UNIX/Linux Sistemlerinde Boru Haberleşmeleri

Kaan Aslan
28 Nisan 2012

1. Giriş

Boru haberleşmeleri yalınlığından dolayı en çok tercih edilen prosesler arası haberleşme yöntemlerindendir. Yöntem kendi içerisinde eş zamanlılığı da barındırdığından programcının ayrıca eş zamanlılık sorunuyla uğraşmasına gerek kalmaz. Boru haberleşmeleri ilk *UNIX* sistemlerinden beri neredeyse tüm *UNIX* türevi sistemlerce desteklenmiştir. Yöntem başta *Windows* olmak üzere pek çok işletim sisteminde de benzer biçimde uygulanmaktadır.

Boru (*pipe*) işletim sistemi tarafından çekirdek alanında organize edilen bir *FIFO* kuyruk sistemidir. Boru haberleşmesi tipik olarak boruya yazma yapan ve borudan okuma yapan iki proses arasında gerçekleştirilir. Yazan taraf boruya yazma yaptığında, okuyan proses onları aynı sırada elde eder. Yazma ve okuma işlemi ilk giren ilk çıkar (*FIFO*) prensibiyle gerçekleştirilir. Örneğin boruya yazan proses sırasıyla *b0*, *b1*, *b2*, ..., *bn byte*'larını yazdıysa, okuyan taraf da bu *byte*'ları *b0*, *b1*, *b2*, ..., *bn* sırasıyla elde edecektir. Boru haberleşmesinde yazan taraf okuyan tarafı, okuyan taraf da yazan tarafı beklemektedir. Boruların belli bir uzunlukları vardır. Blokeli modda borunun dolması durumunda yazan proses boruya yazma yapmaya çalışırsa, bloke oluşur ve boruda yer açılıp tüm *byte*'lar yazılana kadar proses blokede kalır. Başka bir deyişle, boru dolmuşsa yazan taraf boruda yer açılmasını beklemektedir. Benzer biçimde blokeli modda okuma sırasında da borudan okuma yapan proses boru boşsa karşı taraf boruya yazma yapana kadar blokede bekler. Blokeli modda borudan okuma ve boruya yazma işlemlerinin ayrıntıları izleyen bölümlerde ele alınmaktadır.

UNIX türevi sistemlerde borular geleneksel olarak tek yönlüdür. Borunun tek yönlü olması boru yaratıldığında tek bir haberleşme kanalının bulunuyor olması anlamına gelir. Böyle bir ortamda haberleşen *A* ve *B* prosesleri boruya yazma yaparsa bilgiler aynı kanala aktarılır. Benzer biçimde iki proses aynı borudan okuma yaparsa bilgiler aynı kanaldan alınır. Bu nedenle tek yönlü boru sistemi yalnızca tek yönlü (*half duplex*) bir haberleşme sunmaktadır.

Tek yönlü boru sistemlerinde proseslerden biri boruya yazarken diğeri borudan okuma yapmalıdır. Aksi taktirde bilgiler karışır. Halbuki çift yönlü boruların bulunduğu sistemlerde (*Windows* sistemlerini buna örnek verebiliriz) boru yaratıldığında borunun içerisinde iki ayrı kanal bulunur. A ve B proseslerinin haberleşmeleri sırasında *A* prosesinin boruya yazdığıyla *B* prosesinin yazdığı karışmaz. Çünkü yazılanlar boru içerisindeki farklı kanallara aktarılırlar. Bu sistemlerde *A* prosesinin boruya yazdığını *B* prosesi borudan okurken *B* prosesinin boruya yazdığını da *A* prosesi borudan okuyabilir. Yani çift yönlü boru sistemleri çift yönlü (*full duplex*) bir haberleşme sağlayabilmektedir.

UNIX türevi sistemlerdeki borular tek yönlü olduğuna göre bu sistemlerde boru yoluyla çift yönlü haberleşme ancak iki ayrı borunun yaratılmasıyla gerçekleştirilebilir.

UNIX/Linux sistemlerinde boru haberleşmeleri isimsiz (*unnamed/anonymous*) ve isimli (*named*) olmak üzere iki gruba ayrılmaktadır. İsimsiz borular yalnızca üst ve alt prosesler arasındaki haberleşmelerde, isimli borular ise aynı makinenin herhangi iki prosesi arasındaki haberleşmelerde kullanılabilirler. Bunun dışında isimli borular ile isimsiz boruların kullanımları arasında bir fark yoktur. Borudan okuma ve boruya yazma işlemleri her iki durumda da *read ve write* fonksiyonlarıyla yapılmaktadır. Biz bu bölümde önce isimsiz sonra isimli boru haberleşmelerini ele alacağız.

Boru haberleşmesinde proseslerden birinin boruya yazma yaparken diğerinin de okuma yapması gerektiğini belirtmiştik. Haberleşme normal olarak yazan tarafın boruyu kapatmasıyla sonlandırılmaktadır. Bu durumda okuyan taraf önce boruda kalan bilgileri okur, okuyacak bir şey kalmadığında, borunun yazan taraf tarafından kapatıldığını anlar ve işlemini sonlandırır.

