İŞLETİM SİSTEMLERİ
DERSİ ÖDEVİ

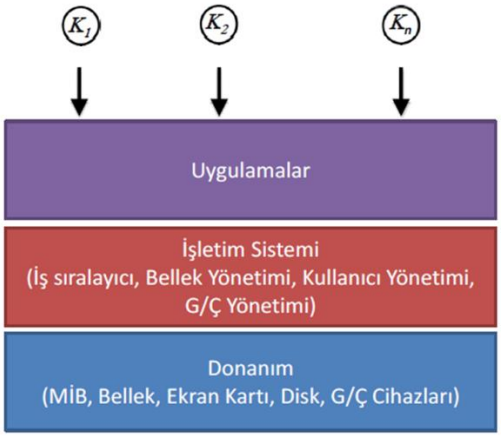
SİSTEM ÇAĞRILARI

1200606074
ERSOY MEVSİM

DERSİN SORUMLUSU:
Arş. Gör. Dr. RABİA KORKMAZ TAN

1. GİRİŞ

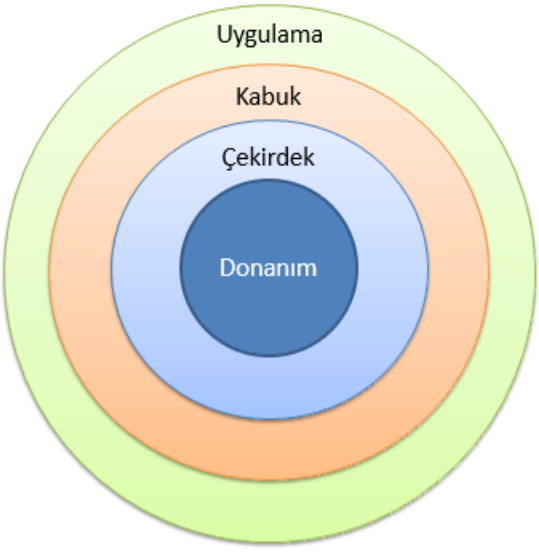
Bilişim sistemleri donanım ve yazılım alt sistemlerinden oluşmaktadır.



Donanım-Yazılım Sistemleri

Yazılım alt sistemi donanım üzerine kuruludur ve farklı mimarilerde tasarlanmış olabilir. Yazılım alt sisteminin en önemli parçasını işletim sistemi oluşturur. İşletim sisteminin temel görevi sistem kaynaklarını kullanıcılar ve uygulamalar arasında paylaştırmaktır. Sistem kaynaklarından en önemlisi işlemcidir. İşletim sisteminde çalışan uygulamaları proses olarak adlandırıyoruz. İşletim sistemi işlemciyi prosesler arasında zamanda paylaştırır. İşletim sisteminin en önemli bileşeni çekirdektir. Çekirdek çok sayıda servisten oluşur: İş sıralayıcı, bellek yöneticisi, kullanıcı yöneticisi, giriş/çıkış yöneticisi vb.

Uygulamalar, sistem çağrıları aracılığı ile işletim sisteminin çekirdeğiyle konuşur. Sıradan kullanıcı için sistem çağrıları oldukça karmaşıktır. Bu karmaşıklığı kullanıcıdan gizlemek için çekirdeği çevreleyen kabuk olarak adlandırılan bir katman yer alır. Kabuğun görevi kullanıcıdan aldığı komutları çekirdeğin anlayacağı bir dizi sistem çağrısına dönüştürmektir. Kabuğu da çevreleyen bir uygulama katmanı bulunur.



İşletim Sisteminin Katmanlı Mimarisi

2. SİSTEM ÇAĞRILARI

İşletim sistemi ile kullanıcı programları arasında tanımlı olan ara yüzdür, işletim sistemi tarafından tanımlanan bir prosedürler kümesidir. İşletim sistemi tarafından tanımlanan bu prosedürlere sistem çağrıları (system calls) denilir.

Tanımlı olan sistem çağrıları kümesi işletim sistemlerinde farklı olabilir. İsim olarak farklı olmasına rağmen arka planda gerçekleştirilen işlemler benzerdir.

Kullanıcı programları donanım ile ya da özel işlemler için bu tanımlı sistem çağrılarını kullanırlar. Sistem çağrıları sayesinde yazılımcı doğrudan donanıma müdahale etmez. Donanım üzerinde gerçekleştireceği işlemi sistem çağrısı kullanarak gerçekleştirir. Bu sayede olası sistem hatalarından kaçınılmış olur.

UNIX, Linux türevi işletim sistemlerinde bulunması gerekli olan sistem çağrıları IEEE tarafından POSIX standartları olarak belirlenmiştir. UNIX sistemlerde genellikle sistem çağrısı ile çağrılacak olan kütüphane fonksiyonunun ismi aynıdır.

Örneğin dosya açmada kullandığımız **"open"** fonksiyonu bir POSIX fonksiyonudur. Linux'ta, MAC OS' da vs. bulunur.

Windows' ta durum biraz farklıdır. Microsoft Win32 API (Application Program Interface) adını verdiği bir prosedür kümesi tanımlamıştır.

Sistem çağrılarının, sistemden sisteme değişiyor olması yazılımcının işini zorlaştırır. Bu değişiklik, bir Windows işletim sistemi için yazılan programın başka bir versiyondaki Windows işletim sistemi üzerinde çalışamamasına sebep olur. Bu yüzden programcılar, doğrudan sistem çağrılar yerine API (application program interface) kullanmayı tercih ederler. Bu ara yüzler (API' ler) tüm Windows işletim sistemleri tarafından genel ölçüde desteklenmektedir.

API' ler programın her Windows işletim sistemi üzerinde çalışabilmesini sağlar. API kullanımı, her sistemin sistem çağrılarını ezbere bilmeyi gerektirmez. Birden fazla sistem işlemini kısa fonksiyonlarla yapabilmeyi sağlar. Programcı API fonksiyonunun içeriğiyle ilgilenmez. Fonksiyonun parametrelerini ve dönüş değerini bilmesi yeterlidir.

Microsoft, sistem çağrılarını kullanıcı programlarından API kullanımı ile izole ederek sahip olduğu işletim sistemleri üzerinde değişiklik yapabilme imkanını elinde bulundurur. İşletim sistemleri değişse bile bir kullanıcı programı API' ler aracılığı ile bütün Windows işletim sistemlerinde sorunsuzca çalışabilmektedir.

Win32 API çağrılarının sayısı binlerle ifade edilecek kadar çoktur.

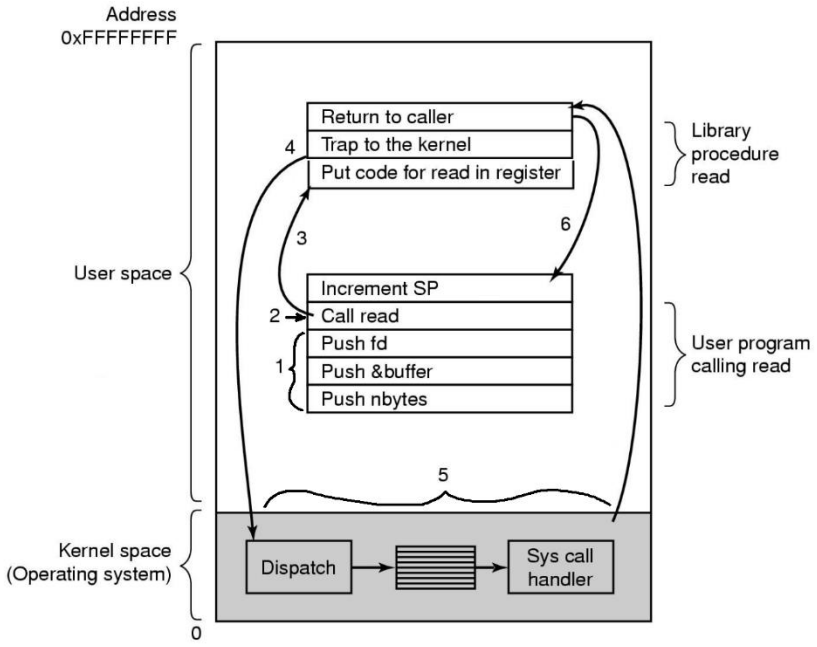
Bir sistem çağrısında gerçekleştirilecek olan işlemler makineye tamamen bağımlıdır bu yüzden sıklıkla assembly programlama dili ile tanımlanırlar. C ve diğer programlama dillerinden bu prosedürleri kullanmak için prosedür kütüphaneleri tanımlanır.

