```
/*----------------------------------------------------------------------------------------------------
--------------------------

                                        C ve Sistem Programcıları Derneği

                                UNIX/Linux Sistem Programlama Kursunda Yapılan Örnekler ve Özet Notlar

                                        Eğitmen: Kaan ASLAN

                Bu notlar Kaan ASLAN tarafından oluşturulmuştur. Kaynak belirtmek koşulu ile her
türlü alıntı yapılabilir.

                                (Notları okurken editörünüzün "Line Wrapping" özelliğini pasif hale
getiriniz.)

----------------------------------------------------------------------------------------------------
--------------------------*/

/*----------------------------------------------------------------------------------------------------
--------------------------
                                                                                                1.
Ders 22/10/2022 - Cumartesi
----------------------------------------------------------------------------------------------------
--------------------------*/

/*----------------------------------------------------------------------------------------------------
--------------------------
    Merhaba UNIX/Linux Programı
----------------------------------------------------------------------------------------------------
--------------------------*/

#include <stdio.h>

int main(void)
{
        printf("Hello UNIX/Linux System Programming...\n");

        return 0;
}

/*----------------------------------------------------------------------------------------------------
--------------------------
                                                                                                4.
Ders 05/11/2022 - Cumartesi
----------------------------------------------------------------------------------------------------
--------------------------*/

/*----------------------------------------------------------------------------------------------------
--------------------------
    UNIX/Linux dünyasında komut satırı argümanlarının oluşturulması için geniş bir kesim
tarafından kullanılan geleneksel bir biçim vardır.
    Bu biçime "GNU biçimi" de denilmektedir. Biz de kursumuzda UNIX/Linux dünyasında yazacağımız
programlarda bu geleneği kullanacağız.
    GNU stilinde komut satırı argümanları üçe ayrılmaktadır:

    1) Argümansız seçenekler
    2) Argümanlı seçenekler
    3) Seçeneksiz argümanlar

    Argümansız seçenekler "-" karakterine yapışık tek bir harften oluşmaktadır. Harflerde büyük
harf - küçük harf duyarlılığı (case sensitivity)
    dikkate alınmaktadır. Örneğin:

    ls -l -i /usr/include

    Burada -l ve -i argümansız seçeneklerdir. /usr/include argümanının bu seçeneklerle hiçbir
ilgisi yoktur. Argümansız seçenekler tek bir
    karakterden oluşturulduğu için birleştirilebilmektedir. Örneğin:
```





```
    ls -li

    Buradaki -li aslında -l -i ile tamamen aynı anlamdadır. Genel olarak GNU stilinde seçenekler
arasındaki sıranın bir önemi yoktur.
    Yani örneğin:

    ls -l -i

    ile

    ls -i -l

    arasında bir farklılık yoktur.

    Argümanlı seçeneklerde bir seçeneğin yanında o seçenekle ilişkili bir argüman da bulunur.
Örneğin:

    gcc -o sample sample.c

    Burada -o seçeneği seçeneği tek başına kullanılmaz. Hedef dosyanın ismi seçeneğin argümanını
oluşturmaktadır. O halde buradaki
    -o seçeneği tipik olarak argümanlı seçeneğe bir örnektir. Argüman seçeneklerin birleştirilmesi
tavsiye edilmez. Ancak birleştirme yapılabilmektedir.
    Örneğin:

    gcc -co sample.o sample.c

    Bu yazım biçimini pek çok program kabul etse de biz tavsiye etmiyoruz. Buradaki argümanların
aşağıdaki gibi belirtilesi daha uygundur:

    gcc -c -o sample.o sample.c

    Programlar, argümanlı seçeneklerde seçeneğin argümanı hiç boşluk karakterleriyle ayrılmasa
bile bunu kabul edebilmektedir. Örneğin:

    gcc -osample sample.c

    Burada -o argümanlı seçenek olduğu için onu başka bir seçenek izleyemeyeceğinden dolayı
"sample" -o seçeneğinin argümanı olarak
    ele alınmaktadır.

    Seçeneklerle ilgisi olmayan argümanlara "seçeneksiz argüman" denilmektedir. Örneğin:

    gcc -o sample sample.c

    Burada "sample.c" argümanı herhangi bir seçenekle ilgili değildir. Örneğin:

    cp x.txt y.txt

    Buradaki "x.txt" ve "y.txt" argümanları da seçeneklerle ilgili değildir. Seçeneksiz
argümanların sonda bulunması gerekmez. Örneğin:

    gcc sample.c -o sample

----------------------------------------------------------------------------------------------------
--------------------------*/

/*----------------------------------------------------------------------------------------------------
--------------------------
    Eskiden yalnızca tek karakterden oluşan kısa seçenekler kullanılıyordu. Ancak daha sonraları
bu kısa seçeneklerin yetersiz kaldığı ve
    okunabilirliği bozduğu gerekçesiyle uzun seçenekler de kullanılmaya başlanmıştır. POSIX
standartları uzun seçenekleri desteklememektedir.
    Ancak UNIX/Linux dünyasında yaygın biçimde kullanılmaktadır. Uzun seçenekler "--" öneki ile
başlatılmaktadır. Örneğin:

    prog --count -a -b --length 100

    Uzun seçenekler de argümanlı ve argümansız olabilmektedir. Yukarıdaki örnekte "--count"
```





```
argümansız uzun seçenek, "-a" ve "-b" argümansız seçenekler
    ve "--length 100" ise argümanlı uzun seçenektir.

    Uzun seçeneklerde "isteğe bağlı argüman (optional argument)" denilen özel bir argüman da
kullanılmaktadır. İsmi üzerinde "isteğe bağlı argüman"
    uzun seçeneklerin yanında verilip verilmemesi isteğe bağlı olan argümanlardır. Uzun
seçeneklerin isteğe bağlı argümanları "=" sentaksı ile
    yapışık bir biçimde belirtilmektedir. Örneğin:

    prog --size=512

    Burada --size uzun seçeneğinin argümanı isteğe bağlıdır. Yani bu uzun seçenek argümansız da
aşağıdaki gibi kullanılabilirdi:

    prog --size

    Günümüzde genel olarak programlar kısa seçenekleri de uzun seçenekleri de bir arada
kullanmaktadır. Programcılar bazı kısa seçeneklerin
    alternatif uzun seçeneklerini oluşturabilmektedir. Yukarıda da belirttiğimiz gibi POSIX
standartları uzun seçenekleri desteklememektedir.
----------------------------------------------------------------------------------------------------
--------------------------*/

/*----------------------------------------------------------------------------------------------------
--------------------------
    UNIX/Linux dünyasında kullanılan komut satırı argümanlarını parse etmek için getopt ve
getopt_long isimli iki fonksiyon bulundurulmuştur.
    getopt fonksiyonu bir POSIX fonksiyonudur. Ancak bu fonksiyon uzun seçenekleri parse
etmemektedir. getopt_long ise uzun seçenekleri de parse eden
    getopt fonksiyonunun daha gelişmiş bir biçimidir. Ancak getopt_long bir POSIX fonksiyonu
değildir. Ancak libc kütüphanesinde bulunmaktadır.
    Bu fonksiyonlar Windows sistemlerinde hazır bir biçimde herhangi bir kütüphanede
bulunmamaktadır. Zaten yukarıda da belirttiğimiz gibi Windows
    sistemlerindeki komut satırı argüman stili UNIX/Linux sistemlerindekinden farklıdır.
----------------------------------------------------------------------------------------------------
--------------------------*/

/*----------------------------------------------------------------------------------------------------
--------------------------
    getopt fonksiyonunun prototipi şöyledir:

    #include <unistd.h>

    int getopt(int argc, char * const argv[], const char *optstring);

    getopt fonksiyonunun ilk iki parametresi main fonksiyonunun argc ve argv parametreleri
gibidir. Yani programcı main fonksiyonunun bu parametrelerini
    getopt fonksiyonuna geçirir. Fonksiyonun üçüncü parametresinde kısa seçenekler
belirtilmektedir. Bu parametre bir yazı biçiminde girilir.
    Bu yazıdaki her bir karakter bir kısa seçeneği belirtir. Bir karakterin yanında ':' karakteri
varsa bu ':' karakterinin solundaki seçeneğin
    argümanlı bir seçenek olduğunu belirtmektedir. Örneğin "ab:c" burada -a, -b ve -c seçenekleri
belirtilmiştir. Ancak -b seçeneğinin bir argümanı da
    vardır.

    getopt fonksiyonu bir kez çağrılmaz. Bir döngü içerisinde çağrılmalıdır. Çünkü fonksiyon her
çağrıldığında bir kısa seçeneği bulmaktadır.
    Fonksiyon bütün kısa seçenekleri bulduktan sonra artık bulacak bir seçenek kalmadığında -1
değerine geri dönmektedir. O halde fonksiyonun
    çağrılma kalıbı şöyle olmalıdır:

    int result;
    ...

    while ((result = getopt(argc, argv, "ab:c")) != -1) {
        ...
    }

    getopt, her kısa seçeneği bulduğunda o kısa seçeneğe ilişkin karakterle (yani o karakterin
```





```
sayısal karşılığı ile) geri dönmektedir.
    O halde bizim getopt fonksiyonunun geri dönüş değerini switch içerisinde ele almamız gerekir:

    while ((result = getopt(argc, argv, "ab:c")) != -1) {
        switch (result) {
            case 'a':
                ...
                break;
            case 'b':
                ...
                break;
            case 'c':
                ...
                break;
        }
    }

    getopt fonksiyonu, olmayan (yani üçüncü parametresinde belirtilmeyen) bir kısa seçenekle
karşılaştığında ya da argümanı olması gerektiği
    halde girilmemiş bir kısa seçenekle karşılaştığında '?' özel değerine geri dönmektedir.
Programcının switch deyimine bu case bölümünü
    ekleyerek bu durumu da değerlendirmesi uygun olur. Örneğin:

    while ((result = getopt(argc, argv, "ab:c")) != -1) {
        switch (result) {
            case 'a':
                ...
                break;
            case 'b':
                ...
                break;
            case 'c':
                ...
                break;
            case '?':
                ...
                break;
        }
    }

    getopt fonksiyonunun kullandığı dört global değişken vardır. Bu global değişkenler
kütüphanenin içerisinde tanımlanmıştır. Bunları biz
    extern bildirimi ile kullanabiliriz. Ancak bunların extern bildirimleri zaten <unistd.h>
dosyası içerisinde yapılmış durumdadır:

    extern int opterr;
    extern int optopt;
    extern int optind;
    extern char *optarg;

    Default durumda, getopt fonksiyonu geçersiz bir seçenekle (yani üçüncü parametresinde
belirtilmeyen bir seçenekle) karşılaştığında
    stderr dosyasına (ekranda çıkacaktır) kendisi hata mesajını yazdırmaktadır. Programcılar
genellikle bunu istemezler. getopt fonksiyonunun
    geçersiz seçenekler için hata mesajını yazdırması opterr değişkenine 0 değeri atanarak
sağlanabilir. Yani opterr değişkeni sıfır dışı
    bir değerdeyse (default durum) fonksiyon mesajı stderr dosyasına kendisi de yazar, sıfır
değerindeyse fonksiyon hata mesajını stderr dosyasına
    yazmaz.

    getopt fonksiyonu geçersiz bir seçenekle ya da argümanı girilmemiş argümanlı bir seçenekle
karşılaştığında '?' geri dönmekle birlikte aynı zamanda
    optopt global değişkenine geçersiz seçeneğin karakter karşılığını yerleştirmektedir. Böylece
programcı daha yeterli bir mesaj verebilmektedir.
    Örneğin:

    opterr = 0;
    while ((result = getopt(argc, argv, "ab:c")) != -1) {
        switch (result) {
```

```
            case 'a':
                printf("-a given...\n");
                break;
            case 'b':
                printf("-b given...\n");
                break;
            case 'c':
                printf("-c given...\n");
                break;
            case '?':
            if (optopt == 'b')
                fprintf(stderr, "-b option given without argument!...\n");
            else
                fprintf(stderr, "invalid option: -%c\n", optopt);
            break;
        }
    }
```

Argümanlı bir kısa seçenek bulunduğunda getopt fonksiyonu, optarg global değişkenini o kısa seçeneğin argümanını gösterecek biçimde
set eder. Ancak optarg, yeni bir argümanlı kısa seçenek bulunduğunda bu kez onun argümanını gösterecek biçimde set edilmektedir. Yani
programcı argümanlı kısa seçeneği bulduğu anda optarg değişkenine başvurmalı gerekirse onu başka bir göstericide saklamalıdır.

Pekiyi seçeneksiz argümanları nasıl edebiliriz? Seçeneksiz argümanlar argv dizisinin herhangi bir yerine bulunuyor olabilir. İşte getopt
fonksiyonu her zaman seçeneksiz argümanları girildiği sırada argv dizisinin sonuna taşır ve onların başladığı indeksi de optind global
değişkeninin göstermesini sağlar. O halde programcı getopt ile işini bitirdikten sonra (yani while döngüsünden çıktıktan sonra) optind
indeksinden argc indeksine kadar ilerleyerek tüm seçeneksiz argümanları elde edebilmektedir. Örneğin:

```
./sample -a ali -b veli selami -c
```

Burada "ali" ve "selami" seçeneksiz argümanlardır. getopt bu argv dizisini şu halde getirmektedir:

```
./sample -a -b veli -c ali selami
```

Şimdi burada optind indeksi artık "ali" argümanının başladığı indeksi belirtecektir. Onun ötesindeki tüm argümanlar seçeneksiz argümanlardır.
Bu argümanları while döngüsünün dışında şöyle yazdırabiliriz:

```
for (int i = optind; i < argc; ++i)
    puts(argv[i]);
```

Programcının girilmiş olan seçenekleri saklayıp programın ilerleyen aşamalarında bunları kullanması gerekebilmektedir. Bunun için şöyle
bir kalıp önerilebilir:

- Her seçenek için bir flag değişkeni tutulur. Bu flag değişkenlerine başlangıçta 0 atanır.
- Her argümanlı seçenek için bir gösterici tutulur.
- Her seçenekle karşılaşıldığında flag değişkenine 1 atanarak o seçeneğin kullanıldığı kaydedilir.
- Argümanlı seçeneklerle karşılaşıldığında onların argümanları göstericilerde saklanır.

```
-------------------------------------------------------------------------------------------------------------------------*/

/*-------------------------------------------------------------------------------------------------------------------------
    getopt fonksiyonun kullanımına ilişkin tipik bir kalıp aşağıda verilmiştir. Aşağıdaki örnekte -a, -b, -d argümansız seçenekler,
    -c ve -e ise argümanlı seçeneklerdir. Bu kalıbı kendi programlarınızda da kullanabilirsiniz. Bu örnekte ayrıştırma işleminde
    bir hata oluştuğunda programın devam etmemesini isteriz. Ancak tüm hataların rapor edilmesi de gerekmektedir. Bunun için bir
    flag değişkeninden faydalanılabilir. O flag değişkeni hata durumunda set edilir. Çıkışta
```





```
kontrol edilip duruma göre
    program sonlandırılır.
-------------------------------------------------------------------------------------------------------------------------
-------------------------*/

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main(int argc, char *argv[])
{
        int result;
        int a_flag, b_flag, c_flag, d_flag, e_flag, err_flag;
        char *c_arg, *e_arg;

        a_flag = b_flag = c_flag = d_flag = e_flag = err_flag = 0;

        opterr = 0;
        while ((result = getopt(argc, argv, "abc:de:")) != -1) {
                switch (result) {
                        case 'a':
                                a_flag = 1;
                                break;
                        case 'b':
                                b_flag = 1;
                                break;
                        case 'c':
                                c_flag = 1;
                                c_arg = optarg;
                                break;
                        case 'd':
                                d_flag = 1;
                                break;
                        case 'e':
                                e_flag = 1;
                                e_arg = optarg;
                                break;
                        case '?':
                                if (optopt == 'c' || optopt == 'e')
                                        fprintf(stderr, "-%c option must have an argument!\n",
optopt);
                                else
                                        fprintf(stderr, "-%c invalid option!\n", optopt);
                                err_flag = 1;
                        }
        }

        if (err_flag)
                exit(EXIT_FAILURE);

        if (a_flag)
                printf("-a option given\n");
        if (b_flag)
                printf("-b option given\n");
        if (c_flag)
                printf("-c option given with argument \"%s\"\n", c_arg);
        if (d_flag)
                printf("-d option given\n");
        if (e_flag)
                printf("-e option given with argument \"%s\"\n", e_arg);

        if (optind != argc)
                printf("Arguments without option:\n");
        for (int i = optind; i < argc; ++i)
                puts(argv[i]);

        return 0;
}
```





```
/*-------------------------------------------------------------------------------------------------------------------------
-------------------------
        getopt fonksiyonun kullanımına bir örnek. Bu örnekte disp isimli program şu komut satırı argümanlarını almaktadır:

        -x (display hex)
        -o (display octal)
        -t (display text)
        -n (number of character per line)

        Burada -x, -o ve -t seçeneklerinden yalnızca bir tanesi kullanılabilmektedir. Eğer hiçbir seçenek kullanılmazsa default durum "-t"
        biçimindedir. -n seçeneği yalnızca hex ve octal görüntülemede kullanılabilmektedir. Bu seçenek de belirtilmezse sanki "-n 16" gibi
        bir belirleme yapıldığı varsayılmaktadır.
-------------------------------------------------------------------------------------------------------------------------
-------------------------*/

#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#include <stdbool.h>
#include <unistd.h>

#define DEFAULT_LINE_CHAR               16

bool disp_text(FILE *f);
bool disp_hex(FILE *f, int n_arg);
bool disp_octal(FILE *f, int n_arg);
int check_number(const char *str);

int main(int argc, char *argv[])
{
        int result;
        int t_flag, o_flag, x_flag, n_flag, err_flag;
        int n_arg;
        FILE *f;

        t_flag = o_flag = x_flag = n_flag = err_flag = 0;
        n_arg = DEFAULT_LINE_CHAR;
        opterr = 0;

        while ((result = getopt(argc, argv, "toxn:")) != -1) {
                switch (result) {
                        case 't':
                                t_flag = 1;
                                break;
                        case 'o':
                                o_flag = 1;
                                break;
                        case 'x':
                                x_flag = 1;
                                break;
                        case 'n':
                                n_flag = 1;
                                if ((n_arg = check_number(optarg)) < 0) {
                                        fprintf(stderr, "-n argument is invalid!...\n");
                                        err_flag = 1;
                                }
                                break;
                        case '?':
                                if (optopt == 'n')
                                        fprintf(stderr, "-%c option given without argument!...\n",
optopt);
                                else
                                        fprintf(stderr, "invalid option: -%c\n", optopt);
                                err_flag = 1;

                        break;
```





```
                }
        }

        if (err_flag)
                exit(EXIT_FAILURE);

        if (t_flag + o_flag + x_flag > 1) {
                fprintf(stderr, "only one of -[tox] option may be specified!...\n");
                exit(EXIT_FAILURE);
        }

        if (t_flag + o_flag + x_flag == 0)
                t_flag = 1;

        if (t_flag && n_flag) {
                fprintf(stderr, "-n option cannot be used with -t option!...\n");
                exit(EXIT_FAILURE);
        }

        if (argc - optind == 0) {
                fprintf(stderr, "file must be specified!...\n");
                exit(EXIT_FAILURE);
        }
        if (argc - optind > 1) {
                fprintf(stderr, "too many files specified!...\n");
                exit(EXIT_FAILURE);
        }

        if ((f = fopen(argv[optind], t_flag ? "r" : "rb")) == NULL) {
                fprintf(stderr, "cannot open file: %s\n", argv[optind]);
                exit(EXIT_FAILURE);
        }
        if (t_flag)
                result = disp_text(f);
        else if (x_flag)
                result = disp_hex(f, n_arg);
        else if (o_flag)
                result = disp_octal(f, n_arg);

        if (!result) {
                fprintf(stderr, "cannot read file: %s\n", argv[optind]);
                exit(EXIT_FAILURE);
        }

        fclose(f);

        return 0;
}

bool disp_text(FILE *f)
{
        int ch;

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

        return feof(f);
}

bool disp_hex(FILE *f, int n_arg)
{
        size_t i;
        int ch;

        for (i = 0;(ch = fgetc(f)) != EOF; ++i) {
                if (i % n_arg == 0) {
                        if (i != 0)
                                putchar('\n');
                        printf("%08zX ", i);
```

```
                }
                printf("%02X ", ch);

        }
        putchar('\n');

        return feof(f);
}

bool disp_octal(FILE *f, int n_arg)
{
        size_t i;
        int ch;

        for (i = 0;(ch = fgetc(f)) != EOF; ++i) {
                if (i % n_arg == 0)
                        printf("%08zo ", i);

                printf("%03o ", ch);
                if (i % n_arg == n_arg - 1)
                        putchar('\n');
        }
        putchar('\n');

        return feof(f);
}

int check_number(const char *str)
{
        const char *temp;
        int result;

        while (isspace(*str))
                ++str;

        temp = str;

        while (isdigit(*str))
                ++str;

        if (*str != '\0')
                return -1;

        result = atoi(temp);
        if (!result)
                return -1;

        return result;
}

/*-------------------------------------------------------------------------------------------------
-------------------------
    Aşağıdaki örnekte mycalc isimli bir program yazılmıştır. Program iki komut satırı argümanı ile
aldığı değerler üzerinde dört
    işlem yapmaktadır. Aşağıdaki seçeneklere sahiptir:

    -a: Toplama işlemi
    -m: Çarpma işlemi
    -d: Bölme işlemi
    -s: Çıkartma işlemi
    -D msg: Çıktının başında "msg: " kısmını ekler

-------------------------------------------------------------------------------------------------
------------------------*/

/* mycalc.c */

#include <stdio.h>
#include <stdlib.h>
```





```
#include <unistd.h>

int main(int argc, char *argv[])
{
        int result;
        int a_flag, m_flag, M_flag, d_flag, s_flag, err_flag;
        char *M_arg;
        double arg1, arg2, calc_result;

        a_flag = m_flag = M_flag = d_flag = s_flag = err_flag = 0;

        opterr = 0;

        while ((result = getopt(argc, argv, "amM:ds")) != -1) {
                switch (result) {
                        case 'a':
                                a_flag = 1;
                                break;
                        case 'm':
                                m_flag = 1;
                                break;
                        case 'M':
                                M_flag = 1;
                                M_arg = optarg;
                                break;
                        case 'd':
                                d_flag = 1;
                                break;
                        case 's':
                                s_flag = 1;
                                break;
                        case '?':
                                if (optopt == 'M')
                                        fprintf(stderr, "-M option must have an argument!\n");
                                else
                                        fprintf(stderr, "invalid option: -%c\n", optopt);
                                err_flag = 1;
                        }
                }

        if (err_flag)
                exit(EXIT_FAILURE);

        if (a_flag + m_flag + d_flag + s_flag > 1) {
                fprintf(stderr, "only one option must be specified!\n");
                exit(EXIT_FAILURE);
        }
        if (a_flag + m_flag + d_flag + s_flag == 0) {
                fprintf(stderr, "at least one of -amds options must be specified\n");
                exit(EXIT_FAILURE);
        }

        if (argc - optind != 2) {
                fprintf(stderr, "two number must be specified!\n");
                exit(EXIT_FAILURE);
        }

        arg1 = atof(argv[optind]);
        arg2 = atof(argv[optind + 1]);

        if (a_flag)
                calc_result = arg1 + arg2;
        else if (m_flag)
                calc_result = arg1 * arg2;
        else if (d_flag)
                        calc_result = arg1 / arg2;
        else
                calc_result = arg1 - arg2;
```





```
        if (M_flag)
                printf("%s: %f\n", M_arg, calc_result);
        else
                printf("%f\n", calc_result);

        return 0;
}

/*-------------------------------------------------------------------------------------------------
-------------------------
    Daha önceden de belirttiğimiz gibi komut satırında uzun seçenek kullanımı POSIX
standartlarında yoktur. Ancak Linux gibi pek çok
    sistemdeki çeşitli yardımcı programlar uzun seçenekleri desteklemektedir. Programlarda bazı
kısa seçeneklerin eşdeğer uzun seçenekleri
    bulunmaktadır. Bazı uzun seçeneklerin ise kısa seçenek eşdeğeri bulunmamaktadır. Bazı kısa
seçeneklerin de uzun seçenek eşdeğerleri yoktur.

    Uzun seçenekleri parse etmek için getopt_long isimli fonksiyon kullanılmaktadır. Uzun
seçenekler POSIX standartlarında olmadığına göre
    getopt_long fonksiyonu da bir POSIX fonksiyonu değildir. Ancak GNU'nun glibc kütüphanesinde
bir eklenti biçiminde bulunmaktadır.
    getopt_long fonksiyonu işlevsel olarak getopt fonksiyonunu kapsamaktadır. Ancak fonksiyonun
kullanımı biraz daha zordur. Fonksiyonun
    prototipi şöyledir:

    #include <getopt.h>

    int getopt_long(int argc, char * const argv[], const char *optstring, const struct option
*longopts, int *longindex);

    Fonksiyonun birinci ve ikinci parametrelerine, main fonksiyonundan alınan argc ve argv
parametreleri geçirilir. Fonksiyonun üçüncü parametresi
    yine kısa seçeneklerin belirtildiği yazının adresini almaktadır. Yani fonksiyonun ilk üç
parametresi tamamen getopt fonksiyonu ile aynıdır.
    Fonksiyonun dördüncü parametresi uzun seçeneklerin belirtildiği struct option türünden bir
yapı dizisinin adresini almaktadır. Her uzun seçenek
    struct option türünden bir nesneyle ifade edilmektedir. struct option yapısı şöyle
bildirilmiştir:

    struct option {
        const char *name;
        int         has_arg;
        int        *flag;
        int         val;
    };

    Fonksiyon bu yapı dizisinin bittiğini nasıl anlayacaktır? İşte yapı dizisinin son elemanına
ilişkin yapı nesnesinin tüm elemanları 0'larla
    doldurulmalıdır. (0 sabitinin göstericiler söz konusu olduğunda NULL adres anlamına geldiğini
de anımsayınız.)

    struct option yapısının name elemanı uzun seçeneğin ismini belirtmektedir. Yapının has_arg
elemanı üç değerden birini alabilir:

    no_argument (0)
    required_argument (1)
    optional_argument (2)

    Bu eleman uzun seçeneğin argüman alıp almadığını belirtmektedir. Yapının flag ve val
elemanları birbirleriyle ilişkilidir. Yapının val elemanı
    uzun seçenek bulunduğunda bunun hangi sayısal değerle ifade edileceğini belirtir. İşte bu flag
elemanına int bir nesnenin adresi geçilirse bu durumda
    uzun seçenek bulunduğunda bu val değeri bu int nesneye yerleştirilir. getopt_long ise bu
durumda 0 değeri ile geri döner. Ancak bu flag
    göstericisine NULL adres de geçilebilir. Bu durumda getopt_long uzun seçenek bulunduğunda val
elemanındaki değeri geri dönüş değeri olarak verir.
    Örneğin:

    struct option options[] = {
```





```
        {"count", required_argument, NULL, 'c'},
        {0, 0, 0, 0}
    };

    Burada uzun seçenek "--count" biçimindedir. Bir argümanla kullanılmak zorundadır. Bu uzun
seçenek bulunduğunda flag parametresi NULL adres
    geçildiği için getopt_long fonksiyonu 'c' değeri ile geri dönecektir. Örneğin:

    int count_flag;
    ...

    struct option options[] = {
        {"count", required_argument, &count_flag, 1},
        {0, 0, 0, 0}
    };

    Burada artık uzun seçenek bulunduğunda getopt_long fonksiyonu 0 ile geri dönecek ancak 1
değeri count_flag nesnesine yerleştirilecektir.

    getopt_long fonksiyonunun son parametresi uzun seçenek bulunduğunda o uzun seçeneğin option
dizisindeki kaçıncı indeksli uzun seçenek olduğunu
    anlamak için kullanılmaktadır. Burada belirtilen adresteki nesneye uzun seçeneğin option
dizisi içerisindeki indeks numarası yerleştirilmektedir.
    Ancak bu bilgiye genellikle gereksinim duyulmamaktadır. Bu parametre NULL geçilebilir. Bu
durumda böyle bir yerleştirme yapılmaz.

    Bu durumda getopt_long fonksiyonunun geri dönüş değeri beş biçimden biri olabilir:

    1) Fonksiyon bir kısa seçenek bulmuştur. Kısa seçeneğin karakter koduyla geri döner.
    2) Fonksiyon bir uzun seçenek bulmuştur ve option yapısının flag elemanında NULL adres vardır.
Bu durumda fonksiyon option yapısının
    val elemanındaki değerle geri döner.
    3) Fonksiyon bir uzun seçenek bulmuştur ve option yapısının flag elemanında NULL adres yoktur.
Bu durumda fonksiyon val değerini bu adrese
    yerleştirir ve 0 değeri ile geri döner.
    4) Fonksiyon geçersiz (yani olmayan) bir kısa ya da uzun seçenekle karşılaşmıştır ya da
argümanlı bir kısa seçenek ya da uzun seçeneğin
    argümanı girilmemiştir. Bu durumda fonksiyon '?' karakterinin değeriyle geri döner.
    5) Parse edecek argüman kalmamıştır fonksiyon -1 ile geri döner.

    getopt fonksiyonundaki yardımcı global değişkenlerin aynısı burada da kullanılmaktadır:

    opterr: Hata mesajının fonksiyon tarafından stderr dosyasına basılıp basılmayacağını belirtir.
    optarg: Argümanlı bir kısa ya da uzun seçenekte argümanı belirtmektedir. Eğer "isteğe bağlı
argümanlı" bir uzun seçenek bulunmuşsa
    ve bu uzun seçenek için argüman girilmemişse optarg nesnesine NULL adres yerleştirilmektedir.
    optind: Bu değişken yine seçeneksiz argümanların başladığı indeksi belirtmektedir.
    optopt: Bu değişken geçersiz bir uzun ya da kısa seçenek girildiğinde hatanın nedenini
belirtmektedir.

    getopt_long geçersiz bir seçenekle karşılaştığında '?' geri dönmekle birlikte optopt
değişkenini şu biçimlerde set etmektedir:

    1) Eğer fonksiyon argümanlı bir kısa seçenek bulduğu halde argüman girilmemişse o argümanlı
kısa seçeneğin karakter karşılığını
    optopt değişkenine yerleştirir.
    2) Eğer fonksiyon argümanlı bir uzun seçenek bulduğu halde argüman girilmemişse o argümanlı
uzun seçeneğin option yapısındaki val değerini
    optopt değişkenine yerleştirmektedir.
    3) Eğer fonksiyon geçersiz bir kısa seçenekle karşılaşmışsa bu durumda optopt geçersiz kısa
seçeneğin karakter karşılığına geri döner.
    4) Eğer fonksiyon geçersiz bir uzun seçenekle karşılaşmışsa bu durumda optopt değişkenine 0
değeri yerleştirilmektedir.

    Maalesef getopt_long olmayan bir uzun seçenek girildiğinde bunu bize vermemektedir. Ancak
GNU'nun getopt_long gerçekleştirimine bakıldığında
    bu geçersiz uzun seçeneğin argv dizisinin "optind - 1" indeksinde olduğu görülmektedir. Yani
bu geçersiz uzun seçeneğe argv[optind - 1]
    ifadesi ile erişilebilmektedir. Ancak bu durum glibc dokümanlarında belirtilmemiştir. Bu
```

```
nedenle bu özelliğin kullanılması uygun değildir.
-------------------------------------------------------------------------------------------------------------
-------------------------*/

/*-------------------------------------------------------------------------------------------------------------
-------------------------
                                                                                                        5.
Ders 06/11/2022 - Pazar
-------------------------------------------------------------------------------------------------------------
-------------------------*/

/*-------------------------------------------------------------------------------------------------------------
-------------------------
        Aşağıdaki örnekteki komut satırı argümanları şunlardır:

        -a
        -b
        -c <arg> ya da --count <arg>
        --verbose
-------------------------------------------------------------------------------------------------------------
-------------------------*/

#include <stdio.h>
#include <stdlib.h>
#include <getopt.h>

int main(int argc, char *argv[])
{
        int a_flag, b_flag, c_flag, verbose_flag;
        int err_flag;
        char *c_arg;
        int result;

        struct option options[] = {
                {"count", required_argument, NULL, 'c'},
                {"verbose", no_argument, &verbose_flag, 1},
                {0, 0, 0, 0}
        };

        a_flag = b_flag = c_flag = verbose_flag = err_flag = 0;

        opterr = 0;
        while ((result = getopt_long(argc, argv, "abc:", options, NULL)) != -1) {
                switch (result) {
                case 'a':
                        a_flag = 1;
                        break;
                case 'b':
                        b_flag = 1;
                        break;
                case 'c':
                        c_flag = 1;
                        c_arg = optarg;
                        break;
                case '?':
                        if (optopt == 'c')
                                fprintf(stderr, "option -c or --count without argument!...\n");
                        else if (optopt != 0)
                                fprintf(stderr, "invalid option: -%c\n", optopt);
                        else
                                fprintf(stderr, "invalid long option!...\n");
                        /* fprintf(stderr, "invalid long option: %s\n", argv[optind - 1]); */
                        err_flag = 1;
                        break;
                }
        }

        if (err_flag)
                exit(EXIT_FAILURE);
```





```
        if (a_flag)
                printf("-a option given\n");
        if (b_flag)
                printf("-b option given\n");
        if (c_flag)
                printf("-c or --count option given with argument \"%s\"\n", c_arg);
        if (verbose_flag)
                printf("--verbose given\n");

        if (optind != argc) {
                printf("Arguments without options");
                for (int i = optind; i < argc; ++i)
                        printf("%s\n", argv[i]);
        }

        return 0;
}

/*-------------------------------------------------------------------------------------------------------------
-------------------------
    getopt_long fonksiyonun kullanımına diğer bir örnekte aşağıda verilmiştir. Aşağıda programın
komut satırı argümanları şunlardır:

    -a
    -b <arg>
    -c
    -h ya da --help
    --count <arg>
    --line[=<arg>]

-------------------------------------------------------------------------------------------------------------
-------------------------*/

#include <stdio.h>
#include <stdlib.h>
#include <getopt.h>

int main(int argc, char *argv[])
{
        int result;
        int a_flag, b_flag, c_flag, h_flag, count_flag, line_flag;
        char *b_arg, *count_arg, *line_arg;
        int err_flag;
        int i;

        struct option options[] = {
                {"help", no_argument, &h_flag, 1},
                {"count", required_argument, NULL, 2},
                {"line", optional_argument, NULL, 3},
                {0, 0, 0, 0 },
        };

        a_flag = b_flag = c_flag = h_flag = count_flag = line_flag = 0;
        err_flag = 0;

        opterr = 0;
        while ((result = getopt_long(argc, argv, "ab:ch", options, NULL)) != -1) {
                switch (result) {
                        case 'a':
                                a_flag = 1;
                                break;
                        case 'b':
                                b_flag = 1;
                                b_arg = optarg;
                                break;
                        case 'c':
                                c_flag = 1;
                                break;
```





```
                        case 'h':
                                h_flag = 1;
                                break;
                        case 2:                 /* --count */
                                count_flag = 1;
                                count_arg = optarg;
                                break;
                        case 3:                 /* --line */
                                line_flag = 1;
                                line_arg = optarg;
                                break;
                        case '?':
                                if (optopt == 'b')
                                        fprintf(stderr, "-b option must have an argument!...\n");
                                else if (optopt == 2)
                                        fprintf(stderr, "argument must be specified with --count
option\n");
                                else if (optopt != 0)
                                        fprintf(stderr, "invalid option: -%c\n", optopt);
                                else
                                        fprintf(stderr, "invalid long option!...\n");

                                err_flag = 1;

                                break;
                }
        }

        if (err_flag)
                exit(EXIT_FAILURE);

        if (a_flag)
                printf("-a option given...\n");

        if (b_flag)
                printf("-b option given with argument \"%s\"...\n", b_arg);

        if (c_flag)
                printf("-c option given...\n");

        if (h_flag)
                printf("-h or --help option given...\n");

        if (count_flag)
                printf("--count option specified with \"%s\"...\n", count_arg);

        if (line_flag) {
                if (line_arg != NULL)
                        printf("--line option given with optional argument \"%s\"\n", line_arg);
                else
                        printf("--line option given without optional argument...\n");
        }

        if (optind != argc) {
                printf("Arguments without options:\n");
                for (i = optind; i < argc; ++i)
                        printf("%s\n", argv[i]);
        }

        return 0;
}

/*-------------------------------------------------------------------------------------------------------------
-------------------------
    getopt_long fonksiyonun kullanılmasına başka bir örnek. Bu örnekteki seçenekler şöyledir:

    -a: argümansız kısa seçenek
    -b: argümanlı kısa seçenek
    --all: argümansız uzun seçenek
```





```
    --length: argümanlı uzun seçenek
    --number: isteğe bağlı argümanlı uzun seçenek

-------------------------------------------------------------------------------------------------------------
-------------------------*/

#include <stdio.h>
#include <stdlib.h>
#include <getopt.h>

int main(int argc, char *argv[])
{
        int result;
        struct option options[] = {
                {"all", no_argument, NULL, 1},
                {"length", required_argument, NULL, 2},
                {"number", optional_argument, NULL, 3},
                {0, 0, 0, 0},
        };
        int a_flag, b_flag, all_flag, length_flag, number_flag, err_flag;
        char *b_arg, *length_arg, *number_arg;

        a_flag = b_flag = all_flag = length_flag = number_flag = err_flag = 0;
        opterr = 0;
        while ((result = getopt_long(argc, argv, "ab:", options, NULL)) != -1) {
                switch (result) {
                        case 'a':
                                a_flag = 1;
                                break;
                        case 'b':
                                b_flag = 1;
                                b_arg = optarg;
                                break;
                        case 1:
                                all_flag = 1;
                                break;
                        case 2:
                                length_flag = 1;
                                length_arg = optarg;
                                break;
                        case 3:
                                number_flag = 1;
                                number_arg = optarg;
                                break;
                        case '?':
                                if (optopt == 'b')
                                        fprintf(stderr, "-b option without argument!\n");
                                else if (optopt == 2)
                                        fprintf(stderr, "--length option without argument!\n");
                                else if (optopt != 0)
                                        fprintf(stderr, "invalid option: -%c\n", optopt);
                                else
                                        fprintf(stderr, "invalid long option!\n");
                                err_flag = 1;
                        }
        }
        if (err_flag)
                exit(EXIT_FAILURE);

        if (a_flag)
                printf("-a option given\n");
        if (b_flag)
                printf("-b option given with argument \"%s\"\n", b_arg);
        if (all_flag)
                printf("--all option given\n");
        if (length_flag)
                printf("--length option given with argument \"%s\"\n", length_arg);
        if (number_flag)
                if (number_arg != NULL)
```

```
                        printf("--number option given with argument \"%s\"\n", number_arg);
                else
                        printf("--number option given without argument\n");

        if (optind != argc)
                printf("Arguments without options:\n");
        for (int i = optind; i < argc; ++i)
                puts(argv[i]);

        return 0;
}

/*

test girişi: ./sample --all --length 100 --number=300 -a ali veli selami
Çıktısı şöyledir:

-a option given
--all option given
--length option given with argument "100"
--number option given with argument "300"
Arguments without options:
ali
veli
selami

*/

/*-------------------------------------------------------------------------------------------------
-------------------------
    getopt_long fonksiyonunda struct option yapısındaki flag elemanına NULL adres yerine int bir
nesnenin adresi geçirilirse
    bu durumda getopt_long bu uzun seçenek girildiğinde doğrudan yapının val elemanındaki değeri
bu nesneye yerleştirir ve 0 ile geri
    döner. Böylece programcı isterse argümansız uzun seçenekleri switch içerisinde işlemeden
doğrudan onun bayrağına set işlemi
    yapabilir. Ayrıca programlarda kısa seçeneklerin uzun seçenek eşdeğerleri de
bulunabilmektedir. Bunu sağlamanın en kolay yolu
    uzun seçeneğe ilişkin struct option yapısındaki val elemanına kısa seçeneğe ilişkin karakter
kodunu girmektir.
-------------------------------------------------------------------------------------------------
-------------------------*/

#include <stdio.h>
#include <stdlib.h>
#include <getopt.h>

int main(int argc, char *argv[])
{
        int result;
        int a_flag, b_flag, all_flag, length_flag, number_flag, err_flag;
        char *b_arg, *length_arg, *number_arg;
        struct option options[] = {
                {"all", no_argument, &all_flag, 1},
                {"length", required_argument, NULL, 'l'},
                {"number", optional_argument, NULL, 3},
                {0, 0, 0, 0},
        };

        a_flag = b_flag = all_flag = length_flag = number_flag = err_flag = 0;
        opterr = 0;
        while ((result = getopt_long(argc, argv, "ab:l:", options, NULL)) != -1) {
                switch (result) {
                        case 'b':
                                b_flag = 1;
                                b_arg = optarg;
                                break;
                        case 1:
                                all_flag = 1;
```





```
                                break;
                        case 'l':
                                length_flag = 1;
                                length_arg = optarg;
                                break;
                        case 3:
                                number_flag = 1;
                                number_arg = optarg;
                                break;
                        case '?':
                                if (optopt == 'b')
                                        fprintf(stderr, "-b option without argument!\n");
                                else if (optopt == 2)
                                        fprintf(stderr, "--length option without argument!\n");
                                else if (optopt != 0)
                                        fprintf(stderr, "invalid option: -%c\n", optopt);
                                else
                                        fprintf(stderr, "invalid long option!\n");
                                err_flag = 1;
                }
        }
        if (err_flag)
                exit(EXIT_FAILURE);

        if (a_flag)
                printf("-a option given\n");
        if (b_flag)
                printf("-b option given with argument \"%s\"\n", b_arg);
        if (all_flag)
                printf("--all option given\n");
        if (length_flag)
                printf("--length option given with argument \"%s\"\n", length_arg);
        if (number_flag)
                if (number_arg != NULL)
                        printf("--number option given with argument \"%s\"\n", number_arg);
                else
                        printf("--number option given without argument\n");

        if (optind != argc)
                printf("Arguments without options:\n");
        for (int i = optind; i < argc; ++i)
                puts(argv[i]);

        return 0;
}

/*-------------------------------------------------------------------------------------------------
-------------------------
                                                                                                6.
Ders 12/11/2022 - Cumartesi
-------------------------------------------------------------------------------------------------
-------------------------*/

/*-------------------------------------------------------------------------------------------------
-------------------------
    Bir kullanıcı ile login olunduğunda login programı /etc/passwd dosyasında belirtilen programı
çalıştırır.
    Biz istersek bu programı değiştirip kendi istediğimiz bir programın çalıştırılmasını
sağlayabiliriz. Kendi programımız
    myshell isimli program olsun ve onu /bin dizinine kopyalamış olalım. /etc/passwd dosyasının
içeriğini şöyle değiştirebiliriz:

    ali:x:1002:1001::/home/ali:/bin/myshell

-------------------------------------------------------------------------------------------------
-------------------------*/

/* myshell.c */
```





```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

#define MAX_CMD_LINE            4096
#define MAX_CMD_PARAMS          128

typedef struct tagCMD {
        char *name;
        void (*proc)(void);
} CMD;

void parse_cmd_line(char *cmdline);
void dir_proc(void);
void clear_proc(void);
void pwd_proc(void);

char *g_params[MAX_CMD_PARAMS];
int g_nparams;

CMD g_cmds[] = {
        {"dir", dir_proc},
        {"clear", clear_proc},
        {"pwd", pwd_proc},
        {NULL, NULL}
};

int main(void)
{
        char cmdline[MAX_CMD_LINE];
        char *str;
        int i;

        for (;;) {
                printf("CSD>");
                if (fgets(cmdline, MAX_CMD_LINE, stdin) == NULL)
                        continue;
                if ((str = strchr(cmdline, '\n')) != NULL)
                        *str = '\0';
                parse_cmd_line(cmdline);
                if (g_nparams == 0)
                        continue;
                if (!strcmp(g_params[0], "exit"))
                        break;
                for (i = 0; g_cmds[i].name != NULL; ++i)
                        if (!strcmp(g_params[0], g_cmds[i].name)) {
                                g_cmds[i].proc();
                                break;
                        }
                if (g_cmds[i].name == NULL)
                        printf("bad command: %s\n", g_params[0]);
        }

        return 0;
}

void parse_cmd_line(char *cmdline)
{
        char *str;

        g_nparams = 0;
        for (str = strtok(cmdline, " \t"); str != NULL; str = strtok(NULL, " \t"))
                g_params[g_nparams++] = str;
}

void dir_proc(void)
{
        printf("dir command executing...\n");
```





```
}

void clear_proc(void)
{
        system("clear");
}

void pwd_proc(void)
{
        char cwd[4096];

        if (g_nparams > 1) {
                printf("pwd command must be used withoud argument!...\n");
                return;
        }

        getcwd(cwd, 4096);

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

/*-------------------------------------------------------------------------------------------------
-------------------------
                                                                                                7.
Ders 13/11/2022 - Pazar
-------------------------------------------------------------------------------------------------
-------------------------*/

/*-------------------------------------------------------------------------------------------------
-------------------------
    Bir hata değerinin yazısını elde etmek için strerror fonksiyonu kullanılabilir. Fonksiyon
bizden EXXX biçimindeki hata kodunu
    parametre olarak alır, bize statik düzeyde tahsis edilmiş hata yazısının adresini verir. Biz
de POSIX fonksiyonu başarısız
    olduğunda errno değerini bu biçimde yazıya dönüştürüp rapor edebiliriz.

    Aşağıda buna bir örnek verilmiştir.
-------------------------------------------------------------------------------------------------
-------------------------*/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>
#include <fcntl.h>

int main(void)
{
        int fd;

        if ((fd = open("xxx.txt", O_RDONLY)) == -1) {
                fprintf(stderr, "open failed: %s\n", strerror(errno));
                exit(EXIT_FAILURE);
        }
        printf("success\n");

        return 0;
}

/*-------------------------------------------------------------------------------------------------
-------------------------
    strerror fonksiyonu ile alınan error yazısı default durumda İngilizce'dir. POSIX
standartlarına göre bu yazının içeriği locale'in
    LC_MESSAGES kategorisine göre ayarlanmaktadır. Dolayısıyla eğer mesajları Türkçe bastırmak
istiyorsanız LC_MESSAGES kategorisine ilişkin
    locale'i setlocale fonksiyonu ile değiştirmelisiniz. Tabii genel olarak tüm kategorilerin
değiştirilmesi yoluna gidilmektedir.
    Türkçe UNICODE UTF-8 locale'i "tr_TR.UTF-8" ile temsil edilmektedir. Dolayısıyla bu işlemi
şöyle yapabilirsiniz:
```

```
        if (setlocale(LC_ALL, "tr_TR.UTF-8") == NULL) {
                fprintf(stderr, "cannot set locale!...\n");
                exit(EXIT_FAILURE);
        }

-------------------------------------------------------------------------------------------------
------------------------*/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>
#include <locale.h>

int main(void)
{
        if (setlocale(LC_ALL, "tr_TR.UTF-8") == NULL) {
                fprintf(stderr, "cannot set locale!...\n");
                exit(EXIT_FAILURE);
        }

        puts(strerror(EPERM));

        return 0;
}

/*-------------------------------------------------------------------------------------------------
-------------------------
    POSIX fonksiyonlarında oluşan hatayı rapor etmek için perror isimli daha pratik kullanımı olan
bir POSIX fonksiyonu (aynı zamanda
    standart C fonksiyonudur) bulundurulmuştur. Fonksiyonun prototipi şöyledir:

        #include <stdio.h>

        void perror(const char *str);

    Fonksiyon argüman olarak girilen yazıyı stderr dosyasına yazdırır. Sonra hemen yanına ':'
karakterini ve bir SPACE karakterini basar ve sonra da o andaki
    errno değerinin yazısını yazdırır. İmleci aşağı satırın başına geçirir. Fonksiyon aşağıdaki
gibi yazılabilir:

        void perror(const char *str)
        {
                fprintf(stderr, "%s: %s\n", str, strerror(errno));
        }

-------------------------------------------------------------------------------------------------
------------------------*/

#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>

int main(void)
{
        int fd;

        if ((fd = open("xxx.txt", O_RDONLY)) == -1) {
                perror("open");
                exit(EXIT_FAILURE);
        }
        printf("success\n");

        return 0;
}

/*-------------------------------------------------------------------------------------------------
-------------------------
```





```
    Biz kursumuzda, bir POSIX fonksiyonu başarısız olduğunda genellikle (ancak her zaman değil)
programımızı sonlandıracağız.
    Bu durumda daha az tuşa basmak için bir exit_sys isimli "sarma (wrapper)" fonksiyondan
faydalanacağız. Bu fonksiyon önce perror
    fonksiyonu ile hatayı stderr dosyasına yazdıracak sonra da exit fonksiyonu ile program
sonlandıracaktır:

        void exit_sys(const char *msg)
        {
                perror(msg);

                exit(EXIT_FAILURE);
        }

-------------------------------------------------------------------------------------------------
------------------------*/

#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <errno.h>
#include <fcntl.h>

void exit_sys(const char *msg);

int main(void)
{
        int fd;

        if ((fd = open("xxx.txt", O_RDONLY)) == -1)
                exit_sys("open");

        printf("success\n");

        return 0;
}

void exit_sys(const char *msg)
{
        perror(msg);

        exit(EXIT_FAILURE);
}

/*-------------------------------------------------------------------------------------------------
-------------------------
    Bazı programcılar yukarıdaki exit_sys fonksiyonunu printf fonksiyonuna benzetmektedir.
(Örneğin Stevens "Advanced Programming in the
    UNIX Environment)" kitabında böyle bir sarma fonksiyon kullanmıştır. Böyle bir sarma
fonksiyona örnek şu olabilir:

        void exit_sys(const char *format, ...)
        {
                va_list ap;

                va_start(ap, format);
                vfprintf(stderr, format, ap);
                fprintf(stderr, ": %s\n", strerror(errno));

                va_end(ap);

                exit(EXIT_FAILURE);
        }

-------------------------------------------------------------------------------------------------
------------------------*/

#include <stdio.h>
#include <string.h>
```





```
#include <stdlib.h>
#include <stdarg.h>
#include <errno.h>
#include <fcntl.h>

void exit_sys(const char *format, ...);

int main(void)
{
        int fd;
        char path[] = "xxx.txt";

        if ((fd = open(path, O_RDONLY)) == -1)
                exit_sys("open (%s)", path);

        printf("success\n");

        return 0;
}

void exit_sys(const char *format, ...)
{
        va_list ap;

        va_start(ap, format);
        vfprintf(stderr, format, ap);
        fprintf(stderr, ": %s\n", strerror(errno));
        va_end(ap);

        exit(EXIT_FAILURE);
}

/*-------------------------------------------------------------------------------------------------
-------------------------
    C standartlarında errno değeri çok kısıtlı bir biçimde kullanılmıştır. Yani C standartlarında
pek az fonksiyon errno değişkenini
    set etmektedir. Ancak standartlar çeşitli standart C fonksiyonlarının errno değişkenini
derleyiciye bağlı olarak set edebileceğini
    belirtmektedir. POSIX standartlarına göre her standart C fonksiyonu aynı zamanda bir POSIX
fonksiyonu olarak ele alınmaktadır.
    Standart C fonksiyonları aynı zamanda errno değişkenini de set etmektedir. Örneğin biz fopen
fonksiyonu ile bir dosyayı açmak istesek
    fopen başarısız olduğunda UNIX/Linux sistemleri errno değerini uygun biçimde set
edebilmektedir. Böylece biz standart C fonksiyonlarındaki
    hata mesajlarını da aşağıdaki gibi yazdırabilmekteyiz:

        if ((f = fopen("test.dat", "r")) == NULL)
                exit_sys("fopen");

    Ya da örneğin:

        if ((p = malloc(SIZE)) == NULL)
                exit_sys("malloc");

    Her ne kadar standart C fonksiyonları UNIX/Linux sistemlerinde errno değişkenini set ediyorsa
da biz standart C uyumunu korumak için
    kursumuzda standart C fonksiyonlarında set edilen errno değişkenini kullanmayacağız. Örneğin:

        if ((f = fopen("test.dat", "r")) == NULL) {
                fprintf(stderr, "cannot open file!...\n");
                exit(EXIT_FAILURE);
        }

-------------------------------------------------------------------------------------------------
------------------------*/

/*-------------------------------------------------------------------------------------------------
-------------------------
    Aslında errno değişkeni Linux'ta çekirdek tarafından set edilen bir değişken değildir. errno
```





```
değişkeni tamamen user moddaki
    POSIX kütüphanesi tarafından set edilmektedir. Tipik olarak Linux çekirdeğinde bir sistem
fonksiyonu başarısız olduğunda
    negatif errno değerine geri dönmektedir. Sistem fonksiyonunu çağıran POSIX fonksiyonu bu
negatif errno değerini pozitife
    dönüştürerek errno değişkenini set etmektedir.
-------------------------------------------------------------------------------------------------
------------------------*/

/*-------------------------------------------------------------------------------------------------
-------------------------
    UNIX/Linux sistemlerinde her dosyanın bir kullanıcı id'si (user id) ve grup id'si (group id)
bulunmaktadır. Bu sistemlerde
    tüm dosyalar "open" isimli bir POSIX fonksiyonu tarafından yaratılmaktadır. Bir dosyanın
kullanıcı id'si onu yaratan
    prosesin etkin kullanıcı id'si olarak set edilmektedir. Dosyanın grup id'si ise iki seçenekten
biri olarak set edilebilmektedir.
    Bazı sistemler dosyanın grup id'sini onu yaratan prosesin etkin grup id'si olarak set
etmektedir. Bu biçim klasik AT&T
    UNIX sistemlerinin uyguladığı biçimdir. Linux böyle davranmaktadır. İkinci seçenek BSD
sistemlerinde olduğu gibi dosyanın grup
    id'sinin onun içinde bulunduğu dizinin grup id'si olarak set edilmesidir. POSIX standartları
her iki durumu da geçerli kabul etmektedir.
    Linux sistemlerinde "mount parametreleriyle" BSD tarzı davranış istenirse
oluşturulabilmektedir. Aynı zamanda Linux sistemlerinde
    "dosyanın içinde bulunduğu dizinde set group id" bayrağı set edilerek de aynı etki
oluşturulabilmektedir.
-------------------------------------------------------------------------------------------------
------------------------*/

/*-------------------------------------------------------------------------------------------------
-------------------------
                                                                                            8.
Ders 20/11/2022 - Cumartesi
-------------------------------------------------------------------------------------------------
------------------------*/

/*-------------------------------------------------------------------------------------------------
-------------------------
    Bir dosya üzerinde işlem yapmak isteyen proses erişme biçimini de (okumak için mi, yazmak için
mi, hem okuyup hem yazmak için mi,
    yoksa dosyadaki kodu çalıştırmak için mi) belirtmektedir. Bu durumda işletim sistemi sırasıyla
şu kontrolleri yapmaktadır (bu işlemler else-if
    biçiminde sıralanmıştır):

    1) Eğer işlem yapmak isteyen prosesin etkin kullanıcı id'si (etkin grup id'sinin burada önemi
yoktur) 0 ise işlem yapmak isteyen proses
    yetkili kullanıcının bir prosesidir. Bu tür proseslere "root prosesler" ya da "super user
prosesler" ya da "öncelikli (priviledged)
    prosesler" denilmektedir. Bu durumda işletim sistemi yapılmak istenen işlem ne olursa olsun bu
işleme onay verir.

    2) Eğer işlem yapmak isteyen prosesin etkin kullanıcı id'si (effective user id) dosyanın
kullanıcı id'si ile aynıysa bu durumda "dosyanın sahibinin
    dosya üzerinde işlem yaptığı gibi mantıksal bir çıkarım" yapılmaktadır. Yapılmak istenen işlem
ile dosyanın sahiplik (owner) erişim bilgileri karşılaştırılır.
    Eğer bu erişim bilgileri işlemi destekliyorsa işleme onay verilir. Değilse işlem
başarısızlıkla sonuçlanır.

    3) Eğer işlem yapmak isteyen prosesin etkin grup id'si (effective group id) ya da "ek grup
(supplementary groups)" id'lerinden biri dosyanın grup id'si ile
    aynıysa bu durumda "dosya ile aynı grupta bulunan bir kullanıcının dosya üzerinde işlem
yaptığı gibi mantıksal bir çıkarım" yapılmaktadır. Yapılmak istenen
    işlem ile dosyanın grupluk (group) erişim bilgileri karşılaştırılır. Eğer bu erişim bilgileri
işlemi destekliyorsa işleme onay verilir. Değilse işlem
    başarısızlıkla sonuçlanır.

    4) İşlem yapmak isteyen proses herhangi bir proses ise bu durumda yapılmak istenen işlem ile
dosyanın "diğer (other)" erişim bilgileri karşılaştırılır.
```

```
    Eğer bu erişim bilgileri işlemi destekliyorsa işleme onay verilir. Değilse işlem
başarısızlıkla sonuçlanır.

    Örneğin aşağıdaki gibi bir dosya söz konusu olsun:

    -rw-r--r-- 1 kaan study    20 Kas 13 13:54 test.txt

    Dosyaya erişim yapmak isteyen proses, "okuma ve yazma amaçlı" erişim yapmak istesin. Eğer
prosesin etkin kullanıcı id'si 0 ise bu işlem onaylanacaktır.
    Eğer prosesin etkin kullanıcı id'si "kaan" ise bu işlem yine onaylanacaktır. Ancak prosesin
etkin grup id'si ya da ek grup id'lerinden biri study ise
    işlem onaylanmayacaktır. Çünkü erişim hakları gruptaki üyelere yalnızca okuma izni
vermektedir. Benzer biçimde prosesin etkin kullanıcı id'si ya da etkin
    grup id'si (ve ek grup id'leri) burada belirtilenlerin dışında ise yine prosese bu işlem için
onay verilmeyecektir.

    Yukarıdaki maddeler else-if biçiminde düşünülmelidir. Örneğin dosya aşağıdaki gibi olsun:

    -r--rw-r-- 1 kaan study    20 Kas 13 13:54 test.txt

    Burada dosyanın sahibi (yani etkin kullanıcı id'si dosyanın kullanıcı id'si ile aynı olan
proses) dosya üzerinde yazma yapamayacaktır. Ancak aynı
    grupta olan prosesler bunu yapabilecektir. Tabii bu biçimdeki erişim hakları mantıksal olarak
tuhaf ve anlamsızdır. Yani dosyanın sahibine
    verilmeyen bir hakkın gruba ya da diğerlerine verilmesi normal bir durum değildir.

    Çalıştırılabilir bir dosya 'x' hakkı ile temsil edilmiştir. Bu durumda biz bir program
dosyasının başkaları tarafından çalıştırılması engelleyebiliriz. Örneğin:

    -rwxr--r-- 1 kaan study 16816 Kas 13 13:49 sample

    Burada dosyanın sahibi (ve tabii root kullanıcısı) bu dosyayı çalıştırabilir. Ancak diğer
kullanıcılar bu dosyayı çalıştıramazlar. Örneğin:

    -rw-r--r-- 1 kaan study 16816 Kas 13 13:49 sample

    Burada artık root kullanıcısı da dosyayı çalıştıramaz. root kullanıcısının dosyayı
çalıştırabilmesi için sahiplik, grupluk ya da diğer erişim bilgilerinin
    en az birinde 'x' hakkının belirtilmiş olması gerekmektedir.
-------------------------------------------------------------------------------------------------
------------------------*/

/*-------------------------------------------------------------------------------------------------
-------------------------
    POSIX standartlarında erişim mekanizması üzerinde açıklamalar yapılırken "root önceliği" ya da
"prosesin etkin kullanıcı id'sinin
    0 olması" gibi bir anlatım uygulanmamıştır. Onun yerine POSIX standartlarında "appropriate
privileges" terimi kullanılmıştır.
    Çünkü bir POSIX sistemi "ya hep ya hiç" biçiminde tasarlanmak zorunda değildir. Gerçekten de
örneğin Linux sistemlerinde "capability"
    denilen bir özellik bulunmaktadır. Bu "capability" sayesinde bir prosesin etkin kullanıcı
id'si 0 olmamasına karşın o proses belirlenen bazı şeyleri
    yapabilir duruma getirilebilmektedir. İşte POSIX standartlarındaki "appropriate privileges"
terimi bunu anlatmaktadır. Yani buradaki
    "appropriate privileges" terimi "prosesin etkin kullanıcı id'si 0 ya da 0 olmasa da prosesin
bu işlemi yapabilme yeteneğinin" olduğunu
    belirtmektedir.
-------------------------------------------------------------------------------------------------
------------------------*/

/*-------------------------------------------------------------------------------------------------
-------------------------
    Prosesin çalışma dizini getcwd isimli POSIX fonksiyonuyla elde edilebilmektedir. Fonksiyonun
prototipi şöyledir:

        #include <unistd.h>

        char *getcwd(char *buf, size_t size);
```





```
    Fonksiyonun birinci parametresi yol ifadesinin yerleştirileceği dizinin adresini, ikinci
parametresi ise bu dizinin null karakter
    dahil olmak üzere uzunluğunu almaktadır. Fonksiyon başarı durumunda birinci parametresiyle
belirtilen adresin aynısına, başarısızlık durumunda
    NULL adrese geri dönmektedir. Fonksiyonun ikinci parametresinde belirtilen uzunluk eğer yol
ifadesini ve null karakteri içerecek büyüklükte değilse
    fonksiyon başarısız olmaktadır.

    UNIX/Linux sistemlerinde bir yol ifadesinin maksimum karakter sayısı (null karakter dahil
