

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

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

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

    return 0;
}
```

```
/
*-----
-----
    Programın komut satırı argümanları
-----
-----*/
```

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

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

    for (i = 0; i < argc; ++i)
        puts(argv[i]);

    return 0;
}
```

```
/
*-----
-----
    getopt örneği
-----
-----*/
```

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

```
int main(int argc, char *argv[])
{
    int result;
    int a_flag = 0, b_flag = 0, c_flag = 0;
    char *a_arg, *c_arg;
    int i;

    opterr = 0;
    while ((result = getopt(argc, argv, "a:bc:")) != -1) {
```

```

switch (result) {
    case 'a':
        a_flag = 1;
        a_arg = optarg;
        break;
    case 'b':
        b_flag = 1;
        break;
    case 'c':
        c_flag = 1;
        c_arg = optarg;
        break;
    case '?':
        if (optopt == 'c')
            fprintf(stderr, "c switch without argument!..\n");
        else if (optopt == 'a')
            fprintf(stderr, "a switch without argument!..\n");
        else
            fprintf(stderr, "invalid switch: -%c\n", optopt);
        exit(EXIT_FAILURE);
}
}

```

```

if (a_flag)
    printf("a switch specified --> %s\n", a_arg);

```

```

if (b_flag)
    printf("b switch specified\n");

```

```

if (c_flag)
    printf("c switch specified --> %s\n", c_arg);

```

```

printf("arguments without switch:\n");

```

```

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

```

```

return 0;

```

```

}

```

```

/

```

```

*-----

```

```

-----

```

```

    getopt örneği

```

```

-----

```

```

-----*/

```

```

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

```

```

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

```

```

    int m_flag = 0, a_flag = 0, s_flag = 0, d_flag = 0;

```

```

int result;
double op_result, op1, op2;

while ((result = getopt(argc, argv, "masd")) != -1) {
    switch (result) {
        case 'm':
            m_flag = 1;
            break;
        case 'a':
            a_flag = 1;
            break;
        case 's':
            s_flag = 1;
            break;
        case 'd':
            d_flag = 1;
            break;
        case '?':
            fprintf(stderr, "invalid switch: -%c\n", optopt);
            exit(EXIT_FAILURE);
    }
}

if (m_flag + a_flag + s_flag + d_flag != 1) {
    fprintf(stderr, "only one switch must be specicified!\n");
    exit(EXIT_FAILURE);
}

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

op1 = strtod(argv[optind], NULL);
op2 = strtod(argv[optind + 1], NULL);

if (m_flag)
    op_result = op1 * op2;
else if (a_flag)
    op_result = op1 + op2;
else if (s_flag)
    op_result = op1 - op2;
else if (d_flag)
    op_result = op1 / op2;

printf("%f\n", op_result);

return 0;
}

```

/

*-----

getopt_long örneği

```

-----*/

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

int main(int argc, char *argv[])
{
    int result;
    int a_flag = 0, b_flag = 0, c_flag = 0, head_flag = 0, tail_flag = 0;
    char *b_arg, *head_arg;
    int i;
    struct option options[] = {
        {"head", required_argument, NULL, 'h'},
        {"tail", no_argument, NULL, 't'},
        {0, 0, 0, 0}
    };

    opterr = 0;
    while ((result = getopt_long(argc, argv, "ab:c", 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':
                head_flag = 1;
                head_arg = optarg;
                break;
            case 't':
                tail_flag = 1;
                break;
            case '?':
                fprintf(stderr, "invalid option!\n");
                exit(EXIT_FAILURE);
        }
    }

    if (a_flag)
        printf("a switch given\n");
    if (b_flag)
        printf("b switch given with argument %s\n", b_arg);
    if (c_flag)
        printf("c switch given\n");
    if (head_flag)
        printf("head switch given with argument %s\n", head_arg);

```

```

    if (tail_flag)
        printf("tail switch given\n");

    printf("Arguments without switch:\n");
    for (i = optind; i < argc; ++i)
        printf("%s\n", argv[i]);

    /* .... */

    return 0;
}

/
*-----
-----
    getopt_long örneği
    mycat [-opt] [--top[=n] --header] <dosya listesi>
-----
-----*/

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

/* Symbolic Constans */

#define DEF_LINE          10
#define HEX_OCTAL_LINE_LEN 16

/* Function Prorotypes */

int print_text(FILE *f, int nline);
int print_hex_octal(FILE *f, int nline, int hexflag);

int main(int argc, char *argv[])
{
    int result, err_flag = 0;
    int x_flag = 0, o_flag = 0, t_flag = 0, top_flag = 0, header_flag = 0;
    char *top_arg;
    struct option options[] = {
        {"top", optional_argument, NULL, 1},
        {"header", no_argument, NULL, 'h'},
        {0, 0, 0, 0}
    };
    FILE *f;
    int i, nline = -1;

    opterr = 0;
    while ((result = getopt_long(argc, argv, "xoth", options, NULL)) != -1)
    {
        switch (result) {
            case 'x':
                x_flag = 1;
                break;

```

```

        case 'o':
            o_flag = 1;
            break;
        case 't':
            t_flag = 1;
            break;
        case 'h':
            header_flag = 1;
            break;
        case 1:
            top_flag = 1;
            top_arg = optarg;
            break;
        case '?':
            if (optopt != 0)
                fprintf(stderr, "invalid switch: -%c\n", optopt);
            else
                fprintf(stderr, "invalid switch: %s\n", argv[optind -
                    1]); /* argv[optind - 1] dokümanite edilmemiş */
            err_flag = 1;
    }
}

if (err_flag)
    exit(EXIT_FAILURE);

if (x_flag + o_flag + t_flag > 1) {
    fprintf(stderr, "only one option must be specified from -o, -t, -
        x\n");
    exit(EXIT_FAILURE);
}

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

if (top_flag)
    nline = top_arg != NULL ? (int) strtol(top_arg, NULL, 10) :
        DEF_LINE;

if (optind == argc) {
    fprintf(stderr, "at least one file must be specified!..\n");
    exit(EXIT_FAILURE);
}

for (i = optind; i < argc; ++i) {
    if ((f = fopen(argv[i], "rb")) == NULL) {
        fprintf(stderr, "cannot open file: %s\n", argv[i]);
        continue;
    }

    if (header_flag)
        printf("%s\n\n", argv[i]);

    if (t_flag)
        result = print_text(f, nline);
    else if (x_flag)

```

```

        result = print_hex_octal(f, nline, 1);
    else
        result = print_hex_octal(f, nline, 0);

    if (i != argc - 1)
        putchar('\n');

    if (!result)
        fprintf(stderr, "cannot read file: %s\n", argv[i]);

    fclose(f);
}

return 0;
}

int print_text(FILE *f, int nline)
{
    int ch;
    int count;

    if (nline == -1)
        while ((ch = fgetc(f)) != EOF)
            putchar(ch);
    else {
        count = 0;
        while ((ch = fgetc(f)) != EOF && count < nline) {
            putchar(ch);
            if (ch == '\n')
                ++count;
        }
    }

    return !ferror(f);
}

int print_hex_octal(FILE *f, int nline, int hexflag)
{
    int ch, i, count;
    const char *off_str, *ch_str;

    off_str = hexflag ? "%07X " : "%012o";
    ch_str = hexflag ? "%02X%c" : "%03o%c";

    if (nline == -1)
        for (i = 0; (ch = fgetc(f)) != EOF; ++i) {
            if (i % HEX_OCTAL_LINE_LEN == 0)
                printf(off_str, i);
            printf(ch_str, ch, i % HEX_OCTAL_LINE_LEN == HEX_OCTAL_LINE_LEN
                - 1 ? '\n' : ' ');
        }

    else {
        count = 0;
        for (i = 0; (ch = fgetc(f)) != EOF && count < nline; ++i) {

```

```

        if (i % HEX_OCTAL_LINE_LEN == 0)
            printf(off_str, i);
        printf(ch_str, ch, i % HEX_OCTAL_LINE_LEN == HEX_OCTAL_LINE_LEN -
1 ? '\n' : ' ');
        if (ch == '\n')
            ++count;
    }
}

if (i % HEX_OCTAL_LINE_LEN != 0)
    putchar('\n');

return !ferror(f);
}

```

```

/
*-----
-----
    POSIX fonksiyonlarında hataların tespit edilmesi
-----
-----*/

```

```

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

int main(void)
{
    if (chdir("xxxxx") == -1) {
        fprintf(stderr, "error: %s (%d)\n", strerror(errno), errno);
        exit(EXIT_FAILURE);
    }
    printf("Ok\n");

    return 0;
}

```

```

/
*-----
-----
    POSIX fonksiyonlarında hataların tespit edilmesi
-----
-----*/

```

```

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

int main(void)
{
    if (chdir("xxxxx") == -1) {
        perror("chdir");
        exit(EXIT_FAILURE);
    }
}

```



```

    }
    printf("Ok\n");

    return 0;
}

/
*-----
-----
    exit_sys fonksiyonu ile hata kontrolünün kısaltılması
-----
-----*/

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

void exit_sys(const char *msg);

int main(void)
{
    if (chdir("xxxxx") == -1)
        exit_sys("chdir");

    printf("Ok\n");

    return 0;
}

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

    exit(EXIT_FAILURE);
}

/
*-----
-----
    open fonksiyonunun kullanılması
-----
-----*/

#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;

```

```

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

    printf("success\n");

    close(fd);

    return 0;
}

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

    exit(EXIT_FAILURE);
}

/
*-----
-----
    open fonksiyonun kullanılması
-----
-----*/

#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.txt", O_RDWR)) == -1)
        exit_sys("open");

    printf("success\n");

    close(fd);

    return 0;
}

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

    exit(EXIT_FAILURE);
}

```

```

/
*-----
-----
    read fonksiyonun kullanımı
-----
-----*/

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

#define SIZE    100

void exit_sys(const char *msg);

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

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

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

    buf[result] = '\0';
    puts(buf);

    close(fd);

    return 0;
}

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

    exit(EXIT_FAILURE);
}

/
*-----
-----
    read fonksiyonunun EOF'a kadar döngü içerisinde çağrılmasına bir örnek
-----
-----*/

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

```

```

#define CHUNK_SIZE      10

void exit_sys(const char *msg);

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

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

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

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

    putchar('\n');

    close(fd);

    return 0;
}

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

    exit(EXIT_FAILURE);
}

/
*-----
-----
    read fonksiyonun döngü içerisinde çağrılıp okunan bilgilerin hex biçimde
    ekrana yazdırılması
-----
-----*/

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

#define CHUNK_SIZE      10

void exit_sys(const char *msg);

int main(void)
{
    int fd;

```

```

int i, k;
unsigned char buf[CHUNK_SIZE];
ssize_t result;

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

k = 0;
while ((result = read(fd, buf, CHUNK_SIZE)) > 0)
    for (i = 0; i < result; ++i) {
        printf("%02X%c", buf[i], k % 16 == 15 ? '\n' : ' ');
        ++k;
    }

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

putchar('\n');

close(fd);

return 0;
}

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

    exit(EXIT_FAILURE);
}

/
*-----
-----
    write fonksiyonun kullanımı
-----
-----*/

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

#define CHUNK_SIZE    10

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("write");

```

```

    if (write(fd, "istanbul", 8) == -1)
        exit_sys("write");

    printf("Ok\n");

    close(fd);

    return 0;
}

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

    exit(EXIT_FAILURE);
}

/
*-----
-----
    Dosya kopyalama örneği (kaynak dosyadan blok blok okuma yapıp hedef
    dosyaya yazılıyor)
-----
-----*/

/* mycp.c */

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

#define CHUNK_SIZE      4096

void exit_sys(const char *msg);

int main(int argc, char *argv[])
{
    int fds, fdd;
    char buf[CHUNK_SIZE];
    ssize_t result_r, result_w;

    if (argc != 3) {
        fprintf(stderr, "wrong number of arguments!\n");
        fprintf(stderr, "usage: mycp <source path> <destination path>\n");
        exit(EXIT_FAILURE);
    }

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

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

```

```

        exit_sys("open");

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

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

close(fds);
close(fdd);

return 0;
}

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

    exit(EXIT_FAILURE);
}

/
*-----
-----
Dosya göstericisini EOF durumuna alıp dosyaya yazma yaparak dosyayı
büyütme örneği
-----
-----*/

```

```

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

void exit_sys(const char *msg);

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

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

    if (lseek(fd, 0, SEEK_END) == -1)
        exit_sys("lseek");

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

```

```

    close(fd);

    printf("Ok\n");

    return 0;
}

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

    exit(EXIT_FAILURE);
}

/
*-----
-----
    Dosya deliklerinin oluşturulması ve delikten okuma yapma
-----
-----*/

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

void exit_sys(const char *msg);

int main(void)
{
    int fd;
    char wbuf[] = "this is a test\n";
    char rbuf[64];
    ssize_t result;
    off_t pos;
    int i;

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

    if ((pos = lseek(fd, 1000000, SEEK_SET)) == -1)
        exit_sys("lseek");

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

    if (lseek(fd, 1000000, SEEK_SET) == -1)
        exit_sys("lseek");

    if ((result = read(fd, rbuf, 64)) == -1)
        exit_sys("read");

    for (i = 0; i < 64; ++i)

```



```

        printf("%02X%c", (unsigned char)rbuf[i], i % 16 == 15 ? '\n' : ' ');

    putchar('\n');

    close(fd);

    return 0;
}

```

```

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

    exit(EXIT_FAILURE);
}

```

```

/
*-----
-----
Linux sistemlerinde POSIX standartlarında bulunmayan copy_file_range
isimli bir sistem fonksiyonu ve bunu sarmalayan bir
kütüphane fonksiyonu bulunmaktadır. Bu fonksiyon iki dosya
betimleyicisini alarak hiç user mod tampon kullanmadan belli bir miktar
byte'ı bir betimleyicinin kaynağından diğer betimleyiciin hedefine
kopyalamaktadır. Fonksiyonun prototipi şöyledir:

```

```

ssize_t copy_file_range(int fd_in, loff_t *off_in, fd_out, loff_t
    *off_out, size_t len, unsigned int flags);

```

Fonksiyonun ayrıntılı çalışması man sayfalarında açıklanmaktadır.

```

-----*/

```

```

/
*-----
-----
    chmod POSIX fonksiyonun kullanımı

```

```

-----*/

```

```

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

```

```

void exit_sys(const char *msg);

```

```

int main(void)
{
    if (chmod("test.txt", S_IRUSR|S_IWUSR|S_IRGRP) == -1)
        exit_sys("chmod");

    return 0;
}

```

```

}

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

    exit(EXIT_FAILURE);
}

/
*-----
-----
    fchmod POSIX fonksiyonun kullanımı
-----
-----*/

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

void exit_sys(const char *msg);

int main(void)
{
    int fd;

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

    if (fchmod(fd, S_IRUSR|S_IWUSR|S_IRGRP|S_IROTH) == -1)
        exit_sys("chmod");

    close(fd);

    return 0;
}

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

    exit(EXIT_FAILURE);
}

/
*-----
-----
    chmod komutunun octal set etme işlemini yapan örnek biçimi
-----
-----*/

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

```

```

#include <string.h>
#include <errno.h>
#include <sys/stat.h>

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

int main(int argc, char *argv[])
{
    int i;
    long mode;
    mode_t modes[] = {S_IRUSR, S_IWUSR, S_IXUSR, S_IRGRP, S_IWGRP, S_IXGRP,
        S_IROTH, S_IWOTH, S_IXOTH};
    mode_t result_mode;

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

    if (!is_octal(argv[1]) || (mode = (mode_t)strtoul(argv[1], NULL, 8)) >
        0x777) {
        fprintf(stderr, "invalid octal digits!..\n");
        exit(EXIT_FAILURE);
    }

    result_mode = 0;
    for (i = 8; i >= 0; --i)
        if (mode >> i & 1)
            result_mode |= modes[8 - i];

    for (i = 2; i < argc; ++i)
        if (chmod(argv[i], result_mode) == -1)
            fprintf(stderr, "%s: %s\n", argv[i], strerror(errno));

    return 0;
}

int is_octal(const char *str)
{
    int i;

    for (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);
}

```

```
/
*-----
-----
    chown fonksiyonun kullanımı (change own restricted özelliği)
-----
-----*/
```

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

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

```
int main(void)
{
    if (chown("test.txt", 1001, -1) == -1)
        exit_sys("chown");

    printf("Ok\n");

    return 0;
}
```

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

    exit(EXIT_FAILURE);
}
```

```
/
*-----
-----
    Dizinler de dosyalar gibi open fonksiyonuyla açılabilirler
-----
-----*/
```

```
#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", O_RDONLY)) == -1)
        exit_sys("open");

    printf("Ok\n");

    close(fd);
}
```

```

    return 0;
}

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

    exit(EXIT_FAILURE);
}

/
*-----
-----
    Dizinlerde 'x' hakkı yol ifadesinde içinden geçilebilirlik anlamına
    gelmektedir. x hakkına sahip olmayan izinlerle
    yol ifadesi oluşturmak istersek sistem fonksiyonları başarısız
    olacaktır. Aşağıdaki örnekte test dizininin x hakkı
    kaldırılmıştır. Bu nedenle open fonksiyonu başarısız olmaktadır.
-----
-----*/

#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/test.txt", O_RDONLY)) == -1)
        exit_sys("open");

    printf("Ok\n");

    close(fd);

    return 0;
}

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

    exit(EXIT_FAILURE);
}

/
*-----
-----
    unlink ve remove fonksiyonları

```

```

-----*/
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

void exit_sys(const char *msg);

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

    printf("Ok\n");

    return 0;
}

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

    exit(EXIT_FAILURE);
}

/
*-----
    mkdir POSIX fonksiyonu (umask'ten etkilenmektedir)
-----
-----*/

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

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

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

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

    if (!is_octal(argv[1]) || (mode = (mode_t)strtoul(argv[1], NULL, 8)) >
        0x777) {
        fprintf(stderr, "invalid octal digits!..\n");
    }
}

```

```

        exit(EXIT_FAILURE);
    }

    result_mode = 0;
    for (i = 8; i >= 0; --i)
        if (mode >> i & 1)
            result_mode |= modes[8 - i];

    umask(0);
    if (mkdir(argv[2], result_mode) == -1)
        exit_sys("mkdir");

    return 0;
}

```

```

int is_octal(const char *str)
{
    int i;

    for (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);
}

```

```

/
*-----
-----
    Dizinlerin rmdir POSIX fonksiyonuyla silinmesi
-----
-----*/

```

```

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

```

```

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

    if (argc == 1) {
        fprintf(stderr, "wrong number of arguments!..\n");
        exit(EXIT_FAILURE);
    }
}

```

```

    for (i = 1; i < argc; ++i)
        if (rmdir(argv[i]) == -1)
            fprintf(stderr, "%s: %s\n", argv[i], strerror(errno));

    return 0;
}

```

```

/
*-----
-----
    rename fonksiyonula dizin girişinin isminin değiştirilmesi
-----
-----*/

```

```

#include <stdio.h>
#include <stdlib.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 (rename(argv[1], argv[2]) == -1)
        exit_sys("rename");

    return 0;
}

```

```

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

    exit(EXIT_FAILURE);
}

```

```

/
*-----
-----
    stat fonksiyonun kullanımı
-----
-----*/

```

```

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

```

```

const char *lsmode(mode_t mode);

```



```

int main(int argc, char *argv[])
{
    struct stat finfo;
    struct tm *pt;
    char buf[50];
    int i;

    if (argc == 1) {
        fprintf(stderr, "wrong number of arguments!..\n");
        exit(EXIT_FAILURE);
    }

    for (i = 1; i < argc; ++i) {
        if (i != 1)
            printf("-----\n");

        if (stat(argv[i], &finfo) == -1) {
            fprintf(stderr, "%s: %s\n", argv[i], strerror(errno));
            continue;
        }

        printf("Name: %s\n", argv[i]);
        printf("Inode No: %lu\n", (unsigned long)finfo.st_ino);
        printf("Hard Link: %lu\n", (unsigned long)finfo.st_nlink);
        printf("Access modes: %s\n", lsmode(finfo.st_mode));
        printf("User Id: %ld\n", (unsigned long)finfo.st_uid);
        printf("Group Id: %lu\n", (unsigned long)finfo.st_gid);
        printf("File Size: %ld\n", (long)finfo.st_size);
        printf("File System Block (Cluster) Size: %ld\n",
            (long)finfo.st_blksize);
        printf("Number Of Blocks (512 Bytes): %ld\n",
            (long)finfo.st_blocks);

        pt = localtime(&finfo.st_mtime);
        strftime(buf, 50, "%b %d %H:%M", pt);
        printf("Last Update: %s\n", buf);

        pt = localtime(&finfo.st_atime);
        strftime(buf, 50, "%b %d %H:%M", pt);
        printf("Last Access: %s\n", buf);

        pt = localtime(&finfo.st_atime);
        strftime(buf, 50, "%b %d %H:%M", pt);
        printf("Last Status Change: %s\n", buf);
    }

    return 0;
}

const char *lsmode(mode_t mode)
{
    static char modetxt[11];
    mode_t modes[] = {S_IRUSR, S_IWUSR, S_IXUSR, S_IRGRP, S_IWGRP, S_IXGRP,
        S_IROTH, S_IWOTH, S_IXOTH};

```

```

int i;

if (S_ISREG(mode))
    modetxt[0] = '-';
else if (S_ISDIR(mode))
    modetxt[0] = 'd';
else if (S_ISCHR(mode))
    modetxt[0] = 'c';
else if (S_ISBLK(mode))
    modetxt[0] = 'b';
else if (S_ISFIFO(mode))
    modetxt[0] = 'p';
else if (S_ISLNK(mode))
    modetxt[0] = 'l';
else if (S_ISSOCK(mode))
    modetxt[0] = 's';
else
    modetxt[0] = '?';

for (i = 0; i < 9; ++i)
    modetxt[1 + i] = (mode & modes[i]) ? "rwx"[i % 3] : '-';

return modetxt;
}

/
*-----
-----
    getpwnam fonksiyonu kullanıcı ismini alarak bize /etc/passwd
    dosyasındaki kullanıcı bilgilerini vermektedir
-----
-----*/

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

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

    if (argc == 1) {
        fprintf(stderr, "wrong number of arguments!..\n");
        exit(EXIT_FAILURE);
    }

    for (i = 1; i < argc; ++i) {
        if (i != 1)
            printf("-----\n");
        if ((pass = getpwnam(argv[i])) == NULL) {
            fprintf(stderr, "cannot find user: %s\n", argv[i]);
            continue;
        }
    }
}

```

```

        printf("User name: %s\n", pass->pw_name);
        printf("Password: %s\n", pass->pw_passwd);
        printf("User id: %ld\n", (long)pass->pw_uid);
        printf("Group id: %ld\n", (long)pass->pw_gid);
        printf("User info: %s\n", pass->pw_passwd);
        printf("Home directory: %s\n", pass->pw_dir);
        printf("Login Prog: %s\n", pass->pw_shell);
    }

    return 0;
}

/
*-----
-----
    getpwnam fonksiyonu kullanıcı id'sini alarak bize /etc/passwd
    dosyasındaki kullanıcı bilgilerini vermektedir
-----
-----*/

#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <pwd.h>

void exit_sys(const char *msg);

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

    if (argc == 1) {
        fprintf(stderr, "wrong number of arguments!..\n");
        exit(EXIT_FAILURE);
    }

    for (i = 1; i < argc; ++i) {
        if (i != 1)
            printf("-----\n");

        uid = strtoul(argv[i], NULL, 10);
        if ((pass = getpwnam((uid_t)uid)) == NULL) {
            fprintf(stderr, "cannot find user: %s\n", argv[i]);
            continue;
        }

        printf("User name: %s\n", pass->pw_name);
        printf("Password: %s\n", pass->pw_passwd);
        printf("User id: %ld\n", (long)pass->pw_uid);
        printf("Group id: %ld\n", (long)pass->pw_gid);
        printf("User info: %s\n", pass->pw_passwd);
        printf("Home directory: %s\n", pass->pw_dir);
        printf("Login Prog: %s\n", pass->pw_shell);
    }
}

```

```

    }

    return 0;
}

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

    exit(EXIT_FAILURE);
}

/
*-----
-----
    getpwent, setpwent ve endpwent fonskiyonlarının kullanımı
-----
-----*/

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

void exit_sys(const char *msg);

int main(void)
{
    struct passwd *pass;

    while ((pass = getpwent()) != NULL)
        printf("User name: %s, User id:%lu\n", pass->pw_name, (unsigned
            long)pass->pw_uid);

    printf("-----\n");

    setpwent();

    while ((pass = getpwent()) != NULL)
        printf("User name: %s, User id:%lu\n", pass->pw_name, (unsigned
            long)pass->pw_uid);

    endpwent();

    return 0;
}

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

    exit(EXIT_FAILURE);
}

```

```

/
*-----
-----
    getgrnam fonksiyonu grup ismini alarak /etc/group dosyasındaki grupla
    ilgili bilgileri bize vermektedir
-----
-----*/

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

int main(int argc, char *argv[])
{
    struct group *grp;
    int i, k;

    if (argc == 1) {
        fprintf(stderr, "wrong number of arguments!..\n");
        exit(EXIT_FAILURE);
    }

    for (i = 1; i < argc; ++i) {
        if (i != 1)
            printf("-----\n");

        if ((grp = getgrnam(argv[i])) == NULL) {
            fprintf(stderr, "cannot find user: %s\n", argv[i]);
            continue;
        }

        printf("Group name: %s\n", grp->gr_name);
        printf("Group password: %s\n", grp->gr_passwd);
        printf("Group id: %ld\n", (long)grp->gr_gid);
        printf("Group members: ");
        for (k = 0; grp->gr_mem[k] != NULL; ++k) {
            if (k != 0)
                printf(", ");
            printf("%s", grp->gr_mem[k]);
        }
        printf("\n");
    }

    return 0;
}

/
*-----
-----
    getgrgid fonksiyonu grubun id'sini alarak /etc/group dosyasından grup
    bilgilerini alıp bize vermektedir.
-----
-----*/

#include <stdio.h>

```

```

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

int main(int argc, char *argv[])
{
    struct group *grp;
    int i, k;
    long gid;

    if (argc == 1) {
        fprintf(stderr, "wrong number of arguments!..\n");
        exit(EXIT_FAILURE);
    }

    for (i = 1; i < argc; ++i) {
        if (i != 1)
            printf("-----\n");

        gid = strtol(argv[i], NULL, 10);
        if ((grp = getgrgid((gid_t)gid)) == NULL) {
            fprintf(stderr, "cannot find user: %s\n", argv[i]);
            continue;
        }

        printf("Group name: %s\n", grp->gr_name);
        printf("Group password: %s\n", grp->gr_passwd);
        printf("Group id: %ld\n", (long)grp->gr_gid);
        printf("Group members: ");
        for (k = 0; grp->gr_mem[k] != NULL; ++k) {
            if (k != 0)
                printf(", ");
            printf("%s", grp->gr_mem[k]);
        }
        printf("\n");
    }

    return 0;
}

/
*-----
-----
    Dosya bilgilerini ls -l stili ile yazdıran program (birden fazla dosya
    için hizalama uygulanmadı)
-----
-----*/

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

void exit_sys(const char *msg);

int main(void)
{

```

```

    struct group *grp;

    while ((grp = getgrent()) != NULL)
        printf("Group name: %s, Group id:%lu\n", grp->gr_name, (unsigned
            long)grp->gr_gid);

    printf("-----\n");

    setgrent();

    while ((grp = getgrent()) != NULL)
        printf("Group name: %s, Group id:%lu\n", grp->gr_name, (unsigned
            long)grp->gr_gid);

    endgrent();

    return 0;
}

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

    exit(EXIT_FAILURE);
}

/
*-----
-----
    Dosya bilgilerini ls -l stili ile yazdıran progam (birden fazla dosya
    için hizalama uygulanmadı)
-----
-----*/

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

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

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

    if (argc == 1) {
        fprintf(stderr, "wrong number od arguments!..\n");
    }

```

```

        exit(EXIT_FAILURE);
    }

    for (i = 1; i < argc; ++i) {
        if (stat(argv[i], &finfo) == -1) {
            fprintf(stderr, "file not found: %s\n", argv[i]);
            continue;
        }
        if ((lsi = get_ls(argv[i], &finfo)) == NULL)
            fprintf(stderr, "cannot get file info: %s\n", argv[i]);
        puts(lsi);
    }

    return 0;
}

const char *get_ls(const char *name, const struct stat *finfo)
{
    static char ltext[2048];
    char datebuf[32];
    struct passwd *pass;
    struct group *grp;
    struct tm *pt;

    mode_t modes[] = {S_IRUSR, S_IWUSR, S_IXUSR, S_IRGRP, S_IWGRP, S_IXGRP,
        S_IROTH, S_IWOTH, S_IXOTH};
    int i;

    if (S_ISREG(finfo->st_mode))
        ltext[0] = '-';
    else if (S_ISDIR(finfo->st_mode))
        ltext[0] = 'd';
    else if (S_ISCHR(finfo->st_mode))
        ltext[0] = 'c';
    else if (S_ISBLK(finfo->st_mode))
        ltext[0] = 'b';
    else if (S_ISFIFO(finfo->st_mode))
        ltext[0] = 'p';
    else if (S_ISLNK(finfo->st_mode))
        ltext[0] = 'l';
    else if (S_ISSOCK(finfo->st_mode))
        ltext[0] = 's';
    else
        ltext[0] = '?';

    for (i = 0; i < 9; ++i)
        ltext[1 + i] = (finfo->st_mode & modes[i]) ? "rwx"[i % 3] : '-';

    if ((pass = getpwuid(finfo->st_uid)) == NULL)
        return NULL;

    if ((grp = getgrgid(finfo->st_gid)) == NULL)
        return NULL;

    pt = localtime(&finfo->st_mtime);

```



```

    strftime(datebuf, 32, "%b %e %H:%M", pt);

    sprintf(lstext + 10, " %lu %s %s %ld %s %s", (unsigned long)finfo-
        >st_nlink, pass->pw_name, grp->gr_name, (long)finfo->st_size, datebuf,
        name);

    return lstext;
}

/
*-----
-----
    /dev/zero aygıt sürücüsünden okuma yapıldığında hep 0 okunur, yazma
    yapıldığında yazılanlar dikkate alınmaz
-----
-----*/

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

#define SIZE    1024

void exit_sys(const char *msg);

int main(void)
{
    int fd;
    unsigned char buf[SIZE];
    ssize_t result;
    int i;

    if ((fd = open("/dev/zero", O_RDONLY)) == -1)
        exit_sys("open");

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

    for (i = 0; i < result; ++i) {
        if (i % 16 == 0)
            printf("%08X ", i);
        printf("%02X%c", buf[i], i % 16 == 15 ? '\n' : ' ');
    }
    if (i % 16 != 0)
        putchar('\n');

    close(fd);

    return 0;
}

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

```

```
    exit(EXIT_FAILURE);
}
```

```
/
```

```
*-----
```

```
-----
```

```
Homework-4 için ipucu
```

```
-----
```

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

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

```
#define MAX_LINE_LEN      4096
```

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

```
struct passwd *csd_getpwnam(const char *uname)
{
```

```
    static struct passwd pass;
    static char buf[MAX_LINE_LEN];
    FILE *f;
    char *tok;
    int success = 0;
```

```
    if ((f = fopen("/etc/passwd", "r")) == NULL)
        return NULL;
```

```
    while (fgets(buf, MAX_LINE_LEN, f) != NULL) {
        tok = strtok(buf, ":\n");
        if (tok == NULL)
            return NULL;
        if (!strcmp(tok, uname)) {
            success = 1;
            break;
        }
    }
}
```

```
if (!success)
    return NULL;
```

```
pass.pw_name = tok;
tok = strtok(NULL, ":");
if (tok == NULL)
    return NULL;
pass.pw_passwd = tok;
tok = strtok(NULL, ":");
if (tok == NULL)
    return NULL;
```

```
pass.pw_uid = (uid_t)strtoul(tok, NULL, 10);
/* etc... */
```

```

    return &pass;
}

int main(void)
{
    struct passwd *pass;

    if ((pass = csd_getpwnam("csd")) == NULL) {
        fprintf(stderr, "cannot find user!..\n");
        exit(EXIT_FAILURE);
    }

    printf("%s, %lu\n", pass->pw_name, (unsigned long)pass->pw_uid);

    return 0;
}

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

    exit(EXIT_FAILURE);
}

/
*-----
-----
    dup fonksiyonu parametresiyle belirtilen betimleyicinin gösterdiği dosya
    nesnesi ile aynı dosya nesnesini gösteren
    yeni bir dosya betimleyicisi tahsis eder ve onun değerine geri döner.
    dup en düşük boş betimleyiciyi tahsis etmektedir.
-----
-----*/

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

void exit_sys(const char *msg);

int main(void)
{
    int fd1, fd2;
    char buf[5 + 1];
    ssize_t result;

    if ((fd1 = open("test.txt", O_RDONLY)) == -1)
        exit_sys("open");

    if ((fd2 = dup(fd1)) == -1)
        exit_sys("dup");

    if ((result = read(fd1, buf, 5)) == -1)
        exit_sys("read");

```

```

    buf[result] = '\\0';
    puts(buf);

    if ((result = read(fd2, buf, 5)) == -1)
        exit_sys("read");

    buf[result] = '\\0';
    puts(buf);

    printf("fd1: %d, fd2: %d\\n", fd1, fd2);

    close(fd1);
    close(fd2);

    return 0;
}

```

```

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

    exit(EXIT_FAILURE);
}

```

```

/
*-----
-----
    C'de deęişken sayıda argüman alan fonksiyonların yazımı
-----
-----*/

```

```

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

void exit_sys(const char *msg);

int add(int a, ...);

int main(void)
{
    int result;

    result = add(5, 10, 20, 30, 40, 50);
    printf("%d\\n", result);

    return 0;
}

int add(int n, ...)
{
    va_list arg;

```

```

    int result;
    int i;

    va_start(arg, n);

    result = 0;
    for (i = 0; i < n; ++i)
        result += va_arg(arg, int);

    va_end(arg);

    return result;
}

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

    exit(EXIT_FAILURE);
}

/
*-----
-----
    fcntl fonksiyonu betimleyici üzerinde çeşitli get set işlemleri yapan
    genel amaçlı bir fonslyondur. Örneğin aşağıda dosya
    açılırken kullanılan erişim hakları daha sonra elde edilmiştir
-----
-----*/

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

void exit_sys(const char *msg);

int main(void)
{
    int fd;
    int result;

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

    if ((result = fcntl(fd, F_GETFL)) == -1)
        exit_sys("fcntl");

    printf("%s\n", (result & O_ACCMODE) == O_WRONLY ? "Yes\n" : "No\n");

    close(fd);

    return 0;
}

```

```

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

    exit(EXIT_FAILURE);
}

/
*-----
-----
    fcntl fonksiyonu ile aslında biz dup işlemi de yapabiliriz. Bunun için
    F_DUPFD command kodu kullanılmaktadır.
-----
-----*/

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

void exit_sys(const char *msg);

int main(void)
{
    int fd;
    int result;

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

    if ((result = fcntl(fd, F_DUPFD, 0)) == -1)
        exit_sys("fcntl");

    printf("New file descriptor: %d\n", result);

    close(fd);

    return 0;
}

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

    exit(EXIT_FAILURE);
}

/
*-----
-----
    1 Numaralı (Ctrl + Alt + F1) terminal aygıt sürücüsünü write only
    modunda açıp oraya bir şeyler yazmak
-----
-----*/

```

```
#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("/dev/tty1", O_WRONLY)) == -1)
        exit_sys("open");

    if (write(fd, "test\n", 5) == -1)
        exit_sys("write");

    close(fd);

    return 0;
}
```

```
void exit_sys(const char *msg)
```

```
{
    perror(msg);

    exit(EXIT_FAILURE);
}
```

```
/
```

```
*-----
```

```
0, 1, ve 2 numaralı dosya betimleyicileri terminal aygı sürücüsüne
ilişkin dosya nesnelerini göstermektedir.
```

```
0 Numaralı betimleyiciden okuma yapıldığında terminal okuma yapılmış
olur, 1 betimleyici kullanılarak yazma yapıldığında
yazılan şeyler terminale yani ekrana yazılırlar. 2 numaralı betimleyici
1 numaralı betimleyici dup yağılarak elde edilmiştir.
```

```
Dolayısıyla default durumda 2 numaralı betimleyici ile yazılanlar da
ekrana çıkacaktır.
```

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

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

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

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

```

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

    return 0;
}

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

    exit(EXIT_FAILURE);
}

/
*-----
-----
    0 numaralı betimleyiciden read fonksiyonuyla okum ayaparsak aslında
    terminal aygıt sürücüsünden yani
    klavyeden okuma yapmış oluruz.
-----
-----*/

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

void exit_sys(const char *msg);

int main(void)
{
    ssize_t result;
    char buf[1024 + 1];

    if ((result = read(0, buf, 1024)) == -1)
        exit_sys("write");

    buf[result] = '\0';
    puts(buf);

    return 0;
}

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

    exit(EXIT_FAILURE);
}

/
*-----
-----
    stderr dosyasının anlamı. Programdaki bütün hata mesajlarını stderr
    dosyasına yazdırmalıyız. Böylelikle
    normal çıktılarla hata mesajları yönlendirme yoluyla ayrıştırılabilir

```



```

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

#include <stdio.h>

int main(void)
{
    printf("stdout\n");
    fprintf(stderr, "stderr\n");

    return 0;
}

/
*-----
-----
    Aslında stdout dosyasının yönlendirilmesi çok kolaydır. Tek yapılacak
    şey 1 numaralı betimleyiciyi kapatıp sonra
    open ile yönlendirmenin yapılacağı dosyayı açmaktır. open'ın en düşük
    betimleyiciyi verdiğini anımsayınız
-----
-----*/

```

```

#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;
    int i;

    close(1);

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

    for (i = 0; i < 10; ++i)
        printf("%d\n", i);

    return 0;
}

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

    exit(EXIT_FAILURE);
}

```

```

/
*-----
-----
    Dosya yönlendirmelerinin dup2 fonksiyonuyla yapılması daha iyi bir
    tekniktir. Çünkü dup2 yönlendirme işlemini atomik bir
    biçimde yapmaktadır. Dolayısıyla multithreaded uygulamalarda ve
    reentrant uygulamalarda herhangi bir sorun oluşmaz.
-----
-----*/

#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;
    int i;

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

    if (dup2(fd, 1) == -1)
        exit_sys("dup2");

    for (i = 0; i < 10; ++i)
        printf("%d\n", i);

    close(fd);

    return 0;
}

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

    exit(EXIT_FAILURE);
}

/
*-----
-----
    stdin dosyasının (0 numaralı betimleyicinin) yönlendirilmesi
-----
-----*/

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

```

```

#include <unistd.h>

void exit_sys(const char *msg);

int main(void)
{
    int fd;
    int i;
    int val;

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

    if (dup2(fd, 0) == -1)
        exit_sys("dup2");

    while (scanf("%d", &val) != EOF)
        printf("%d\n", val);

    close(fd);

    return 0;
}

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

    exit(EXIT_FAILURE);
}

/
*-----
-----
    Dizin listesini elde edtmek için opendir, readdir ve closedir POSIX
    fonksiyonları kullanılmaktadır.
-----
-----*/

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

void exit_sys(const char *msg);

int main(void)
{
    DIR *dir;
    struct dirent *ent;

    if ((dir = opendir("/usr/include")) == NULL)
        exit_sys("opendir");

    errno = 0;

```

```

while ((ent = readdir(dir)) != NULL)
    printf("%s, %lu\n", ent->d_name, ent->d_ino);

if (errno)
    exit_sys("readdir");

closedir(dir);

return 0;
}

```

```

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

    exit(EXIT_FAILURE);
}

```

```

/
*-----
-----
Aksi söylenmediği sürece bir POSIX fonksiyonu başarı durumunda da errno
değişkenini et edebilir. Ancak pratikte GNU libc
kütüphanesi böylesi gereksiz bir set işlemi yapmamaktadır. Yukarıdaki
kodda bu anlamda küçük bir kusur vardır. printf başarı
durumunda standart bağlamında errno değişkenini set edebileceği için
errno değişkenine her yinelemede yeniden 0 atanması uygun olur
-----
-----*/

```

```

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

```

```

void exit_sys(const char *msg);

```

```

int main(void)
{
    DIR *dir;
    struct dirent *ent;

    if ((dir = opendir("/usr/include")) == NULL)
        exit_sys("opendir");

    while (errno = 0, (ent = readdir(dir)) != NULL)
        printf("%s\n", ent->d_name);

    if (errno)
        exit_sys("readdir");

    closedir(dir);

    return 0;
}

```

```

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

    exit(EXIT_FAILURE);
}

```

```

/
*-----
-----
    link fonksiyonun kullanımı
-----
-----*/

```

```

#include <stdio.h>
#include <stdlib.h>
#include <errno.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);
}

```

```

/
*-----
-----
    Dizin ağacını dolaşan bir örnek program
-----
-----*/

```

```

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

```

```

#include <dirent.h>

void exit_sys(const char *msg);
void walk_dir(const char *path, int level);

int main(int argc, char *argv[])
{
    walk_dir("/home/csd/Study", 0);

    return 0;
}

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

    exit(EXIT_FAILURE);
}

void walk_dir(const char *path, int level)
{
    DIR *dir;
    struct dirent *ent;
    struct stat finfo;

    if (chdir(path) == -1) {
        fprintf(stderr, "%s:%s\n", path, strerror(errno));
        return;
    }

    if ((dir = opendir(".")) == NULL) {
        perror("opendir");
        return;
    }

    while (errno = 0, (ent = readdir(dir)) != NULL) {
        if (!strcmp(ent->d_name, ".") || !strcmp(ent->d_name, ".."))
            continue;
        printf("%*s%s\n", level * 4, "", ent->d_name);

        if (lstat(ent->d_name, &finfo) == -1) {
            perror("stat");
            continue;
        }
        if (S_ISDIR(finfo.st_mode)) {
            walk_dir(ent->d_name, level + 1);
            chdir("..");
        }
    }
    closedir(dir);
}

```

```

/
*-----
-----
    Yukarıdaki dizin ağacını dolaşma örneğinin fonksiyon göstericisi
    kullanılarak genelleştirilmiş hali
-----
-----*/

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

void exit_sys(const char *msg);
void walk_dir(const char *path, int level, void (*proc)(const struct dirent
    *, int));

void disp(const struct dirent *ent, int level)
{
    printf("%*s%s\n", level * 4, "", ent->d_name);
}

int main(int argc, char *argv[])
{
    walk_dir("/home/csd/Study", 0, disp);

    return 0;
}

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

    exit(EXIT_FAILURE);
}

void walk_dir(const char *path, int level, void (*proc)(const struct dirent
    *, int))
{
    DIR *dir;
    struct dirent *ent;
    struct stat finfo;

    if (chdir(path) == -1) {
        fprintf(stderr, "%s:%s\n", path, strerror(errno));
        return;
    }

    if ((dir = opendir(".")) == NULL) {
        perror("opendir");
        return;
    }

```

```

while (errno = 0, (ent = readdir(dir)) != NULL) {
    if (!strcmp(ent->d_name, ".") || !strcmp(ent->d_name, ".."))
        continue;
    proc(ent, level);
    if (lstat(ent->d_name, &finfo) == -1) {
        perror("stat");
        continue;
    }
    if (S_ISDIR(finfo.st_mode)) {
        walk_dir(ent->d_name, level + 1, proc);
        chdir("..");
    }
}
closedir(dir);
}

/
*-----
-----
    Yukarıdaki dizi nağacını dolaşan programın callback fonksiyon tarafından
    dolaşımın durdurulabildiği versiyonu
-----
-----*/

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

void exit_sys(const char *msg);
int walk_dir(const char *path, int level, int (*proc)(const struct dirent *,
int));
int disp(const struct dirent *ent, int level);

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

    result = walk_dir("/home/csd/Study", 0, disp);
    printf("result: %d\n", result);

    return 0;
}

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

    exit(EXIT_FAILURE);
}

```



```

int walk_dir(const char *path, int level, int (*proc)(const struct dirent *,
int))
{
    DIR *dir;
    struct dirent *ent;
    struct stat finfo;
    int result;

    result = 1;
    if (chdir(path) == -1) {
        fprintf(stderr, "%s:%s\n", path, strerror(errno));
        goto EXIT2;
    }

    if ((dir = opendir(".")) == NULL) {
        perror("opendir");
        goto EXIT2;
    }

    while (errno = 0, (ent = readdir(dir)) != NULL) {
        if (!strcmp(ent->d_name, ".") || !strcmp(ent->d_name, ".."))
            continue;
        if (!proc(ent, level)) {
            result = 0;
            goto EXIT1;
        }

        if (lstat(ent->d_name, &finfo) == -1) {
            perror("stat");
            continue;
        }
        if (S_ISDIR(finfo.st_mode)) {
            result = walk_dir(ent->d_name, level + 1, proc);
            chdir("..");
            if (!result)
                goto EXIT1;
        }
    }
EXIT1:
    closedir(dir);

EXIT2:
    return result;
}

int disp(const struct dirent *ent, int level)
{
    printf("%*s%s\n", level * 4, "", ent->d_name);

    if (!strcmp(ent->d_name, "sample.c")) {
        printf("Buldu\n");
        return 1;
    }
}

```

```

        return 1;
    }

/
*-----
-----
    scandir fonksiyonun kullanımı
-----
-----*/

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

void exit_sys(const char *msg);

int fselect(const struct dirent *ent)
{
    if (ent->d_name[0] == 'a' || ent->d_name[0] == 's' || ent->d_name[0] ==
        'd')
        return 1;

    return 0;
}

int main(int argc, char *argv[])
{
    struct dirent **ent;
    int count;
    int i;

    if ((count = scandir("/usr/include", &ent, fselect, alphasort)) == -1)
        exit_sys("scandir");

    for (i = 0; i < count; ++i)
        puts(ent[i]->d_name);

    for (i = 0; i < count; ++i)
        free(ent[i]);

    free(ent);

    return 0;
}

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

    exit(EXIT_FAILURE);
}

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

```

scandir fonksiyonunda son parametrenin (karşılaştırma fonksiyonun)
oluşturulması

```
-----*/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <dirent.h>

void exit_sys(const char *msg);

int fcompare(const struct dirent **de1, const struct dirent **de2)
{
    return -strcmp((*de1)->d_name, (*de2)->d_name);
}

int fselect(const struct dirent *ent)
{
    if (ent->d_name[0] == 'a' || ent->d_name[0] == 's' || ent->d_name[0] ==
        'd')
        return 1;

    return 0;
}

int main(int argc, char *argv[])
{
    struct dirent **ent;
    int count;
    int i;

    if ((count = scandir("/usr/include", &ent, fselect, fcompare)) == -1)
        exit_sys("scandir");

    for (i = 0; i < count; ++i)
        puts(ent[i]->d_name);

    for (i = 0; i < count; ++i)
        free(ent[i]);

    free(ent);

    return 0;
}

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

    exit(EXIT_FAILURE);
}
```

```
/
*-----
-----
C standartlarına göre başlangıçta stdin ve stdout dosyaları interaktif
bir aygıtta yönlendirilmiş ise tam tamponlamalı modda
olamazlar. Satır tamponlamalı ya da sıfır tamponlamalı modda
olabilirler. Linux'ta standart C kütüphanesinde stdout ve stdin
dosyaları başlangıçta satır tamponlamalı moddadır. stderr ise
başlangıçta ister interaktif aygıtta yönlendirilmiş olsun isterse
normal bir dosyaya yönlendirilmiş olsun hiçbir zaman tam tamponlamalı
olamamaktadır. Aynı zamanda standartlar stdin ve stdout
dosyaları işin başında interaktif olmayan bir aygıtta yönlendirilmişse
kesinlikle bunların tam tamponlamalı modda olacağını
söylemektedir. Aşağıdaki örnekte ekranda bir şey göremeyebilirsiniz
-----
-----*/
```

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

```
int main(int argc, char *argv[])
{
    printf("this is a test");

    for (;;)
        ;

    return 0;
}
```

```
/
*-----
-----
Yukarıdaki gibi bir durumda bizim yazdıklarımızın görünmesi için fflush
fonksiyonunu çağırmamız ya da yazının sonuna '\n'
karakterini eklememiz gerekir.
-----
-----*/
```

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

```
int main(int argc, char *argv[])
{
    printf("this is a test");
    fflush(stdout);

    for (;;)
        ;

    return 0;
}
```

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

Yine C standartlarına göre stdin dosyasından okuma yapan fonksiyonlar isteğe bağlı olarak stdout dosyasını flush edebilirler.

```
-----*/
#include <stdio.h>

int main(int argc, char *argv[])
{
    printf("this is a test");
    fflush(stdout);

    getchar();

    return 0;
}

/
*-----
Aşağıdaki örnekte döngü süresince Linux sistemlerinde ekranda bir şey
göremeyebiliriz
-----*/
```

```
#include <stdio.h>

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

    for (i = 0; i < 20; ++i) {
        printf("%d ", i);
        sleep(1);
    }
    printf("\n");

    return 0;
}

/
*-----
Yukarıdaki anomaliyi ortadan kaldırmak için fflush yapmalıyız
-----*/
```

```
#include <stdio.h>

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

    for (i = 0; i < 20; ++i) {
        printf("%d ", i);
```

```

        fflush(stdout);
        sleep(1);
    }
    printf("\n");

    return 0;
}

/
*-----
-----
    setbuf standart C fonksiyonu açılan dosya için tamponu değiştirmekte ya
    da sıfır tamponlamalı moda geçmekte kullanılır.
    Aşağıdaki örnekte artık tampon olarak g_buf dizisi kullanılmaktadır.
    g_buf dizisinin başında "this is a test" yazısı
    bulunacaktır.
-----
-----*/

#include <stdio.h>

char g_buf[BUFSIZ];

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

    setbuf(stdout, g_buf);

    printf("this is a test\n");

    for (i = 0; i < 64; ++i) {
        if (i % 16 == 0)
            printf("%08X ", (unsigned int)i);
        printf("%02X%c", (unsigned int)g_buf[i], i % 16 == 15 ? '\n' : ' ');
    }
    if (i % 16 != 0)
        putchar('\n');

    return 0;
}

/
*-----
-----
    setbuf fonksiyonuyla dosyanın tamponlamalı modunu sıfır tamponlamalı mod
    haline getirebiliriz
-----
-----*/

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

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

```

```

    setbuf(stdout, NULL);

    printf("this is a test");
    sleep(10);

    return 0;
}

/
*-----
-----
    setvbuf standart C fonksiyonu setbuf fonksiyonunun gelişmiş bir
    biçimidir. Bu fonksiyonla biz hem taponun büyüklüğünü,
    hem de modunu ve yerini değiştirebilmekteyiz.
-----
-----*/

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

int main(int argc, char *argv[])
{
    FILE *f;
    int ch;

    if ((f = fopen("test.txt", "r")) == NULL) {
        fprintf(stderr, "cannot open file!\n");
        exit(EXIT_FAILURE);
    }
    if (setvbuf(f, NULL, _IONBF, BUFSIZ * 2) != 0) {
        fprintf(stderr, "cannot change buffer mode!\n");
        exit(EXIT_FAILURE);
    }

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

    fclose(f);

    return 0;
}

/
*-----
-----
    Bir dosyayı byte byte kopyalayan iki programın zaman karşılaştırması
    time komutuyla yapılmıştır. Bu işlemi tamponlu biçimde
    standart C fonksiyonlarıyla yaptığımızda POSIX read fonksiyonuna göre
    yaklaşık 10 kat daha hızlı çalışmaktadır.
-----
-----*/

/* cp1.c */

#include <stdio.h>

```

```

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

void exit_sys(const char *msg);

int main(int argc, char *argv[])
{
    int fds, fdd;
    char buf[1];
    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, 1)) > 0)
        if (write(fdd, buf, 1) == -1)
            exit_sys("write");

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

    close(fds);
    close(fdd);

    return 0;
}

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

    exit(EXIT_FAILURE);
}

/* cp2.c */

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

int main(int argc, char *argv[])
{
    FILE *fs, *fd;
    int ch;

```



```

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

if ((fs = fopen(argv[1], "r")) == NULL) {
    fprintf(stderr, "cannot open file: %s\n", argv[1]);
    exit(EXIT_FAILURE);
}

if ((fd = fopen(argv[2], "w")) == NULL) {
    fprintf(stderr, "cannot open file: %s\n", argv[2]);
    exit(EXIT_FAILURE);
}

while ((ch = fgetc(fs)) != EOF)
    if (fputc(ch, fd) == -1) {
        fprintf(stderr, "cannot write file!..\n");
        exit(EXIT_FAILURE);
    }

if (ferror(fs)) {
    fprintf(stderr, "cannot read file!..\n");
    exit(EXIT_FAILURE);
}

fclose(fs);
fclose(fd);

return 0;
}

```

```

/
*-----
-----
    Yukarıdaki örnekte cp2.c programında standart C dosyalarının tamponlama
    modunu Unbuffered hale getirirsek performans cp1.c
    ile yaklaşık aynı olmaktadır.
-----
-----*/

```

```

/* cp2.c */

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

int main(int argc, char *argv[])
{
    FILE *fs, *fd;
    int ch;

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

```

```

if ((fs = fopen(argv[1], "r")) == NULL) {
    fprintf(stderr, "cannot open file: %s\n", argv[1]);
    exit(EXIT_FAILURE);
}

if ((fd = fopen(argv[2], "w")) == NULL) {
    fprintf(stderr, "cannot open file: %s\n", argv[2]);
    exit(EXIT_FAILURE);
}

setbuf(fs, NULL);
setbuf(fd, NULL);

while ((ch = fgetc(fs)) != EOF)
    if (fputc(ch, fd) == -1) {
        fprintf(stderr, "cannot write file!..\n");
        exit(EXIT_FAILURE);
    }

if (ferror(fs)) {
    fprintf(stderr, "cannot read file!..\n");
    exit(EXIT_FAILURE);
}

fclose(fs);
fclose(fd);

return 0;
}

```

/

*-----

getchar fonksiyonu stdin tamponundan 1 byte alarak ona geri döner. eğer tampon boşsa gerçekten read fonksiyonu ile 0 numaralı betimleyiciyi kullanarak tampona okuma yapacaktır. Ancak tamponda en az 1 byte varsa getchar stdin dosyasından gerçek anlamda okuma yapmayacaktır. Hem Linux ve Mac OS X hem de Windows sistemlerinde genel olarak stdin dosyasının default tamponlama modu "satır tamponlamalı (line buffered)" moddur. Yani getchar eğer tamponda hiç byte kalmışsa bir satırlık bilgiyi klavyeden (stdin dosyasından) okuyarak tampona yerleştirmektedir. Aşağıdaki programda getchar ikinci çağrışta gerçek anlamda klavyeden okuma istemeyecektir.

-----*/

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

```

int main(void)
{
    int ch;

    ch = getchar();

```

```

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

    ch = getchar();
    printf("%c\n", ch);

    return 0;
}

/
*-----
-----
    Her getchar fonksiyonunda yeniden klavyeden giriş istenmesini istiyorsak
    bu durumda tampondaki byte'ları '\n'yi görene kadar
    okumalıyız. Bu işlemin daha pratik bir yöntemi yoktur.
-----
-----*/

```

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

```

int main(void)
{
    int ch;

    ch = getchar();
    printf("%c\n", ch);

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

    ch = getchar();
    printf("%c\n", ch);

    return 0;
}

```

```

/
*-----
-----
    stdin tamponunu boşaltan döngüyü bir fonksiyon olarak da yazabiliriz
-----
-----*/

```

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

```

void clear_stdin(void)
{
    int ch;

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

```

```

int main(void)
{
    int ch;

```

```

    ch = getchar();
    printf("%c\n", ch);

    clear_stdin();

    ch = getchar();
    printf("%c\n", ch);

    return 0;
}

```

```

/
*-----
-----
    Yukarıdaki fonksiyon bir makro biçiminde de yazılabilir (do-while'a
    dikkat)
-----
-----*/

```

```

#include <stdio.h>

```

```

#define clear_stdin() \
do \
{ \
    int ch; \
    while ((ch = getchar()) != '\n' && ch != EOF) \
        ; \
} while (0)

```

```

int main(void)
{
    int ch;

    ch = getchar();
    printf("%c\n", ch);

    clear_stdin();

    ch = getchar();
    printf("%c\n", ch);

    return 0;
}

```

```

/
*-----
-----
    gets fonksiyonu stdin dosyasından '\n' görene kadar ('\n' dahil olmak
    üzere) ya da EOF'agelinene kadar okuma yapar. Okunanları
    parametresiyle belirtilen adresten itibaren yerleştirir. Sonuna da '\0'
    ekler. gets hiçbir okuma yapmadan EOF ile karşılaşarsa
    NULL adrese geri dönmektedir. Ancak en az 1 karakter okuma yapmış ise
    parametresiyle belirtilen adresin aynısına geri döner.

```

gets fonksiyonu aşağıdaki gibi yazılmıştır

```
-----*/  
  
#include <stdio.h>  
  
char *mygets(char *buf)  
{  
    int ch;  
    size_t i;  
  
    for (i = 0; (ch = getc(stdin)) != '\n' && ch != EOF; ++i)  
        buf[i] = ch;  
  
    if (i == 0 && ch == EOF)  
        return NULL;  
  
    buf[i] = '\0';  
  
    return buf;  
}  
  
int main(void)  
{  
    char buf[1024];  
    char ch;  
  
    printf("Bir yazı giriniz:");  
    mygets(buf);  
  
    printf("Bir karakter giriniz:");  
    ch = getchar();  
  
    puts(buf);  
    printf("%c\n", ch);  
  
    return 0;  
}
```

```
/
```

gets_s fonksiyonunun gerçekleştirimi

```
-----*/  
  
#include <stdio.h>  
  
char *mygets_s(char *buf, size_t size)  
{  
    size_t i;  
    int ch = 0;  
  
    for (i = 0; i < size - 1; ++i) {  
        if ((ch = getc(stdin)) == '\n' || ch == EOF)
```

```

        break;
        buf[i] = ch;
    }

    if (i == 0 && ch == EOF)
        return NULL;

    buf[i] = '\\0';

    return buf;
}

```

```

int main(void)
{
    char buf[10];
    char ch;

    printf("Bir yazı giriniz:");
    mygets_s(buf, 10);
    puts(buf);

    return 0;
}

```

```

/
*-----
-----
scanf fonksiyonu başarılı biçimde yerleştirdiği parça sayısına geri
döner. Baştaki leading space'leri atmaktadır. Ancak
trailing space'lere dokunulmaz.
-----
-----*/

```

```

#include <stdio.h>

```

```

int main(void)
{
    int a = -1, b = -1;
    int result;

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

    return 0;
}

```

```

/
*-----
-----
scanf fonksiyonundan sonra gets ya da gets_s kullanırken dikkat ediniz
-----
-----*/

```

```

#include <stdio.h>

```

```
int main(void)
{
    int no;
    char name[1024];

    printf("Numaranızı giriniz:");
    scanf("%d", &no);

    printf("Adınızı giriniz:");
    gets(name);

    printf("No: %d, Ad: %s\n", no, name);

    return 0;
}
```

/

*-----

Yukarıdaki programın düzeltilmiş hali şöyledir

-----*/

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

```
void clear_stdin(void)
```

```
{
    int ch;

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

```
int main(void)
```

```
{
    int no;
    char name[1024];

    printf("Numaranızı giriniz:");
    scanf("%d", &no);

    clear_stdin();

    printf("Adınızı giriniz:");
    gets(name);

    printf("No: %d, Ad: %s\n", no, name);

    return 0;
}
```

/

*-----

scanf stdin dosyasından girdileri bir döngü içerisinde karakter karakter alıp işleme sokmaktadır. Eğer formak karakterine uygun bir girişle karşılaşmazsa o karakteri tampona geri bırakır ve işlemini sonlandırır

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

#include <stdio.h>

int main(void)
{
    int a = -1, b = -1;
    int result;
    char buf[1024];

    result = scanf("%d%d", &a, &b);    /* 10 ali */
    printf("result = %d, a = %d, b = %d\n", result, a, b); /* result = 1, a
    = 10, b = -1 */
    gets(buf);
    printf("\n%s\n", buf);    /* "ali "

    return 0;
}

/
*-----
-----
scanf fonksiyonuna uygun karakter girilmemesi sonucunda oluşacak hatalı
durumun ele alınması
-----
-----*/
```

```
#include <stdio.h>

int get_menu_option(void);
void clear_stdin(void);

int main(void)
{
    int option;

    for (;;) {
        option = get_menu_option();
        switch (option) {
            case 0:
                printf("Geçersiz giriş!\n\n");
                clear_stdin();
                break;
            case 1:
                printf("Kayıt ekleme işlemi\n\n");
                break;
            case 2:
                printf("Kayıt bul işlemi\n\n");
                break;
            case 3:
```



```

        printf("Kayıt sil işlemi\n\n");
        break;
    case 4:
        goto EXIT;
    }
}

EXIT:
    return 0;
}

int get_menu_option(void)
{
    int option = 0;

    printf("1) Kayıt Ekle\n");
    printf("2) Kayıt Bul\n");
    printf("3) Kayıt Sil\n");
    printf("4) Çıkış\n");

    printf("Seçiminiz:");
    scanf("%d", &option);

    return option;
}

void clear_stdin(void)
{
    int ch;

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

/
*-----
-----
scanf fonksiyonunda format karakterlerinin yanın bir white space "white
space görmeyene kadar stdin'den okuma yap" anlamına gelmektedir.
Bu nedenle sonda yanlışlıkla bırakılan white space'ler ciddi sorun
oluşturur.
-----
-----*/

#include <stdio.h>

int main(void)
{
    int a = -1, b = -1;
    int result;

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

    return 0;
}

```

```

}

/
*-----
-----
scanf fonksiyonunda format karakterlerinin arasında white space olmayan
karakterler girişte de aynı pozisyonda bulundurulmak zorundadır.
-----
-----*/

#include <stdio.h>

int main(void)
{
    int day, month, year;

    scanf("%d/%d/%d", &day, &month, &year);
    printf("%02d/%02d/%02d\n", day, month, year);

    return 0;
}

/
*-----
-----
scanf fonksiyonuyla yönlendirme drumunda EOF ya da geçersiz bir giriş
görene kadar okuma yapma
-----
-----*/

#include <stdio.h>

int main(void)
{
    int val;
    int result;

    while ((result = scanf("%d", &val)) != EOF && result != 0)
        printf("%d\n", val);

    return 0;
}

/
*-----
-----
fileno fonksiyonu C tarzı stream'i parametre olarak alır. FILE yapısı
içerisindeki dosya betimleyicisi ile geri döner.
-----
-----*/

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

```

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

```
int main(void)
{
    FILE *f;
    int fd;
    char buf[100 + 1];
    int result;

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

    if ((fd = fileno(f)) == -1)
        exit_sys("fileno");

    if ((result = read(fd, buf, 100)) == -1)
        exit_sys("read");
    buf[result] = '\0';
    puts(buf);

    fclose(f);

    return 0;
}
```

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

    exit(EXIT_FAILURE);
}
```

```
/
```

```
*-----
```

```
fdopen fonksiyonu işlevsel olarak fileno fonksiyonunun tersini
yapmaktadır. Yani bizden bir betimleyici alıp bize FILE *
verir. Dolayısıyla biz artık standart dosya fonksiyonlarını
kullanabiliriz.
```

```
-----*
```

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

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

```
int main(void)
{
    int fd;
    FILE *f;
    int ch;
```

```

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

    if ((f = fdopen(fd, "r")) == NULL)
        exit_sys("fdopen");

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

    fclose(f);

    return 0;
}

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

    exit(EXIT_FAILURE);
}

/
*-----
-----
    Programın iki noktası arasında geçen zaman o anda sistemin yüküne bağlı
    olarak ciddi farklılıklar gösterebilir
-----
-----*/

#include <stdio.h>
#include <time.h>

int main(void)
{
    long i;
    time_t start, end;

    start = time(NULL);

    for (i = 0; i < 1000000000; ++i)
        ;

    end = time(NULL);

    printf("%lu\n", (unsigned long)(end - start));

    return 0;
}

/
*-----
-----
    Thread'ler (ya da thread yoksa prosesler) IO yoğun ve CPU yoğun olmak
    üzere kabaca iki ayrılabilir.

```

IO yoğun thread'ler onalara verilen quanta süresini çok az kullanıp hemen bloke olurlar. CPU yoğun thread'ler kendilerine verilen quanta süresini büyük ölçüde harcarlar. Genel olarak thread'ler IO yoğun olma eğilimindedir. Ancak bir döngü içerisinde hiç bloke olmayan thread'ler CPU yoğundur. IO yoğun thread'ler yüzlerce olsa bile sistemde ciddi bir yavaşlığa yol açmazlar. Örneğin aşağıdaki program IO yoğun bir thread'e örnektir.

```
-----*/

#include <stdio.h>

int main(void)
{
    int val;

    for (;;) {
        scanf("%d", &val);
        if (val == 0)
            break;
        printf("%d", val * val);
    }

    return 0;
}

/
*-----
Kütüphanelerdeki sleep gibi fonksiyonlar aslında CPU'yu meşgul ederek
bekleme yapmazlar. Blokeye yol açarak bekleme yaparlar.
Bu nedenle aşağıdaki gibi programlar IO yoğundur ve zaman harcama
bakımından sisteme hiç yük bindirmezler
-----*/

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

int main(void)
{
    int i;

    for (i = 0; i < 10; ++i) {
        sleep(1);
        printf("%d ", i);
        fflush(stdout);
    }
    printf("\n");

    return 0;
}
```

```

/
*-----
-----
fork fonksiyonu ile yeni bir prosesin yaratılması
-----
-----*/

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

void exit_sys(const char *msg);

int main(void)
{
    pid_t pid;

    printf("before fork\n");

    if ((pid = fork()) == -1)
        exit_sys("fork");

    if (pid != 0) {
        /* parent */
        printf("parent process...\n");
    }
    else {
        /* child */
        printf("child process...\n");
    }

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

    return 0;
}

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

    exit(EXIT_FAILURE);
}

/
*-----
-----
fork işlemi
-----
-----*/

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

void exit_sys(const char *msg);

```

```

int main(void)
{
    pid_t pid;

    printf("before fork\n");

    if ((pid = fork()) == -1)
        exit_sys("fork");

    if (pid != 0) {
        printf("Parent's parent process id: %ld\n", (long)getppid());
        printf("Parent process process id: %ld\n", (long)getpid());
        printf("Parent process fork return value: %ld\n", (long)pid);
    }
    else {
        printf("Child process id: %ld\n", (long) getpid());
        printf("Child's parent process id: %ld\n", (long)getppid());
    }

    return 0;
}

```

```

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

    exit(EXIT_FAILURE);
}

```

```

/
*-----
-----
Aşağıdaki örnekte ekrana toplamda kaç tane "ends..." yazısı çıkar?
-----
-----*/

```

```

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

void exit_sys(const char *msg);

int main(void)
{
    if (fork() == -1)
        exit_sys("fork");

    if (fork() == -1)
        exit_sys("fork");

    printf("ends..\n");

    return 0;
}

```

```

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

    exit(EXIT_FAILURE);
}

/
*-----
-----
    Aşağıdaki örnekte ekrana toplamda kaç tane "ends..." yazısı çıkar?
-----
-----*/

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

void exit_sys(const char *msg);

int main(void)
{
    int i;

    for (i = 0; i < 3; ++i)
        if (fork() == -1)
            exit_sys("fork");

    printf("ends..\n");

    return 0;
}

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

    exit(EXIT_FAILURE);
}

/
*-----
-----
    fork işleminden sonra artık üst proses ile alt prosesin sanal bellek
    alanları tamamen ayrılmıştır. Yani bunlar birbirlerinden
    izole edilmiş aynı kod ve data'ya sahip iki farklı proses durumundadır.
    Dolayısıyla fork işleminden sonra üst prosesteki
    değişiklikler alt proses alt prosesteki değişiklikler üst proses
    etkilemez. Yani fork işlemeinden sonra üst ve alt prosesin
    her hangi iki farklı prosesten çalışma anlamında bir farkı yoktur.
-----
-----*/

#include <stdio.h>

```



```

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

void exit_sys(const char *msg);

int g_x;

int main(void)
{
    pid_t pid;

    g_x = 100;
    if ((pid = fork()) == -1)
        exit_sys("fork");

    if (pid != 0) {
        g_x = 200;

    }
    else {
        sleep(1);
        printf("%d\n", g_x);    /* 100 */
    }

    return 0;
}

```

```

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

    exit(EXIT_FAILURE);
}

```

```

/
*-----
-----
fork işlemi sırasında üst prosesin dosya betimleyici tablosundaki file
nesne adresleri alt prosesin dosya betimleyici
tablosuna kopyalanır. Yani fork işleminden sonra üst ve alt proseslerin
dosya betimleyici tabloları aynı dosya nesnesini (struct file)
gösterir hale gelmektedir. Bu durumda bu proseslerden biri örneğin dosya
göstericisini konumlandırırırsa diğeri de bunu konumlandırmış
olarak görecektir.
-----
-----*/

```

```

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

void exit_sys(const char *msg);

int main(void)

```

```

{
    pid_t pid;
    int fd;

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

    if ((pid = fork()) == -1)
        exit_sys("fork");

    if (pid != 0) {
        lseek(fd, 200, SEEK_SET);
    }
    else {
        char buf[100 + 1];
        ssize_t result;

        sleep(1);

        if ((result = read(fd, buf, 100)) == -1)
            exit_sys("read");
        buf[result] = '\0';
        puts(buf);
    }

    return 0;
}

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

    exit(EXIT_FAILURE);
}

/
*-----
-----
    execl Fonksiyonun kullanımı
-----
-----*/

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

void exit_sys(const char *msg);

int main(void)
{
    printf("program begins...\n");

    if (execl("/bin/ls", "/bin/ls", "-l", "-i", (char *)NULL) == -1)
        exit_sys("execl");
}

```

```

    printf("Unreachable code!..\n");

    return 0;
}

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

    exit(EXIT_FAILURE);
}

/
*-----
-----
    execl fonksiyonun kullanımı
-----
-----*/

/* sample.c */

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

void exit_sys(const char *msg);

int main(void)
{
    printf("sample begins...\n");

    if (execl("mample", "mample", "ali", "veli", "selami", (char *)NULL) ==
        -1)
        exit_sys("execl");

    printf("Unreachable code!..\n");

    return 0;
}

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

    exit(EXIT_FAILURE);
}

/* mample.c */

#include <stdio.h>

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

```

```

    printf("mample is running...\n");

    for (i = 0; i < argc; ++i)
        printf("%s\n", argv[i]);

    return 0;
}

/
*-----
-----
    exec sonrasında yeni bir proses yaratılmamaktadır. Yalnızca mevcut
    proses hayatını başka bir program koduyla
    devam ettirmektedir
-----
-----*/

/* sample.c */

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

void exit_sys(const char *msg);

int main(void)
{
    printf("sample begins...\n");

    printf("sample process id: %ld\n", (long)getpid());

    if (execl("mample", "mample", (char *)NULL) == -1)
        exit_sys("execl");

    printf("Unreachable code!...\n");

    return 0;
}

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

    exit(EXIT_FAILURE);
}

/* mample.c */

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

int main(void)
{
    printf("mample is running...\n");

```

```

    printf("mample process id: %ld\n", (long)getpid());

    return 0;
}

/
*-----
-----
    execv fonksiyonu çalıştırılacak programın komut satırı argümanlarını bir
    dizi biçiminde bizden ister.
    Bu dizinin sonu NULL adresle bitmelidir.
-----
-----*/

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

void exit_sys(const char *msg);

int main(void)
{
    char *args[] = {"/bin/ls", "-l", "-i", NULL};

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

    if (execv("/bin/ls", args) == -1)
        exit_sys("execl");

    printf("Unreachable code!..\n");

    return 0;
}

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

    exit(EXIT_FAILURE);
}

/
*-----
-----
    Çalıştıracağı programı ve o programın komut satırı argümanlarını
    parametre olarak alan program örneği. Burada execl fonksiyonunun
    kullanılamayacağına onun yerine execv fonksiyonunun kullanılabileceğine
    dikkat ediniz.
-----
-----*/

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

```

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

```
int main(int argc, char *argv[])
{
    if (execv(argv[1], &argv[1]) == -1)
        exit_sys("execv");

    printf("Unreachable code!...\n");

    return 0;
}
```

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

    exit(EXIT_FAILURE);
}
```

```
/
```

```
*-----
```

```
Aslında genellikle (ama her zaman değil) yalnızca fork ya da yalnızca
exec kullanılmaz. fork ve exec birlikte kullanılır.
Üst proses çalışmaya devam edip alt prosesin başka bir kodu
çalıştırabilmesi için önce bir kez fork yapılır, alt proseste
exec uygulanır.
```

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

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

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

```
int main(void)
{
    pid_t pid;

    if ((pid = fork()) == -1)
        exit_sys("fork");

    if (pid == 0)
        if (execl("/bin/ls", "/bin/ls", "-l", "-i", (char *)NULL) == -1)
            exit_sys("execl");

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

    return 0;
}
```

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

```

    exit(EXIT_FAILURE);
}

/
*-----
-----
    Yukarıdaki programda && operatörü de kullanabilirdik. && operatörünün
    önce sol tarafındaki operand yapılır. Sol tarafındaki operand
    0 ise (yanlış ise) zaten sağ tarafındaki operand hiç yapılmaz. Aşağıdaki
    program yukarıdaki ile eşdeğerdir.
-----
-----*/

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

void exit_sys(const char *msg);

int main(void)
{
    pid_t pid;

    if ((pid = fork()) == -1)
        exit_sys("fork");

    if (pid == 0 && execl("/bin/ls", "/bin/ls", "-l", "-i", (char *)NULL) ==
        -1)
        exit_sys("execl");

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

    return 0;
}

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

    exit(EXIT_FAILURE);
}

/
*-----
-----
    exec işleminde exec öncesi açılmış olan dosyalar default durumda
    kapatılmamaktadır. Aşağıdaki örnekte exec işleminde dosyanın
    aslında kapatılmadığı gösterilmeye çalışılmıştır.
-----
-----*/

/* sample.c */

#include <stdio.h>

```

```

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

void exit_sys(const char *msg);

int main(void)
{
    int fd;
    pid_t pid;
    char sfd[10];

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

    sprintf(sfd, "%d", fd);
    if (execl("mample", "mample", sfd, (char *)NULL) == -1)
        exit_sys("execl");

    printf("Unreachable code...\n");

    return 0;
}

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

    exit(EXIT_FAILURE);
}

/* mample.c */

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

int main(int argc, char *argv[])
{
    char buf[100 + 1];
    ssize_t result;
    int fd;

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

    fd = (int)strtol(argv[1], NULL, 10);

    if ((result = read(fd, buf, 100)) == -1) {
        perror("read");
        exit(EXIT_FAILURE);
    }
    buf[result] = '\0';

```



```

    puts(buf);

    return 0;
}

/
*-----
-----
    Daha dosya açılırken eğer açış modeunda O_CLOEXEC bayrağı kullanılırsa
    dosyanın "close on exec" bayrağı set edilmiş olur.
    Böylece artık exec işlemlerinde dosya otomatik biçimde
    kapatılacaktır. Ancak O_CLOEXEC bayrağı POSIX standartlarında yoktur.
    Linux sistemlerinde bulunmaktadır. Aşağıdaki programda çalıştırılan
    mample programının açık dosyayı kullanmadığına dikkat ediniz.
-----
-----*/

/* sample.c */

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

void exit_sys(const char *msg);

int main(void)
{
    int fd;
    pid_t pid;
    char sfd[10];

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

    sprintf(sfd, "%d", fd);
    if (execl("mample", "mample", sfd, (char *)NULL) == -1)
        exit_sys("execl");

    printf("Unreachable code...\n");

    return 0;
}

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

    exit(EXIT_FAILURE);
}

/* mample.c */

#include <stdio.h>

```

```

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

int main(int argc, char *argv[])
{
    char buf[100 + 1];
    ssize_t result;
    int fd;

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

    fd = (int)strtol(argv[1], NULL, 10);

    if ((result = read(fd, buf, 100)) == -1) {
        perror("read");
        exit(EXIT_FAILURE);
    }
    buf[result] = '\0';

    puts(buf);

    return 0;
}

/
*-----
-----
Dosyanın "close on exec" bayrağını set etmenin taşınabilir bir yolu
fcntl fonksiyonunu kullanmaktır. fcntl fonksiyonunda
command kod olarak F_SETFD girilirse dosyanın "betimleyici bayrakları
(file descriptor fşags)" set edilir. Şimdilik POSIX
standartlarında betimleyici bayrağı olarak yalnızca FD_CLOEXEC bayrağı
tanımlanmıştır. Ancak ileriye doğru uyumu korumak için
bu bayrağı set ederken önce get edip bir düzeyinde OR işlemi uygulamak
iyi tekniktir.
-----
-----*/

/* sample.c */

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

void exit_sys(const char *msg);

int main(void)
{
    int fd;
    pid_t pid;
    char sfd[10];

```

```

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

    if (fcntl(fd, F_SETFD, fcntl(fd, F_GETFD)|FD_CLOEXEC) == -1)
        exit_sys("fcntl");

    sprintf(sfd, "%d", fd);
    if (execl("mample", "mample", sfd, (char *)NULL) == -1)
        exit_sys("execl");

    printf("Unreachable code...\n");

    return 0;
}

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

    exit(EXIT_FAILURE);
}

/* mample.c */

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

int main(int argc, char *argv[])
{
    char buf[100 + 1];
    ssize_t result;
    int fd;

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

    fd = (int)strtol(argv[1], NULL, 10);

    if ((result = read(fd, buf, 100)) == -1) {
        perror("read");
        exit(EXIT_FAILURE);
    }
    buf[result] = '\0';

    puts(buf);

    return 0;
}

```

```
/
*-----
-----
Prosesin sonlandırılması C'de normal olarak exit standart C fonksiyonu
ile yapılmalıdır. exit standart C fonksiyonu açılmış olan
stdio dosyalarını flush eder ve kapatır. Ayrıca başka birtakım
sonlandırma işlemlerini de yapabilmektedir. Biz özellikle bir dosya
açmışken, onun içine birşeyler yazmışsak doğrudan _exit fonksiyonunu
kullanırken dikkat etmeliyiz.
-----
-----*/
```

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

void exit_sys(const char *msg);

int main(void)
{
    FILE *f;

    if ((f = fopen("test.txt", "w")) == NULL)
        exit_sys("fopen");

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

    _exit(0);

    return 0;
}

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

    exit(EXIT_FAILURE);
}
```

```
/
*-----
-----
Tabii programcı _exit fonksiyonunu çağırmadan önce dosyayı flush
edebilir ya da kapatabilir. Bu durumda olumsuz bir durumla
karşılaşılmaz.
-----
-----*/
```

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

void exit_sys(const char *msg);

int main(void)
```

```

{
    FILE *f;

    if ((f = fopen("test.txt", "w")) == NULL)
        exit_sys("fopen");

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

    _exit(0);

    return 0;
}

```

```

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

    exit(EXIT_FAILURE);
}

```

```

/
*-----
-----
Üst proses fork işlemi yaptığında alt proseste tüm stdio tamponlarının
da kopyalarının oluşacağına dikkat ediniz. Bu
nedenle alt proseste exit yerine çoğu kez _exit POSIX fonksiyonu tercih
edilmelidir.
-----
-----*/

```

```

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

```

```

void exit_sys(const char *msg);

```

```

int main(void)
{
    FILE *f;
    pid_t pid;

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

    fprintf(f, "xxxxx\n");

    /* fflush(f); */

    if ((pid = fork()) == -1)
        exit_sys("fork");

    if (pid == 0) {
        printf("child\n");
        _exit(EXIT_FAILURE);
    }
}

```

```

    }

    printf("parent\n");

    return 0;
}

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

    exit(EXIT_FAILURE);
}

/
*-----
-----
Alt prosteşte exec işlemi yapıldığında zaten tüm bellek alanı
boşaltılmaktadır. Dolayısıyla yukarıdaki örnekte olduğu gibi
bir flush problemi ortaya çıkmayacaktır. exec ile çalıştırılan program
exit fonksiyonuyla sonlandırılmış olsa da bir problem
oluşmaz.
-----
-----*/

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

void exit_sys(const char *msg);

int main(void)
{
    FILE *f;
    pid_t pid;

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

    fprintf(f, "xxxxx\n");

    if ((pid = fork()) == -1)
        exit_sys("fork");

    if (pid == 0) {
        if (execl("tample", "tample", (char *)NULL) == -1)
            _exit(EXIT_FAILURE);
    }

    printf("parent\n");

    return 0;
}

void exit_sys(const char *msg)

```

```

{
    perror(msg);

    exit(EXIT_FAILURE);
}

/
*-----
-----
wait fonksiyonu ilk alt proses sonlanana kadar beker. Alt prosesin
sonlanma nedenini ve exit kodunu alarak geri döner.
Ancak normal sonlanmalarda exit kodu oluşmaktadır. Alt prosesin normal
bir biçimde sonlandığını anlayabilmek için
WIFEXITED makrosu kullanılmaktadır. Alt prosesin exit kod ise
WEXITSTATUS makrosuyla elde edilir.
-----
-----*/

/* sample.c */

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

void exit_sys(const char *msg);

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

    printf("sample starts...\n");

    if ((pid = fork()) == -1)
        exit_sys("fork");

    if (pid == 0 && execl("mample", "mample", (char *)NULL) == -1)
        exit_sys("execl");

    if (wait(&status) == -1)
        exit_sys("wait");

    if (WIFEXITED(status))
        printf("Child terminated normally: %d\n", WEXITSTATUS(status));
    else
        printf("Child terminated abnormally!...\n");

    return 0;
}

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

```

```

        exit(EXIT_FAILURE);
    }

/* mample.c */

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

int main(void)
{
    int i;

    printf("mample starts\n");

    for (i = 0; i < 10; ++i) {
        printf("%d\n", i);
        sleep(1);
    }

    return 100;
}

/
*-----
-----
    Üst proses kaç kere alt proses yaratmışsa o kadar wait uygulamalıdır. Bu
    durum biraz sıkıntılı olabilmektedir. Tabii
    eğer alt proses zaten sonlanmışsa wait hiç bekleme yapmaz.
-----
-----*/

/* sample.c */

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

void exit_sys(const char *msg);

int main(void)
{
    pid_t pid1, pid2, pid_result;
    int status;
    int i;

    printf("sample starts...\n");

    if ((pid1 = fork()) == -1)
        exit_sys("fork");

    if (pid1 == 0 && execl("mample", "mample", (char *)NULL) == -1)
        exit_sys("execl");

```



```

    if ((pid2 = fork()) == -1)
        exit_sys("fork");

    if (pid2 == 0 && execl("/bin/ls", "/bin/ls", "-l", (char *)NULL) == -1)
        exit_sys("execl");

    for (i = 0; i < 2; ++i) {
        if ((pid_result = wait(&status)) == -1)
            exit_sys("wait");

        if (WIFEXITED(status))
            printf("%s terminated normally: %d\n", pid_result == pid1 ?
                "mample" : "ls", WEXITSTATUS(status));
        else
            printf("%s Child terminated abnormally!..\n", pid_result == pid1
                ? "mample" : "ls");
    }

    return 0;
}

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

    exit(EXIT_FAILURE);
}

/* mample.c */

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

int main(void)
{
    int i;

    printf("mample starts\n");

    for (i = 0; i < 10; ++i) {
        printf("%d\n", i);
        sleep(1);
    }

    return 100;
}

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

```

waitpid fonksiyonu wait fonksiyonun daha gelişmiş bir biçimidir. waitpid fonksiyonu ile biz proses id'sini bildiğimiz

herhangi bir alt prosesin sonlanmasını bekleyebiliriz: waitpid fonksiyonun birinci parametresi değişik seçenekler sunmaktadır. Son parametre tipik olarak 0 geçilebilir ya da WNOHANG geçilebilir. WNOHANG waitpid fonksiyonunun bloke olmasını engellemektedir.

```
-----*/

/* sample.c */

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

void exit_sys(const char *msg);

int main(void)
{
    pid_t pid1, pid2, pid_result;
    int status;
    int i;

    printf("sample starts...\n");

    if ((pid1 = fork()) == -1)
        exit_sys("fork");

    if (pid1 == 0 && execl("mample", "mample", (char *)NULL) == -1)
        exit_sys("execl");

    if ((pid2 = fork()) == -1)
        exit_sys("fork");

    if (pid2 == 0 && execl("/bin/ls", "/bin/ls", "-l", (char *)NULL) == -1)
        exit_sys("execl");

    if (waitpid(pid1, &status, 0) == -1)
        exit_sys("wait");

    if (WIFEXITED(status))
        printf("mample terminated normally: %d\n", WEXITSTATUS(status));
    else
        printf("mample terminated abnormally!..\n");

    if (waitpid(pid2, &status, 0) == -1)
        exit_sys("wait");

    if (WIFEXITED(status))
        printf("ls terminated normally: %d\n", WEXITSTATUS(status));
    else
        printf("ls terminated abnormally!..\n");

    return 0;
}
```

```

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

    exit(EXIT_FAILURE);
}

```

```

/* mample.c */

```

```

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

```

```

int main(void)
{
    int i;

    printf("mample starts\n");

    for (i = 0; i < 10; ++i) {
        printf("%d\n", i);
        sleep(1);
    }

    return 100;
}

```

```

/

```

```

*-----
-----

```

Üst proses çalışmaya devam ederken alt proses sonlanmışsa ve üst proses wait fonksiyonlarını uygulamamışsa alt proses zombie durumda olur. İşletim sistemi sonlanan proseslerin exit kodlarını wait fonksiyonlarıyla onların üst prosesleri alır diye onlara ilişkin proses kontrol bloklarını ve process id değerlerini boşaltmamaktadır. Bu da patolojik bir durumdur. Zombie'lik tipik olarka wait fonksiyonlarıyla engellenebilir. Ancak SIGCHLD sinyali yoluyla da otomatik engelleme yöntemleri vardır. Aşağıdaki program zombie proses oluşturmaktadır. Bu programı çalıştırıp başka bir terminalden ps -al komutu ile alt prosesin durumuna dikkat ediniz.

```

-----*/

```

```

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

```

```

void exit_sys(const char *msg);

```

```

int main(void)
{
    pid_t pid;

```

```

    printf("sample starts...\n");

    if ((pid = fork()) == -1)
        exit_sys("fork");

    if (pid == 0)
        exit(EXIT_SUCCESS);

    printf("Child is zombie now. Press ENTER to exit...\n");
    getchar();

    return 0;
}

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

    exit(EXIT_FAILURE);
}

/
*-----
-----
    exec fonksiyonlar ile çalıştırılabilir (executable) olmayan dosyalar
    da çalıştırılabilir. exec fonksiyonları önce çalıştırılmak
    istenen dosyanın çalıştırılabilir olup olmadığına (yani ELF formatına
    sahip olup olmadığına) bakmaktadır. Eğer dosya çalıştırılabilir
    değilse bu durumda onun ilk satırını okuyup orada belirtilen programı
    çalıştırmaları. Asıl dosyanın yol ifadesini de o dosyaya
    komut satırı argümanı olarak verirler.
-----
-----*/

/* sample.c */

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

void exit_sys(const char *msg);

int main(void)
{
    pid_t pid;

    printf("sample starts...\n");

    if ((pid = fork()) == -1)
        exit_sys("fork");

    if (pid == 0 && execl("sample.py", "sample.py", (char *)NULL) == -1)
        exit_sys("execl");

```

```

    if (wait(NULL) == -1)
        exit(EXIT_FAILURE);

    return 0;
}

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

    exit(EXIT_FAILURE);
}

/* sample.py */

#! /usr/bin/python

for i in range(10):
    print(i)

/
*-----
-----
Kabuk programı (bash) önce dosyayı fork ve exec yaparak çalıştırmaya
çalışır. Eğer exec başarısız olursa onu bir "shell script"
olarak düşünür ve doğrudan kendisi açarak bir scrip biçiminde
çalıştırır. Bu durumda biz shell script'lerin başına shebang
koymasak da onu komu satırında çalıştırabiliriz. Ancak exec yaparak
çalıştıramayız. Aşağıda dosya bu nedenle komut satırında
çalıştırılabilir ancak exec yapılamaz
-----
-----*/

# sample.sh

for i in 10 20 30 40 50
do
    echo $i
done

/
*-----
-----
Tabii bash script dosyalarının da yine shebang'e sahip olması iyi bir
tekniktir
-----
-----*/

#! /bin/bash

for i in 10 20 30 40 50
do
    echo $i
done

```

```
/
*-----
-----
Biz bir shebang'li text dosyayı çalıştırırken komut satırı argümanı da
verebiliriz. Bu durumda bu komut satırı argümanları
shebang'te belirtilen programın komut satırı argümanları olur. Örneğin
test.txt dosyasında shebang olarak belirtilen dosya
mample ise ./test.txt ali veli selami aslında mample test.txt ali veli
selami durumuna gelir. Aşağıdaki programda sample programı
execl ile test.txt dosyasını çalıştırmaktadır. Bu da mample programının
çalıştırılmasına yol açacaktır.
-----
-----*/
```

```
/* sample.c */
```

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

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

```
int main(void)
```

```
{
```

```
    pid_t pid;
```

```
    printf("sample starts...\n");
```

```
    if ((pid = fork()) == -1)
        exit_sys("fork");
```

```
    if (pid == 0 && execl("test.txt", "test.txt", "ali", "veli", "selami",
        (char *)NULL) == -1)
        exit_sys("execl");
```

```
    if (wait(NULL) == -1)
        exit(EXIT_FAILURE);
```

```
    return 0;
```

```
}
```

```
void exit_sys(const char *msg)
```

```
{
```

```
    perror(msg);
```

```
    exit(EXIT_FAILURE);
```

```
}
```

```
/* test.txt */
```

```
#!/home/csd/Study/Unix-Linux-SysProg/mample
```

```
/* mample.c */
```

```

#include <stdio.h>

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

    for (i = 0; i < argc; ++i)
        printf("%s\n", argv[i]);

    return 0;
}

/
*-----
-----
    shebang'te belirtilen programa da shebanh satırında komut satırı
    argümanı verilebilir. Bu argümanlar tek bir komut satırı
    argümanı olarak shebang'te belirtilen programa argv[1] biçiminde
    aktarılmaktadır. Örneğin shabang şöyle olsun:

    #! /home/csd/Study/Unix-Linux-SysProg/mample ankara istanbul adana

    Biz de test.txt dosyasını şöyle çalıştırmış olalım:

    ./test.txt ali veli selami

    Bu durum aşağıdaki gibi bir çalıştırmayla eşdeğer olacaktır:

    /home/csd/Study/Unix-Linux-SysProg/mample "ankara istanbul adana" ./
    test.txt ali veli selami

-----
-----*/
/* test.txt */

#! /home/csd/Study/Unix-Linux-SysProg/mample ankara istanbul adana

/
*-----
-----
    Aşağıdaki programda shebag olarak bir dosyanın içeriğini yazdıran bir
    program kullanılmıştır
-----
-----*/

/* test.txt

/* mycat.c */

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

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

```

```

int ch;

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

if ((f = fopen(argv[1], "r")) == NULL) {
    fprintf(stderr, "cannot open file: %s\n", argv[1]);
    exit(EXIT_FAILURE);
}

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

if (ferror(f)) {
    fprintf(stderr, "Cannot read file: %s\n", argv[1]);
    exit(EXIT_FAILURE);
}

return 0;
}

```

```
/* test.txt */
```

```
#! /home/csd/Study/Unix-Linux-SysProg/mycat
```

```

bugün hava çok güzel
corona virüslerinin hepsi öldü

```

```
/
```

```

*-----
-----

```

```

Shell programları -c ile tek bir komutu çalıştırıp sonlanabilmektedir.
Dolayısıyla biz komut satırında verdiğimiz tüm komutları
aslında shell programına çalıştırtabiliriz.

```

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

```

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

```

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

```

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

```

```
    pid_t pid;
```

```

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

```



```

    if ((pid = fork()) == -1)
        exit_sys("fork");

    if (pid == 0 && execl("/bin/bash", "/bin/bash", "-c", argv[1], (char
        *)NULL) == -1)
        exit_sys("execl");

    if (wait(NULL) == -1)
        exit(EXIT_FAILURE);

    return 0;
}

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

    exit(EXIT_FAILURE);
}

/
*-----
-----
system standart bir C fonksiyonudur. /bin/sh shell programını -c
seçeneği ile çalıştırır. Yani bizim fonksiyona verdiğimiz
komut system tarafından aslında shell programına çalıştırılmaktadır.
system fork ya da waitpid fonksiyonlarında başarısız
olursa -1 değerine exec fonksiyonlarında başarısız olursa 127 değerine
geri döner. Eğer başarılı olursa shell programının exit
koduyla geri dönmektedir. Zaten shell de -c seçeneği ile son
çalıştırdığı komutun exit koduyla geri döner.
-----
-----*/

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

void exit_sys(const char *msg);

int mysystem(const char *command)
{
    pid_t pid;
    int status;

    if (command == NULL)
        return 1;

    if ((pid = fork()) == -1)
        return -1;

    if (pid == 0 && execl("/bin/sh", "/bin/sh", "-c", command, (char *)NULL)
        == -1)
        _exit(127); /* Neden exit değil de _exit? */

```

```

        if (waitpid(pid, &status, 0) == -1)
            return -1;

        return status;
}

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

    result = mysystem("ls -l");
    if (result == -1 || result == 127)
        exit_sys("mysystem");

    printf("mysystem terminated normally with return value %d\n", result);

    return 0;
}

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

    exit(EXIT_FAILURE);
}

/
*-----
-----
    getenv standart C fonksiyonu (aynı zamanda POSIX fonksiyonu) çevre
    değişkeninin ismini (anahtarı) alıp ona karşı gelen
    değeri bize verir. Eğer öyle bir çevre değişkeni yoksa getenv NULL
    adrese geri dönmektedir. getenv fonksiyonu başarısızlık
    durumunda errno değerini set etmez.
-----
-----*/

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

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

    if (argc == 1) {
        fprintf(stderr, "wrong number of arguments!..\n");
        exit(EXIT_FAILURE);
    }

    for (i = 1; i < argc; ++i) {
        if ((result = getenv(argv[i])) == NULL) {
            fprintf(stderr, "cannot get environment variable: %s\n",
                argv[i]);

```

```

        continue;
    }
    printf("%s --> %s\n", argv[i], result);
}

return 0;
}

/
*-----
-----
    setenv POSIX fonksiyonu prosesin çevre değişken listesine yeni bir
    anahtar-değer çifti ekler
-----
-----*/

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

void exit_sys(const char *msg);

int main(void)
{
    char *result;

    if (putenv("city=istanbul") == -1)
        exit_sys("setenv");

    if ((result = getenv("city")) == NULL) {
        fprintf(stderr, "cannot get environment variable!..\n");
        exit(EXIT_FAILURE);
    }
    puts(result);

    return 0;
}

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

    exit(EXIT_FAILURE);
}

/
*-----
-----
    putenv fonksiyonu da setenv fonksiyonuna benzemektedir. Aradaki fark
    putenv fonksiyonun "anahtar=değer" biçiminde tek bir yazı
    almasıdır. Eğer ilgili değişken zaten varsa değeri değiştirilir.
-----
-----*/

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

```

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

```
int main(void)
```

```
{
    char *result;

    if (putenv("city=istanbul") == -1)
        exit_sys("setenv");

    if ((result = getenv("city")) == NULL) {
        fprintf(stderr, "cannot get environment variable!..\n");
        exit(EXIT_FAILURE);
    }
    puts(result);

    return 0;
}
```

```
void exit_sys(const char *msg)
```

```
{
    perror(msg);

    exit(EXIT_FAILURE);
}
```

```
/
```

```
*-----
```

```
Prosesin tüm çevre değişken listesinin yazdırılması. environ global
değişkeniın extern bildirimi hiçbir başlık dosyasında
yapılmamıştır.
```

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

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

```
extern char **environ;
```

```
int main(void)
```

```
{
    int i;

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

    return 0;
}
```

```
/
```

```
*-----
```

```
Çevre değişkenleri birtakım parametrik bilgilerin kolay oluşturulması
için kullanılabilmektedir. Örneğin bir program database
```

dosyasını DATALOC çevre değişkeni ile belirtilen bir dizinde arayacak biçimde yazılmış olabilir.

```
-----*/

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

int main(void)
{
    FILE *f;
    char *val;
    char path[1024] = "datafile";

    if ((val = getenv("DATALOC")) != NULL)
        sprintf(path, "%s/datafile", val);

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

    fclose(f);

    return 0;
}

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

exec fonksiyonun p'li versiyonlarındaki yol ifadelerinde hiç '/'
karakteri yoksa bu durumda bu fonksiyonlar prosesin
PATH isimli çevre değişkenine başvururlar. Bu PATH çevre değişkenin
değeri olan yazıyı ':' karakterlerleirnden parse ederler
sonra sırasıyla ilgili dosyayı o dizinlerde ararlar. İlk bulunan
dizindeki programı exec ederler. Eğer yol ifadesinde en az bir
'/' karakteri varsa bu durumda fonksiyonun davranışı p'siz versiyonlarla
tamamen aynıdır. Aşağıdaki programda "ls" programı PATH
çevre değişkeninde belirtilen dizinlerden birinde bulunacaktır. exec
fonksiyonlarının p'li biçimleri eğer dyol ifadesinde hiç
'/' karakteri yoksa prosesin çalışma dizinine hiç bakmazlar.
-----*/
```

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

void exit_sys(const char *msg);

int main(void)
{
    pid_t pid;
```

```

    if ((pid = fork()) == -1)
        exit_sys("fork");

    if (pid == 0 && execlp("ls", "ls", "-l", (char *)NULL) == -1)
        exit_sys("execl");

    if (waitpid(pid, NULL, 0) == -1)
        exit_sys("waitpid");

    return 0;
}

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

    exit(EXIT_FAILURE);
}

/
*-----
-----
    exec fonksiyonlarının p'li versiyonlarında "./prog" biiminde yol ifadesi
    prosesin çalışma dizinindeki programı
    çalıştırma anlamına gelmektedir.
-----
-----*/

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

void exit_sys(const char *msg);

int main(void)
{
    pid_t pid;

    if ((pid = fork()) == -1)
        exit_sys("fork");

    if (pid == 0 && execlp("./mample", "mample", (char *)NULL) == -1)
        exit_sys("execl");

    if (waitpid(pid, NULL, 0) == -1)
        exit_sys("waitpid");

    return 0;
}

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

```

```

    exit(EXIT_FAILURE);
}

/
*-----
-----
    execvp fonksiyonu execv fonksiyonun p'li biçimidir. Shell programları
    genel olarak komut satırından girilen yol ifadelerini
    exec fonksiyonlarının p'li versiyonlarıyla çalıştırmaktadır. Bu nedenle
    biz prosesin çalışma dizini içerisindeki bir
    programı çalıştırabilmek için "./isim" biçiminde bir yol ifadesi
    kullanmak zorunda kalmaktayız. Shell programları güvenlik
    amacıyla exec fonksiyonlarının p'li versiyonlarını kullanmaktadır.
-----
-----*/

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

void exit_sys(const char *msg);

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

    if (argc == 1) {
        fprintf(stderr, "wrong number of arguments!..\n");
        exit(EXIT_FAILURE);
    }

    if ((pid = fork()) == -1)
        exit_sys("fork");

    if (pid == 0 && execvp(argv[1], &argv[1]) == -1)
        exit_sys("execl");

    if (waitpid(pid, NULL, 0) == -1)
        exit_sys("waitpid");

    return 0;
}

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

    exit(EXIT_FAILURE);
}

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

```

execle fonksiyonu exec fonksiyonlarının 'e' versiyonlarından biridir.
exec fonksiyonlarının e'li versyonları ilgili programı,
çalıştırırken çevre değişken listesini de değiştirmektedir.

```
-----*/

/* sample.c */

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

void exit_sys(const char *msg);

int main(void)
{
    pid_t pid;
    char *envs[] = {"city=istanbul", "name=ali", NULL};

    if ((pid = fork()) == -1)
        exit_sys("fork");

    if (pid == 0 && execle("./mample", "./mample", "ali", "veli", "selami",
        (char *)NULL, envs) == -1)
        exit_sys("execl");

    if (waitpid(pid, NULL, 0) == -1)
        exit_sys("waitpid");

    return 0;
}

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

    exit(EXIT_FAILURE);
}

/* mample.c */

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

extern char **environ;

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

    printf("Command line arguments:");

    for (i = 0; i < argc; ++i)
        puts(argv[i]);
}
```



```

    printf("Environment variables:\n");

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

    return 0;
}

/
*-----
-----
    execve fonksiyonu da execl ile benzerdir. Ancak bu fonksiyon komut
    satırı argümanlarını tek tek değil dizi olarak alır.
-----
-----*/

/* sample.c */

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

void exit_sys(const char *msg);

int main(void)
{
    pid_t pid;
    char *args[] = { "./mample", "ali", "veli", "selami", NULL };
    char *envs[] = { "city=istanbul", "name=ali", NULL };

    if ((pid = fork()) == -1)
        exit_sys("fork");

    if (pid == 0 && execve("./mample", args, envs) == -1)
        exit_sys("execl");

    if (waitpid(pid, NULL, 0) == -1)
        exit_sys("waitpid");

    return 0;
}

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

    exit(EXIT_FAILURE);
}

/* mample.c */

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

```

```
extern char **environ;
```

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

    printf("Command line arguments:");

    for (i = 0; i < argc; ++i)
        puts(argv[i]);

    printf("Environment variables:\n");

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

    return 0;
}
```

```
/
```

```
*-----*
```

```
    fexecve fonksiyonu tamamen execve fonksiyonu gibidir. Tek farkı
    çalıştırılacak dosyayı yol ifadesi ile almak yerine dosya
    betimleyicisi yoluyla almasıdır. Yani çalıştırılacak dosya open
    fonksiyonuyla O_RDONLY modda açılmışsa biz doğrudan fexecve
    fonksiyonunu da kullanabiliriz.
```

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

```
/* sample.c */
```

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

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

```
int main(void)
{
    pid_t pid;
    char *args[] = {"/mample", "ali", "veli", "selami", NULL};
    char *envs[] = {"city=istanbul", "name=ali", NULL};
    int fd;

    if ((pid = fork()) == -1)
        exit_sys("fork");

    if (pid == 0) {
        if ((fd = open("mample", O_RDONLY)) == -1)
            exit_sys("open");
        if (fexecve(fd, args, envs) == -1)
```

```

        exit_sys("execl");
    /* unreachable code */
}

    if (waitpid(pid, NULL, 0) == -1)
        exit_sys("waitpid");

    return 0;
}

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

    exit(EXIT_FAILURE);
}

/* mample.c */

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

extern char **environ;

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

    printf("Command line arguments:");

    for (i = 0; i < argc; ++i)
        puts(argv[i]);

    printf("Environment variables:\n");

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

    return 0;
}

/
*-----
-----
    Sistem fonksiyonlarının numara belirtilerek syscall isimli Linux
    fonksiyonuyla çağrılmasına örnek
-----
-----*/

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

```

```

void exit_sys(const char *msg);

extern char **environ;

int main(void)
{
    char *args[] = {"/bin/ls", "-l", NULL};

    if (syscall(SYS_execve, "/bin/ls", args, environ) == -1)
        exit_sys("execve");

    printf("Unreachable code!");

    return 0;
}

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

    exit(EXIT_FAILURE);
}

/
*-----
-----
    chmod fonksiyonuyla dosyanın set user id, set group id ve sticky
    özelliklerinin set edilmesi (S_IFMT sembolik sabiti dosya türünü
    maskelemek için
    kullanılmaktadır. Bu durumda dosya erişim hakları için bu sembolik
    sabitin tersi ile & işlemi yapmak gerekir.)
-----
-----*/

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

void exit_sys(const char *msg);

int main(void)
{
    struct stat finfo;

    if (stat("mamle", &finfo) == -1)
        exit_sys("stat");

    if (chmod("mamle", finfo.st_mode & ~S_IFMT|S_ISUID|S_ISGID|S_ISVTX) ==
        -1)
        exit_sys("chmod");

    return 0;
}

```

```

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

    exit(EXIT_FAILURE);
}

/
*-----
-----
Etkin kullanıcı id'si csd olan bir kabukta mample isimli kullanıcı
id'si student olan bir programı çalıştırırsak prosesin
etkin kullanıcı id'si yine csd olarak kalır. Fakat mample program
dosyasının "set user id" özelliği set edilmişse bu durumda
etkin kullanıcı id'si csd olan proses mample dosyasını exec yaparsa
artık prosesin etkin kullanıcı id'si "student" olacaktır.
Aşağıdaki mample.c programının program dosyasının set user id
özellığının set edilmiş olduğunu düşünelim. student.txt dosyasının da
erişim hakları rw-r--r-- biçiminde olsun. student.txt dosyasının
kullanıcı id'si de student'tır. İşte bu durumda biz kim olursak olalım
eğer mample dosyasını çalıştırma hakkına sahipsek bu mample student.txt
dosyasını write modda açanilecektir.
-----
-----*/

/* mample.c */

#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("student.txt", O_WRONLY)) == -1)
        exit_sys("open");

    printf("success..\n");

    return 0;
}

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

    exit(EXIT_FAILURE);
}

```

```

/
*-----
-----
Prosesin gerçek kullanıcı id'si getuid fonksiyonuyla, etkin kullanıcı
id'si geteuid fonksiyonuyla, gerçek grup id'si
getgid fonksiyonuyla etkin grup id'si de getegid fonksiyonuyla elde
edilebilir. Saklı kullanıcı ve grup id'lerini elde eden
bir POSIX fonksiyonu yoktur ancak Linux sistemlerinde bir fonksiyo
vardır.
-----
-----*/

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

int main(void)
{
    uid_t ruid, euid;
    gid_t rgid, egid;

    ruid = getuid();
    euid = geteuid();

    rgid = getgid();
    egid = getegid();

    printf("Real User Id: %lu\n", (unsigned long)ruid);
    printf("Effective User Id: %lu\n", (unsigned long)euid);
    printf("Real Group Id: %lu\n", (unsigned long)rgid);
    printf("Effective Group Id: %lu\n", (unsigned long)egid);

    return 0;
}

/
*-----
-----
Aşağıdaki örnekte sample programı başka bir kullanıcıya ait olan ve set
user id özelliği set edilmiş olan mample programını
çalıştırmaktadır. Etkin kullanıcı id'sinin değiştiğine dikkat ediniz.
(Denemeyi yapmadan önce mample programının sahipliğini
değiştirip set user id özelliğini de set ediniz.)
-----
-----*/

/* sample.c */

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

void exit_sys(const char *msg);

int main(void)
{

```

```

    uid_t ruid, euid;
    gid_t rgid, egid;

    ruid = getuid();
    euid = geteuid();

    rgid = getgid();
    egid = getegid();

    printf("Before exec:\n");

    printf("Real User Id: %lu\n", (unsigned long)ruid);
    printf("Effective User Id: %lu\n", (unsigned long)euid);
    printf("Real Group Id: %lu\n", (unsigned long)rgid);
    printf("Effective Group Id: %lu\n\n", (unsigned long)egid);

    if (execl("mample", "mample", (char *)NULL) == -1)
        exit_sys("execl");

    /* unreachable code */

    return 0;
}

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

    exit(EXIT_FAILURE);
}

/* mample.c */

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

int main(void)
{
    uid_t ruid, euid;
    gid_t rgid, egid;

    ruid = getuid();
    euid = geteuid();

    rgid = getgid();
    egid = getegid();

    printf("Real User Id: %lu\n", (unsigned long)ruid);
    printf("Effective User Id: %lu\n", (unsigned long)euid);
    printf("Real Group Id: %lu\n", (unsigned long)rgid);
    printf("Effective Group Id: %lu\n", (unsigned long)egid);

    return 0;
}

```

```

/
*-----
-----
    getresuid ve getresgid isimli fonksiyonlarla biz gerçek, etkin ve saklı
    id'leri alabiliriz. Ancak bu fonksiyon POSIX
    standartlarında yoktur. Linux ve BSD sistemlerinde bulunmaktadır. Bu
    fonksiyonları kullanmadan önce _GNU_SOURCE nitelik
    makrosunu dosyanın tepesine include ediniz. Ya da derleme işleminde -
    D_GNU_SOURCE komut satırı argümanını bulundurunuz
-----
-----*/

#define _GNU_SOURCE

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

void exit_sys(const char *msg);

int main(void)
{
    uid_t ruid, euid, ssuid;
    gid_t rgid, egid, ssgid;

    if (getresuid(&ruid, &euid, &ssuid) == -1)
        exit_sys("getresuid");

    if (getresgid(&rgid, &egid, &ssgid) == -1)
        exit_sys("getresgid");

    printf("Real User Id = %lu, Effective User Id = %lu, Saved Set User Id =
    %lu\n",
        (unsigned long)ruid, (unsigned long)euid, (unsigned long)ssuid);

    printf("Real Group Id = %lu, Effective Group Id = %lu, Saved Set Group
    Id = %lu\n",
        (unsigned long)rgid, (unsigned long)egid, (unsigned long)ssgid);

    return 0;
}

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

    exit(EXIT_FAILURE);
}

/
*-----
-----
    Bir prosesin ek grup id'leri (supplementary group ids) getgroups isimli
    POSIX fonksiyonuyla elde edilebilir.

```


-----*/

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

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

```
int main(void)
{
    gid_t groups[NGROUPS_MAX + 1];
    int result, i;
    struct group *gr;

    if ((result = getgroups(NGROUPS_MAX + 1, groups)) == -1)
        exit_sys("getgroups");

    for (i = 0; i < result; ++i) {
        if ((gr = getgrgid(groups[i])) == NULL)
            exit_sys("getgrgid");
        printf("%s (%lu) ", gr->gr_name, (unsigned long)groups[i]);
    }
    printf("\n");

    return 0;
}
```

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

    exit(EXIT_FAILURE);
}
```

/

*-----

setgroups isimli fonksiyonuyla prosesimizin ek group id'lerini set edebiliriz. Ancak setgroups fonksiyonu bir POSIX fonksiyonu değildir. Fakat yaygın Unix türevi sistemlerde (örneğin Linux sistemlerinde) bu fonksiyon bulunmaktadır. Eğer proses root değilse ya da uygun yeterliliklere sahip değilse fonksiyon başarısız olmaktadır. (Yani programı sudo ile çalıştırmalısınız)

-----*/

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

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

```
int main(void)
```

```
{
    gid_t groups[NGROUPS_MAX + 1] = {115, 123, 127};
    int result, i;
    struct group *gr;

    if ((result = setgroups(3, groups)) == -1)
        exit_sys("setgroups");

    if ((result = getgroups(NGROUPS_MAX + 1, groups)) == -1)
        exit_sys("getgroups");

    for (i = 0; i < result; ++i) {
        if ((gr = getgrgid(groups[i])) == NULL)
            exit_sys("getgrgid");
        printf("%s (%lu) ", gr->gr_name, (unsigned long)groups[i]);
    }
    printf("\n");

    return 0;
}
```

```
void exit_sys(const char *msg)
```

```
{
    perror(msg);

    exit(EXIT_FAILURE);
}
```

```
/
```

```
*-----
-----
```

Bu fonksiyon tipik olarak login programı tarafından kullanılır.
Fonksiyon parametresiyle aldığı kullanıcı ismine ilişkin ek grupları /etc/passwd ve /etc/group dosyalarından elde eder ve setgroups fonksiyonunu çağırarak proses için set eder.
initgroups bir POSIX fonksiyonu değildir. Linux ve BSD gibi sistemlerde bulunmaktadır. Fonksiyon etkin kullanıcı id'si root olmayan ve gerekli yeteneğe sahip olmayan prosesler tarafından çağrılırsa başarısız olmaktadır. (Yani aşağıdaki programı sudo ile çalıştırmalısınız.)

