**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

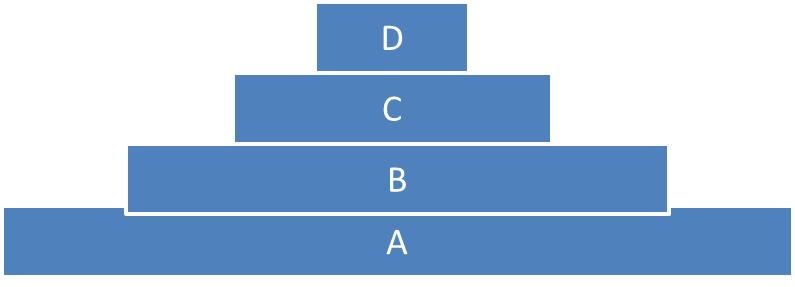
- XENIX

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.

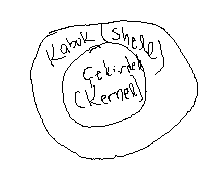
Programı çalıştırabilmek için yalnızca ismi yazılmaz. Onun dizini de belirtilmelidir. Tipik çalıştırma şöyle yapılır:

./sample

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

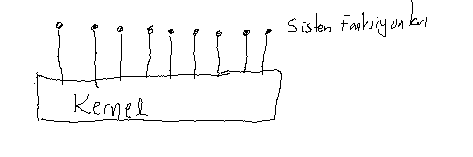
**İşletim Sisteminin Sistem Fonksiyonları, POSIX Fonksiyonları ve Standart C Fonksiyonları**

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.

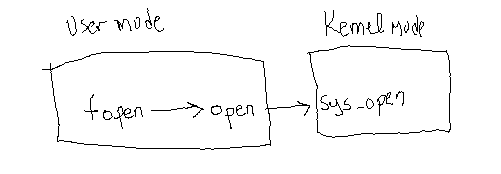
İşletim sistemlerinin çekirdeklerinde binlerce fonksiyon bulunur. Bunların küçük bir kısmı dışarıdan da önemli bazı işleri yapmak için çağrılabilmektedir. Bunlara sistem fonksiyonları (system call) denilmektedir. Her işletim sisteminin sistem fonksiyonlarının isimleri, parametrik yapıları farklıdır.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

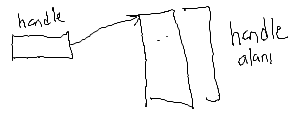
Hata kontrolü bakımından fonksiyonları 2 gruba ayırabiliriz:

1) Her zaman hata kontrolünün yapılması gereken fonksiyonlar: Bunlar sistemin o anki durumuyla ilgili biçimde baaşarısz 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 kontrolü 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>

typedef int DATATYPE;

/\* Type Declarations \*/

typedef struct tagMATRIX {

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\_t rowSize, size\_t colSize)

{

HMATRIX hMatrix;

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

return NULL;

hMatrix->rowSize = rowSize;

hMatrix->colSize = colSize;

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

free(hMatrix);

return NULL;

}

return hMatrix;

}

void SetMatrix(HMATRIX hMatrix, DATATYPE \*vals)

{

size\_t i;

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

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

}

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

{

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

}

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

{

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

}

void DispMatrix(HMATRIX hMatrix)

{

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(HMATRIX hMatrix)

{

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\_

#define MATRIX\_H\_

typedef int DATATYPE;

typedef void \*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 \*/

typedef struct tagMATRIX {

size\_t rowSize;

size\_t colSize;

DATATYPE \*pMatrix;

} MATRIX;

/\* Function Definitions \*/

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

{

MATRIX \*matrix;

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

return NULL;

matrix->rowSize = rowSize;

matrix->colSize = colSize;

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

free(matrix);

return NULL;

}

return matrix;

}

void SetMatrix(HMATRIX hMatrix, 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(HMATRIX hMatrix, size\_t row, size\_t col)

{

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

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

}

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

{

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

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

}

void DispMatrix(HMATRIX hMatrix)

{

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(HMATRIX hMatrix)

{

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 tekniğin deki 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 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(int n)

{

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>

unsigned long factorial(unsigned n)

{

unsigned long 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>

unsigned long factorial(unsigned n)

{

unsigned long result;

if (n == 0)

return 1;

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

return result;

}

int main(void)

{

long result;

