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

**(Eğitmen: Kaan ASLAN)**

**Sistem Programlama Nedir?**

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

- İşletim Sistemleri

- Derleyiciler ve yorumlayıcılar

- Editörler

- Debug Programları

- Virüs ve Antivirüs yazılımları

- Haberleşme programları

- Gömülü sistem programları

- Aygıtların programlanması, aygıt sürücüler

- Veritabanı motorları

- Sanallaştırma yazılımları

- Oyun motorları

- ...

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

**UNIX Türevi İşletim Sistemlerinin Tarihsel Gelişimi**

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

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

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

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

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

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

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

Özgür yazılım akımının temel prensipleri şunlardır:

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

- Programın çalıştırılabilir biçimi istenildiği gibi kopyalanıp dağıtılabilir.

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

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

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

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

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

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

- Understand (Proprietary)

- Source Insight (Proprietary)

- SourceNav (Open Source)

**Masaüstü ve Mobil İşletim Sistemleri**

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

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

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

**Orijinal Kod Temeline Sahip İşletim Sistemleri**

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

- AT&T UNIX

- DOS

- Windows

- Linux

- BSD (belli bir yıldan sonra)

- Solaris

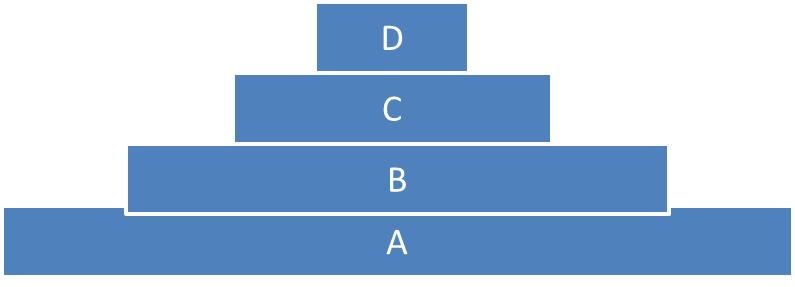
- XENIX

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

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

**Programlamadaki Katmanlı Yapılar**

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



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

**API (Aplication Programming Interface) Kavramı**

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

**Library ve Framework Kavramları**

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

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

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

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

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

**Bazı Çok Bilinmeyen Standart C Fonksiyonları**

Bu bölümde kursta kullanılabilecek bazı yaygın C fonksiyonları ele alınacaktır.

**strtok Fonksiyonu**

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

"ankara,adana, izmir, kars"

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

"10/12/2007"

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

strtok fonksiyonunun prototipi şöyledir:

#include <string.h>

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

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

#include <stdio.h>

#include <string.h>

int main(void)

{

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

char \*str;

str = strtok(text, ",");

while (str != NULL) {

puts(str);

str = strtok(NULL, ",");

}

return 0;

}

Tabi while yerine for döngüsü de kullanılabilirdi:

#include <stdio.h>

#include <string.h>

int main(void)

{

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

char \*str;

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

puts(str);

return 0;

}

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

#include <stdio.h>

#include <string.h>

#include <ctype.h>

char \*mystrtok(char \*str, 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 programını interaktif olmayan modda çalıştırarak parametresiyle belirtilen kabuk komutunun kabuk tarafından çalıştırılmasını sağlar. Fonksiyonun protipi şöyledir:

#include <stdlib.h>

int system(const char \*string);

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

Örneğin:

#include <stdio.h>

#include <stdlib.h>

int main(void)

{

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

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

exit(EXIT\_FAILURE);

}

printf("Ok\n");

return 0;

}

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

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

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

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

**tmpfile Fonksiyonu**

Fonksiyonun prototipi şöyledir:

#include <stdio.h>

FILE \*tmpfile(void);

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

Örneğin:

#include <stdio.h>

#include <stdlib.h>

int main(void)

{

FILE \*f;

int i, val;

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

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

exit(EXIT\_FAILURE);

}

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

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

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

exit(EXIT\_FAILURE);

}

fseek(f, 0, SEEK\_SET);

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

printf("%d ", val);

if (ferror(f)) {

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

exit(EXIT\_FAILURE);

}

printf("\n");

fclose(f);

return 0;

}

**tmpnam Fonksiyonu**

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

#include <stdio.h>

char \*tmpnam(char \*s);

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

Aşağıdaki örnekte bir dosyadaki #'li staırlar silinmektedir:

#include <stdio.h>

#include <stdlib.h>

#include <ctype.h>

#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.

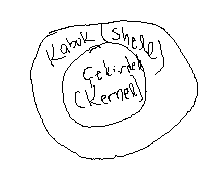
Programı çalıştırabilmek için yalnızca ismi yazılmaz. Onun dizini de belirtilmelidir. Tipik çalıştırma şöyle yapılır:

./sample

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

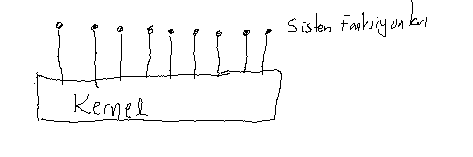
**İşletim Sisteminin Sistem Fonksiyonları, POSIX Fonksiyonları ve Standart C Fonksiyonları**

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.

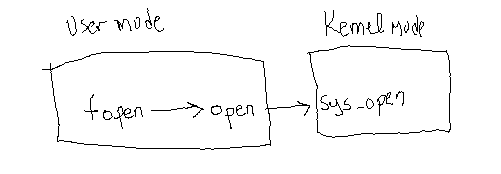
İşletim sistemlerinin çekirdeklerinde binlerce fonksiyon bulunur. Bunların küçük bir kısmı dışarıdan da önemli bazı işleri yapmak için çağrılabilmektedir. Bunlara sistem fonksiyonları (system call) denilmektedir. Her işletim sisteminin sistem fonksiyonlarının isimleri, parametrik yapıları farklıdır.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

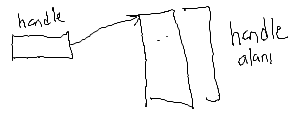
Hata kontrolü bakımından fonksiyonları 2 gruba ayırabiliriz:

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;

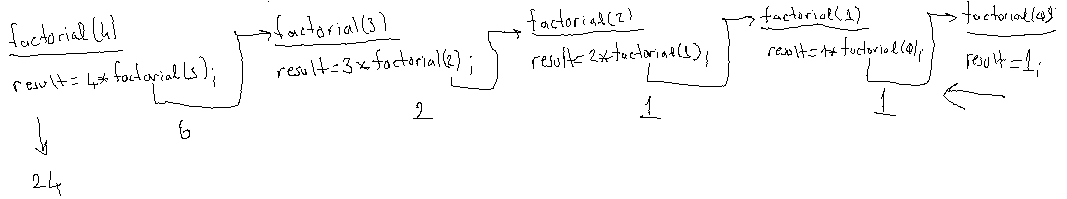
result = factorial(4);

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];

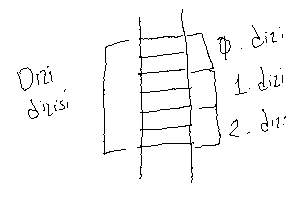
Böyle bir çok boyutlu dizinin başlangıç adresini tutabilecek gösterici şöyle tanımlanabilir:

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);

}

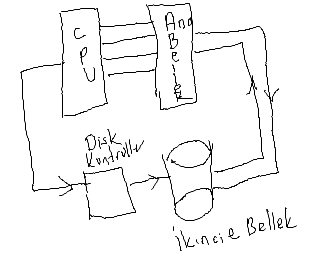
exit(status);

}

**Bellek Sistemleri**

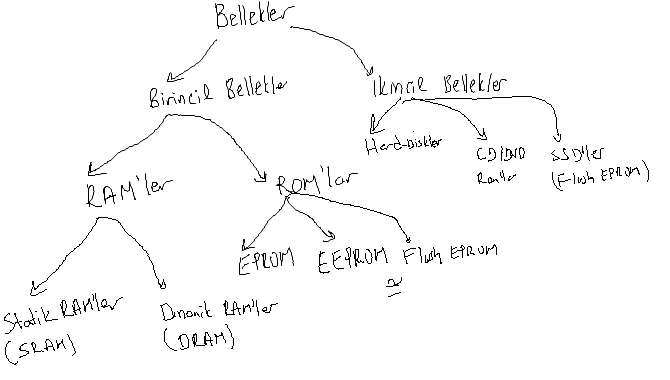
Bilgileri tutmakta kullanılan birimlere bellek (memory) denir. Bilgisayar sistemlerindeki bellekler "Birincil Bellekler (Primary Memory)" ve "İkincil Bellekler "Secondary Memory" olmak üzere ikiye ayrılmaktadır. Birincil belleklere "ana bellekler" ya da halk arasında "RAM" de denilmektedir. Birincil 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 sistem fonksiyonunu çağırmaktadır, Windows'ta CreateFile API fonksiyonu da bir sistem fonksiyonu gibidir) dosyayı açar. Sonra o dosya için bir cache oluşturur. Böylece okuma yazma işlemlerinde bu cache kullanılır. Yani örneğin biz işin başında fgetc fonksiyonu ile dosyadan bir byte okumak istediğimizde, fgetc bir byte değil işletim sisteminin sistem fonksiyonuyla (Linux'ta read, Windows'ta ReadFile) daha fazla bilgiyi okuyarak cache'e çeker. Sonraki okumalarda bize bilgiyi cache'ten verir. Burada cache sistemi tamamen bizim user mod prosesimizin bir parçası biçimindedir. Kernel ile bir ilgisi yoktur. Standart stdio.h fonksiyonlarının kullandığı bu cache sistemi read/write bir sistemdir.

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

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

- Cache read/write biçimdedir.

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

- Her dosyaynın ayrı bir cache'i vardır

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

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

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

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

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

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

setbuf fonksiyonunun prototipi şöyledir:

#include <stdio.h>

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

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

#include <stdio.h>

#include <stdlib.h>

int main(void)

{

FILE \*f;

char buf[BUFSIZ];

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

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

exit(EXIT\_FAILURE);

}

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

/\* ...\*/

fclose(f);

return 0;

}

setvbuf fonksiyonunun parametrik yapısı da şöyledir:

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

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

\_IOFBF (tam tamponlama için)

\_IOLBF (satır tamponlaması için)

\_IONBF (sıfır tamponlama için)

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

#include <stdio.h>

#include <stdlib.h>

int main(void)

{

FILE \*f;

char buf[BUFSIZ];

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

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

exit(EXIT\_FAILURE);

}

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

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

exit(EXIT\_FAILURE);

}

/\* ...\*/

fclose(f);

return 0;

}

**fgetc mi getc mi?**

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

**Aygıt Sürücüler Nasıl Kullanılır?**

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

**stdin, stdout, stderr Dosyaları**

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

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

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

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

**C'nin Gizli Dosya Fonksiyonları**

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

**Dosyaların Yönlendirimesi (IO Redirection)**

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

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

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

./sample > x.txt

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

Benzer biçimde stdin dosyası da şöyle yönlendirilebilir:

./sample < x.txt

Bireden fazla yönlendirme berber de yapılabilir. Örneğin:

./sample > x.txt < y.txt

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

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

Şüphesiz programın aynı zamanda komut satırı argümanları da girilebilir:

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

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

**stderr Dosyasının Anlamı**

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

./sample

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

./sample 2> x.txt

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

./sample 2> /dev/null

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

**Komut Satırında Boru (Pipe) İşlemleri**

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

a | b

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

ls -l | wc

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

ls -l > temp.txt

wc temp.txt

rm temp.txt

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

ps -e | grep "sample"

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

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

/\* lcount.c \*/

#include <stdio.h>

#include <stdlib.h>

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

{

FILE \*f;

int ch;

int count = 0;

if (argc == 1)

f = stdin;

else

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

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

exit(EXIT\_FAILURE);

}

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

if (ch == '\n')

++count;

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

return 0;

}

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

ps -e | ./lcount

Borular kombine edilebilir. Yani aşağıdaki işlem geçerlidir:

a | b | c

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

**stdin Dosyalarında EOF Etkisinin Yaratılması**

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

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

...

}

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

**stdin, stdout ve stderr Dosyalarının Tamponlaması**

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

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

#include <stdio.h>

int main(void)

{

printf("ankara");

for (;;)

;

return 0;

}

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

#include <stdio.h>

int main(void)

{

printf("ankara\n");

for (;;)

;

return 0;

}

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

#include <stdio.h>

int main(void)

{

printf("ankara");

fflush(stdout);

for (;;)

;

return 0;

}

Ya da stdout dosyasının tamponlama modunu sıfır tamponlamalı moda çekebiliriz:

#include <stdio.h>

int main(void)

{

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

printf("ankara");

for (;;)

;

return 0;

}

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

./sample > x.txt

Burada '\n' karakterini görünce değil tampon doulunca dosyaya yazma yapılır.

3) stderr dosyası ister karşılıklı etkileşimli aygıta yönlendirilmiş olsun isterse olmasın hiçbir zaman işin başında tam tamponlamalı modda olamaz. Yani biz stderr dosyasına yazma yaparken yazının sonuna '\n' karakterini eklersek her zaman yazı aygıta transfer edilecektir.

Peki Windows ve Linux'ta stdin dosyasının default tamponlama modu nasıldır? sistemlerin hemen hepsinde eğer stdin terminale yönlendirilmişse (yani kalvyeyi temsil ediyorsa) bu dosyanın default tamponlaması satır tamponlamalı moddur. Yani biz getchat bir fonksiyonla stdin'den bir karakter okumak istediğimizde getchar aygıt sürücüden bir satır talep eder. Biz de ENTER tuşuna basana kadar pek çok karakter girebiliriz. Bastığımız ENTER '\n' anlamına gelir. Bu '\n' karakteriyle birlikte klavyeden girilen tüm karakterler stdin dosyasının tamponuna çekilir. getchar bize onun ilkini verir. Sonra getchar çağormaya devam edersek fonksiyon bize tampondakileri sırasıyla verecektir. En son getchar bize tampondaki '\n' karakterini verir. Artık tampon boşalmıştır. Bir daha getchar çağırırsak o yine tamponu bir satırla doldurmak ister. Örneğin aşağıdaki programda yalnızca 'a' tuşuna ve sonra ENTER tuşuna basmış olalım. Ne olacaktır?

#include <stdio.h>

int main(void)

{

int ch1, ch2;

ch1 = getchar();

ch2 = getchar();

printf("ch1 = %d, ch2 = %d\n", ch1, ch2);

return 0;

}

Burada tamponda "a\n" karakterleri bulunur. Birinci getchar tampondan a'yı alır. İkinci getchar tamponda bilgi olduğundan klavyeden giriş beklemez. O da '\n' yi alır. Bir daha getchar kullansaydık yeniden giriş istenecekti. Peki aşağıdaki örnekte program nasıl çalışır?

#include <stdio.h>

int main(void)

{

int ch;

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

putchar(ch);

return 0;

}

Burada birinci getchar dolayısıyla bizden bir satır istenir. Arık diğer getchar'lar girdiğimiz yazının karakterlerini alarak yazdırır. Tabi tamponun sonundaki '\n' de yazdırılır. Bu sırada imleç aşağıdaki satırın başına geçmiş olur. Sonraki getchar yine bizden bir satır isteyecektir. Döngüden çıkmak için EOF tuş kombinasyonları kullanılır.

**stdin Dosyasını Kullanan Standart C Fonksiyonlarının Davranışı**

Yukarıda zaten getchar fonksiyonu açıklandı. Burada diğer fonksiyonlar üzerinde duracağız.

**gets Fonksiyonu:** Fonksiyonun prototipi şöyledir:

char \*gets(char \*str);

Fonksiyon '\n' görene kadar ('\n' dahil olmak üzere) ya da EOF görene kadar stdin dosyasından karakterleri okur ve parametresiyle aldığı diziye yerleştirir. Fakat '\n' için '\0' karakterini diziye yerleştirir. gets normal durumda parametresiyle aldığı adresin aynısına geri döner. Fakat stdin dosyasından hiç okuma yapamadan EOF görürse NULL adrese geri döner. Örneğin:

#include <stdio.h>

int main(void)

{

int ch;

char s[100];

ch = getchar();

gets(s);

printf("%c-%s\n", ch, s);

return 0;

}

Burada getchar ile biz "anakara" yazısını girip ENTER tuşuna basmış olalım. Girdiğimiz yazının ilk karakterini getchar alır, diğerlerini gets alır. gets tamponun sonundaki '\n' yi de okur fakat onun yerine diziye '\0' yerleştirir. Peki çağrılar ters sırada olsaydı ne olurdu?

#include <stdio.h>

int main(void)

{

int ch;

char s[100];

gets(s);

ch = getchar();

printf("%c-%s\n", ch, s);

return 0;

}

Bırada gets tüm tamponun bir satırla dolmasına yol açar fakat onların hepsini okur. Dolayısıyla getchar tamponu boş göreceğinden bizden yeniden giriş istenecektir. Aşağıdaki programı stdin dosyasını bir dosyaya yönlendirerek test ediniz:

#include <stdio.h>

int main(void)

{

char buf[4096];

while (gets(buf) != NULL)

puts(buf);

return 0;

}

Burada dosyanın sonuna gelindiğinde artok gets hiçbir karakter okuyamadan EOF ile karşılaşır ve NULL adrese geri döner. Böylece döngüden çıkılmış olur.

**scanf Fonksiyonu**

scanf fonksiyonu formatlı okuma yapmak için kullanılmaktadır. Format karakterleri girdinin biçimini ve türünü belirtir. Fonksiyonun prototipi şöyledir:

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

Fonksiyon yerleştirme yapılan nesne sayısına geri döner. Örneğin:

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

scanf önce boşluk karakterlerini atar sonra okumayı yaparak nesneye yerleştirir. Yani scanf baştaki boşluk karakterlerini (leading space) okur fakat sondakileri (trailing space) okumaz, sondakiler stdin tamponunda kalır. scanf stdin dosyasından bilgileri karakter karakter okur, okuduğu karakterin format karakterine uygun olmadığını gördüğünde onu stdin tamponuna geri bırakır (ungetc standard C fonksiyonuna bakınız) ve işlemini sonlandırır. Örneğin:

#include <stdio.h>

int main(void)

{

int a;

int result;

char s[100];

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

gets(s);

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

return 0;

}

Burada scanf için " ali " girişini yapmış olalım. Aşağıdaki gibi bir sonuç elde edilmiştir

a = -858993460, result = 0, s = "ali "

a'ya yerleştirilme yapılmadığı için çöp değer vardır. scanf sıfır ile geri dönmüştür. gets ise tampondan geri kalanların hepsini okumuştur. Örneğin:

#include <stdio.h>

int main(void)

{

int a, b;

int result;

char s[100];

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

gets(s);

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

return 0;

}

Burada klavyeden " 100 ali " girmiş olalım. Şöyle bir sonuç elde edilmiştir:

a = 100, b = -858993460, result = 1, s = "ali "

Görüldüğü gibi a için başarılı yerleştirme yapılmıştır. b için scanf işlemi başlattığında önce ali'nin solundaki boşluk karakterlerini atmıştır. Fakat a'yı beğenmemiş veonu yeniden stdin tamponuna bırakmıştır. gets de geri kalanları almıştır.

scanf fonksiyonunda format karkaterleri arasında başka bir karakter konulursa scanf okuma sırasında bu karakterlerin o pozisyonda bulunmasını ister. Örneğin:

#include <stdio.h>

int main(void)

{

int a, b;

int result;

char s[100];

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

gets(s);

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

return 0;

}

Girişin "100 200" biçiminde yapıldığını düşünelim. scanf 100'den sonra ',' karakteri beklemektedir. Bu gelmediği için oradan okuduğu boşluk karakterini tampona geri bırakır ve işlemini sonlandırır. Şöyle bir çıkış elde edilmiştir:

a = 100, b = -858993460, result = 1, s = " 200"

Format karakterleri arasında hiç boşluk kujllanılmazsa bu durum "önceki boşluk karakterlerinin (leading space) atılacağı anlamına" gelir. Fakat araya bir başka bir karakter getirilirse artık boluklar atılmaz kesinlikle o karakter beklenir. Aşağıdaki iki çağrı arasındaki farkı inceleyiniz:

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

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

Örneğin:

#include <stdio.h>

int main(void)

{

int day, month, year;

scanf("%d/%d/%d", &day, &month, &year);

printf("%d/%d/%d\n", day, month, year);

return 0;

}

Format kısmında tek boşluk "bir ya da birden fazla boşluğu at" anlamına gelir. Örneğin:

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

Burada ilk girişten sonraki boşluk karakterlerinin hepsi atılacaktır. Bunun boşluksuz durumdan işlevsel farklılığı yoktur. Aşağıdaki bir çağrıda ilginç bir durum oluşujr:

scanf("%d%d\n", &a, &b);

Burada formak karakterlerinden sonra bir boşluk karkateri (white space) görüldüğü için scanf bu noktada tüm boşluk karakterlerini atmak ister. Böylece biz boşluk girdiğimiz sürece bizden karkter istemeye devam eder. Yani boşluk karakteri girmeyene kadar scanf bizden karakter isteyecektir.

scanf boşluk karakterlerini ayıraç kullandığı için %s ile okuma yaparken ilk boşluk karkaterini gördüğünde okumayı tamamlar. Yani biz %s ile boşluklu bir yazı okuyamayız.

scanf eğer boşluk karakterlerini attıktan sonra (ya da atmadan önce) dosya sonuyla karşılaşırsa EOF değerine (-1) geri döner.