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

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

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

```

int main(void)
{
    gid_t groups[NGROUPS_MAX + 1];
    int result, i;
    struct group *gr;

    if (initgroups("csd", getegid()) == -1)
        exit_sys("initgroups");

    if ((result = getgroups(NGROUPS_MAX + 1, groups)) == -1)
        exit_sys("getgroups");

    for (i = 0; i < result; ++i) {
        if ((gr = getgrgid(groups[i])) == NULL)
            exit_sys("getgrgid");
        printf("%s (%lu) ", gr->gr_name, (unsigned long)groups[i]);
    }
    printf("\n");

    return 0;
}

```

```

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

    exit(EXIT_FAILURE);
}

```

```

/
*-----
-----
İsimsiz boru (unnamed pipe) örneği. Bu örnekte proses önce boruyu sonra
da alt prosesi yaratır. Üst proses boruya yazma
yapar alt proses de borudan okuma yapmaktadır.
-----
-----*/

```

```

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

```

```

void exit_sys(const char *msg);

```

```

int main(void)
{
    int pfd[2];
    pid_t pid;
    ssize_t result;
    int i, val;

    if (pipe(pfd) == -1)
        exit_sys("pipe");
}

```

```

if ((pid = fork()) == -1)
    exit_sys("fork");

if (pid != 0) {          /* parent writes */
    close(pfds[0]);
    for (i = 0; i < 1000000; ++i)
        if (write(pfds[1], &i, sizeof(int)) == -1)
            exit_sys("write");
    close(pfds[1]);

    if (waitpid(pid, NULL, 0) == -1)
        exit_sys("waitpid");
}
else {                  /* child reads */
    close(pfds[1]);
    while ((result = read(pfds[0], &val, sizeof(int))) > 0) {
        printf("%d ", val);
        fflush(stdout);
    }
    printf("\n");
    if (result == -1)
        exit_sys("read");
    close(pfds[0]);
}

return 0;
}

```

```

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

    exit(EXIT_FAILURE);
}

```

```

/
*-----
-----
Kabuk üzerinde pipe işlemi nasıl yapılmaktadır? a | b biçimindeki bir
komutta kabuk önce isimsiz boruyu yaratır. Sonra a
programı için fork yapar. Henüz exec yapmadan alt prosesin 1 numaralı
betimleyicisini yazma amaçlı boruya yönlendirir.
Sonra exec yapar. Daha sonra yine b için fork yapar. henüz exec yapmadan
alt proste 0 numaralı betimleyiciyi okuma
amaçlı boruya yönlendirir. Sonra exec yapar. Bu işlemler sırasında
gereksiz betimleyicilerin hepsi kapatılmaktadır. Aşağıdaki
örnekte kabuğun yaptığı gibi bir boru işlemi gerçekleştirilmektedir.
-----
-----*/

```

```

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

```

```

void exit_sys(const char *msg);

int main(int argc, char *argv[])    /* ./sample prog args ... "|" prog args
...*/
{
    int pfd[2];
    pid_t pid1, pid2;
    int i, pindex;

    for (pindex = 0; pindex < argc; ++pindex)
        if (!strcmp(argv[pindex], "|"))
            break;
    if (pindex == 0 || pindex == argc || pindex == argc - 1) {
        fprintf(stderr, "invalid argument!\n");
        exit(EXIT_FAILURE);
    }
    argv[pindex] = NULL;

    if (pipe(pfd) == -1)
        exit_sys("pipe");

    if ((pid1 = fork()) == -1)
        exit_sys("fork");
    if (pid1 == 0) {
        if (dup2(pfd[1], 1) == -1)
            exit_sys("dup2");
        close(pfd[0]);
        close(pfd[1]);
        if (execvp(argv[1], &argv[1]) == -1)
            exit_sys("execv");

        /* unreachable code */
    }

    if ((pid2 = fork()) == -1)
        exit_sys("fork");
    if (pid2 == 0) {
        if (dup2(pfd[0], 0) == -1)
            exit_sys("dup2");
        close(pfd[0]);
        close(pfd[1]);
        if (execvp(argv[pindex + 1], &argv[pindex + 1]) == -1)
            exit_sys("execv");

        /* unreachable code */
    }
    close(pfd[0]);
    close(pfd[1]);

    if (waitpid(pid1, NULL, 0) == -1)
        exit_sys("waitpid");
    if (waitpid(pid2, NULL, 0) == -1)
        exit_sys("waitpid");
}

```

```

    return 0;
}

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

    exit(EXIT_FAILURE);
}

/
*-----
-----
mkfifo komutuna benzer bir program. Programın -m, --mode ve -h, --help
komut satırı argümanları vardır. Program belirtilen
isimdeki isimli boruyu yaratır.
-----
-----*/

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

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

int main(int argc, char *argv[])
{
    int result, help_flag = 0, err_flag = 0;
    int mode_flag = 0;
    int i;
    mode_t mode, result_mode;
    mode_t modes[] = {S_IRUSR, S_IWUSR, S_IXUSR, S_IRGRP, S_IWGRP, S_IXGRP,
        S_IROTH, S_IWOTH, S_IXOTH};
    char *mode_arg;
    struct option options[] = {
        {"mode", required_argument, NULL, 'm'},
        {"help", no_argument, NULL, 'h'},
        {0, 0, 0, 0}
    };

    opterr = 0;
    while ((result = getopt_long(argc, argv, "m:h", options, NULL)) != -1) {
        switch (result) {
            case 'm':
                mode_flag = 1;
                mode_arg = optarg;
                break;
            case 'h':
                help_flag = 1;
                break;
            case '?':
                if (optopt != 0)

```

```

        fprintf(stderr, "invalid switch: -%c\n", optopt);
    else
        fprintf(stderr, "invalid switch: %s\n", argv[optind -
            1]); /* argv[optind - 1] dokümente edilmemiş */
    err_flag = 1;
}
}
if (err_flag)
    exit(EXIT_FAILURE);

if (help_flag) {
    fprintf(stdout, "mymkfifo [-m |--mode <mode>] <file list>\n");
    exit(EXIT_SUCCESS);
}

if (mode_flag) {
    if (!is_octal(mode_arg) || (mode = (mode_t)strtoul(mode_arg, NULL,
        8)) > 0x777) {
        fprintf(stderr, "invalid octal digits!..\n");
        exit(EXIT_FAILURE);
    }

    result_mode = 0;
    for (i = 8; i >= 0; --i) {
        if (mode >> i & 1)
            result_mode |= modes[8 - i];
    }
}
else
    result_mode = S_IRUSR|S_IWUSR|S_IRGRP|S_IROTH;

umask(0);
for (i = optind; i < argc; ++i)
    if (mkfifo(argv[i], result_mode) == -1) {
        perror("mkfifo");
        continue;
    }

return 0;
}

int is_octal(const char *str)
{
    int i;

    for (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);
}

/
*-----
-----
    Blokeli modda isimli borular open fonksiyonuyla O_RDONLY modunda
    açıldığında başka bir proses boruyu O_WRONLY ya da O_RDWR
    modunda açana kadar open blokede kalır. Benzer biçimde isimli borular
    open fonksiyonuyla O_WRONLY modunda açıldığında başka
    bir proses boruyu O_RDONLY ya da O_RDWR modunda açana kadar open blokede
    kalır. Ancak isimli boru O_RDWR modunda açılmaya
    çalışırsa bloke oluşmaz. Aşağıda isimli boru örneği verilmiştir.
    -----
    -----*/

/* named-pipe-proc1.c */

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

void exit_sys(const char *msg);

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

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

    if ((pipefd = open(argv[1], O_WRONLY)) == -1)
        exit_sys("open");

    for (i = 0; i < 1000000; ++i)
        if (write(pipefd, &i, sizeof(int)) == -1)
            exit_sys("write");

    close(pipefd);

    return 0;
}

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

    exit(EXIT_FAILURE);
}

```



```
/* named-pipe-proc2.c */
```

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

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

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

```
    int pipefd;
    int val;
    int result;
```

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

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

```
    while ((result = read(pipefd, &val, sizeof(int))) > 0)
        printf("%d ", val), fflush(stdout);
    printf("\n");
```

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

```
    close(pipefd);
```

```
    return 0;
```

```
}
```

```
void exit_sys(const char *msg)
```

```
{
    perror(msg);

    exit(EXIT_FAILURE);
}
```

```
/
```

```
*-----
```

```
-----
Boru dosyasını O_NONBLOCK bayrağı ile blokesiz modda açtığımızda read
fonksiyonu hiç bloke olmaz. read ile n byte okunmak
istendiğinde eğer boruda az sayıda bilgi varsa read n byte okunana kadar
beklemez. Boruda olanı okur ve okuduğu byte sayısı
ile geri döner. Eğer boruda 0 byte varsa read 0 ile geri dönmez -1 ile
geri döner yani başarısız olur. Ancak errno değeri
EAGAIN biçiminde özel bir değere set edilir. Programcı da boruda hiçbir
şey yoksa arka plan işlemleri yapabilir. Benzer
biçimde blokesiz modda write fonksiyonu da n byte'ın hepsi yazılana
kadar blokede beklemes. Yazabildiği byte sayısını yazar
```

yazabildiği byte sayısına geri döner. Boru tamamen doluysa write başarısız olur -1 değerine geri döner ve errno EAGAIN değeri ile set edilir. Blokesiz modda open fonksiyonu da asla bloke olmaz. O_RDONLY ya da O_RDWR modunda boru açılmaya çalışılırsa open başarılı olur. open O_WRONLY modunda boru açılmak istenirse başka bir proses "read" modda boruyu açmamışsa başarılı olur.

```
-----*/

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

void exit_sys(const char *msg);

int main(int argc, char *argv[])
{
    int pipefd;
    int result;
    int i;
    char buf[11];

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

    if ((pipefd = open(argv[1], O_RDONLY|O_NONBLOCK)) == -1)
        exit_sys("open");

    sleep(1);

    for (i = 0; i < 100; ++i) {
        if ((result = read(pipefd, buf, 10)) == -1) {
            if (errno == EAGAIN) {
                printf("%d ", i), fflush(stdout);
                sleep(1);
                continue;
            }
        }
        buf[result] = '\0';
        puts(buf);
        sleep(1);
    }

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

    close(pipefd);

    return 0;
}
```

```

}

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

    exit(EXIT_FAILURE);
}

/
*-----
-----
Nonblocking pipe örneği (önce nonblocking-pipe1.c programını
çalıştırınız). Nonblocking işlemlerde meşgul döngü (busy loop)
önemli bir problemdir. Bu problemi ortadan kaldırmak için select, poll
gibi fonksiyonlar ve asenkron io yöntemleri kullanılır.
-----
-----*/

/* nonblocking-pipe1.c */

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

void exit_sys(const char *msg);

int main(int argc, char *argv[])
{
    int pipefd;
    int result;
    int val;

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

    if ((pipefd = open(argv[1], O_RDONLY|O_NONBLOCK)) == -1)
        exit_sys("open");
    sleep(10);
    for (;;) {
        if ((result = read(pipefd, &val, sizeof(int))) == -1)
            if (errno == EAGAIN)
                continue;
            else
                exit_sys("read");
        if (result == 0)
            break;
        printf("%d ", val), fflush(stdout);
    }
    printf("\n");
}

```

```

    close(pipefd);

    return 0;
}

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

    exit(EXIT_FAILURE);
}

/* nonblocking-pipe2.c */

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

void exit_sys(const char *msg);

int main(int argc, char *argv[])
{
    int pipefd;
    int result;
    int val;

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

    if ((pipefd = open(argv[1], O_RDONLY|O_NONBLOCK)) == -1)
        exit_sys("open");
    sleep(10);
    for (;;) {
        if ((result = read(pipefd, &val, sizeof(int))) == -1)
            if (errno == EAGAIN)
                continue;
            else
                exit_sys("read");
        if (result == 0)
            break;
        printf("%d ", val), fflush(stdout);
    }

    close(pipefd);

    return 0;
}

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

```

```

    exit(EXIT_FAILURE);
}

/
*-----
-----
    Bir kabuk komutunu (dolayısıyla bir programı) çalıştırıp onun stdout
    dosyasına yazdıklarını ya da stdin dosyasından okuduklarını
    boruya yönlendirebiliriz. Bunun için popen ve pclose POSIX fonksiyonları
    kullanılmaktadır.
-----
-----*/

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

void exit_sys(const char *msg);

int main(void)
{
    FILE *f;
    int ch;

    if ((f = popen("gcc -o mample mample.c", "r")) == NULL)
        exit_sys("popen");

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

    pclose(f);

    return 0;
}

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

    exit(EXIT_FAILURE);
}

/
*-----
-----
    popen ve pclose fonksiyonlarının örnek bir gerçekleştirimi
-----
-----*/

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

```

```

void exit_sys(const char *msg);

static pid_t g_pid;

FILE *csd_popen(const char *command, const char *mode)
{
    pid_t pid;
    int pfd[2];
    FILE *f;

    if (mode[1] != '\0' || (mode[0] != 'r' && mode[0] != 'w'))
        return NULL;

    if (pipe(pfd) == -1)
        return NULL;

    if ((pid = fork()) == -1)
        return NULL;

    if (pid == 0) {
        if (mode[0] == 'r') {
            if (dup2(pfd[1], 1) == -1)
                _exit(EXIT_FAILURE);
            close(pfd[0]);
            close(pfd[1]);
        }
        else {
            if (dup2(pfd[0], 0) == -1)
                _exit(EXIT_FAILURE);
            close(pfd[0]);
            close(pfd[1]);
        }
        if (execl("/bin/bash", "/bin/bash", "-c", command, (char *) NULL) ==
            -1)
            _exit(EXIT_FAILURE);
    }
    g_pid = pid;
    if (mode[0] == 'r') {
        close(pfd[1]);
        f = fdopen(pfd[0], "r");
    }
    else {
        close(pfd[0]);
        f = fdopen(pfd[1], "w");
    }

    return f;
}

int csd_pclose(FILE *f)
{
    int status;

    if (waitpid(g_pid, &status, 0) == -1)
        return -1;

```

```

        fclose(f);

        return status;
    }

int main(void)
{
    FILE *f;
    int ch;

    if ((f = csd_popen("ls -l", "r")) == NULL) {
        fprintf(stderr, "csd_open failed\n");
        exit(EXIT_FAILURE);
    }
    while ((ch = fgetc(f)) != EOF)
        putchar(ch);

    csd_pclose(f);

    return 0;
}

```

```

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

    exit(EXIT_FAILURE);
}

```

```

/
*-----
-----
    Boru yoluyla client-server haberleşme örneği
-----
-----*/

```

```

/* client.c */

```

```

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>
#include <fcntl.h>
#include <unistd.h>
#include <signal.h>

```

```

/* Symbolic Constants */

```

```

#define SERVER_PIPE        "serverpipe"
#define MAX_CMD_LEN        1024
#define MAX_MSG_LEN        32768
#define MAX_PIPE_PATH      1024

```

```

/* Type Declaration */

```

```

typedef struct tagCLIENT_MSG {
    int msglen;
    int client_id;
    char msg[MAX_MSG_LEN];
} CLIENT_MSG;

typedef struct tagSERVER_MSG {
    int msglen;
    char msg[MAX_MSG_LEN];
} SERVER_MSG;

typedef struct tagMSG_CONTENTS {
    char *msg_cmd;
    char *msg_param;
} MSG_CONTENTS;

typedef struct tagMSG_PROC {
    const char *msg_cmd;
    int (*proc)(const char *msg_param);
} MSG_PROC;

/* Function Prototypes */

void sigpipe_handler(int sno);
int putmsg(const char *cmd);
int get_server_msg(int fdp, SERVER_MSG *smsg);
void parse_msg(char *msg, MSG_CONTENTS *msgc);
void check_quit(char *cmd);
int connect_to_server(void);
int cmd_response_proc(const char *msg_param);
int disconnect_accepted_proc(const char *msg_param);
int invalid_command_proc(const char *msg_param);
void clear_stdin(void);
void exit_sys(const char *msg);

/* Global Data Definitions */

MSG_PROC g_msg_proc[] = {
    {"CMD_RESPONSE", cmd_response_proc},
    {"DISCONNECT_ACCEPTED", disconnect_accepted_proc},
    {"INVALID_COMMAND", invalid_command_proc},
    {NULL, NULL}
};

int g_client_id;
int g_fdps, g_fdpc;

/* Function Definitions */

int main(void)
{
    char cmd[MAX_CMD_LEN];
    char *str;
    SERVER_MSG smsg;

```



```

MSG_CONTENTS msgc;
int i;

if (signal(SIGPIPE, sigpipe_handler) == SIG_ERR)
    exit_sys("signal");

if ((g_fdps = open(SERVER_PIPE, O_WRONLY)) == -1)
    exit_sys("open");

if (connect_to_server() == -1) {
    fprintf(stderr, "cannot connect to server! Try again...\n");
    exit(EXIT_FAILURE);
}

for (;;) {
    printf("Client>");
    fflush(stdout);
    fgets(cmd, MAX_CMD_LEN, stdin);
    if ((str = strchr(cmd, '\n')) != NULL)
        *str = '\0';

    check_quit(cmd);

    if (putmsg(cmd) == -1)
        exit_sys("putmsg");

    if (get_server_msg(g_fdpc, &smsg) == -1)
        exit_sys("get_client_msg");

    parse_msg(smsg.msg, &msgc);

    for (i = 0; g_msg_proc[i].msg_cmd != NULL; ++i)
        if (!strcmp(msgc.msg_cmd, g_msg_proc[i].msg_cmd)) {
            if (g_msg_proc[i].proc(msgc.msg_param) == -1) {
                fprintf(stderr, "command failed!\n");
                exit(EXIT_FAILURE);
            }
            break;
        }
    if (g_msg_proc[i].msg_cmd == NULL) { /* command not found */
        fprintf(stderr, "Fatal Error: Unknown server message!\n");
        exit(EXIT_FAILURE);
    }
}

return 0;
}

void sigpipe_handler(int sno)
{
    printf("server down, exiting...\n");

    exit(EXIT_FAILURE);
}

```

```

int putmsg(const char *cmd)
{
    CLIENT_MSG cmsg;
    int i, k;

    for (i = 0; isspace(cmd[i]); ++i)
        ;
    for (k = 0; !isspace(cmd[i]); ++i)
        cmsg.msg[k++] = cmd[i];
    cmsg.msg[k++] = ' ';
    for (; isspace(cmd[i]); ++i)
        ;
    for (; (cmsg.msg[k++] = cmd[i]) != '\0'; ++i)
        ;
    cmsg.msglen = (int)strlen(cmsg.msg);
    cmsg.client_id = g_client_id;

    if (write(g_fdps, &cmsg, 2 * sizeof(int) + cmsg.msglen) == -1)
        return -1;

    return 0;
}

int get_server_msg(int fdp, SERVER_MSG *smsg)
{
    if (read(fdp, &smsg->msglen, sizeof(int)) == -1)
        return -1;

    if (read(fdp, smsg->msg, smsg->msglen) == -1)
        return -1;

    smsg->msg[smsg->msglen] = '\0';

    return 0;
}

void parse_msg(char *msg, MSG_CONTENTS *msgc)
{
    int i;

    msgc->msg_cmd = msg;

    for (i = 0; msg[i] != ' ' && msg[i] != '\0'; ++i)
        ;
    msg[i++] = '\0';
    msgc->msg_param = &msg[i];
}

void check_quit(char *cmd)
{
    int i, pos;

    for (i = 0; isspace(cmd[i]); ++i)
        ;
    pos = i;

```

```

    for (; !isspace(cmd[i]) && cmd[i] != '\\0'; ++i)
        ;
    if (!strncmp(&cmd[pos], "quit", pos - i))
        strcpy(cmd, "DISCONNECT_REQUEST");
}

int connect_to_server(void)
{
    char name[MAX_PIPE_PATH];
    char cmd[MAX_CMD_LEN];
    char *str;
    SERVER_MSG smsg;
    MSG_CONTENTS msgc;
    int response;

    printf("Pipe name:");
    fgets(name, MAX_PIPE_PATH, stdin);
    if ((str = strchr(name, '\\n')) != NULL)
        *str = '\\0';

    if (access(name, F_OK) == 0) {
        do {
            printf("Pipe already exists! Overwrite? (Y/N)");
            fflush(stdout);
            response = tolower(getchar());
            clear_stdin();
            if (response == 'y' && remove(name) == -1)
                return -1;
        } while (response != 'y' && response != 'n');
        if (response == 'n')
            return -1;
    }

    sprintf(cmd, "CONNECT %s", name);

    if (putmsg(cmd) == -1)
        return -1;

    while (access(name, F_OK) != 0)
        usleep(300);

    if ((g_fdpc = open(name, O_RDONLY)) == -1)
        return -1;

    if (get_server_msg(g_fdpc, &smsg) == -1)
        exit_sys("get_client_msg");

    parse_msg(smsg.msg, &msgc);

    if (strcmp(msgc.msg_cmd, "CONNECTED"))
        return -1;

    g_client_id = (int)strtol(msgc.msg_param, NULL, 10);

    printf("Connected server with '%d' id...\n", g_client_id);

```

```

    return 0;
}

int cmd_response_proc(const char *msg_param)
{
    printf("%s\n", msg_param);

    return 0;
}

int disconnect_accepted_proc(const char *msg_param)
{
    if (putmsg("DISCONNECT") == -1)
        exit_sys("putmsg");

    exit(EXIT_SUCCESS);

    return 0;
}

int invalid_command_proc(const char *msg_param)
{
    printf("invalid command: %s\n", msg_param);

    return 0;
}

void clear_stdin(void)
{
    int ch;

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

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

    exit(EXIT_FAILURE);
}

/* server.c */

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

/* Symbolic Constants */

```

```

#define SERVER_PIPE            "serverpipe"
#define MAX_MSG_LEN            32768
#define MAX_PIPE_PATH          1024
#define MAX_CLIENT              1024

/* Type Declaration */

typedef struct tagCLIENT_MSG {
    int msglen;
    int client_id;
    char msg[MAX_MSG_LEN];
} CLIENT_MSG;

typedef struct tagSERVER_MSG {
    int msglen;
    char msg[MAX_MSG_LEN];
} SERVER_MSG;

typedef struct tagMSG_CONTENTS {
    char *msg_cmd;
    char *msg_param;
} MSG_CONTENTS;

typedef struct tagMSG_PROC {
    const char *msg_cmd;
    int (*proc)(int, const char *msg_param);
} MSG_PROC;

typedef struct tagCLIENT_INFO {
    int fdp;
    char path[MAX_PIPE_PATH];
} CLIENT_INFO;

/* Function Prototypes */

int get_client_msg(int fdp, CLIENT_MSG *cmsg);
int putmsg(int client_id, const char *cmd);
void parse_msg(char *msg, MSG_CONTENTS *msgc);
void print_msg(const CLIENT_MSG *cmsg);
int invalid_command(int client_id, const char *cmd);
int connect_proc(int client_id, const char *msg_param);
int disconnect_request_proc(int client_id, const char *msg_param);
int disconnect_proc(int client_id, const char *msg_param);
int cmd_proc(int client_id, const char *msg_param);
void exit_sys(const char *msg);

/* Global Data Definitions */

MSG_PROC g_msg_proc[] = {
    {"CONNECT", connect_proc},
    {"DISCONNECT_REQUEST", disconnect_request_proc},
    {"DISCONNECT", disconnect_proc},
    {"CMD", cmd_proc},
    {NULL, NULL}
};

```

```
CLIENT_INFO g_clients[MAX_CLIENT];
```

```
/* Function Definitions */
```

```
int main(void)
```

```
{
```

```
    int fdp;
```

```
    CLIENT_MSG cmsg;
```

```
    MSG_CONTENTS msgc;
```

```
    int i;
```

```
    if ((fdp = open(SERVER_PIPE, O_RDWR)) == -1)
```

```
        exit_sys("open");
```

```
    for (;;) {
```

```
        if (get_client_msg(fdp, &cmsg) == -1)
```

```
            exit_sys("get_client_msg");
```

```
        print_msg(&cmsg);
```

```
        parse_msg(cmsg.msg, &msgc);
```

```
        for (i = 0; g_msg_proc[i].msg_cmd != NULL; ++i)
```

```
            if (!strcmp(msgc.msg_cmd, g_msg_proc[i].msg_cmd)) {
```

```
                if (g_msg_proc[i].proc(cmsg.client_id, msgc.msg_param)) {
```

```
                }
```

```
                break;
```

```
            }
```

```
        if (g_msg_proc[i].msg_cmd == NULL)
```

```
            if (invalid_command(cmsg.client_id, msgc.msg_cmd) == -1)
```

```
                continue;
```

```
    }
```

```
    close(fdp);
```

```
    return 0;
```

```
}
```

```
int get_client_msg(int fdp, CLIENT_MSG *cmsg)
```

```
{
```

```
    if (read(fdp, &cmsg->msglen, sizeof(int)) == -1)
```

```
        return -1;
```

```
    if (read(fdp, &cmsg->client_id, sizeof(int)) == -1)
```

```
        return -1;
```

```
    if (read(fdp, cmsg->msg, cmsg->msglen) == -1)
```

```
        return -1;
```

```
    cmsg->msg[cmsg->msglen] = '\0';
```

```
    return 0;
```

```
}
```

```
int putmsg(int client_id, const char *cmd)
```

```
{
```

```

SERVER_MSG smsg;
int fdp;

strcpy(smsg.msg, cmd);
smsg.msglen = strlen(smsg.msg);

fdp = g_clients[client_id].fdp;

return write(fdp, &smsg, sizeof(int) + smsg.msglen) == -1 ? -1 : 0;
}

void parse_msg(char *msg, MSG_CONTENTS *msgc)
{
    int i;

    msgc->msg_cmd = msg;

    for (i = 0; msg[i] != ' ' && msg[i] != '\0'; ++i)
        ;
    msg[i++] = '\0';
    msgc->msg_param = &msg[i];
}

void print_msg(const CLIENT_MSG *cmsg)
{
    printf("Message from \"%s\": %s\n", cmsg->client_id ? g_clients[cmsg->client_id].path : "", cmsg->msg);
}

int invalid_command(int client_id, const char *cmd)
{
    char buf[MAX_MSG_LEN];

    sprintf(buf, "INVALID_COMMAND %s", cmd);
    if (putmsg(client_id, buf) == -1)
        return -1;

    return 0;
}

int connect_proc(int client_id, const char *msg_param)
{
    int fdp;
    char buf[MAX_MSG_LEN];

    if (mkfifo(msg_param, S_IRUSR|S_IWUSR|S_IRGRP|S_IROTH) == -1) {
        printf("CONNECT message failed! Params = \"%s\"\n", msg_param);
        return -1;
    }

    if ((fdp = open(msg_param, O_WRONLY)) == -1)
        exit_sys("open");

    g_clients[fdp].fdp = fdp;
    strcpy(g_clients[fdp].path, msg_param);
}

```

```

    sprintf(buf, "CONNECTED %d", fdp);
    if (putmsg(fdp, buf) == -1)
        exit_sys("putmsg");

    return 0;
}

int disconnect_request_proc(int client_id, const char *msg_param)
{
    if (putmsg(client_id, "DISCONNECT_ACCEPTED") == -1)
        return -1;

    return 0;
}

int disconnect_proc(int client_id, const char *msg_param)
{
    close(g_clients[client_id].fdp);

    if (remove(g_clients[client_id].path) == -1)
        return -1;

    return 0;
}

int cmd_proc(int client_id, const char *msg_param)
{
    FILE *f;
    char cmd[MAX_MSG_LEN] = "CMD_RESPONSE ";
    int i;
    int ch;

    if ((f = popen(msg_param, "r")) == NULL) {
        printf("cannot execute shell command!..\n");
        return -1;
    }

    for (i = 13; (ch = fgetc(f)) != EOF; ++i)
        cmd[i] = ch;
    cmd[i] = '\0';

    if (putmsg(client_id, cmd) == -1)
        return -1;

    return 0;
}

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

    exit(EXIT_FAILURE);
}

```



```
/
*-----
-----
XSI Shared Memory: shmget, shmat ve shmdt fonksiyonlarının kullanımları
-----
-----*/
```

```
/* proc1.c */
```

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/ipc.h>
#include <sys/shm.h>
```

```
#define SHM_KEY      0x12345678
```

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

```
int main(void)
```

```
{
    int shmid;
    void *addr;
    char *str;

    if ((shmid = shmget(SHM_KEY, 8192, IPC_CREAT|0600)) == -1)
        exit_sys("shmget");

    if ((addr = shmat(shmid, NULL, 0)) == (void *)-1)
        exit_sys("shmat");

    str = (char *)addr;

    strcpy(str, "This is a test....\n");

    printf("press ENTER to continue...\n");
    getchar();

    shmdt(addr);

    return 0;
}
```

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

    exit(EXIT_FAILURE);
}
```

```
/* proc2.c */
```

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

```

#include <sys/ipc.h>
#include <sys/shm.h>

#define SHM_KEY      0x12345678

void exit_sys(const char *msg);

int main(void)
{
    int shmid;
    void *addr;
    char *str;

    if ((shmid = shmget(SHM_KEY, 8192, IPC_CREAT|0600)) == -1)
        exit_sys("shmget");

    if ((addr = shmat(shmid, NULL, 0)) == (void *)-1)
        exit_sys("shmat");

    printf("press ENTER to continue..\n");
    getchar();

    str = (char *)addr;
    puts(str);

    shmdt(addr);

    return 0;
}

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

    exit(EXIT_FAILURE);
}

/
*-----
-----
    Paylaşılan bellek alanı nesnesi (shared memory segment) onu hiçbir
    proses kullanmıyor olsa bile sistem boot edilene kadar
    yaşamaya devam eder. Yani shmat ile yapılan işlem shmdt ile geri
    alınmaktadır. Ancak shmget ile yapılan işlem eğer silinmezse
    sistem boor edilene kadar kalıcıdır. (Buna "kernel persistency"
    denilmektedir.) Bu nesnesyi silmek için shmctl fonksiyonunu
    IPC_RMID parametresiyle çağırmak gerekir. Silme işlemi için ipcrm isimli
    bir kabuk komutu da bulunmaktadır.
    -----
    -----*/

if (shmctl(shmid, IPC_RMID, NULL) == -1)
    exit_sys("shmctl");

```

```

/
*-----
-----
    Paylaşılan bellek alanı için kernel tarafından oluşturulan shmid_ds
    yapısının bazı elemanları shmctl fonksiyonunun
    IPC_STAT ve IPC_SET parametreleriyle değiştirilebilir
-----
-----*/

#include <stdio.h>
#include <stdlib.h>
#include <sys/ipc.h>
#include <sys/shm.h>

#define SHM_KEY      0x12345678

void exit_sys(const char *msg);

int main(void)
{
    int shmid;
    struct shmid_ds ds;

    if ((shmid = shmget(SHM_KEY, 8192, IPC_CREAT|0600)) == -1)
        exit_sys("shmget");

    if (shmctl(shmid, IPC_STAT, &ds) == -1)
        exit_sys("shmctl");

    ds.shm_perm.mode = 0666;

    if (shmctl(shmid, IPC_SET, &ds) == -1)
        exit_sys("shmctl");

    return 0;
}

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

    exit(EXIT_FAILURE);
}

/
*-----
-----
    Paylaşılan bellek alanları yöntemi maalesef kendi içerisinde bir
    senkronizasyona sahip değildir. Bir prosesin bu alana
    sürekli bir şeyler yazıp diğerinin bunları okuması problemine "üretici,-
    tüketici problemi (producer-consumer problem"
    denilmektedir. Üretici tüketici problemi "semaphore" denilen özel
    senkronizasyon nesneleriyle çözülmektedir. Aşağıdaki

```

örnekte bir senkronizasyon olmazsa okuyan prosesin aynı bilgiyi birden fazla kez alabildiği ve bilgi kaçırabildiği gösterilmek istenmiştir.

```
-----*/
/* proc1. c */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include <unistd.h>
#include <sys/ipc.h>
#include <sys/shm.h>

#define SHM_KEY      0x12345678

struct SHARED_MEM_INFO {
    int val;
    /* ... */
};

void exit_sys(const char *msg);

int main(void)
{
    int shmid;
    void *addr;
    struct SHARED_MEM_INFO *smi;

    srand(time(NULL));

    if ((shmid = shmget(SHM_KEY, 8192, IPC_CREAT|0600)) == -1)
        exit_sys("shmget");

    if ((addr = shmat(shmid, NULL, 0)) == (void *)-1)
        exit_sys("shmat");

    smi = (struct SHARED_MEM_INFO *)addr;

    for (int i = 0; i < 100; ++i) {
        smi->val = i;
        usleep(rand() % 300000);
    }

    shmdt(addr);

    if (shmctl(shmid, IPC_RMID, NULL) == -1)
        exit_sys("shmctl");

    return 0;
}

void exit_sys(const char *msg)
```

```

{
    perror(msg);

    exit(EXIT_FAILURE);
}

/* proc2.c */

#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <unistd.h>
#include <sys/ipc.h>
#include <sys/shm.h>

#define SHM_KEY      0x12345678

struct SHARED_MEM_INFO {
    int val;
    /* ... */
};

void exit_sys(const char *msg);

int main(void)
{
    int shmid;
    void *addr;
    struct SHARED_MEM_INFO *smi;

    srand(time(NULL));

    if ((shmid = shmget(SHM_KEY, 8192, IPC_CREAT|0600)) == -1)
        exit_sys("shmget");

    if ((addr = shmat(shmid, NULL, 0)) == (void *)-1)
        exit_sys("shmat");

    smi = (struct SHARED_MEM_INFO *)addr;

    for (;;) {
        printf("%d ", smi->val);
        fflush(stdout);
        usleep(rand() % 300000);
        if (smi->val == 99)
            break;
    }
    printf("\n");

    shmdt(addr);

    return 0;
}

void exit_sys(const char *msg)

```

```

{
    perror(msg);

    exit(EXIT_FAILURE);
}

/
*-----
-----
Linux'a özgü biçimde shmat fonksiyonunun son parametresinde SHM_EXEC
bayrağı kullanılabilir. Bu durumda paylaşılan bellek alanına
yerleştirilen kod çalıştırılabilir durumda olmaktadır.
-----
-----*/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/ipc.h>
#include <sys/shm.h>

#define SHM_KEY      0x12345678

void exit_sys(const char *msg);
void foo(void);

int main(void)
{
    int shmid;
    void *addr;
    void (*pf)(void);

    if ((shmid = shmget(SHM_KEY, 8192, IPC_CREAT|0777)) == -1)
        exit_sys("shmget");

    if ((addr = shmat(shmid, NULL, SHM_EXEC)) == (void *)-1)
        exit_sys("shmat");

    memcpy(addr, foo, 20);
    pf = (void (*)(void))addr;
    pf();

    shmdt(addr);

    if (shmctl(shmid, IPC_RMID, NULL) == -1)
        exit_sys("shmctl");

    return 0;
}

void foo(void)
{
}

```

```

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

    exit(EXIT_FAILURE);
}

/
*-----
-----
System 5 IPC nesneleri anahtar hareketle id oluşturmaktadır. Anahtarlar
key_t türündendir. Ancak anahtar numaraları okunabilir
değildir. Bunun yerine bir dosyadan hareketle anahtar değeri uyduran
ftok isimli bir POSIX fonksiyonundan faydalanılabilmektedir.
ftok dosyanın i-node numarası, dosyanın içerisinde bulunduğu aygıt
numarası ve bizim verdiğimiz değeri kombine ederek bir anahtar
uydurmaktadır. Böylece iki program aynı dosya isminden hareketle ortak
paylaşım alanı oluşturabilmektedir.
-----
-----*/

/* proc1.c */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/ipc.h>
#include <sys/shm.h>

void exit_sys(const char *msg);

int main(void)
{
    int shmid;
    void *addr;
    char *str;
    key_t key;

    if ((key = ftok("myfile", 123)) == -1)
        exit_sys("ftok");

    if ((shmid = shmget(key, 8192, IPC_CREAT|0600)) == -1)
        exit_sys("shmget");

    if ((addr = shmat(shmid, NULL, 0)) == (void *)-1)
        exit_sys("shmat");

    str = (char *)addr;

    strcpy(str, "this is a test...");

    printf("Press ENTER to continue...\n");
    getchar();
}

```

```

    shmdt(addr);

    if (shmctl(shmid, IPC_RMID, NULL) == -1)
        exit_sys("shmctl");

    return 0;
}

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

    exit(EXIT_FAILURE);
}

/* proc2.c */

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/ipc.h>
#include <sys/shm.h>

void exit_sys(const char *msg);

int main(void)
{
    int shmid;
    void *addr;
    char *str;
    key_t key;

    if ((key = ftok("myfile", 123)) == -1)
        exit_sys("ftok");

    if ((shmid = shmget(key, 8192, IPC_CREAT|0600)) == -1)
        exit_sys("shmget");

    if ((addr = shmat(shmid, NULL, 0)) == (void *)-1)
        exit_sys("shmat");

    str = (char *)addr;

    puts(str);

    shmdt(addr);

    return 0;
}

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

```



```

    exit(EXIT_FAILURE);
}

/
*-----
-----
    POSIX paylaşılan bellek alanları shm_open, ftruncate, mmap, munmap ve
    shm_unlink fonksiyonları yardımıyla oluşturulmaktadır.
    shm_open fonksiyonuyla yaratılmış olan paylaşılan bellek alanı nesnesi
    sistem reboot edilene kadar ya da shm_unlink fonksiyonuyla
    silme yapılana kadar kalmaya devam etmektedir. mmap fonksiyonu UNIX
    türevi sistemlerde kullanılan genel amaçlı bir mapping
    fonksiyonudur.
-----
-----*/

/* proc1.1 */

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

#define SHM_PATH      "/this_is_a_text"

void exit_sys(const char *msg);

int main(void)
{
    int fdshm;
    void *addr;
    char *str;

    if ((fdshm = shm_open(SHM_PATH, O_CREAT|O_RDWR, S_IRUSR|S_IWUSR|S_IRGRP|
        S_IROTH)) == -1)
        exit_sys("shm_open");

    if (ftruncate(fdshm, 4096) == -1)
        exit_sys("ftruncate");

    if ((addr = mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_SHARED, fdshm,
        0)) == MAP_FAILED)
        exit_sys("mmap");

    str = (char *)addr;
    strcpy(str, "this is a test...");

    printf("press ENTER to continue...\n");
    getchar();

    munmap(addr, 4096);

```

```

    close(fdshm);

    if (shm_unlink(SHM_PATH) == -1)
        exit_sys("shm_unlink");

    return 0;
}

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

    exit(EXIT_FAILURE);
}

/* proc2.c */

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

#define SHM_PATH        "/this_is_a_text"

void exit_sys(const char *msg);

int main(void)
{
    int fdshm;
    void *addr;
    char *str;

    if ((fdshm = shm_open(SHM_PATH, O_RDWR, 0)) == -1)
        exit_sys("shm_open");

    if ((addr = mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_SHARED, fdshm,
        0)) == MAP_FAILED)
        exit_sys("mmap");

    str = (char *)addr;

    puts(str);

    munmap(addr, 4096);

    close(fdshm);

    return 0;
}

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

```

```

    exit(EXIT_FAILURE);
}

/
*-----
-----
    Bir dosyanın tamamının ya da belli bir kısmının otomatik biçimde belleğe
    çekilip dosya fonksiyonları yerine göstericilerle
    bellek üzerinde dosya işlemleri yapmaya "bellek tabanlı dosyalar (memory
    mapped files) denilmektedir". Bellek tabanlı dosyalar
    için open, mmap, munmap, close fonksiyonlarından faydalanılır. Dosya önce
    open fonksiyonuyla açılır sonra mmap fonksiyonu ile
    dosyanın tamamı ya da bir parçası belleğe çekilir.
-----
-----*/

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

void exit_sys(const char *msg);

int main(void)
{
    int fd;
    void *addr;
    char *str;
    int len;
    int i;

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

    len = (int)lseek(fd, 0, SEEK_END);

    if ((addr = mmap(NULL, len, PROT_READ|PROT_WRITE, MAP_SHARED, fd, 0)) ==
        MAP_FAILED)
        exit_sys("mmap");

    str = (char *)addr;

    for (i = 0; i < len; ++i)
        putchar(str[i]);
    putchar('\n');

    munmap(addr, len);

    close(fd);

    return 0;
}

void exit_sys(const char *msg)

```

```

{
    perror(msg);

    exit(EXIT_FAILURE);
}

/
*-----
-----
    Bir dosya bellek tabanlı biçimde açıldığında bellek üzerinde dosya
    içeriği güncellendiğinde bu değişiklikler hemen bu dosyayı
    open ile açıp read ile okuyan proseslerde gözükür mü? POSIX
    standardlarına göre bunun bir garantisi yoktur. Bizim bunu garanti
    altına almamız için sync isimli POSIX fonksiyonunu MS_SYNC bayrak değeri
    ile çağırmamız gerekir. Ancak Linux işletim sistemi
    dosya sistemi gerçekleştiriminden dolayı bu tür değişikliklerin başka
    prosesler tarafından hiç msync yapılmadan görülmesine
    olanak vermektedir. (Yani Linux'ta aslında başka bir proses open
    fonksiyonuyla dışarıdan dosyayı açıp read ile okuduğu zaman
    çekirdek dosyanın ilgili parçasının o anda belleğe map edildiğini
    bildiği için zaten read fonksiyonu diske başvurmadan onu
    bellekten almaktadır.) Aşağıdaki programda dosya bellekte strcpy
    fonksiyonu ile güncellenmiştir. Bu güncellemede sonra
    klavyeden ENTER tuşuna basılana kadar bekleme yapılmıştır. Bu noktada
    başka bir terminalden gireek dosyayı cat ile açıp okumayı
    deneyiniz. Tabii biz bellek tabanlı dosyayı bellekte güncellediğimizde
    dosyanın diskteki mevcudiyeti güncellenmek zorunda değildir.
    Bunu garanti altına almak için msync fonksiyonu MS_SYNC bayrağı ile
    çağrılmalıdır.
    -----
    -----*/

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

void exit_sys(const char *msg);

int main(void)
{
    int fd;
    void *addr;
    char *str;
    int len;
    int i;

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

    len = (int)lseek(fd, 0, SEEK_END);

```

```

if ((addr = mmap(NULL, len, PROT_READ|PROT_WRITE, MAP_SHARED, fd, 0)) ==
    MAP_FAILED)
    exit_sys("mmap");

str = (char *)addr;

for (i = 0; i < len; ++i)
    putchar(str[i]);
putchar('\n');

strcpy(str, "aaaaaaaaaaaaaaaaaaaaa");
printf("press ENTER to continue...\n");
getchar();

munmap(addr, len);

close(fd);

return 0;
}

```

```

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

    exit(EXIT_FAILURE);
}

```

/

*-----

Yukarıdaki işlemin tersine ilişkin örnek de aşağıdadır. Yani dosya bellek tabanlı bir biçimde açılmış fakat o noktada ENTER tuşuna basılana kadar beklenmiştir. Dosyayı dışarıdan bir editör yardımıyla güncelleyiniz. Linux msync fonksiyonuna gereksinim duyulmadan bellekteki halini güncelleyecektir. (Tabii aslında yine editör güncellemeyi disk üzerinde değil zaten bellekteki map edilmiş alan üzerinde yapmaktadır.) Fakat POSIX standartlarında bunun garanti edilmesi için msync fonksiyonunun MS_INVALIDATE bayrağı ile çağırılması gerekmektedir.

-----*/

```

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

void exit_sys(const char *msg);

int main(void)
{
    int fd;

```

```

void *addr;
char *str;
int len;
int i;

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

len = (int)lseek(fd, 0, SEEK_END);

if ((addr = mmap(NULL, len, PROT_READ|PROT_WRITE, MAP_SHARED, fd, 0)) ==
    MAP_FAILED)
    exit_sys("mmap");

printf("press ENTER to continue...\n");
getchar();

str = (char *)addr;

for (i = 0; i < len; ++i)
    putchar(str[i]);
putchar('\n');

munmap(addr, len);

close(fd);

return 0;
}

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

    exit(EXIT_FAILURE);
}

/
*-----
-----
Her ne kadar Linux için gerek olmasa da taşınabilir bir program için
POSIX standartlarında belirttiği gibi msync fonksiyonunun
MS_SYNC ya da MS_INVALIDATE bayraklarıyla çağrılması gerekir. MS_SYNC
bellekten diskteki dosyaya, MS_INVALIDATE ise diskteki
dosyadan belleğe tazeleme yapmaktadır.
-----
-----*/

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

```

```

void exit_sys(const char *msg);

int main(void)
{
    int fd;
    void *addr;
    char *str;
    int len;
    int i;

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

    len = (int)lseek(fd, 0, SEEK_END);

    if ((addr = mmap(NULL, len, PROT_READ|PROT_WRITE, MAP_SHARED, fd, 0)) ==
        MAP_FAILED)
        exit_sys("mmap");

    printf("press ENTER to continue...\n");
    getchar(); getchar();

    if (msync(addr, len, MS_INVALIDATE) == -1)
        exit_sys("msync");

    str = (char *)addr;

    for (i = 0; i < len; ++i)
        putchar(str[i]);
    putchar('\n');

    strcpy(str, "xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx");

    printf("press ENTER to continue...\n");
    getchar();

    if (msync(addr, len, MS_SYNC) == -1)
        exit_sys("msync");

    munmap(addr, len);

    close(fd);

    return 0;
}

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

    exit(EXIT_FAILURE);
}

```

/

*-----

mmap fonksiyonunda MAP_PRIVATE yapılan değişikliklerin asıl dosyaya yansıtılmayacağı gerektiğinde swap dosyasına yapılacağı anlamına gelir. Bu copy on write mekanizmasıdır. MAP_SHARED yazılanların asıl dosyaya aktarılacağı anlamına gelmektedir. Prosesler paylaşılan bellek alanlarını ya da dosyayı MAP_PRIVATE ile map ederlerse birbirlerinin yazdıklarını göremezler. Bunun için MAP_SHARED gerekmektedir. Linux sistemlerinde POSIX'te belirtilen bayraklardan çok daha fazlası vardır. Bu ekstra bayrakların en önemlisi MAP_ANONYMOUS bayrağıdır. Buna "anonymous mapping" denilmektedir. Bu bayrak MAP_PRIVATE ve MAP_SHARED ile birlikte kullanılabilir. Anonymous mapping dosya ile ilgili olmadan yani yalnızca sanal bellek alanında sayfa tabanlı tahsisat yapmak için kullanılmaktadır. Dolayısıyla MAP_ANOYMOUS bayrağı belirtildiğinde artık mmap fonksiyonunun son iki parametresi (fd, offset) dikkate alınmamaktadır. MAP_ANOYMOUS|MAP_PRIVATE mapping sanal bellek alanında dosyadan bağımsız tahsisat yapmak için ancak alt proste fork yapıldığında bu alanların farklı kopyalarının kullanılması için tercih edilmektedir. MAP_ANONYMOUS|MAP_SHARED ise fork işleminden sonra üst ve alt proseslerin aynı bellek bölgesini görmesi için kullanılır. Aşağıdaki örnekte üst ve alt prosesler bu yolla haberleşebilmektedir.

-----*/

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

void exit_sys(const char *msg);

int main(void)
{
    void *addr;
    char *str;
    pid_t pid;

    if ((addr = mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_ANONYMOUS|
        MAP_SHARED, 0, 0)) == MAP_FAILED)
        exit_sys("mmap");
    str = (char *)addr;

    if ((pid = fork()) == -1)
        exit_sys("fork");

    if (pid != 0) { /* parent process */
        strcpy(str, "this is a test...");
        if (waitpid(pid, NULL, 0) == -1)
            exit_sys("waitpid");
    }
}
```



```

else {
    /* child process */
    sleep(1);
    puts(str);
}

munmap(addr, 4096);

return 0;
}

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

    exit(EXIT_FAILURE);
}

/
*-----
-----
Linux'ta POSIX shared memory nesneleri aslında /dev/shm dizininde birer
dosya biçiminde görünmektedir. Yani bizim shm_open
fonksiyonuyla kök dizinde bir isimle açtığımız dosya gerçekte /dev/shm
dizinde yaratılmaktadır. /dev/shm dizini tmpfs
denilen RAM tabanlı dosya sistemine ilişkindir. Yani /dev/shm dizini
içerisindeki dosyalar aslında diskte değil RAM'da tutulmaktadır.
Sistem reboot edilene kadar ya da shm_unlink fonksiyonu çağrılana kadar
o arada durmaya devam ederler.
-----
-----*/

csd@csd-vm:~/Study/Unix-Linux-SysProg$ ls /dev/shm -l
total 4
-rw-r--r-- 1 csd study 4096 Jun 11 21:47 this_is_a_text

/
*-----
-----
Sistem 5 mesaj kuyrukları msgget fonksiyonuyla yaratılır ya da açılır,
msgsnd fonksiyonuyla kuyruğa mesaj gönderilir.
msgrcv fonksiyonu ile kuyruktan mesaj alınır. msgctl IPC_RMID kodu ile
mesaj kuyruğu silinir. Mesajı gönderirken mesaj bilgisinin
önünde onun türünü belirten long bir type alanı olmalıdır. Mesajın
uzunluğuna bu alan dahil değildir. Mesaj alınırken de
her zaman mesajın yerleştirileceği alanın önünde yine long bir type
alanı olmalıdır. Mesajı alan taraf onun türünü de alır.
Böylece farklı prosesler aynı kuyruktan farklı type değerlerine ilişkin
mesajları okuyabilirler. 0 type değeri kullanılmaz,
özel anlam ifade eder.
-----
-----*/

/* proc1.c */

#include <stdio.h>

```

```

#include <stdlib.h>
#include <string.h>
#include <ctype.h>
#include <sys/ipc.h>
#include <sys/msg.h>

#define MAX_LEN      1024

void exit_sys(const char *msg);

typedef struct tagMSG {
    long type;
    char buf[MAX_LEN];
} MSG;

int main(void)
{
    key_t key;
    int msgid;
    MSG msg;
    char *str;
    int ch;

    if ((key = ftok("mymsg", 123)) == -1)
        exit_sys("ftok");

    if ((msgid = msgget(key, IPC_CREAT|0666)) == -1)
        exit_sys("msgget");

    for (;;) {
        printf("Bir type değeri ve yanına bir yazı giriniz:");
        scanf("%ld", &msg.type);
        while ((ch = getchar()), isspace(ch))
            ;
        ungetc(ch, stdin);
        fgets(msg.buf, MAX_LEN, stdin);
        if ((str = strchr(msg.buf, '\n')) != NULL)
            *str = '\0';
        if (msgsnd(msgid, &msg, strlen(msg.buf) + 1, 0) == -1)
            exit_sys("msgsnd");
        if (!strcmp(msg.buf, "quit"))
            break;
    }

    if (msgctl(msgid, IPC_RMID, 0) == -1)
        exit_sys("msgctl");

    return 0;
}

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

    exit(EXIT_FAILURE);
}

```

```

}

/* proc2.c */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/ipc.h>
#include <sys/msg.h>

#define MAX_LEN      1024

typedef struct tagMSG {
    long type;
    char buf[MAX_LEN];
} MSG;

void exit_sys(const char *msg);

int main(void)
{
    key_t key;
    int msgid;
    MSG msg;

    if ((key = ftok("mymsg", 123)) == -1)
        exit_sys("ftok");

    if ((msgid = msgget(key, 0666)) == -1)
        exit_sys("msgget");

    for (;;) {
        if (msgrcv(msgid, &msg, MAX_LEN, 0, 0) == -1)
            exit_sys("msgrcv");
        printf("type: %ld, message: \"%s\"\n", msg.type, msg.buf);
        if (!strcmp(msg.buf, "quit"))
            break;
    }

    return 0;
}

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

    exit(EXIT_FAILURE);
}

/
*-----
-----
msgctl IPC_STAT koduyla çekirdek tarafından oluşturulan msgqid_ds yapısı
içerisindeki değerleri alabiliriz. Burada kuyrukta

```

toplam kaç mesajın olduğu, kuyrukta toplam kaç byte'ın bulunduğu, kuyrukta olabilecek maksimum byte sayısı gibi bilgiler de vardır.

```
-----*/

/* msgctl.c */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <getopt.h>
#include <sys/ipc.h>
#include <sys/msg.h>

void exit_sys(const char *msg);

/* ./msgctl [-q][Q] [id or key] */

int main(int argc, char *argv[])
{
    int result;
    int q_flag = 0, Q_flag = 0;
    char *arg;
    long argval;
    int msgid;
    struct msqid_ds msqds;

    opterr = 0;
    while ((result = getopt(argc, argv, "q:Q:")) != -1) {
        switch (result) {
            case 'q':
                q_flag = 1;
                arg = optarg;
                break;
            case 'Q':
                Q_flag = 1;
                arg = optarg;
                break;
            case '?':
                if (optopt == 'q' || optopt == 'Q')
                    fprintf(stderr, "c switch without argument!...\n");
                else
                    fprintf(stderr, "invalid switch: -%c\n", optopt);
                exit(EXIT_FAILURE);
        }
    }

    if (q_flag + Q_flag == 0) {
        fprintf(stderr, "neither -q nor -Q flag was given!\n");
        exit(EXIT_FAILURE);
    }

    if (q_flag + Q_flag > 1) {
        fprintf(stderr, "both -q and -Q flag must not be given!\n");
    }
}
```

```

        exit(EXIT_FAILURE);
    }

    if (optind != argc) {
        fprintf(stderr, "too many arguments!...\n");
        exit(EXIT_FAILURE);
    }

    argval = strtol(arg, NULL, 10);

    if (Q_flag) {
        if ((msgid = msgget(argval, 0)) == -1)
            exit_sys("msgget");
    }
    else
        msgid = argval;

    if (msgctl(msgid, IPC_STAT, &msqds) == -1)
        exit_sys("msgctl");

    printf("Number of messages in queue: %lu\n", (unsigned
        long)msqds.msg_qnum);
    printf("Maximum number of bytes allowed in queue: %lu\n", (unsigned
        long)msqds.msg_qbytes);
    printf("Current number of bytes in queue: %lu\n", (unsigned
        long)msqds.__msg_cbytes);

    return 0;
}

```

```

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

    exit(EXIT_FAILURE);
}

```

/

*-----

POSIX mesaj kurukları mq_open fonksiyonuyla yaratılır ya da açılır. mq_send fonksiyonuyla kuyruğa mesaj bırakılır, mq_receive fonksiyonuyla da kuyruktaki mesaj alınır. Kuyruğun mq_attr ile temsil edilen dört özelliği vardır. Bu özellikler kuyruk yaratılırken mq_open fonksiyonunda girilebilmektedir. Kuyruk özellikleri mq_getattr fonksiyonuyla alınır. Eğer kuyruk yaratılırken bu özellikler ilgili parametre NULL geçilerek belirtilmemişse default özelliklerle kuyruk yaratılmaktadır. POSIX mesaj kuyrukları tıpkı POSIX paylaşılan bellek alanları gibi "kernel persistent" biçimdedir. Yani mq_unlink fonksiyonuyla silinmezlerse reboot işlemine kadar yaşamaya devam ederler. mq_receive fonksiyonunda buffer uzunluğunun en az mq_attr'de belirtilen uzunluk kadar olması gerekmektedir. Kuyruklar da dosyalar gibi mq_close fonksiyonuyla kapatılırlar. Linux işletim sistemi POSIX kuyruklarının

handle değerlerini dosya betimleyicisi biçiminde almaktadır. Ancak diğer sistemlerde kuyruk handle değerlerinin dosya betimleyicisi olması zorunlu değildir. POSIX mesaj kuyrukları da kuyruk için ayrılan alan dolduysa mq_send fonksiyonunda, kuyrukta hiç mesaj yoksa mq_receive fonksiyonunda bloke olurlar. Tabii O_NONBLOCK aış bayrağı ile nonblocking işlemler yapılabilir. Bu durumda mq_receive ve mq_send fonksiyonları bloke olmazlar -1 değerine geri dönerler, errno değişkeni de bu durumda EAGAIN değeriyle set edilmektedir.

Linux sistemlerinde mq_setattr fonksiyonunda mq_attr yapısının yalnızca flags elemanın set edilmesimümkündür.

```
-----*/  
  
/* proc1.c */  
  
#include <stdio.h>  
#include <stdlib.h>  
#include <string.h>  
#include <ctype.h>  
#include <sys/stat.h>  
#include <mqueue.h>  
  
#define MSG_QUEUE_PATH      "/this_is_a_message_queue_last"  
  
void exit_sys(const char *msg);  
  
int main(void)  
{  
    mqd_t mq;  
    char *buf;  
    struct mq_attr attr;  
    int prio;  
    int ch;  
    char *str;  
  
    if ((mq = mq_open(MSG_QUEUE_PATH, O_CREAT|O_WRONLY, S_IRUSR|S_IWUSR|  
        S_IRGRP|S_IROTH, NULL)) == -1)  
        exit_sys("mq_open");  
  
    if (mq_getattr(mq, &attr) == -1)  
        exit_sys("mq_getattr");  
  
    printf("Default attributes of the queue:\n");  
    printf("Max msg: %ld\n", attr.mq_maxmsg);  
    printf("Max msg size: %ld\n", attr.mq_msgsize);  
    printf("Number of messages in queue: %ld\n", attr.mq_curmsgs);  
    printf("-----\n");  
  
    if ((buf = (char *)malloc(attr.mq_msgsize)) == NULL)  
        exit_sys("malloc");  
  
    for (;;) {  
        printf("Bir öncelik derecesi ve yanına bir yazı giriniz:");  
        scanf("%d", &prio);
```

```

        while ((ch = getchar()), isspace(ch))
            ;
        ungetc(ch, stdin);
        fgets(buf, MAX_LEN, stdin);
        if ((str = strchr(buf, '\n')) != NULL)
            *str = '\0';

        if (mq_send(mq, buf, strlen(buf) + 1, prio))
            exit_sys("mq_send");

        if (!strcmp(buf, "quit"))
            break;
    }

    free(buf);
    mq_close(mq);

    if (mq_unlink(MSG_QUEUE_PATH) == -1)
        exit_sys("mq_unlink");

    return 0;
}

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

    exit(EXIT_FAILURE);
}

/* proc2.c */

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

#define MSG_QUEUE_PATH    "/this_is_a_message_queue_last"

void exit_sys(const char *msg);

int main(void)
{
    mqd_t mq;
    char *buf;
    unsigned int prio;
    ssize_t result;
    struct mq_attr attr;

    if ((mq = mq_open(MSG_QUEUE_PATH, O_RDONLY)) == -1)
        exit_sys("mq_open");

    if (mq_getattr(mq, &attr) == -1)
        exit_sys("mq_getattr");

```

```

printf("Default attributes of the queue:\n");
printf("Max msg: %ld\n", attr.mq_maxmsg);
printf("Max msg size: %ld\n", attr.mq_msgsize);
printf("Number of messages in queue: %ld\n", attr.mq_curmsgs);
printf("-----\n");

if ((buf = (char *)malloc(attr.mq_msgsize)) == NULL)
    exit_sys("malloc");

for (;;) {
    if ((result = mq_receive(mq, buf, MAX_LEN, &prio)) == -1)
        exit_sys("mq_receive");
    printf("%ld bytes received at priority %u: \"%s\"\n", (long)result,
        prio, buf);
    if (!strcmp(buf, "quit"))
        break;
}

free(buf);

mq_close(mq);

return 0;
}

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

    exit(EXIT_FAILURE);
}

/
*-----
-----
mq_open fonksiyonunda kuyruk özellikleri mq_attr parametresinde
belirtilebilir. mq_open fonksiyonu bu yapıdaki yalnızca mq_maxmsg ve
mq_msgsize
elemanlarını dikkate almaktadır. Kuyruk yaratıldıktan sonra ise yalnızca
mq_setattr ile yapının mq_flags elemanı değiştirilebilmektedir.
Ancak kuyruktaki maksimum mesaj sayısı mevcut Linux sistemlerinde
default durumda en fazla 10 yapılabilmektedir. Böylece biz mesaj
büyüklüğünü değiştirip tampon alanımızı ona uygun oluşturabiliriz.
Çekirdek tarafından izin verilen maksimum değerler
/proc/sys/fs/mqueue dizinin içerisindeki dosyalarda belirtilebilmektedir.
Etkin user id'si 0 olan prosesler bu dosyadaki limitlere takılmazlar.
-----
-----*/

/* proc1.c */

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

```



```

#include <ctype.h>
#include <sys/stat.h>
#include <mqueue.h>

#define MSG_QUEUE_PATH      "/this_is_a_message_queue_last"
#define MAX_MSG_LEN        1024
#define MAX_MSG             10

void exit_sys(const char *msg);

int main(void)
{
    mqd_t mq;
    char buf[MAX_MSG_LEN];
    struct mq_attr attr;
    int prio;
    int ch;
    char *str;

    attr.mq_maxmsg = MAX_MSG;
    attr.mq_msgsize = MAX_MSG_LEN;

    if ((mq = mq_open(MSG_QUEUE_PATH, O_CREAT|O_WRONLY, S_IRUSR|S_IWUSR|
        S_IRGRP|S_IROTH, &attr)) == -1)
        exit_sys("mq_open");

    if (mq_getattr(mq, &attr) == -1)
        exit_sys("mq_getattr");

    printf("Default attributes of the queue:\n");
    printf("Max msg: %ld\n", attr.mq_maxmsg);
    printf("Max msg size: %ld\n", attr.mq_msgsize);
    printf("Number of messages in queue: %ld\n", attr.mq_curmsgs);
    printf("-----\n");

    for (;;) {
        printf("Bir öncelik derecesi ve yanına bir yazı giriniz:");
        scanf("%d", &prio);
        while ((ch = getchar()), isspace(ch))
            ;
        ungetc(ch, stdin);
        fgets(buf, MAX_MSG_LEN, stdin);
        if ((str = strchr(buf, '\n')) != NULL)
            *str = '\0';

        if (mq_send(mq, buf, strlen(buf) + 1, prio))
            exit_sys("mq_send");

        if (!strcmp(buf, "quit"))
            break;
    }

    mq_close(mq);

    if (mq_unlink(MSG_QUEUE_PATH) == -1)

```

```

        exit_sys("mq_unlink");

    return 0;
}

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

    exit(EXIT_FAILURE);
}

/* proc2.c */

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

#define MSG_QUEUE_PATH    "/this_is_a_message_queue_last"
#define MAX_MSG_LEN      1024

void exit_sys(const char *msg);

int main(void)
{
    mqd_t mq;
    char buf[MAX_MSG_LEN];
    unsigned int prio;
    ssize_t result;
    struct mq_attr attr;

    if ((mq = mq_open(MSG_QUEUE_PATH, O_RDONLY)) == -1)
        exit_sys("mq_open");

    if (mq_getattr(mq, &attr) == -1)
        exit_sys("mq_getattr");

    printf("Default attributes of the queue:\n");
    printf("Max msg: %ld\n", attr.mq_maxmsg);
    printf("Max msg size: %ld\n", attr.mq_msgsize);
    printf("Number of messages in queue: %ld\n", attr.mq_curmsgs);
    printf("-----\n");

    for (;;) {
        if ((result = mq_receive(mq, buf, MAX_MSG_LEN, &prio)) == -1)
            exit_sys("mq_receive");
        printf("%ld bytes received at priority %u: \"%s\"\n", (long)result,
            prio, buf);
        if (!strcmp(buf, "quit"))
            break;
    }

    mq_close(mq);

```

```

    return 0;
}

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

    exit(EXIT_FAILURE);
}

```

```

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

```

Sistem Limitleri: POSIX standartlarında sisteme ilişkin çeşitli limit değerler <limits.h> dosyası içerisinde sembolik sabitler olarak bildirilmiştir. <limits.h> içerisinde sembolik sabitler çeşitli başlıklar altında gruplandırılmıştır. Bu başlıklar ve anlam ları şöyledir:

Runtime Invariant Values (Possibly Indeterminate): Buradaki sembolik sabitlerin değeri ilgili sistemde değişmemektedir. Eğer buradaki değerler _POSIX_XXX ile belirtilen minimum değerlerden büyük fakat belirsiz (unpecified) ise bu sembolik sabitler <limits.h> içerisinde bulunmamak zorundadır. Bu durumda bunların değerlerini almak için sysconf fonksiyonu çağrılmalıdır. Fakat sysconf fonksiyonunun bu değerler için vereceği değerler aynıdır. Burada belirsiz demekle ilgili değerin bazı donanım ve sistem özelliklerine göre derleme zamanının belirlenemiyor lması anlatılmaktadır.

Pathname Variable Values: Buradaki sembolik sabit değerleri eğer _POSIX_XXX'ten büyük fakat dosya sistemine bağlı olarak değişiyorsa <limits.h> içerisinde bulunmamak zorundadır. Bu duurmda bunların gerçek değerleri pathonf ya da fpathconf fonksiyonlarıyla alınmalıdır.

Runtime Increasable Values: Buradaki sembolik sabitlerin hepsi <limits.h> içerisinde define edilmek zorundadır. Fakat <limits.h> içerisinde define edilmiş değerler o sistemdeki minimum değerlerdir. Çünkü bu değerler çeşitli biçimlerde (örneğin setrlimit fonksiyonu ile) artırılmış olabilmektedir. O sistemdeki o andaki gerçek değerler yine sysconf fonksiyonuyla elde edilmelidir.

O halde belli bir özellik eğer "Runtime Invariant Values" ya da "Pathname Variable Values" grububdaysa önce ifdef ile ilgili sembolik sabitin define edilmiş olup olmadığına bakılmalı eğer bu sembolik sabit define edilmişse o değeri almalı, define edilmemişse sysconf ya da pathconf/fpathconf çağırması yapılmalıdır. "Runtime Increasable Values" lar için doğrudan o değer alınabilir ya da sysconf ile o andaki gerçek değerler alınabilir. Herhangi bir özelliği elde etmek için bir fonksiyon yazmak iyi bir teknik olabilir.

sysconf fonksiyonu parametre olarak _POSIX_XXX için _SC_XXX değerlerini kullanmaktadır. Fonksiyon başarısız olursa -1 değerine geri döner ve errno EINVAL olarak set edilir. Başarı durumunda ilgili limite geri döner. Değerin belirlenememesi (indeterminate)

durumunda ise sysconf -1'e geri döner errno değerini değiştirmez. Genellikle programcılar değer belirlenememişse yüksek bir değer uydururlar.

```
-----*/

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

void exit_sys(const char *msg);

long child_max(void)
{
    static long result = 0;

#define CHILD_MAX_INDETERMINATE_GUESS    4096

#ifdef CHILD_MAX
    result = CHILD_MAX;
#else
    if (result == 0) {
        errno = 0;
        if ((result = sysconf(_SC_CHILD_MAX)) == -1 && errno == 0)
            result = CHILD_MAX_INDETERMINATE_GUESS;
    }
#endif

    return result;
}

int main(void)
{
    long cmax;

    if ((cmax = child_max()) == -1)
        exit_sys("child_max");

    printf("Child max = %ld\n", cmax);

    return 0;
}

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

    exit(EXIT_FAILURE);
}

/*-----
```

Bir dosyanın maksimum ismi ne olabilir? Eğer NAME_MAX sembolik sabiti <limits.h> içerisinde bildirilmişse onu almalıyız, bildirilmemişse pathconf ile asıl değeri elde etmeliyiz. pathconf bizden ilgilendiğimiz dosya sistemine ilişkin bir dizin'in yol ifadesini de parametre olarak almaktadır. pathconf ve fpathconf fonksiyonlarının ikinci parametreleri _POSIX_XXX için _PC_XXX biçimindeki sembolik sabitlerdir.

```
-----*/

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

void exit_sys(const char *msg);

long name_max(const char *dirpath)
{
    static long result = 0;

#define NAME_MAX_INDETERMINATE_GUESS    1024

#ifdef NAME_MAX
    result = NAME_MAX;
#else
    if (result == 0) {
        errno = 0;
        if ((result = (pathconf(dirpath, _PC_NAME_MAX))) == -1 && errno ==
            0)
            result = NAME_MAX_INDETERMINATE_GUESS;
    }
#endif

    return result;
}

int main(void)
{
    long nmax;

    if ((nmax = name_max(".")) == -1)
        exit_sys("name_max");

    printf("Name max = %ld\n", nmax);

    return 0;
}

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

    exit(EXIT_FAILURE);
}
```

```

}

/
*-----
-----
    Bir yol ifadesini yerleştireceğimiz char türden dizinin dinamik tahsis
    edilmesi örneği (Stevens kitabında eski POSIX versiyon
    larını da dikkate almış. Fakat artık gereksiz olabilmektedir.)
-----
-----*/

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

void exit_sys(const char *msg);

long path_max(void)
{
    static long result = 0;

#define PATH_MAX_INDETERMINATE_GUESS    4096

#ifdef PATH_MAX
    result = PATH_MAX;
#else
    if (result == 0) {
        errno = 0;
        if ((result = pathconf("/", _PC_PATH_MAX)) == -1 && errno == 0)
            result = PATH_MAX_INDETERMINATE_GUESS;
    }
#endif

    return result;
}

int main(void)
{
    long pmax;
    char *path;

    if ((pmax = path_max()) == -1)
        exit_sys("path_max");

    printf("Path max = %ld\n", pmax);

    if ((path = (char *)malloc((size_t)(pmax))) == NULL)
        exit_sys("malloc");

    free(path);

    return 0;
}

```

```

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

    exit(EXIT_FAILURE);
}

/
*-----
-----
Prosesin kaynak liitlerini elde etmek için getrlimit fonksiyonu
kullanılır. Bu fonksiyonun birinci parametresi hangi kaynağın
elde edileceğini beelirten RLIMIT_XXXX değeridir. İkinci parametresi ise
struct rlimit türünden bir yapının adresidir.
Prosesin her kaynağının "soft limit" ve "hard limit" denilen iki eşik
değeri vardır. İşletim sisteemi "soft limit" kontrolü yapmaktadır.
Ancak programcı setrlimit fonksiyonu ile soft limiti hard limite kadar
yükseltebilir. Hard limiti ise ancak etkin kullanıcı id'si
0 olan prosesler ya da ilgili yeteneğe (capability) sahip prosesler
yükseltebilirler. Örneğin aşağıdaki programda prosesin açabileceği
maksimum dosya sayısının soft ve hard limitleri getrlimit fonksiyonuyla
elde edilip yazdırılmıştır. Buradan 1024 ve 4096 değerleri elde
edilmiştir.
Yani biz prosesimizde ancak 1024 tane dosya açabiliriz. Ancak bunu
setrlimit ile ancak 4096'ya çıkartabiliriz.
-----
-----*/

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/resource.h>

void exit_sys(const char *msg);

int main(void)
{
    struct rlimit rlim;

    if (getrlimit(RLIMIT_NOFILE, &rlim) == -1)
        exit_sys("getrlimit");

    printf("Soft limit: %lu\n", (unsigned long)rlim.rlim_cur);
    printf("Hard limit: %lu\n", (unsigned long)rlim.rlim_max);

    return 0;
}

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

    exit(EXIT_FAILURE);
}

```

```
/
*-----
-----
Biz bir proses limitini setrlimit fonksiyonu ile ancak hard limit kadar
yükseltebiliriz. Onun ötesine yükseltebilmemiz için
hard limiti yükseltmemiz gerekir. İşte setrlimit fonksiyonu ile hard
limiti yükseltebilmemiz için bizim root prosesi olmamız
gerekir. Aşağıdaki programda prosesin açabileceği dosya sayısı 1024'ten
hard limit olan 4096'ya yükseltilmiştir. Tabii proses
setrlimit fonksiyonu ile kendi hard limitini düşürebilir.
-----
-----*/
```

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

void exit_sys(const char *msg);

int main(void)
{
    struct rlimit rlim;
    int i;
    int fd;

    if (getrlimit(RLIMIT_NOFILE, &rlim) == -1)
        exit_sys("getrlimit");

    printf("Soft limit: %lu\n", (unsigned long)rlim.rlim_cur);
    printf("Hard limit: %lu\n", (unsigned long)rlim.rlim_max);

    for (i = 0;; ++i) {
        if ((fd = open("sample.c", O_RDONLY)) == -1)
            break;
        printf("%d ", fd);
        fflush(stdout);
    }
    printf("\n");

    for (i = 3; i < 1024; ++i)
        close(i);

    rlim.rlim_cur = 4096;
    if (setrlimit(RLIMIT_NOFILE, &rlim) == -1)
        exit_sys("setrlimit");

    for (i = 0;; ++i) {
        if ((fd = open("sample.c", O_RDONLY)) == -1)
            break;
        printf("%d ", fd);
        fflush(stdout);
    }
}
```



```

    printf("\n");

    return 0;
}

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

    exit(EXIT_FAILURE);
}

/
*-----
-----
Aşağıdaki örnekte prosesin açabileceği dosya sayısı hard limit ve soft
limit yükseltilerek 8192'ye çıkartılmıştır.
Tabii programın sudo ile çalıştırılması gerekir.
-----
-----*/

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

void exit_sys(const char *msg);

int main(void)
{
    struct rlimit rlim;
    int i;
    int fd;

    if (getrlimit(RLIMIT_NOFILE, &rlim) == -1)
        exit_sys("getrlimit");

    printf("Soft limit: %lu\n", (unsigned long)rlim.rlim_cur);
    printf("Hard limit: %lu\n", (unsigned long)rlim.rlim_max);

    rlim.rlim_cur = 8192;
    rlim.rlim_max = 8192;
    if (setrlimit(RLIMIT_NOFILE, &rlim) == -1)
        exit_sys("setrlimit");

    for (i = 0;; ++i) {
        if ((fd = open("sample.c", O_RDONLY)) == -1)
            break;
        printf("%d ", fd);
        fflush(stdout);
    }
    printf("\n");

    return 0;
}

```

```

}

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

    exit(EXIT_FAILURE);
}

/
*-----
-----
komut satırındaki ulimit internal komutu -a ile shell prosesinin tüm
soft limitlerini göstermektedir. Soft limitler -S ile hard
limitler -H ile gösterilir. Aynı zamanda bu limitler değiştirile de
bilmektedir. Prosesin tüm limitleri fork işlemi
sırasında üst prosesten alt prosese aktarılmaktadır. Yani biz örneğin
kabuk üzerinde aşağıdaki gibi kabuğun açabileceği
dosya sayısının soft limitini değiştirebiliriz:

ulimit -S -n 4096

Şimdi artık kabuktan çalıştıracığımız programlar bu limiti
kullanacaktır.

Proseslerin soft ve hard limitlerini kalıcı hale getirmek için Linux
sistemlerinde /etc/security/limits.conf dosyası
kullanılmaktadır. Bu dosyanın formatını ilgili docümanalardan
inceleyiniz. Ancak bu dosyada spesifik bir kullanıcı için
ve bir gruba dahil tüm kullanıcılar için limitler belirlenebilmektedir.
Tabii bu belirlemeler sistem reboot edilince
etki gösterir. Örneğin csd kullanıcısının proseslerinin açabileceği
dosya syısını 8192 yapabilmek için şu satırları dosyaya
eklemeliyiz:

csd      hard      nofile      8192
csd      soft      nofile      8192

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

/
*-----
-----
POSIX thread fonksiyonlarına pthread kütüphanesi denilmektedir. Tüm
thread fonksiyonları pthread_xxx biçiminde isimlendirilmiştir.
pthread_create fonksiyonu thread'i yaratır ve çalıştırır. pthread_create
fonksiyonundan çıkıldıktan sonra artık prosesin
iki bağımsız çizelgelenen akışı olacaktır. Thread fonksiyonlarının geri
dönüş değerleir ve parametreleri void * türünden
olmak zorundadır. Thread fonksiyonlarının büyük bölümü int geri dönüş
değerine sahiptir. Bu fonksiyonlar başarı durumunda 0 değerine,
başarısızlık durumunda errno değerinin kendisine geri dönerler. errno
değerini set etmezler.

```

```

-----*/

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

void *thread_proc(void *param);

int main(void)
{
    int result;
    pthread_t tid;
    int i;

    if ((result = pthread_create(&tid, NULL, thread_proc, NULL)) != 0) {
        fprintf(stderr, "pthread_create: %s\n", strerror(result));
        exit(EXIT_FAILURE);
    }

    for (int i = 0; i < 10; ++i) {
        printf("main thread\n");
        sleep(1);
    }

    return 0;
}

void *thread_proc(void *param)
{
    for (int i = 0; i < 10; ++i) {
        printf("other thread\n");
        sleep(1);
    }

    return NULL;
}

/
*-----
-----
    Thread fonksiyonları başarız olduğunda hata mesajını ekrana yazdırıp
    prosesi sonlandıran yardımcı bir exit_sys_thread
    fonksiyonu kullanacağız.
-----
-----*/

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

```

```

void exit_sys_thread(const char *msg, int err);
void *thread_proc(void *param);

int main(void)
{
    int result;
    pthread_t tid;
    int i;

    if ((result = pthread_create(&tid, NULL, thread_proc, NULL)) != 0)
        exit_sys_thread("pthread_create", result);

    for (int i = 0; i < 10; ++i) {
        printf("main thread\n");
        sleep(1);
    }

    return 0;
}

void exit_sys_thread(const char *msg, int err)
{
    fprintf(stderr, "%s: %s\n", msg, strerror(err));
    exit(EXIT_FAILURE);
}

void *thread_proc(void *param)
{
    for (int i = 0; i < 10; ++i) {
        printf("other thread\n");
        sleep(1);
    }

    return NULL;
}

/
*-----
-----
    Thread fonksiyonuna geçirilecek argüman stack'teki bir nesnenin adresi
    olmamalıdır. Argüman bir tamsayı ise void *'a dönüştürülerek,
    therad fonksiyonuna geçirilmeli oradan da yeniden tamsayı türüne
    dönüştürülmelidir. Thread fonksiyonuna geçirilecek adresin
    static ömürlü bir nesnenin adresi ya da heap'teki bir nesnenin adresi
    olması gerekir.
-----
-----*/

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

void exit_sys_thread(const char *msg, int err);

```

```

void *thread_proc(void *param);

int main(void)
{
    int result;
    pthread_t tid;
    int i;

    if ((result = pthread_create(&tid, NULL, thread_proc, "other thread")) != 0)
        exit_sys_thread("pthread_create", result);

    for (int i = 0; i < 10; ++i) {
        printf("main thread\n");
        sleep(1);
    }

    return 0;
}

void exit_sys_thread(const char *msg, int err)
{
    fprintf(stderr, "%s: %s\n", msg, strerror(err));
    exit(EXIT_FAILURE);
}

void *thread_proc(void *param)
{
    for (int i = 0; i < 10; ++i) {
        printf("%s\n", (char *)param);
        sleep(1);
    }

    return NULL;
}

```

```

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

```

Eğer thread'e birden fazla parametre geçirilmek isteniyorsa bir yapı bildirilmeli, bu yapı heap'te tahsis edilip, yapı nesnesinin adresi geçirilmelidir.

```

-----*/

```

```

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

```

```

void exit_sys_thread(const char *msg, int err);
void *thread_proc(void *param);

```

```

struct THREAD_PARAM {
    char name[64];
    int val;
};

int main(void)
{
    int result;
    pthread_t tid;
    int i;
    struct THREAD_PARAM *thread_param;

    if ((thread_param = (struct THREAD_PARAM *)malloc(sizeof(struct
        THREAD_PARAM))) == NULL) {
        fprintf(stderr, "cannot allocate memory!..\n");
        exit(EXIT_FAILURE);
    }
    strcpy(thread_param->name, "Oter thread");
    thread_param->val = 123;

    if ((result = pthread_create(&tid, NULL, thread_proc, thread_param)) !=
        0)
        exit_sys_thread("pthread_create", result);

    for (int i = 0; i < 10; ++i) {
        printf("main thread\n");
        sleep(1);
    }

    return 0;
}

void exit_sys_thread(const char *msg, int err)
{
    fprintf(stderr, "%s: %s\n", msg, strerror(err));
    exit(EXIT_FAILURE);
}

void *thread_proc(void *param)
{
    struct THREAD_PARAM *thread_param = (struct THREAD_PARAM *)param;

    for (int i = 0; i < 10; ++i) {
        printf("%s, %d\n", thread_param->name, thread_param->val);
        sleep(1);
    }

    free(thread_param);

    return NULL;
}

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

```

main fonksiyonu bittiğinde exit fonksiyonuyla proses sonlandırıldığı için diğer thread'ler de exit sırasında yok edilmektedir.

```
-----*/

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

void exit_sys_thread(const char *msg, int err);
void *thread_proc(void *param);

int main(void)
{
    int result;
    pthread_t tid;
    int i;

    if ((result = pthread_create(&tid, NULL, thread_proc, "other thread")) != 0)
        exit_sys_thread("pthread_create", result);

    for (int i = 0; i < 5; ++i) {
        printf("main thread: %d\n", i);
        sleep(1);
    }

    return 0;
}

void exit_sys_thread(const char *msg, int err)
{
    fprintf(stderr, "%s: %s\n", msg, strerror(err));
    exit(EXIT_FAILURE);
}

void *thread_proc(void *param)
{
    for (int i = 0; i < 10; ++i) {
        printf("%s: %d\n", (char *)param, i);
        sleep(1);
    }

    return NULL;
}

/
*-----

pthread_join fonksiyonu id'si ile verilen thread'in sonlanmasını bekler.
Onun exit kodunu alarak thread için ayrılan alanı yok eder.
Thread'ler için de -prosesler kadar önemli olmasa da- hortlaklıktan
bahasedebiliriz. Yani normal olarak detached olmayan thread'leri
```

pthread_join ile beklemeliyiz. pthread_join fonksiyonu proseslerde kullandığımız wait fonksiyonlarına işlev olarak benzetilmektedir. Ancak pthread_join fonksiyonu herhangi bir thread akışı tarafından prosesin herhangi bir thread'ini beklemek için kullanılabilir.

```
-----*/

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

void exit_sys_thread(const char *msg, int err);
void *thread_proc(void *param);

int main(void)
{
    int result;
    pthread_t tid;
    int i;

    if ((result = pthread_create(&tid, NULL, thread_proc, "other thread")) != 0)
        exit_sys_thread("pthread_create", result);

    for (int i = 0; i < 5; ++i) {
        printf("main thread: %d\n", i);
        sleep(1);
    }

    if ((result = pthread_join(tid, NULL)) != 0)
        exit_sys_thread("pthread_join", result);

    return 0;
}

void exit_sys_thread(const char *msg, int err)
{
    fprintf(stderr, "%s: %s\n", msg, strerror(err));
    exit(EXIT_FAILURE);
}

void *thread_proc(void *param)
{
    for (int i = 0; i < 10; ++i) {
        printf("%s: %d\n", (char *)param, i);
        sleep(1);
    }

    return NULL;
}
```



```
/
*-----
-----
thread'lerin exit kodlarının void * türünden olduğuna dikkat ediniz.
Eğer int bir exit kodu vermek istiyorsanız int değeri
void * türüne dönüştürmelisiniz. (Yani int değeri sanki bir adresmiş
gibi vermelisiniz.)
-----
-----*/
```

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

```
void exit_sys_thread(const char *msg, int err);
void *thread_proc(void *param);
```

```
int main(void)
{
    int result;
    pthread_t tid;
    int i;
    void *retval;

    if ((result = pthread_create(&tid, NULL, thread_proc, "other thread")) != 0)
        exit_sys_thread("pthread_create", result);

    for (int i = 0; i < 5; ++i) {
        printf("main thread: %d\n", i);
        sleep(1);
    }

    if ((result = pthread_join(tid, &retval)) != 0)
        exit_sys_thread("pthread_join", result);

    printf("Thread exited with %ld\n", (long)retval);

    return 0;
}
```

```
void exit_sys_thread(const char *msg, int err)
{
    fprintf(stderr, "%s: %s\n", msg, strerror(err));
    exit(EXIT_FAILURE);
}
```

```
void *thread_proc(void *param)
{
    for (int i = 0; i < 10; ++i) {
        printf("%s: %d\n", (char *)param, i);
        sleep(1);
    }
}
```

```

    return (void *)123;
}

/
*-----
-----
    Bir thread'in exit kodunu almak istemiyorsak onu "joinable" durumdan
    çıkartıp "detach" duruma sokmamız gerekir. Therad'i
    detach duruma sokmak için pthread_detach fonksiyonu kullanılmaktadır. Bu
    işlem thread yaratılırken thread attribute bilgisiyle de
    yapılabilir. Detach duruma sokulmuş bir thread sonlandığında
    işletim sistemi thread'in tüm kaynaklarını yok edecektir.
    Tabii detach bir thread pthread_join fonksiyonu ile beklenemez. Bu
    durumda pthread_join fonksiyonu başarısız olur.
-----
-----*/

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

void exit_sys_thread(const char *msg, int err);
void *thread_proc(void *param);

int main(void)
{
    int result;
    pthread_t tid;
    int i;
    void *retval;

    if ((result = pthread_create(&tid, NULL, thread_proc, "other thread")) != 0)
        exit_sys_thread("pthread_create", result);

    if ((result = pthread_detach(tid)) != 0) /* thread detach
        duruma sokuluyor */
        exit_sys_thread("pthread_detach", result);

    for (int i = 0; i < 5; ++i) {
        printf("main thread: %d\n", i);
        sleep(1);
    }

    if ((result = pthread_join(tid, &retval)) != 0) /* detach duruma
        sokulan thread pthread_join ile beklenemez */
        exit_sys_thread("pthread_join", result);

    printf("Thread exited with %ld\n", (long)retval);

    return 0;
}

```

```
void exit_sys_thread(const char *msg, int err)
{
    fprintf(stderr, "%s: %s\n", msg, strerror(err));
    exit(EXIT_FAILURE);
}
```

```
void *thread_proc(void *param)
{
    for (int i = 0; i < 10; ++i) {
        printf("%s: %d\n", (char *)param, i);
        sleep(1);
    }

    return (void *)123;
}
```

```
/
*-----
-----
Prosesin herhangi bir thread'i prosesin herhangi bir thread'ini
pthread_cancel fonksiyonuyla sonlandırabilir. Ancak thread'in
sonlanabilmesi için özel bazı POSIX donksiyonlarının içerisinde girmiş
olması gerekir. Bu POSIX fonksiyonlarına "thread cancellation points"
denilmektedir.
-----
-----*/
```

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

```
void exit_sys_thread(const char *msg, int err);
void *thread_proc(void *param);
```

```
int main(void)
{
    int result;
    pthread_t tid;
    int i;

    if ((result = pthread_create(&tid, NULL, thread_proc, "other thread")) != 0)
        exit_sys_thread("pthread_create", result);

    printf("Press ENTER to cancel...\n");
    getchar();

    if ((result = pthread_cancel(tid)) != 0)
        exit_sys_thread("pthread_cancel", result);

    if ((result = pthread_join(tid, NULL)) != 0)
        exit_sys_thread("pthread_join", result);
}
```

```

    printf("other thread cancelled!\n");

    for (int i = 0; i < 10; ++i) {
        printf("main thread: %d\n", i);
        sleep(1);
    }

    return 0;
}

void exit_sys_thread(const char *msg, int err)
{
    fprintf(stderr, "%s: %s\n", msg, strerror(err));
    exit(EXIT_FAILURE);
}

void *thread_proc(void *param)
{
    for (int i = 0; i < 10; ++i) {
        printf("%s: %d\n", (char *)param, i);
        sleep(1);
    }

    return NULL;
}

/
*-----
-----
    Eğer pthread_cancel uygulanan thread cancellation point olan bir POSIX
    fonksiyonuna girmiyorsa sonlandırma yapılamaz.
-----
-----*/

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

void exit_sys_thread(const char *msg, int err);
void *thread_proc(void *param);

int main(void)
{
    int result;
    pthread_t tid;

    if ((result = pthread_create(&tid, NULL, thread_proc, "other thread")) != 0)
        exit_sys_thread("pthread_create", result);

    printf("Press ENTER to cancel...\n");
    getchar();
}

```

```

    if ((result = pthread_cancel(tid)) != 0)
        exit_sys_thread("pthread_cancel", result);

    if ((result = pthread_join(tid, NULL)) != 0)
        exit_sys_thread("pthread_join", result);

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

    return 0;
}

void exit_sys_thread(const char *msg, int err)
{
    fprintf(stderr, "%s: %s\n", msg, strerror(err));
    exit(EXIT_FAILURE);
}

void *thread_proc(void *param)
{
    long i;

    for (i = 0; i < 1000000000; ++i)
        ;

    return NULL;
}

/
*-----
-----
    Eğer thread pthread_cancel ile sonlandırılmışsa pthread_join
    fonksiyonundan thread exit kodu olarak PTHREAD_CANCELED
    özel değeri elde edilir. (Bu değer void * türündendir.)
-----
-----*/

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

void exit_sys_thread(const char *msg, int err);
void *thread_proc(void *param);

int main(void)
{
    int result;
    pthread_t tid;
    void *retval;

    if ((result = pthread_create(&tid, NULL, thread_proc, "other thread")) != 0)
        exit_sys_thread("pthread_create", result);

```

```

printf("Press ENTER to cancel...\n");
getchar();

if ((result = pthread_cancel(tid)) != 0)
    exit_sys_thread("pthread_cancel", result);

if ((result = pthread_join(tid, &retval)) != 0)
    exit_sys_thread("pthread_join", result);

if (retval == PTHREAD_CANCELED)
    printf("other thread canceled!...\n");
else
    printf("other thread finished normally...\n");

return 0;
}

```

```

void exit_sys_thread(const char *msg, int err)
{
    fprintf(stderr, "%s: %s\n", msg, strerror(err));
    exit(EXIT_FAILURE);
}

```

```

void *thread_proc(void *param)
{
    int i;

    for (i = 0; i < 10; ++i) {
        printf("%s: %d\n", (char *)param, i);
        sleep(1);
    }

    return NULL;
}

```

```

/
*-----
-----
Aşağıdaki örnekte bir döngü içerisinde 10 thread yaratılmış ve bu 10
thread'in sonlanması beklenmiştir. Tüm thread'ler aynı
fonksiyondan farklı parametreler alarak çalışmaya başlamaktadır.
-----
-----*/

```

```

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

```

```

#define MAX_THREAD    10

```

```

void exit_sys_thread(const char *msg, int err);
void *thread_proc(void *param);

```

```

int main(void)
{
    int result;
    pthread_t tids[MAX_THREAD];
    int i;
    char *arg;

    for (i = 0; i < MAX_THREAD; ++i) {
        if ((arg = (char *)malloc(32)) == NULL) {
            fprintf(stderr, "cannot allocate memory!..\n");
            exit(EXIT_FAILURE);
        }
        sprintf(arg, "Thread-%d", (i + 1));
        if ((result = pthread_create(&tids[i], NULL, thread_proc, arg)) !=
            0)
            exit_sys_thread("pthread_create", result);
    }

    for (i = 0; i < MAX_THREAD; ++i)
        if ((result = pthread_join(tids[i], NULL)) != 0)
            exit_sys_thread("pthread_join", result);

    return 0;
}

void exit_sys_thread(const char *msg, int err)
{
    fprintf(stderr, "%s: %s\n", msg, strerror(err));
    exit(EXIT_FAILURE);
}

void *thread_proc(void *param)
{
    int i;

    for (i = 0; i < 10; ++i) {
        printf("%s: %d\n", (char *)param, i);
        sleep(1);
    }

    free(param);

    return NULL;
}

```

/

*-----

Therad'in çeşitli özellikleri (attributes) vardır. Thread özellikleri pthread_attr_t türüyle temsil edilmiştir. Bu tür pek çok sistemde bir yapı belirtir. Özellik set etmek için önce pthread_attr_init fonksiyonu ile nesnenin initialize edilmesi gerekir. Özellik nesnesinin elemanlarını set etmek için ise bir grup pthread_attr_setxxx isimli POSIX fonksiyonu bulundurulmuştur.

Örneğin thread'in detached mi yoksa joinable mı olacağını belirlemek için pthread_attr_setdetachstate fonksiyonu, thread'in stack uzunluğunu belirlemek için (Linux'ta default 8 MB) pthread_attr_setstacksize fonksiyonu kullanılır. En sonunda bu özellik nesnesinin pthread_destroy ile yok edilmesi gerekmektedir. (pthread_create fonksiyonu bu nesneyi kopya yoluyla kullanır. Yani pthread_destroy işlemi hemen pthread_create fonksiyonundan sonra da yapılabilir.

```
-----*/

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

void exit_sys_thread(const char *msg, int err);
void *thread_proc(void *param);

int main(void)
{
    int result;
    pthread_t tid;
    pthread_attr_t attr;

    if ((result = pthread_attr_init(&attr)) != 0)
        exit_sys_thread("pthread_attr_init", result);

    if ((result = pthread_attr_setstacksize(&attr, 65536)) != 0)
        exit_sys_thread("pthread_attr_setstacksize", result);

    if ((result = pthread_attr_setdetachstate(&attr,
        PTHREAD_CREATE_DETACHED)) != 0)
        exit_sys_thread("pthread_attr_setstacksize", result);

    if ((result = pthread_create(&tid, &attr, thread_proc, "other
        thread")) != 0)
        exit_sys_thread("pthread_create", result);

    pthread_attr_destroy(&attr);

    printf("Press ENTER to exit...\n");
    getchar();

    return 0;
}

void exit_sys_thread(const char *msg, int err)
{
    fprintf(stderr, "%s: %s\n", msg, strerror(err));
    exit(EXIT_FAILURE);
}
```



```

void *thread_proc(void *param)
{
    int i;

    for (i = 0; i < 10; ++i) {
        printf("%s: %d\n", (char *)param, i);
        sleep(1);
    }

    return NULL;
}

/
*-----
-----
pthread_self fonksiyonu hangi thread akışında çağrılırsa o thread'in
kendi thread id'si elde edilir.
-----
-----*/

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

void exit_sys_thread(const char *msg, int err);
void *thread_proc(void *param);

int main(void)
{
    int result;
    pthread_t tid;

    if ((result = pthread_create(&tid, NULL, thread_proc, "other thread")) != 0)
        exit_sys_thread("pthread_create", result);

    if ((result = pthread_join(tid, NULL)) != 0)
        exit_sys_thread("pthread_join", result);

    return 0;
}

void exit_sys_thread(const char *msg, int err)
{
    fprintf(stderr, "%s: %s\n", msg, strerror(err));
    exit(EXIT_FAILURE);
}

void *thread_proc(void *param)
{
    pthread_t tid;
    int i;

```

```

    tid = pthread_self();

    pthread_cancel(tid);          /* pthread_exit daha normal */

    for (i = 0; i < 10; ++i) {
        printf("%s: %d\n", (char *)param, i);
        sleep(1);
    }

    return NULL;
}

/
*-----
-----
pthread_attr_t nesnesinin içerisindeki elemanlar pthread_attr_getxxx
fonksiyonlarıyla alınabilirler. Örneğin pthread_attr_getstacksize
thread'in stack uzunluğunu almak için, pthread_attr_getdetachstate
thread'in joinable olup olmadığını almak için kullanılabilir.
Maalesef POSIX'te thread'in özellikleri elde edilememektedir. Ancak
bunun için Linux'ta pthread_getattr_np fonksiyonu bulunmaktadır.
Bu fonksiyon Linux'a özgüdür. (Fonksiyon isminin sonundaki _np
"nonportable" dan geliyor.)
-----
-----*/

#define _GNU_SOURCE

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

void exit_sys_thread(const char *msg, int err);
void *thread_proc(void *param);

int main(void)
{
    int result;
    pthread_t tid;

    if ((result = pthread_create(&tid, NULL, thread_proc, "other thread")) != 0)
        exit_sys_thread("pthread_create", result);

    if ((result = pthread_join(tid, NULL)) != 0)
        exit_sys_thread("pthread_join", result);

    return 0;
}

void exit_sys_thread(const char *msg, int err)
{

```

```

    fprintf(stderr, "%s: %s\n", msg, strerror(err));
    exit(EXIT_FAILURE);
}

```

```

void *thread_proc(void *param)
{

```

```

    pthread_t tid;
    pthread_attr_t attr;
    int result;
    size_t stacksize;

```

```

    tid = pthread_self();

```

```

    if ((result = pthread_getattr_np(tid, &attr)) != 0)
        exit_sys_thread("pthread_getattr_np", result);

```

```

    if ((result = pthread_attr_getstacksize(&attr, &stacksize)) != 0)
        exit_sys_thread("pthread_attr_getstacksize", result);

```

```

    printf("Stack size: %lu\n", (unsigned long)stacksize);

```

```

    if ((result = pthread_attr_destroy(&attr)) != 0)
        exit_sys_thread("pthread_attr_getstacksize", result);

```

```

    return NULL;
}

```

```

/
*-----
-----
Mutex nesnesi ile kritik kod oluşturmak için önce pthread_mutex_t
türünden global bir nesne tanımlanır. Daha sonra bu nesne
PTHREAD_MUTEX_INITIALIZER makrosuyla ilkdeğer verilerek statik biçimde
ya da pthread_mutex_init fonksiyonuyla dinamik biçimde
ilkdeğerlenir. Kritik kod da pthread_mutex_lock ve pthread_mutex_unlock
çağrılarını arasına yerleştirilir. İşlemler bittikten sonra
mutex nesnesi pthread_mutex_destroy fonksiyonuyla boşaltılmalıdır.
pthread_mutex_destroy ile mutex nesnesini yok etmek için mutex
nesnesinin locked durumda olmaması gerekir. Aksi halde "tanımsız
davranış (undefined behavior)" oluşacaktır.
-----
-----*/

```

```

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

```

```

void exit_sys_thread(const char *msg, int err);
void *thread_proc1(void *param);
void *thread_proc2(void *param);

```

```

pthread_mutex_t g_mutex;

```

```

int g_count;

int main(void)
{
    int result;
    pthread_t tid1, tid2;

    if ((result = pthread_mutex_init(&g_mutex, NULL)) != 0)
        exit_sys_thread("pthread_mutex_init", result);

    if ((result = pthread_create(&tid1, NULL, thread_proc1, NULL)) != 0)
        exit_sys_thread("pthread_create", result);

    if ((result = pthread_create(&tid2, NULL, thread_proc2, NULL)) != 0)
        exit_sys_thread("pthread_create", result);

    if ((result = pthread_join(tid1, NULL)) != 0)
        exit_sys_thread("pthread_join", result);

    if ((result = pthread_join(tid2, NULL)) != 0)
        exit_sys_thread("pthread_join", result);

    printf("g_count = %d\n", g_count);

    pthread_mutex_destroy(&g_mutex);

    return 0;
}

void exit_sys_thread(const char *msg, int err)
{
    fprintf(stderr, "%s: %s\n", msg, strerror(err));
    exit(EXIT_FAILURE);
}

void *thread_proc1(void *param)
{
    int i;
    int result;

    for (i = 0; i < 1000000; ++i) {
        if ((result = pthread_mutex_lock(&g_mutex)) != 0)
            exit_sys_thread("pthread_mutex_lock", result);

        ++g_count;          /* Critical section */

        if ((result = pthread_mutex_unlock(&g_mutex)) != 0)
            exit_sys_thread("pthread_mutex_lock", result);
    }

    return NULL;
}

void *thread_proc2(void *param)
{

```

```

int i;
int result;

for (i = 0; i < 1000000; ++i) {
    if ((result = pthread_mutex_lock(&g_mutex)) != 0)
        exit_sys_thread("pthread_mutex_lock", result);

    ++g_count;          /* Critical section */

    if ((result = pthread_mutex_unlock(&g_mutex)) != 0)
        exit_sys_thread("pthread_mutex_lock", result);
}

return NULL;
}

/
*-----
-----
Mutex nesnesi ile kritik kod oluşturma (mutex nesnesinin sahipliği
pthread_mutex_unlock ile yalnızca onu almış thread tarafından
bırakılabilir.)
-----
-----*/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include <unistd.h>
#include <pthread.h>

void exit_sys_thread(const char *msg, int err);
void *thread_proc1(void *param);
void *thread_proc2(void *param);
void do_something(const char *name);

pthread_mutex_t g_mutex = PTHREAD_MUTEX_INITIALIZER;

int main(void)
{
    int result;
    pthread_t tid1, tid2;

    srand(time(NULL));

    if ((result = pthread_create(&tid1, NULL, thread_proc1, NULL)) != 0)
        exit_sys_thread("pthread_create", result);

    if ((result = pthread_create(&tid2, NULL, thread_proc2, NULL)) != 0)
        exit_sys_thread("pthread_create", result);

    if ((result = pthread_join(tid1, NULL)) != 0)
        exit_sys_thread("pthread_join", result);
}

```

```

    if ((result = pthread_join(tid2, NULL)) != 0)
        exit_sys_thread("pthread_join", result);

    printf("g_count = %d\n", g_count);

    pthread_mutex_destroy(&g_mutex);

    return 0;
}

void exit_sys_thread(const char *msg, int err)
{
    fprintf(stderr, "%s: %s\n", msg, strerror(err));
    exit(EXIT_FAILURE);
}

void *thread_proc1(void *param)
{
    int i;

    for (i = 0; i < 10; ++i) {
        do_something("thread1");
        usleep(rand() % 30000);
    }

    return NULL;
}

void *thread_proc2(void *param)
{
    int i;

    for (i = 0; i < 10; ++i) {
        do_something("thread2");
        usleep(rand() % 30000);
    }

    return NULL;
}

void do_something(const char *name)
{
    int result;

    if ((result = pthread_mutex_lock(&g_mutex)) != 0)
        exit_sys_thread("pthread_mutex_lock", result);

    printf("%s- Step 1\n", name);
    usleep(rand() % 30000);
    printf("%s- Step 2\n", name);
    usleep(rand() % 30000);
    printf("%s- Step 3\n", name);
    usleep(rand() % 30000);
    printf("%s- Step 4\n", name);
    usleep(rand() % 30000);
}

```

```

printf("%s- Step 5\n", name);
printf("-----\n");

if ((result = pthread_mutex_unlock(&g_mutex)) != 0)
    exit_sys_thread("pthread_mutex_unlock", result);
}

/
*-----
-----
pthread_mutex_timedlock fonksiyonu en kötü olasılıkla kilit açılmamışsa
zaman aşımından dolayı blokeyi kaldırabilmektedir.
Ancak fonksiton zaman aşımı olarak epoch'tan itibaren gerçek zamanı
istemektedir. Yani örneğin biz 5 saniyelik bir zaman aşımını
belirteceksek önce epoch'tan itibaren (01/01/1970) şimdiye kadar geçen
saniye sayısını bulmamız sonra buna 5 saniye eklememiz gerekir.
time fonksiyonun epoch'tan geçen saniye sayısını bize verdiğini
anımsayınız. Eğer pthread_mutex_timedlock fonksiyonu zaman aşımından
dolayı sonlanmışsa error kodu ETIMEDOUT biçiminde elde edilmektedir.
Aşağıdaki programda birinci thread mutex'i kilitlemiş ve 20
saniye beklemiştir. İkinci thread ise 5 saniye kadar mutex kilidi
açılmamışsa blokeyi sonlandırmaktadır.

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

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include <errno.h>
#include <unistd.h>
#include <pthread.h>
#include <sys/time.h>

void exit_sys_thread(const char *msg, int err);
void *thread_proc1(void *param);
void *thread_proc2(void *param);

pthread_mutex_t g_mutex;

int main(void)
{
    int result;
    pthread_t tid1, tid2;

    if ((result = pthread_mutex_init(&g_mutex, NULL)) != 0)
        exit_sys_thread("pthread_mutex_init", result);

    if ((result = pthread_create(&tid1, NULL, thread_proc1, NULL)) != 0)
        exit_sys_thread("pthread_create", result);

    sleep(1);

    if ((result = pthread_create(&tid2, NULL, thread_proc2, NULL)) != 0)

```

```

        exit_sys_thread("pthread_create", result);

    if ((result = pthread_join(tid1, NULL)) != 0)
        exit_sys_thread("pthread_join", result);

    if ((result = pthread_join(tid2, NULL)) != 0)
        exit_sys_thread("pthread_join", result);

    pthread_mutex_destroy(&g_mutex);

    return 0;
}

void exit_sys_thread(const char *msg, int err)
{
    fprintf(stderr, "%s: %s\n", msg, strerror(err));
    exit(EXIT_FAILURE);
}

void *thread_proc1(void *param)
{
    int result;

    if ((result = pthread_mutex_lock(&g_mutex)) != 0)
        exit_sys_thread("pthread_mutex_lock", result);

    sleep(20);

    if ((result = pthread_mutex_unlock(&g_mutex)) != 0)
        exit_sys_thread("pthread_mutex_unlock", result);

    printf("First thread unlocked mutex!\n");

    return NULL;
}

void *thread_proc2(void *param)
{
    struct timespec ts;
    int result;

    localtime(&ts.tv_sec);

    ts.tv_sec += 5;
    ts.tv_nsec = 0;

    if ((result = pthread_mutex_timedlock(&g_mutex, &ts)) != 0)
        if (result == ETIMEDOUT) {
            printf("Time out! Do something else!\n");
            return NULL;
        }

    /* some code */

    if ((result = pthread_mutex_unlock(&g_mutex)) != 0)

```



```

        exit_sys_thread("pthread_mutex_unlock", result);

    return NULL;
}

/
*-----
-----
    Mutex yaratıldığında recursive değildir. Yani bir thread mutex
    nesnesinin sahipliğini aldığı anda yeniden onu almaya çalışırsa
    kendi kendini kilitletler. Aşağıda örnekte bu biçimde bir deadlock
    (kilitlenme) oluşacaktır.
-----
-----*/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <pthread.h>

void exit_sys_thread(const char *msg, int err);
void *thread_proc1(void *param);
void *thread_proc2(void *param);
void foo(void);
void bar(void);

pthread_mutex_t g_mutex;

int main(void)
{
    int result;
    pthread_t tid1, tid2;

    if ((result = pthread_mutex_init(&g_mutex, NULL)) != 0)
        exit_sys_thread("pthread_mutex_init", result);

    if ((result = pthread_create(&tid1, NULL, thread_proc1, NULL)) != 0)
        exit_sys_thread("pthread_create", result);

    if ((result = pthread_join(tid1, NULL)) != 0)
        exit_sys_thread("pthread_join", result);

    pthread_mutex_destroy(&g_mutex);

    return 0;
}

void exit_sys_thread(const char *msg, int err)
{
    fprintf(stderr, "%s: %s\n", msg, strerror(err));
    exit(EXIT_FAILURE);
}

void *thread_proc1(void *param)

```

```

{
    foo();

    return NULL;
}

void foo(void)
{
    int result;

    if ((result = pthread_mutex_lock(&g_mutex)) != 0)
        exit_sys_thread("pthread_mutex_lock", result);

    printf("thread locked mutex first time!..\n");
    bar();

    if ((result = pthread_mutex_unlock(&g_mutex)) != 0)
        exit_sys_thread("pthread_mutex_unlock", result);
}

void bar(void)
{
    int result;

    if ((result = pthread_mutex_lock(&g_mutex)) != 0)
        exit_sys_thread("pthread_mutex_lock", result);

    printf("thread locked mutex second time!..\n");

    if ((result = pthread_mutex_unlock(&g_mutex)) != 0)
        exit_sys_thread("pthread_mutex_unlock", result);
}

```

/

*-----

Mutex özelliklerini set etmek için önce pthread_mutexattr_t türünden bir nesne tanımlanır. Sonra bu nesne pthread_mutexattr_init fonksiyonuyla ilkdeğerlenir. Daha sonra da pthread_mutexattr_setxxx fonksiyonlarıyla özellikler set edilir. Sonra da bu özellik nesnesi pthread_mutex_init fonksiyonuna parametre olarak geçirilir. Bu işlemden sonra artık bu özellik nesnesi pthread_mutexattr_destroy fonksiyonuyla yok edilebilir. pthread_mutexattr_settype mutex nesnesinin türünü set etmek için kullanılmaktadır. Burada PTHREAD_MUTEX_RECURSIVE tür olarak bir thread birden fazla kez aynı mutex nesnesinin sahipliğini almaya çalıştığında deadlock oluşmaması için kullanılmaktadır. Tabii recursive mutex'lerde ne kadar lock yapılmışsa nesneyi bırakmak için o sayıda unlock yapılması gerekmektedir. Burada type olarak PTHREAD_MUTEX_ERRORCHECK girilirse mutex kendi kendini kilitlemez error koduyla geri döner. PTHREAD_MUTEX_NORMAL kendi kendini kilitleyen mutex'tir.,

Mutex default durumda tür olarak PTHREAD_MUTEX_DEFAULT türüne sahiptir. POSIX standartları PTHREAD_MUTEX_DEFAULT türünün diğer üç türden bir tanesi olarak alınabileceğini söylemektedir. Linux sistemlerinde PTHREAD_MUTEX_DEFAULT türü PTHREAD_MUTEX_NORMAL biçiminde alınmıştır.