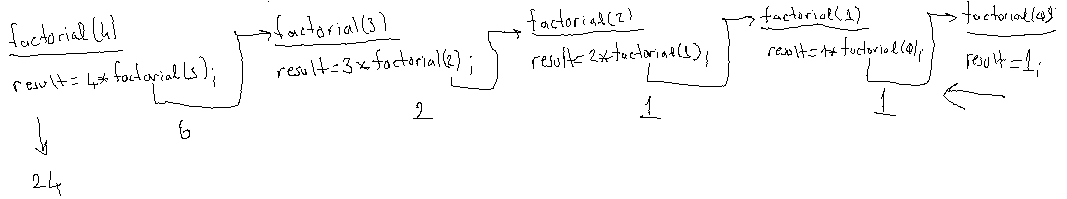
result = factorial(4);

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

return 0;

}

Bu fonksiyon şöyle çalışmaktadır:



Fonksiyon daha basit şöyle de yazılabilirdi:

unsigned long factorial(unsigned n)

{

if (n == 0)

return 1;

return n \* factorial(n - 1);

}

**2) Yazının Yersten 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(const char \*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(const char \*str)

{

const char \*temp = str;

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

++str;

do {

--str;

putchar(\*str);

} while (str != temp);

}

Fonksiyonun özyinelemeli gerçekleştirimi de şöyle olabilir:

#include <stdio.h>

void putsrev(const char \*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, int left, int right)

{

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(unsigned n)

{

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(int n)

{

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(int n)

{

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(int row, int col, char ch)

{

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\_t size)

{

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>

#define SIZE 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(int row, int col)

{

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(int row, int col)

{

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(int a)

{

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(double a)

{

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

return 0;

}

double (\*bar(long a))(double)

{

return foo;

}

double(\*(\*tar(int a))(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, int size, 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(int period, 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 \*/

#define MAX\_CMD\_LINE 1024

#define MAX\_PARAMS 32

/\* Type Declarations \*/

typedef struct tagCMD{

const char \*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];

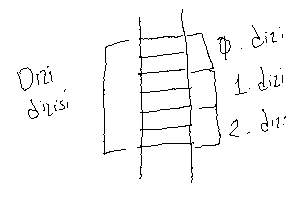
Böyle bir çok boyutlu dizinin başlangıç adresini tutabilecek gösterici şöyle tanımlanabilir:

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 paraztezle 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 başvurmaktadır. Bu çevre değişkenleri yoksa Windows yine bu sürücülerin kök diziblerinden 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 akarakter dahil yol ifadesinin uzunluğundan küçükse fonksiyon başarısız olur ve NULL adresler 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(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);

}

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, başarıszlık 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 propmt'ta belirtilen çalışma dizininden itibaren yer belirtir. Örneğin:

#include <stdio.h>

#include <string.h>

#include <Windows.h>

/\* Symbolic Constants \*/

#define MAX\_CMD\_LINE 1024

#define MAX\_PARAMS 32

/\* Type Declarations \*/

typedef struct tagCMD{

const char \*cmdText;

void(\*proc)(void);

} CMD;

/\* Function Prototypes \*/

void print\_syserr(const char \*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(const char \*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 Çalışma Dizini Nerededir?**

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(const char \*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(const char \*msg);

void WalkDir(const char \*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(const char \*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(const char \*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(const char \*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>

#define TABSIZE 4

void PutSpace(int count);

void WalkDirRecur(const char \*path, int tabCount)

{

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(const char \*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(int count)

{

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(const char \*path, BOOL(\*Proc)(const WIN32\_FIND\_DATA \*, int), int level)

{

HANDLE hFF;

WIN32\_FIND\_DATA finfo;

BOOL bResult;

if (!SetCurrentDirectory(path))

return;

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

if (hFF == INVALID\_HANDLE\_VALUE)

return TRUE;

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(const char \*path, BOOL(\*Proc)(const WIN32\_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(const WIN32\_FIND\_DATA \*fd, int level)

{

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

return FALSE;

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

return TRUE;

}

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(const char \*path)

{

DIR \*dir;

struct dirent \*dire;

struct stat 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>

#define FALSE 0

#define TRUE 1

typedef int bool\_t;

bool\_t walkdir\_recur(const char \*path, bool\_t(\*proc)(const char \*, struct stat \*, int), int level)

{

DIR \*dir;

struct dirent \*dire;

struct stat finfo;

bool\_t result;

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

return FALSE;

if (chdir(path) < 0) {

closedir(dir);

return FALSE;

}

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(const char \*path, bool\_t(\*proc)(const char \*, struct stat \*, int))

{

char cwd[4096];

bool\_t result;

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

return FALSE;

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

if (chdir(cwd) < 0)

return FALSE;

return result;

}

bool\_t disp(const char \*path, struct stat \*finfo, int level)

{

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

return TRUE;

