr.s\_addr == INADDR\_NONE) {

if ((host = gethostbyname(HOSTNAME)) == NULL) {

fprintf(stderr, "gethosybyname failed: %lu\n", WSAGetLastError());

exit(EXIT\_FAILURE);

}

memcpy(&sinClient.sin\_addr.s\_addr, host->h\_addr\_list[0], host->h\_length);

}

for (;;) {

printf("Text:");

gets(buf);

if (sendto(clientSock, buf, strlen(buf) + 1, 0, (structsockaddr \*)&sinClient, sizeof(sinClient)) == SOCKET\_ERROR) {

fprintf(stderr, "send failed!\n");

exit(EXIT\_FAILURE);

}

if (!strcmp(buf, "exit"))

break;

}

closesocket(clientSock);

WSACleanup();

return 0;

}

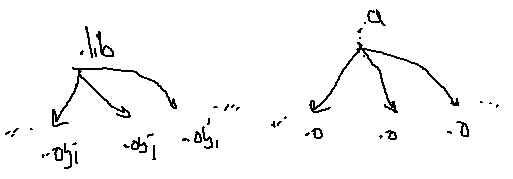
**Kütüphanelerin Oluşturulması ve Kullanılması**

İçerisnde derlenmiş bir biçimde fonksiyonların bulunduğu dosyalara kütüphane (library) denilmektedir. Kütüphaneler statik ve dinamik olmak üzere ikiye ayrılırlar. Windows’ta statik kütüphane dosyalarının uzantıları .lib (library), UNIX/Linux sistemlerinde .a (archieve) biçimindedir. Benzer biçimde dinamik kütüphanelerin de uzantıları Windows sistemlerinde .dll (dynamic link library), UNIX/Linux sistemlerinde .so (shaed object) biçimindedir.

Kütüphane dosyaları herhangi bir dilde yazılıp derlenerek oluşturulabilir. Ancak en çok karşılaşılan durum bunların C/C++ gibi dillerde oluşturulmuş olmasıdır. .NET gibi Java gibi platformlada ara kod sistemi kullanıldığı için buradaki kütüphanelerin oluşturulması ve kullanılması C/C++ gibi doğal kodlu sistemlerden farklıdır. (Örneğin biz Windows’ta bir dll dosyası gördüğümüzde bu dll .NET dünyası için oluşturumuş bir dll olabilir ya da doğal kod içeren bir dll olabilir. Biz kurusumuzda doğal kod içeren kütüphaneleri ele alacağız.)

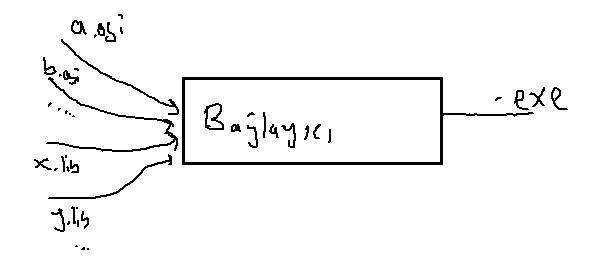
**Statik Kütüphaneler**

Statik kütüphane dosyaları Windows’ta ve UNIX/Linux sistemlerinde aslında obejct modül dosyalarını tutan bir dizin gibidir.



Object modüller içerisinde de derlenmiş fonksiyonlar bulunmaktadır. Biz C’de bir grup fonksiyonu bir kütüphane içerisine yerleştirmek istersek önce onları bir kaynak dosya içerisine yazıp derleriz ve object modül elde ederiz. Sonra bu object modülü kütüphane dosyası içerisine yerleştiririz.

Normalde bağlama işlemine girdi olarak birden fazla object modül ve statik kütüphane dosyası sokulabilmektedir. Örneğin:



Biz C’de bağımsız olarak derlediğimiz object modülleri ve onların kullandığı fonksiyonların bulunduğu kütüphane dosyalarını linker programına girdi olarak verebiliriz. Linker bizim verdiğimiz object modülleri birleştirir. Bu object modüllerden çağrılmış olan ve kütüphaneler içerisinde bulunan fonksiyonları tespit eder. O fonksiyonların içinde bulunduğu object modülleri de kütüphane dosyalarının içinden alarak onu da .exe dosyaya ekler. Böylece biz bir statik kütüphaneden tek bir fonksiyon çağırmış olsak bile o fonksiyonun içinde bulunduğu object modülün tamamı çalıştırılabilen dosyaya eklenecektir. Görüldüğü gibi statik kütüphanelerden bir çağrı yapıldığında çağrılan fonksiyon oradan alınarak çalıştırılabilen dosyanın içerisine yerleştirilmektedir. Böylece program çalışırken artık statik kütüphaneye gereksinim duyulmamaktadır.

Statik kütüphaneler tek parça bir çalıştırılabilen program oluşturduğu için kullanım ve konuşlandırma (deployment) bakımından bazı durumlarda avantaj sağlamaktadır. Ancak statik kütüphaneler çalıştırılabilen dosyaları büyütme eğiliminde olduğu için toplamda disk alanı bakımından bir dezavantaja sahiptir. Ayrıca ileride de ele alınacağı gibi dinamik kütüphaneler kodların ortak kullanımını mümkün hale getirebildiğinden bir optimizasyona da yol açabilmektedir.

Visual Studio IDE’sinde standart C kütüphanesinin default olarak dinamik versiyonu (yani DLL versiyonu) kullanılmaktadır. Ancak komut satırında /MT seçeneği ile ya da Visual Studio IDE’sinde “Project Properties/C-C++/Code Generation/Runtime Library” girişinden statik kütüphane kullanımı sağlanabilmektedir.

Linux’ta gcc ile derleme yaparken yine default durumda standart C kütüphanesinin dinamik versiyonu (.so versiyonu) kullanılmaktadır. Fakat biz açıkça derleme yaparken yine standart C kütüphanesinin static versiyonunun kullanılmasını sağlayabiliriz.

**Anahtar Notlar:** Microsoft’un C ve C++ derleyicisinin ismi cl.exe’dir. Linux’taki ağırlıklı kullanılan C derleyicisi ise gcc’dir. Her iki derleyici de esas olarak komut satırından çalıştırılmaktadır. Visual Studio IDE’si ya da Linux’taki IDE’ler derleme yaparken aslında bu komut satırından çalıştırılan derleyicileri kullanırlar. Yani örneğin Visual Studio’nun ayrı bir C/C++ derleyicisi yoktur. Visual Studio derleme sırasında cl.exe derleyicisini kullanmaktadır. cl.exe default durumda derlemeyi yaptıktan sonra Microsoft’un bağlayıcı programı olan “link.exe” programını ürettiği object kodları ve standart kütüphaneleri girdi yaparak çalıştırmaktadır. Ancak biz istersek /c ya da –c seçeneği (only compile) ile cl.exe’nin link.exe’yi çalıştırmasını engelleyebiliriz. Benzer biçimde aynı çalışma sistematiği Linux’ta da böyledir. Linux’un C derleyicisi gcc’dir. gcc default durumda Linux’un bağlayıcı programı olan “ld” programını çalıştırır. Eğer biz bunu istemiyorsak yine gcc’yi –c seçeneği ile (only compile) çalıştırmalıyız.

Soru: Windows’ta Microsoft’un C derleyici ile komut satırında sample.c programı nasıl derlenir ve link edilir?

Yanıt: Aşağıdaki gibi:

cl sample.c

Tabii biz istesek cl.exe’ye birden fazla kaynak dosya verebiliriz. Bu durumda cl.exe onları bağımsız olarak derler. Tüm object dosyaları link.exe’ye girdi olarak verir. Örneğin:

cl a.c b.c c.c

Soru: Windows’ta cl.exe “link.exe” bağlayıcısını çalıştırırken girdi olarak ona ne verir?

Yanıt: cl.exe’den üretilen object modülleri ve standart C kütüphanesini ve Windows’un bazı API’lerinin bulunduğu temel kütüphaneleri

Soru: Biz gerek Windows’ta gerekse Linux’ta derleme ve bağlama işlemlerini ayrı ayrı yapabilir miyiz?

Yanıt: Evet. Bunun için derleyiciyi –c seçeneğiyle (only compile) çalıştırıp objectg dosyayı elde etmek ve sonra yine “link.exe” ya da “ld” bağlayıcısnı komut satırından çalıştırmak gerekir.

Soru: Visual Studio IDE’sinin cl.exe’yi ve link.exe’yi hangi seçeneklerle çalıştırdığını anlayabilir miyiz?

Yanıt: Evet bunun için “Tools/Options/Projects and Solutions/Build And Run/Ms Build project output verbosity” menüsü kullanılır.

**Statik Kütüphanalerin Oluşturulması**

Statik kütüphane dosyaları Windows’ta Microsoft’un “lib.exe” isimli yardımcı programıyla oluşturulur. lib.exe ile sıfırdan bir static kütüphane dosyası yaratıp içerisine eklemeler şöyle yapabiliriz:

lib /OUT: MyStaticLib.lib A.obj B.obj C.Obj

Statik kütüphanenin içerisindeki object modülleri /LIST seçeneği ile görüntüleyebiliriz:

lib /LIST MyStaticLib.lib

/REMOVE seçeneği ile bir object modülü kütüphaneden atabiliriz. Örneğin:

lib /REMOVE C.obj

Daha sonra kütüphaneye yeni object modüller ekleyebiliriz:

lib MyStaticLib.lib D.obj

Windows’ta Visuyal Studio IDE’si ile statik kütüphanelerin oluşturulması da oldukça kolaydır. Tek yapılacak şey proje yaratılırken (File/New/Project/Win32 Console Application) proje türü olarak “Application Settings”te “Static Library” seçeneğini kullanmaktır. Bu durumda build işlemi yaptığımızda bizim projeye eklediğimiz .c dosyaları derlenir, bunlar object modül haline dönüştürülür. Sonrada lib.exe programı çalıştırılarak .lib dosyası oluşturulur.

UNIX/Linux sistemlerinde statik kütüphane dosyaları “ar” (archive) denilen programla oluşturulup işlemler yapılmaktadır. (Yani UNIX/Linux sistemlerindeki “ar” programını Windows’taki lib.exe’nin karşılığı olarak düşünebiliriz.) ar programı ile bir statik kütüphane dosyası yaratıp (c seçeneği) içerisine object modül ekleme (r seçeneği) işlemi şöyle yapılabilmektedir:

ar cr mystaticlib.a a.o b.o c.o

Eğer zaten var olan bir kütüphaneye object modül ekleyeceksek bu durumda yalnızca “r” seçeneği yeterlidir. Örneğin:

ar r mystaticlib.a d.o

Kütüphanedeki object modülleri görüntülemek için ise “t” seçeneği kullanılır:

ar t mystaticlib.a

ar programının diğer pek çok seçeneği vardır. Bunlar ilgili dokümanlardan öğrenilebilir.

**Statik Kütüphanalerin Kullanılması**

Windows’ta Microsoft’un cl.exe derleyicisi UNIX/Linux sistemlerinde gcc derleyicisi bağlayıcı programları kendi içelerinde çalıştırırken bağlayıcının bazı kütüphane dosyalarına bakmasını sağlamaktadır. Örneğin bu nedenle standartC fonksiyonları bağlayıcı tarafından bulunabilmektedir. Fakat biz kendi kütüphanelerimizin de bağlayı tarafından ele alınmasını isteyebiliriz. Bu işlem çeşitli biçimlerde yapılabilmektedir. Aslında Windows’taki “link.exe” ve UNIX/Linux sistemlerindeki “ld” bağlayıcıları tek tek girdileri komut satırından bizden ister. Bu girdileri yukarıda da belirtildiği gibi komut satırı derleyicileri oluşturmaktadır.

Microsoft’un “cl.exe” derleyicisinde komut satırında .lib dosyaları belirtilirse cl.exe bunu “link.exe” programını çalıştırırken ona komut satırı argümanı yapar. Böylece biz cl.exe ile komut satırında kendi kütüphanemize bakılacak biçimde derlemeyi aşağıdaki yapabiliriz:

cl App.c MyStaticLib.lib

Tabii biz bu işlemi iki aşamada da aşağıdaki gibi yapabilirdik:

cl –c App.c

link App.obj MyStaticLib.lib

Microsoft’un link.exe isimli bağlayıcı programı otomatik olarak standart C fonksiyonlarının ve API fonksiyonlarının bulunduğu kütüphaneleri de link aşamasına dahil etmektedir. Fakat programcı isterse bazı seçeneklerle bu default özelliği kaldırabilir. Microsoft’un “link.exe” isimli bağlayıcısının daha pek çok komut satırı argüman seçenei vardır.

UNIX/Linıux sistemlerindeki statik kütüphane kullanımı tamamen Windows sistemlerine benzemektedir. gcc derleyicisinde komut satırında “.a” uzantılı statik kütüphane dosyaları belirtilirse gcc bunları benzer biçimde “ld” bağlayıcısına girdi yapmaktadır. Örneğin:

gcc –o app app.c mystaticlib.a

UNIX/Linux sistemlerinde statik kütüphane dosyaları kaynak dosya listesinin sonunda belirtilmelidir. Aksi taktirde onun sağındakiler için bu kütüphaneler link işlemine sokulmamaktadır.

Visual Studio IDE’sinde bir statik kütüphaneyi link aşamasına dahil etmek için iki yol izlenebilir:

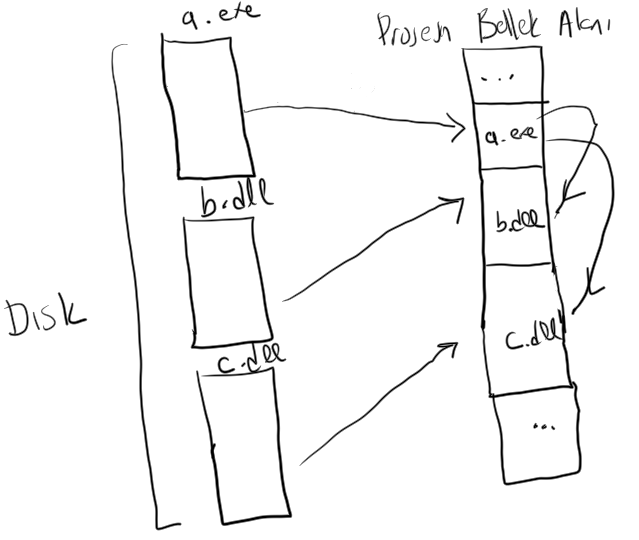
1) .lib dosyası projeye “Add Existing Item” seçeneği ile eklenirse, proje build edilirken bu dosya linker programına girdi olarak verilir.

2) Proje seçeneklerinden “Linker/Input” sekmesine gelinir. “Additional Dependencies” kısmına statik kütüphanenin yalnızca ismi (yol ifadesi olmadan) eklenir. Fakat onun bulunduğu dizin de “Linker/General/Additional Library Directories” kısmına girilir.

**Dinamik Kütüphaneler**

Günüzmüzde dinamik kütüphaneler statik kütüphanelere oldukça yoğun bir biçimde kullanılmaktadır. Windows’ta dinamik kütüphane dosyalarının uzantıları .dll (dynamic link library), UNIX/Linux sistemlerinde ise .so (shared object) biçimindedir. Windows’taki dinamik kütüphanelerin yüklenmesi ve kullanıma hazır hale getirilmesi ile UNIX/Linux’taki dinamik kütüphanelerin yüklenmesi ve kullanıma hazır hale getirilmesi birbirlerinden biraz farklıdır. Fakat genel kullanımları fikir olarak birbirlerine çok benzemektedir.

Bir dinamik kütüphaneden bir fonksiyon kullandığımızda bağlayıcı onu oradan alarak çalıştırılabilen dosyanın içerisine yazmaz. Bağlayıcı yalnızca çalıştırılabilen dosyanın içerisine o programın hangi dinamik kütüphanelerden hangi fonksiyonları kullandığı bilgisini yazar. Böylece bu fonksiyonlar çalıştırılabilen dosyanın içerisinde yer kaplamazlar. İşte işletim sisteminin yükleyicisi dinamik kütüphane kullanan bir programı yüklerken o programın hangi dinamik kütüphaneleri kullandığını belirler. Sonra çalıştırılabilen dosyayla birlikte dinamik kütüphanenin tamamını prosesin bellek alanına yükler. Örneğin a.exe programı b.dll ve c.dll dosyalarını kullanıyor olsun. İşletim sistemi a.exe çalıştırılmak istendiğinde yalnızca o dosyayı değil b.dll ve c.dll dosyalarını da prosesin bellek alanına yüklemektedir.



Artık program çalışırken dll çağırması yapıldığında akış bellekteki dll’in içerisine geçerek çalıştırılır. Tabii bu sistemde programı başka bir makineye taşımak istediğimizde yalnızca çalıştırılabilen programı değil onun kullandığı dinamik kütüphaneleri de o makineye taşımak gerekir.

Pekiyi dinamik kütüphanelerin statik kütüphanelere göre avantajları ve dezavantajları nelerdir?

1) Dinamik kütüphane kullanan uygulamaları başka bir makineye taşırken o dinamik kütüphanelerin de uygun yerlere (ileride açıklanacak) kopyalanması gerekir. Yani dinamik kütüphane kullanan programların konuşlandırılması daha zordur. Halbuki statik kütüphane kullanan programlar yalnızca çalıştırılabilen dosya biçimindedir.

2) Statik link işleminde her çalıştırılabilen dosya statik kütüphaneledeki fonksiyonları tekrar tekrar bulundurmaktadır. Halbuki dinamik kütüphanelerden çağrılar yapıldığında bunlar çalıştırılabilen dosyaların içerisine yazılmazlar. Yani dinamik kütüphanelerin toplamda disk kullanımı konusunda avantajı vardır.

3) Dinamik kütüphane kullanan programlar toplamda bellekte daha fazla yer kaplama eğilimindedir. Çünkü dinamik kütüphaneler bir bütün olarak (yani onların içerisindeki tek bir fonksiyonu çağırsak bile) belleğe yüklenmektedir. Fakat zaten dinamik kütüphanelerin kullanıldığı sistemlerin bellek alanları geniştir ve bu sistemlerin neredeyse hepsi sanal bellek kullanmaktadır.

4) Birden fazla prosesin aynı dinamik kütüphaneyi kullanması durumunda işletim sisteminin sanal bellek ve sayfalama mekanizmaları da devreye girerek bu dinamik kütüphanenin tüm sayfaları aslında fiziksel belleğe defalarca yüklenmek durumunda kalmayabilir. Halbuki statik kütüphanelerde işletim sisteminin bunu ayarlamasının bir yolu yoktur. (Yani örneğin 10 ayrı program statik link işlemi ile printf fonksiyonunu kullanıyor olsun. Bu printf fonksiyonunun kodlarının hepsi bir biçimde ayrı ayrı belleğe yüklenecektir.)

5) Dinamik kütüphanelerin kullanıldığı sistemlerde uygulamada bazı değişiklikler yeniden link işlemi gerekmeden de yapılabilir. Örneğin bazı resimler, yazılar, fonksiyonlar bir dinamik kütüphanenin içerisindedir. Uygulamayı yazan kişi bu dinamik kütüphanenin yeni versiyonunu oluşturup onu hedef makineye taşıyabilir. Böylece program hiç yeniden derleyip link edilmeden değiştirilmiş olur.

**Anahtar Notlar:** Windows’ta doğal kod içeren dll’lerin oluşturulması ve kullanılması ile .NET ortamındaki dll’lerin oluşturulması ve kullanılması tamamen farklı biçimlerde yapılmaktadır. Her iki ortam için de hazırlanmış olan dinamik kütüphanelerin uzantıları .dll biçimindedir. Yani her iki ortamdaki dosyaların uzantılarının .dll olması bunların içeriğinin, oluşturulma ve kullanım biçimlerinin aynı olduğu anlamına gelmemektedir. .NET ortamndaki dll’lerin oluşturulması ve kullanılması çok kolaydır.

**Dinamik Kütüphanelerin Oluşturulması**

Aslında dinamik kütüphanelerin oluşturulması bağlayıcı programlar tarafından yapılmaktadır. Windows’ta Microsoft’un “link.exe” bağlayıcısında /DLL komut satırı seçeneği hedef dosyanın .exe değil .dll yapılacağını belirtir. Ayrıca cl.exe derleyicisinde /LD seçeneği zaten “link.exe” bağlayıcını /DLL seçeneğiyle çalıştırmaktadır. Yani biz doğrudan cl.exe ile derleme yaparken /LD seçeneği ile hedef dosyanın dll olmasını sağlayabiliriz. Örneğin:

cl /DLL mydll.c

Visual Studio IDE’sinde New Project menüsünde “Win32 Console Application” seçilir. “Application Settings”de DLL seçilir. Artık build yapıldığında IDE zaten cl.exe’yi /LD seçeneğiyle çalıştırır ve ürün olarak .dll dosyası elde edilir. (Tabii bir dll dosyasının IDE’den çalıştırılmak istenmesi normal bir durum değildir. Yalnızca bıild işlemi yapılmalıdır.)

Windows’ta dinamik kütüphanelerin içerisindeki fonksiyonlarındışarıdan (örneğin başka bir dll ya da exe dosyasından) çağrılabilmesi için fonksiyonun prototipinin ve/veya tanımlamasının önüne \_\_declspec(dllexport) belirleyicisinin yerleştirilmesi gerekir. \_\_declspec Microsoft derleyicilerine özgü bir eklentidir. \_\_declpec(dllexport) belirleyicisi ilgili fonksiyonun adresini DLL dosyasının export tablosuna yazmaktadır. Örneğin:

#include<stdio.h>

\_\_declspec(dllexport) int Add(inta, intb)

{

returna + b;

}

\_\_declspec(dllexport) int Multiply(inta, intb)

{

returna \* b;

}

DLL’in içerisindeki global değişkenler de dışarıdan kullanılabilirler. Ancak bunların da yine \_\_declspec(dllexport) ile bilridilmesi gerekir. Örneğin:

\_\_declspec(dllexport) int g\_a;

Eğer C++’ta çalışıyorsak bir sınıfın yalnızca belirli bir üye fonksiyonunu export edebiliriz. Ya da sınıfın tüm üye fonkisyonlarını export edebiliriz. Sınıfın public, protected ya da private fonksiyonları export edilebilir. Erişim belirleyicisi ile export işleminin bir ilgisi yoktur. Örneğin:

classSample {

\_\_declspec(dllexport) void Foo();

//...

};

voidSample::Foo()

{

//...

}

Bir sınıfın tüm üye fonksiyonlarını (private ve protected bölümdekiler de dahil olmak üzere) export edebilmek için \_\_declspec(dllexport) belirleyicisi class anahtar sözcüğü ile sınıf isminin arasına getirilir. Örneğin:

class\_\_declspec(dllexport) Sample {

void Foo();

void Bar();

//...

};

voidSample::Foo()

{

//...

}

voidSample::Bar()

{

//...

}

Sınıfın static olmayan veri elemanlarının export edilmesinin bir anlamı yoktur. Çünkü onlar zaten aslında dll’in içerisinde değildir. Fakat sınıfın static veri elemanları dll’in içerisindedir. Onlar da aynı biçimde export edilebilirler. Örneğin:

classSample {

\_\_declspec(dllexport) staticint ms\_a;

\_\_declspec(dllexport) void Foo();

\_\_declspec(dllexport) void Bar();

//...

};

intSample::ms\_a;

voidSample::Foo()

{

//...

}

voidSample::Bar()

{

//...

}

UNIX/Linux sistemlerinde dinamik kütüphane oluşturabilmek için öncelikle derleme işleminde kodun “konumdan bağımsız (position independent)” biçimde oluşturulması gerekir. Çünkü bu sistemler Windows sistemlerinden farklı bir yükleme biçimine sahiptir. Konumdan bağımsız (position Independent) kodların anlamı “UNIX/Linux Sistem Programlama” kursunda ele alınmaktadır. gcc ya da g++ derleyici ile konumdan bağımsız derleme yapmak için komut satırında –f PIC seçeneğinin girilmesi gerekir. Ayrıca derleme işleminden sonra “ld” bağlayıcısının dinamik kütüphane oluşturabilmesi için –shared seçeneği ile çalıştırılşması gerekir (Bu Microsoft’taki /DLL seçeneğine benzemektedir). Mademli gcc’nin kendisi “ld” bağlayıcısını çalıştırmaktadır. O halde –shared seçeneği gcc komut satırında da belirtilebilir. Bu durumda dinamik kütüphane dosyasının oluşturulması aşağıdaki gibi yapılabilmektedir:

gcc -c -fPIC -shared -o mysharedlib.so mysharedlib.c

UNIX/Linux sistemlerinde konumdan bağımsız kod tekniği nedeniyle Microsoft sistemlerinde olduğu gibi fonksiyonu export etmee kavramı yoktur. Yani biz dinamik kütüphanedeki tüm fonksiyonları ve statik dataları dışarıdan kullanabiliriz.

**Dinamik Kütüphanelerin Kullanılması**

Windows’ta bir dinamik kütüphaneden çağırma yapan programı link ederken link aşamasında “o dinamik kütüphanenin import kütüphanesi” denilen bir .lib dosyasını kullanmamız gerekir. Yani biz Windows’ta bir dll oluştururken aslında bize ürün olarak iki dosya verilmektedir: Bir .dll dosyası ve bir de .lib dosyası. Asıl fonksiyon kodları .dll dosyası içerisindedir. Buradaki .lib dosyasına “dll’in import kütüphanesi” denir. Bu .lib dosyasının içerisinde fonksiyonların kodları yoktur. O .dll içerisinde hangi fonksiyonların olduğu, onların sıra numaraları (ordinal number) ve dekore edilmiş ve edilmemiş isimleri vs. bulunur. Yani import kütüphanesi adeta dll’in bir dizin dosyası gibidir. O halde örneğin aşağıdaki gibi bir derleme işleminden biz iki dosya elde ederiz:

cl /DLL mydll.c

Bu dosyalar mydll.dll ve mydll.lib dosyalarıdır.

Bir DLL’i kullanmak için kullanan programın link aşamasında DLL’in import kütüphanesini link işlemine dahil etmesi gerekir. Bu dahil etme işlemi taamamen statik kütüphanelerdeki gibidir. Yani statik kütüphaneler konusunda da belirtildiği gibi bu işlem iki biçimde yapılabilmektedir:

1) Import kütüphane dosyası projeye “Add Existing Item” seçeneği ile eklenirse, proje build edilirken bu dosya linker programına girdi olarak verilir.

2) Proje seçeneklerinden “Linker/Input” sekmesine gelinir. “Additional Dependencies” kısmına import kütüphanesinin yalnızca ismi (yol ifadesi olmadan) eklenir. Fakat onun bulunduğu dizin de “Linker/General/Additional Library Directories” kısmına girilir.

Ancak bir dinamik kütüphane kullanılırken dinamik kütüphane içerisindeki fonksiyonların prototipleri bulundurulmalı (tercihen bir başlık dosyasında) ve bu fonksiyonların prototiplerinin başına \_\_declspec(dllimport) bildirimi yerleştirilmelidir. Örneğin:

\_\_declspec(dllimport) int Add(int a, int b);

Aslında \_\_declspec(dllimport) belirleyicisi DLL’deki fonksiyonları kullanırken mutlak anlamda gerekli değildir. Fakat bu belirleyicinin yerleştirilmesi daha etkin kod üretilmesine yol açmaktadır.

**Anahtar Notlar:** Eğer bu belirleyiciyi derleyici görmezse derleyici çağrı sırasında doğrudan CALL komutu kullanır. Bu durumda bağlayıcı bu komutları düzelterek onları dolaylı CALL komutu haline getirmektedir. İşte \_\_declspec(dllimport) işin başında derleyicinin daha etkin kod üretmesini sağlar.

Sonuç olarak DLL içerisindeki fonksiyonları dışarıdan kullanabilmek için \_\_declspec(dllexport), dışardan kullanırken \_\_declspec(dllimport) belirleyicisinin kullanılması gerekir. Bu işlemi kolaylaştırmak için şöyle bir yol tavsiye edilmektedir: DLL’i hazırlayan kişi dışarıdan kullanılacak (export edilecek) fonksiyonların prototiplerini bir başlık dosyasında toplar. Ancak burada #ifdef işlemiyle aşağıdaki gibi bir düzenleme yapar:

#ifndef MYDLL\_H\_

#defineMYDLL\_H\_

#ifdefDLLEXPORT

#defineDLLSPEC \_\_declspec(dllexport)

#else

#define DLLSPEC \_\_declspec(dllimport)

#endif

/\* Function Prototypes \*/

DLLSPECint Add(int a, int b);

DLLSPECint Multiply(int a, int b);

/\* extern data Declarations \*/

DLLSPECexternint g\_a;

#endif

Burada görüldüğü gibi DLLEXPORT isimli bizim uydurduğumuz makro eğer daha önce define edilmişse yine bizim uydurduğumuz DLLSPEC yerine önişlemci \_\_declspec(dllexport) belirleyicisini yerleştirecektir. Eğer bu makro daha önce define edilmemişse bu sefer DLLEXPORT yerine önişlemci \_\_declspec(dllimport) belirleyicisini yerleştirir. Bu durumda bu başlık dosyası hem DLL oluşturulurken hem de kullanılırken include edilir. Örneğin bu durumda DLL’i oluşturan kod şöyle olacaktır:

#defineDLLEXPORT

#include<stdio.h>

#include"MyDll.h"

DLLSPECint g\_a;

DLLSPECint Add(inta, intb)

{

returna + b;

}

DLLSPECint Multiply(inta, intb)

{

returna \* b;

}

DLL’i kullanan kod da şöyle olacaktır:

#include<stdio.h>

#include"MyDll.h"

int main(void)

{

int result;

result = Add(10, 20);

printf("%d\n", result);

result = Multiply(10, 20);

printf("%d\n", result);

return 0;

}

Tabii DLL’i kullanan kodda import kütüphanesinin link aşamasına dahil edilmesi gerekir.

Görüldüğü gibi bir DLL oluşturulurken o DLL’i kullanacak kişi üç dosyaya gereksinim duymaktadır:

1) DLL dosyasının kendisine (örneğimizdeki MyDll.dll)

2) DLL’in import kütüphanesine (örneğimizdeki MyDll.lib)

3) DLL içerisindeki fonksiyonların ya da sınıfların bildirimlerinin \_\_declspec(dllimport) ile yapıldığı bir başlık dosyasına. (örneğmizdeki MyDll.h)

Dinamik kütüphane kullanan bir program çalıştırılırken kullandığı DLL’in de o makinede bulunuyor olması gerekir. Fakat kullanılan DLL hangi dizinde olmalıdır? İşte Windows sırasıyla DLL’leri bazı dizinlerde aramaktadır. Ancak bu konuda Windows’un versiyonları arasında küçük bazı farklılıklar vardır. Özellikle bu farklılık 32 bit Windows sistemleriyle 64 bit Windows sistemleri arasında daha belirgindir. Burada bazı küçük ayrıntıları göz ardı ederek bu arama dizinlerini sırasıyla aşağıda belirtiyoruz:

1) Programın .exe dosyasının bulunduğu dizin

2) CreateProcess fonksiyonunu uygulayan prosesin (yani exe’yi çalıştıran programın. Bu program Windows’un komut satırı ya da masaüstü (explorer.exe) de olabilir.) çalışma dizini.