2. read ve write Fonksiyonlarının Borulardaki Davranışları

Borular işletim sistemi tarafından kullanıcılara sanki birer dosyaymış gibi gösterilmektedir. Bu nedenle borulardan okuma *read* fonksiyonuyla, borulara yazma da *write* fonksiyonuyla yapılmaktadır. Yukarıda da belirttiğimiz gibi boru haberleşmeleri blokeli ya da blokesiz modda gerçekleştirilebilmektedir. Blokeli modda boru boş olduğunda okuma yapılırken, boru dolu olduğunda da yazma yapılırken bloke oluşur. Halbuki blokesiz modda boru boş da olsa dolu da olsa yazan ve okuyan taraflar bloke olmazlar. Boru haberleşmesini iyi anlayabilmek için *read* ve *write* fonksiyonlarının borular üzerindeki davranışlarının ayrıntılı bir biçimde ele alınmasının gerekli olacağını düşünüyoruz. *read* ve *write* fonksiyonlarının blokeli modda borularla çalışırken davranışları şöyledir:

1) *read* fonksiyonu ile blokeli modda borudan okuma yapılmak istendiğinde bu fonksiyon eğer boruda en az bir *byte* bilgi varsa blokeye yol açmadan okuyabildiği kadar bilgiyi okur ve okuyabildiği *byte* sayısı ile geri döner. Örneğin boruda halen 50 *byte* bulunuyor olsun, biz de *read* fonksiyonuyla 100 byte okumak isteyelim. Bu durumda *read* fonksiyonu blokeye yol açmadan 50 *byte*'ı okur ve 50 değeri ile geri döner. Eğer boruda hiç bilgi yoksa read fonksiyonu prosesi (*thread*'li ortamda ilgili *thread*'i) bloke ederek boruya bilgi yazılana kadar bekletir. Boruya en az 1 byte yazıldığında bloke çözülecek *ve read* fonksiyonu yine okuyabildiği kadar bilgiyi okuyarak geri dönecektir. Bunların dışında, *read* fonksiyonu eğer boruya yazma yapabilecek tüm betimleyiciler kapatılmışsa (yani artık boruya yazma yapabilecek hiçbir betimleyici bulunmuyorsa) sıfır değeri ile geri döner. *read* fonksiyonunun sıfır ile geri dönmesi yazan tarafın boruyu kapattığı anlamına gelir. Bu durumda okuyan tarafın da kendi boru betimleyicisini kapatarak haberleşmeyi sonlandırması gerekir.

2) Blokeli modda *write* fonksiyonu boruya tüm bilgiler yazılana kadar prosesi (*thread*'li ortamda ilgili *thread*'i) bloke ederek bekletir. Yani blokeli modda boruda tüm bilgilerin yazılacağı kadar yer açılıncaya kadar proses blokede kalır. Yeterli yer açıldığında *write* fonksiyonu işlemini tamamlayarak boruya yazdığı byte sayısı ile geri döner.

read ve *write* fonksiyonlarının blokesiz moddaki davranışı da şöyledir:

1) *read* fonksiyonu ile blokesiz modda okuma yapılırken eğer boruda en az bir *byte* varsa fonksiyon okuyabildiği kadar bilgiyi okur ve okuyabildiği byte sayısı ile geri döner. Eğer boruda hiç bilgi yoksa *read* fonksiyonu blokeye yol açmaz; başarısızlıkla geri döner ve *errno* değişkeni *EAGAIN* değeri ile doldurulur.

2) *write* fonksiyonu ile blokesiz modda boruya yazma yapılırken eğer boruda yeterince boş yer varsa fonksiyon tüm bilgiyi yazarak yazdığı *byte* sayısı ile geri döner. Eğer boruda tüm bilgileri yazacak kadar yer yoksa *write* fonksiyonu başarısız olur ve *errno* değişkeni *EAGAIN* değeriyle doldurulur.

3. İsimsiz Boru Haberleşmeleri

İsimsiz boru haberleşmesi yalnızca üst ve alt prosesler arasındaki haberleşmelerde kullanılabilen bir yöntemdir. Bu yöntemde proseslerden biri yazan taraf diğeri okuyan taraf olmak zorundadır. İsimsiz borularla haberleşmeler şu aşamalardan geçilerek yapılmaktadır:

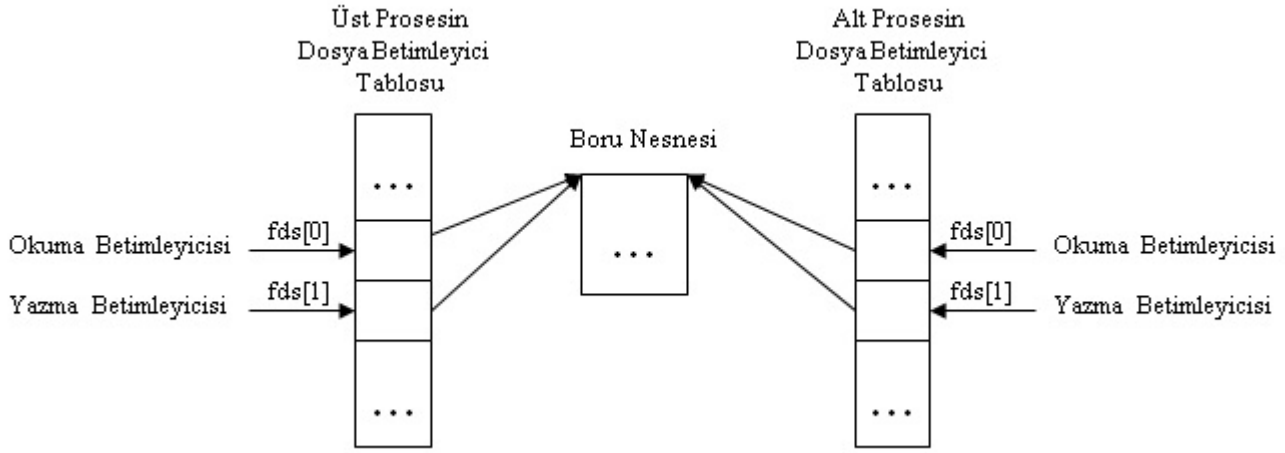
1) İsimsiz boru *pipe* fonksiyonuyla yaratılır. *pipe* fonksiyonunun prototipi şöyledir:[1]

```
#include <unistd.h>

int pipe(int filedes[2]);
```