Dolayısıyla Linux sistemlerinde mutex kendini kilitler durumdadır.

```
-----*/

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

void exit_sys_thread(const char *msg, int err);
void *thread_proc1(void *param);
void foo(void);
void bar(void);

pthread_mutex_t g_mutex;

int main(void)
{
    int result;
    pthread_t tid1;
    pthread_mutexattr_t mattr;

    pthread_mutexattr_init(&mattr);

    if ((result = pthread_mutexattr_settype(&mattr,
        PTHREAD_MUTEX_RECURSIVE)) != 0)
        exit_sys_thread("pthread_mutexattr_settype", result);

    if ((result = pthread_mutex_init(&g_mutex, &mattr)) != 0)
        exit_sys_thread("pthread_mutex_init", result);

    if ((result = pthread_mutexattr_destroy(&mattr)) != 0)
        exit_sys_thread("pthread_mutexattr_destroy", result);

    if ((result = pthread_create(&tid1, NULL, thread_proc1, NULL)) != 0)
        exit_sys_thread("pthread_create", result);

    if ((result = pthread_join(tid1, NULL)) != 0)
        exit_sys_thread("pthread_join", result);

    pthread_mutex_destroy(&g_mutex);

    return 0;
}

void exit_sys_thread(const char *msg, int err)
{
    fprintf(stderr, "%s: %s\n", msg, strerror(err));
    exit(EXIT_FAILURE);
}

void *thread_proc1(void *param)
{
    foo();
```

```

    return NULL;
}

void foo(void)
{
    int result;

    if ((result = pthread_mutex_lock(&g_mutex)) != 0)
        exit_sys_thread("pthread_mutex_lock", result);

    printf("thread locked mutex first time!..\n");
    bar();

    if ((result = pthread_mutex_unlock(&g_mutex)) != 0)
        exit_sys_thread("pthread_mutex_unlock", result);
}

```

```

void bar(void)
{
    int result;

    if ((result = pthread_mutex_lock(&g_mutex)) != 0)
        exit_sys_thread("pthread_mutex_lock", result);

    printf("thread locked mutex second time!..\n");

    if ((result = pthread_mutex_unlock(&g_mutex)) != 0)
        exit_sys_thread("pthread_mutex_unlock", result);
}

```

```

/
*-----
-----
Mutex nesneleri farklı proseslerin thread'leri arasında da
kullanılabilir. Bunun için mutex nesnesinin (yani pthread_mutex_t
nesnesinin)
paylaşılan bir bellek alanında yaratılmış olması gerekir. Ancak nesnenin
paylaşılan bellek alanında yaratılmış olması yetmemektedir.
Ayrıca mutex'in "shared" moda sokulması gerekir. Bu işlem de
pthread_mutexattr_setpshared fonksiyonuyla yapılmaktadır. Bu
fonksiyonda
parametre olarak PTHREAD_PROCESS_SHARED geçilmelidir.
-----
-----*/

```

```

/* proc1.c */

```

```

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

```

```

#define SHM_PATH          "/shaed_memory_mutex_test"

void exit_sys(const char *msg);
void exit_sys_thread(const char *msg, int err);

int main(void)
{
    int fdshm;
    void *addr;
    pthread_mutex_t *mutex;
    pthread_mutexattr_t mattr;
    int result;

    if ((fdshm = shm_open(SHM_PATH, O_CREAT|O_RDWR, S_IRUSR|S_IWUSR|S_IRGRP|
        S_IROTH)) == -1)
        exit_sys("shm_open");

    if (ftruncate(fdshm, 4096) == -1)
        exit_sys("ftruncate");

    if ((addr = mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_SHARED, fdshm,
        0)) == MAP_FAILED)
        exit_sys("mmap");

    mutex = (pthread_mutex_t *)addr;

    pthread_mutexattr_init(&mattr);
    if ((result = pthread_mutexattr_setpshared(&mattr,
        PTHREAD_PROCESS_SHARED)) != 0)
        exit_sys_thread("pthread_mutexattr_setpshared", result);

    if ((result = pthread_mutex_init(mutex, &mattr)) != 0)
        exit_sys_thread("pthread_mutex_init", result);

    if ((result = pthread_mutex_lock(mutex)) != 0)
        exit_sys_thread("pthread_mutex_lock", result);

    printf("Press ENTER to release mutex...\n");
    getchar();

    if ((result = pthread_mutex_unlock(mutex)) != 0)
        exit_sys_thread("pthread_mutex_unlock", result);

    sleep(2);

    pthread_mutexattr_destroy(&mattr);

    munmap(addr, 4096);

    close(fdshm);

    if (shm_unlink(SHM_PATH) == -1)
        exit_sys("shm_unlink");
}

```

```

    return 0;
}

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

    exit(EXIT_FAILURE);
}

void exit_sys_thread(const char *msg, int err)
{
    fprintf(stderr, "%s: %s\n", msg, strerror(err));
    exit(EXIT_FAILURE);
}

/* proc2.c */

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

#define SHM_PATH          "/shaed_memory_mutex_test"

void exit_sys(const char *msg);
void exit_sys_thread(const char *msg, int err);

int main(void)
{
    int fdshm;
    void *addr;
    pthread_mutex_t *mutex;
    int result;

    if ((fdshm = shm_open(SHM_PATH, O_CREAT|O_RDWR, S_IRUSR|S_IWUSR|S_IRGRP|
        S_IROTH)) == -1)
        exit_sys("shm_open");

    if ((addr = mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_SHARED, fdshm,
        0)) == MAP_FAILED)
        exit_sys("mmap");

    mutex = (pthread_mutex_t *)addr;

    printf("waiting for unlock...\n");
    if ((result = pthread_mutex_lock(mutex)) != 0)
        exit_sys_thread("pthread_mutex_lock", result);

    printf("got mutex ownership!..\n");

    if ((result = pthread_mutex_unlock(mutex)) != 0)

```

```

        exit_sys_thread("pthread_mutex_unlock", result);

munmap(addr, 4096);

close(fdshm);

return 0;
}

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

    exit(EXIT_FAILURE);
}

void exit_sys_thread(const char *msg, int err)
{
    fprintf(stderr, "%s: %s\n", msg, strerror(err));
    exit(EXIT_FAILURE);
}

/
*-----
-----
    Bir thread bir mutex'in sahipliğini aldıktan sonra sonlanırsa be olur?
    İşte default durumda pthread_mutex_lock fonksiyonunda
    bekleyen thread'ler beklemeye devam ederler. Yeni thread'ler bu
    fonksiyona girerlerse onlar da beklerler. Zaten bu durumdaki
    mutex'in açılması söz konusu değildir. Bu durumdaki mutex'ler "abandoned
    mutexes" denilmektedir. Fakat mutex "robust" denilen
    bir mutex özelliği de vardır. pthread_mutexattr_setrobust fonksiyonuyla
    bu durum değiştirilebilir. Bu fonksiyonda PTHREAD_MUTEX_ROBUST
    parametresi girildiğinde artık mutex "robust" olur. Robust mutex'lere
    sahip thread'ler sonlansa bile deadlock oluşmaz.
    pthread_mutex_lock fonksiyonları EOWNERDEAD hata koduyla geri dönerler.
    Bu durumda mutex nesnesi unlcok yapılırsa sorun oluşmaz. Ancak
    işlevini yerine getiremez durumdadır. Bu tür duurmdaki mutex'lere
    bozulmuş (inconsistent) mutex denilmektedir. Bozulmuş mutex'leri eski
    haline getirmek
    için pthread_mutex_consistent fonksiyonu çağrılmalıdır. Aşağıdaki bu
    biçimde bir deadlock örneği verilmiştir.
    -----
    -----*/

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

void exit_sys_thread(const char *msg, int err);
void *thread_proc1(void *param);
void *thread_proc2(void *param);

```

```

pthread_mutex_t g_mutex;

int main(void)
{
    int result;
    pthread_t tid1, tid2;

    if ((result = pthread_mutex_init(&g_mutex, NULL)) != 0)
        exit_sys_thread("pthread_mutex_init", result);

    if ((result = pthread_create(&tid1, NULL, thread_proc1, NULL)) != 0)
        exit_sys_thread("pthread_create", result);

    sleep(2);

    if ((result = pthread_create(&tid2, NULL, thread_proc2, NULL)) != 0)
        exit_sys_thread("pthread_create", result);

    if ((result = pthread_join(tid1, NULL)) != 0)
        exit_sys_thread("pthread_join", result);

    if ((result = pthread_join(tid2, NULL)) != 0)
        exit_sys_thread("pthread_join", result);

    pthread_mutex_destroy(&g_mutex);

    return 0;
}

void exit_sys_thread(const char *msg, int err)
{
    fprintf(stderr, "%s: %s\n", msg, strerror(err));
    exit(EXIT_FAILURE);
}

void *thread_proc1(void *param)
{
    int result;

    if ((result = pthread_mutex_lock(&g_mutex)) != 0)
        exit_sys_thread("pthread_mutex_lock", result);

    printf("thread1 gets ownership and terminate!..\n");

    pthread_exit(NULL);

    if ((result = pthread_mutex_unlock(&g_mutex)) != 0)
        exit_sys_thread("pthread_mutex_unlock", result);

    return NULL;
}

void *thread_proc2(void *param)
{

```



```

int result;

printf("second thread waits abandoned mutex (deadlock)...\n");
if ((result = pthread_mutex_lock(&g_mutex)) != 0)
    exit_sys_thread("pthread_mutex_lock", result);

/* .... */

if ((result = pthread_mutex_unlock(&g_mutex)) != 0)
    exit_sys_thread("pthread_mutex_unlock", result);

return NULL;
}

/
*-----
-----
Abandoned mutex'in yeniden kullanılabilir (consistent) duruma sokulması
-----
-----*/

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

void exit_sys_thread(const char *msg, int err);
void *thread_proc1(void *param);
void *thread_proc2(void *param);

pthread_mutex_t g_mutex;

int main(void)
{
    int result;
    pthread_t tid1, tid2;
    pthread_mutexattr_t mattr;

    pthread_mutexattr_init(&mattr);

    if ((result = pthread_mutexattr_setrobust(&mattr, PTHREAD_MUTEX_ROBUST))
        != 0)
        exit_sys_thread("pthread_mutexattr_settype", result);

    if ((result = pthread_mutex_init(&g_mutex, &mattr)) != 0)
        exit_sys_thread("pthread_mutex_init", result);

    if ((result = pthread_mutexattr_destroy(&mattr)) != 0)
        exit_sys_thread("pthread_mutexattr_destroy", result);

    if ((result = pthread_create(&tid1, NULL, thread_proc1, NULL)) != 0)
        exit_sys_thread("pthread_create", result);

```

```

sleep(2);

if ((result = pthread_create(&tid2, NULL, thread_proc2, NULL)) != 0)
    exit_sys_thread("pthread_create", result);

if ((result = pthread_join(tid1, NULL)) != 0)
    exit_sys_thread("pthread_join", result);

if ((result = pthread_join(tid2, NULL)) != 0)
    exit_sys_thread("pthread_join", result);

pthread_mutex_destroy(&g_mutex);

return 0;
}

void exit_sys_thread(const char *msg, int err)
{
    fprintf(stderr, "%s: %s\n", msg, strerror(err));
    exit(EXIT_FAILURE);
}

void *thread_proc1(void *param)
{
    int result;

    if ((result = pthread_mutex_lock(&g_mutex)) != 0)
        exit_sys_thread("pthread_mutex_lock", result);

    printf("thread1 gets ownership and terminate!..\n");

    pthread_exit(NULL);

    if ((result = pthread_mutex_unlock(&g_mutex)) != 0)
        exit_sys_thread("pthread_mutex_unlock", result);

    return NULL;
}

void *thread_proc2(void *param)
{
    int result;

    printf("second thread waits abandoned mutex...\n");
    if ((result = pthread_mutex_lock(&g_mutex)) != 0 && result ==
        EOWNERDEAD)
        if ((result = pthread_mutex_consistent(&g_mutex)) != 0)
            exit_sys_thread("pthread_mutex_consistent", result);

    /* ... */

    if ((result = pthread_mutex_unlock(&g_mutex)) != 0)
        exit_sys_thread("pthread_mutex_unlock", result);
}

```

```
printf("Ok\n");

return NULL;
}
```

/

*-----

Durum değişkenleri (condition variables) mutex nesneleriyle birlikte kullanılan senkronizasyon nesneleridir. Genellikle bir koşul sağlanana kadar bir thread'i bloke ederek bekletmek için kullanılırlar. Durum değişkenlerinesneleri pthread_cond_t türüyle temsil edilmektedir. Bu nesne pthread_cond_init fonksiyonuyla dinamik olarak ya da PTHREAD_COND_INITIALIZER makrosuyla statik olarak ilkdeğerlenir. Bekleme işlemi pthread_cond_wait fonksiyonuyla durum değişken nesnesi ve mutex nesnesi verilerek sağlanmaktadır. Bekleme sırasında ilgili mutex kilitlenmiş olmalıdır. Bu nedenle bu fonksiyon pthread_mutex_lock ve pthread_mutex_unlock fonksiyonlarının arasında çağrılır. pthread_cond_wait fonksiyonu çağrıldığında thread uykuya dalmadan önce atomik bir biçimde mutex nesnesinin kilidini açar. Thread'i uykudan uyandırmak için başka bir thread'in pthread_cond_signal fonksiyonunu çağırması gerekir. Bu durumda bekleyen thread atomik bir biçimde mutex'i yeniden kilitleyerek uyanır. Ancak maalesef pthread_cond_wait fonksiyonundan uyanmanın tek nedeni pthread_cond_wait uygulanması değildir. Sistemler bazen gereksiz uyandırmalar yapabilmektedir. Bu nedenle programcının uyanma sonrasında global bir durum değişkenine bakarak uyandırmanın gerçekliğinden emin olması gerekir. Bu tipik olarak bir while döngüsüyle sağlanmaktadır:

```
while (global koşul değişkeni set edilmediği sürece)
    pthread?cond_wait(...);
```

Bu durumda tipik bekleme kalıbı şöyle olmalıdır:

```
pthread_mutex_lock(&g_mutex);
```

```
while (g_flag == 0)
    pthread_cond_wait(&g_cond, &g_mutex);
```

```
pthread_mutex_unlock(&g_mutex);
```

pthread_cond_wait fonksiyonundan thread'in gereksiz uyandırılmasına İngilizce "spurious wakeup" denilmektedir.

Diğer thread uyuyan thread'i uyandırmak için pthread_cond_signal fonksiyonunu çağırmadan önce global flag değişkenini uygun biçimde set etmelidir.

Tabii bu işlem (zounlu olmasa da) yime mutex kontrolü içerisinde yapılmalıdır.

```
pthread_mutex_lock(&g_mutex);
g_flag = 1;
pthread_mutex_unlock(&g_mutex);
```

```
pthread_cond_signal(&g_cond);
```

pthread_cond_signal fonksiyonu kritik kod içerisinde de çağrılabilir. Bu durumda uyandırma yapılır ancak uyandırılan thread mutex kilitli olduğu için henüz tam uyanamaz. Bu durumda signal uygulayan thread mutex'i bıraktınca diğer thread tam olarak uyanır. Durum değişken nesnesi işlem bitince pthread_cond_destroy fonksiyonuyla geri bırakılmalıdır.

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

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

```
void exit_sys_thread(const char *msg, int err);
void *thread_proc1(void *param);
void *thread_proc2(void *param);
```

```
pthread_mutex_t g_mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t g_cond = PTHREAD_COND_INITIALIZER;
int g_condition = 0;
```

```
int main(void)
{
    int result;
    pthread_t tid1, tid2;

    if ((result = pthread_create(&tid1, NULL, thread_proc1, NULL)) != 0)
        exit_sys_thread("pthread_create", result);

    if ((result = pthread_create(&tid2, NULL, thread_proc2, NULL)) != 0)
        exit_sys_thread("pthread_create", result);

    if ((result = pthread_join(tid1, NULL)) != 0)
        exit_sys_thread("pthread_join", result);

    if ((result = pthread_join(tid2, NULL)) != 0)
        exit_sys_thread("pthread_join", result);

    pthread_cond_destroy(&g_cond);
    pthread_mutex_destroy(&g_mutex);

    return 0;
}
```

```
void exit_sys_thread(const char *msg, int err)
{
    fprintf(stderr, "%s: %s\n", msg, strerror(err));
    exit(EXIT_FAILURE);
}
```

```

void *thread_proc1(void *param)
{
    int result;

    printf("thread1 is running...\n");
    sleep(5);

    if ((result = pthread_mutex_lock(&g_mutex)) != 0)
        exit_sys_thread("pthread_mutex_lock", result);

    g_condition = 1;

    if ((result = pthread_mutex_unlock(&g_mutex)) != 0)
        exit_sys_thread("pthread_mutex_unlock", result);

    if ((result = pthread_cond_signal(&g_cond)) != 0)
        exit_sys_thread("pthread_cond_signal", result);

    sleep(5);

    return NULL;
}

void *thread_proc2(void *param)
{
    int result;

    printf("thread2 waiting for the condition...\n");

    if ((result = pthread_mutex_lock(&g_mutex)) != 0)
        exit_sys_thread("pthread_mutex_lock", result);

    while (g_condition == 0)
        if ((result = pthread_cond_wait(&g_cond, &g_mutex)) != 0)
            exit_sys_thread("pthread_cond_wait", result);

    if ((result = pthread_mutex_unlock(&g_mutex)) != 0)
        exit_sys_thread("pthread_mutex_unlock", result);

    printf("condition granted...\n");

    sleep(5);

    return NULL;
}

/
*-----
-----
Eğer pthread_cond_wait fonksiyonunda aynı koşul değişken nesnesi ve aynı
mutex eşliğinde uykuya dalmış birden fazla bekleyen
thread varsa pthread_cond_signal bunlardan yalnızca bir tanesini
uyandırmak ister. Diğerleri pthread_cond_wait fonksiyonunda
uyumaya devam ederler. Ancak maalesef bazı gerçekleştirimler
pthread_cond_signal uygulandığında tek bir thread'i uyandırmayı

```

başaramayıp bunların hepsini uyandırabilmektedir. Mademki pthread_cond_signal fonksiyonunun anlamı uyuyanlardan yalnızca bir tanesini uyandırmaktır. Bu durumda programcı spurious wakeup için kendi önlemini almalıdır. Bu önlem tipik olarak ilgili global flag'in yeniden eski durumuna çekilmesi ile alınabilir.

```
-----*/

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

void exit_sys_thread(const char *msg, int err);
void *thread_proc1(void *param);
void *thread_proc2(void *param);
void *thread_proc3(void *param);

pthread_mutex_t g_mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t g_cond = PTHREAD_COND_INITIALIZER;
int g_condition = 0;

int main(void)
{
    int result;
    pthread_t tid1, tid2, tid3;

    if ((result = pthread_create(&tid1, NULL, thread_proc1, NULL)) != 0)
        exit_sys_thread("pthread_create", result);

    if ((result = pthread_create(&tid2, NULL, thread_proc2, NULL)) != 0)
        exit_sys_thread("pthread_create", result);

    if ((result = pthread_create(&tid3, NULL, thread_proc3, NULL)) != 0)
        exit_sys_thread("pthread_create", result);

    if ((result = pthread_join(tid1, NULL)) != 0)
        exit_sys_thread("pthread_join", result);

    if ((result = pthread_join(tid2, NULL)) != 0)
        exit_sys_thread("pthread_join", result);

    if ((result = pthread_join(tid3, NULL)) != 0)
        exit_sys_thread("pthread_join", result);

    pthread_cond_destroy(&g_cond);
    pthread_mutex_destroy(&g_mutex);

    return 0;
}

void exit_sys_thread(const char *msg, int err)
{

```

```

    fprintf(stderr, "%s: %s\n", msg, strerror(err));
    exit(EXIT_FAILURE);
}

void *thread_proc1(void *param)
{
    int result;

    printf("thread1 is running...\n");

    printf("press ENTER to signal!...\n");
    getchar();

    if ((result = pthread_mutex_lock(&g_mutex)) != 0)
        exit_sys_thread("pthread_mutex_lock", result);

    g_condition = 1;

    if ((result = pthread_mutex_unlock(&g_mutex)) != 0)
        exit_sys_thread("pthread_mutex_unlock", result);

    if ((result = pthread_cond_signal(&g_cond)) != 0)
        exit_sys_thread("pthread_cond_signal", result);

    sleep(5);

    if ((result = pthread_mutex_lock(&g_mutex)) != 0)
        exit_sys_thread("pthread_mutex_lock", result);

    g_condition = 1;

    if ((result = pthread_mutex_unlock(&g_mutex)) != 0)
        exit_sys_thread("pthread_mutex_unlock", result);

    if ((result = pthread_cond_signal(&g_cond)) != 0)
        exit_sys_thread("pthread_cond_signal", result);

    return NULL;
}

void *thread_proc2(void *param)
{
    int result;

    printf("thread2 waiting for the condition...\n");

    if ((result = pthread_mutex_lock(&g_mutex)) != 0)
        exit_sys_thread("pthread_mutex_lock", result);

    while (g_condition == 0)
        if ((result = pthread_cond_wait(&g_cond, &g_mutex)) != 0)
            exit_sys_thread("pthread_cond_wait", result);

    g_condition = 0;

```

```

    if ((result = pthread_mutex_unlock(&g_mutex)) != 0)
        exit_sys_thread("pthread_mutex_unlock", result);

    printf("thread2 condition granted...\n");

    return NULL;
}

void *thread_proc3(void *param)
{
    int result;

    printf("thread3 waiting for the condition...\n");

    if ((result = pthread_mutex_lock(&g_mutex)) != 0)
        exit_sys_thread("pthread_mutex_lock", result);

    while (g_condition == 0)
        if ((result = pthread_cond_wait(&g_cond, &g_mutex)) != 0)
            exit_sys_thread("pthread_cond_wait", result);

    g_condition = 0;

    if ((result = pthread_mutex_unlock(&g_mutex)) != 0)
        exit_sys_thread("pthread_mutex_unlock", result);

    printf("thread3 condition granted...\n");

    return NULL;
}

/
*-----
-----
Eğer programcı pthread_cond_wait nedeniyle uykuya dalmış tüm thread'leri
uyandırmak istiyorsa bu durumda pthread_cond_broadcast
fonksiyonu kullanılmalıdır. Bu fonksiyon tüm thread'leri uykudan
uyandırır. Ancak bunlardan bir tanesi mutex'i kilitler
fakat neyseki o thread mutex'in kilidini açtığında sıradaki mutex'i
kilitleyerek çıkışı yapar. Yani bir deyişle broadcast işlemi
tüm bekleyen thread'leri uyandırmakla birlikte bu uyandırılmış
thread'ler mutex'i de kilitleyene kadar yine uykuda beklerler.
-----
-----*/

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

void exit_sys_thread(const char *msg, int err);
void *thread_proc1(void *param);
void *thread_proc2(void *param);
void *thread_proc3(void *param);

```



```

pthread_mutex_t g_mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t g_cond = PTHREAD_COND_INITIALIZER;
int g_condition = 0;

int main(void)
{
    int result;
    pthread_t tid1, tid2, tid3;

    if ((result = pthread_create(&tid1, NULL, thread_proc1, NULL)) != 0)
        exit_sys_thread("pthread_create", result);

    if ((result = pthread_create(&tid2, NULL, thread_proc2, NULL)) != 0)
        exit_sys_thread("pthread_create", result);

    if ((result = pthread_create(&tid3, NULL, thread_proc3, NULL)) != 0)
        exit_sys_thread("pthread_create", result);

    if ((result = pthread_join(tid1, NULL)) != 0)
        exit_sys_thread("pthread_join", result);

    if ((result = pthread_join(tid2, NULL)) != 0)
        exit_sys_thread("pthread_join", result);

    if ((result = pthread_join(tid3, NULL)) != 0)
        exit_sys_thread("pthread_join", result);

    pthread_cond_destroy(&g_cond);
    pthread_mutex_destroy(&g_mutex);

    return 0;
}

void exit_sys_thread(const char *msg, int err)
{
    fprintf(stderr, "%s: %s\n", msg, strerror(err));
    exit(EXIT_FAILURE);
}

void *thread_proc1(void *param)
{
    int result;

    printf("thread1 is running...\n");

    printf("press ENTER to signal!..\n");
    getchar();

    if ((result = pthread_mutex_lock(&g_mutex)) != 0)
        exit_sys_thread("pthread_mutex_lock", result);

    g_condition = 1;

    if ((result = pthread_mutex_unlock(&g_mutex)) != 0)

```

```

        exit_sys_thread("pthread_mutex_unlock", result);

    if ((result = pthread_cond_broadcast(&g_cond)) != 0)
        exit_sys_thread("pthread_cond_broadcast", result);

    return NULL;
}

void *thread_proc2(void *param)
{
    int result;

    printf("thread2 waiting for the condition...\n");

    if ((result = pthread_mutex_lock(&g_mutex)) != 0)
        exit_sys_thread("pthread_mutex_lock", result);

    while (g_condition == 0)
        if ((result = pthread_cond_wait(&g_cond, &g_mutex)) != 0)
            exit_sys_thread("pthread_cond_wait", result);

    if ((result = pthread_mutex_unlock(&g_mutex)) != 0)
        exit_sys_thread("pthread_mutex_unlock", result);

    printf("thread2 condition granted...\n");

    return NULL;
}

void *thread_proc3(void *param)
{
    int result;

    printf("thread3 waiting for the condition...\n");

    if ((result = pthread_mutex_lock(&g_mutex)) != 0)
        exit_sys_thread("pthread_mutex_lock", result);

    while (g_condition == 0)
        if ((result = pthread_cond_wait(&g_cond, &g_mutex)) != 0)
            exit_sys_thread("pthread_cond_wait", result);

    if ((result = pthread_mutex_unlock(&g_mutex)) != 0)
        exit_sys_thread("pthread_mutex_unlock", result);

    printf("thread3 condition granted...\n");

    return NULL;
}

/
*-----

```

Üretici-Tüketici Problemi (Producer-Consumer Problem) gerçek hayatta en sık karşılaşılan thread senkronizasyon kalıbıdır.

Bu problemde üretici thread bir döngü içerisinde bir faaliye sonucunda bir değer elde eder. Bu değeri kendisi işlemez. Paylaşılan bir global değişkene yazar. Diğer thread yine bir döngü içerisinde o global değişkendeki değeri alarak onu işler. Yani problemde thread'lerden biri değerleri elde etme işini diğeri ise bunları işleme işini yapmaktadır. Değerleri elde eden thread'e üretici thread, değerleri alıp işleyen thread'e de tüketici thread denilmektedir. Şüphesiz tek bir değer değerleri elde edip kendisi işleyebilirdi.

Ancak değerlerin elde edilmesi ve işlenmesi iki farklı thread'e yaptırıldığında hız kazancı sağlanmaktadır. Buradaki problemde şöyle bir senkronizasyon sorunu vardır: Üretici thread elde ettiği değeri global değişkene yazdıktan sonra yeni bir değeri tüketici onu almadan global değişkene yerleştirmemelidir. Benzer biçimde tüketici thread de eski değeri ikinci kez alıp işlememelidir. Yani üretici thread ancak tüketici thread önceki değeri aldıysa yeni değeri yerleştirmelidir. Benzer biçimde tüketici thread de üretici thread yeni bir değer yerleştirdiyse onu almalıdır. Aşağıda bir senkroniasyon uygulanmadan yapılan bir simülasyonu görüyorsunuz.

```
-----*/

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

void exit_sys_thread(const char *msg, int err);
void *thread_proc_producer(void *param);
void *thread_proc_consumer(void *param);

int g_shared;

int main(void)
{
    int result;
    pthread_t tid_producer, tid_consumer;

    srand(time(NULL));

    if ((result = pthread_create(&tid_producer, NULL, thread_proc_producer,
        NULL)) != 0)
        exit_sys_thread("pthread_create", result);

    if ((result = pthread_create(&tid_consumer, NULL, thread_proc_consumer,
        NULL)) != 0)
        exit_sys_thread("pthread_create", result);

    if ((result = pthread_join(tid_producer, NULL)) != 0)
        exit_sys_thread("pthread_join", result);

    if ((result = pthread_join(tid_consumer, NULL)) != 0)
        exit_sys_thread("pthread_join", result);
}
```

```

    return 0;
}

void exit_sys_thread(const char *msg, int err)
{
    fprintf(stderr, "%s: %s\n", msg, strerror(err));
    exit(EXIT_FAILURE);
}

void *thread_proc_producer(void *param)
{
    unsigned int seedval;
    int i;

    seedval = (unsigned int)time(NULL) + 20000;

    i = 0;
    for (;;) {
        usleep(rand_r(&seedval) % 300000);
        g_shared = i;
        if (i == 99)
            break;
        ++i;
    }

    return NULL;
}

void *thread_proc_consumer(void *param)
{
    unsigned int seedval;
    int val;

    seedval = (unsigned int)time(NULL);

    for (;;) {
        val = g_shared;
        usleep(rand_r(&seedval) % 300000);
        printf("%d ", val);
        fflush(stdout);
        if (val == 99)
            break;
    }
    printf("\n");

    return NULL;
}

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

```

Üretici-Tüketici problemi semaphore nesneleriyle ya da koşul değişken nesneleriyle çözülebilir. Aslında semaphore'lar

bu konuda daha kolay bir kod oluşturmaktadır. Ancak çözüm yöntemi aynıdır. Çözümdeki temel fikir üreticinin tüketiciyi uyandırması, tüketicinin de üreticiyi uyandırmasıdır. İki koşul değişkeni alınır. Koşulu belirten global bir flag de başlangıçta 0 ilkdeğeriyle tanımlanır. Üretici bu değişken 1 olduğu sürece, tüketici de 0 olduğu sürece bekleyecektir. İşin başında bu flag sıfır olduğuna göre tüketici bekler durumdadır. Fakat üretici uykuya dalmaz. Üretici paylaşılan alana değeri yerleştirir. Fakat flag değişkenini kendisi 1 yapar. Tüketiciiyi pthread_cond_signal ile uyandırır. Tüketici uyandığında flag atık 1 olduğu için durum değişkeninden geçer ancak üretici bu sefer flag 1 olduğu için beklemektedir. İşte tüketici paylaşılan alandan bilgiyi alır. flag'i yeniden 0 yapıp üreticiyi uyandırmak için pthread_cond_signal çağrısı yapar. Burada bir tateravalli gibi üretici tüketiciyi, tüketici de üreticiyi uyandırmaktadır.

```
-----*/

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

void exit_sys_thread(const char *msg, int err);
void *thread_proc_producer(void *param);
void *thread_proc_consumer(void *param);

pthread_mutex_t g_mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t g_cond_producer = PTHREAD_COND_INITIALIZER;
pthread_cond_t g_cond_consumer = PTHREAD_COND_INITIALIZER;

int g_shared;
int g_flag;

int main(void)
{
    int result;
    pthread_t tid_producer, tid_consumer;

    if ((result = pthread_create(&tid_producer, NULL, thread_proc_producer,
        NULL)) != 0)
        exit_sys_thread("pthread_create", result);

    if ((result = pthread_create(&tid_consumer, NULL, thread_proc_consumer,
        NULL)) != 0)
        exit_sys_thread("pthread_create", result);

    if ((result = pthread_join(tid_producer, NULL)) != 0)
        exit_sys_thread("pthread_join", result);

    if ((result = pthread_join(tid_consumer, NULL)) != 0)
        exit_sys_thread("pthread_join", result);

    pthread_cond_destroy(&g_cond_producer);
```

```

    pthread_cond_destroy(&g_cond_consumer);
    pthread_mutex_destroy(&g_mutex);

    return 0;
}

void exit_sys_thread(const char *msg, int err)
{
    fprintf(stderr, "%s: %s\n", msg, strerror(err));
    exit(EXIT_FAILURE);
}

void *thread_proc_producer(void *param)
{
    unsigned int seedval;
    int result, i;

    seedval = (unsigned int)time(NULL) + 20000;

    i = 0;
    for (;;) {
        usleep(rand_r(&seedval) % 300000);

        if ((result = pthread_mutex_lock(&g_mutex)) != 0)
            exit_sys_thread("pthread_mutex_lock", result);

        while (g_flag == 1)
            if ((result = pthread_cond_wait(&g_cond_producer, &g_mutex)) !=
                0)
                exit_sys_thread("pthread_cond_wait", result);

        g_shared = i;
        g_flag = 1;

        if ((result = pthread_cond_signal(&g_cond_consumer)) != 0)
            exit_sys_thread("pthread_cond_signal", result);

        if ((result = pthread_mutex_unlock(&g_mutex)) != 0)
            exit_sys_thread("pthread_mutex_unlock", result);

        if (i == 99)
            break;
        ++i;
    }

    return NULL;
}

void *thread_proc_consumer(void *param)
{
    unsigned int seedval;
    int result, val;

    seedval = (unsigned int)time(NULL);

```

```

for (;;) {
    if ((result = pthread_mutex_lock(&g_mutex)) != 0)
        exit_sys_thread("pthread_mutex_lock", result);

    while (g_flag == 0)
        if ((result = pthread_cond_wait(&g_cond_consumer,
            &g_mutex)) != 0)
            exit_sys_thread("pthread_cond_wait", result);

    val = g_shared;
    g_flag = 0;

    if ((result = pthread_cond_signal(&g_cond_producer)) != 0)
        exit_sys_thread("pthread_cond_signal", result);

    if ((result = pthread_mutex_unlock(&g_mutex)) != 0)
        exit_sys_thread("pthread_mutex_unlock", result);

    usleep(rand_r(&seedval) % 300000);
    printf("%d ", val);
    fflush(stdout);
    if (val == 99)
        break;
}
printf("\n");

return NULL;
}

```

```

/
*-----
-----
    Üretici-Tüketici probleminde ortada paylaşılan alan tek bir değişken
    yerine bir kuyruk sistemi olabilir. Dolayısıyla bu durumda
    toplam bloke miktarı azalacak ve performans yükselecektir. Aşağıdaki
    örnekte kuyruk sistemi index kaydırma yöntemiyle gerçekleştirilmiştir.
    Üretici elde ettiği değeri kuyruğun sonuna yerleştirir. Tüketici de
    kuyruğun başındaki elemanı alır.
    -----
    -----*/

```

```

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

```

```

#define QSIZE      10

```

```

void exit_sys_thread(const char *msg, int err);
void *thread_proc_producer(void *param);
void *thread_proc_consumer(void *param);

```

```

pthread_mutex_t g_mutex = PTHREAD_MUTEX_INITIALIZER;

```

```

pthread_cond_t g_cond_producer = PTHREAD_COND_INITIALIZER;
pthread_cond_t g_cond_consumer = PTHREAD_COND_INITIALIZER;

int g_queue[QSIZE];
int g_count;
int g_head, g_tail;

int main(void)
{
    int result;
    pthread_t tid_producer, tid_consumer;

    if ((result = pthread_create(&tid_producer, NULL, thread_proc_producer,
        NULL)) != 0)
        exit_sys_thread("pthread_create", result);

    if ((result = pthread_create(&tid_consumer, NULL, thread_proc_consumer,
        NULL)) != 0)
        exit_sys_thread("pthread_create", result);

    if ((result = pthread_join(tid_producer, NULL)) != 0)
        exit_sys_thread("pthread_join", result);

    if ((result = pthread_join(tid_consumer, NULL)) != 0)
        exit_sys_thread("pthread_join", result);

    pthread_cond_destroy(&g_cond_producer);
    pthread_cond_destroy(&g_cond_consumer);
    pthread_mutex_destroy(&g_mutex);

    return 0;
}

void exit_sys_thread(const char *msg, int err)
{
    fprintf(stderr, "%s: %s\n", msg, strerror(err));
    exit(EXIT_FAILURE);
}

void *thread_proc_producer(void *param)
{
    unsigned int seedval;
    int result, i;

    seedval = (unsigned int)time(NULL) + 20000;

    i = 0;
    for (;;) {
        usleep(rand_r(&seedval) % 300000);

        if ((result = pthread_mutex_lock(&g_mutex)) != 0)
            exit_sys_thread("pthread_mutex_lock", result);

        while (g_count == QSIZE)

```



```

        if ((result = pthread_cond_wait(&g_cond_producer, &g_mutex)) !=
            0)
            exit_sys_thread("pthread_cond_wait", result);

        g_queue[g_tail++] = i;
        g_tail %= QSIZE;
        ++g_count;

        if ((result = pthread_cond_signal(&g_cond_consumer)) != 0)
            exit_sys_thread("pthread_cond_signal", result);

        if ((result = pthread_mutex_unlock(&g_mutex)) != 0)
            exit_sys_thread("pthread_mutex_unlock", result);

        if (i == 99)
            break;
        ++i;
    }

    return NULL;
}

void *thread_proc_consumer(void *param)
{
    unsigned int seedval;
    int result, val;

    seedval = (unsigned int)time(NULL);

    for (;;) {
        if ((result = pthread_mutex_lock(&g_mutex)) != 0)
            exit_sys_thread("pthread_mutex_lock", result);

        while (g_count == 0)
            if ((result = pthread_cond_wait(&g_cond_consumer,
                &g_mutex)) != 0)
                exit_sys_thread("pthread_cond_wait", result);

        val = g_queue[g_head++];
        g_head %= QSIZE;
        --g_count;

        if ((result = pthread_cond_signal(&g_cond_producer)) != 0)
            exit_sys_thread("pthread_cond_signal", result);

        if ((result = pthread_mutex_unlock(&g_mutex)) != 0)
            exit_sys_thread("pthread_mutex_unlock", result);

        usleep(rand_r(&seedval) % 300000);
        printf("%d ", val);
        fflush(stdout);
        if (val == 99)
            break;
    }
    printf("\n");
}

```

```

    return NULL;
}

/
*-----
-----
Durum değişkenleri için de attribute girilebilir. Attribute oluşturmak
için pthread_condattr_t türünden bir nesne alınır.
Önce bu nesne pthread_condattr_init ile ilkdeğerlenir. Sonra
pthread_condattr_setxxx fonksiyonlarıyla özellikler set edilir.
İşte bu attribute nesnesi durum değişken nesnesi pthread_cond_init
fonksiyonuyla yaratılırken ona parametre olarak geçilmektedir.
Durum değişken nesnesi yaratıldıktan sonra pthread_condattr_destroy
fonksiyonu ile attribute nesnesi yok edilir. Aslında toplamda
yalnızca bir tek özellik vardır. O da durum değişken nesnesinin
proseslerarası kullanımı ile ilgilidir. pthread_condattr_setpshared
fonksiyonunda PTHREAD_PROCESS_SHARED parametresi kullanılırsa durum
değişken nesnesi prosesler arasında kullanılabilir.
Aşağıdaki örnekte bir paylaşılan bellek alanı oluşturulmuş. Sonra
üretici-tüketici problemi için gereken tüm nesneler bu paylaşılan
bellek alanında yaratılmıştır. Böylece iki farklı proses kendi
aralarında üretici tüketici işlemlerini yapmaktadır.
-----
-----*/

/* producer.c */

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

#define SHM_PATH          "/shared_memory_cond_test"
#define QSIZE             10

typedef struct {
    pthread_mutex_t mutex;
    pthread_cond_t cond_producer;
    pthread_cond_t cond_consumer;
    int queue[QSIZE];
    int head;
    int tail;
    int count;
} PROD_CONS;

void exit_sys(const char *msg);
void exit_sys_thread(const char *msg, int err);

int main(void)

```

```

{
    int fdshm;
    void *addr;
    pthread_mutexattr_t mattr;
    pthread_condattr_t cattr;
    PROD_CONS *prod_cons;
    int result;
    int i;

    srand(time(NULL));

    if ((fdshm = shm_open(SHM_PATH, O_CREAT|O_RDWR, S_IRUSR|S_IWUSR|S_IRGRP|
        S_IROTH)) == -1)
        exit_sys("shm_open");

    if (ftruncate(fdshm, 4096) == -1)
        exit_sys("ftruncate");

    if ((addr = mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_SHARED, fdshm,
        0)) == MAP_FAILED)
        exit_sys("mmap");

    prod_cons = (PROD_CONS *)addr;

    pthread_mutexattr_init(&mattr);
    if ((result = pthread_mutexattr_setpshared(&mattr,
        PTHREAD_PROCESS_SHARED)) != 0)
        exit_sys_thread("pthread_mutexattr_setpshared", result);

    if ((result = pthread_mutex_init(&prod_cons->mutex, &mattr)) != 0)
        exit_sys_thread("pthread_mutex_init", result);

    pthread_mutexattr_destroy(&mattr);

    if ((result = pthread_condattr_init(&cattr)) != 0)
        exit_sys_thread("pthread_condattr_init", result);

    if ((result = pthread_condattr_setpshared(&cattr, PTHREAD_PROCESS_SHARED
        )) != 0)
        exit_sys_thread("pthread_condattr_setpshared", result);

    if ((result = pthread_cond_init(&prod_cons->cond_producer, &cattr)) !=
        0)
        exit_sys_thread("pthread_condinit", result);

    if ((result = pthread_cond_init(&prod_cons->cond_consumer, &cattr)) !=
        0)
        exit_sys_thread("pthread_condinit", result);

    pthread_condattr_destroy(&cattr);

    i = 0;
    for (;;) {
        usleep(rand() % 300000);
    }
}

```

```

        if ((result = pthread_mutex_lock(&prod_cons->mutex)) != 0)
            exit_sys_thread("pthread_mutex_lock", result);

        while (prod_cons->count == QSIZE)
            if ((result = pthread_cond_wait(&prod_cons->cond_producer,
                &prod_cons->mutex)) != 0)
                exit_sys_thread("pthread_cond_wait", result);

        prod_cons->queue[prod_cons->tail++] = i;
        prod_cons->tail %= QSIZE;
        ++prod_cons->count;

        if ((result = pthread_cond_signal(&prod_cons->cond_consumer)) != 0)
            exit_sys_thread("pthread_cond_signal", result);

        if ((result = pthread_mutex_unlock(&prod_cons->mutex)) != 0)
            exit_sys_thread("pthread_mutex_unlock", result);

        if (i == 99)
            break;
        ++i;
    }

    sleep(5);

    pthread_cond_destroy(&prod_cons->cond_producer);
    pthread_cond_destroy(&prod_cons->cond_consumer);
    pthread_mutex_destroy(&prod_cons->mutex);
    munmap(addr, 4096);
    close(fdshm);
    if (shm_unlink(SHM_PATH) == -1)
        exit_sys("shm_unlink");

    return 0;
}

void exit_sys_thread(const char *msg, int err)
{
    fprintf(stderr, "%s: %s\n", msg, strerror(err));
    exit(EXIT_FAILURE);
}

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

    exit(EXIT_FAILURE);
}

/* consumer.c */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>

```

```

#include <fcntl.h>
#include <sys/stat.h>
#include <unistd.h>
#include <pthread.h>
#include <sys/mman.h>

#define SHM_PATH        "/shared_memory_cond_test"
#define QSIZE           10

typedef struct {
    pthread_mutex_t mutex;
    pthread_cond_t cond_producer;
    pthread_cond_t cond_consumer;
    int queue[QSIZE];
    int head;
    int tail;
    int count;
} PROD_CONS;

void exit_sys(const char *msg);
void exit_sys_thread(const char *msg, int err);

int main(void)
{
    int fdshm;
    void *addr;
    PROD_CONS *prod_cons;
    int result;
    int val;

    srand(time(NULL));

    if ((fdshm = shm_open(SHM_PATH, O_RDWR, 0)) == -1)
        exit_sys("shm_open");

    if ((addr = mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_SHARED, fdshm,
        0)) == MAP_FAILED)
        exit_sys("mmap");

    prod_cons = (PROD_CONS *)addr;

    for (;;) {
        if ((result = pthread_mutex_lock(&prod_cons->mutex)) != 0)
            exit_sys_thread("pthread_mutex_lock", result);

        while (prod_cons->count == 0)
            if ((result = pthread_cond_wait(&prod_cons->cond_consumer,
                &prod_cons->mutex)) != 0)
                exit_sys_thread("pthread_cond_wait", result);

        val = prod_cons->queue[prod_cons->head++];
        prod_cons->head %= QSIZE;
        --prod_cons->count;

        if ((result = pthread_cond_signal(&prod_cons->cond_producer)) != 0)

```

```

        exit_sys_thread("pthread_cond_signal", result);

        if ((result = pthread_mutex_unlock(&prod_cons->mutex)) != 0)
            exit_sys_thread("pthread_mutex_unlock", result);

        usleep(rand() % 300000);

        printf("%d ", val);
        fflush(stdout);
        if (val == 99)
            break;
    }

    printf("\n");

    close(fdshm);

    return 0;
}

```

```

void exit_sys_thread(const char *msg, int err)
{
    fprintf(stderr, "%s: %s\n", msg, strerror(err));
    exit(EXIT_FAILURE);
}

```

```

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

    exit(EXIT_FAILURE);
}

```

```

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

barrier nesneleri n tane thread'in belli bir noktaya geldikten sonra
yollarına devam etmesi için düşünülmüştür. Barrier nesnesi
pthread_barrier_t türüyle temsil edilmiştir. Bu nesne
pthread_barrier_init fonksiyonuyla ilkdeğerlenir. Bu fonksiyonda bir
sayaç değeri
de belirtilmektedir. Bundan sonra bekleme işlemi için
pthread_barrier_wait fonksiyonu kullanılır. İşte sayaç değeri n olmak
üzere
n tane thread'e kadar her thread pthread_barrier_wait fonksiyonunda
bloke olarak bekler. En son n'inci thread de pthread_barrier_wait
çağrısına geldiğinde uyuyan tüm thread'ler uyanarak yoluna devam
ederler. Bu nesne genellikle bir işin çeşitli parçalarını yapan
thread'lerin
kendi işlerini bitirdikten sonra bekletilmeleri için kullanılmaktadır.
Bu tür uygulamalarda genellikle bu thread'lerden bir tanesi
barrier çıkışında özel bir işlem yapar. Programcı bu thread'i kendisi
belirleyebilir. Ancak barrier tasarımcıları bu işi kolaşlaştırmak
için n thread'in n - 1 tanesini 0 ile yalnızca herhangi bir tanesini
PTHREAD_BARRIER_SERIAL_THREAD değeri ile geri dönmektedir. Programcının

```

bu değeri error değeri olarak yotumlamaması gerekir. Aşağıdaki örnekte 4 farklı thread (main thread de dahil) bir barrier nesnesine ulaşana kadar bekletilmiştir.

```
-----*/

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

void exit_sys_thread(const char *msg, int err);
void *thread_proc1(void *param);
void *thread_proc2(void *param);
void *thread_proc3(void *param);

pthread_barrier_t g_barrier;

int main(void)
{
    int result;
    pthread_t tid1, tid2, tid3;

    if ((result = pthread_barrier_init(&g_barrier, NULL, 4)) != 0)
        exit_sys_thread("pthread_barrier_init", result);

    if ((result = pthread_create(&tid1, NULL, thread_proc1, NULL)) != 0)
        exit_sys_thread("pthread_create", result);

    sleep(1);

    if ((result = pthread_create(&tid2, NULL, thread_proc2, NULL)) != 0)
        exit_sys_thread("pthread_create", result);

    sleep(1);

    if ((result = pthread_create(&tid3, NULL, thread_proc3, NULL)) != 0)
        exit_sys_thread("pthread_create", result);

    sleep(1);

    printf("main threads enters barrier...\n");

    if ((result = pthread_barrier_wait(&g_barrier)) != 0 && result !=
        PTHREAD_BARRIER_SERIAL_THREAD)
        exit_sys_thread("pthread_barrier_wait", result);

    printf("main thread exits barrier...\n");

    if ((result = pthread_join(tid1, NULL)) != 0)
        exit_sys_thread("pthread_join", result);

    if ((result = pthread_join(tid2, NULL)) != 0)
```

```

        exit_sys_thread("pthread_join", result);

    if ((result = pthread_join(tid3, NULL)) != 0)
        exit_sys_thread("pthread_join", result);

    pthread_barrier_destroy(&g_barrier);

    return 0;
}

void exit_sys_thread(const char *msg, int err)
{
    fprintf(stderr, "%s: %s\n", msg, strerror(err));
    exit(EXIT_FAILURE);
}

void *thread_proc1(void *param)
{
    int result;

    printf("thread1 enters barrier...\n");

    if ((result = pthread_barrier_wait(&g_barrier)) != 0 && result !=
        PTHREAD_BARRIER_SERIAL_THREAD)
        exit_sys_thread("pthread_barrier_wait", result);

    printf("thread1 exits barrier...\n");

    return NULL;
}

void *thread_proc2(void *param)
{
    int result;

    printf("thread2 enters barrier...\n");

    if ((result = pthread_barrier_wait(&g_barrier)) != 0 && result !=
        PTHREAD_BARRIER_SERIAL_THREAD)
        exit_sys_thread("pthread_barrier_wait", result);

    printf("thread2 exits barrier...\n");

    return NULL;
}

void *thread_proc3(void *param)
{
    int result;

    printf("thread3 enters barrier...\n");

    if ((result = pthread_barrier_wait(&g_barrier)) != 0 && result !=
        PTHREAD_BARRIER_SERIAL_THREAD)
        exit_sys_thread("pthread_barrier_wait", result);

```



```

    printf("thread3 exits barrier...\n");

    return NULL;
}

/
*-----
-----
Aşağıda bir barrier kullanım örneği verilmiştir. 50000000 rastgele
değerlerdn oluşan int türden bir dizi 10 parçaya
bölünmüş ve her parça bir thread tarafından qsort fonksiyonu ile sort
edilmiştir. Sonra da bu 10 kendi aralarında sıralı olan
parçalar birleştirilmiştir. 4 çekirdekli bir Linux sisteminde (sanal
makine) toplam zaman 8.50 saniye civarındadır. Daha sonra
aynı dizi tek bir thread'le (ana thread'le) yine qsort fonksiyonu
kullanılarak sıraya dizilmiştir. Bu da 26.36 saniye civarında
bir zaman almıştır. Görüldüğü gibi thread'li versiyonda yaklaşık 3 kat
daha hızlı sonuç elde edilmiştir.
-----
-----*/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include <unistd.h>
#include <pthread.h>

/* Thread'siz 50000000 için 11.30 saniye (100000000 için 43.1)*/

#define SIZE          50000000
#define NTHREADS      10

void exit_sys_thread(const char *msg, int err);
void *thread_proc(void *param);
int comp(const void *pv1, const void *pv2);
void merge(void);
int check(void);

pthread_barrier_t g_barrier;
int g_nums[SIZE];
int g_snums[SIZE];

int main(void)
{
    int result;
    int i;
    pthread_t tids[NTHREADS];

    srand(time(NULL));
    for (i = 0; i < SIZE; ++i)
        g_nums[i] = rand();

```

```

    if ((result = pthread_barrier_init(&g_barrier, NULL, NTHREADS)) != 0)
        exit_sys_thread("pthread_barrier_init", result);

    for (i = 0; i < NTHREADS; ++i)
        if ((result = pthread_create(&tids[i], NULL, thread_proc, (void
            *)i)) != 0)
            exit_sys_thread("pthread_create", result);

    for (i = 0; i < NTHREADS; ++i)
        if ((result = pthread_join(tids[i], NULL)) != 0)
            exit_sys_thread("pthread_join", result);

    pthread_barrier_destroy(&g_barrier);

    printf(check() ? "Sorted\n" : "Not Sorted\n");

    return 0;
}

void exit_sys_thread(const char *msg, int err)
{
    fprintf(stderr, "%s: %s\n", msg, strerror(err));
    exit(EXIT_FAILURE);
}

void *thread_proc(void *param)
{
    int part = (int)param;
    int result;

    qsort(g_snums + part * (SIZE / NTHREADS), SIZE / NTHREADS, sizeof(int),
        comp);

    result = pthread_barrier_wait(&g_barrier);
    if (result != 0 && result != PTHREAD_BARRIER_SERIAL_THREAD)
        exit_sys_thread("pthread_barrier_wait", result);
    if (result == PTHREAD_BARRIER_SERIAL_THREAD)
        merge();

    /* other jobs... */

    return NULL;
}

int comp(const void *pv1, const void *pv2)
{
    const int *pi1 = (const int *)pv1;
    const int *pi2 = (const int *)pv2;

    return *pi1 - *pi2;
}

void merge(void)
{
    int indexes[NTHREADS];

```

```

int min, min_index;
int i, k;
int partsize;

partsize = SIZE / NTHREADS;

for (i = 0; i < NTHREADS; ++i)
    indexes[i] = i * partsize;

for (i = 0; i < SIZE; ++i) {
    min = indexes[0];
    min_index = 0;

    for (k = 1; k < NTHREADS; ++k)
        if (indexes[k] < (k + 1) * partsize && g_nums[indexes[k]] < min)
        {
            min = g_nums[indexes[k]];
            min_index = k;
        }
    g_snums[i] = min;
    ++indexes[min_index];
}

}

int check(void)
{
    int i;

    for (i = 0; i < SIZE - 1; ++i)
        if (g_snums[i] > g_snums[i + 1])
            return 0;

    return 1;
}

```

/

*-----

Posix semaphore nesneleri sem_init ya da sem_open fonksiyonuyla yaratılır. Kritik kod sem_wait ve sem_post arasına yerleştirilir. sem_wait fonksiyonu eğer semaphore sayacı 0 ise blokeye yol açar. Eğer semaphore sayacı 0'dan büyükse bu fonksiyon sayacı 1 eksilterek kritik koda girişe izin verir. sem_post ise semaphore sayacını 1 artırmaktadır. Semaphore'un başlangıçtaki sayacı sem_init ya da sem_open fonksiyonlarında belirtilir. Bu değer kritik koda en fazla kaç thread akışının girebileceğini belirtmektedir. Semaphore sayacı 1 olan semaphore'lara "binary semaphore" denilmektedir. Binary semaphore'lar mutex nesnelere benzer. Fakat mutex nesnelerinin thread temelinde sahipliği vardır. Yani başka bir thread mutex'in sahipliğini bırakamaz. Ancak semaphore'lar böyle değildir. Bu nedenle üretici-tüketici tarzı problemler semaphore nesneleri ile çok kolay çözülebilmektedir. Semaphore nesneleri tipik olarak n tane kaynağın thread'lere

paylaştırılması için tercih edilmelidir. Semaphore nesneleri kullanım bittikten sonra sem_destroy fonksiyonula yok edilmelidir. Aşağıda bir binary semaphore örneği verilmiştir.

```
-----*/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <pthread.h>
#include <semaphore.h>

#define MAX      1000000

void exit_sys(const char *msg);
void exit_sys_thread(const char *msg, int err);

void *thread_proc1(void *param);
void *thread_proc2(void *param);

sem_t g_sem;
int g_count;

int main(void)
{
    int result;
    int i;
    pthread_t tid1, tid2;

    if (sem_init(&g_sem, 0, 1) == -1)
        exit_sys("sem_init");

    if ((result = pthread_create(&tid1, NULL, thread_proc1, NULL)) != 0)
        exit_sys_thread("pthread_create", result);

    if ((result = pthread_create(&tid2, NULL, thread_proc2, NULL)) != 0)
        exit_sys_thread("pthread_create", result);

    if ((result = pthread_join(tid1, NULL)) != 0)
        exit_sys_thread("pthread_join", result);

    if ((result = pthread_join(tid2, NULL)) != 0)
        exit_sys_thread("pthread_join", result);

    sem_destroy(&g_sem);

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

    return 0;
}

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

```

    exit(EXIT_FAILURE);
}

void exit_sys_thread(const char *msg, int err)
{
    fprintf(stderr, "%s: %s\n", msg, strerror(err));
    exit(EXIT_FAILURE);
}

```

```

void *thread_proc1(void *param)
{
    int i;

    for (i = 0; i < MAX; ++i) {
        if (sem_wait(&g_sem) == 1)
            exit_sys("sem_wait");

        ++g_count;

        if (sem_post(&g_sem) == 1)
            exit_sys("sem_post");
    }

    return NULL;
}

```

```

void *thread_proc2(void *param)
{
    int i;

    for (i = 0; i < MAX; ++i) {
        if (sem_wait(&g_sem) == 1)
            exit_sys("sem_wait");

        ++g_count;

        if (sem_post(&g_sem) == 1)
            exit_sys("sem_post");
    }

    return NULL;
}

```

```

/
*-----
-----
Aşağıdaki örnekte global değişken yerine heap'te tahsis edilen bir alan
thread'lere parametre yoluyla aktarılmıştır.
-----
-----*/

```

```

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <pthread.h>

```

```

#include <semaphore.h>

#define MAX          1000000

void exit_sys(const char *msg);
void exit_sys_thread(const char *msg, int err);
void *thread_proc1(void *param);
void *thread_proc2(void *param);

typedef struct {
    sem_t sem;
    int count;
} THREAD_PARAM;

int main(void)
{
    int result;
    pthread_t tid1, tid2;
    THREAD_PARAM *tparam;

    if ((tparam = (THREAD_PARAM *)malloc(sizeof(THREAD_PARAM))) == NULL) {
        fprintf(stderr, "cannot allocate memory!..\n");
        exit(EXIT_FAILURE);
    }

    tparam->count = 0;

    if (sem_init(&tparam->sem, 0, 1) == -1)
        exit_sys("sem_init");

    if ((result = pthread_create(&tid1, NULL, thread_proc1, tparam)) != 0)
        exit_sys_thread("pthread_create", result);

    if ((result = pthread_create(&tid2, NULL, thread_proc2, tparam)) != 0)
        exit_sys_thread("pthread_create", result);

    if ((result = pthread_join(tid1, NULL)) != 0)
        exit_sys_thread("pthread_join", result);

    if ((result = pthread_join(tid2, NULL)) != 0)
        exit_sys_thread("pthread_join", result);

    sem_destroy(&tparam->sem);

    printf("%d\n", tparam->count);

    free(tparam);

    return 0;
}

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

```

```

    exit(EXIT_FAILURE);
}

void exit_sys_thread(const char *msg, int err)
{
    fprintf(stderr, "%s: %s\n", msg, strerror(err));
    exit(EXIT_FAILURE);
}

void *thread_proc1(void *param)
{
    int i;
    THREAD_PARAM *tparam = (THREAD_PARAM *)param;

    for (i = 0; i < MAX; ++i) {
        if (sem_wait(&tparam->sem) == -1)
            exit_sys("sem_wait");

        ++tparam->count;

        if (sem_post(&tparam->sem) == -1)
            exit_sys("sem_post");
    }

    return NULL;
}

void *thread_proc2(void *param)
{
    int i;
    THREAD_PARAM *tparam = (THREAD_PARAM *)param;

    for (i = 0; i < MAX; ++i) {
        if (sem_wait(&tparam->sem) == -1)
            exit_sys("sem_wait");

        ++tparam->count;

        if (sem_post(&tparam->sem) == -1)
            exit_sys("sem_post");
    }

    return NULL;
}

/
*-----

```

Üretici-Tüketici problemi semaphore nesneleri ile daha sade bir biçimde
 çözülebilir. Aşağıdaki örnekte QSIZE kadar bir kuyruk
 oluşturulmuştur. Üretici bu kuyruğa elde ettiği değerleri eklerken
 tüketici de kuyruktan değerleri almaktadır. İki semaphore
 kullanılmıştır. Üretici semaphore'unun sayacı başlangıçta kuyruk
 uzunluğu (QSIZE), tüketici semaphore'unun sayacı ise başlangıçta

0 durumundadır. Üretici tüketicinin, tüketici de üreticinin semaphore sayaçlarını sem_post ile artırmaktadır. Kuyruktaki eleman sayısını tutmanın bir gereği yoktur. Üretici ve tüketici kuyrukta aynı bölüme üzerinde çalışmadıklarından aynı anda kuyruğa erişimlerinde bir sorun olmayacaktır.

```
-----*/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <pthread.h>
#include <semaphore.h>

#define QSIZE      10

void exit_sys(const char *msg);
void exit_sys_thread(const char *msg, int err);
void *thread_proc_producer(void *param);
void *thread_proc_consumer(void *param);

sem_t g_sem_producer;
sem_t g_sem_consumer;

int g_queue[QSIZE];
int g_head, g_tail;

int main(void)
{
    int result;
    pthread_t tid_producer, tid_consumer;

    if ((result = sem_init(&g_sem_producer, 0, QSIZE)) != 0)
        exit_sys_thread("sem_init", result);

    if ((result = sem_init(&g_sem_consumer, 0, 0)) != 0)
        exit_sys_thread("sem_init", result);

    if ((result = pthread_create(&tid_producer, NULL, thread_proc_producer,
                                NULL)) != 0)
        exit_sys_thread("pthread_create", result);

    if ((result = pthread_create(&tid_consumer, NULL, thread_proc_consumer,
                                NULL)) != 0)
        exit_sys_thread("pthread_create", result);

    if ((result = pthread_join(tid_producer, NULL)) != 0)
        exit_sys_thread("pthread_join", result);

    if ((result = pthread_join(tid_consumer, NULL)) != 0)
        exit_sys_thread("pthread_join", result);

    sem_destroy(&g_sem_producer);
    sem_destroy(&g_sem_consumer);
}
```



```

    return 0;
}

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

    exit(EXIT_FAILURE);
}

void exit_sys_thread(const char *msg, int err)
{
    fprintf(stderr, "%s: %s\n", msg, strerror(err));
    exit(EXIT_FAILURE);
}

void *thread_proc_producer(void *param)
{
    unsigned int seedval;
    int i;

    seedval = (unsigned int)time(NULL) + 20000;

    i = 0;
    for (;;) {
        usleep(rand_r(&seedval) % 300000);

        if (sem_wait(&g_sem_producer) == -1)
            exit_sys("sem_wait");

        g_queue[g_tail++] = i;
        g_tail %= QSIZE;

        if (sem_post(&g_sem_consumer) == 1)
            exit_sys("sem_post");

        if (i == 99)
            break;
        ++i;
    }

    return NULL;
}

void *thread_proc_consumer(void *param)
{
    unsigned int seedval;
    int val;

    seedval = (unsigned int)time(NULL);

    for (;;) {
        if (sem_wait(&g_sem_consumer) == -1)
            exit_sys("sem_wait");
    }
}

```

```

        val = g_queue[g_head++];
        g_head %= QSIZE;

        if (sem_post(&g_sem_producer) == -1)
            exit_sys("sem_wait");

        usleep(rand_r(&seedval) % 300000);
        printf("%d ", val);
        fflush(stdout);
        if (val == 99)
            break;
    }
    printf("\n");

    return NULL;
}

```

```

/
*-----
-----
    POSIX semaphore nesneleri iki biçimde prosesler arasında kullanılır.
    Birincisi semaphore nesnesini sem_init ile paylaşılan bellek alanında
    pshared parametresi sıfır dışı geçilerek yaratmaktır. İkincisi POSIX
    paylaşılan bellek alanları ya da mesaj kuyruklarında olduğu gibi
    kök dizinde bir dosya ismi vererek sem_open fonksiyonuyla yaratmak ve
    açmaktır. (Buna "isimli semaphore'lar" da denilmektedir.)
    sem_open fonksiyonuyla yaratılan ya da açılan semaphore nesneleri
    sem_close ile kapatılmalıdır (sem_destroy ile değil).
    Aşağıda üretici tüketici probleminin isimli semaphore'larla prosesler
    arasında gerçekleştirimine ilişkin örnek bir kod bulunmaktadır.
    -----
    -----*/

```

```

/* producer.c */

```

```

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/mman.h>
#include <semaphore.h>

```

```

#define SHM_PATH          "/shared_memory_producer_consumer"
#define SEM_PATH_PRODUCER "/semaphore_producer"
#define SEM_PATH_CONSUMER "/semaphore_consumer"

```

```

#define QSIZE          10

```

```

void exit_sys(const char *msg);
typedef struct {
    int queue[QSIZE];
    int head;

```

```

    int tail;
} PROD_CONS;

int main(void)
{
    int fdshm;
    void *addr;
    sem_t *sem_producer, *sem_consumer;
    PROD_CONS *prod_cons;
    int i;

    srand(time(NULL));

    if ((fdshm = shm_open(SHM_PATH, O_CREAT|O_RDWR, S_IRUSR|S_IWUSR|S_IRGRP|
        S_IROTH)) == -1)
        exit_sys("shm_open");

    if (ftruncate(fdshm, 4096) == -1)
        exit_sys("ftruncate");

    if ((addr = mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_SHARED, fdshm,
        0)) == MAP_FAILED)
        exit_sys("mmap");

    prod_cons = (PROD_CONS *)addr;

    if ((sem_producer = sem_open(SEM_PATH_PRODUCER, O_CREAT|O_RDWR, S_IRUSR|
        S_IWUSR|S_IRGRP|S_IROTH, QSIZE)) == NULL)
        exit_sys("sem_open");

    if ((sem_consumer = sem_open(SEM_PATH_CONSUMER, O_CREAT|O_RDWR, S_IRUSR|
        S_IWUSR|S_IRGRP|S_IROTH, 0)) == NULL)
        exit_sys("sem_open");

    i = 0;
    for (;;) {
        usleep(rand() % 300000);
        if (sem_wait(sem_producer) == -1)
            exit_sys("sem_wait");

        prod_cons->queue[prod_cons->tail++] = i;
        prod_cons->tail %= QSIZE;

        if (sem_post(sem_consumer) == -1)
            exit_sys("sem_post");
        if (i == 99)
            break;
        ++i;
    }

    munmap(addr, 4096);
    close(fdshm);
    if (shm_unlink(SHM_PATH) == -1)
        exit_sys("shm_unlink");
    sem_close(sem_producer);

```

```

    sem_close(sem_consumer);

    return 0;
}

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

    exit(EXIT_FAILURE);
}

/* consumer.c */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/mman.h>
#include <semaphore.h>

#define SHM_PATH          "/shared_memory_producer_consumer"
#define SEM_PATH_PRODUCER "/semaphore_producer"
#define SEM_PATH_CONSUMER "/semaphore_consumer"

#define QSIZE            10

void exit_sys(const char *msg);

typedef struct {
    int queue[QSIZE];
    int head;
    int tail;
} PROD_CONS;

int main(void)
{
    int fdshm;
    void *addr;
    sem_t *sem_producer, *sem_consumer;
    PROD_CONS *prod_cons;
    int i;

    srand(time(NULL));

    if ((fdshm = shm_open(SHM_PATH, O_CREAT|O_RDWR, S_IRUSR|S_IWUSR|S_IRGRP|
        S_IROTH)) == -1)
        exit_sys("shm_open");

    if (ftruncate(fdshm, 4096) == -1)
        exit_sys("ftruncate");

```

```

if ((addr = mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_SHARED, fdshm,
0)) == MAP_FAILED)
    exit_sys("mmap");

prod_cons = (PROD_CONS *)addr;

if ((sem_producer = sem_open(SEM_PATH_PRODUCER, O_CREAT|O_RDWR, S_IRUSR|
S_IWUSR|S_IRGRP|S_IROTH, QSIZE)) == NULL)
    exit_sys("sem_open");

if ((sem_consumer = sem_open(SEM_PATH_CONSUMER, O_CREAT|O_RDWR, S_IRUSR|
S_IWUSR|S_IRGRP|S_IROTH, 0)) == NULL)
    exit_sys("sem_open");

i = 0;
for (;;) {
    usleep(rand() % 300000);
    if (sem_wait(sem_producer) == -1)
        exit_sys("sem_wait");

    prod_cons->queue[prod_cons->tail++] = i;
    prod_cons->tail %= QSIZE;

    if (sem_post(sem_consumer) == -1)
        exit_sys("sem_post");
    if (i == 99)
        break;
    ++i;
}

munmap(addr, 4096);
close(fdshm);
if (shm_unlink(SHM_PATH) == -1)
    exit_sys("shm_unlink");
sem_close(sem_producer);
sem_close(sem_consumer);

return 0;
}

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

    exit(EXIT_FAILURE);
}

```

Sistem 5 semaphore'ları aslında 3 sistem IPC nesnesinin üçüncüsüdür. Genel kullanım biçimi sistem 5 paylaşılan bellek alanı ve sistem 5 mesaj kuyruklarında olduğu gibidir. Semaphore kullanımı için semget, semctl, semop isimli üç fonksiyondan faydalanılmaktadır.

Sistem 5 semaphore'ları aynı prosesin thread'ler arasındaki senkronizasyon için değil farklı prosesler arasındaki senkronizasyon için kullanılmaktadır. (Zaten o zamanlar thread kavramı yoktu.) Arayüz biraz karışıktır. Bu karışıklığın en önemli nedeni bu fonksiyonların tek bir semaphore üzerinde değil bir grup semaphore üzerinde (semaphore set) işlem yapıyor olmasındandır. Ayrıca semaphore sayaç mekanizması da biraz karmaşık tasarlanmıştır. semget fonksiyonu yine anahtar verildiğinde id'nin elde edilmesi amacıyla kullanılır. Ancak semget fonksiyonu tek semaphore değil bir grup semaphore (semaphore set) yaratmaktadır. Semaphore kümesinin kaç semaphore'dan oluşacağı semget fonksiyonun ikinci parametresinde belirtilir.

```
int semget(key_t key, int nsems, int semflg);
```

Semaphore kümesi semget ile yaratıldıktan sonra küme içerisindeki semaphore'ların sayaçları set edilmelidir. Bunun için semctl fonksiyonu kullanılır:

```
int semctl(int semid, int semnum, int cmd, ...);
```

Bu fonksiyonun son parametresi (ellipsis yerindeki parametresi) aşağıdaki gibi bir birlik (union) olmalıdır. Ancak bu birlik herhangi bir başlık dosyasında bildirilmemiştir.

Dolayısıyla programcı tarafından bildirilmelidir:

```
union semun {
    int val;
    struct semid_ds *buf;
    unsigned short *array;
} arg;
```

Fonksiyonun ikinci parametresi semaphore kümesi içerisindeki hangi semaphore'un sayacı ile ilgili işlem yapılacağını belirtir.

Kümedeki semaphore'ların numaraları 0'dan başlamaktadır. Üçüncü parametre yapılacak işlemi belirtir. SETVAL semaphore sayacının set edileceği

anlamına gelir. Bu durumda son parametredeki birliğin val elemanı set edilecek semaphore sayaç değerini tutmalıdır. GETVAL benzer biçimde ilgili semaphore'un sayaç değeri ile geri dönüleceği anlamına gelir.

SETALL tüm kümedeki semaphore'ların sayaç değerlerinin tek hamlede set edileceği

anlamına gelmektedir. Bu durumda tüm sayaç değerleri short int bir diziye yazılmalı bu dizinin başlangıç adresi de birliğin array elemanında

bulunmalıdır. GETALL ise bunun tersini yapar. Tabii semctl'de cmd parametresi için IPC_RMID değeri de girilebilir. Bu da semaphore kümesinin

silineceği anlamına gelmektedir.

Sistem 5 semaphore'larında kritik kod semop fonksiyonuyla oluşturulmaktadır:

```
int semop(int semid, struct sembuf *sops, size_t nsops);
```

Bu fonksiyon da aslında tek hamlede kümedeki nsops tane semaphore için işlem yapabilmektedir. Bu işlemler fonksiyonun ikinci parametresindeki struct sembuf içerisinde kodlanır. Bu yapı <sys/sem.h> başlık dosyasında aşağıdaki gibi bildirilmiştir:

```
struct sembuf {
    unsigned short sem_num;
    short          sem_op;
    short          sem_flg;
}
```

Buradaki sem_num elemanı semahore kümesindeki semaphore numarasını belirtir. sem_flag parametresi 0 geçilebilir. Asıl önemli parametre sem_op parametresidir.

semop pozitif negatif ya da sıfır değerini alabilir. Kritik kodun başında programcı sem_op değerini negatif yaparak (tipik olarak -1) kritik koda giriş yapmaya çalışır. Kritik kodun sonunda da sem_op değeri pozitif yapılır (tipik olarak 1). Eğer semaphore sayacı n iken programcı

semop fonksiyonunu sem_op değeri -k (k'nın mutlak değeri n'den büyük olsun) ile çağırırsa bloke oluşur. Başka bir deyişle semaphore sayacını 0'ın altına indirme

girişimi blokeye yol açmaktadır. Bu blokedan kurtulmanın yolu. Semaphore sayacını en azından 0'a çıkartacak bir semop işlemi yapmaktır. sem_op değeri pozitif ise

bu değer semaphore sayacına toplanır. Örneğin semaphore sayacının değeri 1 olsun. Şimdi biz bu semaphor'ı semop fonksiyonuyla sem_op değeri -1 olacak biçimde

beklemek istersek bloke olmayız. Ancak sem_op değeri -2 olacak biçimde beklemeye çalışırsak bloke oluşur. Bu durumda bizim blokedan kurtulmamız için semaphore sayacının

2'ye yükseltilmesi gerekmektedir. Semahore sayacına semval diyelim. Eğer semop fonksiyonunda sem_op değeri negatifse ve bu değer mutlak değeri semval'dan küçük ise

sayaç eksiltilir ve bloke oluşmaz. (Örneğin semval 2 ikin sem_op -1 ise bloke oluşmaz semval 1 olur.) Ancak sem_op değeri negatif ve mutlak değeri semval değerinden

büyükse budurumda semaphore sayacı eksiltilmez ancak bloke oluşur. Bloke semval değeri pozitif sem_op değerine erişince ortadan kalkar. (Örneğin semval değeri

2 olsun. Biz sem_op -3 ile beklersek semval 2'de kalır ancak bloke oluşur. Bu blokenin çözülmesi için semval değerinin 3 haline gelmesi gerekektir.)

Nihayet sem_op değeri 0 ise ancak semahore sayacı 0 olduğunda bloke oluşur. Bu seçenek pek kullanılmamaktadır.

Mademki sistem 5 semaphore'larının arayüzleri biraz karmaşıktır. O halde sistem 5 semaphore'larını POSIX semaphore'ları gibi kullanabilmek için aşağıdaki gibi sarma (wrapper) fonksiyonlar yazılabilir

-----*/

```

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#include "general.h"

int sem_create(int key, int mode);
int sem_open(int key);
int sem_init(int semid, int val);
int sem_post(int semid);
int sem_wait(int semid);
int sem_destroy(int semid);

int sem_create(int key, int mode)
{
    return semget(key, 1, IPC_CREAT|mode);
}

int sem_open(int key)
{
    return semget(key, 1, 0);
}

int sem_init(int semid, int val)
{
    union semun {
        int val;
        struct semid_ds *buf;
        unsigned short *array;
        struct seminfo *__buf;
    } su;

    su.val = val;

    return semctl(semid, 0, SETVAL, su);
}

int sem_post(int semid)
{
    struct sembuf sb;

    sb.sem_num = 0;
    sb.sem_op = 1;
    sb.sem_flg = 0;

    return semop(semid, &sb, 1);
}

int sem_wait(int semid)
{
    struct sembuf sb;

    sb.sem_num = 0;
    sb.sem_op = -1;

```



```

    sb.sem_flg = 0;

    return semop(semid, &sb, 1);
}

int sem_destroy(int semid)
{
    return semctl(semid, 0, IPC_RMID);
}

/
*-----
-----
Sistem 5 Semaphore'ları ve Sistem 5 Paylaşılan Bellek Alanları İle
Üretici Tüketici Probleminin Gerçekleştirilmesi
-----
-----*/

/* producer.c */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include <unistd.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <sys/sem.h>

#define QSIZE      10

typedef struct {
    int queue[QSIZE];
    int head;
    int tail;
} PROD_CONS;

void exit_sys(const char *msg);
int sem_create(int key, int mode);
int sem_init(int semid, int val);
int sem_post(int semid);
int sem_wait(int semid);
int sem_destroy(int semid);

int main(void)
{
    key_t shmkey;
    int shmid;
    key_t prod_semkey, cons_semkey;
    int prod_semid, cons_semid;
    void *addr;
    PROD_CONS *prod_cons;
    int i;

    srand(time(NULL));

```

```

if ((shmkey = ftok(".", 123)) == -1)
    exit_sys("ftok");

if ((shmid = shmget(shmkey, 4096, IPC_CREAT|0644)) == -1)
    exit_sys("shmget");

if ((addr = shmat(shmid, NULL, 0)) == (void *)-1)
    exit_sys("shmat");

prod_cons = (PROD_CONS *)addr;

if ((prod_semkey = ftok("producer", 123)) == -1)
    exit_sys("ftok");

if ((cons_semkey = ftok("consumer", 123)) == -1)
    exit_sys("ftok");

if ((prod_semid = sem_create(prod_semkey, 0644)) == -1)
    exit_sys("sem_create");

if ((cons_semid = sem_create(cons_semkey, 0644)) == -1)
    exit_sys("sem_create");

if (sem_init(&prod_cons->prod_sem, 0, 1) == -1)
    exit_sys("sem_init");

if (sem_init(&prod_cons->cons_sem, 0, 1) == -1)
    exit_sys("sem_init");

i = 0;
for (;;) {
    usleep(rand() % 300000);
    if (sem_wait(&prod_cons->prod_sem) == -1)
        exit_sys("sem_wait");

    prod_cons->queue[prod_cons->tail++] = i;
    prod_cons->tail %= QSIZE;

    if (sem_post(&prod_cons->cons_sem) == -1)
        exit_sys("sem_post");
    if (i == 99)
        break;
    ++i;
}

printf("press ENTER to exit\n");
getchar();

shmdt(addr);

if (shmctl(shmid, IPC_RMID, NULL) == -1)
    exit_sys("shmctl");

if (sem_destroy(&prod_cons->prod_sem) == -1)

```

```

        exit_sys("sem_destroy");

    if (sem_destroy(&semid) == -1)
        exit_sys("sem_destroy");

    return 0;
}

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

    exit(EXIT_FAILURE);
}

int sem_create(int key, int mode)
{
    return semget(key, 1, IPC_CREAT|mode);
}

int sem_init(int semid, int val)
{
    union semun {
        int val;
        struct semid_ds *buf;
        unsigned short *array;
        struct seminfo *__buf;
    } su;

    su.val = val;

    return semctl(semid, 0, SETVAL, su);
}

int sem_post(int semid)
{
    struct sembuf sb;

    sb.sem_num = 0;
    sb.sem_op = 1;
    sb.sem_flg = 0;

    return semop(semid, &sb, 1);
}

int sem_wait(int semid)
{
    struct sembuf sb;

    sb.sem_num = 0;
    sb.sem_op = -1;
    sb.sem_flg = 0;

    return semop(semid, &sb, 1);
}

```

```

}

int sem_destroy(int semid)
{
    return semctl(semid, 0, IPC_RMID);
}

/* consumer.c */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include <unistd.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <sys/sem.h>

#define QSIZE      10

typedef struct {
    int queue[QSIZE];
    int head;
    int tail;
} PROD_CONS;

void exit_sys(const char *msg);
int sem_open(int key);
int sem_init(int semid, int val);
int sem_post(int semid);
int sem_wait(int semid);
int sem_destroy(int semid);

int main(void)
{
    key_t shmkey;
    int shmid;
    key_t prod_semkey, cons_semkey;
    int prod_semid, cons_semid;
    void *addr;
    PROD_CONS *prod_cons;
    int val;

    srand(time(NULL));

    if ((shmkey = ftok(".", 123)) == -1)
        exit_sys("ftok");

    if ((shmid = shmget(shmkey, 4096, 0)) == -1)
        exit_sys("shmget");

    if ((addr = shmat(shmid, NULL, 0)) == (void *)-1)
        exit_sys("shmat");

    prod_cons = (PROD_CONS *)addr;

```

```

if ((prod_semkey = ftok("producer", 123)) == -1)
    exit_sys("ftok");

if ((cons_semkey = ftok("consumer", 123)) == -1)
    exit_sys("ftok");

if ((prod_semid = sem_open(prod_semkey)) == -1)
    exit_sys("sem_create");

if ((cons_semid = sem_open(cons_semkey)) == -1)
    exit_sys("sem_create");

for (;;) {

    if (sem_wait(cons_semid) == -1)
        exit_sys("sem_wait");

    val = prod_cons->queue[prod_cons->head++];
    prod_cons->head %= QSIZE;

    if (sem_post(prod_semid) == -1)
        exit_sys("sem_post");

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

    if (val == 99)
        break;

    usleep(rand() % 300000);
}
printf("\n");

shmdt(addr);

return 0;
}

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

    exit(EXIT_FAILURE);
}

int sem_open(int key)
{
    return semget(key, 1, 0);
}

int sem_init(int semid, int val)
{
    union semun {
        int val;
    }

```

```

        struct semid_ds *buf;
        unsigned short *array;
        struct seminfo *__buf;
    } su;

    su.val = val;

    return semctl(semid, 0, SETVAL, su);
}

```

```

int sem_post(int semid)
{
    struct sembuf sb;

    sb.sem_num = 0;
    sb.sem_op = 1;
    sb.sem_flg = 0;

    return semop(semid, &sb, 1);
}

```

```

int sem_wait(int semid)
{
    struct sembuf sb;

    sb.sem_num = 0;
    sb.sem_op = -1;
    sb.sem_flg = 0;

    return semop(semid, &sb, 1);
}

```

```

int sem_destroy(int semid)
{
    return semctl(semid, 0, IPC_RMID);
}

```

```

/
*-----
-----
Yukarıdaki programın sarma fonksiyonlar kullanılmadan yazılmış hali
aşağıda verilmiştir. Aşağıdaki kodda tek bir semaphore kümesi
iki semaphore'u içerecek biçimde yaratılmıştır. Bunların sayaçlarına tek
hamlede semctl fonksiyonu ile SETALL komut kodu ile ilkdeğerleri
verilmiştir.
-----
-----*/

```

```

/* producer.c */

```

```

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

```

```

#include <sys/ipc.h>
#include <sys/shm.h>
#include <sys/sem.h>

#define QSIZE      10

typedef struct {
    int queue[QSIZE];
    int head;
    int tail;
} PROD_CONS;

union semun {
    int val;
    struct semid_ds *buf;
    unsigned short *array;
    struct seminfo *__buf;
};

void exit_sys(const char *msg);

int main(void)
{
    key_t shmkey;
    int shmid;
    key_t semkey;
    int semid;
    void *addr;
    PROD_CONS *prod_cons;
    int i;
    union semun su;
    unsigned short semvals[] = {QSIZE, 0};
    struct sembuf sb;

    srand(time(NULL));

    if ((shmkey = ftok(".", 123)) == -1)
        exit_sys("ftok");

    if ((shmid = shmget(shmkey, 4096, IPC_CREAT|0644)) == -1)
        exit_sys("shmget");

    if ((addr = shmat(shmid, NULL, 0)) == (void *)-1)
        exit_sys("shmat");

    prod_cons = (PROD_CONS *)addr;

    if ((semkey = ftok(".", 123)) == -1)
        exit_sys("ftok");

    if ((semid = semget(semkey, 2, IPC_CREAT|0644)) == -1)
        exit_sys("semget");

    su.array = semvals;
    if (semctl(semid, 0, SETALL, su) == -1)

```

```

        exit_sys("semctl");

i = 0;
for (;;) {
    usleep(rand() % 300000);

    sb.sem_num = 0;
    sb.sem_op = -1;
    sb.sem_flg = 0;

    if (semop(semid, &sb, 1) == -1)
        exit_sys("semop");

    prod_cons->queue[prod_cons->tail++] = i;
    prod_cons->tail %= QSIZE;

    sb.sem_num = 1;
    sb.sem_op = 1;
    sb.sem_flg = 0;

    if (semop(semid, &sb, 1) == -1)
        exit_sys("semop");

    if (i == 99)
        break;
    ++i;
}

printf("press ENTER to exit\n");
getchar();

shmdt(addr);

if (shmctl(shmid, IPC_RMID, NULL) == -1)
    exit_sys("shmctl");

if (semctl(semid, 0, IPC_RMID) == -1)
    exit_sys("semctl");

return 0;
}

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

    exit(EXIT_FAILURE);
}

/* consumer.c */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>

```



```

#include <unistd.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <sys/sem.h>

#define QSIZE      10

typedef struct {
    int queue[QSIZE];
    int head;
    int tail;
} PROD_CONS;

union semun {
    int val;
    struct semid_ds *buf;
    unsigned short *array;
    struct seminfo *__buf;
};

void exit_sys(const char *msg);

int main(void)
{
    key_t shmkey;
    int shmid;
    key_t semkey;
    int semid;
    void *addr;
    PROD_CONS *prod_cons;
    int val;
    struct sembuf sb;

    srand(time(NULL));

    if ((shmkey = ftok(".", 123)) == -1)
        exit_sys("ftok");

    if ((shmid = shmget(shmkey, 4096, 0)) == -1)
        exit_sys("shmget");

    if ((addr = shmat(shmid, NULL, 0)) == (void *)-1)
        exit_sys("shmat");

    prod_cons = (PROD_CONS *)addr;

    if ((semkey = ftok(".", 123)) == -1)
        exit_sys("ftok");

    if ((semid = semget(semkey, 2, 0)) == -1)
        exit_sys("semget");

    for (;;) {
        sb.sem_num = 1;
        sb.sem_op = -1;
    }
}

```

```

    sb.sem_flg = 0;

    if (semop(semid, &sb, 1) == -1)
        exit_sys("semop");

    val = prod_cons->queue[prod_cons->head++];
    prod_cons->head %= QSIZE;

    sb.sem_num = 0;
    sb.sem_op = 1;
    sb.sem_flg = 0;

    if (semop(semid, &sb, 1) == -1)
        exit_sys("semop");

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

    if (val == 99)
        break;

    usleep(rand() % 300000);
}
printf("\n");

shmdt(addr);

return 0;
}

```

```

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

    exit(EXIT_FAILURE);
}

```

/

*-----

read/write lock senkronizasyon nesneleri paylaşılan kaynağa read ve write yapan thread'lerin olduğu durumda kullanılmaktadır. Nesne pthread_rwlock_t türüyle temsil edilmektedir. Nesnenin yaratımı pthread_rwlock_init fonksiyonuyla yapılabileceği gibi, PTHREAD_RWLOCK_INITIALIZER makrosuyla da statik düzeyde yapılabilmektedir. Kilit okuma amaçlı ya da yazma amaçlı elde edilmek istenebilir.

Birden fazla thread kilidi okuma amaçlı kilitleyebilirken, bir thread okuma amaçlı kilitlediye diğer thread'ler o thread kilidi açmadan yazma amaçlı kilitleyememektedir. Benzer biçimde bir thread kilidi yazma amaçlı kilitlediye diğer thread'ler o thread kilidi açmadan kilidi okuma amaçlı kilitleyememektedir. Bu işlemler pthread_rwlock_rdlock ve pthread_rwlock_wrlock fonksiyonlarıyla yapılmaktadır.

Kilidin açılması için pthread_rwlock_unlock fonksiyonu kullanılır.

Blokesiz işlemler için pthread_rwlock_tryrdlock ve pthread_rwlock_trywrlock fonksiyonları da bulundurulmuştur. Yine bu fonksiyonların zaan aşımllı pthread_rwlock_timedrdlock ve pthread_rwlock_timedwrlock isimli biçimleri de vardır. En sonunda nesne pthread_rwlock_destroy fonksiyonuyla yok edilir.

Bu nesnenin de özellikleri vardır. Ancak mevcut durumda tek bir özellik tanımlanmıştır bu da nesnenin prosesler arasında paylaşımı ile ilgilidir. Bu özelliği set etmek için önce pthread_rwlockattr_t türünden nesne alınır. Bu nesne pthread_rwlockattr_init fonksiyonuyla ilkdeğerlenir. Sonra nesneye pthread_rwlockattr_setpshared fonksiyonuyla prosesler arası paylaşım özelliği eklenir. Bu özellik nesnesiyle asıl nesne yaratılır. Sonra da bu özellik nesnesi pthread_rwlockattr_destroy fonksiyonu ile yok edilir.

Aşağıdaki örnekte 4 thread yaratılmıştır. Bu thread'lerin 2 tanesi okuma amaçlı 2 tanesi de yazma amaçlı kritik koda girmek istemektedir. Çıkan sonuca bakarak mekanizmayı gözden geçirebilirsiniz

```
-----*/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include <unistd.h>
#include <pthread.h>

void exit_sys_thread(const char *msg, int err);
void *thread_proc1(void *param);
void *thread_proc2(void *param);
void *thread_proc3(void *param);
void *thread_proc4(void *param);

pthread_rwlock_t g_rwlock = PTHREAD_RWLOCK_INITIALIZER;

int main(void)
{
    int result;
    pthread_t tid1, tid2, tid3, tid4;

    if ((result = pthread_create(&tid1, NULL, thread_proc1, NULL)) != 0)
        exit_sys_thread("pthread_create", result);

    if ((result = pthread_create(&tid2, NULL, thread_proc2, NULL)) != 0)
        exit_sys_thread("pthread_create", result);

    if ((result = pthread_create(&tid3, NULL, thread_proc3, NULL)) != 0)
        exit_sys_thread("pthread_create", result);

    if ((result = pthread_create(&tid4, NULL, thread_proc4, NULL)) != 0)
        exit_sys_thread("pthread_create", result);

    if ((result = pthread_join(tid1, NULL)) != 0)
        exit_sys_thread("pthread_join", result);
```

```

    if ((result = pthread_join(tid2, NULL)) != 0)
        exit_sys_thread("pthread_join", result);

    if ((result = pthread_join(tid3, NULL)) != 0)
        exit_sys_thread("pthread_join", result);

    if ((result = pthread_join(tid4, NULL)) != 0)
        exit_sys_thread("pthread_join", result);

    pthread_rwlock_destroy(&g_rwlock);

    return 0;
}

void exit_sys_thread(const char *msg, int err)
{
    fprintf(stderr, "%s: %s\n", msg, strerror(err));
    exit(EXIT_FAILURE);
}

void *thread_proc1(void *param)
{
    int i;
    int result;
    int seedval;

    seedval = (unsigned int)time(NULL) + 20000;

    for (i = 0; i < 10; ++i) {
        usleep(rand_r(&seedval) % 300000);

        if ((result = pthread_rwlock_rdlock(&g_rwlock)) != 0)
            exit_sys_thread("pthread_rwlock_rdlock", result);

        printf("thread1 ENTERS to critical section for READING...\n");

        usleep(rand_r(&seedval) % 300000);

        printf("thread1 EXITS from critical section...\n");

        if ((result = pthread_rwlock_unlock(&g_rwlock)) != 0)
            exit_sys_thread("pthread_rwlock_rdlock", result);
    }

    return NULL;
}

void *thread_proc2(void *param)
{
    int i;
    int result;
    int seedval;

    seedval = (unsigned int)time(NULL) + 20000;

```

```

for (i = 0; i < 10; ++i) {
    usleep(rand_r(&seedval) % 300000);

    if ((result = pthread_rwlock_rdlock(&g_rwlock)) != 0)
        exit_sys_thread("pthread_rwlock_rdlock", result);

    printf("thread2 ENTERS to critical section for READING...\n");

    usleep(rand_r(&seedval) % 300000);

    printf("thread2 EXITS from  critical section...\n");

    if ((result = pthread_rwlock_unlock(&g_rwlock)) != 0)
        exit_sys_thread("pthread_rwlock_rdlock", result);
}

return NULL;
}

void *thread_proc3(void *param)
{
    int i;
    int result;
    int seedval;

    seedval = (unsigned int)time(NULL) + 20000;

    for (i = 0; i < 10; ++i) {
        usleep(rand_r(&seedval) % 300000);

        if ((result = pthread_rwlock_wrlock(&g_rwlock)) == -1)
            exit_sys_thread("pthread_rwlock_rdlock", result);

        printf("thread3 ENTERS to critical section for WRITING...\n");

        usleep(rand_r(&seedval) % 300000);

        printf("thread3 EXITS from  critical section...\n");

        if ((result = pthread_rwlock_unlock(&g_rwlock)) == -1)
            exit_sys_thread("pthread_rwlock_rdlock", result);
    }

    return NULL;
}

void *thread_proc4(void *param)
{
    int i;
    int result;
    int seedval;

    seedval = (unsigned int)time(NULL) + 20000;

```

```

for (i = 0; i < 10; ++i) {
    usleep(rand_r(&seedval) % 300000);

    if ((result = pthread_rwlock_wrlock(&g_rwlock)) == -1)
        exit_sys_thread("pthread_rwlock_rdlock", result);

    printf("thread4 ENTERS to critical section for WRITING...\n");

    usleep(rand_r(&seedval) % 300000);

    printf("thread4 EXITS from critical section...\n");

    if ((result = pthread_rwlock_unlock(&g_rwlock)) == -1)
        exit_sys_thread("pthread_rwlock_rdlock", result);
}

return NULL;
}

/
*-----
-----
Spinlock (spin dönme anlamına geliyor) meşgul bir döngüde bloke olmadan
kilidin açılması için beklemeye denilmektedir.
Bazı uygulamalarda çok işlemcili ya da çekirdekli sistemlerde kilidi
elde tutan thread eğer bunu makul bir zamanda bırakacaksa
spinlock kullanımı performansı iyileştirmektedir. POSIX sistemlerinde
spinlockpthread_spinlock_t isimli, nesneyle temsil edilmektedir.
Bu nesne pthread_spin_init isimli fonksiyonla ilkdeğerlenir. Kritik kod
pthread_spin_lock ile pthread_spin_unlock çağrılarını
arasına yerleştirilir. İşlem bitince spinlock pthread_spin_destroy
fonksiyonuyla yok edilmektedir. Bir thread pthread_spin_lock
ağrısında kilidi alamazsa bloke olmadan bir döngü içerisinde işlemcinin
set/compare işlemini atomik bir biçimde yapan makine
komutlarıyla kilidi sürekli yoklar (polling). O sırada kilidi elde etmiş
olan thread CPU'yu bırakırsa spinlock'ta bekleyen thread'ler
önemli ölçüde CPU zamanı harcarlar. Bu nedenle spinlock'ların yüksek
öncelikli, thread'ler tarafından uygulanması daha uygun olmaktadır.
Aslında GNU libc kütüphanesinde olduğu gibi pek çok POSIX kütüphanesinde
kilidi almaya çalışmamız mutex, semaphore, read/write lock gibi
nesneler kilit kapalıysa zaten bir süre spin işlemi yapıp ondan sonra
bloke olmaktadır. Yani başka bir deyişle zaten diğer
senkronizasyon nesneleri küçük spinlock görevi de yapmaktadır.
-----*/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <pthread.h>

void exit_sys_thread(const char *msg, int err);
void *thread_proc1(void *param);
void *thread_proc2(void *param);

```

```

pthread_spinlock_t g_spinlock;
int g_count;

int main(void)
{
    int result;
    pthread_t tid1, tid2;

    if ((result = pthread_spin_init(&g_spinlock, 0)) != 0)
        exit_sys_thread("pthread_spin_init", result);

    if ((result = pthread_create(&tid1, NULL, thread_proc1, NULL)) != 0)
        exit_sys_thread("pthread_create", result);

    if ((result = pthread_create(&tid2, NULL, thread_proc2, NULL)) != 0)
        exit_sys_thread("pthread_create", result);

    if ((result = pthread_join(tid1, NULL)) != 0)
        exit_sys_thread("pthread_join", result);

    if ((result = pthread_join(tid2, NULL)) != 0)
        exit_sys_thread("pthread_join", result);

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

    pthread_spin_destroy(&g_spinlock);

    return 0;
}

void exit_sys_thread(const char *msg, int err)
{
    fprintf(stderr, "%s: %s\n", msg, strerror(err));
    exit(EXIT_FAILURE);
}

void *thread_proc1(void *param)
{
    int result;
    int i;

    for (i = 0; i < 100000000; ++i) {
        if ((result = pthread_spin_lock(&g_spinlock)) != 0)
            exit_sys_thread("pthread_spin_lock", result);

        ++g_count;

        if ((result = pthread_spin_unlock(&g_spinlock)) != 0)
            exit_sys_thread("pthread_spin_unlock", result);
    }

    return NULL;
}

```

```
void *thread_proc2(void *param)
```

```
{  
    int result;  
    int i;  
  
    for (i = 0; i < 100000000; ++i) {  
        if ((result = pthread_spin_lock(&g_spinlock)) != 0)  
            exit_sys_thread("pthread_spin_lock", result);  
  
        ++g_count;  
  
        if ((result = pthread_spin_unlock(&g_spinlock)) != 0)  
            exit_sys_thread("pthread_spin_unlock", result);  
    }  
  
    return NULL;  
}
```

```
/
```

```
*-----  
-----
```

Bazı basit işlemler için mutex kullanmak programı yavaşlatabilmektedir.

Örneğin global bir değişkenin 1 artırılması işleminde artırım işlemi birden fazla thread tarafından yapılıyorsa kritik kod işlemi uygulanmalıdır. Öte yandna bu kritk kod işlemi zaman kaybına yol açmaktadır. Bu tür tek makine komutuyla yapılabilecek bazı işlemlerin hiç mutex kullanmadan gerçekleştirilmeleri mümkündür. Intel işlemcilerinde makine komutları bellek operandı alabilmektedir. Makine komutları da atomik olduğu için (yani makine komutu çalışırken threadler arası geçiş olmayacağı için) bu işlemcilerde bellek operandı alan komutlar yoluyla hiç mutex koruması olmadan basit bazı işlemler yapılabilmektedir.

Örneğin Intel işlemcilerinde INC mem makine komutu atomik bir biçimde bellekteki değeri 1 artırabilmektedir. Ancak maalesef çok işlemcili ya da çok çekirdekli sistemlerde bu tür makine komutları farklı işlemci ya da çekirdeklerde aynı bellek bölgesi üzerinde işlem yapılırken geçersiz değerlerin oluşmasına yol açabilmektedir. Intel işlemcileri özellikle bellekteki operand dördün ya da sekizin katlarına hizalanmamışsa bu işlemi tek hamlede yapamaktadır. Intel işlemcilerinde bu tür işlemlerin diğer işlemcilerin müdahalesine kapatılarak yapılabilmesi için komutun başın LOCK önekinin getirilmesi gerekmektedir. ARM işlemcileri Load/Store tarzı çalışmaya sahiptir. Dolayısıyla bellek üzerinde işlem yapan komutlar ARM mimarisinde yoktur. ARM mimarisinde mecburen bellekteki nesne önce CPU yazmacına çekilip orada işleme sokulduktan sonra yeniden belleğe aktarılmaktadır. Fakat yine de ARM işlemcilerinde bu tür basit işlemlerin mutex kontrolü olmadan döngüsel bir yapı ile düşük maliyetli gerçekleştirilmesi de mümkündür.

İşte gcc derleyicisi artırıma, eksiltme ve karşılaştırma gibi basit işlemlerin atomik bir biçimde yapılabilmesi için özel builtin (intrinsic) fonksiyonlar bulundurmıştır. Build-in fonksiyonlar gcc tarafından doğrudan tanınıp bir makro gibi bunlar

için kısa kodlar üretilebilmektedir. gcc'deki atomik builtin fonksiyonlar __sync_xxx biçiminde isimlendirilmiştir. Bunlar builtin olduğu için prototip gereksinimleri yoktur. Ancak daha sonra gcc C++11'deki atomik semantiğini uygulayabilmek için bu builtin fonksiyonları __atomic_xxx ismiyle yenilemiştir. Bu yeni versiyonlar ekstra "memory order" parametresi almaktadır.

Aşağıdak örnekte iki thread aynı global değişkeni hiç mutex ile kritik kod oluşturmadan atomik bir biçimde artırmaktadır. Bu işlem 1000000000 döngü için 25.9 saniye sürmüştür. Aynı programın mutex versiyonu ise 4 dakika 11 saniye sürmüştür.

```
-----*/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <pthread.h>

#define SIZE          1000000000

void exit_sys_thread(const char *msg, int err);
void *thread_proc1(void *param);
void *thread_proc2(void *param);

int g_count;

int main(void)
{
    int result;
    pthread_t tid1, tid2;

    if ((result = pthread_create(&tid1, NULL, thread_proc1, NULL)) != 0)
        exit_sys_thread("pthread_create", result);

    if ((result = pthread_create(&tid2, NULL, thread_proc2, NULL)) != 0)
        exit_sys_thread("pthread_create", result);

    if ((result = pthread_join(tid1, NULL)) != 0)
        exit_sys_thread("pthread_join", result);

    if ((result = pthread_join(tid2, NULL)) != 0)
        exit_sys_thread("pthread_join", result);

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

    return 0;
}

void exit_sys_thread(const char *msg, int err)
{
    fprintf(stderr, "%s: %s\n", msg, strerror(err));
    exit(EXIT_FAILURE);
}
```

```

void *thread_proc1(void *param)
{
    int i;

    for (i = 0; i < SIZE; ++i) {
        __sync_fetch_and_add(&g_count, 1);
    }

    return NULL;
}

```

```

void *thread_proc2(void *param)
{
    int i;

    for (i = 0; i < SIZE; ++i) {
        __sync_fetch_and_add(&g_count, 1);
    }

    return NULL;
}

```

```

/
*-----
-----
    Yukarıdaki programın __atomic_xxx builtin fonksiyonları ile eşdeğeri
    aşağıdadır. __atomic_xxx fonksiyonlarının son
    parametresi olan "memory order" için __ATOMIC_SEQ_CST kullanılmıştır. Bu
    parametre __sync_xxx ile aynı semantiği sağlamaktadır.
    Buradaki memory order parametresi için ilgi dokümanlara
    başvurabilirsiniz.
-----
-----*/

```

```

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <pthread.h>

```

```

#define SIZE          1000000000

```

```

void exit_sys_thread(const char *msg, int err);
void *thread_proc1(void *param);
void *thread_proc2(void *param);

```

```

int g_count;

```

```

int main(void)
{

```

```

    int result;
    pthread_t tid1, tid2;

```

```

    if ((result = pthread_create(&tid1, NULL, thread_proc1, NULL)) != 0)
        exit_sys_thread("pthread_create", result);

```

```

    if ((result = pthread_create(&tid2, NULL, thread_proc2, NULL)) != 0)
        exit_sys_thread("pthread_create", result);

    if ((result = pthread_join(tid1, NULL)) != 0)
        exit_sys_thread("pthread_join", result);

    if ((result = pthread_join(tid2, NULL)) != 0)
        exit_sys_thread("pthread_join", result);

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

    return 0;
}

```

```

void exit_sys_thread(const char *msg, int err)
{
    fprintf(stderr, "%s: %s\n", msg, strerror(err));
    exit(EXIT_FAILURE);
}

```

```

void *thread_proc1(void *param)
{
    int i;

    for (i = 0; i < SIZE; ++i) {
        __atomic_add_fetch(&g_count, 1, __ATOMIC_SEQ_CST);
    }

    return NULL;
}

```

```

void *thread_proc2(void *param)
{
    int i;

    for (i = 0; i < SIZE; ++i) {
        __atomic_add_fetch(&g_count, 1, __ATOMIC_SEQ_CST);
    }

    return NULL;
}

```

```

/
*-----
-----
C'nin 2011 revizyonunda (C11) dile _Atomic isminde bir tür niteleyicisi
(type qualifier) eklenmiştir. Bir nesne bu niteleyici ile
bildirilirse +=, -=, |= gibi işlemler thread güvenli bir biçimde atomik
olarak gerçekleştirilmektedir. Aşağıdaki programda
bu özellik kullanılmıştır. _Atomic anahtar sözcüğü C11 ile geldiği için
gcc derlemesinde -std=c11 ya da -std=c17 seçeneğini
bulundurunuz.
-----
-----*/

```

```

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <pthread.h>

#define SIZE          10000000

void exit_sys_thread(const char *msg, int err);
void *thread_proc1(void *param);
void *thread_proc2(void *param);

_Atomic int g_count;

int main(void)
{
    int result;
    pthread_t tid1, tid2;

    if ((result = pthread_create(&tid1, NULL, thread_proc1, NULL)) != 0)
        exit_sys_thread("pthread_create", result);

    if ((result = pthread_create(&tid2, NULL, thread_proc2, NULL)) != 0)
        exit_sys_thread("pthread_create", result);

    if ((result = pthread_join(tid1, NULL)) != 0)
        exit_sys_thread("pthread_join", result);

    if ((result = pthread_join(tid2, NULL)) != 0)
        exit_sys_thread("pthread_join", result);

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

    return 0;
}

void exit_sys_thread(const char *msg, int err)
{
    fprintf(stderr, "%s: %s\n", msg, strerror(err));
    exit(EXIT_FAILURE);
}

void *thread_proc1(void *param)
{
    int i;

    for (i = 0; i < SIZE; ++i) {
        ++g_count;
    }

    return NULL;
}

void *thread_proc2(void *param)
{
    int i;

```

```

    for (i = 0; i < SIZE; ++i) {
        ++g_count;
    }

    return NULL;
}

/
*-----
-----
C++'ta <atomic> başlık dosyası içerisindeki template atomic sınıfı
atomik işlemler yapmaktadır. Aşağıda bu sınıf kullanılarak
C++ örneği verilmiştir.
-----
-----*/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <pthread.h>
#include <atomic>

#define SIZE          10000000

void exit_sys_thread(const char *msg, int err);
void *thread_proc1(void *param);
void *thread_proc2(void *param);

using namespace std;

atomic<int> g_count;

int main(void)
{
    int result;
    pthread_t tid1, tid2;

    if ((result = pthread_create(&tid1, NULL, thread_proc1, NULL)) != 0)
        exit_sys_thread("pthread_create", result);

    if ((result = pthread_create(&tid2, NULL, thread_proc2, NULL)) != 0)
        exit_sys_thread("pthread_create", result);

    if ((result = pthread_join(tid1, NULL)) != 0)
        exit_sys_thread("pthread_join", result);

    if ((result = pthread_join(tid2, NULL)) != 0)
        exit_sys_thread("pthread_join", result);

    printf("%d\n", (int)g_count);

    return 0;
}

```

```
void exit_sys_thread(const char *msg, int err)
{
    fprintf(stderr, "%s: %s\n", msg, strerror(err));
    exit(EXIT_FAILURE);
}
```

```
void *thread_proc1(void *param)
{
    int i;

    for (i = 0; i < SIZE; ++i) {
        ++g_count;
    }

    return NULL;
}
```

```
void *thread_proc2(void *param)
{
    int i;

    for (i = 0; i < SIZE; ++i) {
        ++g_count;
    }

    return NULL;
}
```

/

*-----

C'nin stdio dosya fonksiyonları POSIX sistemlerinde (fakat standart C'de değil) zaten thread güvenlidir. Dolayısıyla iki thread aynı dosyaya yazma ya da okuma yaparken programcının fwrite, fread gibi fonksiyonları senkronize etmesine gerek yoktur. Aşağıdaki programda iki thread aynı dosyaya fwrite fonksiyonlarıyla 0'dan SIZE'a kadar int değerleri yazmaktadır. Thread'lerden biri tek sayıları yazarken diğeri çift sayıları yazmaktadır. İşlemin sonunda yazma işleminde bir sorun olup olmadığı test edilmiştir.

Her ne kadar standart C'nin stdio fonksiyonları POSIX sistemlerinde thread güvenli olsa da yine de birden fazla art arda çağrılarda senkronizasyon sorunları ortaya çıkabilir. Örneğin iki thread fseek uyguladıktan sonra fwrite işlemi yapsa thread'lerden biri istemediği bir bölgeye yazma yapabilir. Bu durumda bir mutex koruması düşünülebilir. Ancak POSIX sistemlerinde içsel olarak böyle bir mutex kontrolü düşünülmüştür. flockfile ile bir stdio dosyasını kilitlersek funlockfile uygulayana kadar başka bir thread bu dosya üzerinde işlem yapmak istediğinde bloke beklemektedir.

-----*/

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

```

#include <pthread.h>

#define SIZE          1000000

FILE *g_f;

void exit_sys_thread(const char *msg, int err);
void *thread_proc1(void *param);
void *thread_proc2(void *param);
int check(void);

int main(void)
{
    int result;
    pthread_t tid1, tid2;
    int i;

    if ((g_f = fopen("test.dat", "w+")) == NULL) {
        perror("fopen");
        exit(EXIT_FAILURE);
    }

    if ((result = pthread_create(&tid1, NULL, thread_proc1, NULL)) != 0)
        exit_sys_thread("pthread_create", result);

    if ((result = pthread_create(&tid2, NULL, thread_proc2, NULL)) != 0)
        exit_sys_thread("pthread_create", result);

    if ((result = pthread_join(tid1, NULL)) != 0)
        exit_sys_thread("pthread_join", result);

    if ((result = pthread_join(tid2, NULL)) != 0)
        exit_sys_thread("pthread_join", result);

    printf(check() ? "success...\n" : "failed...\n");

    return 0;
}

void exit_sys_thread(const char *msg, int err)
{
    fprintf(stderr, "%s: %s\n", msg, strerror(err));
    exit(EXIT_FAILURE);
}

void *thread_proc1(void *param)
{
    int i;

    for (i = 0; i < SIZE; i += 2)
        if (fwrite(&i, sizeof(int), 1, g_f) != 1) {
            perror("fwrite");
            exit(EXIT_FAILURE);
        }
}

```

```

    return NULL;
}

void *thread_proc2(void *param)
{
    int i;

    for (i = 1; i < SIZE; i += 2)
        if (fwrite(&i, sizeof(int), 1, g_f) != 1) {
            perror("fwrite");
            exit(EXIT_FAILURE);
        }
    return NULL;
}

```

```

int check(void)
{
    char flags[SIZE] = {0};
    int val;
    int i;

    rewind(g_f);
    while (fread(&val, sizeof(int), 1, g_f) == 1) {
        if (flags[val])
            return 0;
        flags[val] = 1;
    }

    if (ferror(g_f)) {
        perror("fread");
        exit(EXIT_FAILURE);
    }

    for (i = 0; i < SIZE; ++i)
        if (!flags[i])
            return 0;

    return 1;
}

```

```

/
*-----
-----
    Bir proses birtakım thread'ler yarattıktan sonra fork işlemi yaparsa alt
    proses her zaman tek bir thread ile çalışmaya devam
    eder. O da üst prosesin fork işleminin yapıldığı thread'tir.
-----
-----*/

```

```

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

```



```

#include <sys/wait.h>
#include <pthread.h>

void exit_sys(const char *msg);
void exit_sys_thread(const char *msg, int err);
void *thread_proc1(void *param);
void *thread_proc2(void *param);

int main(void)
{
    int result;
    pthread_t tid1, tid2;
    pid_t pid;

    if ((result = pthread_create(&tid1, NULL, thread_proc1, NULL)) != 0)
        exit_sys_thread("pthread_create", result);

    if ((result = pthread_create(&tid2, NULL, thread_proc2, NULL)) != 0)
        exit_sys_thread("pthread_create", result);

    if ((pid = fork()) == -1)
        exit_sys("fork");

    if (pid != 0)
        printf("parent process\n");
    else
    {
        printf("child process\n");
        pthread_exit(NULL);
    }

    if ((result = pthread_join(tid1, NULL)) != 0)
        exit_sys_thread("pthread_join", result);

    if ((result = pthread_join(tid2, NULL)) != 0)
        exit_sys_thread("pthread_join", result);

    if (waitpid(pid, NULL, 0) == -1)
        exit_sys("waitpid");

    return 0;
}

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

    exit(EXIT_FAILURE);
}

void exit_sys_thread(const char *msg, int err)
{
    fprintf(stderr, "%s: %s\n", msg, strerror(err));
    exit(EXIT_FAILURE);
}

```

```

void *thread_proc1(void *param)
{
    int i;

    for (i = 0; i < 20; ++i) {
        printf("thread-1: %d\n", i);
        sleep(1);
    }

    return NULL;
}

```

```

void *thread_proc2(void *param)
{
    int i;

    for (i = 0; i < 20; ++i) {
        printf("thread-2: %d\n", i);
        sleep(1);
    }

    return NULL;
}

```

```

/
*-----
-----
    Bir C programının exit fonksiyonuyla ya da main fonksiyonun bitmesiyle
    sonlandığını biliyorsunuz. Ancak programın ana thread'i
    pthread_exit fonksiyonuyla sonlandırılabilir. Bu durumda proses onun son
    thread'i sonlandığında sonlandırılır.

    Aşağıdaki programda fork işlemi ana thread'te değil başka bir thread'te
    uygulanmıştır. Alt proses tek bir thread'le çalışmaya
    başlayacaktır. O da fork fonksiyonunu uygulayan thread'tir. Programın
    çıktısını inceleyiniz.
-----
-----*/

```

```

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/wait.h>
#include <pthread.h>

void exit_sys(const char *msg);
void exit_sys_thread(const char *msg, int err);
void *thread_proc1(void *param);
void *thread_proc2(void *param);

int main(void)
{
    int result;

```

```

pthread_t tid1, tid2;

if ((result = pthread_create(&tid1, NULL, thread_proc1, NULL)) != 0)
    exit_sys_thread("pthread_create", result);

if ((result = pthread_create(&tid2, NULL, thread_proc2, NULL)) != 0)
    exit_sys_thread("pthread_create", result);

if ((result = pthread_join(tid1, NULL)) != 0)
    exit_sys_thread("pthread_join", result);

if ((result = pthread_join(tid2, NULL)) != 0)
    exit_sys_thread("pthread_join", result);

return 0;
}

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

    exit(EXIT_FAILURE);
}

void exit_sys_thread(const char *msg, int err)
{
    fprintf(stderr, "%s: %s\n", msg, strerror(err));
    exit(EXIT_FAILURE);
}

void *thread_proc1(void *param)
{
    int i;
    pid_t pid;

    for (i = 0; i < 20; ++i) {
        printf("thread-1: %d\n", i);
        if (i == 10) {
            if ((pid = fork()) == -1)
                exit_sys("fork");
        }
        sleep(1);
    }

    if (pid != 0 && waitpid(pid, NULL, 0) == -1)
        exit_sys("waitpid");

    return NULL;
}

void *thread_proc2(void *param)
{
    int i;

    for (i = 0; i < 20; ++i) {

```

```

        printf("thread-2: %d\n", i);
        sleep(1);
    }

    return NULL;
}

```

```

/
*-----
-----
Aslında çok thread'li proseslerde fork işlemi organizasyonel bakımdan
karmaşık birtakım sonuçlar doğurmaktadır.
Bu nedenle çok thread'li uygulamalarda ya fork yapılmamalı ya da fork
yapılacaksa alt proseste hemen exec uygulanmalıdır.