3) Windows’un altındaki System32 dizini (bu dizin GetSystemDirectory API fonksiyonuyla elde edilebilir.)

4) Windows’un altındaki System dizini.

5) Windows’un dizinini kendisi (bu fizin GetWindowsDirectory API fonksiyonuyla elde edilebilir)

6) CreateProcess fonksiyonunu uygulayan prosesin PATH çevre değişkeni ile belirlediği dizinler sırasıyla

Windows’ta DLL’lerin yerleştirildiği iki tipik yer vardır: Birincisi geleneksel olarak Windows’un System32 dizinidir. Kurulum programları paylaşılan (yani çok proses tarafından kullanılan DLL’leri buraya çekebilir. Ancak bu yöntem gittikçe daha az tercih edilir olmuştur. Bakınız: “Dll Cehennemi”) Microsoft uzunca süredir her uygulamanın kendi DLL’lerini .exe dosyasının bulunduğu kurulum dizinine çekmesini tavsiye etmektedir.

UNIX/Linux sistemlerinde Windows sistemlerinde olduğu gibi dinamik kütüphanelerin import kütüphaneleri yoktur. Bu sistemlerde dinamik kütüphaneler kullanılırken bizzat dinamik kütüphane dosyasının (yani .so dosyasının) kendisi link işlemine sokulur. (Yani adeta UNIX/Linux sistemlerinde .so dosyasının kendisinin aynı zamanda sanki Windows sistemlerindeki import kütüphanesinin yerine de geçtiği düşünülebilir.) Böylece bağlayıcı program hangi fonksiyonların dinamik kütüphane içerisinde olduğunu buradan anlamaktadır. Örneğin dinamik kütüphane aşağıdaki dosyadan oluşuyor olsun:

/\* mysharedlib.c \*/

#include<stdio.h>

#include"mysharedlib.h"

int g\_a;

int add(inta, intb)

{

returna + b;

}

int multiply(inta, intb)

{

returna \* b;

}

Dinamik kütüphanenin derlenmesi şöyhle yapılmalıdır:

gcc -fPIC -shared -o mysharedlib.so mysharedlib.c

Buradaki mysharedlib.h başlık dosyası aşağıdaki olabilir:

/\* mysharedlib.h \*/

#ifndef MYSHAREDLIB\_H\_

#defineMYSHAREDLIB\_H\_

/\* Function Prototypes \*/

int add(int a, int b);

int multiply(int a, int b);

/\* external data declarations \*/

externint g\_a;

#endif

Dinamik kütüphaneyi kullanan program da şöyle olabilir:

/\* sharedlibusage.c \*/

#include<stdio.h>

#include"mysharedlib.h"

int main(void)

{

int result;

result = add(10, 20);

printf("%d\n", result);

result = multiply(10, 20);

printf("%d\n", result);

printf("%d\n", g\_a);

return 0;

}

Dinamik kütüphaneyi kullanan program da şöyle derlenebilir:

gcc -o sharedlibusage sharedlibusage.c mysharedlib.so

Pekiyi UNIX/Linux sistemlerinde dinamik kütüphaneyi kullanan programı çalıştırırken dinamik kütüphane dosyası (yani .so dosyası) nerede bulunmalıdır? İşte bunun da tıpkı Windows sistemlerinde olduğu gibi bir aranma prosedürü vardır. Bu aranma prosedürü bazı detaylar atlanarak şöyle özetlenebilir:

1) Dinamik kütüphane dosyasının ELF formatı içerisindeki DR\_RPATH dizinine bakılır.

2) exec işlemini yapan prosesin LD\_LIBRARY\_PATH isimli çevre değişkeninde belirtilen dizinlere sırasıyla bakılır.

3) /lib dizinine bakılır

4) /usr/lib dizinine bakılır

Buradaeğer dinamik kütüphane paylaşılacaksa onun /usr/lib ya da /lib içerisine çekilmesi daha uygun olur. /lib dizini çok daha temel kütüphaneleri, /usr/lib dizini ise daha çok uygulama programlarının kullandığı kütüphaneleri barındırmaktadır. Eğer dinamik kütüphane paylaşılmayacaksa LD\_LIBRARY\_PATH çevre değişkenini ayarlamak daha iyi bir yoldur. Örneğin:

export LD\_LIBRARY\_PATH=.

./sharedlibusage

**Dinamik Kütüphanelerin Dinamik Yüklenmesi**

Hem Windows sistemlerinde hem de UNIX/Linux sistemlerinde dinamik kütüphaneler programın çalışma zamanı sırasında dinamik yüklenebilmektedir. Daha önce gördüğümüz dinamik kütüphane kullanma biçimlerinde dinamik kütüphaneler işin başında program belleğe yüklenirken işletim sisteminin yükleyicisi tarafından yükleniyordu. Pekiyi dinamik kütüphanelerin programın çalışma zamanı sırasında yüklenmesine neden gereksinim duyulabilmektedir? Bunun birkaç nedeni olabilir:

1) Normal dinamik kütüphane kullanımında programın kullandığı tüm dinamik kütüphaneler yükleme sırasında işletim sisteminin yükleyicisi tarafından yüklenmektedir. Bu da yükleme zamanını uzatabilmektedir. Halbuki bazı kütüphanelerin gereksinim duyulduğu zamanda yüklenmesi bu zamanı azaltabilir.

2) DLL’ler prosesin bellek alanında yer kaplamaktadır. Halbuki bazı DLL’lerin içerisindeki fonksiyonlara nadiren gereksenimn duyuluyor olabilir. (Ya da bazı koşullarda gereksinim duyuluyor olabilir). Bu tür DLL’lerin dinamik yüklenmesi daha iyi bir tekniktir. Örneğin IDE’lerde olduğu gibi pluag-in ya daa add-in içeren mimarilerde bu eklentileri yazan kişiler bunu bir DLL’de toplarlar ana yazılım gereksinim duyulduğunda bu DLL’leri yükleyebilmektedir.

Windows’ta dinamik kütüphanelerin programın çalışma zamanı sırasında yüklenmesi şöyle gerçekleştirilmektedir:

1) Önce LoadLibrary (ya da onun biraz daha geliştirilmiş biçimi olan LoadLibraryEx) API fonksiyonu ile dinamik kütüphane belleğe yüklenir.

HMODULE WINAPI LoadLibrary(

\_\_in LPCTSTR lpFileName

);

Fonksiyon yüklenecek dinamik kütüphane dosyasının yol ifadesini parametre olarak alır. (Eğer yol ifadesi ‘\’ ya da ‘/’ içermiyorsa daha önce bahsedilen yerlerde sırasıyla arama yapılmaktadır.). Fonksiyon geri dönüş değeri olarak DLL’in belleğe yüklendiği adresi bize verir. Geri dönüş değerinin HMODULE türünden olması 16 bit Wİndows zamanlarından kalmıştır. HMODULE türü void \* olarak typedef edilmiştir. Örneğin:

HMODULE hModule;

if ((hModule = LoadLibrary("MyDll.Dll")) == NULL)

ExitSys("LoadLibrary", EXIT\_FAILURE);

Görüldüğü gibi dinamik kütüphanelerin dinamik yüklenmesi için “import kütüphanesi”ne gereksinim duyulmamaktadır.

2) Dinamik kütüphane yüklendikten sonra onun içerisindeki fonksiyonun ya da global bir nesnenin adresi GetProcAddress fonksiyonuyla elde edilir:

FARPROC WINAPI GetProcAddress(

\_\_in HMODULE hModule,

\_\_in LPCSTR lpProcName

);

Fonksiyonun birinci parametresi bizden LoadLibrary fonksiyonundan elde edilen kütüphanenin belleğe yüklenme adresini ister. İkinci parametre adresini elde edeceğimiz fonksiyonun ya da global nesnenin ismidir. Burada ismi girilen fonksiyon ya da adresin DLL tarafından export edilmiş olması (yani \_\_declspec(dllexport) bildirimi ile derlenmiş olması) gerekmektedir. Çünkü GetProcAddres ilgili sembolü “DLL’in export tablosu” denilen bir yerinde arar. Ancak export edilen semboller “export tablosu”nda bulunmaktadır. Ayrıca burada verilen isimlerin dekore edilmiş isimler olduğuna dikkat edilmelidir. Fonksiyonların ya da global değişkenlerin isimleri C derleyicileri tarafından değiştirilerek (dekore edilerek) amaç koda ve dolayısıyla DLL dosyasına yazılmaktadır. Bu isimlerin dekore edilmiş son halleri DUMPBIN programı ile şöyle elde edilebilir:

DUMPBIN /EXPORTS MyDll.dll

Fonksiyon geri dönüş değeri olarak bize o fonksiyonun adresini verir. Geri dönüş değerindendeki FARPROC bir fonksiyon typedef türüdür. Programcının tür dönüştürmesi ile bu adresi kendi türün e dönüştürmesi gerekir. (Anımsanacağı gibi void \* türü C’de ve C++’ta fonksiyon adreslerini tıtamamaktadır.) FARPROC türü aşağıdaki gibi typedef edilmiştir (tabii bunun bizim için bir önemi yok):

typedef int (\_\_stdcall \*FARPROC)();

Örneğin:

typedef int(\*PADD)(int, int);

if ((padd = (PADD) GetProcAddress(hModule, "Add")) == NULL)

ExitSys("GetProcAddress", EXIT\_FAILURE);

Benzer biçimde global değişkenlerin de adresleri GetPrcAddres fonksiyonu ile elde edilebilir:

typedef int \*GDATA;

GDATA gd;

...

if ((gd = (GDATA)GetProcAddress(hModule, "g\_a")) == NULL)

ExitSys("GetProcAddress", EXIT\_FAILURE);

printf("%d\n", \*gd);

3) Artık fonksiyonlar ya da global değişkenler program içerisinden kullanılabilir. Örneğin:

result = padd(10, 20);

printf("%d\n", result);

4) Kullanım bittikten sonra dinamik kütüphane FreeLibrary API fonksiyonuyla boşaltılabilir:

BOOL WINAPI FreeLibrary(

\_\_in HMODULE hModule

);

Ayrıca biz DLL’i yükleme zamanında yüklenecek biçimde kullanıyor olsak bile onu yine istediğimz zaman FreeLibrary API fonksiyonuyla bellekten boşaltabiliriz.

Fonksiyon parametre olarak DLL’in bellekteki yüklenme adresini almakatdır.

Windows için örnek kullanım programı şöyle olabilir:

#include<stdio.h>

#include<Windows.h>

void ExitSys(LPCSTR lpszMsg, int status);

typedefint(\*PADD)(int, int);

typedefint(\*PMUL)(int, int);

typedefint \*GDATA;

int main(void)

{

HMODULE hModule;

PADD padd;

PMUL pmul;

GDATA gd;

int result;

if ((hModule = LoadLibrary("MyDll.Dll")) == NULL)

ExitSys("LoadLibrary", EXIT\_FAILURE);

if ((padd = (PADD) GetProcAddress(hModule, "Add")) == NULL)

ExitSys("GetProcAddress", EXIT\_FAILURE);

result = padd(10, 20);

printf("%d\n", result);

if ((pmul = (PMUL)GetProcAddress(hModule, "Multiply")) == NULL)

ExitSys("GetProcAddress", EXIT\_FAILURE);

result = pmul(10, 20);

printf("%d\n", result);

if ((gd = (GDATA)GetProcAddress(hModule, "g\_a")) == NULL)

ExitSys("GetProcAddress", EXIT\_FAILURE);

printf("%d\n", \*gd);

FreeLibrary(hModule);

return 0;

}

void ExitSys(LPCSTRlpszMsg, intstatus)

{

DWORD dwLastError = GetLastError();

LPTSTR lpszErr;

if (FormatMessage(FORMAT\_MESSAGE\_ALLOCATE\_BUFFER | FORMAT\_MESSAGE\_FROM\_SYSTEM, NULL, dwLastError,

MAKELANGID(LANG\_NEUTRAL, SUBLANG\_DEFAULT), (LPTSTR)&lpszErr, 0, NULL)) {

fprintf(stderr, "%s: %s", lpszMsg, lpszErr);

LocalFree(lpszErr);

}

exit(status);

}

UNIX/Linux sistemlerinde de dinamik kütüphaneelerin dinamik yüklenmesi Windows sistemlerindekiyle çok benzer biçimde yapılmaktadır. İşlemler sırasıyla şu adımlardan geçilerek yapılabilir:

1) Önce dlopen fonksiyonuyla dinamik kütüphane belleğe yüklenir (dlopen Windows’taki LoadLibrary fonksiyonuna benzetilebilir.)

#include <dlfcn.h>

void \*dlopen(const char \*filename, int flag);

Fonksiyonun birinci parametresi yüklenecek dosyanın yol ifadesini almaktadır. İkinci parametre yüklemeyle ilgili bazı ayrıntılar için düşünülmüştür. Burada bu ayrıntılar üzerinde durulmayacaktır. Bu parametre RTLD\_LAZY oalrak geçilebilir. (lazy load işlemi “UNIX/Linux Sistem Programlama” kurslarında ele alınmaktadır). Fonksiyonun geri dönüş değeri dinamik kütüphanenin yüklendiği adresi belirtir. Dinamik yükleme fonksiyonları libdl kütüphanesinin içerisinde olduğu için link işlemine –ldl seçeneğinin eklenmesi gerekir. Örneğin:

void \*dl;

if ((dl = dlopen("./mysharedlib.so", RTLD\_LAZY)) == NULL) {

fprintf(stderr, "cannot load library!\n");

exit(EXIT\_FAILURE);

}

2) Dinamik kütüphane içerisindeki fonksiyonun ya da global değişkenin adresi dlsym fonksiyonuyla elde edilir. (Bu fonksiyonu Windows sistemlerindeki GetProcAddress fonksiyonuna benzetebiliriz.):

#include <dlfcn.h>

void \*dlsym(void \*handle, const char \*symbol);

Fonksiyonun birinci parametresi dinamik kütüphanenin bellekteki yüklenme adresi, ikinci parametresi ilgili sembolün ismini belirtir. Buradaki isimler de derleyici tarafından dekore edilmiş isimler olabilir. İsimlerin dekaore edilmiş biçimlerini objdump, readelf ya da nm isimli utility’ler ile elde edebiliriz. Örneğin:

readelf –s mysharedlib.so

ya da:

nm mysharedlib.so

Fonksiyonun başarı durumunda sembolün adresine, baaşarıszlık durumunda NULL adrese geri döner. Örneğin:

if ((padd = dlsym(dl, "add")) == NULL) {

fprintf(stderr, "cannot get address!..\n");

exit(EXIT\_FAILURE);

}

**Anahtar Notlar:** Aslında standartlar bağlamında bu fonksiyonun ve GetProcAddress fonksiyonlarının tasarımları yanlıştır. Zira C ve C++’ta fonksiyon adreslerinden data adreslerine, data adreslerinden de fonksiyon adreslerine tür dönüştürme operatörüyle bile dönüştürme tanımlanmamıştır.

3) Artık adresi elde edilmiş olan fonksiyon çağrılabilir. Örneğin:

result = padd(10, 20);

printf("%d\n", result);

4) Nihayet dinaik kütüphane dlclose fonksiyonuyla kapatılır:

#include <dlfcn.h>

int dlclose(void \*handle)

Fonksiyon başarı durumunda 0, başarısızlık durumunda -1 değerine geri döner.

UNIX/Linux sistemlerinde örnek bir kullanım kodu şöyle olabilir:

#include<stdio.h>

#include<stdlib.h>

#include<dlfcn.h>

typedefint(\*PADD)(int, int);

int main(void)

{

void \*dl;

PADD padd;

int result;

if ((dl = dlopen("./mysharedlib.so", RTLD\_LAZY)) == NULL) {

fprintf(stderr, "cannot load library!\n");

exit(EXIT\_FAILURE);

}

if ((padd = dlsym(dl, "add")) == NULL) {

fprintf(stderr, "cannot get address!..\n");

exit(EXIT\_FAILURE);

}

result = padd(10, 20);

printf("%d\n", result);

dlclose(dl);

return 0;

}

**Dinamik Kütüphanelerin Dinamik Kütüphaneleri Kullanması Durumu**

Gerek Windows sistemlerinde gerekse UNIX/Linux sistemlerinde bir dinamik kütüphane diğer bir dinamik kütüphaneyi kullanıyor olabilir. Örneğin Windows’ta biz B.c dosyasını derleyerel B.DLL oluşturmak isteyelim. Ancak B.c ddosyasında da A.dll’deki fonksiyonlardan faydalanmış olalım. O halde komut satırında B.c dosyasını şöyle derleriz:

cl /DLL B.c A.lib

Burada A.lib A.DLL kütüphanesinin import kütüphanesidir. Benzer işlem UNIX/Linux sistemlerinde de benzer biçimde yapılır:

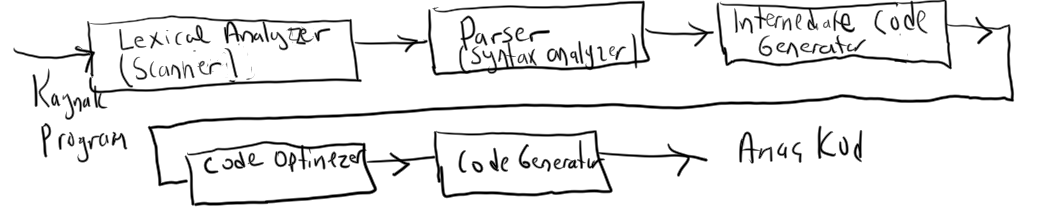
gcc -c -fPIC -shared -o b.so b.c a.so

Burada b.c dosyası dinamik kütüphane oluşturacak şekilde derlenmek istenmiştir. Fakat b.c içerisinden a.so’daki fonksiyonlar kullanılmıştır.

Bir dinamik kütüphanenin diğerini kullanması durumunda işletim sistemi yükleme sırasında söz konusu tüm dinamik kütüphaneleri özyinelemeli olarak yükler. Böylece bizim de programı konuşlandırırken tüm bu dinamik kütüphaneleri çalıştırılabilen dosyayla birlikte hedef makineye taşımamız gerekir.

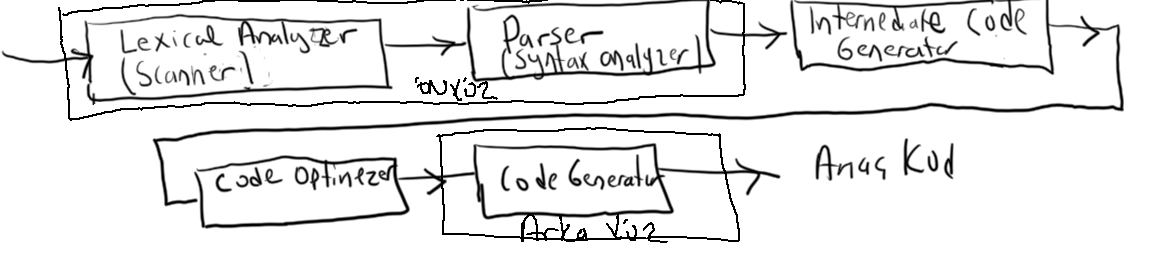
**Derleyicilerin Çalışma Mekanizması**

Bir derleyici tipik şu aşamalardan geçerek derleme işlemini yapar:

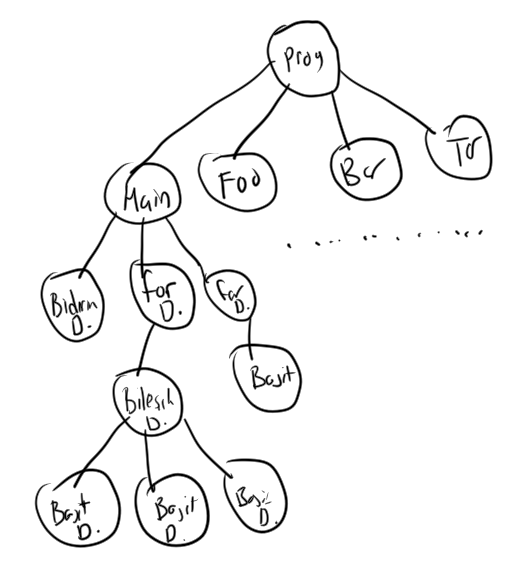


Kaynak kod derleyicinin “lexical anaylzer” ya da “scanner” ya da “tokenizer” denilen modülü tarafından ele alınır. Bu modül kayunak kodu atomlarına ayırmaktadır. Daha sonra “parser” denilen modül devreye girer. Bu modül kaynak programı bir ağaç olarak ifade eder. Bu ağaca “parse tree” denilmektedir. “Parse tree” programın işlenebilir halidir. Yani program bir yazı olmaktan çıkmış bir veri yapısı olarak ifade edilmiştir. Daha sonra bu ağaç özyinelemeli dolaşılır ve ağaçtaki elemanlar için gerçek komutları değil fakat onları temsil eden arakodlar (intermediate codes) üretilir. Bu modüle “intermediate code generator” denilmektedir. Daha sonra bu ara kodlar optimize edilir. Nihayet gerçek makine komutları “Code Generator” modülü tarafından üretilir.

Derleyicilerin kaynak program üzerinde işlem yaptığı modüllere “ön yüz (frontend)”, makine kodu ürettiği modüllere “arka yüz (back end)” ve diğer aradaki modüllere de “orta yüz (middle end)” denilmektedir. Ön yüz kaynak dile arka yüz hedef dile bağlıdır. Derleyicileri yazanlar önyüzlerin çıktılarını, parse tree çıktılarını vs. standart hale getirmeye çalışırlar. Böylece derleyicinin port edilmesi kolaylaşır. Şöyle ki: Bir firma düşünelim bu firma N tane dili M tane makine dili için derleyen derleyiciler yazmak istesin. Normalde bunun için kaç derleyici yazması gerekir? Yanıt: N \* M tane. Halbuki firma N tane dil için önyüz, M tane dil için de arka yüz yazsa toplamda N + M tane faaliyetle bunları birleştirebilir. Örneğin biz bir dil tasarlamış olalım. Ancak bunun derleyicisi için gcc derleyicisinden faydalanmak isteyelim. Mademki gcc işinönemli kısmını yapmaktadır. O halde biz gcc için kendi dilimizi atomlarına ayıran ve onun parse tree modülüne veren bir önyüz yazabiliriz. Kod optimizasyonunu ve kod üretimini gcc’nin modülleri yapabilir. Bu faaliyete gcc için “frontend” yazma faaliyeti denilmektedir. Şimdi biz yeni bir işlemci için C derleyicisi yazmak isteyelim. Ve bunun için gcc derleyicisinden faydalanmak isteyelim. Bu durumda gcc’nin C frontend’i doğrudan kullanılabilir. Bunun için bizim gcc için bir “backend” yazmamız gerekir.



Parse Tree nasıl bir ağaçtır? Bir program bütünden yalına doğru özyinelemeli olarak parçalara ayrılabilir. Örneğin şöyle bir dil tasarlamış olalım: Dilimiz fonksiyonlardan oluşsun. Fonksiyonlar deyimlerden oluşsun. Dilimizde bildirim deyimi, bileşik deyim, basit deyim ve for deyimi olsun. Bu durumda yazdığımız program aşağıdaki bir ağaçla temsil edilebilir:



Görüldüğü gibi program bir metin biçiminden çıkartılıp işlenebilir bir ağaç yapısına dönüştürülmüştür. Sonra bu ağaç yapısı dolaşılıp ara kod üretilecek, sonra o kod optimize edilip gerçek kod üretilecektir. Parse Tree derleyiciler için önemli veri yapılarından biridir.

Yorumlayıcıların (interpreters) temel aşamaları da derleyicilerle aynıdır. Ancak bunların arkayüzü yoktur. Çünkü yorumlayıcı hedef kod üretmeden programı doğrudan çalıştırır. Fakat yorumlayıcılar da kod optimizasyonu yapabilmektedir. Bazı yorumlayıcılar derleyicilerde olduğu gibi önce bir arakod üretir. Sonra bu arakodu adeta bir sanal makinede çalıştırır. Fakat bu arakodu programcı hiç görmeyebilir. Bazı yorumlayıcılar aynı zamanda istendiğinde hedef kod üretebilme yeteneğine de sahiptir.

**Derleyicilerin Kod Optimizasyonları**

Derleyiciler arasındaki en önemli kalite göstergelerinden biri “kod optimizasyonu”dur. Kod optimizasyonu “kodun işlevini değiştirmeden onu daha etkin biçimde çalıştırmak için yeniden düzenleme süreci”dir.

**Anahtar Notlar:** Aslında buradaki optimizasyon terimi yanlış uydurulmuştur. Optimizasyon eniyileme demektir. Halbuki derleyicilerin böyle bir iddiası yoktur. Derleyiciler kodları onların işlevleri değişmemek üzere yeniden düzenleyerek daha iyi hale getirirler. Bu süreç bir optimizasyon süreci değil iyileştirme (improvement) sürecidir. Fakat optimizasyon terimi bu yanlışlık bilinmesine rağmen yaygın olarak kabul görmüştür ve kullanılmaktadır.

Kod optimizasyonu ile ilgili önemli noktalar şunlardır:

- Optimizasyon kaynak kod üzerinde değil üretilen kod üzerinde yapılan bir faaliyettir. Yani optimizasyon ile bizim kaynak kod dosyamızda bir değişilik olmaz.

- Optimizasyon sırasında kodda hiçbir anlam değişikliği olmaması gerekir. Yani optimiza edilmemiş kod ile edilmiş kod tamamen aynı davranışı göstermelidir.

- Optimizasyon genel olarak “hız” ya da “kod büyüklüğü” temel alınarak yapılabilir. Fakat baskın ölçüt ve default durum “hız”optimizasyonudur. Hatta derleyiciler kodun daha hızlı çalışmasını sağlarken kodu büyütebilirler.

- Pek çok derleyicide optimizasyon çeşitli derleme seçenekleri (switch’ler) ile kontrol altına alınabilmektedir. Yani biz programcı olarak derleyicilere hangi tarz optimizasyonu yapması gerektiğini, hangilerini yapmaması gerektiğini söyleyebiliriz. Hatta programcı isterse optimizasyonu tamamen de kapatabilir. Bu durumda derleyici kodu optimize etmeye çalışmaz.

- Kod optimizasyonu C/C++ standartlarının ele aldığı bir konu değildir. Ancak standartlarda “derleyicinin programdaki semantik işlevin değişmemesi şartıyla kodu yeniden düzenleyerek derleyebileceği” belirtilmiştir.

- Derleyicilerin kod optimizasyonları halen tek akışlı ya da tek thread’li sistemlere göre yapılmaktadır. Yani derleyici kodu optimize ederken programda tek bir akış varmış gibi düşünür. Örneğin derleyici global bir değişken kullanılırken onun başka bir thread’te de kullanılıyor olduğuyla ilgilenmez. Dolayısıyla çok thread’li uygulamalarda programcı bu durumu göz önüne almalıdır. (Tabii çok thread’li uygulamalarda yalnızca global değişkenlerin kullanıldığı durumda böyle bir potansiyel tehlike oluşmaktadır. Bunlara da volatile gibi anahtar sözcüklerle ya da diğer bazı yöntemlerle önlem alınabilmektedir.)

**Derleyicilerin Yaptığı Tipik Optimizasyonlar**

Bu bölümde derleyicilerin tipik yaptığı kod optimizasyonları başlıklar halinde ele alınacaktır. Bu optimzasyonların sistem programcısı tarafından bilinmesi önemlidir. Çünkü böylece derleyicilerin hangi optimizasyonları bizim yapabildiği ve hangi maaliyetlerle yapabildiğini anlayabilir. Böylece gerektiğinde bu optimizasyon sürecine katkıda bulunabilir ya da süreci daha iyi kontrol altına alabilir. Optimizasyonlar yerel (local), global (global) ve döngü optimizasyonları olarak da sınıflandırılabilmektedir. Yerel optimizasyonlar kodun küçük bir bölümü üzerinde etkili olur. Global optimizasyonlar birden fazla fonksiyonu içine alacak biçimde geniş bölgeyi etkileyecek biçimde yapılır. Döngü optimizasyonları döngü içlerini optimiza etmeyi hedefler. Burada optimizasyonlar karışık sırada ele alınacaktır.

**1) Ortak Alt İfadelerin Elimine Edilmesi (Common Subexpression Elimination):** Bir gurp işlemde ortak olan bazı alt ifadelerin her defasında yeniden yapılması yerine derleyici bunların sonuçlarını bir yazmaçta ya da geçici değişkende tutup onları kullanabilir. Örneğin:

a = x + y + 10;

b = x + y + 20;

c = x + y + 30;

Burada derleyici her defasında x + y toplamını hesaplamak yerine bu toplamı geçici bir değişkende ya da yazmaçta tutarak oradan alıp kullanabilir. (Örneğin sınıfta Microsoft C derleyicisi ile optimizasyon kapalı ve açık durumda yapılan derlemede üretilen kodun sembolik makine dili çıktısına bakılmıştır. Ve derleyicinin optimizasyon sırasında x + y toplamını ECX yazmacında tutarak oradan kullandığı görülmüştür. Fakat optimizasyon seçeceği kapatıldığında derleyici hep yeniden x + y toplamını hesaplamıştır.)

**2) Döngü içerisinde Değişmeyen İfadelerin Döngü Dışına Çıkartılması (Loop Invariant):** Bu optimizasyon temasında döngü içerisindeki ifade döngü içerisinde kullanılmıyorsa döngünün dışına alınır. Örneğin:

for (i = 0; i < 100; ++i) {

total += i;

x = 10;

}

Burada x = 10 işleminin döngünün çalışmasıyle bir ilgisi yoktur. Yani bu işlemin her yinelemede yeniden yapılmasına gerek yoktur. Derleyici kodu aşağıdaki gibi ele alabilir:

for (i = 0; i < 100; ++i) {

total += i;

}

x = 10;

Microsoft’un C derleyicisinde yukarıdaki kod analiz edildiğinde optimizasyon seçeneği kapalı olduğunda derleyicinin x = 10 ifadesini döngünün dışına çıkarmadığı açık olduğu durumda çıkardığı görülmüştür.

Fonksiyon çağrıları (hele ki derleyici fonksiyonun içini göremiyorsa) döngü dışına çıkartılamaz. Örneğin:

for (i = 0; i < strlen(str); ++i) {

...

}

Burada programcı derleyicinin optimazasyon yaparak strlen fonksiyonunu bir kez çağırmasını beklememelidir. Derleyiciler standart C fonksiyonlarının ne yaptığını bilmediği için onlara normal bir fonksiyon muamelesi yaparlar. Bu nedenle derleyici strlen fonksiyonunu her defasında çağırmak zorundadır. Döngü içerisinde yapılan çağrıların döngü dışına çıkatılamayabileceğine dikkat ediniz. Yukarıdaki kodu programcı şöyle düzenlemeliydi:

len = strlen(str);

for (i = 0; i < len; ++i) {

...