pipe fonksiyonu isimsiz boruyu yaratarak borudan okuma ve boruya yazma yapmakta kullanılacak iki betimleyiciyi parametresi ile aldığı *int* türden diziye yerleştirir. Betimleyicilerden ilki (*filedes[0]* ile belirtilen) borudan okuma yapmak için, diğeri (*filedes[1]* ile belirtilen) boruya yazma yapmak için kullanılır. İşletim sistemi her boru için belirli bir boru alanı oluşturur. Boruların uzunlukları sistemden sisteme değişebilmektedir. Her sistemde o sisteme ilişkin boru uzunluğunun değeri *<limits.h>* içerisindeki *PIPE_BUF* sembolik sabitiyle elde edilebilir. *POSIX* standartlarına göre *PIPE_BUF* değeri aynı dosyada belirtilmiş olan *_POSIX_PIPE_BUF* değerinden küçük olamaz, fakat büyük olabilir. *pipe* fonksiyonu başarı durumunda sıfır değerine başarısızlık durumunda -1 değerine geri dönmektedir.

```
pid_t pid;
int fds[2];
...
if (pipe(fds) < 0) {
    perror("pipe");
    exit(EXIT_FAILURE);
}
...
```

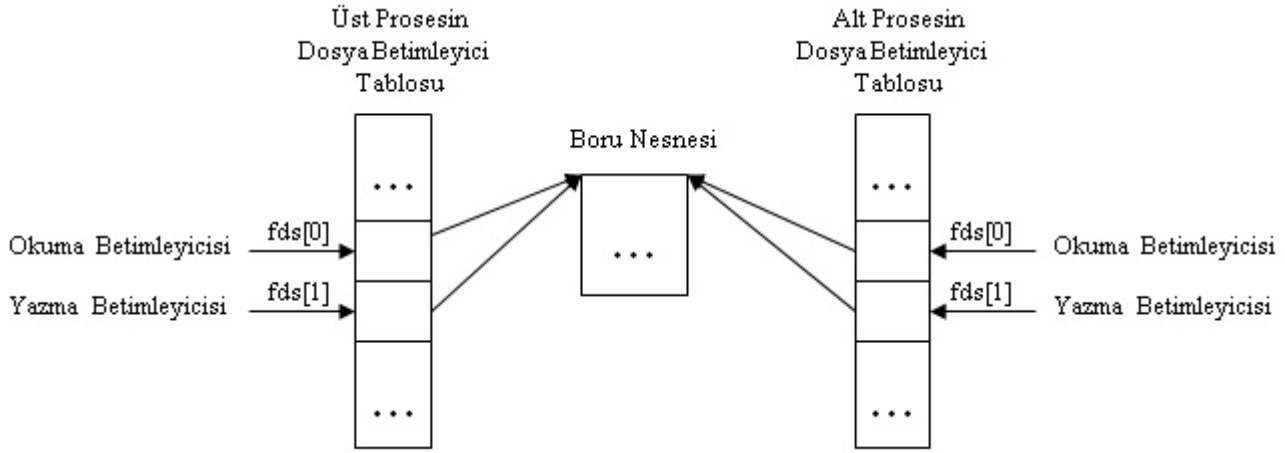
2) İsimsiz boru *pipe* fonksiyonuyla yaratıldıktan sonra *fork* işlemi ile haberleşme yapılacak alt proses yaratılır. Böylece boru betimleyicileri alt prosese de aktarılmış olacaktır. Bu durumu şekilsel olarak şöyle gösterebiliriz:



fork işleminden sonra üst ve alt proseslerde borudan okuma yapma potansiyelinde ve boruya yazma yazma potansiyelinde ikişer betimleyici (toplamda 4 tane) bulunur. Bu aşamada hangi prosesin borudan okuma yapacağına hangisinin boruya yazma yapacağına karar verilmesi gerekir. Çünkü okuma yapacak proses yazma betimleyicisini, yazma yapacak proses de okuma betimleyicisini kapatmalıdır. Burada amaç borudan okuma ve boruya yazma potansiyelinde olan yalnızca birer betimleyicinin kalmasını sağlamaktır. Örneğin üst prosesin yazma, alt prosesin okuma yapacağını düşünelim. Bu durumda üst proses yazma betimleyicisini, alt proses de okuma betimleyicisini kapatmalıdır.

```
...
if ((pid = fork()) < 0) {
    perror("fork");
    exit(EXIT_FAILURE);
}
if (pid) {
    close(fds[0]);     /* üst proses yazma yapacak */
    ...
}
else {
    close(fds[1]);     /* alt proses okuma yapacak */
    ...
}
...
```

Bu işlemden sonra elde edilen durumu şekilsel olarak aşağıdaki gibi gösterebiliriz:



3) Artık boruya yazan taraf *write* fonksiyonuyla yazma işlemini, borudan okuyan taraf da *read* fonksiyonuyla okuma işlemini gerçekleştirebilir. pipe fonksiyonu ile yaratılan boru betimleyicileri blokeli moddadır. Betimleyicilerin blokesiz moda geçirilmesi *fcntl* fonksiyonuyla yapılmaktadır.