İşlemci aynı anda tek komut(instruction) çalıştırabilir. Eğer kullanıcı kipinde çalışan bir kullanıcı programı bir servise ihtiyaç duyarsa, örneğin bir dosyayı okumak isterse, bir sistem çağrısı komutu (system call instruction) çalıştırarak kontrolü işletim sistemine vermelidir. İşletim sistemi programın ne istediğini parametreleri inceleyerek belirler, sistem çağrısını yerine getirir ve kontrolü sistem çağrısını çağıran programa tekrar geri verir.

Örneğin, Unix sistemlerde bulunan read sistem çağrısını inceleyelim. Üç adet parametre alır. Birinci dosyayı belirtir, ikinci parametre tamponu belirtir ve üçüncü parametrede okunacak byte sayısını verir.

sonuc = read(fd, tampon, okunacakByteSayısı);

Eğer sistem çağrısı doğru çalışmamışsa geriye -1 çevrilir.



1. Parametreler yığına(stack) konulur.

2. read sistem çağrısı için read kütüphane fonksiyonu çağrılır.

3. yazmaca read sistem çağrısına karşılık gelen değer konulur.

4. TRAP ile kontrol çekirdeğe verilir.

5. çekirdek hangi sistem çağrısının gerçekleştirileceğine parametre ve yazmaçtaki değeri inceleyerek karar verir. Uygun sistem çağrısı işleyicisini çalıştırır. Sistem çağrısı bittikten sonra çağıran kütüphane fonksiyonunu geri dönülür.

6. Kütüphane fonksiyonu da kendisini çağıran kullanıcı programına geri döner.

2.1. Sistem Çağrısı Çeşitleri

- İşlem Kontrolü
- Dosya Yönetimi
- Cihaz Yönetimi
- Bilgi Sağlama
- İletişim
- Koruma

Windows ve Linux'da sistem çağrısı Örnekleri

	Windows	Unix

Process Control	`CreateProcess()`
	`ExitProcess()`
	`WaitForSingleObject()`
		`fork()`
		`exit()`
		`wait()`

File Manipulation	`CreateFile()`
	`ReadFile()`
	`WriteFile()`
	`CloseHandle()`
		`open()`
		`read()`
		`write()`
		`close()`

Device Manipulation	`SetConsoleMode()`
	`ReadConsole()`
	`WriteConsole()`
		`ioctl()`
		`read()`
		`write()`

Information Maintenance	`GetCurrentProcessID()`
	`SetTimer()`
	`Sleep()`
		`getpid()`
		`alarm()`
		`sleep()`

Communication	`CreatePipe()`
	`CreateFileMapping()`
	`MapViewOfFile()`
		`pipe()`
		`shm_open()`
		`mmap()`

Protection	`SetFileSecurity()`
	`InitlializeSecurityDescriptor()`
	`SetSecurityDescriptorGroup()`
		`chmod()`
		`umask()`
		`chown()`

UNIX	Win32	Description
fork	CreateProcess	Create a new process
waitpid	WaitForSingleObject	Can wait for a process to exit
execve	(none)	CreateProcess = fork + execve
exit	ExitProcess	Terminate execution
open	CreateFile	Create a file or open an existing file
close	CloseHandle	Close a file
read	ReadFile	Read data from a file
write	WriteFile	Write data to a file
lseek	SetFilePointer	Move the file pointer
stat	GetFileAttributesEx	Get various file attributes
mkdir	CreateDirectory	Create a new directory
rmdir	RemoveDirectory	Remove an empty directory
link	(none)	Win32 does not support links
unlink	DeleteFile	Destroy an existing file
mount	(none)	Win32 does not support mount
umount	(none)	Win32 does not support mount
chdir	SetCurrentDirectory	Change the current working directory
chmod	(none)	Win32 does not support security (although NT does)
kill	(none)	Win32 does not support signals
time	GetLocalTime	Get the current time

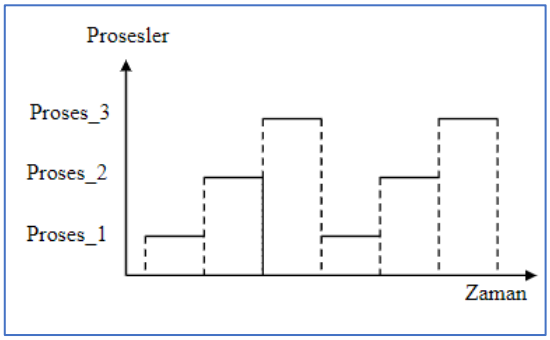
3. PROSESLER

İşletim sisteminin temel görevinin başta işlemci olmak üzere sistem kaynaklarını paylaştırmaktır. İşletim sistemi tasarlanırken bir dizi genel tasarım hedefleri gözetilir. Bu hedeflerin başında çok görevlilik gelir. İşletim sistemi üzerinde çok sayıda uygulamanın çalışması istenir. Bu çalışan uygulamalar, işletim sisteminde, proses olarak adlandırılır. Proses, çalışır durumda olan programa verilen bir isimdir. CPU tarafından işletilmeyen bir program pasif bir yapıya sahip olup komutlar dizisinden ibarettir. Bu komutlar işletime alınmak suretiyle işlevlerini yerine getiren proseslere dönüşmüş olurlar.

Proses yaratıldığında bellekte proses için yer ayrılır. Bunun dışında işletim sistemi çekirdeği her bir proses için proses ile ilgili bilgileri sakladığı bir tablo oluşturur. Bu **tablo proses** tablosu olarak adlandırılır. Bu tablolar, prosesin ana bellekteki yeri, proses aktivitesi, program sayacının değeri, işlemcinin kaydedicileri, tahsis edilen kaynaklar, açık durumdaki dosyalar, hafıza limitleri gibi bilgileri içerirler.

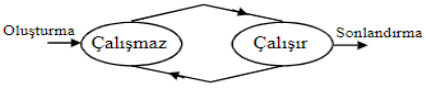
3.1. Proses Modeli

Burada işletim sistemi, tek ya da çok çekirdekli-işlemcili bir sistem üzerinde çalışıyor olabilir. Eğer tek işlemci ya da çekirdek varsa aynı anda birden fazla proses çalışamaz. Çekirdek içinde proses sıralayıcı tarafından yönetilen bir proses kuyruğu bulunur. Her proses belirli bir süre işlemciyi kullanır ve süresi dolduğunda işlemciyi terk eder. Bazen bir prosesin zamanı dolmasa da işlemciyi terk etmesi gerekebilir. İşlemciyi terk eden işi bitmemiş proses kuyruktaki yerini alır ve sıra tekrar kendisine gelen kadar bekler. Böylelikle prosesler işlemciyi zamanda paylaşarak ve vakit buldukça çalışarak ellerindeki işleri tamamlamaya çalışırlar. Birden fazla çekirdek ya da işlemcinin olduğu sistemlerde, çekirdek sayısı kadar kuyruk bulunur ve çekirdek sayısı kadar proses paralel olarak çalışabilir.

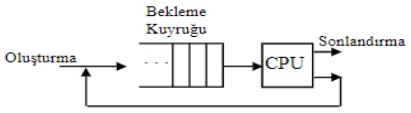


3.1.1. İki durumlu proses modeli:

Herhangi bir zaman diliminde bir proses CPU tarafından ya işletiliyordur ya da işletilmiyordur. Yani proses ya çalışır ya da çalışmaz durumdadır.