}

Pekiyi derleyici döngü çağrıları hiç mi döngü dışına çıkartamaz? İşte eğer çağrı yapılan fonksiyon kodun içerisindeyse ve bir yan etkiye yol açmıyorsa derleyiciler onu döngü dışına çıkartabilmektedir.

Bazı C derleyicilerinde (örneğin Microsoft derleyicilerinde ve gcc derleyicilerinde) “intrinsic” fonksiyon kavramı da vardır. Intrinsic fonksiyonların ne yaptıkları derleyici tarafından bilinir. Böylece derleyici o çağrıları optimize edebilmektedir. Gerçekten de Mikrosoft derleyicilerinde ve gcc derleyicilerinde bazı standart C fonksiyonları ”intrinsic” fonksiyonlardır. Bu “intirinsic” fonksiyonlar için prototip bildirimi de gerekmez. Tabii “intrinsic” fonksiyon kavramı C standartlarında yoktur. Bu nedenle taşınabilirlik bakımından bunların “intirinsic” olarak kullanılması taşınabilirliği bozabilir. Intrinsic fonksiyonlar için derleyiciler doğrudan call komutunu kaldırarak sanki onlar inline fonksiyonlarmış gibi kod da yerleştirebilmektedir.

**3) Sabit İfadelerinin Derleme Aşamasında Ele Alınması (Constant Folding):** Bilindiği gibi sabit ifadelerinin sayısal değerleri derleme aşamasında hesaplanabilmektedir. Örneğin:

a = 100 + 2 \* 5 + b;

Böyle bir ifadedeki 100 + 2 \* 5 alt ifadesinin programın çalışma zamanına bırakılmasının bir anlamı yoktur. Bu ifade derleme aşamasında bir kez yapılıp kod ona göre üretilebilir. Örneğin:

a = 110 + b;

Bu kod yukarıdakiyle aynı anlamdadır. Sabit ifadelerinin derleme aşamasında ele alınması çok temel bir optimizasyondur. Derleyicilerin optimizasyon seçenkerli kapalı olsa bile derleyiciler bu optimizasyonu default olarak yapabilmektedir.

**4) Etkisiz Kodların Elimine Edilmesi (Dead Code Elimination):** Derleyiciler kod üzerinde tkisi olmayan kodları optimizasyon sırasında tamamen kodda çıkartabilmektedir. Örneğin:

int main(void)

{

int i;

int total = 0;

int result;

for (i = 1; i <= 100; ++i)

total += i;

printf("Ok\n");

return 0;

}

Burada aslında döngünün de total değişkenin de result değişkenin de koda bir etkisi yoktur. Yani başka bir deyişle program aşağıdaki gibi olsaydı da yazan için değişen bir şey olmazdı:

int main(void)

{

printf("Ok\n");

return 0;

}

Çünkü her ne kadar total değişkeni üzerinde bir toplama yapılmışsa da bu toplam başka bir yerde kullanılmamaıştır. Benzer biçimde result değişkeni de bildirilmiştir fakat kullanılmamıştır. Tabii bir kodun etkisinin olmadığı iyi bir biçimde analiz edilmelidir. Bazı kodlar dolaylı yan etkilere sahip olabilmektedir. Şimdi aklınıza aşağıdaki gibi bir kodun etkisiz olup olmadığı sorusu gelebilir:

for (i = 0; i < 100; ++i)

;

**5) Koşul İçeren Döngülerin Yeniden Düzenlenmesi (Loop Unswitching)**

Bir döngü içerisinde döngüye bağlı olmayan bir koşul altında farkı işlemler yapılmak istenebilir. Bu durumda döngüsünün içerisinde her defasında bu koşula bakmak yerine koşulun içerisinde döng kullanmak daha hızlı çalışmaya yol açabilir. Örneğin flag değişkeni 1 ise döngü içerisinde bir işlem 0 ise başka bir işelm yapılıyor olsun:

for (i = 1; i < 1000000; ++i)

ifade1;

if (flag)

ifade2;

else

ifade3;

}

Döngü şöyle düzenlenirse daha hızlı çalışacak hale getirilebilir:

if (flag)

for (i = 1; i < 1000000; ++i) {

ifade1;

ifade2;

}

else

for (i = 1; i < 1000000; ++i) {

ifade1;

ifade3;

}

Şüphesiz optimzizasyonlar gerçek anlamda değecek bir kazanç sağlanıyorsa uygulanmalıdır. Örneğin burada optimiza edilmiş biçim kodun daha hızla çalışmasını sağlamasını karşın hem kodu daha karmaşık gibi göstermekte hem de kodu büyütmektedir. Buradaki döngü eğer çok az dönüyorsa derleyicinin böyle bir düzenleme yapması toplamda önemli bir hız kazancı sağlamayacaktır.

**6) Döngü Açımı (Loop Unrolling):** Döngü açımı döngü içerisindeki deyimleri çoklayarak döngünün daha az dönmesini sağlayan ve böylece döngü karşılaştırmasını azaltmayı hedefleyen bir tekniktir. Örneğin:

for (i = 0; i < 100; ++i)  
    foo();

Burada 100 kez foo fonksiyonu çağrılmıştır. Döngü aşağıdaki gibi düzenlenirse daha hızlı çalışacak hale getirilebilir:

for (i = 0; i < 100; i += 4) {  
    foo();  
    foo();  
    foo();  
    foo();  
}

Tabi döngü şöyle de açılabilirdi:

foo();        // 1  
foo();        // 2  
...          
foo();        // 98  
foo();        // 99  
foo();        // 100

Ancak bu açık kodu çok büyütürdü. İşte derleyiciler bunun iyi bir noktasını bulmaya çalışmaktadır.

**7) Döngülerin Ters Çevrilmesi (Loop Inversion):** Bu optimizasyon tekniğinde while döngüsü ters yüz edilerek do-while döngüsü haline getirilir. Örneğin:  
  
while (i < 100)  {  
    foo();  
    ++i;  
}  
  
Bu döngünün makina kodlarında iki jump işlemi vardır. Birincisi döngü kontrolündeki karşılaştırma için uygulanan jump ikincisi de döngünün sonuna gelindiğinde yeniden başa geçmek için kullanılan jump. Örneğin Microsoft C(Version 15.00) derleyicilerinde optimizasyon seçenekleri kapatılarak yapılan derleme işlemi sonucunda aşağıdaki gibi bir kod elde edilmiştir:  
  
$LN2@main:  
    cmp    DWORD PTR \_i$[ebp], 100              
    jge    SHORT $LN1@main  
    call    \_foo  
    mov    eax, DWORD PTR \_i$[ebp]  
    add    eax, 1  
    mov    DWORD PTR \_i$[ebp], eax  
    jmp    SHORT $LN2@main  
$LN1@main:  
  
Burada \_i$[ebp] yerel i değişkenini belirtiyor. İşte yukarıda verdiğimiz while döngüsünün aşağıdaki ters yüz edilmiş hali daha hızlıdır:  
  
if (i < 100)   
    do {  
        foo();  
        ++i;  
    } while (i < 100);  
  
Bu biçimde döngü içerisinde tek bir jump işlemi yapılmaktadır. Derleyiciler   optimizasyon seçenekleri açıldığında while döngülerini bu biçime getirerek hız kazancı sağlamaya çalışabilirler. Örneğin yukarıdaki birinci kod Microsoft C (Version 15.00) derleyicilerinde optimizasyon seçenekleri açıldığında aşağıdaki gibi derlenmektedir:  
  
mov    DWORD PTR \_i$[esp+4], eax  
mov    eax, DWORD PTR \_i$[esp+4]  
cmp    eax, 100                  
jge    SHORT $LN7@main  
$LL2@main:  
call    \_foo  
inc    DWORD PTR \_i$[esp+4]  
mov    ecx, DWORD PTR \_i$[esp+4]  
cmp    ecx, 100                  
jl    SHORT $LL2@main  
  
Jump işlemi pek çok işlemci için pipeline mekanizması üzerinde olumsuz etkileri olan bir işlemdir. İşlemci Jump işlemini gördüğünde sonraki komutların çalışıp çalışmayacağını bilmediğinden onlar için hazırlık işlemleri yapmayabilir. Bu konuda modern işlemciler arasında çeşitli farklılıklar bulunduğunu belirtelim.

**8) Döngü Birleştirmesi (Loop Fusion):** Birden fazla peş peşe döngü tek döngü olarak birleştirilebilir. Örneğin:

int a[100], i;  
  
for (i = 0; i < 100; ++i)   
    a[i] = i;  
  
for (i = 0; i < 100; ++i)   
    foo(i);

Aşağıdaki döngü daha hızlı çalışabilir:

int a[100], i;  
  
for (i = 0; i < 100; ++i){  
    a[i] = i;   
    foo(i);  
}

Ancak cache kullanan bazı işlemcilerde döngüleri birleştirmek yerine tam tersine ayrıştırmak (loop fission) hız kazancı artırabilmektedir. Ya da şöyle diyebiliriz: Cache kullanan bazı işlemcilerde döngü birleştirmesi kar değil zarar oluşturabilir. Bunun nedeni işlemcinin bir bölgeye eriştiğinde oradaki belli miktarda byte’ı cache almasıdır. Bu cache’e alıp bırakma cache tazeleme sorununa yol açabilir ve toplam verimi düşürebilir.

**9) Döngü Yer Değiştirmesi (Loop Interchange):** Özellikle çok boyutlu dizilerin indekslenmesinde kullanılan bir oprimizasyon tekniğidir. Bu teknikte bellekteki atlamaların kısaltılarak cache etkinliğinin artırılması hedeflenir. Örneğin:

for (k = 0; k < colSize; ++k)   
    for (i = 0, i < rowSize; ++i)  
        a[i][k] = val;

C/C++ gibi programlama dillerinde iki boyutlu diziler satır dizilerinin art arda getirilmiyle tek boyutlu olarak oluşturulmaktadır. Yani bu dillerde a[i][k] elemanı tek boyutlu dizinin i \* colSize + k indeksinde bulunur. Yani bu organizasyon dikkate alındığında yukarıdaki örnekte iç döngüde uzun atlamaların olduğu söylenebilir. Dikkat ediniz, iç döngünün her yinelenmesinde bellekte bir sütun uzunluğu kadar atlama yapılmıştır. Bu atlama modern işlemcilerde fazlaca “cache miss” oluşmasına yol açabilir. İşte bu optimizasyon tekniğinde derleyici döngüleri ters sırada dizerek bu sorunu gidermeye çalışır:

for (i = 0, i < rowSize; ++i)  
    for (k = 0; k < colSize; ++k)   
        a[i][k] = val;

Şimdi artık içteki döngünün her yinelenmesinde uzun atlamalar oluşmuyor. Dolayısıyla “cache miss” oranının düşürüleceğini söyleyebiliriz. Ayrıca, döngülerin bu biçimde yer değiştirilmesi vektörel işlem yapan işlemcilerde hız kazancı sağlayabilecek potansiyel bir durumu da oluşturduğunu belirtelim.

**10) Göstericilere İlişkin Optimizasyonlar (Pointer Aliasing):** Bir göstericinin gösterdiği yerdeki bilgi geçici bir değişkende ya da yazmaçta tutulursa tekrar tekrar o göstericinin gösterdiği yere erişme işlemi elimine edilebilir. Çünkü örneğin C’de \*p gibi p[i] gibi işlemler aslında birden fazla makine komutuyla yapılmaktadır. Örneğin:

a = \*p + 1;

b = \*p + 2;

c = \*p + 3

Burada her defasında \*p işlemi yapmak yerine kod aşağıdaki gibi düzenlenirse hız kazancı sağlanabilir:

temp = \*p; /\* temp bir yazmaç da olabilir \*/

a = temp + 1;

b = temp + 2;

c = temp + 3

Tabii derleyicinin bu optimizasyonu yapabilmesi için kesinlikle o sırada bu göstericinin gösterdiği yerdeki değerin değişmmeyeceğini garanti altına alması gerekir. Örneğin:

a = \*p + 1;

foo();

b = \*p + 2;

c = \*p + 3

Burada foo p’nin gösterdiği yerde değişiklik yapıyor olabilir. Bu durumda derleyici foo’nun içini de incelemeden böyle bir kararı veremez.

Eğer bir göstericinin gösterdiği yere yalnızca o gösterici tarafından erişiliyorsa yani o göstericinin dışında oraya erişen başka bir kod yoksa bu yukarıda belirtilen tarzda optimizasyonlara yol açabilir. Örneğin bir sistemde belleğin bir bölgesinden n byte’ı tek hamlede çekerek diğerine tek hamlede aktaran makine komutunun var olduğunu düşünelim. Bu durumda derleyici aşağıdaki gibi bir memcpy fonksiyonunu bu komutu kullanarak optimize edebilir mi?

void \*mymemcpy(void \*dest, void \*source, size\_t size)

{

char \*cdest = (char \*) dest;

char \*csource = (char \*) source;

while (size-- > 0)

\*cdest++ = \*csource++;

return dest;

}

Eğer bloklar çakışıksa derleyici bu makine komutundan faydalanamaz. İşte mymemcpy’yi çağıran kişi böyle bir duruma yol açabileceği için derleyici bu olasılığı göz ardı edemez.

C99 ile birlikte derleyicilerin gösterici optimizasyonlarını iyileştirmek için restrict gösterici kavramı dile sokulmuştur. restrict anahtar sözcüğü göstericinin kendisi kullanılabilir, gösterdiği yer için kullanılamaz. Bir gösterici restrict anahtar sözcüğü ile bildirilmişse artık derleyici o göstericinin yaşamı boyunca o göstericinin gösterdiği yere yalnızca bu gösteri yoluyla erişildiği garantisini alır. Şimdi yukarıdaki memcpy fonksiyonundaki göstericileri restrict yapalım:

void \*mymemcpy(void \* restrict dest, void \* restrict source, size\_t size)

{

char \* restrict cdest = (char \*) dest;

char \* restrict csource = (char \*) source;

while (size-- > 0)

\*cdest++ = \*csource++;

return dest;

}

Artık biz burada derleyiciye şunun garantisini veriyoruz: “Derleyici ben sana iki adres vereceğim. Bu göstericilerin faaliyet alanı bitene kadar bu adreslerdeki bilgilere ben başka yolla erişmeyeceğim. Dolayısıyla source ve dest de çakışık blokları göstermeyecek. Sen bu bilgilerden hareketle gösterici optimizasyonları yapabilirsin. Ben bu kurala uymamışsam cezamı ben çekerim.”

Şimdi önceki örnektektei optimzasyona geri dönelim:

a = \*p + 1;

foo();

b = \*p + 2;

c = \*p + 3

Burada eğer p restrict bir göstericiyse derleyici artık aşağıdaki gibi bir optimizasyonu yapabilir:

temp = \*p; /\* temp bir yazmaç da olabilir \*/

a = temp + 1;

foo();

b = temp + 2;

c = temp + 3

Çünkü artık derleyici p’nin gösterdiği yere foo tarafından da erişilmeyeceğinin sözünü almıştır.

**11) Yazmaç Tahsisatları (Register Allocation):** İşlemcilerde belli sayıda yazmaç vardır. Değişkenler yazmaçlara çekilerek işleme sokulurlar. Bu durumda derleyiciler hangi değişkenlerin hangi yazmaçlara çekileceği kararını vermek zorundadır. Bu karar verilirken çok kullanılan değişkenlerin mümkün olduğu kadar fazla süre yazmaçta kalamasına çalışılır. Böylece derleyici aynı değişkeni tekrar bellekten yazmaca çekmek zorunda kalmaz. İşte hangi değişkenlerin hangi yazmaçlara çekileceği önemli bir optimizasyon konusudur.

**12) Makine Komutlarının Düzenlenmesi ve Yer Değiştirilmesi (Instruction Scheduling and Reordering):**  Bazı makine komutlarının sırasının değiştirilmesi hız kazancı artırabilmektedir. İşte eğer makine komutları arasında girdi çıktı ilişkisi yoksa derleyici bunları yeniden düzenleyerek kodun daha hızlı çalışmasını sağlayabilmektedir. Bu optimizasyon sanıldığı kadar basit değildir ve NP tarzda karmaşıklıklara sahiptir. Dolayısıyla derleme zamanını bu tür optimizasyonlar uzatabilmektedir.

Konuyu kapamadan önce şu uyarıyı tekrarlamakta fayda görüyoruz: Derleyicilerin optimizasyonları tek thread’li akışa göre yapılmaktadırç. Yani eğer çok thread’li uygulama yapılıyorsa derleyicinin bu optimizasyonları bizim kodumuzun anlamını değiştirebilir. Çok thread’li uygulamalarda bu duruma programcının kendisinin dikkat etmesi gerekmektedir. Örneğin:

a = x + y + 10;

----> Bu noktada x, y değişirse ne olur?

b = x + y + 20;

c = x + y + 30;

Burada x ve y’nin global değişkenler olduğunu düşünelim ve ok ile belirtilen noktada başka bir thread’in o kodu x ve y’yi değiştirdiğini varsayalım. Derleyici burada başka bir thread’in x ve y’yi değiştirebileceğini dikkate almaz. Ve optimizasyonunu aşağıdaki gibi yaapabilir:

temp = x + y;

----> Bu noktada x, y değişirse ne olur?

b = temp + 20;

c = temp + 30;

Görüldüğü gibi burada artık x ve y’nin thread tarafından değiştirilen yeni değeri değil eski değeri sonraki işlemlere sokulmuştur. Biz bu tür optimizasyonlara dikkat etmeliyiz. Burada volatile anahtar sözcüğü hayat kurtarıcı olabilir:

volatile x, y;

...

Artık volatile bir nesneye derleyici erişirken her zaman onun o anki değerini bellekten alarak kullanır. Aşağıdaki örneğe dikkat ediniz:

while (g\_flag) {

...

}

Burada programcı g\_flag’in başka bir thread tarafından 0’a çekildiğinde döngüden çıkılmasını istemiş olabilir. Fakat derleyici g\_flag değerini yazmaçta tutarak bir optimizasyon yapabilir. Yani derleyici sürekli olarak g\_flag’in değerini yeniden bellekten almak zorunda değildir. Burada da g\_flag volatile yapılmalıdır.

Derleyiciler eğer multi thread bir ortamı dikkate alarak optimizasyon yapsaydı yukarıda gördüğümüz optimizasyon temalarının çoğu devre dışı kalırdı.

**AŞAĞI SEVİYELİ DİSK İŞLEMLERİ**

Kursumuzda disk sözcüğü yalnızca manyetik temelli ikincil bellekler için değil tüm ikincil bellekler için kullanılan genel bir terim olarak kullanılmaktadır. Bu bağlamda örneğin SSD’ler ya da flash EPROM’lar yarı iletkenlerle imal edildikleri halde onlara da burada disk denilmektedir.

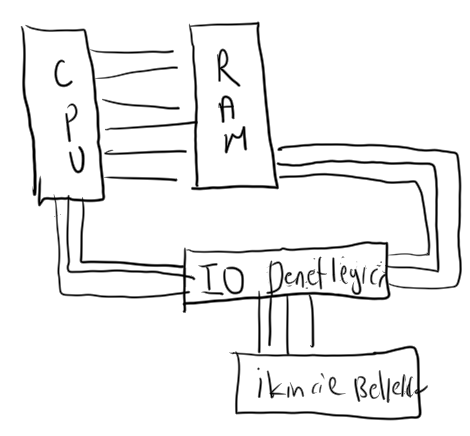
Bugün kullandığımız ikincil bellekler tipik olarak üç gruba ayrılmaktadır:

1) Manyetik tabanlı elektomekanik hard diskler

2) Yarı iletkenlerle imal edilmiş SSD’ler, flash EPROM ve EEPROM bellekler

3) CDROM, DVD ROM gibi ışınsal mekanizmayla çalışan elektormekanik ikincil bellekler

Tipik olarak bir bilgisayar devresinde CPU ile ikincil bellekler elektriksel doğrudan bağlantılı değildir. Bağlantı bir denetleyici yoluyla gerçekleştirilmektedir. CPU’nun en az bir IO denetleyici birimle bağlantısı vardır. CPU bu birimlere elektriksel olarak komutlar gönderir. IO denetleyicileri de bu komutları ikincil bellek birimlerine (yani disklere) yollar. İkincil bellek birimleri de kendi devrelerini harekete geçirerek istenilen işlemi yapar. Böylece örneğin bir hard diskin içerisindeki devreler ve mekanik bileşenler standart değildir. Ancak CPU’nun komut gönderdiği IO denetleyicisi standart bir arayüze sahiptir.



İkincil bellekleri kontrol etmek için IDE, ATA, SATA, SCSI gibi çeşitli denetleyici birimler ve bus arayüzleri bulunmaktadır. CPU bu denetleyicilere komutlar göndererek isteğini bildirir. Bu denetleyiciler elektriksel yollarla asıl ikincil belleğe komutu iletirler. İkincil bellekler de istenilen işlemi yaparlar. Bir okuma işlemi söz konusuysa okunan bilgiler doğrudan RAM’e aktarılır. Bir yazma işlem söz konusuysa RAM’de oluşturulmuş bilgiler ikincil belleğe doğrudan aktarılır. Bu aktarım işlemi DMA (Direct Memory Access) denilen denetleyiciler tarafından yapılmaktadır.

Bu durumda ikincil bellekten bir okuma ya da yazma isteği iki önemli bilgiyi içermek zorundadır:

1) Bu istek ikincil belleğin neresine ilişkindir ve hangi büyüklüktedir?

2) Okunacak ya da yazılacak bilgiler için RAM aktarım adresi neresidir?

Modern çok prosesli işletim sistemleri ikincil belleklerden bir talepte bulunduğunda onun sonucunu beklemezler. Proseslerarası (ya da thread’lerarası) geçiş yaparak istekte bulunan proses ya da thread’i blokede bekletirler. IO denetleyicileri işlem bittiğinde bunu CPU’ya bildirmektedir. işletim sstemi de işleme yol açan proses ya da thread’i çizelgeye yeniden sokarak onun kaldığı yerden çalışmaya devam etmesini sağlar.

**İkincil Belleklerdeki Organizasyon**

İkincil belleklerden bilgiler byte byte değil blok blok transfer edilmektedir. Genel olarak ikincil belleklerden transfer edilecek en küçük birime sektör denir. Bir sektör 512 byte uzunluktadır. (Bazı durumlarda bu büyüklük değişebilir. Ancak 512 standart bir değer olarak kabul görmüştür). Sektör klasik elektomekanik disk sistemleri zamanında uydurulmuş bir terimdir. Günümüzde SSD gibi Flash EPROM ve EEPROM temelli belleklerdeki transfer yöntemleri disklere göre değişiklik gösterse de yine sektörel bir transfer uygulanmaktadır.

**Hard Disklerin Yapısı**

Elektomekanik hard disklerde bilgiler manyetik olarak bir yüzeye kaydedilir ve oradan okunur. Bir hard diskte tipik olarak platter denilen yüzeyle bir çubuğa geçirilmiş duurmdadır. Bu yüzeyler belli bir hızda döndürülür. Okuma yazma amacıyla her yüz için bir kafa (head) bulundurulmuştur. Kafa diske çok yaklaşır fakat onunla temas etmez. Bilgi manyetik olarak kodlanır ve geri alınır.

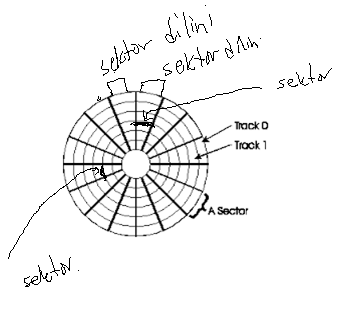


Yukarıdaki resimde üç platter vardır. Disk kafaları aynı eksene monte edilmiştir. Her yüzey için bir kafa bulunmaktadır. Yani yukarıdaki resimde görülen hard diskte toplam 6 kafa vardır. Dodlama ve geri alma disk dönerken kafalar tarafından yapılır. Hard diskler kendi içerisinde küçük bir önbelleğe de sahip olabilirler.

Sektör bir hard diskten okunabilecek ya da hard diske yazılabilecek en küçük bir birimdir. Yani örneğin biz bir diske 10 byte yazamayız. En az bir sektör yazabiliriz. Örneğin biz bir diskteki bir byte’ı değiştirmek isteyelim. Bunun tek yolu şudur: Önce o byte’ın bulunduğu sektör okunur ve RAM’e aktarılır. Değişilik RAM’de yapılır. Sonra o sektör diske geri yazılır.

Hard diskte bilgilerin yazıldığı yollara track denilmektedir. Track’ler en dıştan başlayarak sıfırdan itibaren numaralandırılmıştır. Kafalar track hizasına getirilir, disk dönerken okuma yazma işlemi gerçekleşir.

Eskiden diskler sektör dilimi denilen pasta dilimlerine ayrılıyordu. Track’lerin sektör dilimleri içerisinde kalan parçaları sektörleri oluşturuyordu:



Şekilden de görüldüğü gibi dış track’ler daha uzundur. Fakat eski sistemde treack’lerin sektör dilimleri içerisinde kalan parçalarına hep aynı miktar bilgi (512 byte) depolanıyordu. Ancak dah sonraları (90’lı yıllarda) artık üretici firmalar dış track’lere daha fazla bilgi depolamaya başlamıştır. Dolayısıyla sektör dilimi kavramının pratik bir geçerliliği kalmamıştır. Bugünkü hard disklerde yine sektör temelinde aktarım yapılır. Ancak dış track’lerdeki sektör sayıları iç track’lerden daha fazladır.

Bir hard diskin hızı saniyede okunan ya da yazılan byte sayısıyla (ya da sektör sayısıyla) ölçülmektedir. Bir sektörün okunma ya da yazılma hızı üç bileşenin toplamından oluşur:

1) Disk kafasının Uygun Track’e Konumlanma Zamanı (Seek Time): Disk kafasının uygun track hizasına çekilmesi en önemli zaman kaybını oluşturmaktadır. Dolayısıyla sistem programcısı için en önemli bileşen disk kafa hareketinin optimize edilmesidir.

2) Diskin Dönerek kafa Hizasından Geçene Kadarki Zaman (Rotational Delay): Diskin hızını etkileyen ikinci bileşen disk dönüş hızıdır. Çünkü sektör kafa hizasından geçerken okuma yazma yapılır. İlgili sektörün kafa hizasından geçme zamanı diskin dönüş hızıyla ilgilidir. Bugünki diskler tipik olarak 4200, 5400, 7200, 10000, 15000 hızlardadır. Bizim notebook’larımızda tipik olarak 5400 dönüş hızları kullanımaktadır.

3) Aktarım Zamanı (Transfer Time): Bu zaman en önemsiz zamanı oluşturur. Disk dönerken kafanın bilgiyi okuyup IO denetleyicisine aktardığı zamandır. Bu zaman IO denetleyicisinin tasarımına da, hard diskin teknolojisine de bağlıdır. Örneğin SCSI denetleyiciler daha hızlı bir aktarım sunabilmektedir.

İşletim sistemi tasarımcıları disk kafa hareketini minimize etmeye odaklanırlar. Çünkü diğer bileşenler sistem programcısının kontrolü dışındadır. Eğer işletim sistemi dosyanın parçalarını birbirine sektörlerde tutarsa o dosyaya erişimdeki kafa hareketi minimize edilir. Örneğin “defrag” isimli disk utility programları dosyaların parçalarını yer değiştirerek birbirlerine yaklaştırmaktadır.

**Sektörler İçin Koordinat Sistemi ve Sektör Numaralandırması**

İkincil belleğin türü ne olursa olsun bugün ikincil belleklerle ana bellek arasındaki veri aktarımı sektör düzeyinde yapılmaktadır. Sektörlerin de birer numaraları vardır. Eskiden sektör dilimlerinin anlamlı zamanlarda bir sektörün koordinato yüzey numarası (kafa numarası da denilmektedir), track numarası ve sektör dilimi numarasıyla belirleniyordu. Yani koordinat h:t:s bileşenlerinden oluşmaktaydı. Bugün hala BIOS geçmişe doğru uyumu korumak için bu koordinat sistemini de kullanmaktadır. Ancak daha sonra sektör dilimlerinin fiziksel bir anlamı kalmamaya başlayınca daha sade bir koordinat sistemine geçilmiştir. Bu koordinat sisteminde sektörler tek bir sayıyla belirtilmektedir. Bu sistemde ilk sektörün numarası sıfır olmak üzere her sektöre artan sırada bir numara karşılık düşürülmüştür. Buna Mantıksal Blok Adreslemesi (Logical Block Addressing ya da kısaca LBA) denilmektedir. Numaralandırma en dış yüzeyin en dış track’inden başlayarak (yani sıfıncı track) önce yüzler sonra trackler dolanılarak yapılamkatdır. Yani aşağıdaki sahte koddaki gibi bir numaralandırma söz konusudur:

logical\_address = 0;

for (track = 0; track < MAX\_TRACK; ++track)

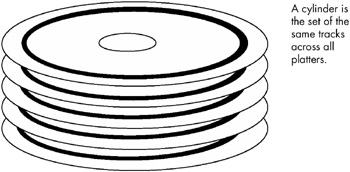
for (head = 0; head < MAX\_HEAD; ++head) {

<logical\_address numarasını oluştur>

}

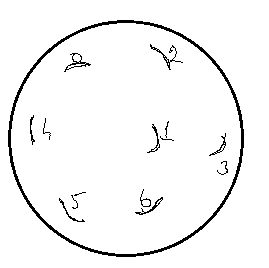
Görüldüğü gibi mantıksal numaralandırma kafa hareketini minimize edecek biçimde yapılmıştır. Numaralandırmada önce en dış yüzeyin ilk track’i dolanılır, sonra diğer yüze geçilip onun en dış track’i dolanalır. Böyle böyle tüm yüzlerin en dış track’leri dolanıldıktan sonra bir sonraki track’e geçilir. Yani mantıksal mantıksal sektör numaraları birbirine yakınsa bunlar daha küçük kafa hareketiyle erişilecek sektörlerdir. Artık IO denetleyicileri de sistem programcısından sektörleri mantıksal sektör numarasıyla istemektedir. Sonuç olarak diskteki her sektörün bir numarası vardır. Ardışıl sektör numaraları aynı kafa hizasına karşılık gelir.

Sektör numaralandırmasında bir de silindir (cylinder) terimi ile karşılaşılmaktadır. Tüm yüzeylerin n numaralı track’lerinin kümesine ‘n’ numaralı silindir denilmektedir. Örneğin 6 yüzeyli (3 platter’lı) bir hard diskte 10 numaralı silindir demek tüm yüzeylerin 10 numaralı track’lerinin kümesidir.