Örneğin üst prosesin 0'dan 1000'e kadar sayıları boruya yazmak istediğini varsayalım. Bu işlem şöyle yapılabilir:

```
int i;
...
for (i = 0; i < 1000; ++i)
    if (write(fds[1], &i, sizeof(int) < 0) {
        perror("write");
        exit(EXIT_FAILURE);
    }

close(fds[1]);
...
```

Alt proses de bu değerleri şöyle okuyacaktır:

```
int result, val;
...
while ((result = read(fds[0], &val, sizeof(int))) > 0) {
    ...
}
if (result < 0) ) {
    perror("read");
    exit(EXIT_FAILURE);
}

close(fds[0]);
...
```

4) Haberleşme yazan tarafın boruyu kapatmasıyla sonlandırılır. Bu durumda okuyan taraf önce boruda kalanları okur, boru tamamen boşalınca read fonksiyonu sıfır ile geri döner ve okuyan taraf da boruyu kapatır.

Borudan *read* işlemi yapılırken eğer boruda hiç bilgi kalmamışsa ve boruya yazma potansiyelinde hiçbir betimleyici yoksa *read* fonksiyonunun sıfır ile geri döndüğünü anımsayınız. Örneğin boruda 50 byte varken yazan taraf boruyu kapatmış olsun. Okuyan tarafın 100 byte okumak istediğini düşünelim. Bu durumda *read* fonksiyonu borudakileri okuyacak ve 50 değeri ile geri dönecektir. Okuyan taraf bir kez daha okuma yapmak istediğinde artık *read* fonksiyonu sıfır ile geri dönecektir.

Şimdi yukarıda adım adım açıkladığımız işlemi küçük bir programla örneklendirelim:

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

void write_pipe(int fd);
void read_pipe(int fd);
void err_sys(const char *msg);

int main(int argc, char *argv[])
{
    int fdp[2];
    int pid;

    if (pipe(fdp) < 0)
        err_sys("pipe");

    if ((pid = fork()) < 0)
        err_sys("fork");

    if (pid) {    /* ust proses yazma yapacak */
        close(fdp[0]);
        write_pipe(fdp[1]);
        close(fdp[1]);
    }
    else {        /* alt proses okuma yapacak */
        close(fdp[1]);
        read_pipe(fdp[0]);
        close(fdp[0]);
    }

    if (wait(NULL) < 0)
        err_sys("wait");

    return 0;
}

void write_pipe(int fd)
{
    int i;

    for (i = 0; i < 10000; ++i)
        if (write(fd, &i, sizeof(int)) < 0)
            err_sys("write");
}

void read_pipe(int fd)
{
    int val;
    int result;
    while ((result = read(fd, &val, sizeof(int))) > 0)
        printf("%d\n", val);