}

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\_t width)

{

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\_t count, size\_t width, int (\*cmp)(const void \*, const void \*))

{

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(const void \*pv1, const void \*pv2)

{

const int \*pi1 = (const int \*)pv1;

const int \*pi2 = (const int \*)pv2;

if (\*pi1 > \*pi2)

return 1;

if (\*pi1 < \*pi2)

return -1;

return 0;

}

typedef struct tagPERSON {

char name[32];

int no;

} PERSON;

int comparer2(const void \*pv1, const void \*pv2)

{

const PERSON \*p1 = (const PERSON \*)pv1;

const PERSON \*p2 = (const PERSON \*)pv2;

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

}

int comparer3(const void \*pv1, const void \*pv2)

{

const PERSON \*p1 = (const PERSON \*)pv1;

const PERSON \*p2 = (const PERSON \*)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\_t count, size\_t width, int (\*cmp)(const void \*, const void \*));

Örneğin:

#include <stdio.h>

#include <stdlib.h>

int comparer1(const void \*pv1, const void \*pv2)

{

const int \*pi1 = (const int \*)pv1;

const int \*pi2 = (const int \*)pv2;

if (\*pi1 > \*pi2)

return 1;

if (\*pi1 < \*pi2)

return -1;

return 0;

}

typedef struct tagPERSON {

char name[32];

int no;

} PERSON;

int comparer2(const void \*pv1, const void \*pv2)

{

const PERSON \*p1 = (const PERSON \*)pv1;

const PERSON \*p2 = (const PERSON \*)pv2;

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

}

int comparer3(const void \*pv1, const void \*pv2)

{

const PERSON \*p1 = (const PERSON \*)pv1;

const PERSON \*p2 = (const PERSON \*)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>

#define BLOCK\_SIZE 10

void ExitSys(LPCSTR lpszMsg, int status);

void Disp(const WIN32\_FIND\_DATA \*files, size\_t size);

int CompareBySize(const void \*pv1, const void \*pv2);

int CompareByName(const void \*pv1, const void \*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(const void \*pv1, const void \*pv2)

{

const WIN32\_FIND\_DATA \*f1 = (const WIN32\_FIND\_DATA \*)pv1;

const WIN32\_FIND\_DATA \*f2 = (const WIN32\_FIND\_DATA \*)pv2;

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

return 1;

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

return -1;

return 0;

}

int CompareByName(const void \*pv1, const void \*pv2)

{

const WIN32\_FIND\_DATA \*f1 = (const WIN32\_FIND\_DATA \*)pv1;

const WIN32\_FIND\_DATA \*f2 = (const WIN32\_FIND\_DATA \*)pv2;

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

}

void Disp(const WIN32\_FIND\_DATA \*files, size\_t size)

{

size\_t i;

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

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

}

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);

}

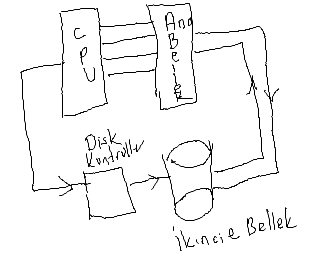
exit(status);

}

**Bellek Sistemleri**

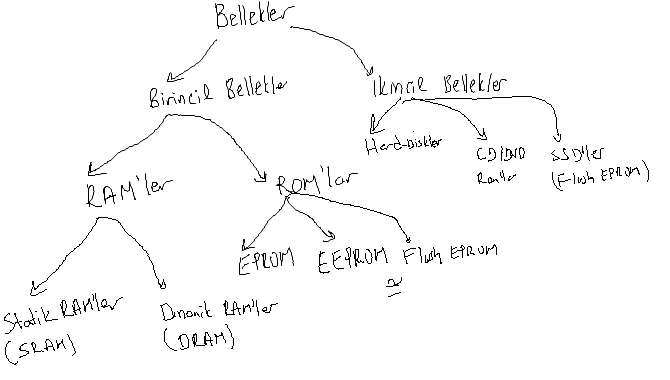
Bilgileri tutmakta kullanılan birimlere bellek (memory) denir. Bilgisayar sistemlerindeki bellekler "Birincil Bellekler (Primary Memory)" ve "İkincil Bellekler "Secondary Memory" olmak üzere ikiye ayrılmaktadır. Birincil belleklere "ana bellekler" ya da halk arasında "RAM" de denilmektedir. Birincil 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 kaafalarını 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. Bınlar 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 saniyer 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'teb 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? Tanıt: 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 ssistem 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 fetgc 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. Byrada 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 dosyay 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 tamponlamalaı 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 dahil 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.