Aşağıdaki gibi bir a.txt dosyası olsun. Dosyanın içerisinde boşluklarla ayrılmış pek çok sayı vardır:

a.txt

------

100 200

300

400

500 600

700 800

900

100

Şimdi biz bu değerleri "IO redirection" ile okumak isteyelim. Komut satırında programı şöyle çalıştırırız:

./sample < a.txt

Aşağıdaki program bunu yapabilir:

#include <stdio.h>

int main(void)

{

int val;

while (scanf("%d", &val) != EOF)

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

return 0;

}

**Anahtar Notlar:** C'nin stdin'den okuma yapan fonksiyonlarının hepsi yalnızca getchar (yani fgetc(stdin)) ve ungetc ile yazılabilir. Benzer biçimde dosyadan formatlı okujma yapan fonksiyonlar da yine fgetc ve ungetc kullanılarak yazılabilir. Örneğin gets fonksiyonunu şöyle yazabiliriz:

char \*mygets(char \*s)

{

int ch;

size\_t i;

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

s[i] = ch;

if (i == 0)

return NULL;

s[i] = '\0';

return s;

}

**Anahtar Notlar:** gets fonksiyonunun tasarımında bir kusur vardır. Bu fonksiyonda her zaman kullanıcı daha fazla giriş yaparsa dizi taşması söz konusu olabilir. gets fonksiyonu C99-TC3'te "deprecated" yapılmıştır ve C11'de tamamen kaldırılmıştır. Bunun yerine gets\_s getirilşmiştir. Bu fonksiyon dizinin uzunluğunu da parametre olarak alır dolayısıyla dizi taşması engellenmiş olur. gets\_s şöyle yazılabilir:

char \*mygets\_s(char \*s, size\_t n)

{

int ch;

size\_t i;

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

s[i] = ch;

if (i == 0)

return NULL;

s[i] = '\0';

return s;

}

Eskiden gets\_s yaygın olarak bir eklenti biçiminde bulunmuyordu. O günlerde gets yerine daha güvenlikli fgets kullanılabiliyordu. (Ne de olsa fgets'in uzunluk parametresi vardır.) Ancak fgets kullanılırken bu fonksiyonun '\n' karakterini de okuduğun unutulmamalıdır. Dizideki '\n' karakterinin silinmesi gerekir. Örneğin:

#include <stdio.h>

#include <string.h>

int main(void)

{

char s[20];

char \*str;

fgets(s, 20, stdin);

if ((str = strchr(s, '\n')) != NULL)

\*str = '\0';

puts(s);

return 0;

}

**stdin Tamponunun Boşaltılması**

Bazen tamponda kalanları değil kullanıcının yeni girişini okumak isteyebiliriz. Öneğin iki getcharf fonksiyonunu peş peeşe kullanacak olalım. İkinci getchar ile yeni bir giriş isteyebiliriz. Ya da gets ile scanf birlikte kullanılırken de benzer bir istek oluşabilir. Örneğin:

#include <stdio.h>

int main(void)

{

int no;

char name[64];

printf("No:");

scanf("%d", &no);

printf("Name:");

gets(name);

printf("No = %d, Name = %s\n", no, name);

return 0;

}

Maalesef C'de standart olarak stdin tamponunu boşaltan bir fonksiyon yoktur. Bazı derleyicilerde fflush(stdin) bunu yapıyor olmakla birlikte, bu kullanım uygıunsuzdur. Çünkü her şeyden önce fflush read-only dosyalarda kullanılamaz ve stdin read only bir dosyadır. Tamponu boşaltmak için küçük bir fonksiyon ya da makro yazılabilir:

#include <stdio.h>

void empty\_stdin(void)

{

int ch;

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

;

}

int main(void)

{

int no;

char name[64];

printf("No:");

scanf("%d", &no);

empty\_stdin();

printf("Name:");

gets(name);

printf("No = %d, Name = %s\n", no, name);

return 0;

}

Tamponu boşaltırken EOF kontrolü de yapılmıştır. Aksi taktirde yönlendirme durumunda sonsuz döngü oluşabilir. Aynı fonksiyonu makro olarak dayazabiliriz:

#define empty\_stdin() \

do { \

int ch; \

\

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

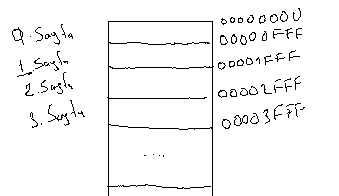
; \

} \

while (0)

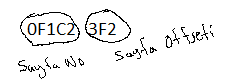
**İşlemcilerin Sayfalama Mekanizması (Paging Mechanisms)**

Modern ve güçlü masaüstü işlemcileri sayfalama (paging) denilen bir mekanizmaya sahiptir. Sayfalama mekanizmasında bellek sayfa (page) denilen ardışıl byte bloklarına ayrılmıştır. Sayfaların büyüklükleri işlemciye bağlıdır. Pek çok işlemci 4K uzunluğunda sayfa kullanmaktadır. Bellekte her bir sayfaya 0'dan başlayark bir numara verilmiştir. Örneğin 4K'lın sayfa kullanan sistemde fiziksel belleğin haritalanması şöyledir:

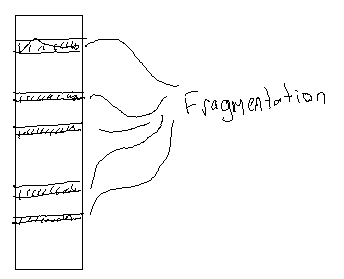


32 bit bir sistemde adresler 4 byte yer kaplar (8 hex digit). Her fiziksel adres aslında bir sayfa içerisindedir ve o sayfada belli bir offset'tedir. Adresin hangi sayfada olduğunu bulmak için onu sayfa uzunluğuna bölmek gerekir. (Tabi 2'lik sistemde bir sayıyı 4096'ya bölmek demek 12 kez sağa kaydırmak, başka bir deyişle sağdaki 12 biti atmak demektir.) Adresin o sayfanın hangi offset'inde olduğunu bulmak için sayfa uzunluğuna bölümünden elde edilen kalanı bulmak gerekir (örneğin sayının 4096'ya bölümünden elde edilen kalan düşük anlamlı 12 bitidir.) Örneğin:

0F1C23F2 adresi hangi sayfa ve offset'tedir?



**Anahtar Notlar**: Bellek sistemlerinde en önemli olgulardan biri bölünme (fragmentation) durumudur. Bölünme arşılık yerleştirilme zorunluluğunun yol açtığı bir sonuçtur. Eğer söz konusu belleğe (disk de olabilir RAM de olabilir) birtakım öğeler ardışıl yerleştiriliyorsa zamanla bunların yaratılıp yok edilmesi sonucunda küçük fakat çok sayıda alan oluşur. Bu alanlar küçük olduğu için pek bir işe yaramazlar. Fakat bellek kullanım oranını ciddi biçimde düşürürler. Bu olguya dışsal bölünme (external fragmentation) denilmektedir. Örneğin dışsal bölünmüş bir bellek şöyle görünümde olabilir (taranmış alanlar boş alanlar):



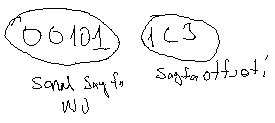
Peki bir program fiziksel belleğe ardışıl yerleştirilmiyorsa nasıl çalışmaktadır? İşte program içerisindeki adreslere "doğrusal adres (linear address), mantıksal adres (logical address) ya da sanal adres (virtual address)" denilmektedir. Biz burada "sanal adres" terimini kullanacağız. İşlemci bu sanal adresleri "sayfa tablosu (page table)" denilen bir tabloya bakarak fiziksel adreslere dönüştürmektedir. İşlemci sayfa tablosunu belli bir yazmacın (register) gösterdiği yerde arar. (Örneğin Intel işlemcilerinde CR3 yazmacı sayfa tablosunun yerini belirtmektedir.) Prosesin sayfa tablosunu işletim sistemi oluşturur ve bu yazmaca onun adresini verir. Böylece işlemci işletim sistemi tarafından oluşturulmuş olan sayfa tablosunu kullanır hale gelmektedir. Peki sayfa tablosu nasıldır? Bazı sistemlerde iki kademeli hatta üç kademeli sayfa tabloları kullanılmaktadır. Örneğin Intel'de iki kademeli sayfa tablosu kullanılır. Ancak biz burada kavramsal karmaşıklık oluşmasın diye tek kademeli bir sayfa tablosunu örnek vereceğiz. Sayfa tablosunun tipik biçimi şöyledir:

|  |  |  |
| --- | --- | --- |
| Sanal Sayfa No  (Aslında Tabloda Yok) | Fiziksel Sayfa No | Sayfa Özellikleri |
| ... | ... | ... |
| 00100 | 01FC3 | RW, P, User, D |
| 00101 | 2FC40 | RW, P, User, D |
| 00102 | 1C167 | RW, P, User, D |
| ... | ... | .... |

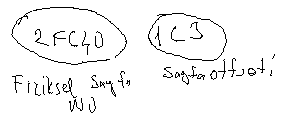
Tablonun ilk sütunu aslında bir indeks numarasıdır. Yani tabloda yoktur. Şöyle ki aslında 00100 sanal sayfasına ilişkin giriş tabloda tablonun başından itibaren bir sayfa girişinin uzunlu ile 0x100'ün çarpımı kadar uzaklıktadır. Tablodaki her bir satıra "sayfa tablosu girişi (page table entry)" denilmektedir. İşlemci bir sanal adresle karşılaştığında önce o sanal adresi sanal sayfa numarasına ve sayfa offsetine ayrıştırır. Sonra sanal sayfa numarasını sayfa tablousna indeks yapar ve oradan fiziksel sayfa numarasını elde eder. Ona da sayfa offset'ini toplar ve nihai fiziksel adresi elde eder. Örneğin aşağıdakiş gibi bir kod olsun:

MOV EAX, [001011C3]

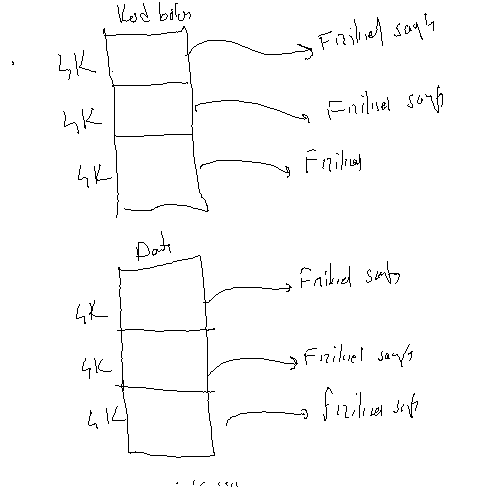
İşlemci bunu aşağıdaki gibi sanal syafa numarasına ve sayfa offset'ine ayırır:



bundan sonra işlemci sayfa tablosuna bakarak 0x101 numaralı sanal sayfa numarasının hangi fiziksel sayfa ile eşleştirildiğine bakar. Yukarıdaki örnekte 0x101 numaralı sanal sayfa 2FC40 fiziksel sayfasıyla eşleştirilmiştir. Bu adresin 4096 ile çarpımına 1C3 eklenir ve şu değer bulunur:



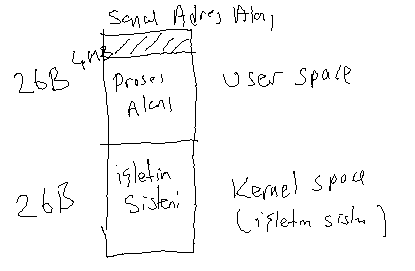
Böylece erişimi fiziksel bellekteki 2FC401C3 adresinden yapar. Böylece programın hepsi aslında 4k'lık fiziksel sayfalara atanır:



Derlenmiş bir kodda derleyicinin üretiiği adreslerin hepsi sanal adreslerdir. Yani derleyici 32 bit bir sistemde sanki program 4GB'lik fiziksel belleğe tek başına yüklenecekmiş gibi kod üretimi yapar. Sanki bütün programlar aynı adresten itibaren fiziksel bellekte tek başlarına çalışacakmış gibi bir koda sahiptir. İşletim sistemi programı yükleyeceği zaman onu sayfalara dağıtır ve sayfa tablosunu uygun biçimde düzenler. Böylece proseslerin sanal adres alanları aynı olmasına karşın onlar farklı fiziksel adreslerde bulunurlar. Başka bir deyişle örneğin Windows'ta ve Linux'ta farklı proseslerdeki aynı sanal adresler aslında fiziksel bellekte aynı yere tekabül etmezler. Aslında bu tür sistemlerde fiziksel belleğin bir önemi yoktur.

Her prosesin ayrı bir sayfa tablosu olduğunu vurgulayalım. İşletim sistemi her prosesler arası geçiş oluştuğunda otomatik olarak işlemcinin kullandığı sayfa tablosunu da değiştirir.

Sayfalama mekanizmasını kullanan işletim sistemlerinde proseslerin bellek alanları tam olarak izole edilmiştir. İşletim sistemi her prosesin sayfa tablosunda onların sanal sayfalarını farklı fiziksel sayfalara yönlendirdiği için istense de bir proses başka bir prosesin alanına erişemez. Tabi işletim sistemi her zaman bellektedir. Örneğin 32 Windows sistemlerinde tipik sanal bellek alanının kullanımı şöyledir:



Tipik olarak Windows sistemlerinde derleyicinin ürettiği kod sanki program fiziksel belleğin 4MB'tan başlayan kısmına yüklenecekmiş gibi oluşturulur. Bu adres .exe dosyanın içerisine yazılır. Ve istenirse Linker ayarlarıyla değiştirilebilmektedir. Yani işletim sistemi bütün proseslerin sayfa tablosunda aynı yerde bulunmaktadır.

Sayfa tablosu girişlerindeki sayfa özellikleri ilgili sayfaya erişim haklarını ve başka birtakım bilgileri içerir. Örneğin bir sayfa read-only ise işlemci oraya yazmaya çalıştığında exception oluşur. İşletim sistemi devreye girer ve prosesi cezalandırarak sonlandırır. Örneğn C'de Microsoft derleyicileri string'leri read-only sayfalara yerleştirmektedir. Böylece bir string update edilmek istendiğinde exception oluşur.

#include <stdio.h>

int main(void)

{

char \*s = "ankara";

\*s = 'x'; /\* read-only sayfaya erişim! \*/

return 0;

}

Peki bu sistemlerde proseslerin bellek alanları izole edilmiştir. Peki bir proses işletim sisteminin alanına erişemez mi? İşte işletim sistemi kendini başka bir mekanizmayla korumaktadır. Her fiziksel sayfa "kernel" ya da "user" biçiminde önceliklendirilmiştir. User sayfalarına ancak "user mod" prosesler erişebilir. "Kernel mod" prosesler her türlü sayfaya erişebilir. Bu durumda biz kendi programımızda işletim sisteminin sayfalarına erişirsek bunlar mod sayfalar olduğu için yine exception oluşur ve prosesimiz sonlandırılır. Sayfanın "user mod" sayfa mı "kernel mod" sayfa mı olduğu yine sayfa tablousu girişinde "ayfa özellikleri" kısmında belirtilmektdir.

Sayfalama mekanizmasını kullana işletim sistemleri bellek yönetiminde şu izlemeleri yapabilmeli ve kararkları verebilmelidir:

- Fiziksel bellekteki sayfaların boş mu dolu mu olduğunun takibi

- Sayfa tablolarının oluşturulması ve programın fiziksel sayfalara atanması

**Sanal Bellek Mekanizması (Virtual Memory)**

Sanal bellek bir programın tamamanın değil bir kısmının fiziksel belleğe yüklenerek disk (ikincil bellekle) ile fiziksel bellek arasında yer değiştirmeli olarak çalıştırılmasına denir. Sanal bellek mekanizmasında bir programın yalnızca belirli kısmı fiziksel belleğe yüklenerek program çalıştırılır. Sonra program çalışırken kod ya da data bakımından fiziksel bellekte olmayan bir kısma erişildiğinde işlemci bunu anlar ve bir içsel kesme oluşturur. Bu içsel kesmeye Intel terminolojisinde "page fault" denilmektedir. Böylece işletim sistemi devreye girer. Prosesin bellekte olmayan sayfasını diskten bulur, onu fiziksel boş bir sayfaya yükler, prosesin sayfa tablosunu günceller ve kesmeden çıkar. Intel terminolojisinde "fault" kesmelerin çılıkdığında çalışma aynı makina komutuyla devam etmektedir. Böylece işletim sistemi işini bitirdiğinde yine akış "fault"a yol açan komutla devam edecektir. Artık ilgili sanal sayfaya sayfa tablosunda bir fiziksel sayfa karşılık getirildiği için fault oluşmaz. Bir sanal sayfanın o anda fiziksel bellekte olup olmadığı sayfa özelliklerindeki P (Present) biti ile anlaşılmaktadır.

Sanal bellek sayesinde bir programın küçük bir kısmı fiziksel belleğe yüklenerek program çalışır. Sanal bellek gerçekleştirimi adım adım şöyle yapılmaktadır:

1) Program kodunda bir sanal adrese erişilmiştir. İşlemci sayfa tablosuna bakar ve bunu fiziksel adrese dönüştürmeye çalışır. eğer sayfanın P biri 0 ise işlemci bunun fiziksel bellekte olmadığına karar verir ve "page fault" oluşturu.

2) İşletim sistemi devreye girer ve prosesin erişilmek istenen sanal sayfasının diskte nerede olduğunu bulur. Onu fiziksel RAM'e yüklemek ister. Tabi bunun için boş bir fiziksel sayfa bulmaya çalışır. Eğer bulursa disktenm o sayfayı o fiziksel sayfaya yükler . Bu sürece İnglizce "Swap-In" denilmektedir. Peki ya fiziksel bellek tıka basa doluysa? Bu durumda işletim sistem, fiziksel bellekten başka bir prosese ait olan bir sayfayı çıkarmak ister. Onu o haliyle yeniden diske yazazacaktır. Bu işleme ise İngilizce "Swap-Out" denilmektedir. Tabi burada işletim sistemi gelecekte en az kullanılacak bir sayfayı fiziksel RAM'den çıkarmak isteyecektir. Tıka basa dolu bir RAM'de her Swap-In işleminde bir Swap-Out yapılması sistemi yavaşlatmaktadır. Bu probleme İngilizce "thrashing" denilmektedir.

3) İşletim sistemi Swap-In yaptıktan sonra prosesin sayfa tablosunu düzeltir ve fault'tan çıkar. Böylece akış fault'a yol açan makina komutuyla devam edecektir. Tabi artık fault oluşmayacaktır.