    if (result < 0)
        err_sys("read");
}
```

```
void err_sys(const char *msg)
{
    perror(msg);
    exit(EXIT_FAILURE);
}
```

Şimdi de boru haberleşmesine ilişkin bazı ayrıntılar üzerinde durmak istiyoruz. Öncelikle boruya yazma yapan tarafın okuma betimleyicisini, okuyan tarafın da yazma betimleyicisini işin başında neden kapattığını açıklayalım. Eğer okuyan taraf yazma betimleyicisini kapatmazsa boruya yazma potansiyelinde olan iki betimleyici olur. Bu durumda aktarımın sonunda yazan taraf betimleyicisini kapattığında okuyan taraf *read* fonksiyonundan sıfır ile geri dönmeyecektir. (Çünkü hala boruya yazma potansiyelinde olan bir betimleyici vardır; o betimleyici de okuyan tarafın kapatmadığı yazma betimleyicisidir.) Böylesi bir durum haberleşmenin normal bir biçimde sonlandırılmasını engelleyecektir. Pekiyi, yazan taraf okuma betimleyicisini kapatmazsa ne olur? Bu durumda iletişimin sonlanmasına ilişkin bir sorun yaşanmasa da elde açık bir betimleyici kalacaktır. Bu betimleyicinin de işin başında kapatılması doğu tekniktir.

UNIX türevi sistemlerde boruların sistem tarafından belirlenmiş olan bir uzunlukları vardır. Boruların ilgili sistemdeki byte uzunluğu <*limits.h*> içerisindeki PIPE_*BUF* sembolik sabiti ile elde edilebilir. Boruya yazma yaparken bu değeri aşmamaya özen göstermemiz gerekir. Çünkü *PIPE_BUF* miktarından daha fazla yazma yapıldığında iç içe geçme durumu oluşabilmektedir. İç içe geçme farklı proseslerin aynı boruya yazması durumunda yazılan bilgilerin birbirleriyle karışabilmesi durumunu anlatır. Örneğin iki farklı prosesin (ya da bir prosesin iki farklı *thread*'inin) aynı boruya *write* fonksiyonuyla 10'ar byte yazdığını düşünelim. Normal olarak bu bilgiler ardışık bir biçimde iç içe geçmeden yazılır. İşte böylesi bir durumda yazılacak miktar *PIPE_BUF* değerinden büyükse iç içe geçme oluşabilmektedir.

Boru haberleşmesinin sonlandırılması için önce yazan tarafın boruyu kapatması gerekir. Eğer önce okuyan taraf boruyu kapatırsa diğer tarafın boruya yazma yapması sırasında *SIGPIPE* isimli sinyal oluşur. *SIGPIPE* sinyalinin varsayılan davranışı prosesin sonlandırılmasıdır. *SIGPIPE* sinyali bloke edilirse ya da dikkate alınmazsa (*ignore* edilirse) bu durumda *SIGPIPE* sinyali oluşmaz, *write* fonksiyonu başarısız olur ve *errno* değişkeni *EPIPE* değeri ile doldurulur. Okuyan tarafın olmadığı bir boruya yazma yapılması durumunda yazma yapan prosesin sistem tarafından sonlandırılmak istendiğine dikkat ediniz.

Şimdi üst prosesin alt prosese bir metin dosyasının içeriğini boru yoluyla ilettiği bir örnek verelim:

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

#define BUFSIZE 32

void do_parent_proc(int fdp, int fdf);
void do_child_proc(int fdp);
void err_sys(const char *msg);
```

```
int main(int argc, char *argv[])
{
    int fdf, fdps[2];
    pid_t pidChild;

    if (argc != 2) {
        fprintf(stderr, "eksik ya da fazla sayida arguman girildi! ");
        exit(EXIT_FAILURE);
    }

    if ((fdf = open(argv[1], O_RDONLY)) < 0)
        err_sys("open");

    if (pipe(fdps)< 0)
        err_sys("pipe");

    if ((pidChild = fork()) < 0)
        err_sys("fork");

    if (pidChild > 0) {
        close(fdps[0]);
        do_parent_proc(fdps[1], fdf);
        close(fdps[1]);
        wait(NULL);
    }
    else {
        close(fdps[1]);
        do_child_proc(fdps[0]);
        close(fdps[0]);
    }

    close(fdf);

    return 0;
}

void do_parent_proc(int fdp, int fdf)
{
    char buf[BUFSIZE];
    ssize_t n;

    while ((n = read(fdf, buf, BUFSIZE)) > 0)
        if (write(fdp, buf, n) < 0)
            err_sys("pipe");
    if (n < 0)
        err_sys("read");
}

void do_child_proc(int fdp)
{
    ssize_t n;
    char buf[BUFSIZE];

    while ((n = read(fdp, buf, BUFSIZE)) > 0)
        if (write(STDOUT_FILENO, buf, n) != n)
            err_sys("write");
    if (n < 0)
        err_sys("read");
}
```

```
void err_sys(const char *msg)
{
    perror(msg);
    exit(EXIT_FAILURE);
}
```

Örneğimizde üst proses gönderici alt proses alıcı konumdadır. Üst prosesin boruyu yaratıp *fork* işlemini uyguladığını görüyorsunuz:

```
...
if (pipe(fdps)< 0)
    err_sys("pipe");

if ((pidChild = fork()) < 0)
    err_sys("fork");
...
```

fork işleminden sonra üst proses okuma betimleyicisini, alt proses de yazma betimleyicisini kapatmıştır. Aktarım *do_parent_proc* ve *do_child_proc fonksiyonları* tarafından yapılmaktadır:

```
if (pidChild > 0) {
    close(fdps[0]);
    do_parent_proc(fdps[1], fdf);
    close(fdps[1]);
    wait(NULL);
}
else {
    close(fdps[1]);
    do_child_proc(fdps[0]);
    close(fdps[0]);
}
```

Aktarım bittikten sonra iki proses de açık olan diğer betimleyicisini kapatmıştır. Programın sonunda hem üst hem de alt prosesin *fdf* betimleyicisi ile belirtilen dosyayı kapattığına dikkat ediniz. Her ne kadar bu dosya yalnızca üst proses tarafından kullanılıyorsa da *fork* işleminden önce açıldığı için betimleyicisi alt prosese aktarılmıştır. (Tabii açılmış dosyaları kapatmak zorunda değiliz. Proses sonlandığında zaten açık dosyalar sistem tarafından otomatik olarak kapatılmaktadır.)

Yukarıdaki örnekte üst proses boruya yazan taraf, alt proses ise borudan okuyan taraf durumundadır. Tabi bunun tam tersi de söz konusu olabilirdi. Yani alt proses boruya yazan taraf, üst proses ise borudan okuyan taraf olabilirdi. Daha önceden de belirttiğimiz gibi boru tek kanallı bir iletişim sunmaktadır. Tek bir boruyla her iki prosesin de birbirlerine bilgi gönderip alması yazılımsal olarak mümkün olsa da uygulamada kullanılabilecek denli etkin bir yöntem değildir. Eğer üst ve alt prosesler arasında iki yönlü bir haberleşme yapılacaksa iki ayrı boru kullanılması doğru tekniktir.