Bir proses birtakım thread'ler yarattıktan sonra herhangi bir thread'te
exec işlemi uygulanırsa programın bütün bellek alanı
boşaltılacağına göre exec yapılan prog tek bir thread'le çalışmasına
devam edecektir. exec işlemi başarı durumunda exec işlemini uygulayan
thread dışında prosesin bütün thread'lerini otomatik olarak yok
etmektedir. Yani exec işlemi sonucunda exec edilen kod her zaman
tek bir thread'le çalışmaya başlar.
-----
-----*/

```

```

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/wait.h>
#include <pthread.h>

void exit_sys(const char *msg);
void exit_sys_thread(const char *msg, int err);
void *thread_proc1(void *param);
void *thread_proc2(void *param);

int main(void)
{
    int result;
    pthread_t tid1, tid2;

    if ((result = pthread_create(&tid1, NULL, thread_proc1, NULL)) != 0)
        exit_sys_thread("pthread_create", result);

    if ((result = pthread_create(&tid2, NULL, thread_proc2, NULL)) != 0)
        exit_sys_thread("pthread_create", result);

    if ((result = pthread_join(tid1, NULL)) != 0)
        exit_sys_thread("pthread_join", result);

    if ((result = pthread_join(tid2, NULL)) != 0)
        exit_sys_thread("pthread_join", result);

    return 0;
}

```

```

}

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

    exit(EXIT_FAILURE);
}

void exit_sys_thread(const char *msg, int err)
{
    fprintf(stderr, "%s: %s\n", msg, strerror(err));
    exit(EXIT_FAILURE);
}

void *thread_proc1(void *param)
{
    int i;
    pid_t pid;

    for (i = 0; i < 20; ++i) {
        printf("thread-1: %d\n", i);
        if (i == 10) {
            if ((pid = fork()) == -1)
                exit_sys("fork");
            if (pid == 0)
                if (execlp("ls", "ls", "-l", (char *)NULL) == -1)
                    exit_sys("execlp");
            /* unreachable code */
        }
        sleep(1);
    }

    if (pid != 0 && waitpid(pid, NULL, 0) == -1)
        exit_sys("waitpid");

    return NULL;
}

void *thread_proc2(void *param)
{
    int i;

    for (i = 0; i < 20; ++i) {
        printf("thread-2: %d\n", i);
        sleep(1);
    }

    return NULL;
}

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

```

Thread'li programların exec işlemi dışında fork yapması organizasyonel birtakım problemler doğurabilmektedir. Üst proseste birtakım senkronizasyon nesneleri varsa alt proseste bu senkronizasyon nesneleri aktratıldığı için organizasyonel sorunlar oluşabilmektedir. Çünkü üst proses fork işlemi sırasına bazı senkronizasyon nesnelerini kilitlemiş olabilir.

Bu senkronizasyon nesneleri kilitli bir biçimde alt proseste aktarıldığında alt proseste kilitlenmelere (deadlocks) yol açabilmektedir.

Bu nedenle üst prosesteki bu senkronizasyon nesnelerinin kararlı bir durumda alt proseste aktarılabilmesi için pthread_atfork isimli bir POSIX

fonksiyonu düşünülmüştür. Bu fonksiyonun "prepare", "parent" ve "child" biçiminde üç fonksiyon göstericisi parametresi vardır. prepare ile belirtilen

fonksiyon fork işleminin hemen öncesinde öncesinde üst proses tarafından otomatik çalıştırılır. Programcının bu fonksiyonda tüm senkronizasyon nesnelerini

kilitlemesi uygun olur. parent isimli fonksiyon fork sonrasında üst proses tarafından child isimli fonksiyon ise fork işlemi sonrasında alt proses

tarafından çalıştırılır. Her iki fonksiyonda da programcının bu kilitlediği senkronizasyon nesnelerini açması beklenmektedir. Böylece hiç olmazsa

alt proseste fork sonrasında bütün senkronizasyon nesnelerinin açık bir biçimde olacağı garanti edilmiş olur. Aşağıdaki örnekte bu semantik uygulanmıştır.

-----*/

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

void exit_sys(const char *msg);
void exit_sys_thread(const char *msg, int err);
void *thread_proc1(void *param);
void *thread_proc2(void *param);

void prepare(void);
void parent(void);
void child(void);

pthread_mutex_t g_mutex1 = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t g_mutex2 = PTHREAD_MUTEX_INITIALIZER;

int main(void)
{
    int result;
    pthread_t tid1, tid2;

    if ((result = pthread_atfork(prepare, parent, child)) != 0)
```

```

        exit_sys_thread("pthread_atfork", result);

    if ((result = pthread_create(&tid1, NULL, thread_proc1, NULL)) != 0)
        exit_sys_thread("pthread_create", result);

    if ((result = pthread_create(&tid2, NULL, thread_proc2, NULL)) != 0)
        exit_sys_thread("pthread_create", result);

    if ((result = pthread_join(tid1, NULL)) != 0)
        exit_sys_thread("pthread_join", result);

    if ((result = pthread_join(tid2, NULL)) != 0)
        exit_sys_thread("pthread_join", result);

    return 0;
}

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

    exit(EXIT_FAILURE);
}

void exit_sys_thread(const char *msg, int err)
{
    fprintf(stderr, "%s: %s\n", msg, strerror(err));
    exit(EXIT_FAILURE);
}

void *thread_proc1(void *param)
{
    int i;
    pid_t pid;

    for (i = 0; i < 20; ++i) {
        printf("thread-1: %d\n", i);
        if (i == 10) {
            if ((pid = fork()) == -1)
                exit_sys("fork");
            /* unreachable code */
        }
        sleep(1);
    }

    if (pid != 0 && waitpid(pid, NULL, 0) == -1)
        exit_sys("waitpid");

    return NULL;
}

void *thread_proc2(void *param)
{
    int i;

```

```

    for (i = 0; i < 20; ++i) {
        printf("thread-2: %d\n", i);
        sleep(1);
    }

    return NULL;
}

void prepare(void)
{
    int result;

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

    if ((result = pthread_mutex_lock(&g_mutex1)) != 0)
        exit_sys_thread("pthread_mutex_lock", result);

    if ((result = pthread_mutex_lock(&g_mutex2)) != 0)
        exit_sys_thread("pthread_mutex_lock", result);
}

void parent(void)
{
    int result;

    printf("parent\n");

    if ((result = pthread_mutex_unlock(&g_mutex1)) != 0)
        exit_sys_thread("pthread_mutex_unlock", result);

    if ((result = pthread_mutex_unlock(&g_mutex2)) != 0)
        exit_sys_thread("pthread_mutex_lock", result);
}

void child(void)
{
    int result;

    printf("child\n");

    if ((result = pthread_mutex_unlock(&g_mutex1)) != 0)
        exit_sys_thread("pthread_mutex_unlock", result);

    if ((result = pthread_mutex_unlock(&g_mutex2)) != 0)
        exit_sys_thread("pthread_mutex_lock", result);
}

```

/

*-----

Global ya da static yerel nesne kullanan fonksiyonlar thread güvenli değildir. Bunları prototiplerini değiştirmeden thread güvenli yapabilmek için thread'e özgü global değişken etkisinin yaratılması gerekir. İşte işletim sistemleri bu tür

thread güvenli kodların yazılabilmesi için thread'e özgü global alanlar oluşturmaktadır. Bu alanlara Windows sistemlerinde "Thread Local Storage", UNIX/Linux sistemlerinde "Thread Specific Data" denilmektedir. Naisl her thread'in bir stack'i varsa bir de TSD alanı vardır. Bu alan slotlardan oluşmaktadır. Slotların numaraları vardır. Bu numaralar pthread_key_t türüyle temsil edilmiştir. Programcı önce pthread_key_create fonksiyonu ile bir slot (ya da anahtar) yaratır. Slotlar thread'ler için ayrı ayrı yaratılmamaktadır. Bir slot yaratıldığında tüm thread'ler için (daha önce yaratılmış ve daha sonra yaratılacak olanlar da dahil olmak üzere) yaratılmaktadır. Slotlara void bir adres yerleştirilip geri alınabilmektedir. Bunun için pthread_setspecific ve pthread_getspecific fonksiyonları kullanılmaktadır. Slot yaratıldığında içerisinde NULL adres olduğu garanti edilmiştir. Programcı tipik olarak malloc ile heap'te bir alan tahsis edip onun adresini slota yerleştirir. Tabii tek bilgi aşağıdaki örnekte olduğu gibi bir adres gibi de doğrudan slotlara yerleştirilebilmektedir. Böylece her thread aynı anahtarı kullansa da aslında kendi thread'inin slotuna erişir. En sonunda yaratılmış olan slot (anahtar) pthread_key_delete fonksiyonu ile yok edilebilmektedir.

Aşağıdaki örnekte rand ve srand fonksiyonları TSD kullanılarak thread güvenli hale getirilmiştir

```
-----*/

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

void exit_sys_thread(const char *msg, int err);
void *thread_proc1(void *param);
void *thread_proc2(void *param);

pthread_key_t g_key;

int myrand(void)
{
    void *val;
    unsigned seed;
    int result;

    if ((val = pthread_getspecific(g_key)) == NULL) {
        if ((result = pthread_setspecific(g_key, (void *)1)) != 0)
            exit_sys_thread("pthread_setspecific", result);
        seed = 1;
    }
    else
        seed = (unsigned) val;

    seed = seed * 1103515245 + 12345;
    if ((result = pthread_setspecific(g_key, (void *)seed)) != 0)
        exit_sys_thread("pthread_setspecific", result);
}
```

```

    return seed / 65536 % 32768;
}

void myrand(unsigned seed)
{
    int result;

    if ((result = pthread_setspecific(g_key, (void *)seed)) != 0)
        exit_sys_thread("pthread_setspecific", result);
}

int main(void)
{
    int result;
    pthread_t tid1, tid2;

    if ((result = pthread_key_create(&g_key, NULL)) != 0)
        exit_sys_thread("pthread_key_create", result);

    if ((result = pthread_create(&tid1, NULL, thread_proc1, NULL)) != 0)
        exit_sys_thread("pthread_create", result);

    if ((result = pthread_create(&tid2, NULL, thread_proc2, NULL)) != 0)
        exit_sys_thread("pthread_create", result);

    if ((result = pthread_join(tid1, NULL)) != 0)
        exit_sys_thread("pthread_join", result);

    if ((result = pthread_join(tid2, NULL)) != 0)
        exit_sys_thread("pthread_join", result);

    if ((result = pthread_key_delete(g_key)) != 0)
        exit_sys_thread("pthread_key_delete", result);

    return 0;
}

void exit_sys_thread(const char *msg, int err)
{
    fprintf(stderr, "%s: %s\n", msg, strerror(err));
    exit(EXIT_FAILURE);
}

void *thread_proc1(void *param)
{
    int i;
    int val;

    for (i = 0; i < 10; ++i) {
        val = myrand();
        printf("Thread-1: %d\n", val % 100);
    }
    printf("\n");
}

```

```

        return NULL;
    }

void *thread_proc2(void *param)
{
    int i;
    int val;

    for (i = 0; i < 10; ++i) {
        val = myrand();
        printf("Thread-2: %d\n", val % 100);
    }
    printf("\n");

    return NULL;
}

```

```

/
*-----
-----
Aşağıda getenv fonksiyonu TSD kullanılarak thread güvenli hale
getirilmiştir. Read/Write lock nesnesi başka bir thread çevre
değişkenlerini
değiştirirken ya da çevre değişkenlerine ekleme yaparken bozulmayı
engellemek için kullanılmıştır. Tabii bunu değiştirecek thread de aynı
nesneyle
write amaçlı kilidi elde etmeye çalışmalıdır. Thread ilk kez slota
yerleştirme yapacağı zaman bellek tahsis edip onun adresini slota
yerleştirmiştir.
Diğer çağrılarda artık bu tahsis ettiği alanı kullanmaktadır. Bu alanın
otomatik free getirilmesi için pthread_key_create fonksiyonunda
"destructor" parametresi
kullanılmıştır. Destructor parametresiyle verilen fonksiyon
pthread_key_delete tarafından değil thread sonlanırken çağrılmaktadır.
Ancak proses sonlanırken bu destructor fonksiyonları çağrılmaz.
Destructor fonksiyonun çağırılması için slotta NULL dışında bir değerin
bulunuyor olması gerekir.
(Yani destructor parametresi girildiği halde slota yerleştirme
yapılmamışsa bu fonksiyon çağrılmamaktadır.) Destructor fonksiyonu
thread pthread_cancel
fonksiyonuyla başka bir thread tarafından sonlandırılırken de
çağrılmaktadır. Eğer thread birden fazla slot için destructor
fonksiyonuna sahipse
bu durumda her slot için destructor fonksiyonu çağrılır. Ancak bunların
sırası hakkında bir belirlemede bulunulmamıştır.
-----
-----*/

```

```

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <pthread.h>

```

```

#define MAX_ENV    4096

```

```

void exit_sys(const char *msg);
void exit_sys_thread(const char *msg, int err);
void *thread_proc1(void *param);
void *thread_proc2(void *param);
char *mygetenv(const char *env);
void destructor(void *param);

extern char **environ;
pthread_key_t g_envkey;
pthread_rwlock_t g_rwlock = PTHREAD_RWLOCK_INITIALIZER;

char *mygetenv(const char *env)
{
    char *pval = NULL;
    void *pv;
    int result;
    size_t len;
    int i;

    if ((pval = (char *)pthread_getspecific(g_envkey)) == NULL) {
        if ((pv = malloc(MAX_ENV)) == NULL)
            return NULL;
        if ((result = pthread_setspecific(g_envkey, pv)) != 0)
            exit_sys_thread("pthread_setspecific", result);
        pval = (char *)pv;
    }

    if ((result = pthread_rwlock_rdlock(&g_rwlock)) != 0)
        exit_sys_thread("pthread_rwlock_rdlock", result);
    for (i = 0; environ[i] != NULL; ++i) {
        len = strlen(env);
        if (!strncmp(env, environ[i], len)) {
            if (environ[i][len] != '=')
                break;
            strcpy(pval, &environ[i][len + 1]);
            break;
        }
    }
    if ((result = pthread_rwlock_unlock(&g_rwlock)) != 0)
        exit_sys_thread("pthread_rwlock_unlock", result);

    return pval;
}

int main(void)
{
    int result;
    pthread_t tid1, tid2;

    if ((result = pthread_key_create(&g_envkey, destructor)) != 0)
        exit_sys_thread("pthread_key_create", result);

    if ((result = pthread_create(&tid1, NULL, thread_proc1, NULL)) != 0)
        exit_sys_thread("pthread_create", result);

```

```

    if ((result = pthread_create(&tid2, NULL, thread_proc2, NULL)) != 0)
        exit_sys_thread("pthread_create", result);

    if ((result = pthread_join(tid1, NULL)) != 0)
        exit_sys_thread("pthread_join", result);

    if ((result = pthread_join(tid2, NULL)) != 0)
        exit_sys_thread("pthread_join", result);

    if ((result = pthread_key_delete(g_envkey)) != 0)
        exit_sys_thread("pthread_key_delete", result);

    return 0;
}

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

    exit(EXIT_FAILURE);
}

void exit_sys_thread(const char *msg, int err)
{
    fprintf(stderr, "%s: %s\n", msg, strerror(err));
    exit(EXIT_FAILURE);
}

void *thread_proc1(void *param)
{
    char *val;

    if ((val = mygetenv("PATH")) == NULL)
        exit_sys("mygetenv");
    puts(val);

    return NULL;
}

void *thread_proc2(void *param)
{
    char *val;

    if ((val = mygetenv("HOME")) == NULL)
        exit_sys("mygetenv");
    puts(val);

    return NULL;
}

void destructor(void *param)
{
    free(param);
}

```

```
/
*-----
-----
C11 ile birlikte ve C++11 ile birlikte C ve C++ dillerine thread_local
isimli bir yer belirleyicisi de eklenmiştir.
Bir global değişkeni ya da static yerel değişkeni biz bu belirleyici ile
bildirdiğimizde derleyici onu thread specific alanda yaratır.
Dolayısıyla bu değişken thread'e özgü global ya da static yerel değişken
durumunda olur. Bu sayede biz Windows ve Linux farklı sistemlerde
thread'e özgü alanları farklı kodlarla oluşturmak zorunda kalmayız.

Aşağıdaki kodu g++ derleyicisi ile -std=c++11 seçeneği ile derleyiniz.
-----
-----*/
```

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

thread_local int g_i;

void exit_sys_thread(const char *msg, int err);
void *thread_proc1(void *param);
void *thread_proc2(void *param);

int main(void)
{
    int result;
    pthread_t tid1, tid2;

    if ((result = pthread_create(&tid1, NULL, thread_proc1, NULL)) != 0)
        exit_sys_thread("pthread_create", result);

    if ((result = pthread_create(&tid2, NULL, thread_proc2, NULL)) != 0)
        exit_sys_thread("pthread_create", result);

    if ((result = pthread_join(tid1, NULL)) != 0)
        exit_sys_thread("pthread_join", result);

    if ((result = pthread_join(tid2, NULL)) != 0)
        exit_sys_thread("pthread_join", result);

    return 0;
}

void exit_sys_thread(const char *msg, int err)
{
    fprintf(stderr, "%s: %s\n", msg, strerror(err));
    exit(EXIT_FAILURE);
}

void *thread_proc1(void *param)
```

```

{
    for (g_i = 0; g_i < 10; ++g_i) {
        printf("thread1: %d\n", g_i);
        sleep(1);
    }

    return NULL;
}

```

```

void *thread_proc2(void *param)
{
    for (g_i = 0; g_i < 10; ++g_i) {
        printf("thread1: %d\n", g_i);
        sleep(1);
    }

    return NULL;
}

```

```

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

Birden fazla thread aynı kod üzerinde ilerlerken belli bir kod
paraçasının yalnızca tek bir thread tarafından çalıştırılmasını
isteyebiliriz. Özellikle TSD uygulamalarında bu tür isteklerle
karşılaşılmaktadır. Bu işlemi pthread_once isimli fonksiyon yapar.
Bu fonksiyona biz bir fonksiyon adresi veririz. Peç çok thread
pthread_once fonksiyonunu görse de bu fonksiyon bizim verdiğimiz
fonksiyonun
yalnızca bir kez ilk thread akışı tarafından çağrılmasını sağlar.

Aşağıdaki örnekte iki thread foo fonksiyonunu çağırılmıştır. foo içerisinde
iki thread de pthread_once çağrısına girmiştir. Ancak
thread'lerden yalnızca bir tanesi pthread_once fonksiyonunda
belirttiğimiz bar fonksiyonunu çağıracaktır.
-----
-----*/

```

```

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <pthread.h>

```

```

void exit_sys_thread(const char *msg, int err);
void *thread_proc1(void *param);
void *thread_proc2(void *param);
void foo(void);
void bar(void);

```

```

pthread_once_t g_once = PTHREAD_ONCE_INIT;

```

```

int main(void)
{
    int result;
    pthread_t tid1, tid2;

```

```

    if ((result = pthread_create(&tid1, NULL, thread_proc1, NULL)) != 0)
        exit_sys_thread("pthread_create", result);

    if ((result = pthread_create(&tid2, NULL, thread_proc2, NULL)) != 0)
        exit_sys_thread("pthread_create", result);

    if ((result = pthread_join(tid1, NULL)) != 0)
        exit_sys_thread("pthread_join", result);

    if ((result = pthread_join(tid2, NULL)) != 0)
        exit_sys_thread("pthread_join", result);

    return 0;
}

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

    exit(EXIT_FAILURE);
}

void exit_sys_thread(const char *msg, int err)
{
    fprintf(stderr, "%s: %s\n", msg, strerror(err));
    exit(EXIT_FAILURE);
}

void *thread_proc1(void *param)
{
    foo();

    return NULL;
}

void *thread_proc2(void *param)
{
    foo();

    return NULL;
}

void foo(void)
{
    int result;

    printf("foo called...\n");

    if ((result = pthread_once(&g_once, bar)) != 0)
        exit_sys_thread("pthread_once", result);
}

void bar(void)
{

```



```
    printf("bar only called once...\n");
}
```

```
/
```

```
*-----
```

pthread_once fonksiyonu genellikle TSD kullanan programlarda karşımıza çıkmaktadır. Programcı TSD slotunu işin başında thread'leri yaratmadan pthread_key_create ile yaratabilir. Ancak bu durumda söz konusu thread güvenli fonksiyon hiç çağrılmazsa bu slot boşuna yaratılmış olacaktır. İşte bu durumu engellemek için thread güvenli hale getirilen fonksiyon ilk kez çağrıldığında TSD slotu yaratılır.

Aşağıdaki örnekte myrand ya da mysrand fonksiyonları ilk kez herhangi bir thread tarafından çağrıldığında TSD slotu pthread_once fonksiyonu sayesinde yaratılmıştır.

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

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

```
void mysrand(unsigned seed);
int myrand(void);
void thread_once_proc(void);
```

```
void exit_sys_thread(const char *msg, int err);
void *thread_proc1(void *param);
void *thread_proc2(void *param);
```

```
pthread_key_t g_key;
pthread_once_t g_once = PTHREAD_ONCE_INIT;
```

```
int myrand(void)
{
    void *val;
    unsigned seed;
    int result;

    if ((result = pthread_once(&g_once, thread_once_proc)) != 0)
        exit_sys_thread("pthread_once", result);

    if ((val = pthread_getspecific(g_key)) == NULL) {
        if ((result = pthread_setspecific(g_key, (void *)1)) != 0)
            exit_sys_thread("pthread_setspecific", result);
        seed = 1;
    }
    else
        seed = (unsigned) val;

    seed = seed * 1103515245 + 12345;
```

```

    if ((result = pthread_setspecific(g_key, (void *)seed)) != 0)
        exit_sys_thread("pthread_setspecific", result);

    return seed / 65536 % 32768;
}

void mysrand(unsigned seed)
{
    int result;

    if ((result = pthread_once(&g_once, thread_once_proc)) != 0)
        exit_sys_thread("pthread_once", result);

    if ((result = pthread_setspecific(g_key, (void *)seed)) != 0)
        exit_sys_thread("pthread_setspecific", result);
}

void thread_once_proc(void)
{
    int result;

    if ((result = pthread_key_create(&g_key, NULL)) != 0)
        exit_sys_thread("pthread_key_create", result);
}

int main(void)
{
    int result;
    pthread_t tid1, tid2;

    if ((result = pthread_create(&tid1, NULL, thread_proc1, NULL)) != 0)
        exit_sys_thread("pthread_create", result);

    if ((result = pthread_create(&tid2, NULL, thread_proc2, NULL)) != 0)
        exit_sys_thread("pthread_create", result);

    if ((result = pthread_join(tid1, NULL)) != 0)
        exit_sys_thread("pthread_join", result);

    if ((result = pthread_join(tid2, NULL)) != 0)
        exit_sys_thread("pthread_join", result);

    return 0;
}

void exit_sys_thread(const char *msg, int err)
{
    fprintf(stderr, "%s: %s\n", msg, strerror(err));
    exit(EXIT_FAILURE);
}

void *thread_proc1(void *param)
{
    int i;
    int val;

```

```

    for (i = 0; i < 10; ++i) {
        val = myrand();
        printf("Thread-1: %d\n", val % 100);
    }
    printf("\n");

    return NULL;
}

```

```

void *thread_proc2(void *param)
{
    int i;
    int val;

    for (i = 0; i < 10; ++i) {
        val = myrand();
        printf("Thread-2: %d\n", val % 100);
    }
    printf("\n");

    return NULL;
}

```

/

*-----

Aslında Linux işletim sisteminde proses ve thread yaratımı benzer biçimde yapılmaktadır. Thread'lerin de birer task_struct yapısı vardır. Bu task_struct içerisinde tüm prosesler ve prosesin thread'leri bağlı listelerle tutulmuştur. task_struct içerisindeki alanlar ve anlamları şöyledir:

- parent ve real_parent: Üst prosesin task_struct adresi
- children: Prosesin alt proses listesindeki ilk alt prosesin task_struct adresi
- sibling: Prosesin alt prosesleri dolaşılırken kullanılan sonraki kardeş prosesin task_struct adresi
- thread_group: Aynı prosesin içerisindeki thread'lerin task_struct listesi. Her task_struct'ın thread_group göstericisi prosesin sonraki bir thread'inin task_struct yapısını gösterir.
- group_leader: Prosesin ana thread'inin task_struct adresi

Linux işletim sistemi her zaman o anda çalışmakta olan kodun task_struct adresini biliyor durumdadır. Kernel içerisindeki current makrosu her zaman o anda çalışmakta olan thread'in task_struct adresini vermektedir. Örneğin bir thread read fonksiyonuyla bir dosyadna okuma yapacak olsa o thread'in task_struct yapısından hareketle prosesin dosya betimleyici tablosuna erişilmektedir. Her task_struct yapısının ayrı bir pid değeri vardır. Ancak POSIX'in getpid fonksiyonu çağrıldığında o anda çalışmakta olan thread'in task_struct pid'si verilmez. prosesin ana thread'inin

pid'si verilir. Çünkü POSIX standartlarında thread'lerin pid'leri yoktur. Yalnızca proseslerin pid'leri vardır. Linux'a özgü gettid fonksiyonu aslında o thread'in pid'sini vermektedir. Sistemde yaratılabilecek toplam task_struct sayısı bellidir. Bu sayı /proc/sys/kernel/threads-max dosyasında belirtilmektedir.

Linux'un sys_clone isimli sistem fonksiyonu aslında proses ve thread yaratımının yapıldığı en genel sistem fonksiyonudur. fork fonksiyonun özel bir clone çağırması olduğunu belirtelim. pthread kütüphanesi de sonuçta thread'i bu clone sistem fonksiyonuyla yaratmaktadır. clone sistem fonksiyonu bir kütüphane fonksiyonu olarak da (tabii ki POSIX fonksiyonu değil) Linux sistemlerinde bulunmaktadır.

Linux'un çizelgeleyici alt sistemi prosesleri çizelgelemez. Thread'leri çizelgeler. Dolayısıyla aslında çizelgeleyici task_struct'lardan hareketle context switch yapmaktadır.

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

POSIX standartlarına göre thread çizelgelemesi thread'lerin çizelgeleme politikalarına (scheduling policies) bağlıdır.

POSIX 4 çizelgeleme politikası tanımlamıştır. Ancak işletim sistemlerinin daha fazla politikaya sahip olabileceği belirtilmiştir. SCHED_FIFO ve SCHED_RR çizelgeleme politikalarına "gerçek zamanlı (real time)" çizelgeleme politikaları denilmektedir. Thread'lerin default çizelgeleme politikaları SCHED_OTHER biçimindedir. SCHED_OTHER çizelgeleme politikası POSIX standartlarında tamamen işletim sistemini yazarların isteğine bırakılmıştır. Yani pratikte karşılaştığımız çizelgeleme politikası hep SCHED_OTHER biçimindedir. Bu da işletim sisteminden işletim sistemine farklılıklar göstermektedir.

POSIX standartlarında bloke olmamış thread'lerin bir run kuyruğunda bekledikleri varsayılmaktadır. Çizelgeleyici alt sistemin görevi ise buradan uygun thread'i seçip CPU'ya atamaktır. SCHED_RR ve SCHED_FIFO çizelgeleme politikalarına sahip olan thread'ler her zaman SCHED_OTHER çizelgeleme politikasına sahip thread'lerden üstündür. Yani çizelgeleyici alt sistem her zaman SCHED_FIFO ya da SCHED_RR politikasına sahip thread'lere öncelik vermektedir. Başka bir deyişle bir SCHED_OTHER thread'inin çalışabilmesi için run kuyruğunda SCHED_FIFO ve SCHED_RR politikalarına sahip thread'lerin olmaması gerekir.

SCHED_FIFO ve SCHED_RR politikalarına sahip thread'lerin birer statik öncelik derecesi vardır. Bu statik öncelik derecesi [1-99] arasında bir değere sahiptir. Sistem her zaman run kuyruğunda SCHED_FIFO ve SCHED_RR thread'lerinin en yüksek öncelikli olanını

olarak CPU'ya verir. Ancak eğer aynı statik önceliğe sahip birden fazla SCHED_FIFO ve SCHED_RR prosesi varsa kuyrukta önce olan thread CPU'ya verilmektedir. Bu thread bir quanta süresi kadar çalıştırılır. Quanta bittiğinde eğer bu thread SCHED_FIFO politikasına sahip ise kuyruğun başına SCHED_RR politikasına sahipse kuyruğun sonuna yerleştirilir. Yeniden run kuyruğuna bakılarak en yüksek öncelikli kuyruğun önündeki thread alınarak CPU'ya atanır. İşlemler böyle devam ettirilir. Örneğin run kuyruğunda şu thread'ler bulunuyor olsun:

FIFO (50), OTHER, RR (50), RR(50) OTHER

Burada en yüksek statik öncelikli kuyruğun önündeki thread ilk thread'tir. Sistem bunu CPU atar. Bir quanta çalıştırır. Quanta bitince kuyruğun başına yerleştirir. Bu durumda yine onu CPU'ya atar. Yani bu durumda bu thread sonlanana kadar ya da bloke olup run kuyruğundan çıkana kadar çalışacaktır. Şimdi bu thread'in bloke olduğunu düşünelim. Run kuyruğu şöyle olacaktır:

OTHER, RR(50), RR(50), OTHER

Burada ikinci sırada bulunan RR (50) thread'i CPU'ya verilecek ve 1 quanta çalıştırılacaktır. Quanta bitince kuyruğun sonuna yerleştirileceği için artık üçüncü sıradaki RR(50) thread'i CPU'ya verilecektir. Böylece bu iki thread bloke olana kadar döngüsel çizelgeleme yöntemiyle çalıştırılacaktır. Görüldüğü gibi SCHED_OTHER thread'ler bunlar varken çalışma fırsatı bulamayacaktır.

Bir SCHED_RR ya da SCHED_FIFO thread çalışırken bloke çözülmüş olan daha yüksek öncelikli bir thread run kuyruğuna yerleştirilirse hemen context switch yapılarak çalışma bu thread'e verilmektedir. Zaten "real time" çizelgelemeden beklenen budur. (POSIX sistemleri real time sistemler değildir.)

Genel olarak SCHED_FIFO ya da SCHED_RR thread'lerin IO yoğun thread'ler olması uygundur. Aksi takdirde diğer diğer thread'ler çalışmaya fırsat bulamayacaklardır.

Birden fazla CPU ya da çekirdek olduğu durumda bu CPU'ya ya da çekirdeklerin aynı run kuyruğuna bakarak seçim yaptıkları düşünülmelidir. Örneğin:

FIFO (50), OTHER, RR (50), RR(50) OTHER

Bu durumda iki CPU varsa, sistem FIFO(50) thread'ini bunlardan birine RR(50) thread'ini diğerine atar. Diğer CPU'da RR(50) thread'leri döngüsel çizelgeleme yoluyla çalışacaktır.

SCHED_OTHER politikasının işlevi POSIX standartlarında işletim sisteminin tanımlamasına bırakılmıştır. Ancak bu politikaların SCHED_FIFO ya da SCHED_RR ile özdeş olabilmesine de izin verilmiştir. Linux işletim sisteminde SCHED_OTHER thread'ler için çizelgeleme algoritması üç kere değiştirilmiştir.

Her ne kadar POSIX standartları SCHED_OTHER thread'ler konusunda belirlemeyi işletim sistemini yazanlara bırakmış olsa da bu thread'lerin bir dinamik önceliğinin olduğu ve bu önceliğin bu thread'lerin kullanacakları quanta sürelerini belirlemeye yaradığından bahsetmiştir. Gerçekten de Linux ve diğer POSIX sistemlerinde SCHED_OTHER politikasına sahip thread'lerin birer dinamik öncelği vardır. Bu dinamik önceliğe "nice değeri" de denilmektedir. Linux'ta SSCHED_OTHER thread'lerinin dinamik önceliği [1-40] arasındadır.

Ancak yüksek öncelik düşük bir quanta süresi anlamına gelmektedir. Bu durumda en çok quanta kullanan dinamik öncelik 1'dir.

Linux sistemlerinde SCHED_OTHER politikasına sahip thread'lerin çizelgeleme algoritmaları üç kere değiştirilmiştir. İlk algoritma tipik olandır. Buna özel bir isim verilmemiştir. İkinci algoritma Linux 2.6 versiyonlarında kullanılmaya başlanmıştır. Buna "O(1) Çizelgelemesi" denilmektedir. Nihayet 2.6.23 ile birlikte şu anda kullanılmakta olan "CFS (completely Fair Scheduling)" algoritmasına geçilmiştir.

SCHED_OTHER thread'lerin dinamik öncelikleri onların kaç mili saniye quanta süresi kullanacaklarını dolaylı olarak belirtmektedir. Yani sistemde n tane SCHED_OTHER thread var ise bunlar döngüsel çizelgelenmekle birlikte hepsi aynı sürede quanta kullanmak zorunda değildir.

Buradakış detay Linux'un versiyonundan versiyonuna değişebilmektedir.

Genel olarak Linux çekirdeğininin pek çok vesiyonunda SCHED_OTHER thread'lerin kullanacakları quanta süreleri onların dinamik öncelikleri ile ilişkilendirilmiştir. Eun kuyruğundaki thread'lerin hepsinin quanta süresi bitmeden yeniden doldurma yapılmamaktadır. Linux'un pek çok versiyonunda timer kesmesi 10 ms'ye kurulmuştur. (Bilgisayarlar hızlanınca artık 1 ms'ye ye kurmaya başladılar.) Her timer kesmesi oluştuğunda task_struct içerisindeki quanta sayacı olan counter elemanı 1 eksiltilir. Bu eleman 0'a düştüğünde thread tüm quanta süresini kullanmış olur.

İşte run kuyruğundaki tüm thread'lerin coub-nter değerleri 0'a düştüğünde yeniden bunlara dinamik öncelik temelinde yeni değerler atanmaktadır.

CFS algoritmasına kadar Linux çekirdekleri genel olarak run kuyruğunda counter değeri en yüksek olan SCHED_OTHER thread'i CPU'ya atamaktadır. Böylece o zamana kadar az CPU kullanan thread'lere öncelik verilmiş olmaktadır. Linux'un CFS sistemine kadar SCHED_OTHER therad'lerin dinamik öncelikleri

default durumda 20 idi. Bu da 200 ms. bir quanta süresine karşılık gelmektedir. Mademki dinamik öncelik en fazla 1 olabilir. Bu durumda $(40 - 1 = 39)$

thread en fazla 390 ms. quanta süresine sahip olabilir.

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

ssched_setscheduler isimli POSIX fonksiyonu bir prosesin id değerini alarak onun çizelgeleme politikasını ve statik ya da dinamik önceliğini değiştirmekte kullanılır. Fonksiyon struct sched_param türünden bir yapı nesnesini de parametre olarak almaktadır. Bu yapının sched_priority isminde tek bir elemanı vardır. Bu eleman SCHED_FIFO ve SCHED_RR için statik önceliği, SCHED_OTHER için dinamik önceliği belirtir. POSIX standartlarına göre bir prosesin çizelgeleme politikası bu fonksiyonla değiştirildiğinde prosesin tüm thread'lerinin çizelgeleme politikaları değiştirilmiş olur. Ancak Linux bunu desteklememektedir. Linux'ta bu fonksiyon çağrıldığında prosesin yalnızca ana thread'inin çizelgeleme politikası değiştirilmiş olur. Prosesin diğer thread'lerinin çizelgeleme politikası Linux'ta bu fonksiyonun pid parametresi yerine gettid fonksiyonuyla elde edilen değerin verilmesiyle yapılabilmektedir. Tabii prosesin çizelgeleme politikasının değiştirilebilmesi ancak root proses için mümkündür. Fonksiyonun pid parametresi yerine 0 da girilebilir. Bu durumda zaten getpid() anlaşılmaktadır.

```
-----*/

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

void exit_sys(const char *msg);

int main(void)
{
    struct sched_param sparam;

    sparam.sched_priority = 50;

    if (sched_setscheduler(getpid(), SCHED_FIFO, &sparam) == -1)
        exit_sys("sched_setscheduler");

    printf("Ok\n");

    return 0;
}

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

    exit(EXIT_FAILURE);
}

/
*-----

Belli bir thread'in (ana thread de dahil olmak üzere) çizelgeleme politikası ve thread önceliği pthread_getschedparam POSIX fonksiyonu ile alınıp pthread_setschedparam POSIX fonksiyonuyla set edilebilir. Tabii bu işlemin yapılabilmesi için prosesin yine root önceliğinde
```

olması gerekir.

```
-----*/

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

void exit_sys_thread(const char *msg, int err);
void *thread_proc(void *param);

int main(void)
{
    int result;
    pthread_t tid;
    struct sched_param param;

    if ((result = pthread_create(&tid, NULL, thread_proc, NULL)) != 0)
        exit_sys_thread("pthread_create", result);

    param.sched_priority = 30;

    if ((result = pthread_setschedparam(tid, SCHED_FIFO, &param)) != 0)
        exit_sys_thread("pthread_setschedparam", result);

    if ((result = pthread_join(tid, NULL)) != 0)
        exit_sys_thread("pthread_join", result);

    return 0;
}

void exit_sys_thread(const char *msg, int err)
{
    fprintf(stderr, "%s: %s\n", msg, strerror(err));
    exit(EXIT_FAILURE);
}

void *thread_proc(void *param)
{
    int i;

    for (i = 0; i < 10; ++i) {
        printf("%d ", i);
        fflush(stdout);
        sleep(1);
    }
    printf("\n");

    return NULL;
}
```



```
/
*-----
-----
gettid fonksiyonu belli bir süredir GNU kütüphanesinde bulunmamaktadır.
Bu Linux'ta bir sistem fonksiyonudur. Bu nedenle
bu fonksiyon syscall fonksiyonu ile çağrılabilir. Aşağıdaki örnekte ana
thread'in ve yaratılan thread'in task_struct pid değerleri
ekrana yazdırılmıştır. Tabii ana thread'in task_struct pid değeri
aslında getpid fonksiyonuyla da elde edilebilir.

Aslında pthread_setschedparam fonksiyonu yerine Linux sistemlerinde
thread'in task_struct pid değeri elde edilerek
sched_setscheduler fonksiyonu da kullanılabilir.
-----
-----*/
```

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <unistd.h>
#include <pthread.h>
#include <sys/syscall.h>

void exit_sys_thread(const char *msg, int err);
void *thread_proc(void *param);

int main(void)
{
    int result;
    pthread_t tid;
    struct sched_param param;
    long task_pid;

    if ((task_pid = syscall(SYS_gettid)) == -1) {           // equivalent
        getpid()
        perror("syscall");
        exit(EXIT_FAILURE);
    }

    printf("Main thread task_struct pid: %ld\n", task_pid);

    if ((result = pthread_create(&tid, NULL, thread_proc, NULL)) != 0)
        exit_sys_thread("pthread_create", result);

    param.sched_priority = 30;

    if ((result = pthread_setschedparam(tid, SCHED_FIFO, &param)) != 0)
        exit_sys_thread("pthread_setschedparam", result);

    if ((result = pthread_join(tid, NULL)) != 0)
        exit_sys_thread("pthread_join", result);

    return 0;
}
```

```

void exit_sys_thread(const char *msg, int err)
{
    fprintf(stderr, "%s: %s\n", msg, strerror(err));
    exit(EXIT_FAILURE);
}

void *thread_proc(void *param)
{
    int i;
    long task_pid;

    if ((task_pid = syscall(SYS_gettid)) == -1) {
        perror("syscall");
        exit(EXIT_FAILURE);
    }

    printf("Thread task_struct pid: %ld\n", task_pid);

    for (i = 0; i < 100; ++i) {
        printf("%d ", i);
        fflush(stdout);
        sleep(1);
    }
    printf("\n");

    return NULL;
}

```

/

*-----

Proceslere ilişkin bilgiler bilindiği gibi ps komutuyla elde edilmektedir. ps komutunda -o parametresi istenilen sütunları ayarlamak için kullanılmaktadır. -T thread bilgilerini de verir. Örneğin:

```
ps -a -T -o pid,pri,tid,cmd,policy
```

Burada aaynı kullanıcının tüm terminallerde çalışan prosesleri (-a) ve threadleri (-T) görüntülenmiştir. Görüntülemeye pid, pri (öncelik), tid (task_pid), cmd (komut), ve policy (çizelgeleme politikası) sütunları kullanılmıştır. Örneğin:

PID	PRI	TID	CMD	POL
13480	19	13480	sudo ./sample	TS
13482	19	13482	./sample	TS
13482	70	13483	./sample	FF
13566	19	13566	ps -a -T -o pid,pri,tid,cmd	TS

Burada ./sample satırlarında pid değeri tid değerine eşit olan satırdaki ./sample prosesin ana thread'ini diğeri ise sonradan yaratılan thread'i belirtmektedir. PRI sütunu SCHED_OTHER prosesler için dinamik önceliği belirtmektedir.

SCHED_FIFO ve SCHED_RR proseslerin statik öncelikleri maksimum dinamik önceliğe (40) eklenerek gösterilmektedir. Örneğin 13483 task_pid değerine sahip olan thread'in çizelgeleme politikası SCHED_FIFO biçimindedir. Tatik önceliği 30'dur.

pstree isimli kabul komutu prosesleri (ve onların thread'lerini) ve alt prosesleri bir ağaç biçiminde göstermektedir. Tipik olarak -p <pid> seçeneğiyle kullanılmaktadır.

Komut satırında bir prosesi belli bir çizelgeleme politikası ve statik/dinamik önceliklerle çalıştırabilmek için chrt komutu kullanılmaktadır. Komutta sırasıyla önce çizelgeleme politikası sonra öncelik sonra da çalıştırılacak komut belirtilir. Örneğin:

```
sudo chrt --fifo 30 ls
```

Burada ls programı SCHED_FIFO çizelgeleme politikasıyla 30 statik önceliğe sahip olacak biçimde çalıştırılmaktadır. Burada çalıştırılan program (örnekte ls) eğer thread yaratırsa bu thread'ler de SCHED_FIFO politikasına sahip olacaktır. Çünkü Linux'ta (ve POSIX standartlarında da böyle) thread'lerin çizelgeleme politikaları onu yaratan thread'ten alınmaktadır.

chrt komutu ile çalışmakta olan programların da çizelgeleme politikaları değiştirilebilir. Bunun için program ismi yerine prosesin id'si belirtilmektedir. Örneğin:

```
sudo chrt -f -p 30 13482
```

```
-----*/
/
*-----
Aslında bir thread'in çizelgeleme politikası ve statik/dinamik önceliği thread yaratıldıktan sonra değil, thread yaratılırken thread attribute bilgisiyle de değiştirilebilmektedir. Bunun için pthread_attr_setpolicy, pthread_attr_setschedparam fonksiyonu ile bu özelliğin set edilmesi gerekir. Default durumda thread'lerin politikaları ve öncelikleri onu yaratan thread'ten alınmaktadır. Bunun olmaması için ayrıca programcının pthread_attr_setinheritsched ile PTHREAD_EXPLICIT_SCHED ayarlama yapması gerekmektedir.
-----*/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <unistd.h>
#include <pthread.h>

void exit_sys_thread(const char *msg, int err);
```

```

void *thread_proc(void *param);

int main(void)
{
    int result;
    pthread_t tid;
    pthread_attr_t attr;
    struct sched_param param;

    if ((result = pthread_attr_init(&attr)) != 0)
        exit_sys_thread("pthread_attr_init", result);

    if ((result = pthread_attr_setschedpolicy(&attr, SCHED_FIFO)) != 0)
        exit_sys_thread("pthread_attr_setschedpolicy", result);

    param.sched_priority = 30;
    if ((result = pthread_attr_setschedparam(&attr, &param)) != 0)
        exit_sys_thread("pthread_attr_setschedparam", result);

    if ((result = pthread_attr_setinheritsched(&attr,
        PTHREAD_EXPLICIT_SCHED)) != 0)
        exit_sys_thread("pthread_attr_setinheritsched", result);

    if ((result = pthread_create(&tid, &attr, thread_proc, NULL)) != 0)
        exit_sys_thread("pthread_create", result);

    pthread_attr_destroy(&attr);

    if ((result = pthread_join(tid, NULL)) != 0)
        exit_sys_thread("pthread_join", result);

    return 0;
}

void exit_sys_thread(const char *msg, int err)
{
    fprintf(stderr, "%s: %s\n", msg, strerror(err));
    exit(EXIT_FAILURE);
}

void *thread_proc(void *param)
{
    int i;

    for (i = 0; ; ++i) {
        printf("%d ", i);
        fflush(stdout);
        sleep(1);
    }
    printf("\n");

    return NULL;
}

```

/

*-----

SCHED_OTHER politikasına sahip prosesin dinamik önceliği nice isimli POSIX fonksiyonuyla değiştirilebilmektedir. nice fonksiyonu mevcut dinamik önceliğe artırım ya da eksiltim yapar. Artırmak quantayı düşürmek eksiltmek quantayı yükseltmek anlamına gelmektedir. (Örneğin 1 nice değeri 10 ms. gibi bir etkiye yol açtığı varsayılabilir.) POSIX standartlarına göre nice fonksiyonu prosesin tüm thread'leri üzerinde etkili olmaktadır. Ancak Linux'ta yalnızca ana thread üzerinde etkili olur. Linux işletim sisteminde bir thread çizelgeleme politikasını ve öncelik derecesini onu yaratan thread'ten almaktadır. Yine nice ile dinamik önceliğin yükseltilmesi için prosesin root önceliğinde olması gerekmektedir.

-----*/

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

```
void exit_sys_thread(const char *msg, int err);
void *thread_proc(void *param);
```

```
int main(void)
{
    int result;
    pthread_t tid;

    if ((result = pthread_create(&tid, NULL, thread_proc, NULL)) != 0)
        exit_sys_thread("pthread_create", result);

    if (nice(-10) == -1) {
        perror("nice");
        exit(EXIT_FAILURE);
    }

    if ((result = pthread_join(tid, NULL)) != 0)
        exit_sys_thread("pthread_join", result);

    return 0;
}
```

```
void exit_sys_thread(const char *msg, int err)
{
    fprintf(stderr, "%s: %s\n", msg, strerror(err));
    exit(EXIT_FAILURE);
}
```

```
void *thread_proc(void *param)
{
    int i;
```

```

    for (i = 0; ; ++i) {
        printf("%d ", i);
        fflush(stdout);
        sleep(1);
    }
    printf("\n");

    return NULL;
}

```

/

*-----

Komut satırından bir programı SCHED_OTHER olarak dinamik önceliği -20, +19 arasında değiştirerek çalıştırabilmek için nice komutu kullanılmaktadır. Örneğin:

```
sudo nice -n -10 ./sample
```

Ayrıca bir de renice isimli bir komut vardır. Ancak bu komut zaten çalışmakta olan bir prosesin nice değerini değiştirmek için kullanılır.

-----*/

/

*-----

Linux sistemlerinde nice fonksiyonunun POSIX uyumlu olmadığını belirtmiştik. Bu fonksiyon Linuc sistemlerinde yalnızca ana thread'in dinamik önceliğini değiştiriyordu. Pekiyi Linux'ta herhangi bir SCHED_OTHER politikasına sahip thread'in dinamik önceliği nasıl değiştirilmektedir? Bunun için aslında ssched_setscheduler fonksiyonu gettid ile kullanılabilir. Ya da benzer biçimde pthread_setschedparam fonksiyonu da kullanılabilir. Fakat sıfır bu iş için getpriority ve setpriority POSIX fonksiyonları da kullanılmaktadır.

-----*/

```

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <unistd.h>
#include <pthread.h>
#include <sys/syscall.h>
#include <sys/resource.h>

```

```

void exit_sys_thread(const char *msg, int err);
void *thread_proc(void *param);

```

```

int main(void)
{

```

```

int result;
pthread_t tid;

if ((result = pthread_create(&tid, NULL, thread_proc, NULL)) != 0)
    exit_sys_thread("pthread_create", result);

if ((result = pthread_join(tid, NULL)) != 0)
    exit_sys_thread("pthread_join", result);

return 0;
}

```

```

void exit_sys_thread(const char *msg, int err)
{
    fprintf(stderr, "%s: %s\n", msg, strerror(err));
    exit(EXIT_FAILURE);
}

```

```

void *thread_proc(void *param)
{
    int i;
    long task_pid;

    if ((task_pid = syscall(SYS_gettid)) == -1) {
        perror("syscall");
        exit(EXIT_FAILURE);
    }

    if (setpriority(PRIO_PROCESS, task_pid, -10) == -1) {
        perror("setpriority");
        exit(EXIT_FAILURE);
    }

    for (i = 0; ; ++i) {
        printf("%d ", i);
        fflush(stdout);
        sleep(1);
    }
    printf("\n");

    return NULL;
}

```

/

*-----

Belli thread'lerin belli CPU ya da çekirdeklere atanması paralel programlama uygulamalarında gerekebilmektedir. Bir thread'in hangi CPU ya da çekirdeklere atanabileceği "affinity mask" denilen bir özellik ile belirlenmektedir. Default durumda thread'ler mevcut tüm CPU ya da çekirdeklere atanabilirler. Biz thread'leri farklı CPU ya da çekirdeklere atayarak onların aynı anda birlikte çalışmasını sağlayabiliriz.

Linux sistemlerinde affinity işlemleri için 4 fonksiyon kullanılmaktadır. sched_getaffinity bir prosesin (prosesin ana thread'inin) affinity değerini verir, sched_setaffinity fonksiyonu ise bunu set etmemizi sağlar. Aşağıdaki örnekte 2 çekirdekli bir sistemde prosesin ana thread'inin hangi CPU ya da çekirdeklere atanabilecekleri gösterilmiştir. Default durumda işletim sistemi thread'leri tüm CPU ya da çekirdeklere atayabilmektedir.

```
-----*/

#define _GNU_SOURCE

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

void exit_sys(const char *msg);

int main(void)
{
    cpu_set_t set;
    int i;

    if (sched_getaffinity(getpid(), sizeof(set), &set) == -1)
        exit_sys("sched_getaffinity");

    for (i = 0; i < 2; ++i)
        printf("%d-CPU: %s\n", i, CPU_ISSET(i, &set) ? "YES" : "NO");

    return 0;
}

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

    exit(EXIT_FAILURE);
}

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

sched_setaffinity fonksiyonu belli bir prosesin (prosesin ana thread'inin) belli CPU ya da çekirdeklerde çalışmasını sağlamak için kullanılmaktadır. Bunun için prosesin etkin kullanıcı id'sinin affinity değişikliği yapılacak prosesin gerçek ya da etkin kullanıcı id'si ile aynı olması ya da root önceliğine sahip olması ya da CAP_SYS_NICE yeteneğine sahip olması gerekmektedir. Biz bu fonksiyon ile bir prosesin belli bir thread'inin affinity'sini de değiştirebiliriz. Tabii bunun için ilgili thread'in task_struct pid değerinin gettid sistem fonksiyonuyla elde edilmesi gerekmektedir.
```


Programı çalıştırıp aşağıdaki gibi ps komutuyla prosesin ana thread'inin 1 numaralı CPU'ya atanıp atanmadığını kontrol ediniz:

```
ps -a -o pid,psr,cmd
```

```
-----*/

#define _GNU_SOURCE

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

void exit_sys(const char *msg);

int main(void)
{
    cpu_set_t set;

    CPU_ZERO(&set);
    CPU_SET(1, &set);

    if ((sched_setaffinity(getpid(), sizeof(set), &set)) == -1)
        exit_sys("sched_setaffinity");

    for (;;)
        ;

    return 0;
}

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

    exit(EXIT_FAILURE);
}

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

Belli bir thread'e affinty uygulamak için yukarıda da belirtildiği gibi
gettid ile sched_setaffinity fonksiyonu kullanılabilir.
Ancak Linux'a özgü bir biçimde zaten pthread_setaffinity_np ve
pthread_getaffinity_np isimli fonksiyonlar da bulundurulmuştur.
Bu fonksiyonlar doğrudan thread id'lerle çalışmaktadır. Aşağıdaki
programda iki thread farklı CPU ya da çekirdeklere atanarak bunlara CPU
yoğun
iş yaptırılmıştır.

-----*/

#define _GNU_SOURCE
```

```

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sched.h>
#include <pthread.h>

void exit_sys_thread(const char *msg, int err);
void *thread_proc1(void *param);
void *thread_proc2(void *param);

int main(void)
{
    int result;
    pthread_t tid1, tid2;
    cpu_set_t set;

    if ((result = pthread_create(&tid1, NULL, thread_proc1, NULL)) != 0)
        exit_sys_thread("pthread_create", result);

    if ((result = pthread_create(&tid2, NULL, thread_proc2, NULL)) != 0)
        exit_sys_thread("pthread_create", result);

    CPU_ZERO(&set);
    CPU_SET(0, &set);

    if ((result = pthread_setaffinity_np(tid1, sizeof(set), &set)) != 0)
        exit_sys_thread("pthread_setaffinity", result);

    CPU_ZERO(&set);
    CPU_SET(1, &set);

    if ((result = pthread_setaffinity_np(tid1, sizeof(set), &set)) != 0)
        exit_sys_thread("pthread_setaffinity", result);

    if ((result = pthread_join(tid1, NULL)) != 0)
        exit_sys_thread("pthread_join", result);

    if ((result = pthread_join(tid2, NULL)) != 0)
        exit_sys_thread("pthread_join", result);

    return 0;
}

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

    exit(EXIT_FAILURE);
}

void exit_sys_thread(const char *msg, int err)
{
    fprintf(stderr, "%s: %s\n", msg, strerror(err));
    exit(EXIT_FAILURE);
}

```

```

void *thread_proc1(void *param)
{
    int i;

    for (i = 0; i < 1000000000; ++i)
        ;

    return NULL;
}

```

```

void *thread_proc2(void *param)
{
    int i;

    for (i = 0; i < 1000000000; ++i)
        ;

    return NULL;
}

```

```

/
*-----
-----
Sinyal oluştuğunda çağrılacak fonksiyonu (signal handler) set etmek için
iki POSIX fonksiyonu kullanılmaktadır: signal ve sigaction.
signal fonksiyonu eskidir ve maalesef semantik konusunda problemleri
vardır. Bu nedenle signal fonksiyonunu değişik sistemler
değişik biçimde gerçekleştirmişleridir. sigaction fonksiyonunda bu
semantik kusurlar ortadan kaldırılmıştır.

Aşağıda SIGINT sinyali oluştuğunda (Klavyeden Ctrl + C tuşuna
basıldığında SIGINT sinyali oluşmaktadır) çağrılacak
fonksiyon set edilmiştir.
-----
-----*/

```

```

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

```

```

void exit_sys(const char *msg);
void sigint_handler(int sno);

```

```

int main(void)
{
    int i;

    if (signal(SIGINT, sigint_handler) == SIG_ERR)
        exit_sys("signal");

    for (i = 0; i < 20; ++i) {
        printf("%d\n", i);
        sleep(1);
    }
}

```

```

    }

    return 0;
}

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

    exit(EXIT_FAILURE);
}

void sigint_handler(int sno)
{
    printf("SIGINT handler running...\n");
}

/
*-----
-----
    signal fonksiyonun ikinci parametresine SIG_DFL özel değeri geçilirse
    sinyal fonksiyonu default duruma çekilir. Yani artık
    sinyal oluştuğunda "default action" uygulanır. Bu parametreye SIG_IGN
    özel değeri geçilirse sinyal işletim sistemi tarafından
    görmeden gelinir (ignore edilir).
-----
-----*/

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

void exit_sys(const char *msg);

int main(void)
{
    int i;

    if (signal(SIGINT, SIG_IGN) == SIG_ERR)
        exit_sys("signal");

    for (i = 0; i < 20; ++i) {
        printf("%d\n", i);
        sleep(1);
    }

    return 0;
}

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

    exit(EXIT_FAILURE);
}

```

```

}

/
*-----
-----
kill isimli POSIX fonksiyonu (ismi yanlış uydurulmuştur) bir prosese ya
da proses grubuna sinyal göndermek için kullanılmaktadır.
Eğer birinci parametresi olan pid "> 0" ise spesifik prosese sinyal
gönderir. Eğer bu parametre "= 0" ise kendi proses grubundaki
tüm proseslere sinyal gönderir. Eğer bu parametre "< 0" ise abs(pid)
değerine sahip proses grubun tüm proseslerine sinyal gönderir.
Eğer bu parametre "= -1" ise sinyal gönderebileceği tüm proseslere
sinyal gönderir.

kill fonksiyonu ile sinyal gönderebilmek için gönderen prosesin gerçek
ya da etkin kullanıcı id'sinin gönderilen prosesin gerçek ya da
saved set kullanıcı id'sine eşit olması gerekir. Tabii root prosesi her
prosese sinyal gönderebilir. (Ya da Linux'ta CAP_KILL yeteneğine
sahip prosesler de tüm proseslere sinyal gönderebilirler.)

Aşağıda proc2 programı SIGUSR1 numaları sinyali proc1'e göndermek için
yazılmıştır. proc1'i önce çalıştırınız. ps -a ile
onun pid değerini bakıp o değerler proc2'yi çalıştırınız.
-----
-----*/

/* proc1.c */

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

void exit_sys(const char *msg);
void sigusr1_handler(int sno);

int main(void)
{
    int i;

    if (signal(SIGUSR1, sigusr1_handler) == SIG_ERR)
        exit_sys("signal");

    for (i = 0;; ++i) {
        printf("%d\n", i);
        sleep(1);
    }

    return 0;
}

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

```

```

        exit(EXIT_FAILURE);
    }

void sigusr1_handler(int sno)
{
    printf("SIGUSR1 occurred...\n");
}

/* proc2.c */

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

void exit_sys(const char *msg);

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

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

    pid = (pid_t)strtol(argv[1], NULL, 10);

    if (kill(pid, SIGUSR1) == -1)
        exit_sys("kill");

    return 0;
}

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

    exit(EXIT_FAILURE);
}

```

*-----

Komut satırından da bir prosese sinyal göndermek için kill isimli komut kullanılmaktadır. kill komutunun basit kullanımı şöyledir:

```
kill -<numara> <process-id'ler>
```

Örneğin:

```
kill -15 15711
```

Numara yerine sinyallerin sembolik sabit isimleri de kullanılabilir. Bunun için SIGXXX isimli sinyal -XXX biçiminde belirtilmelidir.

Örneğin:

```
kill -TERM 15711
```

Eğer kill komutunda sinyal numarası belirtilmezse -TERM (yani -15) belirtilmiş gibi işlem görülür. Bu durumda bir prosesi garantili sonlandırmak için -KILL ya da -9 seçenekleri kullanılmalıdır.

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

Bir prosesi sonlandırmak için sıklıkla iki sinyal kullanılmaktadır: SIGTERM (15) ve SIGKILL (9). SIGTERM sinyali için sinyal fonksiyonu set edilebilir fakat SIGKILL için edilemez. Benzer biçimde SIGTERM sinyali SIG_IGN ile ignore edilebilir ancak SIGKILL sinyali ignore edilemez. Yani SIGTERM ile sonlandırma garanti değildir ancak SIGKILL ile sonlandırma garantidir. Örneğin:

```
kill -KILL 15711
```

Aşağıdaki programı bir terminalde çalıştırığ diğeri ile önce SIGTERM sonra da SIGKILL sinyallerini göndermeyi deneyiniz

```
-----*/  
  
#include <stdio.h>  
#include <stdlib.h>  
#include <unistd.h>  
#include <signal.h>  
  
void exit_sys(const char *msg);  
void sigterm_handler(int sno);  
  
int main(void)  
{  
    int i;  
  
    if (signal(SIGTERM, sigterm_handler) == SIG_ERR)  
        exit_sys("signal");  
  
    for (i = 0;; ++i) {  
        printf("%d\n", i);  
        sleep(1);  
    }  
  
    return 0;  
}  
  
void exit_sys(const char *msg)  
{  
    perror(msg);
```

```

    exit(EXIT_FAILURE);
}

void sigterm_handler(int sno)
{
    printf("SIGTERM occurred...\n");
}

```

```

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

```

signal fonksiyonu ile sinyal set etmek biraz problemlidir. Problem bu fonksiyonun davranışının UNIX türevi sistemlerde farklı olabilmesinden kaynaklanmaktadır. Eski UNIX sistemleri signal fonksiyonu ile sinyal set edildiğinde sinyal fonksiyonu çalışırken aynı sinyalin oluşmasına izin veriyordu. Böylece aynı sinyal fonksiyonu iç içe çalışabiliyordu. Ancak BSD sistemleri sinyal oluştuğunda sinyal fonksiyonu çalıştırılırken bu sinyali bloke edip bekletmektedir. Ta ki sinyal fonksiyonu işini bitiren kadar. Böylece iç içe geçme olmamaktadır. Yine eski UNIX sistemlerinde signal fonksiyonu ile set yapıldığında sinyal fonksiyonu çağrılır çağrılmaz o numaralı sinyal otomatik default'a çekiliyordu (yani set edilmemiş duruma getiriliyordu). Ancak BSD sistemleri bunu yapmıyordu. Yine bu fonksiyon ile set yapıldığında yavaş sistem fonksiyonlarının otomatik restart edilip edilmeyeceği sistemler arasında farklılık gösterebiliyordu.

Linux işletim sisteminde signal ve sigaction isimli iki sistem fonksiyonu vardır. signal sistem fonksiyonu Sistem 5 semantiği ile sinyal fonksiyonunu set etmektedir. sigaction modern olandır. Fakat glibc kütüphanesindeki signal fonksiyonu glibc 2.0'dan sonra signal sistem fonksiyonunu değil sigaction sistem fonksiyonunu çağırarak yazılmış durumdadır.

Linux işletim sistemi signal isimli sistem fonksiyonunda eski UNIX System-5 semantiğini uygulamaktadır. Yani:

- Sinyal fonksiyonu çalıştırılırken sinyal default'a çekilir.
- Aynı sinyal bloke edilmez. Sinyal fonksiyonu iç içe çalışabilir.
- Yavaş sistem fonksiyonları restart edilmez.

glibc 2.0 öncesinde Linux'taki signal POSIX fonksiyonu signal sistem fonksiyonunu çağırıldığı için Sistem 5 semantiğini uyguluyordu. Fakat glibc 2.0'dan sonra Linux'taki signal POSIX fonksiyonu sigaction fonksiyonu çağırılarak BSD semantiğini uygulamaktadır. Yani:

- Sinyal fonksiyonu çalışırken sinyal default'a çekilmez.
- Aynı sinyal otomatik bloke edilir.
- Yavaş sistem fonksiyonları otomatik restart edilir.

Maalesef sistem 5 semantiği default'a çekme yüzünden kusurlu tasarlanmıştır. Çünkü aşağıdaki gibi bir işlemde her zaman program çökebilmektedir:


```

void signal_handler(int sno)
{
    // burada yeniden sinyal gelirse proses sonlanır
    signal(sno, signal_handler);
    ....
}

```

signal fonksiyonu ile sinyal set etmeye UNIX dünyasında "unreliable signal" da denilmektedir.

```

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

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

Sinyal oluştuğunda set çağrılmak üzere set edilen fonksiyonun (signal
handler) int parametresi oluşan sinyalin numarasını
belirtmektedir. Böylece farklı sinyaller için aynı fonksiyon set
edilebilir ve bu parametreye bakarak ayrıştırma yapılabilir.

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

```

```

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

```

```

void exit_sys(const char *msg);
void signal_handler(int sno);

```

```

int main(void)
{
    int i;

    if (signal(SIGUSR1, signal_handler) == SIG_ERR)
        exit_sys("signal");

    if (signal(SIGUSR2, signal_handler) == SIG_ERR)
        exit_sys("signal");

    for (i = 0;; ++i) {
        printf("%d\n", i);
        sleep(1);
    }

    return 0;
}

```

```

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

    exit(EXIT_FAILURE);
}

```

```

void signal_handler(int sno)
{
    if (sno == SIGUSR1)
        printf("SIGUSR1\n");
    else if (SIGUSR2)
        printf("SIGUSR2\n");
}

```

/

*-----

sigaction fonksiyonun ssemantiği her sistemde aynıdır. Dolayısıyla signal fonksiyonu yerine bu fonksiyon tercih edilmelidir.

Bu fonksiyonda struct sigaction isimli bir yapı nesnesinin içi doldurulur ve fonksiyona verilir. Bu yapının doldurulması gereken elemanları şunlardır:

sa_handler: Çağrılacak sinyal fonksiyonun adresi buraya yerleştirilir.

sa_mask: Bu eleman sigset_t türündendir. Sinyal fonksiyonu çalıştığı sürece prosesin signal mask'ine burada belirtilen sinyaller eklenir. Fonksiyonun çalışması bittiğinde eklenmiş olan bu sinyaller prosesin signal mask'inden çıkartılır.

Bu bir dizisi üzerinde işlem yapan sigemptyset, sigfillset, sigaddset, sigdelset ve sigismember isimli fonksiyonlar bulunmaktadır:

sa_flags: Bu elemana SA_XXXX biçiminde çeşitli bayraklar OR'lanarak girilir. SA_RESETHAND sinyal fonksiyonu çalıştırılırken sinyalin default'a çekileceğini belirtir. Bu flag set edilmezse sinyal default'a çekilmemektedir. SA_NODEFER sinyal fonksiyonu çalıştığı sürece aynı numaralı sinyalin bloke edilmeyeceği anlamına gelir. Bu flag belirtilmezse sinyal fonksiyonu çalışırken sa_mask dikkate alınmaksızın aynı numaralı sinyal bloke edilmektedir. SA_RESTART yavaş sistem fonksiyonlarının yeniden otomatik başlatılacağı anlamına gelmektedir.

Sistem 5 semantiği için sa_flags elemanının SA_NODEFER | SA_RESETHAND biçiminde olması gerekir. BSD semantiği için ise flags elemanı SA_RESTART biçiminde olması gerekir. Bu durumda bu eleman 0'da tutulursa otomatik default'a çekme uygulanmaz, sinyal fonksiyonu çalıştığı sürece aynı numaralı sinyal bloke edilir ve sistem fonksiyonları otomatik restart edilmez. Linux'taki signal POSIX fonksiyonu glibc 20'dan sonra sa_flags = SA_RESTART biçimindedir ve BSD semantiğini uygulamaktadır.

-----*/

```

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

```

```

void exit_sys(const char *msg);

```

```

void sigint_handler(int sno);

int main(void)
{
    struct sigaction act;
    int i;

    act.sa_handler = sigint_handler;
    sigemptyset(&act.sa_mask);
    act.sa_flags = 0;

    if (sigaction(SIGINT, &act, NULL) == -1)
        exit_sys("sigaction");

    for (i = 0;; ++i) {
        printf("%d\n", i);
        sleep(1);
    }

    return 0;
}

```

```

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

    exit(EXIT_FAILURE);
}

```

```

void sigint_handler(int sno)
{
    printf("sigint signal occurs...\n");
}

```

/

*-----

Her prosesin bloke edilmiş sinyalleri belirten bir signal mask'i vardır. Prosesin signal mask'i Linux sistemlerinde task_struct içerisinde saklanmaktadır. Bloke edilen sinyal oluşursa işletim sistemi sinyali prosese teslim etmez (deliver etmez). Onu bekletir. Bu beklemedeki bir sinyale "pending" durumda denilmektedir. Eğer proses bloke edilen sinyalin blokesini açarsa bu durumda işletim sistemi artık o pending durumda olan sinyali prosese teslim eder. Prosesin signal mask'ine yeni sinyallerin yerleştirilmesi ya da oradan sinyal çıkartılması sigprocmask isimli POSIX fonksiyonuyla yapılmaktadır.

UNIX türevi sistemlerde klasik sinyallerde kuyruklama yoktur. Yani bloke edilmiş bir sinyal birden fazla kez oluşursa sinyalin blokesi çözüldüğünde yalnızca 1 kez bu sinyalden oluşur.

Aşağıdaki örnekte SIGINT sinyali önce prosesin signal mask'ine eklenerek bloke edilmiş sonra da açılmıştır.

```

-----*/

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

void exit_sys(const char *msg);
void sigint_handler(int sno);

int main(void)
{
    int i;
    sigset_t sset;
    struct sigaction act;

    act.sa_handler = sigint_handler;
    sigemptyset(&act.sa_mask);
    act.sa_flags = 0;

    if (sigaction(SIGINT, &act, NULL) == -1)
        exit_sys("sigaction");

    sigemptyset(&sset);
    sigaddset(&sset, SIGINT);

    if (sigprocmask(SIG_BLOCK, &sset, NULL) == -1)
        exit_sys("sigprocmask");

    for (i = 0;; ++i) {
        printf("%d\n", i);
        if (i == 10)
            if (sigprocmask(SIG_UNBLOCK, &sset, NULL) == -1)
                exit_sys("sigprocmask");
            sleep(1);
    }

    return 0;
}

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

    exit(EXIT_FAILURE);
}

void sigint_handler(int sno)
{
    printf("SIGINT occurred...\n");
}

```

```

/
*-----
-----
    pause isimli POSIX fonksiyonu bir sinyal oluşana kadar blokede beklemeye
    yol açar. Eğer bir sinyal için sinyal fonksiyonu
    set edilmişse pause fonksiyonu sinyal fonksiyonu çalıştırıldıktan sonra
    geri dönmektedir. Fonksiyon her zaman -1 değerine geri
    döner ve errno değeri de her zaman EINTR olarak set edilir.
-----
-----*/

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

void exit_sys(const char *msg);
void sigint_handler(int sno);

int main(void)
{
    struct sigaction act;

    act.sa_handler = sigint_handler;
    sigemptyset(&act.sa_mask);
    act.sa_flags = 0;

    if (sigaction(SIGINT, &act, NULL) == -1)
        exit_sys("sigaction");

    printf("waiting for a signal\n");

    pause();

    printf("ok\n");

    return 0;
}

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

    exit(EXIT_FAILURE);
}

void sigint_handler(int sno)
{
    printf("SIGINT occurred...\n");
}

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

```

Bazen programlar sonsuz döngüde sinyalleri bekleyerek işlevlerini gerçekleştirebilmektedir.

```
-----*/

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

void exit_sys(const char *msg);
void sigint_handler(int sno);

int main(void)
{
    struct sigaction act;

    act.sa_handler = sigint_handler;
    sigemptyset(&act.sa_mask);
    act.sa_flags = 0;

    if (sigaction(SIGINT, &act, NULL) == -1)
        exit_sys("sigaction");

    printf("waiting for a signal\n");

    for (;;)
        pause();

    return 0;
}

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

    exit(EXIT_FAILURE);
}

void sigint_handler(int sno)
{
    printf("SIGINT occurred...\n");
}

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

abort standart C fonksiyonu parametresizdir ve geri dönüş değerine sahip
değildir. Bu fonksiyon anormal çıkışlar için düşünülmüştür.
Bir programda çok ciddi birtakım sorunlar tespit edilmişse çıkış exit
ile değil abort ile yapılmalıdır. abort SIGABRT isimli
sinyalin oluşmasına yol açmaktadır.
-----
-----*/
```

```

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

int main(void)
{
    printf("program runs...\n");

    abort();

    printf("unreachable code...\n");

    return 0;
}

/
*-----
-----
    abort fonksiyonu çağrıldığında SIGABRT sinyali "raise" edilmektedir.
    (Sinyalin raise edilmesi sonraki örneklerde ele alınmaktadır.)
    Bu durumda eğer sinyal fonksiyonu set edilmişse o fonksiyon çalıştırılır
    fonksiyonun çalışması bittiğinde proses sonlandırılır.
    SIGABRT sinyali her zaman core dosyasının oluşmasına yol açmaktadır.

    Aşağıdaki örnekte abort fonksiyonu çağrılarda set edilen fonksiyonu
    çalışacak bu fonksiyon bittiğinde program sonlandırılacaktır.
    Yani abort'tan sonraki "ok" yazısı ekranda görülmeyecektir.
-----
-----*/

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

void exit_sys(const char *msg);
void sigabrt_handler(int sno);

int main(void)
{
    struct sigaction act;

    act.sa_handler = sigabrt_handler;
    sigemptyset(&act.sa_mask);
    act.sa_flags = 0;

    if (sigaction(SIGABRT, &act, NULL) == -1)
        exit_sys("sigaction");

    printf("waiting for a signal\n");

    abort();

    printf("ok\n");

    return 0;
}

```

```

}

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

    exit(EXIT_FAILURE);
}

void sigabrt_handler(int sno)
{
    printf("SIGABRT occurred...\n");
}

/
*-----
-----
SIGABRT sinyali abort tarafından değil de kill gibi bir fonksiyon
tarafından gönderilirse sinyal fonksiyonu çalıştıktan sonra
program sonlanmaz. Çünkü programı raise işlemi sonrasında abort
fonksiyonu sonlandırmaktadır.

Aşağıdaki programı çalıştırıp başka bir terminalden kill -ABRT <pid>
komutu ile SIGABRT sinyalini gönderiniz.
-----
-----*/

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

void exit_sys(const char *msg);
void sigabrt_handler(int sno);

int main(void)
{
    struct sigaction act;

    act.sa_handler = sigabrt_handler;
    sigemptyset(&act.sa_mask);
    act.sa_flags = 0;

    if (sigaction(SIGABRT, &act, NULL) == -1)
        exit_sys("sigaction");

    printf("waiting for a signal\n");

    pause();

    printf("ok\n");

    return 0;
}

```



```

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

    exit(EXIT_FAILURE);
}

```

```

void sigabrt_handler(int sno)
{
    printf("SIGABRT occurred...\n");
}

```

```

/
*-----
-----
C'de goto deyimi aynı fonksiyon içerisindeki atlamalarda kullanılır. Bir
fonksiyondan başka bir fonksiyona atlamaya
"nonlocal jump" ya da "long jump" denilmektedir. Ancak programcı
herhangi bir yere long jump yapamaz. Ancak daha önce
geçmiş olduğu bir noktaya long jump yapabilir. İşte setjmp fonksiyonu
ile önce geri dönecek yer belirlenir. Sonra longjmp ile
buraya geri dönülür. setjmp fonksiyonu ilk çağrıda 0 ile geri
dönmektedir. longjmp yapıldığında akış yine setjmp'ın içerisinde
çıkar.
Fakat bu kez setjmp 0 ile değil longjmp'de belirtilen değerle geri
dönmektedir. Nesne yönelimli programlama dillerindeki try-catch-throw
mekanizmaları da aynı yöntemle derleyici tarafından sağlanmaktadır.
-----
-----*/

```

```

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

```

```

void foo(void);
void bar(void);

```

```

jmp_buf g_jb;

```

```

int main(void)
{
    int result;

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

    if ((result = setjmp(g_jb)) == 0)
        printf("first set...\n");
    else if (result == 1) {
        printf("return from longjmp...\n");
        exit(EXIT_SUCCESS);
    }

    foo();

    return 0;
}

```

```

}

void foo(void)
{
    printf("foo begins...\n");
    bar();
    printf("foo ends...\n");
}

void bar(void)
{
    printf("bar begins...\n");
    longjmp(g_jb, 1);
    printf("bar ends...\n");
}

/
*-----
-----
    Eğer SIGABRT sinyali set edilen fonksiyonda longjmp yapılırsa abort
    fonksiyonu prosesi sonlandıramaz.
-----
-----*/

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

void exit_sys(const char *msg);
void sigabrt_handler(int sno);

jmp_buf g_jb;

int main(void)
{
    struct sigaction act;
    int result, i;

    act.sa_handler = sigabrt_handler;
    sigemptyset(&act.sa_mask);
    act.sa_flags = 0;

    if (sigaction(SIGABRT, &act, NULL) == -1)
        exit_sys("sigaction");

    if ((result = setjmp(g_jb)) == 1)
        goto CONTINUE;

    abort();

CONTINUE:
    for (i = 0; i < 10; ++i) {
        printf("%d ", i), fflush(stdout);
    }
}

```

```

        sleep(1);
    }
    printf("\n");

    return 0;
}

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

    exit(EXIT_FAILURE);
}

void sigabrt_handler(int sno)
{
    printf("SIGABRT occurred...\n");

    longjmp(g_jb, 1);
}

/
*-----
-----
    raise POSIX fonksiyonu kendi prosesine sinyal göndermek için kullanılır.
    Aslında raise(signo) ile kill(getpid(), signo) tamamen
    eşdeğerdir. raise aynı zamanda standart bir C fonksiyonudur. raise
    fonksiyonu ile sinyal gönderildiğinde fonksiyon geri dönmeden
    kesinlikle
    sinyal fonksiyonun çalışmış olacağı garanti edilmiştir. Aynı garanti
    kill fonksiyonu ile kendi prosesimize sinyal gönderirken de
    verilmiştir.
-----
-----*/

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

void exit_sys(const char *msg);
void sigusr1_handler(int sno);

int main(void)
{
    struct sigaction act;

    act.sa_handler = sigusr1_handler;
    sigemptyset(&act.sa_mask);
    act.sa_flags = 0;

    if (sigaction(SIGUSR1, &act, NULL) == -1)
        exit_sys("sigaction");

    printf("raise will be called...\n");

```

```

    if (raise(SIGUSR1) == -1)
        exit_sys("raise");

    printf("raise called...\n");

    return 0;
}

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

    exit(EXIT_FAILURE);
}

void sigusr1_handler(int sno)
{
    printf("SIGUSR1 occurred...\n");
}

```

```

/
*-----
-----
alarm fonksiyonu belli bir saniye sonra kendi prosesine SIGALRM
sinyalinin gönderilmesine yol açar. Eğer daha önce bir
alarm set edilmişse o silinir yenisi set edilmiş olur. Fonksiyonun
parametresi 0 olarak geçilirse eski alarm iptal edilmektedir.
Fonksiyon bir önce set edilmiş alarm'daki kalan saniye sayısını
vermektedir.
-----
-----*/

```

```

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

void exit_sys(const char *msg);
void sigalrm_handler(int sno);

int main(void)
{
    struct sigaction act;

    act.sa_handler = sigalrm_handler;
    sigemptyset(&act.sa_mask);
    act.sa_flags = 0;

    if (sigaction(SIGALRM, &act, NULL) == -1)
        exit_sys("sigaction");

    alarm(10);

    pause();
}

```

```

        return 0;
    }

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

    exit(EXIT_FAILURE);
}

void sigalrm_handler(int sno)
{
    printf("SIGALRM occurred...\n");
}

/
*-----
-----
    Programcılar genellikle periyodik timer oluşturmak için alarm sinyali
    fonksiyonunda yeniden alarm çağırması yaparlar.
-----
-----*/

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

void exit_sys(const char *msg);
void sigalrm_handler(int sno);

int main(void)
{
    struct sigaction act;

    act.sa_handler = sigalrm_handler;
    sigemptyset(&act.sa_mask);
    act.sa_flags = 0;

    if (sigaction(SIGALRM, &act, NULL) == -1)
        exit_sys("sigaction");

    alarm(1);

    for(;;)
        pause();

    return 0;
}

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

```

```

    exit(EXIT_FAILURE);
}

void sigalrm_handler(int sno)
{
    printf("SIGALRM occurred...\n");

    alarm(1);
}

```

```

/
*-----

```

Bir sinyal için sinyal fonksiyonu (signal handler) set ettiğimizde o sinyal fonksiyonun içerisinde çağıracağımız fonksiyonlara dikkat etmeliyiz. Sinyal fonksiyonlarının içerisinde ancak "sinyal güvenli (async-signal safe)" fonksiyonları çağırabiliriz. POSIX standartlarında sinyal güvenli fonksiyonların neler olduğu listelenmiştir. Tabii bazı koşullar altında dikkat olmak koşuluyla sinyal fonksiyonlarının içerisinde sinyal güvenli olmayan fonksiyonları da çağırabiliriz.

Bir POSIX fonksiyonu çalışırken onun içerisinde sinyal oluşursa ve sinyal fonksiyonu da aynı fonksiyonu çağırırsa üstelik bu fonksiyon sinyal güvenli değilse sorun oluşabilir. Thread güvenlilikle sinyal güvenlilik aynı anlamda değildir. Bir fonksiyon şu kategorilerden birine girebilir:

- 1) Hem thread güvenli hem de sinyal güvenli
- 2) Thread güvenli fakat sinyal güvenli değil
- 3) Sinyal güvenli ama thread güvenli değil
- 4) Thread güvenli de değil, sinyal güvenli de değil

Örneğin aşağıdaki fonksiyon thread güvenli olduğu halde sinyal güvenli değildir:

```

thread_local int g_i;

void foo(void)
{
    g_i = 10;
    ...
    g_i = 20;
    ...
    g_i = 30;
}

```

Bazı POSIX fonksiyonlarının thread güvenli olduğu olduğu halde sinyal güvenli olmadığına dikkat ediniz: Örneğin malloc, calloc, stdio fonksiyonları gibi...

```

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

```

/

*-----

Bir sinyal fonksiyonu içerisinde errno'yu değiştiren bir POSIX fonksiyonu kullanılırsa sinyal kestiği fonksiyonun set ettiği errno değeri bozulabilir. Bunun için eğer böyle bir fonksiyon çağrılacaksa sinyal fonksiyonlarının başında errno değerini saklayıp sonunda yeniden set etmek iyi bir tekniktir. Örneğin:

```
void signal_handler(int signo)
{
    int temp = errno;
    ....
    errno = temp;
}
```

POSIX fonksiyonları başarı durumunda errno değerini 0 olarak set etmezler. POSIX standartlarına göre errno hiçbir fonksiyon tarafından 0'a çekilmemektedir. POSIX standartları başarı durumunda errno değerinin fonksiyon tarafından değiştirilebileceğini açıkça belirtmiştir. Ancak başarı durumunda errno değerini başka değerlere set eden POSIX fonksiyonları vardır. Bu nedenle programcının errno değişkenini saklayıp geri yerleştirmesi uygun olur.

-----*/

/

*-----

Alt proses sonlandığında işletim sistemi o prosesin üst prosesine SIGCHLD sinyali göndermektedir. Bu sinyalin ismi AT&T UNIX sistemlerinde eskiden SIGCLD biçimindeydi. Ancak POSIX BSD ismi olan SIGCHLD ismini tercih etti. O zamanlar SIGCLD sinyalinin semantiği ile SIGCHLD sinyalinin semantiği arasında da küçük farklılıklar vardı. POSIX standartları bu farklılıkları "implementation dependent" hale getirmiştir.

-----*/

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

```
void exit_sys(const char *msg);
void sigchld_handler(int sno);
```

```
int main(void)
{
    struct sigaction act;
    pid_t pid;

    act.sa_handler = sigchld_handler;
```

```

sigemptyset(&act.sa_mask);
act.sa_flags = 0;

if (sigaction(SIGCHLD, &act, NULL) == -1)
    exit_sys("sigaction");

if ((pid = fork()) == -1)
    exit_sys("fork");

if (pid == 0) {
    sleep(5);
    _exit(EXIT_FAILURE);
}

pause();

if (waitpid(pid, NULL, 0) == -1)
    exit_sys("waitpid");

return 0;
}

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

    exit(EXIT_FAILURE);
}

void sigchld_handler(int sno)
{
    printf("SIGCHLD occurred...\n");
}

/
*-----
-----
    SIGCHLD sinyali genellikle zombie proses oluşumunu otomatik engellemek
    için kullanılmaktadır. Yani tipik olarak programcı
    bu sinyale ilişkin sinyal fonksiyonu içerisinde wait fonksiyonlarını
    uygular.
-----
-----*/

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

void exit_sys(const char *msg);
void sigchld_handler(int sno);

int main(void)
{

```



```

struct sigaction act;
pid_t pid;

act.sa_handler = sigchld_handler;
sigemptyset(&act.sa_mask);
act.sa_flags = 0;

if (sigaction(SIGCHLD, &act, NULL) == -1)
    exit_sys("sigaction");

if ((pid = fork()) == -1)
    exit_sys("fork");

if (pid == 0) {
    sleep(5);
    _exit(EXIT_FAILURE);
}

pause();

return 0;
}

```

```

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

    exit(EXIT_FAILURE);
}

```

```

void sigchld_handler(int sno)
{
    if (waitpid(-1, NULL, 0) == -1)
        exit_sys("waitpid");
}

```

```

/
*-----
-----
Üst prosesin birden fazla alt proses yarattığı durumda bu alt prosesler
birbirlerine yakın zamanlarda sonlanırsa sinyaller
kuyruklanmadığından her alt proses için bir kez SIGCHLD sinyali
oluşmayabilir. Bu durum da otomatik zombie engellemeyi
sekteye uğratabilir. Örneğin aşağıda 10 tane alt proses yaratılmıştır.
Ancak sinyal fonksiyonu 10 kere çağrılmamaktadır.
-----
-----*/

```

```

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

```

```

void exit_sys(const char *msg);

```

```

void sigchld_handler(int sno);

int main(void)
{
    struct sigaction act;
    pid_t pids[10];
    int i;

    act.sa_handler = sigchld_handler;
    sigemptyset(&act.sa_mask);
    act.sa_flags = 0;

    if (sigaction(SIGCHLD, &act, NULL) == -1)
        exit_sys("sigaction");

    for (i = 0; i < 10; ++i) {
        if ((pids[i] = fork()) == -1)
            exit_sys("fork");
        if (pids[i] == 0) {
            sleep(5);
            _exit(EXIT_FAILURE);
        }
    }

    for (;;)
        pause();

    return 0;
}

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

    exit(EXIT_FAILURE);
}

void sigchld_handler(int sno)
{
    int temp = errno;

    printf("SIGCHLD occurred...\n");

    errno = temp;
}

```

/

*-----

Yukarıdaki problem aşağıdaki gibi SIGCHLD sinyal fonksiyonunda bir döngü içerisinde wait fonksiyonları uygulanarak giderilebilir. Ekrana yazdırılan yazıların sayısına bakarak 10 tane olduğunu görünüz.

```

-----*/

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>
#include <errno.h>
#include <signal.h>

void exit_sys(const char *msg);
void sigchld_handler(int sno);

int main(void)
{
    struct sigaction act;
    pid_t pids[10];
    int i;

    act.sa_handler = sigchld_handler;
    sigemptyset(&act.sa_mask);
    act.sa_flags = 0;

    if (sigaction(SIGCHLD, &act, NULL) == -1)
        exit_sys("sigaction");

    for (i = 0; i < 10; ++i) {
        if ((pids[i] = fork()) == -1)
            exit_sys("fork");
        if (pids[i] == 0) {
            sleep(5);
            _exit(EXIT_FAILURE);
        }
    }

    for (;;)
        pause();

    return 0;
}

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

    exit(EXIT_FAILURE);
}

void sigchld_handler(int sno)
{
    int temp = errno;

    while (waitpid(-1, NULL, WNOHANG) > 0)
        printf("prevented child to be zombie!..\n");
}

```

```
    errno = temp;
}
```

```
/
```

```
*-----
```

POSIX standartlarına göre otomatik zombie engellenemenin iki yolu daha vardır:

- 1) SIGCHLD sinyali ignore'a çekilir (SIG_IGN)
- 2) SIGCHLD sinyali için sigaction fonksiyonunda set yapılırken SA_NOCLDWAIT bayrağı da kullanılır.

POSIX standartları SIGCHLD sinyali için SA_NOCLDWAIT bayrağı kullanıldığında sinyal fonksiyonun çağrılıp çağrılmayacağını işletim sisteminin isteğine bırakmıştır. Linux sistemleri sinyal fonksiyonunu çağdırmaktadır.

Her ne kadar SIGCHLD sinyalinin default eylemi (default action) "ignore" olsa da burada "ignore'a çekmek" açıkça signal fonksiyonuyla ya da sigaction fonksiyonuyla SIG_IGN kullanmak anlamındadır.

Bu biçimde otomatik zombie engellendiğinde artık wait fonksiyonları kullanılmamalıdır. Eğer kullanılırsa bu fonksiyonlar -1'e geri dönerler ve errno da ECHILD olarak set edilir. Dolayısıyla aşağıdaki örnekte waitpid fonksiyonu otomatik zombie engellendiği için başarısızlıkla geri dönecektir.

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

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

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

```
int main(void)
{
```

```
    struct sigaction act;
    pid_t pids[10];
    int i;

    act.sa_handler = SIG_IGN;
    sigemptyset(&act.sa_mask);
    act.sa_flags = 0;
```

```
    if (sigaction(SIGCHLD, &act, NULL) == -1)
        exit_sys("sigaction");
```

```
    /* signal(SIGCHLD, SIG_IGN); */
```

```

for (i = 0; i < 10; ++i) {
    if ((pids[i] = fork()) == -1)
        exit_sys("fork");
    if (pids[i] == 0) {
        _exit(EXIT_FAILURE);
    }
}

printf("press ENTER to continue...\n");
getchar();

if (waitpid(-1, NULL, 0) == -1)    /* waitpid will fail!.. */
    exit_sys("waitpid");

return 0;
}

```

```

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

    exit(EXIT_FAILURE);
}

```

```

/
*-----
-----
Eğer sigaction fonksiyonunda SIGCHLD sinyali için hem fonksiyon set
edilir hem de SA_NOCLDWAIT bayrağı kullanılırsa
bu durumda otomatik zombie yine engellenmekle birlikte sinyal
fonksiyonun çağrılıp çağrılmayacağı işletim sistemine
bırakılmıştır (implementation dependent). Linux'ta çağırma
yapılmaktadır. SA_NOCLDWAIT bayrağı yalnızca SIGCHLD sinyali
için anlamlıdır.
-----
-----*/

```

```

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>
#include <errno.h>
#include <signal.h>

void exit_sys(const char *msg);
void sigchld_handler(int sno);

int main(void)
{
    struct sigaction act;
    pid_t pids[10];
    int i;

    act.sa_handler = sigchld_handler;

```

```

sigemptyset(&act.sa_mask);
act.sa_flags = SA_NOCLDWAIT;

if (sigaction(SIGCHLD, &act, NULL) == -1)
    exit_sys("sigaction");

for (i = 0; i < 10; ++i) {
    if ((pids[i] = fork()) == -1)
        exit_sys("fork");
    if (pids[i] == 0) {
        sleep(1);
        _exit(EXIT_FAILURE);
    }
}

for(;;)
    pause();

return 0;
}

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

    exit(EXIT_FAILURE);
}

void sigchld_handler(int sno)
{
    int temp = errno;

    printf("SIGCHLD occurred...\n");          /* wait shouldn't be used! */

    errno = temp;
}

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

Bazı sistem fonksiyonları (read, write, sleep gibi) bazı kaynaklarla
çalışırken uzun süre bekleme yapabilmektedir. Çünkü
bu sistem fonksiyonları istenen şey gerçekleşene kadar çizelgeden
çıakrtılıp blokeye yol açmaktadır. Pekiyi bir proses
bu sistem fonksiyonlarında blokede beklerken bir sinyal oluştuğunda ne
olur? İşte işletim sistemi bu durumda sinyali hemen
teslim edebilmek için fonksiyonun yol açtığı blokenin bitmesini
beklemez. Sistem fonksiyonunu başarısızlıkla sonlandırır ve
hemen prosesi çizelgeye sokarak ona sinyali teslim eder. Bu tür sistem
fonksiyonları (yani bunları çağıran POSIX fonksiyonları)
-1 değerine geri dönüp errno değerini EINTR olarak set etmektedir. Bu
durumda biz böylesi bir durumla karşılaştığımızda
fonksiyonun yaptığı işte başarılı olduğunu düşünmemeliyiz. Fonksiyonun
sinyal geldiğinden dolayı başarısız olduğunu ve

```

yeniden çağrılırsa başarılı olabileceğini düşünmeliyiz. O halde bu tür sistem fonksiyonlarını çağırırken bizim eğer başarısızlığın nedeni sinyal ise (EINTR) bu fonksiyonu yeniden çağırmanız gerekir. Bu da maalesef programlama açısından sıkıntılıdır.

Aşağıdaki programda alarm fonksiyonu dolayısıyla SIGALRM sinyali gönderildiğinde scanf fonksiyonu (yani aslında onun çağırdığı read fonksiyonu) başarız olacaktır ve errno EINTR ile set edilecektir.

-----*/

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <signal.h>

void exit_sys(const char *msg);
void sigalrm_handler(int sno);

int main(void)
{
    struct sigaction act;
    int val;

    act.sa_handler = sigalrm_handler;
    sigemptyset(&act.sa_mask);
    act.sa_flags = 0;

    if (sigaction(SIGALRM, &act, NULL) == -1)
        exit_sys("sigaction");

    alarm(1);

    printf("Bir sayı giriniz:");
    fflush(stdout);
    scanf("%d", &val);
    printf("%d\n", val * val);

    return 0;
}

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

    exit(EXIT_FAILURE);
}

void sigalrm_handler(int sno)
{
    /* do something */
}
```

```
/
*-----
-----
İşte bu tür durumlarda yöntemlerden birisi eğer fonksiyon sinyal
yüzünden başarısız olmuşsa bir döngü içerisinde onu
yeniden çağırmaaktır. Tabii eğer programcı kendi programında bir sinyal
işliyorsaa bu yola gitmelidir. Yoksa zaten işlenmeyen
sinyaller için proses sonlandırılır. Bir sinyalin default eylemi
(default action) eğer "ignore" ise bu durumda işletim
sistemi yavaş sistem fonksiyonlarını başarısızlıkla sonlandırmaz.
-----
-----*/
```

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <signal.h>

void exit_sys(const char *msg);
void sigalrm_handler(int sno);

int main(void)
{
    struct sigaction act;
    int val;

    act.sa_handler = sigalrm_handler;
    sigemptyset(&act.sa_mask);
    act.sa_flags = 0;

    if (sigaction(SIGALRM, &act, NULL) == -1)
        exit_sys("sigaction");

    alarm(1);

    printf("Bir sayı giriniz:");
    fflush(stdout);

    while (scanf("%d", &val) == -1 && errno == EINTR)
        ;

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

    return 0;
}

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

    exit(EXIT_FAILURE);
}

void sigalrm_handler(int sno)
```



```

{
    /* do something */
}

/
*-----
-----
    Bu çok sıkıcı bir işelmdir. Bazı programcılar bunun için aşağıdaki gibi
    bir makro kullanabilmektedir:

    #define EINTR_LOOP(statement)    while ((statement) == -1 && errno == -1)
-----
-----*/

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

#define EINTR_LOOP(statement)        while ((statement) == -1 && errno ==
    EINTR)

void exit_sys(const char *msg);
void sigalrm_handler(int sno);

int main(void)
{
    struct sigaction act;
    int val;

    act.sa_handler = sigalrm_handler;
    sigemptyset(&act.sa_mask);
    act.sa_flags = 0;

    if (sigaction(SIGALRM, &act, NULL) == -1)
        exit_sys("sigaction");

    alarm(1);

    printf("Bir sayı giriniz:");
    fflush(stdout);

    EINTR_LOOP(scanf("%d", &val));
    printf("%d\n", val * val);

    return 0;
}

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

    exit(EXIT_FAILURE);
}

```

```

void sigalrm_handler(int sno)
{
    /* do something */
}

/
*-----
-----
    Bazı yavaş sistem fonksiyonlarının çekirdek tarafından otomatik restart edilmesi için sigaction fonksiyonunda sa_flags parametresi SA_RESTART girilmelidir. Bu durumda artık programcının sinyal nedeniyle sonlanmalar için kendisinin bir döngü oluşturmaya gerek kalmaz.
-----
-----*/

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

void exit_sys(const char *msg);
void sigalrm_handler(int sno);

int main(void)
{
    struct sigaction act;
    int val;

    act.sa_handler = sigalrm_handler;
    sigemptyset(&act.sa_mask);
    act.sa_flags = SA_RESTART;

    if (sigaction(SIGALRM, &act, NULL) == -1)
        exit_sys("sigaction");

    alarm(1);

    printf("Bir sayı giriniz:");
    fflush(stdout);

    if (scanf("%d", &val) == -1)
        exit_sys("scanf");

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

    return 0;
}

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

```

```

    exit(EXIT_FAILURE);
}

void sigalrm_handler(int sno)
{
    /* do something */
}

/
*-----
-----
    Anımsanacağı gibi eski signal fonksiyonunda sinyal oluştuğunda yavaş
    fonksiyonlarında otomatik restart işlemi işletim sistemine
    bağlı olarak değişebiliyordu (implementation dependent). Linux
    sistemlerinde otomatik restart yapıldığını anımsayınız.
-----
-----*/

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

void exit_sys(const char *msg);
void sigalrm_handler(int sno);

int main(void)
{
    int val;

    if (signal(SIGALRM, sigalrm_handler) == SIG_ERR)
        exit_sys("signal");

    alarm(1);

    printf("Bir sayı giriniz:");
    fflush(stdout);

    if (scanf("%d", &val) == -1)
        exit_sys("scanf");

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

    return 0;
}

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

    exit(EXIT_FAILURE);
}

void sigalrm_handler(int sno)

```

```

{
    /* do something */
}

/
*-----
-----
    Biz sigaction fonksiyonunda flags parametresinde SA_RESTART girmiş olsak
    bile bazı yavaş sistem fonksiyonları doğası gereği
    restart edilememektedir. SA_RESTART bayrağı belirtildiğinde genel olarak
    POSIX standartlarında hangi fonksiyonların hangi
    koşullar altında restart edileceği belirtilmemiştir. Örneğin sleep,
    nanosleep gibi bekleme fonksiyonları restart işlemi
    yapmazlar. Zaten bu işlemin bu fonksiyonlarda yapılması anlamlı
    değildir.
    read (write vs.) fonksiyonu talep edildiği kadar bilgiyi henüz
    okuyamamışken sinyal oluşabilir. Bu durumda SA_RESTART bayrağı
    set edilmemiş olsa bile fonksiyon başarısız olmaz. Okuduğu kadar bilgiye
    geri döner.
    -----
    -----*/

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

void exit_sys(const char *msg);
void sigusr1_handler(int sno);

int main(void)
{
    struct sigaction act;

    act.sa_handler = sigusr1_handler;
    sigemptyset(&act.sa_mask);
    act.sa_flags = SA_RESTART;

    if (sigaction(SIGUSR1, &act, NULL) == -1)
        exit_sys("sigaction");

    if (sleep(60) != 0)
        if (errno == EINTR)
            printf("sleep finished due to signal...\n");
        else
            exit_sys("sleep");
    else
        printf("sleep finished normally...\n");

    return 0;
}

void exit_sys(const char *msg)
{

```

```

    perror(msg);

    exit(EXIT_FAILURE);
}

void sigusr1_handler(int sno)
{
    /* do something */
}

/
*-----
-----
    open, read write gibi fonksiyonları bloke durumunda sinyal geldiğinde
    eğer proses sonlanmazsa başarısız olmaktadır.
    Aşağıdaki örnekte "myfifo" isimli bir boru dosyası açılmak istenmiştir.
    Başka bir proses boruyu yazma modunda açana kadar
    bloke olacaktır. Bu durumda başka bir terminalden SIGUSR1 sinyalini
    prosese göndererek open fonksiyonunun EINTR errno
    değeriyle başarısız olduğunu görünüz.
-----
-----*/

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

void exit_sys(const char *msg);
void sigusr1_handler(int sno);

int main(void)
{
    struct sigaction act;
    int fd;
    ssize_t size;
    char buf[10];

    act.sa_handler = sigusr1_handler;
    sigemptyset(&act.sa_mask);
    act.sa_flags = 0;

    if (sigaction(SIGUSR1, &act, NULL) == -1)
        exit_sys("sigaction");

    if ((fd = open("myfifo", O_RDONLY)) == -1)
        exit_sys("open");

    printf("pipe opened successfully...\n");

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

```

```

    printf("read read successfully\n");

    close(fd);

    return 0;
}

```

```

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

    exit(EXIT_FAILURE);
}

```

```

void sigusr1_handler(int sno)
{
    /* do something */
}

```

/

POSIX'e 90 yılların ortalarında "realtime extensions" başlığı altında "gerçek zamanlı (realtime)" sinyal kavramı da eklendi. Gerçek zamanlı sinyallerin numaraları [SIGRTMIN, SIGRTMAX] arasındadır. Bunlara ayrı isimler verilmemiştir. Gerçek zamanlı sinyallerin normal sinyallerden (ilk 32 sinyal numarası normal sinyaller için ayrılmıştır) farkları şunlardır:

- 1) Gerçek zamanlı sinyaller kuyruklanmaktadır. Yani birden fazla gerçek zamanlı aynı sinyal oluştuğunda kaç tane oluşmuş olduğu tutulur ve o sayıda prosese teslim edilir.
- 2) Gerçek zamanlı sinyallerde bir bilgi de sinyale iliştirilebilmektedir. Bu bilgi ya int bir değer ya da bir gösterici olur. Gösterici kullanıldığında bu göstericinin gösterdiği yerin hedef proseste anlamlı olması gerekmektedir.
- 3) Gerçek zamanlı sinyallerde bir öncelik ilişkisi (priority) vardır. Birden fazla farklı numaralı gerçek zamanlı sinyal bloke edildiği durumda bloke açılınca bunların oluşma sırası küçük numaradan büyük numaraya göredir.

Gerçek zamanlı sinyaller kill fonksiyonu ile değil sigqueue isimli fonksiyonla gönderilmektedir. Eğer bu sinyaller kill ile gönderilirse kuyruklama yapıp yapılmayacağı sistemden sisteme değişebilmektedir.

Gerçek zamanlı sinyaller alınırken yine sigaction fonksiyonu kullanılmak zorundadır. Bu fonksiyonda sinyal fonksiyonu için artık sigaction yapısının sa_handler elemanına değil sa_sigaction elemanına atama yapılmalıdır. Tabii fonksiyonun bunu anlaması için flags parametresine de ayrıca SA_SIGINFO eklenmelidir. (Yani başka bir deyişle fonksiyon sa_flags parametresinde SA_SIGINFO değerini gördüğünde artık sinyal fonksiyonu için yapının sa_sigaction elemanı bakar.)

Aşağıdaki örnekte proc1 programı n tane gerçek zamanlı sinyal gönderir. proc2 ise bunları almaktadır. Programlaro çalıştırarak kuyruklanmanın yapıldığına dikkat ediniz. sigqueue fonksiyonunda iliştirilen sinyal bilgisi siginfo_t yapısının si_value elemanından alınmaktadır.

sigqueue fonksiyonuyla set edilen inyal fonksiyonundaki siginfo_t yapısının diğer elemanlarını ilgili dokimanlardan inceleyiniz. (Örneğin burada sinyali gönderen proses id'si, gerçek kullanıcı id'si, sinyalin neden gönderildiği gibi bilgiler vardır.)

```
-----*/
/* proc1.c */

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

void exit_sys(const char *msg);

/* ./prog1 <realtime signal no> <process id> <count> */

int main(int argc, char *argv[])
{
    int signo;
    pid_t pid;
    int count;
    int i;
    union sigval val;

    if (argc != 4) {
        fprintf(stderr, "wrong number of arguments!..\n");
        exit(EXIT_FAILURE);
    }

    signo = (int)strtol(argv[1], NULL, 10);
    pid = (pid_t)strtol(argv[2], NULL, 10);
    count = (int)strtol(argv[3], NULL, 10);

    for (i = 0; i < count; ++i) {
        val.sival_int = i;
        if (sigqueue(pid, SIGRTMIN + signo, val) == -1)
            exit_sys("sigqueue");
    }

    return 0;
}

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

```

    exit(EXIT_FAILURE);
}

void sigusr1_handler(int sno)
{
    printf("sigusr1 occurred...\n");
}

/* proc2.c */

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

void exit_sys(const char *msg);
void sigrt_handler(int signo, siginfo_t *info, void *context);

/* ./prog2 <realtime signal no> */

int main(int argc, char *argv[])
{
    struct sigaction act;
    int signo;

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

    signo = (int)strtol(argv[1], NULL, 10);

    act.sa_sigaction = sigrt_handler;
    sigemptyset(&act.sa_mask);
    act.sa_flags = SA_SIGINFO;

    if (sigaction(SIGRTMIN + signo, &act, NULL) == -1)
        exit_sys("sigaction");

    for(;;)
        pause();

    return 0;
}

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

    exit(EXIT_FAILURE);
}

void sigrt_handler(int signo, siginfo_t *info, void *context)
{

```



```

printf("SIGRTMIN + 0 occurred with %d code\n", info-
>si_value.sival_int);
}

```

/

```

*-----
-----

```

Sinyal konusu UNIX türevi sistemlerde 70'lerin başlarından beri vardı. Oysa thread'ler 90'ların ortalarında bu sistemlere sokulmuştur. Yani aslında sinyal konusu thread'siz bir ortamda ortaaya çıkmış ve kullanılmaya başlanmıştır. Ancak thread'ler eklendikten sonra bazı belirlemelerin de yapılması gerekmiştir.

Sinyaller default olarak kill ve sigqueue fonksiyonları tarafından prosese gönderilirler. Proses bu sinyalleri herhangi bir thread akışı tarafından işletebilir. Prosese gönderilen sinyalin hangi thread tarafından işletileceği POSIX standartlarında belirsiz bırakılmıştır.

Aşağıdaki programda 3 thread yaratılmıştır. Ana thread'le birlikte toplam 4 thread vardır. Siz de başka bir terminalden prosese kill komutuyla (kill fonksiyonuyla) SIGUSR1 sinyalini göndererek bu sinyalin hangi thread tarafından ele alındığına bakınız.

```

-----*/

```

```

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <pthread.h>
#include <signal.h>

```

```

void exit_sys(const char *msg);
void exit_sys_thread(const char *msg, int err);
void *thread_proc1(void *param);
void *thread_proc2(void *param);
void *thread_proc3(void *param);
void sigusr1_handler(int sno);

```

```

int main(void)
{
    int result;
    pthread_t tid1, tid2, tid3;
    struct sigaction act;

    act.sa_handler = sigusr1_handler;
    sigemptyset(&act.sa_mask);
    act.sa_flags = 0;

    if (sigaction(SIGUSR1, &act, NULL) == -1)
        exit_sys("sigaction");

    if ((result = pthread_create(&tid1, NULL, thread_proc1, NULL)) != 0)
        exit_sys_thread("pthread_create", result);
}

```

```

    if ((result = pthread_create(&tid2, NULL, thread_proc2, NULL)) != 0)
        exit_sys_thread("pthread_create", result);

    if ((result = pthread_create(&tid3, NULL, thread_proc3, NULL)) != 0)
        exit_sys_thread("pthread_create", result);

    printf("main thread waiting at pause...\n");
    pause();
    printf("main thread resuming...\n");

    if ((result = pthread_join(tid1, NULL)) != 0)
        exit_sys_thread("pthread_join", result);

    if ((result = pthread_join(tid2, NULL)) != 0)
        exit_sys_thread("pthread_join", result);

    if ((result = pthread_join(tid3, NULL)) != 0)
        exit_sys_thread("pthread_join", result);

    return 0;
}

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

    exit(EXIT_FAILURE);
}

void exit_sys_thread(const char *msg, int err)
{
    fprintf(stderr, "%s: %s\n", msg, strerror(err));
    exit(EXIT_FAILURE);
}

void *thread_proc1(void *param)
{
    printf("thread1 waiting at pause...\n");
    pause();
    printf("thread1 resuming...\n");

    return NULL;
}

void *thread_proc2(void *param)
{
    printf("thread2 waiting at pause...\n");
    pause();
    printf("thread2 resuming...\n");

    return NULL;
}

void *thread_proc3(void *param)
{

```

```

    printf("thread3 waiting at pause...\n");
    pause();
    printf("thread3 resuming...\n");

    return NULL;
}

```

```

void sigusr1_handler(int sno)
{
    printf("SIGUSR1 occurred...\n");
}

```

```

/
*-----
-----
Sinyal set işlemi (signal disposition) thread'e özgü bir işlem değildir.
Prosese özgü bir işlemdir. Yani biz falanca thread için
sinyal edemeyiz. Proses için edebiliriz. Ancak istersek prosesin hangi
thread'inin bir sinyali işleyeceğini belirleyebiliriz.
pthread_kill isimli fonksiyon prosesin belli bir thread'ine sinyal
gönderir. Yani kesinlikle sinyal fonksiyonu o thread tarafından
çalıştırılacaktır. Ancak pthread_kill ile başka bir proses başka bir
prosesin thread'ine sinyal gönderemez. Aynı proses kendi
thread'ine sinyal gönderebilir. (Anımsanacağı gibi thread id'sini
belirten pthread_t değeri o proseste anlamlıdır. Sistem genelinde
tek değildir.)

Aşağıdaki programda prosesin ana thread'i prosesin başka bir thread'ine
SIGUSR1 sinyalini göndermiştir.
-----
-----*/

```

```

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <pthread.h>
#include <signal.h>

void exit_sys(const char *msg);
void exit_sys_thread(const char *msg, int err);
void *thread_proc1(void *param);
void *thread_proc2(void *param);
void *thread_proc3(void *param);
void sigusr1_handler(int sno);

int main(void)
{
    int result;
    pthread_t tid1, tid2, tid3;
    struct sigaction act;

    act.sa_handler = sigusr1_handler;
    sigemptyset(&act.sa_mask);
    act.sa_flags = 0;

```

```

    if (sigaction(SIGUSR1, &act, NULL) == -1)
        exit_sys("sigaction");

    if ((result = pthread_create(&tid1, NULL, thread_proc1, NULL)) != 0)
        exit_sys_thread("pthread_create", result);

    if ((result = pthread_create(&tid2, NULL, thread_proc2, NULL)) != 0)
        exit_sys_thread("pthread_create", result);

    if ((result = pthread_create(&tid3, NULL, thread_proc3, NULL)) != 0)
        exit_sys_thread("pthread_create", result);

    sleep(5);

    if (pthread_kill(tid2, SIGUSR1) == -1)
        exit_sys("pthread_kill");

    if ((result = pthread_join(tid1, NULL)) != 0)
        exit_sys_thread("pthread_join", result);

    if ((result = pthread_join(tid2, NULL)) != 0)
        exit_sys_thread("pthread_join", result);

    if ((result = pthread_join(tid3, NULL)) != 0)
        exit_sys_thread("pthread_join", result);

    return 0;
}

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

    exit(EXIT_FAILURE);
}

void exit_sys_thread(const char *msg, int err)
{
    fprintf(stderr, "%s: %s\n", msg, strerror(err));
    exit(EXIT_FAILURE);
}

void *thread_proc1(void *param)
{
    printf("thread1 waiting at pause...\n");
    pause();
    printf("thread1 resuming...\n");

    return NULL;
}

void *thread_proc2(void *param)
{
    printf("thread2 waiting at pause...\n");

```

```

    pause();
    printf("thread2 resuming...\n");

    return NULL;
}

```

```

void *thread_proc3(void *param)
{
    printf("thread3 waiting at pause...\n");
    pause();
    printf("thread3 resuming...\n");

    return NULL;
}

```

```

void sigusr1_handler(int sno)
{
    printf("SIGUSR1 occurred...\n");
}

```

```

/
*-----

```

Başka bir procesten başka bir prosesin thread'ine sinyal göndermek için POSIX standartlarında bir yöntem yoktur. Ancak Linux sistemlerinde bunun için tkill ve tckill isimli sistem fonksiyonları bulundurulmuştur. Fakat maalesef bu sistem fonksiyonlarını çağıran sarma kütüphane fonksiyonları (wrapper) bulunmamaktadır. tckill sistem fonksiyonu tkill fonksiyonundan bir parametre daha fazladır. Bu fonksiyonda biz sinyal göndereceğimiz prosesin id değerini, sinyal göndereceğimiz thread'in task struct pid değerini ve sinyal numarasını veririz. kill fonksiyonuyla (ya da komut satırından kill komutuyla) tid değeri verilerek sinyal gönderilmeye çalışılırsa maalesef bu durumda sinyal thread'e değil thread'in ilişkin olduğu prosesle gönderilmektedir.

Aşağıdaki örnekte tckill.c programı thread'e sinyal göndermektedir. sample.c programı ise üç thread oluşturup sinyal gelmesini beklemektedir. Bu iki programı farklı terminallerde çalıştırınız. tckill ile sinyal göndermeden önce ps -a -L -o pid, tid, cmd komutu ile yaratılmış olan thread'lerin task_struct pid değerlerini görünüz. sample.c programı SIGUSR1 sinyalini işlemektedir. Bu sinyalin Linux sistemlerindekiş numarası 10'dur.