Yukarıdaki işlemin bazı ayrıntıları vardır. Örneğin "Swap-In" yapıcacağı zaman programın ilgili kısmı diskte nereden alınacaktır? Bu konuda işletims sitemlerinin değişik stratejileri vardır. Bazı işletim sistemleri her executable dosyanın kendisini hem de bir swap dosyasını bu amaçla kullanır (örneğin Windows). Bazı sistemlerde executable dosya hiç kullanılmaz ve birden fazla swap dosyası kullanılır. Bazı sistemler swap dosyası yerine ayrı bir disk bölümünü (partition) kullanılmaktadır. Peki neden executable dosya bunun için yeterli olmamaktadır? Eğer bir fiziksel sayfa üzerinde değişiklik yapılmışsa onu executable dosyaya yazmak mümkün değildir. Fakat read-only sayfaları her zaman oradan alabiliriz. Swap-out yapılacağı zaman sayfanın diske geri yazılması gerekir mi? İşte sayfa tablo girişlerinde bir D biti de bulunmaktadır. İşlemci ne zaman bir sayfaya yazma yapsa bu D bitini set eder. İşletim sistemi de Swap-out yapacağı zaman bu D bitine bakar. Bu D biti 0 ise boşuna sayfayı swap dosyasına yazmaz.

Peki proses hiç tahsis etmediği bir alana erişmek isterse ne olur? Çünkü alanın diskte de bir karşılığı yoktur. İşte işletim sistemi page fault oluştuğunda önce fault'a yol açan sanal adresin gerçekten legal bir adres olup olmadığına bakmaktadır. Yani erişlmek istenen adres tahsis edilmiş bir adres değilse prosesi hemen sonlandırır.

Peki aynı program ikinci kez yüklendiğinde yeniden onun bazı sayfaları fiziksel belleğe yüklenir mi? İşte işletim sistemi değişmemiş ve değişmeyecek sayfaları yeniden fiziksel belleğe yüklememektedir. Yani örneğin biz iki kez notepad.exe programını çalıştırsak programın kod bölümü zaten hiç update edilmeyeceği için iki farklı notepad prosesinin sayfa tablosunda bazı girişler aynı fiziksel adrese yönlendirilmektedir. Böylece aynı programı biz iki çalıştırdığımızda aslında tekrar her şey fiziksel belleği yüklenmez.

İşletim sistemi programın ne kadarını fiziksel belleğe yüklemektedir? Prosesin fiziksel bellekteki kısmına İnglizce "Resident Set" ya da "Working Set" denilmektedir. Bu da işletim sistemlerini yazanların kullandıkları algoritmaya bağlı olarak değişmektedir. İşletims sitemlerinin en zor gerçekleştirilen alt sistemlerinden biri "bellek yöneticisi (memory management)" bölümüdür.

**Proseslerin Yaratılması**

Bir prosesi yaratmak için işletim sistemlerinde sistem fonksiyonları bulunur. Windows sistemlerinde CreateProcess isimli API fonksiyonu, UNIX/Linux sistemlerinde fork fonksiyonu proses yaratmak için kullanılmaktadır.

Windows'ta CreateProcess API fonksiyonunun prototipi şöyledir:

BOOL WINAPI CreateProcess(

\_\_in LPCTSTR lpApplicationName,

\_\_in\_out LPTSTR lpCommandLine,

\_\_in LPSECURITY\_ATTRIBUTES lpProcessAttributes,

\_\_in LPSECURITY\_ATTRIBUTES lpThreadAttributes,

\_\_in BOOL bInheritHandles,

\_\_in DWORD dwCreationFlags,

\_\_in LPVOID lpEnvironment,

\_\_in LPCTSTR lpCurrentDirectory,

\_\_in LPSTARTUPINFO lpStartupInfo,

\_\_out LPPROCESS\_INFORMATION lpProcessInformation

);

Fonksiyonun birinci parametresi çalıştırılacak program dosyasının yol ifadesini alır. Burada yol ifadesi mutlak ya da göreli olarak verilebilir. Fonskiyonun ikinci parametresi programın komut satırı argümanlarını belirtir. Komut satırı argümanları tek bir yazı olarak fonksiyona verilmektedir. Sonra C derleyicilerinin başlangıç kodları (start up code) bu yazıyı boşluklarda parse ederek argv dizisini oluşturmaktadır. İlk komut satırı argümanının programın yol ifadesi olması C'de zorunlu tutulmuştur. Fakat işletim sistemi genelinde böyle bir zorunluluk yoktur. Kolaylık olsun diye şöyle bir seçenek de sunulmuştur: Eğer birinci parametre NULL geçilirse ikinci parametredeki ilk boşluksuz kısım sanki birinci parametredeki çalıştırılabilen dosyanın yol ifadesi gibi ele alınmaktadır. Dosya ismini bu yolla vermenin diğerinde bir farkı daha vardır. Bu yöntemde eğer dosya ismi hiç '\' karakteri içermiyorsa Windows onu sırasıyla bazı dizinlerde arar. eğer buradaki dosya ismi '\' karaketeri içeriyorsa Windows onu yol ifadesi ile belirtilen yerde arar fakat başka bir dizine bakmaz. Yine fonksiyonun birinci parametresi NULL geçilmezse Windows programı başka yerde aramamaktadır. Burada ikinci parametrenin const olmayan bir adres olduğuna dikkat ediniz. (\_\_in\_\_out). Biz ikinci parametreye bir string ifadesi vermemeliyiz. Çünkü CreateProcess bu adresteki bilgiyi saklayıp burayı tampon olarak kullanarak fonksiyon çıkışında yeniden orijinal yazıyı burada bırakır. Ama bu diziyi güncellemektedir. Halbuki string ifadeleri güncellenemez. Eğer birinci parametre NULL geçeilirse ve ikinci parametredeki dosya ismi '\' içermiyorsa Windows dosyayı sırasıyla şu dizinlerde arar:

1) CreateProcess fonksiyonunu uygulayan programın program dosyasının bulunduğu dizin

2) CreateProcess uygulayan prossesin o andaki çalışma dizini

3) 32 Bit Windows System dizini (tipik olarak c:\windows\system32 dizini)

4) 16 bit Windows dizini (tipik olarak c:\windows\system)

5) Windows'un kendi dizini (tipik olarak c:\windows)

6) CreateProcess fonksiyonunu uygulayan prossesin PATH çevre değişkeni ile belirtilen dizinleri

Fonkısiyonun üçüncü ve dördüncü parametreleri prossese ve prossesin ana thread'ine ilişkin güvenlik parametreleridir. Bu parametreler NULL olarak geçilebilir. Bu durumda default güvenlik durumu anlaşılır. Fonksiyonun beşinci parametresi kernel nesnelerinin alt prosseslere geçirilebilmesine ilişkin ana şalter görevindedir. Bu parametre FALSE olarak geçilebilir. Altıncı parametre yaratılacak prossese ilişkin çeşitli belirlemeleri içermektedir. Bu parametre bazı bayrakların bit OR işlemine sokulmasıyla oluşturulur. Fakat istenirse bu parametre sıfır geçilebilir. Fonksiyonun yedinci parametresi yaratılacak prossesin çevre değişken listesini belirtir. Bu paramere NULL geçilirse yaratılacak prossesin çevre değişkenleri üst prossesten alınır. Fonksiyonun sekizinci parametresi yaratrılacak pross esin çalışma dizinini belirtir. Eğer bu parametre NULL geçilirse prosesin çalışma dizini üst prosesten alınır. Fonksiyonun dokuzuncu parametresi yaratılacak prosese ilişkin bazı ayrıntıların belirlenmesini sağlar. Bu paramtereye STARTUPINFO türünden bir yapının adresi geçirilmelidir. Bu yapının ilk elemanına yapının sizeof'u yazılmalıdır. Diğer elemanlar boş bırakılabilir. Çünkü yapının elemanları sıfır ise bu default durum anlamına gelir. Fonksiyonun son parametresi PROCESS\_INFORMATION türünden bir yapının adresini alır. Fonksiyon bu yapının içini bizim için doldurur. Bu yaıya fonksiyon yaratılacak proses nesnesinin HANDLE ve Id değerlerini, yaratılacak ana thread nesnesinin HANDLE ve id değerlerini yerleştirir. Prosesin HANDLE değeri Proses Kontrol Bloğuna erişmek için sistem tarafından kullanılmaktadır.

Fonksiyon başarı durumunda sıfır dışı herhangi bir değere, başarısızlık durumunda 0 değerine geri döner.

Aşağıda maksimum default değerler geçilerek bir prosesin yaratılmasına örnek verilmiştir:

#include <stdio.h>

#include <stdlib.h>

#include <Windows.h>

void ExitSys(LPCSTR lpszMsg, int status);

int main(void)

{

char cmdLine[] = "c:\\windows\\system32\\notepad.exe";

STARTUPINFO si = { sizeof(STARTUPINFO) };

PROCESS\_INFORMATION pi;

if (!CreateProcess(NULL, cmdLine, NULL, NULL, FALSE, 0, NULL, NULL, &si, &pi))

ExitSys("CreateProcess", EXIT\_FAILURE);

return 0;

}

void ExitSys(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);

}

**Anahtar Notlar:** Windows API fonksiyonları hem ASCII hem de UNICODE karakter tablosuyla çalışabilmektedir. Visual Studio'da default durumda karakter tablosu UNICODE biçimdedir. Yukarıdaki örnekte bunun ASCII yapılması gerekir. Bunun için proje seçeneklerine gelinmeli "Caharacter Set" seçeneği "Not Set" yapılmalıdır.

**Sınıf Çalışması**: Basit bir C programı yazınız. Programı derlemek için cl.exe'yi aşağıdaki gibi çalıştırınız:

cl.exe test.c

Visual Studio 2013 için cl.exe şuarada bulunmaktadır:

C:\Program Files (x86)\Microsoft Visual Studio 12.0\VC\bin\cl.exe

**Çözüm:**

#include <stdio.h>

#include <stdlib.h>

#include <Windows.h>

#include <winapifamily.h>

void ExitSys(LPCSTR lpszMsg, int status);

int main(void)

{

char cmdLine[] = "\"C:\\Program Files (x86)\\Microsoft Visual Studio 12.0\\VC\\bin\\cl.exe\" "

/\*

"-I \"C:\\Program Files (x86)\\Microsoft Visual Studio 12.0\\VC\\include\" "

"-I \"c:\\Program Files (x86)\\Windows Kits\\8.1\\Include\\um\" "

"-I \"c:\\Program Files (x86)\\Windows Kits\\8.1\\Include\\shared\" "

\*/

" sample.c";

STARTUPINFO si = { sizeof(STARTUPINFO) };

PROCESS\_INFORMATION pi;

if (!CreateProcess(NULL, cmdLine, NULL, NULL, FALSE, 0, NULL, NULL, &si, &pi))

ExitSys("CreateProcess", EXIT\_FAILURE);

return 0;

}

void ExitSys(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);

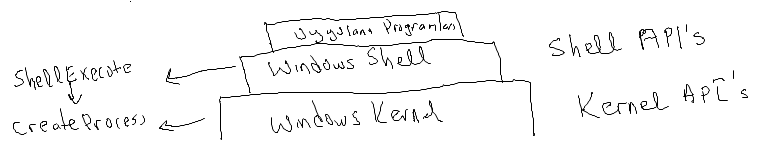
}

exit(status);

}

**ShellExecute Fonksiyonu**

ShellExecute bir sistem fonksiyonu değildir. Zaten kendi içerisinde CrateProcess isimli sistem fonksiyonunu çağırır. ShellExecute bir Shell fonksiyonudur. Windows'un kabuğuna "Windows Explorer" denilmektedir ve proses olarak "explorer.exe" programıdır. Aslında Windows'ta shell tamamen devre dışı bırakılabilir. Tabi bu durumda bu shell fonksiyonlarını kullanamayız. Fakat shell fonksiyonları da Windows'un bir parçası durumundadır.



ShellExecute fonksiyonuna biz çalıştırılmayan bir dosya verebiliriz. ShellExecute "Registry" kayıtlarıne erişerek bu uzantılı dosyanın hangi çalıştırılabilen dosyayla ilişkilendirilmiş olduğunu belirler ve CreateProcess ile o çalıştırılabilen dosyayı çalıştırır. Sonra da bizim verdiğimiz dosyayı ona komut satırı argümanı olarak geçirir.

**Anahtar Notlar:** Bazen Micreosoft bazı API fonksiyonlarını çeşitli nedenlerle geliştirmek istemektedir. Tabi eski fonksiyonların da muhafaza edilmesi gerekir. İşte Microsoft bu tür durumlarda fonksiyonun ilacve yeni bir XXXExli versiyonunu oluşturur (Örneğin CreateWindow, CreateWindowEx gibi.)

Fonksiyonun prototipi şöyledir:

#include <shellapi.h>

HINSTANCE ShellExecute( HWND hwnd,

LPCTSTR lpOperation,

LPCTSTR lpFile,

LPCTSTR lpParameters,

LPCTSTR lpDirectory,

INT nShowCmd

);

Fonksiyonun birinci paraametresi bir GUI penceresinin HANDLE değerini alır. Fonksiyon başarısızlık durumunda bir MessageBox çıkartabildiği için böyle parent pencereye gereksinim duymaktadır. Bu parametre NULL geçilirse masaüstü penceresi anlaşılır. İkinci parametre yapılmak istenen eylemi belirtir. eğer bir program çalıştırılacaksa eylem "open" olmalıdır. Örnek bir ShellWxecute çağrısı şöyle olabilir:

#include <stdio.h>

#include <stdlib.h>

#include <Windows.h>

void ExitSys(LPCSTR lpszMsg, int status);

int main(void)

{

HINSTANCE hInstance;

hInstance = ShellExecute(NULL, "open", "x.txt", NULL, NULL, SW\_SHOWNORMAL);

if ((int)hInstance < 32)

ExitSys("ShellExecute", EXIT\_FAILURE);

return 0;

}

void ExitSys(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 Proseslerin Yaratılması**

UNIX/Linux sistemlerinde prosesler fork isimli bir POSIX fonksiyonla yaratılır. fork POSIX fonksiyonu Linux sistemlerinde doğrudan sys\_fork isimli sistem fonksiyonunu çağırmaktadır. Bu sistemlerde proses yaratmanın başka bir yolu da yoktur. (fork fonksiyonunun vfork isimli bir versiyonu da vardır fakat bunun kullanım gerekçesi hepten ortadan kalkmıştır.) Fonksiyonun prototipi şöyledir:

#include <unistd.h>

pid\_t fork(void);

fork bir prosesin özdeş bir kopyasından oluşturur. Yani yaratılan alt proses (child process) için yeni bir kontrol blok yaratılır. Üst prosesin kontrol bloğundaki bilgiler alt prosese kopyalanır. Alt prosesin sanal bellek alanı tamamen üst prosesten (parent process) kopyalanır. Böylece aynı koda ve data sahip özdeş fakat iki farklı iki proses oluşmuş olur.

UNIX/Linux sistemlerinde her prosesin bir proses id ideğeri vardır. Prosesin id değeri pross kontrol bloğuna erişmekte kullanılan bir handle gibi işlem görür. Prosesin id değeri tamsayısal bir değerdir ve pid\_t türü ile temsil edilir.

fork fonksiyonuna bir proses girer (yani üst proses), alt proses fork2un içerisinde yaratılır. Böylece fork'tan iki proses çıkar. Tabi ikisinin de kodu aynıdır. Fakat bağımsız proseslerdir.

fork fonksiyonundan üst proses alt prosesin id değeri ile, alt proses ise sıfır değeri ile çıkar. Böylece kodda üst proses ile alt proses birbirlerinden yarılabilir. fork başarısız da olabilir. Bu durumda -1 değerine geri döner. O halde tipik olarak fork uygulama kalıbı şöyledir:

pid\_t pid;

if ((pid = fork()) == -1) {

perror("fork");

exit(EXIT\_FAILURE);

}

if (pid != 0) { /\* parent process \*/

}

else { /\* child process \*/

}

Örneğin:

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

int main(void)

{

pid\_t pid;

if ((pid = fork()) == -1) {

perror("fork");

exit(EXIT\_FAILURE);

}

if (pid != 0) { /\* parent process \*/

printf("parent\n");

}

else { /\* child process \*/

printf("child\n");

}

printf("ends...\n");

return 0;

}

Peki aşağıdaki programda ekrana kaç tane "ends" yazısı çıkar?

#include <stdio.h>

#include <unistd.h>

int main(void)

{

int i;

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

fork();

printf("ends\n");

return 0;

}

Yanıt: 8 tane. Programda ne kadar proses yaratılmışsa o kadar "ends" yazısı çıkar. Burada toplam 8 prose yaratılmaktadır. Yukarıdaki programın eşdeğeri aslında aşağıdaki gibidir:

#include <stdio.h>

#include <unistd.h>

int main(void)

{

int i;

fork();

fork();

fork();

printf("ends\n");

return 0;

}

fork ile biz başka bir progrtamı çalıştıramayız. fork ancak bir prosesin özdeş kopyasını çalıştırır. Peki başka bir program dosyasını nasıl çalıştırırız? İşte bu exec fonksiyonlarıyla yapılmaktadır.

**exec Fonksiyonları**

exec POSIX fonksiyonları bir prosesin başka bir kodla çalışmasına devam etmesini sağlar. exec işlemiyle mevcut prosesin çalıştırdığı kod bellekten atılır. Onun yerine exec fonksiyonunda belirtilen dosya belleğe yüklenir ve o dosyadaki kod çalıştırılır. exec işlemleriyle prosesin kontrol bloğu değişmez. Yani prosesin id'si, yetkileri, çalışma dizini, açtığı dosyalar vs. hep öyle kalır. Proses hayatına başka bir program koduyla devam eder.

Aslında exec isimli bir fonksiyon yoktur. exec bir ailesnin ismidir. Bu ailede yoplam 7 fonksiyon vardır. Bu fonksiyonların yaptıkları şey aynı olmasına karşın yalnızca parametrik yapı yani arayüz farklıdır. Bu 7 fonksiyon şöyledir:

#include <unistd.h>

int execl(const char \*path, const char \*arg, ...);

int execlp(const char \*file, const char \*arg, ...);

int execle(const char \*path, const char \*arg, ..., char \* const envp[]);

int execv(const char \*path, char \*const argv[]);

int execvp(const char \*file, char \*const argv[]);

int execvpe(const char \*file, char \*const argv[], char \*const envp[]);

int execve(const char \*filename, char \*const argv[], char \*const envp[]);

Aslında asıl taban fonksyion execve fonksiyonudur. Yani yalnızca execve bir sistem fonksiyonudur. Diğer fonksiyonlar execve fonksiyonunu çağırmaktadır.

En çok kullanılan exec fonksiyonlarından biri execl fonksiyonudur. (Buradaki l list'ten geliyor.)

#include <unistd.h>

int execl(const char \*path, const char \*arg, ...);

Fonksiyonun birinci parametresi çalıştırılabilir (executable) dosyanın yol ifadesini alır. Diğer parametreler programın komut satırı argümanlarını belirtir. Listenin sonunun NULL adresle bitirilmesi gerekir. execl başarılıysa geri dönmez. Başarısızsa -1 değerine geri döner.

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

int main(void)

{

printf("begins...\n");

if (execl("/bin/ls", "/bin/ls", "-l", (char \*)NULL) < 0) {

perror("execl");

exit(EXIT\_FAILURE);

}

printf("ends...\n"); /\* unreachable code \*/

return 0;

}

Fonksiyonun son parametresi NULL girilirken tür dönüştürmesi yapılmalıdır. Yani aşağıdaki çağırma biçimi sorunludur:

if (execl("/bin/ls", "/bin/ls", "-l", NULL) < 0) {

perror("execl");

exit(EXIT\_FAILURE);

}

Çünkü "..." parametresine karşılık NULL argümanı girilirse bu NULL makrosunun nasıl define edilmiş olduğuna bağlı olarak sorunlar çıkabilir. Bilindiği gibi NULL makrosu iki biçimde define edilmiş olabilir:

#define NULL 0

ya da,

#define NULL ((void \*)0)

İşte eğer NULL birinci biçimde olduğu gibi define edilmişse buna karşı gelen parametre gösterici olmadığı için derleyici argüman stack'e 4 byte gönderir. Oysa 64 bit sistemlerde göstericiler 8 byte olduğu için sorun çıkar. Fakat biz bu son parametreyi (char \*)NULL biçiminde girersek bu durumda her halukarda stack'e NULL adres atılacaktır.