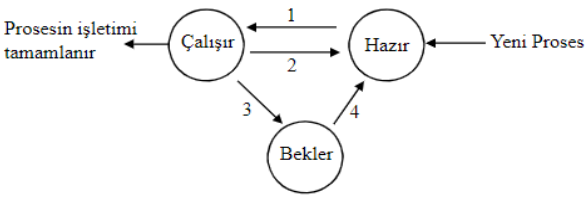
Durum geçişleri

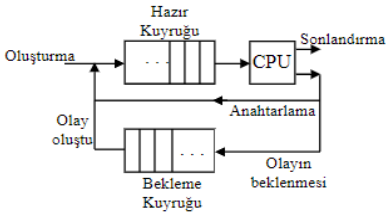


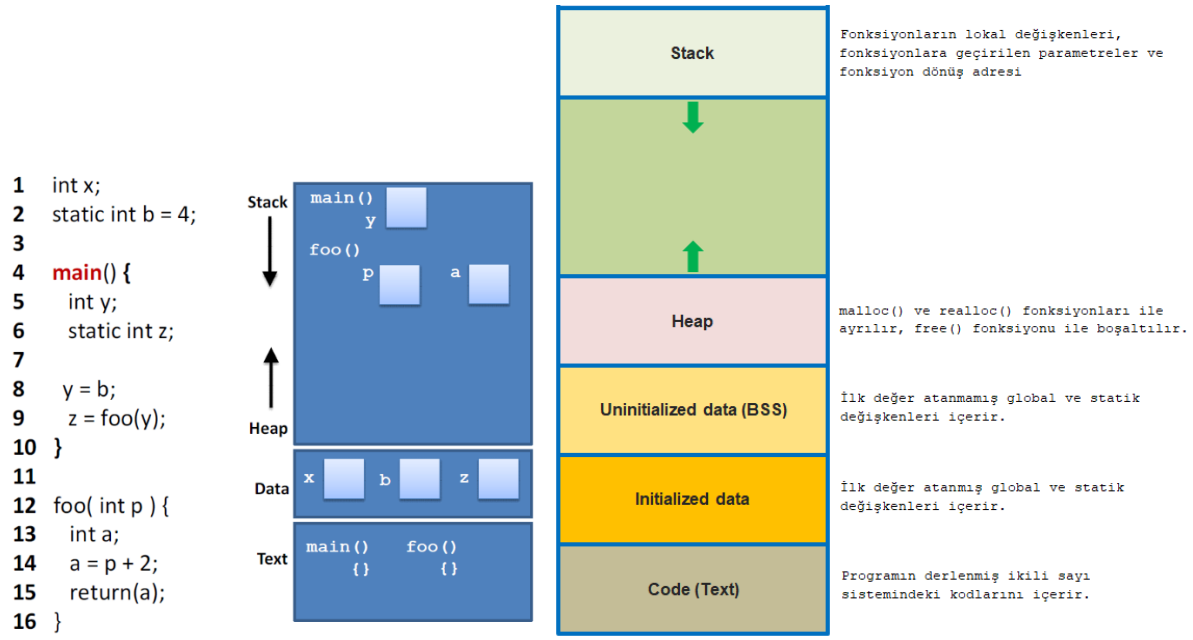
Kuyruk diyagramı

3.1.2. Üç durumlu proses modeli:

Eğer tüm prosesler çalışmaya hazır durumundaysa, kuyruk yapısı ikidurumlu modeldeki gibi düşünülebilir. Fakat giriş/çıkış işleminin sonlanmasını bekleyen bir proses olduğunda kuyruktan sıra ile prosesleri işletime sokmak mümkün olmayacaktır. İki durumlu modeldeki çalışmaz durumu, hazır ve bekler durumu olarak genişletilebilir.



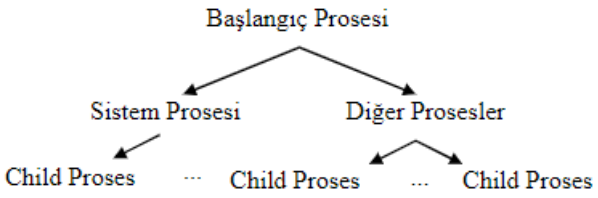




Proses Bellek Modeli

3.2. Parent (anne) ve child (çocuk) proses

Her bir uygulamanın çalıştırılması için bir prosesin yaratılması gerekir. Ama bir uygulama çok prosesli olabilir. Bir proses bir ya da daha fazla çocuk proses yaratabilir. Böylelikle prosesler arasında ebeveyn–çocuk ilişkisi yaratılmış olunur. Bir proses tarafından oluşturulan proses **child** proses, **child** prosesi üreten proses de **parent** proses olarak isimlendirilir. Proses hiyerarşisinde **parent** proses çok sayıda **child** prosese sahip olabilirken **child** proses sadece bir **parent** prosese aittir.



Genellikle işletim sistemi yüklendiğinde bir başlangıç prosesi otomatik olarak oluşturulur ve bu proses tüm proses hiyerarşisinin kökü olur. Bir proses oluşturulduğunda bu köke eklenir.

3.2.1. Çocuk Proses Oluşturulması

Bir proses, **proses oluşturma sistem çağrısını** kullanarak çocuk proses yaratabilir. **Proses oluşturma sistem çağrısı** prosesin bellek görüntüsünün bire bir kopyasını çıkarır. Bu nedenle sistem çağrısı ile proses yaratmak maliyetlidir. Diğer bir maliyet anahtarlamada oluşur. İşlemciyi kullanan proses süresi dolup işlemciyi terk ederken sıra tekrar kendisine geldiğinde kaldığı yerden devam edebilmesi için anahtarlama bilgileri saklanması gerekir. Bu bilgiler saklayıcılar, yığın göstergesi, program sayacı gibi işlemci içindeki elemanlardan oluşur. Bunlar bellekte saklanır. Böylelikle sıra tekrar kendisinde geldiğinde, prosesin bellekte saklanan bu bilgileri işlemciye geri yüklenerek, prosesin kaldığı yerden devam etmesi sağlanır. Prosesin anahtarlama bilgileri kalabalık olduğu için anahtarlama maliyeti de yüksektir. Bu yüzden çok prosesli yapıya farklı bir seçenek olarak çok iplikli (=thread) yapılar kullanılabilir. Bir iplik yaratıldığında onun için sadece yeni bir yığın yaratılır. Bir iplik prosesin heap, text ve data alanlarını paylaşır. Her ipliğin ise kendi yığını vardır.

Linux'da uygulama geliştirmek için aşağıdaki modellerden biri kullanılır.

i. Çok Prosesli Programlama Modeli
ii. Tek Prosesli Çok İplikli Programlama Modeli
iii. Çok Prosesli Çok İplikli Programlama Modeli

Bu modellerin birbirlerine göre kazanım ve yitimleri vardır. Çok prosesli modelin yitimi proses yaratma ve prosesler arasında bağlam anahtarlama maliyetidir. Çok iplikli programlamada iplikler işlemciyi daha verimli kullanır. Üstelik iplik yaratmak daha düşük maliyeti vardır. Buna karşılık ipliklerden biri başarısız olursa uygulama da başarısız olur ve sonlanması gerekir. Bu nedenle genellikle üçüncü model tercih edilir. Kritik görevler ayrı prosesler olarak kodlanır. Her bir proses ipliklerden oluşur.

3.2.2. Linux'da Proses Yaratılması

Linux'da proses yaratmak için **for**k() sistem çağrısını kullanıyoruz. Bu fonksiyon asıl prosesin bire bir kopyasını oluşturur (dosya tanımlayıcıları, yazmaçlar,... herşey). Kopyalama işleminden sonra iki proses (ana ve çocuk) birbirlerinden tamamen ayrılırlar. Kullandıkları veriler kendilerine özgü olur. (Fakat programın text kısmı aynı olduğu için iki proses tarafından paylaşılır).

fork sistem çağrısı yapıldığında çekirdeğin yürüttüğü işlemler:

- Proses tablosunda yer ayırılır
- Çocuk prosese sistemde yeni bir kimlik numarası atanır
- Anne prosesin kopyası çıkarılır
- Dosya erişimi ile ilgili sayaçları düzenlenir
- **fork()** sistem çağrısı, anneye çocuğun kimliğini, çocuğa da 0 değerini döndürür. **fork()** sistem çağrısı, eğer proses yaratılırken hata durumu oluşursa, -1 değerini döndürür.

Ana proses çocuk prosesin bitmesini beklemek isterse ya da beklemesi gerekli ise **waitpid**() isimli sistem çağrısını kullanır.

Linux'da pid ler için, <sys/types.h> dosyasında tanımlı olan pid_t veri tipi kullanılır. **fork()** gibi çeşitli proses yönetim fonksiyonları <unistd.h> kütüphanesinde tanımlıdır.

- **getpid()**: geriye prosese kendi numarasını verir.
- **getppid()**:çalışan prosesin ana proses numarasını verir. (get_parent_program_id)

Prosesten çıkmak için **exit()** sistem çağrısı kullanılır.

Örnek1: `fork()` sistem çağrısının kullanıldığı basit bir örnek aşağıda verilmiştir.