```

-----*/

```

```

/* tckill.c */

```

```

#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <unistd.h>
#include <sys/syscall.h>

```

```

void exit_sys(const char *msg);

```

```

int tgkill(int tgid, int tid, int signo)
{
    return syscall(SYS_tgkill, tgid, tid, signo);
}

/* ./tgkill <process id> <thread task struct process id> <signal no> */

int main(int argc, char *argv[])
{
    int tgid, tid, signo;

    if (argc != 4) {
        fprintf(stderr, "wrong number of arguments!..\n");
        exit(EXIT_FAILURE);
    }

    tgid = (int)strtol(argv[1], NULL, 10);
    tid = (int)strtol(argv[2], NULL, 10);
    signo = (int)strtol(argv[3], NULL, 10);

    if (tgkill(tgid, tid, signo) == -1)
        exit_sys("tgkill");

    return 0;
}

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

    exit(EXIT_FAILURE);
}

/* sample.c */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <pthread.h>
#include <signal.h>

void exit_sys(const char *msg);
void exit_sys_thread(const char *msg, int err);
void *thread_proc1(void *param);
void *thread_proc2(void *param);
void *thread_proc3(void *param);
void sigusr1_handler(int sno);

int main(void)
{
    int result;
    pthread_t tid1, tid2, tid3;
    struct sigaction act;

```

```

act.sa_handler = sigusr1_handler;
sigemptyset(&act.sa_mask);
act.sa_flags = 0;

if (sigaction(SIGUSR1, &act, NULL) == -1)
    exit_sys("sigaction");

if ((result = pthread_create(&tid1, NULL, thread_proc1, NULL)) != 0)
    exit_sys_thread("pthread_create", result);

if ((result = pthread_create(&tid2, NULL, thread_proc2, NULL)) != 0)
    exit_sys_thread("pthread_create", result);

if ((result = pthread_create(&tid3, NULL, thread_proc3, NULL)) != 0)
    exit_sys_thread("pthread_create", result);

if ((result = pthread_join(tid1, NULL)) != 0)
    exit_sys_thread("pthread_join", result);

if ((result = pthread_join(tid2, NULL)) != 0)
    exit_sys_thread("pthread_join", result);

if ((result = pthread_join(tid3, NULL)) != 0)
    exit_sys_thread("pthread_join", result);

return 0;
}

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

    exit(EXIT_FAILURE);
}

void exit_sys_thread(const char *msg, int err)
{
    fprintf(stderr, "%s: %s\n", msg, strerror(err));
    exit(EXIT_FAILURE);
}

void *thread_proc1(void *param)
{
    printf("thread1 waiting at pause...\n");
    pause();
    printf("thread1 resuming...\n");

    return NULL;
}

void *thread_proc2(void *param)
{
    printf("thread2 waiting at pause...\n");
    pause();

```

```

    printf("thread2 resuming...\n");

    return NULL;
}

void *thread_proc3(void *param)
{
    printf("thread3 waiting at pause...\n");
    pause();
    printf("thread3 resuming...\n");

    return NULL;
}

void sigusr1_handler(int sno)
{
    printf("SIGUSR1 occurred...\n");
}

/
*-----
-----
    Bir thread'e gerçek zamanlı sinyal gönderebilmek için ise
    pthread_sigqueue fonksiyonu kullanılmaktadır. Tabii bu fonksiyon da
    bizeden pthread_t aldığı için ancak aynı proses içerisinde
    kullanılabilir. Başka bir procesten başka bir prosesin belli bir
    thread'ine
    gerçek zamanlı sinyali gönderebilmek için bir POSIX fonksiyonu yoktur.
    Ancak Linux sistemlerinde rt_tgsigqueueinfo isimli
    sistem fonksiyonu ile bu yapılabilir. Fakat bu sistem fonksiyonunu
    çağıran sarma bir kütüphane fonksiyonu bulundurulmamıştır.
-----
-----*/

#define _GNU_SOURCE

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <pthread.h>
#include <signal.h>

void exit_sys(const char *msg);
void exit_sys_thread(const char *msg, int err);
void *thread_proc(void *param);
void sigrt_handler(int signo, siginfo_t *info, void *context);

int main(void)
{
    int result;
    pthread_t tid;
    struct sigaction act;
    union sigval val;
    int i;

```



```

act.sa_sigaction = sigrt_handler;
sigemptyset(&act.sa_mask);
act.sa_flags = SA_SIGINFO;

if (sigaction(SIGRTMIN, &act, NULL) == -1)
    exit_sys("sigaction");

if ((result = pthread_create(&tid, NULL, thread_proc, NULL)) != 0)
    exit_sys_thread("pthread_create", result);

sleep(1);

for (i = 0; i < 10; ++i) {
    val.sival_int = i;
    if ((result = pthread_sigqueue(tid, SIGRTMIN, val)) != 0)
        exit_sys_thread("pthread_sigqueue", result);
}

sleep(1);

if ((result = pthread_cancel(tid)) == -1)
    exit_sys_thread("pthread_cancel", result);

if ((result = pthread_join(tid, NULL)) != 0)
    exit_sys_thread("pthread_join", result);

return 0;
}

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

    exit(EXIT_FAILURE);
}

void exit_sys_thread(const char *msg, int err)
{
    fprintf(stderr, "%s: %s\n", msg, strerror(err));

    exit(EXIT_FAILURE);
}

void *thread_proc(void *param)
{
    printf("thread1 waiting at pause...\n");

    for (;;)
        pause();

    return NULL;
}

void sigrt_handler(int signo, siginfo_t *info, void *context)

```

```

{
    printf("SIGRTMIN + 0 occurred with %d code\n", info-
        >si_value.sival_int);
}

```

```

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

```

POSIX sistemlerinde hem prosesin ana şalter görevinde olan bir sinyal maskeleye kümesi (signal mask) hem de thread'lerin sinyal maskeleye kümesi vardır. Eğer sinyal kill fonksiyonuyla prosese gönderilmişse prosesin sinyal maskeleye kümesi dikkate alınmaktadır. Eğer sinyal pthread_kill ile belli bir thread'e gönderiliyorsa bu durumda da thread'in maskeleye kümesi dikkate alınır. Thread'lerin maskeleye kümeleri aynı zamanda prosese gönderilen sinyalin hangi thread tarafından işletileceği konusunda da dolaylı bir etkiye sahiptir. Şöyle ki: Eğer thread ilgili sinyal için bloke edilirse prosese gönderilen bu sinyal artık bu thread tarafından işlenmez. Örneğin programcılar prosese gönderilen sinyalin belli bir thread tarafından işlenmesini istiyorlarsa bu durumda tek bir thread'i sinyale açıp diğer thread'leri bu sinyale kapatabilmektedirler.

Aşağıdaki örnekte ikinci thread dışında tüm thread'ler SIGUSR1 sinyaline kapatılmıştır. Bu durumda diğer terminalden prosese SIGUSR1 sinyali gönderildiğinde bunu iki numaralı thread çalıştıracaktır.

```

-----*/

```

```

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <pthread.h>
#include <signal.h>

void exit_sys(const char *msg);
void exit_sys_thread(const char *msg, int err);
void *thread_proc1(void *param);
void *thread_proc2(void *param);
void *thread_proc3(void *param);
void sigusr1_handler(int sno);

int main(void)
{
    int result;
    pthread_t tid1, tid2, tid3;
    struct sigaction act;
    sigset_t sset;

    act.sa_handler = sigusr1_handler;
    sigemptyset(&act.sa_mask);
    act.sa_flags = 0;

    if (sigaction(SIGUSR1, &act, NULL) == -1)
        exit_sys("sigaction");
}

```

```

    if ((result = pthread_create(&tid1, NULL, thread_proc1, NULL)) != 0)
        exit_sys_thread("pthread_create", result);

    if ((result = pthread_create(&tid2, NULL, thread_proc2, NULL)) != 0)
        exit_sys_thread("pthread_create", result);

    if ((result = pthread_create(&tid3, NULL, thread_proc3, NULL)) != 0)
        exit_sys_thread("pthread_create", result);

    sigemptyset(&sset);
    sigaddset(&sset, SIGUSR1);

    if ((result = pthread_sigmask(SIG_BLOCK, &sset, NULL)) != 0)
        exit_sys_thread("pthread_sigmask", result);

    if ((result = pthread_join(tid1, NULL)) != 0)
        exit_sys_thread("pthread_join", result);

    if ((result = pthread_join(tid2, NULL)) != 0)
        exit_sys_thread("pthread_join", result);

    if ((result = pthread_join(tid3, NULL)) != 0)
        exit_sys_thread("pthread_join", result);

    return 0;
}

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

    exit(EXIT_FAILURE);
}

void exit_sys_thread(const char *msg, int err)
{
    fprintf(stderr, "%s: %s\n", msg, strerror(err));
    exit(EXIT_FAILURE);
}

void *thread_proc1(void *param)
{
    sigset_t sset;
    int result;

    sigemptyset(&sset);
    sigaddset(&sset, SIGUSR1);

    if ((result = pthread_sigmask(SIG_BLOCK, &sset, NULL)) != 0)
        exit_sys_thread("pthread_sigmask", result);

    printf("thread1 waiting at pause...\n");
    pause();
    printf("thread1 resuming...\n");
}

```

```

    return NULL;
}

void *thread_proc2(void *param)
{
    printf("thread2 waiting at pause...\n");
    pause();
    printf("thread2 resuming...\n");

    return NULL;
}

void *thread_proc3(void *param)
{
    sigset_t sset;
    int result;

    sigemptyset(&sset);
    sigaddset(&sset, SIGUSR1);

    if ((result = pthread_sigmask(SIG_BLOCK, &sset, NULL)) != 0)
        exit_sys_thread("pthread_sigmask", result);

    printf("thread3 waiting at pause...\n");
    pause();
    printf("thread3 resuming...\n");

    return NULL;
}

void sigusr1_handler(int sno)
{
    printf("SIGUSR1 occurred...\n");
}

```

```

/
*-----
-----
    Nadiren programcı kritik birtakım işlemler yaparken sinyalleri bloke
    edip sonra açıp pause ile bekleyebilmektedir.

    sprocmask(<sinyalleri bloke et>);

    <kritik kod>

    sprocmask(<sinyalleri aç>);
---> Problem var!
    pause();

```

Programcı sinyalleri açıp pause beklemesini yapmak istediği sırada sinyal prosese gönderilirse henüz akış pause fonksiyona girmeden sinyal teslim edilebilir. Daha sonra akış pause fonksiyonuna girdiğinde buradan çıkılamamış olabilir. Bu problem atomik bir biçimde

sinyalleri açarak pause yapan bir fonksiyonla çözülebilir. İşte sigsuspend fonksiyonu bunu yapmaktadır.

Yukarıda açıklanan temayı uygulayan aşağıdaki kodu inceleyiniz. Prosesin sinyal maskelerini açıp pause ile değil sigsuspend ile beklenmelidir.

```
-----*/

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

void exit_sys(const char *msg);
void sigusr1_handler(int sno);

int main(void)
{
    struct sigaction act;
    sigset_t sset;
    int i;

    act.sa_handler = sigusr1_handler;
    sigemptyset(&act.sa_mask);
    act.sa_flags = 0;

    if (sigaction(SIGUSR1, &act, NULL) == -1)
        exit_sys("sigaction");

    sigfillset(&sset);
    if (sigprocmask(SIG_BLOCK, &sset, NULL) == -1)
        exit_sys("sigprocmask");

    for (i = 0; i < 10; ++i) {
        sleep(1);
        printf("critical region running...\n");
    }

    sigemptyset(&sset);
    if (sigsuspend(&sset) == -1 && errno != EINTR)        /* pause yerine
        sigsuspend kullanılmalı */
        exit_sys("sigsuspend");

    sigfillset(&sset);
    if (sigprocmask(SIG_UNBLOCK, &sset, NULL) == -1)
        exit_sys("sigprocmask");

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

    return 0;
}
```

```

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

    exit(EXIT_FAILURE);
}

void sigusr1_handler(int sno)
{
    printf("SIGUSR1 occurred...\n");
}

```

```

/
*-----
-----
sigwait fonksiyonu programcının belirlediği sinyallerden herhangi biri
oluşana kadar bekleme yapar. Eğer programcının belirlemediği
bir sinyal oluşursa ve sinyal fonksiyonu da set edilmişse fonksiyon
çalıştırılır fakat bekleme devam eder. Eğer programcının
belirlemediği bir sinyal oluşursa fakat bu sinyal için sinyal fonksiyonu
set edilmemişse bu durumda default eylem gerçekleşir (muhtemelen
prosesin sonlandırılması). Eğer sigwait'te beklerken programcının
belirlediği sinyallerden biri oluşursa ve bu sinyal için
sinyal fonksiyonu set edilmişse sinyal fonksiyonu çağrılmaz ancak bu
sinyal için sinyal fonksiyonu set edilmemişse default eylem
uygulanır (muhtemelen prosesin sonlandırılması). Eğer sigwait
çağrıldığında zaten beklelen sinyallerin bazıları pending durumdaysa
sigwait
hiç bekleme yapmaz.

sigwait fonksiyonun sigwaitinfo isimli siginfo_t parametrelili versiyonu
da vardır. Bu fonksiyon yalnız oluşan sinyalin numarasını değil onun
siginfo_t
bilgilerini de vermektedir.

Aşağıdaki örnekte sigwait fonksiyonunda SIGUSR1 ve SIGUSR2 sinyalleri
beklenmektedir. Başka bir sinyal oluşursa proses
sonlandırılır. SIGUSR1 ve SIGUSR2 için sinyal fonksiyonu set edilmiştir.
Bu durumda bu sinyaller oluşursa sinyal fonksiyonu çağrılmadan
bekleme sonlandırılır. pause ve sisuspend fonksiyonlarının sinyal
fonksiyonun çalışmasına yol açtığını anımsayınız.
-----
-----*/

```

```

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

void exit_sys(const char *msg);
void sigusr1_handler(int sno);
void sigusr2_handler(int sno);

int main(void)

```

```

{
    struct sigaction act;
    sigset_t sset;
    int result;
    int signo;

    act.sa_handler = sigusr1_handler;
    sigemptyset(&act.sa_mask);
    act.sa_flags = 0;

    if (sigaction(SIGUSR1, &act, NULL) == -1)
        exit_sys("sigaction");

    act.sa_handler = sigusr2_handler;
    if (sigaction(SIGUSR2, &act, NULL) == -1)
        exit_sys("sigaction");

    sigemptyset(&sset);
    sigaddset(&sset, SIGUSR1);

    printf("process waiting for SIGUSR1 signal...\n");

    if ((result = sigwait(&sset, &signo)) != 0) {
        fprintf(stderr, "sigwait: %s\n", strerror(result));
        exit(EXIT_FAILURE);
    }

    if (signo == SIGUSR1)
        printf("sigwait returns due to SIGUSR1\n");
    else
        printf("sigwait returns due to SIGINT\n");

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

    return 0;
}

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

    exit(EXIT_FAILURE);
}

void sigusr1_handler(int sno)
{
    printf("SIGUSR1 occurred...\n");
}

void sigusr2_handler(int sno)
{
    printf("SIGUSR2 occurred...\n");
}

```

```
/
*-----
sleep fonksiyonu parametresiyle belirtilen saniye kadar thread'i bloke
bekletir. Fonksiyon zamandan dolayı sonlanırsa
0 değerine sinyal dolayısıyla sonlanırsa kalan saniye değerine geri
döner. sleep yeniden başlatılabilen (restartable) bir
fonksiyon değildir.

Aşağıdaki örnekte SIGINT sinyali gelse bile sleep ile 10 saniye
civarında bekleme yapılmıştır.
-----*/
```

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

void exit_sys(const char *msg);
void sigint_handler(int sno);

int main(void)
{
    struct sigaction act;
    int t;

    act.sa_handler = sigint_handler;
    sigemptyset(&act.sa_mask);
    act.sa_flags = 0;

    if (sigaction(SIGINT, &act, NULL) == -1)
        exit_sys("sigaction");

    printf("waiting 10 seconds even if SIGINT occurs...\n");

    t = 10;
    while ((t = sleep(t)) != 0)
        ;

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

    return 0;
}

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

    exit(EXIT_FAILURE);
}

void sigint_handler(int sno)
{

```



```

    printf("SIGINT occurred...\n");
}

/
*-----
-----
    nanosleep nano saniye (saniyenin m,lyarda biri) çözünürlüğüne sahip bir
    POSIX bekleme fonksiyonudur. Fonksiyon timespec
    isimli yapı nesnesini parametre olarak alır. Sinyal oluşursa sonlanır ve
    kalan zamanı da yine bize timespec biçiminde verir.
    Her ne kadar bu fonksiyon nanosaniye çözünürlüğüne sahipse de işletim
    sistemleri genellikle bu çözünürlüğü destekleyememektedir.
    Bu durumda bekleme tam istenen miktarda gerçekleşmez. Örneğin Linux
    işletim sistemlerinde sleep kuyruğundaki thread'ler timer
    kesmesinde kontrol edilmektedir. Dolayısıyla genellikle bu çözünürlük 1
    milisaniyedir.
-----
-----*/

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

void exit_sys(const char *msg);

int main(void)
{
    struct timespec ts;

    printf("waiting 3.5 seconds...\n");
    ts.tv_sec = 3;
    ts.tv_nsec = 500000000;

    if (nanosleep(&ts, NULL) == -1)
        exit_sys("nanosleep");

    printf("sleep finished..\n");

    return 0;
}

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

    exit(EXIT_FAILURE);
}

/
*-----
-----
    Ayrıca POSIX standartlarından 2008'de çıkartılmış olan mikrosaniye
    çözünürlüğüne sahip bir usleep fonksiyonu da vardır. Bu
    fonksiyon Linux sistemlerinde hala desteklenmektedir.

```

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

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

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

```
int main(void)
{
    printf("waiting 3.5 seconds even if SIGINT occurs...\n");

    if (usleep(3500000) == -1)
        exit_sys("usleep");

    printf("sleep finished..\n");

    return 0;
}
```

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

    exit(EXIT_FAILURE);
}
```

```
/
```

```
*-----
```

Amacımız bekleme yapmak değil de zaman ölçmek ise ilk akla gelen POSIX fonksiyonu clock_gettime olmalıdır. Bu fonksiyonun birinci parametresi clockid_t türündendir. Hesabın yapılacağı saat cinsini belirtir. Bu parametre CLOCK_REALTIME biçiminde girilirse 01/01/1970'ten geçen nano saniye sayısı bize verilir. Tabii biz buradan elde ettiğimiz mutlak zamanı C'nin klasik zaman fonksiyonlarına sokaup dönüştürmeler yapabiliriz. Saat ürü olarak CLOCK_MONOTONIC değeri geçilirse belli bir zamandan itibaren geçen görelî zaman elde edilir. Linux sistemleri bu durumda boot zamanından itibaren geçen zamanı bize vermektedir.

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

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

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

```
int main(void)
{
    struct timespec ts;
    struct tm *pt;
```

```

    if (clock_gettime(CLOCK_REALTIME, &ts) == -1)
        exit_sys("clock_gettime");

    printf("Seconds: %ld\n", (long)ts.tv_sec);
    printf("Nanoseconds: %ld\n", (long)ts.tv_nsec);

    puts(ctime(&ts.tv_sec));

    pt = localtime(&ts.tv_sec);
    printf("%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);

    return 0;
}

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

    exit(EXIT_FAILURE);
}

/
*-----
-----
    Programın iki noktası arasında geçen gerçek zamanı taşınabilir bir
    biçimde ölçmek için ilk akla gelecek yöntem clock_gettime
    olmalıdır. Tabii benzer başka yöntemler de kullanılabilir.
-----
-----*/

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

void exit_sys(const char *msg);

int main(void)
{
    struct timespec ts1, ts2;
    int i;
    double result;

    if (clock_gettime(CLOCK_REALTIME, &ts1) == -1)
        exit_sys("clock_gettime");

    for (i = 0; i < 1000000000; ++i)
        ;

    if (clock_gettime(CLOCK_REALTIME, &ts2) == -1)
        exit_sys("clock_gettime");

    result = (ts2.tv_sec - ts1.tv_sec) * 1000000000.0 + ts2.tv_nsec -
        ts1.tv_nsec;
    printf("Elapsed time (nanoseconds): %.0f\n", result);
}

```

```

    printf("Elapsed time (seconds): %f\n", result / 1000000000.0);

    return 0;
}

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

    exit(EXIT_FAILURE);
}

/
*-----
-----
    clock isimli standart C fonksiyonu bize zamanı clock_t türünden verir.
    Ancak geçen zamanın saniye cinsine dönüştürülmesi
    için CLOCKS_PER_SEC değerine bölünmesi gerekmektedir. Genellikle bu
    fonksiyonlar işletim sisteminin timer kesmesi ile tick
    sayısını verecek biçimde yazılmışlardır. Bu durumda zaman ölçmenin C
    genelinde en taşınabilir yolu clock fonksiyonunu
    kullanmak olabilir. Ancak genel olarak clock_gettime fonksiyonunun
    çözünürlüğünün daha yüksek olduğu varsayılmalıdır.
-----
-----*/

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

void exit_sys(const char *msg);

int main(void)
{
    clock_t c1, c2;
    int i;
    double result;

    c1 = clock();

    for (i = 0; i < 1000000000; ++i)
        ;

    c2 = clock();

    result = (double)(c2 - c1) / CLOCKS_PER_SEC;
    printf("Elapsed time (seconds): %f\n", result);

    return 0;
}

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

```

```

    exit(EXIT_FAILURE);
}

/
*-----
-----
    clock_gettime fonksiyonundaki duyarlılık clock_getres fonksiyonuyla elde
    edilebilir. Pek çok işlemcide zamansal nano saniye
    çözünürlük için özel makine komutları bulunmaktadır. Örneğin x64
    işlemcide çalışan sanal makinede bu değer 1 nanosaniye çıkmıştır.
-----
-----*/

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

void exit_sys(const char *msg);

int main(void)
{
    struct timespec ts;

    if (clock_getres(CLOCK_REALTIME, &ts) == -1)
        exit_sys("clock_getres");

    printf("%ld second and %ld nano seconds:\n", (long)ts.tv_sec,
        (long)ts.tv_nsec);

    return 0;
}

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

    exit(EXIT_FAILURE);
}

/
*-----
-----
    clock_gettime fonksiyonunda clockid_t olarak CLOCK_PROCESS_CPUTIME_ID
    geçilirse prosesin yalnızca CPU'da harcadığı zamanların
    toplamı verilir. Bu gerçek zamandan daha kısa olacaktır. eğer proses çok
    thread'ten oluşuyorsa bu hesaba tüm thread'lerin CPU zamanları
    toplanır.
    Fakat clockid_t olarak CLOCK_THREAD_CPUTIME_ID verilirse bu da spesifik
    bir thread'in (fonksiyonu çağıran) CPU zamanını ölçmekte kullanılır.
-----
-----*/

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

```

```

#include <unistd.h>

void exit_sys(const char *msg);

int main(void)
{
    struct timespec ts1, ts2;
    int i;
    double result;

    if (clock_gettime(CLOCK_PROCESS_CPUTIME_ID, &ts1) == -1)
        exit_sys("clock_gettime");

    for (i = 0; i < 10; ++i)
        sleep(1);

    if (clock_gettime(CLOCK_PROCESS_CPUTIME_ID, &ts2) == -1)
        exit_sys("clock_gettime");

    result = (ts2.tv_sec - ts1.tv_sec) * 1000000000.0 + ts2.tv_nsec -
        ts1.tv_nsec;
    printf("Elapsed time (nonosecands): %.0f\n", result);
    printf("Elapsed time (seconds): %f\n", result / 1000000000.0);

    return 0;
}

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

    exit(EXIT_FAILURE);
}

/
*-----
-----
    Başka bir prosese ilişkin clockid_t elde etmek mümkündür. Bu durumda o
    prosese ilişkin zamansal ölçümler yapılabilir.
    Bunun için clock_getcpuclockid POSIX fonksiyonu kullanılmaktadır. Eğer
    spesifik bir thread'e ilişkin clockid_t elde etmek için ise
    pthread_getcpuclockid fonksiyonu kullanılmaktadır.
    -----
    -----*/

/
*-----
-----
    İlgili thread'i duyarlılıklı bir biçimde bekletmek için nanosleep
    fonksiyonu görmüştük. Her ne kadar POSIX standartlarından kaldırılmış
    olsa da Linux sistemlerinde desteklenen mikrosaniye duyarlılıklı bir
    usleep fonksiyonu da vardı. Klasik sleep fonksiyonu ise eski bir
    fonksiyondur ve saniye duyarlılığına sahipti. İşte nanosleep
    fonksiyonunun clock_nanosleep isimli biraz daha gelişmiş bir biçimi
    vardır.

```

Fonksiyonun bu versiyonu clockid_t aldığı için daha yeteneklidir. clock_nanosleep fonksiyonunun nanosleep fonksiyonundan en önemli avantajı gerçek bekleme zamanını ölçebilmesidir. Eğer bekleme nanosleep fonksiyonuyla yapılırsa ve programda sinyal de kullanılıyorsa nanosleep sinyal geldiğinde başarısızlıkla sonlandırılacak (errno = EINTR) programcı da kalan süreyi alarak yeniden nanosleep fonksiyonunu çağırıp beklemeye devam edecektir. Ancak her sinyal geldiğinde sinyal fonksiyonun çalıştırılması zamanı ekstra bir zaman olarak bekleyemeye eklenecektir.

Aşağıdaki örnekte programcı nanosleep kullanarak sinyalli bir ortamda 10 saniye beklemek istemiştir. Ancak SIGINT sinyali oluştuğunda bu toplam bekleme gerçek zamana göre fazlalaşacaktır.

-----*/

```
stdio.h>
#include <stdlib.h>
#include <time.h>
#include <errno.h>
#include <signal.h>

void exit_sys(const char *msg);
void sigint_handler(int sno);

int main(void)
{
    struct sigaction act;
    struct timespec ts, rm;
    struct timespec ts1, ts2;
    double result;

    act.sa_handler = sigint_handler;
    sigemptyset(&act.sa_mask);
    act.sa_flags = 0;

    if (sigaction(SIGINT, &act, NULL) == -1)
        exit_sys("sigaction");

    printf("waiting 10 seconds...\n");
    ts.tv_sec = 10;
    ts.tv_nsec = 0;

    if (clock_gettime(CLOCK_REALTIME, &ts1) == -1)
        exit_sys("clock_gettime");

    while (nanosleep(&ts, &rm) == -1 && errno == EINTR)
        ts = rm;

    if (clock_gettime(CLOCK_REALTIME, &ts2) == -1)
        exit_sys("clock_gettime");
```

```

    result = (ts2.tv_sec - ts1.tv_sec) * 1000000000.0 + ts2.tv_nsec -
        ts1.tv_nsec;
    printf("Elapsed time (nonosecands): %.0f\n", result);
    printf("Elapsed time (seconds): %f\n", result / 1000000000.0);

    return 0;
}

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

    exit(EXIT_FAILURE);
}

void sigint_handler(int sno)
{
    int i;

    for (i = 0; i < 1000000000; ++i)
        ;
}

/
*-----
-----
    clock_nanosleep fonksiyonun nanosleep fonksiyonundan en önemli avantajı
    sinyalli ortamda timer türü TIMER_ABSTIME alınarak
    umulan beklemeyi yapabilmesidir.
-----
-----*/

#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <errno.h>
#include <signal.h>

void exit_sys(const char *msg);
void sigint_handler(int sno);

int main(void)
{
    struct sigaction act;
    struct timespec ts;
    struct timespec ts1, ts2;
    double elapsed;
    int result;

    act.sa_handler = sigint_handler;
    sigemptyset(&act.sa_mask);
    act.sa_flags = 0;

    if (sigaction(SIGINT, &act, NULL) == -1)
        exit_sys("sigaction");

```



```

if (clock_gettime(CLOCK_REALTIME, &ts) == -1)
    exit_sys("clock_gettime");

ts.tv_sec += 10;

if (clock_gettime(CLOCK_REALTIME, &ts1) == -1)
    exit_sys("clock_gettime");

while ((result = clock_nanosleep(CLOCK_REALTIME, TIMER_ABSTIME, &ts,
    NULL)) == EINTR)
    ;

if (clock_gettime(CLOCK_REALTIME, &ts2) == -1)
    exit_sys("clock_gettime");

elapsed = (ts2.tv_sec - ts1.tv_sec) * 1000000000.0 + ts2.tv_nsec -
    ts1.tv_nsec;
printf("Elapsed time (nonosecands): %.0f\n", elapsed);
printf("Elapsed time (seconds): %f\n", elapsed / 1000000000.0);

return 0;
}

```

```

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

    exit(EXIT_FAILURE);
}

```

```

void sigint_handler(int sno)
{
    int i;

    for (i = 0; i < 1000000000; ++i)
        ;
}

```

```

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

```

Periyodik bir biçimde iş yapılmasını sağlayan timer'lara İngilizce "interval timer" denilmektedir. POSIX sistemlerinde iki interval timer mekanizması vardır. setitimer ile oluşturulan interval timer'ın kullanımı kolaydır. Ancak kullanımı zor olan daha yetenekli interval timer mekanizması POSIX standartlarına eklendiği için bu fonksiyon "obsolete" yapılmıştır. setitimer fonksiyonunda interval timer için periyodik işleme başlamak için gereken zaman ile periyot zamanı ayrı ayrı verilir. Yine interbval timer'ın türleri vardır. Her tür zaman dolduğunda farklı bir sinyalin oluşmasına yol açar. ITIMER_REAL timer'ı SIGALRM sinyaline yol açar ve gerçek zamanlı timer'dır. Prosein 3 tüden tek farklı farklı tek interval timer'ları vardır. Fonksiyon ikinci kez çağrıldığında eski timer rest edilir.

Oradaki değerler programcıya verilebilmektedir. Ayrıca getitimer isimli fonksiyon da mevcut interval timer'ın timer değerlerini almak için kullanılabilir.

Aşağıdaki örnekte periyodik işleme 5 saniye sonra başlanıp birer saniye periyotlu işlem yapılmaktadır.

```
-----*/

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

void exit_sys(const char *msg);
void sigalrm_handler(int sno);

int main(void)
{
    struct itimerval itv;
    struct sigaction act;

    act.sa_handler = sigalrm_handler;
    sigemptyset(&act.sa_mask);
    act.sa_flags = 0;

    if (sigaction(SIGALRM, &act, NULL) == -1)
        exit_sys("sigaction");

    itv.it_interval.tv_sec = 1;
    itv.it_interval.tv_usec = 0;

    itv.it_value.tv_sec = 5;
    itv.it_value.tv_usec = 0;

    if (setitimer(ITIMER_REAL, &itv, NULL) == -1)
        exit_sys("setitimer");

    for (;;)
        pause();

    return 0;
}

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

    exit(EXIT_FAILURE);
}

void sigalrm_handler(int sno)
{
    static int i = 0;
```

```

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

if (i == 10) {
    printf("\n");
    exit(EXIT_SUCCESS);
}

++i;
}

```

```

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

```

Periyodik timer (interval timers) yaratmak için yukarıda setitimer fonksiyonunu kullanmıştık. Bu fonksiyon POSIX standartlarında artık "obsolete" ilan edilmiştir. Dolayısıyla gelecekte standartlardan kaldırılabilir. Bunun yerine POSIX'e daha yetenekli ama kullanılması daha zor olan bir grup yeni periyodik timer fonksiyonu eklenmiştir.

Modern periyodik timer mekanizmasını kullanabilmek için önce timer'ın timer_create fonksiyonu ile yaratılması gerekir. timer_create fonksiyonu bizden sigevent türünden bir yapı nesnesinin adresini ister. O yapı nesnesini bizim tanımlayıp içini bizim doldurmamız gerekir. sigevent yapısının sigev_notify elemanı periyot bittiğinde bizim nasıl haberdar edileceğimizi belirtir. Eğer bu elemana SIGEV_SIGNAL değeri girilirse biz sinyal yoluyla haberdar edilirdik. Ancak söz konusu sinyal SIGALRM olmak zorunda değildir. Söz konusu sinyalin numarası yapının sigev_signo elemanına girilmelidir. Eğer sigev_notify elemanı SIGEV_THREAD olarak girilirse işletim sistemi bir thread yaratıp bizim yapının sigev_notify_function elemanı ile belirttiğimiz fonksiyonunu bu thread akışıyla çağırarak bizi haberdar eder. Ancak işletim sisteminin tek bir thread'le mi bu işi yapacağı yoksa her çağrı için ayrı bir thread mi kullanacağı sistemden sisteme değişebilmektedir.

Bu thread'in yaratılmasındaki thread özellikleri yapının sigev_notify_attributes elemanına girilebilir. Aynı zamanda istenirse oluşan sinyale bir değer de iliştilerilebilmektedir. Bu değer sigev_value elemanına yerleştirilir ve sigaction ile siginfo_t parametrelili sinyal fonksiyonu ile elde edilir. timer_create bir timer'ı oluşturur ve onun id'sini fonksiyonun sigev_value timerid parametresine yerleştirir. Bu id sonraki fonksiyonlarda kullanılacaktır.

Timer yaratıldıktan sonra timer'ı kurma işlemi timer_settime fonksiyonuyla yapılmaktadır. Burada yine ilk periyota kadar geçen zaman ve periyot zamanı belirtilmektedir.

Timer kullanıldıktan sonra timer_delete fonksiyonuyla silinir. Yaratılan timer'lar fork işlemi sırasında alt süreçte etkinliğini kaybetmektedir. Yine exec işlemi yapıldığında timer'lar yok edilmektedir.

Aşağıdaki örnekte sinyal yoluyla (SIGUSR1) haberdar edilme yöntemi kullanılmıştır.

```
-----*/

#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <setjmp.h>
#include <unistd.h>
#include <signal.h>

void exit_sys(const char *msg);
void sigusr1_handler(int sno);

jmp_buf g_jb;

int main(void)
{
    struct sigaction act;
    struct sigevent se;
    timer_t mytimer;
    struct itimerspec tspec;

    act.sa_handler = sigusr1_handler;
    sigemptyset(&act.sa_mask);
    act.sa_flags = 0;

    if (sigaction(SIGUSR1, &act, NULL) == -1)
        exit_sys("sigaction");

    se.sigev_notify = SIGEV_SIGNAL;
    se.sigev_signo = SIGUSR1;
    se.sigev_value.sival_int = 0;

    if (timer_create(CLOCK_REALTIME, &se, &mytimer) == -1)
        exit_sys("timer_create");

    tspec.it_value.tv_sec = 5;
    tspec.it_value.tv_nsec = 0;

    tspec.it_interval.tv_sec = 1;
    tspec.it_interval.tv_nsec = 0;

    if (timer_settime(mytimer, 0, &tspec, NULL) == -1)
        exit_sys("timer_settime");

    if (setjmp(g_jb) == 1) {
        if (timer_delete(mytimer) == -1)
            exit_sys("timer_delete");
        exit(EXIT_SUCCESS);
    }

    for (;;)
        pause();
}
```

```

    return 0;
}

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

    exit(EXIT_FAILURE);
}

void sigusr1_handler(int sno)
{
    static int count = 0;

    if (count == 10)
        longjmp(g_jb, 1);

    printf("sigusr1 occurred...\n");

    ++count;
}

/
*-----
-----
    Eğer biz sinyal fonksiyonuna değer de aktarmak istiyorsak bu durumda
    siginfo_t * parametrelili sinyal fonksiyonu set etmeliyiz.
-----
-----*/

#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <setjmp.h>
#include <unistd.h>
#include <signal.h>

void exit_sys(const char *msg);
void sigusr1_handler(int sno, siginfo_t *info, void *ucont);

jmp_buf g_jb;

int main(void)
{
    struct sigaction act;
    struct sigevent se;
    timer_t mytimer;
    struct itimerspec tspec;

    act.sa_sigaction = sigusr1_handler;
    sigemptyset(&act.sa_mask);
    act.sa_flags = SA_SIGINFO;

    if (sigaction(SIGUSR1, &act, NULL) == -1)

```

```

        exit_sys("sigaction");

se.sigev_notify = SIGEV_SIGNAL;
se.sigev_signo = SIGUSR1;
se.sigev_value.sival_int = 100;

if (timer_create(CLOCK_REALTIME, &se, &mytimer) == -1)
    exit_sys("timer_create");

tspec.it_value.tv_sec = 5;
tspec.it_value.tv_nsec = 0;

tspec.it_interval.tv_sec = 1;
tspec.it_interval.tv_nsec = 0;

if (timer_settime(mytimer, 0, &tspec, NULL) == -1)
    exit_sys("timer_settime");

if (setjmp(g_jb) == 1) {
    if (timer_delete(mytimer) == -1)
        exit_sys("timer_delete");
    exit(EXIT_SUCCESS);
}

for (;;)
    pause();

return 0;
}

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

    exit(EXIT_FAILURE);
}

void sigusr1_handler(int sno, siginfo_t *info, void *ucontext)
{
    static int count = 0;

    if (count == 10)
        longjmp(g_jb, 1);

    printf("sigusr1 occurred: %d\n", info->si_value.sival_int);

    ++count;
}

```

/

*-----

Periyot dolduğunda bir sinyalin oluşması yerine belirlenen bir fonksiyonun çağrılması da sağlanabilir. Bunun için yapının

sigev_notify elemanı SIGEV_THREAD biçiminde girilmelidir. Çağrılacak fonksiyon da yapının sigev_notify_function elemanına girilir. İşletim sistemi kendisi bir thread yaratıp bizim girdiğimiz fonksiyonu o thread akışının çalıştırmasını sağlamaktadır. Linux bir kere thread yaratıp hep aynı thread'le sinyal fonksiyonu çağırılmaktadır.

```
-----*/

#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <setjmp.h>
#include <unistd.h>
#include <signal.h>

void exit_sys(const char *msg);
void signal_notification_handler(union sigval val);

jmp_buf g_jb;

int main(void)
{
    struct sigevent se;
    timer_t mytimer;
    struct itimerspec tspec;

    se.sigev_notify = SIGEV_THREAD;
    se.sigev_notify_function = signal_notification_handler;
    se.sigev_notify_attributes = NULL;
    se.sigev_value.sival_int = 100;

    if (timer_create(CLOCK_REALTIME, &se, &mytimer) == -1)
        exit_sys("timer_create");

    tspec.it_value.tv_sec = 5;
    tspec.it_value.tv_nsec = 0;

    tspec.it_interval.tv_sec = 1;
    tspec.it_interval.tv_nsec = 0;

    if (timer_settime(mytimer, 0, &tspec, NULL) == -1)
        exit_sys("timer_settime");

    if (setjmp(g_jb) == 1) {
        if (timer_delete(mytimer) == -1)
            exit_sys("timer_delete");
        exit(EXIT_SUCCESS);
    }

    for (;;)
        pause();

    return 0;
}
```

```

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

    exit(EXIT_FAILURE);
}

```

```

void signal_notification_handler(union sigval val)
{
    static int count = 0;

    if (count == 10)
        longjmp(g_jb, 1);

    printf("signal notification function called: %d\n", val.sival_int);

    ++count;
}

```

```

/
*-----
-----
setitimer fonksiyonu ile her cins timer'dan yalnızca bir tane
yaratılabilmektedir. Oysa bu modern timer mekanizmasında
istenildiği kadar çok timer yaratılabilir. Ayrıca haberdar edilme
yönteminin sinyal yoluyla yapılması durumunda girilen sinyal
numarası realtime bir sinyal numarası olsa bile kuyruklama
yapılmamaktadır. Yani örneğin periyot kısa olabilir. Bu süre içerisinde
sinyal fonksiyonu sonlanmamış olabilir. Ancak yalnızca 1 tane sinyal
pending durumda bekleyecektir. Fakat programcı isterse
aslında kaç periyodun geçmiş olduğunu timer_getoverrun isimli
fonksiyonla istediği zaman alabilir. Bu fonksiyon aynı biçimde
fonksiyon yoluyla haberdar edilmede de kullanılır.

```

```

-----*/
/
*-----
-----
Bir grup prosesin oluşturduğu gruba "proses grubu" denilmektedir. Proses
grubu kavramı bir grup prosese sinyal gönderebilmek için
uydurulmuştur. Gerçekten de kill sistem fonksiyonunun birinci
parametresi olan pid sıfırdan bir küçük bir sayı olarak girilirse
abs(pid) numaralı proses grubuna sinyal gönderilmektedir. Bir sinyal bir
proses grubuna gönderilirse o proses grubunun bütün üyeleri olan
proseslere gönderilmiş olur. kill fonksiyonunun birinci parametresi 0
girilirse bu durumda sinyal kill fonksiyonunu uygulayan prosesin
proses grubuna gönderilir. Yani proses kendi proses grubuna sinyali
göndermektedir.

```

Bir proses grubunun id'si vardır. Proses gruplarının id'si o proses grubundaki bir prosesin proses id'si ile aynıdır. İşte proses id'si proses grup id'sine eşit olan prosese o proses grubunun "proses grup lideri (process group leader)" denilmektedir.

Proses grup lideri genellikle proses grubunu yaratan procestir. Alt prosesin proses grubu onu yaratan üst procesten alınmaktadır.

Bir prosesin ilişkin olduğu proses grubunun id'sini alabilmek için getpgrp ya da getpgid POSIX fonksiyonları kullanılır.

```
pid_t getpgrp(void);
pid_t getpgid(pid_t pid);
```

getpgid fonksiyonu herhangi bir prosesin proses grup id'sini almakta kullanılabilir. Bu fonksiyonun parametresi 0 geçilirse fonksiyonu çağıran prosesin proses grup id'si alınmış olur Yani aşağıdaki çağrı eşdeğerdir:

```
pgid = getpgrp();
pgid = getpgid();
```

Proses grup lideri sonlanmış olsa bile proses grubunun id'si aynı biçimde kalmaya devam eder. Proses grubu gruptaki son prosesin sonlanması ya da grup değiştirmesiyle ömrünü tamamlamaktadır.

Bir programı kabul üzerinden çalıştırdığımızda kabuk yeni bir proses grubu oluşturur ve çalıştırılan programa ilişkin proses o proses grubunun proses grup lideri yapar. Aşağıdaki programı çalıştırıp sonucunu inceleyiniz. Yapılan denemeden şöyle bir sonuç elde edilmiştir:

```
Parent process id: 9921
Parent process id of the parent: 1894
Parent process group id: 9921
Child process id: 9922
Parent process id of the child: 9921
Child process group id: 9921
```

-----*/

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

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

```
int main(void)
{
    pid_t pgid;
    pid_t pid;

    if ((pid = fork()) == -1)
        exit_sys("fork");

    if (pid != 0) {        /* parent process */
        printf("Parent process id: %ld\n", (long)getpid());
```

```

    printf("Parent process id of the parent: %ld\n", (long)getppid());
    pgid = getpgrp();
    printf("Parent process group id: %ld\n", (long)pgid);

    if (waitpid(pid, NULL, 0) == -1)
        exit_sys("waitpid");
}
else { /* child process */
    sleep(1);
    printf("Child process id: %ld\n", (long)getpid());
    printf("Parent process id of the child: %ld\n", (long)getppid());
    pgid = getpgrp();
    printf("Child process group id: %ld\n", (long)pgid);
}

return 0;
}

```

```

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

    exit(EXIT_FAILURE);
}

```

/

*-----

Yeni bir proses grubu yaratmak için ya da bir prosesin proses grubunu değiştirmek için setpgid POSIX fonksiyonu kullanılmaktadır.

```
int setpgid(pid_t pid, pid_t pgid);
```

Fonksiyonun birinci parametresi proses grup id'si değiştirilecek prosesi ikinci parametresi de hedef proses grup id'sini belirtmektedir. Eğer bu iki parametre aynı ise yeni bir proses grubu yaratılır ve bu yeni grubun lideri de buradaki proses olur. Bir proses (root olsa bile) ancak kendisinin ya da kendi alt proseslerinin proses grup id'lerini değiştirebilir. Ancak üst proses alt proses exec uyguladıktan sonra onun proses grup id'sini artık değiştirememektedir. setpgid fonksiyonu ile proses kendisinin ya da alt proseslerinin proses grup id'lerini aynı "oturum (session)" içerisindeki bir proses grup id'si olarak değiştirebilmektedir.

Örneğin kabuk çalıştırdığımız programı yeni proses grubunun grup lideri şöyle yapmaktadır: Kabuk önce fork yapar. Üst ya da alt procese setpgid fonksiyonuyla yeni proses grubunu oluşturup alt prosesin bu grubun lideri olmasını sağlar.

-----*/

```

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

```

```

#include <sys/wait.h>

void exit_sys(const char *msg);

int main(void)
{
    pid_t pgid;
    pid_t pid;

    if ((pid = fork()) == -1)
        exit_sys("fork");

    if (pid != 0) { /* parent process */
        printf("Parent process id: %ld\n", (long)getpid());
        printf("Parent process id of the parent: %ld\n", (long)getppid());
        pgid = getpgrp();
        printf("Parent process group id: %ld\n", (long)pgid);

        if (waitpid(pid, NULL, 0) == -1)
            exit_sys("waitpid");
    }
    else { /* child process */
        sleep(1);
        if (setpgid(getpid(), getpid() /* 0 */ ) == -1)
            exit_sys("setpgid");

        printf("Child process id: %ld\n", (long)getpid());
        printf("Parent process id of the child: %ld\n", (long)getppid());
        pgid = getpgrp();
        printf("Child process group id: %ld\n", (long)pgid);
    }

    return 0;
}

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

    exit(EXIT_FAILURE);
}

```

*-----

Yukarıda da belirtildiği gibi kill POSIX fonksiyonuyla (ya da kill komutuyla) bir proses grubuna sinyal gönderildiğinde aslında proses grubundaki tüm proseslere sinyal gönderilmektedir. Bunu aşağıdaki programla test edebilirsiniz. Başka bir terminalden girip önce üst ve alt proseslerin id'lerini aşağıdaki komut ile elde ediniz:

```
ps -a -o pid,ppid,pgid,cmd
```

Sonra da proses grubuna kill komutuyla SIGUSR1 sinyalini gönderiniz:

```
kill -USR1 -<proses grup id>
```

Proses grup id'yi eksi değer olarak yazmayı unutmayınız.

```
-----*/

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

void exit_sys(const char *msg);
void sigusr1_handler(int sno);

int main(void)
{
    pid_t pid;
    struct sigaction sa;

    sa.sa_handler = sigusr1_handler;
    sigemptyset(&sa.sa_mask);
    sa.sa_flags = 0;

    if (sigaction(SIGUSR1, &sa, NULL) == -1)
        exit_sys("siagaction");

    if ((pid = fork()) == -1)
        exit_sys("fork");

    if (pid != 0) { /* parent process */
        printf("parent waitint at pause\n");
        pause();
        if (waitpid(pid, NULL, 0) == -1)
            exit_sys("waitpid");
    }
    else { /* child process */
        printf("child waitint at pause\n");
        pause();
    }

    return 0;
}

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

    exit(EXIT_FAILURE);
}

void sigusr1_handler(int sno)
{
    printf("SIGUSR1 occurred in process %ld\n", (long)getpid());
}
```

}

/

*-----

Kabuk ile birden fazla programı boru eşliğinde çalıştırdığımızı düşünelim. Örneğin:

```
cat | grep "xxx"
```

Burada kabuk cat ve grep için fork yapar. Bu iki proses kardeştir. Ancak aralarında ğstlük-altlık ilişkisi yoktur.

Kabuk bu iki prosesi aynı gruba sokar ve birinci prosesi de (burada cat) bu grubun grup lideri yapar. Bu komutu bir terminalden çalıştırıp diğer bir terminalde aşağıdaki komutla durumu gözlemleyiniz:

```
ps -a -o pid,ppid,pgid,cmd
```

İşte bu durumda Ctrl+C ve Ctrl + Delete tuşları SIGINTR ve SIGQUT sinyallerini bu proses grubuna yollayacak böylece buradaki iki proses de (eğer sinyali sinyali işlememişlerse) sonlanacaktır.

-----*/

/

*-----

Oturum (session) arka plan çalışmayı düzene sokmak için uydurulmuş bir kavramdır. Bir oturum proses gruplarından oluşur.

Oturumu oluşturan proses gruplarından yalnızca biri "ön plan (foreground)", diğerlerinin hepsi "arka plan (background)" gruplarıdır. İşte aslında klavye sinyallerini terminal sürücüsü (tty) oturumun ön plan proses grubuna yollamaktadır.

Kabuk üzerinden bir komut yazıp sonuna & karakteri getirilirse bu karakter "bu komutu arka planda çalıştır" anlamına gelir.

Böylece kabul komutu wait ile beklemez. Yeniden prompt'a düşer. Ancak o komut çalışmaya devam etmektedir. İşte kabuk & ile çalıştırılan prosesleri oturumun arka plan prosesi durumuna getirir.

Sonunda & olmadan çalıştırılan prosesler ise ön plan proses grubunu oluşturur. Aslında kabuk da ön plan proses grubunun içerisinde. Dolayısıyla kabuğun kendisi de aynı oturumdadır.

O halde durum özetle şöyledir: Aslında kabuk bir oturum yaratıp kendini oturumun lideri yapmıştır. Sonra kabuk sonu & ile biten komutlar için yarattığı proses guplarını oturumun arka plan proses grupları yapar. Sonunda & olmayan komutları da oturumun ön plan proses grubu yapmaktadır. Terminal sürücüsü de oturumun ön plan proses grubuna SIGINT ve SIGQUT sinyallerini göndermektedir.

Oturumların da proses gruplarında olduğu gibi id'leri vardır.

Oturumların id'leri oturum içerisindeki bir proses grubunun liderinin

id'si ile aynıdır. Oturum terminal sürücüsüyle ilişkili bir kavram olarak uydurulmuştur. Dolayısıyla oturumların bir "ilişkin olduğu terminal (controlling terminal)" vardır. Bu terminal gerçek terminal ise tty terminallerinde biridir. Sahte (Psudo) bir terminal ise pts terminallerinden biridir. Pencere yöneticilerinin içerisinde açılan terminaller sahte terminallerdir. Ancak işlev olarak gerçek terminallerden bir farkları yoktur. Klavyeden Ctrl+C ve Ctrl+Backspace tuşlarına basıldığında SIGINTR ve SIGQUIT sinyalleri bu terminal tarafından (aslında terminal sürücüsü tarafından) oturumun ön plan proses grubuna gönderilmektedir.

-----*/

/

*-----

Bir prosesin ilişkin olduğu oturum id'si (session id) getsid POSIX fonksiyonula alınmaktadır:

```
pid_t getsid(pid_t pid);
```

Aşağıdaki programda bir prosesin ve onun alt prosesinin id bilgileri ekrana yazdırılmıştır. Üst ve alt proseslerin aynı session id'ye sahip olduğuna onun da bash'in session id'si olduğuna dikkat ediniz.

-----*/

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

void exit_sys(const char *msg);

int main(void)
{
    pid_t pgid;
    pid_t pid;

    if ((pid = fork()) == -1)
        exit_sys("fork");

    if (pid != 0) { /* parent process */
        printf("Parent process id: %ld\n", (long)getpid());
        printf("Parent process id of the parent: %ld\n", (long)getppid());
        pgid = getpgrp();
        printf("Parent process group id: %ld\n", (long)pgid);
        printf("Parent process session id: %ld\n", (long)getsid(0));

        if (waitpid(pid, NULL, 0) == -1)
            exit_sys("waitpid");
    }
    else { /* child process */
        sleep(1);
    }
}
```

```

    if (setpgid(getpid(), getpid() /* 0 */ ) == -1)
        exit_sys("setpgid");

    printf("Child process id: %ld\n", (long)getpid());
    printf("Parent process id of the child: %ld\n", (long)getppid());
    pgid = getpgrp();
    printf("Child process group id: %ld\n", (long)pgid);
    printf("Child process session id: %ld\n", (long)getsid(0));
}

return 0;
}

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

    exit(EXIT_FAILURE);
}

/
*-----
-----
Yeni bir oturum (session) yaratmak için setsid fonksiyonu
kullanılmaktadır:

pid_t setsid(void);

Fonksiyon şunları yapar:

- Yeni bir oturum oluşturur
- Bu oturum içerisinde yeni bir proses grubu oluşturur.
- Oluşturulan oturumun ve proses grubunun lideri fonksiyonu çağıran
  procestir.

setsid fonksiyonu tipik olarak kabuk programları tarafından işin başında
çağrılmaktadır. Böylece kabuk yeni bir oturumun hem lideri olur
hem de o oturum içerisinde yaratılmış olan bir proses grubunun lideri
olur. O halde bir komut uygulanmamış durumdaki kabuk ortamında bir
oturum
ve bir de proses grubu vardır. Kabuk bu ikisinin de lideri durumundadır.
Sonra kabukta sonu & ile bitmeyen bir komut çalıştırıldığında kabuk
bu komuta ilişkin proses için yeni bir proses grubu yaratır ve bu grubu
oturumun ön plan proses grubu yapar.

setsid fonksiyonunu çağıran proses eğer zaten bir proses grubunun grup
lideri ise fonksiyon başarısız olmaktadır. Örneğin
biz kabuktan çalıştırdığımız bir programda setsid çağrısı yaparsak
başarısız oluruz.

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

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

```

```

#include <unistd.h>

void exit_sys(const char *msg);

int main(void)
{
    if (setsid() == -1)      /* function possibly will fail! */
        exit_sys("setsid");

    return 0;
}

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

    exit(EXIT_FAILURE);
}

/
*-----
-----
    Eğer yeni bir oturum yaratılmak isteniyorsa programın nasıl
    çalıştırılacağı bilinmediğine göre önce fork uygulayıp alt proseste
    setsid uygulamak gerekir.
-----
-----*/

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

void exit_sys(const char *msg);

int main(void)
{
    pid_t pid;

    if ((pid = fork()) == -1)
        exit_sys("fork");

    if (pid != 0)
        exit(EXIT_SUCCESS);

    if (setsid() == -1)      /* function possibly will fail! */
        exit_sys("setsid");

    printf("Ok, i am session leader of the new session!\n");

    return 0;
}

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

```



```

    exit(EXIT_FAILURE);
}

/
*-----
-----
    Oturum lideri open fonksiyonuyla O_NOCTTY bayrağı kullanılmadan bir
    terminal aygıt sürücüsünü açtığında artık o terminal
    oturumun ilişkin olduğu terminal (controlling terminal) durumuna gelir.
    Eğer terminal o anda başka bir oturumun terminaliyse oradan
    kopartılmaktadır.
    open işlemini yapan prosese ise "terminali kontrol eden proses
    (controlling process)" denilmektedir. Normal olarak kabuk programı
    (bash)
    termini kontrol eden proses (controlling process) durumundadır.
    Dosyaların betimleyicileri üst procesten alt prosese aktarıldığına göre
    bu terminal betimleyicisi her proses de gözükecektir. Buna "ilgili
    prosese ilişkin terminal (process controlling terminal)" denilmektedir.
    Anımsanacağı gibi aslında 0 numaralı betimleyici terminal aygıt
    sürücüsünün (controlling terminal) O_RDONLY modunda açılmasıyla,
    1 numaralı betimleyici aynı aygıt sürücünün O_WRONLY moduyla açılmasıyla
    ve stderr de 1 numaralı betimleyinin dup yapılmasıyla
    oluşturulmaktadır.
    Yani aslında 0, 1 ve 2 betimleyiciler aynı dosyaya ilişkindir. Buna
    terminal ya da bu bağlamda "controlling terminal" denilmektedir.
    -----*/

/
*-----
-----
    Oturumdaki ön plan proses grubunun hangisi olduğu tcgetpgrp POSIX
    fonksiyonuyla elde edilebilir. Oturumun ön plan proses grubu da
    tcsetpgrp POSIX fonksiyonula değiştirilebilir.

    pid_t tcgetpgrp(int fildes);
    int tcsetpgrp(int fildes, pid_t pgid_id);
    -----*/

/
*-----
-----
    Oturumun arka plan bir plan prosesi prosesin ilişkin olduğu terminalden
    (controlling terminal) okuma yapmak isterse terminal
    sürücüsü o arka plan prosesin içinde bulunduğu proses grubuna SIGTTIN
    sinyali göndermektedir.
    Bu sinyalin default durumu prosesin durdurulmasıdır. Bu biçimde
    durdurulmuş olan prosesler SIGCONT sinyali ile yeniden çalıştırılmak
    istenebilirler.
    Ancak yeniden okuma yapılırsa yine proses durdurulur. Bu tür prosesler
    kabuk üzerinden fg komutuyla ön plana çekilebilir. Bu durumda kabuk
    önce prosesin proses grubunu ön plan proses

```

grubu yapar sonra da onu SOGCONT sinyali ile uyandırır.

Aşağıdaki programı komut satırında sonuna & gerirerek çalıştırınız.

Program 10 saniye sonra stdin dosyasından okuma yapmaya çalışacak ve bu nedenden dolayı SIGTTIN sinyali gönderilerek durdurulacaktır. Prosesin durdurulmuş olduğunu ps -l komutu ile ya da ps -o stat,cmd komutuyla gözleyiniz

```
-----*/

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

void exit_sys(const char *msg);

int main(void)
{
    sleep(10);
    fgets(s, 100, stdin);
    puts(s);

    return 0;
}

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

    exit(EXIT_FAILURE);
}

/
*-----
Arka plan proses grubundaki proses SIGTTIN sinyalini işleyebilir. Bu
durumda eğer yeniden başlatılabilir olmayan bir sistem fonksiyonu
içerisinde bulunuluyorsa bu sistem fonksiyonu EINTR hata koduyla geri
döner.

Aşağıda programı sonuna & getirerek çalıştırıp log dosyasını
inceleyiniz.
-----*/

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

void exit_sys(const char *msg);
void sigttin_handler(int sno);

FILE *g_f;
```

```

int main(void)
{
    char s[100];
    struct sigaction sa;

    if ((g_f = fopen("log.txt", "w")) == NULL)
        exit_sys("fopen");

    sa.sa_handler = sigttin_handler;
    sigemptyset(&sa.sa_mask);
    sa.sa_flags = 0;

    if (sigaction(SIGTTIN, &sa, NULL) == -1)
        exit_sys("sigaction");

    sleep(10);
    if (fgets(s, 100, stdin) == NULL && errno == EINTR)
        fprintf(g_f, "gets terminated by signal!..\n");
    puts(s);

    return 0;
}

```

```

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

    exit(EXIT_FAILURE);
}

```

```

void sigttin_handler(int sno)
{
    fprintf(g_f, "SIGTTIN occurred!..\n");
}

```

/

*-----

Arka plan proses grubundaki bir prosesin ilişkin terminale (controlling terminal) bir şeyler yazmaya çalışması da uygun değildir. Bu durumda da terminal sürücüsü prosesin ilişkin olduğu arka plan proses grubuna SIGTTOU sinyalini göndermektedir. Bu sinyalin default eylemi yine prosesin durdurulmasıdır. Ancak bu sinyalin aygıt sürücüsü tarafından arka plan proses grubuna gönderilmesi için Linux'ta terminalin TOSTOP modunda olması gerekir. Eğer terminal bu modda değilse proses durdurulmaz ancak yazılanlar da kaybolur.

Aşağıdaki programı sonuna & koyarak çalıştırmayı deneyiniz

-----*/

```

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

```

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

```
int main(void)
{
    sleep(10);

    printf("test\n");

    sleep(10);

    return 0;
}
```

```
/
```

```
*-----
```

Bir prosesin durudulması çizelgeden çıkartılıp bloke edilmesi anlamına gelir. Ancak bunun normal bir blokedden farkı vardır. Bloke olma bir nedene dayalıdır. O neden ortadan kalkınca (ya da sağlanınca) bloke sonlanır. Ancak prosesin durudulması herhangi bir olayı beklemek biçiminde bi bloke değildir. Durdurulmuş prosesler SIGCONT sinyali ile yeniden normal yaşamlarına dönerler. Prosesi durdurmak için ona SIGSTOP sinyaliin gönderilmesi gerekir. SIGSTOP sinyali için sinyal fonksiyonu set edilemez ve bu sinyal ignore da edilemez. Bu anlamda SIGSTOP sinyali SIGKILL sinyaline benzemektedir. SIGSTOP sinyaline benzer diğer bir sinyale de SIGTSTP sinyali denilmektedir. Bu sinyal terminal sürücüsü tarafından klavyeden Ctrl + Z tuşuna basıldığında ön plan porses grubuna gönderilmektedir. Bu sinyalin de -ismi üzer,nde- default eylemi proseslerin durdurulmasıdır. Ancak SIGTSTP sinyali ignore edilebilir ya da bu sinyal için sinyal fonksiyonu set edilebilir. (SIGSTOP sinyaliin ignore edilemediğini ve bunun için bir sinyal fonksiyonu set edilemediğine dikkat ediniz.)

Aşağıdaki programı çalıştırıp Ctrl + Z tuşlarına basmayı deneyiniz.

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

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <unistd.h>
#include <signal.h>
```

```
void exit_sys(const char *msg);
void sigtstop_handler(int sno);
```

```
int main(void)
{
    int i;
    struct sigaction sa;

    sa.sa_handler = sigtstop_handler;
```

```

sigemptyset(&sa.sa_mask);
sa.sa_flags = 0;

if (sigaction(SIGTSTP, &sa, NULL) == -1)
    exit_sys("sigaction");

for (i = 0;; ++i) {
    printf("%d\n", i);
    sleep(1);
}

return 0;
}

```

```

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

    exit(EXIT_FAILURE);
}

```

```

void sigtstop_handler(int sno)
{
    printf("SIGTSTOP occurred!..\n");
}

```

/

*-----

read ve write fonksiyonları POSIX standartlarına göre sistem genelinde atomiktir. Yani örneğin iki proses dosyanın aynı bölgesine write ile yazma yaparsa kesinlikle iç içe geçme olmaz. Ya onun ya da diğerinin yazdıkları gözükür. Benzer biçimde bir proses dosyanın belli bir bölümüne yazma yaparken aynı anda başka bir proses aynı bölgeden okumak yapmak isterse okuyan bozuk bir şeyi okumaz. Ya önceki durumu okur ya da diğerinin tüm yazdıklarını okur.

Ancak VTYS (Veritabanı Yönetim Sistemleri) gibi programlar birden fazla read ya da write ile ilişkili okuma ve yazma yapabilmektedir. Farklı read ve write çağrıları tabii ki atomik değildir. Örneğin bir proses data dosyasına bir yazdıktan sonra ona ilişkin bazı bilgileri indeks dosyasına yazıyor olabilir. Başka bir proses de aynı işlemi yapıyor olabilir. Burada iç içe geçme olabilir. O halde bu gibi ayrık okuma ve yazmaları eğer gerekiyorsa senkronize etmek gerekir. Senkronizasyon için bir mutex kullanmak iyi bir tek nik değildir. Çünkü bütünsel bir kilitlemeye yol açar. Halbuki dosya işlemlerinde yalnızca çakışan offset aralıkları için kilitleme yapılırsa performans artar. İşte bu nedenle zamanla UNIX türevi sistemlere "dosya kilitleme (file locking)" mekanizmaları eklenmiştir.

Dosya kilitleme bütünsel olarak ya da offset temelinde bölgesel olarak (kayıtsal olarak) yapılabilir. Genel olarak bütünsel kilitleme çoğu kez kötü bir tekniktir.

-----*/

/

*-----

Dosyayı bütünsel kilitlemek için flock fonksiyonu kullanılabilir. Bu fonksiyon bir POSIX fonksiyonu değildir. Ancak Linux dahil olmak üzere pek çok UNIX türevi sistemde bulunmaktadır.

```
int flock(int fd, int operation);
```

Fonksiyonun birinci parametresi kilitlenecek dosyaya ilişkin betimleyicidir. İkincisi parametresi kilitleme biçimidir. Bu biçim şunlardan birisi olabilir:

LOCK_SH: Okuma için kilidi alma
LOCK_EX: Yazma için kilidi alma
LOCK_UN: Kilidi bırakma

İstaniirse bunlardan birinin yanı sıra LOCK_NB bayrağı da eklenebilir. Tıpkı okuma yazma kilitlelerinde olduğu gibi birbirleriyle çelişen kilitleme blokeye yol açmaktadır.

Bu biçimdeki kilitler alt prosese de fork işlemi sırasında geçirilmektedir. Eğer kilit hiç açılmazsa kilidin ilişkin olduğu dosya ilişkin son betimleyici kapatıldığında kilit otomatik olarak açılır.

Dosyayı bütünsel kilitlemek yerine proseslerarası çalışan bir senkronizasyon nesnesiyle benzer bir etki yaratılabilir. Ancak toplamda böyle bir senkronizasyon nesnesi oluşturmanın klost maliyeti flock fonksiyonundan çok daha fazla olmaktadır.

Dosya hangi modda açılırsa açılsın flock ile kilitleme yapılabilir.

-----*/

/

*-----

Offset temelinde kilitlemede fcntl fonksiyonu kullanılmaktadır. Anımsanacağı gibi bu fonksiyon açılmış dosyaların birtakım özelliklerini değiştirmek için de kullanılmaktaydı. Fonksiyonun prototipi şöyledir:

```
int fcntl(int fd, int cmd, ... );
```

Offset temelinde kilitleme için kilitlenecek dosyayı temsil eden bir dosya betimleyicisi kullanılır (Birinci parametre). Fonksiyonun parametresine F_SETLK, F_SETLKW ya da F_GETLK değerlerinden biri girilir. Üçüncü parametresine ise her zaman flock isimli bir yapı nesnesinin adresi yerleştirilmelidir. Bu yapı şöyledir:

```
struct flock {
    short l_type;
    short l_whence;
    off_t l_start;
    off_t l_len;
    pid_t l_pid;
};
```

Bu yapı nesnesinin elemanları pid dışında programcı tarafından doldurulur. l_type elemanı kilitlenmenin cinsini belirtir. Bu elemana F_RDLCK, F_WRLCK, F_UNLCK değerlerinden biri girilmelidir. Yapının l_whence elemanı offset için orijini belirtmektedir. Bu elemana da SEEK_SET, SEEK_CUR ya da SEEK_END değerleri girilmelidir. l_start kilitlenecek bölgenin başlangıç offset'ini l_len ise uzunluğunu belirtmektedir. l_pid F_GETLK komutu tarafından doldurulmaktadır.

l_len değeri 0 ise bu l_start değerinden itibaren dosyanın sonuna ve ötesine kadar kitleme anlamına gelmektedir. (Bu durumda örneğin lseek ile dosya göstericisi EOF ötesine konumlandırılrsa bile kilit geçerli olmaktadır.)

F_SETLK blokesiz F_SETLKW blokeli kitleme yapmaktadır. Kilitlenmek istenen alan daha önce kilitlenmiş olan başka alanları kapsıyor ya da kesişiyor olabilir. Bu durumda çelişme tüm kapsanan ve kesişen alanlar dikkate alınarak belirlenmektedir.

F_SETLK ile kitlemek isteyen proses eğer bu alan başka bir proses tarafından çelişki yaratacak biçimde kilitlenmişse fcntl -1 değerine geri döner ve errno EACCESS ya da EAGAIN değeriyle set edilir. F_SETLKW zaten bu durumda bloke olmaktadır. F_GETLK komutu için de programcının flock nesnesini oluşturmuş olması gerekir. Bu durumda fcntl bu alanın isteğe bağlı biçimde kilitlenip kilitlenmeyeceğini bize söyler. Yani bu durumda fcntl kitleme yapmaz ama sanki yapacakmış gibi duruma bakar. Eğer çelişki yoksa fcntl yalnızca yapının type elemanını F_UNLCK haline getirir. Eğer çelişki varsa bu çelişkiye yol açan kilit bilgilerini yapı nesnesinin içerisine doldurur. Fakat o alan birden fazla proses tarafından farklı biçimde kilitlenmişse bu durumda fcntl bu kilitlerin herhangi birinin bilgisini bize verecektir.

fcntl ile offset temelindeki kilitler fork işlemi sırasında alt prosese aktarılmazlar. Yani üst prosesin kitlemiş olduğu alanlar alt proses tarafından da kilitlenmiş gibi olmazlar. exec işlemleri sırasında offset temelindeki kitlemeler varlığını devam ettirmektedir.

Bir proses sonlandığında ya da o i-node elemanına ilişkin tüm betimleyiciler kapatıldığında prosesin ilgili dosyadaki kilitleri serbest bırakılır.

Prosesin dosyaya F_WRLCK koyabilmesi için dosyanın yazma modunda, F_RDLCK koyabilmesi için ise dosyanın okuma modunda açılmış olması gerekir.

Aşağıdaki program offset temelinde kayıt kilitlemeyi betimlemek için yazılmıştır. Program çalıştırıldığında bir komut satırına düşülecek ve komut satırında şu komutlar verilecektir.

```
<fcntl command code> <lock type> <starting offset> <length>
```

Programı kilitlemede kullanılacak dosyanın yol ifadesini komut satırı argümanı biçiminde vererek çalıştırınız. Örneğin:

```
./rlock test.txt
```

Tabii vereceğiniz dosyanın boş olmaması gerekir. Örnek komutlar şöyledir:

Örneğin:

```
CSD (32567): F_SETLK F_WRLCK 0 64
CSD (32767): F_SETLK F_UNLCK 0 64
```

Command kodlar şunlardır: F_SETLK, F_SETLKW, F_GETLK

lock türleri de şunlardır: F_RDLCK, F_WRLCK, F_UNLCK

```
-----*/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>
#include <fcntl.h>
#include <unistd.h>

#define MAX_CMDLINE    4096
#define MAX_ARGS       64

void parse_cmd(void);
int get_cmd(struct flock *fl);
void disp_flock(const struct flock *fl);
void exit_sys(const char *msg);

char g_cmd[MAX_CMDLINE];
int g_count;
char *g_args[MAX_ARGS];

int main(int argc, char *argv[])
{
    int fd;
    pid_t pid;
    char *str;
    struct flock fl;
    int fcntl_cmd;

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



```

}

pid = getpid();

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

for (;;) {
    printf("CSD (%ld)>", (long)pid), fflush(stdout);
    fgets(g_cmd, MAX_CMDLINE, stdin);
    if ((str = strchr(g_cmd, '\n')) != NULL)
        *str = '\0';
    parse_cmd();
    if (g_count == 0)
        continue;
    if (g_count == 1 && !strcmp(g_args[0], "quit"))
        break;
    if (g_count != 4) {
        printf("invalid command!\n");
        continue;
    }

    if ((fcntl_cmd = get_cmd(&f1)) == -1) {
        printf("invalid command!\n");
        continue;
    }

    if (fcntl(fd, fcntl_cmd, &f1) == -1)
        if (errno == EACCES || errno == EAGAIN)
            printf("Locked failed!..\n");
        else
            exit_sys("fcntl");
    if (fcntl_cmd == F_GETLK)
        disp_flock(&f1);
}

close(fd);

return 0;
}

void parse_cmd(void)
{
    char *str;

    g_count = 0;
    for (str = strtok(g_cmd, " \t"); str != NULL; str = strtok(NULL, " \t"))
        g_args[g_count++] = str;
}

int get_cmd(struct flock *f1)
{
    int cmd, type;

    if (!strcmp(g_args[0], "F_SETLK"))

```

```

        cmd = F_SETLK;
    else if (!strcmp(g_args[0], "F_SETLKW"))
        cmd = F_SETLKW;
    else if (!strcmp(g_args[0], "F_GETLK"))
        cmd = F_GETLK;
    else
        return -1;

    if (!strcmp(g_args[1], "F_RDLCK"))
        type = F_RDLCK;
    else if (!strcmp(g_args[1], "F_WRLCK"))
        type = F_WRLCK;
    else if (!strcmp(g_args[1], "F_UNLCK"))
        type = F_UNLCK;
    else
        return -1;

    fl->l_type = type;
    fl->l_whence = SEEK_SET;
    fl->l_start = (off_t)strtol(g_args[2], NULL, 10);
    fl->l_len = (off_t)strtol(g_args[3], NULL, 10);

    return cmd;
}

```

```

void disp_flock(const struct flock *fl)
{
    switch (fl->l_type) {
        case F_RDLCK:
            printf("Read Lock\n");
            break;
        case F_WRLCK:
            printf("Write Lock\n");
            break;
        case F_UNLCK:
            printf("Unlocked (can be locked)\n");
    }

    printf("Whence: %d\n", fl->l_whence);
    printf("Start: %ld\n", (long)fl->l_start);
    printf("Length: %ld\n", (long)fl->l_len);
    if (fl->l_type != F_UNLCK)
        printf("Process Id: %ld\n", (long)fl->l_pid);
}

```

```

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

    exit(EXIT_FAILURE);
}

```

```

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

```

Eğer çok sayıda lock işlemi yapılacaksa fcntl çağrısını yapan bir satma fonksiyon kullanılabilir.

```
-----*/
int setlock_wrapper(int fd, int type, int whence, off_t start, off_t len)
{
    struct flock fl;

    fl.l_type = type;
    fl.l_whence = whence;
    fl.l_start = start;
    fl.l_len = len;

    return fcntl(fd, F_SETLK, &fl);
}

int getlock_wrapper(int fd, int type, int whence, off_t start, off_t len)
{
    if (fcntl(fd, F_GETLK, &fl) == -1)
        return -1;

    return fl.l_type;
}

/*-----
```

Yukarıdaki sarma fonksiyonların bir benzeri POSIX standartlarında lockf (flock ile karıştırmayınız) ismiyle bulunmaktadır.

```
int lockf(int fildes, int function, off_t size);
```

Bu fonksiyon her zaman dosya göstericisinin gösterdiği yerden itibaren size kadar byte'ı kilitlet. size değeri negatif olabilir. İkinci parametre olan function şunlardan biri olabilir:

```
F_ULOCK
F_LOCK
F_TLOCK
F_TEST
```

F_ULOCK "unlock" anlamındadır. F_LOCK kilitleme yapmaya çalışır. Eğer çelişkili bir durum varsa blokeye yol açar. F_TLOCK ise çelişki durumunda bloke oluşturmayıp -1 ile geri döner. errno değeri de yine EACCES ya da EAGAIN değeri ile set edilir. F_TEST kilitleme yapmaz ancak durumu test eder. Eğer test çelişkili ise fonksiyon -1'e geri döüp errno değerini EACCES ya da EAGAIN değerine set etmektedir.

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

/

*-----

Biz yukarıda "tavsiye niteliğinde (advisory)" kilitleme yaptık. Burada "tavsiye niteliğinde (advisory)" demekle şu anlatılmak istenmektedir: Bir proses fcntl (ya da lockf ya da flock ile) kilitleme yaptığı anda aslında başka bir proses isterse read ve write fonksiyonlarıyla istediği zaman okuma yazma yapabilmektedir. Yani read ve write fonksiyonları ilgili alanın kilitli olup olmadığına bakmamaktadır. Tabii programcı tüm proseslerin kodlarını kendisi yazdığı için (ya da bir protokol eşliğinde başkaları da yazmış olabilir) her zaman okuma yazma işlemlerinde öncelikle kilide bakçak biçimde bir strateji izler. İşte kilitlemenin diğer bir türüne de "zorlamalı kilitleme (mandatory locking)" denilmektedir. Artık bu kilitleme biçiminde başka bir proses kilitlenmiş alanı çelişki durumunda istede de read ve write ile okuyup yazamamaktadır.

Zorlamalı kilitleme POSIX standartların bulunmamaktadır. POSIX standartları fcntl fonksiyonun açıklamasında bunun gerekçelerini belirtmiştir. Bazı işletim sistemleri bunu hiç desteklememektedir. Linux 2.4 çekirdeği ile birlikte zorlamalı dosya kilitlemesini destekler hale gelmiştir. Zorlamalı kilitleme için aslında fcntl fonksiyonunda ek bir şey yapılmaz. Linux'ta zorlamalı kilitleme için şu koşulların sağlanmış olması gerekmektedir:

- 1) Dosyanın içinde bulunduğu dosya sisteminin -o mand ile mount edilmiş olması gerekir. (remount -o mand, remount <device> <mount point>)
- 2) İlgili dosyanın setg-group id bayrağı set edilip x hakkı varsa reset edilmelidir. (chmod <dosya> g+s g-x)

Zorlamalı kilitlemeye çekirdek her read/write işleminde kilide denk gelinmiş mi diye kontroller yapmaktadır. Bu da çalışmayı yavaşlatmaktadır. Ayrıca zorlamalı kilitleme kötüye kullanıma da açıktır.

Aşağıda zorlamalı kilitlemeyi test etmek için iki program verilmiştir. Programlardan ilki belli bir dosyanın belli bir yerinden n byte okuma yazma yapar. Örneğin:

```
./mlocktest test.txt r 0 64
./mlocktest test.txt w 30 50
```

Yazma sırasında hep 0'lar yazılır.

Diğer program yukarıdaki rlock.c programıdır.

Bu denemeyi yapmadan önce şu işlemleri uygulayınız:

```
sudo mount -o remount,mand /dev/sda2 /
chmod g+s-x test.txt
```

Sonra programları farklı terminallerde dosya ismi vererek (burada test.txt) çalıştırınız.

-----*/

```

/* mlocktest.c */

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

void exit_sys(const char *msg);

/* ./mlocktest <dosya ismi> <r/w> <offset> <uzunluk> */

int main(int argc, char *argv[])
{
    int fd;
    int operation;
    off_t offset;
    off_t len;
    char *buf;
    ssize_t result;

    if (argc != 5) {
        fprintf(stderr, "wrong number of arguments!..\n");
        exit(EXIT_FAILURE);
    }

    if (strcmp(argv[2], "r") && strcmp(argv[2], "w")) {
        fprintf(stderr, "invalid operation!\n");
        exit(EXIT_FAILURE);
    }

    offset = (off_t)strtol(argv[3], NULL, 10);
    len = (off_t)strtol(argv[4], NULL, 10);

    if ((buf = (char *)calloc(len, 1)) == NULL) {
        fprintf(stderr, "cannot allocate memory!..\n");
        exit(EXIT_FAILURE);
    }

    if ((fd = open(argv[1], argv[2][0] == 'r' ? O_RDONLY : O_WRONLY)) == -1)
        exit_sys("open");

    lseek(fd, 0, offset);
    if (argv[2][0] == 'r') {
        if ((result = read(fd, buf, len)) == -1)
            exit_sys("read");
        printf("%ld bytes read\n", (long)result);
    }
    else {
        if ((result = write(fd, buf, len)) == -1)
            exit_sys("write");
        printf("%ld bytes written\n", (long)result);
    }
}

```

```

    free(buf);
    close(fd);

    return 0;
}

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

    exit(EXIT_FAILURE);
}

/* rlock.c */

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

#define MAX_CMDLINE    4096
#define MAX_ARGS       64

void parse_cmd(void);
int get_cmd(struct flock *fl);
void disp_flock(const struct flock *fl);
void exit_sys(const char *msg);

char g_cmd[MAX_CMDLINE];
int g_count;
char *g_args[MAX_ARGS];

int main(int argc, char *argv[])
{
    int fd;
    pid_t pid;
    char *str;
    struct flock fl;
    int fcntl_cmd;

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

    pid = getpid();

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

    for (;;) {
        printf("CSD (%ld)>", (long)pid), fflush(stdout);
        fgets(g_cmd, MAX_CMDLINE, stdin);
    }
}

```

```

    if ((str = strchr(g_cmd, '\n')) != NULL)
        *str = '\0';
    parse_cmd();
    if (g_count == 0)
        continue;
    if (g_count == 1 && !strcmp(g_args[0], "quit"))
        break;
    if (g_count != 4) {
        printf("invalid command!\n");
        continue;
    }

    if ((fcntl_cmd = get_cmd(&f1)) == -1) {
        printf("invalid command!\n");
        continue;
    }

    if (fcntl(fd, fcntl_cmd, &f1) == -1)
        if (errno == EACCES || errno == EAGAIN)
            printf("Locked failed!..\n");
        else
            exit_sys("fcntl");
    if (fcntl_cmd == F_GETLK)
        disp_flock(&f1);
}

close(fd);

return 0;
}

void parse_cmd(void)
{
    char *str;

    g_count = 0;
    for (str = strtok(g_cmd, " \t"); str != NULL; str = strtok(NULL, " \t"))
        g_args[g_count++] = str;
}

int get_cmd(struct flock *f1)
{
    int cmd, type;

    if (!strcmp(g_args[0], "F_SETLK"))
        cmd = F_SETLK;
    else if (!strcmp(g_args[0], "F_SETLKW"))
        cmd = F_SETLKW;
    else if (!strcmp(g_args[0], "F_GETLK"))
        cmd = F_GETLK;
    else
        return -1;

    if (!strcmp(g_args[1], "F_RDLCK"))
        type = F_RDLCK;

```

```

    else if (!strcmp(g_args[1], "F_WRLCK"))
        type = F_WRLCK;
    else if (!strcmp(g_args[1], "F_UNLCK"))
        type = F_UNLCK;
    else
        return -1;

    fl->l_type = type;
    fl->l_whence = SEEK_SET;
    fl->l_start = (off_t)strtol(g_args[2], NULL, 10);
    fl->l_len = (off_t)strtol(g_args[3], NULL, 10);

    return cmd;
}

void disp_flock(const struct flock *fl)
{
    switch (fl->l_type) {
        case F_RDLCK:
            printf("Read Lock\n");
            break;
        case F_WRLCK:
            printf("Write Lock\n");
            break;
        case F_UNLCK:
            printf("Unlocked (can be locked)\n");
    }

    printf("Whence: %d\n", fl->l_whence);
    printf("Start: %ld\n", (long)fl->l_start);
    printf("Length: %ld\n", (long)fl->l_len);
    if (fl->l_type != F_UNLCK)
        printf("Process Id: %ld\n", (long)fl->l_pid);
}

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

    exit(EXIT_FAILURE);
}

int setlock_wrapper(int fd, int type, int whence, off_t start, off_t len)
{
    struct flock fl;

    fl.l_type = type;
    fl.l_whence = whence;
    fl.l_start = start;
    fl.l_len = len;

    return fcntl(fd, F_SETLK, &fl);
}

int getlock_wrapper(int fd, int type, int whence, off_t start, off_t len)

```



```

{
    if (fcntl(fd, F_GETLK, &fl) == -1)
        return -1;

    return fl.l_type;
}

/
*-----
-----
Linux sistemlerinde sistemdeki tüm kilitler /proc/locks isimli dosyay
çekirdek tarafından yazılmaktadır. Örneğin:

1: POSIX  ADVISORY  WRITE 3515 08:02:393260 0 63
2: POSIX  ADVISORY  READ  2339 08:02:786493 128 128
3: POSIX  ADVISORY  READ  2339 08:02:786491 1073741826 1073742335
4: FLOCK  ADVISORY  WRITE 770 00:17:614 0 EOF

Burada birinci sütun kilidin flock fonksiyonuyla mı yoksa fcntl
fonksiyonuyla mı oluşturulduğunu belirtir. İkinci sütun
kilidin isteğe bağlı mı yoksa zorlamalı mı olduğunu belirtmektedir. Üçüncü
sütun kilidin yazmayı mı okumayı mı kilitlediğini
belirtmektedir. Sonraki sütun kilitleyen prosesin id'sini sonraki sütun ise
kilitlenmiş dosyanın dosya sistemindek, aygıt
numarasını ve i-node numarasını belirtmektedir. Nihayet son sütun
kilitleme offset aralığını belirtmektedir.
-----
-----*/
/
*-----
-----

```

POSIX sistemlerinde ileri IO işlemleri 4 bölüme ayrılarak incelenebilir:

- 1) Multiplexed IO: Bir grup betimleyici izlemeye alınır. Bu betimleyicilerde ilgilenilen olay (read/write/error) yoksa bloke beklenir. Ta ki bu betimleyicilerden en az birinde ilgilenilen olay gerçekleşene kadar. Multiplexed IO için select ve poll POSIX fonksiyonları kullanılmaktadır. Ancak Linux epoll isimli daha yetenekli bir fonksiyona da sahiptir.
- 2) Signal Driven IO: Burada belli betimleyiciler izlemeye alınır. Ancak bloke beklenmez. Bu betimleyicilerde olay gerçekleştiğinde SIGIO isimli sinyal oluşur. Programcı da bu sinyal oluşturulduğunda blokeye maruz kalmadan read/write yapılabilir.
- 3) Asenkron IO: Burada read/write işlemleri başlatılır. Ancak bir bloke oluşmaz. Arka planda çekirdek tarafından okuma ve yazma bir yandan yapılır. Ancak aktarım bittiğinde programcı haberdar edilir. Bunun signal driven IO'dan farkı şudur: Signal driven IO'da aktarım yapılmamaktadır. Yalnızca okuma yazma yallırsa bloke olunmayacağı prosese söylenmektedir. Halbuki asenkron IO'da okuma ve yazma işlemi bloke çözüldüğünde arka planda gerçekleştirilmekte ve yalnızca işlemin bittiği haber verilmektedir.

4) Scatter-Gather IO: Burada okuma birden fazla adrese yazma ise birden fazla adresten kaynağa yapılmaktadır.

-----*/

/

*-----

Multiplexed IO blokeye yol açabilecek aynı anda birden fazla dosya betimleyici ile işlem yapmayı anlatan bir IO terimidir. Multiplexed IO için select, pselect, poll ve epoll fonksiyonları kullanılmaktadır.

select fonksiyonu ilk kez BSD sistemlerinde denenmiştir. Sonra POSIX standartlarına sokulmuştur. Fonksiyonun prototipi şöyledir:

```
int select(int nfd, fd_set *readfds, fd_set *writefds, fd_set *errorfds, struct timeval *timeout);
```

Fonksiyonun birinci parametresi ilgilenilecek betimleyicilerin en büyük değerini almaktadır. En yüksek betimleyici numarası FD_SETSIZE ile belirtilmiştir. Bu parametre doğrudan FD_SETSIZE (1024) olarak girilebilir. Ancak bu parametrenin uygun girilmesi bir hız kazancı sağlayabilmektedir. Fonksiyon parametrelerindeki fd_set türü bir bit dizisi belirtmektedir. Bu bit dizisinin belli bitini 1 yapmak için FD_SET, belli bitini 0 yapmak için FD_CLR, tüm bitlerini sıfır yapmak için FD_ZERO, belli bitini test etmek için ise FD_ISSET makroları kullanılmaktadır. Biz fonksiyona üç tane fd_set nesnesi veririz.

Bunlardan ilki "okuma", ikinci "yazma" üçüncüsü "error" kontrolü yaptırmak içindir. Fonksiyon son parametresi "zaman aşımı" belirtmektedir. Bu parametreler için NULL adres girilebilir. Zaman aşımı için NULL adres girilirse zaman aşımının kullanılmayacağı anlaşılır. Eğer zaman aşımı için yapının her iki elemanı 0'da girilebilir. Budurumda select fonksiyonu hemen testini yapar ve geri döner.

select fonksiyonu okuma takibi için tipik olarak şöyle kullanılır: Programcı okuma izlemesi yapılacak betimleyicilerin numaralarına ilişkin bitleri

bir fd_set içerisinde set eder. Bunu fonksiyonun ikinci parametresine verir. Fonksiyon da yalnızca 1 olan bitlere ilişkin betimleyicileri izleyecektir. Programcı bu betimleyicilerden okuma yapılamayacak durumda ise (yani okuma girişiminde bloke oluşacak durumda ise) select bloke akışı bekletir. En az bir betimleyicide okuma eylemi yapılabilecek durumdaysa bloke çözülür. Fonksiyon hangi betimleyicilerde okuma işleminin yapılabileceğini

yine bizim ona verdiğimiz fd_set nesnesinin ilgili 1 yaparak bize iletmektedir. Yani bizim fonksiyona verdiğimiz fd_set nesnesi fonksiyon geri döndüğünde

artık bozmuş durumdadır yani hangi betimleyicilerin okumaya elvirişli olduğunu gösterecek biçimde değiştirilmiş durumdadır. Buradaki aynı çalışma

biçimi "yazma" ve "error" işlemi için de aynı biçimde söz konusudur. Programcı aynı zamanda isterse üç ayrı fd_set nesnesini de fonksiyona verebilir.

select fonksiyonu başarısızlık durumunda -1 değerine geri döner. Eğer hiçbir betimleyicide olay gerçekleşmemiş ancak zaman aşımı dolmuşsa 0 değerine geri döner.

Eğer en az bir betimleyicide ilgili olay gerçekleşmişse toplam olay gerçekleşen betimleyici sayısına geri döner. (Aynı betimleyici örneğin hem okuma hem de yazma için izleniyorsa ve bu betimleyicide hem okuma hem de yazma olayı gerçekleşmişse bu değer 2 artırılmaktadır.)

select fonksiyonunun normal disk dosyaları için kullanılması anlamsızdır. Uzun süre beklemeye yol açabilecek terminal gibi, boru gibi, soket gibi aygıtlar için kullanılmalıdır. Normal olarak select ile bekeleneyecek betimleyicilere ilişkin kaynaklar "bloke"li modda açılmalıdır.

Aşağıdaki örnekte 0 numaralı betimleyici select ile izlenmiştir. Bu betimleyici terminal sürücüsüne ilişkindir. Kullanıcı bir yazı girip ENTER tuşuna bastığında terminal sürücüsü select fonksiyonunun blokesini çözecektir.

-----*/

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/select.h>

void exit_sys(const char *msg);

int main(int argc, char *argv[])
{
    fd_set rset;
    int result;
    char buf[1024 + 1];
    ssize_t n;

    FD_ZERO(&rset);
    FD_SET(0, &rset);

    if ((result = select(1, &rset, NULL, NULL, NULL)) == -1)
        exit_sys("select");

    printf("result = %d\n", result);
    if ((n = read(0, buf, 1024)) == -1)
        exit_sys("read");
    buf[n] = '\0';
    printf("%s", buf);

    return 0;
}
```

```

}

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

    exit(EXIT_FAILURE);
}

/
*-----
-----
    Yukarıdaki programa 5 saniyelik bir zaman aşımı ekleyelim.
-----
-----*/

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

void exit_sys(const char *msg);

int main(int argc, char *argv[])
{
    fd_set rset;
    int result;
    char buf[1024 + 1];
    ssize_t n;
    struct timeval tv;

    FD_ZERO(&rset);
    FD_SET(0, &rset);

    tv.tv_sec = 5;
    tv.tv_usec = 0;

    if ((result = select(1, &rset, NULL, NULL, &tv)) == -1)
        exit_sys("select");

    if (result == 0)
        printf("Timeout!\n");
    else {
        printf("result = %d\n", result);
        if ((n = read(0, buf, 1024)) == -1)
            exit_sys("read");
        buf[n] = '\0';
        printf("%s", buf);
    }

    return 0;
}

void exit_sys(const char *msg)

```

```

{
    perror(msg);

    exit(EXIT_FAILURE);
}

/
*-----
-----
    POSIX standartları zaman aşımı için verdiğimiz timeval yapı nesnesinin
    fonksiyon tarafından güncellenip güncellenmeyeceğini
    isteğe bağlı olarak sisteme bırakmıştır. Linux'ta select sonlanmadan
    önce bu yapıya kalan zaman miktarı yerleştirilir.
-----
-----*/

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

void exit_sys(const char *msg);

int main(int argc, char *argv[])
{
    fd_set rset;
    int result;
    char buf[1024 + 1];
    ssize_t n;
    struct timeval tv;

    FD_ZERO(&rset);
    FD_SET(0, &rset);

    tv.tv_sec = 10;
    tv.tv_usec = 0;

    if ((result = select(1, &rset, NULL, NULL, &tv)) == -1)
        exit_sys("select");

    if (result == 0)
        printf("Timeout!\n");
    else {
        printf("result = %d\n", result);
        if ((n = read(0, buf, 1024)) == -1)
            exit_sys("read");
        buf[n] = '\0';
        printf("%s", buf);
        printf("Remaining time: %ld.%03ld\n", (long)tv.tv_sec,
            (long)tv.tv_usec / 1000);
    }

    return 0;
}

```

```

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

    exit(EXIT_FAILURE);
}

```

```

/
*-----
-----
select ile birden fazla betimleyiciyi izlerken bloke çözüldüğünde
programcının hangi betimleyiciler dolayısıyla blokenin
çözüldüğünü belirlemesi gerekir. Bunun için FD_ISSET makrosu ile izlenen
tüm betimleyiciler kontrol edilmelidir. Aslında
bu kontrol yapılırken 0'dan FD_SETSIZE değerine kadar bile bir döngü
kullanılabilir. Naisl olsa izlenmeyen tüm betimleyicilerin
bitleri 0 olacaktır. Tabii programcı döngüyü kısaltmak için eğer
betimleyicilerin numaralarını bir yerde saklamışsa yalnızca onu da
sorgulayabilir.

```

Aşağıdaki birden fazla isimli borundan okuma yapmaya çalışan bir örnek bulunmaktadır. Boruların isimleri komut satırı argümanlarıyla verilmektedir. Bu isimli borular açılıp bunların betimleyicileri bir dizide saklanmıştır. select'in blokesi çözüldüğünde hangi borularda okuma eyleminin yapılabileceği FD_ISSET le kontrol edilmiştir. Yazan taraf boruyu kapattığında bu da bir okuma eylemi gibi select'in blokesini çözer. Ancak bu durumda borudan 0 byte okunacaktır. Aşağıdaki uygulama için önce isimli borular yaratıp farklı terminallerden bu boruları cat ile aşağıdaki gibi açınız.

```

cat > boru ismi

```

```

-----*/

```

```

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

```

```

#define MAX_ARGS      32
#define BUFFER_SIZE   1024

```

```

void exit_sys(const char *msg);

```

```

int main(int argc, char *argv[])
{
    int fds[MAX_ARGS];
    int i;
    fd_set rset, rset_o;
    char buf[BUFFER_SIZE + 1];
    ssize_t result;

```

```

int maxfds, nfd;

if (argc == 1) {
    fprintf(stderr, "too few arguments!..\n");
    exit(EXIT_FAILURE);
}

printf("waiting at open...\n");

maxfds = 0;
nfd = 0;
FD_ZERO(&rset_o);
for (i = 0; i < argc - 1; ++i) {
    if ((fds[i] = open(argv[i + 1], O_RDONLY)) == -1)
        exit_sys("open");
    FD_SET(fds[i], &rset_o);
    if (fds[i] > maxfds)
        maxfds = fds[i];
    ++nfd;
}
printf("waiting at select...\n");
for (;;) {
    rset = rset_o;
    if (select(maxfds + 1, &rset, NULL, NULL, NULL) == -1)
        exit_sys("select");
    for (i = 0; i < argc - 1; ++i)
        if (FD_ISSET(fds[i], &rset)) {
            if ((result = read(fds[i], buf, BUFFER_SIZE)) == -1)
                exit_sys("read");
            if (!result) {
                printf("pipe closing...\n");
                FD_CLR(fds[i], &rset_o);
                close(fds[i]);
                --nfd;
                if (!nfd)
                    goto EXIT;
            }

            buf[result] = '\0';
            printf("%s", buf);
        }
}

EXIT:
return 0;
}

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

    exit(EXIT_FAILURE);
}

```

```
/
*-----
-----
Aşağıdaki örnekte ise boruların betimleyicileri bir dizide
toplanmıştır.
-----
-----*/
```

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

#define MAX_ARGS      32
#define BUFFER_SIZE   1024

void exit_sys(const char *msg);

int main(int argc, char *argv[])
{
    int fd;
    fd_set rset, rset_o;
    char buf[BUFFER_SIZE + 1];
    ssize_t result;
    int maxfds, nfds;
    int i;

    if (argc == 1) {
        fprintf(stderr, "too few arguments!..\n");
        exit(EXIT_FAILURE);
    }

    printf("waiting at open...\n");

    maxfds = 0;
    nfds = 0;
    FD_ZERO(&rset_o);
    for (i = 0; i < argc - 1; ++i) {
        if ((fd = open(argv[i + 1], O_RDONLY)) == -1)
            exit_sys("open");
        FD_SET(fd, &rset_o);
        if (fd > maxfds)
            maxfds = fd;
        ++nfds;
    }
    printf("waiting at select...\n");
    for (;;) {
        rset = rset_o;
        if (select(maxfds + 1, &rset, NULL, NULL, NULL) == -1)
            exit_sys("select");
        for (i = 0; i <= maxfds; ++i)
            if (FD_ISSET(i, &rset)) {
                if ((result = read(i, buf, BUFFER_SIZE)) == -1)
```



```

        exit_sys("read");
    if (!result) {
        printf("pipe closing...\n");
        FD_CLR(i, &rset_o);
        close(i);
        --nfds;
        if (!nfds)
            goto EXIT;
    }

    buf[result] = '\0';
    printf("%s", buf);
}

EXIT:
    return 0;
}

```

```

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

    exit(EXIT_FAILURE);
}

```

/

pselect fonksiyonu select fonksiyonun biraz daha gelişmiş bir versiyonudur. Prototipi şöyledir:

```

int pselect(int nfds, fd_set *readfds, fd_set *writefds, fd_set *errorfds,
    const struct timespec *timeout, const sigset_t *restrict sigmask);

```

Fonksiyonun select fonksiyonundan yalnızca üç farkı vardır. Diğer bütün davranışı aynıdır.

- 1) Zaman aşımı için timeval yapısı değil timespec yapısı kullanılmıştır. Bu da nanosaniye çözünürlük anlamına gelmektedir.
- 2) Fonksiyon bir "signal set" parametresine sahiptir. Biz istersek fonksiyon çalışana kadar belli sinyallerin bloke edilmesini sağlayabiliriz. Tabii bu parametreyi NULL da geçebiliriz. Fonksiyon sonlandığında otomatik olarak bu sinyaller prosesin sinyal mask kümesinden çıkarılmaktadır.
- 3) pselect fonksiyonunun zaman aşımı parametresi const biçimdedir. Yani fonksiyon tarafından güncellenmemektedir.

select ve pselect fonksiyonları eğer bloke edilmemişse ilgili sinyal oluştuğunda -1 ile geri döner ve errno EINTR ile set edilir. Bu fonksiyonlar "restartable" yapılamazlar.

-----*/

/

*-----

poll fonksiyonu amaç bakımından select fonksiyonuna çok benzemektedir. select ve poll aynı işi yapan farklı arayüzler biçiminde düşünülebilir. Eskiden select BSD sistemlerinde, poll ise AT&T UNIX sistemlerinde kullanılıyordu. Tabii uzun süredir bu iki fonksiyon da POSIX standartlarında bulunmaktadır.

```
int poll(struct pollfd fds[], nfds_t nfds, int timeout);
```

poll fonksiyonu ilgilenilen betimleyicileri ve olayları tek tek bir yapı dizisi biçiminde bizden ister. Yani biz bir yapı dizisi oluşturup onun içini doldururuz. Sonra bu yapı dizisinin adresini poll fonksiyonuna veririz. poll fonksiyonu da select fonksiyonunda olduğu gibi bu olayları bloke izler. Bu betimleyicilerden herhangi birinde bir olay olduğunda blokeyi çözer. Programcı da girdiği diziyi kontrol ederek hangi olayların olduğunu anlayıp uygun işlemleri yapar. Fonksiyonun timeout parametresi -1 girilirse zaman aşımı ortadan kaldırılmaktadır. Bu parametre 0 girilirse fonksiyon betimleyicilerin durumlarına hemen bakıp geri döner. Sıfır dışı değer milisaniye cinsinden zaman aşımı belirtmektedir. poll fonksiyonu başarısızlık durumunda -1 değerine, zaman aşımından dolayı sonlanmalarda 0 değerine ve normal sonlanmalarda ise olay gerçekleşen betimleyici sayısına geri dönmektedir.

poll ile izlenecek en önemli iki olay (events) POLLIN ve POLLOUT olaylarıdır. poll fonksiyonunun bize verdiği olaylar (revents) ise tipik olarak şunlardan oluşmaktadır: POLLIN, POLLOUT, POLLERR, POLLHUP. POLLHUP boru, soket gibi aygıtlar kapatıldığında oluşur. POLLERR ise borularda okuyan taraf botuyu kapattığında soketlerde ise soket kapatıldığında oluşmaktadır.

-----*/

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

void exit_sys(const char *msg);

int main(int argc, char *argv[])
{
    struct pollfd pfd[1];
    int result;
    char buf[1024 + 1];
    ssize_t n;

    pfd[0].fd = 0;
    pfd[0].events = POLLIN;
```

```

    if ((result = poll(pfds, 1, -1)) == -1)
        exit_sys("poll");

    printf("%d event(s) occurred\n", result);

    if (pfds[0].revents & POLLIN) {
        if ((n = read(0, buf, 1024)) == -1)
            exit_sys("read");
        buf[n] = '\0';
        printf("%s", buf);
    }

    return 0;
}

```

```

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

    exit(EXIT_FAILURE);
}

```

```

/
*-----
-----
Aşağıdaki programda birden fazla isimli boru ile poll fonksiyonu
kullanılarak multiplexed IO işlemi yapılmıştır.
Programı çalıştırırken komut satırı argümanı olarak boru isimlerini
giriniz. Girdiğiniz boruları başka terminallerden
cat > boru ismi komutu ile açınız.
-----
-----*/

```

```

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

#define MAX_ARGS      32
#define BUFFER_SIZE   1024

void exit_sys(const char *msg);

int main(int argc, char *argv[])
{
    int i;
    int fd;
    char buf[BUFFER_SIZE + 1];
    int result;
    ssize_t n;
    int nfds;
    struct pollfd pfds[MAX_ARGS];

    if (argc == 1) {

```

```

        fprintf(stderr, "too few arguments!..\n");
        exit(EXIT_FAILURE);
    }

    printf("waiting at open...\n");

    nfds = 0;
    for (i = 0; i < argc - 1; ++i) {
        if ((fd = open(argv[i + 1], O_RDONLY)) == -1)
            exit_sys("open");
        pfds[i].fd = fd;
        pfds[i].events = POLLIN;
        ++nfds;
    }

    printf("waiting at poll...\n");
    for (;;) {
        if ((result = poll(pfds, nfds, -1)) == -1)
            exit_sys("poll");
        for (i = 0; i < nfds; ++i) {
            if (pfds[i].revents & POLLIN) {
                if ((n = read(pfds[i].fd, buf, BUFFER_SIZE)) == -1)
                    exit_sys("read");
                buf[n] = '\0';
                printf("%s", buf);
            }
            else if (pfds[i].revents & POLLHUP) {
                close(pfds[i].fd);
                --nfds;
                if (!nfds)
                    goto EXIT;
            }
        }
    }
EXIT:
    return 0;
}

```

```

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

    exit(EXIT_FAILURE);
}

```

/

*-----

select fonksiyonunun sinyal blokesi yapan pselect biçiminde bir versiyonu vardı. İşte poll fonksiyonun da Linux sistemlerinde sinyal blokesi yapan ppoll isimli bir versiyonu vardır. Ancak pselect POSIX standartlarında bulunduğu halde ppoll bulunmamaktadır. ppoll Linux'a özgüdür:

```
int ppoll(struct pollfd *fds, nfds_t nfds, struct timespec *timeout_ts,
const sigset_t *sigmask);
```

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

select ve poll fonksiyonları bazı sistemlerde yüksek performansla çalışabilmektedir. Ancak Linux'ta bu fonksiyonların izlediği betimleyici sayısı arttıkça fonksiyonlar önemli bir yavaşlama içerisine girmektedir. Yani maalesef Linux sistemlerinde select ve poll fonksiyonları iyi biçimde ölçeklendirilmemiştir. İşte Linux'ta daha iyi ölçeklendirilmiş epoll isimli bir sistem fonksiyonu bulundurulmuştur. Yüksek performans isteyen server programlar Linux'ta epoll sistemini tercih etmektedir. Tabii epoll POSIX standartlarında mevcut değildir. Yalnızca Linux sistemlerinde bulunmaktadır. epoll sistemi şöyle kullanılmaktadır:

1) Programcı önce epoll_create isimli fonksiyonla bir betimleyici elde eder. Bu etimleyicinin IO olaylarının izleneceği betimleyici ile bir ilgisi yoktur. Bu betimleyici diğer fonksiyonlara bir handle gibi geçirilmektedir:

```
int epoll_create(int size);
```

Fonksiyonun parametresi kaç betimleyicinin izlenileceğine yönelik bir ip ucu değeri alır. Programcı burada verdiği değerden daha fazla betimleyiciyi izleyebilir. Daha sonra bu parametre tasarımcıları rahatsız etmiş ve epoll_create1 isimli fonksiyonla kaldırılmıştır.

```
int epoll_create1(int flags);
```

Buradaki flags şimdilik yalnızca FD_CLOEXEC değerini ya da 0 değerini alabilmektedir. Fonksiyonların geri dönüş değeri başarı durumunda handle görevind eolan bir betimleyicidir.

2) Artık programcı izleyeceği betimleyicileri epoll sistemine epoll_ctl fonksiyonuyla ekler. Örneğin programcı 5 boru betimleyicisini izleyecekse bu 5 betimleyici için de ayrı ayrı epoll_ctl çağrısı yapmalıdır.

```
int epoll_ctl(int epfd, int op, int fd, struct epoll_event *event);
```

Fonksiyonun birinci parametresi epoll_create ya da epoll_create1 fonksiyonundan elde edilen handle değeridir. İkinci parametre EPOLL_CTL_ADD, EPOLL_CTL_MOD, EPOLL_CTL_DEL değerlerinden birini alır. EPOLL_CTL_ADD ekleme için, EPOLL_CTL_DEL çıkarma için ve EPOLL_CTL_MOD mevcut eklenmiş betimleyicide izleme değişikliği yapmak için kullanılmaktadır. Üçüncü parametre izlenecek betimleyiciyi belirtir. Son parametre izlenecek olayı belirtmektedir. sturct epoll_event yapısı şöyle bildirilmiştir:

```
struct epoll_event {
    uint32_t    events;
```

```
    epoll_data_t data;
};
```

Yapının events elemanı tıpkı poll fonksiyonunda olduğu gibi EPOLLIN, EPOLLOUT değerlerini almaktadır. Geri döndürülen olay da yine EPOLLIN, EPOLLOUT, EPOLLERR ve EPOLLHUP gibi olaylardır. Yapının data elemanı aslında çekirdek tarafından saklanıp epoll_wait fonksiyonu yoluyla bize geri verilmektedir. Bu eleman bir birlik biçiminde bildirilmiştir:

```
typedef union epoll_data {
    void        *ptr;
    int         fd;
    uint32_t    u32;
    uint64_t    u64;
} epoll_data_t;
```

select ve poll fonksiyonları "düzey tetiklemeli (level triggered)" çalışmaktadır. epoll fonksiyonu da default durumda düzey tetiklemeli çalışır. Ancak events parametresine bit or işlemi ile EPOLLET eklenirse o betimleyici için "kenar tetiklemeli (edge triggered)" mod kullanılır. Düzey tetiklemeli mod demek (select, poll daki durum ve epoll'daki default durum) bir okuma ya da yazma olayı açılıp bloke çözüldüğünde programcı eğer okuma ya da yazma yapmayıp yeniden bu fonksiyonları çağırırsa bekleme yapılmayacak demektir. Yani örneğin biz select ya poll ile stdin dosyasını izliyorsak ve klavyeden bir giriş yapıldıysa bu fonksiyonlar blokeyi çözer. Fakat biz read ile okuma yapmazsak ve yeniden select ve poll fonksiyonlarını çağırırsak artık bloke oluşmaz. Halbuki kenar tetiklemeli modda biz okuma yapmasak bile yeni okuma eylemi oluşana kadar yine blokede kalırız.

3) Asıl bekleme ve izleme işlemi epoll_wait fonksiyonu tarafından yapılmaktadır. Bu fonksiyon poll select ve poll fonksiyonu gibi bloke oluşturur ve arka planda betimleyicileri izler.

```
int epoll_wait(int epfd, struct epoll_event *events, int maxevents, int
    timeout);
```

Fonksiyonun birinci parametresi epoll_create ya da epoll_create1 fonksiyonundan elde edilmiş olan handle değeridir. İkinci parametre oluşan olayların depolanacağı yapı dizisinin adresidir. Biz bu yapının events elemanından oluşan olayın ne olduğunu anlarız. Yapının data elemanı epoll_ctl sırasında verdiğimiz değeri belirtir. Bizim en azından epoll_ctl fonksiyonund ilgili betimleyiciyi bu data elemanında girmiş olmamız gerekir. Fonksiyonun üçüncü parametresi ikinci parametresiyle belirtilen dizinin uzunluğudur. (Normal olarak bu dizinin eklenmiş olan betimleyici sayısı kadar olması gerekir. Ancak buradaki değer toplam izlenecek betimleyici asısından az olabilir. Bu parametre tek en fazla hamlede kaç betimleyici hakkında bilgi verileceğini belirtmektedir.) Son parametre yine milisaniye cinsinden

zaman aşımını belirtir. -1 değeri zaman aşımının kullanılmayacağını, 0 değeri hemen bakıp çıkılacağını belirtmektedir. Fonksiyon başarı durumunda diziye doldurduğu eleman sayısı ile başarısızlık durumunda -1 ile geri döner. Fonksiyon 0 değeri ile geri dönerse sonlanmanın zaman aşımından dolayı oluştuğu anlaşılır.

Sistemin kapatılması için `epoll_create` ya da `epoll_create1` ile elde edilen betimleyici kapatılabilir.