**Dosya Kavramı İle Sektörler Arasındaki İlişki**

Aslında dosya (file) kavramı uydurma bir kavramdır. İkincil belleklerdeki tek gerçek, sektör gerçeğidir. İşletim sistemi birtakım bilgileri diskin sektörlerinde saklar buna da bir isim karşılık düşürür. Kullanıcılara bunu dosya kavramı biçiminde gösterir. Aslında dosyalar ardışıl byte’lardan oluşmamaktadır. Diskteki sektörlerde bulunan bilgilerden oluşmaktadır. İşletim sistemi bize o sektörlerdeki bilgiyi sanki ardışıl byte topluluğuymuş gibi gösterir. Örneğin 7 sektörden oluşan bir dosya aşağıdaki gibi disk yüzeyine (hatta yüzeylerine) dağılmış olabilir.



İşletim sistemi dosyaların parçalarını sektörlere neden ardışıl yerleştirmemektedir? İşte daha önceden de gördüğümüz gibi ardışıl yerleştirme zorunluluğu bölünme (fragmentation) denilen olguya yol açar. Bölünme tahsis etme ve silme işlemleri sonucunda bellekte ardışıl olmayan çok sayıda küçük blokların oluşması durumudur. Bu diskin kullanım verimini çok düşürür. İşte sistemleri dosyaların parçalarını disk kafa hareketlerini azaltmak için mümkün olduğu kadar ardışıl yerleştirmeye çalışmaktadır. Ancak ardışıl yerleştirme bir zorunluluk biçiminde değildir.

Mademki işletim sistemi dosyaların parçalarını ardışıl olmayan sektörlerde tutabilmektedir, o halde dosyarın hangi parçalarının diskin neresinde olduğu bir biçimde işletim sistemi tarafından tutulmalıdır.

**Cluster ve Block Kavramları**

Aslında sektörler dosya parçalarını saklamak için küçük birimlerdir (bir sektörün 512 byte olduğunu anımsayınız). İşte işletim sistemleri dosya parçalarını sektörler yerine “cluster” ya da “block” denilen ardışıl sektörlerde saklarlar. Cluster ile block kavramları bu bağlamda eş anlamlıdır. Microsoft “cluster” terimini tercih ederken UNIX/Linux sistemlerinde terimi tercih edilmektedir. Bir cluster ya da blok ardışıl n tane sektörden oluşturulmaktadır (burada n sayısı genellikle 2’nin bir kuvvetidir). İşletim sistemleri diski cluster ya da bloklara ayırır ve her cluster ya da bloğa bir numara verir. Tabii cluset ve block kavramları aslında yapauy kavramlardır. Diske sektör numaralatıylşa erişilir.

İşletim sistemleri dosyaların parçalarını cluster ya da block’larda saklar. Bir cluster ya da block bir dosyanın parçası olabilecek en küçük birimdir. Örneğin dosya 1 byte bile uzunlukta olsa o diskte bir cluster ya da block yer kaplar. Dosyanın parçalarının sektörlerde değilde ardışıl sektör bloklarında tutulmasının iki nedeni vardır:

1) Bu sayede dosyaların hangi parçalarının nerede tutulduğuna ilişkin meta-data alanı küçültülmüş olur (çünkü dosya daha az parçadan oluşacaktır)

2) Bu sayede dosya diskte daha az yayılmış bir biçimde tutulur. Bu da dosyay erişmek için gerekn disk kafa hareketlerinin azalmasına yol açar.

Pekiyi bir cluster ya da block kaç sektörden oluşmalıdır? İşte eğer bir cluster ya da block az sayıda sektörden oluşursa diskteki meta-data alanları büyür, dosya erişmek için gereken kafa hareketleri artar. Fakat dosyaların son cluster ya da bloklarında kullanılmayan alanlar (yani içsel bölünme) azaltılır. Eğer bir cluster ya da block çok sektörden oluşturulursa bu durumda disk meta-data alanları küçülür, kafa hareketleri azalır fakat içsel bölünme miktarı artar. İşte işletim sistemleri disk büyüdükçe cluster ya da bloğun daha fazla sektörden oluşturmayı, disk küçüldükçe daha az sektörden oluşturmayı tercih etmektedir. Bazı işletim sistemlerinde disk formatlanırken bunu sistem yöneticisi de belirleyebilmektedir.

**Disk Editörleri**

Nasıl text editörler varsa, audio ve video editörleri varsa diski analiz etmek ve görüntülemekiçin de disk editörleri vardır. Bazı disk editörleri paralıdır. Bazıları açık kaynak kodlu ve/veya bedavadır. Bedava iki önemli disk editörü “Active Disk Editor” ve “HxD” editörüdür. Bu editörler sayesinde istediğimiz disk sektörlerini inceleyip onlar üzerinde değişiklikler yapabiliriz. Dosya sistemi üzerinde çeşitli incelemeler yapabiliriz.

**Dosya Sistemleri**

Bir işletim sisteminin dosyalarla ilgili işlemler yapan alt sistemine dosya sistemi (file system) denilmektedir. Dosya sisteminin gerçekleştime bakımından iki tarafı vardır:

1) Ana Bellek Tarafı: İşletim sistemi açılan her dosya için kernel alanı içerisinde o dosyaya yönelik kayıtlar tutar. Sonra bu kayıtlardan faydalanarak dosya işlemini gerçekleştirir. Bu kayıtlar dosya sisteminin ana bellek atafını oluşturmaktadır.

2) Disk Tarafı: İşletim sistemleri diskin sektörleri organize ederek dosyaları diskte düzenli bir biçimde saklarlar. İşte işletim sistemlerinin disk üzerindeki organizasyonuna o dosya sisteminin disk tarafı denilmektedir. Biz kursumuzda bir dosya sisteminin disk organizasyonu dediğimizde o dosya sisteminin hangi disk sektörlerini hangi amaçla kullandığını anlamalıyız.

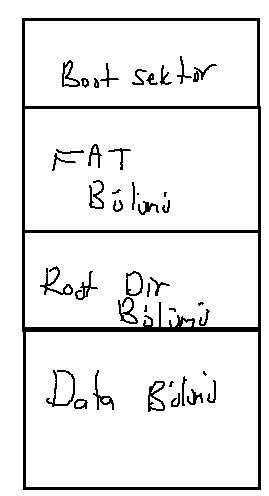
Bazı işletim sistemlerinin dosya sistemlerinin ana bellek tarafı birden fazla disk organizasyonunu destekleyebilmektedir. Örneğin Linux sistemleri Microsoft’un FAT ve NTFS olarak düzenlenmiş disk organizasyonunu tanıyıp o diskler üzerinde dosya işlemleri yapabilmektedir. Ancak Microsoft, Linux sistemlerinde kullanılan EXT-2 EXT-3 gibi disk organizasyonlarını desteklememktedir.

**FAT Dosya Dosya Sistemlerinin Disk Organizasyonu**

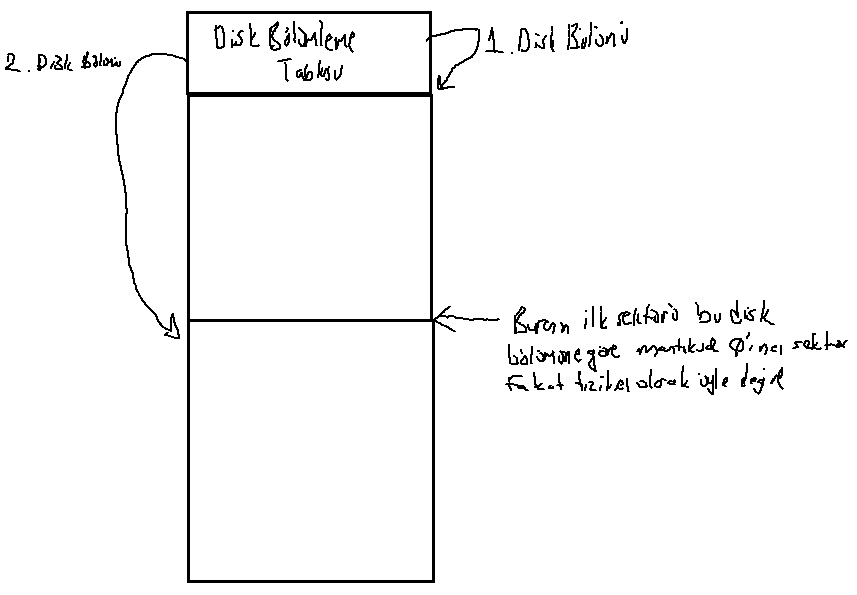
FAT dosya sistemleri Microsoft’un ilk DOS işletim sisteminden beri kullandığı temel dosya sistemidir. FAT dosya sistemi ilk yıllarda önemli bir yenilik olarak değerlendirilmiştir. Bu dosya sisteminde ileride ele alınacağı gibi ismine FAT (File Allocation Table) denilen önemli bir meta-data bölümü vardır. Bu bölüm dosya sistemine ismini vermiştir.

FAT dosya sisteminin FAT12, FAT16 ve FAT32 biçiminde üç değişik biçimi vardır. FAT12’de FAT elemanları 12 bit, FAT16’da 16 bit ve FAT32’de 32 bittir. Ayrıca FAT32’deki meta-data organizasyonunda bazı önemli değişiklikler de yapılmıştır. FAT32 daha büyük disk bölümlerini kullanabilmektedir. Genel kapasite olarak FAT12 ve FAT16 sistemlerinin geliştirilmiş bir biçimidir.

FAT12 ve FAT16 dosya sistemiyle formatlanmış bir disk bölümü aşağıdaki organizasyonel bölümlerden oluşur:



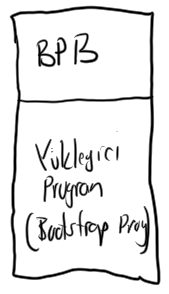
Hard disk’ler, SSD’ler ve Flash EPROM bellekler birden fazla disk bölümüne sahip olabilirler ve bu disk bölümlerine farklı dosya sistemleri yüklenebilir. Bu medyalarda hangi disk bölümlerinin hangi sektörden başlayıp ne kadar uzunlukta olduğu medyanın içerisinde saklanmaktadır. Disk bölümlerinin nereden başladığı ve ne uzunlukta olduğunu tutan meta-tablosuna genel olarak Disk Bölümleme Tablosu (Disk Partition Table) denilmektedir. Disk bölümlerine ayrılmış bir diskteki sektör numaralandırmaları iki biçimde yapılabilmektedir: Fiziksel numaralandırma ve mantıksal numaralandırma. Fiziksel numaralandırma kontrol kartının uyguladığı gerçek numaralandırmadır. Burada medyanın tamamı tek bir medya olarak değerlendirilir ve ilk sektör 0’dan başlatılarak her sektöre artan bir numara karşılık düşürülmüştür. Mantıksal numaralandırmada ilgili disk bölümünün baını orijin noktası olarak kabul eder. Yani her disk bölümü adeta ayrı bir diskmiş gibi 0’dan başlayarak numaralandırılmaktadır. Örneğin:



Bir diskteki kullanıma hazırlanmış disk bölümlerine (partitions) “volume” de denilmektedir. Microsoft “volume” sözcüğünü bu bağlamda çok kullanmaktadır.

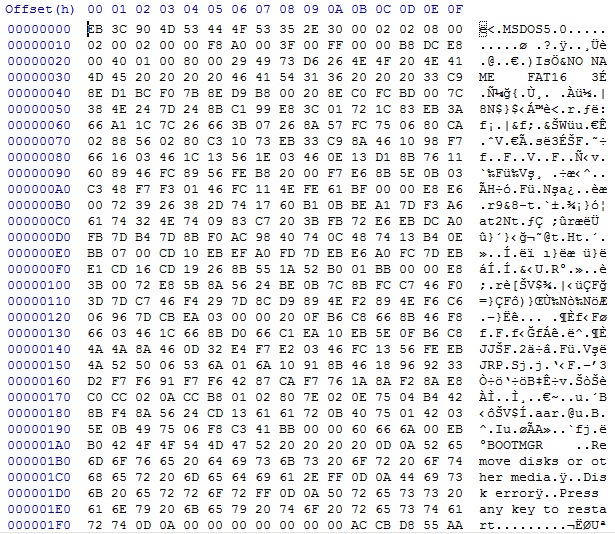
**Boot Sektör**

FAT dosya sistemlerinde volüm’ün ilk sektörüne (yani mantıksal 0’ıncı sektörüne) boot sektör sektör denilmektedir. Boot sektör kendi içerisinde iki bölümden oluşmaktadır: “BPB (Bios Paramter Block)” ve “Yükleyici Program (Bootstrap Program)”



BPB FAT dosya sisteminde volümün parametrik bilgilerinin tutulduğu bölümdür. Bott sektörün başından başlar. Yaklaşık 30-40-50 byte uzuluğundadır (uzunluğu FAT türüne göre ve versiyona göre değişebilmektedir.)

BPB bölümü FAT dosya sisteminin kalbidir. Volümün tüm parametrik bilgileeri, meta-data alanlarının yerleri vs. bu bölümde tutulmaktadır. Boot sektörün bölümleri iyi anlaşılsın diye FAT16 bir volümün boot sektörü aşağıda verilmiştir:



Şimdi BPB bölümünü byte byte inceleyelim:

|  |  |  |
| --- | --- | --- |
| **Offset**  **(Hex)** | **Uzunluk** | **İçerik** |
| 00 | 3 byte | Jmp Code |
| 03 | 8 byte | OEM Yorum |
| 0B | WORD | Sektördeki Byte Sayısı |
| 0D | BYTE | Cluster’ın Sektör Uzunluğu |
| 0E | WORD | Ayrılmış Sektörlerin Sayısı |
| 10 | BYTE | FAT kopyalarının Sayısı |
| 11 | WORD | Root Girişlerinin Sayısı |
| 13 | WORD | Volümdeki Toplam Sektör Sayısı |
| 15 | BYTE | Ortam Belirleyicisi (Media Descriptor) |
| 16 | WORD | Fat’in Bir Kopyasındaki Sektör Sayısı |
| 18 | WORD | Track Başına Düşen Sektör Sayısı |
| 1A | WORD | Diskteki Kafa (yüzey) Sayısı |
| 1C | DWORD | Saklı Sektörlerin Sayısı |
| 20 | DWORD | Yeni Toplam Sektör Uzunluğu |
| 24 | BYTE | Fiziksel Sürücü Numarası |
| 25 | BYTE | Kullanılmıyor |
| 26 | BYTE | Genişletilmiş Boot Id’si (Extended Boot Signature) |
| 27 | DWORD | Volümün Seri Numarası (Volume Serial Number) |
| 2B | 11 Byte | Volüm Etiketi (Volume Label) |
| 36 | 8 byte | Dosya Sisteminin Türü |

**Jmp Code:** Boot sektörün hemen başında BPB bloğunu atlamak için kullanılan jmp makine komutu vardır.

**OEM Yorum:** Burada format programlarının yazdıkları 8 byte’lık bir yazı bulunur. Bu yazının ne olduğunun bir önemi yoktur. Silinse de sorun oluşmaz. Genellikle bu alana Microsoft işletim sisteminin versiyon numarasını yazmaktadır. (Burada MSDOS 5.0 görürseniz şaşırmayınız. Çünkü FAT dosya sistemlerinde BPB’nin son gelişmiş hali MSDOS 5.0 zamanında yapılmıştır)

**Sektördeki Byte Sayısı:** Bu alanda volümün bir sektöründe kaç byte olduğu bilgisi vardır. Burada her zaman 512 değerini görürüz. Gerçi formatla parametreleriyle eskiden flopy’ler sektördeki byte sayısı 512’den farklı olacak biçimde formatlanabiliyordu. Fakat 512 artık standart bir değerdir.

**Cluster’ın Sektör Uzunluğu:** Burada bir cluster’ın kaç sektörden oluştuğu bilgisi vardır. Küçük volümlerde bir cluster az sektörden büyük volümlerde çok sektörden oluşur.

**Ayrılmış Sektörlerin Sayısı:** Bazen boot sektörden hemen sonra FAT bölümü gelmeyebilir. Örneğin hard disklerde FAT’in bir sonraki track başına hizalanması işleyişi hızlandırabilmektedir. Bazen de boot sektör yükleyici programı daha büyük olabilir. Bu durumda da FAT’in yeri ötelenebilir. İşte bu alanda FAT’in hangi mantıksal sektörden başladığı bilgisi vardır. Bu alandaki sayı 1 ise FAT hemen boot sektörden sonra başlar. Örneğin bu alandaki sayı 100 ise boot sektörden sonra 99 sektör başka bir amaçla kullanılmıştır ve FAT 100’üncü mantıksal sektörden başlamaktadır.

**FAT Kopyalarının Sayısı:** Microsoft FAT’in iki kopyasını bulundurmaktadır. Aslında bu iki kopyanın hata düzeltme bakımından bir faydasının dokunduğunu söylemek de çok zordur. Ancak Microsoft geleneksel olarak hala FAT için iki kopya kullanmaktadır. Hatta Microsoft bu iki sayısını o kadar içselleştirmiştir ki boot sektör BPB bloğundaki değeri artık dikkate bile almamaktadır. O halde bu alanda biz hemen her zaman 2 değerini görürüz. FAT kopytalarının arasında hiç sektörel boşluk yoktur. Birinci kopyayı hemen ikinci kopya izler. Normal olarak işletim sistemi FAT’in bir kopyası üzerinde güncelleme yapar yapmaz ikinci kopyası üzerinde de güncelleme yapar. Yani bu kopya normal durumda aynı içeriğe sahiptir. Bizim de iki kopyaya birden bakmak gibi bir gereksnimimiz olmaz. Burada yeniden vurgulamak gerekirse: Microsoft FAT’in iki kopyasını biri bozulursa diğerinden faydalanılsın diye oluşturmuştur. Ancak pratikte bu durumun faydası olduğu söylenemez.

**Root Girişlerinin Sayısı:** Yukarıdaki şekilden de görüldüğü gibi FAT bölümünü “Root Dir” bölümü izlemektedir. “ROOT Dir” bölümü kök dizindeki “dizin girişlerinden oluşur”. Bir dizin girişi FAT12 ve FAT16’da 32 byte uzunluğundadır. Bu alanda kaç tane dizin girişi olduğu bilgisi vardır. Buradan Root Dir bölümünün sektör sayısı dolaylı olarak hesaplanabilmektedir. Yani Root Dir bölümünün sektör uzunluğu şöyle hesaplanır:

Root Dir SektörUzunluğu = Root girişlerinin sayısı \* 32 / 512

FAT12 ve FAT16 sistemlerinde kök dizindeki dosya sayısının en fazla bu alanda belirtilen miktarda olabileceğine dikkat ediniz. Ancak alt dizinlerdeki dosyalar için bir kısıtlama yoktur.

**Volümdeki Toplam Sektör Sayısı:** Bu alanda ilgili disk bölümündeki toplam sektör sayısı bulunur. Maalesef Microsoft ilk dönemlerde bu alanı WORD olarak ayırmıştır. Böylece volümdeki toplam sektör sayısı en fazla 216 tane olabilmektedir. Volümün kapasitesi de 29 \* 216 = 225 = 32MB ile sınırlıdır. Bu durum DOS 3.30’a kadar böyle devam etmiştir. DOS 3.30’da Microsoft BPB’nin sonuna (0x20 offsetine) DWORD olarak yeni bir toplam sektör sayısı alanı yerleştirmiştir. Böylece yeni DOS versiyonları (ve tabii ki günümüz Windows’larına kadarki tüm Windows versiyonları) önce 0x13 offset’inden WORD çekerler. Eğer orada 0 görürlerse bu kez 0x20 offset’inden DWORD çekerek toplam sektör sayısını elde ederler.

**Ortam Belirleyicisi (Media Descriptor):** Bu alanda medyanın nasıl bir medya olduğu bilgisi yer almaktadır. Bu alan artık kullanılan bir alan değildir. Eskiden bu alanın bir işlevi vardı. Fakat artık ciddi bir işlevi olduğu söylenemez. Bu alanda eğer F8 varsa ortam hard disk’tir. F0 varsa ortam floppy’dir. Biz burada artık hep F8 görürüz.

**Fat’in Bir Kopyasındaki Sektör Sayısı:** Bu alanda FAT’in bir kopyasındaki sektör sayısı bulunmaktadır. Böylelikle FAT’in toplam sektör uzunluğu hesaplanabilir. FAT’in iki kopyasının bulunduğunu anımsayınız.

**Track Başına Düşen Sektör Sayısı:** Bu alanın şimdilerde bir önemi kalmamıştır. Ancak fiziksel koordinat sisteminin kullanıldığı (INT 13h) DOS sistemlerinde bu alanda bir track’in kaç sektör diliminden oluştuğu bilgisi bulunuyordu.

**Diskteki Kafa (yüzey) Sayısı:** Bu alanın şimdilerde bir önemi kalmamıştır. Ancak fiziksel koordinat sisteminin kullanıldığı (INT 13h) DOS sistemlerinde bu alanda diskteki toplam kafa sayısı bulunmaktadır.

**Saklı Sektörlerin (Hidden Sectors) Sayısı:** Burada ilgili disk bölümünün başından itibaren kaçıncı sektörde boot-sektörün yer aldığı bilgisi vardır. Eğer buradaki değer 0 ise boot sektör hemen ilgili disk bölümünün başındadır. Fakat bazen boot sektör hemen ilgili disk bölümünün başından başlatılmaz. Track hizalaması için biraz öteye alınabilir.

**Yeni Toplam Sektör Uzunluğu:** Bu alanda yukarıda da belirtildiği gibi daha uzun bir toplam sektör sayısı bilgisi DWORD olarak tutulmaktadır.

**Fiziksel Sürücü Numarası:** Burada IO kontrol kartlarının arayüzleri için fiziksel sürücüye erişmekte kullanılan numara bulunmaktadır. Birinci hard disk için bu numara 0x80, ikinci hard disk için 0x81 biçiminde devam etmektedir. Yani volüme bir hard diskteyse o hard diskin kaçıncı hard disk olduğu bu alanda yazmaktadır.

**Genişletilmiş Boot Id’si (Extended Boot Signature):** Burada 0x28 ya da 0x29 değeri bulunur. Bu değer ilgili disk bölümündeki dosya sisteminin türü hakkında bilgi verir. 0x29 FAT dosya sistemi olduğu anlamına gelmektedir.

**Volümün Seri Numarası (Volume Serial Number):** Volüm formatlanırken ona 4 byte’lık rastgele bir seri numarası verilir. Eskiden bu seri numarası floppy zamanlarında bazı programlar tarafından floppy’nin değiştirilip değiştirilmediğini kontrol amacıyla kullanılırdı. Bugün pek bir kullanım amacı kalmamıştır. (Bu değer teil bir değer değildir. Kopya koruma gibi amaçlarla bu değerden faydalanılması doğru yaklaşım değildir.)

**Volüm Etiketi (Volume Label):** Bu alanda 11 byte’lık volüme etiketi bulunmaktadır. Ancak volüm etiketi FAT dosya sistemlerinde kök dizinde “Volume Label” özniteliğine (attribute) sahip bir dizin girişi olarak da tutulmaktadır. DOS ve Windows uzun süredir volüm etiketini BOOT sektör BPB bloğunbdan değil bu dosyadan almaktadır.

**Dosya Sisteminin Türü:** Burada dosya sisteminin türü yazı olarak tutulmaktadır (örneğin “FAT32”, “FAT16”, “FAT12” gibi). Dolayısıyla dosya sistemninin türü hemen disk editörle görsel olarak belirlenebilir.

BPB bloğundaki bilgilerden hareketle tüm data alanlarının hangi mantıksal (relative) sektörden başlayıp ne uzunlukta sürdüğü belirlenebilir:

1) Boot sektör volümün ilk sektörüdür. Onun yeri bellidir.

2) FAT bölümü BPB’deki 0x0E offsetinden çekilen “Ayrılmış sektörlerin Sayısı” mantıksal sektöründen başlar. Uzunluğu BPB’deki FAT’in Bir Kopyasının Sektör Sayısı ile FAT Kopyalarının Sayısı alanındaki bilgilerin çarpımıyla bulunabilir.

3) Root Dir bölümü FAT’in bitişinden başlar ve uzunluğu BPB’deki Root Dir Girişlerinin Sayısı alanındaki değerden faydalanılarak bulunur.

4) Data bölümü Root Dir bölümünün bitişinden başlar.

**Boot Sektör Yükleyici Programı**

Boot Sektör yükleyici programı boot sektör’de hemen BPB’nin bittiği yerden başlar ve boot sektörün sonuna kadar devam eder. Boot sektör yükleyici programı sembolik makine dilinde yazılıp buraya yerleştirilmiştir. Boot işlemi sırasında eğer boot edilecek disk bölümü bu bölümse boot yükleyici programı (boot loader) FAT dosya sisteminin boot sektörünü diskten okuyarak kontrolü buradaki programa aktarır. Bu program işletim sisteminin yüklenme sürecini başlatır.



Eğer bu FAT disk bölümü bootable değilse yani yüklenecek bir işletim sistemine sahip değilse buradaki küçük yükleyici program bir error mesajı ekrana basarak sistemi yeniden başlatır.

**Windows’ta Aşağı Seviyeli Disk İşlemlerinin Yapılması**

Bilindiği gibi disk sektörlerinin okunup yazılması aşağı seviyeli olarak Disk IO deneteleyicisinin (IDE, SATA gibi) programlanmasıyla yapılmaktadır. Ancak Windows ve Linux gibi işletim sistemleri bu işlemi yapan kodları zaten kendi içlerinde bulundururlar. User moddan bizim bu kodları kullanabilmemizi aygıt sürücülerle sağlarlar. Bu aygıt sürücüler diski sanki bir dosyaymış gibi bize göstermektedir. Biz bu aygıt sürücüleri Windows sistemlerindeki dosya açan CreateFile API fonksiyonuyla açarız. Sonra SetFilePointer API fonksiyonuyla ilgili sektörün başladığı pozisyona dosya göstericisini konumlandırırz. Oradan ReadFile API fonksiyonuyla okuma yapıp, WriteFile API fonksiyonuyla oraya yazma yapabiliriz. Bu işlemleri yapan iki dosya DiskIO.h ve DiskIO.c ismiyle kurs için hazırlanmıştır:

/\* DiskIO.h \*/

#ifndef DISKIO\_H\_

#defineDISKIO\_H\_

#include<Windows.h>

/\* Symbolic Constants \*/

#defineBYTE\_PER\_SECTOR 512

/\* Function Prototypes \*/

HANDLE OpenDisk(int disk, DWORD dwDesiredAccess);

HANDLE OpenVolume(int drive, DWORD dwDesiredAccess);

BOOL ReadSector(HANDLE hDrive, \_\_int64 sector, DWORD count, void \*buf);

BOOL WriteSector(HANDLE hDrive, \_\_int64 sector, DWORD count, constvoid \*buf);

#endif

/\* DiskIO.c \*/

#include<stdio.h>

#include<Windows.h>

#include"DiskIO.h"

/\* static Function Prototypes \*/

staticBOOL SetFilePosition(HANDLE hDrive, \_\_int64 distance, DWORD MoveMethod);

/\* Function definitions \*/

HANDLE OpenDisk(intdisk, DWORDdwDesiredAccess)

{

HANDLE hDisk;

char driverName[32];

DWORD shareMode = 0;

sprintf(driverName, "\\\\.\\PhysicalDrive%d", disk);

if (dwDesiredAccess&GENERIC\_READ)

shareMode |= FILE\_SHARE\_READ;

if (dwDesiredAccess&GENERIC\_WRITE)

shareMode |= FILE\_SHARE\_WRITE;

returnCreateFile(driverName, dwDesiredAccess, shareMode, NULL, OPEN\_EXISTING, FILE\_ATTRIBUTE\_NORMAL, NULL);

}

HANDLE OpenVolume(intdrive, DWORDdwDesiredAccess)

{

HANDLE hDisk;

char driverName[32];

DWORD shareMode = 0;

sprintf(driverName, "\\\\.\\%c:", drive + 'A');

if (dwDesiredAccess&GENERIC\_READ)

shareMode |= FILE\_SHARE\_READ;

if (dwDesiredAccess&GENERIC\_WRITE)

shareMode |= FILE\_SHARE\_WRITE;

returnCreateFile(driverName, dwDesiredAccess, shareMode, NULL, OPEN\_EXISTING, FILE\_ATTRIBUTE\_NORMAL, NULL);

}

BOOL ReadSector(HANDLEhDrive, \_\_int64sector, DWORDcount, void \*buf)

{

\_\_int64 location = (\_\_int64)sector \* BYTE\_PER\_SECTOR;

DWORD dwBytes = count \* BYTE\_PER\_SECTOR;

DWORD dwRead;

if (!SetFilePosition(hDrive, location, FILE\_BEGIN))

returnFALSE;

if (!ReadFile(hDrive, buf, dwBytes, &dwRead, NULL))

returnFALSE;

returnTRUE;

}

BOOL WriteSector(HANDLEhDrive, \_\_int64sector, DWORDcount, void \*buf)

{

\_\_int64 location = (\_\_int64)sector \* BYTE\_PER\_SECTOR;

DWORD dwBytes = count \* BYTE\_PER\_SECTOR;

DWORD dwRead, status;

if (!SetFilePosition(hDrive, location, FILE\_BEGIN))

returnFALSE;

if (!DeviceIoControl(hDrive, FSCTL\_DISMOUNT\_VOLUME, NULL, 0, NULL, 0, &status, NULL))

returnFALSE;

if (!WriteFile(hDrive, buf, dwBytes, &dwRead, NULL))

returnFALSE;

returnTRUE;

}

staticBOOL SetFilePosition(HANDLEhDrive, \_\_int64distance, DWORDMoveMethod)

{

LARGE\_INTEGER li;

li.QuadPart = distance;

li.LowPart = SetFilePointer(hDrive, li.LowPart, &li.HighPart, MoveMethod);

if (li.LowPart == 0xFFFFFFFF && GetLastError() != NO\_ERROR)

returnFALSE;

returnTRUE;

}

Burada OpenDisk tüm diski (disk bölümleri dikkate alınmaksızın) tek bir dosya gibi açmak için, OpenVolume ise yalnızca ilgili disk bölümünü bir dosya açmak için kullanılır. Örneğin biz G volümünün boot sektörünü şöyle ekrana yazdırabiliriz:

#include<stdio.h>

#include<Windows.h>

#include"DiskIO.h"

void ExitSys(LPCSTR lpszMsg, int status);

int main(void)

{

BYTE buf[512];

HANDLE hVolume;

int i;

if ((hVolume = OpenVolume(6, GENERIC\_READ)) == INVALID\_HANDLE\_VALUE)

ExitSys("OpenVolume", EXIT\_FAILURE);

if (!ReadSector(hVolume, 0, 1, buf))

ExitSys("ReadSector", EXIT\_FAILURE);

for (i = 0; i < 512; ++i) {

printf("%02X ", buf[i]);

if (i % 16 == 15)

printf("\n");

elseif (i % 8 == 7)

printf(" ");