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/wait.h>

void gotoxy(int x,int y)
{
    printf("%c[%d;%df",0x1B,y,x);
}

int main (void){
    pid_t f;
    int status,sutun=1;
    printf("Program çalışıyor: Kimliğim= %d\n", getpid());
    f=fork();
    if (f==0) /*çocuk*/
    {
        sutun=45; sleep(4);
        gotoxy(sutun,4);    printf("Ben çocuk. Kimliğim= %d", getpid());
        gotoxy(sutun,5);    printf("Annemin kimliği= %d", getppid());
        //exit(0);
    }
    else if (f>0)/* anne */
    {
        sleep(1);
        printf("\nBen anne. Kimliğim= %d", getpid());
        printf("\nAnnemin kimliği= %d", getppid());
        printf("\nÇocuğumun kimliği= %d\n", f);
        if (waitpid(f, &status, 0) == -1) {
            perror("waitpid");
        }
        sleep(2);
        //exit(0);
    }

    gotoxy(sutun,7); printf("Bitti.");
    gotoxy(sutun,8); printf("------------------------------");

    exit(0);
}
```

Örnek2: `CreateProcess()` sistem çağrısının kullanıldığı basit bir örnek aşağıda verilmiştir.

```
#include <windows.h>
#include <stdio.h>
int main(void)
{
	STARTUPINFO si;
	PROCESS_INFORMATION pi;
	ZeroMemory(&si, sizeof(si));
	si.cb = sizeof(si);
	ZeroMemory(&pi, sizeof(pi));

	// const TCHAR* target = _T("c:\\WINDOWS\\system32\\calc.exe");
	const TCHAR* target = _T("C:\\Program Files\\Microsoft Office\\root\\Office16\\Excel.exe");

	// Start the child process. If not OK...
	if(!CreateProcess(target, // module name.
		NULL, // Command line.
		NULL, // Process handle not inheritable.
		NULL, // Thread handle not inheritable.
		FALSE, // Set handle inheritance to FALSE.
		0, // No creation flags.
		NULL, // Use parent's environment block.
		NULL, // Use parent's starting directory.
		&si, // Pointer to STARTUPINFO structure.
		&pi) // Pointer to PROCESS_INFORMATION structure.
	)
		// Then, give some prompt...
		printf("\nSorry! CreateProcess() failed.\n\n");
	else
	{
		printf("\nWell, CreateProcess() looks OK.\n");
	}

	//-- wait for the process to finish
	while (true)
	{
		 //-- see if the task has terminated
		 DWORD dwExitCode = WaitForSingleObject(pi.hProcess, 0);
		 if (   (dwExitCode == WAIT_FAILED   )
		    || (dwExitCode == WAIT_OBJECT_0 )
		    || (dwExitCode == WAIT_ABANDONED) )
		 {
			break;
		 }
	}
	// Close process and thread handles. If the function succeeds, the return value is nonzero.
	// If the function fails, the return value is zero.

	if(CloseHandle(pi.hThread) != 0)
	printf("The thread handle has been closed successfully\n");

	if(CloseHandle(pi.hProcess) !=0)
	printf("The process handle has been closed successfully\n");

	ExitProcess(0);
}
```

4. İPLİKLER (İş parçacıkları – Lifler – Thread)

Proses model iki temel kavram üzerine kurulmuştur: Kaynakların gruplandırılması ve programın icrası.

Her bir proses ana bellek üzerinde kendine ait bir adres alanına sahiptir. Bu adres alanında program metni, veriler ve prosesin çalışması için gerekli olan kaynakların bilgileri yer alır. Bu kaynaklar açık olan dosyalar, çocuk prosesler, bekleyen alarmlar, sinyal bilgileri, hesaplama için kullanılan parametreler vb. bilgilerdir.

Proses çalışması için gerekli olan bütün kaynakları proses kendi adres bölgesinde gruplandırarak, daha hızlı çalışan ve daha kolay yönetilebilen bir forma sokulmuş olur.

Diğer kavram ise, prosesin icra edilen kısmı ki, bu kısma thread (iplik-iş parçacığı) adı verilmektedir.

İplik, bir sonraki komutun adresini tutan bir program sayacına, çağrılmış ama daha sonlanmamış her bir prosedür (fonksiyon) için ayrı bir alan tutan yığına ve işlemci içindeki registerlardaki değerleri saklayan register değişkenlerine sahiptir.

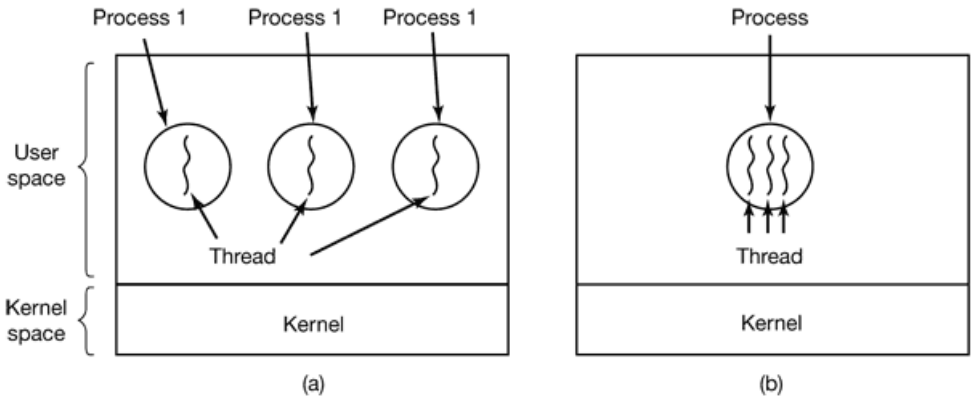
Prosesler kaynakları gruplandırmak için kullanılırken, iplikler işlemci içerisine giren ve prosesin işlem kısmını icra eden kod parçalarıdır.

İplikler, bir proses içinde birden fazla işlem biriminin çalışmasına olanak sağlayan yapılardır.

Tek bir proses içerisinde çoklu ipliklerin paralel (multi-threading) çalışması, multiprogramming yapısına sahip bir işletim sisteminde birden fazla prosesin aynı anda çalışmasına benzerdir.

Bir proses içerisindeki tüm iplikler, aynı adres uzayını, açık dosyaları, genel değişkenler gibi kaynakları paylaşırlar. Prosesler ise fiziksel belleği, diskleri, yazmaçları ve diğer genel kaynakları paylaşırlar.

Aşağıdaki Şekil a'da her biri tek ipliği içeren üç farklı proses gösterilmektedir. Bu proseslerin her biri ayrı bir adres alanına sahiptir. Şekil b'de ise üç ipliğe sahip bir proses gösterilmiştir. Şekil b'deki iplikler aynı prosese ait oldukları için aynı adres alanını kullanırlar.



İpliklerin yönetiminde kullanılan birçok sistem çağırısı bulunmaktadır.

Bu sistem çağrılarından bazıları: **thread_create, thread_exit, thread_wait, thread_yield**.

4.1. PTHREAD Kütüphanesini Kullanarak İplik Yaratmak

PThread Kütüphanesi kullanarak iplik yaratmak için pthread_create fonksiyonunu kullanıyoruz:

```
#include <pthread.h>
int pthread_create(pthread_t *thread,
                   const pthread_attr_t *attr,
                   void*(*start_routine)(void*),
                   void *arg)
```

Burada birinci parametre ipliğin kimliğini saklamak istediğimiz bellek gözünün adresini ve ikinci parametre ise yaratmak istediğimiz ipliğin özelliklerini almaktadır. Üçüncü parametre ile ipliğin çalıştırmasını istediğimiz fonksiyonun adresini ve dördüncü parametre ile bu fonksiyona geçirmek istediğimiz parametrenin değerini veriyoruz. PTHREAD kütüphanesindeki tüm fonksiyonlar ve veri yapıları **pthread_** ön eki ile başlar.

Aşağıdaki uygulamada 5 adet iplik yaratılmaktadır. İplikler çalışmaya başladıktan sonra ekrana bir ileti göndermektedir. Ebeveyn proses ise bu beş ipliğin işini bitirmesini beklemek üzere **pthread_join** çağrısı yapmaktadır. Yaratılan her ipliğe ulaşmak üzere **pthread_t** tipinde bir veri yapısı iliştirilir. Tüm PTHREAD kütüphane fonksiyonlarda iplik üzerinde işlem yapmak amacı ile referans olarak bu değer kullanılır.

Yaratılan ipliğin yürüteceği fonksiyon herhangi bir tipten değer alabilir ve benzer şekilde herhangi bir tipten değer döndürebilir. Parametre aktarımı **pthread_create** çağrısı ile iplik yaratılırken gerçekleşir:

```
pthread_create( &iplik[i],
        NULL,
        metot,
        (void *)(i));
```

İpliğin döndürdüğü değere ise pthread_join çağrısı üzerinden erişilir:

```
pthread_join( iplik[i],(void **)&return_value);
```

ÖRNEK 3:

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <windows.h>
#include <iostream>
using namespace std;

#define NUM_THREADS 5
pthread_t iplik[NUM_THREADS];

void* metot(void * arg) {
        int a= *((int*)(&arg));
        cout << "Thread " << a << endl ;
        //return ((void *)a);
}
```

```
void thread_start() {
	for (int i = 0; i < NUM_THREADS; i++) {
		pthread_create( &iplik[i], NULL, metot, (void *)(i));
	}
}
void thread_wait() {
	for (int i = 0; i < NUM_THREADS; i++) {
		pthread_join( iplik[i],NULL);
	}
}
int main() {
	thread_start();
	thread_wait();
	return 0;
}
```

4.2. C++11'de İplik Yaratmak

C++, 2011 yılında çıkan yeni sürümü ile günümüz modern programlama dillerinde bulunan birçok ileri düzey özelliğe kavuşmuştur. Nihayet paralel programlama için kullandığımız iplikler işletim sistemine bağımlı olmaktan kurtuldu ve dilin bir parçası haline geldi. C++11'de thread kütüphanesi ile gelen thread adında yeni bir sınıf yer alır. C++11'de iplik yaratmanın 3 farklı yolu vardır. Fonksiyon kullanımı, struct kullanımı, λ(lambda) ifadesi kullanımı.