olmak üzere) <limits.h> içerisindeki
    PATH_MAX sembolik sabitiyle belirtilmiştir. Ancak bu konuda bazı ayrıntılar vardır. Bazı
sistemlerde bu PATH_MAX sembolik sabiti tanımlı değildir.
    Dolayısıyla bazı sistemlerde maksimum yol ifadesi uzunluğu pathconf denilen özel bir fonksiyon
ile elde edilebilmektedir. Linux sistemlerinde
    <limits.h> dosyası içerisinde PATH_MAX 4096 olarak define edilmiştir.
-------------------------------------------------------------------------------------------------
------------------------*/

#include <stdio.h>
#include <stdlib.h>
#include <limits.h>
#include <unistd.h>

void exit_sys(const char *msg);

int main(void)
{
        char buf[PATH_MAX];

        if (getcwd(buf, PATH_MAX) == NULL)
                exit_sys("getcwd");

        puts(buf);

        return 0;
}

void exit_sys(const char *msg)
{
        perror(msg);

        exit(EXIT_FAILURE);
}

/*-------------------------------------------------------------------------------------------------
-------------------------
    Prosesin çalışma dizinini chdir isimli POSIX fonksiyonuyla değiştirebiliriz. Fonksiyonun
prototipi şöyledir:

        #include <unistd.h>

        int chdir(const char *path);

    Fonksiyon yeni çalışma dizinin yol ifadesini parametre olarak alır. Başarı durumunda 0
değerine, başarısızlık durumunda -1 değerine
    geri döner.
-------------------------------------------------------------------------------------------------
------------------------*/

#include <stdio.h>
#include <stdlib.h>
#include <limits.h>
#include <unistd.h>

void exit_sys(const char *msg);

int main(void)
{
        char buf[PATH_MAX];
```





```
        if (getcwd(buf, PATH_MAX) == NULL)
                exit_sys("getcwd");

        puts(buf);

        if (chdir("/usr/bin") == -1)
                exit_sys("chdir");

        if (getcwd(buf, PATH_MAX) == NULL)
                exit_sys("getcwd");

        puts(buf);

        return 0;
}

void exit_sys(const char *msg)
{
        perror(msg);

        exit(EXIT_FAILURE);
}

/*-------------------------------------------------------------------------------------------------
-------------------------
    Daha önce yazmış olduğumuz kabuk programına cd komutunu aşağıdaki gibi ekleyebiliriz. Bu
örnekteki getenv fonksiyonunu henüz görmedik.
-------------------------------------------------------------------------------------------------
------------------------*/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>
#include <limits.h>
#include <unistd.h>

#define MAX_CMD_LINE            4096
#define MAX_CMD_PARAMS          128

typedef struct tagCMD {
        char *name;
        void (*proc)(void);
} CMD;

void parse_cmd_line(char *cmdline);
void dir_proc(void);
void clear_proc(void);
void pwd_proc(void);
void cd_proc(void);

void exit_sys(const char *msg);

char *g_params[MAX_CMD_PARAMS];
int g_nparams;
char g_cwd[PATH_MAX];

CMD g_cmds[] = {
        {"dir", dir_proc},
        {"clear", clear_proc},
        {"pwd", pwd_proc},
        {"cd", cd_proc},
        {NULL, NULL}
};

int main(void)
{
        char cmdline[MAX_CMD_LINE];
```





```
        char *str;
        int i;

        if (getcwd(g_cwd, PATH_MAX) == NULL)
                exit_sys("fatal error (getcwd)");

        for (;;) {
                printf("CSD:%s>", g_cwd);
                if (fgets(cmdline, MAX_CMD_LINE, stdin) == NULL)
                        continue;
                if ((str = strchr(cmdline, '\n')) != NULL)
                        *str = '\0';
                parse_cmd_line(cmdline);
                if (g_nparams == 0)
                        continue;
                if (!strcmp(g_params[0], "exit"))
                        break;
                for (i = 0; g_cmds[i].name != NULL; ++i)
                        if (!strcmp(g_params[0], g_cmds[i].name)) {
                                g_cmds[i].proc();
                                break;
                        }
                if (g_cmds[i].name == NULL)
                        printf("bad command: %s\n", g_params[0]);
        }

        return 0;
}

void parse_cmd_line(char *cmdline)
{
        char *str;

        g_nparams = 0;
        for (str = strtok(cmdline, " \t"); str != NULL; str = strtok(NULL, " \t"))
                g_params[g_nparams++] = str;
}

void dir_proc(void)
{
        printf("dir command executing...\n");
}

void clear_proc(void)
{
        system("clear");
}

void pwd_proc(void)
{
        printf("%s\n", g_cwd);
}

void cd_proc(void)
{
        char *dir;

        if (g_nparams > 2) {
                printf("too many arguments!\n");
                return;
        }
        if (g_nparams == 1) {
                if ((dir = getenv("HOME")) == NULL)
                        exit_sys("fatal error (getenv");
        }
        else
                dir = g_params[1];

        if (chdir(dir) == -1) {
```

```
                printf("%s\n", strerror(errno));
                return;
        }

        if (getcwd(g_cwd, PATH_MAX) == NULL)
                exit_sys("fatal error (getcwd)");
}

void exit_sys(const char *msg)
{
        perror(msg);

        exit(EXIT_FAILURE);
}

/*--------------------------------------------------------------------------------------------------------
------------------------
                                                                                                    9.
Ders 20/11/2022 - Pazar
--------------------------------------------------------------------------------------------------------
------------------------*/

/*--------------------------------------------------------------------------------------------------------
------------------------
    Dizinler de işletim sistemi tarafından birer dosyaymış gibi ele alınmaktadır. Gerçekten de
dizinleri sanki "içerisinde
    dosya bilgilerini tutan dosyalar" gibi düşünebiliriz. Dolayısıyla UNIX/Linux sistemlerinde bir
dosyayı silmek için,
    bir dosya yaratmak için, bir dosyanın ismini değiştirmek için prosesin o dizine "w" hakkının
olması gerekir. Yukarıda belirttiğimiz
    üç işlem de aslında dizine yazma yapma anlamına gelmektedir. Yani bizim bir dosyayı silebilmek
için dosyaya "w" hakkına sahip olmamız
    gerekmez, dosyanın içinde bulunduğu dizine "w" hakkına sahip olmamız gerekir. Bir dizin için
"r" hakkı demek o dizinin içeriğinin
    okunabilmesi hakkı demektir. Yani bizim bir dizinin içeriğini elde edebilmemiz (ya da ls gibi
bir komutla görüntüleyebilmemiz)
    için o dizine "r" hakkına sahip olmamız gerekir.
--------------------------------------------------------------------------------------------------------
------------------------*/

/*--------------------------------------------------------------------------------------------------------
------------------------
    Dizinlerde "x" hakları farklı bir anlama gelmektedir. İşletim sistemi, bir yol ifadesi
verildiğinde yol ifadesinde hedeflenen
    dizin girişi için bilgileri elde etmek isteyecektir. Örneğin:

    "/home/kaan/Study/C/sample.c"

    Burada hedeflenen dosya "sample.c" dosyasıdır. Ancak işletim sistemi bu dosyanın yerini
bulabilmek için yol ifadesindeki
    bileşenlerin üzerinden geçer. Bu işleme "pathname resolution" denilmektedir. İşte "pathname
resolution" işleminde dizin geçişleriyle
    hedefe ulaşılabilmesi için prosesin yol ifadesine ilişkin dizin bileşenlerinin "x" hakkına
sahip olması gerekir. Yani dizinlerdeki "x"
    hakkı "içinden geçilebilirlik" gibi bir anlama gelmektedir. Biz bir dizinimizdeki "x" hakkını
kaldırırsak, işletim sistemi pathname
    resolution işleminde başarısız olur. Dolayısıyla pathname resolution işleminin başarılı
olabilmesi için yol ifadesindeki
    dizin bileşenlerinin hepsine (son bileşen de dahil olmak üzere) prosesin "x" hakkına sahip
olması gerekir. Yukarıdaki örnekte
    pathname resolution işleminin bitirilebilmesi için prosesin "home" dizini "kaan" dizini
"Study" dizini ve "C" dizini için "x"
    hakkına sahip olması gerekir. "x" hakkı bir dizin ağacında bir noktaya duvar örmek için
kullanılabilmektedir. mkdir gibi
    kabuk komutları dizin yaratırken zaten "x" hakkını default durumda vermektedir. Proses id'si 0
olan "root" prosesler her zaman
    pathname resolution sırasında dizinler içerisinden geçebilirler.
--------------------------------------------------------------------------------------------------------
------------------------*/
```





```
/*--------------------------------------------------------------------------------------------------------
------------------------
    Bir işletim sisteminin dosyalarla uğraşan kısmına "dosya sistemi (file system)" denilmektedir.
Dosya sisteminin iki yönü vardır:
    Disk ve Bellek. İşletim sistemi dosya organizasyonu için diskte belli bir biçim
kullanmaktadır. Ancak bir dosya açıldığında işletim sistemi
    çekirdek alanı içerisinde bazı veri yapıları oluşturur bu da dosya sisteminin bellek tarafı
ile ilgilidir.

    Pek çok POSIX uyumlu işletim sistemi dosya işlemleri için 5 sistem bulundurmaktadır:

    - Dosya açmak için gereken sistem fonksiyonu (Linux'ta sys_open)
    - Dosya kapatmak için gereken sistem fonksiyonu (Linux'ta sys_close)
    - Dosyadan okuma yapmak için gereken sistem fonksiyonu (Linux'ta sys_read)
    - Dosyaya yazma yapmak için gereken sistem fonksiyonu (Linux'ta sys_write)
    - Dosya göstericisini konumlandırmak için gereken sistem fonksiyonu (Linux'ta sys_lseek)

    Bu 5 sistem fonksiyonunu çağıran 5 POSIX fonksiyonu bulunmaktadır: open, close, read, write ve
lseek

    Biz bir UNIX/Linux sisteminde hangi düzeyde çalışıyor olursak olalım eninde sonunda dosya
işlemleri bu 5 POSIX fonksiyonu ile yapılmaktadır.
    Programlama dili ne olursa olsun durum böyledir.
--------------------------------------------------------------------------------------------------------
------------------------*/

/*--------------------------------------------------------------------------------------------------------
------------------------
    Bir dosya açıldığında işletim sistemi açılacak dosyanın bilgilerini pathname resolution işlemi
sonucunda diskte bulur.
    Dosyanın bilgilerini kernel alanı içerisinde bir alana çeker. Bu alana "dosya nesnesi (file
object)" denilmektedir. Buradaki "nesne (object)"
    terimi tahsis edilmiş yapı alanları için kullanılmaktadır. Dosya nesnesi Linux'un kaynak
kodlarında "struct file" ile temsil edilmiştir.
    İşletim sistemi, bir proses bir dosyayı açtığında açılan dosyayı o proses ile ilişkilendirir.
Yani dosya nesnelerine proses kontrol blokları
    yoluyla erişilmektedir. Güncel Linux çekirdeklerinde bu durum biraz karmaşıktır:

    task_struct (files) ---> files_struct (fdt) ---> fdtable (fd) ---> file * türünden bir dizi --
-> file

    Linux'ta proses kontrol bloktan dosya nesnesine erişim birkaç yapıdan geçilerek yapılmaktadır.
Ancak biz bu durumu
    şöyle basitleştirerek ifade edebiliriz: "proses kontrol blokta bir eleman bir diziyi
göstermektedir. Bu diziye "dosya betimleyici tablosu (file desctiptor table)"
    denilmektedir. Dosya betimleyici tablosunun her elemanı bir dosya nesnesini göstermektedir.
Yani biz yukarıdaki yapıyı aşağıdaki gibi
    sadeleştirerek kavramsallaştırıyoruz:

    proses kontrol block ---> betimleyici tablosu --> dosya nesneleri

    Dosya betimleyici tablosu (file descriptor table) açık dosyalara ilişkin dosya nesnelerinin
adreslerini tutan bir gösterici dizisidir.
    Dosya betimleyici tablosuna proses kontrol bloktan hareketle erişilmektedir. Her prosesin ayrı
bir dosya betimleyici tablosu vardır.
    İşletim sistemi her açılan dosya için bir dosya nesnesi tahsis etmektedir. Aynı dosya ikinci
kez açıldığında o dosya için yine yeni bir
    dosya nesnesi oluşturulur. Dosya göstericisinin konumu da dosya nesnesinin içerisinde
saklanmaktadır.
--------------------------------------------------------------------------------------------------------
------------------------*/

/*--------------------------------------------------------------------------------------------------------
------------------------
    UNIX/Linux sistemlerinde dosyayı açmak için open isimli POSIX fonksiyonu kullanılmaktadır.
(Örneğin fopen standart C fonksiyonu da
    UNIX/Linux sistemlerinde aslında open fonksiyonunu çağırmaktadır.) Fonksiyonun prototipi
şöyledir:
```





```
        #include <fcntl.h>

        int open(const char *path, int flags, ...);

    open fonksiyonu isteğe bağlı (optional) olarak bir üçüncü argüman alabilmektedir. Eğer
fonksiyon 3 argümanla çağrılacaksa üçüncü argüman
    mode_t türünden olmalıdır. Her ne kadar prototipteki "..." "istenildiği kadar argüman
girilebilir" anlamına geliyorsa da open ya iki argümanla
    ya da üç argümanla çağrılmalıdır. open fonksiyonunu daha fazla argümanla çağırmak "tanımsız
davranışa (undefined behavior)" yol açmaktadır.

    Fonksiyonun birinci parametresi açılacak dosyanın yol ifadesini belirtir. İkinci parametre
açış bayraklarını (modlarını) belirtmektedir.
    Bu parametre O_XXX biçiminde isimlendirilmiş sembolik sabitlerin "bit OR" işlemine
sokulmasıyla oluşturulur. Açış sırasında aşağıdaki sembolik sabitlerden
    yalnızca biri belirtilmek zorundadır.

    O_RDONLY
    O_WRONLY
    O_RDWR
    O_SEARCH (at'li fonksiyonlar için bulundurulmuştur, ileride ele alınacaktır)
    O_EXEC (fexecve fonksiyonu için bulundurulmuştur, ileride ele alınacaktır)

    Buradaki O_RDONLY "yalnızca okuma yapma amacıyla", O_WRONLY "yalnızca yazma yapma amacıyla" ve
O_RDWR "hem okuma hem de yazma yapma amacıyla"
    dosyanın açılmak istendiği anlamına gelmektedir. İşletim sistemi, prosesin etkin kullanıcı
id'sine ve etkin grup id'sine ve dosyanın kullanıcı ve grup
    id'lerine bakarak prosesin dosyaya "r", "w" hakkının olup olmadığını kontrol eder. Eğer proses
bu hakka sahip değilse open fonksiyonu başarısız olur.
    Buradaki O_SEARCH bayrağı bazı POSIX fonksiyonlarının "at"li versiyonları için, O_EXEC bayrağı
ise "fexecve" fonksiyonu için bulundurulmuştur. Bu bayraklar
    ileride ele alınacaktır.

    open fonksiyonu yalnızca olan dosyayı açmak için değil aynı zamanda yeni bir dosya yaratmak
için de kullanılmaktadır. O_CREAT bayrağı
    dosya varsa etkili olmaz. Ancak dosya yoksa dosyanın yaratılmasını sağlar. Yani O_CREAT
bayrağı "dosya varsa olanı aç, yoksa yarat ve aç"
    anlamına gelmektedir. Bir dosya yaratılırken dosyanın erişim haklarını, dosyayı yaratan kişi
open fonksiyonun üçüncü parametresinde vermek zorundadır.
    Yani dosyanın erişim haklarını dosyayı yaratan kişi belirlemektedir. Biz O_CREAT bayrağını
açış moduna eklemişsek bu durumda "dosya yaratılabilir"
    fikri ile erişim haklarını open fonksiyonun üçüncü parametresinde belirtmemiz gerekir. Erişim
hakları tüm bitleri sıfır tek biti 1 olan sembolik sabitlerin
    "bit OR" işlemine sokulmasıyla oluşturulmaktadır. Bu sembolik sabitlerin hepsi S_I öneki
başlar. Bunu R, W ya da X harfi izler. Bunu da USR, GRP ya da OTH
    harfleri izlemektedir. Böylece 9 tane erişim hakkı şöyle isimlendirilmiştir:

    S_IRUSR
    S_IWUSR
    S_IXUSR
    S_IRGRP
    S_IWGRP
    S_IXGRP
    S_IROTH
    S_IWOTH
    S_IXOTH

    Örneğin S_IRUSR|S_IWUSR|S_IRGRP|S_IROTH erişim hakları "rw-r--r--" anlamına gelmektedir.

    Ayrıca <sys/stat.h> içerisisinde aşağıdaki sembolik sabitler de bildirilmiştir:

    S_IRWXU
    S_IRWXG
    S_IRWXO

    Bu sembolik sabitler şöyle oluşturulmuştur:

    #define S_IRWXU (S_IRUSR|S_IWUSR|S_IXUSR)
```





```
    #define S_IRWXG (S_IRGRP|S_IWGRP|S_IXGRP)
    #define S_IRWXO (S_IROTH|S_IWOTH|S_IXOTH)

    Bu durumda örneğin S_IRWXU|S_IRWXG|S_IRWXO işlemi "rwxrwxrwx" anlamına gelmektedir.

    Yukarıdaki S_IXXX biçimindeki sembolik sabitlerin değerlerinin eskiden sistemden sisteme
değişebileceği dikkate alınmıştır. Bu nedenle
    POSIX standartları başlarda bu sembolik sabitlerin sayısal değerlerini işletim sistemlerini
oluşturanların belirlemesini istemiştir.
    Ancak daha sonraları (2008 ve sonrasında, SUS 4) bu sembolik sabitlerin değerleri POSIX
standartlarında açıkça belirtilmiştir. Dolayısıyla
    programcılar artık bu sembolik sabitleri kullanmak yerine bunların sayısal karşılıklarını da
kullanabilir duruma gelmiştir. Ancak eski
    sistemler dikkate alındığında bunların sayısal karşılıkları yerine yukarıdaki sembolik
sabitlerin kullanılması tavsiye edilmektedir. Bu
    sembolik sabitler aynı zamanda okunabilirliği de artırmaktadır. POSIX standartları belli bir
sürümden sonra bu sembolik sabitlerin sayısal
    değerlerini aşağıdaki gibi belirlemiştir:

    S_IRWXU             0700
    S_IRUSR             0400
    S_IWUSR             0200
    S_IXUSR             0100
    S_IRWXG             070
    S_IRGRP             040
    S_IWGRP             020
    S_IXGRP             010
    S_IRWXO             07
    S_IROTH             04
    S_IWOTH             02
    S_IXOTH             01
    S_ISUID             04000
    S_ISGID             02000
    S_ISVTX             01000

    Yani belli bir süreden sonra artık rwxrwxrwx biçiminde owner, group ve other bilgilerine
ilişkin S_IXXX biçimindeki sembolik sabitler
    gerçekten yukarıdaki sıraya göre bitleri temsil eder hale gelmiştir. Örneğin S_IWGRP sembolik
sabiti 000010000 bitlerinden oluşmaktadır.
    Bu durumda belli bir süreden sonra örneğin S_IRUSR|S_IWURS|S_IRGRP|S_IROTH bir erişim hakkını
biz 0644 octal değeri ile edebiliriz.
    Bu sembolik sabitlerin binary karşılıklarını da vermek istiyoruz.

    S_IRUSR        100 000 000
    S_IWUSR        010 000 000
    S_IXUSR        001 000 000

    S_IRGRP        000 100 000
    S_IWGRP        000 010 000
    S_IXGRP        001 001 000

    S_IROTH        000 000 100
    S_IWOTH        000 010 010
    S_IXOTH        001 001 001

    open fonksiyonunda O_CREAT bayrağı belirtilmemişse erişim haklarının girilmesinin hiçbir
anlamı yoktur. Kaldı ki
    O_CREAT bayrağı girildiğinde dosya varsa erişim hakları yine dikkate alınmayacaktır.

    POSIX sistemlerinde yukarıdaki S_IXXX biçimindeki sembolik sabitler mode_t türüyle temsil
edilmiştir. mode_t türü
    <sys/types.h> ve bazı başlık dosyalarında sistemi oluşturanların belirlediği bir tamsayı türü
olarak typedef edilmiştir.

    O_TRUNC açış bayrağı "eğer dosya varsa onu sıfırlayarak aç" anlamına gelmektedir. Ancak
O_TRUNC ancak yazma modunda açılan
    dosyalarda kullanılabilmektedir. Yani O_TRUNC bayrağını kullanabilmek için O_WRONLY ya da
O_RDWR bayraklarından birinin de
    belirtilmiş olması gerekmektedir. Örneğin O_WRONLY|O_CREAT|O_TRUNC açış modu "dosya yoksa
```

```
yarat ancak varsa sıfırlayarak aç"
    anlamına gelmektedir. O_TRUNC bayrağı için dosyanın yaratılıyor olması gerekmez.
O_WRONLY|O_TRUNC geçerli bir açış modudur.
    Bu durumda dosya yoksa open başarısız olur. Ancak dosya varsa sıfırlanarak açılır.

    O_APPEND bayrağı yazma işlemlerinin dosyanın sonuna yapılacağı anlamına gelmektedir. Yani bu
bayrak kullanılırsa tüm yazma
    işlemlerinde işletim sistemi dosya göstericisini dosyanın sonuna çekip sonra yazmayı
yapmaktadır. Bu açış modu da O_WRONLY
    ya da O_RDWR için anlamlıdır. Örneğin O_RDWR|O_APPEND burada dosyaya her yazılan sona
eklenecektir. Ancak dosyanın herhangi bir yerinden
    okuma yapılabilecektir.

    O halde standart C'nin fopen fonksiyonundaki açış modlarının POSIX karşılıkları şöyle
oluşturulabilir:

    Standart C fopen                    POSIX

    "w"                                 O_WRONLY|O_CREAT|O_TRUNC
    "w+"                                O_RDWR|O_CREAT|O_TRUNC
    "r"                                 O_RDONLY
    "r+"                                O_RDWR
    "a"                                 O_WRONLY|O_CREAT|O_APPEND
    "a+"                                O_RDWR|O_CREAT|O_APPEND

    O_EXCL bayrağı "exclusive" açım kullanılmaktadır. Bu bayrak O_CREAT ile birlikte
kullanılmalıdır. O_CREAT|O_EXCL biçiminde
    açış modu "dosya yoksa yarat, varsa yaratma başarısız ol" anlamına gelmektedir. O_EXCL
bayrağının O_CREAT olmadan kullanılması
    "tanımsız davranışa" yol açmaktadır.

    O_DIRECTORY bayrağının tek işlevi açılmak istenen dosya bir dizin dosyası değilse açımın
başarısız olmasını sağlamak içindir.

    open fonksiyonunun diğer açış modları ileride başka konular içerisinde ele alınacaktır.

    Erişim hakları open fonksiyonu tarafından (yani open fonksiyonunun çağırdığı sistem fonksiyonu
tarafından) kontrol edilmektedir.
    Örneğin biz dosyayı O_RDWR modunda açmak isteyelim bu durumda prosesimizin dosyaya "r" ve "w"
haklarına sahip olması gerekir. Eğer
    prosesimiz dosya için bu haklara sahip değilse open başarısız olur ve errno EACCESS değeri ile
set edilir. Burada önemli olan nokta
    kontrolün en başta open tarafından yapılmasıdır. Yani O_RDWR modunda açma istendiğinde eğer
proses bu haklara sahip değilse
    açma başarılı olup read ya da write fonksiyonlarındaki hatadan dolayı başarısız olma söz
konusu değildir. Direkt açmanın kendisi
    başarısız olmaktadır.

    open fonksiyonu başarı durumunda int türden "dosya betimleyicisi (file descriptor)" denilen
bir değerle geri dönmektedir.
    Dosya betimleyicisi bir handle olarak diğer fonksiyonlar tarafından istenmektedir. open
başarısız olursa -1 ile geri döner ve errno
    uygun biçimde set edilir. open fonksiyonunun başarısız olması için pek çok neden söz
konusudur. Bundan dolayı açma işleminin başarısı
    kesinlikle test edilmelidir.
-------------------------------------------------------------------------------------------------
-------------------------*/

/*-------------------------------------------------------------------------------------------------
-------------------------

10. Ders 26/11/2022 - Cumartesi
-------------------------------------------------------------------------------------------------
-------------------------*/

/*-------------------------------------------------------------------------------------------------
-------------------------

    open fonksiyonu işletim sisteminin dosya açan sistem fonksiyonunu (Linux'ta sys_open)
```





```
çağırmaktadır. Bu sistem fonksiyonu
    açılacak dosyaya ilişkin bilgileri diskten bulur ve o bilgileri "dosya nesnesi (file object)"
denilen bir yapının içerisine
    yerleştirir. Dosya nesnesi Linux'un kaynak kodlarında "struct file" türü ile temsil
edilmiştir. İşletim sistemi dosya nesnesinin
    içini doldurduktan sonra dosya betimleyici tablosunda boş bir slot bulur ve o slota dosya
nesnesinin adresini yazar. Anımsanacağı gibi
    dosya betimleyici tablosu dosya nesnelerinin adreslerini tutan bir gösterici dizisi biçiminde
organize edilmiştir. Dosya betimleyici tablosunun
    yeri prosesin kontrol bloğundan hareketle elde edilmektedir. İşte open fonksiyonunun bize geri
döndürdüğü dosya betimleyicisi aslında
    dosya betimleyici tablosunda (yani gösterici dizisinde) bir indeks belirtmektedir.

    Bir program çalıştığında genellikle dosya betimleyici tablosunun ilk üç betimleyicisi dolu
diğerleri boştur. Dosya betimleyici tablosunun
    0'ıncı slotu (yani 0 numaralı betimleyici) terminal aygıt sürücüsü için oluşturulmuş dosya
nesnesini göstermektedir. Buna stdin
    dosya betimleyicisi denilmektedir. 1 ve 2 numaralı betimleyiciler yine terminal aygıt sürücüsü
oluşturulmuş dosya betimleyicisini göstermektedir.
    (1 ve 2 numaralı betimleyiciler aynı nesneyi göstermektedir) Bu betimleyicilere de sırasıyla
stdout ve stderr denilmektedir.
    Böylece ilk boş betimleyici genellikle 3 numaralı betimleyici olmaktadır. open fonksiyonun
dosya betimleyici tablosunda ilk boş betimleyiciyi vermesi
    POSIX standartlarında garanti edilmiştir.

    Her prosesin proses kontrol bloğu ve dolayısıyla dosya betimleyici tablosu birbirinden
farklıdır. O halde dosya betimleyicileri
    kendi prosesinin dosya betimleyici tablosunda bir indeks belirtmektedir. Yani dosya
betimleyicileri prosese özgü bir anlama sahiptir.

    Bu durumda tipik olarak işletim sisteminin dosya açan sistem fonksiyonu sırasıyla şu işlemleri
yapmaktadır:

    1) Dosya betimleyici tablosunda ilk boş betimleyiciyi bulmaya çalışır. Boş betimleyiciyi
bulamazsa başarısız olur ve errno
    değerini EMFILE ise set eder.

    2) Dosya nesnesini tahsis eder ve bunun içini diskten elden ettiği bilgilerle doldurur. Bunun
adresini de dosya betimleyici tablosunda
    ilk boş betimleyiciye ilişkin slota yazar.

    3) Dosya betimleyici tablosunda indeks belirten betimleyici ile geri döner.

    C'nin fopen fonksiyonunda dosya açımı sırasında "text mode", "binary mode" gibi bir kavram
vardır. Halbuki işletim sisteminde
    böyle bir kavram yoktur. İşletim sistemine göre dosya byte'lardan oluşmaktadır. Text mode,
binary mode C ve diğer diller tarafından
    uydurulmuş olan yapay bir kavramdır.

    Bir proses her open işlemi yaptığında kesinlikle yeni bir dosya nesnesi oluşturulur. Bu
durumda bir proses aynı dosyayı aynı biçimde
    ikinci kez açmış olsa bile dosya aynı dosya nesnesi kullanılmaz. Her iki open iki farklı dosya
nesnesinin ve dosya betimleyicisinin
    oluşmasına yol açmaktadır.
-------------------------------------------------------------------------------------------------
-------------------------*/

#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <sys/stat.h>

void exit_sys(const char *msg);

int main(void)
{
        int fd;

        if ((fd = open("test.txt", O_WRONLY|O_CREAT|O_TRUNC, S_IRUSR|S_IWUSR|S_IRGRP|S_IROTH)) ==
```





```
-1)
                exit_sys("open");

        printf("file opened: %d\n", fd);

        return 0;
}

void exit_sys(const char *msg)
{
        perror(msg);

        exit(EXIT_FAILURE);
}

/*-------------------------------------------------------------------------------------------------
-------------------------
    Açılan her dosyanın kapatılması gerekir. Bir dosyanın kapatılması sırasında işletim sistemi
dosyanın açılması sırasında yapılan
    işlemleri geri almaktadır. Tipik olarak UNIX/Linux sistemlerinde dosya kapatıldığında şunlar
yapılmaktadır:

    1) Dosya nesnesi eğer onu gösteren tek bir betimleyici varsa yok edilir.
    2) Dosya betimleyici tablosundaki betimleyiciye ilişkin slot boşaltılır.

    İleride de görüleceği gibi dosya betimleyici tablosunda birden fazla betimleyici aynı dosya
nesnesini gösteriyor durumda olabilir.
    Bu durumda işletim sistemi dosya nesnesi içerisinde bir sayaç tutup bu sayacı artırıp
eksiltmektedir. Sayaç 0'a geldiğinde nesneyi silmektedir.
    (Linux'un kaynak kodlarında bu sayaç struct file yapısının f_count elemanında tutulmaktadır.)

    Bir dosya artık kullanılmayacaksa onu kapatmak iyi bir tekniktir. Çünkü bu sayede:

    1) Dosya betimleyici tablosunda gereksiz bir slot tahsis edilmiş durumda olmaz.
    2) Dosya nesnesi gereksiz bir biçimde kernel alanı içerisinde yer kaplamaz.

    Tabii işletim sistemi, proses dosyayı kapatmasa bile proses sonlandırılırken dosya prosesin
dosya betimleyici tablosunu inceler
    ve açık dosyaları bu biçimde kapatır. Yani biz bir dosyayı kapatmasak bile proses bittiğinde
dosyalar zaten kapatılmaktadır.

    Dosyanın kapatılması için close isimli POSIX fonksiyonu bulundurulmuştur. Bu POSIX fonksiyonu
doğrudan işletim sisteminin
    dosyayı kapatan sistem fonksiyonunu çağırmaktadır. close fonksiyonunun prototipi şöyledir:

        #include <unistd.h>

        int close(int fd);

    Fonksiyon parametre olarak dosya betimleyicisini alır. close fonksiyonu başarı durumunda 0,
başarısızlık durumunda -1 değerine geri
    dönmektedir. Fonksiyonun geri dönüş değeri genellikle kontrol edilmez.
-------------------------------------------------------------------------------------------------
-------------------------*/

#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <sys/stat.h>

void exit_sys(const char *msg);

int main(void)
{
        int fd;

        if ((fd = open("test.txt", O_WRONLY|O_CREAT|O_TRUNC, S_IRUSR|S_IWUSR|S_IRGRP|S_IROTH)) ==
-1)
                exit_sys("open");
```





```
        printf("file opened: %d\n", fd);

        close(fd);

        return 0;
}

void exit_sys(const char *msg)
{
        perror(msg);

        exit(EXIT_FAILURE);
}

/*-------------------------------------------------------------------------------------------------
-------------------------
    İlk UNIX sistemlerinden beri creat isimli bir fonksiyon da open fonksiyonun bir sarma
fonksiyonu biçiminde bulundurulmaktadır.
    creat fonksiyonu POSIX standartlarında var olan bir fonksiyondur. Fonksiyonun prototipi
şöyledir:

        #include <fcntl.h>

        int creat(const char *path, mode_t mode);

    Fonksiyonun birinci parametresi dosyanın yol ifadesini belirtmektedir. İkinci parametre erişim
bilgisini belirtir. Görüldüğü gibi
    fonksiyonda açış modu belirten flags parametresi yoktur. Çünkü bu parametre
O_WRONLY|O_CREAT|O_TRUNC biçiminde alınmaktadır. creat fonksiyonu
    aşağıdaki gibi yazılmıştır:

        int creat(const char *path, mode_t mode)
        {
                return open(path, O_WRONLY|O_CREAT|O_TRUNC, mode);
        }

-------------------------------------------------------------------------------------------------
-------------------------*/

/*-------------------------------------------------------------------------------------------------
-------------------------
    Dosyadaki her bir byte'a bir offset numarası karşı getirilmiştir. Buna ilgili byte'ın offset'i
denilmektedir. Dosya göstericisi
    okuma ve yazma işlemlerinin hangi offset'ten itibaren yapılacağını gösteren bir offset
belirtmektedir. Okuma ya da yazma miktarı
    kadar dosya göstericisi otomatik olarak ilerletilmektedir. Dosya ilk açıldığında dosya
göstericisi 0 durumundadır. Dosya göstericisinin
    dosyanın son byte'ından sonraki byte'ı göstermesi durumuna EOF durumu denir. EOF durumunda
okuma yapılamaz. Ancak yazma yapılabilir.
    Bu durumda yazılanlar dosyaya eklenmiş olur. Dosyada araya bir şey eklemek (insert) diye bir
kavram yoktur. Dosya boyutunu değiştirmek
    için dosya göstericisi EOF'a çekilip yazma yapılmalıdır.

    Dosya göstericisin konumu dosya nesnesi içerisinde saklanmaktadır. (Linux'un kaynak kodlarında
"struct file" yapısının f_pos
    elemanı dosya göstericisinin konumunu tutmaktadır.) Biz aynı dosyayı ikinci kez açmış olsak
bile yeni bir dosya nesnesi
    dolayısıyla yeni bir dosya göstericisi elde etmiş oluruz.
-------------------------------------------------------------------------------------------------
-------------------------*/

/*-------------------------------------------------------------------------------------------------
-------------------------
    Dosyadan okuma yapmak için read POSIX fonksiyonu kullanılmaktadır. Pek çok sistemde bu POSIX
fonksiyonu doğrudan işletim sisteminin
    okuma yapan sistem fonksiyonunu (Linux'ta sys_read) çağırmaktadır. read fonksiyonunun
prototipi şöyledir:
```

```
    #include <unistd.h>

    ssize_t read(int fd, void *buf, size_t nbyte);

    Fonksiyonun birinci parametresi okuma işleminin yapılacağı dosya betimleyicisini
belirtmektedir. İşletim sistemi, bu betimleyiciden
    hareketle dosya nesnesine erişmektedir. İkinci parametre bellekteki transfer adresini
belirtmektedir. Üçüncü parametre okunacak
    byte sayısını belirtir.

    Fonksiyon başarı durumunda okuyabildiği byte ile geri döner. read fonksiyonu ile eğer dosya
göstericisinin gösterdiği yerden itibaren dosya sonuna kadar
    mevcut olan byte miktarından daha fazla byte okunmak istenirse, read fonksiyonu okuyabildiği
kadar byte'ı okur ve okuyabildiği byte sayısına
    geri döner. Dosya göstericisi EOF durumunda ise read hiç okuma yapamayacağı için 0 ile geri
dönmektedir. Ancak argümanların
    yanlış girilmesinde ya da IO hatalarında read başarısız olur ve -1 değerine geri döner.
ssize_t <unistd.h> ve <sys/types.h>
    içerisinde işaretli bir tamsayı türü biçiminde typedef edilmek durumunda olan POSIX'e özgü bir
typedef ismidir.

    read fonksiyonu ile dosyadan 0 byte okunmak istendiğinde read fonksiyonu temel bazı
kontrolleri yapar. (Örneğin; dosyanın okuma modunda
    açılmış olup olmadığı kontrol edilir.) Eğer bu kontrollerde bir sorun çıkarsa -1 değerine geri
döner. Eğer bu kontrollerde bir sorun çıkmazsa
    0 değerine geri döner ve herhangi bir okuma işlemi yapmaz.
-------------------------------------------------------------------------------------------------
------------------------*/

#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <unistd.h>

void exit_sys(const char *msg);

int main(void)
{
        int fd;
        char buf[10 + 1];
        ssize_t result;

        if ((fd = open("test.txt", O_RDONLY)) == -1)
                exit_sys("open");

        if ((result = read(fd, buf, 10)) == -1)
                exit_sys("read");

        buf[result] = '\0';

        printf(":%s:\n", buf);

        close(fd);

        return 0;
}

void exit_sys(const char *msg)
{
        perror(msg);

        exit(EXIT_FAILURE);
}

/*-------------------------------------------------------------------------------------------------
-------------------------
    Simdi bir dosyayı (örneğimizde içerisinde yazı olan bir dosyayı) dosya sonuna kadar read
fonksiyonu ile bir döngü içerisinde
```





```
    okuyalım. Bu tür durumlarda klasik yöntem aşağıdaki gibi bir döngü oluşturmaktır:

        while ((result = read(fd, buf, BUFSIZE)) > 0) {
                buf[result] = '\0';
                printf("%s", buf);
        }

        if (result == -1)
                exit_sys("read");

    Bu döngüden IO hatası oluşunca ya da dosya göstericisi dosyanın sonuna geldiğinde
çıkılacaktır.
-------------------------------------------------------------------------------------------------
------------------------*/

#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <unistd.h>

#define BUFSIZE         4096

void exit_sys(const char *msg);

int main(void)
{
        int fd;
        char buf[BUFSIZE + 1];
        ssize_t result;

        if ((fd = open("sample.c", O_RDONLY)) == -1)
                exit_sys("open");

        while ((result = read(fd, buf, BUFSIZE)) > 0) {
                buf[result] = '\0';
                printf("%s", buf);
        }

        if (result == -1)
                exit_sys("read");

        close(fd);

        return 0;
}

void exit_sys(const char *msg)
{
        perror(msg);

        exit(EXIT_FAILURE);
}

/*-------------------------------------------------------------------------------------------------
-------------------------

11. Ders 27/11/2022 - Pazar
-------------------------------------------------------------------------------------------------
------------------------*/

/*-------------------------------------------------------------------------------------------------
-------------------------
    Dosyaya yazma yapmak için write isimli POSIX fonksiyonu kullanılmaktadır. Bu fonksiyon da pek
çok sistemde doğrudan işletim sisteminin
    dosyaya yazma yapan sistem fonksiyonunu (Linux'ta sys_write) çağırmaktadır. Prototipi
şöyledir:

    #include <unistd.h>
```





```
    ssize_t write(int fd, const void *buf, size_t nbyte);

    Fonksiyonun birinci parametresi yazma yapılacak dosyaya ilişkin dosya betimleyicisini
belirtir. İkinci parametre yazılacak
    bilgilerin bulunduğu bellek adresidir. Üçüncü parametre yazılacak byte sayısını belirtir.
write fonksiyonu başarılı olarak yazılan
    byte sayısı ile geri dönmektedir. Normal olarak bu değer üçüncü parametrede belirtilen
yazılmak istenen byte sayısıdır. Ancak çok seyrek
    bazı durumlarda (örneğin diskin dolu olması gibi) write talep edilenden daha az byte'ı
yazabilir. Bu durumda yazabildiği byte sayısı ile geri döner.
    write başarısız olursa -1 değerine geri dönmektedir.

    write fonksiyonu ile dosyaya 0 byte yazılmak istendiğinde gerçek bir yazma yapılmaz. write
fonksiyonu bu durumda yazma konusunda
    gerekli kontrolleri yapar (örneğin dosyanın yazma modunda açılıp açılmadığı gibi). Eğer bu
kontrollerde başarısızlık oluşursa
    write fonksiyonu -1 ile geri döner. Eğer bu kontrollerde başarısızlık oluşmazsa write
fonksiyonu 0 ile geri döner. Ancak yukarıda da
    belirttiğimiz gibi bu durumda gerçek bir yazma yapılmamaktadır. POSIX standartları normal
dosyaların dışında (yani "regular" olmayan dosyaların dışında)
    0 byte yazma işlemini "unspecified" olarak belirtmiştir. Dolayısıyla ileride göreceğimiz boru
gibi dosyalara 0 byte yazıldığında ne olacağı
    o sisteme bağlı bir durumdur.
-------------------------------------------------------------------------------------------------
------------------------*/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <unistd.h>

void exit_sys(const char *msg);

int main(void)
{
        int fd;
        char buf[] = "this is a test";
        ssize_t result;

        if ((fd = open("test.txt", O_WRONLY|O_CREAT|O_TRUNC, S_IRUSR|S_IWUSR|S_IRGRP|S_IROTH)) ==
-1)
                exit_sys("open");

        if (write(fd, buf, strlen(buf)) == -1)
                exit_sys("write");

        close(fd);

        return 0;
}

void exit_sys(const char *msg)
{
        perror(msg);

        exit(EXIT_FAILURE);
}

/*-------------------------------------------------------------------------------------------------
-------------------------
    UNIX/Linux sistemlerinde dosya kopyalama bir döngü içerisinde kaynak dosyadan hedef dosyaya
blok blok okuma yazma işlemi ile
    yapılmaktadır. Ancak bazı UNIX türevi işletim sistemleri dosya kopyalama işlemi için sistem
fonksiyonları da bulundurabilmektedir.
    Örneğin Linux sistemlerinde copy_file_range isimli sistem fonksiyonu doğrudan disk üzerinde
blok kopyalaması yoluyla dosya
```





```
    kopyalamasını hiç user mode işlem yapmadan gerçekleştirebilmektedir. Ancak bu işlemin
taşınabilir yolu yukarıda belirttiğimiz gibi
    kaynaktan hedefe aktarım yapmaktır. Pekiyi bu kopyalama işleminde hangi büyüklükte bir tampon
kullanılmalıdır? Tipik olarak dosya sistemindeki
    blok uzunluğu bunun için tercih edilir. stat, fstat, lstat gibi fonksiyonlar bunu bize
verirler. Blok uzunlukları 512'nin
    katları biçimindedir.

    Aşağıdaki örnekte blok kopyalaması yoluyla dosya kopyalaması yapılmıştır.
-------------------------------------------------------------------------------------------------
------------------------*/

#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <unistd.h>

#define BUFFER_SIZE             4096

void exit_sys(const char *msg);

int main(int argc, char *argv[])
{
        char buf[BUFFER_SIZE];
        int fds, fdd;
        ssize_t result;

        if (argc != 3) {
                fprintf(stderr, "wrong number of arguments!...\n");
                exit(EXIT_FAILURE);
        }

        if ((fds = open(argv[1], O_RDONLY)) == -1)
                exit_sys(argv[1]);

        if ((fdd = open(argv[2], O_WRONLY|O_CREAT|O_TRUNC, S_IRUSR|S_IWUSR|S_IRGRP|S_IROTH)) ==
-1)
                exit_sys(argv[2]);

        while ((result = read(fds, buf, BUFFER_SIZE)) > 0)
                if (write(fdd, buf, result) != result) {
                        fprintf(stderr, "cannot write file!...\n");
                        exit(EXIT_FAILURE);
                }

        if (result == -1)
                exit_sys("read");

        close(fds);
        close(fdd);

        return 0;
}

void exit_sys(const char *msg)
{
        perror(msg);

        exit(EXIT_FAILURE);
}

/*-------------------------------------------------------------------------------------------------
-------------------------
    write çok çok seyrek de olsa başarılı olduğu halde talep edilen miktar kadar hedef dosyaya
yazamayabilir. Örneğin diskin dolu
    olması durumunda ya da bir sinyal oluşması durumunda write talep edilen miktar kadar yazma
yapamayabilir. Bu tür durumları diğer
    durumlardan ayırmak için ayrı bir kontrol yapmak gerekebilir.
```

```
-------------------------------------------------------------------------------------------------------
------------------------*/

#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <unistd.h>

#define BUFFER_SIZE             4096

void exit_sys(const char *msg);

int main(int argc, char *argv[])
{
        char buf[BUFFER_SIZE];
        int fds, fdd;
        ssize_t size, result;

        if (argc != 3) {
                fprintf(stderr, "wrong number of arguments!...\n");
                exit(EXIT_FAILURE);
        }

        if ((fds = open(argv[1], O_RDONLY)) == -1)
                exit_sys(argv[1]);

        if ((fdd = open(argv[2], O_WRONLY|O_CREAT|O_TRUNC, S_IRUSR|S_IWUSR|S_IRGRP|S_IROTH)) ==
-1)
                exit_sys(argv[2]);

        while ((result = read(fds, buf, BUFFER_SIZE)) > 0) {
                if ((size = write(fdd, buf, result)) == -1)
                        exit_sys("write");
                if (size != result) {
                        fprintf(stderr, "cannot write file!...\n");
                        exit(EXIT_FAILURE);
                }
        }

        if (result == -1)
                exit_sys("read");

        close(fds);
        close(fdd);

        return 0;
}

void exit_sys(const char *msg)
{
        perror(msg);

        exit(EXIT_FAILURE);
}

/*-------------------------------------------------------------------------------------------------
-------------------------
    read ve write POSIX fonksiyonları yukarıda da belirttiğimiz gibi dosya göstericisinin
gösterdiği yerden itibaren okuma ve
    yazma işlemlerini yapmaktadır. Bu fonksiyonlar dosya göstericisinin konumunu okunan ya da
yazılan miktar kadar ilerletmektedir.
    İşte read ve write fonksiyonlarının pread ve pwrite biçiminde bir versiyonu da bulunmaktadır.
pread ve pwrite fonksiyonları,
    işlemlerini dosya göstericisinin gösterdiği yerden itibaren değil, parametreleriyle belirtilen
offset'ten yapmaktadır. Bu
    fonksiyonlar dosya göstericisinin konumunu değiştirmezler. Uygulamada pread ve pwrite
fonksiyonları seyrek kullanılmaktadır.
    Örneğin dosyanın farklı yerlerinden sürekli okuma/yazma yapıldığı durumlarda bu fonksiyonlar
```





```
kullanım kolaylığı sağlayabilmektedir.
    Fonksiyonların prototipleri şöyledir:

    #include <unistd.h>

    ssize_t pread(int fildes, void *buf, size_t nbyte, off_t offset);
    ssize_t pwrite(int fildes, const void *buf, size_t nbyte, off_t offset);

    pread ve pwrite fonksiyonlarının read ve write fonksiyonlarından tek farkı offset
parametresidir. Bu fonksiyonlar dosya
    göstericisinin gösterdiği yerden değil, son parametreleriyle belirtilen yerden okuma ve yazma
işlemini yaparlar. Fonksiyonların
    dosya göstericisinin konumunu değiştirmediğine dikkat ediniz.

    Dosyalara okuma yazma işlemi genellikle ardışıl bir biçimde yapıldığı için bu fonksiyonlar
seyrek kullanılmaktadır. Ancak
    örneğin veritabanı işlemlerinde yukarıda da belirttiğimiz gibi dosyanın farklı offset'lerinden
sıkça okuma ve yazmanın
    yapıldığı durumlarda bu fonksiyonlar tercih edilebilmektedir.

    pread ve pwrite fonksiyonları da doğrudan ilgili sistem fonksiyonlarını çağırmaktadır. (Linux
sistemlerinde sys_pread ve
    sys_pwrite). Tabii bu işlemler önce dosya gösterisicini saklayıp, sonra konumlandırıp, sonra
read/write işlemlerini yapıp,
    sonra da yeniden dosya göstericisini eski konumuna yerleştirmekle yapılabilir. Ancak pread ve
pwrite işlemlerini yapan
    sistem fonksiyonları bu biçimde değil, daha doğrudan aynı işlemi yapmaktadır.
-------------------------------------------------------------------------------------------------------
------------------------*/

/*-------------------------------------------------------------------------------------------------
-------------------------
    Dosya göstericisi dosya açıldığında 0'ıncı offset'tedir. Ancak okuma ve yazma yapıldığında
okunan ya da yazılan miktar kadar otomatik ilerletilmektedir.
    Dosya göstericisini belli bir konuma almak için lseek isimli POSIX fonksiyonu
kullanılmaktadır. Bu fonksiyon da pek çok işletim
    sisteminde doğrudan dosyayı konumlandıran sistem fonksiyonunu (Linux'ta sys_lseek)
çağırmaktadır. lseek fonksiyonun genel kullanımı
    fseek standart C fonksiyonuna çok benzemektedir. Fonksiyonun prototipi şöyledir:

    #include <unistd.h>

    off_t lseek(int fd, off_t offset, int whence);

    Fonksiyonun birinci parametresi dosya göstericisi konumlandırılacak dosyayaya ilişkin dosya
betimleyicisini belirtir. Dosya göstericisi
    dosya nesnesinin (Linux'ta struct file) içerisinde tutulmaktadır. İkinci parametre
konumlandırma offset'ini belirtir. off_t
    <unistd.h> ve <sys/types.h> içerisinde işaretli bir tamsayı türü biçiminde typedef edilmiş
olan bir tür ismidir. Üçüncü parametre konumlandırma
    orijinini belirtmektedir. Bu üçüncü parametre 0, 1 ya da 2 olarak girilebilir. Tabii sayısal
değer girmek yerine yine SEEK_SET (0), SEEK_CUR (1)
    ve SEEK_END (2) sembolik sabitlerini girebiliriz. Bu sembolik sabitler <unistd.h> ve <stdio.h>
içerisinde de bildirilmiştir.
    Fonksiyon başarı durumunda konumlandırılan offset'e, başarısızlık durumunda -1 değerine geri
dönmektedir.

    SEEK_SET konumlandırmanın dosyanın başından itibaren yapılacağını, SEEK_CUR konumlandırmanın o
anda dosya göstericisinin gösterdiği
    yerden itibaren yapılacağını ve SEEK_END de konumlandırmanın EOF durumundan itibaren
yapılacağını belirtmektedir. En normalk durum
    SEEK_SET orijininde ikinci parametrenin >= 0, SEEK_END orijininde <= 0 biçiminde girilmesidir.
SEEK_CUR orijininde ikinci
    parametre pozitif ya da negatif girilebilir. Örneğin dosya göstericisini EOF durumuna şöyle
konumlandırabiliriz:

    lseek(fd, 0, SEEK_END);

    Dosya sistemine de bağlı olarak UNIX/Linux sistemleri dosya göstericisini EOF'un ötesine
```





```
konumlandırmaya izin verebilmektedir.
    Bu özel bir durumdur. Bu tür durumlarda dosyaya yazma yapıldığında "dosya delikleri (file
holes)" oluşmaktadır. Dosya delikleri konusu ileride
    ele alınacaktır.

    Aslında dosya açarken O_APPEND modu atomik bir biçimde her write işleminden önce dosya
göstericisini EOF durumuna çekmektedir. Bu nedenle
    her yazılan dosyanın sonuna eklenmektedir.

    Aşağıdaki örnekte "test.txt" O_WRONLY modunda açılmış ve dosya göstericisi EOF durumuna
çekilerek dosyaya ekleme yapılmıştır.
-------------------------------------------------------------------------------------------------------
------------------------*/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <unistd.h>

void exit_sys(const char *msg);

int main(void)
{
        int fd;
        char buf[] = "\nthis is a test";

        if ((fd = open("test.txt", O_WRONLY)) == -1)
                exit_sys("open");

        lseek(fd, 0, SEEK_END);

        if (write(fd, buf, strlen(buf)) == -1)
                exit_sys("write");

        close(fd);

        return 0;
}

void exit_sys(const char *msg)
{
        perror(msg);

        exit(EXIT_FAILURE);
}

/*-------------------------------------------------------------------------------------------------
-------------------------
    Bir C/C++ programcısı olarak UNIX/Linux sistemlerinde dosya işlemleri yapmak için üç seçenek
söz konusu olabilir:

    1) C'nin ya da C++'ın standart dosya fonksiyonlarını kullanmak
    2) POSIX dosya fonksiyonlarını kullanmak
    3) Sistem fonksiyonlarını kullanmak

    Burada en taşınabilir olan standart C/C++ fonksiyonlarıdır. Dolayısıyla ilk tercih bunlar
olmalıdır. Ancak C ve C++'ın standart dosya fonksiyonları
    spesifik bir sistemin gereksinimini karşılayacak biçimde yazılmamıştır. Bunedenle bazen
doğrudan POSIX fonksiyonlarının kullanılması
    gerekebilmektedir. Genellikle dosya işlemleri yapan sistem fonksiyonlarının kullanılması hiç
gerekmez. Çünkü Linux'ta olduğu gibi
    pek çok UNIX türevi sistemde yukarıda gördüğümüz POSIX fonksiyonları zaten doğrudan sistem
fonksiyonlarını çağırmaktadır.
-------------------------------------------------------------------------------------------------------
------------------------*/

/*-------------------------------------------------------------------------------------------------
```





```
-------------------------
    POSIX standartlarına göre dosyaya yapılan read ve write işlemleri sistem genelinde atomiktir.
Yani örneğin iki program
    aynı anda aynı dosyanın aynı yerine yazma yapsa bile iç içe geçme oluşmaz. Önce birisi yazar
daha sonra diğeri yazar.
    Tabii hangi prosesin önce yazacağını bilemeyiz. Ancak burada önemli lan nokta iç içe geçmenin
olmamasıdır. Benzer biçimde
    bir read ile bir dısyanın bir yerinden n byte okumak istediğimizde başka bir proses aynı
dosyanın aynı yerine yazma yaptığında
    biz ya o prosesin yazdıklarını okuruz ya da onun yazmadan önceki dosya değerlerini okuruz.
Yarısı eski yarısı yeni bir bilgi okumayız.
    Ancak işletim sistemi farklı read ve write çağrılarını bu anlamda senktronize etmemektedir.
Yani örneğin biz bir dosyanın belli bir yerine
    iki farklı write fonksiyonu ile ardışık şeyler yazdığımızı düşünelim. Birinci write işleminden
sonra başka bir proses artık orayı
    değiştirebilir. Dlayısıyla bu anlamda bir iç içe girme durumu oluşabilir. Veritabanı
programlarında bu tür durumlarla sık karşılaşılmaktadır.
    Örneğin veri tabanı programı bir kaydı "data" dosyasına yazıp ona ilişkin indeksleri "index"
dosyasına yazıyor olabilir. Bu durumda
    iki write işlemi söz konusudur. Data dosyasına bilgiler yazıldktan sonra henüz indeks
dosyasına bilgi yazılmadan başka bir proses bu iki işlemi hızlı
    davranarak yaparsa data ve indeks bütünlüğü bozulur. İşletim sisteminin burada bir sorumluluğu
yoktur. Bu tarz işlemlerde senkroznizasyon
    programcılar tarafından sağlanmak zorundadır. Bu tür senktronizasyonlar senkronizasyon
nesneleriyle (semaphore gibi, mutex gibi) dosya bütününde
    yapılabilir. Ancak tüm dosyaya erişimin engellenmesi iyi bir teknik değildir. İşte bu tür
durumlar için işletim sistemleri çekirdeğe
    entegre edilmiş olan "dosya kilitleme (file locking)" mekanizması bulundurmaktadır. Dosya
kilitleme tüm dosyayı değil dosyanın belli offset'lerine erişimi
    engelleme amacındadır.
-------------------------------------------------------------------------------------------------------
------------------------*/

/*-------------------------------------------------------------------------------------------------
-------------------------
    Biz open fonksiyonu ile bir dosya yaratırken yaratacağımız dosyaya verdiğimiz erişim hakları
dosyaya tam olarak yansıtılmayabilir.
    Yani örneğin biz gruba "w" hakkı vermek istesek bile bunu sağlayamayabiliriz. Çünkü belirtilen
erişim değerlerini maskeleyen (yani ortadan kaldıran)
    bir mekanizma vardır. Buna prosin umask değeri denilmektedir. Prosesin umask değeri mode_t
türü ile ifade edilir; sahiplik, grupluk ve diğerlik
    bilgilerini içerir. Bu bilgiler aslında maskeleneck değerleri belirtmektedir. Örneğin prosesin
umask değerinin S_IWGRP|S_IWOTH olduğunu varsayalım.
    Bu umask değeri "biz open fonksiyonu ile bir dosyayı yaratırken grup için ve diğerleri için
"w" hakkı versek bile bu hak dosyaya
    yansıtılmayacak" anlamına gelmektedir. Eğer prosesin umask değeri 0 ise bu durumda
maskelenecek bir şey yoktur dolayısıyla verilen
    hakların hepsi dosyaya yansıtılır. Prosesin umask değerinin umask olduğunu varsayalım. Dosyaya
vermek istediğimiz erişim haklarının da
    mode olduğunu varsayalım. (Yani mode S_IXXX gibi tek biti 1 olan değerlerin bit düzeyinde
OR'lanması ile oluşturumuş değr olsun.)
    Bu durumda dosyaya yansıtılacak erişim hakları mode & ~umask olacaktır. Yani prosesin umask
değerindeki bitler maskelenecek erişim
    haklarını belirtmektedir.

    Prosesin başlangıçtaki umask değeri üst prosesten aktarılmaktadır. Örneğin biz kabuktan
program çalıştırırken çalıştırdığımız programın umask
    değeri kabuğun (örneğin bash prosesinin) umask değeri olarak bizim prosesimize geçirilecektir.
Kabuğun umask değeri "umask" isimli
    komutla elde edilebilir. Bu değer genellikle "0022" ya da "0002" gibi bir değerde olacaktır.
Buradaki basamaklar octal sayı (sekizlik sistemde sayı)
    belirtmektedir. Bir octal digit 3 bitle açılmaktadır. Dolayısıyla bu bitler maskelenecek
erişim haklarının durumunu belirtir:

    ? owner group other

    En yüksek anlamlı octal digit şimdiye kadar görmediğimiz başka haklarla ilgilidir. Bu haklara
"set user id", "set group id" ve "sticky"
    hakları denilmektedir. Ancak diğer 3 octal digit sırasıyla owner, group ve other maskeleme
```