}

printf("\n");

CloseHandle(hVolume);

return 0;

}

void ExitSys(LPCSTRlpszMsg, intstatus)

{

DWORD dwLastError = GetLastError();

LPTSTR lpszErr;

if (FormatMessage(FORMAT\_MESSAGE\_ALLOCATE\_BUFFER | FORMAT\_MESSAGE\_FROM\_SYSTEM, NULL, dwLastError,

MAKELANGID(LANG\_NEUTRAL, SUBLANG\_DEFAULT), (LPTSTR)&lpszErr, 0, NULL)) {

fprintf(stderr, "%s: %s", lpszMsg, lpszErr);

LocalFree(lpszErr);

}

exit(status);

}

**FAT Dosya Sisteminde BPB Bilgilerinin Elde Edilmesi**

Yukarıda da belirtildiği gibi FAT dosya sisteminin kalbi boot sektörün başındaki BPB (Bios Parameter Block) alanıdır. O alan okunup yorumlanmadıktan sonra bu dosya sistemi üzerinde işlemler yapılamaz. Bu nedenle sistem programcısı öncelikle BPB alanını okuyarak bir yapıya yerleştirmelidir. BPB alanını okuyup bir yapıya yerleştiren bir fonksiyon şöyhle yazılabilir:

/\* FatSys.h \*/

#ifndef FATSYS\_H\_

#defineFATSYS\_H\_

#include<Windows.h>

/\* Symbolic contants \*/

#defineFILE\_INFO\_LENGTH 32

/\* Type Declaration \*/

typedefstructtagBPB {

HANDLE hVolume; /\* volume \*/

WORD fatLen; /\* number of sectors of FAT (A) \*/

WORD rootLen; /\* number of sectors of ROOT \*/

WORD fatCopyNum; /\* number of copies of FAT (A) \*/

DWORD totalSec; /\* total sector (A) \*/

WORD bps; /\* byte per sector(A) \*/

WORD spc; /\* sector per cluster(A) \*/

WORD reservedSect; /\* reserved sector(A) \*/

BYTE medDes; /\* media descriptor byte(A) \*/

WORD spt; /\* sector per track(A) \*/

WORD rootEntryNum; /\* root entry(A) \*/

WORD headNum; /\* number of heads(A) \*/

WORD hidNum; /\* number of hidden sector(A)\*/

WORD tph; /\* track per head \*/

WORD fatOrigin; /\* fat directory location \*/

WORD rootOrigin; /\* root directory location \*/

WORD dataOrigin; /\* first data sector location \*/

DWORD serialNumber; /\* Volume Serial Number (A) \*/

BYTE volumeName[12]; /\* Volume Name (A) \*/

} BPB;

/\* Function Prototypes \*/

BOOL GetBPB(HANDLE hVolume, BPB \*pBPB);

/\* FatSys.c \*/

#include<stdio.h>

#include<windows.h>

#include"DiskIO.h"

#include"FatSys.h"

BOOL GetBPB(HANDLEhVolume, BPB \*pBPB)

{

BYTE bootSect[BYTE\_PER\_SECTOR];

if (!ReadSector(hVolume, 0, 1, bootSect))

returnFALSE;

pBPB->hVolume = hVolume;

pBPB->bps = \*(WORD \*)(bootSect + 0x0B);

pBPB->spc = \*(BYTE \*)(bootSect + 0x0D);

pBPB->reservedSect = \*(WORD \*)(bootSect + 0x0E);

pBPB->fatLen = \*(WORD \*)(bootSect + 0x16);

pBPB->rootLen = \*(WORD \*)(bootSect + 0x11) \* FILE\_INFO\_LENGTH / pBPB->bps;

pBPB->fatCopyNum = \*(BYTE \*)(bootSect + 0x10);

if (\*(WORD \*)(bootSect + 0x13))

pBPB->totalSec = \*(WORD \*)(bootSect + 0x13);

else

pBPB->totalSec = \*(DWORD \*)(bootSect + 0x20);

pBPB->medDes = \*(bootSect + 0x15);

pBPB->spt = \*(WORD \*)(bootSect + 0x18);

pBPB->rootEntryNum = \*(WORD \*)(bootSect + 0x11);

pBPB->headNum = \*(WORD \*)(bootSect + 0x1A);

pBPB->hidNum = \*(WORD \*)(bootSect + 0x1C);

pBPB->tph = (WORD)(pBPB->totalSec / pBPB->spt / pBPB->headNum);

pBPB->fatOrigin = pBPB->reservedSect;

pBPB->rootOrigin = pBPB->reservedSect + pBPB->fatLen \* pBPB->fatCopyNum;

pBPB->dataOrigin = pBPB->rootOrigin + pBPB->rootLen;

pBPB->serialNumber = \*(DWORD \*)(bootSect + 0x27);

memcpy(pBPB->volumeName, bootSect + 0x2B, 11);

pBPB->volumeName[11] = '\0';

returnTRUE;

}

Şöyle bir kodla test işlemi yapılabilir:

/\* Test.c \*/

#include<stdio.h>

#include<Windows.h>

#include"DiskIO.h"

#include"FatSys.h"

void ExitSys(LPCSTR lpszMsg, int status);

int main(void)

{

BYTE buf[512];

HANDLE hVolume;

BPB bpb;

if ((hVolume = OpenVolume(6, GENERIC\_READ)) == INVALID\_HANDLE\_VALUE)

ExitSys("OpenVolume", EXIT\_FAILURE);

if (!GetBPB(hVolume, &bpb))

ExitSys("GetBPB", EXIT\_FAILURE);

printf("%ld\n", bpb.totalSec);

printf("%d\n", bpb.fatOrigin);

printf("%d\n", bpb.rootOrigin);

/\* .... \*/

CloseHandle(hVolume);

return 0;

}

void ExitSys(LPCSTRlpszMsg, intstatus)

{

DWORD dwLastError = GetLastError();

LPTSTR lpszErr;

if (FormatMessage(FORMAT\_MESSAGE\_ALLOCATE\_BUFFER | FORMAT\_MESSAGE\_FROM\_SYSTEM, NULL, dwLastError,

MAKELANGID(LANG\_NEUTRAL, SUBLANG\_DEFAULT), (LPTSTR)&lpszErr, 0, NULL)) {

fprintf(stderr, "%s: %s", lpszMsg, lpszErr);

LocalFree(lpszErr);

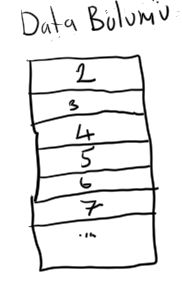
}

exit(status);

}

**Cluster’ların Numaralandırılması**

FAT dosya sisteminde Data bölümü clusterlar’dan oluşmaktadır. Her cluster’a ilki 2’den başlamak üzere (0 ve 1 numaralı cluster’lar bir sebepten dolayı kullanılmamaktadır) bir cluster numarası karşılık düşürülmüştür. Yani data bölümünün hemen başında 2 numaralı cluster bulunmaktadır. Örneğin bir cluster’ın 4 sektörden oluştuğunu varsayalım. Bu durumda data bölümünün ilk 4 sektörü 2 numaralı cluster’a sonraki 4 sektörü 3 numaralı cluster’a karşılık gelmektedir.



Bir cluster’ı okuyup yazan fonksiyonlar oluşturabiliriz. Örneğin:

BOOL ReadCluster(constBPB \*pBPB, DWORDclu, void \*buf)

{

DWORD sect = pBPB->dataOrigin + (clu - 2) \* pBPB->spc;

return ReadSector(pBPB->hVolume, sect, pBPB->spc, buf);

}

BOOL WriteCluster(constBPB \*pBPB, DWORDclu, constvoid \*buf)

{

DWORD sect = pBPB->dataOrigin + (clu - 2) \* pBPB->spc;

return WriteSector(pBPB->hVolume, sect, pBPB->spc, buf);

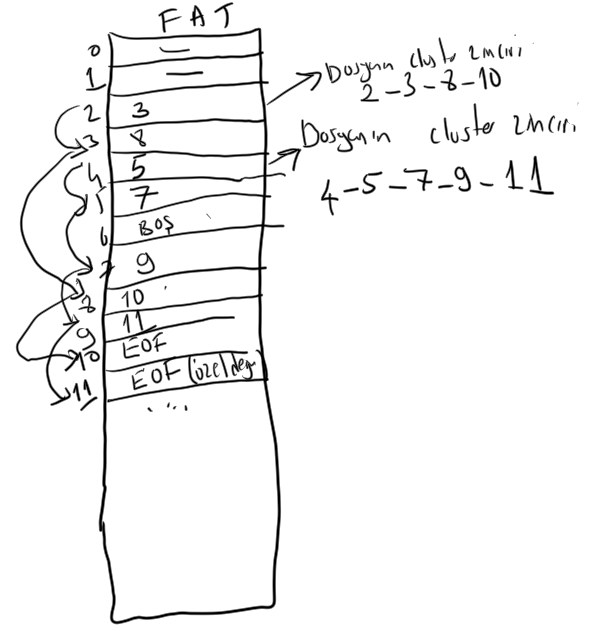
}

Burada görüldüğü gibi önce okunacak ya da yazılacak cluster’ın data bölümünün kaçıncı sektöründen başladığı hesaplanmıştır. Sonra da okuma ve yazma işlemleri ReadSector ve WriteSector fonksiyonlarıyla yapılmıştır.

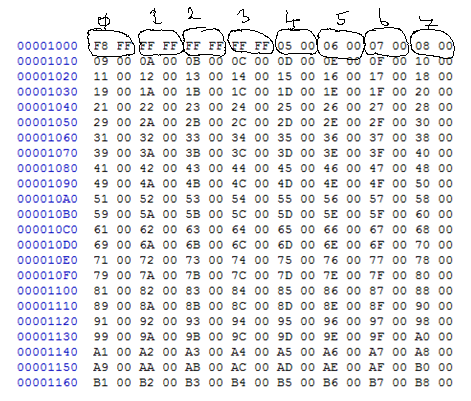
**FAT Bölümü**

Daha önceden de belirtildiği gibi dosyaları oluşturan cluster’lar data bölümünde dağınık olarak bulunabilirler. İşte hangi dosyaların data bölümünün hangi cluster’larında olduğu bilgisi FAT tablosunda bulunmaktadır. FAT tablosunda bütün dosyaların parçalarının tek tek hangi cluster’larda olduğu bilgisi vardır. Örneğin “a.dat” isimli dosyanın cluster’ları sırasıyla 12, 23, 24, 36, 41 olabilir. Bu numaralara ilgili dosyanın cluster zinciri (cluster chain) denilmektedir.

FAT bağlı listelerden oluşan bir yapıdadır. Bir dosyanın ilk cluster bilgisi dizin girişinden alınır (bu konu ileride ele alınmaktadır). FAT tablosu FAT elemanlarından oluşmaktadır. Her FAT elemanının ilki sıfırdan başlamak üzere (ancak 0 ve 1’in kullanılmadığını anımsayınız). Bir dosyanın FAT zinciri şöyle bulunur: Öncelikle dosyanın ilk cluster’ının ne olduğu bilinmek zorundadır. Bu bilgi dizin girişlerinden elde edilir. Sonra FAT’te o fat elemanına gidilir. Orada bulunan değer dosyanın sonraki cluster’ını belirtmektedir. Böyle böyle zincir bağlı listede takip edilir. Özel bir değer görüldüğünde (EOF) durulur. Örneğin aşağıdaki çizimde iki dosyanın FAT zinciri gösterilmektedir.



FAT bölümü FAT sektörlerinden FAT sektörleri de FAT elemanlarındna oluşur. Pekiyi bir FAT elemanı ne uzunluktadır? İşte FAT dosya sisteminin FAT12, FAT16 ve FAT32 olmak üzere üç değişik biçimi vardır. FAT12’de bir FAT elemanı 12 bit, FAT16’da 16 bit ve FAT32’de 32 bittir. Bu durumda örneğin FAT16 olan bir sistemin FAT bölümünün bir kısmı aşağıda verilmiş olsun:



FAT’in ilk iki elemanında özel değerler bulunmaktadır. İşletim sistemi bu özel değerleri (magic numbers) kontrol edebilir. (Eskiden DOS ve Windows bunu kontrol ediyordu. Artık Windows kontrol etmemektedir.) Bu dğerler yanlış ise FAT bölümünün bozulmuş olduğu sonucu çıkarılabilmektedir. FAT12’de özel değer F8 FF FF (Burada F8 ortam betimleyicisidir), FAT16’da F8 FF FF FF ve FAT32’de F8 FF FF FF FF FF FF FF biçimindedir. Yukarıdaki FAT örneğinde 2 ve 3 numaralı cluster’larda tek cluster’lık küçük iki dosya vardır. Çünkü bu FAT elemanlarındaki FF FF değeri özel bir değerdir ve EOF anlamına gelir. Yani FAT16’da cluster zincirini izlerken biz en sonunda FF FF ile karşılaşırız. Bu durum zincirin bittiğini gösterir.

FAT elemanlarındaki her değer “sonraki cluster numarası” anlamına gelmemektedir. Bazı özel değerler bazı özel anlamlara gelmektedir. İşte aşağıda o özel değerler belirtiliyor:

|  |  |
| --- | --- |
| **FAT Elemanındaki Değer (HEX)** | **Özel Anlamı** |
| FAT12: 000  FAT16: 0000  FAT32: 00000000 | Boş cluster. Bu cluster boştur. Dolayısıyla işletim sistemi tarafından tahsis edilebilir. Volüm formatlandığında FAT elemanlarında sıfır değerleri vardır. |
| FAT12: 001  FAT16: 0001  FAT32: 00000001 | 1 numaralı cluster kullanılmamaktadır. Dolayısıyla FAT tablosu içerisinde normal olarak böyle bir FAT elemanı bulunmaz |
| FAT12: [002:FEF]  FAT16: [0002:FFEF]  FAT32: [00000002:0FFFFFEF] | Sonraki cluster numarası. Bir FAT elemanında bulunan normalk değer cluster zincirindeki sonraki elemanın değeridir. |
| FAT12: [FF0:FF6]  FAT16: [FFF0:FFF6]  FAT32: [0FFFFFF0:0FFFFFF6] | Reserved. Bu değerler bazı sistemler tarafından onlara özgü bazı amaçlarla kullanılsın diye ayrılmıştır. |
| FAT12: FF7  FAT16: FFF7  FAT32: 0FFFFFF7 | Bad cluster. Aşağı seviyeli formatlama yapılırken eğer bir sektörde sorun çıkarsa o sektörün içinde bulunduğu cluster “bad cluster” olarak işaretlenir. İşletim sistemi de oraya dosya yerleştirmez. |
| FAT12: [FF8:FFF]  FAT16: [FFF8:FFFF]  FAT32: [0FFFFFF8:0FFFFFFF] | Son cluster. Bu değerleri dosyanın cluster zincirindeki son cluster’ı belirtmektedir. |

FAT32’de sanıldığının aksine her FAT elemanı 32 bit değil 28 bittir. Dolayısıyla FAT tablosunda FAT elemanlarının yüksek anlamlı 4 biti (yani yüksek anlamlı hex digiti) her zaman sıfırdır.

**FAT Elemanlarının Elde Edilmesi**

FAT tablosunun aşağıdaki gibi unsigned char türünden bir dizi içerisine okunduğunu varsayalım:

unsigned char fat[FAT\_SIZE];

Şimdi clu numaralı FAT elemanını elde eden bir fonksiyonu 16 bit, 12 bit ve 32 bit FAT için yazalım.

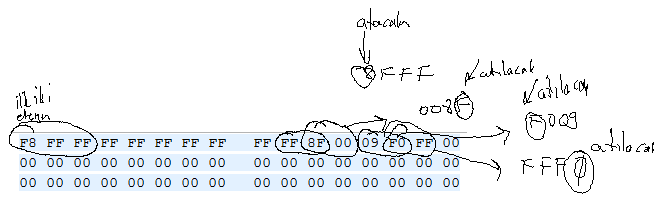
WORD GetNextClu16(WORD clu)

{

return \*(WORD \*)(fat + clu \* 2);

}

FAT16’da bir FAT elemanı 2 byte olduğu için cluster numarası 2 ile çarpılmıştır. Şimdi aynı fonksiyonu FAT12 için yazalım. FAT12’de bir FAT elemanı 12 bit yani 3 hex digit’tir. Fat elemanının FAT’teki konumuna cluster numarasının 1.5 ile çarpılmasıyla gidilir. Ancak oradan çekilen WORD değerin hangi 4 bitlik kısmının atılacağı cluster numarasının tek mi çift mi olduğuna bağlıdır. Örneğin aşağıda 12 bit için girişler verilmektedir:



12 bit FAT’te normal olarak 3 hex digit elde edilirken eğer cluster numarası çift ise bizim sonraki byte’ın düşük anlamlı hex digitini almamız, cluseter numarası tek ise bizim ilk byte’ın yüksek anlamlı hex digitini almamız gerekir. Bunun formülüze edilmiş hali şöyle açıklanabilir: Biz her zaman FAT12’de cluster numarasını 1.5 ile çarpıp oradan WORD çekeriz. Eğer cluster numarası çift ise yüksek anlamlı 4 biti atarız, tek ise düşük anlamlı 4 biti aratız. O halde GetNextClu12 fonksiyonu şöyle yazılabilir:

WORD GetNextClu12(WORD clu)

{

if (clu % 2 == 0)

return \*(WORD \*)(fat + clu \* 3 / 2 ) & 0x0FFF;

else

return \*(WORD \*)(fat + clu \* 3 / 2 ) >> 4;

}

Aynı fonksiyon şöyle de yazılabilirdi:

WORD GetNextClu12(WORD clu)

{

WORD wordVal = \*(WORD \*)(fat + clu \* 3 / 2 );

return clu & 1 ? wordVal >> 4 : wordVal &0x0FFF;

}

FAT32’de ilgili FAT elemanın byte offset’i cluster numarası 4 ile çarpılarak elde edilir:

DWORD GetNextClu32(DWORD clu)

{

return \*(DWORD \*)(fat + clu \* 4);

}

Yukarıdaki fonksiyonları bir döngü içerisinde çağırarak dosyanın cluster zincirini elde edebiliriz:

WORD \*GetClusterChain12(BYTE \*fat, WORDfirstClu)

{

staticWORD chain[MAX\_FAT12\_CHAIN];

WORD clu = firstClu, i;

for (i = 0; clu < 0xFF8; ++i) {

chain[i] = clu;

clu = GetNextClu12(fat, clu);

}

chain[i] = 0;

return chain;

}

WORD \*GetClusterChain16(BYTE \*fat, WORDfirstClu)

{

staticWORD chain[MAX\_FAT16\_CHAIN];

WORD clu = firstClu, i;

for (i = 0; clu < 0xFFF8; ++i) {

chain[i] = clu;

clu = GetNextClu16(fat, clu);

}

chain[i] = 0;

return chain;

}

DWORD \*GetClusterChain32(BYTE \*fat, DWORDfirstClu)

{

staticDWORD chain[MAX\_FAT32\_CHAIN];

WORD clu = firstClu, i;

for (i = 0; clu < 0xFFF8; ++i) {

chain[i] = clu;

clu = GetNextClu32(fat, clu);

}

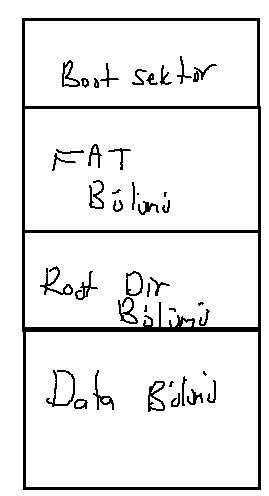
chain[i] = 0;

return chain;

}

**FAT12 ve FAT16 Dosya Sistemlerinde Dizin Girişleri ve Dizinlerin Organizasyonu**

Bir dosyanın tüm bilgileri FAT12 VE FAT16’da 32 byte’lık dizin girişlerinde saklanmaktadır. FAT32’de uzun dosya isimleri de devreye sokulmuştur. Bu nedenle FAT32 daha sonra ele alınacaktır. Örneğin Windows’taki FindFirstFile ve FindNextFile API fonksiyonları dosya bilgilerini bu 32 byte’lık girişlerden elde eder. Volümün RootDir bölümünde kök dizindeki dosyalara ilişkin girişler tutulmaktadır. Bu girişlerin sayısının sınırlı olduğunu anımsayınız.



FAT dosya sistemlerinde dosya ile dizin arasında organizasyonel bir fark yoktur. Dizinlerin de tıpkı dosyalar gibi cluster zincirleri vardır. Fakat dizinlerin cluster zincirlerinde 32 byte’lık dizin girişi bulunur. Başka bir deyişle dizinler onların girişlerini tutan dosyalar gibidir. Örneğin:



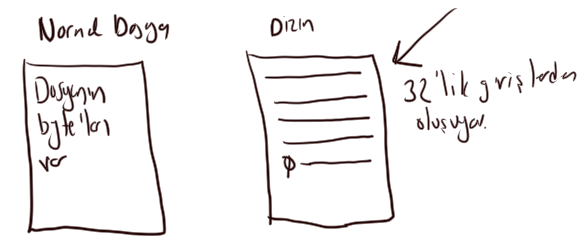
32 byte’lık dizin girişinin formatı şöyledir:

|  |  |  |
| --- | --- | --- |
| **Offset (Hex)** | **Uzunluk** | **Açıklama** |
| [00:0A] | 11 byte | Dosyanın ismi ve uzantısı. FAT12 ve FAT16 sistemlerinde bir dosyanın ismi 8 karakteri, uzantısı da 3 karakteri geçemyordu. Eğer isim ya da uzantı 8 ve 3’ten kısaydsa geri kalan alanlar boşluk karakteriyle (0x20) doldurulmaktadır. |
| 0B | BYTE | Dosyanın özelliği. Dosya özelliği kursun başlarında da ele alındığı gibi bit bit anlamlı bir alandır.    Görüldüğü gibi girişin bir dosya mı yoksa bir dizin mi olduğu yalnızca özellik byte’ının 4’üncü bitinden anlaşılmaktadır. |
| [0C:15] | 10 byte | Reserved. Bu alan FAT 12 ve FAT16’da reserved bırakılmıştır. Ancak uzun dosya isimlerinin desteklendiği VFAT sisteminde bu alan başka amaçla kullanılmaktadır. |
| 16 | WORD | Dosyanın son değiştirildiği zamanı belirtir. Burası bir bit alanı biçiminde aşağıdaki gibi kodlanmıştır. Saniye olarak yalnızca çift saniyelerin tutulduğuna dikkat ediniz. Saniye kısmındaki değer iki ile çarpılmalıdır: |
| 18 | WORD | Burada dosyanın son değiştirildiği tarih bilgisi bulunmaktadır. Bu alandaki bilgi de aşağıdaki gibi bit bit kodlanmıştır. Yıl için orijin noktası 1980’dir.    Yani yıl alanında yazan değere 1980 toplamak gerekir. |
| 2A | WORD | Dosyanın ilk cluster numarası. Bilindiği gibi dosyanın FAT zincirinin izlenebilmesi için onun ilk cluster numarasının bilinmesi gerekir. İşte dosyaların ilk cluster numarfaları dizin girişlerinde bulunmaktadır. |
| 2C | DWORD | Dosyanın byte uzunluğu. Bu alanda dosyanın DWORD olarak byte uzunluğu bulunmaktadır. |

İşletim sistemi dizin girişlerini tararken nerede duracaktır? İşte 32’lik dizin girişlerinin ilk byte’ı eğer 0 ise bu dizin listesinin sonuna gelindiği anlamına gelir. Dosya silindiğinde dosyanın FAT zinciri sıfırlanır. Ancak dizin girişi sıfırlanmaz. Yalnızca onun ilk byte’ı 0xE5 yapılır. Artık 0xE5 ile başlayan dizin girişlerine yeni bir dosya çekilebilecektir.

**FAT Dosya Sisteminde Alt Dizinlerin Organizasyonu**

Aslında dizinlerle dosyalar arasında organizasyonel olarak hiçbir fark yoktur. Dizinler de tamamen dosyalar gibi 32’lik girişlere ve FAT zincirine sahiptir. Dizinlerle dosyalar arasındaki en önemli fark dosyaların içerisinde gerçek dosya bilgileri varken dizinlerin içerisinde o dizinlerdeki dosyaların hangi dosyalar olduğunu gösteren 32’lik dizin girişlerinin olmasıdır. Dizinler de çok fazla dosya içerebilirler. Bu durumda onların da cluster zincirleri birden fazla cluster’dan oluşabilir.



Windows ve UNIX/Linux sistemlerinde kullanılan dosya sistemlerinde bir alt dizin yaratıldığında işletim sistemi o alt dizinin içerisinde “.” ve “..” isimli iki alt dizin girişini otomatik olarak oluşturmaktadır. Gerçekten de FAT dosya sisteminde bir alt dizin yarattığımızda onun ilk iki 32’lik girişinde “.” ve “..” dizinlerini görürüz. “.” dizini bulunaln dizinin 32’lik girişinin aynısını, “..” girişi de üst dizinin dizin girişinin aynısını barındırır.

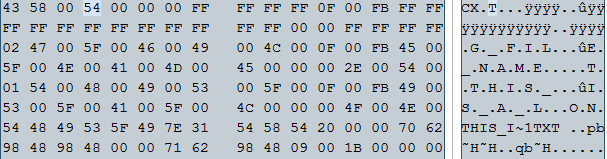
**VFAT Dizin Girişleri**

Klasik FAT12 ve FAT16 sistemlerinde yukarıda da gördüğümüz gibi dizin girişleri 32 byte uzunluğundadır. 32 byte’lık dizin girişlerinde dosya isimleri ve uzantıları 8 + 3 karakter olabilmektedir. Ayrıca 32’lik dizin girişlerinde dosyalar için yalnızca son değiştirilme tarihi ve zamanının tutulduğunu anımsayınız. Microsoft Windows sistemleriyle birlikte FAT’teki 32’lik dizin girişlerini geçmişe doğru uyumu koruyacak biçimde genişletmiştir. Bu özelliğe VFAT (virtual FAT) denilmektedir. VFAT yeni bir dosya sistemi değildir. Uzun dosya isimlerinin dizin girişlerinde bulunabilmesi için 32’lik dizin girişlerinin genişletilmiş halidir. VFAT geçmişe doğru uyumludur. Ayrıca VFAT’teki dosya isimleri UNICODE karakterlerden oluşturulabilmektedir. Dosya isimlerinde boşluk karakterleri de VFAT ile birlikte bulunabilir hale getirilmiştir. Ayrıca dosya isminde birden fazla ‘.’ karakteri de bulundurulabilmektedir. VFAT sisteminde her uzun dosya ismi için bir de onun kısası olan 32’lik ayrı bir giriş de bulundurulmaktadır. Bu kısa isme ilişkin 32’lik giriş uzun isimlere ilişkin 32’lik girişlerin sonunda bulunmaktadır.

Uzun dosya girişleri birden fazla 32’lik girişlerde tutulur. 32’lik girişler ters dizilmiştir. Yani dosya isimlerinin sonraki karakterlerine karşı gelen 32’lik girişler önde bulunur. Uzun dosya girişlerinin 32’lik bir parçasının formatı şöyledir:

|  |  |  |
| --- | --- | --- |
| **Offset (Hex)** | **Uzunluk** | **Açıklama** |
| 0 | BYTE | 32’lik girişlerin sıra numarası ve tahsisat durumunu belirtir. Buradaki byte’ın bitleri şu şöyle organize edilmiştir:    İlk 6 bit uzun dosya için ayrılan 32’lik girişlerin sıra numarasını belirtir. Sıra numaraları büyükten küçüğe dizilmektedir. 6’ıncı bit ilgili girişin son giriş olup olmadığını, 7’inci bit ilgili girişin silinmiş olup olmadığını belirtmektedir. |
| [01:0A] | 10 byte | Burada her biri 2 byte olmak üzere dosya isminin beş karakteri bulunur. |
| 0B | BYTE | Burada klasik eskiye uyumlu 32’lik girişte olduğu gibi dosya özelliği bilgisi vardır. |
| 0C | BYTE | reserved |
| OD | BYTE | Checksum. Burada bilgilerin bozulup bozulmadığının anlaşılması için bir checksum değeri tutulmaktadır. |
| [0E:19] | 12 byte | Bu alanda dosya isminin sonraki 6 karakteri UNICODE olarak tutulmaktadır. |
| 1A | BYTE | Burada her zaman 0 değeri bulunur. |
| [1C:1F] | 4 byte | Dosyanın sonraki UNICODE olarak iki karakteri bulunur. |

Görüldüğü gibi bir 32’lik giriş uzun dosya isminin 13 karakterini tutabilmektedir. Örneğin “THIS\_IS\_A\_LONG\_FILE\_NAME.TXT” isimli bir dosya ismi toplam 28 karakterdir (nokta isme dahildir ve herhangi bir karakter gibidir.) Bu durumda bu dosya için 13 + 13 + 2 biçiminde toplam 32’lik 3 giriş oluşturulacaktır. Bu üç girişin hemen aşağısında bu dosyanın kısa ismine ilişkin bir 32’lik giriş daha bulundurulur.



**Yol İfadelerinin Çözümlenmesi (Pathname Resolution)**

Bir yol ifadesi verildiğinde işletim sisteminin yol ifadesiyle hedeflenen dosyaya ilişkin bilgileri ele geçerimesi gerekir. Bu sürece yol ifadelerinin çözümlenmesi (pathname resolution) denilmektedir. Anımsanacağı gibi yol ifadeleri mutlak (absolute) ve göreli (relative) olmak üzere ikiye ayrılmaktadır. Eğer yol ifadesinin başında (sürücü ismi varsa bundan sonrasında) ters bölü varsa (UNIX/Linux sistemlerinde düz bölü) bu tür yol ifadelerine mutlak yol ifadeleri denilmektedir. Eğer yol ifadelerinin başında bir ters bölü (ya da düz bölü) yoksa böyle yol ifadelerine de göreli yol ifadeleri denilmektedir. Mutlak yol ifadeleri kök dizinden itibaren çözümlenir. Göreli yol ifadeleri ise prosesin kontrol bloğunda tutulan “çalışma dizininden (current working directory)” itibaren çzöümlenir. Şimdi burada FAT dosya sistemi için mutlak bir yol ifadesinin nasıl çözümlendiğine ilişkin bir örnek verelim. Yol ifademiz “\A\B\C\D.TXT” olsun. İşletim sistemi burada hedef olarak D.TXT dosyasının bilgilerine erişecektir. Bunun önce işletim sisgtemi kök dizine gider. FAT dosya sisteminde kkö dizinin yeri bellidir. Oradaki 32’lik girişlerde sıralı arama yöntemiyle A dizin girişini arayacaktır. Bu dizin girişini bulduğunda onun bir dizine ilişkin olduğunu “özellik byte’ına” bakarak doğrular. A dizin girişinin cluster zincirini elde eder. A’nun cluster’ları içerisindeki bilgileri 32’lik dizin girişi gibi değerlendirir. Bu sefer A’nın cluster’ları içerisinde B girişini arar. Bulursa bunun da bir dizin olduğunu doğrular bu sefer B’nin cluster zincirini elde eder. B’nin cluster’larında C’yi arar. C’nin cluster zincirininde de benzer biçimde D’yi arayacaktır.