Şimdi komut satırı üzerinde kabuk programları yoluyla yapılan boru işlemlerine göz gezdirelim. Bildiğiniz gibi komut satırında kullanılan ‘|’ karakterinin özel bir anlamı vardır. Bu karakterin solunda ve sağında çalıştırılabilen iki program dosyası bulunur. Böylesi bir durumda kabuk bir boru yaratarak bu iki programı da çalıştırır. Sonra sol taraftaki programın *stdout* dosyasını yazma amaçlı olarak, sağ taraftaki programın da *stdin* dosyasını okuma amaçlı olarak boruya yönlendirir. Böylece sol taraftaki programın ekrana yazdıklarını sağ taraftaki program klavyeden okuyormuş gibi elde eder. Örneğin:

```
study@linux-sdhv:~> ls | wc
     18      18     161
```

Burada *ls* programının *stdout* dosyasına yazdığı şeyleri, *wc* programı *stdin* dosyasından okuyacaktır. *wc* (*word count*) programının bir metin dosyasında kaç satır, kaç sözcük ve kaç karakter bulunduğu bilgisini *stdout* dosyasına yazdırdığını anımsayınız. Normal olarak *wc* programı işlem yapacağı dosyayı komut satırı argümanıyla alır. Örneğin:

```
study@linux-sdhv:~> ls > test.txt
study@linux-sdhv:~> wc test.txt
 18  18 161 test.txt
```

Burada önce *ls* komutu ile dizin içeriği *test.txt* isimli bir dosyaya aktarılmış, sonra da *wc* programı ile bu dosyanın içerisindeki satır, sözcük ve karakterler elde edilmiştir. Pek çok *UNIX/Linux* komutunda olduğu gibi *wc* komutunda da işlem yapılacak dosya ismi belirtilmemişse bilgi *stdin* dosyasından alınmaktadır. Örneğin:

```
study@linux-sdhv:~> wc
ali
veli
selami
      3       3      16
```

ls | *wc* örneğinde *ls* komutunun çıkışı *stdout* dosyasına yazdığına, *wc* komutunun da girişi *stdin* dosyasından okuduğuna dikkat ediniz. *a* | *b* gibi bir komut verildiğinde boru işlemi kabuk tarafından (bazı ayrıntıları göz ardı ediyoruz) şöyle yapılmaktadır:

1) Kabuk tarafından isimsiz bir boru yaratılır.

2) *a* ve *b* prosesleri için *fork* işlemi yapılır.

3) *a* programı için yaratılan alt prosesinin *1* numaralı (*STDOUT_FILENO*) dosya betimleyicisi yazma amacıyla, *b* programı için yaratılan alt prosesinin de *0*numaralı *(STDIN_FILENO)* betimleyicisi okuma amacıyla *dup2* fonksiyonu ile boruya yönlendirilir. Yönlendirme işleminden sonra alt proseslerde boru betimleyicileri kapatılır.

4) Yaratılan alt prosesler için *exec* işlemi uygulanır.

5) Üst prosese ilişkin boru betimleyicileri kapatılır ve *wait* fonksiyonlarıyla alt proseslerin sonlanması beklenir.

Yukarıda açıkladığımız adımları uygulayarak boru yönlendirmesi yapan örnek bir programı aşağıda veriyoruz:

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

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

```
pid_t exec_pipe(const char *filename, const int fdps[2], int rw)
{
    pid_t pid;

    if ((pid = fork()) < 0)
        err_sys("fork");

    if (pid == 0) {
        if (dup2(fdps[rw], rw) == -1)
            err_sys("dup2");

        close(fdps[0]);
        close(fdps[1]);

        if (execlp(filename, filename, (char *) NULL) == -1)
            err_sys("execlp");
    }

    return pid;
}

int main(int argc, char *argv[])
{
    pid_t pid1, pid2;
    int fdps[2];

    if (argc != 3) {
        fprintf(stderr, "kullanim: %s <prog1> <prog2> \n", argv[0]);
        exit(EXIT_FAILURE);
    }

    if (pipe(fdps) == -1)
        err_sys("pipe");

    pid1 = exec_pipe(argv[1], fdps, 1);
    pid2 = exec_pipe(argv[2], fdps, 0);

    close(fdps[0]);
    close(fdps[1]);

    if (waitpid(pid1, NULL, 0) < 0)
        err_sys("waitpid");

    if (waitpid(pid2, NULL, 0) < 0)
        err_sys("waitpid");

    return 0;
}

void err_sys(const char *msg)
{
    perror(msg);
    exit(EXIT_FAILURE);
}
```

Programı iki komut satırı argümanıyla şöyle çalıştırmalısınız:

```
<./isim> <prog1> <prog2>
```

Program *prog1* ve *prog2* ile belirtilen programları çalıştırır, *prog1* programının *stdout* dosyasına yazdıklarını *prog2* programının *stdin* dosyasından okumasını sağlar. Programımızda *main* fonksiyonunda önce borunun yaratıldığını görüyorsunuz. *exec_pipe* isimli fonksiyon *fork* fonksiyonunu uygulayıp alt prosesi yaratmakta, yönlendirmeyi yaparak *execlp* fonksiyonuyla ilgili programı çalıştırmaktadır:

```
pid_t exec_pipe(const char *filename, const int fdps[2], int rw);
```

Fonksiyonun birinci parametresi çalıştırılacak program dosyasının ismini, ikinci parametresi yaratılmış olan boruya ilişkin betimleyicilerin bulunduğu betimleyici dizisinin adresini alır. Son parametre boruya yönlendirilecek betimleyiciyi belirlemekte kullanılmaktadır. Eğer bu parametre için *0* değeri girilirse çalıştırılan programın *stdin* betimleyicisi, *1* değeri değeri girilirse *stdout* betimleyicisi boruya yönlendirilir. Üst prosesin betimleyicilerinin *exec_pipe fonksiyonunun* sonunda kapatıldığına dikkat ediniz.