bitlerini belirtmektedir.

    Kabuk üzerinde umask komutuyla aynı zamanda kabuğun umask değeri de değiştirilebilir. Bu durumda yine değiştirme değerleri octal digitler
    biçiminde verilmelidir. Örneğin:

    umask 022

    Burada en yüksek anlamlı octal verilmediğine göre 0 kabul edilir. O halde burada belirtilern umask değeri grup için ve diğerleri için
    "w" hakkını maskeleyecektir. (Zaten pek çok kabulta umask değerin default durumu böyledir.) Bazen programcı umask değerini tamamen sıfırlamak da isteyebilir.
    Bu işlem şöyle yapılabilir:

    umask 0

    Burada yüksek anlamlı üç octal digit de 0 kabul edilmektedir. Bu durumda artık çalıştırdığımız programda open fonksiyonun tüm erişim hakları
    dosyalara yansıtılacaktır.

    Bir proses başlangıçta umask değerini üst prosesten almaktadır. Ancak proses istediği zaman umask isimli POSIX fonksiyonu ile kendi umask
    değerini değiştirebilmektedir. umask fonksiyonunun prototipi şöyledir:

    #include <sys/stat.h>

    mode_t umask(mode_t cmask);

    Fonksiyon belirtilen değerle prosesin umask değerini set eder ve prosesin eski umask değerine geri döner. Fonksiyon başarısız olamaz.

    umask fonksiyonu ile kendi prosesimizin umask değerini almak için onu değişltirmemiz gerekir. Bu durumda bu işlem aşağıdaki bir kodla
    yapılabilir:

    mode_t mode;

    mode = umask(0);
    umask(mode);

    Tabii programcı umask fonksiyonuna octal digitler girebilir. Ancak sistemlerde bu ocatl digitler tam olarak S_IXXX sembolik sabitlerinin
    değerlerine karşı gelmeyebilir. Ancak daha önceden de bahsedildiği gibi POSIX standartlarında belli bir zamandan sonra bu S_IXXX sembolik sabitlerinin
    değerleri açıkça belirtilmiştir. Örneğin:

    umask(00022);               /* Eskiden bu biçimde belirleme taşınabilir değildi, eski sistemlerde dikkat edilmesi gerekir */
    umask(S_IWGRP|S_IWOTH);     /* Bu biçimde belirleme daha okunabilirdir. */

    Aşağıdaki örnekte prosesin umask değeri önce sıfırlanmış, sonra bir dosya yaratılmıştır. open fonksiyonunda verilen erişim hakları
    artık dosyaya tamamen yansıtılacaktır.
----------*/

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>

void exit_sys(const char *msg);

int main(void)
{
        int fd;

        umask(0);
```





```
        if ((fd = open("x.dat", O_WRONLY|O_CREAT, S_IRWXU|S_IRWXG|S_IRWXO)) == -1)
                exit_sys("open");

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

        close(fd);

        return 0;
}

void exit_sys(const char *msg)
{
        perror(msg);

        exit(EXIT_FAILURE);
}
```

/*----------

12. Ders 03/12/2022 - Cumartesi
----------*/

/*----------
    Aşağıdaki daha önce yapmış olduğumuz shell programına umask komutunu ekliyoruz.
----------*/

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>
#include <sys/stat.h>
#include <limits.h>
#include <unistd.h>

#define MAX_CMD_LINE            4096
#define MAX_CMD_PARAMS          128

typedef struct tagCMD {
        char *name;
        void (*proc)(void);
} CMD;

void parse_cmd_line(char *cmdline);

void dir_proc(void);
void clear_proc(void);
void pwd_proc(void);
void cd_proc(void);
void umask_proc(void);

int check_umask_arg(const char *str);

void exit_sys(const char *msg);

char *g_params[MAX_CMD_PARAMS];
int g_nparams;
char g_cwd[PATH_MAX];

CMD g_cmds[] = {
        {"dir", dir_proc},
        {"clear", clear_proc},
        {"pwd", pwd_proc},
        {"cd", cd_proc},
        {"umask", umask_proc},
```





```
        {NULL, NULL}
};

int main(void)
{
        char cmdline[MAX_CMD_LINE];
        char *str;
        int i;

        if (getcwd(g_cwd, PATH_MAX) == NULL)
                exit_sys("fatal error (getcwd)");

        for (;;) {
                printf("CSD:%s>", g_cwd);
                if (fgets(cmdline, MAX_CMD_LINE, stdin) == NULL)
                        continue;
                if ((str = strchr(cmdline, '\n')) != NULL)
                        *str = '\0';
                parse_cmd_line(cmdline);
                if (g_nparams == 0)
                        continue;
                if (!strcmp(g_params[0], "exit"))
                        break;
                for (i = 0; g_cmds[i].name != NULL; ++i)
                        if (!strcmp(g_params[0], g_cmds[i].name)) {
                                g_cmds[i].proc();
                                break;
                        }
                if (g_cmds[i].name == NULL)
                        printf("bad command: %s\n", g_params[0]);
        }

        return 0;
}

void parse_cmd_line(char *cmdline)
{
        char *str;

        g_nparams = 0;
        for (str = strtok(cmdline, " \t"); str != NULL; str = strtok(NULL, " \t"))
                g_params[g_nparams++] = str;
}

void dir_proc(void)
{
        printf("dir command executing...\n");
}

void clear_proc(void)
{
        system("clear");
}

void pwd_proc(void)
{
        printf("%s\n", g_cwd);
}

void cd_proc(void)
{
        char *dir;

        if (g_nparams > 2) {
                printf("too many arguments!\n");
                return;
        }
        if (g_nparams == 1) {
                if ((dir = getenv("HOME")) == NULL)
```





```
                        exit_sys("fatal error (getenv");
        }
        else
                dir = g_params[1];

        if (chdir(dir) == -1) {
                printf("%s\n", strerror(errno));
                return;
        }

        if (getcwd(g_cwd, PATH_MAX) == NULL)
                exit_sys("fatal error (getcwd)");
}

void umask_proc(void)
{
        mode_t mode;
        int argval;

        if (g_nparams > 2) {
                printf("too many arguments in umask command!...\n");
                return;
        }

        if (g_nparams == 1) {
                mode = umask(0);
                umask(mode);

                printf("%04o\n", (int)mode);

                return;
        }

        if (!check_umask_arg(g_params[1])) {
                printf("%s octal number out of range!...\n", g_params[1]);
                return;
        }

        sscanf(g_params[1], "%o", &argval);
        umask(argval);
}

int check_umask_arg(const char *str)
{
        if (strlen(str) > 4)
                return 0;

        for (int i = 0; str[i] != '\0'; ++i)
                if (str[i] < '0' || str[i] > '7')
                        return 0;

        return 1;
}

void exit_sys(const char *msg)
{
        perror(msg);

        exit(EXIT_FAILURE);
}
```