Diğer bir çok kullanılan exec fonksiyonu da execv fonksiyonudur (Buradaki 'v' "vector" sözcüğünden kısaltmadır.) execv fonksiyonunda komut satırı argümanları bir gösterici dizisine yerleştirilip geçirilir. Fonksiyonun prototipi şöyledir:

#include <unistd.h>

int execv(const char \*path, char \*const argv[]);

Örneğin:

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

int main(void)

{

char \*args[] = { "/bin/ls", "-l", NULL };

if (execv("/bin/ls", args) < 0) {

perror("execv");

exit(EXIT\_FAILURE);

}

return 0;

}

Söz konusu gösterici dizisinin NULL adresle sonlandırılmış olması gerekir. (Burada NULL makrosununun dönüştürme yapılması gerekmemektedir.)

Örneğin bir program kendi komut satırı argümanlarıyla aldığı başka programı çalıştırsın. Örneğin programamız sample olsun o da ls'yi çalıştırsın:

./sample /bin/ls -l

execv fonksiyonu bunun çok uygundur:

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

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

{

char \*args[] = { "/bin/ls", "-l", NULL };

if (execv(argv[1], &argv[1]) < 0) {

perror("execv");

exit(EXIT\_FAILURE);

}

return 0;

}

exec fonksiyonunun p'li versiyonları çalıştırılacak dosyayı exec uygulayan prosesin PATH çevre değişkeni ile belirtilen dizinlerde arar. PATH çevre değişkeni ':' karakterleriyle ayrılan dizinlerden oluşan bir yazı biçimindedir. Örneğin:

echo $PATH

/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:/usr/local/games

Ancak exec fonksiyonunun p'li vers,yonlarının PATH çevre değişkenine bakması için yol ifadesinde hiç bir '/' karakterinin bulunmaması gerekir. Aksi halde bu fonksiyonlar PATH çevre değişkenine bakmazlar ve tamamen p'siz versiyonlar gibi yol ifadesinin mutlak ya da göreli olması durumuna göre aramalarını yaparlar. Örneğin:

execlp("a/sample", "a/sample", (char \*)NULL);

Burada execlp PATH çevre değişkenine bakmamaktadır. sample dosyasını prosesin çalışma dizininin altındaki a dizininin içerisinde arar. Ancak yol ifadesinde hiç '/ ' karakteri yoksa bu p'li versiyonlar yalnızca PATH çevre değişkeni ile belirtilen dizinlerde arama yaparlar. Ayrıca çalışma dizinine bakmazlar. Örneğin:

execlp("sample", "a/sample", (char \*)NULL);

Burada sample programı bulunulan dizinde olsa bile oraya bakılmamaktadır. Yalnızca PATH çevre değişkeni ile belirtilen dizinlere bakılmaktadır.

shell programları exec fonksiyonlarının p'li versiyonlarını kullanarak programları çalıştırmaktadır. İşte biz de bu yüzden programları çalıştırırken "./sample" biçiminde isimleri belirtiriz. Eğer böyle bir program "sample" biçiminde çalıştırılmaya çalışılırsa exec fonksiyonlarının p'li versiyonları onları yalnızca PATH çevre değişkeni belirtilen dizinlerde arayacak, dolayısıyla bulamayacaktır. "./sample" yol ifadesi aslında "sample" ile aynı anlama geliyor olsa bile işin içerisinde bir '/' karakteri karıştırılmıştır.

Ayrıca exec fonksiyonlarının bir de e'li versiyonları vardır. (Burada 'e' "environment" sözcüğünden gelmektedir.) Bu versiyonlar program çalıştırılırken çevre eğişken takımının değiştirilmesine yol açmaktadır.

**fork ve exec Fonksiyonlarının Bir Arada Kullanılması**

Bilindiği yalnızca fork fonksiyonu bit prosesin özdeş yeni bir kopyasını oluşturur. Yeni prosesin çalıştırdığı kod aynı kod olacaktır. Yalnız başına exec fonksiyonları ise proses yaratmayıp mevcut prosesin başka bir kodla çalışmasına devam etmesini sağlar. Peki hem bizim programımız çalışmaya devam ederken hem de başka bir programı naısl çalıştırabiliriz? İşte bunun için fork ve exec birlijte kullanılmalıaıdır. Önce bir kez fork yapılır. alt proseste exec uygulanır. Tipik kalıp şöyledir:

if ((pid = fork()) < 0) {

perror("fork");

exit(EXIT\_FAILURE);

}

if (pid == 0)

if (execl(...) < 0) {

perror("execl");

exit(EXIT\_FAILURE);

}

/\* parent devam eder, else'e gerek yok \*/

Ya da daha kompakt yazım uygulanabilir:

if ((pid = fork()) < 0) {

perror("fork");

exit(EXIT\_FAILURE);

}

if (pid == 0 && execl(...) < 0) {

perror("execl");

exit(EXIT\_FAILURE);

}

Örneğin:

#include <stdlib.h>

#include <unistd.h>

int main(void)

{

pid\_t pid;

printf("parent begins\n");

if ((pid = fork()) < 0) {

perror("fork");

exit(EXIT\_FAILURE);

}

if (pid == 0 && execlp("ls", "ls", "-l", (char \*)NULL) < 0) {

perror("execlp");

exit(EXIT\_FAILURE);

}

printf("parent ends..\n");

return 0;

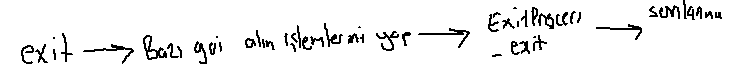
}

Windows sistemleriyle UNIX/Linux sistemlerinin proses yaratma bakımından farklı olduğuna dikkat ediniz. Windows'ta fork ve exec benzeri bir mekanizma yoktur. Orada CreateProcess adeta buradaki fork ve exec'in birliktge kullanılmasına benzemektedir. Yani Windows sisyemlerinde zaten yeni bir proses bir kodla başlatılır.

**Proseslerin Sonlandıroması ve Exit Kodları**

Prosesler işletim sisteminin sistem fonksiyonlarıyla sonlandırılır. Windows sistemlerine ExitProcess API fonksiyonu, POSIX sistemlerinde \_exit (Linux'ta sys\_exit'i çağırır) fonksiyonu prosesi sonlandırmakta kullanılır. C'nin standart exit fonksiyonu ise dolaylı olarak bu fonksiyonları çağırmaktadır.

C'nin standart exit fonksiyonu işletim sisteminin sistem fonksiyonlarını çağırmadan önce standart kütüphaneye ilişkin çeşitli sonlandırma işlemlerini yapmaktadır. Dolayısıyla C'de çalışıyorsak programı işletim sisteminin API fonksiyonları ile değil exit standart C fonkisyonuyla sonlandırmamız daha uygun olur. Örneğin biz bir dosya açıp içine birşeyler yazmış olalım. Bu durumda prosesi işletimn sistem fonksiyonlarıyla sonlandırmamız uygun olmaz. Çünkü stdio fonksiyonlarının oluşturduğu tampon fclose işlemi sırasında fklush edilmektedir. Oysa işletim sisteminin sistem fonksiyonlarının bu tampondan haberi yoktur. Onlar dosyayı işletim sistemi düzeyinde kapatırlar. Fakat tabi eğer biz standart C kütüphanesi ile ilgili önemli işlemler yapmamışsak prosesi doğrudan işletim sisteminin sistem fonksiyonlarıyla da sonlandırabiliriz.



Bir proses sonlandığında işletim sistemine "exit kodu" denilen bir kod iletilir. Prosesin exit kodu C'de exit fonksiyonuna verilen argümandır. main fonksiyonunda return uygulanırsa bu da aynı anlama gelmektedir. C standartlarına göre main fonksiyonun geri dönüş değeri (eğer geri dönerse) exit fonksiyonuna argüman yapılmaktadır. Yani C standartlarına göre bir C programı şöyle çalıştırılır:

exit(main(...));

Ayrıca C'de (main fonksiyonu için istisna olarak) eğer main'de return uygulanmazsa sanki ana bloğun sonunda 0 ile geri dönülmüş gibi işlem uygulanır. Yani:

int main(void)

{

...

}

ile,

int main(void)

{

...

return 0;

}

aynı anlamdadır.

Peki exit kodu ne işe yarar? İşletim sistemi için exit kodunun kaç olduğunun bir önemi yoktur. İşletim sistemi bu kodu alır, saklar. Eğer üst proeses isterse ona verir. Böylece üst proses alt prosesin nasıl sonlandığını bilmiş olur. Duruma göre birşeyler yapabilir. Geleneksel olarak başarılı sonlanmalarda sıfır değeri, başarız sonlanmalarda sıfır dışı değerler tercih edilmektedir.

**Anahtar Notlar:** Shell programları hem bir komut satırı sunarlar hem de bunlar bir yorumlayıcı (interpreter) da içermektedir. Yani her shell ortamının bir "shell script" dili vardır. Bugün için en çok kullanılan shell "bash (bourne again shell)" isimli shell'dir. bash'in de bir script dili vardır. Bu dilin öğrenilmesi zor değildir. Bir shell script bir kaynak dosyadır ve genellikle uzantısı .sh biçimindedir. Shell script dosyaları shell programları tarafından yorumlanarak çalıştırılırlar. Onları çalıştırmanın iki yolu vardır. Birincisi çalıştırılacak script dosyasını shell programına komut satırı argümanı olarak vermektir. Bu durumda shell interaktif modda çalışmayacak, ilgili dosyayı çalıştırıp sonlanacaktır. Örneğin:

/bin/bash sample.sh

İkinci yöntemde script dosyasına sanki çalıştırılabilir bir dosya gibi 'x' hakkı vermektir:

chmod +x sample.sh

Bu durumda dosyanın ilk satırının yorumlayıcı programın yol ifadesini içermesi gerekmektedir.

/\* sample.sh Dosyası \*/

#! /bin/bash

...

exec fonksiyonları böyle bir dosyayı yüklerken onun gerçek bir çalıştırılabilir dosya olmadığını anlar (örneğin bunların ELF başlığı yoktur). Onun birinci satırına bakar ve orada belirtilen programı çalıştırır. Bu programa söz konusu dosyayı komut satırı argümanı olarak geçrir. Yani yine çalışma aşağıdaki gibi olmuç olacaktır:

/bin/bash sample.sh

Tabi biz bu yöntemle aslında her türlü dosyayı çalıştırabiliriz. Örneğin perl, python dosyalarını vs.

Proseslerin exit kodları nasıl alınabilir? Tabi bunun için önce prosesin sonlanması gerekir. Windows'ta GetExitCodeProcess fonksiyonu prosesin exit kodunu almakta kullanılır:

BOOL WINAPI GetExitCodeProcess(

\_\_in HANDLE hProcess,

\_\_out LPDWORD lpExitCode

);

Fonksiyonun birinci parametresi exit kodunun alınacağı prosesin handle değeridir. İkinci parametresi exit kodunun yerleştirileceği DWORD türden nesnenin adresini alır. Eğer söz konusu proses henüz sonlanmamışsa exit kodu olarak 259 (STILL\_ACTIVE) elde edilir.

#include <stdio.h>

#include <stdlib.h>

#include <Windows.h>

void ExitSys(LPCSTR lpszMsg, int status);

int main(void)

{

char cmdLine[] = "..\\TestApp\\Debug\\TestApp.exe";

STARTUPINFO si = { sizeof(STARTUPINFO) };

PROCESS\_INFORMATION pi;

DWORD dwExitCode;

if (!CreateProcess(NULL, cmdLine, NULL, NULL, FALSE, 0, NULL, NULL, &si, &pi))

ExitSys("CreateProcess", EXIT\_FAILURE);

WaitForSingleObject(pi.hProcess, INFINITE);

if (!GetExitCodeProcess(pi.hProcess, &dwExitCode))

ExitSys("GetExitCodeProcess", EXIT\_FAILURE);

if (dwExitCode == STILL\_ACTIVE) {

printf("Program is still running...\n");

exit(EXIT\_SUCCESS);

}

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

return 0;

}

void ExitSys(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 exit kodu waitxxx isimli fonksiyonlarla elde edilmektedir. wait fonksiyonu yetersiz kaldığı için waitpid isimli fonksiyon da daha sonra POSIX sistemlerine dahil edilmiştir. wait fonksiyonun prototipi şöyledir:

#include <sys/wait.h>

pid\_t wait(int \*status);

wait fonksiyonu hem prosesin sonlanmasını bekler hem de sonlandığında onun exit kodunu alır.

wait ilk sonlanan alt prosesin exit kodunu alır ve sonlanan alt prosesin proses id değeri ile geri döner. (Tabi zaten sonlanmış bir alt proses varsa hemen onun exit kodunu alır, bekleme yapmaz.) Fonksiyon sonlanan prosesin exit kod bilgisini aparametresiyle aldığı int türden nesneye yerleştirir. Başarısızlık durumunda ise -1 değerine geri döner.

Aslında fonksiyonun parametreye yerleştirdiği değer yalnızca exit kodu değildir. Bu int nesnesnin bazı bitleri prosesin neden sonlandığına yönelik bilgi de içermektedir. İşte hangi bitlerin hangi amaçla kullanıldığı sistemden sisteme değişebildiği için standart birkaç makro bulundurulmuştur. WIFEXITED(status) makrosu eğer proses normal bir biçimde sonlanmışsa sıfır dışı değer verir. WEXITSTATUS(statıs) makrosu ise bize exit kodunu verir. Eğer fonksiyonun parametresi NULL geçilirse exit kod elde edilmez. Fakat bekleme yapılır. (Yani wait fonksiyonu Windows'taki WaitForSingleObject ve GetExitCodeProcess fonksiyonlarının birleşimi gibidir.) Örneğin:

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <sys/wait.h>

int main(void)

{

pid\_t pid;

int status;

printf("parent begins\n");

if ((pid = fork()) < 0) {

perror("fork");

exit(EXIT\_FAILURE);

}

if (pid == 0 && execl("app", "app", (char \*)NULL) < 0) {

perror("execl");

exit(EXIT\_FAILURE);

}

if (wait(&status) < 0) {

perror("wait");

exit(EXIT\_FAILURE);

}

if (WIFEXITED(status))

printf("%d\n", WEXITSTATUS(status));

else

printf("Process doesn't finish normally\n");

return 0;

}

**UNIX/Linux Sistemlerinde Hortlak (Zombie) Proses Kavramı**

UNIX/Linux sistemlerinde prosesin exit kodu Proses Kontrol Bloğuna yazılır ve buradan alınır. Bu nedenle bir proses bittiği halde henüz onun exit kodunu wait fonksiyonlarıyla üst proses almamışsa prosesin kontrol bloğu işletim sistemi tarafından serbest bırakılmaz. İşte sonlandığı halde exit kodu alınmamasından dolayı kontrol bloğu boşaltılmamış proseslere hortlak prosesler denilmektedir. wait fonksiyonlarıyla exit kod alındığında hortlaklık ortadan kalkar.

UNIX/Linux sistemlerinde eğer üst proses alt prosesten daha önce sonlanmışsa bu durumdaki alt proseslere öksüz (orphan) prosesler denilmektedir. Sistem öksüz duruma düşen prosese init ptosesini (id'si 1 olan proses) üst proses olarak atar. init de başarılı bir biçimde alt proses sonlandığında onun exit kodunu alarak onu zombi olmaktan kurtarır. O halde zombi durumu yalnızca üst proses yaşarken fakat alt proses sonlanmışsa oluşur. eğer alt proses sonlandoktan sonra üst proses de sonlanırsa yine init alt prosesin exit kodunu almaktadır. Peki hortlaklığın önemi nedir? Bazı programlar çok uzun süre (örneğin bir seneden fazla) çalışabilmektedir. Bunlar sürekli hortlak proses üretirse sistem kaynaklarını tüketebilir. Ayrıca hortlak prosesin id değeri de başka bir prosese verilememektedir. wait fonksiyonlarının dışında otomatik hortlaklığı engellemenin başka yolları da vardır.

**Proseslerin Çevre Değişkenleri (Environment Variables)**

Modern işletim sistemlerinin hemen hepsi çevre değişkeni kavramını içermektedir. Çevre değişkeni aslında bir anahtara karşı bir değerin getirildiği bir çifti belirtir. Anahtar da değer de birer yazıdır. Çevre değişkeni denildiğinde genellikle bu anahtardan bahsedilir. Örneğin anahtar (yani çevre değişkeni) "City" olsun. Buna karşı gelen değer "New York" olabilir. Ya da anahtar "Count" olsun. Buna karşı gelen değer "123" olabilir. Böylece her prosesin bir çevre değişken takımı vardır. Her ne kadar böyle bir veri yapısı tamamen manuel olarak oluşturulabilse de çevre değişkenlerinin organizasyonu ve kullanımı çekirdek düzeyinde yapılmaktadır ve taban bir kavramdır. Yani prosesin hiçbir şeyi yokken çevre değişkenlerini kullanabilir. Ayrıca bazı çevre değişkenleri bizzat çekirdek tarafından da kullanılmaktadır (PATH gibi).

Programcı çevre değişkenleriyle ilgili dört şeyi yapabilir:

1) Yeni çevre değişkenini değerle birlikte prosesin çevre değişken listesine ekleyebilir.

2) Bir çevre değişkenini prosesin çevre değişken listesinden silebilir.

3) Bir çevre değişkenine karşılık gelen değeri elde edebilir

4) Prosesin tüm çevre değişken listesini elde edebilir.

Eçvre değişkenleri prosese özgüdür. Yani her prosesin ayrı bir çevre değişken listesi vardır. Genel olarak çevre değişkenleri pek çok işletim sisteminde üst prosesten alt prosese aktarılmaktadır. Yani üst proses bir proses yarattığında üst prosesin çevre değişken listesi alt prosese aktarılmaktadır (tabi bu bir kopyalama gibidir. Bu kopyalamadn asonra üst ve alt proseslerin çevre değişken listesi bağımsız olur.)

Peki prosesin çevre değişken listesi nerede tutulmaktadır? Bazı sistemlerde çevre dğeişkenleri Proses Kontrol Bloğunda tutulurken, modern sistemlerin çoğunda (örneğin Windows ve Linux sistemlerinde) prosesin çevre değişkenleri prosesin bellek alanında tutulmaktadır.

Prosesin eçvre değişkeni verildiğinde onun değerini bize veren getenv isimli satndar bir C fonksiyonu vardır:

#include <stdlib.h>

char \*getenv(const char \*name);

Fonksiyon parametre olarak çevre değişkeninin ismini alır, geri dönüş değeri olarak onun değerinin bulunduğu dizinin adresini verir. Bu adres statik olarak tahsis edilmiştir. Fonksiyon başarısız olabilir. Bu durumda NULL adrese geri döner. Örneğin:

#include <stdio.h>

#include <stdlib.h>

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

{

char \*value;

if (argc != 2) {

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

exit(EXIT\_FAILURE);

}

if ((value = getenv(argv[1])) == NULL) {

fprintf(stderr, "environment variable not found!\n");

exit(EXIT\_FAILURE);

}

puts(value);

return 0;

}

Ayrıca getenv standart C fonksiyonunun dışında Windows sistemlerinde GetEnvironmentVariable isimli API fonksyionu aynı işlemi yapmaktadır (zaten Windows altında getenv standart C fonksiyonuı bunu çağırıyor):

DWORD WINAPI GetEnvironmentVariable(

\_\_in LPCTSTR lpName,

\_\_out LPTSTR lpBuffer,

\_\_in DWORD nSize

);

Prosesin çevre değişken listesine yeni bir ekeleme yapmak için kullanılabilecek bir standart C fonksiyonu yoktur. Bunun için UNIX/Linux sistemlerinde setenv ve putenv POSIX fonksyonları Windows sistemlerinde SetEnvironmentVariable API fonksiyonu kullanılmaktadır. setenv fonksiyonunun prototipi şöyledir:

#include <stdlib.h>

int setenv(const char \*name, const char \*value, int overwrite);

Fonksiyonun birinci parametresi çevre değişkenini, ikinci parametresi ise bunun değerini belirtir. Üçüncü parametre sıfır ya da sıfır dışı bir değer olarak girilir. Eğer sıfır girilirse bu eçvre değişkeni zaten varsa fonksiyon balarısız olur. Eğer sıfır dışı olarak girilirse eski değer silinir. Yerine yenisi gelir. Fonksiyon başarı durumunda sıfır, başarısızlık durumunda -1 değerine geri döner. Örneğin:

#include <stdio.h>

#include <stdlib.h>

int main(void)

{

char \*value;

if (setenv("CITY", "Istanbul", 1) < 0) {

perror("setenv");

exit(EXIT\_FAILURE);

}

if ((value = getenv("CITY")) == NULL) {

perror("getenv");

exit(EXIT\_FAILURE);

}

puts(value);

return 0;

}

putenv fonksiyonunun prototipi de şöyledir:

#include <stdlib.h>

int putenv(char \*string)

Fonksiyonun parametresi "anahtar=değer" biçiminde tek bir yazı alır. İlgili çevre değişkeniş varsa üstüne yazar. Başarı durumunda sıfır, başarısızlık durumunda -1 değerine geri döner.