Fonksiyon kullanımı

thread sınıfının kurucu fonksiyonu ilk parametre olarak çalıştırılacak fonksiyonu alır. Aşağıdaki örnekte t1 ve t2 adında iki iplik yaratılır. Bu iplikler hello fonksiyonunu çalıştırırlar.

İplik fonksiyonuna parametre aktarmak mümkündür. Bunun için basitçe thread sınıfının yapıcısına çalıştırılacak fonksiyondan sonra sırayla parametre değerleri geçilir:

thread t1(fonksiyon, parametre değeri)

ÖRNEK 4:

```
#include <iostream>
#include <thread>
#include <windows.h>
using namespace std;

void hello(int a){
	cout << "Thread " <<a<< endl ;
}
int main(){
	thread t1(hello,1);
	thread t2(hello,2);
	t1.join();
	t2.join();
	cout << "Done." << endl;
}
```

5. SİNYAL İŞLEME

Bir sinyal bir prosese gönderilen yazılım kesmesidir. İşletim sistemi, sinyalleri, çalışan bir yazılıma olağandışı durumları raporlamakta kullanır. Bazı sinyaller geçersiz bellek adreslerine erişim gibi durumlarda hata raporlamakta, bazıları da bir telefon hattının kapanması gibi rasgele olayları raporlamakta kullanılır.

C kütüphaneleri her biri başka bir çeşit olaya karşılık olmak üzere çeşitli sinyal türleri tanımlar. Bazı olaylar, bir yazılımın çalışmasını imkansız kılabilir, bu tür olayları raporlayan sinyaller yazılımın çalışmasını durdurmasına sebep olur. Diğer sinyal çeşitleri zararsız olayları raporlar ve bunlar ön tanımlı olarak yok sayılır.

Bir olayın sinyallere sebep olacağını umuyorsanız, sinyalle tetiklenen bir işlev tanımlayıp, işletim sistemine böyle sinyaller geldiğinde bu işlevi çalıştırmasını belirtebilirsiniz.

Son olarak, bir proses başka bir sürece sinyal gönderebilir; bu bir prosesin kendi alt prosesini durdurması gerektiğinde ya da birbiriyle haberleşerek eşzamanlı çalışması gereken prosesler arasında kullanılabilir.

5.1. Bazı Sinyal Çeşitleri

Bir sinyal olağandışı bir olayın varlığını raporlar. Bir sinyale sebep olan (üreten ya da ortaya çıkaran) olayların bazıları:

- Sıfırla bölme ya da geçerli bir aralık dışında bir adres gösterme gibi yazılım hataları.
- Kullanıcı tarafından yazılımın durdurulmak ya da sonlandırılmak istenmesi. Çoğu ortam kullanıcıya C-z tuşlayarak uygulamayı durdurabilme veya C-c tuşlayarak uygulamayı sonlandırabilme imkanı sağlar. Bu tuş vuruşları algılandığında işletim sistemi sürece bu isteği belirten bir sinyal gönderir.
- Bir alt prosesin sonlanması.
- Alarm veya zamanlayıcının zamanaşımına uğraması.
- Aynı proses tarafından yapılan bir kill veya raise çağrısı.
- Başka bir proses tarafından yapılan bir kill çağrısı (sinyallerin prosesler arası iletişim için sınırlı ama kullanışlı biçimidir).
- Yapılamayacak bir G/Ç işlemini yapmaya çalışma.

Bu olayların her biri (açıkça yapılan kill ve raise çağrıları dışında) kendine özel bir sinyal üretir.

Genellikle, sinyalleri üreten olaylar üç ana sınıf altında incelenir:

Hatalar, dış olaylar, doğrudan yapılan istekler.

Doğrudan yapılan istekler, amacı özellikle sinyal üretmek olan **kill** sistem çağrısı ile yapılır.

5.2. Standart Sinyaller

1. Yazılım Hatalarının Sinyalleri - Yazılımınızdaki hataları raporlayan sinyaller.

2. Sonlandırma Sinyalleri - Bir yazılımı durduran ya da sonlandıran sinyaller.

3. Alarm Sinyalleri - Zamanlayıcıların zamanaşımına uğraması.

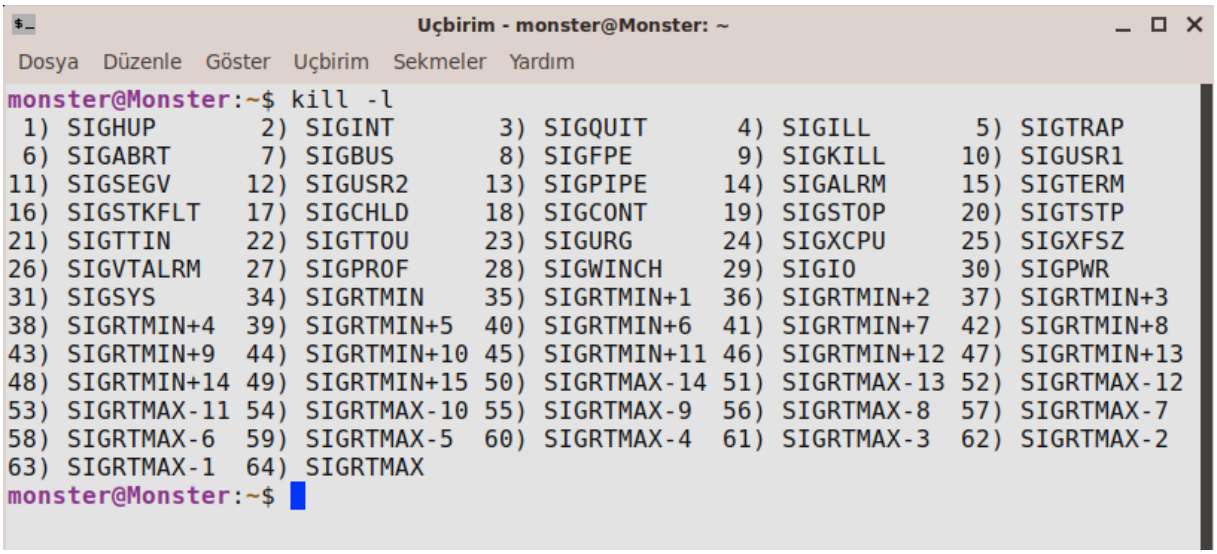
4. Eşzamansız G/Ç Sinyalleri - Girdi varlığını bildiren sinyaller.

5. İş Denetim Sinyalleri - İş denetimi için kullanılan sinyaller.

6. İşlemsel Hata Sinyalleri - İşlemsel sistem hatalarını raporlayan sinyaller.

7. Çeşitli Sinyaller - Muhtelif sinyaller.

8. Sinyal İletileri - Bir sinyali açıklayan bir iletinin basılması.



5.3. Sinyallerin Üretilmesi

5.3.1. Kendine Sinyal Gönderme

Bir sürecin kendine sinyal göndermesi için **raise** sistem çağrısı kullanılabilir. Bu işlev signal.h kütüphane dosyasında bildirilmiştir.

int raise (int sinyalnum)

İşlev çağrıldığı sürece sinyalnum sinyalini gönderir. İşlem başarılı olursa sıfırla, aksi takdirde sıfırdan farklı bir değerle döner. Başarısızlığın tek sebebi sinyalnum değerinin geçersiz olmasıdır.