/*----------
    UNIX/Linux sistemlerinde open, close, read, wri,te ve lseek fonksiyonlarının yanı sıra pek çok yardımcı dosya fonksiyonları da vardır.
    Bu yardımcı dosya fonksiyonları dosyalar üzerinde bazı önemli işlemleri yapmaktadır. Bu bölümde bu fonlksiyonların önemli olanlarını
    tanıtacağız.
----------

```
--------------------------*/

/*-------------------------------------------------------------------------------------------------
-------------------------
    Bir dosyaya ilişkin bilgileri elde etmek için stat, lstat ve fstat isimli üç fonksiyon
kullanılmaktadır. Bu fonksiyonlar
    aslında aynı şeyi yaparlar. Fakat parametrik yapı bakımından ve semantik bakımdan bunların
arasında küçük farklılıklar vardır.
    Fonksiyonların prototipleri şöyledir:

    #include <sys/stat.h>

    int stat(const char *path, struct stat *buf);
    int fstat(int fd, struct stat *buf);
    int lstat(const char *path, struct stat *buf);

    stat fonksiyonları bir dosyanın bilgilerini elde etmek amacıyla kullanılmaktadır. Örneğin
dosyanın erişim hakları, kullanıcı ve grup id'leri,
    dosyanın uzunluğu, dosyanın tarih zaman bilgileri bu stat fonksiyonlarıyla elde edilmektedir.
ls komutu -l seneği ile kullanıldığında
    aslında dosya bilgilerini bu stat fonksiyonuyla elde edip ekrana yazdırmaktadır.

    stat fonksiyonlarından en çok kullanılanı stat fonksiyonudur:

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

    Fonksiyonun birinci parametresi bilgisi elde edilecek dosyanın yol ifadesini belirtmektedir.
İkinci parametresi dosya
    bilgilerinin yerleştirileceği struct stat isimli bir yapı nesnesinin adresini almaktadır. stat
isimli yapı <sys/stat.h>
    içerisinde bildirilmiştir. Fonksiyon başarı durumunda 0, başarısızlık durumunda -1 değerine
geri döner.

    struct stat yapısının elemanları şöyledir:

    struct stat {
        dev_t     st_dev;         /* ID of device containing file */
        ino_t     st_ino;         /* Inode number */
        mode_t    st_mode;        /* File type and mode */
        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;     /* Block size for filesystem I/O */
        blkcnt_t  st_blocks;      /* Number of 512B blocks allocated */

        /* Since Linux 2.6, the kernel supports nanosecond
            precision for the following timestamp fields.
            For the details before Linux 2.6, see NOTES. */

        struct timespec st_atim;  /* Time of last access */
        struct timespec st_mtim;  /* Time of last modification */
        struct timespec st_ctim;  /* Time of last status change */

    #define st_atime st_atim.tv_sec      /* Backward compatibility */
    #define st_mtime st_mtim.tv_sec
    #define st_ctime st_ctim.tv_sec
    };

    Yapının st_dev elemanı dosyanın içinde bulunduğu aygıtın aygıt numarasını belirtir. Genellikle
programcılar bu bilgiye gereksinim duymazlar. dev_t
    türü herhangi bir tamsayı türü biçiminde typedef edilebilecek bir tür ismidir.

    stat fonksiyonları dosya bilgilerini aslında diskten elde etmektedir. UNIX/Linux sistemlerinde
kullanılan dosya sistemlerinin disk organşizasyonunda
    i-node tablosu denilen bir tablo vardır. i-node tablosu i-node elemanlarından oluşmaktadır.
Her i-node elemanı bir dosyaya ilişkin bilgileri tutar.
    İşte bir dosyanın bilgilerinin hangi i-node elemanında olduğu stat yapısının st_ino elemanına
```





```
yerleştirilmektedir. Dosyanın i-node elemanı i-node
    tablosunda bir indeks belirtmektedir. Dosyaların i-node numarları ls komutunda -i seçeneği ile
gösterilmektedir. ino_t türü işaretsiz olmak koşuluyla
    herhangi bir tamsayı türü biçiminde typedef edilebilmektedir.

    Yapının st_mode elemanı dosyanın erişim bilgilerini ve türünü içermektedir. Yine bu elemanın
içerisindeki değerler bitler biçiminde oluşturulmuştur.
    1 olan bitler ilgili özelliğin olduğunu belirtmektedir. Belli bir erişim hakkının (örneğin
S_IWGRP gibi) olup olmadığını anlamak için programcı ilgili
    bitin set edilip edilmediğine st_mode & S_IXX işlemi ile bakmalıdır. Dosyanın türü de yine
aynı elemanın içerisine bitsel olarak kodlanmıştır. Ancak hangi bitlerin
    hangi türleri belirttiği POSIX standartlarında belirtilmemiştir. Bu durum sistemden sisteme
değişebilmektedir. (Anımsanacağı gibi eskiden aynı durum S_IXXX
    sembolik sabitleri için de geçerliydi. Ancak daha sonra bu sembolik sabitlerinyonları sayısal
değerleri yani bit pozisyonları POSIX standartlarında belirlendi.)
    Dosyanın türünü anlamak için iki yöntem bulunmaktadır. Birincisi <sys/stat.h> içerisindeki
S_ISXXX biçimindeki makroları kullanmaktır. Bu makrolar
    eğer dosya ilgili türdense sıfır dışı bir değer ilgili türden değilse sıfır değerini verir.
Makrolar şunlardır:

    S_ISBLK(m)      Blok aygıt sürücü dosyası mı? (ls -l'de 'b' dosya türü)
    S_ISCHR(m)      Karakter aygıt sürücü dosyası mı? (ls -l'de 'c' dosya türü)
    S_ISDIR(m)      Dizin dosyası mı? (ls -l'de 'd' dosya türü)
    S_ISFIFO(m)     Boru dosyası mı? (ls -l'de 'p' dosya türü)
    S_ISREG(m)      Sıradan bir disk dosyası mı? (ls -l'de '-' dosya türü)
    S_ISLNK(m)      Sembolik bağlantı dosyası mı? (ls -l'de 'l' dosya türü)
    S_ISSOCK(m)     Soket dosyası mı? (ls -l'de 's' dosya türü)

    Dosya türünün tespiti için ikinci yöntem st_mode içerisindeki dosya tür bitlerinin S_IFMT
sembolik sabiti ile bit AND işlemi ile
    elde edilip aşağıdaki sembolik sabitlerle karşılaştırılmasıdır.

    S_IFBLK         Blok aygır dosyası
    S_IFCHR         Karakter aygıt dosyası
    S_IFIFO         Boru dosyası
    S_IFREG         Sıradan disk dosyası
    S_IFDIR         Dizin dosyası
    S_IFLNK         Sembolik bağlantı dosyası
    S_IFSOCK        Soket dosyası

    st_mode değeri S_IFMT değeri ile bir AND işlemine sokulduktan sonra bu sembolik sabitlerle
karşılaştırılmalıdır. Bu sembolik sabitlerin
    tek biti 1 değildir. Yani karşılaştırma (mode & S_IFMT) == S_IFXXX biçiminde yapılmalıdır.

    Yapının st_nlink elemanı dosyanın "hard link" sayısını belirtmektedir. Hard link kavramı
ileride ele alınacaktır. nlink_t türü
    bir tamsayı türü olmak koşuluyla herhangi bir tür olarak typedef edilebilmektedir.

    Yapının st_uid elemanı dosyanın kullanıcı id'sini belirtmektedir. Tabii ls -l komutu bu id'yi
sayı olarak değil /etc/passwd dosyasına başvurarak
    isim biçiminde yazdırmaktadır. uid_t türü herhangi bir tamsayı türü olarak typedef
edilebilmektedir.

    Yapının st_gid elemanı dosyanın grup id'sini belirtmektedir. Tabii ls -l komutu bu id'yi sayı
olarak değil /etc/group dosyasına başvurarak
    isim biçiminde yazdırmaktadır. ugid_t türü herhangi bir tamsayı türü olarak typedef
edilebilmektedir.

    Yapının st_rdev elemanı eğer dosya bir aygıt dosyası ise temsil ettiği atgıtın numarasını bize
vermektedir. Bu eleman da dev_t türündedir.

    Yapının st_size elemanı dosyanın uzunluğunu bize vermektedir. off_t türü daha önceden de
belittiğimiz gibi işaretli bir tamsayı
    türü biçiminde typedef edilmek zorundadır.

    Yapının st_blksize elemanı dosyanın içinde bulunduğu dosya sisteminin kullandığı blok
uzunluğunu belirtmektedir. Dosyaların parçaları
    diskte "block" denilen ardışıl byte topluluklarında tutulmaktadır. İşte bir bloğun kaç byte
olduğu bilgisi bu elemanla belirtilmektedir.
```





```
    Aynı zamanda programcılar dosya kopyalama gibi işlemlerde bu büyüklüğü tampon büyüklüğü
(buffer size) olarak da kullanmaktadır.
    blksize_t işaretli bir tamsayı türüolarak typedef edilmek zorundadır.

    Yapının st_blocks elemanı dosyanın diskte kapladığı blok sayısını belirtmektedir. (Ancak
buradaki sayı 512 byte'lık blokların sayısıdır.
    Yani dosya sistemindeki dosyanın parçaları olan bloklara ilişkin sayı değildir.) blkcnt_t
işaretli bir tamsayı türü olarak typedef
    edilmek zorundadır.

    UNIX/Linux sistemlerinde kullanılan i-node tabanlı dosya sistemleri bir dosya için üç zaman
bilgisi tutmaktadır:

    1) Dosyanın son değiştirilme zamanı
    2) Dosyanın son okunma zamanı
    3) Dosyanın i-node bilgilerinin son değiştirilme zamanı

    POSIX standartları hangi POSIX fonksiyonlarının hangi zamanları dosya için güncellediğini
belirtmektedir. Örneğin read fonksiyonu
    dosyanın son okuma zamanını, write fonksiyonu son yazma ve i-node bilgilerinin değiştirilme
zamanını güncellemektedir.

    stat yapısının bu zamanı tutan elemanları eski POSIX standartlarında time_t türündendi ve
isimleri st_atime, st_mtime ve st_ctime
    biçimindeydi. Bu elemanlar epoch olan 01/01/1970'ten geçen saniye sayısını tutuyordu (C
Programlama Dili'nde epoch'un 01/01/1970 olması
    zorunlu değildir. Ancak POSIX standartlarında bu zorunludur.) Ancak daha sonra POSIX
standartlarında bu zaman bilgisini nanosaniye çözünürlüğe
    çektiler. Dolayısıyla zamansal bilgiler time_t türü ile değil timespec bir yapıyla
belirtilmeye başlandı. Yapı elemanlarının isimleri de
    st_atime, st_mtim ve st_ctim olarak değiştirildi. timespec yapısı geçmişe doğru uyumu
koruyabilmek için aşağıdaki gibi bildirilmiştir:

    struct timespec {
        time_t  tv_sec;
        long    tv_nsec;
    };

    Yapının tv_sec elemanı yine 01/01/1970'ten geçen saniye saniye sayısını tv_nsec elemanı ise o
saniyeden sonraki nano saniye sayısını tutmaktadır.
    Sistemlerin çoğu POSIX standartlarında bu konuda değişiklik yapılmış olsa da eski doğru uyumu
şöyle korumuştur:

    struct stat {
        ...
        struct timespec st_atim;        /* Time of last access */
        struct timespec st_mtim;        /* Time of last modification */
        struct timespec st_ctim;        /* Time of last status change */

    #define st_atime st_atim.tv_sec             /* Backward compatibility */
    #define st_mtime st_mtim.tv_sec
    #define st_ctime st_ctim.tv_sec
    };

    Bu durumda programcı sisteminin yeni POSIX standartlarını destekleyip desteklemediğine bakmalı
ve duruma göre yapının eski ya da yeni
    elemanlarını kullanmalıdır.
-------------------------------------------------------------------------------------------------
--------------------------*/

/*-------------------------------------------------------------------------------------------------
-------------------------

13. Ders 04/12/2022 - Pazar
-------------------------------------------------------------------------------------------------
--------------------------*/

/*-------------------------------------------------------------------------------------------------
-------------------------
```





```
    Aşağıda dosya bilgilerini stat fonksiyonu ile alıp yazdıran bir örnek verilmiştir.
-------------------------------------------------------------------------------------------------
--------------------------*/

#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <sys/stat.h>

void exit_sys(const char *msg);
void disp_mode(mode_t mode);

int main(int argc, char *argv[])
{
        struct stat finfo;
        struct tm *pt;

        if (argc == 1) {
                fprintf(stderr, "file(s) must be specified!\n");
                exit(EXIT_FAILURE);
        }

        for (int i = 1; i < argc; ++i) {
                if (stat(argv[i], &finfo) == -1)
                        exit_sys("stat");

                printf("i-node no: %llu\n", (unsigned long long)finfo.st_ino);
                printf("file mode: ");
                disp_mode(finfo.st_mode);
                printf("number of hard links: %llu\n", (unsigned long long)finfo.st_nlink);
                printf("user id: %llu\n", (unsigned long long)finfo.st_uid);
                printf("group id: %llu\n", (unsigned long long)finfo.st_gid);
                printf("file size: %lld\n", (long long)finfo.st_size);
                printf("file block size: %lld\n", (long long)finfo.st_blksize);
                printf("number of blocks: %lld\n", (long long)finfo.st_blocks);

                pt = localtime(&finfo.st_mtim.tv_sec);
                printf("last modification: %02d/%02d/%04d %02d:%02d:%02d\n", pt->tm_mday, pt-
>tm_mon + 1, pt->tm_year + 1900,
                                        pt->tm_hour, pt->tm_min, pt->tm_sec);
                pt = localtime(&finfo.st_atim.tv_sec);
                printf("last access (read): %02d/%02d/%04d %02d:%02d:%02d\n", pt->tm_mday, pt-
>tm_mon + 1, pt->tm_year + 1900,
                                        pt->tm_hour, pt->tm_min, pt->tm_sec);
                pt = localtime(&finfo.st_ctim.tv_sec);
                printf("last i-node changed: %02d/%02d/%04d %02d:%02d:%02d\n", pt->tm_mday, pt-
>tm_mon + 1, pt->tm_year + 1900,
                                        pt->tm_hour, pt->tm_min, pt->tm_sec);

                if (argc > 2)
                        printf("-----------------\n");
        }

        return 0;
}

void exit_sys(const char *msg)
{
        perror(msg);

        exit(EXIT_FAILURE);
}

void disp_mode(mode_t mode)
{
        static mode_t modes[] = {S_IRUSR, S_IWUSR, S_IXUSR, S_IRGRP, S_IWGRP, S_IXGRP, S_IROTH,
S_IWOTH, S_IXOTH};
        static mode_t ftypes[] = {S_IFBLK, S_IFCHR, S_IFIFO, S_IFREG, S_IFDIR, S_IFLNK, S_IFSOCK};
```

```
        for (int i = 0; i < 7; ++i)
                if ((mode & S_IFMT) == ftypes[i]) {
                        putchar("bcp-dls"[i]);
                        break;
                }
        /*

        alternatifi

        if (S_ISBLK(mode))
                putchar('b');
        else if (S_ISCHR(mode))
                putchar('c');
        else if (S_ISDIR(mode))
                putchar('d');
        else if (S_ISFIFO(mode))
                putchar('p');
        else if (S_ISREG(mode))
                putchar('-')
        else if (S_ISLNK(mode))
                putchar('l');
        else if (S_ISSOCK(mode))
                putchar('s');
        else
                putchar('?');

        */

        for (int i = 0; i < 9; ++i)
                putchar(mode & modes[i] ? "rwx"[i % 3] : '-');
        putchar('\n');
}

/*-------------------------------------------------------------------------------------------------
-------------------------
    Aşağıdaki örnekte get_ls isimli fonksiyon bizden stat yapısını ve dosyanın ismini alarak char
türden static bir dizinin içerisine
    dosya bilgilerini ls -l formatında kodlamaktadır. Ancak biz henüz kullanıcı id'sini ve grup
id'sini /etc/passwd ve /etc/group
    dosyalarına başvurarak isimlere dönüştürmedik. Bu nedenle bu örnekte dosyaların kullanıcı ve
grup id'leri yazı olarak değil
    sayı olarak kodlanmıştır.
-------------------------------------------------------------------------------------------------
-------------------------*/

#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <sys/stat.h>

#define LS_BUFSIZE              4096

void exit_sys(const char *msg);
char *get_ls(struct stat *finfo, const char *name);

int main(int argc, char *argv[])
{
        struct stat finfo;
        struct tm *pt;

        if (argc == 1) {
                fprintf(stderr, "file(s) must be specified!\n");
                exit(EXIT_FAILURE);
        }

        for (int i = 1; i < argc; ++i) {
                if (stat(argv[i], &finfo) == -1)
                        exit_sys("stat");
                printf("%s\n", get_ls(&finfo, argv[i]));
```





```
        }

        return 0;
}

void exit_sys(const char *msg)
{
        perror(msg);

        exit(EXIT_FAILURE);
}

char *get_ls(struct stat *finfo, const char *name)
{
        static char buf[LS_BUFSIZE];
        static mode_t modes[] = {S_IRUSR, S_IWUSR, S_IXUSR, S_IRGRP, S_IWGRP, S_IXGRP, S_IROTH,
S_IWOTH, S_IXOTH};
        static mode_t ftypes[] = {S_IFBLK, S_IFCHR, S_IFIFO, S_IFREG, S_IFDIR, S_IFLNK, S_IFSOCK};
        int index;
        struct tm *ptime;

        index = 0;
        for (int i = 0; i < 7; ++i)
                if ((finfo->st_mode & S_IFMT) == ftypes[i]) {
                        buf[index++] = "bcp-dls"[i];
                        break;
                }

        for (int i = 0; i < 9; ++i)
                buf[index++] = finfo->st_mode & modes[i] ? "rwx"[i % 3] : '-';

        ptime = localtime(&finfo->st_mtim.tv_sec);

        index += sprintf(buf + index, " %llu", (unsigned long long)finfo->st_nlink);
        index += sprintf(buf + index, " %llu", (unsigned long long)finfo->st_uid);
        index += sprintf(buf + index, " %llu", (unsigned long long)finfo->st_uid);
        index += sprintf(buf + index, " %llu", (unsigned long long)finfo->st_size);
        index += strftime(buf + index, LS_BUFSIZE, " %b %2e %H:%M", ptime);

        sprintf(buf + index, " %s", name);

        return buf;
}

/*-------------------------------------------------------------------------------------------------
-------------------------
    fstat fonksiyonu stat fonksiyonunun yol ifadesi değil dosya betimleyicisi alan biçimidir.
Prototipi şöyledir:

    int fstat(int fd, struct stat *buf);

    Genel olarak işletim sisteminin dosya betimleyicisinden hareketle i-node bilgilerine erişmesi
yol ifadesinden hareketle
    erişmesinden daha hızlı olmaktadır. Çünkü open fonksiyonunda zaten open dosyanın i-node
bilgilerine erişip onu dosya nesnesinin
    içerisine almaktadır. Tabii önce dosyayı açıp sonra fstat uygulamak anlamsız bir yöntemdir.
Ancak zaten biz bir dosyayı
    başka amaçla açmışsak onun bilgilerini fstat ile daha hızlı elde edebiliriz.
-------------------------------------------------------------------------------------------------
-------------------------*/

#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <unistd.h>

#define LS_BUFSIZE              4096
```





```
void exit_sys(const char *msg);
char *get_ls(struct stat *finfo, const char *name);

int main(int argc, char *argv[])
{
        int fd;
        struct stat finfo;

        if (argc != 2) {
                fprintf(stderr, "wrong number of arguments!...\n");
                exit(EXIT_FAILURE);
        }

        if ((fd = open(argv[1], O_RDONLY)) == -1)
                exit_sys("open");

        /* burada dosyayla ilgili birtakım işlemler yapılıyor */

        if (fstat(fd, &finfo) == -1)
                exit_sys("fstat");

        printf("%s\n", get_ls(&finfo, "sample.c"));

        close(fd);

        return 0;
}

void exit_sys(const char *msg)
{
        perror(msg);

        exit(EXIT_FAILURE);
}

char *get_ls(struct stat *finfo, const char *name)
{
        static char buf[LS_BUFSIZE];
        static mode_t modes[] = {S_IRUSR, S_IWUSR, S_IXUSR, S_IRGRP, S_IWGRP, S_IXGRP, S_IROTH,
S_IWOTH, S_IXOTH};
        static mode_t ftypes[] = {S_IFBLK, S_IFCHR, S_IFIFO, S_IFREG, S_IFDIR, S_IFLNK, S_IFSOCK};
        int index;
        struct tm *ptime;

        index = 0;
        for (int i = 0; i < 7; ++i)
                if ((finfo->st_mode & S_IFMT) == ftypes[i]) {
                        buf[index++] = "bcp-dls"[i];
                        break;
                }

        for (int i = 0; i < 9; ++i)
                buf[index++] = finfo->st_mode & modes[i] ? "rwx"[i % 3] : '-';

        ptime = localtime(&finfo->st_mtim.tv_sec);

        index += sprintf(buf + index, " %llu", (unsigned long long)finfo->st_nlink);
        index += sprintf(buf + index, " %llu", (unsigned long long)finfo->st_uid);
        index += sprintf(buf + index, " %llu", (unsigned long long)finfo->st_uid);
        index += sprintf(buf + index, " %llu", (unsigned long long)finfo->st_size);
        index += strftime(buf + index, LS_BUFSIZE, " %b %2e %H:%M", ptime);

        sprintf(buf + index, " %s", name);

        return buf;
}

/*-------------------------------------------------------------------------------------------------
```





```
-------------------------
    Bir dosyayı işaret eden özel dosyalara "sembolik bağlantı dosyaları (symbolic link files)"
denilmektedir. Sembolik bağlantı dosyaları
    aynı zamanda "soft link" dosyalar biçiminde de isimlendirilmektedir. Sembolik bağlantı
dosyaları bir dosyayı işaret eden dosyalardır.
    Bunlar gerçek anlamda birer dosya değildir. Adeta bir "pointer" dosyadır. İşletim sistemleri
sembolik bağlantı dosyaları için diskte
    yalnızca bir i-node elemanı tutmaktadır. Sembolik bağlantı dosyaları komut satırında ln -s
komutuyla yaratılabilirler. Örneğin:

    ln -s x.dat y.dat

    Burada "x.dat" dosyanının "y.dat" isimli bir sembolik bağlantı dosyası oluşturulmuştur. ls -l
komutunda sembolik bağlantı dosyaları ok işaretiyle
    gösterilmektedir. Örneğin:

    kaan@kaan-virtual-machine:~/Study/Unix-Linux-SysProg$ ls -l x.dat y.dat

    -rwxr-xr-x 1 kaan study 0 Kas 27 13:07 x.dat
    lrwxrwxrwx 1 kaan study 5 Ara 10 10:11 y.dat -> x.dat

    Sembolik bağlantı dosyaları "l" dosya türü ile gösterilmektedir. Bir sembolik bağlantı dosyası
başka bir sembolik bağlantı dosyasını gösterebilir.
    Örneğin:

    kaan@kaan-virtual-machine:~/Study/Unix-Linux-SysProg$ ls -l x.dat y.dat z.dat
    -rwxr-xr-x 1 kaan study 0 Kas 27 13:07 x.dat
    lrwxrwxrwx 1 kaan study 5 Ara 10 10:11 y.dat -> x.dat
    lrwxrwxrwx 1 kaan study 5 Ara 10 10:48 z.dat -> y.dat

    Sembolik bağlantı dosyaları yaratıldığında erişim hakları otomatik olarak "lrwxrwxrwx"
biçiminde oluşturulmaktadır. Sembolik bağlantı dosyalarının
    kendi erişim haklarının bir önemi yoktur. Bu dosyaların kendi erişim haklarısistem tarafından
herhangi bir biçimde kullanılmamaktadır.

    open gibi POSIX fonksiyonlarının pek çoğu sembolik bağlantı dosyalarında bağlantıyı
izlemektedir. Yani örneğin biz open fonksiyonu
    ile bir sembolik bağlantı dosyasını açmaya çalışsak open fonksiyonu o dosyayı değil o dosyanın
gösterdiği dosyayı açmaya çalışır.
    Yukarıdaki örnekte biz "z.dat" dosyasını açmak istesek aslında "x.dat" dosyası açılacaktır. Bu
durum ileride ele alacağımız POSIX fonksiyonlarının
    hemen hepsinde böyledir. Ancak lstat fonksiyonu istisnalardan biridir.

    Bir dosya fonksiyonuna yol ifadesi olarak sembolik bağlantı dosyası verildiğinde fonksiyon
(lstat dışındaki fonksiyonlar) sembolik bağlantıyı
    izlemektedir. Ancak bu izleme sırasında bir döngü oluşabilir. Örneğin a sembolik bağlantı
dosyası b sembolik bağlantı dosyasını,
    b sembolik bağlantı dosyası da c sembolik bağlantı dosyasını gösteriyor olabilir. c sembolik
bağlantı dosyası da yeniden a sembolik bağlantı
    dosyasını gösteriyor olabilir. Böyle bir işlemde sonsuz döngü söz konusu olmaktadır. İşte
dosya fonksiyonları bu durumu da dikkate alır ve böylesi bir
    döngüsellik varsa başarısızlıkla geri döner. Bu başarısızlık durumunda errno değeri ELOOP
biçiminde set edilmektedir. Aslında POSIX sistemlerinde işletim
    sistemi tarafından belirlenmiş maksimum link izleme sayısı vardır. Bu sayı aşıldığında ilgili
fonksiyomn başarısız olup errno değişkeni ELOOP değeri
    ile set edilmektedir. (POSIX standartlarında maksimum link izleme değeri <sys/limits.h>
içerisinde SYMLOOP_MAX sembolik sabitiyle belirtilmektedir.
    Ancak bu sembolik sabit define edilmiş olmak zorunda değildir. Ayrıca POSIX sistemlerinde
olabilecek en düşük sembolik link izleme sayısı da _POSIX_SYMLOOP_MAX (8)
    değeri ile belirlenmiştir.) Yani aslında sembolik bağlantıların döngüye genellikle girmesi
maksimum sayacın aşılması ile anlaşılmaktadır.

    Bir sembolik bağlantı dosyasının gösterdiği dosya silinirse burada tuhaf bir durum oluşur.
İşte bu tür durumlarda bu sembolik bağlantı dosyası
    kullanıldığında (örneğin open fonksiyonuyla açılmaya çalışıldığında) sanki dosya yokmuş gibi
bir hata oluşurb (ENOENT). Çünkü bağlantının işaret ettiği
    bir dosya bulunmamaktadır. Windows sistemlerinde sembolik bağlantı dosyalarının bir benzerleri
"kısayol (shortcut)" dosyalar biçiminde karşımıza çıkmaktadır.
```

lstat fonksiyonu ile stat fonksiyonu arasındaki tek fark eğer stat bilgisi elde edilecek dosya bir sembolik bağlantı dosyası ise
stat fonksiyonun bu bağlantının gösterdiği dosyanın bilgisini alması ancak lstat fonksiyonun sembolik bağlantı dosyasının kendi bilgisini almasıdır.
Diğer dosyalar için bu fonksiyon arasında bir farklılık yoktur. Örneğin:

```
parallels@ubuntu-linux-20-04-desktop:~/Study/Unix-Linux-SysProg$ ls -l sample.c x
-rw-rw-r-- 1 parallels parallels 1748 Dec  4 13:46 sample.c
lrwxrwxrwx 1 parallels parallels    8 Dec  4 13:47 x -> sample.c
```

Burada "x" bir sembolik bağlantı dosyasıdır ve bu dosya "sample.c" dosyasını göstermektedir. İşte biz "x" dosyasının stat bilgilerini
stat fonksiyonu ile almaya çalışırsak stat bize aslında "sample.c" dosyasının bilgilerini verir. Ancak biz "x" dosyasının stat bilgilerini
lstat fonksiyonu ile alırsak lstat bize "x" dosyasının kendi bilgisini verir.

Aşağıdaki örnekte sembolik bağlantı dosyasının lstat ve stat fonksiyonlarıyla stat bilgileri alınmıştır.

```
-------------------------------------------------------------------------------------------------------------------------*/

#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <sys/stat.h>

#define LS_BUFSIZE              4096

void exit_sys(const char *msg);
char *get_ls(struct stat *finfo, const char *name);

int main(int argc, char *argv[])
{
        struct stat finfo;
        struct tm *pt;

        if (argc == 1) {
                fprintf(stderr, "file(s) must be specified!\n");
                exit(EXIT_FAILURE);
        }

        for (int i = 1; i < argc; ++i) {
                if (lstat(argv[i], &finfo) == -1)
                        exit_sys("stat");
                printf("%s\n", get_ls(&finfo, argv[i]));
        }

        return 0;
}

void exit_sys(const char *msg)
{
        perror(msg);

        exit(EXIT_FAILURE);
}

char *get_ls(struct stat *finfo, const char *name)
{
        static char buf[LS_BUFSIZE];
        static mode_t modes[] = {S_IRUSR, S_IWUSR, S_IXUSR, S_IRGRP, S_IWGRP, S_IXGRP, S_IROTH,
S_IWOTH, S_IXOTH};
        static mode_t ftypes[] = {S_IFBLK, S_IFCHR, S_IFIFO, S_IFREG, S_IFDIR, S_IFLNK, S_IFSOCK};
        int index;
        struct tm *ptime;

        index = 0;
        for (int i = 0; i < 7; ++i)
                if ((finfo->st_mode & S_IFMT) == ftypes[i]) {
```





```
                        buf[index++] = "bcp-dls"[i];
                        break;
                }

        for (int i = 0; i < 9; ++i)
                buf[index++] = finfo->st_mode & modes[i] ? "rwx"[i % 3] : '-';

        ptime = localtime(&finfo->st_mtim.tv_sec);

        index += sprintf(buf + index, " %llu", (unsigned long long)finfo->st_nlink);
        index += sprintf(buf + index, " %llu", (unsigned long long)finfo->st_uid);
        index += sprintf(buf + index, " %llu", (unsigned long long)finfo->st_gid);
        index += sprintf(buf + index, " %llu", (unsigned long long)finfo->st_size);
        index += strftime(buf + index, LS_BUFSIZE, " %b %2e %H:%M", ptime);

        sprintf(buf + index, " %s", name);

        return buf;
}

/*-------------------------------------------------------------------------------------------------------------------------
    Bir dosyanın stat bilgilerini görüntülemek için stat isimli kabuk komutu da bulundurulmuştur. Tabii bu komut aslında
    stat ve lstat POSIX fonksiyonlarını çağırarak elde ettikleri bilgileri yazdırmaktadır. Örneğin:

    kaan@kaan-virtual-machine:~/Study/Unix-Linux-SysProg$ stat sample.c
    File: sample.c
    Size: 329           Blocks: 8          IO Block: 4096   normal dosya
    Device: 805h/2053d  Inode: 1207667     Links: 2
    Access: (0644/-rw-r--r--)  Uid: ( 1000/    kaan)   Gid: ( 1001/   study)
    Access: 2022-12-10 10:59:52.700330245 +0300
    Modify: 2022-12-10 10:59:46.620211508 +0300
    Change: 2022-12-10 11:41:11.151049064 +0300

-------------------------------------------------------------------------------------------------------------------------*/

/*-------------------------------------------------------------------------------------------------------------------------

14. Ders 10/12/2022 - Cumartesi
-------------------------------------------------------------------------------------------------------------------------*/

/*-------------------------------------------------------------------------------------------------------------------------
    Bir dosyayı silmek için remove ve unlink isimli fonksiyonlar kullanılmaktadır. remove bir standart C fonksiyonudur. unlink ise
    bir POSIX fonksiyonudur. Bu iki fonksiyon tamamen aynı şeyi yapmaktadır. Fonksiyonların prototipleri şöyledir:

    #include <stdio.h>

    int remove(const char *path);

    #include <unistd.h>

    int unlink(const char *path);

    Fonksiyonlar başarı durumunda 0 değerine, başarısızlık durumunda -1 değerine geri dönmektedir.

    remove ve unlink fonksiyonlarıyla bir dosyayı silebilmek için prosesin dosyanın kendisine "w" hakkının olması gerekmez.
    Ancak dosyanın içinde bulunduğu dizine "w" hakkının olması gerekir. Bizim eğer dosyanın içinde bulunduğu dizine "w"
    hakkımız varsa dosyanın sahibi olmasak bile dosyayı silebiliriz. Tabii proses id'si 0 olan prosesler herhangi bir kontrol
```





uygulamadan bu silme işlemini yapabilirler.

Bir dosya unlink ya da remove fonksiyonlarıyla silindiğinde dizin girişi silinir. Ancak dosyanın kendisi dosyanın hard link sayacı
0'a düşmüşse silinmektedir. Ynai unlink ve remove fonksiyonları dosyayı dizin girişinden silerler. Sonra dosyanın hard link sayacını 1
eksiltirler. Eğer hard link sayacı 0'a düşmüşse dosyayı fiziksel olarak silerler. HArd link sayacının ne anlama geldiği ileride
ele alınacaktır.

Aşağıdaki örnekte komut satırından verilen yol ifadelerine ilişkin dosyalar silinmeye çalışılmıştır.

```
-------------------------------------------------------------------------------------------------------------------------*/

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main(int argc, char *argv[])
{
        if (argc == 1) {
                fprintf(stderr, "file name(s) must be specified!...\n");
                exit(EXIT_FAILURE);
        }

        for (int i = 1; i < argc; ++i)
                if (unlink(argv[i]) == -1)
                        perror("unlink");

        return 0;
}

/*-------------------------------------------------------------------------------------------------------------------------
    Daha önceden de belirttiğimiz gibi aslında "dizinler" birer dosya gibi organize edilmiştir. Dizin dosyalarının içerisinde
    "dizin girişleri (directory entries)" bulunmaktadır. Bir dizin girişinin formatı dosya sisteminden dosya sistemine değişebilmektedir.
    Ancak özet olarak bir dizin dosyasının içeriği şöyledir:

    Dizi Dosyası
    ------------
    dosya_ismi  i-node no
    dosya_ismi  i-node no
    dosya_ismi  i-node no
    ...
    dosya_ismi  i-node no
    dosya_ismi  i-node no
    dosya_ismi  i-node no

    Dosyaların asıl bilgileri (yani stat fonksiyonuyla elde ettiğimiz bilgiler) Diskte "I-Node Block" denilen bir bölgede saklanmaktadır.
    I-Node Block i-node elemanlarından oluşur. Her i-node elemanına ilk eleman 0 olmak üzere artan sırada bir numara karşı düşürülmüştür.
    İşletim sistemi bir dosya ile ilgili işlem yaparken kesinlikle o dosyanın i-node elemanına erişmek ve oradaki bilgileri kullanmak
    zorundadır.

    Bir dosya unlink ya da remove fonksiyonlarıyla silindiğinde kesinlikle dizin girişi silinmektedir. Ancak dosyanın silinip
    silinmeyeceği hard-link sayacına bağlıdır.

    Farklı dizin girişleri farklı isimlerle aynı i-node numaralarını işaret ediyorsa buna "hard link" denilmektedir. Örneğin:

    Dizin Dosyası
    --------------
    a.txt   12345678
```





```
    b.txt   12345678
    ...

    Burada bizim open fonksiyonuyla "a.txt" ya da "b.txt" dosyalarını açmamız arasında hiçbir farklılık yoktur. Çünkü dosyanın bütün bilgileri
    i-node elemanının içerisindedir. İşte biz bu dosyalardan örneğin "a.txt" dosyasını silersek aslında yalnızca dizin girişini silmiş oluruz.
    Çünkü işletim sistemi "a.txt" dosyasının işaret ettiği i-node elemanının başka bir giriş tarafından kullanıldığını gördüğü için i-node elemanını
    ve dosyanın diskteki varlığını silmez. İşte bu durum "har link sayacı" ile kontrol edilmektedir. Yukarıdaki örnekte dosyanın hard link sayacı 2'dir.
    Biz bu dizin girişlerinden birini sildiğimizde hard link sayacı 1'e düşer. Diğerini de sildiğimizde hard link sayacı 0'a düşer ve dosya gerçekten silinir.

    Bir dosyanın hard link'ini oluşturmak için ln kabuk komutu kullanılmaktadır. Örneğin:

    ln sample.c mample.c

    kaan@kaan-virtual-machine:~/Study/Unix-Linux-SysProg$ ls -li sample.c mample.c
    1207667 -rw-r--r-- 2 kaan study 329 Ara 10 10:59 mample.c
    1207667 -rw-r--r-- 2 kaan study 329 Ara 10 10:59 sample.c

    Dosyanın hard link sayacının 2 olduğuna dikkat ediniz.

    Bir dizin yaratıldığında onun içerisinde "." ve ".." biçiminde iki dizin girişi otomatik olarak yaratılmaktadır. (UNIX/Linux
    sistemlerinde başı "." ile başlayan dizin girişleri ls komutunda default olarak görüntülenmemektedir. Bunların görüntülenmesi için
    -a (all) seçeneğinin de kullanılması gerekir.) "." dizin girişi kendi dizin dosyasının i-node elemanını ".." dizin girişi ise
    üst dizinin i-node elemanını göstermektedir. Bu nedenle bir dizin yaratıldığında dizin dosyasına ilişkin hard-link sayacı
    2 olur. O dizinin içerisinde yaratılan her dizin ".." girişini içereceğinden dolayı o dizinin hard link sayacını artıracaktır.

    Belli bir i-node elemanını gösteren dizin girişlerinin elde edilmesine yönelik bu sistemlerde pratik bir yol yoktur. Yapılacak şey
    diskteki tüm dosyaları gözden geçirip i-node numaralarından onların aynı i-node elemanını gösterip göstermediğini anlamaktır.
-------------------------------------------------------------------------------------------------------------------------*/

/*-------------------------------------------------------------------------------------------------------------------------
    Yukarıda da belirttiğimiz gibi dosya bilgileri disk üzerinde i-node bloktaki i-node elemanının içerisinde tutulmaktadır.
    stat fonksiyonları erişim bilgilerini buradan almaktadır (ls komutu da stat fonksiyonları kullanılarak yazılmıştır).
    Dosyanın erişim hakları yine anımsayacağınız gibi open fonksiyonunda dosya yaratılırken belirlenmektedir. İşte bir dosyanın erişim haklarını
    dosyanın içine dokunmadan chmod isimli POSIX fonksiyonu ile değiştirebiliriz. Fonksiyonun prototipi şöyledir:

    #include <sys/stat.h>

    int chmod(const char *path, mode_t mode);

    Fonksiyonun birinci parametresi dosyanın yol ifadesini, ikinci parametresi erişim haklarını belirtmektedir. Fonksiyon başarı durumunda 0 değerine,
    başarısızlık durumunda -1 değerine geri dönmektedir. Erişim hakları 2008 stnadralarına kadar S_IXXX sembolik sabitleriyle
    oluşturulmak zorundaydı. Ancak 2008 ve sonrasında artık bu S_IXXX sembolik sabitlerinin değerleri belirlendiği için programcı doğrudan
    octal bir sayı biçiminde bu erişim haklarını verebilir. Fakat tavsiye edilen yine S_IXXX sembolik sabitlerinin kullanılmasıdır.

    Bir dosyanın erişim haklarını chmod fonksiyonuyla değiştirebilmek için prosesin etkin kullanıcı id'sinin dosyanın kullanıcı id'si ile
    aynı olması ya da prosesin etkin kullanıcı id'sinin 0 olması gerekmektedir. Dosyanın dördüncü
```

3 btilik S_ISUID, S_ISGID ve S_ISVTX
    erişim hakları da bu fonksiyonla set edilmeye çalışılabilir. Ancak bazı sistemler S_ISUID ve S_ISGID erişim haklarını değiştirmeye izin vermeyebilmektedir.

    chmod POSIX fonksiyonu prosesin umask değerini dikkate almamaktadır. Yani fonksiyonda belirttiğimiz erişim haklarının hepsi
    dosyaya yansıtılmaktadır.

    Aşağıdaki girilen octal digitlerle dosyaların erişim haklarını değiştiren bir örnek verilmiştir. Bu örnekte doğrudan chmod fonksiyonunda
    bitmask değerler sayısal olarak kullanılmıştır. Bu durumun eski sistemlerde sorunlu olabileceğini bir kez daha vuruguluyoruz.

```
----------------------------------------------------------------------------------------------------------*/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/stat.h>

int check_mode(const char *str);
void exit_sys(const char *msg);

int main(int argc, char *argv[])
{
        int mode;

        if (argc < 3) {
                fprintf(stderr, "too few parameters!...\n");
                exit(EXIT_FAILURE);
        }

        if (!check_mode(argv[1])) {
                fprintf(stderr, "invalid mode: %s\n", argv[1]);
                exit(EXIT_FAILURE);
        }

        sscanf(argv[1], "%o", &mode);
        for (int i = 2; i < argc; ++i)
                if (chmod(argv[i], mode) == -1)
                        fprintf(stderr, "cannot change mode: %s\n", argv[1]);

        return 0;
}

int check_mode(const char *str)
{
        if (strlen(str) > 4)
                return 0;

        for (int i = 0; str[i] != '\0'; ++i)
                if (str[i] < '0' || str[i] > '7')
                        return 0;

        return 1;
}

void exit_sys(const char *msg)
{
        perror(msg);

        exit(EXIT_FAILURE);
}

/*----------------------------------------------------------------------------------------------------------
--------------------------
```

    Aşağıdaki örnekte eski POSIX standartları da dikkate alınarak mode bilgisi S_IXXX sembolik sabitlerinin bit düzeyinde
    OR'lanması ile oluşturulmuştur.





```
----------------------------------------------------------------------------------------------------------
------------------------*/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/stat.h>

int check_mode(const char *str);
void exit_sys(const char *msg);

int main(int argc, char *argv[])
{
        int modeval;
        mode_t modes[] = {S_ISUID, S_ISGID, S_ISVTX, S_IRUSR, S_IWUSR, S_IXUSR, S_IRGRP, S_IWGRP,
S_IXGRP, S_IROTH, S_IWOTH, S_IXOTH};
        mode_t mode;

        if (argc < 3) {
                fprintf(stderr, "too few parameters!...\n");
                exit(EXIT_FAILURE);
        }

        if (!check_mode(argv[1])) {
                fprintf(stderr, "invalid mode: %s\n", argv[1]);
                exit(EXIT_FAILURE);
        }

        sscanf(argv[1], "%o", &modeval);

        mode = 0;
        for (int i = 11; i >= 0; --i)
                if (modeval >> i & 1)
                        mode |= modes[11 - i];

        for (int i = 2; i < argc; ++i)
                if (chmod(argv[i], mode) == -1)
                        fprintf(stderr, "cannot change mode: %s\n", argv[1]);

        return 0;
}

int check_mode(const char *str)
{
        if (strlen(str) > 4)
                return 0;

        for (int i = 0; str[i] != '\0'; ++i)
                if (str[i] < '0' || str[i] > '7')
                        return 0;

        return 1;
}

void exit_sys(const char *msg)
{
        perror(msg);

        exit(EXIT_FAILURE);
}

/*----------------------------------------------------------------------------------------------------------
--------------------------
```

    chmod POSIX fonksiyonunun yanı sıra bir de dosya betimleyicisi ile çalışan fchmod fonksiyonu vardır. Eğer dosyayı zaten açmışsak chmod yerine
    fchmod fonksiyonu daha hızlı bir çalışma sunmaktadır. Fonksiyonun prototipi şöyledir:

```
    #include <sys/stat.h>
```





```
    int fchmod(int fd, mode_t mode);

----------------------------------------------------------------------------------------------------------
------------------------*/

/*----------------------------------------------------------------------------------------------------------
--------------------------
```

    Dosyanın erişim haklarını değiştirmek için chmod isimli bir kabuk komutu da bulunmaktadır. Bu kabuk komutu tabii chmod POSIX
    fonksiyonu kullanılarak yazılmıştır. Bu kabuk komutunun kullanımının birkaç biçimi vardır. Tipik olarak komutta erişim hakları
    octal digitlerle belirtilmektedir. Örneğin:

```
    chmod 664 a.txt b.txt
```

    Burada 664'ün bit karşılığı şöyledir: 110 110 100. Bu erişim hakları olarak şu anlama gelmektedir: rw-rw-r--. Komutun ikinci kullanımı
    + ve -'li kullanımıdır. Örneğin:

```
    chmod +w a.txt
```

    Burada "a.txt" dosyasının "owner", "group" ve "other" "w" hakkı eklemektedir. Komutta "-" ilgili hakkın çıkartılacağınıbelirtmektedir.
    Bunların önüne u, g, o ya da a harfleri getirilebilir. Örneğin:

```
    chmod o+w a.txt
```

    Burada yalnızca "other" için "w" hakkı eklenmiştir. a hepsine anlamına gelir. Örneğin:

```
    chmod a-w a.txt
```

    Burada owner, group ve other için "w" hakları silinmiştir. Tabii birden fazlası kombine edilebilir. Örneğin:

```
    chmod 0 a.txt
    chmod ug+rw a.txt
```

    Komutta octal sayı belirtilirse umask etkili olmaz. Ancak ocatl sayıyerine ugua ve rwx belirtilirse bu durumda
    kabuğun umask değeri etkili olmaktadır.