Modern işletim sistemleri yol ifadesinin çözümlenmesi sırasında eriştikleri dizin girişlerini bir cache sistemi içerisinde saklamaktadır. Bu cache sistemine sıklıkla “denty cache” denilmektedir. “Dentry Cache” genellikle bir hash tablosu biçiminde oluşturulur. Hash tablosunda anahtar olarak yol ifadesi verilir. Değer olarak da dizin girişi bilgileri elde edilir. Böylece son erişilen dosyalara ilişkin girişler cache’te saklandığı için onlara erişilmek istendiğinde artık hiç disk okuması yapılmadan doğrudan giriş elde edilebilir.

Yol ifadelerinin çözümlenmesine ilişkin bir örnek aşağıdaki gibi olabilir. Aşağıdaki örnekte hiçbir iyileştirme yapılmamıştır. Örneğin her defasında FAT bölümünün tamamı okunmaktadır. Ancak yol ifadelerinin çözümlenmesi için bir fikir vermesi bakımından örnek incelenmelidir:

/\* PathNameResolution.c \*/

#include<string.h>

#include<ctype.h>

#include"DiskIO.h"

#include"PathNameResolution.h"

/\* static Function Prototypes \*/

DIRENTRY \*findDirEntry(void \*cluster, size\_t size, constchar \*fileName);

char g\_cwd[MAX\_PATH] = "g:\\";

BOOL PathNameLookup(constchar \*path, DIRENTRY \*dirEntry)

{

int drive;

char absPath[MAX\_PATH];

char \*pathPos, \*pathComp, \*dirEntries, \*cluCont;

size\_t dirEntrySize, cluSize;

HANDLE hVolume;

BPB bpb;

DIRENTRY \*dentry;

BOOL foundFlag;

WORD clu;

BYTE \*fat;

if (path[1] == ':') {

strcpy(absPath, path);

drive = toupper(path[0]) - 'A';

if (path[2] == '\\')

pathPos = absPath + 3;

else

pathPos = absPath + 2;

}

elseif (path[0] == '\\') {

strcpy(absPath, path);

drive = toupper(g\_cwd[0]) - 'A';

pathPos = absPath+ 1;

}

else {

sprintf(absPath, "%s\\%s", g\_cwd, path);

drive = toupper(absPath[0]) - 'A';

pathPos = absPath + 3;

}

if ((hVolume = OpenVolume(drive, GENERIC\_READ)) == INVALID\_HANDLE\_VALUE)

returnFALSE;

if (!GetBPB(hVolume, &bpb))

returnFALSE;

dirEntrySize = bpb.rootLen \* bpb.bps;

if ((dirEntries = (char \*)malloc(dirEntrySize)) == NULL)

returnFALSE;

if (!ReadSector(hVolume, bpb.rootOrigin, bpb.rootLen, dirEntries))

returnFALSE;

if ((pathComp = strtok(pathPos, "\\")) == NULL)

returnFALSE;

if ((dentry = findDirEntry(dirEntries, dirEntrySize, pathComp)) == NULL) {

printf("cannot find!..\n");

returnFALSE;

}

if ((fat = (BYTE \*)malloc(bpb.fatLen \* bpb.bps)) == NULL)

returnFALSE;

if (!ReadSector(hVolume, bpb.fatOrigin, bpb.fatLen, fat))

returnFALSE;

cluSize = bpb.spc \* bpb.bps;

if ((cluCont = (char \*)malloc(cluSize)) == NULL)

returnFALSE;

while ((pathComp = strtok(NULL, "\\")) != NULL) {

clu = dentry->firstClu;

foundFlag = FALSE;

while (clu < 0xFFF8) {

if (!ReadCluster(&bpb, clu, cluCont))

returnFALSE;

if ((dentry = findDirEntry(cluCont, cluSize, pathComp)) == NULL)

clu = GetNextClu16(fat, clu);

else {

foundFlag = TRUE;

break;

}

}

if (!foundFlag)

returnFALSE;

}

\*dirEntry = \*dentry;;

returnTRUE;

}

DIRENTRY \*findDirEntry(void \*cluster, size\_tsize, constchar \*fileName)

{

DIRENTRY \*dirEntry = (DIRENTRY \*)cluster;

unsignedint i, k;

char fname[11] = { 0x20, 0x20, 0x20, 0x20, 0x20, 0x20, 0x20, 0x20, 0x20, 0x20, 0x20 };

k = 0;

for (i = 0; fileName[i] != '\0'; ++i) {

if (fileName[i] != '.')

fname[k++] = fileName[i];

else

k = 8;

}

for (i = 0; dirEntry->fileName[0] != 0 && i <size / 32; ++i) {

if (!\_strnicmp(dirEntry->fileName, fname, 11))

return dirEntry;

++dirEntry;

}

returnNULL;

}

/\* Test.c \*/

#include<stdio.h>

#include<Windows.h>

#include"DiskIO.h"

#include"FatSys.h"

#include"PathNameResolution.h"

void ExitSys(LPCSTR lpszMsg, int status);

int main(void)

{

DIRENTRY dentry;

int i;

if (!PathNameLookup("g:\\x\\y\\z\\t.txt", &dentry))

ExitSys("PathNameLookup", EXIT\_FAILURE);

printf("%.11s\n", dentry.fileName);

return 0;

}

void ExitSys(LPCSTRlpszMsg, intstatus)

{

DWORD dwLastError = GetLastError();

LPTSTR lpszErr;

if (FormatMessage(FORMAT\_MESSAGE\_ALLOCATE\_BUFFER | FORMAT\_MESSAGE\_FROM\_SYSTEM, NULL, dwLastError,

MAKELANGID(LANG\_NEUTRAL, SUBLANG\_DEFAULT), (LPTSTR)&lpszErr, 0, NULL)) {

fprintf(stderr, "%s: %s", lpszMsg, lpszErr);

LocalFree(lpszErr);

}

exit(status);

}

Pekiyi yol ifadelerinin çözümlenmesi işleminde nasıl iyileştirmeler yapılabilir? Bazıları şunlardır:

- FAT’in tamamı her defasında (yani örneğimizde PathNameLookup) fonksiyonunda okunmayabilir. Toplamda bir kez işin başında okunup global bir alana yerleştirilebilir.

- FAT’in tamamının okunması yerine FAT için bir cache sistemi oluşturulabilir. Çünkü büyük volümlerde FAT çok büyük olabilmektedir. Cache’leme FAT elemanı temelinde yapılabilir (örneğin Linux böyle yapmaktadır). Sector temelinde yapılabilir (örneğin Windows böyle yapmaktadır). Her iki durumda da önce aranan bilginin cache’te olup olmadığına bakılmalıdır. Bunun için hash tabloları kullanmak uygundur.

- Dosyalara ilişkin dizin girişleri için de bir cache sisteminin oluşturulması uygun olur. Böylece aynı dosya ya da dizinler aranırken boşuna tekrar tekrar aynı işlemler yapılmaz. Bu cache sistemine Linux sistemlerinde “dentry cache” denilmektedir.

- Göreli yol ifadelerinin çözümlenmesi için prosesin çalışma dizininin (current working directory) proses kontrol bloğunda saklanması gerekir. Ancak orada çalışma dizininin yalnızca yol ifadesi olarak değil dizin girişi olarak da saklanması uygun olur. Böylece doğrudan göreli yol ifadeleri o dizin girişinden itibaren çözümlenebilir. (Yukarıdaki örnekte prosesin geçerli dizininin g\_cwd dizisi içerisinde yalnızca yol ifadesi olarak saklandığına dikkat ediniz. Bu nedenle o örnekte göreli yol ifadeleri her defasında baştan itibaren çözümlenecektir.)

FAT’in sektör temelinde cache’lenmesine ilişkin bir örnek aşağıdaki gibi olabilir. Bu örnekte FAT’in son kullanılan sektörleri bir cache’te saklanmaktadır. Bu cache’ten eleman çıkartnmada LRU (Least Recently Used) algoritması kullanılmaktadır. Örnek yalnızca FAT16 sistemi için gerçekleştirilmiştir. GetNextClu16 fonksiyonunun cache’e bakarak çalıştığına dikkat ediniz.

/\* Fat.h \*/

#ifndef FAT\_H\_

#defineFAT\_H\_

#include<Windows.h>

#include"BPB.h"

#defineFAT\_CACHE\_TABLE\_SIZE 31

#defineFAT\_CACHE\_SIZE 4

typedefstructtagFATCACHE\_NODE {

DWORD sect;

BYTE \*data;

structtagFATCACHE\_NODE \*next;

structtagFATCACHE\_NODE \*prev;

structtagFATCACHE\_NODE \*lruNext;

structtagFATCACHE\_NODE \*lruPrev;

} FATCACHE\_NODE;

DWORD GetNextFat16(constBPB \*pBPB, DWORD clu);

#endif

/\* Fat.c \*/

#include<stdio.h>

#include<Windows.h>

#include"DiskIO.h"

#include"BPB.h"

#include"Fat.h"

staticFATCACHE\_NODE \*g\_cache[FAT\_CACHE\_TABLE\_SIZE];

BYTE g\_fatCache[FAT\_CACHE\_TABLE\_SIZE];

DWORD g\_count;

FATCACHE\_NODE \*g\_lruHead;

FATCACHE\_NODE \*g\_lruTail;

staticDWORD HashFunc(DWORDdw)

{

returndw % FAT\_CACHE\_TABLE\_SIZE;

}

DWORD GetNextFat16(constBPB \*pBPB, DWORDclu)

{

DWORD sect;

BYTE \*buf;

DWORD hash;

FATCACHE\_NODE \*fcn, \*newNode;

sect = pBPB->fatOrigin + clu \* 2 / 512;

hash = HashFunc(sect);

fcn = g\_cache[hash];

while (fcn != NULL) {

if (fcn->sect == sect) {

if (g\_lruTail != fcn) {

if (fcn->lruPrev != NULL)

fcn->lruPrev->lruNext = fcn->lruNext;

else

g\_lruHead = fcn->lruNext;

if (fcn->lruNext != NULL)

fcn->lruNext->lruPrev = fcn->lruPrev;

fcn->lruPrev = g\_lruTail;

g\_lruTail->lruNext = fcn;

g\_lruTail = fcn;

fcn->lruNext = NULL;

}

return \*(WORD \*)(fcn->data + (clu \* 2 % 512));

}

fcn = fcn->next;

}

if (g\_count == FAT\_CACHE\_SIZE) {

FATCACHE\_NODE \*temp;

temp = g\_lruHead;

g\_lruHead->lruNext->lruPrev = NULL;

g\_lruHead = g\_lruHead->lruNext;

if (temp->prev != NULL) {

temp->prev->next = temp->next;

if (temp->next != NULL)

temp->next->prev = temp->prev;

}

else {

g\_cache[HashFunc(temp->sect)] = temp->next;

}

buf = temp->data;

newNode = temp;

}

else {

if ((buf = (BYTE \*)malloc(BYTE\_PER\_SECTOR)) == NULL)

return 0;

if ((newNode = (FATCACHE\_NODE \*)malloc(sizeof(FATCACHE\_NODE))) == NULL)

return 0;

++g\_count;

}

newNode->data = buf;

newNode->sect = sect;

newNode->prev = NULL;

newNode->next = g\_cache[hash];

g\_cache[hash] = newNode;

if (!ReadSector(pBPB->hVolume, sect, 1, buf))

return 0;

if (g\_lruTail == NULL) {

g\_lruHead = g\_lruTail = newNode;

newNode->lruPrev = NULL;

}

else {

g\_lruTail->lruNext = newNode;

newNode->lruPrev = g\_lruTail;

g\_lruTail = newNode;

}

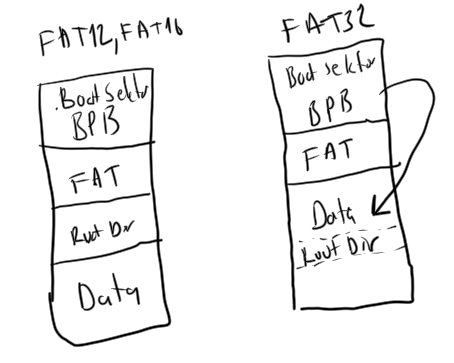
newNode->lruNext = NULL;

return \*(WORD \*)(buf + (clu \* 2 % 512));

}

**FAT32 Sistemlerinin Farklılıkları**

FAT32 sistemlerinde FAT elemanlarının 32 bit (aslında 28 bit) olmasının yanı sıra aynı zamanda boot sektör BPB formatında da farklılıklar vardır. FAT32 sistemlerinde “Root Dir” bölümü data bölümünün içerisine taşınmıştır. Yani kök dizin de sanki bir alt dizin gibi data bölümünde bulunur. Tabii BPB bloğu içerisinde kök dizinin hangi cluster’da olduğu bilgisi tutulmaktadır.



“Root Dir” bölümünün Data bölümüne taşınmasıyla kök dizindeki dosya sayısı sınırı da ortadan kalkmış olmaktadır.

FAT32’deki Boot Sektör BPB Bloğu FAT12 ve FAT16’dakinin biraz değiştirilmiş biçimidir. FAT32 BPB Bloğunun ilk 32 byte’ı (0x20) FAT12 ve FAT16 BPB’sinin aynısı olduğuna dikkat ediniz:

|  |  |  |
| --- | --- | --- |
| **Offset**  **(Hex)** | **Uzunluk** | **İçerik** |
| 00 | 3 byte | Jmp Code |
| 03 | 8 byte | OEM Yorum |
| 0B | WORD | Sektördeki Byte Sayısı |
| 0D | BYTE | Cluster’ın Sektör Uzunluğu |
| 0E | WORD | Ayrılmış Sektörlerin Sayısı |
| 10 | BYTE | FAT kopyalarının Sayısı |
| 11 | WORD | Root Girişlerinin Sayısı |
| 13 | WORD | Volümdeki Toplam Sektör Sayısı |
| 15 | BYTE | Ortam Belirleyicisi (Media Descriptor) |
| 16 | WORD | Fat’in Bir Kopyasındaki Sektör Sayısı |
| 18 | WORD | Track Başına Düşen Sektör Sayısı |
| 1A | WORD | Diskteki Kafa (yüzey) Sayısı |
| 1C | DWORD | Saklı Sektörlerin Sayısı |
| 20 | DWORD | Yeni Toplam Sektör Uzunluğu |
| **24** | **DWORD** | **FAT’in bir kopyasındaki sektör sayısı** |
| **28** | **WORD** | **FAT Aynalama Bilgisi** |
| **2A** | **WORD** | **FAT32 Versiyon Numarası** |
| **2C** | **DWORD** | **Root Dir Bölümünün Cluster Numarası** |
| **30** | **WORD** | **FSINFO Yapısnın Hangi Sektörde Olduğu Bilgisi** |
| **32** | **WORD** | **Boot Sektör Kopyasının Yeri** |
| **34** | **12 Byte** | **Reserved** |
| **40** | **BYTE** | **Fiziksel Drive Numarası** |
| **41** | **BYTE** | **Reserved** |
| **42** | **BYTE** | **Genişletilmiş Boot ID’si** |
| **43** | **DWORD** | **Volümün Seri Numarası (Volume Serial Number)** |
| **47** | **11 Byte** | **Volüm Etiketi (Volume Label)** |
| **52** | **8 byte** | **Dosya Sisteminin Türü** |

**Fat’in Bir Kopyasındaki Sektör Sayısı:** FAT12 ve FAT16’da FAT zaten çok büyük olamıyordu. FAT sektörlerinin sayısı WORD ile (0x16’ıncı offset’e bakınız) belirtilebiliyordu. Ancak FAT32’de FAT çok büyük olduğu için bu alan yenilenmiş ve DWORD’e yükseltilmiştir.

**FAT Aynalama Bilgisi:** Bu alan bitsel olarak yorumlanmaktadır. FAT32’de FAT kopyaları beraber güncellenebilir ya da güncellenmeyebilir (buna İngilizce “mirroring” denilmektedir). Eğer aynalama kapılsa hangi kopyanın aktif olduğu da burada belirtilmektedir. Ayrıntılı bilgi için “FAT32 Filse System Specification” dokümanını inceleyiniz.

**FAT32 Versiyon Numarası:** Burada FAT32 sisteminin versiyon numarası bulunur.

**Root Dir Bölümünün Cluster Numarası:** Burada Root Dir bölümünün Data bölümünün kaçıncı cluster’ından başladığı bilgisi bulunmaktadır. (Anımsanacağı gibi Data bölümünün ilk cluster’ı 2 numaralı cluster’dır. 0 ve 1 cluster değerleri kullanılmamaktadır.)

**FSINFO Yapısnın Hangi Sektörde Olduğu Bilgisi:** FAT32 sistemlerinde FAT12 ve FAT16 sistemlerinde bulunmayan bazı bilgiler FSINFO isimli bir yapı biçiminde tutulmaktadır. Bu alanda FSINFO yapısının hangi sektörde bulunduğu bilgisi vardır. Buradaki değer genellikle 1 biçiminde bulunur (yani boot sektörden hemen sonraki sektörde FSINFO bilgileri vardır.) FSINFO yapısındaki iki önemli eleman FSI\_Free\_Count ve FSI\_Next\_Free elemanlarıdır. FSI\_Free\_Count elemanı FAT’te boş olan cluster’ların sayısını, FSI\_Next\_Free ise ilk boş olan cluster numarasını tutmaktadır. Anımsanacağı gibi FAT12 ve FAT16’da volümdeki boş cluster’ların sayısı herhangi bir yerde tutulmamaktadır. FAT bölümündeki 0 olan FAT elemanları syaılarak boş alan miktarı hesaplanmaktadır.

**Boot Sektör Kopyasının Yeri:** FAT32’de boot sektörün bir kopyası (backup’ı) da ayrıca bir yerde tutulmaktadır. İşte burada boot sektörün kopyasının nerede olduğu bilgi vardır.

**Fiziksel Drive Numarası:** Burada INT 13 kesmesi için volümün fiziksel sürücü numarası bulunmaktadır (Aynı bilgi FAT12 ve FAT16 BPB’sinin 0x24’üncü offsetindebulunmaktadır.)

**Genişletilmiş Boot Id’si (Extended Boot Signature):** Bu bilginin aynısı FAT12 ve FAT16’nın BPB’sinde de vardır. Burada 0x28 ya da 0x29 değeri bulunur. Bu değer ilgili disk bölümündeki dosya sisteminin türü hakkında bilgi verir. 0x29 FAT dosya sistemi olduğu anlamına gelmektedir.

**Volümün Seri Numarası (Volume Serial Number):** Bu bilgi de FAT12 ve FAT16 BPB’sinde vardır. Volüm formatlanırken ona 4 byte’lık rastgele bir seri numarası verilir. Eskiden bu seri numarası floppy zamanlarında bazı programlar tarafından floppy’nin değiştirilip değiştirilmediğini kontrol amacıyla kullanılırdı. Bugün pek bir kullanım amacı kalmamıştır. (Bu değer teil bir değer değildir. Kopya koruma gibi amaçlarla bu değerden faydalanılması doğru yaklaşım değildir.)

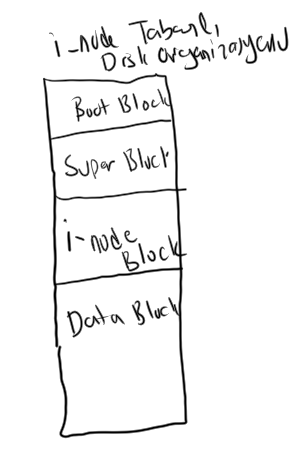
**Volüm Etiketi (Volume Label):** Bu bilgi de FAT12 ve FAT16 BPB’sinde vardır. Bu alanda 11 byte’lık volüme etiketi bulunmaktadır. Ancak volüm etiketi FAT dosya sistemlerinde kök dizinde “Volume Label” özniteliğine (attribute) sahip bir dizin girişi olarak da tutulmaktadır. DOS ve Windows uzun süredir volüm etiketini BOOT sektör BPB bloğunbdan değil bu dosyadan almaktadır.

**Dosya Sisteminin Türü:** Bu bilgi de FAT12 ve FAT16 BPB’sinde vardır. Burada dosya sisteminin türü yazı olarak tutulmaktadır (örneğin “FAT32”, “FAT16”, “FAT12” gibi). Dolayısıyla dosya sistemninin türü hemen disk editörle görsel olarak belirlenebilir.

**UNIX/Linux Sistemlerinde Kullanılan I-NODE Tabanlı Dosya Sistemlerinin Disk Organizasyonu**

UNIX/Linux sistemlerinde kullanılan dosya sistemleri birbirlerine yapı bakımından oldukça benzemektedir. Bu aileye “i-node tabanlı dosya sistemi ailesi” denebilir. Örneğin Ext2-Ext3-Ext4 gibi sistemler i-node tabanlı ailenin birer üyesidir. (Yani örneğin FAT sistemleri nasıl birbirlerine benziyorsa fakat aralarında bazı farkılıklar da varsa UNIX/Linux sistemlerindeki i-node tabanlı dosya sistemleri de yapı olarak birbirlerine benzemektedir.)

UNIX/Linux sistemlerindeki i-node tabanlı dosya sistemlerinin disk organizasyonları temel olarak şöyledir:



Yukarıdaki organizasyonda “block” terimi ardışıl sektör topluluklarını ifade etmektedir. Diskin başında “boot blok” vardır. Bu blok 1024 byte (iki sektör) uzunluğundadır.

**Anahtar Notlar:** Linux sistemlerinde bir dosya sanki bir vloümmüş gibi sisteme gösterilebilmektedir. Bu tür aygıtlara “looback device” denilmektedir. Böylece i-node temelli dosya sistemleri üzerinde denemeler bu biçimde çok daha kolay yapılabilmektedir. Bunun sırasıyla şu adımlar izlenmelidir:

1) Öncelikle volümü temsil eden içi sıfırlarla dolu bir dosya oluşturmak gerekir. Bu işlem komut satırında dd komutuyla kolay biçimde yapılabilir. Örneğin:

dd if=/dev/zero of=mydisk.dat bs=512 count=2880

Bu komut if ile belirtilen dosyadan of ile belirtilen dosyaya bs \* count kadar byte’ı kopyalamaktadır. /dev/zero aygıt sürücüsü her okunduğunda sıfır değerini veren bir dosya gibi davranır. O halde bu komutla mydisk.dat dosyası oluşturulacak için 512 \* 2880 byte sıfır ile doldurulacaktır.

2) Daha sonra oluşturulan bu dosya “loopback device” olarak tanıtılır:

sudo losetup /dev/loop0 mydisk.dat

Artık /dev/llop0 aygıt sürücüsü mydisk.dat dosyasını kullanacak hale getiirlmiştir.

3) Oluşturduğumuz volümü mkfs ile formatlayabiliriz:

sudo mkfs -t ext2 /dev/loop0

4) Artık oluşturulan dosya sistemi mount edilebilir:

sudo mount -t ext2 /dev/loop0 /mnt/mydisk

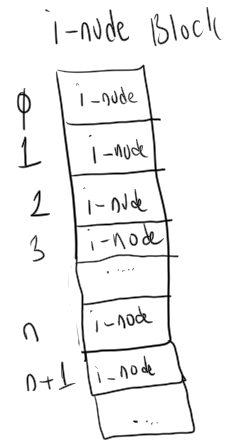
5) İşlemler bittikten sonra loopback device yok aşağıdaki gibi yok edilebilir:

sudo losetup -d /dev/loop0

Boot blokta -tıpkı FAT dosya sisteminde olduğu gibi- bir yükleyici program vardır. Bu program işletim yüklenmesine ön ayak olur. Süper blok da 2 1024 byte (yani 2 sektör) uzunluğundadır. Süper blokta ilgili i-node dosya sisteminin metadata bilgileri bulunmaktadır. Süper blok FAT dosya sistemindeki BPB bloğuna benzetilebilir. Pekiyi süper blokta nasıl bilgiler vardır? Örneğin Ext-2 dosya sisteminin dokümanlarında süper blok aşağıdaki elemanlara sahiptir:

|  |  |  |
| --- | --- | --- |
| **Offset (bytes)** | **Size (bytes)** | **Description** |
| 0 | 4 | [s\_inodes\_count](http://www.nongnu.org/ext2-doc/ext2.html#S-INODES-COUNT) |
| 4 | 4 | [s\_blocks\_count](http://www.nongnu.org/ext2-doc/ext2.html#S-BLOCKS-COUNT) |
| 8 | 4 | [s\_r\_blocks\_count](http://www.nongnu.org/ext2-doc/ext2.html#S-R-BLOCKS-COUNT) |
| 12 | 4 | [s\_free\_blocks\_count](http://www.nongnu.org/ext2-doc/ext2.html#S-FREE-BLOCKS-COUNT) |
| 16 | 4 | [s\_free\_inodes\_count](http://www.nongnu.org/ext2-doc/ext2.html#S-FREE-INODES-COUNT) |
| 20 | 4 | [s\_first\_data\_block](http://www.nongnu.org/ext2-doc/ext2.html#S-FIRST-DATA-BLOCK) |
| 24 | 4 | [s\_log\_block\_size](http://www.nongnu.org/ext2-doc/ext2.html#S-LOG-BLOCK-SIZE) |
| 28 | 4 | [s\_log\_frag\_size](http://www.nongnu.org/ext2-doc/ext2.html#S-LOG-FRAG-SIZE) |
| 32 | 4 | [s\_blocks\_per\_group](http://www.nongnu.org/ext2-doc/ext2.html#S-BLOCKS-PER-GROUP) |
| 36 | 4 | [s\_frags\_per\_group](http://www.nongnu.org/ext2-doc/ext2.html#S-FRAGS-PER-GROUP) |
| 40 | 4 | [s\_inodes\_per\_group](http://www.nongnu.org/ext2-doc/ext2.html#S-INODES-PER-GROUP) |
| 44 | 4 | [s\_mtime](http://www.nongnu.org/ext2-doc/ext2.html#S-MTIME) |
| 48 | 4 | [s\_wtime](http://www.nongnu.org/ext2-doc/ext2.html#S-WTIME) |
| 52 | 2 | [s\_mnt\_count](http://www.nongnu.org/ext2-doc/ext2.html#S-MNT-COUNT) |
| 54 | 2 | [s\_max\_mnt\_count](http://www.nongnu.org/ext2-doc/ext2.html#S-MAX-MNT-COUNT) |
| 56 | 2 | [s\_magic](http://www.nongnu.org/ext2-doc/ext2.html#S-MAGIC) |
| 58 | 2 | [s\_state](http://www.nongnu.org/ext2-doc/ext2.html#S-STATE) |
| 60 | 2 | [s\_errors](http://www.nongnu.org/ext2-doc/ext2.html#S-ERRORS) |
| 62 | 2 | [s\_minor\_rev\_level](http://www.nongnu.org/ext2-doc/ext2.html#S-MINOR-REV-LEVEL) |
| 64 | 4 | [s\_lastcheck](http://www.nongnu.org/ext2-doc/ext2.html#S-LASTCHECK) |
| 68 | 4 | [s\_checkinterval](http://www.nongnu.org/ext2-doc/ext2.html#S-CHECKINTERVAL) |
| 72 | 4 | [s\_creator\_os](http://www.nongnu.org/ext2-doc/ext2.html#S-CREATOR-OS) |
| 76 | 4 | [s\_rev\_level](http://www.nongnu.org/ext2-doc/ext2.html#S-REV-LEVEL) |
| 80 | 2 | [s\_def\_resuid](http://www.nongnu.org/ext2-doc/ext2.html#S-DEF-RESUID) |
| 82 | 2 | [s\_def\_resgid](http://www.nongnu.org/ext2-doc/ext2.html#S-DEF-RESGID) |
| -- EXT2\_DYNAMIC\_REV Specific -- | | |
| 84 | 4 | [s\_first\_ino](http://www.nongnu.org/ext2-doc/ext2.html#S-FIRST-INO) |
| 88 | 2 | [s\_inode\_size](http://www.nongnu.org/ext2-doc/ext2.html#S-INODE-SIZE) |
| 90 | 2 | [s\_block\_group\_nr](http://www.nongnu.org/ext2-doc/ext2.html#S-BLOCK-GROUP-NR) |
| 92 | 4 | [s\_feature\_compat](http://www.nongnu.org/ext2-doc/ext2.html#S-FEATURE-COMPAT) |
| 96 | 4 | [s\_feature\_incompat](http://www.nongnu.org/ext2-doc/ext2.html#S-FEATURE-INCOMPAT) |
| 100 | 4 | [s\_feature\_ro\_compat](http://www.nongnu.org/ext2-doc/ext2.html#S-FEATURE-RO-COMPAT) |
| 104 | 16 | [s\_uuid](http://www.nongnu.org/ext2-doc/ext2.html#S-UUID) |
| 120 | 16 | [s\_volume\_name](http://www.nongnu.org/ext2-doc/ext2.html#S-VOLUME-NAME) |
| 136 | 64 | [s\_last\_mounted](http://www.nongnu.org/ext2-doc/ext2.html#S-LAST-MOUNTED) |
| 200 | 4 | [s\_algo\_bitmap](http://www.nongnu.org/ext2-doc/ext2.html#S-ALGO-BITMAP) |
| -- Performance Hints -- | | |
| 204 | 1 | [s\_prealloc\_blocks](http://www.nongnu.org/ext2-doc/ext2.html#S-PREALLOC-BLOCKS) |
| 205 | 1 | [s\_prealloc\_dir\_blocks](http://www.nongnu.org/ext2-doc/ext2.html#S-PREALLOC-DIR-BLOCKS) |
| 206 | 2 | (alignment) |
| -- Journaling Support -- | | |
| 208 | 16 | [s\_journal\_uuid](http://www.nongnu.org/ext2-doc/ext2.html#S-JOURNAL-UUID) |
| 224 | 4 | [s\_journal\_inum](http://www.nongnu.org/ext2-doc/ext2.html#S-JOURNAL-INUM) |
| 228 | 4 | [s\_journal\_dev](http://www.nongnu.org/ext2-doc/ext2.html#S-JOURNAL-DEV) |
| 232 | 4 | [s\_last\_orphan](http://www.nongnu.org/ext2-doc/ext2.html#S-LAST-ORPHAN) |
| -- Directory Indexing Support -- | | |
| 236 | 4 x 4 | [s\_hash\_seed](http://www.nongnu.org/ext2-doc/ext2.html#S-HASH-SEED) |
| 252 | 1 | [s\_def\_hash\_version](http://www.nongnu.org/ext2-doc/ext2.html#S-DEF-HASH-VERSION) |
| 253 | 3 | padding - reserved for future expansion |
| -- Other options -- | | |
| 256 | 4 | [s\_default\_mount\_options](http://www.nongnu.org/ext2-doc/ext2.html#S-DEFAULT-MOUNT-OPTIONS) |
| 260 | 4 | [s\_first\_meta\_bg](http://www.nongnu.org/ext2-doc/ext2.html#S-FIRST-META-BG) |
| 264 | 760 | Unused - reserved for future revisions |

I-node Blok i-node elemanlarından oluşmaktadır. Yani i-node Blok, i-node elemanlarından oluşan bir dizi biçimindedir. I-Node bloktaki her i-node elemanının ilki 0 olmak üzere bir numarası vardır.