Örnek 5:

```
#include <iostream>
using namespace std;
#include <signal.h>

/* Bir durdurma sinyali geldiğinde, eylemi önce öntanımlı eyleme
  ayarlayalım, temizliği yaptıktan sonra da sinyali yeniden
  gönderelim. */

void tstp_handler (int sig)
{
        //signal (SIGTSTP, SIG_DFL);
        /* Temizlik işlemleri */
        cout<<"\nTSTP Handler çalıştı."<<sig<<"\n";
        raise (SIGCONT);
}

/* Süreç çalışmaya kaldığı yerden devam edeceği zaman
  sinyal eylemciyi yeniden kuralım. */

void cont_handler (int sig)
{
        cout<<"Sinyal dinleme tekrar basladi.\n";
        signal (SIGCONT, cont_handler);
        signal (SIGTSTP, tstp_handler);
}

/* Her iki eylemciyi de yazılım başlatıldığında etkinleştirelim. */

int
main (void)
{
        cout<<"Program basladi.\n";
        signal (SIGCONT, cont_handler);
        signal (SIGTSTP, tstp_handler);
        cout<<"Sinyal dinleme basladi.\n";

        // işlemler

         getchar();
}
```

5.3.2. Başka Bir Sürece Sinyal Gönderme

kill sistem çağrısı bir sinyalin başka bir sürece gönderilmesi için kullanılabilir. İsmine rağmen, bir sürecin sonlandırılmasına sebep olmaktan farklı bir şeyler yapmak için de kullanılabilir.

kill işlevi signal.h dosyasında bildirilmiştir.

int kill (pid_t pid, int sinyalnum)

kill işlevi pid ile belirtilen proses ya da proses grubuna sinyalnum sinyalini gönderir. Standart sinyallere ilaveten, ayrıca pid kimliğini doğrulamak için sıfır değerini de kullanabilirsiniz.

Bir proses kill (getpid(), sinyal) gibi bir çağrı ile kendisine bir sinyal gönderebilir ve sinyal engellenmez.

Sinyal gönderme başarılı olduğunda kill sıfır ile döner. Aksi takdirde sinyal gönderilmemiş demektir ve -1 ile döner.

kill (getpid (), sinyal) çağrısı raise (sinyal) çağrısı ile aynı etkiye sahiptir.

Örnek 6: child prosesten parent prosese kapanma çağrısının gönderilmesi örneği

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/wait.h>

int sutun=1, devam=1, status;
void gotoxy(int x,int y)
{
   printf("%c[%d;%df",0x1B,y,x);
}
void child_durdu(int sig){
   devam=0;    gotoxy(sutun,7); printf("Child durdu!...\n");
}

int main (void){
   pid_t f;

   printf("Program çalışıyor: Kimliğim= %d\n", getpid());
   f=fork();
   if (f==0) /*çocuk*/
   {
      sutun=45; sleep(3);
      gotoxy(sutun,4);    printf("Ben çocuk. Kimliğim= %d\n", getpid());
      gotoxy(sutun,5);    printf("Annemin kimliği= %d\n", getppid());
      sleep(3);
   }
   else if (f>0)/* anne */
   {
      printf("\nBen anne. Kimliğim= %d", getpid());
      printf("\nAnnemin kimliği= %d", getppid());
      printf("\nÇocuğumun kimliği= %d\n", f);

      signal(SIGCHLD, child_durdu);       while(devam);
      sleep(3);
   }
   gotoxy(sutun,8); printf("Bitti.");
   gotoxy(sutun,9); printf("--------------------------------");
   exit(0);
}
```

Örnek 7: kill sistem çağrısı örneği

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/wait.h>

int sutun=1, devam=1;
void gotoxy(int x,int y)
{
	printf("%c[%d;%df",0x1B,y,x);
}
void sinyal(int sig){
	gotoxy(sutun,7); printf("Child prosesten %d numaralı SIGUSR1 sinyali geldi!...\n", sig);
}

int main (void){
	pid_t f;
	int status;

	printf("Program çalışıyor: Kimliğim= %d\n", getpid());
	f=fork();
	if (f==0) /*çocuk*/
	{
		sutun=55; sleep(3);
		gotoxy(sutun,4);    printf("Ben çocuk. Kimliğim= %d\n", getpid());
		gotoxy(sutun,5);    printf("Annemin kimliği= %d\n", getppid());
		kill(getppid(), SIGUSR1);
		sleep(3);
	}
	else if (f>0)/* anne */
	{
		printf("\nBen anne. Kimliğim= %d", getpid());
		printf("\nAnnemin kimliği= %d", getppid());
		printf("\nÇocuğumun kimliği= %d\n", f);

		signal(SIGUSR1, sinyal);

		if (waitpid(f, &status, 0) == -1) {
		   perror("waitpid");
		}
		printf("\n"); sleep(3);
	}
	gotoxy(sutun,8); printf("Bitti.");
	gotoxy(sutun,9); printf("--------------------------------\n");

	exit(0);
}
```

Dosya ve Dizin Yönetimi

Bellekte sakladığımız verilere uygulama sona erdiğinde ya da makinayı kapattığımızda tekrar ulaşamayız. Bellek kalıcı bir depolama ortamı değildir. Verileri kalıcı olarak saklamak istediğimizde dosya sisteminden yararlanıyoruz. Dosya sistemini ayrıca uygulamaları birbirleri ile tümleştirmek için de kullanabiliriz. Bir uygulama aktarmak istediği veriyi dosyaya yazar, diğer uygulama ise bu verileri aynı dosyadan okur. Dosya burada paylaşılan bir kaynak işlevi görür. Verileri dosyada uygulamanın ihtiyaçlarına göre farklı şekillerde organize edilebilir.

Bir dosya sisteminin, verilerin dosyalarda organize edilmesi dışında başka temel görevleri daha bulunur:

- Dosya yaratma
- Dosya silme
- Dosya açma
- Dosya kapatma
- Dosyayı koruma ve paylaşım
- Dosyaya okuma ve yazma

Dosya Tipleri

Verileri diskte iki farklı şekilde organize edebiliriz:

- **Karakter dizisi**
  Verileri dosyaya karakter dizisi olarak yazdığımızda, diskte kaç bayt yer kaplayacağını, verinin tipi değeri belirler. Örneğin C'de int tipinden bir veri bellekte 32 bit yer kaplar. Ancak bu tipten bir değişkenin değeri 42 ise diskte 2 bayt, değişkenin değeri 1.000.000 ise 7 bayt yer kaplar. Karakter dizisi olarak diske yazılan veriyi herhangi bir metin editörü ile açıp okuyabilir içeriğini düzenleyebilirsiniz.

- **Sekizli dizisi**
  Verileri dosyaya bayt dizisi olarak yazdığımızda, bellekte kaç bayt yer kaplıyor ise veriler diskte de o kadar bayt yer kaplar. Ancak bu sefer dosyanın içeriğini bir metin editörü ile açıp okuyamazsınız. Eğer dosyanın tüketicisi bir yazılım ise bu yöntemi tercih edebilirsiniz.

Dosya diskte iki bölümden oluşur:

- **Veriler**: Karakter dizisi ya da sekizli dizisi olarak organize edilmiş olabilir.
- **İndeks bölümü**: dosyanın tipi, verilerin saklandığı blokların disk üzerindeki fiziksel konumları, haklar, dosyanın sahibi, dosyanın boyutu gibi bilgileri kapsar.

Dosya ve Dizin Yönetimi Çağrıları

UNIX	Win32	Tanımı
open	CreateFile	Bir dosya oluşturur ya da varolan bir dosyayı açar.
close	CloseHandle	Dosyayı kapatır.
read	ReadFile	Dosyadan veri okur.
write	WriteFile	Dosyaya veri yazar.
Iseek	SetFilePointer	Dosya işaretçisini taşır.
stat	GetFileAttributesEx	Çeşitli dosya özelliklerini alır.
mkdir	CreateDirectory	Yeni bir dizin oluşturur.
rmdir	RemoveDirectory	Boş olan bir dizini siler.
link	(none)	Win32 bu özelliği desteklemez. Name2 adında yeni bir dosya oluşturur ve name1'e bağlar.
unlink	DeleteFile	Varolan bir dosyayı siler.
mount	(none)	Win32 mount özelliğini desteklemez Bir dosya sistemi ekler
unmount	(none)	Win32 unmount özelliğini desteklemez Bir dosya sistemi siler.