    Komutun başka ayrıntıları da vardır. Bunun için ilgili dokimanlara başvurabilirsiniz.

```
----------------------------------------------------------------------------------------------------------
------------------------*/

/*----------------------------------------------------------------------------------------------------------
--------------------------
```

    Bir dosyanın kullanıcı id'si ve grup id'si dosya yaratılırken belirleniyordu. Ancak programcı isterse dosyanın kullajcı id'sini
    ver grup id'sini chown isimli POSIX fonksiyonu ile değiştirebilir. Fonksiyonun prototipi şöyledir:

```
    #include <unistd.h>

    int chown(const char *path, uid_t owner, gid_t group);
```

    Fonksiyonun birinci parametresi dosyanın yol ifadesini ikinci parametresi değiştirilecek kullanıcı id'sini ve üçüncü parametresi de değiştirilecek
    grup id'sini belirtmektedir. Bir dosyanın kullanıcı ve grup id'lerinin değiştirilmesi kötüye kullanıma açık bir durum oluşturabilmektedir.
    (Yani örneğin kaan kullancısı kendi dosyasını sanki ali'nin dosyayıymış gibi gösterirse burada bir kötü niyet de söz konusu olabilir.)
    Bu nedenle bu fonksiyonun kullanımı üzerinde bazı kısıtlar vardır. Şöyle ki:

    1) Eğer prosesin etkin kullanıcı id'si dosyanın kullanıcı id'si ile aynı ise bu durumda chwon fonksiyonu dosyanın grup id'sini
    kendi grup id'si olarak ya da ek gruplarının birinin id'si olarak eğiştirebilmektedir. Ancak dosyanın kullanıcı id'sinin değiştirilmesi
    işletim sisteminin izninе bağlıdır. Modern sistemler bu izni vermemektedir. Ancak bazı eski





sistemler bu izni vermektedir. Bu izin
    "change own restricted" ismiyle ifade edilmektedir. İlgili sistemin bu izni verip vermediği <unisth.h> dosyası içerisindeki _POSIX_CHOWN_RESTRICTED
    sembolik sabitiyle derleme aşamsında sorgulanbilir.

    2) Proses id'si 0 olan root prosesler her zaman dosyanın kullanıcı ve grup id'sini istedikleri gibi değiştirebilirler.

    Fonksiyon ile yalnızca kullanıcı id'si ya da grup id'si değiştirilebilir. Bu durumda değiştirilmeyecek değer için -1 girilmelidir.
    Fonksiyon başarı durumunda 0 değerine, başarısızlık durumunda -1 değerine geri dönmektedir. Change on restricted durumu aşağıdaki gibi
    #ifdef komutuyla sorgulanabilir:

```
        #include <stdio.h>
        #include <unistd.h>

        int main(int argc, char *argv[])
        {
        #ifdef _POSIX_CHOWN_RESTRICTED
                printf("chown restricted\n");
        #else
                printf("chown not restricted\n");
        #endif

                return 0;
        }
```

    Aşağıda chown fonksiyonun örnek bir kullanımını görüyorsunuz:

```
----------------------------------------------------------------------------------------------------------
------------------------*/

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

void exit_sys(const char *msg);

int main(void)
{
        if (chown("test.txt", 1000, -1) == -1)
                exit_sys("chown");

        printf("Ok\n");

        return 0;
}

void exit_sys(const char *msg)
{
        perror(msg);

        exit(EXIT_FAILURE);
}

/*----------------------------------------------------------------------------------------------------------
--------------------------
```

    truncate isimli POSIX fonksiyonu bir dosyanın boyutunu değiştirmek için kullanılmaktadır. Fonksiyonun prototipi şöyledir:

```
    #include <unistd.h>

    int truncate(const char *path, off_t length);
```

    Fonksiyonun birinci parametresi dosyanın yol ifadesini almaktadır. İkinci parametresi dosyanın yeni uzunluğunu belirtir. Bu fonksiyon
    genellikle dosyanın sonundaki kısmı atarak onun boyutunu küçültmek amacıyla kullanılmaktadır. Burada belirtilen uzunluk dosyanın
    gerçek uzunluğundan küçükse dosyanın sonundaki ilgili kısım yok edilir ve dosya burada

```
belirtilen uzunluğa getirilir. (Fonksiyonun
    ismi tipik olarak dosyaların küçültüleceği fikriyle "trunctate" olarak verilmiştir.) Biz
truncate fonksiyonu ile dosyayı büyütmek
    de isteyebiliriz. Bu durumda dosya büyütülür ve büyütülen kısım 0'larla doldurulur. Bugünkü
sistemlerde dosya sistemi
    "dosya deliklerini (file holes)" destekliyorsa; büyütme, delik (hole) oluşturularak
yapılmaktadır. Fonksiyon, başarı durumunda 0
    değerine, başarısızlık durumunda -1 değerine geri döner ve errno değişkeni uygun biçimde set
edilir. Tabii truncate yapabilmek için
    prosesin dosyaya yazma hakkının olması olması gerekmektedir.

    truncate fonksiyonunun yol ifadesini alarak değil, dosya betimleyicisini alarak aynı işlemi
yapan ftruncate isminde bir benzeri de vardır.
    Fonksiyonun prototipi şöyledir:

    #include <unistd.h>

    int ftruncate(int fd, off_t length);

    Fonksiyonun birinci parametresi dosya betimleyicisini almaktadır. İkinci parametresi dosyanın
yeni uzunluğudur. Fonksiyon başarı durumunda 0
    değerine, başarısızlık durumunda -1 değerine geri döner ve errno değişkeni uygun biçimde set
edilir. Tabii ftruncate yapabilmek için
    prosesin dosyanın yazma modunda açılmış olması gerekmektedir.

    Fonksiyonun işlev bakımından truncate fonksiyonundan hiçbir farkı yoktur.

    Aşağıdaki örnekte daha önce var olan "test.dat" dosyası 1000 byte uzunluğa çekilmiştir.
-------------------------------------------------------------------------------------------------
------------------------*/

#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <unistd.h>

void exit_sys(const char *msg);

int main(void)
{
        int fd;

        if ((fd = open("test.dat", O_RDWR)) == -1)
                exit_sys("open");

        if (ftruncate(fd, 1000) == -1)
                exit_sys("open");

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

        close(fd);

        return 0;
}

void exit_sys(const char *msg)
{
        perror(msg);

        exit(EXIT_FAILURE);
}

/*-------------------------------------------------------------------------------------------------
------------------------

15. Ders 11/12/2022 - Pazar
-------------------------------------------------------------------------------------------------
------------------------*/
```





```
/*-------------------------------------------------------------------------------------------------
------------------------
    chown POSIX fonksiyonunun dosya betimleyicisi ile çalışan fchown isminde bir benzeri de
vardır. Bu fonksiyonun chown fonksiyonundan
    tek farkı dosyanın yol ifadesini değil dosya betimleyicisini larak işlem yapmasıdır. Elimizde
zaten açmış olduğumuz bir dosya varsa
    biz bu betimleyiciyi kullanarak bu işlemi nispeten daha hızlı yapabiliriz. Fonksiyonun
prototipi şöyledir:

    #include <unistd.h>

    int fchown(int fd, uid_t owner, gid_t group);
-------------------------------------------------------------------------------------------------
------------------------*/

/*-------------------------------------------------------------------------------------------------
------------------------
    Dosyanın kullanıcı ve grup id'lerini değiştirebilmek için chown isimli bir kabuk komutu da
bulundurulmuştur. Komut aşağıdaki biçimlerde kullanılmaktadır:

    sudo chwon kaan:study test.txt
    sudo chown kaan test.txt
    sudo chown :study test.txt

-------------------------------------------------------------------------------------------------
------------------------*/

/*-------------------------------------------------------------------------------------------------
------------------------
    Bir dizin (directory) yaratmak için mkdir isimli POSIX fonksiyonu kullanılmaktadır. Dizin
yaratma işlemi open fonksiyonuyla
    yapılamamaktadır. mkdir fonksiyonunun prototipi şöyledir:

    #include <sys/stat.h>

    int mkdir(const char *path, mode_t mode);

    Fonksiyonun birinci parametresi yaratılacak dizinin yol ifadesini, ikinci parametresi ise
erişim haklarını belirtmektedir. Fonksiyon yine
    başarı durumunda 0 değerine, başarısızlık durumunda -1 değerine geri dönmektedir.

    Dizin yaratırken erişim haklarında 'x' hakkını bulundurmayı unutmayınız. Anımsanacağı gibi
dizinlerde 'x' hakkı "içinden geçilebilirlik"
    anlamına geliyordu. mkdir fonksiyonu tıpkı open fonksiyonu gibi prosesin umask değerinden
etkilenmektedir. O halde istediğiniz
    erişim haklarının hepsinin dizine yansıtılmasını istiyorsanız işin başında umask(0) çağrısıyla
prosesinizin umask değerini
    sıfırlayabilirsiniz.

    Bir dizin yaratıldığında içerisinde "." ve ".." isminde iki dizin girişi bulunmaktadır. Daha
önceden de belirtildiği gibi "." dizin
    girişi bulunulan dizinin i-node elemanını, ".." dizin girişi ise üst dizinin i-node elemanını
işaret eder. Bu nedenle bir dizin yaratıldığında
    kendi dizininin ve üst dizinin har link sayaçları bir artırılmaktadır.

    Aşağıda komut satırından verilen isimle bir dizin yaratn örnek verilmiştir.
-------------------------------------------------------------------------------------------------
------------------------*/

#include <stdio.h>
#include <stdlib.h>
#include <sys/stat.h>

void exit_sys(const char *msg);

int main(int argc, char *argv[])
{
        if (argc != 2) {
                fprintf(stderr, "wrong number of arguments!...\n");
```





```
                exit(EXIT_FAILURE);
        }

        if (mkdir(argv[1], S_IRWXU|S_IRWXG|S_IRWXO) == -1)
                exit_sys("mkdir");

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

        return 0;
}

void exit_sys(const char *msg)
{
        perror(msg);

        exit(EXIT_FAILURE);
}

/*-------------------------------------------------------------------------------------------------
------------------------
    Komut satırında dizin yaratmak için mkdir isminde bir kabuk komutu da bulunmaktadır. Tabii bu
komut mkdir POSIX fonksiyonu
    kullanılarak yazılmıştır. Komut default durumda umask değerinden etkilenir. Ancak -m ya da --
mode seçeneği ile biz erişim
    haklarını octal basamaklar biçiminde belirtebilmekteyiz. Örneğin:

    mkdir xxx
    mkdir -m 777 yyy

    Dizinler çin de hard link çıkartılabilmektedir. Ancak bu durum dizin ağacını dolaşan kodların
sonsuz döngüye girmesine yol açabilmektedir.
    Bu nedenle dizinler için hard link çıkartmak yerine soft link (sembolik bağlantı) çıkartmak
tercih edilmektedir.
-------------------------------------------------------------------------------------------------
------------------------*/

/*-------------------------------------------------------------------------------------------------
------------------------
    Bir dizini silmek için unlink ya da remove fonksiyonları kullanılamaz. Dizin silmek için rmdir
isimli özel bir POSIX fonksiyonu
    bulundurulmuştur. Fonksiyonun prototipi şöyledir:

    #include <unistd.h>

    int rmdir(const char *path);

    Fonksiyon parametre olarak silinecek dizinin yol ifadesini alır. Başarı durumunda 0 değerine,
başarısızlık durumunda -1 değerine
    geri döner.

    rmdir fonksiyonu ile içinde dosya olan dizinler silinememektedir. Bu durum güvenlik amacıyla
düşünülmüştür. İçi boş dizin demek
    içinde yalnızca "." ve ".." girişlerinin bulunduğu dizin demektir. Zaten UNIX/Linux, macOS ve
Windows sistemlerinde bu iki özel
    dizin girişi silinememektedir. rmdir fonksiyonuna bir dizini işaret eden sembolik bağlantı
dosyası verilirse fonksiyob bağlantıyı
    izlemez. Başarısız olur ve errno değeri ENOTDIR biçiminde set edilir.

    rmdir fonksiyonun başarılı olabilmesi için prosesin dizine yazma hakkına sahip olması gerekmez
ancak dizinin içinde bulunduğu
    dizine yazma hakkına sahip olması gerekir.
-------------------------------------------------------------------------------------------------
------------------------*/

/*-------------------------------------------------------------------------------------------------
------------------------
    Komut satırından dizin silmek için rmdir isimli bir kabuk komutu da bulunmaktadır. Tabii bu
komut aslıdan rmdir POSIX fonksiyonu
    kullanılarak yazılmıştır. Tabii rmdir komutuyla dizin silmek için yine dizinin boş olması
```





```
gerekir. İçi dolu dizinleri tek hamlede silmek için
    rm komutu -r seçeneği ile kullanılabilir. Örneğin:

    rm -r xxx

-------------------------------------------------------------------------------------------------
------------------------*/

/*-------------------------------------------------------------------------------------------------
------------------------
    Anımsanacağı gibi genel olarak pek çok UNIX/Linux sisteminde kullanıcılar hakkında bilgiler
/etc/passwd ve /etc/group
    dosyalarında tutuluyordu. Bu dosyalardaki satırlar ':' ile ayrılmış olan alanlardan
oluşmaktaydı. Bu dosyalar ve bunların formatları
    POSIX standartlarında belirtilmemiştir. Onun yerine POSIX standartlarında bu dosyalardan okuma
yapan özel fonksiyonlar bulundurulmuştur.
    (Yani aslında bir POSIX sisteminde /etc/passwd ve /etc/group dosyaları bu isimlerde ve
Linux'tak içerikte bulunmak zorunda değildir.
    Ancak bu bilgileri alan aşağıda açıklayacağımız POSIX fonksiyonları bulunmak zorundadır.)
-------------------------------------------------------------------------------------------------
------------------------*/

/*-------------------------------------------------------------------------------------------------
------------------------
    /etc/passwd dosyası üzerinde parse işlemi yapan fonksiyonların prototipleri <pwd.h> dosyası
içerisinde bulundurulmuştur.
    getpwnam POSIX fonksiyonu bir kullanıcının ismini alarak o kullanıcı hakkında /etc/passwd
dosyasında belirtilen bilgileri
    vermektedir. Fonksiyonun prototipi şöyledir:

    #include <pwd.h>

    struct passwd *getpwnam(const char *name);

    Fonksiyon parametre olarak kullanıcı ismini almaktadır. Başarı durumunda o kullanıcıya ilişkin
bilgileri barındıran statik düzeyde tahsis edilmiş olan
    struct passwd isimli bir yapı nesnesinin adresiyle geri dönmektedir. struct passwd yapısı
şöyle bildirilmiştir:

    struct passwd {
        char   *pw_name;       /* username */
        char   *pw_passwd;     /* user password */
        uid_t   pw_uid;        /* user ID */
        gid_t   pw_gid;        /* group ID */
        char   *pw_gecos;      /* user information */
        char   *pw_dir;        /* home directory */
        char   *pw_shell;      /* shell program */
    };

    Aslında bu yapının elemanları /etc/passwd dosyasındaki satır bilgilerinden oluşmaktadır.
Aşağıda /etc/passwd dosyasından birkaç satır
    verilmiştir:

    ...
    kaan:x:1000:1001:Kaan Aslan,,,:/home/kaan:/bin/bash

student:$6$EW3bJuIgtpIfgbdm$Sy4Z4XNdxgBrNlzc7cEnEJn2gp36XCvaIUqaH9p8ZZrtfF3qQZ7KTK7qpM4T54/p5Lck24
ZknXC1EuXm2hnBm1:1001:1001:Student,,,:/home/student:/bin/ulak-shell
    ali:x:1001:1001::/home/ali:/bin/myshell
    veli:x:1002:1002::/home/veli:/bin/bash
    ...

    Yapının pw_nam elemanı kullanıcı ismini, pw_passwd elemanı parola bilgisini, pw_uid ve pw_gid
elemanları login olunduğunda
    çalıştırılacak programa ilişkin prosesin gerçek ve etkin kullanıcı ve group id değerlerini
pw_gecos yorum bilgisini (kullanıya ilişkin ek
    birtakım bilgileri, pw_dir login olunduğunda çalıştırılacak programa ilişkin prosesin çalışma
dizinini ve pw_shell elemanı da login olunduğunda çalıştırılacak
    programı belirtmektedir.)
```

getpwnam fonksiyonu iki nedenden dolayı başarısız olabilir. Birincisi belirtilen isme ilişkin bir kullanıcının /etc/passwd dosyası içerisinde
bulunamamasıdır. İkincisi ise daha patolojik durumlardır. Yani bir IO hatası, /etc/passwd dosyasının silinmiş olması gibi. Programcının
Eğer fonksiyon isme ilişkin bir kayıt bulamadıysa errno değerini değiştirmemektedir. Ancak diğer hatalı durumlarda errno değerini uygun biçimde
set etmektedir. Dolayısıyla programcı bu tür durumlarda fonksiyonu çağırmadan önce errno değerini 0'a çeker. Sonra fonksiyon başarısız olduğunda
errno değerine bakar. Eğer bu değer hala 0 ise fonksiyonun ilgili kullanıcı ismini bulamadığından dolayı başarısız olduğu anlaşılır.

Aşağıdaki örnekte komut satırından ismi alınan kullanıcının /etc/passwd dosyasındaki bilgileri ekrana (stdout dosyasına) yazdırılmıştır.

```
----------------------------------------------------------------------------------------------------
-------------------------*/

#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <pwd.h>

void exit_sys(const char *msg);

int main(int argc, char *argv[])
{
        struct passwd *pass;

        if (argc != 2) {
                fprintf(stderr, "wrong number of arguments!...\n");
                exit(EXIT_FAILURE);
        }

        errno = 0;
        if ((pass = getpwnam(argv[1])) == NULL) {
                if (errno == 0) {
                        fprintf(stderr, "user name cannot found!...\n");
                        exit(EXIT_FAILURE);
                }
                exit_sys("getpwnam");
        }

        printf("User Name: %s\n", pass->pw_name);
        printf("Password: %s\n", pass->pw_passwd);
        printf("User id: %llu\n", (unsigned long long)pass->pw_uid);
        printf("Group id: %llu\n", (unsigned long long)pass->pw_gid);
        printf("Gecos: %s\n", pass->pw_gecos);
        printf("Current Working Directory: %s\n", pass->pw_dir);
        printf("Login Program: %s\n", pass->pw_shell);

        return 0;
}

void exit_sys(const char *msg)
{
        perror(msg);

        exit(EXIT_FAILURE);
}

/*----------------------------------------------------------------------------------------------------
-------------------------
    getpwuid fonksiyonu da getpwnam fonksiyonu gibidir. Yalnızca kullanıcı ismi ile değil kullanıcı id'si ile kullanıcı bilgilerini elde
    etmektedir. Fonksiyonun prototipi şöyledir:

    #include <pwd.h>

    struct passwd *getpwuid(uid_t uid);
```





Fonksiyon yine başarı durumunda statik düzeyde tahsis edilmiş olan struct passwd türünden yapı nesnesinin adresiyle, başarısızlık durumunda
NULL adresle geri dönmektedir. Başarısızlığın nedeni kullanı id'sine ilişkin kullanıcının bulunamaması nedeni ile ise
bu durumda fonksiyon errno değerini değiştirmemektedir.

Aşağıdaki örnekte komut satırından verilen kullanıcı id'sine ilişkin kullanıcı bilgileri ekrana (stdout dosyasına) yazdırılmıştır.

```
----------------------------------------------------------------------------------------------------
-------------------------*/

#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <pwd.h>

void exit_sys(const char *msg);

int main(int argc, char *argv[])
{
        struct passwd *pass;

        if (argc != 2) {
                fprintf(stderr, "wrong number of arguments!...\n");
                exit(EXIT_FAILURE);
        }

        errno = 0;
        if ((pass = getpwuid(atoi(argv[1]))) == NULL) {
                if (errno == 0) {
                        fprintf(stderr, "user name cannot found!...\n");
                        exit(EXIT_FAILURE);
                }
                exit_sys("getpwnam");
        }

        printf("User Name: %s\n", pass->pw_name);
        printf("Password: %s\n", pass->pw_passwd);
        printf("User id: %llu\n", (unsigned long long)pass->pw_uid);
        printf("Group id: %llu\n", (unsigned long long)pass->pw_gid);
        printf("Gecos: %s\n", pass->pw_gecos);
        printf("Current Working Directory: %s\n", pass->pw_dir);
        printf("Login Program: %s\n", pass->pw_shell);

        return 0;
}

void exit_sys(const char *msg)
{
        perror(msg);

        exit(EXIT_FAILURE);
}

/*----------------------------------------------------------------------------------------------------
-------------------------
    Bazen programcı /etc/passwd dosyasındaki tüm kayıtları elde etmek isteyebilir. Bunun için getpwent, endpwent ve setpwend POSIX
    fonksiyonları bulundurulmuştur. Fonksiyonların prototipileri şöyledir:

    #include <pwd.h>

    struct passwd *getpwent(void);
    void setpwent(void);
    void endpwent(void);

    getpwent fonksiyonu her çağrıldığında sıraki bir kullanıcının bilgisini verir. Fonksiyon /etc/passwd dosyasının sonuna gelindiğinde
```





(yani artık bilgisi verilecek kullanıcı kalmadığında) NULL adrese geri döner. Tabii getpwent IO hatası nedeniyle de başarısız olabilir.
Bu durumda errno değerini değiştirmez. Programcı bu sayede başarısızlığın nedenini anlayabilir. İşlem bitince endpwent fonksiyonu
son kez çağrılmalıdır. (Bu fonksiyon arka planda muhtemelen /etc/passwd dosyasını kapatmaktadır.) Eğer dolaşım yeniden yapılacaksa
setpwent fonksiyonu çağrılır. İlk dolaşımda setğwent fonksiyonun çağrılması gerekmemektedir.

Aşağıdaki programda tüm kullanıcı bilgileri bir döngü içerisinde elde edilip ekrana (stdout dosyasına) yazdırılmıştır.

```
----------------------------------------------------------------------------------------------------
-------------------------*/

#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <pwd.h>

void exit_sys(const char *msg);

int main(void)
{
        struct passwd *pass;

        while ((errno = 0, pass = getpwent()) != NULL) {
                printf("User Name: %s\n", pass->pw_name);
                printf("Password: %s\n", pass->pw_passwd);
                printf("User id: %llu\n", (unsigned long long)pass->pw_uid);
                printf("Group id: %llu\n", (unsigned long long)pass->pw_gid);
                printf("Gecos: %s\n", pass->pw_gecos);
                printf("Current Working Directory: %s\n", pass->pw_dir);
                printf("Login Program: %s\n", pass->pw_shell);
                printf("-------------------------------------------------------------\n");
        }

        if (errno != 0)
                exit_sys("getpwent");

        endpwent();

        return 0;
}

void exit_sys(const char *msg)
{
        perror(msg);

        exit(EXIT_FAILURE);
}

/*----------------------------------------------------------------------------------------------------
-------------------------
    Bilindiği gibi pek çok UNIX türevi sistemde grup bilgileri /etc/group isimli bir dosyada tutulmaktadır. (POSIX standartları
    grup bilgilerinin böyle bir dosyada tutulacağına yönelik bir bilgi içermemektedir.) İşte grup bilgilerinin bu dosyadan alınması
    için de benzer bir mekanizma oluşturulmuştur. Aşağıda grup /etc/group dosyasından birkaç satır görüyprsunuz:

    ...
    nm-openvpn:x:133:
    kaan:x:1000:
    sambashare:x:134:kaan
    study:x:1001
    test:x:1002
    ...

    Grup bilgilerini elde etmek için kullanılan POSIX fonksiyonları da şöyledir:
```





```
    #include <grp.h>

    struct group *getgrnam(const char *name);
    struct group *getgrgid(gid_t gid);
    struct group *getgrent(void);
    void setgrent(void);
    void endgrent(void);
```

Bu fonksiyonlardaki struct group yapısı <grp.h> dosyası içerisinde şöyle bildirilmiştir:

```
        struct group {
                char   *gr_name;                /* group name */
                char   *gr_passwd;              /* group password */
                gid_t   gr_gid;                 /* group ID */
                char  **gr_mem;                 /* NULL-terminated array of pointers to names of
group members */
        };
```

Yapının gr_name elemanı grubun ismini belirtmektedir. gr_passwd elemanı grubun parola bilgisini belirtir. Gruplarda da parola
kavramı vardır. Ancak seyrek kullanılmaktadır. gr_gid elemanı grubun numarısını belirtir. Anımsanacağı gibi bir kullanıcı
birdenfazla gruba üye olabilmektedir. Kullanıcının asıl grubu /etc/passwd dosyasında belirtilen grup id'ye ilişkin gruptur.
Örneğin /etc/group dosyasında aşağıdaki gibi bir satır bulunuyor olsun:

```
    study:x:1001:ali,veli,selami
```

Burada grup bilgilerinin sonundaki ali, vel, selami bu study grubuna ek grup olarak dahil edilen kullanıcıları belirtmektedir.
Örneğin kaan kullanıcısının asıl grubu project olabilir. Ancak kaan kullanıcısı aynı zamanda "ek grup (supplementary group)" olarak
study grubuna da dahil olabilir. Yani sistemin bir kullanıcının ek gruplarını elde edebilmesi için /etc/group dosyasını baştan sona gözden geçirip
kullanıcının hangi satırların ':' ayrılmış son bölümünde geçtiğini belirlemesi gerekmektedir. İşte group yapısının gr_mem
elemanı bir göstericiyi gösteren göstericidir ve bu gruba ait olan kullanıcıları belirtmektedir. Tabii bu gr_mem ile belirtilmiş olan
gösterici dizisinin son elemanı NULL adres içermektedir.

getgrnam fonksiyonu grubun isminden hareketle grup bilgilerini, getgrgid fonksiyonu ise grup id'sinden hareketle grup bilgilerini vermektedir.
Tıpkı kullanıcı bilgilerinde olduğu gibi grup bilgilerinin de tek tek elde edilmesi benzer biçimde get_grent, endgrent ve setgrent
fonksiyonlarıyla yapılmaktadır.