#include <stdio.h>

#include <stdlib.h>

int main(void)

{

char \*value;

if (putenv("CITY=Istanbul") < 0) {

perror("setenv");

exit(EXIT\_FAILURE);

}

if ((value = getenv("CITY")) == NULL) {

perror("getenv");

exit(EXIT\_FAILURE);

}

puts(value);

return 0;

}

Microsoft da putenv POSIX fonksiyonunu \_putenv ismiyle desteklemektedir. (Tabi bu da aslında SetEnvironmentVariable API fonksiyonu çağırıyor).

Windows'ta SetEnvironmentVariable API fonksiyonunun prototipi de şöyledir:

BOOL WINAPI SetEnvironmentVariable(

\_\_in LPCTSTR lpName,

\_\_in LPCTSTR lpValue

);

Fonksiyonun birinci parametresi çevre değişkenini, ikinci parametresi bunun değerini belirtir. Çevre zaten varsa silinerek yeni değer yerleştirilir.

Bir çevre değişkenini silmek bazı sistemlerde mümkünken bazı sistemlerde mümkün değildir.

Bir prosesin tüm çevre değişkenleri nasıl elde edilir? Windows sistemlerinde GetEnvironmentString isimli API fonksiyonu bize prosesin tüm çevre değişkenlerini ve onların değerlerini verir:

LPTCH WINAPI GetEnvironmentStrings(void);

Fonksiyonun geri dönüş değeri charf türden bir göstericidir. Verilen yazı şu biçimdedir:

K1=V1\0K2=V2\0K3=V3\0K4=V4\0\0

Örneğin:

#include <stdio.h>

#include <windows.h>

int main(void)

{

char \*env;

env = GetEnvironmentStrings();

while (\*env != '\0') {

puts(env);

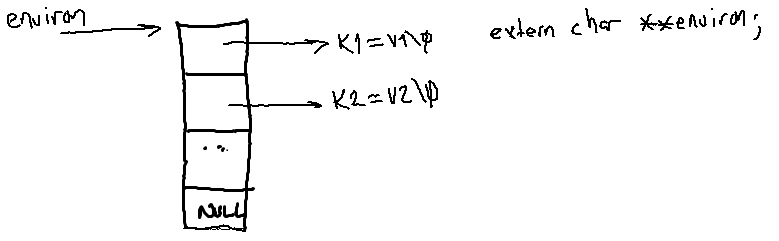
env += strlen(env) + 1;

}

return 0;

}

UNIX/Linux sistemlerinde kütüphane içerisinde tanımlanmış char \*\* türünden environ isimli bir global göstericiyi gösteren gösterici vardır. Bunun extern bildirimi maalesef hiçbir başlık dosyasında yoktur. Programcının extern bildirimini kendisinin yapması gerekmektedir. Bu göstericiyi gösteren gösterici anahtra değer çiftlerini tutan gösterici dizisinin başlangıç adresini gösterir. Dizinin sonunda NULL adres vardır:



Örneğin:

#include <stdio.h>

extern char \*\*environ;

int main(void)

{

int i;

for (i = 0; environ[i] != NULL; ++i)

puts(environ[i]);

return 0;

}

Windows sistemlerinde komut satırında "set" komutu, UNIX/Linux sistemlerinde ise "env" komutu shell prosesinin (cmd.exe ya da /bin/bash) o andaki çevre değişken listesini görüntüler.

Peki çevre değişkenleri nasıl oluşturulmaktadır? UNIX/Linux sistemlerinde fork işlemi sırasında üst prosesin çevre değişkenleri alt prosese aktarılmaktadır. exec sırasında programcı isterse çalıştıracağı programın yeni bir çevre değişken listesi ile çalıştırılmasını sağlayabilir. Yani exec sırasında prosesin çevre değişken listesi atılarak yerine yenisi oluşturulabilmektedir. Bunun için exec fonksiyonlarının e'li versiyonları kullanılır.

#include <unistd.h>

int execle(const char \*path, const char \*arg, ..., char \* const envp[]);

int execvpe(const char \*file, char \*const argv[], char \*const envp[]);

int execve(const char \*filename, char \*const argv[], char \*const envp[]);

Örneğin execle fonksiyonunu şöyle kullanabiliriz:

/\* sample.c \*/

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

int main(void)

{

pid\_t pid;

char \*env[] = { "City=Istanbul", "Name=Ali" };

if ((pid = fork()) < 0) {

perror("fork");

exit(EXIT\_FAILURE);

}

if (pid == 0 && execle("app", "app", (char \*)NULL, env) < 0) {

perror("execle");

exit(EXIT\_FAILURE);

}

wait(NULL);

return 0;

}

/\* app.c \*/

#include <stdio.h>

extern char \*\*environ;

int main(void)

{

int i;

for (i = 0; environ[i] != NULL; ++i)

puts(environ[i]);

return 0;

}

O halde biz UNIX/Linux sistemlerinde shell üzerinden bir program çalıştırdığımızda shell'n çevre değişkenleri bizim programamıza aktarlır. Peki shell'in çevre değişkenleri nasıl oluşturulmuştur? Bu konu ileride ele alınacaktır.

Windows'ta CreateProcess sırasında CreateProcess fonksiyonunun 7'nci parametresi NULL geçilirse yaratılan prosese üst prosesin çevre değişkenleri aktarılır. Değilse, bu 7'nci parametre verilen liste aktarılır:

BOOL WINAPI CreateProcess(

\_\_in LPCTSTR *lpApplicationName*,

\_\_in\_out LPTSTR *lpCommandLine*,

\_\_in LPSECURITY\_ATTRIBUTES *lpProcessAttributes*,

\_\_in LPSECURITY\_ATTRIBUTES *lpThreadAttributes*,

\_\_in BOOL *bInheritHandles*,

\_\_in DWORD *dwCreationFlags*,

\_\_in LPVOID *lpEnvironment*,

\_\_in LPCTSTR *lpCurrentDirectory*,

\_\_in LPSTARTUPINFO *lpStartupInfo*,

\_\_out LPPROCESS\_INFORMATION *lpProcessInformation*

);

UNIX/Linux sistemlerinde shell üzerinde,

export K=V

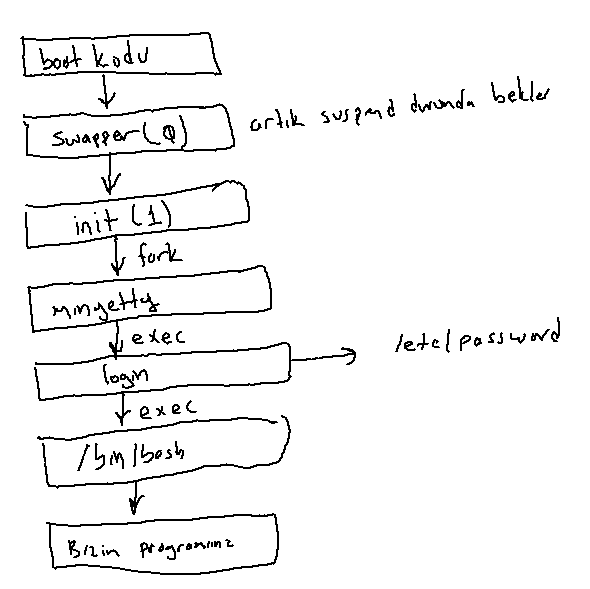
yazılır ENTER tuşuna basılırsa K çevre değişkeni yaratılır ve buna V değeri atanır. (Eğer export komutu belirtilmezse bu değişken shell dilinin değişkeni olur. Fakat shell bunu prosesin çevre değişken listesine eklemez). Shell üzerinden $<çevre değişken ismi> yazılırsa shell bize onun değerini verir. Örneğin:

export Name=Ali

echo $Name

O halde biz shell programına bir çevre değişkeni ekleyip kendi programımızı çalıştırırsak o değişken bizim programımıza da geçecektir.

Peki shell'deki çevre değişkenleri nereden gelmektedir? Tipik bir UNIX/Linux sistemi boot işleminden sonra aşağıdaki gibi proses yaratmaktadır:



Bu hikayedeki çeşitli prosesler birtakım script dosyalarına ve konfigürasyon dosyalarına bakarak birtakım şeyleri ayarlamaktadır. Sistem admin de bu dosyalara uygun şeyleri yazarak açılış sırasında sistemi konfigüre edebilir. Bu bilgiler kernel'a dahil değildir. (Yani örneğin init programı ya da login programı kernel kodları içerisinde yoktur. Bunlar başka proje grupları tarafından yazılmışlardır.)

Peki sistem boıot edilirken hangi proses hangi scrip dosyalarına bakmaktadır? İşte bu da çeşitli dağıtımlara ve dağıtımlarda kullanılan paketlere göre değişebilmektedir.

Linux sistemlerinde pek çok paketin alternatif gerçekleştirimleri vardır. Örneğin init programı için poüler iki paket kullanılmaktadır: sysvinit ve upstart. Örneğin Suse gibi sistemler klasik sysvinit paketini kullanırken, Ubuntu ve türevleri (örneğin Mint gibi) upstart kullanmaktadır. sysvinit paketindeki init programı /etc/inittab dosyasına, upstart paketindeki init rogramı /etc/init/init.conf dosyasına bakmaktadır. Bu dosyaların formatlarına ilişkin bilgi man sayfalarından elde edilebilir.

login prosesi user name ve password sorgulamasını yapar. Password'ü /etc/passwd dosyasına bakarak doğrular ve burada belirtilen programı çalıştırır. /etc/passwd dosyası sistemdeki tüm kullanıcıların kayıtlarının bulunduğu bir text dosyadır. Bu dosyada her bir satır ':' lerle ayrılmış alanlardan oluşur. Örneğin:

csd:x:1000:1000:CSD,,,:/home/csd:/bin/bash

student:x:1001:1000:CSD,,,:/home/student:/bin/bash

usermetrics:x:115:124:User Metrics:/var/lib/usermetrics:/bin/false

clickpkg:x:116:125::/nonexistent:/bin/false

Görüldüğü gibi her kullanıcı için 7 alan vardır. İlk alanda kullanıcının ismi bulunur. İkinci alanda password bulunmaktadır. Burada 'x' karakteri varsa bu, şifrenin /etc/shadow dosyasında olduğunu gösterir. Sonra user id, sonra da grup id değerleri gelmektedir. Son alanda login başarılıysa çalıştırılacak program bulunur. Sondan bir öncekinde o program çalıştırıldığında başlangıç çalıma dizini bulunmaktadır.

login prosesi bazı çevre değişkenlerini prosese dahil etmektedir. Örneğin USER, HOME, PATH, SHELL gibi.

login prgramı shell programını çalıştırır (tipik olarak /bin/bash). Shell tarafından da pek çok çevre değişkeni listeye eklenmektedir. Örneğin: PWD, LINES, COLUMNS, LANG gibi...

Peki biz bir çevre değişkeninin kalıcılığını nasıl sağlayabiliriz? İşletim sistemi başlatılırken çalıştırılan çeşitli script dosyalarının içerisine çevre değişken yaratım komıtlarını eklersek sistem açıldığında otomatik olarak o çevre değişkenleri proses ağacına dahil edilmiş olur. Örneğin çevre değişkenleri için tipik olarak /bin/bash shell'inin çalıştırdığı script dosyaları tercih edilmektedir. Tipik olarak bash iki script dosyasını çalıştırmaktadır. Bunlar: /etc/profile ve ~/.bash\_profile dosyalarıdır. /etc/profile dosyası her kullanıcıyı ilgilendirecek bir dosyadır ve her shell çalıştırıldığında çalıştırılır. Home dizinindeki .bash\_profile dosyası yalnızca ilgili kullanıcı shell'i çalıştırdığında çalıştırılır. Örneğin ~/.bash\_profile dosyasına aşağıdaki satırı yerleştirerek PATH çevre değişkenine ekleme yapabiliriz:

export PATH=$PATH:/home/csd/Study

Windows sistemlerinde de genel mantık aynıdır. Windows'ta masaüstünü "EXPLORER.EXE" isimli proses temsil eder. Windows'ta da bir loin prosesi vardır. Bu proses login işlemini yapar. Explorer.exe prosesi bu proses tarafından çalıştırılır. Biz "Denetim Masası/Sistem/Gelişmiş Sistem Ayarları/Ortam Değişkenleri" menüsünden yeni çevre değişkenleri ekleyip, mevcut olanları değiştirebiliriz. Tabi bu değişiklik o anda çalışmakta olan proseslere yansımaz.

Windows'ta komut satırında (cmd.exe) çevre değişkeni girmek için SET komutu kullanılır. Örneğin:

SET City=Ankara

Yine komut satırında çevre değişkenlerinin anahtarı verildiğinde değeri elde etmek için %Key% ifadesi kullanılır. Örneğin:

SET PATH=%PATH%;c:\Study

**Anahtar Notlar:** Windows'ta çevre değişkenlerinin büyük harf küçük harf duyarlılıkları yoktur. Fakat UNIX/Linux sistemlerinde vardır.

**Çevre Değişkenlerine Neden Gereksinim Duyulmaktadır?**

Çevre değişklenlerine işletim sistemi düzeyindeki global değişkenler gibi bakılabilir. Programlar birtakım dosyaları vs. bazı çevre değikenlerinin belirttiği yerde arayabilirler. Örneğin bir veritabanı dosyası DATABASE isimli bir çevre değişkeninin belirttiği dizinde aranabilir. Bu durumda programcı önce getenv fonksiyonuyla bu çevre dğeiikeninin değerini alır ve dosyayı o dizinde araybilir. Veritabanı dosyası başka dizine yerleştirmek istenirsetek yapılacak şey bu çevre değişkeninin değerini değiştirmektir. Örneğin pek çok C derleyicisi <...> biçiminde include edilmiş dosyaları INCLUDE isimli bir çevre değişkenin belirttiği dizinde aramaktadır. Eğer bu eçvre değişkeni set edilmemişse default bir dizin kullanılmaktadır. Ya da örneğin Java'da çeşitli jar dosyaları ve derlenmiş dosyalar CLASSPATH isimli bir çevre değişkenine bağlı olarak aranmaktadır. Bazı çevre değişkenlerini doğrudan işletim sistemi kullanmaktadır. (Örneğin PATH gibi, LD\_LIBRARY\_PATH gibi).

Bazı çevre değişkenleri işletim sisteminin içinde bulunduğu durum hakkında bilgi veriyor olabilir. Örneğin Windows hangi dizine yüklenmiştir? O anda kullanılan yerel ayarlar nelerdir? Kullanıcının ismi nedir? Home dizini hangisidir vs.

Çevre değişkenleri proseslerarası haberleşmede de bazen kullanılabilmektedir. Örneğin, üst proses belli bir çevre değişkenini set ederek alt prosesi çalıştırır. Alt proses böylece o çevre değişkeninden bilgi alabilir. Böylesi basit bilgi aktarımı için herhangi bir proseslerarası haberleşme yönteminin kullanılmasına gerek yoktur.

**Anahtar Notlar:** UNIX/Linux sistemlerinde (Windows'ta da böyle) shell programları temelde iki modda çalıştırılmaktadır: Interaktif mod ve Interaktif olmayan (yani tek komutluk) mod. İnteraktif modda bir prompt çıkar. Kullanıcı komut girer, o komut çalışır yeniden prompt'a düşülür. Taki exit ya da logout komutu uygulanana kadar. Halbuki interaktif olmayan modda tek bir komut çalıştırılıp shell'den çıkılır. Örneğin system isimli standart C fonksiyonu shell'i böyle çalıştırmaktadır. Tek komutluk çalıştırma için /bin/bash'te (diğer shell'lerde de aynı) -c seçeneği kullanılmaktadır. Örneğin:

/bin/bash -c ls

Bıurada bash ls komutunu çalıştırıp çıkacaktır.

**Proseslerarası Haberleşme (Interprocess Communication - IPC)**

Yukarıda açıklandığı gibi modern sistemlerde sayfa tabloları yoluyla prosesler arasında tam bir izaolasyon sağlanmıştır. Her proses sanki fiziksel belleğe tek başına yüklenip çalışıyormuş gibi bir illüzyon oluşturulur. Bir prosesten diğerine n byte gönderip alma sürecine "Proseslerarası Haberleşme" denilmektedir. Proseslerarası haberleşme kabaca ikiye ayrılır:



Aynı makinanın prosesleri arasındaki haberleşmelerde kullanılan tipik teknikler şunlardır:

- Borular (Pipes)

- Paylaşılan Bellek Alanları (Shared Memory)

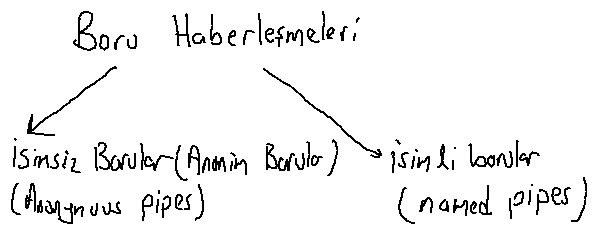
- Mesaj Kuyrukları (Message Queue)

Ağ altında farklı makinaların prosesleri arasında haberleşmede bir protokol de devreye girer. Bu amaçla oluşturulmuş değişik protokoller vardır. Bugün en yaygın kullanılan protokol Internet'in de kullandığı IP protokol ailesidir. Kursumuzda IP protokol ailesi ile haberleşme ele alınacaktır.

**Anahtar Notlar:** Raspberry Pi'ın 2 numaralı modeli oldukça hızlanfırılmıştır (4 core). Microsoft da Windows 10'un Raspberry Pi versiyonunu oluşturmuştur. Bu tür çok sayıda embedded Linux kitleri bulunmaktadır. Ancak Raspberry Pi oldukça standartlaşmış durumdadır ve önemli ölçüde desteğe sahiptir. Dokümantasyonu da oldukça geniştir. Raspberry Pi üzerine çok çeşitli işletim sistemi kurulabiliyorsa da şu an için en çok tercih edilen "Raspbian" isimli Linux dağıtımıdır. Raspbian üzerine IDE'ler, editörler, server programlar kurulabilir. Böylece Raspberry Pi ucuz bir PC olarak kullanılabilmektedir. Bu tür embedded Linux kitleri özellikle IO amaçlı endüstriyel ortamlarda kullanılmaktadır. Yani bu kitlere kameralar, sensörler, step motorlar vs. bağlanarak bunlar başka amaçları gerçekleştirmek için kullanılabilmektedir.

**Boru Haberleşmeleri**

Boru haberleşmeleri hem UNIX/Linux sistemlerinde hem de Windows sistemlerinde kullanılan yaygın bir tekniktir. Bu haberleşme modelinde senkronizasyon da otomatik sağlanmaktadır. Bu bakımdan kullanılması kolay bir yöntemdir. Boru hanerleşmeleri her iki sistemde de "İsimli" ve "İsimsiz (anonim de denir)" olmak üzere ikiye ayrılmaktadır.