OPEN sistem çağrısının birden çok işlevi vardır. Bunlardan en önemlisi, program içinde geçen kütük kimliği ile fiziksel kütüğün, işletim sırasında eşleştirilmesidir. **open** sistem çağrısının bir diğer işlevi kütüklere erişimin denetlenmesi ve paylaşımın sağlanmasıdır. Genelde **open** sistem çağrısı işletilirken erişilmek istenen kütüğe (salt okuma, okuma-yazma gibi) ne amaçla erişileceği de belirtilir. Bu yolla işletim sistemi, çağrı parametreleri arasında yer alan erişim istem kodunu, ilgili görev/kullanıcı hakları ile karşılaştırma olanağı bulur. Bu durumda, open çağrısının parametreleri arasında, open(kütük kimliği, erişim türü) gibi, simgesel kütük kimliğinin yanı sıra bir de erişim türü bilgisi yer alır.

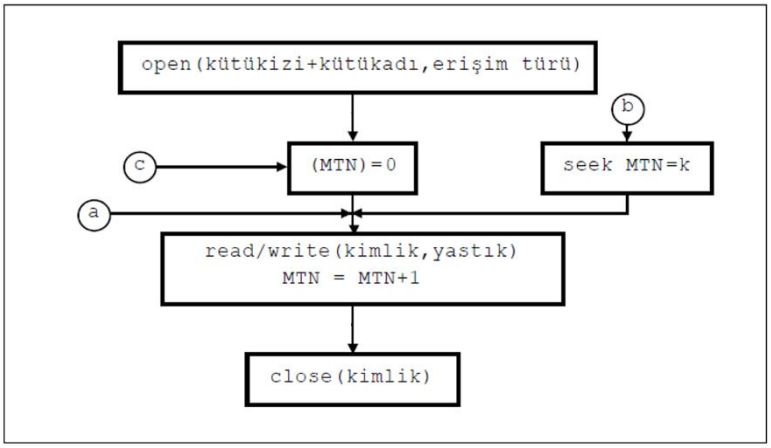
CLOSE sistem çağrısı, open sistem çağrısı ile açılan dosyanın kapatılması işlemini yerine getirir. close sistem çağrısı, open sistem çağrısıyla birlikte düşünülür. open ve close sistem çağrıları, kütüklerin birden çok görev tarafından paylaşılması için gerekli zamanuyumlama düzeneğinin kurulabilmesine de altyapı oluşturabilirler. Aynı kütüğe erişen birden çok görevden open komutunu ilk çalıştıran, ilgili kütüğün erişimini diğer görevlere kapatabilir. İşletim sistemi, open komutunu çalıştıran diğer görevlerin, close komutu işletilene değin ilgili kütüğe ilişkin bekleme kuyruğuna konmasını sağlayabilir. Bu biçimiyle, open ve close sistem çağrıları, üst düzey zaman uyumlama işleçleri gibi de kullanılabilirler.

Sistem çağrıları aracılığıyla gerçekleştirilen kütük işlemlerinin özeti aşağıdaki şekilde verilmiştir. Buna göre herhangi bir kütük üzerinde, herhangi bir işlem yapmadan önce kütük açma işleminin gerçekleştirilmesi, başka bir deyişle program içinde geçen simgesel kütük kimliğinin fiziksel kütük ile eşleştirilmesi gereklidir. Bu işlem bir kez gerçekleştikten sonra,

(a) İlgili kütüğün belirli sayıda tutanağı sıradan okunup yazılabilir

(b) Mantıksal tutanak numarası istenen değerle günlenerek rastgele okuma-yazma işlemi gerçekleştirilebilir. Mantıksal tutanak numarası sıfırlanarak sıradan okuma-yazma işlemleri yinelenebilir.

(c) Kütük üzerinde öngörülen tüm işlemler tamamlandığında kütük kapama işlemi gerçekleştirilir.



Dosya yönetimi için sistem çağrılarının çoğunluğu okuma ve yazma için kullanılır. Bunlardan en önemlileri de read ve write çağrılarıdır. Bu iki çağrı da benzer parametrelere sahiptir.

READ sistem çağrısı üç tane parametreye sahiptir: birincisi dosyayı bildirir, ikincisi buffer belleğin yerini bildirir ve üçüncüsü ise verinin kaç byte uzunluğa sahip olduğunu bildirir.

C programından read sistem çağrısı aşağıdaki şekilde yapılabilir:

```
Data = read(fd, buffer, nbytes)
```

Eğer sistem çağrısı yanlış bir parametre ya da disk hatası nedeniyle çalıştırılamıyorsa, Data değişkeni -1'e ayarlanır. Programlar, eğer bir hata meydana geldiyse bu hatayı görmek için her zaman sistem çağrılarının sonuçlarını kontrol etmelidirler.

WRITE sistem çağrısı da üç tane parametreye sahiptir: Birincisi daha önce açmış olduğumuz dosyayı gösteren bir dosya tanımlayıcısı (File descriptor), ikincisi buffer ise dosyaya yazacağımız verinin adresi, üçüncüsü count ise yazılacak verinin kaç byte olduğunu bildirir.

C programından write sistem çağrısı aşağıdaki şekilde yapılabilir:

```
Sonuc = write(fd, buffer, size_t count);
```

Sistem çağrısı geriye kaç byte yazıldığını döndürür, bir hata oluşmuş ise -1 değerini döndürür.

Dosya İşlemleri

►yaratma / silme

►isim değiştirme

►açma / kapama

►yazma / okuma / ekleme

►dosya işaretçisi konumlandırma

►özellik sorgulama / değiştirme

⇒ sistem çağrıları ile (open, create, read, write, close, ...)

Boş bir klasör oluşturmak için veya silmek için kullanılan **MKDIR** ve **RMDIR** olmak üzere iki çağrı vardır.

LINK çağrısının amacı farklı directory' lerde bulunan iki veya daha çok isim altında gözüken dosyalara izin verir. Tipik bir kullanımı ortak bir dosyayı paylaşmak için aynı programlama grubunun üyelerine izin vermesidir.

CHDIR çağrısı. Bu çağrı, aktif directory'i değiştirmek için kullanılır. Bu sayede uzun dosya isimleri yazmaksızın istenilen dosyaya kolayca ulaşılabilir

UNIX'te her dosya koruma için bir moda sahiptir. Mod, kendisi, grup ve diğerleri için okuma, yazma, çalıştırma bitlerine sahiptir. **CHMOD** sistem çağrısı dosyanın modunu değiştirmeyi gerçekleştirir. Örneğin, bir dosya sahibinden başka herkes için sadece okunur yapılabilir.

C programlama dilinde Temel dizin işlem fonksiyonları aşağıda yer almaktadır:

Fonksiyon	İşlevi
mkdir( )	Bir klasör oluşturur.
rmdir ( )	Bir klasörü siler.
opendir ( )	Bir klasörü açar.
closedir ( )	Bir klasörü kapatır.

Örnek 8: Temel Dizin işlemleri yapan C uygulaması

```
#include <iostream>
using namespace std;
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
#include <dirent.h>

struct stat st = {0};

int main() {
   char dizin[30];
   int c, secim;
   DIR *e;    /*Bu FILE *e gibi düşünelim. Klasor okuyunca buna aktaracaz.*/

   struct dirent *sd;    /* Bu yapıya okuduğumuz klasorun bilgileri aktarılır.
   Bu programda okuduğumuz klasor un alt klasorlerinin isimlerini alacağız. */

   do{
      system("clear");
      printf("1.Dizin olustur\n2.Dizin sil\n3.Dizin oku\n4-ÇIKIŞ\nSeciminizi girin : ");
      scanf("%d",&secim);

      switch(secim)
      {
         case 1:
            printf("\nDizin ismi girin\n");
            scanf("%s",&dizin);
            c=1;
            if (stat(dizin, &st) == -1) c=mkdir(dizin,777);
            /* Görüldüğü gibi mkdir 2 parametre alır ilki dosyanin ismi ikincisi ise verilen izin.
            777 deyince yazılabilir ve okunabilir oluyor. 655 yazarsak klasor yazılamaz olur
            (CHMOD deniyor bu parametreye). İşin ilginç yanı eğer klasör açılırsa 0 oluşmazsa 1 geriye döner.
            İsterseniz sadece mkdir(dizin) yazabilirsiniz varsayılan olarak ikinci parametre 777 olacaktır.*/

            if(c==1)
               printf("\nDizin olusturulamadi");
            else
               printf("\nDizin olusturuldu");
            getchar();
            break;
```

```
        case 2:
           printf("\nDizin ismi girin\n");
           scanf("%s",&dizin);
           c=rmdir(dizin);  /*Bu fonksiyonumuz dizin siler fakat klasorün içi dolu ise silmez.
               Geriye dönen sayı 1 ise silinemedi 0 ise silindi */

           if(c==1)
              printf("\nDizin silinemedi");
           else
              printf("\nDizin silindi");
           getchar();
           break;
        case 3:
           printf("\nAcilacak dizini girin\n");
           scanf("%s",&dizin);
           e=opendir(dizin);            /*Ve klasör açma. Bu aynı fopen fonksiyonu . */
           if(e==NULL)
              printf("\nBu isimle dizin yok");
           else
           {
              printf("\nDizin var\n");
              while((sd=readdir(e))!=NULL)
                 printf("%s\n",sd->d_name);
           }
           /* Evet burda e değişkeni okunuyor. sd struct yapısına gönderiliyor.
           sd pointer oldugu icin "." degil "->" ile bileşenlerine ulaştik.
           Burda dizini girdiğimiz dizin varya yani klasor işte bunun alt klasörlerinin
           ismi d_name (dir name) ile eriştik. Yani bu while döngüsü girdiğimiz klasorun alt klasörlerinin
           ismini ekrana yazacak.*/

           closedir(e);      /* Buda kapatıyor. */
           getchar();
           break;
      }
      getchar();
   }while(secim!=4);
   printf("İşlem Tamam!\n");
}
```