```
-----*/

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/epoll.h>

void exit_sys(const char *msg);

int main(int argc, char *argv[])
{
    int epfd;
    struct epoll_event epe;
    struct epoll_event epe_out[1];
    int result;
    char buf[1024 + 1];
    ssize_t n;

    if ((epfd = epoll_create(1)) == -1)
        exit_sys("epoll_create");

    epe.events = EPOLLIN;
    epe.data.fd = STDIN_FILENO;
    if (epoll_ctl(epfd, EPOLL_CTL_ADD, STDIN_FILENO, &epe) == -1)
        exit_sys("epoll_ctl");

    printf("waiting stdin...\n");

    if ((result = epoll_wait(epfd, epe_out, 1, -1)) == -1)
        exit_sys("epoll_wait");

    if (epe_out[0].events & EPOLLIN) {
        if ((n = read(epe_out[0].data.fd, buf, 1024)) == -1)
            exit_sys("read");
        buf[n] = '\0';
        printf("%d event(s) occurred...\n", result);
        printf("%s", buf);
    }

    return 0;
}

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

```

    exit(EXIT_FAILURE);
}

/
*-----
-----
    epoll sisteminde izlemek istediğimiz betimleyicileri ekledikten sonra
    bunları çıkarmamız gerekmez. Bu betimleyicilere
    ilişkin dosyalar kapatıldığında zaten ilgili betimleyici izlemeden
    otomatik olarak çıkartılmaktadır.

    Aşağıdaki örnekte yine bu kez epoll sistemi ile isimli borularda
    multiplexed io uygulaması yapılmıştır. Yine bu programı
    komut satırı argümanı olarak isimli boruların yolu fadelerini vererek
    çalıştırınız. Diğer terminallerden boruları yazma
    modunda cat > boru ismi biçiminde açınız.
-----
-----*/

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

#define MAX_ARGS      32
#define BUFFER_SIZE   1024

void exit_sys(const char *msg);

int main(int argc, char *argv[])
{
    int epfd;
    struct epoll_event epe;
    struct epoll_event *epe_outs;
    int fd;
    char buf[BUFFER_SIZE + 1];
    int result;
    ssize_t n;
    int nfds;
    int i;

    if (argc == 1) {
        fprintf(stderr, "too few arguments!..\n");
        exit(EXIT_FAILURE);
    }

    if ((epfd = epoll_create(2)) == -1)
        exit_sys("epoll_create");

    printf("waiting at open...\n");

    nfds = 0;
    for (i = 0; i < argc - 1; ++i) {
        if ((fd = open(argv[i + 1], O_RDONLY)) == -1)

```



```

        exit_sys("open");
epe.events = EPOLLIN;
epe.data.fd = fd;
if (epoll_ctl(epfd, EPOLL_CTL_ADD, fd, &epe) == -1)
    exit_sys("epoll_ctl");
++nfds;
}

if ((epe_outs = (struct epoll_event *)malloc(nfds * sizeof(struct
    epoll_event))) == NULL)
    exit_sys("malloc");

printf("waiting at epoll...\n");
for (;;) {
    if ((result = epoll_wait(epfd, epe_outs, nfds, -1)) == -1)
        exit_sys("epoll_wait");
    for (i = 0; i < nfds; ++i) {
        if (epe_outs[i].events & EPOLLIN) {
            if ((n = read(epe_outs[i].data.fd, buf, BUFFER_SIZE)) == -1)
                exit_sys("read");
            buf[n] = '\0';
            printf("%s", buf);
        }
        else if (epe_outs[i].events & EPOLLHUP) {
            close(epe_outs[i].data.fd);
            --nfds;
            if (!nfds)
                goto EXIT;
        }
    }
}

EXIT:
    free(epe_outs);

    return 0;
}

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

    exit(EXIT_FAILURE);
}

/
*-----

```

Linux sistemlerinde epoll özel bir ihtimamla hazırlanmıştır. Bu nedenle epoll sistemi Linux'ta select ve poll fonksiyonlarından oldukça hızlı çalışmaktadır. Pekiye neden? Çünkü Linux çekirdeği dosya nesnesinin içerisinde bu işlem için alan ayırmıştır. Biz bir betimleyiciyi izleme listesine epoll_ctl ile eklediğimizde çekirdek hemen onu ilgili bağlı listelere eklemektedir. Sonra bu betimleyicide olay gerçekleştiğinde zaten bu betimleyiciyi zaten gerçekleşen olay listesine almaktadır. Yani sonuçta

aslında betimleyicilerin çekirdek tarafından izlenmesine, gözden geçirilmesine gerek kalmamaktadır. İzlenen betimleyicilerin artması durumunda zaman kaybı "The Linux Programming Interface" kitabında 1365'inci sayfada verilmiştir:

Number of descriptors monitored (N)	poll() CPU time (seconds)
select() CPU time (seconds)	epoll CPU time (seconds)
10	0.61
0.73	0.41
100	2.9
3.0	0.42
1000	35
35	0.53
10000	990
930	0.66

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

Signal Driven IO işleminde belli bir betimleyicide olay oluşupunda SIGIO isimli sinyal oluşturulmaktadır. Böylece programcı bu oluştuğunda okuma/yazma işlemini yapabilmektedir. Ancak bu model Pgüncel POSIZ standartlarında bulunmamaktadır. Linux bu modeli desteklemektedir. Bunun için sırasıyla şunlar yapılmalıdır:

- 1) Betimleyici open fonksiyonuyla açılır.
- 2) SIGIO sinyali için sinyal fonksiyonu set edilir.
- 3) Betimleyici üzerinde F_SETOWN komut koduyla fcntl uygulanır. Üçüncü parametreye sinyalin gönderileceği prosesin ya da proses grubunun id'si girilmelidir. (Tabii genellikle programcı kendi prosesinin id'sini girer.)
- 4) Betimleyici blokesiz moda sokulur ve aynı zamanda O_SYNC bayrağı da set edilir. Bu işlem fcntl fonksiyonunda F_SETFL komut koduyla yapılabilir.

fcntl(fd, F_SETFL, fcntl(fd, F_GETFL) | O_NONBLOCK | O_ASYNC);

O_ASYNC byrağı POSIX standartlarında bulunmamaktadır.

Bu yöntemde ilgilenilen olay (yani read mı write mı) gizlice open fonksiyonundaki açış modunda belirtilmektedir. Yani örneğin biz open fonksiyonunda dosyayı O_RDONLY modunda açarsak yalnızca okuma ilgilendiğimiz, O_WRONLY modunda açarsak yalnızca yazma ile ilgilendiğimiz, O_RDWR modunda açarsak hem okuma hem de yazma ile ilgilendiğimiz sonucu çıkar.

- 5) Artık normal akışa devam eder. İlgilenilen olay gerçekleştiğinde sinyal oluşturulmaktadır.

Signal driven IO "kenar tetiklemeli (edge triggered)" bir yöntemdir. Blokesiz okuma/yazma yapılabilir bir durumda (örneğin okuma

durumunda boruya okunacak birşeyler gelmesi gibi) sinyal oluşur. Ancak okuma/yazma yapılmasa bile yeni bir benzer durum oluştuğunda yeniden sinyal oluşur. (Örneğin boruya bilgi geldiğinde sinyal oluşur. Biz borudan okuma yapmasak bile yeniden boruya bilgi gelirse yine sinyal oluşur. Halbuki select, poll böyle değildir. Anımsanacağı gibi epoll'da bu durum ayarlanabilemektedir.)

Aşağıdaki programda yine borular komut satırı argümanlarıyla verilmektedir. Borular üzerinde okuma işlemi söz konusu olduğunda SIGIO sinyali oluşacaktır. Okuma işlemi sinyal içerisinde değil dışarıda yapılmıştır. Ancak sinyal fonksiyonunda bir bayrak set edilmiştir. Yazan taraf boruyu kapattığında da yine SIGIO sinyali oluşmaktadır. Tabii bu durumda yine read fonksiyonu blokeye yol açmadan 0 ile geri dönecektir. Örnekte bekleme işleminin sigprocmask ve sigsuspend ile yapıldığına dikkat ediniz. Bu tür durumlarda pause kullanmak -pause öncesinde son bir sinyal gelirse- sorunlara yol açma potansiyelindedir.

Aslında istenirse sinyal fonksiyonu içerisinde hangi betimleyicide olay olduğu anlaşılabilir. Ama bunun için sigaction fonksiyonunda flags parametresinin SA_SIGINFO biçiminde geçilip siginfo_t parametrelili sinyal fonksiyonunun set edilmesi sağlanmalıdır. Bu siginfot yapısında Linux'ta (POSIX'te yok) si_fd elemanı SIGIO oluşmasına yol açan dosya betimleyicisini bulundurmaktadır.

-----*/

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <fcntl.h>
#include <unistd.h>
#include <signal.h>

#define MAX_ARGS      32
#define BUFFER_SIZE   1024

void sigio_handler(int sno);
void exit_sys(const char *msg);

int g_flag;

int main(int argc, char *argv[])
{
    int nfd, nfd_open;
    int fds[MAX_ARGS];
    struct sigaction sa;
    sigset_t sm, osm;
    int i;
    ssize_t n;
    char buf[BUFFER_SIZE];

    if (argc == 1) {
        fprintf(stderr, "too few arguments!..\n");
```

```

        exit(EXIT_FAILURE);
    }

    sa.sa_handler = sigio_handler;
    sigemptyset(&sa.sa_mask);
    sa.sa_flags = SA_RESTART;

    if (sigaction(SIGIO, &sa, NULL) == -1)
        exit_sys("sigaction");

    nfds = 0;
    for (i = 0; i < argc - 1; ++i) {
        if ((fds[i] = open(argv[i + 1], O_RDONLY)) == -1)
            exit_sys("open");
        if (fcntl(fds[i], F_SETOWN, getpid()) == -1)
            exit_sys("fcntl");
        if (fcntl(fds[i], F_SETFL, fcntl(fds[i], F_GETFL) | O_ASYNC |
            O_NONBLOCK) == -1)
            exit_sys("fcntl");
        ++nfds;
    }

    sigemptyset(&sm);
    sigaddset(&sm, SIGIO);
    if (sigprocmask(SIG_BLOCK, &sm, &osm) == -1)
        exit_sys("sigprocmask");

    nfds_open = nfds;
    printf("waiting at sigsuspend...\n");
    for (;;) {
        sigsuspend(&osm);
        if (g_flag) {
            for (i = 0; i < nfds; ++i) {
                if (fds[i] == -1)
                    continue;
                if ((n = read(fds[i], buf, BUFFER_SIZE)) == -1) {
                    if (errno == EAGAIN)
                        continue;
                    exit_sys("read");
                }
                if (n == 0) {
                    close(fds[i]);
                    fds[i] = -1;
                    --nfds_open;
                    if (nfds_open == 0)
                        goto EXIT;
                    continue;
                }
                buf[n] = '\0';
                printf("%s", buf);
            }

            g_flag = 0;
        }
    }
}

```

```

EXIT:
    return 0;
}

void sigio_handler(int sno)
{
    g_flag = 1;
}

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

    exit(EXIT_FAILURE);
}

```

/

*-----

İleri Modellerinden biri de "Asenkron IO Modelidir". Bu modelde okuma/yazma gibi işlemler başlatılır ancak akış devam eder. İşlemler bittiğinde durum programcıya bir sinyal ya da fonksiyon çağrısı ile bildirilir. İşlemler tipik olarak şöyle yürütülmektedir.

1) Önce struct aiocb isimli bir yapı türünden nesne tanımlayıp içinin doldurulması gerekir. Yapı şu biçimdedir:

```

struct aiocb {
    int          aio_fildes;
    off_t        aio_offset;
    volatile void *aio_buf;
    size_t        aio_nbytes;
    int          aio_reqprio;
    struct sigevent aio_sigevent;
    int          aio_lio_opcode;
};

```

Yapının aio_fildes elemanına okuma/yazma yapılmak istenen dosyaya ilişkin dosya betimleyicisi yerleştirilir. Asenkron okuma/yazma işlemleri dosya göstericisinin gösterdiği yerden itibaren yapılmamaktadır. Okuma/yazmanın dosyanın neresinden yapılacağı yapının aio_offset elemanında belirtilir. (Eğer yazma durumu söz konusuysa ve dosya O_APPEND modda açıldıysa bu durumda aio_offset elemanının değeri dikkate alınmaz. Her yazılan dosyaya eklenir.) Yapının aio_buf elemanı transfer yapılacağı bellek adresini belirtir. Bu adresteki dizinin işlem sonlanana kadar yaşıyor durumda olması gerekmektedir. Yapının aio_nbytes elemanı okunacak ya da yazılacak byte miktarını belirtmektedir. Tabii burada belirtilen byte miktarı aslında aio_buf dizisinin uzunluğunu belirtmektedir. Yoksa bildirimde bulunulacak byte sayısını belirtmez. Yani örneğin asenkron biçimde bir borudan 100 byte okumak istesek bize "işlem bitti" bildirimini 100 byte okuduktan sonra gelmek

zorunda değildir. Daha az miktarda okuma olayı gerçekleşmişse de "işlem bitti bildirimi" yapılır. Tabii hiçbir zaman burada belirtilen byte miktarından fazla okuma yazma yapılmayacaktır. Yapının aio_reprio elemanı ise okuma/yazma için bir öncelik derecesi belirtmektedir. Yani bu değer yapılacak transferin önceliğine ilişkin bir ip ucu belirtir. Ancak işletim sisteminin bu ipucunu kullanıp kullanmayacağı isteğe bağlı bırakılmıştır. Bu eleman 0 geçilebilir. Yapının aio_sigevent elemanı işlem bittiğinde yapılacak bildirim hakkında bilgileri barındırmaktadır. Bu sigevent yapısını daha önce görmüştük. Şöyleydi:

```
struct sigevent {
    int         sigev_notify;
    int         sigev_signo;
    union sigval sigev_value;
    void        (*sigev_notify_function)(union sigval);
    void        *sigev_notify_attributes;
};
```

Bu yapının sigev_notify elemanı bildirimin türünü belirtir. Bu tür SIGEV_NONE, SIGEV_SIGNAL, SIGEV_THREAD biçiminde olabilir. Yapının sigev_signo elemanı ise eğer sinyal yoluyla bildirimde bulunulacaksa sinyalin numarasını belirtmektedir. Yapının sigev_value elemanı sinyal fonksiyonuna ya da pthread fonksiyonuna gönderilecek kullanıcı tanımlı bilgiyi temsil etmektedir. Yapının sigev_notify_function elemanı eğer bildirim thread yoluyla yapılacaksa işletim sistemi tarafından yaratılan thread'in çağıracağı fonksiyonu belirtmektedir. Yapının sigev_notify_attributes elemanı ise yaratılacak thread'in özelliklerini belirtir. Bu parametre NULL geçilebilir.

2) Şimdi okuma ya da yazma olayını aio_read ya da aio_write fonksiyonuyla başlatmak gerekir. Artık akış bu fonksiyonlarda bloke olmayacak fakat işlem bitince bize bildirimde bulunulacaktır.

```
int aio_read(struct aiocb *aiocbp);
int aio_write(struct aiocb *aiocbp);
```

Fonksiyonlar başarı durumunda 0 başarısızlık durumunda -1 değerine geri dönmektedir. İşlemlerin devam ettiğine yani henüz sonlanmadığına dikkat ediniz. Bu fonksiyonlara verdiğimiz aiocb yapılarının işlem tamamlanana kadar yaşıyor olması gerekir. Yani fonksiyon bizim verdiğimiz aiocb yapısını çalışırken kullanıyor olabilir.

3) Anımsanacağı gibi biz aiocb yapısının aio_nbytes elemanına maksimum okuma/yazma miktarını vermiştik. Halbuki bundan daha az okuma/yazma yapılması mümkündür. Pekiyi bize bildirimde bulunulduğunda ne kadar miktarda bilginin okunmuş ya da yazılmış olduğunu nasıl anlayacağız? İşte bunun için aio_result isimli fonksiyon kullanılmaktadır:

```
ssize_t aio_return(struct aiocb *aiocbp);
```

Fonksiyon başarı durumunda transfer edilen byte sayısına başarısızlık durumunda -1'e geri dönmektedir. Eğer bildirim gelmeden bu fonksiyon çağrılırsa geri dönüş değeri anlamlı olmayabilir. aio_read ve aio_write fonksiyonları sinyal güvenli değildir ancak aio_Return ve aio_error fonksiyonları sinyal güvenlidir.

Aşağıdaki programda stdin dosyasından asenkron bir biçimde okuma yapılmıştır. Her aio_read için okuma yeniden başlatılmış ve bildirim olarak SIGUSR1 sinyali seçilmiştir. Ana program beklemeyi pause ile değil sigprocmask ve sigsuspend fonksiyonları ile yapmaktadır. Sinyal fonksiyonunda yalnızca bir bayrak set edilmiş asıl işlem dışarıda yapılmıştır. Sinyal fonksiyonun içerisinde aio_return yapılabilir (abii bunun için aiocb yapı nesnesinin global alınması gerekir) ancak maalesef işlemin devam ettirilmesi için aio_read yapılamaz. Çünkü aiored sinyal güvenli değildir. (Tabii bildirim olarak sinyal yerine thread yöntemi kullanılırsa artık thread fonksiyonun içerisinde yeniden aio_read işlemi yapılabilir.)

```
-----  
-----*/  
  
#include <stdio.h>  
#include <stdlib.h>  
#include <string.h>  
#include <signal.h>  
#include <aio.h>  
  
void sigusr1_handler(int sno);  
void exit_sys(const char *msg);  
  
char g_buf[1024 + 1];  
int g_flag;  
  
int main(void)  
{  
    struct aiocb cb;  
    struct sigaction sa;  
    ssize_t size;  
    sigset_t sm, osm;  
  
    sa.sa_handler = sigusr1_handler;  
    sigemptyset(&sa.sa_mask);  
    sa.sa_flags = 0;  
  
    if (sigaction(SIGUSR1, &sa, NULL) == -1)  
        exit_sys("siagaction");  
  
    cb.aio_fildes = 0;          /* stdin */  
    cb.aio_offset = 0;          /* stdin ve pipe için 0 vermek gerekir */  
    cb.aio_nbytes = 1024;  
    cb.aio_buf = g_buf;  
    cb.aio_reqprio = 0;  
    cb.aio_sigevent.sigev_notify = SIGEV_SIGNAL;  
    cb.aio_sigevent.sigev_signo = SIGUSR1;
```

```

cb.aio_sigevent.sigev_value.sival_int = 0;

sigemptyset(&sm);
sigaddset(&sm, SIGUSR1);

if (sigprocmask(SIG_BLOCK, &sm, &osm) == -1)
    exit_sys("sigprocmask");

if (aio_read(&cb) == -1)
    exit_sys("aio_read");

for (;;) {
    sigsuspend(&osm);

    if (g_flag) {
        if ((size = aio_return(&cb)) == -1)
            exit_sys("aio_return");
        g_buf[size] = '\0';
        if (!strcmp(g_buf, "quit\n"))
            break;
        printf("%s", g_buf);
        g_flag = 0;
        if (aio_read(&cb) == -1)
            exit_sys("aio_read");
    }
}

return 0;
}

```

```

void sigusr1_handler(int sno)
{
    g_flag = 1;
}

```

```

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

    exit(EXIT_FAILURE);
}

```

/

*-----

Yukarıdaki örnekte bekleme işlemi bir senkronizasyon sorunu oluşmasın diye sigprocmask ve sigsuspend fonksiyonları yardımıyla yapılmıştır. Aslında bunun yerine istenirse aio_suspend isimli fonksiyondan da faydalanılabilir.

```

int aio_suspend(const struct aiocb *const list[], int nent, const struct
timespec *timeout);

```

Fonksiyonun birinci paramtresi beklenecek asenkron olaylara ilişkin aiocb yapılarının bulunduğu dizinin adresini almaktadır.

Yani fonksiyon aslında birden fazla olayı bekleyebilmektedir. İkinci parametre birinci parametredeki dizinin uzunluğunu belirtir. Son parametre zaman aşımı belirtmektedir. NULL geçilirse zaman aşımı kullanılmaz. Fonksiyon başarı durumunda 0, başarısızlık durumunda (yani zaman aşımı ya da sinyal oluşma durumunda) -1'e geri dönmektedir. Geri dönüş değerinin kontrol edilmesine genellikle gerek olmaz. aio_suspend fonksiyonu eğer başlatılan io olayı sona ermişse hiç bloke oluşturmamaktadır. (Halbuki örneğin pause fonksiyonu yalnızca sinyal oluştuğunda geri dönmektedir.)

Şimdi yukarıdaki programı sigprocmask ve sigsuspend yerine aio_suspend ile yeniden deneyelim.

```
-----*/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <signal.h>
#include <aio.h>

void sigusr1_handler(int sno);
void exit_sys(const char *msg);

char g_buf[1024 + 1];
int g_flag;

int main(void)
{
    struct aiocb cb;
    const struct aiocb *pcb;
    struct sigaction sa;
    ssize_t size;

    sa.sa_handler = sigusr1_handler;
    sigemptyset(&sa.sa_mask);
    sa.sa_flags = 0;

    if (sigaction(SIGUSR1, &sa, NULL) == -1)
        exit_sys("siagaction");

    cb.aio_fildes = 0;          /* stdin */
    cb.aio_offset = 0;          /* stdin ve pipe için 0 vermek gerekir */
    cb.aio_nbytes = 1024;
    cb.aio_buf = g_buf;
    cb.aio_reqprio = 0;
    cb.aio_sigevent.sigev_notify = SIGEV_SIGNAL;
    cb.aio_sigevent.sigev_signo = SIGUSR1;
    cb.aio_sigevent.sigev_value.sival_int = 0;

    if (aio_read(&cb) == -1)
        exit_sys("aio_read");

    pcb = &cb;
    for (;;) {
```

```

    aio_suspend(&pcb, 1, NULL);

    if (g_flag) {
        if ((size = aio_return(&cb)) == -1)
            exit_sys("aio_return");
        g_buf[size] = '\0';
        if (!strcmp(g_buf, "quit\n"))
            break;
        printf("%s", g_buf);
        g_flag = 0;
        if (aio_read(&cb) == -1)
            exit_sys("aio_read");
    }

    return 0;
}

```

```

void sigusr1_handler(int sno)
{
    g_flag = 1;
}

```

```

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

    exit(EXIT_FAILURE);
}

```

/

*-----

aio_read ya da aio_write fonksiyonlarında belirtilen işlem bittiğinde istenirse sigevent yapısı yoluyla bildirim biçimi SIGEV_THREAD seçilebilir. Bu durumda işlem bittiğinde işletim sistemi tarafından bir thread yaratılacak ve o thread akışı tarafından belirlenen fonksiyon çağrılacaktır. İşlemlerin devam ettirilmesi bu thread fonksiyonu tarafından yapılabilir.

Aşağıdaki örnekte program komut argümanlarıyla fifo dosyalarının yol ifadelerini almaktadır. Bunları asenkron io yöntemiyle okumaktadır. Her io olayı bittiğinde belirlenen fonksiyon çağrılmış ve okunan bilgiler ekrana yazdırılmıştır. Bu programda struct aiocb yapısının ve transfer alanının (buffer'ın) bir yapıda tutulduğuna dikkat ediniz. Bu teknik bu tür durumlarda sık kullanılmaktadır. Çünkü bazen (özellikle soket uygulamalarında) programcı tampon alanda biriktirme yapıp birikmiş olan bilgiyi işleme sokmak isteyebilir.

-----*/

```

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

```

```

#include <aio.h>

#define MAX_ARGS      32
#define BUFFER_SIZE   102

void read_completion_proc(union sigval val);
void exit_sys(const char *msg);

typedef struct {
    struct aiocb cb;
    char buf[BUFFER_SIZE + 1];
} IOCB_BUF;

int main(int argc, char *argv[])
{
    int nfd;
    int fds[MAX_ARGS];
    IOCB_BUF *cbbufs[MAX_ARGS];
    int i;

    if (argc == 1) {
        fprintf(stderr, "too few arguments!..\n");
        exit(EXIT_FAILURE);
    }

    printf("waiting for pipe to be opened...\n");
    nfd = 0;
    for (i = 0; i < argc - 1; ++i) {
        if ((fds[i] = open(argv[i + 1], O_RDONLY)) == -1)
            exit_sys("open");

        if ((cbbufs[i] = (IOCB_BUF *)calloc(1, sizeof(IOCB_BUF))) == NULL)
            exit_sys("calloc");

        cbbufs[i]->cb.aio_fildes = fds[i];
        cbbufs[i]->cb.aio_offset = 0;
        cbbufs[i]->cb.aio_buf = cbbufs[i]->buf;
        cbbufs[i]->cb.aio_nbytes = BUFFER_SIZE;
        cbbufs[i]->cb.aio_reqprio = 0;
        cbbufs[i]->cb.aio_sigevent.sigev_notify = SIGEV_THREAD;
        cbbufs[i]->cb.aio_sigevent.sigev_value.sival_ptr = cbbufs[i];
        cbbufs[i]->cb.aio_sigevent.sigev_notify_function =
            read_completion_proc;

        if (aio_read(&cbbufs[i]->cb) == -1)
            exit_sys("aio_read");

        ++nfd;
    }

    printf("waiting at getchar...\n");
    getchar();

    for (i = 0; i < nfd; ++i) {
        close(cbbufs[i]->cb.aio_fildes);
    }
}

```

```

        free(cbbufs[i]);
    }

    return 0;
}

void read_completion_proc(union sigval val)
{
    IOCB_BUF *cbbuf = (IOCB_BUF *)val.sival_ptr;
    ssize_t n;

    if ((n = aio_return(&cbbuf->cb)) == -1)
        exit_sys("aio_return");

    if (n == 0) {
        printf("closing pipe...\n");
        return;
    }

    cbbuf->buf[n] = '\0';
    printf("%s", cbbuf->buf);

    if (aio_read(&cbbuf->cb) == -1)
        exit_sys("aio_read");
}

```

```

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

    exit(EXIT_FAILURE);
}

```

```

/
*-----
-----
    Yukarıdaki program şöyle de düzenlenebilirdi
-----
-----*/

```

```

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

```

```

#define MAX_ARGS      32
#define BUFFER_SIZE   102

```

```

void read_completion_proc(union sigval val);
void exit_sys(const char *msg);

```

```

typedef struct {
    struct aiocb cb;
    char buf[BUFFER_SIZE + 1];
}

```

```

} IOCB_BUF;

int main(int argc, char *argv[])
{
    int fds[MAX_ARGS];
    IOCB_BUF *cbbuf;
    int i;

    if (argc == 1) {
        fprintf(stderr, "too few arguments!..\n");
        exit(EXIT_FAILURE);
    }

    printf("waiting for pipe to be opened...\n");
    for (i = 0; i < argc - 1; ++i) {
        if ((fds[i] = open(argv[i + 1], O_RDONLY)) == -1)
            exit_sys("open");

        if ((cbbuf = (IOCB_BUF *)calloc(1, sizeof(IOCB_BUF))) == NULL)
            exit_sys("malloc");

        cbbuf->cb.aio_fildes = fds[i];
        cbbuf->cb.aio_offset = 0;
        cbbuf->cb.aio_buf = cbbuf->buf;
        cbbuf->cb.aio_nbytes = BUFFER_SIZE;
        cbbuf->cb.aio_reqprio = 0;
        cbbuf->cb.aio_sigevent.sigev_notify = SIGEV_THREAD;
        cbbuf->cb.aio_sigevent.sigev_value.sival_ptr = cbbuf;
        cbbuf->cb.aio_sigevent.sigev_notify_function = read_completion_proc;

        if (aio_read(&cbbuf->cb) == -1)
            exit_sys("aio_read");
    }

    printf("waiting at getchar...\n");

    getchar();

    return 0;
}

void read_completion_proc(union sigval val)
{
    IOCB_BUF *cbbuf = (IOCB_BUF *)val.sival_ptr;
    ssize_t n;

    if ((n = aio_return(&cbbuf->cb)) == -1)
        exit_sys("aio_return");

    if (n == 0) {
        close(cbbuf->cb.aio_fildes);
        free(cbbuf);
        return;
    }
}

```

```

    cbbuf->buf[n] = '\0';
    printf("%s", cbbuf->buf);

    if (aio_read(&cbbuf->cb) == -1)
        exit_sys("aio_read");
}

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

    exit(EXIT_FAILURE);
}

/
*-----
-----
    Bu tür uygulamalarda (aslında multiplexed io da böyle) programcı önce
    biriktirme yapıp sonra belli bir tampon dolduğunda
    biriktirdiklerini işleme sokmak isteyebilir. Bu tür isteklerle özellikle
    soket uygulamalarda çok karşılaşılmaktadır.
    Aşağıda bu biçimde biriktirerek işleme sokmaya yönelik bir örnek
    verilmiştir
-----
-----*/

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

#define MAX_ARGS      32
#define MSG_SIZE      10

void read_completion_proc(union sigval val);
void exit_sys(const char *msg);

typedef struct {
    struct aiocb cb;
    char buf[MSG_SIZE + 1];
    size_t index;
    size_t left;
} IOCB_BUF;

int main(int argc, char *argv[])
{
    int fds[MAX_ARGS];
    IOCB_BUF *cbbuf;
    int i;

    if (argc == 1) {
        fprintf(stderr, "too few arguments!..\n");
        exit(EXIT_FAILURE);
    }

```

```

printf("waiting for pipe to be opened...\n");
for (i = 0; i < argc - 1; ++i) {
    if ((fds[i] = open(argv[i + 1], O_RDONLY)) == -1)
        exit_sys("open");

    if ((cbbuf = (IOCB_BUF *)calloc(1, sizeof(IOCB_BUF))) == NULL)
        exit_sys("malloc");

    cbbuf->index = 0;
    cbbuf->left = MSG_SIZE;
    cbbuf->cb.aio_fildes = fds[i];
    cbbuf->cb.aio_offset = 0;
    cbbuf->cb.aio_buf = cbbuf->buf;
    cbbuf->cb.aio_nbytes = MSG_SIZE;
    cbbuf->cb.aio_reqprio = 0;
    cbbuf->cb.aio_sigevent.sigev_notify = SIGEV_THREAD;
    cbbuf->cb.aio_sigevent.sigev_value.sival_ptr = cbbuf;
    cbbuf->cb.aio_sigevent.sigev_notify_function = read_completion_proc;

    if (aio_read(&cbbuf->cb) == -1)
        exit_sys("aio_read");
}

printf("waiting at getchar...\n");

getchar();

return 0;
}

void read_completion_proc(union sigval val)
{
    IOCB_BUF *cbbuf = (IOCB_BUF *)val.sival_ptr;
    ssize_t n;

    if ((n = aio_return(&cbbuf->cb)) == -1)
        exit_sys("aio_return");

    if (n == 0) {
        close(cbbuf->cb.aio_fildes);
        free(cbbuf);
        return;
    }

    cbbuf->index += n;
    cbbuf->left -= n;

    if (cbbuf->left == 0) {
        cbbuf->buf[MSG_SIZE] = '\0';
        printf("Buffer filled: %s\n", cbbuf->buf);
        cbbuf->index = 0;
        cbbuf->left = MSG_SIZE;
    }
}

```

```

    cbbuf->cb.aio_nbytes = cbbuf->left;
    cbbuf->cb.aio_buf = cbbuf->buf + cbbuf->index;
    if (aio_read(&cbbuf->cb) == -1)
        exit_sys("aio_read");
}

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

    exit(EXIT_FAILURE);
}

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

aio_error isimli fonksiyon herhangi bir durumda başlatılan işlemin
akibeti konusunda bilgi almak için kullanılabilir.

int aio_error(const struct aiocb *aiocbp);

Fonksiyonun geri dönüş değeri bu asenkron işlemin o anda ne durumda
olduğu hakkında bize bilgi verir. Eğer fonksiyon
EINPROGRESS değerine geri dönerse işlemin hala devam ettiği anlamı
çıkar. Geri dönüş değeri ECANCELED ise bu durumda
işlem aio_cancel fonksiyonuyla iptal edilmiştir. Fonksiyon errno
değerini set etmez. Geri dönüş değeri diğer pozitif değerlerden birisi
ise hata ile ilgili başka bir errno değerini belirtir. Fonksiyon işlem
başarılı bir biçimde işlem bitmişse 0
değerine geri döner.
-----
-----*/

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

aio_cancel fonksiyonu ise başlatılmış olan bir işlemi iptal etmek için
kullanılmaktadır.

int aio_cancel(int fd, struct aiocb *aiocbp);

Fonksiyonun birinci parametresi iptal edilecek betimleyiciyi belirtir.
Eğer iocb NULL geçilirse bu betimleyiciye ilişkin
bütün asenkron işlemler iptal edilmektedir.

Fonksiyon AIO_CANCELED değerine geri dönerse iptal başarılıdır.
AIO_NOTCANCELED değerine geri dönerse işlem aktif biçimde
devam etmekte olduğu için iptal başarısızdır. AIO_ALLDONE değeri ise
işlemin zaten bittiğini belirtir. Fonksiyon başarısızlık
durumunda -1 değerine geri döner.
-----
-----*/

```


/

*-----

Pek çok uygulamada değişik adreslerdeki bilgilerin peşi sıraya dosyaya yazılması ya da ters olarak okunması gerekebilmektedir. Örneğin bir kaydı temsil eden aşağıdaki üç bilginin birbiri ardına dosyaya yazılmak istendiğini düşünelim:

```
int record_len;
char record[RECORD_SIZE];
int record_no;
```

Bu bilgilerin dosyaya yazılması için normal olarak üç ayrı write işlemi yapmak gerekir:

```
if (write(fd, &record_len, sizeof(int)) != sizeof(int)) {
    ...
}

if (write(fd, record, RECORD_SIZE) != RECORD_SIZE) {
    ...
}

if (write(fd, &record_no, sizeof(int)) != sizeof(int)) {
    ...
}
```

Üç write işlemi görece bir zaman kaybı oluşturabilmektedir. Tabii zaman kaybı uygulamaların ancak çok azında önem oluşturur. Buradaki zaman kaybının en önemli nedeni her write çağrısının kernel mode'a geçiş yapmasıdır. Eğer bu zaman kaybını aşağı çekmek istiyorsak ilk gelen yöntem önce bu bilgileri başka bir tampona kopyalayıp tek bir write işlemi yapmaktır:

```
char buf[BUFSIZE];

memcpy(buf, &recordlen, sizeof(int));
memcpy(buf + sizeof(int), record, RECORD_SIZE);
memcpy(buf + sizeof(int) + RECORD_SIZE, &record_no, sizeof(int));

if (write(fd, buf, 2 * sizeof(int) + RECORD_SIZE) != 2 * sizeof(int) +
    RECORD_SIZE) {
    ....
}
```

Bu işlem üç ayrı write işlemine göre oldukça hızlıdır. İşte readv ve writev isimli fonksiyonlar farklı adreslerdeki bilgileri yukarıdakine benzer biçimde dosyaya yazıp dosyadan okumaktadır. Bu işlemlere İngilizce "scatter/gather IO" denilmektedir. readv ve writev fonksiyonlarının prototipleri şöyledir:

```
ssize_t readv(int fildes, const struct iovec *iov, int iovcnt);
ssize_t writev(int fildes, const struct iovec *iov, int iovcnt);
```

Fonskiyonların birinci parametreleri okuma ya da yazma işleminin yapılacağı dosya betimleyicisini, ikinci parametreleri kullanılacak tampon uzunluğun belirtildiği yapı dizisinin adresini, üçüncü parametresi de bu dizinin uzunluğunu belirtir. Programcı struct iovec türünden bir yapı dizisi oluşturmuş onun içeriğini doldurmalıdır. Geri dönüş değeri başarısızlık durumunda -1, diğer durumlarda okunan yazılan byte miktarıdır. Okuma ve yazma işlemleri tek parça haline atomik biçimde yapılmaktadır. iovec yapısı şöyle bildirilmiştir:

```
struct iovec {
    void *iov_base;
    size_t iov_len;
};
```

Aşağıdaki programda üç ayrı adresteki yazılar peşi sıra tek bir writev çağırısı ile dosyaya yazdırılmıştır.

```
-----*/

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

#define BUFFER_SIZE      10

void exit_sys(const char *msg);

int main(void)
{
    int fd;
    char *buf1[BUFFER_SIZE];
    char *buf2[BUFFER_SIZE];
    char *buf3[BUFFER_SIZE];
    struct iovec vec[3];

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

    memset(buf1, 'a', BUFFER_SIZE);
    memset(buf2, 'b', BUFFER_SIZE);
    memset(buf3, 'c', BUFFER_SIZE);

    vec[0].iov_base = buf1;
    vec[0].iov_len = BUFFER_SIZE;

    vec[1].iov_base = buf2;
    vec[1].iov_len = BUFFER_SIZE;

    vec[2].iov_base = buf3;
    vec[2].iov_len = BUFFER_SIZE;
```

```

    if (writev(fd, vec, 3) == -1)
        exit_sys("writev");

    close(fd);

    return 0;
}

```

```

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

    exit(EXIT_FAILURE);
}

```

```

/
*-----
-----
    Aşağıdaki programda da readv kullanılarak yukarıdaki işlemin tersi
    yapılmıştır
-----
-----*/

```

```

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

```

```

#define BUFFER_SIZE      10

```

```

void exit_sys(const char *msg);

```

```

int main(void)
{
    int fd;
    char *buf1[BUFFER_SIZE];
    char *buf2[BUFFER_SIZE];
    char *buf3[BUFFER_SIZE];
    struct iovec vec[3];

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

    vec[0].iov_base = buf1;
    vec[0].iov_len = BUFFER_SIZE;

    vec[1].iov_base = buf2;
    vec[1].iov_len = BUFFER_SIZE;

    vec[2].iov_base = buf3;
    vec[2].iov_len = BUFFER_SIZE;
}

```

```

    if (readv(fd, vec, 3) == -1)
        exit_sys("writev");

    write(1, buf1, BUFFER_SIZE);
    write(1, buf2, BUFFER_SIZE);
    write(1, buf3, BUFFER_SIZE);

    close(fd);

    return 0;
}

```

```

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

    exit(EXIT_FAILURE);
}

```

/

Arka planda sessiz sedasız çalışan bir kullanıcı arayüzü olmayan, kullanıcılarla etkileşmeyen programlara Windows dünyasında "service" UNIX/Linux dünyasında "daemon" denilmektedir. Servisler ya da daemon'lar tipik olarak boo sırasında çalışmaya başlatılırlar ve yine tipik olarak makine rebbot edilene kadar çalışmaya devam ederler. Tabii böyle bir zorunluluk yoktur. Servis ya da "daemon" kernel mod bir kavram değildir. Yani servisler ve daemon'lar genellikle "user mode"da yazılırlar. UNIX/Linux dünyasında geleneksel olarak "daemon"lar xxxxd biçiminde sonuna 'd' harfi getirilerek isimlendirilmektedir. Çekirdeğe ilişkin bazı thread'ler de servis benzeri işlemler yaptıkları için bunlar da çoğu kez sonu 'd' ile bitecek ancak başı da 'k' ile başlayacak biçimde isimlendirilmiştir. Bu kernel daemon'ların bizim şu andaki konumuz olan daemon'larla hiçbir ilgisi yoktur. Yalnızca işlev bakımından bir benzerlikleri vardır. UNIX/Linux dünyasında daemon dendiğinde akla tabii ki "server programlar" gelir. Örneğin ftp server programının ismi "ftpd" biçiminde olabilir. Ya da örneğin HTTP server programının ismi "httpd" biçiminde olabilir.

daemon'lar genellikle arka planda önemli işlemler yaptıkları için "root (process id 0)" hakkıyla (yani sudo ile) çalıştırılırlar. Daemon programlar pek çok modern UNIX/Linux sisteminde init paketleri içerisindeki özel utility'ler tarafından başlatılıp, sürdürülüp sonlandırılmaktadır. Yani ilgili dağıtımın bu daemon'ları idare etmek için özel komutları bulunabilmektedir. Linux sistemlerinde init prosesi ve diğer proseslerin kodları ve boot süreci ile ilgili utility'ler "init paketleri" denilen ve farklı proje grupları tarafından oluşturulmuştur. Ve tipik olarak bugüne kadar yaygın 3 init paketi kullanılmıştır:

1) sysvinit: Klasik System5'teki işlevleri yapan init paketi. Linux uzun bir süre bu paketi kullanmıştır.

- 2) Upstart: 2006 yılında oluşturulmuştur ve 2010'ların ortalarına kadar (bazı dağıtımlarda hala) kullanılmaya devam edilmiştir.
- 3) systemd: 2010 yılında oluşturulmuştur ve son yıllarda pek çok Linux dağıtımında kullanılmaya başlanmıştır.

Bir daemon programın yazılması tipik olarak şu aşamalardan geçilerek yapılmaktadır:

1) Daemon programlar bir dosya açmak istediklerinde tam olarak belirlenen haklarla bunu yapmalıdırlar. Bu nedenle bu proseslerin umask değerlerinin 0 yapılması uygun olur.

2) Bir prosesin daemon etkisi yaratması için terminalle bir bağlantısının kalmaması gerekir. Bu maalesef 0, 1, 2 numaralı terminal betimleyicilerinin kapatılmasıyla sağlanamaz. Bunu sağlamanın en temel yolu setsid fonksiyonunu çağırmasıdır. Anımsanacağı gibi setsid fonksiyonu yeni bir oturum (session) ve yeni bir proses grubu oluşturup ilgili prosesi bu proses grubunun ve oturumun lideri yapmaktadır. Ayrıca setsid fonksiyonu prosesin terminal ilişkisini (controlling terminal) ortadan kaldırmaktadır. Ancak setsid uygulayabilmek

için prosesin herhangi bir prosesin grup lideri olmaması gerekir. Aksi takdirde fonksiyon başarısız olmaktadır. Anımsanacağı gibi kabuk programlar çalıştırdıkları programlar için bir proses grubu yaratıp o programı da proses grup lideri yapıyordu. İşte proses grup lideri olmaktan kurtulmak için bir kez fork yapıp üst prosesi sonlandırabiliriz. Aynı zamanda bu işlem kabuk programının hemen komut satırına yeniden düşmesine yol açacaktır. O halde 2'inci aşamada fork yapıp üst proses sonlandırılmalıdır. Bilindiği gibi teeminale bağlı programlar kabukta çalıştırıldıklarında yeniden komut satırına düşsek bile kabuk programları kapatıldığında bu programlar sonlandırılmaktadır.

Yani örneğin biz terminal ilişkisini kesmezsek bir kez fork yapıp alt proses arka planda çalışır gibi olur ancak bu durumda terminal kapatıldığında o alt proses de sonlandırılır.

3) Artık alt proses setsid fonksiyonunu uygulayarak yeni bir oturum yaratır ve terminal ilişkisini keser. Terminal ilişkisinin kesilmesi ile artık terminal kapatılsa bile programımız çalışmaya devam eder. Tabii setsid ile terminal bağlantısının kesilmiş olması programın terminale bir şey yazamayacağı anlamına gelmez. Hala 0, 1, 2 numaralı betimleyiciler açıktır. Terminal açık olduğu sürece oraya yazma yapılabilir.

4) Daemon programların çalışma dizinlerinin (current working directory) sağlam bir dizin olması tavsiye edilir. Aksi takdirde o dizin silinirse arka plan bu programların çalışmaları bozulur. Bu nedenle daemon programların çoğu kök dizini (silinemeyeceği için) çalışma dizini yapmaktadır. Tabii bu zorunlu değildir. Bunun yerine varlığı garanti edilmiş olan herhangi bir dizin de çalışma dizini yapılabilir.

5) Daemon programın o ana kadar açılmış olan tüm betimleyicileri kapatması uygun olur. Örneğin 0, 1, 2 numaralı betimleyiciler

ilgili terminale ilişkindir ve artık o terminal kapatılmış ya da kapatılacak olabilir. Program kendini daemon yaptığı sırada açmış olduğu diğer dosyaları da kapatmalıdır. Bunu sağlamanın basit bir yolu prosesin toplam dosya betimleyici tablosunun uzunluğunu elde edip her bir betimleyici için close işlemi uygulamaktır. Çünkü maalesef biz açık betimleyicileri pratik bir biçimde tespit edememekteyiz. Zaten kapalı bir betimleyiciye close uygulanırsa close başarısız olur ancak program çökmez. Prosesin toplam betimleyici sayısı sysconf çağrısında _SC_OPEN_MAX argümanıya ya da getrlimit fonksiyonunda RLIMIT_NOFILE argümanıya elde edilebilir. İki fonksiyon da aynı değeri vermektedir.

6) Zorunlu olmamakla birlikte ilk üç betimleyiciyi /dev/null aygıtına yönlendirmek iyi bir fikirdir. Çünkü bir biçimde bazı fonksiyonlar bu betimleyicileri kullanıyor olabilirler. Anımsanacağı gibi /dev/null aygıtına yazılanlar kaybolmaktadır. Bu aygıttan okuma yapılmak istendiğinde ise EOF etkisi oluşur.

Pekiye daemon'lar ne yaparlar? İşte daemon2lar arka planda genellikle sürekli bir biçimde birtakım işler yapmaktadır. Bu anlamda en tipik daemon örnekleri "server" programlardır. Örneğin http server aslında httpd isimli bir daemon'dan ibarettir. Bunun gibi UNIX/Linux sistemlerinde genellikle boot zamanında devreye giren onlarca daemon program vardır. Örneğin belli zamanlarda belli işlerin yapılması için kullanılan cron utility'si aslında bir daemon olarak çalışmaktadır.

Aşağıda bir prosesi daemon haline getiren bir fonksiyon örneği verilmiştir. Bu fonksiyon main fonksiyonun çağrılmış böylece proses daemon haline getirilmiştir.

-----*/

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

```
void make_daemon(void)
{
    pid_t pid;
    int maxfd;
    int fd;
    int i;

    umask(0);

    if ((pid = fork()) == -1) {
        perror("fork");
        exit(EXIT_FAILURE);
    }

    if (pid != 0)
```

```

        exit(EXIT_SUCCESS);

if (setsid() == -1) {
    perror("setsid");
    exit(EXIT_FAILURE);
}

if (chdir("/") == -1)
    exit(EXIT_FAILURE);

maxfd = sysconf(_SC_OPEN_MAX);
for (i = 0; i < maxfd; ++i)
    close(i);

if ((fd = open("/dev/null", O_RDWR)) == -1)
    exit(EXIT_FAILURE);

if (dup(fd) == -1 || dup2(fd, 1) == -1)
    exit(EXIT_FAILURE);
}

int main(void)
{
    make_daemon();

    /* daemon kodu, muhtemelen bir server program */

    return 0;
}

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

```

Pekiyi mademki daemon programların terminal ilişkileri yoktur, o zaman bu programlar dış dünyaya nasıl bildirimde bulunacaklardır? İlk akla gelen yöntem önceden belirlenmiş bazı dosyalara yazmak olabilir. Ancak her daemon'ın kendi belirlediği dosyalara yazması karışık bir durum oluşturabilmektedir. Bazı daemon'lar çok uzun süre çalışırlar bu da onların yazdıkları log'ların çok büyümesine yol açar. UNIX türevi sistemlerde daha genel ve merkezi bir log mekâzisması düşünülmüştür. Bu mekanizma sayesinde farklı daemon'lar aynı log dosyasına bildirimleri yazarlar. Böylece hem daha güvenli hem daha makul bir loglama sistemi oluşturulmuş olur.

Merkezi loglama için üç temel POSIX fonksiyonu kullanılmaktadır: openlog, syslog ve closelog fonksiyonları. Log oluşturmadan önce openlog fonksiyonu çağrılarak bazı belirlemeler yapılır. Fonksiyonun prototipi şöyledir:

```
void openlog(const char *ident, int logopt, int facility);
```

Fonksiyonun birinci parametresi log mesajlarına görüntülenecek program ismini belirtmektedir. Genellikle programcılar bu parametre için program ismini argüman olarak verirler. Linux sistemlerinde bu parametreye NULL geçilebilmektedir. Bu durumda sanki bu parametre için program ismi yazılmış gibi işlem yapılır. Ancak POSIX standartlarında NULL geçme durumu belirtilmemiştir. İkinci parametre aşağıdaki sembolik sabitlerin bit or işlemine sokulmasıyla oluşturulabilir:

LOG_PID
LOG_CONS
LOG_NDELAY
LOG_ODELAY
LOG_NOWAIT

Burada LOG_PID log mesajında prosesin proses id'sinin de bulunduğunu belirtir. LOG_CONS log mesajlarının aynı zamanda default console (/dev/console) da yazılacağını belirtmektedir. Bu parametre için argüman 0 olarak da girilebilir. Fonksiyonun üçüncü parametresi log mesajını yollayan prosesin kim olduğu hakkında temel bir bilgi vermek için düşünülmüştür. Bu parametre LOG_USER olarak girilebilir. LOG_USER bir user proses tarafından bu loglamanın yapıldığını belirtmektedir. LOG_KERN mesajın kernel tarafından gönderildiğini belirtir. LOG_DAEMON mesajın bir sistem daemon programı tarafından gönderildiğini belirtmektedir. LOG_LOCAL0'dan LOG_LOCAL7'ye kadarki sembolik sabitler özel log kaynaklarını belirtmektedir. Fonksiyon başarısız olamamaktadır. bu nedene geri dönüş değeri void yapılmıştır. Bu parametre de 0 geçilebilir. Bu durumda LOG_LOCAL0 anlaşılır.

syslog fonksiyonu asıl log işlemini yapan fonksiyondur. Aslında syslog için işin başında openlog çağrısının yapılmasına da gerek yoktur. Bu durumda default belirlemeler kullanılmaktadır. Fonksiyonun parametrik yapısı şöyledir:

```
void syslog(int priority, const char *format, ...);
```

Fonksiyonun birinci parametresi mesajın öncelik derecesini (yani önemini) belirtir. Diğer parametreler tamamen printf fonksiyonundaki gibidir.

Öncelik değerleri şunlardır:

LOG_EMERG
LOG_ALERT
LOG_CRIT
LOG_ERR
LOG_WARNING
LOG_NOTICE
LOG_INFO
LOG_DEBUG

En çok kullanılanlar error mesajları için LOG_ERR, uyarı mesajları için LOG_WARNING ve genel bilgilendirme mesajları için LOG_INFO değerleridir. syslog için openlog çağrılmak zorunda olmadığından syslog fonksiyonunun birinci parametresi ile istenirse openlog fonksiyonunun üçüncü parametresi kombine edilebilir.

Nihayet eğer proses birmeden log sistemi prosese kapatılmak isteniyorsa closelog fonksiyonu çağrılmalıdır:

```
void closelog(void);
```

Aşağıda log fonksiyonlarının kullanımına ilişkin bir örnek verilmiştir

```
-----*/
#include <stdio.h>
#include <stdlib.h>
#include <syslog.h>

void exit_sys(const char *msg);

int main(void)
{
    int i;

    openlog("sample", LOG_PID, LOG_LOCAL0);

    for (i = 0; i < 10; ++i)
        syslog(LOG_INFO, "This is a test: %d\n", i);

    closelog();

    return 0;
}

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

    exit(EXIT_FAILURE);
}
```

/

```
*-----
Tabii log sistemi aslında tipik olarak daemon programlar ve aygıt
sürücüler tarafından kullanılmaktadır. Aşağıda
bu sistemi kullanan bir daemon örneği görüyorsunuz. Bu program
çalışırken tail /var/log/syslog komutu ile syslog isimli
log dosyasının sonuna bakabilirsiniz.
```

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

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

```

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

void make_daemon(void)
{
    pid_t pid;
    int maxfd;
    int fd;
    int i;

    umask(0);

    if ((pid = fork()) == -1) {
        perror("fork");
        exit(EXIT_FAILURE);
    }

    if (pid != 0)
        exit(EXIT_SUCCESS);

    if (setsid() == -1) {
        perror("setsid");
        exit(EXIT_FAILURE);
    }

    if (chdir("/") == -1)
        exit(EXIT_FAILURE);

    maxfd = sysconf(_SC_OPEN_MAX);
    for (i = 0; i < maxfd; ++i)
        close(i);

    if ((fd = open("/dev/null", O_RDWR)) == -1)
        exit(EXIT_FAILURE);

    if (dup(fd) == -1 || dup(fd) == -1)
        exit(EXIT_FAILURE);
}

int main(void)
{
    int i;

    make_daemon();

    openlog("sampled", LOG_PID, LOG_LOCAL0);

    for (i = 0; i < 100; ++i) {
        syslog(LOG_INFO, "daemon running: %d\n", i);
        sleep(1);
    }
}

```

```
closelog();  
  
return 0;  
}
```

```
/
```

```
*-----
```

Pekiysi syslog log mesajlarını nereye yazmaktadır? Aslına log mesajlarını syslog fonksiyonun kendisi yazmaz. Log mesajlarının yazdırılması için özel bir daemon prosten faydalanılmaktadır. Bu proses eskiden syslogd ismindeydi, daha sonra bunun biraz daha gelişmiş versiyonu olan rsyslogd prosesi kullanılmaya başlandı. Kursun yapıldığı zamanlarda Linux sistemlerinde yaygın olarak rsyslogd isimli daemon kullanılmaktadır.

syslogd ya da rsyslogd aslında diğer proseslerden proseslerarası haberleşmeyle yazdırılacak log mesajlarını almaktadır. Tipik olarak syslog fonksiyonu

/dev/log isimli soket dosyasını (datagram UNIX domain soket) kullanarak mesajları syslogd ya da rsyslogd daemon'ına göndermektedir. (Yani aslında syslogd ya da

rsyslogd datagram server görevi de yapmaktadır). Kernel'ın kendisi de (örneğin printk fonksiyonunda) aslında neticede bu syslogd ya da rsyslogd daemon'ına

/dev/log soketi yoluyla datagram mesaj göndererek log mesajlarını yazdırmaktadır. Uzak bağlantı söz konusu olduğunda syslogd ya da rsyslogd datagram mesajları

UDP/IP 514 numaralı porttan almaktadır. İşte aslında log mesajlarının hangi dosyalara yazılacağına syslogd ya da rsyslogd daemon'ları karar vermektedir.

Bu daemon'lar çalışmaya başladıklarında default durumda /etc/syslog.conf ya da /etc/rsyslog.conf dosyalarına bakmaktadır. İşte bu daemon'ların hangi dosyalara

yazacağı bu konfigürasyon dosyalarında sistem yöneticisi tarafından belirlenebilmektedir. Ancak bu dosyada da belirleme yapılmamışsa default olarak

pek çok mesaj grubu (error, warning, info) /var/log/syslog dosyasına yazılmaktadır. O halde programcı bu mesajlar için bu dosyaya başvurmalıdır.

Log dosyalarını incelemek için pek çok utility bulunmaktadır. Örneğin lnav, glogg ksystemlog gibi.

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

```
/
```

```
*-----
```

Yukarıda da belirtildiği gibi pek çok daemon aslında sistem boot edilirken çalıştırılmakta ve sistem kapatılana kadar çalışır durumda kalmaktadır.

Fakat bazı daemon'lar ise gerektiğinde çalıştırılıp, gerekmediğinde durdurulabilmektedir. İşte UNIX/Linux sistemlerinde bu çalıştırma, durdurma gibi faaliyetler için daha yüksek seviyeli araçlar bulundurulmaktadır. Bu araçlar init prosesinin paketinde yer alırlar. Tarihsel süreç içerisinde Linux sistemlerinde boot işleminden ve servis işlemlerinden sorumlu üç önemli paket geliştirilmiştir:

systemVinit (klasik)
upstart
systemd

Kursun yapıldığı zaman diliminde ağırlıklı biçimde systemd paketi kullanılmaktadır. Sisteminizde hangi init paketinin kullanıldığını anlamak için birkaç yol söz konusu olabilir:

```
ls -l /sbin/init  
cat /proc/1/status
```

systemd init paketinin servis yönetici programı systemctl isimli programdır. Bu program yoluyla daemon işlemleri yapabilmek için öncelikle bir .service uzantılı dosyasının oluşturulması gerekir. Bu programın çalıştırabileceği servislere "unit" denilmektedir. .service uzantılı dosyada da unit bildirimleri bulunur. Sonra bu dosyanın /etc/systemd/system dizinine kopyalanması gerekmektedir. Ondan sonra asıl daemon programının da /usr/bin içerisine çekilmesi uygundur. (Tabii burada bazı seçenekler söz konusudur. Ancak biz tipik durumları ele alıyoruz. Ayrıntılı bilgi için systemd dokümantasyonuna bakınız.) Artık şu komutlar uygulanarak servis yönetilebilir:

```
systemctl start <daemon ismi> (daemon'ı çalıştırır)  
systemctl stop <daemon ismi> (daemon'ı durdurur)  
systemctl restart <daemon ismi> (durdurup yeniden başlatır)  
systemctl show <daemon ismi> (daemon'ın durumunu gösterir)  
systemctl enable <service dosyasının ismi> (boot zamanında devreye sokmak için)  
systemctl disable <service dosyasının ismi> (boot zamanında devreden çıkarmak için)
```

systemd için tipik bir .service uzantılı dosyanın içeriği şöyledir:

```
[Unit]  
Description=Example Unit  
[Service]  
Type=forking  
ExecStart=/usr/bin/sampled  
[Install]  
WantedBy=multi-user.target
```

Daemon'lar işleme başlamadna önce çeşitli parametrik bilgileri bazı konfigürasyon dosyalarından (genellikle bunların uzantıları .conf olur) okuyabilmektedir.

Sistem yöneticisi de bu dosyalarda değişiklik yapıp daemon'ı restart edebilmektedir.

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

Farklı makinelerin prosesleri arasında haberleşme (yani ağ altında haberleşme) daha çetrefil bir haberleşme biçimidir. Çünkü burada ilgili işletim sisteminin dışında pek çok belirlemelerin önceden yapılmış olması gerekir. İşte ağ haberleşmesinde önceden belirlenmiş kurallar topluluğuna "protokol" denilmektedir. Ağ haberleşmesi için tarihsel süreç içerisinde pek çok protokol gerçekleştirilmiştir. Bunların bazıları büyük şirketlerin kontrolü altındadır. Ancak açık bir protkol olan IP protokol ailesi günümüzde hemen her zaman tercih edilen protokol ailesidir.

Protokol ailesi (protocol family) denildiğinde birbirleriyle ilişkili bir grup protokol anlaşılır. Ailenin pek çok protokolü başka protokollerin üzerine konumlandırılmıştır. Böylece protokol aileleri katmanlı (layered) bir yapıya sahip olmuştur. Üst seviye bir protokol alt seviye protokolün zaten var olduğu fikriyle o alt seviye protokol kullanılarak oluşturulur.

ISO Bir protokol ailesinde katmanlı olarak hangi tarzda protokollerin bulundurulabileceğine yönelik OSI (Open System Interconnection) isimli bir referans dokümanı oluşturmuştur. Bunlara OSI katmanları denilmektedir. OSI'nin 7 katmanı vardır. Aşağıdaki yukarıya bunlar şöyledir: Physical Layer, Data Link Layer, Network Layer, Transport Layer, Session Layer, Presentation Layer, Application Layer. En aşağı seviyeli elektriksel tanımlamaların yapıldığı katmana "fiziksel katman" denilmektedir. (Örneğin kabloların, konnektörlerin özellikleri vs.)

Veri Bağlantı Katmanı artık bilgisayarlar arasında fiziksel bir adreslemenin yapıldığı ve bilgilerin paketlere ayrılarak gönderilip alındığı

bir ortam tanımlarlar. Örneğin bugün kullandığımız Ethernet kartları MAC adresi denilen dünya genelinde tek olan adresler yoluyla paket paket (packet switching)

bilginin gönderilip alınmasını sağlar. Bu yüzden Ethernet protokolü "veri bağlantı katmanına ilişkindir." Ağ Katmanı (Network layer) artık "internetworking" yapmak

için gerekli kuralları tanımlar. "Internetworking" terimi network'lerden oluşan network'ler anlamına gelir. Tipik olarak internetworking yapmak için

yerel ağlar (local area networks) "router" denilen aygıtlarla birbirine bağlanmaktadır. Ağ katmanında artık fiziksel bir adresleme değil, mantıksal adresleme

sistemi kullanılmaktadır. Ayrıca bilgilerin paketlere ayrılarak router'lerden dolaşıp hedefe varması için rotalama mekanizması da bu katmanda tanımlanmaktadır.

Yani elimizde yalnızca network katmanı varrsa bi yalnızca "internetworking" ortamında belli bir kaynaktan hedefe bir paket yollayabiliriz. Transport katmanları network katmanlarının üzerindedir. Transpot katmanında artık kaynak ile hedef arasında bir bağlantı oluşturulabilmekte ve veri aktarımı daha güvenli olarak yapılabilmektedir. Aynı zamanda transport kavramı "multiplex" bir kaynak hedef yapısı da oluşturmaktadır. Bu sayede hedefe bilgiler oradaki spesifik bir programa gönderilebilmektedir. Oturum katmanı pek çok ailede yoktur. Görevi oturum açma kapama gibi yüksek seviyeli bazı belirlemeleri yapmaktır. Presentation katmanı verilerin sıkıştırılması, şifrelenemesi gibi tanımlamalar içermektedir. Nihayet bu protokolü kullanan bütün programlar aslında uygulama katmanını oluşturmaktadır. Yani ağ ortamında haberleşen her program zaten kendi içerisinde açık ya da gizli bir protokol oluşturmuş durumdadır.

IP ailesi neden bu kadar popüler olmuştur? Bunun en büyük nedeni 1983 yılında hepimizin katıldığı Internet'in (I'nin büyük yazıldığına dikkat ediniz)

bu aileyi kullanmaya başlamasındadır. Böylece IP ailesini kullanarak yazdığımız programlar hem aynı bilgisayarda hem yerel ağımızdaki bilgisayarlarda hem de

Internet'te çalışabilmektedir. Aynı zamanda IP ailesinin açık bir (yani bir şirketin malı değil) protokol olması da cazibeyi çok artırmıştır.

IP ailesi 70'li yıllarda Vint Cerf ve Bob Kahn tarafından geliştirilmiştir. IP ismi Internet Protocol'den gelmektedir. Burada internet "internetworking" anlamında kullanılmıştır.

Bugün hepimizin bağlandığı büyük ağa da "Internet" denilmektedir. Bu ağ ilk kez 1969 yılında Amerika'da Amerikan Savunma Bakanlığının bir soğuk savaş

projesi biçiminde başlatıldı. O zamana kadar yalnızca yerel ağlar vardı. 1969 yılında ilk kez bir "WAN (Wide Area Network)" oluşturuldu.

Bu proje Amerikan savunma bakanlığının DARPA isimli araştırma kurumu tarafından başlatılmıştır ve ARPA.NET ismi verilmiştir. Daha bu ağa Amerika'daki

çeşitli devlet kurumları ve üniversiteler katıldı. Sonra ağ Avrupa'ya sıçradı. 1983 yılında bu ağ NCP protokolünden IP protokol ailesine geçiş yaptı.

Bundan sonra artık APRA.NET ismi yerine "Internet" ismi kullanılmaya başlandı.

IP prtokol ailesi 4 katmanlı bir ailedir. Fiziksel ve Veri Bağlantı Katmanı bir arada düşünülebilir. Bugün bunlar Ethernet ve Wireless protokolleri

biçiminde pratikte kullanılmaktadır. IP ailesinin ağ katmanı aileye ismini veren IP protokolünden oluşmaktadır. Transport katmanı ise TCP ve UDP

protokollerinden oluşur. Nihayet TCP üzerine oturtulmuş olan HTTP, TELNET, SSH, POP3, IMAP gibi pek çok protokol ailenin uygulama katmanını oluşturmaktadır.

IP protokolü tek başına kullanılırsa ancak bir paket gönderip alma işini yapar. Bu nedenle bu protokolün tek başına kullanılması çok seyrek olur. Uygulamada genellikle transport katmanına ilişkin TCP ve UDP protokolleri kullanılmaktadır.

TCP "stream tabanlı", UDP "datagram (paket) tabanlı" bir protokoldür. Stream tabanlı demek tamamen boru haberleşmesinde olduğu gibi gönderen tarafın bilgilerinin bir kuyruk sistemi eşliğinde alıcıda organize edilmesi ve alıcının istediği kadar byte'ı parça parça okuyabilmesi demektir. Datagram tabanlı demek tamamen mesaj kuyruklarında olduğu gibi bilginin paket paket iletilmesi demektir. Yani datagram haberleşmede alıcı taraf gönderen tarafın tüm paketini tek hamlede almak zorundadır.

TCP bağlantılı (connection-oriented) UDP bağlantısız (connectionless) bir protokoldür. Buradaki bağlantı IP paketleriyle yapılan mantıksal bir bağlantıdır. Bağlantı sırasında gönderici ve alıcı birbirlerini tanır ve haberleşme boyunca konuşabilirler. Oysa UDP'de alıcı onun hiç bilmediği bir kullanıcıdan bilgi alabilmektedir. UDP'de gönderen ve alan arasında bir kayıt tutulmaz.

TCP güvenilir (reliable) UDP (güvenilir) olmayan (unreliable) bir protokoldür. TCP'de mantıksal bir bağlantı oluşturulduğu için yolda kaybolan paketlerin telafi edilmesi mümkündür. Alıcı taraf gönderenin bilgilerini eksiksiz ve bozulmadan aldığını bilir.

IP protokol ailesinde ağa bağlı olan birimlere "host" denilmektedir. Host bir bilgisayar olmak zorunda değildir. İşte bu protokolde her host'un mantıksal bir adresi vardır. Bu adrese IP adresi denilmektedir. IP adresi IPv4'te 4 byte uzunlukta, IPv6'da 16 byte uzunlukta. Ancak bir host'ta farklı programlar farklı host'larla haberleşiyor olabilir. İşte aynı host'a gönderilen IP paketlerinin o host'ta ayrıştırılması için "protokol port numarası" diye içsel bir numara uydurulmuştur. Port numarası bir şirketin içerisinde çalışanların dahili numarası gibi düşünülebilir. Port numaraları IPv4'te 2 byte'la, IPv6'da 4 byte'la ifade edilmektedir. İlk 1024 port numarası IP ailesinin uygulama katmanındaki protokoller için ayrılmıştır. Bunlara "well known ports" denilmektedir. Bu nedenle programcıların port numaralarını 1024'ten büyük olacak biçimde almaları gerekir. Bu durumda TCP ve UDP'de bilgiler belirli bir IP adresindeki host'un belirli bir portuna gönderilir. Gönderilen bu bilgiler de o portla ilgilenen programlar tarafından alınmaktadır.

IP haberleşmesi (yani paketlerin, oluşturulması, gönderilmesi alınması vs.) işletim sistemlerinin çekirdekleri tarafından yapılmaktadır.

Tabii User mod programlar için sistem çağrılarını yapana API fonksiyonlarına ve kütüphanelerine gereksinim vardır. İşte bunların en yaygın

kullanılanı "socket kütüphanesi" denilen kütüphanedir. Bu kütüphane ilk kez 1983 yılında 4.2BSD'de gerçekleştirilmiştir ve pek çok UNIX türevi sistem bu kütüphaneyi aynı biçimde benimsemiştir. Microsoft'un Windows sistemleri de bu API kütüphanesini desteklemektedir. Bu kütüphaneye "Winsock"

ya da "WSA (Windows Socket API)" denilmektedir. Microsoft'un Winsock kütüphanesinde hem BSD fonksiyonları orijinal haliyle bulunmakta hem de başı WSAXXX ile başlayan Windows'a özgü fonksiyonlar bulunmaktadır.

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

Bir TCP/Ip uygulamasında server ve client olmak üzere iki ayrı program yazılır. Server program şu fonksiyonlar çağrılarak oluturulmatadır:

socket, bind, listen, accept, read/write/recv/send, shutdown, close

socket fonksiyonu bir handle alanı yaratır ve bize bir dosya betimleyicisi verir. Biz diğer fonksiyonlarda socket dediğimiz bu betimleyiciyi kullanırız.

```
int socket(int domain, int type, int protocol);
```

Fonksiyonun birinci parametresi kullanılacak protokol ailesini belirtir. Bu parametre AF_XXX biçimindeki sembolik sabitlerden biri olarak girilir. IPV4 için bu parametreye AF_INET, IPV6 için AF_INET6 girilmelidir. UNIX protokolü için AF_UNIX kullanılır. İkinci parametre kullanılacak protokolün stream mi datagram mı ya da başka bir türden mi olduğunu belirtir. Stream soketler için SOCK_STREAM, datagram soketler için SOCK_DGRAM kullanılmalıdır. Başka soket türleri de vardır. Üçüncü parametre transport katmanındaki protokolü belirtmektedir.

Ancak zaten ikinci parametreden transport protokolü anlaşılıyorsa üçüncü parametre 0 geçilebilir. Örneğin IP ailesinde üçüncü parametreye gerek duyulmamaktadır. Çünkü SOCK_STREAM zaten TCP'yi SOCK_DGRAM ise UDP'yi anlatmaktadır. Fakat yine de bu parametreye istenirse IP ailesi için IPPROTO_TCP ya da IPPROTO_UDP girilebilir. (Bu sembolik sabitler <netinet/in.h> içerisinde) Fonksiyon başarılıysa soket betimleyicisine başarısızsa -1 değerine geri döner.

Server program soketi yarattıktan sonra onu bağlamalıdır (bind etmelidir). bind işlemi sırasında server'ın hangi portu dinleyeceği ve hangi network arayüzünden (kartından) gelen bağlantı isteklerini kabul edeceği belirlenir.

```
int bind(int sockfd, const struct sockaddr *addr, socklen_t addrlen);
```

Fonksiyonun birinci parametresi yaratılmış olan soket betimleyicisini alır. İkinci parametre her ne kadar sockaddr isimli

yapı türündense de aslında her protokol için ayrı bir yapı adresini almaktadır. Yani sockaddr yapısı genelliği (void gösterici gibi) temsil etmek için kullanılmıştır. IPV4 için kullanılacak yapı sockaddr_in IPV6 için, sockaddr_in6 ve örneğin Unix domain soketler için ise sockaddr_un biçiminde olmalıdır. Üçüncü parametre ikinci parametredeki yapının uzunluğu olarak girilmelidir.

sockaddr_in yapısı şöyledir:

```
struct sockaddr_in {
    sa_family_t    sin_family;
    in_port_t      sin_port;
    struct in_addr  sin_addr;
};
```

Yapının sin_family elemanına protokol ailesini belirten AF_XXX değeri girilmelidir. Bu eleman tipik olarak short biçimde bildirilmiştir. Yapının sin_port elemanı in_port_t elemanı türündendir ve bu tür uint16_t olarak typedef edilmiştir. Bu eleman server'ın dinleyeceği port numarasını belirtir. Yapının sin_addr elemanı IP numarası belirten bir elemandır. Bu eleman in_addr isimli bir yapı türündendir. Bu yapı da şöyle bildirilmiştir:

```
struct in_addr {
    in_addr_t  s_addr;
};
```

in_addr_t 4 byte'lık işaretisiz tamsayı türünü (uint32_t) belirtmektedir. Böylece s_addr 4 byte'lık IP adresini temsil eder.

IP ailesinde tüm sayısal değerler BIG ENDIAN formatıyla belirtilmek zorundadır. Bu ailede "network byte ordering" denildiğinde BIG ENDIAN anlaşılır. Oysa makinelerin belli bir bölümü (örneğin INTEL ve default ARM) LITTLE ENDIAN kullanmaktadır. İşte elimizdeki makinenin endian'lığı ne olursa olsun onu BIG ENDIAN'a dönüştüren htons (host to network byte ordering short) ve htonl (host to network byte ordering long) isimli bir iki fonksiyon vardır. Bu işlemlerin tersini yapan ntohs ve ntohl fonksiyonları da bulunmaktadır. IP adresi olarak INADDR_ANY özel bir değerdir ve "tüm network kartlarından gelen bağlantı isteklerini kabul et" anlamına gelir. Bu durumda sockaddr_in yapısı tipik olarak şöyle doldurulabilir:

```
struct sockaddr_in sinaddr;

sinaddr.sin_family = AF_INET;
sinaddr.sin_port = htons(SERVER_PORT);
sinaddr.sin_addr.s_addr = htonl(INADDR_ANY);
```

server bind işleminden sonra soketi aktif dinleme konumuna sokmak için listen fonksiyonunu çağırmalıdır. Fonksiyonun prototipi şöyledir:

```
int listen(int socket, int backlog);
```

Fonksiyonun birinci parametresi socketin handle değeri, ikinci parametresi kuyruk uzunluğunu belirtir. listen işlemi blokeye yol açmamaktadır. İşletim sistemi listen işleminden sonra ilgili porta gelen bağlantı isteklerini uygulama için bir kuyruk sisteminde biriktirir. accept fonksiyonu bu kuyruğa bakmaktadır. Kuyruk uzunluğunu yüksek tutmak meeşgul server'larda bağlantı isteklerinin kaçırılmamasını sağlayabilir. Linux'ta default durumda verilebilecek en yüksek değer 128'dir. Ancak /proc/sys/net/core/somaxconn dosyasındaki değer yükseltilebilir. Fonksiyon başarı durumunda 0 değerine başarısızlık durumunda -1 değerine geri döner. Bu fonksiyon işletim sisteminin "firewall mekanizması" tarafından denetlenmektedir.

Nihayet asıl bağlantı accept fonksiyonuyla sağlanmaktadır. accept fonksiyonu bağlantı kuyruğuna bakar. Eğer orada bir bağlantı isteği varsa onu alır ve hemen geri döner. Eğer orada bir bağlantı isteği yoksa default durumda blokede bekler. Fonksiyonun prototipi şöyledir:

```
int accept(int socket, struct sockaddr *address, socklen_t
    *address_len);
```

Fonksiyonun birinci parametresi socketin dosya betimleyicisini almaktadır. İkinci parametre bağlanılan client'a ilişkin bilgilerin yerleştirileceği yapının adresini almaktadır. Tabii bu yapı yine protokole göre değişebilen bir yapıdır. Örneğin IPV4 için bu yapı yine sockaddr_in olmalıdır. Bu yapının içinden biz bağlanılan client'ın ip adresini, kaynak port numarasını alabiliriz. Bu bilgiler yine BIG ENDIAN formattadır. accept fonksiyonu başarı durumunda client ile konuşma işinde kullanılacak socket değerine (socket dosya betimleyicisine) başarısızlık durumunda 0 değerine geri dönmektedir.

Server programın listen ve accept işlemlerini yapmak için kullandığı sokete "pasif socket" ya da "dinleme socketi" denilmektedir. Bu pasif socket başka bir amaçla kullanılamaz. Client'larla konuşmak için accept fonksiyonun verdiği socketler kullanılır.

Aşağıda accept işlemine kadar tipik bir server örneği verilmiştir. Bu program dinlenecekprot numarasını komut satırı argümanıya almaktadır.

```
-----*/

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

void exit_sys(const char *msg);

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

```

int sock, sock_client;
struct sockaddr_in sinaddr, sinaddr_client;
socklen_t sinaddr_len;
in_port_t port;

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

port = (in_port_t)strtoul(argv[1], NULL, 10);

if ((sock = socket(AF_INET, SOCK_STREAM, 0)) == -1)
    exit_sys("socket");

sinaddr.sin_family = AF_INET;
sinaddr.sin_port = htons(port);
sinaddr.sin_addr.s_addr = htonl(INADDR_ANY);

if (bind(sock, (struct sockaddr *)&sinaddr, sizeof(sinaddr)) == -1)
    exit_sys("bind");

if (listen(sock, 8) == -1)
    exit_sys("listen");

printf("Waiting for connection...\n");

sinaddr_len = sizeof(sinaddr_client);
if ((sock_client = accept(sock, (struct sockaddr *)&sinaddr_client,
    &sinaddr_len)) == -1)
    exit_sys("accept");

printf("Connected: %s : %u\n", inet_ntoa(sinaddr_client.sin_addr),
    (unsigned)ntohs(sinaddr_client.sin_port));

/* other stuff */

return 0;
}

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

    exit(EXIT_FAILURE);
}

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

```

Client program tipik olarak şu aşamalardan geçerek yazılmaktadır:
 socket, bind (optional), gethostbyname (optional), connect,
 read/write/recv/send, shutdown, close.

Client program yine socket fonksiyonuyla bir soket yaratır. İsterse bu soketi bind eder. Client bind işlemi yapmak zorunda değildir.

Ancak client'ın server'a belirli bir kaynak porttan bağlanması isteniyorsa client bu kaynak portu belirlemek için bind işlemi yapmalıdır.

Eğer client bind işlemi yapmazsa işletim sistemi belli bir aralıkta boş bir kaynak port numarasını tahsis eder. Client program server'ın ip adresini ve port numarasını bilerek ona bağlanacaktır. Ancak IP adresleri akılda zor tutulduğu için IP adreslerine isimler karşı düşürülmüştür.

Pratikte daha çok bu isimler kullanılmaktadır. connect fonksiyonun prototipi şöyledir:

```
int connect(int socket, const struct sockaddr *address, socklen_t
address_len);
```

Fonksiyonun birinci parametresi soket betimleyicisi alır. İkinci parametre yine IPV4 için sockaddr_in IPV6 için sockaddr_in6 türünden bir yapının adresini alır. Böylece IPV4 için programcı bir sockaddr_in yapısı alıp bağlanacağı server'ın ip adresini ve port numarasını bu yapıya yerleştirir. Fonksiyonun son parametresi ikinci parametredeki nesnenin byte uzunluğunu almaktadır. Fonksiyon başarı durumunda 0, başarısızlık durumunda -1 değerine geri dönmektedir.

IP adreslerinin sockaddr_in yapısına BIG ENDIAN (network byte ordering) biçiminde girilmesi gerektiğini anımsayınız. Bu durumda örneğin a.b.c.d biçimindeki bir IP adresi htonl(a << 24 | b << 16 | c << 8 | d) biçiminde oluşturulabilir. Ancak bunun yerine bu dönüşümü yapana inet_addr isimli bir fonksiyon bulundurulmuştur. Bu fonksiyon "a.b.c.d" biçiminde yazısal olarak verilen IP adresini parse ederek BIG ENDIAN formata dönüştürmektedir. Eğer noktalıformdaki (dotted decimal form) IP adresi yazılmış girilmişse fonksiyon INADDR_NONE değerine geri dönmektedir.

Bu işlemin tersini yapan inet_ntoa isimli bir fonksiyon vardır. Bu iki fonksiyon IPV4'te kullanılabilir. Daha sonraları IPV6'da da kullanılacak

biçimde yeni fonksiyonlar inet_pton ve inet_aton fonksiyonlarıdır.

Genellikle client programlar server'ın IP adresini ya da host ismini (domain ismini) alarak çalışacak biçimde yazılırlar. İşte programcının girilen değer öncelikle "noktalı desimal formda (dotted decimal form)" olup olmadığını kontrol etmesi gerekir. Zaten inet_addr fonksiyonu bu kontrolü

yapabilmektedir. O halde girilen yazı önce inet_addr fonksiyonuna sokulmalı eğer oradan INADDR_NONE değeri elde ediliyorsa artık girilen yazının ip adresi

olmadığı anlaşılır. İşte host isimlerinin IP adreslerine dönüştürülmesi için IP protokol ailesinde "DNS protokolü" denilen özel bir protokol bulunmaktadır.

İsimlerin IP adres karşılıkları "Domain Name Server" denilen özel server'larda tutulur. Client program bu server'lara danışarak ismi IP numarasına

dönüştürür. Bu işlem için IPV4'te gethostbyname isimli fonksiyon kullanılıyordu. Daha sonra IPV6'yı da kapsayacak biçimde getnameinfo ve getaddrinfo fonksiyonları oluşturdu. gethostbyname fonksiyonun prototipi şöyledir:

```
struct hostent *gethostbyname(const char *name);
```

Fonksiyon parametre olarak host ismini alır, ve hostent isimli bir yapı adresine geri döner. hostent yapısı şöyledir:

```
struct hostent {
    char *h_name;
    char **h_aliases;
    int h_addrtype;
    int h_length;
    char **h_addr_list;
};
```

Bit host ismine karşı birden fazla IP adresi olabileceği gibi, bir IP adresine karşı da birden fazla host ismi olabilmektedir. Yapının h_addr_list elemanı her biri 4 char'dan oluşan dört elemanlı dizilerin adreslerini tutmaktadır. Bunlar host ismine karşı gelen IP adresleridir.

Bu gösterici dizisinin sonunda NULL adres vardır.

Aşağıdaki programda anlatılan yere kadar bir client program iskeleti verilmiştir. Program komut satırı argümanı olarak server'ın ip adresini (ya da host ismini) ve port numarasını alır ve server'a TCP ile bağlanır.

-----*/

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>
```

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

```
int main(int argc, char *argv[])
{
    int sock;
    struct sockaddr_in sinaddr;
    struct hostent *hent;
    in_port_t port;

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

```

port = (in_port_t)strtol(argv[2], NULL, 10);

if ((sock = socket(AF_INET, SOCK_STREAM, 0)) == -1)
    exit_sys("socket");

sinaddr.sin_family = AF_INET;
sinaddr.sin_port = htons(port);

if ((sinaddr.sin_addr.s_addr = inet_addr(argv[1])) == INADDR_NONE) {
    if ((hent = gethostbyname(argv[1])) == NULL)
        exit_sys("gethostbyname");
    memcpy(&sinaddr.sin_addr.s_addr, hent->h_addr_list[0], hent->h_length);
}

if (connect(sock, (struct sockaddr *)&sinaddr, sizeof(sinaddr)) == -1)
    exit_sys("connect");

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

/* other stuff */

return 0;
}

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

    exit(EXIT_FAILURE);
}

/
*-----
-----
Client taraf soketi yarattiktan sonra kaynak port numarasının belli bir
değerde olmasını sağlayabilir. Bunun için client
tarafın da bind işlemi uygulaması gerekir. Aşağıdaki örnekte client
taraf 5051 numaralı port'a bind işlemi yapmıştır.
-----
-----*/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>

void exit_sys(const char *msg);

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

```

```

{
    int sock;
    struct sockaddr_in sinaddr;
    struct hostent *hent;
    in_port_t port;

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

    port = (in_port_t)strtol(argv[2], NULL, 10);

    if ((sock = socket(AF_INET, SOCK_STREAM, 0)) == -1)
        exit_sys("socket");

    sinaddr.sin_family = AF_INET;
    sinaddr.sin_port = htons(5051);
    sinaddr.sin_addr.s_addr = htonl(INADDR_ANY);

    if (bind(sock, (struct sockaddr *)&sinaddr, sizeof(sinaddr)) == -1)
        exit_sys("bind");

    sinaddr.sin_family = AF_INET;
    sinaddr.sin_port = htons(port);

    if ((sinaddr.sin_addr.s_addr = inet_addr(argv[1])) == INADDR_NONE) {
        if ((hent = gethostbyname(argv[1])) == NULL)
            exit_sys("gethostbyname");
        memcpy(&sinaddr.sin_addr.s_addr, hent->h_addr_list[0], hent->h_length);
    }

    if (connect(sock, (struct sockaddr *)&sinaddr, sizeof(sinaddr)) == -1)
        exit_sys("connect");

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

    /* other stuff */

    return 0;
}

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

    exit(EXIT_FAILURE);
}

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

```

Bağlantı sağlandıktan sonra artık iki taraf da birbirlerine bilgi gönderip alabilirler (full duplex). Bilgi gönderip alma aslında bir tamponlama (buffering) mekanizmasıyla gerçekleştirilmektedir. Her iki tarafta da işletim sistemi "gönderme tamponu (send buffer)" ve "alma tamponu (receive buffer)" isminde iki tampon bulundurur. Biz bir soketle karşı tarafa bilgi göndermek istediğimizde aslında göndermek istediğimiz bilgiler önce kendi bilgisayarımızın gönderme taponuna (send buffer) yazılmaktadır. Bu gönderme tamponuna yazılan bilgiler işletim sistemi tarafından TCP ve dolayısıyla IP paketlerine dönüştürülüp uygun bir zamanda gerçekten gönderilmektedir. Benzer biçimde aslında bilgisayarımıza gelen paketler kesme (interrupt) mekanizması yoluyla işletim sistemi tarafından alma tamponuna yerleştirilmektedir. Biz soketten okuma yapmak istediğimizde bu tampona yerleştirilmiş olanları okuruz. Pekiyi alma tamponu dolarsa ve biz hiç okuma yapmazsak ne olur? İşte TCP protokolü (ama IP değil) akış kontrolüne (flow control) sahiptir. Akış kontrolü tampon taşmasını engellemek için iki tarafın konuşarak birbirlerini gerektiğinde durdurması anlamına gelmektedir.

TCP'de soket arayüzünü kullanarak bilgi göndermek için write ya da send fonksiyonları kullanılmaktadır. send fonksiyonunun write fonksiyonundan fazla bir flags parametresi vardır.

```
ssize_t write(int fd, const char *buf, size_t size);
ssize_t send(int fd, const void *buffer, size_t sşze, int flags);
```

Soketten bilgi okumak için ise read ya da recv fonksiyonları kullanılmaktadır. Yine aslında recv fonksiyonu read fonksiyonundan farklı olarak bir flags parametresine sahiptir:

```
ssize_t read(int fd, void *buf, size_t count);
ssize_t recv(int fd, void *buf, size_t len, int flags);
```

send fonksiyonunun flags parametresi 0 geçilirse, recv fonksiyonunun da flags parametresi 0 geçilirse bunların write ve read'ten hiçbir farklılığı kalmamaktadır.

write ya da send fonksiyonu ile bilgi gönderilirken aslında bilgi "gönderme tamponuna yazılıp" hemen bu fonksiyonlar geri dönerler. write ya da send

fonksiyonları POSIX standartlarına göre normal olarak tüm bilgi gönderme tamponuna yazılana kadar blokeye yol açmaktadır. Örneğin biz bu fonksiyonlarla

100 byte göndermek isteyelim ancak gönderme tamponunda 90 byte'lık yer kalmış olsun. Bu durumda write ya da send fonksiyonları bu 100 byte'ın tamamı yazılana

kadar blokeye yol açmaktadır. (send fonksiyonunun bu davranışı Windows sistemlerinde değişiklik gösterebilmektedir. Bu sistemlerde send fonksiyonu bilginin tamamı tampona aktarılanaya kadar blokeye yol açmak zorunda değildir. Bu sistemlerde send fonksiyonu yazabildiği kadar byte'ı tampona

yazıp yazabildiği byte sayısına geri dönebilmektedir.)

read ya da recv fonksiyonları "alma tamponuna" bakmaktadır. Bu tampon tamamen boş ise bunlar blokeli modda (default durum) en az 1 byte okuyana kadar beklerler. Ancak eğer tamponda en az 1 byte'lık bilgi varsa bu fonksiyonlar blokeye yol açmazlar okuyabildikleri kadar byte'ı okuyarak okuyabildikleri byte sayısına geri dönerler.

TCP/IP haberleşmede önemli bir durum vardır: Bir tarafın tek bir write/send ile gönderdiği bilgiyi diğer taraf tek bir read/recv ile okuyamayabilir.

Çünkü write/send yapan taraf bu bilgileri önce gönderme tamponuna yerleştirir. Buradaki bilgiler işletim sistemi tarafından farklı IP paketleri ile iletilebilir. Bunun sonucunda alıcı taraf bunları parça parça alabilir.

```
-----*/  
  
/* server.c */  
  
#include <stdio.h>  
#include <stdlib.h>  
#include <string.h>  
#include <unistd.h>  
#include <sys/socket.h>  
#include <netinet/in.h>  
#include <arpa/inet.h>  
  
#define BUFFER_SIZE      1024  
  
char *revstr(char *str);  
void exit_sys(const char *msg);  
  
int main(int argc, char *argv[])  
{  
    int sock, sock_client;  
    struct sockaddr_in sinaddr, sinaddr_client;  
    socklen_t sinaddr_len;  
    in_port_t port;  
    ssize_t result;  
    char buf[BUFFER_SIZE + 1];  
  
    if (argc != 2) {  
        fprintf(stderr, "wrong number of arguments!..\n");  
        exit(EXIT_FAILURE);  
    }  
  
    port = (in_port_t)strtoul(argv[1], NULL, 10);  
  
    if ((sock = socket(AF_INET, SOCK_STREAM, 0)) == -1)  
        exit_sys("socket");  
  
    sinaddr.sin_family = AF_INET;
```

```

sinaddr.sin_port = htons(port);
sinaddr.sin_addr.s_addr = htonl(INADDR_ANY);

if (bind(sock, (struct sockaddr *)&sinaddr, sizeof(sinaddr)) == -1)
    exit_sys("bind");

if (listen(sock, 8) == -1)
    exit_sys("listen");

printf("Waiting for connection...\n");

sinaddr_len = sizeof(sinaddr_client);
if ((sock_client = accept(sock, (struct sockaddr *)&sinaddr_client,
    &sinaddr_len)) == -1)
    exit_sys("accept");

printf("Connected: %s : %u\n", inet_ntoa(sinaddr_client.sin_addr),
    (unsigned)ntohs(sinaddr_client.sin_port));

for (;;) {
    if ((result = recv(sock_client, buf, BUFFER_SIZE, 0)) == -1)
        exit_sys("recv");
    if (result == 0)
        break;
    buf[result] = '\0';
    if (!strcmp(buf, "quit"))
        break;
    printf("%ld bytes received from %s (%u): %s\n", (long)result,
        inet_ntoa(sinaddr_client.sin_addr),
        (unsigned)ntohs(sinaddr_client.sin_port), buf);
    revstr(buf);
    if (send(sock_client, buf, strlen(buf), 0) == -1)
        exit_sys("send");
}

/* Other stuff */

return 0;
}

char *revstr(char *str)
{
    size_t i, k;
    char temp;

    for (i = 0; str[i] != '\0'; ++i)
        ;

    for (--i, k = 0; k < i; ++k, --i) {
        temp = str[k];
        str[k] = str[i];
        str[i] = temp;
    }

    return str;
}

```

```

}

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

    exit(EXIT_FAILURE);
}

/* client.c */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>

#define BUFFER_SIZE      1024

void exit_sys(const char *msg);

int main(int argc, char *argv[])
{
    int sock;
    struct sockaddr_in sinaddr;
    struct hostent *hent;
    in_port_t port;
    char buf[BUFFER_SIZE];
    char *str;
    ssize_t result;

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

    port = (in_port_t)strtol(argv[2], NULL, 10);

    if ((sock = socket(AF_INET, SOCK_STREAM, 0)) == -1)
        exit_sys("socket");

    sinaddr.sin_family = AF_INET;
    sinaddr.sin_port = htons(port);

    if ((sinaddr.sin_addr.s_addr = inet_addr(argv[1])) == INADDR_NONE) {
        if ((hent = gethostbyname(argv[1])) == NULL)
            exit_sys("gethostbyname");
        memcpy(&sinaddr.sin_addr.s_addr, hent->h_addr_list[0], hent->h_length);
    }

    if (connect(sock, (struct sockaddr *)&sinaddr, sizeof(sinaddr)) == -1)

```

```

        exit_sys("connect");

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

for (;;) {
    printf("Yazı giriniz:");
    fgets(buf, BUFFER_SIZE, stdin);
    if ((str = strchr(buf, '\n')) != NULL)
        *str = '\0';
    if ((send(sock, buf, strlen(buf), 0)) == -1)
        exit_sys("send");
    if (!strcmp(buf, "quit"))
        break;

    if ((result = recv(sock, buf, BUFFER_SIZE, 0)) == -1)
        exit_sys("recv");
    if (result == 0)
        break;
    buf[result] = '\0';
    printf("%ld bytes received from %s (%u): %s\n", (long)result,
        inet_ntoa(sinaddr.sin_addr),
            (unsigned)ntohs(sinaddr.sin_port), buf);
}

/* Other stuff */

return 0;
}

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

    exit(EXIT_FAILURE);
}

```

Haberleşmenin sonunda TCP soketi nasıl kapatılmalıdır? Mademli soketler UNIX/Linux sistemlerinde birer dosya betimleyicisi gibidir o halde soketi kapatma işlemi close ile yapılmaktadır. Tabii yine close işlemi yapılmazsa işletim sistemi proses normal ya da sinyal gibi nedenlerle sonlandığında otomatik close işleminş yapar. Soket betimleyicileri de dup işlemine sokulabilir. Bu durumda close işlemi soket nesnesinin yok edileceği anlamına gelmez. Benzer biçimde fork işlemi sırasında da betimleyicilerin çoğtlendiğine dikkat ediniz.

Aktif soketlerin doğrudan close ile kapatılması iyi bir teknik değildir. Bu soketler önce shutdown ile haberleşmeden kesilmeli sonra close işlemi uygulanmalıdır. Bu biçimde soketlerin kapatılmasına İngilizce "graceful close (zarif kapatma)" denilmektedir. Pekiyi shutdown

fonksiyonu ne yapmaktadır ve neden gerekmektedir? close işlemi ile bir soket kapatıldığında işletim sistemi sokete ilişkin tüm veri yapılarını ve bağlantı bilgilerini siler. Örneğin biz karşı tarafa send ile bir şey gönderdikten hemen sonraki satırda close yaparsak artık send ile gönderdiklerimizin karşı tarafa ulaşacağını hiçbir garantisi yoktur. Çünkü anımsanacağı gibi send aslında "gönderme tamponuna" bilgiyi yazıp geri dönmektedir. Hemen arkasından close işlemi uygulandığında artık bu sokete ilişkin gönderme ve alma tamponları da yok edileceğinden tamponda gönderilmeyi bekleyen bilgiler hiç gönderilmeyebilecektir. İşte shutdown fonksiyonun üç işlevi vardır:

- 1) Haberleşmeyi TCP çerçevesinde el sıkışarak sonlandırmak.
- 2) Gönderme tamponuna yazılan bilgilerin gönderildiğine emin olmak.
- 3) Okuma ya da yazma işlemini sonlandırıp diğer işleme devam edebilmek.

shutdown fonksiyonun prototipi şöyledir:

```
int shutdown(int socket, int how);
```

Fonksiyonun birinci parametresi sonlandırılacak soketin betimleyicisini, ikinci parametresi biçimini belirtmektedir İkinci parametre şunlardan biri olarak girilebilir:

SHUT_RD: Bu işlemden sonra artık soketten okuma yapılamaz. Fakat sokete yazma yapılabilir. Bu seçenek pek kullanılmamaktadır.

SHUT_WR: Burada artık shutdown daha önce gönderme tamponuna yazılmış olan byte'ların gönderilmesine kadar bloke oluşturabilir.

Bu işlemden sonra artık sokete yazma yapılamaz ancak okuma işlemi devam ettirilebilir.

SHUT_RDWR: En çok kullanılan seçenektir. Burada da artık shutdown daha önce gönderme tamponuna yazılmış olan byte'ların gönderilmesine kadar bloke oluşturabilir. Artık bundan sonra soketten okuma ya da yazma yapılamamaktadır. shutdown başarı durumunda 0 değerine, başarısızlık durumunda -1 değerine geri dönmektedir.

O halde aktif bir soketin kapatılması tipik olarak şöyle yapılmaktadır:

```
shutdown(sock, SHUT_RDWR);  
close(sock);
```

Karşı taraf (peer) soketi shutdown ile SHUT_WR ya da SHUT_RDWR ile sonlandırmışsa artık biz o soketten okuma yaptığımızda read ya da recv fonksiyonları 0 ile geri döner. Benzer biçimde karşı taraf doğrudan soketi close ile kapatmışsa yine biz recv işleminden 0 elde ederiz.

Karşı tarafın soketi kapatıp kapatmadığı tipik olarak recv fonksiyonunda anlaşılabilmektedir. Ancak karşı taraf soketi kapattıktan sonra biz sokete

write ya da send ile birşeyler yazmak istersek default durumda UNIX/Linux sistemlerinde SIGPIPE sinyali oluşmaktadır. Programcı send fonksiyonun

flags parametresine MSG_NOSIGNAL değerini girerse bu durumda send başarısız olmakta ve errno EPIPE değeri ile set edilmektedir.

Karşı taraf soketi kapatmamış ancak bağlantı kopmuş olabilir. Bu durumda send ve recv fonksiyonları -1 ile geri döner.

Yukarıdaki programlara zarif kapatma özelliğini de ekleyebiliriz.

```
-----*/  
  
/* server.c */  
  
#include <stdio.h>  
#include <stdlib.h>  
#include <string.h>  
#include <unistd.h>  
#include <sys/socket.h>  
#include <netinet/in.h>  
#include <arpa/inet.h>  
  
#define BUFFER_SIZE      1024  
  
char *revstr(char *str);  
void exit_sys(const char *msg);  
  
int main(int argc, char *argv[])  
{  
    int sock, sock_client;  
    struct sockaddr_in sinaddr, sinaddr_client;  
    socklen_t sinaddr_len;  
    in_port_t port;  
    ssize_t result;  
    char buf[BUFFER_SIZE + 1];  
  
    if (argc != 2) {  
        fprintf(stderr, "wrong number of arguments!..\n");  
        exit(EXIT_FAILURE);  
    }  
  
    port = (in_port_t)strtoul(argv[1], NULL, 10);  
  
    if ((sock = socket(AF_INET, SOCK_STREAM, 0)) == -1)  
        exit_sys("socket");  
  
    sinaddr.sin_family = AF_INET;  
    sinaddr.sin_port = htons(port);  
    sinaddr.sin_addr.s_addr = htonl(INADDR_ANY);  
  
    if (bind(sock, (struct sockaddr *)&sinaddr, sizeof(sinaddr)) == -1)  
        exit_sys("bind");  
  
    if (listen(sock, 8) == -1)  
        exit_sys("listen");  
  
    printf("Waiting for connection...\n");  
  
    sinaddr_len = sizeof(sinaddr_client);  
    if ((sock_client = accept(sock, (struct sockaddr *)&sinaddr_client,  
        &sinaddr_len)) == -1)  
        exit_sys("accept");
```

```

printf("Connected: %s : %u\n", inet_ntoa(sinaddr_client.sin_addr),
      (unsigned)ntohs(sinaddr_client.sin_port));

for (;;) {
    if ((result = recv(sock_client, buf, BUFFER_SIZE, 0)) == -1)
        exit_sys("recv");
    if (result == 0)
        break;
    buf[result] = '\0';
    if (!strcmp(buf, "quit"))
        break;
    printf("%ld bytes received from %s (%u): %s\n", (long)result,
          inet_ntoa(sinaddr_client.sin_addr),
          (unsigned)ntohs(sinaddr_client.sin_port), buf);
    revstr(buf);
    if (send(sock_client, buf, strlen(buf), 0) == -1)
        exit_sys("send");
}

shutdown(sock_client, SHUT_RDWR);
close(sock_client);
close(sock);

return 0;
}

char *revstr(char *str)
{
    size_t i, k;
    char temp;

    for (i = 0; str[i] != '\0'; ++i)
        ;

    for (--i, k = 0; k < i; ++k, --i) {
        temp = str[k];
        str[k] = str[i];
        str[i] = temp;
    }

    return str;
}

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

    exit(EXIT_FAILURE);
}

/* client.c */

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

```

```

#include <string.h>
#include <unistd.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>

#define BUFFER_SIZE      1024

void exit_sys(const char *msg);

int main(int argc, char *argv[])
{
    int sock;
    struct sockaddr_in sinaddr;
    struct hostent *hent;
    in_port_t port;
    char buf[BUFFER_SIZE];
    char *str;
    ssize_t result;

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

    port = (in_port_t)strtol(argv[2], NULL, 10);

    if ((sock = socket(AF_INET, SOCK_STREAM, 0)) == -1)
        exit_sys("socket");

    sinaddr.sin_family = AF_INET;
    sinaddr.sin_port = htons(port);

    if ((sinaddr.sin_addr.s_addr = inet_addr(argv[1])) == INADDR_NONE) {
        if ((hent = gethostbyname(argv[1])) == NULL)
            exit_sys("gethostbyname");
        memcpy(&sinaddr.sin_addr.s_addr, hent->h_addr_list[0], hent->h_length);
    }

    if (connect(sock, (struct sockaddr *)&sinaddr, sizeof(sinaddr)) == -1)
        exit_sys("connect");

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

    for (;;) {
        printf("Yazı giriniz:");
        fgets(buf, BUFFER_SIZE, stdin);
        if ((str = strchr(buf, '\n')) != NULL)
            *str = '\0';
        if ((send(sock, buf, strlen(buf), 0)) == -1)
            exit_sys("send");
        if (!strcmp(buf, "quit"))
            break;
    }
}

```



```

    if ((result = recv(sock, buf, BUFFER_SIZE, 0)) == -1)
        exit_sys("recv");
    if (result == 0)
        break;
    buf[result] = '\0';
    printf("%ld bytes received from %s (%u): %s\n", (long)result,
        inet_ntoa(sinaddr.sin_addr),
            (unsigned)ntohs(sinaddr.sin_port), buf);
}

shutdown(sock, SHUT_RDWR);
close(sock);

return 0;
}

```

```

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

    exit(EXIT_FAILURE);
}

```

/

*-----

Windows sistemlerindeki soket kütüphanesine "Winsock" denilmektedir. Şu anda bu kütüphanenin 2'inci versiyonu kullanılmaktadır. Winsosk fonksiyonları "UNIX/Linux uyumlu" fonksiyonlar ve Windows'a özgü fonksiyonlar olmak üzere iki biçimde kullanılabilir. Ancak Winsock'un UNIX/Linux uyumlu fonksiyonlarında da birtakım değişiklikler söz konusudur. Bir UNIX/Linux ortamında yazılmış soket uygulamasının Windows sistemlerine aktarılması için şu düzeltmelerin yapılması gerekir:

- 1) POSIX'in soket sistemine ilişkin tüm başlık dosyaları kaldırılır. Onun yerine <winsok2.h> dosyası include edilir.
- 2) xxx_t'li typedef türleri silinir ve onların yerine (dokümanlara da bakabilirsiniz) int, short, unsigned int, unsigned short türleri kullanılır.
- 3) Windows'ta soket sisteminin ilklendirilmesi için WSASStartup fonksiyonu işin başında çağrılır ve işin sonunda da bu işlem WSACleanup fonksiyonuyla geri alınır.
- 4) Windows'ta dosya betimleyicisi kavramı yoktur. (Onun yerine "handle" kavramı vardır.) Dolayısıyla soket türü de int değil, SOCKET isimli bir typedef türüdür.
- 5) shutdown fonksiyonun ikinci parametresi SD_RECEIVE, SD_SEND ve SD_BOTH biçimindedir.
- 6) close fonksiyonu yerine closesocket ile soket kapatılır.
- 7) Windows'ta soket fonksiyonları başarısızlık durumunda -1 değerine geri dönmeyiz. socket fonksiyonu başarısızlık durumunda INVALID_SOCKET değerine, diğerleri ise SOCKET_ERROR değerine geri dönmektedir.

- 8) Visual Studiko IDE'sinde default durumda "deprecated" durumlar "error"e yükseltilmiştir. Bunlar için bir sembolik sabit define edilebilmektedir. Ancak proje ayarlarından "sdl check" disable edilebilir. Benzer biçimde proje ayarlarından "Unicode" "not set" yapılmalıdır.
- 9) Projenin linker ayarlarından Input/Additional Dependencies edit alanına Winsock kütüphanesi olan "Ws2_32.lib" import kütüphanesi eklenir.

Yukarıdaki programların Winsock'a dönüştürülmüş biçimleri şöyledir:

```
-----*/  
  
/* server.c */  
  
#include <stdio.h>  
#include <stdlib.h>  
#include <string.h>  
#include <WinSock2.h>  
  
#define BUFFER_SIZE      1024  
  
char* revstr(char* str);  
void exit_sys(LPCSTR lpszMsg, int status, DWORD dwLastError);  
  
int main(int argc, char* argv[])  
{  
    SOCKET sock, sock_client;  
    struct sockaddr_in sinaddr, sinaddr_client;  
    int sinaddr_len;  
    unsigned short port;  
    int result;  
    char buf[BUFFER_SIZE + 1];  
    WSADATA wsadata;  
  
    if ((result = WSStartup(MAKEWORD(2, 2), &wsadata)) != 0)  
        exit_sys("WSStartup", EXIT_FAILURE, result);  
  
    if (argc != 2) {  
        fprintf(stderr, "wrong number of arguments!..\n");  
        exit(EXIT_FAILURE);  
    }  
  
    port = (unsigned short)strtoul(argv[1], NULL, 10);  
  
    if ((sock = socket(AF_INET, SOCK_STREAM, 0)) == INVALID_SOCKET)  
        exit_sys("socket", EXIT_FAILURE, WSAGetLastError());  
  
    sinaddr.sin_family = AF_INET;  
    sinaddr.sin_port = htons(port);  
    sinaddr.sin_addr.s_addr = htonl(INADDR_ANY);  
  
    if (bind(sock, (struct sockaddr*)&sinaddr, sizeof(sinaddr)) ==  
        SOCKET_ERROR)
```

```

    exit_sys("bind", EXIT_FAILURE, WSAGetLastError());

if (listen(sock, 8) == -1)
    exit_sys("listen", EXIT_FAILURE, WSAGetLastError());

printf("Waiting for connection...\n");

sinaddr_len = sizeof(sinaddr_client);
if ((sock_client = accept(sock, (struct sockaddr*)&sinaddr_client,
    &sinaddr_len)) == SOCKET_ERROR)
    exit_sys("accept", EXIT_FAILURE, WSAGetLastError());

printf("Connected: %s : %u\n", inet_ntoa(sinaddr_client.sin_addr),
    (unsigned)ntohs(sinaddr_client.sin_port));

for (;;) {
    if ((result = recv(sock_client, buf, BUFFER_SIZE, 0)) == -1)
        exit_sys("recv", EXIT_FAILURE, WSAGetLastError());
    if (result == 0)
        break;
    buf[result] = '\0';
    if (!strcmp(buf, "quit"))
        break;
    printf("%ld bytes received from %s (%u): %s\n", (long)result,
        inet_ntoa(sinaddr_client.sin_addr),
        (unsigned)ntohs(sinaddr_client.sin_port), buf);
    revstr(buf);
    if (send(sock_client, buf, strlen(buf), 0) == -1)
        exit_sys("send", EXIT_FAILURE, WSAGetLastError());
}

shutdown(sock_client, SD_BOTH);
closesocket(sock_client);
closesocket(sock);

WSACleanup();

return 0;
}

char *revstr(char* str)
{
    size_t i, k;
    char temp;

    for (i = 0; str[i] != '\0'; ++i)
        ;

    for (--i, k = 0; k < i; ++k, --i) {
        temp = str[k];
        str[k] = str[i];
        str[i] = temp;
    }

    return str;
}

```

```

}

void exit_sys(LPCSTR lpszMsg, int status, DWORD dwLastError)
{
    LPTSTR lpszErr;

    if (FormatMessage(FORMAT_MESSAGE_ALLOCATE_BUFFER |
        FORMAT_MESSAGE_FROM_SYSTEM, NULL, dwLastError,
        MAKELANGID(LANG_NEUTRAL, SUBLANG_DEFAULT), (LPTSTR)&lpszErr, 0,
        NULL)) {
        fprintf(stderr, "%s: %s", lpszMsg, lpszErr);
        LocalFree(lpszErr);
    }

    exit(status);
}

```

/* client.c */

```

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <WinSock2.h>

#define BUFFER_SIZE      1024

void exit_sys(LPCSTR lpszMsg, int status, DWORD dwLastError);

int main(int argc, char* argv[])
{
    SOCKET sock;
    WSADATA wsadata;
    struct sockaddr_in sinaddr;
    struct hostent* hent;
    unsigned short port;
    char buf[BUFFER_SIZE];
    char *str;
    int result;

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

    port = (unsigned short)strtol(argv[2], NULL, 10);

    if ((result = WSStartup(MAKEWORD(2, 2), &wsadata)) != 0)
        exit_sys("WSStartup", EXIT_FAILURE, result);

    if ((sock = socket(AF_INET, SOCK_STREAM, 0)) == INVALID_SOCKET)
        exit_sys("socket", EXIT_FAILURE, WSAGetLastError());

    sinaddr.sin_family = AF_INET;
    sinaddr.sin_port = htons(port);
}

```

```

if ((sinaddr.sin_addr.s_addr = inet_addr(argv[1])) == INADDR_NONE) {
    if ((hent = gethostbyname(argv[1])) == NULL)
        exit_sys("gethostbyname", EXIT_FAILURE, WSAGetLastError());
    memcpy(&sinaddr.sin_addr.s_addr, hent->h_addr_list[0], hent->h_length);
}

if (connect(sock, (struct sockaddr*)&sinaddr, sizeof(sinaddr)) ==
    SOCKET_ERROR)
    exit_sys("connect", EXIT_FAILURE, WSAGetLastError());

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

for (;;) {
    printf("Yazı giriniz:");
    fgets(buf, BUFFER_SIZE, stdin);
    if ((str = strchr(buf, '\n')) != NULL)
        *str = '\0';
    if ((send(sock, buf, strlen(buf), 0)) == SOCKET_ERROR)
        exit_sys("send", EXIT_FAILURE, WSAGetLastError());
    if (!strcmp(buf, "quit"))
        break;

    if ((result = recv(sock, buf, BUFFER_SIZE, 0)) == SOCKET_ERROR)
        exit_sys("recv", EXIT_FAILURE, WSAGetLastError());
    if (result == 0)
        break;
    buf[result] = '\0';
    printf("%ld bytes received from %s (%u): %s\n", (long)result,
        inet_ntoa(sinaddr.sin_addr),
        (unsigned)ntohs(sinaddr.sin_port), buf);
}

shutdown(sock, SD_BOTH);
closesocket(sock);

WSACleanup();

return 0;
}

void exit_sys(LPCSTR lpszMsg, int status, DWORD dwLastError)
{
    LPTSTR lpszErr;

    if (FormatMessage(FORMAT_MESSAGE_ALLOCATE_BUFFER |
        FORMAT_MESSAGE_FROM_SYSTEM, NULL, dwLastError,
        MAKELANGID(LANG_NEUTRAL, SUBLANG_DEFAULT), (LPTSTR)&lpszErr, 0,
        NULL)) {
        fprintf(stderr, "%s: %s", lpszMsg, lpszErr);
        LocalFree(lpszErr);
    }

    exit(status);
}

```

/

*-----

Daha önce write/send ve read/recv fonksiyonlarının davranışları hakkında temel bazı şeyler söylemiştik. Şimdi biraz ayrıntılandıralım. POSIX standartlarına göre send fonksiyonu blokeli modda bloke garantisi vermektedir. Yani örneğin gönderme tamponunda 10 byte'lık boş bir alan varsa fakat biz write/send ile 100 byte göndermeye çalışıyorsak blokeli modda send bu 100 byte'ın tamamını tampona yazana kadar bloke bekler. Böylece send genel olarak bizim yazmak istediğimiz byte miktarına geri dönmektedir. Ancak bu davranış eskiden tam böyle değildi.

Windows sistemlerinde de tam böyle değildir. Windows sistemlerinde send fonksiyonu eğer gönderme tamponu tüm gönderilecek bilgiyi alamayacak biçimde doluysa send blokeye yol açmayabilmektedir. Böylece Windows sistemlerinde send talep edilenden daha düşük bir değerle geri dönebilmektedir. Blokesin modda POSIX standartlarına göre send fonksiyonu yine kısmi yazıma izin vermemektedir. Örneğin gönderme tamponunda yine 10 byte boşluk olsun. Biz de blokesiz modda 100 byte göndermek isteyelim. Bu durumda send tüm bilgileri tampona yazamayacaksa hiçbirini yazmaz ve -1 ile geri döner. errno EAGAIN değeri ile set edilir.

read/recv fonksiyonları daha önceden belirtildiği gibi blokeli modda eğer alma tamponu tamamen boş ise blokeye yol açmaktadır. Alma tamponunda en az 1 byte varsa read/recv bloke oluşturmaz olanı okur ve geri döner. Blokesiz modda write/recv tampon boşsa bile blokeye yol açmamaktadır. Bu durumda bu fonksiyonlar -1 ile geri dönerler ve errno EAGAIN değeriyle set edilir.

Yukarıda da belirtildiği gibi karşı taraf shutdown uygulamışsa ya da soketi kapatmışsa read/recv 0 ile geri döner. Başka diğer hatalarda (örneğin bağlantı kopması) bu fonksiyonlar başarısız olup -1 ile geri dönerler.