Borular UNIX/Linux sistemlerinde tek kanallıdır. Yani bir taraf yazar, diğer taraf okur (half duplex). Fakat tek bir boruyla aynı anda iki tarafın yazıp okuması mümkün değildir. Windows sistemlerinde borular çift kanallıdır. Dolayısıyla bir boru yaratıldığında bir tarafın yazdığını diğer taraf okurken, diğer tarafın yazdığını da karşı taraf okuyabilir. Aynı durumu UNIX/Linux sistemlerinde sağlamak için iki boru yaratmak gerekir.

İsimsiz borular üst ve alt prosesler arasında haberleşme kullanılmaktadır. İsimli borular ise aynı makina içerisindeki herhangi iki proses arasında kullanılabilir.

Tipik bir boru haberleşmesi şöyle yapılır:

1) Önce boru yaratılır.

2) Proseslerden biri boruya bilgiyi yazar. Genellikle borular işletim sistemlerinde sanki birer dosyaymış gibi ele alınırlar. Bu nedenle yazma ve okuma işlemleri dosya fonksiyonlarıyla yapılır. (UNIX/Linux sistemlerinde read ve write POSIX fonksiyonları, Windows sistemlerinde ReadFile ve WriteFile API fonksiyonları).

3) Diğer proses bilgiyi okur. Borular birer FIFO kuyruk sistemi gibidir. Ve borularda stream taabanlı bir haberleşme söz konusudur.

4) Borunun belli bir uzunluğu vardır. Eğer yazan taraf yazar, okuyan taaraf bunları almazsa boru dolar. Boru dolduğu zaman yazan taraf yazmaya çalıştığında bloke olur. Yani donar kalır. Ta ki karşı taraf okuyup boruda yer açılana kadar. Benzer biçimde okuyan taraf borudakilerin hepsini okumuşsa ve yeniden okumak isterse yine bloke olur. Yani donarak bekler. Ta ki diğer taraf boruya birşey yazana kadar.

5) Boru haberleşmesi yazan tarafın boruyu kapatmasıyla sonlandırılmalıdır. Bu durumda okuyan taraf önce boruda kalanları okur. Sonra artık okuma fonksiyonu sıfır ile geri döner. Okuyan taraf da bölece karşı tarafın boruyu kapattığını anlar. O da boruyu kapatır. Önce okuyan tarafın boruyu kapatması normal bir durum değildir. Bu genellikle sistemlerde çökmeye yol açar.

**UNIX/Linux Sistemlerinde İsimsiz Boru Haberleşmeleri**

Blokeli modda UNIX/Linux sistemlerinde isimsiz boru haberleşmesi şöyle yapılır:

1) Üst proses önce pipe isimli fonksiyonla boruyu yaratır.

#include <unistd.h>

int pipe(int pipefd[2]);

**Anahtar Notlar:** C'de fonksiyonun gösterici parametresi dizi formunda belirtilebilir ve köşeli parantezlerin içerisine sabit ifadeleri yazılabilir. Bunun okunabilirlik dışında göstericiden hiçbir farkı yoktur. Dolayısıyla aşağıdaki prototiplerin hepsi eşdeğerdir:

void foo(int \*a);

void foo(int a[]);

void foo(int a[2]);

void foo(int a[100]);

Dolayısıyla pipe fonksiyonunun prototipi şöyle de yazılabilirdi:

int pipe(int \*pipefd);

pipe fonksiyonu başarı durumunda 0, başarısızlık durumunda -1 değerine geri döner. Fonksiyona iki elemanlı bir int dizinin adresi geçirilmelidir. Örneğin:

int pipefds[2];

...

if (pipe(pipefds) < 0) {

perror("pipe");

exit(EXIT\_FAILURE);

}

Fonksiyon bize iki betimleyici (yani iki handle değeri) verir. Birinci betimleyici borudan okuma yapmak için kullanılan betimleyicidir. İkincisi ise boruya yazma yapmak için kullanılacak betimleyicidir.

2) Artık üst proses fork fonksiyonuyla alt prosesi oluşturur. Böylece üst prosesin boru betimleyicileri alt prosese aktarılmış olur. Bu durumda borudan okuma potansiyeli olan ve boruya yazma potansiyeli olan ikişer betimleyici vardır.

if ((pid = fork()) < 0) {

perror("fork1");

exit(EXIT\_FAILURE);

}

3) Şimdi hangi prosesin boruya yazma yapacağı, hangisinin okuma yapacağının belirlenmiş olması gerekir. Yazan taraf okuma betimleyicisini, okuyan taraf da yazma betimleyicisini close fonksiyonuyla kapatmalıdır. Yani boruya yazma ve okuma potansiyelinde olan tek bir betimleyici kalmalıdır.

if ((pid = fork()) < 0) {

perror("fork1");

exit(EXIT\_FAILURE);

}

if (pid != 0) { /\* üst proses boruya yazacak \*/

close(pipefds[0]);

...

}

else { /\* alt proses borudan okuyacak \*/

close(pipefds[1]);

...

}

4) Artık yazan taraf write fonksiyonuyla boruya yazma yapar. Okuyan taraf da read fonksiyonuyla borudan okuma yapacaktır:

#include <unistd.h>

ssize\_t read(int fd, void \*buf, size\_t count);

ssize\_t write(int fd, const void \*buf, size\_t count);

Burada ssize\_t türü POSIX sistemlerinde işaretli bir tamsayı türü olarak typedef edilmek zorunda bir tür ismidir.

Fonksiyonların birinci paarametreleri borunun (dosyanın) handle değeridir. (Dosya handle değerlerine UNIX/Linux sistemlerinde "dosya betimleyicisi (file descriptor) denilmektedir) İkinci parametreler bellekteki okuma ya da yazma için gereken transfer adresidir. Son parametreler okunacak ya da yazılacak byte miktarını belirtir. Fonksiyonlar başarılıysa okunan ya da yazılan byte sayısına, başarısızsa -1 değerine geri dönerler. read fonksiyonu 0 ile geri dönerse bu durum karşı tarafın boruyu kapatmış olduğu ve artık boruda okunacak hiçbirşeyin kalmadığı anlamına gelir. read boru boşsa, write ise boru doluysa zaten geri dönmeden beklerler. Blokeli modda çalışmanın önemli birkaç noktası vardır.

- Borudan n byte okunmak istendiğinde boruda n'den daha az byte varsa read fonksiyonu n byte'ın hepsini okuyana kadar blokede kalır.

- Birden fazla kaynak boruya yazma yaparken iç içe geçme olmaz. Ancak borunun bir uzunluğu vardır (PIPE\_LEN). Bu uzunlktan daha fazla yazmak böylesi bir durumda iç içe geçme oluşturabilir. Yazma sırasında da tüm byte'lar yazılana kadar bloke çözülmez.

Örnek bir program şöyle yazılabilir:

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

void pipe\_write\_proc(int fd);

void pipe\_read\_proc(int fd);

int main(void)

{

pid\_t pid;

int pipefds[2];

if (pipe(pipefds) < 0) {

perror("pipe");

exit(EXIT\_FAILURE);

}

if ((pid = fork()) < 0) {

perror("fork");

exit(EXIT\_FAILURE);

}

if (pid != 0) { /\* parent yazacak \*/

close(pipefds[0]);

pipe\_write\_proc(pipefds[1]);

close(pipefds[1]);

wait(NULL);

}

else { /\* child okuyacak \*/

close(pipefds[1]);

pipe\_read\_proc(pipefds[0]);

close(pipefds[0]);

}

return 0;

}

void pipe\_write\_proc(int fd)

{

int i;

for (i = 0; i < 100000; ++i)

if (write(fd, &i, sizeof(int)) < 0) {

perror("write");

exit(EXIT\_FAILURE);

}

}

void pipe\_read\_proc(int fd)

{

int result, val;

while ((result = read(fd, &val, sizeof(int))) > 0)

printf("%d ", val), fflush(stdout);

if (result < 0) {

perror("read");

exit(EXIT\_FAILURE);

}

}

**Shell Pipe İşlemini Nasıl Yapıyor?**

Önceki konularda da görüldüğü gibi shell üzerinden,

a | b

gibi bir işlemde a'nın stdout dosyasına yazdıklarını b stdin'den okumaktadır. Peki bu nasıl yapılmaktadır? İşte shell bunu kabaca şöyle yapar:

1) Önce pipe'ı yaratır

2) a için fork uygular ve a'nın stdout dosyasını boruya yönlendirir. Sonra a için exec uygular. Böylece a stdout dosyasına yazdığını sanırken aslında boruya yazar.

3) b için fork uygular ve b'nin stdin dosyasını boruya yönlendirir. Sonra b için exec uygular. Böylece b stdin dosyasındna okuduğunu sanırken aslında borudan okur. Tabi aslında boruya a'nın yazdıklarını okur.

4) Shell iki prosesin de bitmesini wait ile bekler.

Kursumuzda IO yönlendirmesinin aşağı seviyeli mekanizması üzerinde durulmamıştır. Bu nedenle bu örnek burada yapılmayacaktır. "UNIX/Linux Sistem Programlama" kursunda bu tür örnekler yapılmaktadır.

**Windows Sistemlerinde İsimsiz Boru Haberleşmesi**

Windows'ta isimsiz boru haberleşmesi de çok benzer yapılmaktadır. İsimsiz boru Windows'ta CreatePipe API fonksiyonuyla yaratılır:

BOOL WINAPI CreatePipe(

\_\_out PHANDLE *hReadPipe*,

\_\_out PHANDLE *hWritePipe*,

\_\_in LPSECURITY\_ATTRIBUTES *lpPipeAttributes*,

\_\_in DWORD *nSize*

);

Fonksiyonun birinci ve ikinci parametreleri borudan okumak ve boruya yazmak için gereken handele nesnelerinin adreslerini alır. Fonksiyon buraya okuma ve yazma için gereken handle değerlerini yerleştirir. Üçüncü parametre kernel nesnelerinin güvenlik parametresidir. NULL geçilebilir. Son parametre ise yaratılacak borunun byte cinsinden uzunluğudur. Fonksiyon başarı durumunda sıfır dışı bir değere, başarı sızlık durumunda sıfır değerine geri döner.

Boru yaratıldıktan sonra handle değerlerinin alt prosese geçirilebilmesi için CreateProcess fonksiyonunun 5. parametresi TRUE geçirilmelidir. Bu konu "Windows Sistem Programlama" kursunda ele alınmaktadır. Alt prosese diğer handle'ın (örneğin üst proses yazma yapıyorsa bu yazma hande'ının) geçirilmesine gerek yoktur. CreateProcess'teki 5. parametre "master switch" biçimindedir. Default kernel nesnelerinin handle değerleri alt prosese aktarılmaz. Fakat biz istersek alt prosese belirli handle'ların aktarılmısını isteyebiliriz. Bu akratımın sağlanması yaratıcı fonksiyonların SECURITY\_ATTRIBUTES parametresiyle yapılabileceği gibi SetHandleInformation API fonksiyonuyla da daha sonra yapılabilir.

İsimsiz boru yaratıldıktan sonra yine CreateProcess fonksiyonuyla alt proses oluşturulur. Sonra yine UNIX/Linux sistemlerinde olduğu gibi okuyan taraf yazma borusunu, yazan taraf da okuma borusunu kaparı. Ondan sonra ReadFile ve WriteFile dosya fonksiyonlarıyla okuma yazma işlemleri gerçekleştirilir. Yine önce boruya yazma yapan tarafın boruyu kapatması gerekir. ReadFile borudan sıfır byte okuduğu zamana borunun kapatılmış olduğunu anlar o da boruyu kapatır.

Windows'taki uygulamanın UNIX/Linux sistemlerinden küçük bir farkı daha vardır. Bu sistemlerde CreateProcess adeta UNIX/Linux sistemlerindeki fork/exec işlemine karşılık gelir. Bu duurmda alt proses üst prosesten aktarılan boru handle değerini bilemez. O halde bu handle değerinin de alt prosese aktarılması gerekir. Bu aktarım komut satırı argümanları yoluyla ya da çevre değişkenleri yoluyla yapılabilir.

İsimsiz Boru Haberleşmesi yapan örnek bir Windows programı şöyle yazılabilir:

/\* Parent.c \*/

#include <stdio.h>

#include <stdlib.h>

#include <Windows.h>

void ExitSys(LPCSTR lpszMsg, int status);

void WritePipeProc(HANDLE hPipeWrite);

int main(void)

{

HANDLE hPipeRead, hPipeWrite;

char cmdLine[1024];

STARTUPINFO sa = { sizeof(STARTUPINFO) };

PROCESS\_INFORMATION pi;

if (!CreatePipe(&hPipeRead, &hPipeWrite, NULL, 1024))

ExitSys("CreatePipe", EXIT\_FAILURE);

SetHandleInformation(hPipeRead, HANDLE\_FLAG\_INHERIT, TRUE);

sprintf(cmdLine, "Child.exe %d", hPipeRead);

if (!CreateProcess(NULL, cmdLine, NULL, NULL, TRUE, 0, NULL, NULL, &sa, &pi))

ExitSys("CreateProcess", EXIT\_FAILURE);

CloseHandle(hPipeRead);

WritePipeProc(hPipeWrite);

CloseHandle(hPipeWrite);

return 0;

}

void WritePipeProc(HANDLE hPipeWrite)

{

int i;

DWORD dwByteWritten;

for (i = 0; i < 100000; ++i)

if (!WriteFile(hPipeWrite, &i, sizeof(int), &dwByteWritten, NULL))

ExitSys("WriteFile", EXIT\_FAILURE);

}

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);

}

/\* Child.c \*/

#include <stdio.h>

#include <stdlib.h>

#include <Windows.h>

void ReadPipeProc(HANDLE hPipeRead);

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

{

HANDLE hPipeRead;

if (argc != 2) {

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

exit(EXIT\_FAILURE);

}

sscanf(argv[1], "%d", &hPipeRead);

ReadPipeProc(hPipeRead);

return 0;

}

void ReadPipeProc(HANDLE hPipeRead)

{

DWORD byteRead;

int val;

BOOL result;

while (ReadFile(hPipeRead, &val, sizeof(int), &byteRead, NULL) && byteRead != 0)

printf("%d ", val), fflush(stdout);

}

**İsimli Boru Haberleşmeleri**

İsimsiz boru haberleşmesi yalnızca üst proses ile alt proses arasındaki haberleşmelerde kullanılabilmektedir. Halbuki aralarında üstlük-altlık ilişkisi olmayan herhangi iki prosesin haberleşmesi için isimli boru haberleşmesi kullanılır.

**UNIX/Linux Sistemlerinde İsimli Boru Haberleşmesi**

UNIX/Linux sistemlerinde isimli borular sanki birer dosyaymış gibi ele alınmaktadır. Onlar dosya sisteminde dizin girişlerinde 'p' dosya özelliğiyle görüntülenmektedir. Boru dosyaları gerçek anlamda diskte yer kaplamazlar. Onların yalnızca bir dizin girişleri vardır. Onların gerçek yerleri ana bellektedir. İsimli borular sırasıyla şu adımlarla oluşturulur:

1) UNIX/Linux sistemlerinde isimli borularla çalışmak için önce bir boru dosyasının oluşturulması gerekir. Boru dosyasını oluşturabilmek için mkfifo isimli POSIX fonksiyonu kullanılabilir. Ya da komut satırından mkfifo isimli komut zaten bu fonksiyonu çağırarak isimli boru dosyası oluşturmaktadır. mkfifo fonksiyonunun prototipi şöyledir:

#include <sys/stat.h>

int mkfifo(const char \*pathname, mode\_t mode);

Fonksiyonun birinci parametresi boru dosyasının yol ifadesini, ikinci parametresi boru dosyasının erişim haklarını belirtmektedir. İkinci parametren S\_IRUSR|S\_IWUSR|S\_IRGRP|S\_IROTH biçiminde girilebilir. Bu parametrenin anlamı ileride open fonksiyonunda anlatılacaktır. Fonksiyon başarı durumunda sıfır değerine, başarıszlık durumunda -1 değerine geri döner.

mkfifo komutu da aşağıdaki gibi kullanılabilir:

mkfifo mypipe

2) Boru dosyası open isimli dosya açan PSOIX fonksiyonuyla açılır. Ya da istenirse bunun için fopen fonksiyonu da kullanılabilir. Ancak fopen kullanılacaksa dosya tamponlama stratejisinin "sıfır tamponlamalı moda" çekilmesi uygun olur. open fonksiyonu ileride ele alınacaktır. Ancak burada üstün körü bir biçimde hedefe yönelik kullanımdan bahsedebiliriz. Açış işlemini bir proses okuma modunda diğeri de yazma modunda yapmalıdır:

if ((fd = open("mypipe", O\_RDONLY)) < 0) {

perror("open");

exit(EXIT\_FAILURE);

}

...

if ((fd = open("mypipe", O\_WRONLY)) < 0) {

perror("open");

exit(EXIT\_FAILURE);

}

...

Blokeli modda proseslerden biri boruyu okuma modunda açmışsa, diğeri yazma modunda açana kadar open blokede kalır. Benzer biçinde bir proses bpruyu yazma modunda açmışsa diğeri okuma modunda açana kadar blokede kalır.

3) Artık yazan taraf write fonksiyonuyla boruya yazma yapar. Okuyan taraf da read fonksiyonuyla okumaya yapar. Tabi yine boru dolduğunda yazan taraf, boru boş olduğunda da okuyan taraf blokede kalır.

4) Yine önce yazan tarafın boruyu kapatması gerekir. Daha sonra karşı taraf read fonksiyonuyla sıfır byte okuduğunda o da boruyu kapatır.

Örnek bir program şöyle yazılabilir:

/\* pipeproc1.c \*/

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <fcntl.h>

#include <sys/stat.h>

void pipe\_write\_proc(int fd);

int main(void)

{

int fd;

if ((fd = open("mypipe", O\_WRONLY)) < 0) {

perror("open");

exit(EXIT\_FAILURE);

}

pipe\_write\_proc(fd);

close(fd);

return 0;

}

void pipe\_write\_proc(int fd)

{

int i;

for (i = 0; i < 100000; ++i)

if (write(fd, &i, sizeof(int)) < 0) {

perror("write");

exit(EXIT\_FAILURE);

}

}

/\* pipeproc.2 \*/

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <fcntl.h>

void pipe\_read\_proc(int fd);

int main(void)

{

int fd;

if ((fd = open("mypipe", O\_RDONLY)) < 0) {

perror("open");

exit(EXIT\_FAILURE);

}

pipe\_read\_proc(fd);

close(fd);

return 0;

}

void pipe\_read\_proc(int fd)

{

int result, val;

while ((result = read(fd, &val, sizeof(int))) > 0)

printf("%d ", val), fflush(stdout);

if (result < 0) {

perror("read");

exit(EXIT\_FAILURE);

}

}

İsimli boru dosyası standart C fonksiyonlarıyla da açılıp, okuma yazma yapılabilir. Tabi yukarıda da belirtildiği gibi bu durumda tamponlama mekanizmasını kaldırmak uygun olur:

/\* pipeproc1.c \*/

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <fcntl.h>

#include <sys/stat.h>

void pipe\_write\_proc(FILE \*f);

int main(void)

{

FILE \*f;

if ((f = fopen("mypipe", "w")) == NULL) {

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

exit(EXIT\_FAILURE);

}

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

pipe\_write\_proc(f);

fclose(f);

return 0;

}

void pipe\_write\_proc(FILE \*f)

{

int i;

for (i = 0; i < 100000; ++i)

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

break;

if (ferror(f)) {

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

exit(EXIT\_FAILURE);

}

}

/\* pipeproc2.c \*/

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <fcntl.h>

void pipe\_read\_proc(FILE \*f);

int main(void)

{

FILE \*f;

if ((f = fopen("mypipe", "r")) == NULL) {

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

exit(EXIT\_FAILURE);

}

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

pipe\_read\_proc(f);

fclose(f);

return 0;

}

void pipe\_read\_proc(FILE \*f)

{

int result, val;

while (fread(&val, sizeof(int), 1, f) > 0)

printf("%d ", val), fflush(stdout);

if (ferror(f)) {

perror("cannot read pipe!\n");

exit(EXIT\_FAILURE);

}

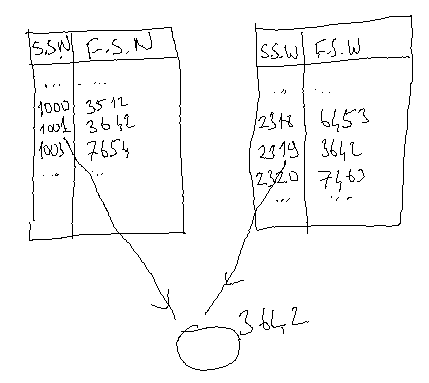
}

**Windows Sistemlerinde İsimli Boru Haberleşmesi**

Bu kursumuzda Windows sistemlerindeki isimli boru haberleşmeleri üzerinde durmayacağız. Çünkü Windows sistemlerinde isimli borular adeta clien-server tarzda kullanılmaktadır. Kavramsal karmaşıklığı fazladır. Bu nedenle isimli borular Derneğimizde "Windows Sistem Programlama" kursunun konusu içerisindedir.