Bir dosyanın ismi dışındaki tüm meta data bilgileri i-node elemanında tutulmaktadır. Yani örneğin dosya ne zaman yaratılmıştır, uzunluğu nedir, erişim özellikleri nelerdir, hard link sayısı kaçtır, dosyanın parçaları data bölümünün hangi bloklarındadır (yani cluster’larındadır)? Aslında kursumuzda daha önce görmül olduğumuz UNIX/Linux sistemlerindeki stat fonksiyonu bilgileri bu i-node elemanından alıp bize vermektedir. stat fonksiyonunun parametrik yapısını yeniden anımsatmak istiyoruz:

#include <sys/stat.h>

int stat(const char \*path, struct stat \*buf);

struct stat {

dev\_t st\_dev; /\* ID of device containing file \*/

ino\_t st\_ino; /\* inode number \*/

mode\_t st\_mode; /\* protection \*/

nlink\_t st\_nlink; /\* number of hard links \*/

uid\_t st\_uid; /\* user ID of owner \*/

gid\_t st\_gid; /\* group ID of owner \*/

dev\_t st\_rdev; /\* device ID (if special file) \*/

off\_t st\_size; /\* total size, in bytes \*/

blksize\_t st\_blksize; /\* blocksize for filesystem I/O \*/

blkcnt\_t st\_blocks; /\* number of 512B blocks allocated \*/

time\_t st\_atime; /\* time of last access \*/

time\_t st\_mtime; /\* time of last modification \*/

time\_t st\_ctime; /\* time of last status change \*/

};

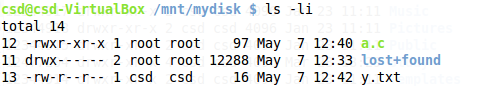
Tabii stat fonksiyonu i-node elemanındaki birçok değeri bize verse de -çok teknik olduğu gerekçesiyle- bazı bilgileri (örneğin dosya parçalarının hangi bloklarda olduğu gibi) de bilgileri bize vermemektedir. Ext-2 dokümanlarında i-node elemanın içeriği şöyle dokümante edilmiştir:

| **Offset (bytes)** | **Size (bytes)** | **Description** |
| --- | --- | --- |
| 0 | 2 | [i\_mode](http://www.nongnu.org/ext2-doc/ext2.html#I-MODE) |
| 2 | 2 | [i\_uid](http://www.nongnu.org/ext2-doc/ext2.html#I-UID) |
| 4 | 4 | [i\_size](http://www.nongnu.org/ext2-doc/ext2.html#I-SIZE) |
| 8 | 4 | [i\_atime](http://www.nongnu.org/ext2-doc/ext2.html#I-ATIME) |
| 12 | 4 | [i\_ctime](http://www.nongnu.org/ext2-doc/ext2.html#I-CTIME) |
| 16 | 4 | [i\_mtime](http://www.nongnu.org/ext2-doc/ext2.html#I-MTIME) |
| 20 | 4 | [i\_dtime](http://www.nongnu.org/ext2-doc/ext2.html#I-DTIME) |
| 24 | 2 | [i\_gid](http://www.nongnu.org/ext2-doc/ext2.html#I-GID) |
| 26 | 2 | [i\_links\_count](http://www.nongnu.org/ext2-doc/ext2.html#I-LINKS-COUNT) |
| 28 | 4 | [i\_blocks](http://www.nongnu.org/ext2-doc/ext2.html#I-BLOCKS) |
| 32 | 4 | [i\_flags](http://www.nongnu.org/ext2-doc/ext2.html#I-FLAGS) |
| 36 | 4 | [i\_osd1](http://www.nongnu.org/ext2-doc/ext2.html#I-OSD1) |
| 40 | 15 x 4 | [i\_block](http://www.nongnu.org/ext2-doc/ext2.html#I-BLOCK) |
| 100 | 4 | [i\_generation](http://www.nongnu.org/ext2-doc/ext2.html#I-GENERATION) |
| 104 | 4 | [i\_file\_acl](http://www.nongnu.org/ext2-doc/ext2.html#I-FILE-ACL) |
| 108 | 4 | [i\_dir\_acl](http://www.nongnu.org/ext2-doc/ext2.html#I-DIR-ACL) |
| 112 | 4 | [i\_faddr](http://www.nongnu.org/ext2-doc/ext2.html#I-FADDR) |
| 116 | 12 | [i\_osd2](http://www.nongnu.org/ext2-doc/ext2.html#I-OSD2) |

Görüldüğü gibi Ext-2’de bir i-node elemanı 128 byte uzunluğundadır.

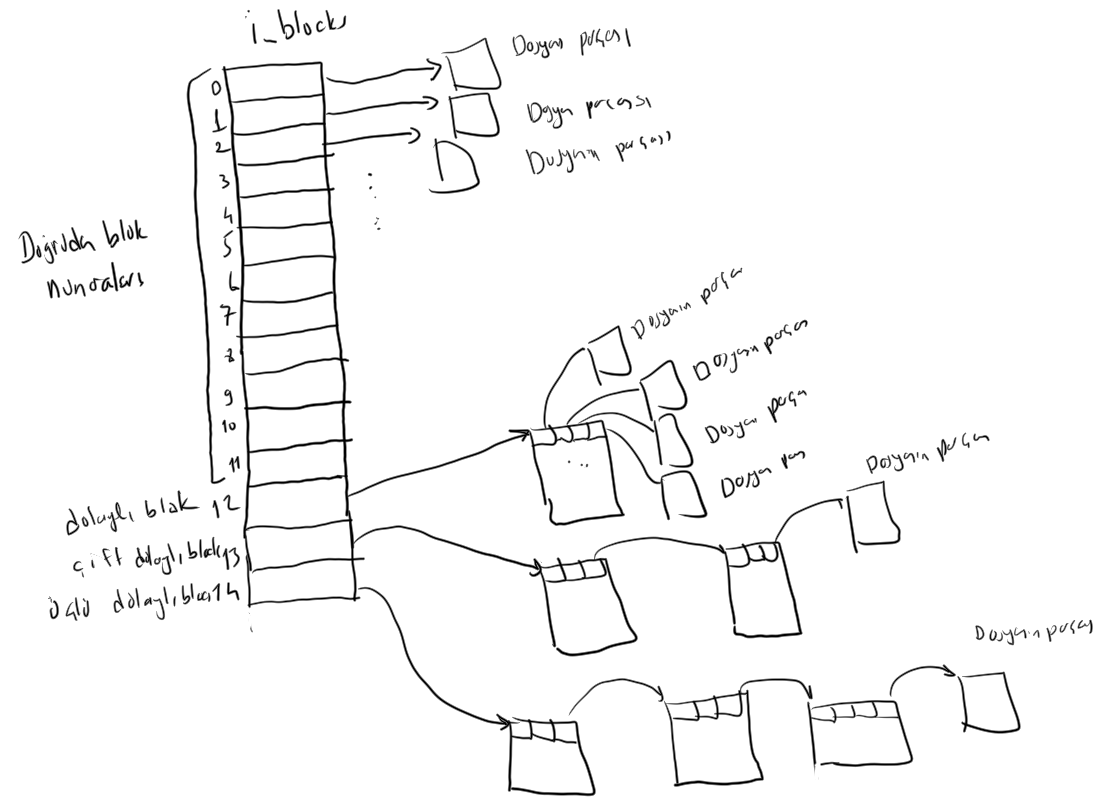
I-node tabanlı dosya sistemlerinde volümdeki toplam dosyaların sayısı i-node blok’ta ayrılmış i-node elemanlarının sayısından fazla olamaz. Çünkü her dosya için (dosyanın büyüklüğü ne olursa olsun) i-node blok’ta bir tane i-node elemanı tahsis edilmek zorundadır.

Bir dosayanın i-node numarası o dosyanın bilgilerinin i-node blok’taki hangi i-node elemanında bulunduğunu belirtir. ls komutunda –i seçeneği ile dosyların i-node numaraları elde edilebilir. Örneğin:



Data bölümünün ilk bloğu 0 numaralı bloktur. (UNIX/Linux sistemlerinde “cluster” yerine “block” teriminin kullanıldığını anımsayınız). Bir dosyanın parçalarının hangi bloklarda tutulduğu i-node elemanında nasıl saklanmaktadır? Örneğin dosya toplamda 150 blok yer kaplıyor olsun. i-node elemanında tek tek bu 150 bloğun hangi numaralı bloklar olduğu tutuluyor olabilir?

İşte i-node sistemide bunun için doğrudan (direct), dolaylı (indirect), çift dolaylı (double indirect) ve üçlü dolaylı (triple indirect) blok kavramları kullanılmaktadır. Örneğin Ext-2 dosya sisteminde bir i-node elemanının içerisinde 40’ıncı (decimal) offset’ten başlayan her biri 4 byte uzunlukta olan 15 tane blok numaralarrını tutan bir dizi vardır. (Bu dizi yukarıdaki Ext-2 dokümanlarından alınan kısımda i\_blocks ile temsil edilmektedir.) Bu 15 blok numaralarına ilşkin dizinin ilk 12 elemanı doğrudan blok numaralarını, 13’üncü elemanı dolaylı blok numarasını, 14’üncü elemanı çift dolaylı blok numarasını ve 15’inci elemanı üçlü dolaylı blok numarasını tutar.



Ext-2’de bir bloğun kaç sektörden oluştuğu bilgisi süper blok”ta belirtilmektedir. Süper bloğun s\_log\_block\_size elemanındaki değer 0 ise blok uzunluğu 1024 byte, 1 ise 2048 byte 2 ise 4096 byte, ... biçimindedir. Başka bir deyişle s\_log\_block\_size elemanındaki değer ile blok uzunluğu arasındaki ilişki şöyledir:

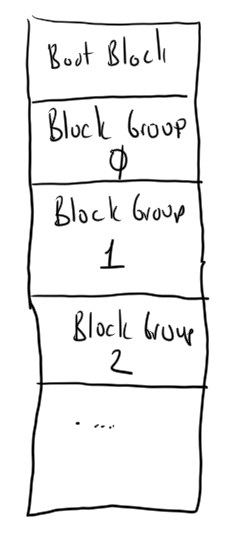
block size = 1024 << s\_log\_block\_size;

Bir bloğun 1024 byte (yani 2 sektör) uzunluğunda olduğunu varsayalım. Bu durumda eğer dosya 12 \* 1024 byte’tan küçük ya da ona eşitse dosyanın bloklarının numaraları doğrudan i-node elemanının i\_blocks dizisi içerisindeki ilk 12 elemandan alınır. Eğer dosyanın uzunluğu 12 \* 1024 (yani 12K)’dan büyükse dosyanın ilk 12K’sının blok numaraları doğurdan bloklarda geri kalanları dolaylı bloklarda bulunur. Dolaylı blok numarası i\_blocks dizisinin 13’üncü elemanındadır. Dizinin bu 13’üncü elemanında bir blok numarası vardır. Ancak o blok numarası dosyanın bir parçasını oluşturan bir bloğun numarası değildir. Dosyanın parçalarının hangi bloklarda tutulduğunu gösteren bloğun numarasıdır. Yani biz 13’üncü elemanda yazılan bloğa gittiğimizde o bloğun içerisinde 4 byte’lık blok numaraları görürüz. Bir bloğun 1024 byte olduğunu varsayarsak blok numaraları 4 byte yer kapladığına göre bir blok içerisinde 1024 /4 = 256 blok numarası bulunur. İşte eğer dosya 12K + 256K’dan daha büyükse artık dolaylı bloklar da yetersiz kalır. Bu durumda çift dolayı bloklar kullanılır. Çift dolaylı blok numarası i\_blocks dizisinin 14’üncü elemanındadır. 14’üncü elemanda belirtilen numaralı bloğa gittiğimizde onun içerisinde 256 tane blok numarası görürüz. O 256 blok numarası dosyanın parçalarını belirtmez. Bilakis dosyanın parçalarının hangi bloklarda olduğunu tutan blokları belirtir. Bu durumda çift dolaylı bir blok içerisinde (bir blok 1K uzunluktaysa) bir dosyanın 256 \* 256 \* 1024 (yaklaşık 67 MB) kadar byte’lık kısmı tutulmuş olur. Üçlü dolaylı bloklar da benzer biçimdedir.

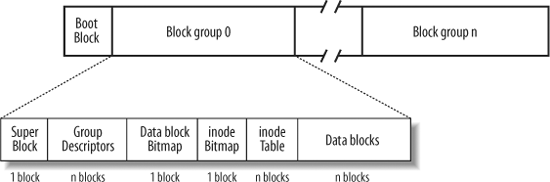
i-node tabanlı dosya sistemleri mimari olarak birbirlerine çok benzese de ayrıntılarda farklılıklar söz konusudur. Bu nedenle burada biz spesik olarak Ext-2 dosya sistemine özgü disk organizasyonu üzerinde de durmak istiyoruz.

**Ext-2 Dosya Sisteminin Disk Organizasyonu**

Ext-2 Dosya sistemi resmi olarak “Ext-2 File System Specification” dokümanında ayrıntılarıyla açıklanmaktadır. Bu dosya sisteminin disk organizasyonu kabaca aşağıdaki gibidir:



Görüldüğü gibi Aslında ext-2 disk organizasyonu blok grouplardan oluşmaktadır. Her blok grubunun kendi i-node tablosu ve data bölümü vardır. Ayrıca süper blok ve “block group descriptor” denilen veri yapılarının backup amaçlı birer kopyası da blok gruplarının başında tutulmaktadır (Ext-2 Revision 1’de bu backup kopyalar her blok grubunda değil yalnızca belirli blok gruplarında tutuluyor). Bir blok grubunun formatı şöyledir:



Tabii blok gruplardaki i-node tablolarındaki i-node elemanlarına yine ardışıl numaralar verilmiştir. Yani örneğin ilk blok grubunun son i-node elemanı n numaralı ise ikinci blok grubunun ilk i-node elemanı n + 1 numaralı elemandır.

**Anahtar Notlar:** Ext-2 dosya sistemindeki meta data alanlarında belirtilen blok numaraları her zaman volümün başından itibaren ve süper blokta belirtilen blok uzunluğu dikkate alınarak verilmiş olan blok numaralarıdır.

“Blok Group Decriptor” tablosu “block decsriptor”larından oluşur. Yani “block group descriptor” tablosu tüm blok gruplarının bilgilerini içeren “block group descriptor” dizisi biçimindedir. “Block group descriptor” tablosu içerisindeki bir “block group decsriptor”ın formatı şöyledir:

| **Offset (bytes)** | **Size (bytes)** | **Description** |
| --- | --- | --- |
| 0 | 4 | [bg\_block\_bitmap](http://www.nongnu.org/ext2-doc/ext2.html#BG-BLOCK-BITMAP) |
| 4 | 4 | [bg\_inode\_bitmap](http://www.nongnu.org/ext2-doc/ext2.html#BG-INODE-BITMAP) |
| 8 | 4 | [bg\_inode\_table](http://www.nongnu.org/ext2-doc/ext2.html#BG-INODE-TABLE) |
| 12 | 2 | [bg\_free\_blocks\_count](http://www.nongnu.org/ext2-doc/ext2.html#BG-FREE-BLOCKS-COUNT) |
| 14 | 2 | [bg\_free\_inodes\_count](http://www.nongnu.org/ext2-doc/ext2.html#BG-FREE-INODES-COUNT) |
| 16 | 2 | [bg\_used\_dirs\_count](http://www.nongnu.org/ext2-doc/ext2.html#BG-USED-DIRS-COUNT) |
| 18 | 2 | [bg\_pad](http://www.nongnu.org/ext2-doc/ext2.html#BG-PAD) |
| 20 | 12 | [bg\_reserved](http://www.nongnu.org/ext2-doc/ext2.html#BG-RESERVED) |

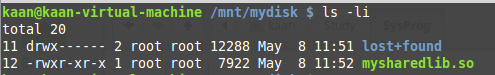
Her blok gruptaki i-node tablosunun yeri vs. “block group descriptor” tablosunun ilgili elemanlarında tutulmaktadır.

Pekiyi bir i-node numarasına ilişkin i-node elemanın hangi blok grubunda olduğu nasıl tespit edilmektedir? İşte süper blok içerisinde blok gruplarına ilişkin bazı bilgiler de vardır. Örneğin bir blok grubunda kaç tane blok bulunduğu süper bloğun s\_blocks\_per\_count elemanında saklanır. Böylece blok gruplarının volümün neresinden başladığı ve hangi uzunlukta olduğu tespit edilebilmektedir. Ayrıca her blok grubunda kaç tane i-node elemanın bulunduğu da yine süper bloğun s\_inodes\_per\_group elemanında saklanmıştır. Bu durumda biz n numaralı bir i-node elemanın hangi grupta olduğunu şöyle tespit edebiliriz:

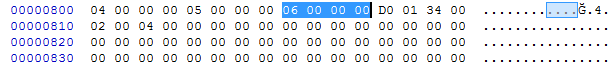
n / s\_inodes\_per\_group

Ext-2 dosya sisteminde ilk i-node elemanının numarası 1’dir. Yani i-node elemanları 1’den başlayarak numaralandırılmaktadır. 1 numaralı i-node elemanı tamamen 0’larla doldurulmuştur (null eleman). 2 numaralı i-node elemanı kök dizine ilişkindir. (n numaralı i-node elemanının i-node tablosunun n – 1’inci elemanında olacağına dikkat ediniz.)

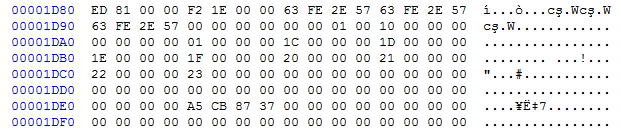
Örneğin loopback aygıt olarak mount edilmiş bir ext-2 dosya sisteminin kök dizininde aşağıdaki dosyalar bulunuyor olsun:



Burada mysharedlib.so dosyasının i-node numarası 12’dir. Bu dosya ilk blok grubundadır. “Block group descriptor” tablosunun içeriği şöyledir:

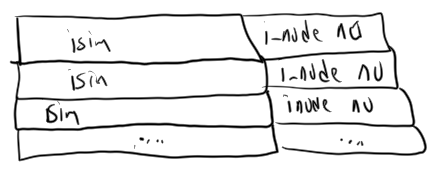


Burada volümün bir tek blok grubundan oluştuğunu görüyorsunuz. Seçilmiş olan alan (8’inci offsetten çekilen DWORD) blok grubunun i-node tablosunun yerini göstermektedir. Bu volümde 1 blok 1024 byte olduğu için i-node tablosunun yeri 6 \* 1024 (0x1800) offset’indedir. 12 numaralı i-node elemanı da i-node tablosunun başından 11 \* 128 byte ileride olacaktır: 6 \* 1024 + 11 \* 128 (0x1D80). Bu i-node elemanını aşağıda görmektesiniz:



**I-Node Tabanlı Dosya Sistemlerinde Dizin Organizasyonu**

I-node tabanlı dosya sistemlerinde dizinler de tamamen dosyalar gibidir. Yani onlar için de bir i-node elemanı vardır. Ancak dizinlerin blokları içerisinde o dizinde bulunan dosyaların isimleri ve i-node numaraları bulunur. Aslında i-node tabanlı dosya sistemleri arasında dizin içeriğinin formatı bakımından farklılıklar vardır. Ancak burada biz bu formatı genel olarak aşağıdaki biçimdeymiş gibi gösterebiliriz:



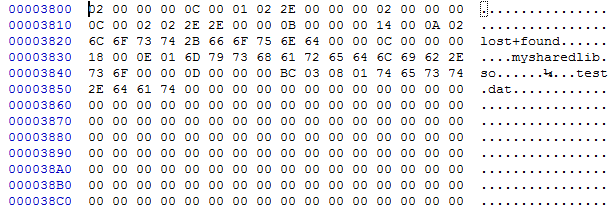
Ext-2 dosya sisteminde dizin elemanlarının formatı şöyledir:

| **Offset (bytes)** | **Size (bytes)** | **Description** |
| --- | --- | --- |
| 0 | 4 | [inode](http://www.nongnu.org/ext2-doc/ext2.html#IFDIR-INODE) |
| 4 | 2 | [rec\_len](http://www.nongnu.org/ext2-doc/ext2.html#IFDIR-REC-LEN) |
| 6 | 1 | [name\_len](http://www.nongnu.org/ext2-doc/ext2.html#IFDIR-NAME-LEN)[[a]](http://www.nongnu.org/ext2-doc/ext2.html#FTN.AEN1829) |
| 7 | 1 | [file\_type](http://www.nongnu.org/ext2-doc/ext2.html#IFDIR-FILE-TYPE)[[b]](http://www.nongnu.org/ext2-doc/ext2.html#FTN.AEN1838) |
| 8 | 0-255 | [name](http://www.nongnu.org/ext2-doc/ext2.html#IFDIR-NAME) |

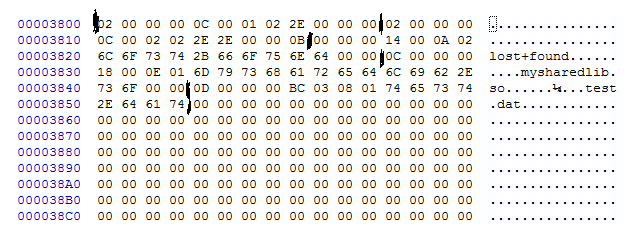
Görüldüğü gibi Ext-2’de dizin içerisindeki dosya listesi sabit uzunluklu kayıtlar biçiminde değildir. Ext-2’de dosya isimleri 256 karaktere kadar uzayabildiği için her kayıdın bu kadar büyük olmasına gerek görülmemiştir. Çünkü dosya isimlerinin çoğu aslında kısadır. Tablodaki inode isimli eleman ilgili dosyanın i-node numarasını tutar. rec\_len ilgili dizin girişinin kaç byte uzunlukta olduğunu belirtir. Yani sonraki giriş brada belirtilen değer kadar ileridedir. name\_len değişken olan dosya isminin kaç karakter uzunluğunda olduğunu tutar. file\_type dosyanın türünü belirtir. Gerçi bu tür bilgisi aynı zamanda i-node elemanın içinde de vardır fakat “yol ifadesinin çözümlenmesi (path name resolution)” işleminin hızlı bir biçimde yapılabilmesi için bu bilgi ayrıca Ext-2’de dizin girişinde de bulundurulmaktadır.

Pekiyi i-node temelli bir dosya sisteminde bir yol ifadesi nasıl çözümlenir? İşte kök dizini i-node elemanın yeri bellidir. Örneğin Ext-2 dosya sisteminde 2 numaralı i-node elemanı (anımsanacağı gibi Ext-2’de i-node numaraları 1’den başlar ve ilk i-node elemanı 0’larla doludur) kök dizine ilişkindir. Sonra kök dizin içerisinde diğer yol bileşeni, onun içerisinde de diğeri bulunarak süreç ilerletilir.

Aşağıda örnek bir dizin içeriği (kök dizin) görüyorsunuz:

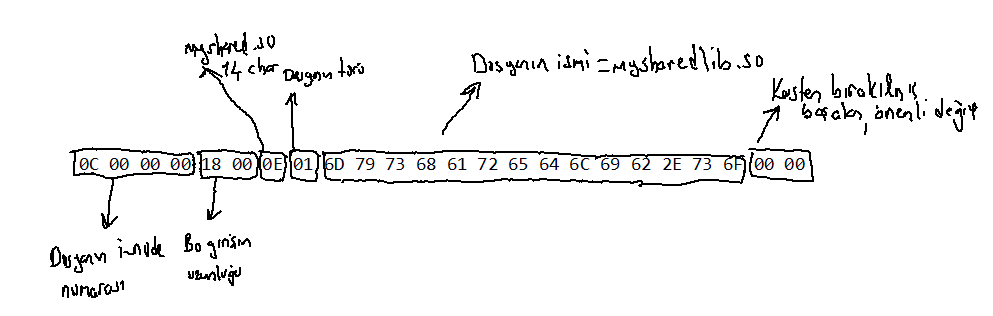


Burada kayıtların yerleri aşağıdaki gibidir:



Burada örneğin mysharedlib.so dosyasının dizin girişi şöyledir:

0C 00 00 00 18 00 0E 01 6D 79 73 68 61 72 65 64 6C 69 62 2E 73 6F 00 00

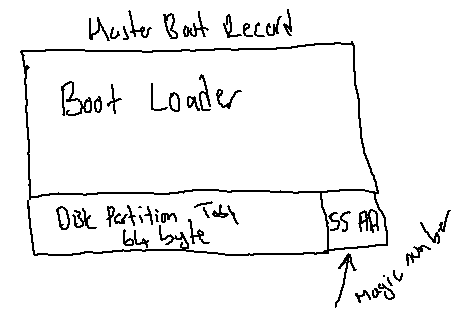


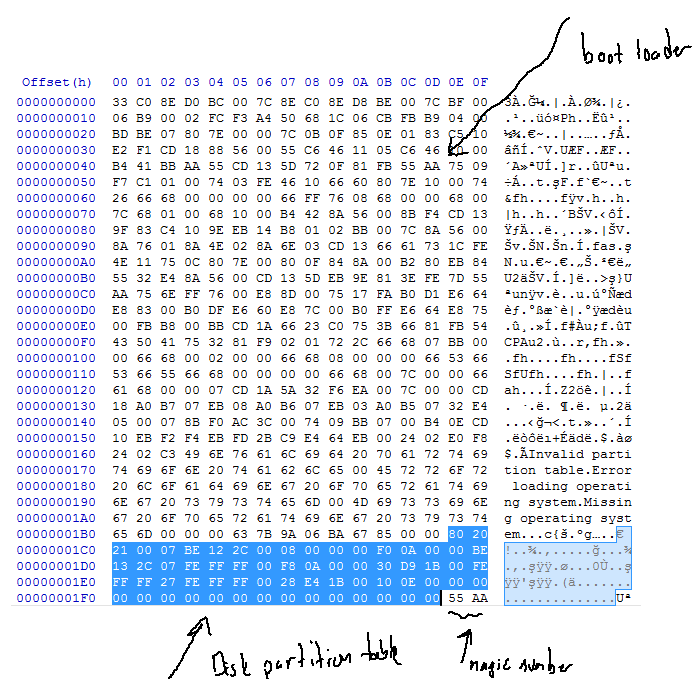
Dizin girişi sıralı olarak taranmaktadır. Dizin listesinin bittiği sonraki kaydın i-node elemanının 0 olmasından anlaşılır. Anımsanacağı gibi Ext-2 dosya sisteminde 0 numaralı i-node elemanı kullanılmamaktadır. i-node elemanları 1’den başlar.

**Diskin Bölümlere Arılması ve Disk Bölümleme Tablosu**

Bir hard diske, SSD’ye ya da flash belleğe birden fazla dosya sistemi yerleştirilebilir. Örneğin biz bir diske hem Windows’u hem Linux’u kurabiliriz. Windows NTFS sistemini kullanıyorken, Linux Ext-2 sistemini kullanıyor olabilir. İşte bunu sağlamak için disk bölümesi kavramı uydurulmuştur. Diskin başında bir disk bölümleme tablosu (disk partition table) bulunur. Burada hangi disk bölümünün hangi sektörden başlayıp hangi uzunlukta olduğu bilgisi yazılıdır. Böylece işletim sistemleri kendi bölümleri dışına çıkmaz. Yani adeta bir disk daha küçük parçalara ayrılmış birden fazla diskmiş gibi kullanılmaktadır.

Disk bölümleme tablosu nerededir? Diskin ilk sektörüne (yani 0’ıncı sektörüne) “Master Boot Record” denilmektedir. Bu ilk sektörün yapısı şöyledir:





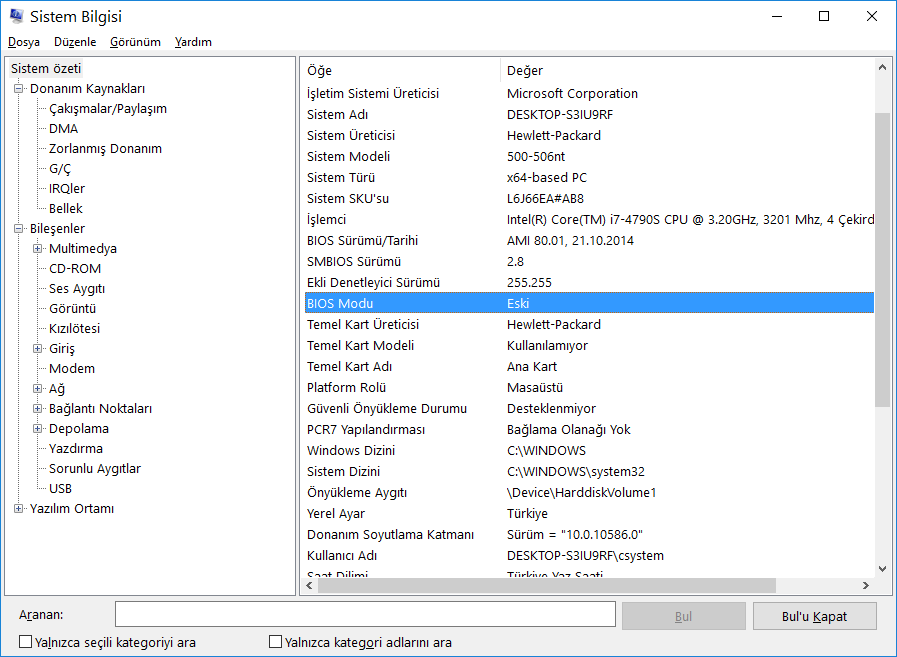
Disk Bölümleme Tablosu her biri 16 byte olan 4 girişe sahiptir. Bu durumda diskte en fazla 4 ana disk bölümü bulunabilmektedir. Daha fazla disk bölümleme için “uzatılmış disk bölümü (extended partition)” kavramı uydurulmuştur. 16 byte’lık disk bölümleme tablosu elemanının formatı şöyledir:

|  |  |  |
| --- | --- | --- |
| Structure of a *16-byte* Partition Table Entry | | |
| **Relative Offsets (*within entry*)** | **Length (*bytes*)** | **Contents** |
| **0** | **1** | **Boot Indicator (80h = *active*)** |
| **1 - 3** | **3** | **Starting CHS values** |
| **4** | **1** | ***Partition-type* Descriptor** |
| **5 - 7** | **3** | **Ending CHS values** |
| **8 - 11** | **4** | **Starting Sector** |
| **12 - 15** | **4** | **Partition Size (in sectors)** |

**Table 4.**

İlk byte disk bölümünün aktif olmadığını belirtir. Burada 0x80 değeri varsa o disk bölümü aktiftir, 0x00 varsa aktif değildir. Dört girişten yalnızca bir tanesi aktif olabilir. Dördüncü offset’te o disk bölümünde (partition type descriptor) hangi dosya sisteminin bulunduğu bilgisi vardır. Görüldüğü gibi her disk bölümü için o disk bölümünün hangi sektörden başladığı ve ne uzunlukta olduğu disk bölümleme tablosu elemanında belirtilmektedir. Tablodaki diğer elemanların artık önemi kalmamıştır.