-----*/

/

*-----

Zamanla bazı klasik soket fonksiyonları yerine onların işlevini yapabilecek daha yetenekli fonksiyonlar oluşturulmuştur. Eski fonksiyonlar IPV4 zamanlarında tasarlanmıştı. IPV6 için bunların bazılarının farklı isimli biçimleri devreye sokuldu. En sonunda hem IPV4'te hem de IPV6 da çalışacak daha yetenekli fonksiyonlar tasarlandı. Programlarınızda bunları kullanmanız artık daha uygun olabilir.

inet_nto fonksiyonu bilindiği gibi noktalı desimal formattaki IP adresini 4 byte'lık IPV4 formuna dönüştürüyordu. İşte bu fonksiyonun inet_ptoa isimli IPV6'yı da kapsayan gelişmiş biçimi vardır:

```
const char *inet_ntop(int af, const void *src, char *dst, socklen_t size);
```

Fonksiyonun birinci parametresi AF_INET (IPv4) ya da AF_INET6 (IPv6) olarak girilmelidir. İkinci parametre nümerik IPv4 ya da IPv6 adresinin bulunduğu adresi belirtmektedir. Fonksiyon yazısal noktalı desimal formatı üçüncü parametresiyle belirtilen adrese yerleştirir. Son parametre üçüncü parametredeki dizinin uzunluğunu belirtir. Bu parametre INET_ADDRSTRLEN ya da INET6_ADDRSTRLEN biçiminde girilebilir. Fonksiyon başarı durumunda üçüncü parametreyle belirtilen adrese başarısızlık durumunda NULL adrese geri döner.Örneğin:

```
char ntopbuf[INET_ADDRSTRLEN];
...
printf("Connected: %s : %u\n", inet_ntop(AF_INET,
&sinaddr_client.sin_addr, ntopbuf, INET_ADDRSTRLEN),
(unsigned)ntohs(sinaddr_client.sin_port));
```

inet_ntoa işleminin tersinin inet_addr ile yapıldığını belirtmiştik. İşte inet_addr fonksiyonunun yerine hem IPv4 hem de IPv6 ile çalışan inet_ptoa fonksiyonu kullanılabilmektedir:

```
int inet_pton(int af, const char *src, void *restrict dst);
```

Fonksiyonun birinci parametresi yine AF_INET ya da AF_INET6 biçiminde geçilir. İkinci parametre noktalı desimal formatın bulunduğu yazının adresini üçüncü parametre ise nümerik adresin yerleştirileceği adrsi almaktadır. Fonksiyon başarı durumunda 1 değerine, başarısızlık durumundurumunda 0 ya da -1 değerine geri döner. Başarısızlık birinci parametreden kaynaklanıyorsa -1, ikinci parametreden yanaklanıyorsa 0) Bu durumda örneğin client programda inet_addr yerine inet_ptoa fonksiyonunu şöyle çağırabilirdik:

```
if (inet_pton(AF_INET, argv[1], &sinaddr.sin_addr.s_addr) == 0) {
    ...
}
```

IPv6 ile birlikte yeni gelen diğer bir fonksiyon da getaddrinfo isimli fonksiyondur. Bu fonksiyonaslında inet_addr ve gethostbyname fonksiyonlarının IPv6'yı da içerecek biçimde genişletilmiş bir biçimidir. Yani getaddrinfo hem noktalı desimal formatı nümerik adrese dönüştürür hem de eğer isim söz konusuysa DNS işlemi yaparak ilgili host'un IP adresini elde eder. Maalesef fonksiyon biraz karışık tasarlanmıştır. Fonksiyonun prototipi şöyledir:

```
int getaddrinfo(const char *node, const char *service, const struct
addrinfo *hints, struct addrinfo **res);
```

Fonksiyonun birinci parametres, "noktalı desimal formatlı IP adresi" ya da "host ismi"dir. İkinci parametre NULL geçilebilir ya da buraya port numarası girilebilir. Ancak bu port numarası girilecekse yazısal girilmelidir. Fonksiyon kendisi bu port numarasını htons yaparak Big Endian formata dönüştürüp bize verecektir. Bu parametre IP ailesinin uygulama katmanına ilişkin spesifik bir protokol ismi olarak da girilebilmektedir (Örneğin "http" gibi). Eğer bu parametre NULL girilirse bize port olarak 0 verilir. Fonksiyonun üçüncü parametresi nasıl bir adres istediğimizi anlatan filtreleme seçeneklerini belirtir. Bu parametre addrinfo isimli bir yapı türündendir.

Bu yapının yalnızca ilk dört elemanı programcı tarafından girilebilmektedir. Ancak POSIX standartları bu yapının elemanlarının sıfırlanmasını öngörmektedir (sıfırlanmak normal tırlar için bit 0, göstericiler için NULL adres). Yapı şöyledir:

```
struct addrinfo {
    int             ai_flags;
    int             ai_family;
    int             ai_socktype;
    int             ai_protocol;
    socklen_t       ai_addrlen;
    struct sockaddr *ai_addr;
    char            *ai_canonname;
    struct addrinfo *ai_next;
};
```

Yapının ai_flags elemanı pek çok bayrak değeri alabilmektedir. Bu değer 0 olarak da geçilebilir. Yapının ai_family elemanı AF_INET girilirse host'a ilişkin IPV4 adresleri, AF_INET6 girilirse IPV6 adresleri AF_UNSPEC girilirse hem IPV4 hem de IPV6 adresleri elde edilir. Yapının ai_socktype elemanı 0 girilebilir ya da SOCK_STREAM veya SOCK_DGRAM girilebilir. Fonksiyonun ayrıntılı açıklaması için dokümanlara başvurunuz.

getaddrinfo fonksiyonun son parametresine bir bağlı listenin ilk elemanını gösteren adres yerleştirilmektedir. Buradaki bağlı listenin baş elemanı struct addrinfo yapısının ai_next elemanıdır. Bu bağlı listenin boşaltımı freeaddrinfo fonksiyonu tarafından yapılmaktadır.

getaddrinfo fonksiyonu başarı durumunda 0 değerine başarısızlık durumunda error koduna geri döner. Bu error kodları klasik errno değerlerinden farklı olduğu için strerror fonksiyonuyla değil gai_strerror fonksiyonuyla yazıya dönüştürülmelidir.

getaddrinfo fonksiyonun tersini yapan getnameinfo isminde bir fonksiyon da sonraları socket kütüphanesine eklenmiştir. getnameinfo aslında inet_ntop, getserbyname (biz görmekdik) fonksiyonlarının bileşimi gibidir. Biz aşağıdaki örnekte bu fonksiyonu kullanmayacağız.


```
int getnameinfo(const struct sockaddr *addr, socklen_t addrlen, char
    *host, socklen_t hostlen, char *serv, socklen_t servlen, int flags);
```

Fonksiyonun birinci parametresi `sockaddr_in` ya da `sockaddr_in6` yapısıdır. İkinci parametre birinci parametredeki yapının uzunluğudur. Fonksiyonun sonraki dört parametresi sırasıyla noktalı hostun yazısal temsilin yerleştirileceği dizinin adresi ve uzunluğu, port numarasına ilişkin yazının (servis ismi) yerleştirileceği dizinin adresi ve uzunluğudur. Son parametre 0 geçilebilir. Maksimum host ismi `NI_MAXHOST` ile maksimum servis ismi ise `NI_MAXSERV` ile belirtilmiştir.

Aşağıda daha önce yazmış olduğumuz server'ın bu yeni fonksiyonlarla yazılmış halini görüyorsunuz.

```
-----*/

/* server.c */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

#define BUFFER_SIZE      1024

char *revstr(char *str);
void exit_sys(const char *msg);

int main(int argc, char *argv[])
{
    int sock, sock_client;
    struct sockaddr_in sinaddr, sinaddr_client;
    socklen_t sinaddr_len;
    char ntopbuf[INET_ADDRSTRLEN];
    in_port_t port;
    ssize_t result;
    char buf[BUFFER_SIZE + 1];

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

    port = (in_port_t)strtoul(argv[1], NULL, 10);

    if ((sock = socket(AF_INET, SOCK_STREAM, 0)) == -1)
        exit_sys("socket");

    sinaddr.sin_family = AF_INET;
    sinaddr.sin_port = htons(port);
    sinaddr.sin_addr.s_addr = htonl(INADDR_ANY);
```

```

if (bind(sock, (struct sockaddr *)&sinaddr, sizeof(sinaddr)) == -1)
    exit_sys("bind");

if (listen(sock, 8) == -1)
    exit_sys("listen");

printf("Waiting for connection...\n");

sinaddr_len = sizeof(sinaddr_client);
if ((sock_client = accept(sock, (struct sockaddr *)&sinaddr_client,
    &sinaddr_len)) == -1)
    exit_sys("accept");

printf("Connected: %s : %u\n", inet_ntop(AF_INET,
    &sinaddr_client.sin_addr, ntopbuf, INET_ADDRSTRLEN),
    (unsigned)ntohs(sinaddr_client.sin_port));

for (;;) {
    if ((result = recv(sock_client, buf, BUFFER_SIZE, 0)) == -1)
        exit_sys("recv");
    if (result == 0)
        break;
    buf[result] = '\0';
    if (!strcmp(buf, "quit"))
        break;
    printf("%ld bytes received from %s (%u): %s\n", (long)result,
        ntopbuf, (unsigned)ntohs(sinaddr_client.sin_port), buf);
    revstr(buf);
    if (send(sock_client, buf, strlen(buf), 0) == -1)
        exit_sys("send");
}

shutdown(sock_client, SHUT_RDWR);
close(sock_client);
close(sock);

return 0;
}

char *revstr(char *str)
{
    size_t i, k;
    char temp;

    for (i = 0; str[i] != '\0'; ++i)
        ;

    for (--i, k = 0; k < i; ++k, --i) {
        temp = str[k];
        str[k] = str[i];
        str[i] = temp;
    }

    return str;
}

```

```

}

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

    exit(EXIT_FAILURE);
}

/* client.c */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>

#define BUFFER_SIZE      1024

void exit_sys(const char *msg);

int main(int argc, char *argv[])
{
    int sock;
    struct addrinfo *ai, *ri;
    struct addrinfo hints = {0};
    char buf[BUFFER_SIZE];
    char *str;
    ssize_t result;
    int sresult;

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

    if ((sock = socket(AF_INET, SOCK_STREAM, 0)) == -1)
        exit_sys("socket");

    hints.ai_family = AF_INET;
    hints.ai_socktype = SOCK_STREAM;

    if ((sresult = getaddrinfo(argv[1], argv[2], &hints, &ai)) != 0) {
        fprintf(stderr, "getaddrinfo: %s\n", gai_strerror(sresult));
        exit(EXIT_FAILURE);
    }

    for (ri = ai; ri != NULL; ri = ri->ai_next)
        if (connect(sock, ri->ai_addr, ri->ai_addrlen) != -1)
            break;

    if (ri == NULL)

```

```

        exit_sys("connect");

freeaddrinfo(ai);

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

for (;;) {
    printf("Yazı giriniz:");
    fgets(buf, BUFFER_SIZE, stdin);
    if ((str = strchr(buf, '\n')) != NULL)
        *str = '\0';
    if ((send(sock, buf, strlen(buf), 0)) == -1)
        exit_sys("send");
    if (!strcmp(buf, "quit"))
        break;

    if ((result = recv(sock, buf, BUFFER_SIZE, 0)) == -1)
        exit_sys("recv");
    if (result == 0)
        break;
    buf[result] = '\0';
    printf("%ld bytes received from %s (%s): %s\n", (long)result,
        argv[1], argv[2], buf);
}

shutdown(sock, SHUT_RDWR);
close(sock);

return 0;
}

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

    exit(EXIT_FAILURE);
}

```

/

*-----

Tıpkı borularda olduğu gibi çok client'lı TCP server programları birden fazla client'tan gelen bilgileri okuyabilmelidir.

Budurumda server programın organizasyonu client programlara göre daha zor olmaktadır. Buradaki server problemi yine şöyledir:

Server bir client için recv yaparken eğer o soketin alma tamponu boşsa bloke oluşur. Bu durumda maalesef diğer client'lardan gelmiş olan bilgileri okuyamaz. İşte bu tür durumlarda tıpkı daha önce borular için yaptığımız "ileri io modelleri" kullanılmalıdır.

Şüphesiz en basit çok client'lı organizasyon fork organizasyonudur.

Şöyle ki bu organizasyonda server her client bağlantısı sağlandığında fork yapar, böylece her client ile aslında ayrı bir proses konuşmuş olur. Şüphesiz her client bağlantısında bir

prosesin oluşturulması verimli bir yöntem değildir. Bu yöntemi uygularken üst prosesin otomatik zombie engelleme yapması gerekir. Çünkü üst proses hep accept işleminde bloke bekleyeceğine göre alt prosesi zombie'likten kurtaramaz.

Aşağıda böyle bir fork modeli örneği verilmiştir

```
-----*/  
  
/* server.c */  
  
#include <stdio.h>  
#include <stdlib.h>  
#include <string.h>  
#include <unistd.h>  
#include <sys/socket.h>  
#include <netinet/in.h>  
#include <arpa/inet.h>  
  
#define BUFFER_SIZE      1024  
  
char *revstr(char *str);  
void exit_sys(const char *msg);  
  
int main(int argc, char *argv[])  
{  
    int sock, sock_client;  
    struct sockaddr_in sinaddr, sinaddr_client;  
    socklen_t sinaddr_len;  
    char ntopbuf[INET_ADDRSTRLEN];  
    in_port_t port;  
    ssize_t result;  
    char buf[BUFFER_SIZE + 1];  
    pid_t pid;  
  
    if (argc != 2) {  
        fprintf(stderr, "wrong number of arguments!..\n");  
        exit(EXIT_FAILURE);  
    }  
  
    port = (in_port_t)strtol(argv[1], NULL, 10);  
  
    if ((sock = socket(AF_INET, SOCK_STREAM, 0)) == -1)  
        exit_sys("socket");  
  
    sinaddr.sin_family = AF_INET;  
    sinaddr.sin_port = htons(port);  
    sinaddr.sin_addr.s_addr = htonl(INADDR_ANY);  
  
    if (bind(sock, (struct sockaddr *)&sinaddr, sizeof(sinaddr)) == -1)  
        exit_sys("bind");  
  
    if (listen(sock, 8) == -1)  
        exit_sys("listen");
```

```

printf("Waiting for connection...\n");

for (;;) {
    sinaddr_len = sizeof(sinaddr_client);
    if ((sock_client = accept(sock, (struct sockaddr *)&sinaddr_client,
        &sinaddr_len)) == -1)
        exit_sys("accept");

    printf("Connected: %s : %u\n", inet_ntop(AF_INET,
        &sinaddr_client.sin_addr, ntopbuf, INET_ADDRSTRLEN),
        (unsigned)ntohs(sinaddr_client.sin_port));

    if ((pid = fork()) == -1)
        exit_sys("fork");
    if (pid == 0) {
        for(;;) {
            if ((result = recv(sock_client, buf, BUFFER_SIZE, 0)) == -1)
                exit_sys("recv");
            if (result == 0)
                break;
            buf[result] = '\0';
            if (!strcmp(buf, "quit"))
                break;
            printf("%ld bytes received from %s (%u): %s\n",
                (long)result, ntopbuf,
                (unsigned)ntohs(sinaddr_client.sin_port), buf);
            revstr(buf);
            if (send(sock_client, buf, strlen(buf), 0) == -1)
                exit_sys("send");
        }
        shutdown(sock_client, SHUT_RDWR);
        close(sock_client);
        exit(EXIT_SUCCESS);
    }
}

close(sock);

return 0;
}

char *revstr(char *str)
{
    size_t i, k;
    char temp;

    for (i = 0; str[i] != '\0'; ++i)
        ;

    for (--i, k = 0; k < i; ++k, --i) {
        temp = str[k];
        str[k] = str[i];
        str[i] = temp;
    }
}

```

```

    return str;
}

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

    exit(EXIT_FAILURE);
}

/* client.c */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>

#define BUFFER_SIZE      1024

void exit_sys(const char *msg);

int main(int argc, char *argv[])
{
    int sock;
    struct addrinfo *ai, *ri;
    struct addrinfo hints = {0};
    char buf[BUFFER_SIZE];
    char *str;
    ssize_t result;
    int sresult;

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

    if ((sock = socket(AF_INET, SOCK_STREAM, 0)) == -1)
        exit_sys("socket");

    hints.ai_family = AF_INET;
    hints.ai_socktype = SOCK_STREAM;

    if ((sresult = getaddrinfo(argv[1], argv[2], &hints, &ai)) != 0) {
        fprintf(stderr, "getaddrinfo: %s\n", gai_strerror(sresult));
        exit(EXIT_FAILURE);
    }

    for (ri = ai; ri != NULL; ri = ri->ai_next)
        if (connect(sock, ri->ai_addr, ri->ai_addrlen) != -1)
            break;

```

```

if (ri == NULL)
    exit_sys("connect");

freeaddrinfo(ai);

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

for (;;) {
    printf("Yazı giriniz:");
    fgets(buf, BUFFER_SIZE, stdin);
    if ((str = strchr(buf, '\n')) != NULL)
        *str = '\0';
    if ((send(sock, buf, strlen(buf), 0)) == -1)
        exit_sys("send");
    if (!strcmp(buf, "quit"))
        break;

    if ((result = recv(sock, buf, BUFFER_SIZE, 0)) == -1)
        exit_sys("recv");
    if (result == 0)
        break;
    buf[result] = '\0';
    printf("%ld bytes received from %s (%s): %s\n", (long)result,
        argv[1], argv[2], buf);
}

shutdown(sock, SHUT_RDWR);
close(sock);

return 0;
}

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

    exit(EXIT_FAILURE);
}

```

*-----

Yukarıdaki açıkladığımız fork modeli en basit model olmakla birlikte en verimsiz modeldir. Her client bağlantısında bir prosesin yaratılması client sayısı arttıkça sistemde yük oluşturur. Tabii fork modeli az client için uygulanabilecek bir modeldir. Ancak burada server client'lara müdahale edemez. Çünkü her server her client için ayrı proses yaratmıştır. İşte fork modeli yerine thread modeli her bakımdan daha etkindir. Bu modelde server her client bağlantısında bir proses değil thread yaratır.

Aşağıda thread modeline ilişkin örnek görülmektedir.

-----*/


```

/* server.c */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <pthread.h>

typedef struct {
    int sock;
    struct sockaddr_in sinaddr;
    char ntopbuf[INET_ADDRSTRLEN];
} CLIENT_INFO;

#define BUFFER_SIZE      1024

void *client_proc(void *param);
char *revstr(char *str);
void exit_sys(const char *msg);

int main(int argc, char *argv[])
{
    int sock, sock_client;
    struct sockaddr_in sinaddr, sinaddr_client;
    socklen_t sinaddr_len;
    in_port_t port;
    ssize_t result;
    CLIENT_INFO *ci;
    pthread_t tid;

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

    port = (in_port_t)strtoul(argv[1], NULL, 10);

    if ((sock = socket(AF_INET, SOCK_STREAM, 0)) == -1)
        exit_sys("socket");

    sinaddr.sin_family = AF_INET;
    sinaddr.sin_port = htons(port);
    sinaddr.sin_addr.s_addr = htonl(INADDR_ANY);

    if (bind(sock, (struct sockaddr *)&sinaddr, sizeof(sinaddr)) == -1)
        exit_sys("bind");

    if (listen(sock, 8) == -1)
        exit_sys("listen");

    printf("Waiting for connection...\n");

```

```

for (;;) {
    sinaddr_len = sizeof(sinaddr_client);
    if ((sock_client = accept(sock, (struct sockaddr *)&sinaddr_client,
        &sinaddr_len)) == -1)
        exit_sys("accept");

    if ((ci = (CLIENT_INFO *)malloc(sizeof(CLIENT_INFO))) == NULL)
        exit_sys("malloc");

    ci->sock = sock_client;
    ci->sinaddr = sinaddr_client;
    inet_ntop(AF_INET, &sinaddr_client.sin_addr, ci->ntopbuf,
        INET_ADDRSTRLEN);

    printf("Connected: %s : %u\n", ci->ntopbuf,
        (unsigned)ntohs(sinaddr_client.sin_port));

    if ((result = pthread_create(&tid, NULL, client_proc, ci)) != 0) {
        fprintf(stderr, "pthread_create: %s\n", strerror(result));
        exit(EXIT_FAILURE);
    }
    if ((result = pthread_detach(tid)) != 0) {
        fprintf(stderr, "pthread_detach: %s\n", strerror(result));
        exit(EXIT_FAILURE);
    }
}

close(sock);

return 0;
}

void *client_proc(void *param)
{
    CLIENT_INFO *ci = (CLIENT_INFO *)param;
    char buf[BUFFER_SIZE + 1];
    ssize_t result;

    for (;;) {
        if ((result = recv(ci->sock, buf, BUFFER_SIZE, 0)) == -1)
            exit_sys("recv");
        if (result == 0)
            break;
        buf[result] = '\0';
        if (!strcmp(buf, "quit"))
            break;
        printf("%ld bytes received from %s (%u): %s\n", (long)result, ci-
            >ntopbuf, (unsigned)ntohs(ci->sinaddr.sin_port), buf);
        revstr(buf);
        if (send(ci->sock, buf, strlen(buf), 0) == -1)
            exit_sys("send");
    }

    shutdown(ci->sock, SHUT_RDWR);
}

```

```

    close(ci->sock);

    free(ci);

    return NULL;
}

char *revstr(char *str)
{
    size_t i, k;
    char temp;

    for (i = 0; str[i] != '\0'; ++i)
        ;

    for (--i, k = 0; k < i; ++k, --i) {
        temp = str[k];
        str[k] = str[i];
        str[i] = temp;
    }

    return str;
}

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

    exit(EXIT_FAILURE);
}

/* client.c */

/* server.c */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

#define BUFFER_SIZE    1024

char *revstr(char *str);
void exit_sys(const char *msg);

int main(int argc, char *argv[])
{
    int sock, sock_client;
    struct sockaddr_in sinaddr, sinaddr_client;
    socklen_t sinaddr_len;
    char ntopbuf[INET_ADDRSTRLEN];
    in_port_t port;

```

```

ssize_t result;
char buf[BUFFER_SIZE + 1];

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

port = (in_port_t)strtoul(argv[1], NULL, 10);

if ((sock = socket(AF_INET, SOCK_STREAM, 0)) == -1)
    exit_sys("socket");

sinaddr.sin_family = AF_INET;
sinaddr.sin_port = htons(port);
sinaddr.sin_addr.s_addr = htonl(INADDR_ANY);

if (bind(sock, (struct sockaddr *)&sinaddr, sizeof(sinaddr)) == -1)
    exit_sys("bind");

if (listen(sock, 8) == -1)
    exit_sys("listen");

printf("Waiting for connection...\n");

sinaddr_len = sizeof(sinaddr_client);
if ((sock_client = accept(sock, (struct sockaddr *)&sinaddr_client,
    &sinaddr_len)) == -1)
    exit_sys("accept");

printf("Connected: %s : %u\n", inet_ntop(AF_INET, &sinaddr_client,
    ntopbuf, INET_ADDRSTRLEN), (unsigned)ntohs(sinaddr_client.sin_port));

for (;;) {
    if ((result = recv(sock_client, buf, BUFFER_SIZE, 0)) == -1)
        exit_sys("recv");
    if (result == 0)
        break;
    buf[result] = '\0';
    if (!strcmp(buf, "quit"))
        break;
    printf("Connected: %s : %u\n", inet_ntop(AF_INET, &sinaddr_client,
        ntopbuf, INET_ADDRSTRLEN),
        (unsigned)ntohs(sinaddr_client.sin_port));
    revstr(buf);
    if (send(sock_client, buf, strlen(buf), 0) == -1)
        exit_sys("send");
}

shutdown(sock_client, SHUT_RDWR);
close(sock_client);
close(sock);

return 0;
}

```

```

char *revstr(char *str)
{
    size_t i, k;
    char temp;

    for (i = 0; str[i] != '\0'; ++i)
        ;

    for (--i, k = 0; k < i; ++k, --i) {
        temp = str[k];
        str[k] = str[i];
        str[i] = temp;
    }

    return str;
}

```

```

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

    exit(EXIT_FAILURE);
}

```

/

*-----

Çok client'lı server soketler için en çok kullanılan modellerden biri "select modeli"dir. Bu model hem UNIX/Linux hem de Windows sistemlerine kullanılabilir. Bu model şöyle özetlenebilir:

- 1) Önce select fonksiyonunun için "read set" parametresi için içib oş bir fd_set nesnesi yaratılır.
- 2) Dinleme soketi (pasif soket) yaratılır, bind işlemi uygulanır ve bu soket "read set" içerisinde FD_SET ile eklenir. Artık dinleme soketinde bir bağlantı isteği olduğunda sanki bir okuma durumu oluşmuş gibi select blokesi çözülecektir. Eğer select'in blokesi dinleme soketi yüzünden çözülmüşse o noktada sizin accept işlemini yapmamız gerekir.
- 3) accept işlemiyle bir client ile bağlantı sağlandığında bu client'ın soketi de "read set" içerisine eklenir. Böylece artık "read set" içerisinde hem dinleme soketi hem de de client soketler bulunur.
- 4) Program bir döngü içerisinde select fonksiyonun çağrılması biçiminde oluşturulur. select'in blokesi çözüldüğünde hangi sokette okuma eyleminin oluştuğuna bakılır. Eğer dinleme soketinde okuma eylemi oluşmuşsa bu noktada yukarıda da belirtildiği gibi accepy yapılmalıdır, eğer bir client sokette okuma eylemi oluşmuşsa o client soketten read/recv yapılır.
- 5) Bir client close ya da shutdown uyguladığında bu da select için bir okuma eylemi oluşmuş gibi ele alınır. Bu durumda programcı read/recv

yapacak ve buradan eğer 0 değeri elde ederse karşı tarafın soketi kapattığını anlayacaktır. Artık programcının bu noktada ilgili client soketi kapatıp onu "read set" kümesinden çıkartması gerekir.

Aşağıda bir select server modeli görülmektedir.

```
-----*/

/* server.c */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <sys/select.h>

#define BUFFER_SIZE      1024

char *revstr(char *str);
void exit_sys(const char *msg);

int main(int argc, char *argv[])
{
    int sock, sock_client;
    struct sockaddr_in sinaddr, sinaddr_client;
    socklen_t sinaddr_len;
    char ntopbuf[INET_ADDRSTRLEN];
    in_port_t port;
    ssize_t result;
    char buf[BUFFER_SIZE + 1];
    fd_set rset, tset;
    int maxfd;
    int i;

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

    port = (in_port_t)strtoul(argv[1], NULL, 10);
    FD_ZERO(&rset);

    if ((sock = socket(AF_INET, SOCK_STREAM, 0)) == -1)
        exit_sys("socket");

    sinaddr.sin_family = AF_INET;
    sinaddr.sin_port = htons(port);
    sinaddr.sin_addr.s_addr = htonl(INADDR_ANY);

    if (bind(sock, (struct sockaddr *)&sinaddr, sizeof(sinaddr)) == -1)
        exit_sys("bind");
```

```

FD_SET(sock, &rset);
maxfd = sock;

if (listen(sock, 8) == -1)
    exit_sys("listen");

printf("Waiting for connection...\n");

for (;;) {
    tset = rset;
    if (select(maxfd + 1, &tset, NULL, NULL, NULL) == -1)
        exit_sys("select");
    for (i = 0; i <= maxfd; ++i)
        if (FD_ISSET(i, &tset)) {
            if (i == sock) {
                sinaddr_len = sizeof(sinaddr_client);
                if ((sock_client = accept(sock, (struct sockaddr
                    *)&sinaddr_client, &sinaddr_len)) == -1)
                    exit_sys("accept");

                inet_ntop(AF_INET, &sinaddr_client.sin_addr, ntopbuf,
                    INET_ADDRSTRLEN);
                printf("Connected: %s : %u\n", ntopbuf,
                    (unsigned)ntohs(sinaddr_client.sin_port));
                FD_SET(sock_client, &rset);
                if (sock_client > maxfd)
                    maxfd = sock_client;
                continue;
            }

            if ((result = recv(i, buf, BUFFER_SIZE, 0)) == -1)
                exit_sys("recv");
            if (result > 0) {
                buf[result] = '\0';
                sinaddr_len = sizeof(sinaddr_client);
                if (getpeername(i, (struct sockaddr *)&sinaddr_client,
                    &sinaddr_len) == -1)
                    exit_sys("getpeername");

                inet_ntop(AF_INET, &sinaddr_client.sin_addr, ntopbuf,
                    INET_ADDRSTRLEN);
                printf("%ld bytes received from %s (%u): %s\n",
                    (long)result, ntopbuf,
                    (unsigned)ntohs(sinaddr_client.sin_port), buf);
                revstr(buf);
                if (send(i, buf, strlen(buf), 0) == -1)
                    exit_sys("send");
            }
            else {
                shutdown(i, SHUT_RDWR);
                close(i);
                FD_CLR(i, &rset);
            }
        }
}

```

```

    }

    close(sock);

    return 0;
}

char *revstr(char *str)
{
    size_t i, k;
    char temp;

    for (i = 0; str[i] != '\0'; ++i)
        ;

    for (--i, k = 0; k < i; ++k, --i) {
        temp = str[k];
        str[k] = str[i];
        str[i] = temp;
    }

    return str;
}

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

    exit(EXIT_FAILURE);
}

/* client.c */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>

#define BUFFER_SIZE      1024

void exit_sys(const char *msg);

int main(int argc, char *argv[])
{
    int sock;
    struct addrinfo *ai, *ri;
    struct addrinfo hints = {0};
    char buf[BUFFER_SIZE];
    char *str;
    ssize_t result;
    int sresult;

```



```

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

if ((sock = socket(AF_INET, SOCK_STREAM, 0)) == -1)
    exit_sys("socket");

hints.ai_family = AF_INET;
hints.ai_socktype = SOCK_STREAM;

if ((sresult = getaddrinfo(argv[1], argv[2], &hints, &ai)) != 0) {
    fprintf(stderr, "getaddrinfo: %s\n", gai_strerror(sresult));
    exit(EXIT_FAILURE);
}

for (ri = ai; ri != NULL; ri = ri->ai_next)
    if (connect(sock, ri->ai_addr, ri->ai_addrlen) != -1)
        break;

if (ri == NULL)
    exit_sys("connect");

freeaddrinfo(ai);

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

for (;;) {
    printf("Yazı giriniz:");
    fgets(buf, BUFFER_SIZE, stdin);
    if ((str = strchr(buf, '\n')) != NULL)
        *str = '\0';
    if (!strcmp(buf, "quit"))
        break;

    if ((send(sock, buf, strlen(buf), 0)) == -1)
        exit_sys("send");

    if ((result = recv(sock, buf, BUFFER_SIZE, 0)) == -1)
        exit_sys("recv");
    if (result == 0)
        break;
    buf[result] = '\0';
    printf("%ld bytes received from %s (%s): %s\n", (long)result,
        argv[1], argv[2], buf);
}

shutdown(sock, SHUT_RDWR);
close(sock);

return 0;
}

void exit_sys(const char *msg)

```

```
{
    perror(msg);

    exit(EXIT_FAILURE);
}
```

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

Yukarıdaki programda daha önce görmediğimiz getpeername isimli bir fonksiyon kullanılmıştır. Bu fonksiyon bağlı bir soketi parametre olarak alır ve karşı tarafın ip adresini ve port numarasını bize sockaddr_in ya da sockaddr_in6 biçiminde verir. Tabii aslında server bağlantıyı yaptığında karşı tarafın bilgisini zaten accept fonksiyonunda almaktadır. B bilgi saklanarak kullanılabilir. Ancak bu bilgi saklanmamışsa istenildiği zaman getpeername fonksiyonuyla alınabilmektedir. Fonksiyonun prototipi şöyledir:

```
int getpeername(int sockfd, struct sockaddr *addr, socklen_t *addrlen);
```

Fonksiyonun birinci parametresi soket betimleyicisidir. İkinci parametre duruma göre bilgilerin yerleştirileceği struct sockaddr_in ya da struct sockaddr_in6 yapı nesnesinin adresini alır. Son parametre ikinci parametredeki yapının uzunluğudur. Eğer buraya az bir uzunluk girilirse kırma yapılır ve gerçek uzunluk verdiğimiz adresteki nesneye yerleştirilir. fonksiyon başarı durumunda 0 değerine başarısızlık durumunda -1 değerine geri dönmektedir.

getppername fonksiyonunun ters işlemini getsockname fonksiyonu yapmaktadır. Bu fonksiyon kendi bağlı soketimizin ip adresini ve port numarasını elde etmek için kullanılır. Genellikle b1 fonksiyona gereksinim duyulmamaktadır.

```
int getsockname(int sockfd, struct sockaddr *addr, socklen_t *addrlen);
```

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

poll modeli tamamen select modeline benzemektedir. Zaten poll fonksiyonu select fonksiyonunun değişik arayüzlü bir benzeri gibidir. Her ne kadar select ve poll aynı işlevlere sahip olsalar da poll fonksiyonun arayüzünün select fonksiyonundan daha iyi olduğu söylenebilir. Örneğin select fonksiyonunda maximum betimleyici sayısı FD_SETSIZE kadardır ve bu taşınabilir biçimde artırılamamaktadır. Oysa poll arayüzünde böyle bir kısıt yoktur. Yine pek çok sistemde select ve poll birbirlerine çok yakın performans göstermektedir. Fakat poll fonksiyonunun az daha iyi performanlolu olma eğilimi vardır. poll modeli de kısaca şu adımlardan geçilerek uygulanır.

- 1) Önce poll fonksiyonu için struct pollfd türünden uygun genişlikte bir dizi yaratılır. (Tabii programcı isterse dinamik dizi de kullanabilir)
- 2) Dinleme soketi (pasif soket) yaratılır, bind işlemi uygulanır ve bu soket, POLLIN koduyla diziyeye eklenir. Şimdi dizinin aktif eleman 1'dir. Artık dinleme soketinde bir bağlantı isteği olduğunda sanki POLLIN işlemi sağlanmış gibi select blokesi çözülecektir. poll fonksiyonunda dinleme soketine bağlantı isteği geldiğinde POLLIN olayı gerçekleşir. Eğer poll fonksiyonunun blokesi dinleme soketi yüzünden çözülmüşse o noktada bizim accept işlemini yapmamız gerekir.
- 3) accept işlemiyle bir client ile bağlantı sağlandığında bu client'ın soketi ve POLLIN olayı da dizinin sonuna eklenir. Böylece dizinin aktif eleman sayısı gittikçe artacaktır. Dizinin içerisinde hem dinleme soketinin hem de client soketlerin olduğuna dikkat ediniz.
- 4) Program bir döngü içerisinde poll fonksiyonun çağrılması biçiminde oluşturulur. poll'un blokesi çözüldüğünde hangi sokette POLLIN eyleminin oluştuğuna bakılır. Eğer dinleme soketinde POLLIN eylemi oluşmuşsa bu noktada yukarıda da belirtildiği gibi accept yapılmalıdır, eğer bir client sokette POLLIN eylemi oluşmuşsa o client soketten read/recv yapılır.
- 5) Bir client close ya da shutdown uyguladığında bu da poll için bir POLLIN eylemi oluşturur. Bu durumda programcı read/recv yapacak ve buradan eğer 0 değerini elde ederse karşı tarafın soketi kapattığını anlayacaktır. Artık programcının bu noktada ilgili client soketi kapatıp onu diziden çıkartması gerekir.

Aşağıda bir poll server modeli görülmektedir.

```
-----*/  
  
/* server.c */  
  
#include <stdio.h>  
#include <stdlib.h>  
#include <string.h>  
#include <unistd.h>  
#include <sys/socket.h>  
#include <netinet/in.h>  
#include <arpa/inet.h>  
#include <poll.h>  
  
#define BUFFER_SIZE      1024  
#define MAX_CLIENT      1024  
  
char *revstr(char *str);  
void exit_sys(const char *msg);  
  
int main(int argc, char *argv[])  
{  
    int sock, sock_client;
```

```

struct sockaddr_in sinaddr, sinaddr_client;
socklen_t sinaddr_len;
char ntopbuf[INET_ADDRSTRLEN];
in_port_t port;
ssize_t result;
char buf[BUFFER_SIZE + 1];
struct pollfd pfd[10];
int npfd;
int i;

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

port = (in_port_t)strtol(argv[1], NULL, 10);

if ((sock = socket(AF_INET, SOCK_STREAM, 0)) == -1)
    exit_sys("socket");

sinaddr.sin_family = AF_INET;
sinaddr.sin_port = htons(port);
sinaddr.sin_addr.s_addr = htonl(INADDR_ANY);

if (bind(sock, (struct sockaddr *)&sinaddr, sizeof(sinaddr)) == -1)
    exit_sys("bind");

npfd = 1;
pfd[0].fd = sock;
pfd[0].events = POLLIN;

if (listen(sock, 8) == -1)
    exit_sys("listen");

printf("Waiting for connection...\n");

for (;;) {
    if (poll(pfd, npfd, -1) == -1)
        exit_sys("poll");
    for (i = 0; i < npfd; ++i)
        if (pfd[i].revents & POLLIN) {
            if (pfd[i].fd == sock) {
                sinaddr_len = sizeof(sinaddr_client);
                if ((sock_client = accept(sock, (struct sockaddr
                    *)&sinaddr_client, &sinaddr_len)) == -1)
                    exit_sys("accept");

                inet_ntop(AF_INET, &sinaddr_client.sin_addr, ntopbuf,
                    INET_ADDRSTRLEN);
                printf("Connected: %s : %u\n", ntopbuf,
                    (unsigned)ntohs(sinaddr_client.sin_port));

                pfd[npfd].fd = sock_client;
                pfd[npfd].events = POLLIN;
                ++npfd;
            }
        }
}

```

```

        continue;
    }

    if ((result = recv(pfds[i].fd, buf, BUFFER_SIZE, 0)) == -1)
        exit_sys("recv");
    if (result > 0) {
        buf[result] = '\0';
        sinaddr_len = sizeof(sinaddr_client);

        if (getpeername(pfds[i].fd, (struct sockaddr
            *)&sinaddr_client, &sinaddr_len) == -1)
            exit_sys("getpeername");
        inet_ntop(AF_INET, &sinaddr_client.sin_addr, ntopbuf,
            INET_ADDRSTRLEN);
        printf("%ld bytes received from %s (%u): %s\n",
            (long)result, ntopbuf,
            (unsigned)ntohs(sinaddr_client.sin_port), buf);
        revstr(buf);
        if (send(pfds[i].fd, buf, strlen(buf), 0) == -1)
            exit_sys("send");
    }
    else {
        shutdown(pfds[i].fd, SHUT_RDWR);
        close(pfds[i].fd);
        pfds[i] = pfds[npfds - 1];
        --npfds;
    }
}

}

close(sock);

return 0;
}

char *revstr(char *str)
{
    size_t i, k;
    char temp;

    for (i = 0; str[i] != '\0'; ++i)
        ;

    for (--i, k = 0; k < i; ++k, --i) {
        temp = str[k];
        str[k] = str[i];
        str[i] = temp;
    }

    return str;
}

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

```

```

    exit(EXIT_FAILURE);
}

/* client.c */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>

#define BUFFER_SIZE      1024

void exit_sys(const char *msg);

int main(int argc, char *argv[])
{
    int sock;
    struct addrinfo *ai, *ri;
    struct addrinfo hints = {0};
    char buf[BUFFER_SIZE];
    char *str;
    ssize_t result;
    int sresult;

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

    if ((sock = socket(AF_INET, SOCK_STREAM, 0)) == -1)
        exit_sys("socket");

    hints.ai_family = AF_INET;
    hints.ai_socktype = SOCK_STREAM;

    if ((sresult = getaddrinfo(argv[1], argv[2], &hints, &ai)) != 0) {
        fprintf(stderr, "getaddrinfo: %s\n", gai_strerror(sresult));
        exit(EXIT_FAILURE);
    }

    for (ri = ai; ri != NULL; ri = ri->ai_next)
        if (connect(sock, ri->ai_addr, ri->ai_addrlen) != -1)
            break;

    if (ri == NULL)
        exit_sys("connect");

    freeaddrinfo(ai);

    printf("Connected...\n");
}

```

```

for (;;) {
    printf("Yazı giriniz:");
    fgets(buf, BUFFER_SIZE, stdin);
    if ((str = strchr(buf, '\n')) != NULL)
        *str = '\0';
    if (!strcmp(buf, "quit"))
        break;

    if ((send(sock, buf, strlen(buf), 0)) == -1)
        exit_sys("send");

    if ((result = recv(sock, buf, BUFFER_SIZE, 0)) == -1)
        exit_sys("recv");
    if (result == 0)
        break;
    buf[result] = '\0';
    printf("%ld bytes received from %s (%s): %s\n", (long)result,
        argv[1], argv[2], buf);
}

shutdown(sock, SHUT_RDWR);
close(sock);

return 0;
}

```

```

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

    exit(EXIT_FAILURE);
}

```

/

*-----

Anımsanacağı gibi poll yönteminin Linux sistemlerinde epoll isimli bir benzeri vardı. Linux sistemlerinde en iyi performansı bu epoll modeli vermektedir. Bu modelde önce epoll_create ile bir epoll nesnesi yaratılır. Sonra dinleme soketi epoll_ctl fonksiyonu ile ilgi listesine eklenir. Tıpkı poll modelinde olduğu gibi dinleme soketine bir bağlantı isteği geldiğinde, herhangi bir sokete okunacak bir bilgi geldiğinde ya da bir soket shutdown ya da close fonksiyonuyla kapatıldığında EPOLLIN olayı gerçekleşmektedir. İşte programcı döngü içerisinde epoll_wait fonksiyonu ile bekler. Sonra hangi sokette olay gerçekleşmişse gereğini yapar.

epoll modeli Linux'a özgüdür diğer UNIX türevi sistemlerde yoktur.

Aşağıda epoll modelini uygulayan bir "level triggered" server örneği verilmiştir.

```

-----*/

/* server.c */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <sys/epoll.h>

#define BUFFER_SIZE      1024
#define MAX_CLIENT       1024

char *revstr(char *str);
void exit_sys(const char *msg);

int main(int argc, char *argv[])
{
    int sock, sock_client;
    int epfd;
    struct sockaddr_in sinaddr, sinaddr_client;
    socklen_t sinaddr_len;
    char ntopbuf[INET_ADDRSTRLEN];
    in_port_t port;
    ssize_t result;
    char buf[BUFFER_SIZE + 1];
    int nepes;
    int rnepes;
    int i;
    struct epoll_event epe;
    struct epoll_event epes[MAX_CLIENT];

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

    port = (in_port_t)strtoul(argv[1], NULL, 10);

    if ((epfd = epoll_create(1024)) == -1)
        exit_sys("epoll_create");

    if ((sock = socket(AF_INET, SOCK_STREAM, 0)) == -1)
        exit_sys("socket");

    sinaddr.sin_family = AF_INET;
    sinaddr.sin_port = htons(port);
    sinaddr.sin_addr.s_addr = htonl(INADDR_ANY);

    if (bind(sock, (struct sockaddr *)&sinaddr, sizeof(sinaddr)) == -1)
        exit_sys("bind");

```



```

nepes = 1;
epe.events = EPOLLIN;
epe.data.fd = sock;
if (epoll_ctl(epfd, EPOLL_CTL_ADD, sock, &epe) == -1)
    exit_sys("epoll_ctl");

if (listen(sock, 8) == -1)
    exit_sys("listen");

printf("Waiting for connection...\n");

for (;;) {
    if ((rnepes = epoll_wait(epfd, epes, nepes, -1)) == -1)
        exit_sys("epoll");
    for (i = 0; i < rnepes; ++i)
        if (epes[i].events & EPOLLIN) {
            if (epes[i].data.fd == sock) {
                sinaddr_len = sizeof(sinaddr_client);
                if ((sock_client = accept(sock, (struct sockaddr
                *)&sinaddr_client, &sinaddr_len)) == -1)
                    exit_sys("accept");

                inet_ntop(AF_INET, &sinaddr_client.sin_addr, ntopbuf,
                INET_ADDRSTRLEN);

                epe.events = EPOLLIN;
                epe.data.fd = sock_client;
                if (epoll_ctl(epfd, EPOLL_CTL_ADD, sock_client, &epe) ==
                -1)
                    exit_sys("epoll_ctl");
                ++nepes;

                printf("New client Connected: %s : %u, total descriptor
                interested: %d\n", ntopbuf,
                (unsigned)ntohs(sinaddr_client.sin_port), nepes);

                continue;
            }
        }

    if ((result = recv(epes[i].data.fd, buf, BUFFER_SIZE, 0)) ==
    -1)
        exit_sys("recv");
    if (result > 0) {
        buf[result] = '\0';
        sinaddr_len = sizeof(sinaddr_client);

        if (getpeername(epes[i].data.fd, (struct sockaddr
        *)&sinaddr_client, &sinaddr_len) == -1)
            exit_sys("getpeername");
        inet_ntop(AF_INET, &sinaddr_client.sin_addr, ntopbuf,
        INET_ADDRSTRLEN);
        printf("%ld bytes received from %s (%u): %s\n",
        (long)result, ntopbuf,
        (unsigned)ntohs(sinaddr_client.sin_port), buf);
    }
}

```

```

        revstr(buf);
        if (send(epes[i].data.fd, buf, strlen(buf), 0) == -1)
            exit_sys("send");
    }
    else {
        shutdown(epes[i].data.fd, SHUT_RDWR);
        close(epes[i].data.fd);

        --nepes;
        printf("Client connection ended, total descriptors
            interested: %d\n", nepes);
    }
}

close(sock);
close(epfd);

return 0;
}

```

```

char *revstr(char *str)
{
    size_t i, k;
    char temp;

    for (i = 0; str[i] != '\0'; ++i)
        ;

    for (--i, k = 0; k < i; ++k, --i) {
        temp = str[k];
        str[k] = str[i];
        str[i] = temp;
    }

    return str;
}

```

```

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

    exit(EXIT_FAILURE);
}

```

/

*-----

Daha önceden de epoll IO modelinin hem düzey tetiklemeli (default) hem de kenar tetiklemeli modda çalışabildiğini belirtmiştik. Düzey tetiklemeli mod aslında select ve poll modelinin modudur. Düzey tetiklemeli modda eğer "read/recv yapıldığında bloke oluşmayacak bir durum varsa" select, poll ve epoll_wait hemen geri döner. Böylece bu modda biz gelen bilgilerin hepsini okumalıyız. Aksi takdirde

sürekli bloke çözülecektir ve bu da sonsuz döngü oluşturabilecektir. Ancak bazı uygulamalarda gelen bilginin hepsinin okunması istenmeyebilir.

Örneğin programcı gelen bilgi belli bir büyüklüğe ulaşınca onu kendi tamponuna taşımak isteyebilir. Bu durumda düzey tetiklemeli çalışma uygulanamaz.

Kenar tetiklemeli modda "yeni bilginin gelip gelmediğine" bakılmaktadır. Yani network tamponunda yeni bilginin olmasıyla değil yeni bir gelmesiyle bloke çözülmektedir. (Başka bir deyişle bir `epoll_wait` işleminden sonra ancak yeni bir bilgi gelmişse tetikleme oluşur.)

Düzyet tetiklemeli IO modellerinde bir bilgi geldiğinde o bilgi okunup işleme sokulurken başka bir bilginin gelmesi durumunda bu yeni gelen bilgi ikinci turda ancak ele alınabilmektedir. Buradaki zaman kaybını engellemek için soketi blokesiz moda sokup döngü içerisinde okuma yapmak uygun bir yöntem olabilmektedir. Örneğin:

```
nepes = epoll(.....);

for (i = 0; i < nepes; ++i) {
    if (epes[i].events & EPOLLIN) {
        for (;;) {
            result = recv(....);
            if (result == -1) {
                if (errno = EAGAIN)
                    break;
                exit_sys("recv");
            }
            ....
        }
    }
}
```

Bu yöntemde sokete bilgi gelmişse ele alınırken yeni bir bilginin gelmesi durumunda bu yeni gelen bilgi ikinci tura bırakılmadan hemen işlenebilmektedir. Bazı performans kritik durumlarda bu yöntem tercih edilebilmektedir. Ancak bu yöntem için de soketin blokesiz moda sokulması gerekir. Soket `accept` ile edildiğinde default blokeli moddadır. Soketi blokesiz moda sokmak `fcntl` fonksiyonuyla yapılabilir. Ancak nonblocking olarak döngüsel okumalarda sorunlardan biri belli bir client'tan çok fazla ve sürekli bilgi geldiğinde bu döngüde çok fazla zaman geçirilmesi ve bunun sonucunda da diğer soketlerle ilgilenilememesi olmaktadır. Programcı böyle bir sorunu belli bir limit koyarak çözebilir.

Kenar tetiklemeli modda eğer programcı her yeni tetiklemede önceden gelmiş olan tüm bilgileri okumayı garanti etmek istiyorsa ya blokeli modda `read/recv` fonksiyonunda talep ettiğinden daha az değerin okunmuş olmasına bakmalı ya da yukarıdaki gibi blokesiz modda bir döngü içerisinde `EAGAIN` durumu oluşana kadar `read/recv` uygulamalıdır.

`Epoll` yönteminde genel olarak kenar tetiklemeli modun düzey tetiklemeli moddan daha verimli olduğu görülmüştür. Kenar tetiklemeli modda

yeni bir bilgi geldiğinde tetiklemenin yapılması çok sayıda betimleyici ile çalışılırken etkin artışına yol açmaktadır.() Bunun en önemli nedenlerinden biri çekirdek kodlarının dha az betimleyiciye bakıyor olmasındandır.)

Yukarıdaki programı mantıksal olarak kenar tetiklemeli ve döngüsel biçime aşağıdaki gibi sokabiliriz.

```
-----*/
/* server.c */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <sys/epoll.h>

#define BUFFER_SIZE      1024
#define MAX_CLIENT      1024

char *revstr(char *str);
void exit_sys(const char *msg);

int main(int argc, char *argv[])
{
    int sock, sock_client;
    int epfd;
    struct sockaddr_in sinaddr, sinaddr_client;
    socklen_t sinaddr_len;
    char ntopbuf[INET_ADDRSTRLEN];
    in_port_t port;
    ssize_t result;
    char buf[BUFFER_SIZE + 1];
    int nepes;
    int rnepes;
    int i;
    struct epoll_event epe;
    struct epoll_event epes[MAX_CLIENT];

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

    port = (in_port_t)strtoul(argv[1], NULL, 10);

    if ((epfd = epoll_create(1024)) == -1)
```

```

    exit_sys("epoll_create");

if ((sock = socket(AF_INET, SOCK_STREAM, 0)) == -1)
    exit_sys("socket");

sinaddr.sin_family = AF_INET;
sinaddr.sin_port = htons(port);
sinaddr.sin_addr.s_addr = htonl(INADDR_ANY);

if (bind(sock, (struct sockaddr *)&sinaddr, sizeof(sinaddr)) == -1)
    exit_sys("bind");

nepes = 1;
epe.events = EPOLLIN|EPOLLET;
epe.data.fd = sock;
if (epoll_ctl(epfd, EPOLL_CTL_ADD, sock, &epe) == -1)
    exit_sys("epoll_ctl");

if (listen(sock, 8) == -1)
    exit_sys("listen");

printf("Waiting for connection...\n");

for (;;) {
    if ((rnepes = epoll_wait(epfd, epe, nepes, -1)) == -1)
        exit_sys("epoll");
    for (i = 0; i < rnepes; ++i)
        if (epe[i].events & EPOLLIN) {
            if (epe[i].data.fd == sock) {
                sinaddr_len = sizeof(sinaddr_client);
                if ((sock_client = accept(sock, (struct sockaddr
                    *)&sinaddr_client, &sinaddr_len)) == -1)
                    exit_sys("accept");
                if (fcntl(sock_client, F_SETFL, fcntl(sock_client,
                    F_GETFL) | O_NONBLOCK) == -1)
                    exit_sys("fcntl");

                inet_ntop(AF_INET, &sinaddr_client.sin_addr, ntopbuf,
                    INET_ADDRSTRLEN);

                epe.events = EPOLLIN|EPOLLET;
                epe.data.fd = sock_client;
                if (epoll_ctl(epfd, EPOLL_CTL_ADD, sock_client, &epe) ==
                    -1)
                    exit_sys("epoll_ctl");
                ++nepes;

                printf("New client Connected: %s : %u, total descriptor
                    interested: %d\n", ntopbuf,
                    (unsigned)ntohs(sinaddr_client.sin_port), nepes);

                continue;
            }
        }

    for (;;) {

```

```

        if ((result = recv(epes[i].data.fd, buf, BUFFER_SIZE,
            0)) == -1) {
            if (errno == EAGAIN)
                break;
            exit_sys("recv");
        }

        if (result > 0) {
            buf[result] = '\0';
            sinaddr_len = sizeof(sinaddr_client);

            if (getpeername(epes[i].data.fd, (struct sockaddr
                *)&sinaddr_client, &sinaddr_len) == -1)
                exit_sys("getpeername");
            inet_ntop(AF_INET, &sinaddr_client.sin_addr,
                ntopbuf, INET_ADDRSTRLEN);
            printf("%ld bytes received from %s (%u): %s\n",
                (long)result, ntopbuf,
                (unsigned)ntohs(sinaddr_client.sin_port), buf);
            revstr(buf);
            if (send(epes[i].data.fd, buf, strlen(buf), 0) ==
                -1)
                exit_sys("send");
        }
        else {
            shutdown(epes[i].data.fd, SHUT_RDWR);
            close(epes[i].data.fd);

            --nepes;
            printf("Client connection ended, total descriptors
                interested: %d\n", nepes);
            break;
        }
    }
}

close(sock);
close(epfd);

return 0;
}

```

```

char *revstr(char *str)
{
    size_t i, k;
    char temp;

    for (i = 0; str[i] != '\0'; ++i)
        ;

    for (--i, k = 0; k < i; ++k, --i) {
        temp = str[k];
        str[k] = str[i];
        str[i] = temp;
    }
}

```

```

    }

    return str;
}

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

    exit(EXIT_FAILURE);
}

```

```

/
*-----
-----
Nihayet soket uygulamalarında asenkron IO modeli de uygulanabilmektedir.
Ancak bu modelim soketlerde taşınabilirlik bakımından bazı
sorunları vardır. POSIX standartları bu modelin soketlerde kullanımı
konusunda gerekli olan bazı gri noktaları açıklamamıştır.
Dolayısıyla asenkron model soketlerde kullanılırken taşınabilirlik
sorunları oluşabilmektedir. Linux sistemlerinde bu model epoll
modelinden
daha yavaş olma eğilimindedir.

Bu modelde daha önce yaptığımız gibi her betimleyici için bir AIO Kontrol
Blok oluşturulur ve sonra okuma işlemi aio_read ile başlatılır.
İşlem bittiğinde sistem bize bir sinyal ya da thread yoluyla bunu
bildirmektedir. Biz de orada yeniden aio_read yaparak işlemi devam
ettiririz.

Bu yöntemde dinleme soketi üzerinde asenkron işlem yapılamamaktadır.
Yani yalnızca accept ile elde edilen soket üzerinde asenkron işlemler
yapılabilmektedir.
-----
-----*/

```

```

/* server.c */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <aio.h>

#define BUFFER_SIZE      1024
#define MAX_CLIENT       1024

typedef struct {
    struct aiocb cb;
    char buf[BUFFER_SIZE + 1];
    char peer_name[INET_ADDRSTRLEN];
    in_port_t peer_port;
}

```

```

} IOCB_BUF;

void read_completion_proc(union sigval val);
char *revstr(char *str);
void exit_sys(const char *msg);

int main(int argc, char *argv[])
{
    int sock, sock_client;
    struct sockaddr_in sinaddr, sinaddr_client;
    socklen_t sinaddr_len;
    in_port_t port;
    IOCB_BUF *aiobuf;

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

    port = (in_port_t)strtoul(argv[1], NULL, 10);

    if ((sock = socket(AF_INET, SOCK_STREAM, 0)) == -1)
        exit_sys("socket");

    sinaddr.sin_family = AF_INET;
    sinaddr.sin_port = htons(port);
    sinaddr.sin_addr.s_addr = htonl(INADDR_ANY);

    if (bind(sock, (struct sockaddr *)&sinaddr, sizeof(sinaddr)) == -1)
        exit_sys("bind");

    if (listen(sock, 8) == -1)
        exit_sys("listen");

    printf("Waiting for connection...\n");

    for (;;) {
        if ((sock_client = accept(sock, (struct sockaddr *)&sinaddr_client,
            &sinaddr_len)) == -1)
            exit_sys("accept");

        if ((aiobuf = (IOCB_BUF *)calloc(1, sizeof(IOCB_BUF))) == NULL)
            exit_sys("calloc");

        inet_ntop(AF_INET, &sinaddr_client.sin_addr, aiobuf->peer_name,
            INET_ADDRSTRLEN);
        aiobuf->peer_port = htons(sinaddr_client.sin_port);

        printf("New client Connected: %s : %u\n", aiobuf->peer_name,
            (unsigned)aiobuf->peer_port);

        aiobuf->cb.aio_fildes = sock_client;
        aiobuf->cb.aio_offset = 0;
        aiobuf->cb.aio_buf = aiobuf->buf;
        aiobuf->cb.aio_nbytes = BUFFER_SIZE;
    }
}

```



```

        aiobuf->cb.aio_reqprio = 0;
        aiobuf->cb.aio_sigevent.sigev_notify = SIGEV_THREAD;
        aiobuf->cb.aio_sigevent.sigev_value.sival_ptr = aiobuf;
        aiobuf->cb.aio_sigevent.sigev_notify_function =
            read_completion_proc;

        if (aio_read(&aiobuf->cb) == -1)
            exit_sys("aio_read");
    }

    close(sock);

    return 0;
}

void read_completion_proc(union sigval val)
{
    IOCB_BUF *aiobuf = (IOCB_BUF *)val.sival_ptr;
    ssize_t n;

    if ((n = aio_return(&aiobuf->cb)) == -1)
        exit_sys("aio_return");

    if (n == 0) {
        shutdown(aiobuf->cb.aio_fildes, SHUT_RDWR);
        close(aiobuf->cb.aio_fildes);
        free(aiobuf);
        return;
    }

    aiobuf->buf[n] = '\0';
    printf("%ld bytes received from %s (%u): %s\n", (long)n, aiobuf->
        >peer_name, (unsigned)aiobuf->peer_port, (char *)aiobuf->cb.aio_buf);
    revstr(aiobuf->buf);
    if (send(aiobuf->cb.aio_fildes, (void *)aiobuf->cb.aio_buf,
        strlen(aiobuf->buf), 0) == -1)
        exit_sys("send");

    if (aio_read(&aiobuf->cb) == -1)
        exit_sys("aio_read");
}

char *revstr(char *str)
{
    size_t i, k;
    char temp;

    for (i = 0; str[i] != '\0'; ++i)
        ;

    for (--i, k = 0; k < i; ++k, --i) {
        temp = str[k];
        str[k] = str[i];
        str[i] = temp;
    }
}

```

```

    return str;
}

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

    exit(EXIT_FAILURE);
}

/* client.c */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>

#define BUFFER_SIZE      1024

void exit_sys(const char *msg);

int main(int argc, char *argv[])
{
    int sock;
    struct addrinfo *ai, *ri;
    struct addrinfo hints = {0};
    char buf[BUFFER_SIZE];
    char *str;
    ssize_t result;
    int sresult;

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

    if ((sock = socket(AF_INET, SOCK_STREAM, 0)) == -1)
        exit_sys("socket");

    hints.ai_family = AF_INET;
    hints.ai_socktype = SOCK_STREAM;

    if ((sresult = getaddrinfo(argv[1], argv[2], &hints, &ai)) != 0) {
        fprintf(stderr, "getaddrinfo: %s\n", gai_strerror(sresult));
        exit(EXIT_FAILURE);
    }

    for (ri = ai; ri != NULL; ri = ri->ai_next)
        if (connect(sock, ri->ai_addr, ri->ai_addrlen) != -1)
            break;

```

```

if (ri == NULL)
    exit_sys("connect");

freeaddrinfo(ai);

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

for (;;) {
    printf("Yazı giriniz:");
    fgets(buf, BUFFER_SIZE, stdin);
    if ((str = strchr(buf, '\n')) != NULL)
        *str = '\0';
    if (!strcmp(buf, "quit"))
        break;

    if ((send(sock, buf, strlen(buf), 0)) == -1)
        exit_sys("send");

    if ((result = recv(sock, buf, BUFFER_SIZE, 0)) == -1)
        exit_sys("recv");
    if (result == 0)
        break;
    buf[result] = '\0';
    printf("%ld bytes received from %s (%s): %s\n", (long)result,
        argv[1], argv[2], buf);
}

shutdown(sock, SHUT_RDWR);
close(sock);

return 0;
}

```

```

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

    exit(EXIT_FAILURE);
}

```

/

*-----

UDP (User Datagram Protocol) bağlantılı olmayan bir protokoldür. Dolayısıyla bir taraf bir tarafa hiç bağlanmadan onun IP adresini ve port numarasını bilerek UDP paketlerini gönderebilir. Gönderen taraf alan tarafın paketi alıp almadığını bilmez. Bir akış kontrolü yoktur. Dolayısıyla alan taraf bilgi kaçırabilir. UDP tabii ki TCP'ye göre daha hızlıdır. Zaten TCP bir bakıma UDP'nin organize edilmiş bağlantılı biçimidir.

UDP bağlantılı olmadığı için burada "client" ve "server" terimleri tam oturmamaktadır. Ancak yine de genellikle bilgiyi gönderen tarafa

"client" alan tarafa "server" denilmektedir. UDP özellikle periyodik kısa bilgilerin gönderildiği ve alındığı durumlarda tercih edilmektedir.

UDP haberleşmesinde bilgi alan tarafın (server) bilgi kaçırabilmesi söz konusu olabileceğinden dolayı böyle kaçırmalarda sistemde önemli bir aksamanın olmaması gerekir. Örneğin birtakım makineler belli periyotlarda server'a "ben çalışıyorum" demek için periyodik UDP paketleri yollayabilir.

Server da hangi makinenin çalıştığını anlayabilir. Bir araba simülatörü arabanın durumunu UDP paketleriyle dış dünyaya verebilir. Ya da örneğin bir görüntü aktarımı UDP paketleriyle yapılabilir. Bir UDP paketi 64K gibi bir sınıra sahiptir. Büyük verilerin UDP ile gönderilmesi için programcının

paketlere kendisinin manuel numara vermesi gerekebilir. Zaten TCP protokolü bu şekilde bir numaralandırmayı kendi içerisinde yapmaktadır. UDP haberleşmesinin

önemli bir farkı da "broadcasting" işlemidir. Yerel ağda belli bir makine tüm host'lara UDP paketini gönderebilir. TCP'de böyle bir broadcasting mekanizması yoktur.

UDP server programın tasarımı şöyledir:

- 1) Server socket fonksiyonuyla soketi SOCK_DGRAM parametresiyle yaratır.
- 2) Soketi bind fonksiyonuyla bağlar.
- 3) recvfrom fonksiyonuyla gelen paketleri alır ve işler
- 4) Haberleşme bitince soketi close ile kapatır.

UDP Client Programın tasarımı da şöyledir:

- 1) Client socket fonksiyonuyla soketi SOCK_DGRAM parametresiyle yaratır.
- 2) Client isteğe bağlı olarak soketi bind fonksiyonuyla bağlayabilir.
- 3) Client server'ın host isminden hareketle server'ın IP adresini gethostbyname ya da getaddrinfo fonksiyonuyla elde edebilir.
- 4) Client sendto fonksiyonuyla UDP paketlerini gönderir.
- 5) Haberleşme birince client close fonksiyonuyla soketi kapatır.

recvfrom fonksiyonu UDP paketini okumak için kullanılmaktadır.

Fonksiyonun prototipi şöyledir:

```
ssize_t recvfrom(int socket, void *buffer, size_t length, int flags, struct sockaddr *address, socklen_t *address_len);
```

Fonksiyonun birinci parametresi bind edilen soketi belirtir. ikinci parametre alınacak bilginin yerleştirileceğin adresi belirtir. Üçüncü parametre

ikinci parametredeki dizinin uzunluğunu belirtir. Eğer buradaki değer değer UDP paketindeki gönderilmiş olan byte sayısından daha az ise kırpılarak

diziye yerleştirme yapılmaktadır. flgas parametresi birkaç seçeneğe sahiptir. 0 girilebilir. fonksiyonun dördüncü parametresi UDP paketin gönderen tarafın

IP adresinin ve port numarasının yerleştirileceği sockaddr_in yapısının adresini alır. Son parametre ise bu yapının uzunluğunu tutan int nesnenin adresini almaktadır.

Eğer bu değer küçük girilirse fonksiyon asıl değeri buraya yerleştirmektedir. Fonksiyon başarı durumunda UDP paketindeki byte sayısına, başarısızlık durumunda -1 değerine geri dönmektedir.

sendto fonksiyonu da şöyledir:

```
ssize_t sendto(int socket, const void *message, size_t length, int flags, const struct sockaddr *dest_addr, socklen_t dest_len);
```

Fonksiyonun parametreleri recvfrom da olduğu gibidir. Ancak buffer yönü terstir. Fonksiyon blokeli modda UDP paketinin tamamı network tamponuna yazılana kadar blokeye yol açmaktadır. Başarı durumunda network tamponuna yazılan byte sayısına başarısızlık durumunda -1'e geri dönmektedir.

Aşağıda örnek bir UDP-Server ve Client_program verilmiştir.

```
-----*/  
  
/* udp-server.c */  
  
#include <stdio.h>  
#include <stdlib.h>  
#include <string.h>  
#include <unistd.h>  
#include <sys/socket.h>  
#include <netinet/in.h>  
#include <arpa/inet.h>  
  
#define BUFFER_SIZE      4096  
  
char *revstr(char *str);  
void exit_sys(const char *msg);  
  
int main(int argc, char *argv[])  
{  
    int sock;  
    struct sockaddr_in sinaddr, sinaddr_client;  
    socklen_t sinaddr_len;  
    char ntopbuf[INET_ADDRSTRLEN];  
    in_port_t port;  
    ssize_t result;  
    char buf[BUFFER_SIZE + 1];  
  
    if (argc != 2) {  
        fprintf(stderr, "wrong number of arguments!...\n");  
        exit(EXIT_FAILURE);  
    }  
  
    port = (in_port_t)strtoul(argv[1], NULL, 10);  
  
    if ((sock = socket(AF_INET, SOCK_DGRAM, 0)) == -1)  
        exit_sys("socket");
```

```

sinaddr.sin_family = AF_INET;
sinaddr.sin_port = htons(port);
sinaddr.sin_addr.s_addr = htonl(INADDR_ANY);

if (bind(sock, (struct sockaddr *)&sinaddr, sizeof(sinaddr)) == -1)
    exit_sys("bind");

printf("Waiting for client data...\n");

for (;;) {
    sinaddr_len = sizeof(sinaddr_client);
    if ((result = recvfrom(sock, buf, BUFFER_SIZE, 0, (struct sockaddr
        *)&sinaddr_client, &sinaddr_len)) == -1)
        exit_sys("recvfrom");

    buf[result] = '\0';
    inet_ntop(AF_INET, &sinaddr_client.sin_addr, ntopbuf,
        INET_ADDRSTRLEN);
    printf("%ld bytes received from %s (%u): %s\n", (long)result,
        ntopbuf, (unsigned)ntohs(sinaddr_client.sin_port), buf);

    revstr(buf);
    if (sendto(sock, buf, strlen(buf), 0, (struct sockaddr
        *)&sinaddr_client, sizeof(struct sockaddr)) == -1)
        exit_sys("sendto");
}

close(sock);

return 0;
}

char *revstr(char *str)
{
    size_t i, k;
    char temp;

    for (i = 0; str[i] != '\0'; ++i)
        ;

    for (--i, k = 0; k < i; ++k, --i) {
        temp = str[k];
        str[k] = str[i];
        str[i] = temp;
    }

    return str;
}

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

    exit(EXIT_FAILURE);
}

```

```

}

/* udp-client.c */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>

#define BUFFER_SIZE      4096

void exit_sys(const char *msg);

int main(int argc, char *argv[])
{
    int sock;
    struct addrinfo *ai;
    struct addrinfo hints = {0};
    struct sockaddr_in sinaddr;
    int sinaddr_len;
    char buf[BUFFER_SIZE];
    char ntopbuf[INET_ADDRSTRLEN];
    char *str;
    ssize_t result;
    int sresult;

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

    if ((sock = socket(AF_INET, SOCK_DGRAM, 0)) == -1)
        exit_sys("socket");

    hints.ai_family = AF_INET;
    hints.ai_socktype = SOCK_DGRAM;

    if ((sresult = getaddrinfo(argv[1], argv[2], &hints, &ai)) != 0) {
        fprintf(stderr, "getaddrinfo: %s\n", gai_strerror(sresult));
        exit(EXIT_FAILURE);
    }

    for (;;) {
        printf("Yazı giriniz:");
        fgets(buf, BUFFER_SIZE, stdin);
        if ((str = strchr(buf, '\n')) != NULL)
            *str = '\0';
        if (!strcmp(buf, "quit"))
            break;
    }
}

```

```

    if (sendto(sock, buf, strlen(buf), 0, ai->ai_addr, sizeof(struct
        sockaddr)) == -1)
        exit_sys("sendto");

    sinaddr_len = sizeof(sinaddr);
    if ((result = recvfrom(sock, buf, BUFFER_SIZE, 0, (struct sockaddr
        *)&sinaddr, &sinaddr_len)) == -1)
        exit_sys("recvfrom");

    buf[result] = '\0';
    inet_ntop(AF_INET, &sinaddr.sin_addr, ntopbuf, INET_ADDRSTRLEN);
    printf("%ld bytes received from %s (%u): %s\n", (long)result,
        ntopbuf, (unsigned)ntohs(sinaddr.sin_port), buf);
}

freeaddrinfo(ai);
close(sock);

return 0;
}

```

```

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

    exit(EXIT_FAILURE);
}

```

/

*-----

Server uygulamalarında server'ın bir client'tan gelen isteği yerine getirmesi bir zaman kaybı oluşturabilmektedir. Server bir client ile uğraşırken diğer client'ların istekleri mecburen bekletilir. İşte bu durumu en aza indirmek için select, poll, epoll modellerinde server bir client'ın isteğini bir thread ile yerine getirebilir. Böylece birden fazla client'a aynı anda hizmet verebilecektir.

Örneğin select modelinde bu işlem şöyle yapılabilir:

```

for (;;) {
    select(.....);
    if (<belli bir sokete bilgi geldi>) {
        <thread oluştur ve işlemi thread'e devret>
    }
}

```

Tabii burada küçük bir işlem için yeni bir thread'in yaratılıp yok edilmesi etkin bir yöntem değildir. Çünkü bilindiği gibi thread'lerin yaratılıp yok edilmeleri de dikkate değer bir zaman kaybı oluşturmaktadır. İşte bu tür durumlarda "thread havuzları (thread pools)" kullanılabilir. Thread havuzlarında zaten belli bir miktar thread yaratılmış ve bekler durumda (suspend durumda) tutulmaktadır. Böylece client'tan gelen

isteğin bu thread'lerden biri tarafından gerçekleştirilmesi uygun olur. Tabii havuzda ne kadar thread'in bekletileceği neselesi vardır. Eğer genel amaçlı bir thread havuzu yazılacaksa havuzda bekletilecek thread dinamik bir biçimde belirlenebilir. (Yani çok fazla gereksinim olduğunda havuzu büyütmek, gereksinim azaldığında küçültmek gibi.) Daha önceden de belirttiğimiz gibi UNIX/Linux sistemlerinde standart bir thread havuzu mekanizması yoktur. Halbuki örneğin Windows sistemlerinde işletim sistemi tarafından sunulan threda havuz mekanizması bulunmaktadır. Bu durumda UNIX/Linux sistemlerinde başkaları tarafından yazılmış thread havuzları kullanılabilir ya da uygulamaya yönelik basit bir havuz mekanizması oluşturulabilir.

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

recv ve send fonksiyonlarının read ve write fonksiyonlarından tek farkı flags parametresidir. Biz yukarıdaki örneklerde bu parametreyi 0 geçtik. Dolayısıyla yukarıdaki örneklerde kullandığımız recv ve send fonksiyonlarının read ve write fonksiyonlarından hiçbir farkı yoktur. Pekiyi bu flag değerleri neler olabilir? İşte POSIX standartlarında recv fonksiyonundaki flag değerleri şunlardan biri ya da birden fazlası olabilir:

MSG_PEEK: Bu seçenekte bilgi reive tamponundan alınır ama oradan silinmez. (Belki de programcı mesaj uygunsa okumak istemektedir.)

MSG_OOB: Outof-band data (urgent data) denilen okumalar için kullanılmaktadır.

MSG_WAITALL: Bu seçenekte n byte okunmak istendiğinde bu n byte kesin olarak okunana kadar blokede beklenir. Fakat bu durumda bir sinyal geldiğinde yine recv -1 ile geri döner ve errno EINTR ile set edilir.

send fonksiyonundaki POSIX bayrakları da şunlardır:

MSG_EOR: Soket türü SOCK_SEQPACKET ise kaydı sonlandırmakta kullanılır.

MSG_OOB: Outof-band data gönderimi için kullanılmaktadır.

MSG_NOSIGNAL: Normal olarak send ya da write işlemi yapılırken karşı taraf soketi kapatmışsa bu fonksiyonların çağrıldığı tarafta

SIGPIPE sinyali oluşmaktadır. Bu bayrak kullanılırsa SIGPIPE sinyali oluşmaz send -1 ile geri döner ve errno EPIPE değeri ile set edilir.

Linux POSIX'in bayraklarından daha fazlasını bulundurmaktadır. Örneğin recv ve send işleminde MSG_DONTWAIT bir çağrılımlık "nonblocking" etki yaratmaktadır. Yani recv sırasında network tamponunda hiç bilgi yoksa recv bloke olmaz, -1 ile geri döner errno EAGAIN değeri ile set edilir. send işlemi sırasında da network tamponu dolu ise send bloke olmaz -1 ile geri döner errno EAGAIN değeri ile set edilir.

```

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

Client server uygulamaların çoğunda client server'a isteğini yazısal
biçimde iletmektedir. Fakat istek binary biçimde de
iletilebilir. bu durumda server'ın soketten belli bir miktar
okuaybilecek biçimde orhganize edilmesi gerekir. Bazen client
önce server'a okunacak mesajın uzunluğunu gönderir. Sonra da mesajın
kendisi gönderir. Bu yöntem text ya da binary tabanlı
yapılabilmektedir.
Bu durumda programcının belli bir miktar okuyana kadar biriktirme
yapması gerekir. Garantili n byte okumak için recv fonksiyonu
MSG_WAITALL
bayrağı ile kullanılabilir. Ancak bu da sıkıntılıdır. Çünkü select,
poll, ya da asenkron io modellerinde n byte okuyana kadar bloke
oluşturmak
tasarıma terstir. Bu tür durumlarda daha önce borularda yaptığımız gibi
"index, left" mekanizmasını kullanabiliriz.

Aşağıda her defasında BUFFER_SIZE (10) byte bilginin hepsi okunduğunda
onları işleme soken bir epoll modeli server örneği
verilmiştir.

-----*/

/* server.c */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <sys/epoll.h>

#define BUFFER_SIZE      10
#define MAX_CLIENT       1024

typedef struct {
    int fd;
    char buf[BUFFER_SIZE + 1];
    ssize_t index;
    ssize_t left;
} CLIENT_INFO;

char *revstr(char *str);
void exit_sys(const char *msg);

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

```

```

int sock, sock_client;
int epfd;
struct sockaddr_in sinaddr, sinaddr_client;
socklen_t sinaddr_len;
char ntopbuf[INET_ADDRSTRLEN];
in_port_t port;
ssize_t result;
int nepes;
int rnepes;
int i;
struct epoll_event epe;
struct epoll_event epes[MAX_CLIENT];
CLIENT_INFO *ci;

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

port = (in_port_t)strtol(argv[1], NULL, 10);

if ((epfd = epoll_create(1024)) == -1)
    exit_sys("epoll_create");

if ((sock = socket(AF_INET, SOCK_STREAM, 0)) == -1)
    exit_sys("socket");

sinaddr.sin_family = AF_INET;
sinaddr.sin_port = htons(port);
sinaddr.sin_addr.s_addr = htonl(INADDR_ANY);

if (bind(sock, (struct sockaddr *)&sinaddr, sizeof(sinaddr)) == -1)
    exit_sys("bind");

nepes = 1;
epe.events = EPOLLIN;
epe.data.fd = sock;
if (epoll_ctl(epfd, EPOLL_CTL_ADD, sock, &epe) == -1)
    exit_sys("epoll_ctl");

if (listen(sock, 8) == -1)
    exit_sys("listen");

printf("Waiting for connection...\n");

for (;;) {
    if ((rnepes = epoll_wait(epfd, epes, nepes, -1)) == -1)
        exit_sys("epoll");
    for (i = 0; i < rnepes; ++i)
        if (epes[i].events & EPOLLIN) {
            if (epes[i].data.fd == sock) {
                sinaddr_len = sizeof(sinaddr_client);
                if ((sock_client = accept(sock, (struct sockaddr *)
                    &sinaddr_client, &sinaddr_len)) == -1)
                    exit_sys("accept");
            }
        }
    }

```

```

    inet_ntop(AF_INET, &sinaddr_client.sin_addr, ntopbuf,
              INET_ADDRSTRLEN);

    if ((ci = (CLIENT_INFO *)malloc(sizeof(CLIENT_INFO))) ==
        NULL)
        exit_sys("malloc");

    ci->fd = sock_client;
    ci->index = 0;
    ci->left = BUFFER_SIZE;
    epe.events = EPOLLIN;
    epe.data.ptr = ci;

    if (epoll_ctl(epfd, EPOLL_CTL_ADD, sock_client, &epe) ==
        -1)
        exit_sys("epoll_ctl");
    ++nepes;

    printf("New client Connected: %s : %u, total descriptor
        interested: %d\n", ntopbuf,
        (unsigned)ntohs(sinaddr_client.sin_port), nepes);

    continue;
}

ci = (CLIENT_INFO *)epes[i].data.ptr;
if ((result = recv(ci->fd, ci->buf + ci->index, ci->left,
    0)) == -1)
    exit_sys("recv");
if (result > 0) {
    ci->index += result;
    ci->left -= result;
    if (ci->left == 0) {
        ci->buf[ci->index] = '\0';
        sinaddr_len = sizeof(sinaddr_client);

        if (getpeername(ci->fd, (struct sockaddr
            *)&sinaddr_client, &sinaddr_len) == -1)
            exit_sys("getpeername");
        inet_ntop(AF_INET, &sinaddr_client.sin_addr,
            ntopbuf, INET_ADDRSTRLEN);
        printf("%ld bytes received from %s (%u): %s\n",
            (long)ci->index, ntopbuf,
            (unsigned)ntohs(sinaddr_client.sin_port), ci->buf);
        ci->index = 0;
        ci->left = BUFFER_SIZE;
    }
}
else {
    shutdown(ci->fd, SHUT_RDWR);
    close(ci->fd);
    --nepes;
    free(ci);
}

```

```

        printf("Client connection ended, total descriptors
        interested: %d\n", nepes);
    }
}

close(sock);
close(epfd);

return 0;
}

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

    exit(EXIT_FAILURE);
}

/* client.c */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>

#define BUFFER_SIZE      1024

void exit_sys(const char *msg);

int main(int argc, char *argv[])
{
    int sock;
    struct addrinfo *ai, *ri;
    struct addrinfo hints = {0};
    char buf[BUFFER_SIZE];
    char *str;
    int result;

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

    if ((sock = socket(AF_INET, SOCK_STREAM, 0)) == -1)
        exit_sys("socket");

    hints.ai_family = AF_INET;
    hints.ai_socktype = SOCK_STREAM;

    if ((result = getaddrinfo(argv[1], argv[2], &hints, &ai)) != 0) {

```

```

        fprintf(stderr, "getaddrinfo: %s\n", gai_strerror(result));
        exit(EXIT_FAILURE);
    }

    for (ri = ai; ri != NULL; ri = ri->ai_next)
        if (connect(sock, ri->ai_addr, ri->ai_addrlen) != -1)
            break;

    if (ri == NULL)
        exit_sys("connect");

    freeaddrinfo(ai);

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

    for (;;) {
        printf("Yazı giriniz:");
        fgets(buf, BUFFER_SIZE, stdin);
        if ((str = strchr(buf, '\n')) != NULL)
            *str = '\0';
        if (!strcmp(buf, "quit"))
            break;

        if ((send(sock, buf, strlen(buf), 0)) == -1)
            exit_sys("send");
    }

    shutdown(sock, SHUT_RDWR);
    close(sock);

    return 0;
}

```

```

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

    exit(EXIT_FAILURE);
}

```

/

```

*-----
-----

```

Aşağıda biriktirmeli işlem yapan server'ın select ile gerçekleştirimi görülmektedir.

```

-----*/

```

```

/* server.c */

```

```

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

```

```

#include <netinet/in.h>
#include <arpa/inet.h>
#include <sys/select.h>

#define BUFFER_SIZE      10
#define MAX_CLIENT      1024

char *revstr(char *str);
void exit_sys(const char *msg);

typedef struct {
    int fd;
    char buf[BUFFER_SIZE + 1];
    ssize_t index;
    ssize_t left;
} CLIENT_INFO;

int main(int argc, char *argv[])
{
    int sock, sock_client;
    struct sockaddr_in sinaddr, sinaddr_client;
    socklen_t sinaddr_len;
    char ntopbuf[INET_ADDRSTRLEN];
    in_port_t port;
    ssize_t result;
    fd_set rset, tset;
    CLIENT_INFO clients[MAX_CLIENT];
    int maxfd;
    int nclients = 0;
    int i;

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

    port = (in_port_t)strtoul(argv[1], NULL, 10);
    FD_ZERO(&rset);

    if ((sock = socket(AF_INET, SOCK_STREAM, 0)) == -1)
        exit_sys("socket");

    sinaddr.sin_family = AF_INET;
    sinaddr.sin_port = htons(port);
    sinaddr.sin_addr.s_addr = htonl(INADDR_ANY);

    if (bind(sock, (struct sockaddr *)&sinaddr, sizeof(sinaddr)) == -1)
        exit_sys("bind");

    FD_SET(sock, &rset);
    maxfd = sock;

    if (listen(sock, 8) == -1)
        exit_sys("listen");

```

```

printf("Waiting for connection...\n");

for (;;) {
    tset = rset;
    if (select(maxfd + 1, &tset, NULL, NULL, NULL) == -1)
        exit_sys("select");

    if (FD_ISSET(sock, &tset)) {
        sinaddr_len = sizeof(sinaddr_client);
        if ((sock_client = accept(sock, (struct sockaddr
            *)&sinaddr_client, &sinaddr_len)) == -1)
            exit_sys("accept");

        clients[nclients].fd = sock_client;
        clients[nclients].index = 0;
        clients[nclients].left = BUFFER_SIZE;
        ++nclients;

        inet_ntop(AF_INET, &sinaddr_client.sin_addr, ntopbuf,
            INET_ADDRSTRLEN);
        printf("Connected: %s : %u\n", ntopbuf,
            (unsigned)ntohs(sinaddr_client.sin_port));
        FD_SET(sock_client, &rset);
        if (sock_client > maxfd)
            maxfd = sock_client;
    }

    for (i = 0; i < nclients; ++i)
        if (FD_ISSET(clients[i].fd, &tset)) {
            if ((result = recv(clients[i].fd, clients[i].buf +
                clients[i].index, clients[i].left, 0)) == -1)
                exit_sys("recv");
            if (result > 0) {
                clients[i].index += result;
                clients[i].left -= result;
                if (clients[i].left == 0) {
                    clients[i].buf[clients[i].index] = '\0';
                    sinaddr_len = sizeof(sinaddr_client);
                    if (getpeername(clients[i].fd, (struct sockaddr
                        *)&sinaddr_client, &sinaddr_len) == -1)
                        exit_sys("getpeername");

                    inet_ntop(AF_INET, &sinaddr_client.sin_addr,
                        ntopbuf, INET_ADDRSTRLEN);
                    printf("%ld bytes received from %s (%u): %s\n",
                        (long)clients[i].index, ntopbuf,
                        (unsigned)ntohs(sinaddr_client.sin_port),
                        clients[i].buf);

                    clients[i].index = 0;
                    clients[i].left = BUFFER_SIZE;
                }
            }
        }
    else {

```



```

        shutdown(clients[i].fd, SHUT_RDWR);
        close(clients[i].fd);
        FD_CLR(clients[i].fd, &rset);
        clients[i] = clients[nclients - 1];
        --nclients;
    }
}

close(sock);

return 0;
}

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

    exit(EXIT_FAILURE);
}

/* client.c */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>

#define BUFFER_SIZE      1024

void exit_sys(const char *msg);

int main(int argc, char *argv[])
{
    int sock;
    struct addrinfo *ai, *ri;
    struct addrinfo hints = {0};
    char buf[BUFFER_SIZE];
    char *str;
    int result;

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

    if ((sock = socket(AF_INET, SOCK_STREAM, 0)) == -1)
        exit_sys("socket");

    hints.ai_family = AF_INET;
    hints.ai_socktype = SOCK_STREAM;

```

```

if ((result = getaddrinfo(argv[1], argv[2], &hints, &ai)) != 0) {
    fprintf(stderr, "getaddrinfo: %s\n", gai_strerror(result));
    exit(EXIT_FAILURE);
}

for (ri = ai; ri != NULL; ri = ri->ai_next)
    if (connect(sock, ri->ai_addr, ri->ai_addrlen) != -1)
        break;

if (ri == NULL)
    exit_sys("connect");

freeaddrinfo(ai);

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

for (;;) {
    printf("Yazı giriniz:");
    fgets(buf, BUFFER_SIZE, stdin);
    if ((str = strchr(buf, '\n')) != NULL)
        *str = '\0';
    if (!strcmp(buf, "quit"))
        break;

    if ((send(sock, buf, strlen(buf), 0)) == -1)
        exit_sys("send");
}

shutdown(sock, SHUT_RDWR);
close(sock);

return 0;
}

```

```

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

    exit(EXIT_FAILURE);
}

```

/

*-----

Bir tarafın diğer tarafa bir dosyanın içeriğini göndermesi (dosya aktarımı) sık rastlanan durumlardandır. Bunun şüphesiz klasik gerçekleştirim

biçimi dosyadan blok blok okuma yapıp onları soketten yollamak ve benzer biçimde bir döngü içerisinde recv fonksiyonuyla alarak dosyaya yazmak olabilir.

Linux sistemlerinde (başka bazı sistemlerde de var) bu işlem için sendfile isimli bir sistem fonksiyonu bulundurulmuştur. Fonksiyonu prototipi şöyledir:

```
ssize_t sendfile(int out_fd, int in_fd, off_t *offset, size_t count);
```

Fonksiyonun birinci parametresi soket betimleyicisini almaktadır. İkinci parametre normal bir dosya betimleyicisidir. Fonksiyonun üçüncü parametresi

aktarımın başlatılacağı offset değerini belirtmektedir. Bu parametre NULL geçilirse aktarım ikinci parametreyle belirtilen dosyanın başından itibaren yapılır.

Fonksiyon sonlandıktan sonra dosya göstericisinin yeni değeri bu adrese yerleştirilmektedir. Bu parametre NULL da geçilebilir. Bu durumda kopyalama ikinci parametreyle

belirtilen dosyanın başından itibaren yapılır. Eğer bu offset parametresi NULL geçilmediyse ikinci parametreyle belirtilen dosyaya ilişkin dosya göstericisinin

değeri değiştirilmez ama bu parametre NULL geçilmişse bu dosyaya ilişkin dosya göstericisinin değeri değiştirilir. Son parametre kaç byte transfer edileceğidir.

Fonksiyon başarı durumunda aktarılan byte sayına başarısızlık durumunda -1 değerine geri döner. Bu fonksiyon sayesinde hiç user düzeyinde tampon kullanmadan tek bir hamlede

bir dosyanın içeriğini soketten karşı tarafa gönderebiliriz. Fonksiyon başarı durumunda tampona yazılmış olan byte sayısı ile başarısızlık durumunda -1 değeri ile geri döner

sendfile fonksiyonun POSIX standartlarında bulunmadığına Linux'a özgü olduğuna dikkat ediniz.

sendfile sistem fonksiyonu kernel 2.6.33'e kadar yalnızca dosyadan sokete kopyalama yapmak için kullanılıyordu. Bu versiyondan sonra fonksiyon herhangi iki betimleyici

arasında çalışabilir hale getirilmiştir. copy_file_range fonksiyonu ise halen yalnızca normal dosyalar için çalışabilmektedir.

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

```
/
```

```
*-----
```

TCP/IP client server uygulamaların önemli bir bölümünde mesajlaşma text tabanlı yapılmaktadır. Yani client server'a isteğini bir yazı göndererek iletir, server da client'a yanıtı bir yazı biçiminde verir. Gerçekten de IP ailesinin TELNET, SSH, HTTP, POP3, IMAP gibi protokolleri hep yazısal işlem yapmaktadır. Bir satır bilginin gönderilmesinde hiçbir problem yoktur. Ancak bir satırlık bilginin etkin bir biçimde alınabilmesi için bir algoritma kullanmak gerekir. Çünkü karakterlerin tek tek read ya da recv ile okunması yavaş bir seçenektir. Aşağıda her çağrıldığında soketten bir satır okuyan örnek bir fonksiyon verilmiştir.

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

```
int sock_readline(int sock, char *buf, size_t len)
{
    char *bufx = buf;
```

```

static char *bp;
static ssize_t count = 0;
static char b[2048];

if (len <= 2)
    return -1;

while (--len > 0) {
    if (--count <= 0) {
        if ((count = recv(sock, b, sizeof(b), 0)) == -1)
            return -1;
        if (count == 0)
            return 0;
        bp = b;
    }
    *buf++ = *bp++;
    if (buf[-1] == '\n' && buf[-2] == '\r') {
        *buf = '\0';
        break;
    }
}

return buf - bufx;
}

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

```

IP ailesinin uygulama katmanındaki Telnet, SSH, HTTP, POP3, SMTP gibi protokoller yukarıda belirtildiği gibi hep yazısal işlem yapmaktadır. Örneğin POP3 (Post Office Version 3) protokolü kabaca şöyledir:

- 1) Client server'a (POP3 server) 110 numaralı porttan (POP3 için well known port) bağlanır. Bağlanma başarılı ise Server Client'a "+OK <mesaj>\r\n" biçiminde bir satır yollar.
- 2) Bundan sonra client "USER <user name>\r\n" komutu ile mail adresini iletir. (Örneğin test@csystem.org). Server eğer bu işlem başarılıysa "+OK\r\n" yazısını gönderir.
- 3) Client bundan sonra "PASS <password>\r\n" yazısını server'a göndererek password'ü iletmış olur. Eğer password başarılı ise server "+OK Logged in.\r\n" yazısını balarısız ise "-ERR [AUTH] Authentication failed.\r\n" yazısını göndermektedir.
- 4) Şimdi client posta kutusundaki postalar hakkında "LIST\r\n" komutu ile bilgileri elde eder. Bu komuta karşı server birden fazla satırdan oluşan şöyle bir vermektedir:

```

1 8088
2 1568
3 3962
.

```

Son satırın '.' ile bittiğine dikkat ediniz.

5) Client herhangi bir postayı "RETR <no>\r\n" ile elde edebilir. Buna karşı server e-postanın içeriğini birden fazla satır biçiminde yollayacaktır.
E-posta düz text ya da genellikle HTML olarak gönderilmiş olabilir. e-posta'nın içeriğinde MIME Type zaten belirtilebilmektedir. Bu konuda POP3 ve MIME dokümanlarına başvurabilirsiniz.

Bu protokolde server'ın olumlu yanıtlarında "+OK" olumsuz yanıtlarında "-ERR" biçiminde satırlar gönderilmektedir.

6) Nihayet client işini bitirdikten sonra server'a "QUIT\t\n" komutunu gönderir. Server'da soketi kapatır. Client da sketin kapatılmış olduğunu anlar ve işini sonlandırır.

Aşağıda POP3 işlemi yapan TELNET benzeri bir program örneği verilmiştir. Programda bir thread sürekli server'dan gelen satırları blokeli biçimde okumaktadır.

```
-----*/  
  
/* pop3client.c */  
  
#include <stdio.h>  
#include <stdlib.h>  
#include <string.h>  
#include <unistd.h>  
#include <errno.h>  
#include <pthread.h>  
#include <signal.h>  
#include <sys/socket.h>  
#include <netinet/in.h>  
#include <arpa/inet.h>  
#include <netdb.h>  
  
#define BUFFER_SIZE      1024  
  
void *thread_proc(void *param);  
int sock_readline(int sock, char *buf, size_t len);  
void sigusr1_handler(int sno);  
void exit_sys(const char *msg);  
  
typedef struct {  
    int sock;  
    pthread_t tid;  
} THREAD_PARAM;  
  
int main(int argc, char *argv[])  
{  
    int sock;  
    struct addrinfo *ai, *ri;  
    struct addrinfo hints = {0};
```

```

char buf[BUFFER_SIZE + 1];
char *str;
int result;
pthread_t tid;
struct sigaction sa;
THREAD_PARAM *tp;

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

sa.sa_handler = sigusr1_handler;
sa.sa_flags = 0;
sigemptyset(&sa.sa_mask);

if (sigaction(SIGUSR1, &sa, NULL) == -1)
    exit_sys("sigaction");

if ((sock = socket(AF_INET, SOCK_STREAM, 0)) == -1)
    exit_sys("socket");

hints.ai_family = AF_INET;
hints.ai_socktype = SOCK_STREAM;

if ((result = getaddrinfo(argv[1], argv[2], &hints, &ai)) != 0) {
    fprintf(stderr, "getaddrinfo: %s\n", gai_strerror(result));
    exit(EXIT_FAILURE);
}

for (ri = ai; ri != NULL; ri = ri->ai_next)
    if (connect(sock, ri->ai_addr, ri->ai_addrlen) != -1)
        break;

if (ri == NULL)
    exit_sys("connect");

freeaddrinfo(ai);

if ((tp = malloc(sizeof(THREAD_PARAM))) == NULL)
    exit_sys("malloc");
tp->sock = sock;
tp->tid = pthread_self();
if ((result = pthread_create(&tid, NULL, thread_proc, tp) != 0)) {
    fprintf(stderr, "pthread_create: %s\n", strerror(result));
    exit(EXIT_FAILURE);
}

for (;;) {
    if (fgets(buf, BUFFER_SIZE, stdin) == NULL)
        break;
    if (errno == EINTR)
        break;
    if ((str = strchr(buf, '\n')) != NULL)
        *str = '\0';
}

```

```

        if (!strcmp(buf, "quit"))
            break;

        strcat(buf, "\r\n");

        if (send(sock, buf, strlen(buf), 0) == -1)
            exit_sys("send");
    }

    shutdown(sock, SHUT_RDWR);
    close(sock);

    return 0;
}

void *thread_proc(void *param)
{
    char buf[BUFFER_SIZE + 1];
    THREAD_PARAM *tp = (THREAD_PARAM *)param;
    int result;

    for (;;) {
        if ((result = sock_readline(tp->sock, buf, BUFFER_SIZE)) == -1)
            exit_sys("sock_readline");
        if (!result)
            break;

        printf("%s", buf);
    }

    if ((result = pthread_kill(tp->tid, SIGUSR1)) != 0) {
        fprintf(stderr, "pthread_create: %s\n", strerror(result));
        exit(EXIT_FAILURE);
    }

    return NULL;
}

int sock_readline(int sock, char *buf, size_t len)
{
    char *bufx = buf;
    static char *bp;
    static ssize_t count = 0;
    static char b[2048];

    if (len <= 2)
        return -1;

    while (--len > 0) {
        if (--count <= 0) {
            if ((count = recv(sock, b, sizeof(b), 0)) == -1)
                return -1;
            if (count == 0)
                return 0;
            bp = b;
        }
        *bufx++ = *bp++;
    }

```

```

    }
    *buf++ = *bp++;
    if (buf[-1] == '\n' && buf[-2] == '\r') {
        *buf = '\0';
        break;
    }
}

return buf - bufx;
}

```

```

void sigusr1_handler(int sno)
{
}

```

```

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

    exit(EXIT_FAILURE);
}

```

/

Aslında POP3'te olduğu gibi pek çok protokol her komuta en az 1 satır yanıt vermektedir. Eğer server komuta karşı bir satırdan daha fazla bir yanıt veriyorsa genellikle bu ilk satırda ya da son satırda açığa vurulmaktadır. Örneğin POP3 protokolünde LIST komutu birden fazla satırlık bilgi gönderebilmektedir. Ancak son satır özellikle protokolda '.' biçiminde tutulmuştur. Böylece client programı yazan programcı bu '.' satırını gördüğünde satır okuma işlemini bitirilebilmektedir. Benzer biçimde POP3 protokolünde RETR komutunun birinci satırında e-postanın kaç byte'lık bir bilgi içerdiği belirtilmektedir. Böylece POP3 client programını yazan programcı bu sayıyı alarak tam o kadar byte okuyabilir. Diğer diğer protokollerdeki mantıkta aslında çok benzerdir.

Aşağıdaki program yukarıdaki programın thread'siz dolayısıyla daha uygun bir versiyondur. Bu programda POP3 komutlarındaki yanıtta da bakılarak porttan uygun miktarda okumalar yapılmıştır.

-----*/

```

#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#include <string.h>
#include <unistd.h>
#include <errno.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>

```



```

#define BUFFER_SIZE          1024

void proc_list(int sock);
void proc_retr(int sock);
void getcmd(const char *buf, char *cmd);
int sock_readline(int sock, char *buf, size_t len);
void exit_sys(const char *msg);

int main(int argc, char *argv[])
{
    int sock;
    struct addrinfo *ai, *ri;
    struct addrinfo hints = {0};
    char bufsend[BUFFER_SIZE + 1];
    char bufrecv[BUFFER_SIZE + 1];
    char *str;
    ssize_t result;
    char cmd[BUFFER_SIZE];

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

    if ((sock = socket(AF_INET, SOCK_STREAM, 0)) == -1)
        exit_sys("socket");

    hints.ai_family = AF_INET;
    hints.ai_socktype = SOCK_STREAM;

    if ((result = getaddrinfo(argv[1], argv[2], &hints, &ai)) != 0) {
        fprintf(stderr, "getaddrinfo: %s\n", gai_strerror(result));
        exit(EXIT_FAILURE);
    }

    for (ri = ai; ri != NULL; ri = ri->ai_next)
        if (connect(sock, ri->ai_addr, ri->ai_addrlen) != -1)
            break;

    if (ri == NULL)
        exit_sys("connect");

    freeaddrinfo(ai);

    if (sock_readline(sock, bufrecv, BUFFER_SIZE) == -1)
        exit_sys("sock_readline");
    printf("%s", bufrecv);

    for (;;) {
        printf(">");
        fflush(stdout);
        if (fgets(bufsend, BUFFER_SIZE, stdin) == NULL)
            break;
        if ((str = strchr(bufsend, '\n')) != NULL)

```

```

        *str = '\\0';

    strcat(bufsend, "\\r\\n");

    if (send(sock, bufsend, strlen(bufsend), 0) == -1)
        exit_sys("send");

    getcmd(bufsend, cmd);
    if (!strcmp(cmd, "LIST"))
        proc_list(sock);
    else if (!strcmp(cmd, "RETR"))
        proc_retr(sock);
    else {
        if ((result = sock_readline(sock, bufrecv, BUFFER_SIZE)) == -1)
            exit_sys("sock_readline");
        if (result == 0)
            break;
        printf("%s", bufrecv);
        if (!strcmp(cmd, "QUIT"))
            break;
    }
}

shutdown(sock, SHUT_RDWR);
close(sock);

return 0;
}

void proc_list(int sock)
{
    ssize_t result;
    char bufrecv[BUFFER_SIZE];

    do {
        if ((result = sock_readline(sock, bufrecv, BUFFER_SIZE)) == -1)
            exit_sys("sock_readline");
        if (result == 0)
            break;
        printf("%s", bufrecv);
    } while (*bufrecv != '.');
}

void proc_retr(int sock)
{
    ssize_t result;
    char bufrecv[BUFFER_SIZE + 1];
    ssize_t n;
    int i, ch;

    for (i = 0;; ++i) {
        if ((result = recv(sock, &ch, 1, 0)) == -1)
            exit_sys("sock_readline");
        if (result == 0)
            return;
    }
}

```

```

        if ((bufrecv[i] = ch) == '\n')
            break;
    }
    bufrecv[i] = '\0';

    printf("%s", bufrecv);
    n = (ssize_t)strtol(bufrecv + 3, NULL, 10);
    printf("%ld\n", n);

    while (n > 0) {
        if ((result = recv(sock, bufrecv, BUFFER_SIZE, 0)) == -1)
            exit_sys("recv");
        if (result == 0)
            break;
        bufrecv[result] = '\0';
        printf("%s", bufrecv);
        fflush(stdout);
        n -= result;
    }
}

void getcmd(const char *buf, char *cmd)
{
    int i;

    for (i = 0; buf[i] != '\0' && !isspace(buf[i]); ++i)
        cmd[i] = buf[i];
    cmd[i] = '\0';
}

int sock_readline(int sock, char *buf, size_t len)
{
    char *bufx = buf;
    static char *bp;
    static ssize_t count = 0;
    static char b[2048];

    if (len <= 2)
        return -1;

    while (--len > 0) {
        if (--count <= 0) {
            if ((count = recv(sock, b, sizeof(b), 0)) == -1)
                return -1;
            if (count == 0)
                return 0;
            bp = b;
        }
        *buf++ = *bp++;
        if (buf[-1] == '\n' && buf[-2] == '\r') {
            *buf = '\0';
            break;
        }
    }
}

```

```

    return buf - bufx;
}

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

    exit(EXIT_FAILURE);
}

```

```

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

```

Soketler yaratıldıktan sonra onların bazı özellikleri setsockopt isimli fonksiyonla değiştirilebilir ve getsockopt isimli fonksiyonla da alınabilir. setsockopt fonksiyonunun prototipi şöyledir:

```

int setsockopt(int socket, int level, int option_name, const void
    *option_value, socklen_t option_len);

```

Fonksiyonun birinci parametresi özelliği değiştirilecek soketi belirtir. ikinci parametresi değişimin hangi düzeyde yapılacağını belirten bir sembolik sabit olarak girilir. Soket düzeyi için her zaman SOL_SOCKET girilmelidir. Üçüncü parametre hangi özelliğin değiştirileceğini belirtmektedir.

Dördüncü parametre değiştirilecek özelliğin değerinin bulunduğu nesnenin adresini almaktadır. Son parametre dördüncü parametredeki nesnenin uzunluğunu belirtmektedir.

Fonksiyon başarı durumunda 0, başarısızlık durumunda -1 değerine geri döner.

Soket seçeneğini elde etmek için de getsockopt fonksiyonu kullanılmaktadır:

```

int getsockopt(int socket, int level, int option_name, void *restrict
    option_value, socklen_t *restrict option_len);

```

Parametreler setsockopt'ta olduğu gibidir. Yalnızca dördüncü parametrenin yönü değişiktir.

Tipik soket seçenekleri (üçüncü parametre) şunlardan biri olabilir:

```

SO_ACCEPTCONN
SO_BROADCAST
SO_DEBUG
SO_DONTROUTE
SO_ERROR
SO_KEEPALIVE
SO_LINGER
SO_OOBINLINE
SO_RCVBUF
SO_RCVLOWAT
SO_RCVTIMEO
SO_REUSEADDR
SO_SNDBUF

```

SO_SNDLOWAT
SO_SNDTIMEO
SO_TYPE

Burada bizim için şimdilik önemli olan birkaç seçenek vardır:
SO_BROADCAST, SO_OOBLINE, SO_SNDBUF, SO_RECVBUF, SO_REUSEADDR.

Örneğin:

```
int buflen;  
int optsize = sizeof(int)  
...  
if (getsockopt(sock_client, SOL_SOCKET, SO_RECVBUF, &buflen, &optsize) ==  
    -1)  
    exit_sys("getsockopt");
```

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

SO_REUSEADDR seçeneği belli bir port için bind işlemi yapmış bir server'ın sonlanması sonucunda bu server'ın yeniden çalıştırılıp aynı portu bind edebilmesi için kullanılmaktadır. Bir portu bind eden server, bir client ile bağlandıktan sonra çökerse, ya da her hangi bir biçimde sonlanırsa işletim sistemleri o portun yeniden belli bir süre bind edilmesini engellemektedir. Bunun nedeni eski çalışan server ile

yeni çalışacak olan server'ın göndereceği ve alacağı paketlerin karışabilme olasılığıdır. Eski paketlerin ağda maksimum bir geçerlilik süresi vardır.

İşletim sistemi de bunun iki katı kadar bir süre (2 dakika civarı, neden iki katı olduğu protokolün aşağı seviyeli çalışması ile ilgilidir) bu protun yeniden

bind edilmesini engellemektedir. İşte eğer bu soket seçeneği kullanılırsa artık sonlanan ya da çöken bir server hemen yeniden çalıştırıldığında bind işlemi sırasında "Address already in use" biçiminde bir hata ile karşılaşılacaktır. Örneğin:

```
int sockapt = 1;  
...  
if (setsockopt(sock, SOL_SOCKET, SO_REUSEADDR, &sockapt,  
    sizeof(sockapt)) == -1)  
    exit_sys("setsockopt");
```

SO_REUSEADDR seçeneğini set etmek için int bir nesne alıp onun içerisine sıfır dışı bir değer yerleştirip, onun adresini setsockopt fonksiyonunun dördüncü

parametresine girmek gerekir. Bu nesneye 0 girip fonksiyonu çağırırsak bu özelliği kapatmış oluruz. Fonksiyonun dördüncü parametresine adresi girilecek nesnenin

seçeneğe göre değişebileceğine dikkat ediniz.

SO_REUSEADDR bayrağı daha önce başka bir program tarafındna bind edilmiş soketin ikinci kez diğer bir program tarafından bind edilmesi için kullanılmamaktadır.

Eğer böyle bir ihtiyaç varsa (nadiren olabilir) Linux'ta (fakat POSIX'te değil) SO_REUSEPORT soket seçeneği kullanılmalıdır. Bu soket seçeneği benzer biçimde

Windows sistemlerinde SO_EXCLUSIVEADDRUSE biçimindedir.

Aşağıdaki server programını client ile bağlandıktan sonra Ctrl+C ile sonlandırınız. Sonra yeniden çalıştırmaya çalışınız.

SO_REUSEADDR seçeneği kullanıldığından dolayı bir sorun ile karşılaşılacaktır.

Daha sonra server programdan o kısmı silerek yeniden denemeyi yapınız.

```
-----*/
/* server.c */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

#define BUFFER_SIZE      1024

void exit_sys(const char *msg);

int main(int argc, char *argv[])
{
    int sock, sock_client;
    struct sockaddr_in sinaddr, sinaddr_client;
    socklen_t sinaddr_len;
    char ntopbuf[INET_ADDRSTRLEN];
    in_port_t port;
    ssize_t result;
    char buf[BUFFER_SIZE + 1];
    int sockopt = 1;

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

    port = (in_port_t)strtoul(argv[1], NULL, 10);

    if ((sock = socket(AF_INET, SOCK_STREAM, 0)) == -1)
        exit_sys("socket");

    if (setsockopt(sock, SOL_SOCKET, SO_REUSEADDR, &sockopt,
        sizeof(sockopt)) == -1)
        exit_sys("setsockopt");
```

```

sinaddr.sin_family = AF_INET;
sinaddr.sin_port = htons(port);
sinaddr.sin_addr.s_addr = htonl(INADDR_ANY);

if (bind(sock, (struct sockaddr *)&sinaddr, sizeof(sinaddr)) == -1)
    exit_sys("bind");

if (listen(sock, 8) == -1)
    exit_sys("listen");

printf("Waiting for connection...\n");

sinaddr_len = sizeof(sinaddr_client);
if ((sock_client = accept(sock, (struct sockaddr *)&sinaddr_client,
    &sinaddr_len)) == -1)
    exit_sys("accept");

printf("Connected: %s : %u\n", inet_ntop(AF_INET, &sinaddr_client,
    ntopbuf, INET_ADDRSTRLEN), (unsigned)ntohs(sinaddr_client.sin_port));

for (;;) {
    if ((result = recv(sock_client, buf, BUFFER_SIZE, 0)) == -1)
        exit_sys("recv");
    if (result == 0)
        break;
    buf[result] = '\0';
    if (!strcmp(buf, "quit"))
        break;
    printf("%ld bytes received from %s (%u): %s\n", (long)result,
        ntopbuf, (unsigned)ntohs(sinaddr_client.sin_port), buf);
}

shutdown(sock_client, SHUT_RDWR);
close(sock_client);
close(sock);

return 0;
}

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

    exit(EXIT_FAILURE);
}

/* client.c */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

```

```

#include <netdb.h>

#define BUFFER_SIZE      1024

void exit_sys(const char *msg);

int main(int argc, char *argv[])
{
    int sock;
    struct addrinfo *ai, *ri;
    struct addrinfo hints = {0};
    char buf[BUFFER_SIZE];
    char *str;
    int result;

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

    if ((sock = socket(AF_INET, SOCK_STREAM, 0)) == -1)
        exit_sys("socket");

    hints.ai_family = AF_INET;
    hints.ai_socktype = SOCK_STREAM;

    if ((result = getaddrinfo(argv[1], argv[2], &hints, &ai)) != 0) {
        fprintf(stderr, "getaddrinfo: %s\n", gai_strerror(result));
        exit(EXIT_FAILURE);
    }

    for (ri = ai; ri != NULL; ri = ri->ai_next)
        if (connect(sock, ri->ai_addr, ri->ai_addrlen) != -1)
            break;

    if (ri == NULL)
        exit_sys("connect");

    freeaddrinfo(ai);

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

    for (;;) {
        printf("Yazı giriniz:");
        fgets(buf, BUFFER_SIZE, stdin);
        if ((str = strchr(buf, '\n')) != NULL)
            *str = '\0';
        if (!strcmp(buf, "quit"))
            break;

        if ((send(sock, buf, strlen(buf), 0)) == -1)
            exit_sys("send");
    }

    shutdown(sock, SHUT_RDWR);
}

```



```

    close(sock);

    return 0;
}

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

    exit(EXIT_FAILURE);
}

```

```

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

```

OOB (Out-Of-Band) Data bazı stream protokollerinde olan bir özelliktir. Örneğin TCP protokolü bunu 1 byte olarak desteklemektedir. OOB Verisine TCP'de "Acil (Urgent)" veri de denilmektedir. Bunun amacı OOB verisinin normal stream sırasında değil daha önce gönderilenlerin -eğer hedef host'ta henüz ele alınmamışlarsa- önünde ele alınabilmesidir. Yani biz TCP'de birtakım verileri gönderdikten sonra OOB verisini gönderirsek bu veri önce göndermiş olduklarımızdan daha önde işleme sokulabilir. Böylece uygulamalarda önce gönderilen birtakım işlemlerin iptal edilmesi gibi gerekçelerle kullanılabilmektedir.

OOB verisini gönderebilmek için send fonksiyonunun flags parametresine MSG_OOB girmek gerekir. Tabii TCP yalnızca 1 byte uzunluğunda OOB verisinin gönderilmesine izin vermektedir. Bu durumda eğer send ile birden fazla byte MSG_OOB bayrağı ile gönderilmek istenirse gönderilenlerin yalnızca son byte'ı OOB olarak gönderilir.