**Paylaşılan Bellek Alanları (Shared Memory) Yöntemi İle Proseslerarası Haberleşme**

Anımsanacağı ağı gibi modern sistemlerde işlemcinin sağladığı bir sayafalama ve buna bağlı olarak bir sanal bellek mekanizması vardır. Bu mekanizma proseslerin bellek alanlarını tamamen birbirlerinden izole eder. Paylaşılan bellek alanları yönteminde işletim sistemi iki ya da daha fazla prosesin farklı sanal sayfalarını aynı fiziksel sayfaya yönlendirir. Örneğin:



Burada soldaki prosesin 1001 numaralı sanal sayfası ile sağdaki prosesin 2319 numaralı sanal sayfası aynı fiziksel sayfaya yönlendirilmiştir.

Paylaşılan bellek alanları yöntemi çok hızlı bir yöntemdir. Prosesler hiç kernel moda geçmeden haberleşebilirler. Ancak bu yöntem kendi içerisinde borularda olduğu gibi bir senkronizasyon içermez. Yani proseslerden biri paylaşılan bellek alanına yazdığında diğeri onun yazdığını ne zaman alacaktır? Diğeri yeni birşey yazarsa eskisi ezilmez mi? İşte senkronizasyon için üretici-tüketici problemi (producer-consumer problem) gibi bir yöntemin kullanılması gerekir.

Paylaşılan bellek alanları hem Windows hem de UNIX/Linux sistemlerinde var olan bir yöntemdir. Windows'taki ve UNIX/Linux sistemlerindeki bellek tabanlı dosyalar (memory mapped files) paylaşılan bellek alanları konusuyla birleştirilmiştir.

**Windows Sistemlerinde Paylaşılan Bellek Alanlarının Oluşturulması**

Windows sistemlerinde paylaşılan bellek alanları şu adımlardan geçilerek oluşturulur:

1) Öncelikle bir "file mapping" nesnesinin yaratılması gerekir. Bu işlem CreateFileMapping fonksiyonuyla yapılır:

HANDLE WINAPI CreateFileMapping(

\_\_in HANDLE *hFile*,

\_\_in LPSECURITY\_ATTRIBUTES *lpAttributes*,

\_\_in DWORD *flProtect*,

\_\_in DWORD *dwMaximumSizeHigh*,

\_\_in DWORD *dwMaximumSizeLow*,

\_\_in LPCTSTR *lpName*

);

Fonksiyonun birinci parametresi eğer "bellek tabanlı dosya (memory mapped file)" kullanılacaksa o dosyanın handle değerini alır. eğer yalnızca paylaşılan bellek alanı oluşturulacaksa bu parametreye INVALID\_HANDLE\_VALUE özel değeri geçilmelidir. Fonksiyonun ikinci parametresi file mapping nesnesinin güvenlik parametresidir. NULL geçilebilir. Üçüncü parametre oluşturaln bellek alanının koruma özelliğini belirtir. Bu parametre PAGE\_READWRITE girilirse bu bölgeye hem okuma hem de yazma yapılabilir. Fonksiyonun dördüncü ve beşinci parametreleri paylaşılacak bellek alanın uzunluğunu belirtir. Aslında bu iki parametre 8 byte'lık bir tamsayının düşük ve yüksek anlamlı 4'er byte'ıdır. Bu değerin sayfa katlarında olması anlamlıdır. Son parametre proseslerarası paylaşım için gereken isimdir. Fonksiyon başarı durumunda mapping nesnesine ilişkin bir handle değeri ile, baaşarısızlık durumunda NULL adresle geri döner.

Bu fonksiyon iki proseste de çağrılır. İlk çağıran proses nesneyi yaratmış olur. Artık diğer çağırandaki parametreler etkili olmaz. İkinci çağıran proses yalnızca nesneyi açmış olur. Dolayısıyla fonksiyonu sonradan çağıran kişi için üçüncü, dördüncü, beşinci parametrelerin bir önemi yoktur. Tabi iki proseste de aynı değerler kullanılırsa hangisinin önce çalıştığının bir önemi kalmaz. Eğer istenirse proseslerden biri CreateFileMapping fonksiyonuyla nesneyi yaratırken diğeri OpenFileMapping fonksiyonuyla açabilir. OpenFileMapping yalnızca olanı açma özelliğine sahiptir. Yaratma özelliğine sahip değildir.

2) Bundan sonra MapViewOfFile isimli API fonksiyonuyla gerçek bellek adresi elde edilir. Yani işletim sistemi bu fonksiyonla paylaşılan fiziksel bellek alanını gören bir sanal sayfa numarasını sayfa tablosunda oluşturup bize oranın sanal adresini verecektir:

LPVOID WINAPI MapViewOfFile(

\_\_in HANDLE *hFileMappingObject*,

\_\_in DWORD *dwDesiredAccess*,

\_\_in DWORD *dwFileOffsetHigh*,

\_\_in DWORD *dwFileOffsetLow*,

\_\_in SIZE\_T *dwNumberOfBytesToMap*

);

Fonksiyonun birinci parametresi mapping nesnesinden elde edilen handle değeridir. İkinci parametre sayafay erişim özelliğini belirtir. Tabi buaradaki parametre mapping nesnesi yaratılırken verilen parametreden daha geniş haklara sahip olamaz. Tipik olarak bu parametreye FILE\_MAP\_READ|FILE\_MAP\_WRITE değeri girilmektedir. Fonksiyonun üçüncü parametreleri map edilecek kısmın offetini alır. Tabi bu parametreler bellek tabanlı dosyalar için anlamlıdır. eğer biz yalnızca paylaşılan bellek alanı oluşturuyorsak bu parametreye sıfır geçmemeiz anlamlı olur. Son parametre paylaşılacak bellek alan ın büyüklüğüdür. Mapping nesnesinde belirtilen büyüklük maksimum büyüklüktür. Buradaki değer ondan küçük olabilir ya da ona eşit olabilir. Ancak ondan büyük olamaz. Fonksiyon başarı durumunda paylaşılan bellek alanına erişmekte kullanılan adrese, başarısızlık durumda NULL adrese geri döner.

3) Kullanım bitince paylaşılan bellek alanının sisteme iade edilmesi gerekir. Bu UnmapViewOfFile fonksiyonuyla yapılır:

BOOL WINAPI UnmapViewOfFile(

\_\_in LPCVOID *lpBaseAddress*

);

Fonksiyon paylaşılan bellek alanının başlangıç adresini parametre olarak alır ve o alanı serbest bırakır.

4) Nihayet proses mapping nesnesini de CloseHandle fonksiyonuyla serbest bırakmalıdır. Tabi bittiğinde bu son iki aşama otomatik yapılır.

BOOL WINAPI CloseHandle(

\_\_in HANDLE *hObject*

);

Örneğin:

/\* Process1.c \*/

#include <stdio.h>

#include <stdlib.h>

#include <Windows.h>

void ExitSys(LPCSTR lpszMsg, int status);

int main(void)

{

HANDLE hFileMapping;

char \*memstr;

hFileMapping = CreateFileMapping(INVALID\_HANDLE\_VALUE, NULL, PAGE\_READWRITE, 0, 4096, "ThisIsATest");

if (hFileMapping == NULL)

ExitSys("CreateFileMapping", EXIT\_FAILURE);

memstr = (char \*)MapViewOfFile(hFileMapping, FILE\_MAP\_READ | FILE\_MAP\_WRITE, 0, 0, 4096);

if (memstr == NULL)

ExitSys("MapViewOfFile", EXIT\_FAILURE);

printf("Process-1: Press ENTER to write into shared memory\n");

getchar();

strcpy(memstr, "this is a test");

printf("Process-1: Press ENTER to EXIT");

getchar();

UnmapViewOfFile(memstr);

CloseHandle(hFileMapping);

return 0;

}

void ExitSys(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);

}

/\* Process2.c \*/

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <Windows.h>

void ExitSys(LPCSTR lpszMsg, int status);

int main(void)

{

HANDLE hFileMapping;

char \*memstr;

hFileMapping = CreateFileMapping(INVALID\_HANDLE\_VALUE, NULL, PAGE\_READWRITE, 0, 4096, "ThisIsATest");

if (hFileMapping == NULL)

ExitSys("CreateFileMapping", EXIT\_FAILURE);

memstr = (char \*)MapViewOfFile(hFileMapping, FILE\_MAP\_READ | FILE\_MAP\_WRITE, 0, 0, 4096);

if (memstr == NULL)

ExitSys("MapViewOfFile", EXIT\_FAILURE);

printf("Process-2: Press ENTER to read from shared memory\n");

getchar();

puts(memstr);

printf("Process-2: Press ENTER to exit\n");

getchar();

UnmapViewOfFile(memstr);

CloseHandle(hFileMapping);

return 0;

}

void ExitSys(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 Paylaşılan Bellek Alanlarının Oluşturulması**

UNIX/Linux sistemlerinde paylaşılan bellek alanlarını oluşturmak için iki grup fonksiyon vardır. Bunlardan biri en eskiden beri var olan "System V" fonksiyonları denilen gruptur. İkincisi ise 90'lı yıllarda UNIX türevi sistemlere sokulmuş olan modern fonksiyonlardır. Bunlara "POSIX shared memory" fonksiyonları denilmektedir. Aslında her iki fonksiyon grubu da POSIX standartlarında vardır. Biz burada daha sonra çıkmış olan modern paylaşılan bellek alanı fonksiyonlarını göreceğiz. Diğer fonksiyonlar "UNIX/Linux Sistem Programlama" kursunda ele alınmaktadır.

UNIX/Linux sistemlerinde paylaşılan bellek alanı oluşturma işlemi şu adımlardan geçilerek yapılır:

1) Paylaşılan bellek alanı nesnesi shm\_open fonksiyonuyla yaratılır. shm\_open hem nesneyi ilk kez yaratmak için hem de yaratılmış olanı açmak için kullanılır:

#include <sys/mman.h>

#include <sys/stat.h> /\* For mode constants \*/

#include <fcntl.h> /\* For O\_\* constants \*/

int shm\_open(const char \*name, int oflag, mode\_t mode);

Fonksiyonun birinci parametresi paylaşılan bellek alanın proseslerarası kullanımı için gereken isimdir. Bu ismin kök dizinde sanki bir dosya ismiymiş gibi verilmesi gerekmektedir. (Tabi aslında böyle bir dosya yoktur. Dizin girişinde de görülmeyecektir.) İkinci parametre açış modunu belirtir. Açış modeu open fonksiyonunda olduğu gibidir. O\_CREAT "yoksa yarat, varsa olanı aç" anlamına gelir. Bununla birlikte O\_RDONLY ya da O\_RDWR baurakları OR'lanmalıdır. O\_EXCL "nesne varsa açma, başarısızlıkla sonlan" anlamına gelmektedir. Üçüncü parametre nesnenin erişim haklarını belirtir. Bu konu ileride ele alınacaktır. Fakat tipik olarak S\_IRUSR|S\_IWUSR|S\_IRGRP|S\_IROTH biçiminde girilir. Bu üçüncü parametre nesne yaratılırken kullanılır. Nesne yaratılmışsa, yaratılmış olan açılacaksa bu argüman dikkate alınmaz. Fonksiyon başarı durumunda sıfır değerine, başarıszlık durumunda -1 değerine geri döner.

2) Paylaşılan bellek alanına ilişkin dosya için ftuncate fonksiyonuyla bir alanın ayrılması gerekir:

#include <unistd.h>

int ftruncate(int fd, off\_t length);

Fonskiyonun birinci parametresi paylaşılan bellek alanının betimleyicisi, ikinci parametresi alanın uzunluğudur. Fonksiyon başarı durumunda sıfır değerine, başarısızlık durumunda -1 değerine geri döner.

3) Paylaşılan bellek alanı nesnesi için mmap fonksiyonuyla bir sanal adres oluşturulur. İşletim sistemi bu sanal adrese ilişkin sayfayı ortak bir fiziksel sayfaya yönlendirecektir:

#include <sys/mman.h>

void \*mmap(void \*addr, size\_t length, int prot, int flags, int fd, off\_t offset);

Fonksiyonun birinci parametresi önerilen adrestir. Bu parametre NULL geçilirse sistem bize istdeği sanal adresi verir. İkinci parametre paylaşılan bellek alanının uzunluğunu belirtir. Üçüncü parametre erişim bilgilerini belirtmektedir. Bu parametre aşağıdaki bir ya da birden fazla bayrağın OR'lanmasıyla oluşturulabilir:

**PROT\_EXEC** Pages may be executed.

**PROT\_READ** Pages may be read.

**PROT\_WRITE** Pages may be written.

**PROT\_NONE**

Dördüncü parametre ya MAP\_SHARED ya da MAP\_PRIVATE geçilmelidir. Paylaşılan bellek alanı için MAP\_SHARED kullanılmalıdır. Beşinci parametre shm\_open fonksiyonundan elde edilen dosya betimleyicisidir. Son parametrede ayrılan alanın neresinden itibaren map edileceğini belirtir. Bu parametre paylaşılan alan için sıfır geçilmelidir. Fonksiyon başarı durumunda paylaşılan bellek alanın sanal adresiyle, başarısızlık durumunda MA\_FAILED değerine geri döner.

**Anahtar Notlar:** Paylaşılan bellek alanının kullanan fonksiyonlar libc kütüphanesi içerisinde değildir. Dolayısıyla linker bu kütüphaneye default olarak bakmamaktadır. Bu nedenle derleme yapılırken -lrt seçeneği ile librt kütüphanesi devreye sokulmalıdır.

4) Paylaşılan bellek alanı munmap fonksiyonuyla geri bırakılır:

#include <sys/mman.h>

int munmap(void \*addr, size\_t length);

Fonksiyonun birinci parametresi paylaşılan alanın sanal bellek adresi ikinci parametresi ise ne kadar alanın serbesy bırakılacağını belirtir. Fonksiyon başarı durumunda sıfır değerine, başarıszlık durumunda -1 değerine geri döner.

5) Paylaşılan bellek alanı nesnesi close fonksiyonuyla yok edilir.

int close(int fd);

Örnek bir paylaşılan bellek alanı uygulaması şöyle yapılabilir:

/\* shmprocess1.c \*/

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <unistd.h>

#include <sys/mman.h>

#include <sys/stat.h>

#include <fcntl.h>

int main(void)

{

int shmfd;

char \*memstr;

if ((shmfd = shm\_open("/this\_is\_a\_test", O\_CREAT | O\_RDWR, S\_IRUSR | S\_IWUSR | S\_IRGRP | S\_IROTH)) < 0) {

perror("shmfd");

exit(EXIT\_FAILURE);

}

if (ftruncate(shmfd, 4096) < 0) {

perror("ftruncate");

exit(EXIT\_FAILURE);

}

if ((memstr = (char \*)mmap(NULL, 4096, PROT\_READ | PROT\_WRITE, MAP\_SHARED, shmfd, 0)) == MAP\_FAILED) {

perror("mmap");

exit(EXIT\_FAILURE);

}

printf("Process-1: Press ENTER to write into shared memory\n");

getchar();

strcpy(memstr, "this is a test");

printf("Process-1: Press ENTER to EXIT");

getchar();

munmap(memstr, 4096);

close(shmfd);

return 0;

}

/\* shmprocess2.c \*/

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <unistd.h>

#include <sys/mman.h>

#include <sys/stat.h>

#include <fcntl.h>

int main(void)

{

int shmfd;

char \*memstr;

if ((shmfd = shm\_open("/this\_is\_a\_test", O\_CREAT | O\_RDWR, S\_IRUSR | S\_IWUSR | S\_IRGRP | S\_IROTH)) < 0) {

perror("shmfd");

exit(EXIT\_FAILURE);

}

if (ftruncate(shmfd, 4096) < 0) {

perror("ftruncate");

exit(EXIT\_FAILURE);

}

if ((memstr = (char \*)mmap(NULL, 4096, PROT\_READ | PROT\_WRITE, MAP\_SHARED, shmfd, 0)) == MAP\_FAILED) {

perror("mmap");

exit(EXIT\_FAILURE);

}

printf("Process-2: Press ENTER to read from shared memory\n");

getchar();

puts(memstr);

printf("Process-2: Press ENTER to exit\n");

getchar();

munmap(memstr, 4096);

close(shmfd);

return 0;

}

**Anahtar Notlar:** Özellikle gömülü ortamlarda bazı fiziksel bellek bölgeleri çeşitli IO uçlarına yönlendirilmiştir. Yani biz adeta bellekte o yere veri aktardığımızda aslında o IO uçlarına 5V göndermiş oluruz. Benzer biçimde belleğin o bölgesini okuduğumuzda o uçlardaki gerilim değerini okumuş oluruz. Bu tekniğe "bellek tabanlı IO tekniği (mempry mapped IO)" denilmektedir. Örneğin Raspberry PI bu tekniği kullanmaktadır.ş Raspberry PI'da fiziksel belleğim 512 megabyte'ından itibaren IO bölgesi başlar.

**Mesaj Kuyrukları Yöntemi**

Bu yöntem kursumuzda ele alınmayacaktır. "UNIX/Linux Sistem Programlama" kursunda ele alınmaktadır. Bu yöntemde ismine mesaj kuyruğu (message queue) denilen bir kuruk sistemi yaratılır. Bir proses bu kuyruğa yazma yaparken diğeri okuma yapar. Biçim bakımından boru haberleşmesine benzemektedir. Ancak mesaj kuyruklarında paket tarzı bir aktarım vardır. Bir taraf mesaj adı altında bir paket bilgi gönderir. Diğeri bunu tamamen alır. Bir kısmını almak sonra kalanını almak gibi bir durum söz konusu değildir. Boru haberleşmesi "stream" tabanlı bir haberleşme sunduğu halde mesaj kuyrukları paket tabanlı (datagram) tarzı bir haberleşme sunmaktadır.

**Aşağı Seviyeli Fonksiyonlarla Dosya İşlemleri**

Bilindiği gibi dosya sistemi tamamen işletim sisteminin kontrolü altındadır. Biz hangi dille ya da hangi kütüphaneyle çalışıyor olursak olalım eninde sınıfında dosya işlemleri işletim sisteminni sistem fonksiyonlarıyla kernel moda geçerek gerçekleştirilmektedir.

Peki neden standart C fonksiyonları varken işletim sisteminin aşağı seviyeli sistem fonksiyonlarıyla dosya işlemi yapmak isteriz? Standat C fonksiyonları her sistemde olabilecek düşük bir parametrik yapı sunmaktadır. Oysa sistemlerin kendine özgü özellikleri vardır. Örneğin Windows'ta birisi dosyayı açmışsa başka birisinin dosyayı açmasını istemeyebiliriz. Bu işlem fopen ile yapılamaz. Aygıt sürücülerle çalışırken mecburen fopen'da olmayan bazı açış modlarını kullanmak zorunda kalabiliriz. Aynı zamanda işletim sisteminin aşağı seviyeli dosya fonksiyonları bize işletim sisteminin dosya sisteminin tasarımı hakkında da daha iyi bilgiler vermektedir.