Aşağıdaki örnekte tüm gruplara ilişkin grup bilgileri ekrana (stdout dosyasına) yazdırılmıştır.

```
----------------------------------------------------------------------------------------------------
-------------------------*/

#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <grp.h>

void exit_sys(const char *msg);

int main(void)
{
        struct group *grp;

        while ((errno = 0, grp = getgrent()) != NULL) {
                printf("Group name: %s\n", grp->gr_name);
                printf("Password: %s\n", grp->gr_passwd);
                printf("Group id: %llu\n", (unsigned long long)grp->gr_gid);
                printf("Supplemenray userf of this group: ");
                for (int i = 0; grp->gr_mem[i] != NULL; ++i) {
                        if (i != 0)
```

```
                                printf(", ");
                        printf("%s", grp->gr_mem[i]);
                }
                printf("\n-----------------------------------------------------\n");
        }

        if (errno != 0)
                exit_sys("getgrent");

        endgrent();

        return 0;
}

void exit_sys(const char *msg)
{
        perror(msg);

        exit(EXIT_FAILURE);
}

/*--------------------------------------------------------------------------------------------------
-------------------------

16. Ders 17/12/2022 - Cumartesi
--------------------------------------------------------------------------------------------------
-------------------------*/

/*--------------------------------------------------------------------------------------------------
-------------------------
    Daha önce ls -l komutu formatında dosya bilgilerini yazdıran bir örnek yapmıştık. Ancak o
örnekte kullanıcı ve grup isimleri
    isim olarak değil kullanıcı ve grup id'leri olarak ekrana (stdout dosyasına) yazdırılmıştı.
Şimdi artık getpwuid ve getgrgid
    fonksiyonları ile bu sayısal id değerlerinden kullanıcı ve grup isimlerini elde edebiliriz.

    Aşağıda ls -l komutunun düzeltilmiş hali verilmektedir.
--------------------------------------------------------------------------------------------------
-------------------------*/

#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <sys/stat.h>
#include <pwd.h>
#include <grp.h>

#define LS_BUFSIZE              4096

void exit_sys(const char *msg);
char *get_ls(struct stat *finfo, const char *name);

int main(int argc, char *argv[])
{
        struct stat finfo;

        if (argc == 1) {
                fprintf(stderr, "file(s) must be specified!\n");
                exit(EXIT_FAILURE);
        }

        for (int i = 1; i < argc; ++i) {
                if (lstat(argv[i], &finfo) == -1)
                        exit_sys("stat");
                printf("%s\n", get_ls(&finfo, argv[i]));
        }

        return 0;
}
```





```
void exit_sys(const char *msg)
{
        perror(msg);

        exit(EXIT_FAILURE);
}

char *get_ls(struct stat *finfo, const char *name)
{
        static char buf[LS_BUFSIZE];
        static mode_t modes[] = {S_IRUSR, S_IWUSR, S_IXUSR, S_IRGRP, S_IWGRP, S_IXGRP, S_IROTH,
S_IWOTH, S_IXOTH};
        static mode_t ftypes[] = {S_IFBLK, S_IFCHR, S_IFIFO, S_IFREG, S_IFDIR, S_IFLNK, S_IFSOCK};
        int index;
        struct tm *ptime;
        struct passwd *pw;
        struct group *gr;

        pw = getpwuid(finfo->st_uid);
        gr = getgrgid(finfo->st_gid);

        index = 0;
        for (int i = 0; i < 7; ++i)
                if ((finfo->st_mode & S_IFMT) == ftypes[i]) {
                        buf[index++] = "bcp-dls"[i];
                        break;
                }

        for (int i = 0; i < 9; ++i)
                buf[index++] = finfo->st_mode & modes[i] ? "rwx"[i % 3] : '-';

        ptime = localtime(&finfo->st_mtim.tv_sec);

        index += sprintf(buf + index, " %llu", (unsigned long long)finfo->st_nlink);
        if (pw == NULL)
                index += sprintf(buf + index, " %llu", (unsigned long long)finfo->st_uid);
        else
                index += sprintf(buf + index, " %s", pw->pw_name);
        if (gr == NULL)
                index += sprintf(buf + index, " %llu", (unsigned long long)finfo->st_gid);
        else
                index += sprintf(buf + index, " %s", gr->gr_name);

        index += sprintf(buf + index, " %llu", (unsigned long long)finfo->st_size);
        index += strftime(buf + index, LS_BUFSIZE, " %b %2e %H:%M", ptime);

        sprintf(buf + index, " %s", name);

        return buf;
}

/*--------------------------------------------------------------------------------------------------
-------------------------
    Anımsanacağı gibi dizinler (directories) de aslında tamamen dosyalar gibi organize ediliyordu.
Dizinlerin içerisinde aşağıdaki gibi
    dizin girişleri bulunyordu:

    isim    i-node_no
    isim    i-node_no
    isim    i-node_no
    ...

    Dizin dosyalarının gerçek formatları biraz daha detay içerebilmektedir. Kursumuzun sonlarında
doğru Ext-2 dosya sisteminin disk organizasyonu
    üzerinde duracağız.

    Bir dizini erişim hakları yeterliyse open fonlsiyonuyla açabiliriz. Ancak POSIX
standartlarında dizin dosyalarından okuma, yazma ve konumlandırma işlemlerinin
```





```
    yapılıp yapılamayacağı işletim sistemini yazanların isteğine bırakılmıştır. Linux, BSD, macOS
gibi sistemler dizin dosyalarından read ve write fonksiyonları ile
    okuma ve yazma yapmaya izin vermemektedir. Ancak bu sistemler lseek fonksiyonuyla dizin
dosyalarının dosya göstericilerinin konumlandırılmasına izin vermektedir.
    Pekiyi mademki işletim sistemleri dizin dosyalarından okuma yazma yapmaya izin
vermeyebiliyorlar, bu durumda open fonksiyonuyla dizin dosyalarını hangi modda
    açabiliriz? İşte bunun POSIX standartlarında O_SEARCH isimli bir mod bulunmaktadır. Bu mod
aslında ileride ele alacağımız at'li POSIX
    fonksiyonları için düşünülmüştür. Eğer O_SEARCH modunda bir dizin açılırsa bu dizinden okuma
yazma yapılamaz ancak bu at'li fonksiyonlar kullanılabilir.
    Ancak O_SEARCH modu Linux tarafıdan desteklenmemektedir. Bu durumda mecburen Linux'ta bir
dizini açacaksak işletim sistemi read fonksiyonu ile okuma yapılmasına
    izin vermiyor olsa da biz açış modu olarak O_RDONLY kullanırız.

    Pekiyi bir dizini O_SEARCH modunda açmak ile O_RDONLY modunda açmak arasında ne fark vardır?
O_SEARCH modu POSIX standartlarına bir dizin üzerinde "read", "write"
    yapmamak ancak başka işlemlerde kullanılmak amacıyla kullanılmak için eklenmiştir. Dolayısıyla
bir işletim sistemi örneğin dizin dosyalarından read fonksiyonu ile
    okuma yapmaya izin veriyorsa bu durumda biz o dizini O_SEARCH modunda açarsak okuma yapamayız.
Ancak O_RDONLY modunda açarsak okuma yapabiliriz.
    Linux ve macOS O_SEARCH modunu desteklememektedir. Ancak BSD türevi sistemler bu modu
desteklemektedir.

    Aşağıdaki Linux sistemlerinde bir dizin'inin open fonksiyonuyla açılmasına örnek verdik. Linux
açış modu olarak O_SREACH modunu
    desteklemediği için O_RDONLY modunu kullandık. İşletim sistemleri genel olarak dizinlerin
write modda açılmasına izin vermemektedir.
--------------------------------------------------------------------------------------------------
-------------------------*/

#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <unistd.h>

void exit_sys(const char *msg);

int main(void)
{
        int fd;

        if ((fd = open(".", O_RDONLY)) == -1)
                exit_sys("open");

        printf("Ok\n");

        close(fd);

        return 0;
}

void exit_sys(const char *msg)
{
        perror(msg);

        exit(EXIT_FAILURE);
}

/*--------------------------------------------------------------------------------------------------
-------------------------
    Pekiyi mademki işletim sistemlerinin çoğu bir dizin üzerinde read ve write fonksiyonları ile
işlem yapmaya izin vermiyorsa bu durumda
    bir dizini open fonksiyonu ile açmanın ne anlamı vardır? İşte anımsanacağı gibi yol ifadesi
alan POSIX dosya fonksiyonlarının başı f ile başlayan
    dosya betimleyicisi alan biçimleri de vardı. Örneğin stat ve lstat fonksiyonları yol ifadesi
alırken fstat fonksiyonu dosya betimleyicisi
    alıyordu. Benzer biçimde chmod için fchmod, chown fchown fonksiyonları bulunmaktaydı. İşte bu
f'li fonksiyonların bir de at'li versiyonları vardır.
    Örneğin fstatat, fchmodat, fchownat gibi. Ayrıca başı f ile başlamayan çeşitli dosya
```





```
fonksiyonlarının da at'li versiyonarı bulunmaktadır.
    Örneğin open fonksiyonun da bir at'li versiyonu vardır. Aslında at'li versiyonlar seyrek
kullanılan fonksiyonlardır. Ancak biz kursumuzda bunlar
    hakkında açıklama yapmayı da uygun görüyoruz. Pekiyi bu at'li fonksiyonlar ne yapmaktadır.
Aşağıda openat fonksiyonunun prototipini
    görüyorsunuz:

    #include <fcntl.h>

    int openat(int fd, const char *path, int oflag, ...);

    Fonksiyonun prototipini open fonksiyonu ile karşılaştırınız:

    int open(const char *path, int oflag, ...);

    Fonksiyonların at'li vesiyonları genel olarak bir dosya betimleyicisi de almaktadır. Bu dosya
betimleyicisinin bir dizin'e ilişkin olması
    gerekir. Eğer bu dosya betimleyicisi bir dizin'e ilişkin değilse fonksiyon başarısız olur.
at'li versiyonlara bir dizine ilişkin dosya betimleyicisi
    verdikten sonra ayrıca bu fonksiyonlar bir de yol ifadesi de alırlar. Buradaki yol ifadesi
eğer mutlak (absolute) ise bu at'li versiyonların at'siz
    versiyonlardan (flag parametreleri dışında) hiçbir farkı kalmaz. Dolayısıyla bu durumda
geçerli olsa da at'li versiyonları kullanmanın anlamı kalmamaktadır.
    (Bazı at'li versiyonlar flag parametresine de sahiptir. Bu parametrenin işlevinden faydalanmak
için de at'li fonksiyonlar kullanılabilmektedir.)
    Yani fonksiyon bu dizin betimleyicisinden faydalanmamaktadır. Ancak yol ifadesi göreli
(relative) ise bu durumda dosya prosesin çalışma dizininden itibaren değil
    dizin betimleyicisinin belirttiği dizinden itibaren orijin belirtmektedir. Yani biz at'li
versiyonlarla göreli yol ifadelerinin orijinlerini prosesin
    çalışma dizinin dışında başka bir dizine kaydırabilmekteyiz. Tabii fonksiyonların at'li
versiyonları kullanılacaksa bu durumda dizin dosyalarının
    O_SEARCH modunda açılması daha uygundur. Çünkü bu at'li versiyonlar için dizin dosyalarının
okuma modunda açılması gerekmemektedir. Zaten POSIX'te
    O_SEARCH modu bu at'li fonksiyonlar için bulundurulmuştur. Linux ve macOS sistemleri O_SEARCH
modunu desteklemediğine göre bu sistemlerde
    at'li fonksiyonları kullanırken dizin'leri O_RDONLY modda açmamız gerekir. POSIX
standartlarına göre at'li fonksiyonlarda eğer dizin O_SEARCH modunda
    açılmışsa belirtilen dizinin "x" hakkına sahilik kontrolü yapılmaz. Eğer dizin O_SERACH yerine
diğer modlarla (örneğin O_RDONLY) açılmışsa
    bu durumda belirtilen dizinde "x" hakkı kontrolü yapılmaktadır. Ayrıca fonksiyonların at'li
versiyonlarında dizine ilişkin dosya betimleyicisine özel olarak
    AT_FDCWD değeri geçirilirse bu durumda sanki prosesin çalışma dizinine ilişkin dizin
betimleyicisi geçirilmiş gibi bir etki oluşmaktadır.
    Tabii bu durumda fonksiyonun at'li versiyonu ile at'siz versyonu arasında bir fark kalmaz.
Ancak fonksiyonaın at'li versiyonlarının ekstra parametreleri de
    olabilmektedir (genellikle bu ekstra parametre flag parametresi biçimindedir). İşte programcı
bu ekstra parametrelerden faydalanabilmek için dosya
    betimleyici parametresini AT_FDCWD biçiminde geçebilmektedir. Ayrıca fonksiyonların at'li
versiyonlarında biz yol ifadesi olarak mutlak ifadesi geçtiğimizde
    fonksiyonun dizin betimleyici parametresi zaten hiç kontrol edilmemektedir (yani bu parametre
geçersiz bir betimleyici belirtse bile eğer yol ifadesi mutlak ise
    fonksiyon için bir sorun oluşturmamaktadır.)

    Diğer at'li bazı fonksiyonların da prototipleri şöyledir:

    int fchmodat(int fd, const char *path, mode_t mode, int flag);
    int fchownat(int fd, const char *path, uid_t owner, gid_t group, int flag);
    int fstatat(int fd, const char *restrict path, struct stat *restrict buf, int flag);

    Aşağıda openat fonksiyonun kullanımına bir örnek verilmiştir. Burada çalılşma dizininde
"test.txt" dosyası bulunduğu halde fonksiyon
    başarısız olacaktır.
--------------------------------------------------------------------------------------------------
-------------------------*/

#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <unistd.h>
```

```
void exit_sys(const char *msg);

int main(void)
{
        int fddir, fd;

        if ((fddir = open("/usr/include", O_RDONLY)) == -1)
                exit_sys("open");

        if ((fd = openat(fddir, "test.txt", O_RDONLY)) == -1)
                exit_sys("openat");

        printf("Ok\n");

        close(fd);
        close(fddir);

        return 0;
}

void exit_sys(const char *msg)
{
        perror(msg);

        exit(EXIT_FAILURE);
}

/*----------------------------------------------------------------------------------------------------------
    Bir dizin dosyası içerisindeki "dizin girişlerini (directory entry)" elde etmek için bir grup
POSIX fonksiyonu bulundurulmuştur.
    (Dizin dosyalarının open ya da openat ile fonksiyonu ile açılabildiğine ancak pek çok sistemde
read fonksiyonu ile okunamadığına dikkat ediniz.
    Ayrıca dizin dosyalarının iç formatı dosya sisteminden dosya sistemine değişebilmektedir. Bu
nedenle POSIX standartlarında bu işi yapan ayrı
    fonksiyonlar bulundurulmuştur.)

    Linux sistemlerinde dizin girişlerinin okunması için getdents isimli bir sistem fonksiyonu
bulundurulmuştur. Dolayısıyla aşağıda açıklayacağımız
    POSIX fonksiyonları arka planda Linuz sistemlerinde getdents sistem fonksiyonunu
çağırmaktadır.
----------------------------------------------------------------------------------------------------------
----------------------------*/

/*----------------------------------------------------------------------------------------------------------
    Dizin girişlerini elde etmek için önce dizin'in opendir fonksiyonuyla açılması gerekmektedir.
Bunun için dizin'e okuma hakkının
    bulunuyor olması gerekmektedir. opendir fonksiyonunun prototipi şöyledir:

    #include <dirent.h>

    DIR *opendir(const char *dirname);

    Fonksiyon parametre olarak açılacak dizin'in yol ifadesini almaktadır. Fonksiyonun geri dönüş
değeri DIR isimli bir yapı türünden (DIR bir typedef ismidir)
    bir adrestir. Bu DIR adresi bir handle gibi kullanılmaktadır. Fonksiyon başarısızlık durumunda
NULL adrese geri döner ve errno uygun biçimde değer
    alır. opendir fonksiyonun fdopendir isimli bir versiyonu da vardır. Bu versiyon eğer zateb
dizin O_SEARCH ya da O_RDONLY modda açılmışsa o dizine ilişkin betimleyici
    yoluyla DIR adresini vermektedir:

    #include <dirent.h>

    DIR *fdopendir(int fd);

    Dizin opendir ya da fdopendir fonksiyonuyla açılıp, DIR handle'ı elde edildikten sonra, artık
dizin girişleri readdir POSIX
```





```
    fonksiyonuyla tek tek bir döngü içerisinde okunabilir. readdir fonksiyonu her çağrıldığında
bir sonraki dizin girişi elde edilir.
    Fonksiyonun prototipi şöyledir:

    #include <dirent.h>

    struct dirent *readdir(DIR *dirp);

    Fonksiyon parametre olarak DIR yapısının adresini alır sıradaki dizin girişini elde eder. Bu
dizin girişinin bilgilerini statik
    ömürlü struct dirent isimli bir yapı nesnesinin içerisine yerleştirir. Bize de onun adresini
verir. Eğer readdir dizin listesinin
    sonuna gelirse NULL adrese geri dönmektedir. Ancak fonksiyon IO hatalarından dolayı da
başarısız olabilir. Bu durumda başarısızlığın
    dizin sonuna gelmekten dolayı mı yoksa IO hatalarından dolayı mı olduğunu anlamak gerekebilir.
İşte readdir fonksiyonu eğer dizin sonuna
    gelindiğinden dolayı NULL adrese geri dönmüş ise bu durumda errno değişkeninin değerini
değiştirmemektedir. O halde programcı fonksiyonu çağırmadan
    önce rrno değişkenine 0 atamalı sonra fonksiyonu çağırmalıdır. Eğer fonksiyon NULL adrese geri
dönmüşse errno değişkenine bakmalı eğer errno
    hala 0 ise fonksiyonun dizin sonuna gelindiğinden dolayı başarısız olduğu sonucunu
çıkarmalıdır. O halde fonksiyon tipik olarak şöyle kullanılmalıdır:

    struct dirent *de;
    ...
    while (errno = 0, (de = readdir(dir)) != NULL) {
        /* ... */
    }
    if (errno != 0)
        exit_sys("readdir");

    dirent yapısı POSIX standartlarına göre en az iki elemana sahip olmak zorundadır. Bu elemanlar
d_ino ve d_name elemanlarıdır.
    d_ino elemanı ino_t türündendir. d_name elemanı ise char türden bir dizidir. Ancak işletim
sistemleri genellikle bu dirent yapısında
    daha fazla eleman bulundurmaktadır. Örneğin Linux'taki dirent yapısı şöyledir:

    struct dirent {
        ino_t          d_ino;       /* Inode number */
        off_t          d_off;       /* Not an offset; see below */
        unsigned short d_reclen;    /* Length of this record */
        unsigned char  d_type;      /* Type of file; not supported
                                       by all filesystem types */
        char           d_name[256]; /* Null-terminated filename */
    };

    Görüldüğü gibi Linux'ta yapının içerisinde d_off, d_reclen ve d_type elemanları da
bulunmaktadır. d_off ve d_reclen elemanları
    önemli değildir. Ancak d_type elemanı dosyanın ne dosyası olduğunu belirtmektedir. Bu eleman
sayesinde programcı dosyanın türünü
    anlamak için stat fonksiyonlarını çağırmak zorunda kalmaz. Gerçekten de i-node tabanlı dosya
sistemleri dizin girişlerinde dosyanın
    türünü de zaten tutmaktadır. Ancak POSIX standartlarında bu elemanlar zorunlu olarak
belirtilmediğinden taşınabilir programlarda
    yalnızca yapının d_ino ve d_name elemanları kullanılmalıdır.

    dirent yapısının d_ino elemanı bize dosyanın i-node numarasını verir. d_name elemanı ise dizin
girişinin ismini vermektedir.
    Linux sistemlerinde d_type bit düzeyinde kodlanmamıştır. Aşağıdaki değerlerden birine eşit
olmak zorundadır:

    DT_BLK      block device
    DT_CHR      character device
    DT_DIR      directory
    DT_FIFO     named pipe (FIFO)
    DT_LNK      symbolic link
    DT_REG      regular file.
    DT_SOCK     UNIX domain socket.
    DT_UNKNOWN  Bilinmeyen bir tür
```





```
    readdir ile dizin girişleri dosya sistemindeki kayıtlara göre verilmektedir. Halbuki ls komutu
default durumda önce dizin girişlerini
    isme göre sıraya dizmekte sonra onları göstermektedir. (Linux'ta -f'den sonra -l'yi
kullanınız, ters sırada çalışmıyor.) Doğal sıranın
    ne anlam ifade ettiği dosya sistemlerinin anlatıldığı bölümde ele alımacaktır.

    Dizin girişleri elde edildikten sonra dizin closedir POSIX fonksiyonuyla kapatılmaldır:

    #include <dirent.h>

    int closedir(DIR *dirp);

    Fonksiyon başarı durumunda 0, başarısızlık durumunda -1 değerine geri dönmektedir.

    closedir fonksiyonu kendi içerisinde kullandığı betimleyicileri close etmektedir. Örneğin biz
DIR nesnesini (directory stream)
    fdopendir ile dizin betimleyicisini vererek yaratmış olalım. closedir bu betimleyiciyi kendisi
close etmektedir.
----------------------------------------------------------------------------------------------------------
----------------------------*/

#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <dirent.h>

void exit_sys(const char *msg);

int main(int argc, char *argv[])
{
        DIR *dir;
        struct dirent *de;

        if (argc != 2) {
                fprintf(stderr, "wrong number of arguments!...\n");
                exit(EXIT_FAILURE);
        }

        if ((dir = opendir(argv[1])) == NULL)
                exit_sys("opendir");

        while (errno = 0, (de = readdir(dir)) != NULL)
                printf("%s\n", de->d_name);

        if (errno != 0)
                exit_sys("readdir");

        closedir(dir);

        return 0;
}

void exit_sys(const char *msg)
{
        perror(msg);

        exit(EXIT_FAILURE);
}

/*----------------------------------------------------------------------------------------------------------
    Aşağıdaki örnekte bir dizindeki dosyaların hepsini ls -l stili ile yazdırıyoruz. Bu örnekte
bazı noktalara dikkat ediniz:

    - Biz argv[1] ile görüntülenecek dizini alıyoruz. Ancak bu dizindeki dosyaların stat bilgileri
elde edilirken yok ifadesinin
    dosya isminin başına eklenmesi gerekmektedir:
```





```
    while (errno = 0, (de = readdir(dir)) != NULL) {
        sprintf(path, "%s/%s", argv[1], de->d_name);
        if (lstat(path, &finfo) == -1)
            exit_sys("stat");

        printf("%s\n", get_ls(&finfo, de->d_name));
    }

    Aslında bu tür durumlarda fonksiyonların at'li versiyonlarını kullanmak daha uygun
olabilmektedir.

    - Biz burada bir hizalama yapmadık. Halbuki orijinal ls -l komutu yazısal sütnları karakter
sayısına göre hizalayıp sola
    dayalı olarak, sayısal sütunları ise hizalayıp sağa dayalı olarak yazdırmaktadır. Tabii bunun
için dütnun en geniş elemanının
    bulunması da gerekmektedir. Bu işlem "çalışma sorusu" olarak sorulacaktır.

    - Biz bu örnekte dizin girişlerini doğal sıraya göre görüntüledik. Halbuki ls -l komutu önce
onları isme göre sıraya dizip
    sonra görüntülemektedir.
----------------------------------------------------------------------------------------------------------
----------------------------*/

#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <errno.h>
#include <sys/stat.h>
#include <dirent.h>
#include <pwd.h>
#include <grp.h>

#define LS_BUFSIZE              4096

void exit_sys(const char *msg);
char *get_ls(struct stat *finfo, const char *name);

int main(int argc, char *argv[])
{
        struct stat finfo;
        DIR *dir;
        struct dirent *de;
        char path[4096];

        if (argc != 2) {
                fprintf(stderr, "wrong number of arguments!...\n");
                exit(EXIT_FAILURE);
        }

        if ((dir = opendir(argv[1])) == NULL)
                exit_sys("opendir");

        while (errno = 0, (de = readdir(dir)) != NULL) {
                sprintf(path, "%s/%s", argv[1], de->d_name);
                if (lstat(path, &finfo) == -1)
                        exit_sys("stat");

                printf("%s\n", get_ls(&finfo, de->d_name));
        }
        if (errno != 0)
                exit_sys("readdir");

        closedir(dir);

        return 0;
}

void exit_sys(const char *msg)
{
```

```
        perror(msg);

        exit(EXIT_FAILURE);
}

char *get_ls(struct stat *finfo, const char *name)
{
        static char buf[LS_BUFSIZE];
        static mode_t modes[] = {S_IRUSR, S_IWUSR, S_IXUSR, S_IRGRP, S_IWGRP, S_IXGRP, S_IROTH,
S_IWOTH, S_IXOTH};
        static mode_t ftypes[] = {S_IFBLK, S_IFCHR, S_IFIFO, S_IFREG, S_IFDIR, S_IFLNK, S_IFSOCK};
        int index;
        struct tm *ptime;
        struct passwd *pw;
        struct group *gr;

        pw = getpwuid(finfo->st_uid);
        gr = getgrgid(finfo->st_gid);

        index = 0;
        for (int i = 0; i < 7; ++i)
                if ((finfo->st_mode & S_IFMT) == ftypes[i]) {
                        buf[index++] = "bcp-dls"[i];
                        break;
                }

        for (int i = 0; i < 9; ++i)
                buf[index++] = finfo->st_mode & modes[i] ? "rwx"[i % 3] : '-';

        ptime = localtime(&finfo->st_mtim.tv_sec);

        index += sprintf(buf + index, " %llu", (unsigned long long)finfo->st_nlink);
        if (pw == NULL)
                index += sprintf(buf + index, " %llu", (unsigned long long)finfo->st_uid);
        else
                index += sprintf(buf + index, " %s", pw->pw_name);
        if (gr == NULL)
                index += sprintf(buf + index, " %llu", (unsigned long long)finfo->st_gid);
        else
                index += sprintf(buf + index, " %s", gr->gr_name);

        index += sprintf(buf + index, " %llu", (unsigned long long)finfo->st_size);
        index += strftime(buf + index, LS_BUFSIZE, " %b %2e %H:%M", ptime);

        sprintf(buf + index, " %s", name);

        return buf;
}

/*--------------------------------------------------------------------------------------------
-------------------------
    Yukarıda da belirttiğimiz üzere aslında opendir, readdir, closedir gibi POSIX fonksiyonları
arka planda işletim sisteminin sistem fonksiyonlarını
    çağırmaktadır. Örneğin Linux'ta aslında işletim sistemi düzeyinde işlemler önce sys_open
sistem fonksiyonu ile dizin'in açılması
    sonra sys_getdents sistem fonksiyonu ile dizin girişlerinin okunması ve nihayet sys_close
fonksiyonu ile dizin'in kapatılması yoluyla
    yapılmaktadır. Ancak POSIX standartlarında bu işlemler taşınabilir biçimde opendir, readdir ve
closedir fonksiyonlarına devredilmiştir.
    Şüphesiz bu fonksiyonlar asında dizini açıp onun betimleyicisini DIR yapısının içerisinde
saklamaktadır. İşte elimizde DIR yapısı
    varsa biz de açık dizin'in betimleyicisini elde etmek isttyorsak bunun için dirfd isimli POSIX
fonksiyonundan faydalanabiliriz:

    #include <dirent.h>

    int dirfd(DIR *dirp);

    Fonksiyon parametre olarak DIR yapısının adresini alır, geri dönüş değeri olarak dizine
```





```
ilişkin betimleyiciyi verir. Fonksiyon başarısızlık durumunda
    -1 değerine geri dönmektedir.

    Yukarıdaki örneği fstatat fonksiyonunu kullanarak basitleştirebiliriz. fstatat fonksiyonunun
prototipi şöyledir:

    #include <sys/stat.h>

    int fstatat(int fd, const char *restrict path, struct stat *restrict buf, int flag);

    Fonksiyonun fd parametresinin yanı sıra aynı zamanda bir flag parametresinin olduğuna dikkat
ediniz. Bu parametre stat semantiğinin mi yoksa
    lstat semantiğinin mi uygulanacağını belirtmektedir. Eğer bu parametreye 0 geçilirse bu
durumda stat semantiği uygulanır. Eğer bu parametreye
    AT_SYMLINK_NOFOLLOW değeri geçilirse bu durumda lstat semantiği uygulanmaktadır.
AT_SYMLINK_NOFOLLOW sembolik sabiti <sys/stat.h> içerisinde değil
    <fcntl.h> içerisinde bildirilmiştir. İşte biz yukarıdaki örnekte önce dizin'in betimleyicisini
dirfd fonksiyonu ile alıp bunu fstatat
    fonksiyonunda kullanırsak yol ifadesini düzenlememize gerek kalmaz. Aşağıdaki örnekte bu çözüm
verilmiştir.
--------------------------------------------------------------------------------------------
-------------------------*/

#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <errno.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <dirent.h>
#include <pwd.h>
#include <grp.h>

#define LS_BUFSIZE              4096

void exit_sys(const char *msg);
char *get_ls(struct stat *finfo, const char *name);

int main(int argc, char *argv[])
{
        struct stat finfo;
        DIR *dir;
        struct dirent *de;
        int fd;

        if (argc != 2) {
                fprintf(stderr, "wrong number of arguments!...\n");
                exit(EXIT_FAILURE);
        }

        if ((dir = opendir(argv[1])) == NULL)
                exit_sys("opendir");

        if ((fd = dirfd(dir)) == -1)
                exit_sys("dirfd");

        while (errno = 0, (de = readdir(dir)) != NULL) {
                if (fstatat(fd, de->d_name, &finfo, AT_SYMLINK_NOFOLLOW) == -1)
                        exit_sys("stat");

                printf("%s\n", get_ls(&finfo, de->d_name));
        }
        if (errno != 0)
                exit_sys("readdir");

        closedir(dir);

        return 0;
}
```





```
void exit_sys(const char *msg)
{
        perror(msg);

        exit(EXIT_FAILURE);
}

char *get_ls(struct stat *finfo, const char *name)
{
        static char buf[LS_BUFSIZE];
        static mode_t modes[] = {S_IRUSR, S_IWUSR, S_IXUSR, S_IRGRP, S_IWGRP, S_IXGRP, S_IROTH,
S_IWOTH, S_IXOTH};
        static mode_t ftypes[] = {S_IFBLK, S_IFCHR, S_IFIFO, S_IFREG, S_IFDIR, S_IFLNK, S_IFSOCK};
        int index;
        struct tm *ptime;
        struct passwd *pw;
        struct group *gr;

        pw = getpwuid(finfo->st_uid);
        gr = getgrgid(finfo->st_gid);

        index = 0;
        for (int i = 0; i < 7; ++i)
                if ((finfo->st_mode & S_IFMT) == ftypes[i]) {
                        buf[index++] = "bcp-dls"[i];
                        break;
                }

        for (int i = 0; i < 9; ++i)
                buf[index++] = finfo->st_mode & modes[i] ? "rwx"[i % 3] : '-';

        ptime = localtime(&finfo->st_mtim.tv_sec);

        index += sprintf(buf + index, " %llu", (unsigned long long)finfo->st_nlink);
        if (pw == NULL)
                index += sprintf(buf + index, " %llu", (unsigned long long)finfo->st_uid);
        else
                index += sprintf(buf + index, " %s", pw->pw_name);
        if (gr == NULL)
                index += sprintf(buf + index, " %llu", (unsigned long long)finfo->st_gid);
        else
                index += sprintf(buf + index, " %s", gr->gr_name);

        index += sprintf(buf + index, " %llu", (unsigned long long)finfo->st_size);
        index += strftime(buf + index, LS_BUFSIZE, " %b %2e %H:%M", ptime);

        sprintf(buf + index, " %s", name);

        return buf;
}

/*--------------------------------------------------------------------------------------------
-------------------------

17. Ders 18/12/2022 - Pazar
--------------------------------------------------------------------------------------------
-------------------------*/

/*--------------------------------------------------------------------------------------------
-------------------------
    rewinddir isimli POSIX fonksiyonu dolaşımı yeniden başlatmak amacıyla kullanılır. Yani bu
işlem adeta dosya göstericisinin
    dizin dosyasının başına çekilmesi işlemi gibidir.

    Aşağıdaki örnekte dizin girişleri rewindir fonksiyonu ile iki kez elde edilmiştir.
--------------------------------------------------------------------------------------------
-------------------------*/
```





```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <dirent.h>

void exit_sys(const char *msg);

int main(int argc, char *argv[])
{
        DIR *dir;
        struct dirent *de;

        if (argc != 2) {
                fprintf(stderr, "wrong number of arguments!...\n");
                exit(EXIT_FAILURE);
        }

        if ((dir = opendir(argv[1])) == NULL)
                exit_sys("opendir");

        while (errno = 0, (de = readdir(dir)) != NULL)
                printf("%s\n", de->d_name);

        if (errno != 0)
                exit_sys("readdir");

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

        rewinddir(dir);

        while (errno = 0, (de = readdir(dir)) != NULL)
                printf("%s\n", de->d_name);

        if (errno != 0)
                exit_sys("readdir");

        closedir(dir);

        return 0;
}

void exit_sys(const char *msg)
{
        perror(msg);

        exit(EXIT_FAILURE);
}

/*--------------------------------------------------------------------------------------------
-------------------------
    Dizin girişlerini dolaşırken belli bir noktada dizin dosyasının dosya göstericisinin konumunu
telldir POSIX fonksiyonu
    ile alabiliriz ve o offset'e seekdir POSIX fonksiyonu ile yeniden konumlandırma yapabiliriz.
Fonksiyonların prototipleri şöyledir:

    #include <dirent.h>

    long telldir(DIR *dirp);
    void seekdir(DIR *dirp, long loc);

    Tabii biz belli bir konumu okuduktan sonra kaydedersek bu durumda okumadan dolayı dizin
dosyasının dosya göstericisi ilerletilmiş
    olacaktır. Aşağıdaki örnekte dizin içerisinde "sample.c" dosyası bulunup onun konumu telldir
fonksiyonu ile saklanmıştır. Sonra seekdir
    fonksiyonu ile konuma konumlandırma yapılmıştır. Tabii burada kaydedilen konum "sample.c"
dosyasından sonraki dosyanın konumdur.
--------------------------------------------------------------------------------------------
-------------------------*/
```

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>
#include <dirent.h>

void exit_sys(const char *msg);

int main(int argc, char *argv[])
{
        DIR *dir;
        struct dirent *de;
        long loc;

        if (argc != 2) {
                fprintf(stderr, "wrong number of arguments!...\n");
                exit(EXIT_FAILURE);
        }

        if ((dir = opendir(argv[1])) == NULL)
                exit_sys("opendir");

        while (errno = 0, (de = readdir(dir)) != NULL) {
                printf("%s\n", de->d_name);
                if (!strcmp(de->d_name, "sample.c"))
                        loc = telldir(dir);
        }

        if (errno != 0)
                exit_sys("readdir");

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

        seekdir(dir, loc);

        while (errno = 0, (de = readdir(dir)) != NULL)
                printf("%s\n", de->d_name);

        if (errno != 0)
                exit_sys("readdir");

        closedir(dir);

        return 0;
}

void exit_sys(const char *msg)
{
        perror(msg);

        exit(EXIT_FAILURE);
}

/*-------------------------------------------------------------------------------------------------
-------------------------
    Dizin ağacının dolaşılması özyinelemeli bir algoritmayla yapılmaldır. Bu işlem çeşitli
biçimlerde gerçekleştirilebilir.
    En basit gerçekletirimi dolaşılacak ağacın kök yol ifadesini alan özyinelemeli bir fonksiyon
yazmaktır. Bu fonksiyon
    dizin girişlerini tek tek elde eder. Eğer söz konusu dizin girişi bir dizine ilişkinse o
dizinin yol ifadesiyle kendini çağırır.
    Bu algoritmada dikkat edilmesi gereken birkaç nokta vardır:

    1) Dizin girişleri dolaşılırken "." ve ".." dizinleri continue ile geçilmelidir. Aksi takdirde
sonsuz döngü oluşabilir.
    2) stat fonksiyonu yerine lstat fonksiyonu kullanılmalıdır. Çünkü dizin ağacı dolaşılırken
sembolik bağlantı bir dizine ilişkinse
    sembolik bağlantının hedefine gidilmesi özyinelemeyi bozup sonsuz döngülere yol açabilir.
    3) readdir fonksiyonu dizin girişini okuduğunda bize yalnız girişin ismini vermektedir.
```





```
Dolayısıyla lstat fonksiyonu uygulanırken
    prosesin çalışma dizinin uygun olması gerekir. Bunu sağlayabilmek için her dizine geçişte
chdir fonksiyonu ile prosesin çalışma dizinini değiştebiliriz.
    Ya da alternatif olarak mutlak bir yol ifadesi sürekli güncellenebilir. Aslında burada
seçeneklerden biri de fonksiyonların at'li
    biçimlerini kullanmak olabilir.
    4) Her özyineleme bittiğinde opendir ile açılan dizin closedir ile kapatılmalıdır.
    5) Genellikle böylesi fonksiyonlar bir fatal error ile programı sonlandırmamalıdır. chdir
fonksiyonu ile prosesin çalışma dizini
    değiştirilemeyebilir. Ya da örneğin opendir ile biz bir dizini açamayabiliriz. Bu tür
durumlarda hata stderr dosyasına rapor edilip işlemin
    devem ettirilmesi uygun olabilir.
    6) Özyinelemeli dolaşım bittikten sonra prosesin çalışma dizini orijinal halde bırakılmalıdır.

    Aşağıda tipik bir özyinelemeli "depth-first" dolaşım örneği verilmiştir. Ancak burada prosesin
çalışma dizini özyineleme bittikten sonra
    orijinal dizin ile yeniden set edilmemiştir.
-------------------------------------------------------------------------------------------------
------------------------*/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>
#include <sys/stat.h>
#include <unistd.h>
#include <dirent.h>

void walkdir(const char *path);
void exit_sys(const char *msg);

int main(int argc, char *argv[])
{
        if (argc != 2) {
                fprintf(stderr, "wrong number of arguments!...\n");
                exit(EXIT_FAILURE);
        }

        walkdir(argv[1]);

        return 0;
}

void walkdir(const char *path)
{
        DIR *dir;
        struct dirent *de;
        struct stat finfo;

        if ((dir = opendir(path)) == NULL) {
                fprintf(stderr, "cannot read directory: %s\n", path);
                return;
        }

        if (chdir(path) == -1) {
                fprintf(stderr, "directory cannot change: %s\n", path);
                goto EXIT;
        }

        while (errno = 0, (de = readdir(dir)) != NULL) {
                printf("%s\n", de->d_name);
                if (!strcmp(de->d_name, ".") || !strcmp(de->d_name, ".."))
                        continue;
                if (lstat(de->d_name, &finfo) == -1) {
                        fprintf(stderr, "cannot get stat info: %s\n", de->d_name);
                        continue;
                }

                if (S_ISDIR(finfo.st_mode)) {
```





```
                        walkdir(de->d_name);
                        if (chdir("..") == -1) {
                                fprintf(stderr, "directory cannot change: %s\n", path);
                                goto EXIT;
                        }
                }
        }
        if (errno != 0)
                fprintf(stderr, "cannot read directory info: %s\n", path);
EXIT:
        closedir(dir);
}

void exit_sys(const char *msg)
{
        perror(msg);

        exit(EXIT_FAILURE);
}

/*-------------------------------------------------------------------------------------------------
-------------------------
    Özyineleme çağırmada hangi kademede bulunulduğunu belirten bir bilginin de özyinelemeli
fonksiyona parametre yoluyla aktarılması
    faydaları olabilmektedir. Örneğin bu sayede biz ağacı kademeli bir biçimde görüntüleyebiliriz.