Disk bölümleme tablosu UEFI Bios kullanan sistemlerde “GUID Partition Table” ismiyle modernize edilmiştir. Sistemimizde UEFI BIOS olduğunu nasıl anlarız? Windows’ta “msinfo32” programı çalıştırılıp “System Summary (Sistem Özeti)” kısmındaki “BIOS Modu”na bakılabilir:



Linux’ta sistemin UEFI ile boot edilip edilmediğini anlamak için “/sys/firmware/efi” dosyasının var olup olmadığna bakılır. Bilgisayarımız UEFI’yi desteklediği halde boot işlemi klasik BIOS’tan yapılıyor olabilir. Bunun için “CMOS Setup”a girişte UEFI desteğini açmak gerekebilir.

MBR sektörünün başında küçük bir “boot loader” programı bulunur. Bu program disk bölümleme tablosunun dört bölümünden hangisinin aktif olduğunu tespit eder. O disk bölümünnün ilk sektörünü belleğe yükleyerek oraya jump eder. Her disk bölümünün ilk sektörü kendi işletim sistemini yüklemekle yükümlüdür. (Anımsanacağı gibi FAT dosya sisteminde ilk sektöre zaten “boot sector” deniliyordu)

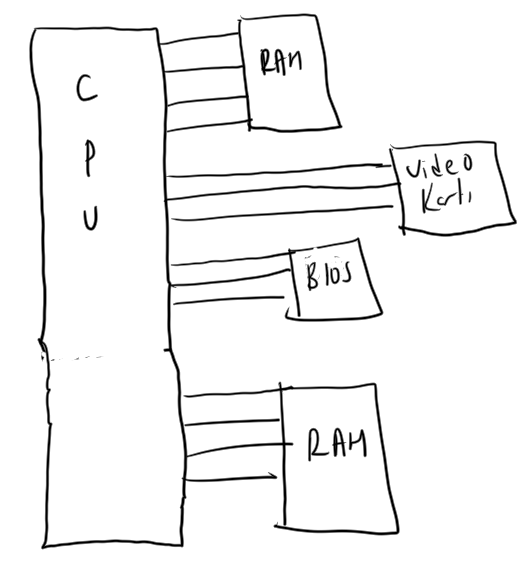
Biz burada disk bölümleme tablosundaki genişletilmiş (extended) disk bölümlerinin nasıl oluşturulduğu ve GUID disk bölümleme tablosunun yapısını ele almayacağız. Bu konu “Sistem Programalama ve İleri C Uygulamaları II” numaralı kursun konusu içerisindedir.

**Intel Tabanlı Bilgisayarların Boot Süreci**

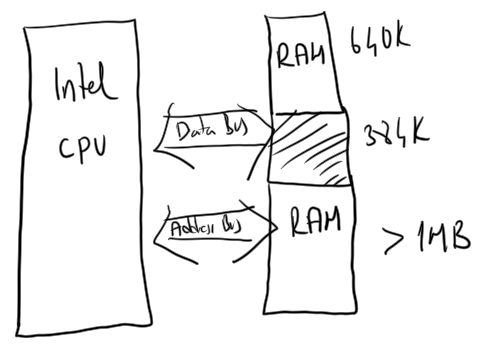
Bugün kullandığımız Intel tabanlı masaüstü ve dizüstü bilgisayarlarımızı açtığımızda işletim sisteminin otomatik olarak yüklendiğini görüyoruz. Pekiyi bu süreç nasıl gerçekleşmektedir? İşte çok ayrıntılarına girmeden PC’lerimizdeki tipik boot süreci şöyledir:

1) Her işlemci reset edildiğinde belli bir adresten çalışmaya başlayacak biçimde tasarlanmıştır. Bu adrese “reset vektörü” denilmektedir. İşlemci RAM’e bağlı olduğuna göre ve bilgisayarımızı açtığımızda RAM’de sıfırlandığına göre işlemci hangi komutlarla çalışmaya başlayacaktır? İşte bilgisayar sistemlerinde CPU’nun reset vektörüne bilgisayarı kapattığımızda da bilgileri tutan tarzda bir bellek yerleştirilmektedir. Eskiden bu tür bellekler EPROM olarak üretiliyordu. Bugün artık bu tür bellekler için EEPROM teknolojisi kullanılmaktadır.

Biz PC’mizi reset ettiğimizde çalışma EEPROM içerisindeki bir yerden başlar. PC açıldığında hazır bulunan ve kalıcı belekte bulunan bu kodlara PC terminolojisinde “BIOS (Basic Input Output System)” denilmektedir. Aslında bir bilgisayar sisteminde RAM sürekli olmak zorunda değildir. RAM’in bazı bölgeleri başka aygıtların içerisindeki belleklere yönlendirilmiş olabilir (Memory Mapped IO).Bu yönlendirme eletriksel düzeyde kolay bir biçimde yapılabilmektedir. Örneğin:



Tabii bizim sistem programcısı olarak adres alanının hangi kısımlarının RAM yerine bağlı olduğunu bilmemiz gerekebilir. Örneğin PC mimarisinde ilk 1MB belleğin son 384K’sı başka aygıtlara yönlendirilmiştir. (Örneğin bu nedenle DOS işletim sistemi 1024K – 384K = 640K kullanabiliyordu). Yani PC RAM’i aslında sürekli değildir:

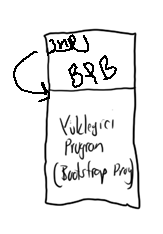


İşte PC sisteminde klasik BIOS ilk 1MB’nin son 64K’sında bulunur. Bugünkü 32 bit ve 64 bit Intel işlemcileri reset edildiğinde çalışma reset vektöründen başlar. Oradan BIOS’a jump edilmektedir ve BIOS’taki boot kodu çalışmaktadır. BIOS’taki boot kodunun ilk yaptığı şey POST (Power On Self Test) işlemidir. Bu işlem sırasında sisteme hangi aygıtların bağlı olduğu tespit edilmektedir.

2) BIOS’taki POST kodundan sonra sistemin boot edilmesi için gereken kodlar çalıştırılmaktadır. BIOS’taki “bootstrap” kod “CMOS Setup”ta belirdeğimiz sıraya göre aygıtlara bakar. Örneğin C sürücüsünün (yani birinci hard diskin) ilk boot aygıtı olarak seçildiğini düşünelim.

3) BIOS’taki bootstrap kod boot aygıtının (örneğin C sürücüsünün) ilk sektörünü belleğe yükler (7C00 adresine) ve oradaki koda jump eder. Böylece diskin MBR sekötüründeki “yükleyici program” çalıştırılmış olur.

4) MBR’deki yükleyici program aktif disk bölümünü tespit ve onun ilk sektörünü belleğe yükleyip oraya jump eder. Böylece ilgili disk bölümünün ilk sektöründeki program çalışmış olur. Her dosya sisteminde ilk sektörler boot işlemi için ayrılmıştır. Örneğin FAT dosya sisteminde ilk sektöre boot sektör denilmektedir ve boot sektörün organizasyonu şöyledir:



Burada BPB’nin başındaki jmp komutu FAT dosya sistemindekiş yükleyici programa atlamaktadır. Benzer biçimde Ext-2 dosya sisteminde de boot blokta işletim sistemini yükleyen bir kod vardır. Görüldüğü gibi işletim sistemini asıl yükleyen kod ilgili disk bölümünün ilk sektöründeki yükleyici koddur.

Boot işleminin özeti şöyledir: Makina açıldığında akış BIOS’tan başlar. BIOS diskin ilk sektörünü (MBR) belleğe okuyarak oraya jump eder. Buradaki program aktif disk bölümünü tespit ederek onun ilk sektörünü belleğe okur ve oraya jump eder. İşletim sistemini de oradaki kod yükler.

**Örnek Çalışma: Bilgisayar Açıldığında Bizim Yerleştirdiğimiz Bir Programın Çalıştırılması**

Bunu sağlamanın en pratik yolu sembolik makine dilinde bir program yazıp onu hard diskin, flash belleğin ya da CD’nin ilk sektörüne yerleştirmektir. Ve makineyi o sürücüden boot edilecek biçimde “CMOS Setup”tan ayarlamaktır. Ekrana Intel gerçek modda merhaba dünya yazısını çıkartan bir nasm programı şöyle yazılabilir:

[BITS 16]

mov ax, 07C0h

mov ds, ax

mov si, message

REPEAT:

mov al, [si]

test al, al

jz EXIT

mov ah, 0x0e

mov bl, 7

int 0x10

inc si

jmp REPEAT

EXIT:

jmp $

message db "this is a test", 0

times 510 - ($ - $$) db 0

dw 0xAA55

Derleme işlemi şöyle yapılabilir:

nasm -f bin -o boot.bin boot.asm

Elde edilen boot.bin dosyası flash belleğe “Win32DiskImager” gibi bir programla yazılıp makine boot edilebilir.

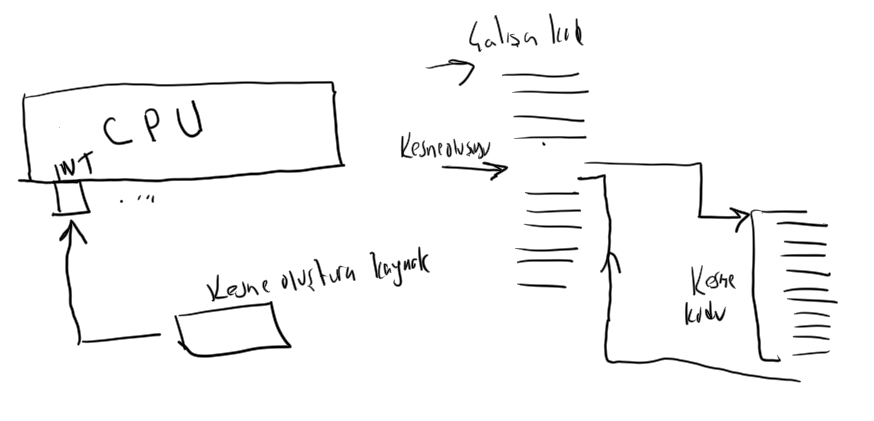
**Kesmeler (Interrupts)**

Bir kodun çalışmasına ara verilip başka bir kodun çalıştırılması ve bu bitince önceki kodun kaldığı yerden çalışmaya devam etmesi sürecine “kesme (interrupt)” denilmektedir. “Interrupt” sözcüğü İngilizce “araya girme (özellikle konuşmalarda vs.) anlamına gelmektedir”.

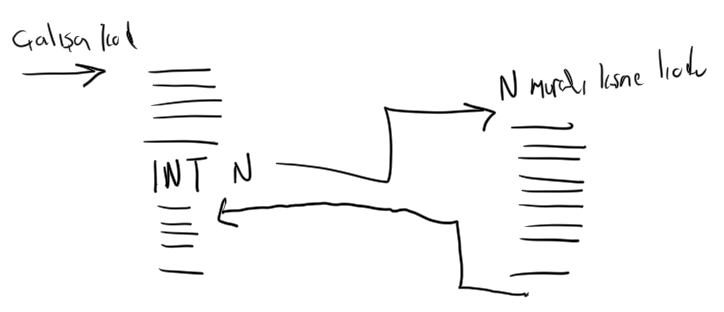
Kesme işlemciye özgü bir kavramdır. İşlemcilerin hemen hepsinde (mikrodenetleyiciler de dahil olmak üzere) çeşitli biçimlerde kesme kavramları vardır. Kesmeler fonksiyon çağrılarına (yani CALL işlemine) benziyor olsa da aslında çeşitli bakımlardan ondan farklıdır.

Kesmeler oluş biçimine göre üçe ayrılmaktadır:

1) Donanım kesmeleri (hardware interrupts): Kesme dendiğinde default olarak donanım kesmeleri anlaşılır. Neredeyse hemen her işlemcide ve mikrodenetleyicide donanım kesmeleri vardır. Donanım kesmeleri işlemcinin özel bir pini (bu pine genellikle INT pini denir) dışarıdan elektriksel olarak uyarılarak oluşturulmaktadır. (Yani donanım kesmeleri asenkron olarak dışarıdan başka bir donanım birimi tarafından işlemcinin kesme uyarılarak oluştuurlur):



2) Yazılım Kesmeleri (Software Interrupts): Yazılım kesmeleri programcı tarafından makine koduyla koda dahil edilerek oluşturulan kesmelerdir. Örneğin Intel işlemcilerinde INT makine komutu yazılım kesmesi oluşturmak için kullanılmaktadır. Yazılım kesmelerinde kesmeyi oluşturan kaynak bizzat programcının kendisidir. Zaten yazılım kesmelerinin mekanizma olarak fonksiyon çağrılarından (CALL işleminden) çok büyük farkları yoktur. Pek çok işletim sisteminde sistem fonksiyonları CALL makine komutu yerine yazılım kesmeleriyle (örneğin Intel’de INT makine komutuyla) çağrılmaktadır.



3) İçsel kesmeler (internal interrupts): Bu kesmeler bizzat işlemcinin kendisi tarafından oluşturulmaktadır. Intel terminolojisinde bunlara “fault” ya da “exception” da denir. Sayfalama mekznizması ve koruma mekanizması hep bu tür içsel kesmelerle yönetilmektedir. Örneğin işlemci yetkisi olmayan kod bir bellek alanına erişlmeye çalıştığında “page fault” denilen bir içsel kesme oluşturur. Bu içsel kesme sonucunda işletim sisteminin kesme kodu çalışır. O kod da prosesi sonlandırır. Ya da örneğin biz Windows gibi Linux gibi sistemlerde bir göstericiye rastgele bir adres yerleştirip onun gösterdiği yere erişmek istersek bu biçimde bir içsel kesme oluşacaktır. Yani içsel kesmeler işlemcinin bir makine komutunu çalıştırırken bazı uygunsuzluklar yüzünden kendisinin oluşturduğu kesmelerdir.

Kesmenin cinsi ne olursa olsun bir kesme oluştuğunda araya girilip çalıştırılan koda “kesme kodu (interrupt handler)” denilmektedir. Pek çok işlemcide kesmelerin numaraları vardır. Yani numaralar sayesinde çok sayıda kesme varmış gibi bir etki oluşturulabilmektedir. Örneğin Intel işlemcilerinde toplam 256 kesme vardır. Zaten örneğin INT makine komutu kesme numarasını argüman olarak alır. Örneğin:

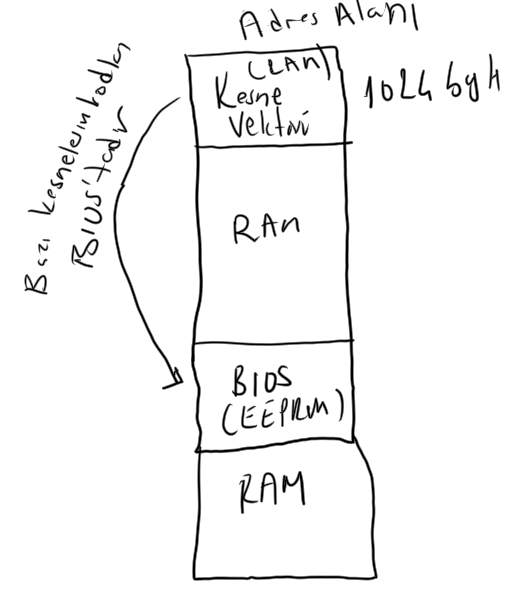
....

int 0x10 ; 10h numaralı kesme yazaılımsal olarak çağrılıyor

....

Geleneksel olarak kesme numaraları Intel’de 16’lık sistemde belirtilmektedir.

Pekiyi bir kesme oluştuğunda kimin kodu çalıştırılmaktadır? İşte Intel işlemcilerinde bir kesme oluştuğunda işlemci hangi kodu çalıştıracağına RAM’de “kesme vektörü (interrupt vector)” denilen bir yere bakarak karar verir. Kesme vektörü gerçek modda belleğin tepesindeki ilk 1024 byte’dır. Her kesme kodunun segment:offset değerleri (adresleri) burada yazılıdır. İşlemci n mumarlı bir kesme oluştuğunda vektörün n numaralı elemanına bakarak kesme koduna dallanır. Tersten gidersek Intel işlemcilerinde gerçek modda n numaralı bir kesme oluştuğunda kendi kodumuzun alıştıırlabilmesini sağlamak için kodumuzun adresini kesme vektörünün n’inci elemanına yazmamız gerekir. Kesme vektörü yukarıda da belirtildiği gibi RAM’in tepesindeki ilk 1024 byte’tır. Buraya bir değer yazmak gerek C’de gerekse sembolik makine dilinde çok kolaydır.



Burada özellikle bir kesme oluştuğunda sistem programcısının kesme vektörüyle oymayarak kendi kendi kodunun çalışmasını sağlayabildiğine dikkat ediniz. Kullandığımız PC’ler reset edildiğinde kesme vekötürü de sıfırlanmış durumdadır. Çalışma BIOS’tab naşlar buradaki kod kesme vektörünü de doldurmaktadır. Pek çok kesmenin kodu (örneğin 10h, 13h, 16h gibi) BIOS’tadır. Dolayısıyla BIOS’taki başlangıç kodu kesme vektörünü bu kodları gösterecek biçimde düzenlemektedir.

Bir kesmenin orifinal kodunu vektörden alıp saklayabiliriz. Sonra vektörü kendi kodumuzu gösterecek biçimde değiştirebiliriz. Kendi kodumuz içerisinde de orijinal kesme kodunu çağırabiliriz. Bu sürece kesmenin kancalanması (hook edilmesi) denilmektedir. DOS zamanındaki pek çok virüs 8h gibi 13h gibi BIOS kesmelerini kancalıyor ve bulaşmayı böyle sağlıyordu.

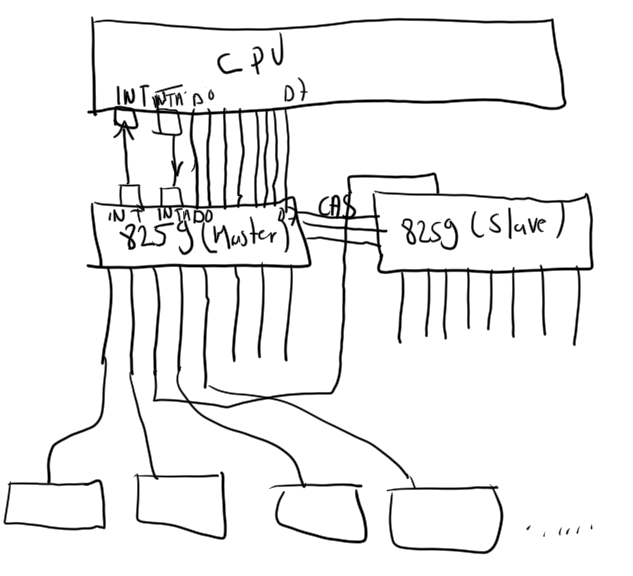
Intel işlemcileri reset edildiğinde “gerçek mod (real mode)” denilen bir modda başlar. Gerçek mod işlemciin 16 bit 8086 gibi (yani DOS’un kullanıldığı işlemci) çalıştığı moddur. Intel işlemcilerinin “korumalı mod (protected mode)” denilen diğer bir çalışma modu daha vardır. Örneğin Windows, Linux korumalı modda çalışmaktadır. Bu işletim sistemleri yüklenirken işlemci gerçek moddan korumalı moda geçirilmektedir.

Korumalı modda kesme vektörünün formatı ve yeri değişmektedir. Korumalı modda artık kesme vektörüne “kesme betimlici tablosu (interrupt descriptor table)” denir. Kesme betimleyici tablosu belleğin herhangi bir yerinde oluşturulabilmektedir. İşlemci kesme betimleyici tablosunu “IDTR (Interrupt Descriptor Table Register)” yazmacının gösterdiği yerde aramaktadır. İşletim sistemlerinin yükleyicileri korumalı moda geçmeden önce kesme betimleyici tablosunu oluşturur. IDTR yazmacının burayı göstermesini sağlar.

Pekiyi korumalı modda Windows ya da Linux’ta biz bir kesme oluştuğunda kendi kodumuzun çalışmasını sağlayabilir miyiz? Bunu doğrudan yapamayız. Çünkü Windows ve Linux sistemlerinde kesme vektörüne ya da kesme betimleyici tablosuna “user mode”tan erişeyemeyiz. Bunun için aygıt sürücü oluşturmak gerekir. Tabii aygıt sürücülerin belli mimarileri vardır. Biz aygıt sürücünün içerisinde kesme vektörüne ya da kesme betimleyici tablosuna yine doğrudan erişmeyiz. İşletim sisteminin bizim için uygun gördüğü bir biçimde kesme işlemleri için devreye girebiliriz. İşletim sistemleri kesmelerden pek çok aygıt sürücü faydalanabilsin diye genellikle bir zincir oluşturma yöntemi izlemektedir.

**PC’ler Donanım Kesmeleri**

PC’lerde işlemcinin kesme ucuna (INT ucuna) doğrudan donanım kaynakları bağlanmamıştır. Çünkü birden fazla kaynağın elektriksel olarak bu ucu uyarması sorunlara yol açabilmektedir. İşte bunu düzene sokmak için ilk PC’lerde (IBM PC ve uyumlu makineler) Intel’in 8259 Kesme Denetleyicisi (Programmable Interrupt Controller) kullanılmıştır. 8259 kesme denetleyicisinin 8 giriş ucu vardır. Bu uçlara donanım birimleri bağlanabilir. Bu birimler bu uçları elektriksel olarak uyardığında kesme denetleyicisi de CPU’yunun INT ucunu uyarmaktadır. AT’lerle (PC’lerin 80286 işlemcisiyle yapılmış biçimi) birlikte kesme denetleyicisi sayısı ikiye çıkartılmıştır. Intel’in 8259 kesme denetleyicisinin birbirlerine bağlanabilmesi için birinci kesme denetleyicisinin (master) bir giriş ucunun ikinci kesme denetleyicine (slave) bağlanması gerekmektedir. İşte IBM birinci kesme denetleyicisinin iki numaralı ucunu bu amaçla kullanmıştır. Bugün hala kullandığımız PC ve notebook’lardaki kesme sistemi bu mimariye dayanmaktadır. 8259 bir chip set’in içerisine bizim görmeyeceğimiz biçimde yerleştirilmiş durumdadır. Bugünki PC’lerde kesme sistemi aşağıdaki şekilde görüldüğü gibidir:



Tipik kesme oluşturma protokolü de şöyledir:

1) Donanım birimleri kesme denetleyicisinin bir ucunu (elektriksel olarak) uyarır.

2) Kesme denetleyicisi CPU’nun INT ucunu uyarır.

3) CPU bunu kabul ederse bunu INTA ucunu aktive ederek kesme denetleyicisine bildirir.

4) Kesme denetleyicisi kesmenin kabul edildiğini INTA ucundan anlar kesme dumarasını D0-D7 uçlarından CPU’ya bildirir.

5) CPU o anda çalıştırdığı koda ara verir. Kesme vektörüne ya da kesme betimleyici tablosuna bakarak kesme koduna dallanır.

CPU INT ucu uyarıldığında donanım kesmesini kabul etmek zorunda değildir. CPU donanım kesmelerine FLAGS yazmacının IF (Interrupt flag) set edilerek kaapatılabilir. Bu işlem CLI (Clear Interrupt Flag) makine komutuyla tek hamlede de yapılabilmektedir. CPU’yu yeniden donanım kesmelerine açmak için ise STI (Set Interrupt Flag) makine komutu kullanılabilmektedir.

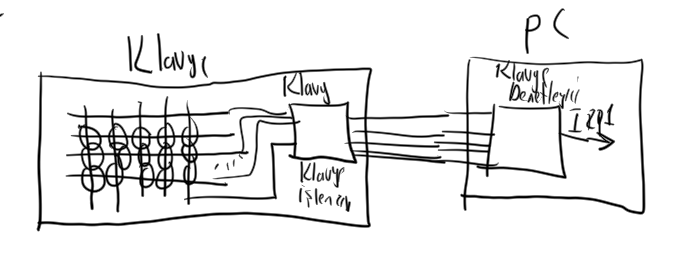
Bugünkü PC mimarisinde kesme denetleyicisinin giriş uçlarından her birine IRQ denilmektedir. Birinci kesme denetleyicinin ilk ucu IRQ0 olmak üzere her IRQ’ya bir numara verilmiştir. IRQ2 hattı kaskat bağlantı için kullanıldığından bu hatta bir donanım birimi bağlı değildir.

Şimdi IRQ hatlarına bağlı donanım birimlerini tek tek inceleyelim:

**IRQ0:** Bu hatta Intel’in 8254 zamanlayıcısı (Programmable Interval Timer) bağldır. Bu denetleyici programlanarak saniyede belli sayıda pals üretmesi sağlanabilmektedir. İşte her palse kesme denetleyicisinin IRQ0 hattını aktive dere. Böylece saniyede belli bir periyotta donanım kesmesi oluşturulmuş olur. BIOS bu denetleyiciyi saniyede 18.2 kere darbe üretecek biçimde programlar.

Windows gibi Linux işletim sistemlerinin thread’ler arası geçiş (context switch) mekanizması tamamen IRQ0 şle yapılmaktadır. Bu işletim sistemleri artık makine hızlıysa bu denetleyiciyi saniyede 1000 kere darbe üretecek biçimde kurmaktadır (yani 1 milisaniye periyoda). Böylece işletim sistemi her timer kesmesi geldiğinde bazı ayarlamalar yapar. Bu kesmeler belli bir sayıya gelince (örneğin 60 tanede bir. Yani 60 nilisaniyede) thread’lerarası geçiş uygular. 8254 timer kesmesi kapatılırsa sistem hemen çökebilir. İşte bu nedenle CLI gibi makine komutları koruma mekaznizması altında yalnızca kernel moddaki kodlar tarafından kullanılabilmektedir.

**IRQ1:** Bu hatta Intel’in 8042 (ya da benzeri) klavye denetleyici birimi bağlıdır. Klavye denetleyicisi de klavyenin içerisindeki klavye işlemcisi ile (keyboard encoder) bağlantı halindedir. Bu durum şekilsel olarak şöyle gösterilebilir:



Klavye deveresi bir çeşit matris gibidir. Tellerin kesişim noktasında tuşlar vardır. Böylece elektrisksel olarak hangi tuşa basıldığı anlaşılabilmektedir. Bu matrisin uçları klavye içerisindeki kalvye işlemcisine (keyboard encoder) giridi olarak verilir. Klavye işlemcisi hangi tuşa baıslmış olduğunu tespit eder ve bunu klavye kablosu yoluyla bilgisayar tarafındaki kalvye denetleyicisine iletir. Klavye denetleyicisi de IRQ1 hattından kesme oluşturur. İşletim sisteminin kesme kodu (ya da default olarak BIOPS’taki kod) devreye girer. Klavye denetleyicisinden basılan tuşu alır.

Klavye denetleyicisi basılan tuşun numarasını tespit etmektedir. Tuşların üzerindeki sembollerin konuyla bir ilgisi yoktur. Klavye üzerindeki her tuşa bir numara verilmiştir. Buna ilgili tuşun “tarama kodu (scan code)” denilmektedir. Klavye içerisindeki işlemci de klavye denetleyicisine bu tarama kodunu göndermektedir. Kesme kodu bu tarama kodunu aldığında tuşun hangi tuş olduğunu anlar. İşletim sistemleri bölgesel ayarlara bakarak o tuşun aslında hangi karakterle (klavye üzerinde gördüğümüz sembol) ilişkili olduğunu tespit eder. Sanki kalvyeden o tuşa basılmış gibi bir etki yaratır. Yani klavyedeki karakterlerin yer değiştirmesi (klavyenin transpoze edilmesi) tamamen yzılım youluyla yapılmaktadır. (Örneğin Türkçe ve İngilizce klavyenin devreleri arasında bir fark yoktur.

Klavyede yalnızca tuşa basıldığında klavye denetleyicisine bildirmde bulunulmaz aynı zamanda el tuştan çekildiğinde de bildirimde bulunulmaktadır. Klavyenin tuşuna basıldığında gönderilen koda “make code”, el tuştan çekildiğinde gönderilen koda ise “break code” denilmektedir. Bilgisayar tarafındaki klavye denetleyicisi hem make code için hem de break code için kesme oluşturmaktadır. Örneğin önce A tuşuna basmış olalım sonra parmağımızı çekmeden B tuşuna basalım sonra paramağımızı önce A tuşundan sonmra B tuşundan çekelim. Şu kodlar oluşturulacaktır:

- A tuşu için make code

- B tuşu için make code

- A tuşu için break code

- B tuşu için break code

Bugün kullandığımız PC klavyelerinde klavye içerisindeki işlemci 1 byte make code’u seri olarak bilgisayar tarafına iletmektedir. El tuştan çekildiğinde klavye içerisindeki işlemci bu sefer önce bir F0 byte’ını sonra da make code’un aynısını (toplam 2 byte) break code olarak gönderir.

Klavye üzerindeki ışıklı tuşların ışıkları klavye devresi tarafından yakılıp söndürülür ancak bu işlem otomatik yapılmamaktadır. Biz örneğin NUM-LOCK tuşuna bastığımızda bu tuşa basıldığı bilgisayar tarafıne iletilir. Oluşan kesmenin kodu bu tuşun NUM-LOCK tuşu olduğunu anlar bu kez klavye içerisindeki işlemciyi programlayarak bu ışığın yakılmasını sağlar.

Klavyedeki tuşlara yönelik önemli bir durumda “typematic” sürecidir. Bir tuşa basıp parmağımızı o tuşta beklettiğimizde sürekli o tuşa basılıyormuş gibi oluşan işleme “typematic” denilmekedir. Typemetic periyodu ve ilk basıştan ne kadar zaman sonra typematic’e başlanacağı klavye içerisindeki işlemci tarafından belirlenir. Bu işlemci bilgisayar tarafındaki klavye denetleyici yoluyla (tabii dolayısıyla yazılım yoluyla) programlanabilmektedir.

**IRQ2:** Bu hat kaskat bağlantığından dışsal bir donanım birimine bağlı değildir.

**IRQ3 ve IRQ4:** Bu hatlara 8250 ya da 16550 UART işlemcisi bağlıdır.