**Windows Sistemlerinde Aşağı Seviyeli Dosya İşlemleri**

Windows sistemlerinde temel dosya işlemlerini yapmak için 5 API fonksiyonu kullanılır:

CreateFile

ReadFile (ReadFileEx)

WriteFile (WriteFileEx)

CloseHandle

SetFilePointer (SetFilePointerEx)

ReadFile, WriteFile ve SetFilePointer fonksiyonlarının daha sonra Ex'li (IO Completion port özelliği ile) daha yeetenekli versiyonları oluşturulmuştur. CreateFile fonksiyonun prototipi şöyledir:

HANDLE WINAPI CreateFile(

\_\_in LPCTSTR *lpFileName*,

\_\_in DWORD *dwDesiredAccess*,

\_\_in DWORD *dwShareMode*,

\_\_in LPSECURITY\_ATTRIBUTES *lpSecurityAttributes*,

\_\_in DWORD *dwCreationDisposition*,

\_\_in DWORD *dwFlagsAndAttributes*,

\_\_in HANDLE *hTemplateFile*

);

Fonksiyonun birinci parametresi açılacak dosyanın yol ifadesidir. İkinci parametre açış modunu belirtir. Bu parametre en azından GENERIC\_READ, GENERIC\_WRITE bayraklarından yalnızca birini ya da her ikisini içermelidir. Diğer bayraklar için MSDN dokümanlarına başvurulabilir. Üçüncü parametre dosyanın paylaşım modunu belirtir. Bu parametre FILE\_SHAE\_READ, FILE\_SHARE\_WRITE ve FILE\_SHARE\_DELETE bayraklarıyla oluşturulur. Örneğin FILE\_SHARE\_READ bayrağı "başkası bu dosyayı açsın ama read modda açsın" anlamına gelir. Eğer bu parametre sıfır girilirse başkası dosyayı açamaz. Örneğin fopen fonksiyonu dosyayı her zaman FILE\_SHAREREAD|FILE\_SHARE\_WRITE|FILE\_SHARE\_DELETE modunda açmaktadır. Fonksiyonun dördüncü parametresi kernel nesnesinin güvenlik durumunu belirtir. NULL geçilebilir. Beşinci parametre yaratım modunu belirtir. Örneğin bu parametrede OPEN\_EXISTING bayrağı "dosya varsa aç, yoksa başarısz ol" anlamına gelir. Bu bayrak fopen'daki "r" moduna benzetilebilir. CREATE\_ALWAYS "dosya varsa sıfırla ve aç, dosya yoksa yarat ve aç" anlamına gelir. Bu fopen'daki "w" modu gibidir. CREATE\_NEW "dosya yoksa yarat ve aç, dosya varsa başarısız ol" anlamına gelir. TRUNCATE\_EXISTING ise "dosya yoksa başarıs ol, varsa sıfırla ve aç" anlamına gelir. Fonksiyonun altıncı parametresi yaratılacak dosyanın dosya özelliklerini belirtir. Bu parametre FILE\_ATTRIBUTE\_NORMAL geçilebilir. Bu durumda tipik bazı dosya özellikleri (ayrıntılar için MSDN'e bakınız) verilmiş olur. Eğer olan dosya açılacaksa bu parametre sıfır olarak da geçilebilir. Son parametre zaten açık bir dosyanın handle'ı elimnizde varsa onun özelliklerine sahip bir dosya yaratmak için kullanılır. Bu parametre NULL geçilebilir. Fonksiyon başarı durumunda dosyanın handle değerine başarıszlık durumunda INVALID\_HANDLE\_VALUE değerine geri döner.

Dosyadan okuma yapmak için ReadFile API fonksiyonu kullanılır:

BOOL WINAPI ReadFile(

\_\_in HANDLE *hFile*,

\_\_out LPVOID *lpBuffer*,

\_\_in DWORD *nNumberOfBytesToRead*,

\_\_out LPDWORD *lpNumberOfBytesRead*,

\_\_in LPOVERLAPPED *lpOverlapped*

);

Fonksiyonun birinci parametresi okunanacak dosyanın handle değerini, ikinci parametresi okunan bilgilerinin yerleştirileceği adresi, üçüncü parametresi okunacak byte miktarını, dördüncü parametresi okunmuş olan byte miktarını belirtir. Son parametre "overllaped io" için gereken yapıdır. Bu parametre NULL geçilebilir.

Dosyaya yazmak için WriteFile API fonksiyonu kullanılır. Bu fonksiyonun parametrik yapısı ReadFile fonksiyonundaki gibidir:

BOOL WINAPI WriteFile(

\_\_in HANDLE *hFile*,

\_\_in LPCVOID *lpBuffer*,

\_\_in DWORD *nNumberOfBytesToWrite*,

\_\_out LPDWORD *lpNumberOfBytesWritten*,

\_\_in LPOVERLAPPED *lpOverlapped*

);

Fonksiyon tam ters olarak bizim veridiğimiz adresten itibaren belirttiğimiz miktarda byte'ı dosya göstericisinin gösterdiği yerden itibaren yazar.

Dosya nihayet CloseHandle API fonksiyonuyla kapatılır. CloseHandle yalnızca dosyaları kapatmak için değil ismine "kernel nesnesi" denilen tüm handle alanlarını kapatmak için ortak kullanılan bir fonksiyondur.

Dosya göstericini konumlandırmakl için SetFilePointer API fonksiyonu kullanılmaktadır:

DWORD WINAPI SetFilePointer(

\_\_in HANDLE *hFile*,

\_\_in LONG *lDistanceToMove*,

\_\_in\_out\_opt PLONG *lpDistanceToMoveHigh*,

\_\_in DWORD *dwMoveMethod*

);

Fonksiyonun birinci parametresi dosyanın handle değerini alır. İkinci ve üçüncü parametreler konumlandırma offset'ini belirtmektedir. Üçüncü parametre NULL geçilebilir. NULL geçilmezse buraya konumlandırma offset'inin yüksek anlamlı 4 byte'ı yazılmaktadır. Son parametre konumlandırma orijinini belirtmektyedir. Bu aparemtre FILE\_BEGIN, FILE\_CURRENT ya da FILE\_END biçiminde girilmelidir. Fonksiyon başarı durumunda dosya göstericisinin yewni konumunun düşük anlamlı 4 byte'ına geri döner. Başarısızlık INVALID\_SET\_FILE\_POINTER değerine geri döner.

Aşağı seviyeli dosya kullanımına bir örnek şöyle verilebilir:

#include <stdio.h>

#include <stdlib.h>

#include <Windows.h>

void ExitSys(LPCSTR lpszMsg, int status);

int main(void)

{

HANDLE hFile;

DWORD dwBytresRead, dwBytesWritten;

char rbuf[10 + 1];

char wbuf[] = "xxxxx";

if ((hFile = CreateFile("test.txt", GENERIC\_READ|GENERIC\_WRITE, FILE\_SHARE\_READ, NULL, OPEN\_EXISTING, FILE\_ATTRIBUTE\_NORMAL, NULL)) == INVALID\_HANDLE\_VALUE)

ExitSys("CreateFile", EXIT\_FAILURE);

if (!ReadFile(hFile, rbuf, 10, &dwBytresRead, NULL))

ExitSys("ReadFile", EXIT\_FAILURE);

rbuf[dwBytresRead] = '\0';

puts(rbuf);

if (!WriteFile(hFile, wbuf, strlen(wbuf), &dwBytesWritten, NULL))

ExitSys("ReadFile", EXIT\_FAILURE);

CloseHandle(hFile);

return 0;

}

void ExitSys(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 Aşağı Seviyeli Dosya İşlemleri**

UNIX/Linux sistemlerinde aşağı seviyeli dosya işlemleri için birkaç POSIX fonksiyonları kullanılmaktadır. Bu fonksiyonlar doğrudan işletim sisteminin sistem fonksiyonlarını çağırırlar. UNIX/Linux sistemlerindeki temel POSIX dosya fonksiyonları şunlardır:

open

read

write

close

lseek

open fonksiyonun prototipi şöyledir:

#include <sys/stat.h>

#include <fcntl.h>

int open(const char \*pathname, int flags, ...);

Fonksiyon iki ya da üç parametreli kullanılır. Fonksiyonun birinci parametresi açılacak dosyanın yol ifadesini belirtir. İkinci parametre açış modunu belirtmektedir. Açış modu en azından şunlardan yalnızca birini içermek zorudadır:

O\_RDONLY

O\_WRONLY

O\_RDWR

Buna ilaveten aşağıdaki bayraklar da OR'lanarak açış modunda kullanılabilir:

O\_CREAT: Bu modda dosya yoksa yaratılır ve açılır, varsa olan dosya açılır.

O\_TRUNC: Bu modda dosya açılırken varsa aynı zamanda sıfırlanmaktadır. Örneğin O\_CREAT|O\_TRUNC modu "dosya yoksa yarat ve aç, varsa sıfırla ve aç" anlamına gelir. Yani fopen'daki "w" modudur.

O\_APPEND: Bu modda dosyadan okuma yapılabilir. Ancak her yazma işlemi önce dosya göstericisinin dosyanın sonuna çekilmesiyle yapılmaktadır. Yani her write işlemi sona eklemeye yol açar.

O\_EXCL: Bu modda yaratım sırasında dosya varsa open başarısız olur. Yani olmayan bir dosyayı yaratmanın garanti altına alınması için bu mod kullanılmaktadır. O\_EXCL tek başına kullanılamaz. Mutlaka O\_CREAT ile birlikte kullanılmalıdır.

Diğer modlar için ilgili dokümanlara başvurabilirsiniz.

Fonksiyon fonksiyonun son parametresi yaratma riski varsa kullnılmalıdır. Yani ikinci parametrede O\_CREAT belirtilmişse üçüncü parametre girilmelidir. Üçüncü parametre dosyanın erişim haklarını belirtir. Erişim hakları ls -l komutunda aşağıdaki gibi görüntülenmektedir:

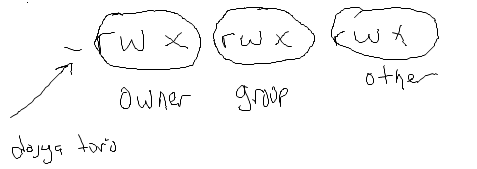
-rwxrwxrwx

Erişim hakları için toplam 9 karakter kullanılır. En soldaki karakter dosyanın türünü belirtir. '-' normal dosya (regular file) anlamına gelir. 'd' dizin anlamına gelir. Diğer erişim hakları 3'lük 3 grup oluşturmaktadır:

UNIX/Linux sistemlerinde her prosesin bir "user id" ve "group id" değeri vardır. Bu değerler sayısaldır ve prosesin kontrol bloğunda saklanır. Üst prosesten alt prosese fork işlemi sırasında aktarılır ve exec işlemi sırasında değişmez. Bu id değerleri login programı tarafından password başarılı girilmişse /etc/passwd ve /etc/group dosyalarına bakılarak set edilir. Ayrıca sistem "user id"leri "user name"lerle, "group id"leri de "group name"lerle ilişkilendirmiştir. Bu ilişkilendirme yine /etc/passwd ve /etc/gropu dosyalarında belirtilmektedir. Böylece konuşurken ya da görüntülerken "user id"ler yerine "user name"ler, "group id"ler yerine "group name"ler söz konusu edilir. Tabi işletim sisteminin çekirdeği "user id" ve "group id" sayısal değerleriyle çalışmaktadır.

UNIX/Linux sistemlerinde her dosyanın ayrıca bir "user id" ve "group id" si vardır.

Bir dosyanın erişim haklarının ilk üç karakterine "owner", sonraki üç karakterine "group" ve sonraki üç karakterine "other" denilmektedir:



İlgili erişim hakkı varsa r, w, x sembolleri yoksa '-' sembolü bulunmaktadır. Örneğin:

- rw- r-- ---

Burada owner'a "rw", gruba yalnızca "r" ve other'a hiçbir hak verilmemiştir.

Erişim hakları open fonksiyonu atrafından kontrol edilmektedir. Ve algoritması şöyledir (maddeleri else-if olarak değerlendiriniz):

1) Önce open kendisini çağıran prosesin user id'si 0 mı diye bakar. Sıfır numaralı user id'ye sahip prosese "super user" ya da "priviled user" ya da "root user" denilmektedir. Eğer dosyaya erişmek isteyen prosesin user id'si 0 ise erişim kabul edilir. 0 numaralı user id'ye sahip proses hiçbir engele takılmaz.

2) open fonksiyonu dosyaya erişmek isteyen prosesin user id'si dosyanın user id'si ile aynı mı diye bakar. (Yani erişim yapmak isteyen proses dosyanın sahibi midir?). Eğer böyleyse ilk üçlük (owner) erişim hakkını dikkate alır.

3) open fonksiyonu dosyaya erişmek isteyen prosesin group id'si dosyanın group id'si ile aynı mı diye bakar. (Yani erişim yapmak isteyen proses dosyanın grubuyla aynı gruptan mıdır?). Eğer böyleyse ikinci üçlük (group) erişim hakkını dikkate alır.

4) Bu durumda open other erişim hakkını dikkate alır.

Örneğin aşağıdaki haklara sahip "x.txt" dosyasını open ile açmaya çalışalım:

- rw- r-- r--

if ((fd = open("x.txt", O\_RDWR)) < 0) {

perror("open");

exit(EXIT\_FAILURE);

}

Eğer bu çağrıyı dosyanın sahibi yaparsa open başarılı olur. Aynı gruptan kişi ya da herhangi bir yaparsa başarısız olur.

Peki bir dosyanın user id'sinin group id'sinin ve erişim haklarının nasıl olacağına kim nasıl karar vermektedir? İşte aslında bunlara dosyayı open fonksiyonuyla yaratn proses karar vermektedir. Şöyle ki:

1) Yeni yaratılan dosyaının user id'si her zaman onu yaratan prosesin user id'si olarak alınır. (Yani dosyayı hangi user yaratmışsa dosyanın user id'si de o olacaktır.)

2) Yeni yaratılan dosyanın group id'si POSIX standartlarında iki seçenekten biri olarak verilebilir:

- Dosyayı yaratan prosesin group id'si olarak

- Dosyanın içinde bulunduğu dizinin group id'si olarak

Maalesef bu durum POSIX standartları oluşturulduğu sırada izlenen iki farklı stratejiden kaynaklanmaktadır. Linux'ta default olarak yeni yaratılan dosyanın group id'si onu yaratan prosesin group id'si olarak alınmaktadır.

3) Dosyanın erişim hakları dosya yaratılırken open fonksiyonun üçüncü parametresinde verilir. Bu üçüncü parametre aşağıdaki sembolik sabitlerin OR'lanmasıyla oluşturulur

S\_IRUSR

S\_IWUSR

S\_IXUSR

S\_IRGRP

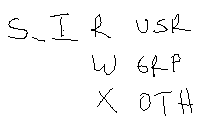
S\_IWGRP

S\_IXGRP

S\_IROTH

S\_IWOTH

S\_IXOTH



Örneğin:

if ((fd = open("x.txt", O\_RDWR|O\_CREAT, S\_IRUSR|S\_WUSR|S\_RGRP)) < 0) {

perror("open");

exit(EXIT\_FAILURE);

}

Open fonksiyonu başarı durumunda dosya betimleyicisi (file descriptor) denien handle değerine başarısızlık durununda -1 değerine geri döner.

**Anahtar Notlar:** Bilgisayar için iki önemli online sözlük kullanılabilir:

1) FOLDOC: Burada bilgisayara ilişkin terimler ve anlamları bulunmaktadır (http://foldoc.org/)

2) DADS: Bu veri yapıları ve algoritnmalara ilişkin sözlğktür (http://xlinux.nist.gov/dads/)

UNIX/Linux sistemlerinde dosyadan okuma yapmak için read isimli POSIX fonksiyonu, dosyaya yazma yapmak için write isimli POSIX fonksiyonu kullanılmaktadır.

#include <[unistd.h](http://linux.die.net/include/unistd.h)>

ssize\_t read(int fd, void \*buf, size\_t count);

ssize\_t write(int *fd*, const void \**buf*, size\_t *count*);

Fonksiyonların birinci parametreleri dosya betimleyicisini alır. İkinci parametreler bellekteki transfer adresini son parametreler ise yazılacak ya da okunacak byte sayısını belirtir. read fonksiyonu okuyabildiği byte sayısına geri döner. Dosyada kalandan daha fazla byte okunmak istenebilir. Bu durumda read okuyabildiği kadarını okur. write fonksiyonu da yazabildiği byte sayısıyla geri döner. Her iki fonksiyon da başarısızlık durumunda -1 değerine geri dönmektedir.

UNIX/Linux sistemlerinde dosya göstericisini konumlandırmak için lseek fonksiyonu kullanılmaktadır.

#include <unistd.h>

off\_t lseek(int fd, off\_t offset, int whence);

Fonksiyonun birinci parametresi dosya betimleyicisini, ikinci parametresi konumlandırmaoffsetini, üçüncü parametresi ise konumlandırma orijinini belirtir. (Genel kullanımı fseek gibidir.). Fonksiyon başarı durumunda konumlanma offset'ine başarısızlık durumunda -1 değerine geri döner. lseek ile dosya göstericisi dosyanın uzunluğunun ötesine konumlandırılabilir. Bu durumda ilk yazma yapıldığında dosya deliği (file hole) oluşur.

Nihayet dosya close fonksiyonuyla kapatılır:

#include <[unistd.h](http://linux.die.net/include/unistd.h)>

int close(int fd);

#include <stdio.h>

#include <stdlib.h>

#include <fcntl.h>

#include <sys/types.h>

#include <unistd.h>

int main(void)

{

int fd;

char str[] = "This is a test\n";

if ((fd = open("test.txt", O\_RDWR | O\_CREAT, S\_IRUSR | S\_IWUSR | S\_IRGRP)) < 0) {

perror("open");

exit(EXIT\_FAILURE);

}

lseek(fd, 0, SEEK\_END);

if (write(fd, str, sizeof(str)) < 0) {

perror("write");

exit(EXIT\_FAILURE);

}

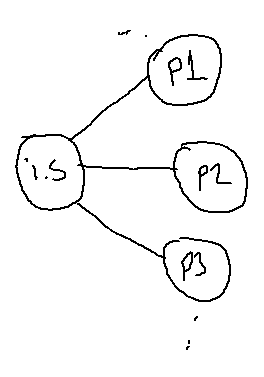
close(fd);

return 0;

}

**İşletim Sistemlerinde Zaman Paylaşımlı Çalışma**

Eskiden thread'ler yokken her prosesin tek bir akışı vardı. İşte örneğin C programlama dilinde tipik olarak proses akışı main fonksiyonundan başlatılmaktadır. Pekiyi çokprosesli işletim sistemlerinde tek bir CPU olduğu durumda birden fazla program nasıl aynı anda çalışabilmektedir? Aslında programlar tek CPU'lu bir sistemde aynı anda çalışmamaktadır. Bir proses CPU'ya atanır. Bir süre çalıştırılır. Sonra çalışmasına ara verilir. Diğeri yine bir süre çalıştırılır. Çalışma bu biçimde devam ettirilir. Sonra prosesler kaldıkları yerden hep böyle çalıştırılırlar. Dışarıdan bakıldığında sanki, bunlar aynı anda çalışıyormuş gibi bir izlenim edinilmektedir. Fakat aslında "biraz ondan biraz bundan" biçiminde zaman paylaşımlı bir çalışma söz konusudur.



Bir prosesin parçalı çalışma süresine quanta süresi ya da quantum denilmektedir. Örneğin Windows sistemlerinde tipik quanta süresi 20 ms. civarındadır. UNIX/Linux sistemlerinde 60 ms. gibi quanta süreleri tercih edilmektedir. Quanta süreleri işletim sisteminden işletim sistemine hatta versiyondan versiyona değişebilir. Sistem yöneticisini genel olarak bunu ayarlama durumunda değildir.

**Thread'lerle İşlemler**

Thread sözcüğü etimolojik olarak akışların ipliğe benzetilmesinden hareketle "iplik" sözcüğünden gelmektedir. Thread'ler bir prosesin bağımsız çizelgelenen akışlarını belirtir. Proses çalışmakta olan programın tamamaını kavramsal olarak anlatmaktadır. Thread ise yalnızca bir akış belirtir. Dolayısıyla thread'ler proses kavramının içerisinde yer alır.