C programlama dilinde Temel dosya işlem fonksiyonları aşağıda yer almaktadır:

Fonksiyon	İşlevi
fopen( )	Bir dosya açar.
fclose( )	Bir dosyayı kapatır.
fputc( ) ve putc()	Dosyaya bir karakter yazar.
fgetc( ) ve getc()	Dosyadan bir karakter okur.
fseek( )	Bir dosyadaki belirli bir byte'ı bulur(Arama).
fprintf( )	Yapılandırılmış verileri dosyaya yazar.
fwrite()	
fscanf( )	Yapılandırılmış verileri dosyadan okur.
fread()	
feof( )	Dosya sonu geldiğinde doğru bir değer verir.
ferror( )	Bir hata durumunda doğru bir değer verir.
rewind( )	Dosya aktif konumunu başa alır.
remove( )	Dosyayı siler.

fopen() fonksiyonuna argüman olarak geçirilen ve mod ile gösterilen karakter dizisi ise dosyanın ne şekilde açılacağını belirler. Burada, mod ifadesi yerine kullanılabilecek değerler aşağıdaki tabloda yer almaktadır:

MOD	Anlamı
r	Okuma için bir metin dosyası açar.
w	Yazma için bir metin dosyası oluşturur.
a	Bir metin dosyasına ekleme yapar.
rb	Okuma için bir dosyayı ikili sistemde açar.
wb	Yazma için ikili sistemde bir dosya oluşturur.
ab	İkili sistemde bir dosyaya ekleme yapar.
r+	Okuma ve yazma için bir metin dosyası açar.
w+	Okuma ve yazma için bir metin dosyası oluşturur.
a+	Okuma ve yazma için bir metin dosyası oluşturur veya ekleme yapar.
r+b	Okuma ve yazma için bir ikili sistem dosyası açar.
w+b	Okuma ve yazma için bir ikili sistem dosyası oluşturur.
a+b	Okuma ve yazma için bir ikili sistem dosyasına ekleme yapar.

Örnek 9: Dosya işlemleri yapan C uygulaması

```
#include<stdio.h>
#include<dirent.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <stdlib.h>
#include <conio.h>
#include <string.h>
#include <windows.h>

struct stat st = {0};

struct kayit{
	int numara;
	char ad[30];
	char adres[30];
}k;

void gotoxy(short x, short y)
{
	COORD pos ={x,y};
	SetConsoleCursorPosition(GetStdHandle(STD_OUTPUT_HANDLE), pos);
}

int main() {
	char dizin[5]="data";
	char dosya[30];
	char path[100];
	char cevap;
	int c, secim, satir;
	DIR *dir;    /*Bu FILE *e gibi düşünelim. Klasor okuyunca buna aktaracaz.*/
	FILE *d;

	struct dirent *sd;    /* Bu yapıya okuduğumuz klasorun bilgileri aktarılır.
	Bu programda okuduğumuz klasor un alt klasorlerinin isimlerini alacağız. */

	dir=opendir(dizin);
	if(dir==NULL){
	    c=mkdir(dizin);
	    if(c==1){
	        printf("\nData Dizini olusturulamadi");
	        exit(0);
	    }
	    else
	        printf("\nData Dizini olusturuldu");
	}
	else closedir(dir);

   do{
	system("cls");
	printf("1.Data Dizinini Ac\n2.Dosya Olustur\n3.Kayit Ekle\n");
	printf("4-Dosya Icerigini goster\n5-Dosya Sil\n6-CIKIS\n");
	printf("Seciminizi girin : ");
	scanf("%d",&secim);
```

```
switch(secim)
{
    case 1:
        dir=opendir(dizin); /*Ve klasör açma. Bu aynı fopen fonksiyonu . */
        if(dir==NULL)
            printf("\nBu isimde dizin yok");
        else
        {
            printf("\nDizin var\n");
            while((sd=readdir(dir))!=NULL)
                printf("%s\n",sd->d_name);
        }
        closedir(dir);      getch();   break;

    case 2:
        printf("\nDosya ismi girin: ");
        scanf("%s",&dosya);         strcpy(path, dizin);
        strcat(path,"/");           strcat(path,dosya);

        if ((d=fopen(path,"r"))==NULL)
        {
            if((d=fopen(path,"w"))==NULL)
                printf("\nDosya olusturulamadı.");
            else
                printf("\nDosya olusturuldu.");
        }
        else
        {
            printf("\nDosya zaten mevcut.");   fclose(d);
        }
        getch();    break;

    case 3:
        printf("\nDosya ismi girin: ");
        scanf("%s",&dosya);         strcpy(path, dizin);
        strcat(path,"/");           strcat(path,dosya);

        if ((d=fopen(path,"r"))==NULL)
        {
            printf("\nDosya mevcut degil!");          getch();
            break;
        }
        fclose(d);
        if ((d=fopen(path,"a"))==NULL)
        {
            printf("\nDosya acilamadı.");             getch();
            break;
        }
        else printf("\nDosya acildi.\n");

        do{
            printf("\nNumara: ");  scanf("%d",&k.numara);
            printf("Ad: ");          scanf("%s",k.ad);
            printf("Adres: ");       scanf("%s",k.adres);
            fwrite(&k,sizeof(k),1,d);
            printf("Baska kayit girecek misiniz?(e/h)");    cevap=getch();
        }while(cevap=='e' || cevap=='E');
```

```
            fclose(d);
            printf("\nKayitlar eklendi!...");        getch();
            break;

        case 4:
            printf("\nDosya ismi girin: ");
            scanf("%s",&dosya);  strcpy(path, dizin);
            strcat(path,"/"); strcat(path,dosya);

            if ((d=fopen(path,"r"))==NULL)
            {
                printf("\nDosya mevcut degil!"); getch();
                break;
            }

            system("cls");
            gotoxy(1,1); printf("NUMARA");
            gotoxy(15,1); printf("AD");
            gotoxy(40,1); printf("ADRES");
            gotoxy(1,2);
            printf("--------------------------------------------------");
            satir=3;
            fread(&k,sizeof(k),1,d);
            while(!feof(d)){
                gotoxy(1,satir); printf("%d",k.numara);
                gotoxy(15,satir); printf("%s",k.ad);
                gotoxy(40,satir); printf("%s",k.adres);
                satir++;
                fread(&k,sizeof(k),1,d);
            }
            gotoxy(1,satir);
            printf("--------------------------------------------------");
            fclose(d);
            getch();    break;

        case 5:
            printf("\nDosya ismi girin: ");
            scanf("%s",&dosya);  strcpy(path, dizin);
            strcat(path,"/"); strcat(path,dosya);

            if ((d=fopen(path,"r"))==NULL)
            {
                printf("\nDosya mevcut degil!"); getch();
                break;
            }
            fclose(d);  remove(path);
            printf("Dosya Silindi!"); getch();
            break;
    }
  }while(secim!=6);
  printf("Islem Tamam!\n");
}
```

KAYNAKLAR

- http://www.belgeler.org/glibc/glibc-Signal-Handling.html
- https://web.itu.edu.tr/bkurt/Courses/os/lecture_1_to_5_2spp.pdf
- http://www.omegaegitim.com/books/os/
- https://www.harunalp.com/system-calls-sistem-cagrilari
- https://www.bilgigunlugum.net/prog/cprog/c_dosya
- https://web.karabuk.edu.tr/yasinortakci/dersnotlari.html
- https://silo.tips/download/letim-sistemlerine-giri-5