**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

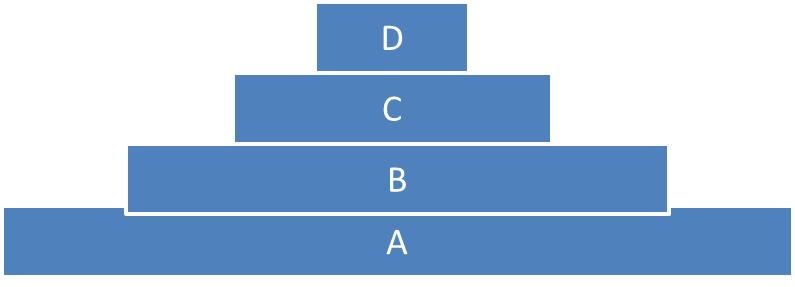
- XENIX

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, const char \*delim)

{

static char \*pos;

char \*beg;

if (str != NULL)

pos = str;

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

++pos;

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

return NULL;

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>

#define MAX\_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 programı 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>

#define LINE\_LEN 4096

int issharp(const char \*str)

{

while (isspace(\*str))

++str;

return \*str == '#';

}

int main(int argc, 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.

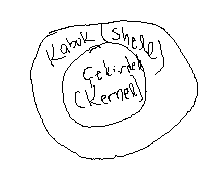
Programı çalıştırabilmek için yalnızca ismi yazılmaz. Onun dizini de belirtilmelidir. Tipik çalıştırma şöyle yapılır:

./sample

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

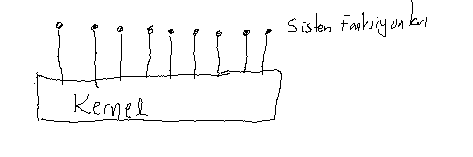
**İşletim Sisteminin Sistem Fonksiyonları, POSIX Fonksiyonları ve Standart C Fonksiyonları**

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.

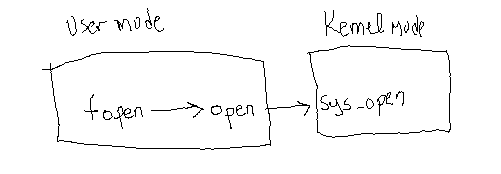
İşletim sistemlerinin çekirdeklerinde binlerce fonksiyon bulunur. Bunların küçük bir kısmı dışarıdan da önemli bazı işleri yapmak için çağrılabilmektedir. Bunlara sistem fonksiyonları (system call) denilmektedir. Her işletim sisteminin sistem fonksiyonlarının isimleri, parametrik yapıları farklıdır. 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 byte2lık 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 önişlemsi 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(LPCSTR lpszMsg, int status)

{

DWORD dwLastError = GetLastError();

LPTSTR lpszErr;

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

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

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

LocalFree(lpszErr);

}

exit(status);

}

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(LPCSTR lpszMsg, int status)

{

DWORD dwLastError = GetLastError();

LPTSTR lpszErr;

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

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

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

LocalFree(lpszErr);

}

exit(status);

}

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(LPCSTR lpszMsg, int status)

{

DWORD dwLastError = GetLastError();

LPTSTR lpszErr;

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

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

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

LocalFree(lpszErr);

}

exit(status);

}

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(LPCSTR lpszMsg, int status)

{

DWORD dwLastError = GetLastError();

LPTSTR lpszErr;

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

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

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

LocalFree(lpszErr);

}

exit(status);

}

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)