    Aşağıdaki örnekte walkdir fonksiyonuna bir kademe bilgisi de eklenmiştir.
-------------------------------------------------------------------------------------------------
------------------------*/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>
#include <sys/stat.h>
#include <unistd.h>
#include <dirent.h>

void walkdir(const char *path, int level);
void exit_sys(const char *msg);

int main(int argc, char *argv[])
{
        if (argc != 2) {
                fprintf(stderr, "wrong number of arguments!...\n");
                exit(EXIT_FAILURE);
        }

        walkdir(argv[1], 0);

        return 0;
}

void walkdir(const char *path, int level)
{
        DIR *dir;
        struct dirent *de;
        struct stat finfo;

        if ((dir = opendir(path)) == NULL) {
                fprintf(stderr, "cannot read directory: %s\n", path);
                return;
        }

        if (chdir(path) == -1) {
                fprintf(stderr, "directory cannot change: %s\n", path);
                goto EXIT;
        }
```





```
        while (errno = 0, (de = readdir(dir)) != NULL) {
                printf("%*s%s\n", level * 4, "", de->d_name);
                if (!strcmp(de->d_name, ".") || !strcmp(de->d_name, ".."))
                        continue;
                if (lstat(de->d_name, &finfo) == -1) {
                        fprintf(stderr, "cannot get stat info: %s\n", de->d_name);
                        continue;
                }

                if (S_ISDIR(finfo.st_mode)) {
                        walkdir(de->d_name, level + 1);
                        if (chdir("..") == -1) {
                                fprintf(stderr, "directory cannot change: %s\n", path);
                                goto EXIT;
                        }
                }
        }
        if (errno != 0)
                fprintf(stderr, "cannot read directory info: %s\n", path);
EXIT:
        closedir(dir);
}

void exit_sys(const char *msg)
{
        perror(msg);

        exit(EXIT_FAILURE);
}

/*-------------------------------------------------------------------------------------------------
-------------------------
    Aslında yukarıdaki walkdir fonksiyonu bir sarma fonksiyonla daha iyi hale getirilebilir. Bu
sayede level parametresi kullanıcıdan gizlenebilir
    ve prosesin çalışma dizini alınıp geri set edilebilir.

    Aşağıdaki örnekte walkdir fonksiyonu asıl özyineleme işlemini yapan walkdir_recur fonksiyonunu
çağırmaktadır. Fonksiyonda kademeli yazım
    için printf fonksiyonu şöyle çağrılmıştır:

    printf("%*s%s\n", level * 4, "", de->d_name);

    Burada * format karakteri level * 4 ile eşleştirilmiştir. İlk %s format karakteriyle de ""
biçiminde boş string eşleşecektir.
    O halde biz yalnızca satırın başında level * 4 kadar boşluk oluşturmuş oluyoruz.
-------------------------------------------------------------------------------------------------
------------------------*/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>
#include <sys/stat.h>
#include <unistd.h>
#include <dirent.h>

void walkdir(const char *path);
void walkdir_recur(const char *path, int level);
void exit_sys(const char *msg);

int main(int argc, char *argv[])
{
        if (argc != 2) {
                fprintf(stderr, "wrong number of arguments!...\n");
                exit(EXIT_FAILURE);
        }

        walkdir(argv[1]);
```

```
        return 0;
}

void walkdir(const char *path)
{
        char cwd[PATH_MAX];

        if (getcwd(cwd, PATH_MAX) == NULL) {
                perror("getcwd");
                return;
        }

        walkdir_recur(path, 0);

        if (chdir(cwd) == -1) {
                perror("chdir");
                return;
        }
}

void walkdir_recur(const char *path, int level)
{
        DIR *dir;
        struct dirent *de;
        struct stat finfo;

        if ((dir = opendir(path)) == NULL) {
                fprintf(stderr, "cannot read directory: %s\n", path);
                return;
        }

        if (chdir(path) == -1) {
                fprintf(stderr, "directory cannot change: %s\n", path);
                goto EXIT;
        }

        while (errno = 0, (de = readdir(dir)) != NULL) {
                printf("%*s%s\n", level * 4, "", de->d_name);
                if (!strcmp(de->d_name, ".") || !strcmp(de->d_name, ".."))
                        continue;
                if (lstat(de->d_name, &finfo) == -1) {
                        fprintf(stderr, "cannot get stat info: %s\n", de->d_name);
                        continue;
                }

                if (S_ISDIR(finfo.st_mode)) {
                        walkdir_recur(de->d_name, level + 1);
                        if (chdir("..") == -1) {
                                fprintf(stderr, "directory cannot change: %s\n", path);
                                goto EXIT;
                        }
                }
        }
        if (errno != 0)
                fprintf(stderr, "cannot read directory info: %s\n", path);
EXIT:
        closedir(dir);
}

void exit_sys(const char *msg)
{
        perror(msg);

        exit(EXIT_FAILURE);
}

*-------------------------------------------------------------------------------------------
-------------------------
    Dizin ağacını dolaşırken her defasında prosesin çalışma dizinini değiştirmek yerine
```





```
fonksiyonların at'li biçimlerinden de
    faydalanabiliriz. Aşağıdaki örnekte özyinelemeli fonksiyona üst dizinin betimleyicisi (fdp) ve
dosyanın ismi geçirilmiştir.
    at'li fonksiyonların eğer yol ifadesi mutlak ise at'siz fonksiyonlar gibi davrandığını
anımsayınız.
-------------------------------------------------------------------------------------------
-------------------------*/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <unistd.h>
#include <dirent.h>

void walkdir(const char *path);
void walkdir_recur(int fddir, const char *fname, int level);
void exit_sys(const char *msg);

int main(int argc, char *argv[])
{
        if (argc != 2) {
                fprintf(stderr, "wrong number of arguments!...\n");
                exit(EXIT_FAILURE);
        }

        walkdir(argv[1]);

        return 0;
}

void walkdir(const char *path)
{
        int fddir;

        if ((fddir = open(path, O_RDONLY)) == -1)
                exit_sys(path);

        walkdir_recur(fddir, path, 0);

        close(fddir);
}

void walkdir_recur(int fdp, const char *fname, int level)
{
        DIR *dir;
        int fdc;
        struct dirent *de;
        struct stat finfo;

        if ((fdc = openat(fdp, fname, O_RDONLY)) == -1) {
                fprintf(stderr, "cannot open file: %s\n", fname);
                return;
        }
        if ((dir = fdopendir(fdc)) == NULL) {
                fprintf(stderr, "cannot read directory: %s\n", fname);
                close(fdp);
                return;
        }

        while (errno = 0, (de = readdir(dir)) != NULL) {
                printf("%*s%s\n", level * 4, "", de->d_name);
                if (!strcmp(de->d_name, ".") || !strcmp(de->d_name, ".."))
                        continue;
                if (fstatat(fdc, de->d_name, &finfo, AT_SYMLINK_NOFOLLOW) == -1) {
                        fprintf(stderr, "cannot get stat info: %s\n", de->d_name);
                        continue;
```





```
                }

                if (S_ISDIR(finfo.st_mode))
                        walkdir_recur(fdc, de->d_name, level + 1);
        }
        if (errno != 0)
                fprintf(stderr, "cannot read directory info: %s\n", fname);
EXIT:
        closedir(dir);
}

void exit_sys(const char *msg)
{
        perror(msg);

        exit(EXIT_FAILURE);
}

/*-------------------------------------------------------------------------------------------
-------------------------
    Dizin ağacını dolaşırken genelleştirme sağlamak için fonksiyon göstericilerinden
faydalanabiliriz. Yani fonksiyonumuz dizin ağacını
    dolaşırken dosya isimlerini ekrana yazdırmak yerine parametresiyle aldığı bir callback
fonksiyonu çağırabilir.

    Aşağıda dizin girişi bulundukça çağrılan bir callback mekanizması örneği verilmiştir. Buradaki
fonksiyonun prototipi şöyledir:

    int walkdir(const char *path, int (*proc)(const char *, const struct stat *, int));

    Fonksiyonun birinci parametresi dolaşılacak dizinin yol ifadesini belirtir. İkinci parametre
callback fonksiyonun adresini almaktadır.
    callback fonksiyonun virinci parametresi bulunan dizin girişinin ismini (tüm yol ifadesi
değil), ikinci parametresi bu dosyanın
    stat bilgilerini belirtmektedir. Üçüncü parametre ise özyinleme için kademe bilgisini
belirtir. Callback fonksiyon 0 ile geri dönerse
    özyineleme devam ettirlir. Ancak sıfır dışı bir değerle geri dönerse özyineleme sonlandırılır
ve walkdir fonksiyonu da bu değerler
    geri döner. Bu durumda walkdir fonksiyonun geri dönüş değeri üç biçimde olabilir:

    -1: POSIX fonksiyonlarından birinin hatayla geri dönmesi
    > 0: Erken sonlanmayı belirtir.
    0: Normal sonlanmayı belirtir.

-------------------------------------------------------------------------------------------
-------------------------*/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>
#include <sys/stat.h>
#include <unistd.h>
#include <dirent.h>

int walkdir(const char *path, int (*proc)(const char *, const struct stat *, int));
int walkdir_recur(const char *path, int level, int (*proc)(const char *, const struct stat *,
int));

int disp(const char *fname, const struct stat *finfo, int level)
{
        printf("%*s%s\n", level * 4, "", fname);

        if (!strcmp(fname, "d.dat"))
                return 1;

        return 0;
}
```





```
int main(int argc, char *argv[])
{
        int result;

        if (argc != 2) {
                fprintf(stderr, "wrong number of arguments!...\n");
                exit(EXIT_FAILURE);
        }

        if ((result = walkdir(argv[1], disp)) == -1) {
                fprintf(stderr, "function terminates problematically!\n");
                exit(EXIT_FAILURE);
        }
        if (result != 0)
                printf("function terminates prematurely with %d code\n", result);
        else
                printf("function terminates normally!...\n");

        return 0;
}

int walkdir(const char *path, int (*proc)(const char *, const struct stat *, int))
{
        char cwd[PATH_MAX];
        int result;

        if (getcwd(cwd, PATH_MAX) == NULL) {
                perror("getcwd");
                return - 1;
        }

        result = walkdir_recur(path, 0, proc);

        if (chdir(cwd) == -1) {
                perror("chdir");
                return -1;
        }

        return result;
}

int walkdir_recur(const char *path, int level, int (*proc)(const char *, const struct stat *,
int))
{
        DIR *dir;
        struct dirent *de;
        struct stat finfo;
        int result = 0;

        if ((dir = opendir(path)) == NULL) {
                fprintf(stderr, "cannot read directory: %s\n", path);
                return -1;
        }

        if (chdir(path) == -1) {
                fprintf(stderr, "directory cannot change: %s\n", path);
                result = -1;
                goto EXIT;
        }

        while (errno = 0, (de = readdir(dir)) != NULL) {
                if (!strcmp(de->d_name, ".") || !strcmp(de->d_name, ".."))
                        continue;
                if (lstat(de->d_name, &finfo) == -1) {
                        fprintf(stderr, "cannot get stat info: %s\n", de->d_name);
                        continue;
                }
                if ((result = proc(de->d_name, &finfo, level)) != 0) {
                        result = -1;
```

```
                        goto EXIT;
                }

                if (S_ISDIR(finfo.st_mode)) {
                        result = walkdir_recur(de->d_name, level + 1, proc);
                        if (chdir("..") == -1) {
                                fprintf(stderr, "directory cannot change: %s\n", path);
                                result = -1;
                                goto EXIT;
                        }
                        if (result != 0)
                                goto EXIT;
                }
        }
        if (errno != 0)
                fprintf(stderr, "cannot read directory info: %s\n", path);
EXIT:
        closedir(dir);

        return result;
}

/*----------------------------------------------------------------------------------------------------------
-------------------------
    scandir bir dizindeki belli koşulları sağlayan girişleri veren biraz karmaşık parametreye
sahip bir POSIX fonksiyonudur.
    Fonksiyonun parametrik yapısı şöyledir:

    #include <dirent.h>

    int alphasort(const struct dirent **d1, const struct dirent **d2);
    int scandir(const char *dir, struct dirent ***namelist, int (*sel)(const struct dirent *),
                int (*compar)(const struct dirent **, const struct dirent **));

    scandir fonksiyonunun birinci parametresi dizin'in yol ifadesini almaktadır. İkinci
parametreye struct dirent türünden
    göstericiyi gösteren bir göstericinin adresi geçirilmelidir. Üçüncü parametre filte işleminde
kullanılacak fonksiyonu belirtmektedir.
    Her dizin girişi bulundukça bu fonksiyon çağrılır. Eğer bu fonksiyon sıfır dışı bir değerle
geri dönerse dizin girişi biriktirilir.
    Bu parametre NULL adres geçilebilir. Bu durumda dizindeki tüm girişler elde edilir. Son
parametre filtrelenen girişlere ilişkin
    gösterici dizisini sort etmek için kullanılacak karşılaştırma fonksiyonunu belirtmektedir. Bu
karşılaştırma fonksiyonunun prototipi şöyle olmalıdır:

    int cmp(const struct **direnet1, const struct **dirent2);

    Fonksiyon tıpkı qsort fonksiyonunda olduğu gibi birinci parametresiyle belirtilmiş olan dizin
girişi ikinci parametresiyle belirtilmiş
    olan dizin girişinden büyükse pozitif herhangi bir değere, küçükse negatif herhangi bir değere
ve eşitse sıfır değerine geri dönmelidir.
    Alfabetik sıralamayı sağlamak amacıyla zaten hazır bir alphasort isimli fonksiyon
bulundurulmuştur.

    scandir fonksiyonu başarı durumunda gösterici dizisine yerleştirilen eleman sayısı ile
başarısızlık durumunda -1 ile geri döner ve errno uygun biçimde değer
    alır.

    scandir fonksiyonu tüm tahisatları malloc fonksiyonunu kullanarak yapmaktadır. Dolayısıyla
programcının tahsis edilen bu alanları kendisinin
    free hale getirmesi gerekmektedir.

    scandir kendi içerisinde her biriktirilecek dizin girişi için malloc fonksiyonu ile bir truct
dirent yapısı tahsis eder, bunların adreslerini
    de yine tahsis ettiği bir gösterici dizisine yerleştirir. Bu gösterici dizisinin adresini de
bizim adresini geçtiğimiz
    göstericiyi gösteren göstericinin içerisine yerleştirmektedir.

    Aşağıdaki örnekte komut argümanı olarak girilen bir dizinde başı 'a' ya da 'A' harfi ile
```





```
başlayan girişler elde edilmiştir.
----------------------------------------------------------------------------------------------------
-------------------------*/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <dirent.h>

void exit_sys(const char *msg);

int myfilter(const struct dirent *de)
{
        return de->d_name[0] == 'a' || de->d_name[0] == 'A';
}

int main(int argc, char *argv[])
{
        int result;
        struct dirent **dents;

        if (argc != 2) {
                fprintf(stderr, "wrong number of arguments!...\n");
                exit(EXIT_FAILURE);
        }

        if ((result = scandir(argv[1], &dents, myfilter, alphasort)) == -1)
                exit_sys("scandir");

        for (int i = 0; i < result; ++i)
                printf("%s\n", dents[i]->d_name);

        for (int i = 0; i < result; ++i)
                free(dents[i]);
        free(dents);

        return 0;
}

void exit_sys(const char *msg)
{
        perror(msg);

        exit(EXIT_FAILURE);
}

/*----------------------------------------------------------------------------------------------------
-------------------------
    scandir fonksiyonun tasarımında bize göre kusurlar vardır. Fonksiyonun dirent yapılarını
biriktirmesi karşılaştırma fonksiyonu yazacak
    kişiler için yük oluşturmaktadır. Buradaki daha doğru tasarım yeni bir yapı bildirip yapının
içerisinde hem dirent bilgilerinin hem de
    stat bilgilerinin bulunması olabilir.

    Aşağıda bir karşılaştırma fonksiyonu yazımı örnek verilmiştir.
----------------------------------------------------------------------------------------------------
-------------------------*/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/stat.h>
#include <dirent.h>

void exit_sys(const char *msg);

int myfilter(const struct dirent *de)
{
```





```
        return de->d_name[0] == 'a' || de->d_name[0] == 'A';
}

int cmp_size(const struct dirent **de1, const struct dirent **de2)
{
        struct stat finfo1, finfo2;

        if (stat((**de1).d_name, &finfo1) == -1)
                exit_sys("stat");

        if (stat((**de2).d_name, &finfo2) == -1)
                exit_sys("stat");

        if (finfo1.st_size > finfo2.st_size)
                return 1;

        if (finfo1.st_size < finfo2.st_size)
                return -1;

        return 0;
}

int main(int argc, char *argv[])
{
        int result;
        struct dirent **dents;

        if (argc != 2) {
                fprintf(stderr, "wrong number of arguments!...\n");
                exit(EXIT_FAILURE);
        }

        if (chdir(argv[1]) == -1)
                exit_sys("chdir");

        if ((result = scandir(argv[1], &dents, myfilter, cmp_size)) == -1)
                exit_sys("scandir");

        for (int i = 0; i < result; ++i)
                printf("%s\n", dents[i]->d_name);

        for (int i = 0; i < result; ++i)
                free(dents[i]);
        free(dents);

        return 0;
}

void exit_sys(const char *msg)
{
        perror(msg);

        exit(EXIT_FAILURE);
}

/*----------------------------------------------------------------------------------------------------
-------------------------

18. Ders 24/12/2022 - Cumartesi
----------------------------------------------------------------------------------------------------
-------------------------*/

/*----------------------------------------------------------------------------------------------------
-------------------------
    Dizin ağacını özyinelemeli biçimde dolaşan ftw (file traverse walk) ve nftw (new file traverse
walk) isimli POSIX fonksiyonları
    bulunmaktadır. Aslında eskiden yalnızca ftw fonksiyonu vardı. Ancak bu fonksiyona bazı
eklemeler yapılıp nftw fonksiyonu oluşturuldu
    ve ftw fonksiyonu "deprecated" yapıldı. Yani bugün hem ftw hem de nftw fonksiyonları bulunuyor
```





```
olsa da nftw fonksiyonun kullanılması
    önerilmektedir. Zaten nftw fonksiyonu işlevsel olarak ftw fonksiyonu kapsamaktadır. nftw
fonksiyonunun prototipi şöyledir:

    #include <ftw.h>

    int nftw(const char *path, int (*fn)(const char *, const struct stat *, int, struct FTW *),
int fd_limit, int flags);

    Linux altında bu fonksiyonu libc kütüphanesi ile kullanırken "feature test macro"
oluşturulmalıdır. Burada başlık dosyalarının
    yukarısında aşağıdaki gibi bir sembolik sabit bulundurmak gerekir:

    #define _XOPEN_SOURCE 500

    Feature test macro kavramından daha sonra bahsedilecektir. Budaraki sayının 500'e eşit ya da
daha büyük olması gerekmektedir.

    Fonksiyonun birinci parametresi özyinelemeli dolaşılacak dizin'in yol ifadesini almaktadır.
İkinci parametre her dizin girişi bulundukça
    çağrılacak "callback" fonksiyonun adresini almaktadır. Buradaki fonksiyonun aşağıdaki
parametrik yapıya sahip olması gerekir:

    int callback(const char *path, const struct stat *finfo, int flag, struct FTW *ftw);

    nftw fonksiyonun üçüncü parametresi kullanılacak maksimum dosya betimleyici sayısını
belirtmektedir. Fonksiyon her derine indikçe
    o dizini opendir fonksiyonu ile açtığı için (bizde öyle yapmıştık) dosya betimleyici
tablosunda bir betimleyici harcamaktadır. Linux'ta
    default durumda prosesin dosya betimleyici tablosunda 1024 tane betimleyici için yer vardır.
Dolayısıyla derine inildikçe bu tabloda
    betimleyici yer kaplayacağından derin ağaçlarda betimleyici sıkıntısı çekilebilir. İşte
fonksiyonun dördüncü parametresi (fd_limit)
    fonksiyonun en fazla kaç betimleyiciyi açık olarak tutacığını belirtmektedir. Programcı bu
parametreye ortalama bir değer girebilir.
    Fonksiyon kendi içerisinde burada belirtilen derinlik aşıldığında özyineleme yaparken üst
dizin'in betimleyicisini kapatıp geri dönüşte
    yeniden açmaktadır. Ayrıca fonksiyonun dokümantasyonunda fonksiyonun her kademe için en fazla
bir tane betimleyici kullanacağı belirtilmiştir.
    Fonksiyonun son parametresi özyinelemeli dolaşım sırasında bazı belirlemeler için
kullanılmaktadır. Bu parametre çeşitli sembolik sabitlerin
    bit düzeyinde OR'lanması ile oluşturulmaktadır. Bu sembolik sabitler şunlardır:

    FTW_CHDIR: Eğer bu bayrak belirtilirse fonksiyon her dizine geçtiğinde prosesin çalışma
dizinini de o dizin olarak değiştirmektedir.

    FTW_DEPTH: Normalde dolaşım "pre-order" biçimde yapılmaktadır. Bu bayrak girilirse "post-
order" dolaşım yapılır. Bayrağın ismi yanlış verilmiştir.
    "pre-order" dolaşım demek bir dizin ile karşılaşıldığında önce dizin girişinin ele alınması
sonra özyineleme yapılması demektir. "post-order" dolaşım ise
    önce özyineleme yapılıp sonra dizin girişinin ele alınması demektir. Defaul durum "pre-order"
dolaşım biçimindedir.

    FTW_MOUNT: Bu bayrak belirtilirse özyineleme yapılırken bir "mount point" ile karşılaşılırsa o
dosya sistemine girilmez. Default durumda
    özyineleme sırasında bir "mount point" ile kaşılaşılırsa özyineleme o dosya sisteminin içine
girilerek devam ettirilmektedir.

    FTW_PHYS: Default durumda nftw fonksiyonu bir sembolik link dosyası ile karşılaştığında linki
izler ve link'in hedefine
    yönelik hareket eder. Daha önce bir böyle bir durumun sonsuz döngüye yol açabileceğinden
bahsetmiştik. Bu nedenle biz özyinelemede stat fonksiyonu yerine
    lstat fonksiyonunu kullanmıştık. İşte bu bayrak belirtilirse artık nftw fonksiyonu sembolik
link dosyası ile karşılaştığında link'i izlemez,
    sembolik link dosyasının kendisi hakkında bilgi verir.

    Programcı bu dördüncü parametreye hiçbir bayrak geçmek istemezse 0 girebilir.

    nftw fonksiyonun geri dönüş değeri fonksiyon başarıssa -1, başarılıysa 0'dır. Ancak aslında
```

fonksiyon başarılı durumda callback fonksiyonun
    geri dönüş değeri ile geri döner. Şöyle ki: Biz callback fonksiyonu 0 ile geri döndürürsek özyinelemeye devam etmek istediğimizi belirtmiş
    oluruz. Bu durumda bir IO hatası da olmazsa nftw fonksiyonu 0 ile geri döner. Eğer biz bu fonksiyondan sıfır dışı bir değerle geri dönersek.
    nftw fonksiyonu özyinelemeyi bırakıp hemen geri çıkar ve bizim callback fonksiyondan döndürdüğümüz sıfır dışı değerle geri döner.

    Şimdi de callback fonksiyonun parametrelerine gelelim:

    int callback(const char *path, const struct stat *finfo, int flag, struct FTW *ftw);

    Fonksiyonun birinci parametresine bulunun dizin girişinin yol ifadesi yerleştirilir. Bu yol ifadesinin baş kısmı tamamen
    bizim nftw fonksiyonuna verdiğimiz dizin ifadesinden oluşmaktadır. (Yani biz nftw fonksiyonuna mutlak bir yol iafdesi verirsek buraya
    mutlak bir yol ifadesi geçirilir, biz nftw fonksiyonuna göreli bir yol ifadesi verirsek burada göreli bir yol ifadesi geçirilir.)
    Fonksiyonun ikinci parametresi bulunan dizin girişine ilişkin struct stat yapısının adresini belirtmektedir. Fonksiyonun üçüncü parametresi ise
    bulunan dizin girişinin türünü belirtmektedir. Bu tür şunlardan birine tam eşit olmak zorundadır:

    FTW_D: Bulunan giriş bir dizin girişidir.

    FTW_DNR: Bulunan giriş bir dizin girişidir. Ancak bu dizin'in içi okunamamaktadır. Dolayısıyla bu dizin özyinelemede dolaşılamayacaktır.

    FTW_DP: Post-order dolaşımda bir dizinle karşılaşıldığında bayrak FTW_D yerine FTW_DP olarak set edilmektedir.

    FTW_F: Bulunan dizin girişi sıradan bir dosyadır (regular file).

    FTW_NS: Bulunan dizin girişi için stat ya da lstat fonksiyonu başarısız olmuştur. Dolayısıyla fonksiyona geçirilen stat yapısı da
    anlamlı değildir.

    FTW_SL: Bulunan giriş bir sembolik bağlantı dosyasına ilişkindir. Sembolik bağlantı dosyasının hedefi mevcuttur.

    FTW_SLN: Bulunan giriş bir sembolik bağlantı dosyasına ilişkindir. Sembolik bağlantı dosyasının hedefi mevcut değildir ("danging link" durumu).

    callback fonksiyonun son parametresi FTW isimli bir yapı türündendir Bu yapı şöyle bildirilmiştir:

    struct FTW {
        int base;
        int level;
    };

    Yapının level elemanı ağaçtaki derinlik düzeyini belirtmektedir. Bu değer 0'dan başlayarak derine indikçe artırılmaktadır.
    base elemanı ise dizin girişinin birinci parametrede belirtilen yol ifadesinin kaçıncı indeksinden başladığını belirtmektedir. Örneğin
    biz "/home/kaan/Study" dizinini dolaşmak istemiş olalım. Fonksiyon da dizin girişi olarak "sample.c" bulmuş olsun. Fonksiyon bize bu girişi
    "/home/kaan/Study/sample.c" biçiminde verecektir. İşte buradaki base 17 olarak verilecektir.

    Aşağıda nftw fonksiyonunun kullanımına bir örnek verilmiştir.
----------*/

#define _XOPEN_SOURCE 500

#include <stdio.h>
#include <stdlib.h>
#include <ftw.h>





```
int callback(const char *path, const struct stat *finfo, int flag, struct FTW *ftw);
void exit_sys(const char *msg);

int main(int argc, char *argv[])
{
        int result;

        if (argc != 2) {
                fprintf(stderr, "wrong number of arguments!...\n");
                exit(EXIT_FAILURE);
        }

        if ((result = nftw(argv[1], callback, 100, FTW_PHYS)) == -1)
                exit_sys("nftw");

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

        return 0;
}

int callback(const char *path, const struct stat *finfo, int flag, struct FTW *ftw)
{
        switch (flag) {
                case FTW_DNR:
                        printf("%*s%s (cannot read directory)\n", ftw->level * 4, "", path + ftw->base);
                        break;
                case FTW_NS:
                        printf("%*s%s (cannot get statinfo)\n", ftw->level * 4, "", path + ftw->base);
                        break;
                default:
                        printf("%*s%s\n", ftw->level * 4, "", path + ftw->base);
        }

        return 0;
}

void exit_sys(const char *msg)
{
        perror(msg);

        exit(EXIT_FAILURE);
}

/*----------------------------------------------------------------------------------------------------------
    Bir dosyanın hard link'ini programlama yoluyla oluşturabilmek için link isimli POSIX fonksiyonu kullanılmaktadır. Linux sistemlerinde
    link fonksiyonu doğrudan işletim sisteminin sys_link isimli sistem fonksiyonunu çağırmaktadır. link fonksiyonunun prototipi şöyledir:

    #include <unistd.h>

    int link(const char *oldpath, const char *newpath);

    Fonksiyonun birinci parametresi hard link'i çıkartılacak dosyanın yol ifadesini, ikinci parametresi yeni dizin girişinin ismini belirtmektedir.
    Tabii prosesin ilgili dizine yazma hakkının olması gerekir. POSIX standartlarına göre bir sembolik bağlantı dosyasının har link'i çıkartılırken
    bu sembolik bağlantının kendisinin mi yoksa onun hedefinin mi link'inin çıkartılacağı işletim sistemini yazanların isteğine bırakılmıştır.
    Fonksiyon başarı durumunda 0 değerine, başarısızlık durumunda -1 değerine geri döner ve errno uygun biçimde değer alır.

    Bir dizin'in hard link'inin çıkartılması özyinelemeli fonksiyonları sonsuz döngüye sokabilmektedir. Bu nedenle dizinler üzerinde
    hard link çıkartma şüpheli bir durumdur. POSIX standartları bir dizin'in hard link'ini çıkartılabilmesi için prosesin
```





```
    bunu yapabilecek önceliğe sahip olması gerektiğini (yani etkin kullanıcı id'sinin 0 olması gerektiğini) ve işletim sisteminin de
    dizinlerin hard link'lerinin çıkartılabilmesine izin vermesi gerektiğini belirtmektedir. Eskiden Linux sistemleri root prosesler için
    dizinlerin hard link'lerinin çıkartılmasına izin veriyordu. Ancak sonraları bunu da kaldırdı. Yani Linux istemlerinde dizinlerin
    hard link'leri artık çıkartılamamaktadır.

    Daha önceden de belirtildiği gibi bir dosyanın hard link'i komut satırından ln komutuyla oluşturulabilmektedir. Tabii aslında bu program
    link fonksiyonu çağrılarak yazılmıştır. Örneğin

    ln a b

    Aşağıdaki örnekte komut satırından verilen bir dosyanın hard link'i çıkartılmıştır.
----------*/

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

void exit_sys(const char *msg);

int main(int argc, char *argv[])
{
        if (argc != 3) {
                fprintf(stderr, "wrong number of arguments!...\n");
                exit(EXIT_FAILURE);
        }

        if (link(argv[1], argv[2]) == -1)
                exit_sys("link");

        return 0;
}

void exit_sys(const char *msg)
{
        perror(msg);

        exit(EXIT_FAILURE);
}

/*----------------------------------------------------------------------------------------------------------
    link fonksiyonunun linkat ismiyle "at"li bir versiyonu da vardır. linkat fonksiyonunun prototipi şöyledir:

    #include <fcntl.h>

    int linkat(int fd1, const char *path1, int fd2, const char *path2, int flag);

    Fonksiyonun birinci parametresi ikinci parametresiyle belirtilen yok ifadesi göreli ise aramanın yapılacağı dizinin betimleyicisini alır.
    Üçüncü parametres ise dörddüncü parametrede belirtilen yol ifadesi göreli ise aramanın yapılacağı dizin'in betimleyicisini almaktadır.
    Son parametre sembolik bağlantının izlenip inzlenmeyeceğini belirtir. Eğer bu parametre AT_SYMLINK_FOLLOW biçiminde girilirse semboli
    bağlantı izlenir. Eğer bu parametre 0 girilirse sembolik bağlantı izlenmez.
----------*/

/*----------------------------------------------------------------------------------------------------------
    Bir dosyanın sembolik bağlantı dosyası symlink isimli POSIX fonksiyonuyla oluşturulmaktadır. Bu fonksiyon Linux sistemlerinde
    doğrudan sys_symlink isimli sistem fonksiyonunu çağırmaktadır. Fonksiyonun prototipi şöyledir:
```





```
    #include <unistd.h>

    int symlink(const char *path1, const char *path2);

    Fonksiyonun birinci parametresi sembolik bağlantısı çıkartılacak dosyanın ypl ifadesini, ikinci parametresi ise sembolik bağlantı dosyasının
    yol ifadesini alır. Fonksiyon başarı durumunda sıfır değerine, başarısızlık durumunda -1 değerine geri döner. Prototipten de gördüğünüz
    gibi sembolik bağlantı dosyasının kendisine ilişkin erişim hakları bizden istenmemektedir. Çünkü sembolik bağlantı dosyasının kendi erişim haklarının
    bir önemi yoktur. Bu fonksiyon bu erişim hakları için "rwxrwxrwx" haklarını vermektedir. Sembolik bağlantı dosyasını izleyen fonksiyonlar
    hedef dosyanın erişim haklarını dikkate alırlar. Sembolik bağlantı dosyasının kendi erişim haklarının bir önemi yoktur.

    symlink fonksiyonu ile "dangling" link oluşturulabilmektedir. Yani başka bir deyişle fonksiyonun birinci parametresinde belirtilen
    dosyanın bulunuyor olması gerekmez.

    Bir POSIX fonksiyonun sembolik bağlantı dosyasını izleyip izlemediğine dikkat ediniz. Şüphe duyarsanız dokümanlardan bunu doğrulayınız.
    Örneğin open fonksiyonu sembolik bağlantıyı izlemektedir. Ancak remove ve unlik fonksiyonları sembolik bağlantı dosyalarını izlememektedir.
    (Yani remove ve unlink ile sembolik bağlantı dosyası silinmeye çalışılırsa bu fonksiyonlar sembolik bağlantı dosyasının kendisini silmektedir.)
    Genel olarak POSIX fonksiyonlarının büyük bölümü sembolik bağlantı dosyasını izlemektedir.

    Bir POSIX fonksiyonu "pathname resolution" işlemini yaparken belli sayıdadan fazla sembolik link üzerinden geçilmişse
    döngüsel bir durumun oluştuğunu düşünerek ELOOP özel değeri ile geri dönmektedir. Örneğin:

    a -> b
    b -> a

    Bu durumda biz "a" dosyasını open fonksiyonuyla açmak istersek fonksiyon başarısız olur ve errno ELOOP değerini alır.

    Dizin'lerin hard link'lerinin çıkartılması sorunlu bir durum oluşturduğundan yukarı söz etmiştik. Halbuki aynı durum sembolik
    bağlantılar için geçerli değildir. Yani sıradan bir proses bir dizin'in sembolik bağlantısını oluşturabilmektedir.

    Daha önceden de belirtildiği gibi bir dosyanın sembolik bağlantısı ln -s kabuk komutuyla oluşturulabilmektedir. Örneğin:

    ln -s a b

    Burada b sembolik bağlantı dosyası a dosyasını göstermektedir.

    Aşağıda komut satırından hareketle bir sembolik bağlantı oluşturma örneği verilmiştir.
----------*/

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

void exit_sys(const char *msg);

int main(int argc, char *argv[])
{
        if (argc != 3) {
                fprintf(stderr, "wrong number of arguments!...\n");
                exit(EXIT_FAILURE);
        }

        if (symlink(argv[1], argv[2]) == -1)
                exit_sys("symlink");
```