Normal olarak OOB verisi recv fonksiyonunda MSG_OOB bayrağı ile alınır. Ancak bu bayrak kullanılarak recv çağrıldığında eğer bir OOB verisi sırada yoksa recv başarısız olmaktadır. recv fonksiyonun MSG_OOB bayraklı çağrısında başarılı olabilmesi için o anda bir OOB verisinin gelmiş olması gerekir.

Pekiyi OOB verisinin geldiğini nasıl anlarız? İşte tipik yöntem SIGURG sinyalinin kullanılmasıdır. Çünkü sokete bir OOB verisi geldiğinde işletim sistemi

SIGURG sinyali oluşturabilmektedir. Bu sinyalin default durumu IGNORE biçimindedir. (Yani eğer set edilmemişse sanki sinyal oluşmamış gibi davranılır.)

Fakat default olarak OOB verisi geldiğinde sinyal oluşmamaktadır. Bunu mümkün hale getirmek için socket üzerinde fcntl fonksiyonu ile F_SETOWN komut kodunu

kullanarak set işlemi yapmak gerekir. fcntl fonksiyonun son parametresi bu durumda sinyalin gönderileceği prosesin id değeri olarak girilmelidir.

Eğer bu parametre negatif bir proses grup id'si olarak girilirse bu durumda işletim sistemi tüm proses grubuna sinyal göndermektedir. Örneğin:

```
if (fcntl(sock_client, F_SETOWN, getpid()) == -1)
    exit_sys("fcntl");
```

Aşağıdaki server programda bir OOB verisi geldiğinde SIGURG sinyali oluşturulmaktadır. Bu sinyalin içerisinde recv fonksiyonu MSG_OOB bayrağı ile çağrılmıştır. OOB verisinin okunması için MSG_OOB bayrağı gerekir, ancak OOB verisinin olmadığı bir durumda bu bayrak kullanılırsa recv başarısız olur.

0 halde SIGURG sinyali geldiğinde recv MSG_OOB ile çağrılmalıdır. Bu durumda TCP'de her zaman yalnızca 1 byte okunabilmektedir. Ayrıca server programda SIGURG sinyali set edilirken sigaction yapısının flags parametresinin SA_RESTART biçiminde egeçildiğine dikkat ediniz. Bu recv üzerinde beklerken oluşabilecek SIGURG sinyalinden sonra recv'in otomatik yeniden başlatılması için kullanılmıştır.

Aşağıdaki client programda başı "u" harfi ile başlayan yazılar MSG_OOB bayrağı kullanılarak gönderilmiştir. Bu durumda yalnızca son byte'ın OOB verisi olacağını anımsayınız. OOB verileri her zaman mümkünse diğer verilerden önce ele alınmaktadır.

-----*/

```
/* oobserver.c */
```

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>
#include <unistd.h>
#include <signal.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

#define BUFFER_SIZE      1024

void sigurg_handler(int sno);
void exit_sys(const char *msg);

int sock_client;

int main(int argc, char *argv[])
{
    int sock;
    struct sockaddr_in sinaddr, sinaddr_client;
    socklen_t sinaddr_len;
    char ntopbuf[INET_ADDRSTRLEN];
    in_port_t port;
    ssize_t result;
    char buf[BUFFER_SIZE + 1];
```

```

struct sigaction sa;

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

sa.sa_handler = sigurg_handler;
sa.sa_flags = SA_RESTART;
sigemptyset(&sa.sa_mask);

if (sigaction(SIGURG, &sa, NULL) == -1)
    exit_sys("sigaction");

port = (in_port_t)strtoul(argv[1], NULL, 10);

if ((sock = socket(AF_INET, SOCK_STREAM, 0)) == -1)
    exit_sys("socket");

sinaddr.sin_family = AF_INET;
sinaddr.sin_port = htons(port);
sinaddr.sin_addr.s_addr = htonl(INADDR_ANY);

if (bind(sock, (struct sockaddr *)&sinaddr, sizeof(sinaddr)) == -1)
    exit_sys("bind");

if (listen(sock, 8) == -1)
    exit_sys("listen");

printf("Waiting for connection...\n");

sinaddr_len = sizeof(sinaddr_client);
if ((sock_client = accept(sock, (struct sockaddr *)&sinaddr_client,
    &sinaddr_len)) == -1)
    exit_sys("accept");

if (fcntl(sock_client, F_SETOWN, getpid()) == -1)
    exit_sys("fcntl");

printf("Connected: %s : %u\n", inet_ntop(AF_INET, &sinaddr_client,
    ntopbuf, INET_ADDRSTRLEN), (unsigned)ntohs(sinaddr_client.sin_port));

for (;;) {
    if ((result = recv(sock_client, buf, BUFFER_SIZE, 0)) == -1)
        exit_sys("recv");
    if (result == 0)
        break;
    buf[result] = '\0';
    if (!strcmp(buf, "quit"))
        break;
    printf("%ld bytes received from %s (%u): %s\n", (long)result,
        ntopbuf, (unsigned)ntohs(sinaddr_client.sin_port), buf);
}

shutdown(sock_client, SHUT_RDWR);

```

```

        close(sock_client);
        close(sock);

        return 0;
    }

void sigurg_handler(int sno)
{
    char oob;

    if (recv(sock_client, &oob, 1, MSG_OOB) == -1)
        exit_sys("recv");

    printf("OOB Data received: %c\n", oob);
}

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

    exit(EXIT_FAILURE);
}

/* oobclient.c */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>

#define BUFFER_SIZE      1024

void exit_sys(const char *msg);

int main(int argc, char *argv[])
{
    int sock;
    struct addrinfo *ai, *ri;
    struct addrinfo hints = {0};
    char buf[BUFFER_SIZE];
    char *str;
    int result;

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

    if ((sock = socket(AF_INET, SOCK_STREAM, 0)) == -1)
        exit_sys("socket");

```

```

hints.ai_family = AF_INET;
hints.ai_socktype = SOCK_STREAM;

if ((result = getaddrinfo(argv[1], argv[2], &hints, &ai)) != 0) {
    fprintf(stderr, "getaddrinfo: %s\n", gai_strerror(result));
    exit(EXIT_FAILURE);
}

for (ri = ai; ri != NULL; ri = ri->ai_next)
    if (connect(sock, ri->ai_addr, ri->ai_addrlen) != -1)
        break;

if (ri == NULL)
    exit_sys("connect");

freeaddrinfo(ai);

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

for (;;) {
    printf("Yazı giriniz:");
    fgets(buf, BUFFER_SIZE, stdin);
    if ((str = strchr(buf, '\n')) != NULL)
        *str = '\0';
    if (!strcmp(buf, "quit"))
        break;

    if ((send(sock, buf, strlen(buf), buf[0] == 'u' ? MSG_OOB : 0)) ==
        -1)
        exit_sys("send");
}

shutdown(sock, SHUT_RDWR);
close(sock);

return 0;
}

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

    exit(EXIT_FAILURE);
}

```

*-----

OOB verisi geldiğinde select fonksiyonu "sokette bir işlem oluştu" biçiminde geri dönmektedir. Ancak oluşan işlem "exceptional" kabul edilmektedir. Yani select fonksiyonunda dördüncü parametredeki bayraklara set işlemi yapmamız gerekir. Benzer biçimde OOB geldiğinde poll fonksiyonu da POLLPRI bayrağını set etmektedir.

Aşağıdaki örnekte OOB verisi select fonksiyonu ile tespit edilmiştir.

```

-----*/

/* oobserver.c */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <sys/select.h>

#define BUFFER_SIZE      1024

void exit_sys(const char *msg);

int main(int argc, char *argv[])
{
    int sock, sock_client;
    struct sockaddr_in sinaddr, sinaddr_client;
    socklen_t sinaddr_len;
    char ntopbuf[INET_ADDRSTRLEN];
    in_port_t port;
    ssize_t result;
    char buf[BUFFER_SIZE + 1];
    fd_set rset, trset, eset, teset;
    char oob;

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

    port = (in_port_t)strtoul(argv[1], NULL, 10);

    if ((sock = socket(AF_INET, SOCK_STREAM, 0)) == -1)
        exit_sys("socket");

    sinaddr.sin_family = AF_INET;
    sinaddr.sin_port = htons(port);
    sinaddr.sin_addr.s_addr = htonl(INADDR_ANY);

    if (bind(sock, (struct sockaddr *)&sinaddr, sizeof(sinaddr)) == -1)
        exit_sys("bind");

    if (listen(sock, 8) == -1)
        exit_sys("listen");

    printf("Waiting for connection...\n");

    sinaddr_len = sizeof(sinaddr_client);

```

```

if ((sock_client = accept(sock, (struct sockaddr *)&sinaddr_client,
    &sinaddr_len)) == -1)
    exit_sys("accept");

printf("Connected: %s : %u\n", inet_ntop(AF_INET, &sinaddr_client,
    ntopbuf, INET_ADDRSTRLEN), (unsigned)ntohs(sinaddr_client.sin_port));

FD_ZERO(&eset);
FD_ZERO(&teset);

FD_SET(sock_client, &rset);
FD_SET(sock_client, &eset);

for (;;) {
    trset = rset;
    teset = eset;
    if (select(sock_client + 1, &trset, NULL, &teset, NULL) == -1)
        exit_sys("select");

    if (FD_ISSET(sock_client, &trset)) {
        if ((result = recv(sock_client, buf, BUFFER_SIZE, 0)) == -1)
            exit_sys("recv");
        if (result == 0)
            break;
        buf[result] = '\0';
        if (!strcmp(buf, "quit"))
            break;
        printf("%ld bytes received from %s (%u): %s\n", (long)result,
            ntopbuf, (unsigned)ntohs(sinaddr_client.sin_port), buf);
    }
    if (FD_ISSET(sock_client, &teset)) {
        if ((result = recv(sock_client, &oob, 1, MSG_OOB)) == -1)
            exit_sys("recv");
        printf("OOB Data received: %c\n", oob);
    }
}

shutdown(sock_client, SHUT_RDWR);
close(sock_client);
close(sock);

return 0;
}

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

    exit(EXIT_FAILURE);
}

/* oobclient.c */

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

```

```

#include <string.h>
#include <unistd.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>

#define BUFFER_SIZE          1024

void exit_sys(const char *msg);

int main(int argc, char *argv[])
{
    int sock;
    struct addrinfo *ai, *ri;
    struct addrinfo hints = {0};
    char buf[BUFFER_SIZE];
    char *str;
    int result;

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

    if ((sock = socket(AF_INET, SOCK_STREAM, 0)) == -1)
        exit_sys("socket");

    hints.ai_family = AF_INET;
    hints.ai_socktype = SOCK_STREAM;

    if ((result = getaddrinfo(argv[1], argv[2], &hints, &ai)) != 0) {
        fprintf(stderr, "getaddrinfo: %s\n", gai_strerror(result));
        exit(EXIT_FAILURE);
    }

    for (ri = ai; ri != NULL; ri = ri->ai_next)
        if (connect(sock, ri->ai_addr, ri->ai_addrlen) != -1)
            break;

    if (ri == NULL)
        exit_sys("connect");

    freeaddrinfo(ai);

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

    for (;;) {
        printf("Yazı giriniz:");
        fgets(buf, BUFFER_SIZE, stdin);
        if ((str = strchr(buf, '\n')) != NULL)
            *str = '\0';
        if (!strcmp(buf, "quit"))
            break;
    }
}

```



```

        if ((send(sock, buf, strlen(buf), buf[0] == 'u' ? MSG_OOB : 0)) ==
            -1)
            exit_sys("send");
    }

    shutdown(sock, SHUT_RDWR);
    close(sock);

    return 0;
}

```

```

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

    exit(EXIT_FAILURE);
}

```

/

*-----

UNIX Domain soketler aynı makinenin prosesleri arasında haberleşmeler için kullanılabilir. Bu bakımdan bu soketler IP ailesini kullanan soketlerden farklıdır. Unix domain soket yaratabilmek için socket fonksiyonun birinci parametresi AF_UNIX geçilmelidir. Unix domain soketlerin TCPI/IP ya da UDP/IP soketlerle bir ilgisi yoktur. Bu soketler UNIX/Linux sistemlerinde oldukça etkin bir biçimde gerçekleştirilmektedir. Dolayısıyla aynı makinenin prosesleri arasında haberleşmede borulara, mesaj kuyruklarına, paylaşılan bellek alanlarına bir seçenek olarak kullanılabilir. Hatta bazı UNIX türevi sistemlerde (ama Linux'ta böyle değil) aslında çekirdek tarafından önce bu protokol gerçekleştirilip daha sonra boru mekanizması bu protokol kullanılarak gerçekleştirilmektedir. Böylece örneğin aynı makinedeki iki prosesin haberleşmesi için UNIX domain soketler TCP/IP ve UDP/IP soketlerine göre çok daha hızlı çalışmaktadır. Aynı makine üzerinde çok client'lı uygulamalar için UNIX domain soketler boru haberleşmesine göre organizasyonel avantaj bakımından tercih edilebilmektedir.

UNIX domain soketlerin kullanımı en çok boru kullanımına benzemektedir. Ancak Unix domain soketlerin borulara olan bir üstünlüğü "full duplex" haberleşme sunmasıdır. Bilindiği gibi borular "half duplex" bir haberleşme sunmaktadır. Ancak genel olarak boru haberleşmeleri UNIX domain soket haberleşmelere göre daha hızlı olma eğilimindedir.

UNIX domain soketler kullanım olarak daha önce görmüş olduğumuz TCP/IP ve UDP/IP protokollerine çok benzemektedir. Yani işlemler sanki TCP/IP ya da UDP/IP client server program yazılıyormuş gibi yapılır. Başka bir deyişle UNIX domain soketlerinde client ve server programların genel yazım adımları TCP/IP ve UDP/IP ile aynıdır.

UNIX domain soketlerde client'ın server'a bağlanması için gereken adres bir dosya ismi yani yol ifadesi biçimindedir. Kullanılacak yapı sockaddr_in değil sockaddr_un yapısıdır. Bu yapı en azından şu elemanlara sahip olmak zorundadır:

```
struct sockaddr_un {
    sa_family_t sun_family;
    char sun_path[108];
};
```

Yapının sun_family elemanı AF_UNIX biçiminde girilmeli, sun_path elemanı da soketi temsil eden dosyanın yol ifadesi biçiminde girilmelidir. Burada yol ifadesi verilen dosya bind işlemi tarafından yaratılmaktadır. Yaratılan bu dosyanın türü "(s)ocket" biçiminde görüntülenmektedir. Eğer bu dosya zaten varsa bind fonksiyonu başarısız olur. Dolayısıyla bu dosyanın varsa silinmesi gerekmektedir. O halde client ve server programların işin başında bir isim altında anlaşmaları gerekmektedir. sockaddr_un yapısı kullanılmadan önce sıfırlanmalıdır. bind tarafından yaratılan bu soket dosyaları normal l bir dosya değildir. Yani open fonksiyonuyla açılamamaktadır.

Aşağıda stream tabanlı örnek bir UNIX domain uygulaması verilmiştir. Bu örnekte client yalnızca bilgi göndermekte server ise bilgiyi alıp yazdırmaktadır.

-----*/

```
/* uds-server.c */
```

```
/* server.c */
```

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>
#include <unistd.h>
#include <sys/socket.h>
#include <sys/un.h>
```

```
#define BUFFER_SIZE      1024
```

```
char *revstr(char *str);
void exit_sys(const char *msg);
```

```
int main(int argc, char *argv[])
{
    int sock, sock_client;
    socklen_t sunaddr_len;
    struct sockaddr_un sunaddr, sunaddr_client;
    ssize_t result;
    char buf[BUFFER_SIZE + 1];
```

```

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

if ((sock = socket(AF_UNIX, SOCK_STREAM, 0)) == -1)
    exit_sys("socket");

memset(&sunaddr, 0, sizeof(sunaddr));
sunaddr.sun_family = AF_UNIX;
strcpy(sunaddr.sun_path, argv[1]);

if (remove(argv[1]) == -1 && errno != ENOENT)
    exit_sys("remove");

if (bind(sock, (struct sockaddr *)&sunaddr, sizeof(sunaddr)) == -1)
    exit_sys("bind");

if (listen(sock, 8) == -1)
    exit_sys("listen");

printf("Waiting for connection...\n");

sunaddr_len = sizeof(sunaddr_client);
if ((sock_client = accept(sock, (struct sockaddr *)&sunaddr_client,
    &sunaddr_len)) == -1)
    exit_sys("accept");

printf("Connected: %s\n", sunaddr_client.sun_path);

for(;;) {
    if ((result = recv(sock_client, buf, BUFFER_SIZE, 0)) == -1)
        exit_sys("recv");
    if (result == 0)
        break;
    buf[result] = '\0';
    if (!strcmp(buf, "quit"))
        break;
    printf("%ld bytes received: %s\n", (long)result, buf);

    revstr(buf);
    if (send(sock_client, buf, strlen(buf), 0) == -1)
        exit_sys("send");
}

close(sock_client);
close(sock);

return 0;
}

char *revstr(char *str)
{
    size_t i, k;
    char temp;

```

```

    for (i = 0; str[i] != '\0'; ++i)
        ;

    for (--i, k = 0; k < i; ++k, --i) {
        temp = str[k];
        str[k] = str[i];
        str[i] = temp;
    }

    return str;
}

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

    exit(EXIT_FAILURE);
}

/* uds-client.c */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/socket.h>
#include <sys/un.h>

#define BUFFER_SIZE      1024

void exit_sys(const char *msg);

int main(int argc, char *argv[])
{
    int sock;
    struct sockaddr_un sunaddr;
    char buf[BUFFER_SIZE];
    char *str;
    ssize_t result;
    int sresult;

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

    if ((sock = socket(AF_UNIX, SOCK_STREAM, 0)) == -1)
        exit_sys("socket");

    memset(&sunaddr, 0, sizeof(sunaddr));
    sunaddr.sun_family = AF_UNIX;
    strcpy(sunaddr.sun_path, argv[1]);

    if (connect(sock, (struct sockaddr *)&sunaddr, sizeof(sunaddr)) == -1)

```

```

        exit_sys("socket");

    for (;;) {
        printf("Yazı giriniz:");
        fgets(buf, BUFFER_SIZE, stdin);
        if ((str = strchr(buf, '\n')) != NULL)
            *str = '\0';
        if ((send(sock, buf, strlen(buf), 0)) == -1)
            exit_sys("send");
        if (!strcmp(buf, "quit"))
            break;

        if ((result = recv(sock, buf, BUFFER_SIZE, 0)) == -1)
            exit_sys("recv");
        if (result == 0)
            break;
        buf[result] = '\0';
        printf("%ld bytes received: %s\n", (long)result, buf);
    }

    close(sock);

    return 0;
}

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

    exit(EXIT_FAILURE);
}

/
*-----
-----
Stream tabanlı UNIX domain soketlerde server accept uyguladığında
client'a ilişkin sockaddr_un yapısında ne almaktadır?
Aslında bu protokolde bir port kavramı olmadığına göre server'ın
bağlantıdan bir bilgi elde etmeyecektir. Fakat yine de
client program da bind uygulayıp ondan sonra sokete bağlanabilir. Bu
durumda server client bağlantısından sonra sockaddr_un
yapısından client'ın bind ettiği soket dosyanın yol ifadesini elde eder.

Multi client UNIX domain soket uygulamalarında tamamen TCP/IP'de yapmış
olduğumuz server modellerinin aynıları kullanılabilir.

-----*/

/
*-----
-----
UNIX domain soketlerde datagram haberleşme de yapılabilir. Bu haberleşme
mesaj kuyruklarına bir seçenektir. UNIX domain
soket datagram haberleşmede gönderilen datagram'ların aynı sırada
alınması garanti edilmiştir. Yani gönderim UDP/IP'de olduğu gibi

```

güvensiz değil güvenlidir.

Aşağıda UNIX domain soketler kullanılarak bir datagram haberleşme örneği verilmektedir. Burada server hiç bağlantı sağlamadan herhangi bir client'tan paketi alır, oradaki yazıyı ters çevirip ona geri gönderir.

Hem client hem de server ayrı ayrı iki dosya ismi ile bind işlemi paymaktadırlar.

Server program komut satırı argümanı olarak kendi bind edeceği soket dosyasının yol ifadesini, client program ise hem kendi bind edeceği soket dosyasının yol ifadesini hem de server soketin yol ifadesini almaktadır.

```
-----*/
/* uds-dg-server.c */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>
#include <unistd.h>
#include <sys/socket.h>
#include <sys/un.h>

#define BUFFER_SIZE      4096

char *revstr(char *str);
void exit_sys(const char *msg);

int main(int argc, char *argv[])
{
    int sock;
    struct sockaddr_un sunaddr, sunaddr_client;
    socklen_t sunaddr_len;
    ssize_t result;
    char buf[BUFFER_SIZE + 1];

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

    if ((sock = socket(AF_UNIX, SOCK_DGRAM, 0)) == -1)
        exit_sys("socket");

    memset(&sunaddr, 0, sizeof(sunaddr));
    sunaddr.sun_family = AF_UNIX;
    strcpy(sunaddr.sun_path, argv[1]);

    if (remove(argv[1]) == -1 && errno != ENOENT)
        exit_sys("remove");

    if (bind(sock, (struct sockaddr *)&sunaddr, sizeof(sunaddr)) == -1)
        exit_sys("bind");
```

```

printf("Waiting for client data...\n");

for (;;) {
    sunaddr_len = sizeof(sunaddr_client);
    if ((result = recvfrom(sock, buf, BUFFER_SIZE, 0, (struct sockaddr
        *)&sunaddr_client, &sunaddr_len)) == -1)
        exit_sys("recvfrom");

    buf[result] = '\0';
    printf("%ld bytes received from %s: %s\n", (long)result,
        sunaddr_client.sun_path, buf);

    revstr(buf);
    if (sendto(sock, buf, strlen(buf), 0, (struct sockaddr
        *)&sunaddr_client, sizeof(struct sockaddr)) == -1)
        exit_sys("sendto");
}

close(sock);

return 0;
}

char *revstr(char *str)
{
    size_t i, k;
    char temp;

    for (i = 0; str[i] != '\0'; ++i)
        ;

    for (--i, k = 0; k < i; ++k, --i) {
        temp = str[k];
        str[k] = str[i];
        str[i] = temp;
    }

    return str;
}

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

    exit(EXIT_FAILURE);
}

/* uds-dg-client.c */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>
#include <unistd.h>
#include <sys/socket.h>

```

```

#include <sys/un.h>

#define BUFFER_SIZE          4096

void exit_sys(const char *msg);

int main(int argc, char *argv[])
{
    int sock;
    struct sockaddr_un sunaddr, sunaddr_server;
    socklen_t sunaddr_len;
    char buf[BUFFER_SIZE];
    char *str;
    ssize_t result;

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

    if ((sock = socket(AF_UNIX, SOCK_DGRAM, 0)) == -1)
        exit_sys("socket");

    memset(&sunaddr, 0, sizeof(sunaddr));
    sunaddr.sun_family = AF_UNIX;
    strcpy(sunaddr.sun_path, argv[1]);

    if (remove(argv[1]) == -1 && errno != ENOENT)
        exit_sys("remove");

    if (bind(sock, (struct sockaddr *)&sunaddr, sizeof(sunaddr)) == -1)
        exit_sys("bind");

    memset(&sunaddr_server, 0, sizeof(sunaddr));
    sunaddr_server.sun_family = AF_UNIX;
    strcpy(sunaddr_server.sun_path, argv[2]);

    for (;;) {
        printf("Yazı giriniz:");
        fgets(buf, BUFFER_SIZE, stdin);
        if ((str = strchr(buf, '\n')) != NULL)
            *str = '\0';
        if (!strcmp(buf, "quit"))
            break;

        if (sendto(sock, buf, strlen(buf), 0, (struct sockaddr
            *)&sunaddr_server, sizeof(sunaddr_server)) == -1)
            exit_sys("sendto");

        sunaddr_len = sizeof(sunaddr_server);
        if ((result = recvfrom(sock, buf, BUFFER_SIZE, 0, (struct sockaddr
            *)&sunaddr_server, &sunaddr_len)) == -1)
            exit_sys("recvfrom");

        buf[result] = '\0';
    }
}

```



```

        printf("%ld bytes received from %s: %s\n", (long)result,
               sunaddr_server.sun_path, buf);
    }

    close(sock);

    return 0;
}

```

```

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

    exit(EXIT_FAILURE);
}

```

/

*-----

UNIX domain soketler "isimsiz boru haberleşmesine" benzer biçimde de kullanılabilmektedir. Anımsanacağı gibi isimsiz boru hanerleşmesinde yalnızca üst alt prosesler arası haberleşme yapılabilmekteydi. pipe fonksiyonu bize iki betimleyici veriyordu. Biz de fork işlemi ile bu betimleyicileri alt prosese geçiriyorduk. Burada da benzer bir model uygulanmaktadır.

İsimsiz (unbound) soket yaratımı socketpair isimli fonksiyonuyla yapılmaktadır. Fonksiyonun prototipi şöyledir:

```
int socketpair(int domain, int type, int protocol, int sv[2]);
```

Fonksiyonun birinci parametresi protokol ailesinin ismini alır. Her ne kadar fonksiyon genel olsa da pek çok işletim sistemi bu fonksiyonu yalnızca UNIX domain soketler için gerçekleştirmektedir. (Gerçekten de üst ve alt prosesler arasında UNIX domain soketler varken örneğin TCP/IP soketleriyle haberleşmenin zarardan başka bir faydası olmayacaktır.) Fonksiyonun ikinci parametresi kullanılacak soketin türünü belirtir. Bu parametre yine SOCK_STREAM ya da SOCK_DGRAM biiminde girilmelidir. Üçüncü parametre kulalhılacak transport katmanını belirtmektedir. Bu parametre 0 olarak geçilebilir. Son parametre bir çift soket betimleyicisinin yerleştirileceği iki elemanlı int türden dizinin başlangıç adresini almaktadır. Fonksiyon başarı durumunda 0 değerine, başarısızlık durumunda -1 değerine geri dönmektedir.

socketpair fonksiyonu SOCK_STREAM soketler için zaten bağlantı sağlanmış iki soketi bize vermektedir. Yani bu fonksiyon çağrıldıktan sonra listen, accept, connect gibi fonksiyonların çağrılması gereksizdir.

Pekiye isimsiz borularla socketpair fonksiyonuyla oluşturulan isimsiz UNIX domain soketler arasında ne fark vardır? Aslında bu iki kullanım benzer etkilere sahiptir. Ancak en önemli farklılık UNIX domain soketlerin çift yönlü (full duplex) bir haberleşme sağlamasıdır. Ayrıca isimsiz mesaj kuyrukları olmadığına dikkat ediniz. Halbuki isimsiz UNIX domain soketler sanki isimsiz

mesaj kuyrukları gibi de kullanılabilmektedir.

Aşağıdaki programda tıpkı isimsiz boru haberleşmesinde olduğu gibi üst ve alt prosesler birbirileri arasında isimsiz UNIX domain soketler yoluyla haberleşmektedir. Buradaki soketlerin çift yönlü haberleşmeye olanak verdiğini anımsayınız.

```
-----*/  
  
/* uds-socketpair.c */  
  
#include <stdio.h>  
#include <stdlib.h>  
#include <string.h>  
#include <unistd.h>  
#include <sys/wait.h>  
#include <sys/socket.h>  
  
#define BUFFER_SIZE      1024  
  
char *revstr(char *str);  
void exit_sys(const char *msg);  
  
int main(void)  
{  
    int socks[2];  
    char buf[BUFFER_SIZE + 1];  
    char *str;  
    ssize_t result;  
    pid_t pid;  
  
    if (socketpair(AF_UNIX, SOCK_STREAM, 0, socks) == -1)  
        exit_sys("socketpair");  
  
    if ((pid = fork()) == -1)  
        exit_sys("fork");  
  
    if (pid != 0) {        /* parent */  
        close(socks[1]);  
  
        for (;;) {  
            if ((result = recv(socks[0], buf, BUFFER_SIZE, 0)) == -1)  
                exit_sys("recv");  
            if (result == 0)  
                break;  
            buf[result] = '\0';  
            if (!strcmp(buf, "quit"))  
                break;  
            revstr(buf);  
            if (send(socks[0], buf, strlen(buf), 0) == -1)  
                exit_sys("send");  
        }  
  
        if (waitpid(pid, NULL, 0) == -1)  
            exit_sys("waitpid");  
    }  
}
```

```

        close(socks[0]);

        exit(EXIT_SUCCESS);
    }
    else {
        /* child */
        close(socks[0]);
        for (;;) {
            printf("Yazı giriniz:");
            fgets(buf, BUFFER_SIZE, stdin);
            if ((str = strchr(buf, '\n')) != NULL)
                *str = '\0';
            if ((send(socks[1], buf, strlen(buf), 0)) == -1)
                exit_sys("send");
            if (!strcmp(buf, "quit"))
                break;
            if ((result = recv(socks[1], buf, BUFFER_SIZE, 0)) == -1)
                exit_sys("recv");
            if (result == 0)
                break;
            buf[result] = '\0';
            printf("%ld bytes received: %s\n", (long)result, buf);
        }
        close(socks[1]);
        exit(EXIT_SUCCESS);
    }

    return 0;
}

char *revstr(char *str)
{
    size_t i, k;
    char temp;

    for (i = 0; str[i] != '\0'; ++i)
        ;

    for (--i, k = 0; k < i; ++k, --i) {
        temp = str[k];
        str[k] = str[i];
        str[i] = temp;
    }

    return str;
}

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

    exit(EXIT_FAILURE);
}

```

/

*-----

TCP Protokolünün Seviyeli Çalışma Biçimi

TCP protokolü RFC 793 dokümanlarında (<https://tools.ietf.org/html/rfc793>) tanımlanmıştır. Sonradan protokole bazı revizyonlar ve eklemeler yapılmıştır.

Paket tabanlı protokollerin hepsinde gönderilip alınan veriler paket biçimindedir (yani bir grup byte biçimindedir). Bu paketlerin "başlık (header) ve veri (data)" kısmı vardır. Örneğin Ethernet paketinin başlık ve data kısmı, IP paketinin başlık ve data kısmı TCP paketinin başlık ve data kısmı bulunmaktadır. Öte yandan TCP protokolü aslında IP protokolünün üzerine konumlandırılmıştır. Yani aslında

TCP Paketleri IP paketleri gibi gönderilip alınmaktadır. İşin gerçeği bu paketler bilgisayarımıza Ethernet paketi olarak gelmektedir.

Paketlerin başlık kısımlarında önemli "meta data" bilgileri bulunmaktadır. O halde aslında bizim network kartımıza bilgiler sanki ethernet paketiymiş gibi gelmektedir.

O zaman network kartımıza gelen paketin bir ethernet başlığı ve bir de ethernet data kısmı olmalıdır. İşte aslında IP paketi de Ethernet paketinin data kısmında

konuslandırılır. TCP paketi de aslında IP paketinin data kısmında konuşlandırılmaktadır. O halde aslında bize gelen ethernet paketinin data kısmında ip paketi,

IP paketinin data kısmında TCP paketi bulunmaktadır. TCP'de gönderdiğimiz verier aslında TCP paketinin data kısmını oluşturmaktadır. Aşağıdaki şekilde bir paketin

datas'ı TAB ile girintilenerek gösterilmiştir. İşte bize gelen Ethernet Paketi şu görünümündedir:

```
Ether Başlığı
  IP Başlığı
    TCP Başlığı
      TCP Datası
```

Örneğin biz send fonksiyonuyla "ankara" yazısını gönderiyor olsak bu "ankara" yazısını oluşturan byte'lar aslında TCP Data'sındadır.

```
Ether Başlığı
  IP Başlığı
    TCP Başlığı
      "Ankara"
```

Tabii IP paketleri aslında yalnızca bilgisayarımıza gelirken Ethernet paketi olarak gelir. Dışarda rotalanırken Ethernet paketi söz konusu değildir.

TCP Başlığı şöyledir:

```
Source Port | Destination Port
Sequence Number
Acknowledge Number
```

Data Offset | 000 | Flags | Window Size
Checksum | Urgent Pointer
Options

Burada her satır 32 bit yani 4 byte yer kaplamaktadır. TCP başlığı 20 byte'tan 60 byte'a kadar değişen uzunlukta olabilir. Başlıktaki Data Offset

TCP data'sının hangi offsetten başladığını dolayısıyla TCP başlığının uzunluğunu belirtir. Options 4 bit uzunluğundadır. Buradak değer 4 ile çarpılmalıdır.

Böylece Data Offset kısmında en az 5 (toplam 20 byte) en fazla 15 (toplam 60 byte) değeri bulunabilir. Bu başlıkta kaynak ve hedef IP adreslerinin ve

TCP data kısmının uzunluğunun bulunmadığına dikkat ediniz. Çünkü bu bilgiler zaten IP başlığında doğrudan ya da dolaylı biçimde bulunmaktadır.

TCP paketi her zaman IP paketinin data kısmında kodlanır.

Başlıktaki Flags alanı 6 bitten oluşmaktadır. Her bir bir özelliği temsil eder. Buradaki belli bitler set edildiğinde başlıktaki belli alanlar da

anamlı hale gelebilmektedir. Buradaki bitler şöyledir: URG, ACK, PSH, RST, SYN, FIN. Flags alanındaki bir'den fazla bit 1 olabilir. Bir TCP paketi (TCP segment)

yalnızca başlık içerebilir. Yani hiç data içermeyebilir.

TCP protokolünde o anda iki tarafında bulunduğu bir durum vardır.

Taraflar belli eylemler sonucunda durumdan duruma geçiş yaparlar. Bu nedenle TCP'nin çalışması

bir "sonlu durum makinesi (finite state mschine)" biçiminde ele alınıp açıklanabilir. Henüz bağlantı yoksa iki taraf da CLOSED denilen durumdadır.

TCP bağlantısının kurulması için client ile server bazı paketleri gönderip almaktadır. Buna el sıkışma (hand shaking) denilmektedir.

TCP'de bağlantı kurulması

için yapılan el sıkışma 3'lü (three way) ya da 4'lü (four way) olabilir.

Uygulamada 3'lü el sıkışma kullanılmaktadır. Normal olarak bağlantının kurulabilmesi

için iki tarafın da birbirlerine SYN biti set edilmiş data kısmı olmayan TCP paketi (20 byte) gönderip karşı taraftan ACK biti set edilmiş (data'sı olmayan)

TCP paketi alması gerekir. Bunun ikiyolu olabilir:

Client	Server
CLOSED	CLOSED
----- SYN ----->	
SYN-SENT	SYN-RECEIVED
<----- ACK -----	
<----- SYN -----	
ESTABLISHED	
----- ACK ----->	
	ESTABLISHED

Burada 4 paket kullanıldığı için buna 4'lü el sıkışma denilmektedir.
Ancak pratikte 3'lü el sıkışma kullanılmaktadır:

Client	Server
CLOSED	CLOSED
----- SYN ----->	
SYN-SENT	SYN-RECEIVED
<-- SYN + ACK ----	
ESTABLISHED	
----- ACK ----->	
	ESTABLISHED

Bağlantı kopartılması için iki tarafın da birbirilerine FIN biti set edilmiş paketler gönderip ACK biti set edilmiş paketleri alması gerekir.

Bağlantının kopartılması da tipik olarak 3'lü ya da 4'lü el sıkışma yoluyla yapılmaktadır. Bağlantının kopartılması talebini herhangi bir taraf başlatabilir.

3'lü el sıkışma ile bağlantının kopartılması şöyle yapılmaktadır

Peer-1	Peer-2
ESTABLISHED	ESTABLISHED
----- FIN ----->	
FIN-WAIT-1	CLOSE_WAIT
<-- FIN + ACK ----	
FIN-WAIT-2	LAST-ACK
----- ACK ----->	
TIME-WAIT	CLOSED
CLOSED	

Burada özetle bir taraf önce karşı tarafa FIN paketi yollamıştır. Karşı taraf buna ACK+FIN ile karşılık vermiştir. Diğer taraf da son olarak karşı tarafa ACK

yollamıştır. Ancak bağlantıyı koprtmak isteyen taraf bu ACK yollama işinden sonra MSL (Maximum Segment Life) denilen bir zaman aralığının iki katı kadar

beklemektedir (Tipik olarak 2 dakika). Eğer bu taraf beklemene hemen CLOSED duruma geçseydi şöyle bir sorun oluşabilirdi: Bu taraf hemen yeniden programı çalıştırıp

karşı tarafa paket yollayıp sanki eski bağlantıyı devam ettirme gibi bozuk bir durum söz konusu olabilirdi. Halbuki diğer taraf zaten bu son ACK paketini almamışsa

belli zaman sonra CLOSED duruma geçmektedir.

Eğer shutdown işlemi ile sonlandırma yapılırsa sonlandırma 4'lü el sıkışmayla gerçekleşir. Pratikte daha çok bu durumla karşılaşılmaktadır. 4'lü el sıkışmaya

"yarı kapama (half close)" el sıkışması da denmektedir. Yarı kapama bir tarafın artık send yapmayacağını ancak receive yapabileceğini belirtir. Bu durumda bağlantı devam

edebilir. Ta ki karşı taraf da bu half close işlemini yapana kadar.

Peer-1	Peer-2
--------	--------

```

ESTABLISHED          ESTABLISHED
----- FIN ----->
FIN-WAIT-1           CLOSE-WAIT
<-- ACK -----
                      CLOSING

```

Artık Peer1-1 data gönderemez ama alabilir. Peer2 ise alamaz ama gönderebilir.

```

<-- FIN -----
TIME_WAIT
---- ACK ---->
                      TIME_WAIT
CLOSED              CLOSED

```

TCP'de akış kontrolü için "acknowledgement" yani alındı bildirimi kullanılmaktadır. Bir taraf bir tarafa birşeyler gönderdiği zaman karşı taraf bunu aldığını Flags kısmındaki ACK biti set edilmiş bir paket ile bildirir. ACK paketleri boş olabilir ya da dolu olabilir. Yani karşı taraf bir bilgiyi gönderirken de aynı anda daha önce almış olduğu bilgiler için ACK yollayabilir. TCP'de kümülatif bir "acknowledgement" sistemi kullanılmaktadır. Gönderilen her byte'ın bir başlangıç sıra numarası vardır. Buna "sequence number" denilmektedir. Sequence number TCP başlığında 32 bitlik bir alandır. Bu alandaki sayı son noktasına gelirse yeniden başa dönmektedir. Sequence number bağlantı kurulduğunda sıfırdan başlamaz, rastgele bir değerden başlatılmaktadır. Örneğin belli bir anda bir tarafın sequence number değeri 1552 olsun. Şimdi bu taraf karşı tarafa 300 byte göndersin. Artık bu sequence number 1852 olacaktır. Yani bir sonraki gönderimde bu taraf sequence number olarak 1852 kullanacaktır. Sequence number her gönderimde bulundurulmak zorundadır. Karşı taraf bilgiyi aldığını ACK bitiyle onaylarken TCP başlığındaki "acknowledgement number" denilen sayıyı da gönderir. Acknowledgement karşı taraftan talep edilen sonraki sequence number değeri olarak gönderilir. Örneğin:

Peer-1	Peer-2
300 byte (sequence Number: 3560) ----->	
100 byte (sequence Number: 3860) ----->	
	<---- ACK (Acknowledgement Number: 3960)
50 byte (sequence Number: 3960) ----->	
	<---- ACK (Acknowledgement Number: 4010)
10 byte (sequence Number: 4010) ----->	

Pekiye bir TCP paketi (TCP segment) gönderici (sender) tarafından gönderildikten sonra alıcı (receiver) bunu alamamışsa ne olacaktır? Çünkü TCP'nin güvenli bir protokol olması demek bir biçimde böyle bir durumda bir telafinin yapılması demektir. TCP protokolü şöyle yöntem izlemektedir: Gönderen taraf her gönderdiği paket (TCP segment) için bir zamanlayıcı kurar. Bu zamanlayıcıya "retransmission timer" denilmektedir. Eğer belli süre içerisinde gönderilen TCP paketini kapsayan bir ACK gelediyse gönderici taraf aynı paketi yeniden göndermektedir. Böylece aslında gönderilen paket henüz onun için ACK gelmedikçe gönderme tamponundan atılmaz. Retransmission Timer bazı değerlere göre dinamik bir biçimde oluşturulmaktadır. Bunun detayları için önerilen kaynaklara bakılabilir. Tabii böyle bir sistemde alıcı taraf aynı paketi birden fazla kez alabilmektedir. Bu durumda bu paketlerin yalnızca tek bir kopyasını alıp diğerlerini atmak alıcı tarafın sorumluluğundadır.

Akış kontrolünün en önemli unsurlarından biri de tampon taşmasını engellemektir. Örneğin gönderici taraf sürekli bilgi gönderirse fakat alıcı taraftaki proses recv işlemiyle hiç okuma yapmazsa alıcı taraftaki işletim sisteminin alıcı tamponu dolabilir ve sistem çökebilir. İşte akış kontrolü sayesinde alıcı taraf gönderici tarafa "artık gönderme, benim tamponum doldu" diyebilmektedir. TCP'de bunun için "pencere (window)" kavramı kullanılmaktadır. Pencerenin bir büyüklüğü (window size) vardır. Pencere büyüklüğü demek "hiç ACK gelmediği durumda göndericinin en fazla en fazla gönderebileceği byte sayısı" demektir. Örneğin pencerenin 8K olması demek "alıcı ACK" göndermedikten sonra gönderici en fazla 8K gönderebilmesi demektir. Pencere genişliği alıcı taraf tarafından gönderici tarafa bildirilir. Örneğin pencere genişliği 8K olsun. Bu durumda gönderici taraf sırasıyla 1K + 1K + 1K + 1K + 1K uzunluğunda toplam 5K'lık bilgiyi karşı tarafa göndermiş olsun. Eğer henüz ACK gelmemişse gönderici taraf en fazla 3K kadar daha bilgi gönderebilir. Tam bu sırada iki ayrı 1K için kümülatif olarak ACK geldiğini düşünelim. Şimdi pencere genişliği gönderen taraf için ne olmalıdır? İşte bu durum alıcı tarafın ACK gönderirken aslında ACK gönderdiği bilgileri tamponunda eritmiş olup olmadığına bağlıdır. Pekala alıcı taraf ACK gönderdiği halde henüz oradaki proses recv yapmadığı için bu bilgileri 8K'lık tamponunda hala bekletiyor olabilir. Bu durumda gönderici tarafın pencere genişliğinin 5K'ya düşürülmesi gerekir. İşte TCP'de her ACK sırasında yeni pencere genişliği de aslında karşı tarafa gönderilmektedir. O halde aslında pencere genişliği her ACK bildiriminde dinamik olarak ayarlanmaktadır. Tabii TCP çift taraflı olduğuna göre her iki tarafın ayrı sequence number, acknowledgement number ve pencere genişlikleri vardır. Ayrıca ACK paketlerinin illa da daha önce alınmış olan bir paketin alındığını bildirmek amaçlı kullanılmayabileceğini de belirtelim. Bir taraf pencere genişliğini

ayarlamak için de ACK gönderebilir. Tabi bu durumda "acknowledgement number" yeni aynı değerde olmalıdır.

Pekiye sequence number, pencere genişlikleri nasıl iki tarafa bildirilmektedir. İşte bağlantı kurulurken client taraf SYN paketi içerisinde kendi başlangıç sequence number'ını karşı tarafa iletmektedir. Server de bağlantıyı kabul ederken yine SYN (ya da SYN + ACK) paketinde kendi sequence number'ını karşı tarafa bildirmektedir. Aslında SYN paketi yalnızca başlık içermektedir. Data içermemektedir. Ancak TCP protokolüne göre özel bir durum olarak SYN paketlerinde sanki tek bir byte varmış gibi acknowledgement number alınmaktadır. Pencere genişliği de aslında ilk kez bağlantı yapılırkenki ACK paketlerinde belirtilmektedir. Yukarıda belirtildiği gibi pencere genişliği her ACK paketinde bildirilir.

TCP/IP stack gerçekleştirmeleri ACK stratejisi için bazı yöntemler uygulamaktadır. Örneğin eğer gönderilecek paket varsa bununla birlikte ACK paketinin gönderilmesi, ACK'ların iki paket biriktirildikten sonra gönderilmesi gibi. Bunun için "Tcp IP Protocol Suite 466'ıncı sayfaya başvurabilirsiniz."

TCP paketindeki önemli Flag'lerden birisi de "RST" bitidir. Buna "reset isteği" denilmektedir. Bir taraf RST bayrağı set edilmiş paket alırsa artık karşı tarafın "abnormal" bir biçimde bağlantıyı kopartıp yeniden bağlanma talep ettiği anlaşılır. Normal sonlanma el sıkışarak başarılı bir biçimde yapılırken RST işlemi anormal sonlanmaları temsil eder. Örneğin socket kütüphanelerinde hiç shutdown yapmadan socket close edilirse close eden taraf karşı tarafa RST paketi göndermektedir. Halbuki önce shutdown yapılırsa el sıkışmalı sonlanma gerçekleştirilir. O halde her zaman aktif socketler shutdown yapıldıktan sonra close edilmelidir.

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

UDP protokolü aslında saf IP protokolüne çok benzerdir. UDP'yi IP'den ayıran iki önemli farklılık şudur:

- 1) UDP'nin port numarası kavramına sahip olması
- 2) UDP'nin hata için bir checksum kontrolü uygulayabilmesidir.

Bir UDP paketi yine aslında IP paketinin data kısmında bulunmaktadır. IPV4 Formatı şöyledir:

Source Port		Destination Port
Total Length		Checksum

Data

Burada UDP paketinin toplam uzunluğunun bulunması aslında gereksizdir. Çünkü uzunluk TCP'de olduğu gibi aslında IP paketinin başlığına bakılarak tespit edilebilmektedir. Ancak hesaplama kolaylı oluşturmak için bu uzunluk UDP başlığında ayrıca bulundurulmuştur. Aslında checksum UDP paketlerinde bulunmak zorunda değildir. Eğer gönderici checksum kontrolü istemiyorsa burayı 0 bitleriyle doldurur. (Eğer zaten checksum 0 ise burayı 1 bitleriyle doldurmaktadır.) Alan taraf checksum hatasıyla karşılaşırsa TCP'de olduğu gibi paketi yeniden talep etmez. Yalnızca onu atar.

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

İçerisinde derlenmiş bir biçimde fonksiyonların bulunduğu dosyalara kütüphane (library) denilmektedir. Kütüphaneler yalnızca fonksiyon değil

global nesneler de içerebilirler. Kütüphaneler "statik" ve "dinamik" olmak üzere ikiye ayrılmaktadır. Statik kütüphane dosyalarının uzantıları

UNIX/Linux sistemlerinde ".a", Windows sistemlerinde ".lib" biçimindedir. Kütüphanelerin nasıl oluşturulacağı ve nasıl kullanılacağı iki ayrı alt konuyu içermektedir.

Statik kütüphaneler aslında "object modülleri (yani .o dosyalarını)" tutan bir kap gibidir. Statik kütüphaneler aslında object modüllerden oluşmaktadır.

Statik kütüphanelere link aşamasında linker tarafından bakılır. Bir program statik kütüphane dosyasından bir çağırma yaptıysa (ya da o kütüphaneden bir global değişkeni kullandıysa)

linker o statik kütüphane içerisinde ilgili fonksiyonun bulunduğu object modülü link aşamasında statik kütüphane dosyasından çekerek çalıştırılabilir dosyaya

yazar. (Yani statik kütüphaneden bir tek fonksiyon çağırırsak bile aslında o fonksiyonun bulunduğu object modülün tamamı çalıştırılabilen dosyaya yazılmaktadır.)

Statik kütüphaneleri kullanan programlar artık o statik kütüphaneler olmadan çalıştırılabilirler.

Statik kütüphanelerin şu dezavantajları vardır:

– Kütüphaneyi kullanan farklı programlar aynı fonksiyonun (onun bulunduğu object modülün) bir kopyasını çalıştırılabilir dosya içerisinde bulundururlar.

Yani örneğin printf fonksiyonu statik kütüphanede ise her printf kullanan C programı aslında printf fonksiyonun bir kopyasını da barındırıyor durumda olur.

Bu da disk hacmini azaltır. Programların gereksinim duyacağı sanal bellek miktarını artırır.

- Statik kütüphane kullanan programlar dinamik kütüphane kullanan programlara göre kimi zaman daha geç yüklenebilirler.
- Statik kütüphanede bir değişiklik yapıldığında onu kullanan programların yeniden link edilmesi gerekir.

Statik kütüphane dosyalarının şu avantajları vardır:

- Kolay konuşlandırılabilirler.
- Kullanımları kolaydır ve build işlemini sadeleştirirler.

UNIX/Linux sistemlerinde statik kütüphane dosyaları `ar` isimli utility programla oluşturulur. Tipik kullanım şöyledir:

```
ar r libsample.a a.o b.o c.o
```

`r` seçeneği (yanında `-` olmadığına dikkat ediniz) ilgili object modüllerin kütüphaneye yerleştirilmesini sağlar. kütüphane yoksa ayrıca kütüphane dosyası da yaratılmaktadır. `t` seçeneği kütüphane içerisindeki object modüllerin listesini almakta kullanılır. Örneğin:

```
ar t libsample.a
```

`d` seçeneği kütüphaneden bir object modülü silmekte, `x` seçeneği onu `.o` dosyası biçiminde save etmekte ve `m` seçeneği yeni versiyonu eski versiyonla değiştirmekte kullanılır. O halde `a.c` dosyasının içindekileri statik kütüphaneye eklemek şöyle yapılmalıdır:

```
gcc -c a.c  
ar r libsample.a a.o
```

Statik kütüphaneler link aşamasında linker için komut satırında belirtilmelidir. Örneğin:

```
gcc -o app app.c libsample.a
```

`gcc .a` uzantılı dosyaları linker'a pass etmektedir.

Komut satırında kütüphane dosyalarının komut satırı argümanlarının sonunda belirtilmesi uygundur. Çünkü `gcc` programı kütüphane dosyalarının solundaki dosyalar link edilirken ilgili kütüphane dosyasını bu işleme dahil ederler.

Şüphesiz statik kütüphane kullanmak yerine aslında object modülleri de doğrudan link işlemine sokabiliriz. Örneğin:

```
gcc -o sample sample.c a.o b.o
```

Çok sayıda object modül söz konusu olduğunda bu işlemin zorlaşacağına dikkat ediniz. Yani object modüller dosyalara benzetilirse statik kütüphane dosyaları dizinler gibi düşünülebilir.

Derleme işlemi sırasında kütüphane dosyası -l<isim> biçiminde de belirtilebilir. Bu durumda arama sırasında "lib" öneki aramaya dahil edilmektedir.

Yani örneğin:

```
gcc -o sample sample.c -lsample
```

İşleminde aslında libsample.a (ya da libsample.so) dosyaları aranmaktadır. Arama işlemi sırasıyla bazı dizinlerde yapılmaktadır. Örneğin /lib dizini, /usr/lib dizini gibi dizinlere bakılmaktadır. Ancak bulunulan dizine bakılmamaktadır. Belli bir dizine bakılması isteniyorsa -L seçeneği ile dizin eklenebilir.

Örneğin:

```
gcc -o sample sample.c -lsample -L.
```

Buradaki '.' çalışma dizinini temsil etmektedir.

Bir statik kütüphane başka bir statik kütüphaneye bağımlı olabilir.

Örneğin biz liby.a kütüphanesindeki kodda libx.a kütüphanesindeki fonksiyonları kullanmış olabiliriz.

Bu durumda liby.a kütüphanesini kullanan program libx.a kütüphanesini de komut satırında belirtmek zorundadır. Örneğin:

```
gcc -o sample sample.c libx.a liby.a
```

-----*/
/
*-----

Dinamik kütüphane dosyalarının UNIX/Linux sistemlerinde uzantıları ".so" (so shared object'ten kısaltma), Windows sistemlerinde ise .dll (Dynamic Link Library) biçimindedir.

Bir dinamik kütüphaneden bir fonksiyon çağrıldığında linker statik kütüphanede olduğu gibi gidip fonksiyonun kodunu çalıştırılabilen dosyaya yazmaz.

Bunun yerine çalıştırılabilen dosyaya çağrılan fonksiyonun hangi dinamik kütüphanede olduğu bilgisini yazar. Çalıştırılabilen dosyayı yükleyen işletim sistemi

o dosyanın çalışması için gerekli olan dinamik kütüphaneleri çalıştırılabilen dosyayla birlikte bütünsel olarak sanal bellek alanına yüklemektedir. Böylece birtakım

ayarlamalar yapıldıktan sonra artık çağrılan fonksiyon için gerçekten o anda sanal belleğe yüklü olan dinamik kütüphane kodlarına gidilmektedir.

Dinamik kütüphane kullanımının avantajları şunlardır:

- Çalıştırılabilen dosyalar fonksiyon kodlarını içermezler. Dolayısıyla önemli bir disk hacmi kazanılmış olur.
- Dinamik kütüphaneler birden fazla program tarafından tekrar tekrar yüklenmeden kullanılabilir. Yani işletim sistemi arka planda aynı dinamik kütüphaneyi kullanan programlarda bu kütüphaneyi tekrar tekrar fiziksel belleğe yüklememektedir. Bu da statik kütüphanelere göre önemli bir avantaj oluşturmaktadır. Bu durum eğer dinamik kütüphane daha önce yüklenmişse programın daha hızlı yüklemesine de yol açabilmektedir.
- Dinamik kütüphaneleri kullanan programlar bu dinamik kütüphanelerdeki değişikliklerden etkilenmezler. Yani biz dinamik kütüphanenin yeni bir versiyonunu oluşturduğumuzda bunu kullanan programları derlemek ya da link etmek zorunda kalmayız.

Dinamik kütüphaneler proseslerin sanal bellek alanlarının farklı yerlerine yüklenebilirler. Bunların yüklenme adreslerinin değişik olabilmesi "relocation"

denilen sorunu ortaya çıkarmaktadır. "Relocation" çalıştırılabilir ya da dinamik kütüphane dosyalarının sanal bellekte herhangi bir yere yüklenebilmesi

anlamına gelir. Ancak relocation işlemi basit bir işlem değildir. Çünkü bu dosyalar derlenirken derleyici onların sanal belleğin neresine yükleneceğini bilmemektedir.

Relocation işlemi için temelde iki teknik kullanılmaktadır:

- 1) Relocation Tablosu Tekniği
- 2) Konumdan Bağımsız Kod (Position Independent Code) Tekniği

Windows işletim sistemi "relocation tablosu" tekniğini kullanmaktadır.

Burada dosyada "relocation" bilgilerinin bulunduğu bir alan vardır. Bu alanda adresleri düzeltilecek offset'ler dosyan başından itibaren tek tek belirtilmektedir. Yükleyici DLL'i uygun yere yükledikten sonra bu relocation tablosundaki offsetleri tek tek düzeltir. Böylece kod yüklendiği yerde çalışır hale getirilir. UNIX/Linux sistemlerinde ise ağırlıklı olarak

"Konumdan Bağımsız Kod" Tekniği kullanılmaktadır. Burada üretilen kodlar zaten hep görelî adresler içerir. Böylece üretilen bu kodlar zaten görelî adresler

içerdiği için nereye yüklenirse çalışabilir.

Relocation Tablosu yönteminde "relocation" işlemi belli bir zaman almaktadır. Bu da programın yüklenme zamanını uzatabilmektedir. Fakat dinamik kütüphane

"reloaction" yapıldığında çalışma hızlıdır. Konumdan Bağımsız Kod tekniğinde "reloaction" zamanı minimize edilmiştir. Fakat görelî adreslerle çalışan kodlar

toplamda daha yavaş olma eğilimindedir.

UNIX/Linux sistemlerinde bir dinamik oluşturma işlemi şöyle yapılır:

- 1) Önce dinamik kütüphaneye yerleştirilecek object modüllerin `-fPIC` seçeneği ile Konumdan Bağımsız Kod tekniği kullanılarak derlenmesi gerekir
- 2) Link aşamasında `-shared` seçeneğini kullanılması gerekir. `-shared` kullanılmazsa dinamik kütüphane değil normal çalıştırılabilir dosya oluşturulur.

```
gcc -fPIC a.c b.c c.c
gcc -o libsample.so -shared a.o b.o c.o
```

Dinamik kütüphanelere daha sonra dosya eklenip çıkartılamaz. Onların her defasında yeniden oluşturulmaları gerekmektedir. Yukarıdaki işlem tek hamlede şöyle de yapılabilir:

```
gcc -o libsample.so -shared -fPIC a.c b.c c.c
```

Dinamik kütüphane kullanan bir program link edilirken kullanılan dinamik kütüphanenin komut satırında belirtilmesi gerekir. Örneğin:

```
gcc -o sample sample.c libsample.so
```

Tabii bu işlem yine `-l` seçeneği ile de yapılabilirdi:

```
gcc -o sample sample.c -lsample -L.
```

```
-----*/
/
*-----
```

Default durumda gcc (ve tabii clang) derleyicileri standart C fonksiyonlarının ve POSIX fonksiyonlarını (libc kütüphanedi) dinamik kütüphaneden alarak kullanır. Ancak programcı isterse `-static` seçeneği ile statik link işlemi de yapabilir. Bu durumda bu fonksiyonlar statik kütüphanelerden alınarak çalıştırılabilen dosyalara yazılacaktır. Örneğin:

```
gcc -o sample -static sample.c
```

Tabii bu biçimde statik link işlemi yapıldığında çalıştırılabilen dosyanın boyutu çok büyüyecektir.

Bir programın kullandığı dinamik kütüphaneler `ldd` isimli utility program ile basit bir biçimde görüntülenebilir. Örneğin:

```
csd@csd-vm:~/Study/Unix-Linux-SysProg/static-lib$ ldd sample
linux-vdso.so.1 (0x00007fff38162000)
libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f7ec0b5c000)
/lib64/ld-linux-x86-64.so.2 (0x00007f7ec114f000)
```

```
-----*/
/
*-----
```

Pekiye bizim programımız örneğin libsample.so isimli bir dinamik kütüphaneden çağrı yapıyor olsun. Bu libsample.so dosyasının program çalıştırılırken nerede bulundurulması gerekir? İşte program çalıştırılırken ilgili dinamik kütüphane dosyasının özel bazı dizinlerde bulunuyor olması gerekir. Yükleme sırasında hangi dizinlere bakıldığı "man ld.so" sayfasından elde edilebilir. Bu dokümanda belirli sırada tek tek hangi dizinlere bakıldığı belirtilmiştir. Ayrıntılar bir yana bırakılırsa kabaca LD_LIBRARY_PATH isimli çevre değişkeninde belirtilen dizinlere (bu çevre değişkeni de PATH çevre değişkeni gibi ':'ler ile ayrılmış dizinler oluşmaktadır. '.' dizini yine "o anki çalışma dizini" anlamına gelmektedir.) ve /lib, /usr/lib dizinlerine bakılmaktadır. O halde basit bir seçenek bizim dinamik kütüphanemizi /lib ya da /usr/lib dizinlerinin birinin içine çekmek ya da LD_LIBRARY_PATH çevre değişkenine dinamik kütüphanemizin bulunduğu dizini eklemektedir. /lib dizini işletim sisteminin boot edilebilmesi için gereken birtakım araçların da kullandığı daha temel bir kütüphane dizinidir. Uygulama programcılarının kütüphanelerini buraya değil /usr/lib içerisine yerleştirmesi daha uygun olur.

Dinamik kütüphanelerin aranacağı yer aslında doğrudan çalıştırılabilen dosyanın içerisine (dynamic isimli bölüme (section)) da yazılabilir. Bunun için

-rpath <yol ifadesi> linker seçeneği kullanılmalıdır. Buradaki yol ifadesinin mutlak olması gerekir. Ancak eskiden bunun için DT_RPATH isimli tag kullanılırken daha sonra bu tag "deprecated" yapılmış DT_RUNPATH tag'ı kullanılmaya başlanmıştır. Bu seçeneği linker'a geçirebilmek için gcc'de -Wl seçeneğini kullanmak gerekir. -Wl seçeneği bitişik yazılan virgüllü alanlardan oluşmaktadır. gcc bını ld linler'ına virgüller yerine boşluklar koyarak geçirmektedir. Örneğin:

```
gcc -o sample -Wl,-rpath,/home/csd/Study/Unix-Linux-SysProg/static-lib,sample.c libsample.so
```

Burada ELF formatının DT_RUNPATH tag'ına yerleştirme yapılmaktadır. Eğer DT_RPATH tag'ına yerleştirme yapılmak isteniyorsa ayrıca --disable-new-dtags seçeneğinin de girilmesi gerekmektedir. Ancak burada küçük bir pürüz vardır. Kurulum sırasında dinamik kütüphane ya başka dizine yerleştirilirse ne olacaktır? İşte -rpath seçeneğinde '\$ORIGIN' program dosyasının bulunduğu dizin anlamına gelir. Örneğin:

```
gcc -o sample -Wl,-rpath,'$ORIGIN'/. sample.c libsample.so
```

Burada artık neresi olursa olsun dinamik kütüphane çalıştırılan programla aynı dizinde bulunursa sorun oluşmayacaktır. Pekiyi DT_RPATH tag'ı neden deprecated yapılıp DT_RUNPATH tag'ı eklenmiştir? İşte aslında arama sırası bakımından DT_RPATH tag'ının en yukarıda olması (daha soğrusu LD_LIBRARY_PATH'in yukarısında olması) yanlış bir tasarımıdır. Geriye doğru uyumu koruyarak bu yanlış tasarım telafi edilmiştir.

Dinamik kütüphanenin aranması sırasında /lib ve /usr/lib dizilerine bakılmadan önce özel bir dosyaya da bakılmaktadır. Bu dosya /etc/ld.so.cache isimli dosyadır.

/etc/ld.so.cache dosyası aslında binary bir dosyadır. Hızlı arama yapılabilen sözlük tarzı ilgili dosyanın hangi dizinde olduğunu gösteren bir yapıdadır.

Yani bu dosta .so dosyalarının hangi dizinlerde olduğunu belirten binary bir dosyadır. Pekiyi bu dosyanın içerisinde hangi so dosyaları vardır? Aslında bu dosyanın

içerisinde /lib ve /usr/lib dizinindeki so dosyalarının hepsi vardır. Ama programcı isterse kendi dosyalarını da bu cache dosyasının içerisine yerleştirebilir.

Pekiyi neden böyle bir cache dosyasına geeks'inim duyulmuştur? Dinamik kütüphaneler yüklenirken /lib ve /usr/lib dizinlerinin taranması görece olarak uzun zaman

almaktadır. Bu da programın yüklenme süresini uzatabilmektedir. Halbuki bu dizinlere bakılmadan önce bu cache dosyasına bakılırsa ilgili dosyanın olup olmadığı varsa

nerede olduğu çok daha hızlı bir biçimde elde edilebilmektedir. Burada dikkat edilmesi gereken nokta bu cache dosyasına /lib ve /usr/lib dizinlerinden daha önce bakıldığı

ve bu dizinlerin içeriğinin de zaten bu cache dosyasının içerisinde olduğudur. O halde aslında /lib ve /usr/lib dizinlerinde arama çok nadir olarak yapılır.

Pekiyi /etc/ld.so.cache dosyasına biz nasıl bir dosya ekleriz? Aslında programcı bunu dolaylı olarak yapar. Şöyle ki: /sbin/ldconfig isimli bir program vardır.

Bu program /etc/ld.so.conf isimli bir text dosyaya bakar. Bu dosya dizinlerden oluşmaktadır. Bu dizinlerin içindeki so dosyalarını /etc/ld.so.cache dosyasına

ekler. Şimdiler de /etc/ld.so.conf dosyasının içeriği şöyledir:

```
include /etc/ld.so.conf.d/*.conf
```

Bu ifade /etc/ld.so.conf.d dizinindeki tüm .conf uzantılı dosyaların bu işleme dahil edileceğini belirtir.

Biz ldconfig programını çalıştırdığımızda bu program /lib, /usr/lib ve /etc/ld.so.conf (dolayısıyla /etc/ld.so.conf.d dizinindeki .conf dosyaları)na bakarak

/etc/ld.so.cache dosyasını yeniden oluşturmaktadır. O halde bizim bu cache'e ekleme yapmak için tek yapacağımız şey /etc/ld.so.conf.d dizinindeki bir .conf

dosyasına yeni bir satır olarak bir dizinin yol ifadesini girmektir. (.conf dosyaları her satırda bir dizinin yol ifadesinden oluşmaktadır) Tabii programcı isterse bu dizine yeni bir .conf dosyası da ekleyebilir. İşte programcı bu işlemi yaptıktan sonra /sbin/ldconfig programını çalıştırınca artık onun eklediği dizinin içerisindeki so dosyaları da /etc/ld.so.cache dosyasının içerisine eklenmiş olacaktır. Daha açık bir anlatımla programcı bu cache dosyasına ekleme işini adım adım şöyle yapar:

- 1) Önce so dosyasını bir dizine yerleştirir.
- 2) Bu dizinin ismini /etc/ld.so.conf.d dizinindeki bir dosyanın sonuna ekler.
- 3) /sbin/ldconfig programını çalıştırır.

Programcı /etc/ld.so.conf.d dizinindeki herhangi bir dosyaya değil de -f seçeneği ile kendi belirlediği bir dosyaya da ilgili dizini yazabilmektedir. ldconfig her çalıştırıldığında sıfırdan yeniden cache dosyasını oluşturmaktadır.

Programcı /lib ya da /usr/lib dizinine bir so dosyası eklediğinde ldconfig programını çalıştırması zorunlu olmasa da iyi bir tekniktir. Çünkü o dosya da cache dosyasına yazılacak ve daha hızlı bulunacaktır.

-----*/
/
*-----

UNIX/Linux sistemlerinde dinamik kütüphane dosyalarına isteğe bağlı olarak birer versiyon numarası verilmektedir. Bu versiyon numarası dosya isminin bir parçası durumundadır. Linux sistemlerine izlenen tipik numaralandırma (convention) şöyledir:

<dosya ismi>.so.<majör numara>.<minör yüksek numara>.<minör alçak numara>

Örneğin:

libsample.so.2.4.6

Majör numara büyük değişiklikleri, minör numaralar ise küçük değişiklikleri anlatmaktadır. Majör numara değişirse yeni dinamik kütüphane eskisiyle uyumlu değildir. Burada "uyumlu değildir" lafı eski dinamik kullanan programların yenisini kullanamayacağı anlamına gelmektedir. Çünkü muhtemelen bu yeni versiyonda fonksiyonların isimlerinde, parametrik yapılarında değişiklikler söz konusu olmuş olabilir. Bazı fonksiyonlar silinmiş olabilir.

Fakat majör numarası aynı ancak minör numaraları farklı olan kütüphaneler birbirleriyle uyumludur. Yani alçak minör numarayı kullanan program yüksek minör numarayı kullanırsa sorun olmayacaktır. Bu durumda yüksek minör numaralı kütüphanede hiçbir fonksiyonun ismi, parametrik yapısı değişmemiştir ve hiçbir fonksiyon silinmemiştir. Örneğin fonksiyonlar daha hızlı çalışacak biçimde optimizasyonlar yapılmış olabilir. Ya da örneğin yeni birtakım fonksiyonlar da eklenmiş olabilir.

Linux sistemlerinde bir konvansiyon olarak bir kütüphanenin üç ismi bulunmaktadır: Gerçek ismi (real name), so ismi (so name) ve linker ismi (linker name).

Kütüphanenin majör ve çift minör versiyonlu ismine gerçek ismi denilmektedir. Örneğin:

libsample.so.2.4.6

so ismi ise yalnızca majör numara içerisinde ismidir. Örneğin yukarıdaki gerçek ismin so ismi şöyledir:

libsample.so.2

Linker ismi ismi ise hiç versiyon numarası içermeyen ismidir. Örneğin yukarıdaki kütüphanelerin linker ismi ise şöyledir:

libsample.so

Tabii aslında kütüphaneden bir tane vardır. O da gerçek isimli kütüphanedir. so ismi gerçek isme sembolik link, linker ismi de so ismine sembolik link yapılır.

linker ismi ---> so ismi ---> gerçek ismi

Örneğin:

gcc -o libsample.so.1.0.0 -shared -fPIC a.c b.c c.c (gerçek isimli kütüphane dosyası oluşturuldu)

ln -s libsample.so.1.0.0 libsample.so.1 (so ismi oluşturuldu)

ln -s libsample.so.1 libsample.so (linker ismi oluşturuldu)

```
lrwxrwxrwx 1 csd study 14 Nov 26 22:32 libsample.so -> libsample.so.1
lrwxrwxrwx 1 csd study 18 Nov 26 22:31 libsample.so.1 -> libsample.so.1.0.0
-rwxr-xr-x 1 csd study 7552 Nov 26 22:30 libsample.so.1.0.
```

Dinamik kütüphanelerin linker isimleri link aşamasında kullanılan isimlerdir. Bu sayede link işlemini yapan programcılar daha az tuşa basarak

genel bir isim kullanması sağlanmıştır. Bu durumda örneğin biz libsample isimli kütüphane kullanan programı link etmek istersek şöyle yapabiliriz:

```
gcc -o sample sample.c libsample.so
```

Ya da şöyle yapabiliriz:

```
gcc -o sample sample.c -lsample -L.
```

Dinamik kütüphanelerin so isimleri aslında kütüphanenin tüm uyumlu (compatible) versiyonlarının ortak ismi gibi düşünülebilir.

Kütüphanelerin SO isimleri aslında aynı zamanda kütüphanelerin gerçek isimlerine ilişkin kütüphane dosyalarının içerisinde SONAME tag'ıyla yazılmaktadır. Yani dinamik kütüphane dosyalarının içerisinde SONAME isimli bir alan vardır. Bu SONAME alanı link işlemi sırasında -soname seneği ile doldurulur. Dinamik kütüphaneyi oluşturan programcı genel olarak bu SONAME alanına kütüphanenin soname'ini yazmalıdır. Örneğin:

```
gcc -o libsample.so.1.0.0 -Wl,-soname,libsample.so.1 -shared a.o b.o c.o
```

Tabii programcı kütüphane dosyasında bu SONAME alanına hiç bir şey yazmayabilir. Örneğin:

```
gcc -o libsample.so.1.0.0 -shared a.o b.o c.o
```

Bir kütüphane dosyasının SONAME alanı readelf -d seçeneği ile görüntülenebilir. (-d seçeneği ".dynamic" isimli bölümü görüntülemektedir. SONAME alanı bu bölüm (section) içerisinde.) Örneğin:

```
readelf -d libsample.so.1.0.0 | grep SONAME
```

Aynı işlemi objdump utility'si ile de şöyle yapabiliriz:

```
objdump -x libsample.so.1.0.0 | grep SONAME
```

Dinamik kütüphanelerin so isimleri yükleme sırasında gerçek yüklenecek dosyayı belirtmektedir. Yani örneğin libsample.so.1.0.0 dinamik kütüphane

dosyasının SONAME alanında libsample.so.1 ismi yazıyorsa aslında bu kütüphane yerine yükleyici libsample.so.1 isimli kütüphaneyi yükleyecektir. Tabii mademki

libsample.so.1 dosyası libsample.so.1.0.0 dosyasına sembolik link yapılmıştır. Bu durumda yükleyici yine libsample.so.1.0.0 dosyasını yükler. Şimd, bu işlemleri yeniden sırasıyla en baştan yapalım:

```
gcc -c -fPIC a.c b.c c.c
```

```
gcc -o libsample.so.1.0.0 -shared -Wl,-soname,libsample.so.1 a.o b.o c.o
```

```
ln -s libsample.so.1.0.0 libsample.so.1
```

```
ln -s libsample.so.1 libsample.so
```

Şimdi de kütüphaneyi kullanan sample isimli programı derleyip link edelim:

```
gcc -o sample sample.c libsample.so
```

Şimdi ldd utility'si ile sample programının kullandığı dinamik kütüphanelere bakalım:

```
linux-vdso.so.1 (0x00007fffd809b6000)
libsample.so.1 => ./libsample.so.1 (0x00007ff0f491d000)
libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007ff0f452c000)
/lib64/ld-linux-x86-64.so.2 (0x00007ff0f4d21000)
```

Görüldüğü gibi aslında sample programı libsample.so sembolik linkinin izlediği libsample.so.1.0.0 dosyasını değil bu dosyanın SONAME alanında yazılı olan libsample.so.1 dosyasını yüklemek istemektedir. Tabii neticede kütüphanelerin so isimleri gerçek isimlerine sembolik link yapıldığına göre aslında yine yükleyici libsample.so.1.0.0 dinamik kütüphanesini yükleyecektir.

Pekiye çalıştırılabilen bir dosyanın hangi dinamik kütüphaneleri kullandığı nerede yazmaktadır? İşte aslında bu bilgi çalıştırılabilen dosyanın içersinde .dynamic isimli bölümde DT_NEEDED taglarıyla belirtilir. ldd programı da aslında bu tag'lara bakmaktadır. Biz bu tagı yine istersek readelf -d ile görüntüleyebiliriz. Örneğin:

```
readelf -d sample | grep NEEDED
0x0000000000000001 (NEEDED)           Shared library: [libsample.so.1]
0x0000000000000001 (NEEDED)           Shared library: [libc.so.6]
```

İşte aslında dinamik kütüphane kullanan program link edilirken linker kütüphane dosyasının SONAME tag alanına bakıp orada yazan dosyayı çalıştırılabilen dosyanın NEEDED alanına yazmaktadır. Yükleyici aslında her zaman bu NEEDED alanındaki dinamik kütüphaneleri yüklemeye çalışmaktadır.

Pekiye kütüphanenin SONAME alanına bir şey yazılmazsa ne olur? İşte bu durumda linker link işlemi sırasında belirtilen dinamik kütüphane dosyasını çalıştırılabilen dosyanın NEEDED alanına yerleştirir. Örneğin:

```
gcc -c -fPIC a.c b.c c.c
gcc -o libsample.so.1.0.0 a.o b.o c.o
ln -s libsample.so.1.0.0 libsample.so.1
ln -s libsample.so.1 libsample.so
```

```
gcc -o sample sample.c libsample.so
```

Burada sample programının NEEDED tag'ı şöyle oluşturulacaktır:

```
readelf -d sample | grep NEEDED
x0000000000000001 (NEEDED)           Shared library: [libsample.so]
```

0x0000000000000001 (NEEDED)

Shared library: [libc.so.6]

Tabii sonuç olarak oluşturulan semblik linlerden dolayı yine yükleyici libsample.so.1.0.0 dosyasını yükleyecektir.

Peki bu durumda so isminin anlamı nedir? İşte eğer biz asıl isme ilişkin dosyanın so ismine yerleştirme yaparsak çalıştırılabilen dosya o isimdeki dosyası yüklemek isteyecektir. Biz de sembolik linkleri değiştirerek onun başka bir minör versyonu yüklemesini sağlayabiliriz. Yani örneğin yukarıdaki uygulamada sample programı aslında sample.so.1 dosyasını yüklemek istemektedir. Normalde de bu dosya sample.so.1.0.0 dosyasını göstermektedir. Şimdi biz bu dosyanın ileride örneğin sample.so.1.0.1 dosyasını göstermesini sağlayabiliriz. Artık sample programı çalıştırıldığında bu yeni versiyonu yüklemeye çalışacaktır.

so ismine ilişkin sembolik link çıkartma ve /etc/ld.so.cache dosyasının güncellenmesi işlemi ldconfig tarafından otomatik yapılmaktadır. Yani aslında biz kütüphanenin gerçek isimli dosyasını /lib ya da /usr/lib içerisinde yerleştirip ldconfig programını çalıştırdığımızda bu program zaten so ismine ilişkin sembolik linki de oluşturmaktadır.

Özetle Dinamik kütüphane kullanırken şu konvansiyona uymak iyi bir tekniktir:

- Kütüphane ismini lib ile başlatarak vermek
- Kütüphane ismine minör ve minör numara vermek
- Gerçek isimli kütüphane dosyasını oluştururken so ismi olarak -Wl,-soname seçeneği ile kütüphanenin so ismini yazmak
- Kütüphane için linker ismi ve so ismini sembolik link biçiminde oluşturmak
- Kütüphane paylaşılacaksa onu /usr/lib dizinine yerleştirmek ve ldconfig programı çalıştırarak /etc/ld.so.cache dosyasının güncellenmesini sağlamak. Bu işlem eğer kütüphaneyi /usr/lib içersine yerleştirmiyorsak yapılmayabilir.

-----*/
/
*-----

Dinamik kütüphane dosyaları program çalıştırıldıktan sonra çalışma zamanı sırasında belli bir noktada da yüklenebilir. Bu işlemin bazı avantajları şunlar olabilmektedir:

- Program daha hızlı yüklenebilir.
- Programın sanal bellek alanı gereksiz bir biçimde doldurulmayabilir. (Örneğin nadiren çalışılacak bir fonksiyon dinamik kütüphanede olabilir. Bu durumda o dinamik kütüphanenin işin başında yüklenmesi gereksiz bir yükleme zamanı ve bellek israfına yol açabilir.)

Dinamik kütüphanelerin dinamik yüklenmesi için `dlopen`, `dlsym`, `dlerror` ve `dlclose` fonksiyonlarıyla yapılmaktadır. Bu fonksiyonlarda `libdl` kütüphanesi içerisinde. Dolayısıyla link işlemi için `-ldl` seçeneğinin bulundurulması gerekir. `dlopen` fonksiyonun prototipi şöyledir:

```
void *dlopen(const char *filename, int flag);
```

Fonksiyonun birinci parametresi yüklenecek dinamik kütüphanenin yol ifadesini, ikinci parametresi seçenek belirten bayrakları almaktadır. Fonksiyon başarı durumunda kütüphaneyi temsil eden bir handle değerine başarısızlık durumunda `NULL` adrese geri dönmektedir. Başarısızlık durumunda fonksiyon `errno` değişkenini set etmez. Başarısızlığa ilikin yazı doğrudan `dlerror` fonksiyonuyla elde edilmektedir:

```
char *dlerror(void);
```

`dlopen` fonksiyonun birinci parametresindeki dinamik kütüphane isminde eğer hiç / karakteri yoksa bu durumda kütüphanenin aranması daha önce ele aldığımız prosedüre göre yapılmaktadır. Eğer dosya isminde en az bir / karakteri varsa dosya yalnızca bu mutlak ya da görelî yol ifadesinde aranmaktadır.

Dinamik yükleme sırasında yüklenecek kütüphanenin `SONAME` alanında yazılan isme hiç bakılmamaktadır. (Bu `SONAME` alanındaki isim yalnızca link aşamasında kullanılır)

Kütüphanenin adres alanından boşaltılması ise `dlclose` fonksiyonuyla yapılmaktadır:

```
int dlclose(void *handle);
```

Fonksiyon başarı durumunda 0, başarısızlık durumunda 0 dışı bir değere geri dönmektedir. Aynı kütüphane ikinci kez yüklenebilir. Bu durumda gerçek bir yükleme yapılmaz.

Ancak yüklenen sayıda `close` işleminin yapılması gerekmektedir.

Kütüphanenin içerisindeki fonksiyonlar ya da global nesneler adresleri elde edilerek kullanılırlar. Bunların adreslerini elde edebilmek için `dlsym` isimli

fonksiyon kullanılmaktadır:

```
void *dlsym(void *handle, const char *symbol);
```

Fonksiyon başarı durumunda ilgili sembolün adresine başarısızlık durumunda `NULL` adrese geri döner.

Aşağıda bir dinamik kütüphane dinamik olarak yüklenmiş ve oradan bir fonksiyon ve data adresi alınarak kullanılmıştır.

-----*/

```

#include <stdio.h>
#include <stdlib.h>
#include <dlfcn.h>

int main(void)
{
    void *dl;
    int (*padd)(int ,int);
    int result;
    int *pi;

    if ((dl = dlopen("libsamle.so", RTLD_NOW)) == NULL) {
        fprintf(stderr, "dlopen: %s\n", dlerror());
        exit(EXIT_FAILURE);
    }

    if ((* (void **)&padd = dlsym(dl, "add")) == NULL) {
        fprintf(stderr, "dlsym: %s\n", dlerror());
        exit(EXIT_FAILURE);
    }

    result = padd(10, 20);
    printf("%d\n", result);

    if ((pi = (int *)dlsym(dl, "x")) == NULL) {
        fprintf(stderr, "dlsym: %s\n", dlerror());
        exit(EXIT_FAILURE);
    }
    printf("%d\n", *pi);

    dlclose(dl);

    return 0;
}

```

/

*-----

Dinamik kütüphane dlopen fonksiyonuyla yüklenirken global değişkenlerin ve fonksiyonların nihai yükleme adresleri bu dlopen işlemi sırasında hesaplanabilir ya da onlar kullanıldıklarında hesaplanabilir. İkisi arasında kullanıcı açısından bir fark olmamakla birlikte tüm sembollerin adreslerinin yükleme sırasında hesaplanması bazen yükleme işlemini (eğer çok sembol varsa) uzatabilmektedir. Bu durumu ayarlamak için dlopen fonksiyonun flags parametresi kullanılır. Bu parametre RTLD_NOW olarak girilirse tüm sembollerin adresleri dlopen sırasında, RTLD_LAZY girilirse kullanıldıkları noktada hesaplanmaktadır. İki biçim arasında çoğu kez programcı için bir farklılık oluşmamaktadır. Ancak aşağıdaki örnekte bu iki biçimin ne anlama geldiği gösterilmektedir.

Aşağıdaki örnekte libmample.so kütüphanesindeki foo fonksiyonu gerçekte olmayan bir bar fonksiyonunu çağırmıştır. Bu fonksiyonun gerçekte olmadığı foo fonksiyonun sembol çözümlemesi yapıldığında anlaşılacaktır. İşte eğer bu kütüphaneyi kullanan sample.c programı kütüphaneyi RTLD_NOW ile yüklerse tüm semboller o anda çözülmeye çalışılacağından dolayı bar'ın bulunmuyor olması hatası da dlopen sırasında oluşacaktır. Eğer kütüphane RTLD_LAZY ile yüklenirse bu durumda sembol çözümlemesi foo2nun kullanıldığı noktada (yani dlsym fonksiyonunda) gerçekleşecektir. Dolayısıyla hata da o o noktada oluşacaktır.

```
-----*/  
/* mample.c */  
  
#include <stdio.h>  
  
void bar(void);  
  
void foo(void)  
{  
    bar();  
}  
  
/* sample.c */  
  
#include <stdio.h>  
#include <stdlib.h>  
#include <dlfcn.h>  
  
int main(void)  
{  
    void *dl;  
    void (*pf)(void);  
  
    if ((dl = dlopen("libmample.so", RTLD_LAZY)) == NULL) { /* RTLD_NOW ile  
        de deneyiniz */  
        fprintf(stderr, "dlopen: %s\n", dlerror());  
        exit(EXIT_FAILURE);  
    }  
  
    printf("Ok\n");  
  
    if ((* (void **) &pf = dlsym(dl, "foo")) == NULL) {  
        fprintf(stderr, "dlsym: %s\n", dlerror());  
        exit(EXIT_FAILURE);  
    }  
  
    pf();  
  
    dlclose(dl);  
  
    return 0;
```


}

/

*-----

Bazen bir dinamik kütüphane içerisindeki sembollerin o dinamik kütüphaneyi kullanan kodlar tarafından kullanılması istenmeyebilir. Örneğin dinamik kütüphanede bar isimli bir fonksiyon vardır. Bu fonksiyon bu dinamik kütüphanenin kendi içerisinde başka fonksiyonlar tarafından kullanılıyor olabilir. Ancak bu fonksiyonun dinamik kütüphanenin dışından kullanılması istenmeyebilir. (Bunun çeşitli nedenleri olabilir. Örneğin kapsülleme sağlamak için, dışarıdaki sembol çakışmalarını ortadan kaldırmak için vs.) İşte bunu sağlamak amacıyla gcc'ye özgü `__attribute__((...))` eklentisinden faydalanılmaktadır. `__attribute__((...))` eklentisi pek çok seçeneğe sahip platform spesifik bazı işlemlere yol açmaktadır. Bu eklentinin seçeneklerini gcc dokümanlarından elde edebilirsiniz. Bizim bu amaçla kullanacağımız `__attribute__((...))` seçeneği "visibility" isimli seçenektir.

Aşağıdaki örnekte bar fonksiyonu foo fonksiyonu tarafından kullanılmaktadır. Ancak kütüphanin dışından bu fonksiyonun kullanılması istenmemiştir.

Burada fonksiyon özelliğinin (yani `__attribute__` sentaksının) fonksiyon isminin hemen sola getirildiğine ve çift parantez kullanıldığına dikkat ediniz.

Burada kullanılan özellik "visibility" isimli özelliktir ve değeri "hidden" biçimde verilmiştir.

Default durumda dinamik kütüphanedeki bütün global semboller dışarıdan kullanılabilir. Buradaki `__attribute__((visibility("hidden")))` özelliğinin fonksiyonu static yapmakla aynı şey olmadığına dikkat ediniz. Fonksiyon static yapılırsa o dinamik kütüphanedeki diğer modüller tarafından da kullanılamamaktadır.

-----*/

/* mample.c */

#include <stdio.h>

void bar(void);

void foo(void)

{
 printf("foo\n");
 bar();

}

```
void __attribute__((visibility("hidden"))) bar(void)
{
    printf("bar\n");
}
```

```
/
```

```
*-----
```

Bir dinamik kütüphane normal olarak ya da dinamik olarak yüklendiğinde birtakım ilk işlerin yapılması gerekebilir. (Örneğin kütüphane thread güvenli olma iddiasındadır ve birtakım senkronizasyon nesnelerinin ve thread'e özgü alanların yaratılması gerekebilir.)

Bunun için __attribute__((constructor)) fonksiyon özelliği kullanılmaktadır. Benzer biçimde dinamik kütüphane programın adres alanından

boşaltılırken de birtakım son işlemler için __attribute__((destructor)) ile belirtilen fonksiyon çağrılmaktadır. (Aslında bu constructor ve destructor

fonksiyonları normal programlarda da kullanılabilir. Bu durumda ilgili fonksiyonlar main fonksiyonundan önce ve main fonksiyonundan sonra çağrılmaktadır.)

Dinamik kütüphane birden fazla kez yüklendiğinde yalnızca ilk yüklemede toplamda bir kez constructor fonksiyonu çağrılmaktadır. Benzer biçimde destructor fonksiyonu da yalnızca bir kez çağrılır.

```
-----*/
```

```
/* mample.c */
```

```
#include <stdio.h>
```

```
void __attribute__((constructor)) dynamic_init()
{
    printf("dinamik kütüphane kullanılıyor..\n");
}
```

```
void __attribute__((destructor)) dynamic_exit()
{
    printf("dinamik kütüphane boşaltılıyor...\n");
}
```

```
void foo(void)
{
    printf("foo\n");
}
```

```
/* sample.c */
```

```
#include <stdio.h>
#include <stdlib.h>
#include <dlfcn.h>
```

```
int main(void)
{
```

```

void *dl;

if ((dl = dlopen("libmample.so", RTLD_NOW)) == NULL) {
    fprintf(stderr, "dlopen: %s\n", dlerror());
    exit(EXIT_FAILURE);
}

printf("ok\n");

if ((dl = dlopen("libmample.so", RTLD_NOW)) == NULL) {
    fprintf(stderr, "dlopen: %s\n", dlerror());
    exit(EXIT_FAILURE);
}

dlclose(dl);

return 0;
}

```

```

/
*-----
-----
O anda makinemizdeki işletim sistemi hakkındaki bilgi uname komutuyla
elde edilebilir. Bu komut -r ile kullanılırsa o makinede yüklü olan
kernel versiyonu elde edilmektedir. Örneğin:

uname -r
4.15.0-20-generic
-----
-----*/

```

```

/
*-----
-----
Kernel'ın bir parçası gibi işlev gören, herhangi bir koruma engeline
takılmayan, kernel modda çalışan özel olarak hazırlanmış modüllere
(yani kod parçalarına) Linux dünyasında "çekirdek modülleri (kernel
modules)" denilmektedir. Çekirdek modülleri eğer kesme gibi bazı
mekanizmaları
kullanıyorsa ve bir donanım aygıtını yönetme iddiasındaysa bunlara özel
olarak "aygıt sürücüler (device drivers)" denilmektedir.

Genellikle bir Linux sistemini yüklediğimizde zaten kernel modüllerini
ve aygıt sürücülerini oluşturabilmek için gereken kütüphaneler ve başlık
dosyaları
zaten yüklü biçimde bulunmaktadır. Tabii programcı kernel kodlarını da
kendi makinesine indirmek isteyebilir. Bunun için aşağıdaki komut
kullanılabilir:

sudo apt-get install linux-source

```

Eğer sisteminizde Linux'un kaynak kodları yüklü ise bu kaynak kodlar /usr/src dizininde bulunmaktadır. Bu dizindeki linux-headers-\$(uname -r)

dizini kaynak kodlar yüklü olmasa bile bulunan bir dizindir ve bu dizin çekirdek modülleri ve aygıt sürücülerin "build edilmeleri" için gereken başlık dosyalarını barındırmaktadır. Benzer biçimde /lib/modules isimli dizinde \$(uname -r) isimli bir dizin vardır. Bu dizin çekerek modüllerinin build edilmesi için gereken bazı kodları ve kütüphaneleri bulundurmaktadır.

-----*/
/
*-----

Bir kernel modülünde biz user mod için yazılmış kodları kullanamayız. Çünkü orası bir dünyadır. Kernel modüllerinde biz yalnızca kernel içerisindeki bazı fonksiyonları kullanabiliriz. Bunlara "kernel tarafından export edilmiş fonksiyonlar" denilmektedir. "Kernel tarafından export edilmiş fonksiyon kavramıyla "sistem fonksiyonu" kavramının bir ilgisi yoktur. Sistem fonksiyonları user moddan çağrılmak üzere tasarlanmış ayrı bir grup fonksiyondur. Oysa kernel tarafından export edilmiş fonksiyonlar user moddan çağrılmazlar. Yalnızca kernel modüllerinden çağrılabilirler. Buradan çıkan sonuç şudur: Bir kernel modül yazılırken ancak kernel'ın export ettiği fonksiyon ve datalar kullanılabilir. Kernel kaynak kodları çok büyüktür buradaki kısıtlı sayıda fonksiyon export edilmiştir. Benzer biçimde programcının oluşturduğu bir kernel modül içerisindeki belli fonksiyonları da programcı export edebilir. Bu durumda bu fonksiyonlar da başka kernel modüllerinden kullanılabilirler.

-----*/
/
*-----

Mademki kernel modüller işletim sisteminin kernel kodlarındaki fonksiyon ve dataları kullanabiliyorlar o zaman kernel modüller o anda çalışılan kernel'ın yapısına da bağlı durumdadırlar. İşletim sistemlerinde "aygıt sürücü yazmak" ya da "kernel modül yazmak" biçiminde genel bir konu yoktur. Her işletim sisteminin kernel modül ve aygıt sürücü mimarisi diğerlerinden farklıdır. Dolayısıyla bu konu spesifik bir işletim sistemi için geçerli olabilecek oldukça platform bağımlı bir konudur. Hatta işletim sistemlerinde bazı versiyonlarda genel aygıt sürücü mimarisi bile değiştirilebilmektedir. Dolayısıyla eski aygıt sürücülerini yeni versiyonlarda çalışmamakta yenileri de eski versiyonlarda çalışmamaktadır. İşletim sisteminin yeni versiyonlarında kernel mimarisi de değiştirilmiş olabilmektedir. Dolayısıyla daha farklı kernel fonksiyonları eskilerinin yerlerini almış olabilir. Yeni birtakım başka fonksiyonlar export edilmiş olabilir.

```
-----*/
/
*-----
-----

Bir kernel modülünü derlemek ve link etmek maalesef sanıldığından daha
zordur. Herne kadar kernel modüller de ELF object dosları
iseler de bunlarda özel bazı bölümler (sections) bulunmaktadır.
Dolayısıyla bu modillerin derlenmesinde özel gcc seçenekleri devreye
sokulur.
Bunların link edilmeleri de bazı kütüphane dsoyalarının devreye
sokulmasıyla yapılır. Dolayısıyla bir kernel modülün "build edilmesi"
biraz
ayrıntılı bilgi gerektirmektedir. İşte kernel tasarımcıları bu sıkıcı
işlemleri kolaylaştırmak için özel "make dosyaları" düzenlemişlerdir.
Programcı
bu make dosyalarından faydalananarak build işlemini çok daha kolay
yapabilmektedir.

Kernel modüller için build işlemini yapan örnek bir Makefile aşağıdaki
gibi olabilir. Burada önce /lib/modules/$(uname -r)/builf dizinindeki
Makefile
çalıştırılmış onadan sonra çalışma bu dosyadan devam ettirilmiştir.
Özetle b Make dosyası "helloworld.c" isimli dosyanın derlenerek kernel
modül biçimind elink
edilmesini sağlamaktadır. Kernel modül birden fazla kaynak dosyadan
oluşturulabilir. Bu durumda obj-m += a.o b.o c.o... biçiminde tüm
dosyalar belirtilir. Ya da bunlar
teker teker şöyle de belirtilebilir:

obj-m += a.o
obj-m += b.o
obj-m += c.o
...

-----*/

# Makelfile

obj-m += helloworld.o

all:
    make -C /lib/modules/$(shell uname -r)/build M=${PWD} modules
clean:
    make -C /lib/modules/$(shell uname -r)/build M=${PWD} clean

/
*-----
-----

Tabii aslında make dosyası parametrik biçimde de oluşturabilmektedir.
Tabii bu durumda make programı çalıştırılırken bu parametrenin değeri de
belirtilmelidir. Örneğin:

make file=helloworld
```

-----*/

```
obj-m += $(file).o
```

```
all:
```

```
    make -C /lib/modules/$(shell uname -r)/build M=${PWD} modules
```

```
clean:
```

```
    make -C /lib/modules/$(shell uname -r)/build M=${PWD} clea
```

/

*-----

Tipik bir "helloworld" modülü şöyle oluşturulabilir. Bu modülü yukarıdaki make dosyası ile aşağıdaki gibi build yapmalısınız:

```
make file=helloworld
```

Kernel modüllerin yüklenmesi için insmod isimli program kullanılmaktadır. Tabii bu program sudo ile çalıştırılmalıdır. Örneğin:

```
sudo insmod helloworld.ko
```

Kernel modüller istenildiği zaman rmmod isimli programla boşaltılabilirler. Örneğin:

```
sudo rmmod helloworld.ko
```

-----*/

```
/* helloworld.c */
```

```
#include <linux/module.h>
```

```
#include <linux/kernel.h>
```

```
int init_module(void)
```

```
{  
    printk(KERN_INFO "Hello World...\n");
```

```
    return 0;
```

```
}
```

```
void cleanup_module(void)
```

```
{  
    printk(KERN_INFO "Goodbye World...\n");
```

```
}
```

/

*-----

En basit bir kernel modülde aşağıdaki iki temel dosya include edilmelidir:

```
#include <linux/module.h>
#include <linux/kernel.h>
```

Bu iki dosya /lib/modules/\$(uname -r)/build/include dizini içerisindedir. (Yani libc ve POSIX kütüphanelerinin başlık dosyalarının bulunduğu /usr/include içerisinde değildir.) Yukarıda açıklanan Make dosyası include dosyalarının bu zidinde aranmasını sağlamaktadır. Bir kernel modül yüklendiğinde kernel modül içeisinde belirlenmiş olan bir fonksiyon çağrılır (bu "constructor" gibi düşünülebilir.) Default çağrılacak fonksiyonun ismi init_module biçimindedir. Bu fonksiyonun geri dönüş değeri int türdendir ve parametresi yoktur. Fonksiyon başarı durumunda 0 değerine başarısızlık durumunda negatif hata koduna geri dönmelidir. Bu fonksiyon başarısızlıkla geri dönerse modülün yüklenmesinden vazgeçilmektedir. Benzer biçimde bir modül kernel alanından boşaltılırken de yine bir fonksiyon çağrılmaktadır. (Bu fonksiyon "destructor" gibi düşünülebilir.) Default çağrılacak fonksiyonun ismi cleanup_module biçimindedir. Bu fonksiyonun geri dönüş değeri ve parametresi void biçimdedir.

Kernel modüller tıpkı daemon'lar gibi ekrana değil log dosyalarına ileri yazarlar. Bunun için kernel içindeki printk isimli fonksiyon kullanılmaktadır. printk fonksiyonun genel kullanımı printf gibidir. Default durumda yeni kernel'larda bu fonksiyon mesajların /var/log/syslog dosyasına yazdırılması sağlamaktadır. printk fonksiyonun prototipi <linux/kernel.h> dosyası içerisindedir.

printk fonksiyonun örnek kullanımı şöyledir:

```
printk(KERN_INFO "This is test\n");
```

Mesajın solundaki KERN_XXX biçimindeki makrolar aslında bir string açımı yapmaktadır. Dolayısıyla yan yana iki string birleştirildiği için mesaj yazısının başında küçük bir alan bulunur. Bu alan (yani bu makro) mesajın türünü ve aciliyetini belirtmektedir. Tipik KERN_XXX makroları şunlardır:

```
KERN_EMERG
KERN_ALERT
KERN_CRIT
KERN_ERR
KERN_WARN
KERN_NOTICE
KERN_INFO
KERN_DEBUG
```

Bu makroların tipik biçimi şöyledir:

```
#define KERN_EMERG    "<0>"
#define KERN_ALERT    "<1>"
#define KERN_CRIT     "<2>"
#define KERN_ERR      "<3>"
#define KERN_WARNING  "<4>"
```

```
#define KERN_NOTICE    "<5>"
#define KERN_INFO      "<6>"
#define KERN_DEBUG     "<7>"
```

Aslında KERN_XXX makroları ile printk fonksiyonunu kullanmak yerine pr_XXX makroları da kullanılabilir. Şöyle ki:

```
printk(KERN_INFO "Hello World...\n");
```

ile

```
pr_info("Hello World...\n");
```

tamamen eşdeğerdir. Aşağıdaki makrolar bulunmaktadır:

```
pr_emerg
pr_alert
pr_crit
pr_err
pr_warning
pr_notice
pr_info
pr_debug
```

printk fonksiyonun yazdıklarını /var/log/syslog dosyasına bakarak görebiliriz. Örneğin:

```
tail /var/log/syslog
```

Ya da dmesg programı ile de aynı bilgi elde edilebilir.

Belli bir anda yüklenmiş olan modüller /proc/modules dosyasından elde edilebilir. Aslında lsmod isimli bir program daha vardır. lsmod zaten /proc/modules dosyasını okuyarak sonu basitleştirerek yazdırmaktadır.

```
-----*/
/
*-----
-----
Aslında init_module ve cleanup_module fonksiyonlarının ismi
değiştirilebilir. Fakat bunun için bildirimde bulunmak gerekir.
Bildirimde bulunmak için ise module_init(...) ve module_exit(...)
makroları kullanılmaktadır. Bu makrolar kaynak kodun herhangi bir yerine
yazılabilir. Ancak makro içerisinde belirtilen fonksiyonların
bildirimlerinin bu makroların yerleştiği yere kadar yapılmış olması
gerekmektedir.
Bu makrolar tipik olarak kaynak kodun sonuna yerleştirilmektedir.
-----*/

/* helloworld.c */

#include <linux/module.h>
```



```

#include <linux/kernel.h>

int helloworld_init(void)
{
    printk(KERN_INFO "Hello World...\n");

    return 0;
}

void helloworld_exit(void)
{
    printk(KERN_INFO "Goodbye World...\n");
}

module_init(helloworld_init);
module_exit(helloworld_exit)

/
*-----
-----
    Genellikle kernel modül içerisindeki global değişkenlerin ve
    fonksiyonların "internal linkage" yapılması tercih edilmektedir.
-----
-----*/

/* helloworld.c */

#include <linux/module.h>
#include <linux/kernel.h>

static int helloworld_init(void)
{
    printk(KERN_INFO "Hello World...\n");

    return 0;
}

static void helloworld_exit(void)
{
    printk(KERN_INFO "Goodbye World...\n");
}

module_init(helloworld_init);
module_exit(helloworld_exit);

/
*-----
-----
    Kernel modüllerde başlangıç ve bitiş fonksiyonlarında fonksiyon
    isimlerinin soluna __init ve __exit makroları getirilebilmektedir.
    Bu makrolar <linux/init.h> dosyası içerisinde yer almaktadır. __init ilgili
    fonksiyonu özel bir ELF bölümüne (section) yerleştirir. Modül
    yüklendikten sonra

```

bu bölüm kernel alanından atılmaktadır. __exit makrosu ise kernel'ın içine gömülmüş modüllerde fonksiyonun dikkate alınmayacağını (dolayısıyla hiç yüklenmeyeceğini) belirtir. Ancak sonradan yüklemelerde bu makronun bir etkisi yoktur.

```
-----*/

/* helloworld.c */

#include <linux/module.h>
#include <linux/kernel.h>

static int __init helloworld_init(void)
{
    printk(KERN_INFO "Hello World...\n");

    return 0;
}

static void __exit helloworld_exit(void)
{
    printk(KERN_INFO "Goodbye World...\n");
}

module_init(helloworld_init);
module_exit(helloworld_exit);

/
*-----
-----
insmod ile yüklediğimiz her modül için /sys/modules dizinin içerisinde
ismi modül ismiyle aynı olan bir dizin yaratılmaktadır.
/proc/modules dosyası ile bu dizini karıştırmayınız. /proc/modules
dosyasının satırları yüklü olan modüllerin isimlerini tutar.
Modüllere ilikin asıl önemli bilgiler kenel tarafından /sys/modules
dizininde tutulmaktadır.
-----*/

/
*-----
-----
Modüllere parametre geçirebiliriz. Parametre geçirme işlemi insmod ile
modül yüklenirken komut satırında modül isminden sonra
değişken=değer çiftleriyle yapılmaktadır. Örneğin:

sudo insmod ./helloworld number=10 msg="\nThis is a test\n"
values=10,20,30,40,50

Kernel modüllere bu biçimde değer aktarmak için module_param ve
module_param_array isimli makrolar kullanılır. Makronun üç parametresi
vardır:

module_param(name, type, perm)
```

name parametresi ilgili değişkenin ismini belirtmektedir. Bu ismin komut satırındaki argüman ismiyle aynı olması gerekir. type ilgili parametrenin türünü belirtir. Bu tür şunlardan biri olabilir: int, long, short, uint, ulong, ushort, charp, bool, invbool. perm ise /sys/modules/<modül ismi> dizininde yaratılacak olan parameters dizinin erişim haklarını belirtir. Bu makrolar global alanda herhangi bir yere yerleştirilebilir.

module_param_array makrosu da şöyledir:

```
define module_param_array(name, type, nump, perm)
```

Makronun birinci ve ikinci parametreleri yine değişken ismi ve türüdür. Tabii buradaki değişken isminin bir dizi ismi olarak girilmesi gerekmektedir.

Üçüncü parametre toplam kaç değerın modüle dizi biçiminde aktarıldığını belirten değişkenin adresini (ismini değil) alır. Son parametre yine erişim haklarını belirtmektedir.

Aşağıdaki örnekte üç parametre komut satırından kernel modüle geçirilmiştir. Komut satırındaki isimlerle programın içerisindeki değişken isimlerinin aynı olması gerekmektedir. Yazıların geçirilmesinde iki tırnaklar kullanılır. Ancak kabuk programının söz konusu yazıyı tek bir parametre olarak ele alabilmesi için yazının dışarıdan da tırnaklanması gerekmektedir. Kabuktaki iki tırnak ile tek tırnak arasında yalnızca küçük bir farklılık vardır. Dizi geçirirken yanlışlıkla boşluk karakteri kullanılmamalıdır.

Aşağıdaki programı şöyle bir örnekle çalıştırabilirsiniz:

```
sudo insmod helloworld.ko count=100 msg='"this is a test"'
values=10,20,30,40,50
```

```
-----*/
/* helloworld.c */
```

```
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/moduleparam.h>
```

```
static int count;
static char *msg;
static int values[5];
static int size;
```

```
module_param(count, int, S_IRUSR|S_IWUSR|S_IRGRP|S_IROTH);
module_param(msg, charp, S_IRUSR|S_IWUSR|S_IRGRP|S_IROTH);
module_param_array(values, int, &size, S_IRUSR|S_IWUSR|S_IRGRP|S_IROTH);
```

```
static int __init helloworld_init(void)
{
```

```

int i;

printk(KERN_INFO "Hello World...\n");
printk(KERN_INFO "count = %d, msg = %s\n", count, msg);

printk(KERN_INFO "values:\n");
for (i = 0; i < size; ++i)
    printk(KERN_INFO "%d\n", values[i]);

return 0;
}

static void __exit helloworld_exit(void)
{
    printk(KERN_INFO "Goodbye World...\n");
}

module_init(helloworld_init);
module_exit(helloworld_exit);

```

```

/
*-----
-----
Her kernel modül parametresi /sys/modules/<modül_ismi>/parameters
dizinin içerisinde bir dosyayla temsil edilir. Örneğin yukarıdaki
kodda biz bu dizin içerisinde count, msg ve values isimli üç dosya
görürüz.
-----
-----*/

```

```

/
*-----
-----
errno değişkeni aslında libc kütüphanesinin (standart C ve POSIX
kütüphanesi) içerisinde tanımlanmış bir değişkendir.
Kernel modda yani kernel'ın içerisinde errno isimli bir değişken yoktur.
Bu nedenle kerneldaki fonksiyonlar POSIX fonksiyonları gibi
başarısızlık durumunda -1 ile geri dönmeyiz. Başarısızlık durumunda
negatif errno değeri ile geri dönerler. Örneğin open POSIX
fonksiyonu sys_open kernel fonksiyonunu çağırdığında onun negatif bir
değerler geri dönüp dönmediğine bakar. Eğer sys_open
negatif errno değeriyle geri dönerse bu durumda bu değer pozitifliğini
errno değişkenine yerleştirip open -1 ile geri dönmektedir.
Kernel modül yazan programcılarında bu konvansiyona uyması iyi bir
tekniktir. Örneğin:

```

```

if (işler ters gitti)
    return -EINVAL;

```

POSIX arayüzünde adrese geri dönen fonksiyonlar genel olarak başarısızlık durumunda NULL adrese geri dönmektedir. Oysa kernel kodlarında adrese geri dönen fonksiyonlar başarısız olduklarında yine sanki bir adresmiş gibi negatif erro değerine geri dönmektedir.

Kernel kodlarındaki ERR_PTR isimli makro bir tamsayı değeri alıp onu adres türüne dönüştürmektedir. Bu nedenle adrese geri dönen fonksiyonlarda şöylesi kodlar görülebilir:

```
void *foo(void)
{
    ...
    if (işler ters gitti)
        return ERR_PTR(-EINVAL);
    ...
}
```

Bu işlemin tersi de PTR_ERR makrosuyla yapılmaktadır. Yani PTR_ERR makrosu bir adresi alıp onu tamsayıya dönüştürmektedir. Bu durumda kernel kodlarında adrese geri dönen fonksiyonların başarısızlığı aşağıdaki gibi kodlarla kontrol edilmektedir:

```
void *ptr;

ptr = foo();
if (PTR_ERR(ptr) < 0)
    return PTR_ERR(ptr)
```

Kernel modül programcılarının da buradaki konvansiyona uygun kod yazması iyi bir tekniktir.

Linux çekirdeğindeki EINVAL sembolik sabitleri POSIX arayüzündeki EINVAL sabitleriyle aynı değerdedir.

```
-----*/
/
*-----
```

Linux'ta bir kernel modül artık user mod'tan kullanılabilir hale getirildiyse buna "aygıt sürücüsü (device driver)" denir. Aygıt sürücüler open fonksiyonuyla bir dosya gibi açılırlar. Bu açma işleminden bir dosya betimleyicisi elde edilir. Bu dosya betimleyicisi read, write, lseek, close gibi fonksiyonlarda kullanılabilir. Aygıt sürücülere ilişkin dosya betimleyicileri bu fonksiyonlarla kullanıldığında aygıt sürücüsü içerisindeki belirlenen bazı fonksiyonlar çağrılmaktadır. Yani tersten gidersek biz örneğin read fonksiyonu çağrıldığında aygıt sürücümüzdeki belli bir fonksiyonun çalışmasını sağlayabiliriz. Böylece aygıt sürücüsü ile user mod arasında veri transferleri yine sanki dosyaymış gibi read, write gibi fonksiyonlarla yapılır. Pekiyi user mod'dan aygıt sürücümüzdeki herhangi bir fonksiyonu çağırabilir miyiz? Yanıt evet. Bunun için ioctl isimli POSIX fonksiyonu kullanılmaktadır. Aygıt sürücüsü içerisinde fonksiyonlara birer kod numarası atanır. Sonra ioctl fonksiyonunda bu kod numarası belirtilir. Böylece akış user mod'tan kernel moda geçerek belirlenen fonksiyonu kernel modda çalıştıracaktır.

Pekiye aygıt sürücülerini açmak için open fonksiyonunda yol ifadesi olarak (yani dosya ismi olarak) ne verilecektir? İşte aygıt sürücüler dosya sisteminde bir dizin girişiyle temsil edilir. O dizin girişi open ile açıldığında aslında o dizin girişinin temsil ettiği aygıt sürücüsü açılmış olur. İşte aygıt sürücülerini temsil eden dizin girişlerine "aygıt dosyaları (device files)" denilmektedir.

_____*/

/

*_____

_____*/

/

*_____

_____*/

/

*_____

_____*/

/

*_____

_____*/

/

*_____

_____*/

/

*_____

_____ *

/

*

/

*

/

*

/

*

/

*

/

*

/

*

/

*

/

*-----

-----*

/

*-----

-----*

/

*-----

-----*

/

*-----

-----*

/

*-----

-----*

/

*-----

-----*

/

*-----

-----*

/

*-----

-----*/

/
*-----

-----*/

/
*-----

-----*/

/
*-----

-----*/

/
*-----

-----*/

/
*-----

-----*/

/
*-----

-----*/

/
*-----

-----*/

/
*-----

-----* /

/

*-----

*/

* _____

*/

*_____

*_____

-----* /

/

*-----

-----*

/

*-----

-----*

/

*-----

-----* /

/

*

*/

/

*

*/

/

*

*/

/

*

*/

/

*

*/

/

*

*/

/

*

*/

/

*

*/

/

*-----

-----*

/

*-----

-----*

/

*-----

-----*

/

*-----

-----*

/

*-----

-----*

/

*-----

-----*

/

*-----

-----*

/

*-----

-----*/

/
*-----

-----*/

/
*-----

-----*/

/
*-----

-----*/

/
*-----

-----*/

/
*-----

-----*/

/
*-----

-----*/

/
*-----

-----*/

/
*-----

-----* /

/

*-----

*/

* _____

*/

*_____

*_____

-----* /

/

*-----

-----*

/

*-----

-----*

/

*-----

-----* /

/

*

*/

/

*

*/

/

*

*/

/

*

*/

/

*

*/

/

*

*/

/

*

*/

/

*

*/

/

*-----

-----*

/

*-----

-----*

/

*-----

-----*

/

*-----

-----*

/

*-----

-----*

/

*-----

-----*

/

*-----

-----*

/

*-----

-----*/

/
*-----

-----*/

/
*-----

-----*/

/
*-----

-----*/

/
*-----

-----*/

/
*-----

-----*/

/
*-----

-----*/

/
*-----

-----*/

/
*-----

-----*/

/
*-----

-----*/

/
*-----

-----*/

/
*-----

-----*/

/
*-----

-----*/

/
*-----

-----*/