4. İsimli Boru Haberleşmeleri

Önceki bölümde gördüğümüz isimsiz borular ile yalnızca aynı kökten gelen prosesler arasında haberleşme yapabiliriz. Çünkü yaratılan borulara ilişkin betimleyiciler ancak *fork* işlemiyle alt prosesler tarafından kullanılabilir duruma gelmektedir. Halbuki isimli borularla aynı kökten gelmeyen farklı prosesler arasında haberleşmeler yapılabilmektedir. İsimli borulara *UNIX* türevi sistemlerde *fifo* (*first in first out* sözcüklerinden kısaltılmıştır) da denilmektedir. İsimli borular işletim sistemi tarafından sanki birer dosyaymış gibi ele alındığından bunlar üzerinde işlemler yapmak oldukça kolaydır. İsimli borularla işlemler tipik olarak şu adımlardan geçilerek gerçekleştirilmektedir:

1) İsimli boru proseslerin biri tarafından *mkfifo* fonksiyonuyla yaratılır. *mkfifo* fonksiyonunun prototipi şöyledir:

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

int mkfifo(const char *path, mode_t mode);
```

Fonksiyonun birinci parametresi boruya ilişkin yol ifadesini belirtir. İsimli borular tamamen normal dosyalar gibi herhangi bir dizinde yaratılabilirler. İsimli boruların da erişim hakları vardır. Fonksiyonun ikinci parametresi isimli borunun erişim haklarını belirtir. Burada belirtilen erişim hakları prosesin umask değerinden etkilenmektedir. Fonksiyon başarı durumunda sıfır değerine, başarısızlık durumunda -1 değerine geri döner. Başarısızlık durumunda errno değişkeninin alabileceği değerlerden bazıları şunlardır:

EACCES	:	Yol bileşenlerinden birine ilişkin dizinlere erişim hakkı yoktur.
EEXIST	:	Belirtilen dosya zaten vardır.
ENOENT	:	Yol bileşenlerinden birine ilişkin dizin yoktur ya da yol ifadesi boş bir *string*'ten oluşmaktadır.
...	:	...

Örneğin:

```
if (mkfifo("testfifo", S_IRUSR|S_IWUSR|S_IRGRP|S_IROTH) < 0) {
    perror("mkfifo")
    exit(EXIT_FAILURE);
}
```

İsimli borular kabuk üzerinden *mkfifo* ya da *mknod* komutuyla da yaratılabilirler. *mkfifo* komutunun tipik kullanımı şöyledir:

```
study@linux-sdhv:~> mkfifo -m 644 testfifo
```

Komutun *–m* parametresi erişim haklarını belirlemekte kullanılmaktadır. *mknod* komutu isimli boruların dışında aygıt dosyaları yaratmak için de kullanılan genel bir komuttur. Aynı boru *mknod* komutuyla aşağıdaki biçimde yaratılabilirdi:

```
study@linux-sdhv:~> mknod -m 644 testfifo p
```

Komutun sonundaki *p*, isimli boru oluşturulacağını belirtir. *mkfifo* ve *mknod* komutlarının ayrıntılarını *man* sayfalarından inceleyebilirsiniz. İsimli borular *ls –l kabuk* komutunda *'p'* dosya türü ile gösterilirler. Örneğin:

```
study@linux-sdhv:~> mkfifo -m 644 testfifo
study@linux-sdhv:~> ls -l testfifo
prw-r--r-- 1 study users 0 2011-02-11 13:58 testfifo
```

2) Her iki proses de isimli boruyu *open* fonksiyonuyla açar. İsimli borular da tek taraflı bir haberleşme sunduğuna göre boruya yazacak prosesin boruyu O_*WRONLY* modunda , borudan okuma yapacak prosesin de boruyu *O_RDONLY* modunda açması gerekir. İsimli borular *open* fonksiyonuyla açılırken O_*CREAT* ya da *O_CREAT*|*O_EXCL* bayrakları kullanılmaz. Yalnızca *O_RDONLY*, *O_WRONLY* ya da *O_RDWR* bayrakları kullanılabilir. Örneğin boruyu okuma amaçlı aşağıdaki gibi açabiliriz:

```
int fdp;
...
if ((fd = open("testfifo", O_RDONLY)) < 0) {
    perror("open");
    exit(EXIT_FAILURE);
}
```

Yazma amaçlı olarak da şöyle açabiliriz:

```
int fdp;
...
if ((fdp = open("testfifo", O_WRONLY)) < 0) {
    perror("open");
    exit(EXIT_FAILURE);
}
```

Eğer boru blokeli modda *open* fonksiyonu ile *O_RDONLY* modunda açılmak istenirse, en az bir proses boruyu yazma yapma niyeti ile (yani *O_WRONLY ya* da *O_RDWR* modunda) açana kadar *open* fonksiyonu blokeye neden olur. Benzer biçimde eğer boru *O_WRONLY* modunda açılmak istenirse, boruyu başka bir proses okuma niyetiyle (yani *O_RDONLY* ya da *O_RDWR* modunda) açana kadar *open* fonksiyonu blokeye neden olmaktadır. Örneğin haberleşme sırasında isimli boruyu önce okuyan taraf *O_RDONLY* modunda açmak istesin. Bu durumda diğer proses boruyu *O_WRONLY* ya da *O_RDWR modunda* açana kadar *open* fonksiyonu geri dönmeyecektir. Okuyan bir prosesin olmaması durumunda boruya yazma yapılmasının *SIGPIPE* sinyalinin oluşmasına yol açtığını anımsayınız. İsimli borularda açış sırasında böyle bir durumla

karşılaşılmayacaktır. Eğer açış sırasında bloke oluşmasaydı yazan tarafa ilişkin prosesin önce çalıştırılması soruna yol açardı değil mi? İsimli borunun *O_RDWR* modunda açılması geçerli bir işlemdir. Bu durumda aynı proses boruya hem yazma hem de okuma yapma potansiyelinde olduğu için bloke oluşmaz. Tabi isimli borunun *O_RDWR* modunda açılmasına ilişkin anlamlı örnekler bulmanın da zor olduğunu belirtmeliyiz.

3) Boruya yazan taraf *write* fonksiyonu kullanarak, borudan okuyan taraf da *read* fonksiyonunu kullanarak yazma ve okuma işlemlerini gerçekleştirir. İsimli borular açıldıktan sonra artık geri kalan işlemler tamamen isimsiz borularda olduğu gibidir.

4) Yazan taraf boruyu kapatır. Okuyan taraf önce boruda kalanları okur. Boruda bilgi kalmayınca *read* fonksiyonu sıfır ile geri döner. Böylece okuyan taraf da boruyu kapatır. Gördüğünüz gibi isimli borularda da boruyu önce yazan tarafın kapatması gerekir. Okuma potansiyelinde hiçbir prosesin olmaması durumunda yazan taraf yazmaya çalıştığında yine *SIGPIPE* sinyali oluşmaktadır.

5) Boru *remove* ya da *unlink* fonksiyonları ile proseslerden biri tarafından silinir. Silme işlemi kabuk üzerinden herhangi bir zaman normal dosyalarda olduğu gibi *rm* komutuyla da yapılabilir.

İsimli borular tamamen normal dosyalar gibi dosya sisteminde kalıcı bir biçimde yer alırlar. Örneğin *mkfifo* fonksiyonuyla isimli boru yaratmış olalım. Sistemi kapatıp yeniden açtığımızda boru dosyası diskte bulunmaya devam edecektir. İsimli boruların uzunlukları her zaman 0 olarak görüntülenir. İsimli borular *stat* yapısının *st_mode* elemanına bakılarak tespit edilebilirler.

Şimdi isimli boruya ilişkin basit bir örnek vermek istiyoruz. Örneğimizde üç ayrı program bulunuyor. pipewrite isimli program isimli boruya yazma yaparken, piperead isimli program isimli borudan okuma yapmaktadır. pipeop ise isimli boruyu yaratmak ve silmek için kullanılan programdır:

```
/* pipeop.c */

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

void err_sys(const char *msg);
void err_usr(const char *msg);

int main(int argc, char *argv[])
{
    int ch, cflag, rflag;

    opterr = 0;
    cflag = 0, rflag = 0;

    while ((ch = getopt(argc, argv, "cr")) != -1) {
        switch (ch) {
            case 'c':
                cflag = 1;
                break;
            case 'r':
                rflag = 1;
                break;
            case '?':
                err_usr("gecersiz secenek!\n");
        }
    }
```

```
    if (!cflag && !rflag)
        err_usr("-c ya da -r seçeneklerinden biri girilmeli!\n");

    if (cflag && rflag)
        err_usr("-c ve -r seçenekleri birlikte girilemez!\n");

    if (argc - optind != 1)
        err_usr("yanlis sayıda argüman girildi!\n");

    if (cflag) {
        if (mkfifo(argv[optind], S_IRUSR|S_IWUSR|S_IRGRP|S_IROTH) < 0)
            err_sys("mkfifo");
    }
    else
        if (unlink(argv[optind]) < 0)
            err_sys("unlink");

    return 0;
}

void err_sys(const char *msg)
{
    perror(msg);
    exit(EXIT_FAILURE);
}

void err_usr(const char *msg)
{
    fprintf(stderr, "%s", msg);
    exit(EXIT_FAILURE);
}
```

Gördüğünüz gibi programın *–c* ve *–r* biçiminde iki seçeneği vardır. *–c* boruyu yaratmak için, *-r* de silmek için kullanılır. Programı aşağıdaki gibi derleyip çalıştırabilirsiniz:

```
study@linux-sdhv:~> gcc -o pipeop pipeop.c general.c
study@linux-sdhv:~> ./pipeop -c testpipe
```

Boruya yazma yapan *pipewrite* programını aşağıda veriyoruz. Program boru ismini komut satırından alıyor ve döngü içerisinde 100'e kadar sayıları boruya yazıyor:

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

void err_sys(const char *msg);
void err_usr(const char *msg);
```

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

    if (argc != 2)
        err_usr("yanlis sayida arguman girildi!\n");

    if ((fd = open(argv[1], O_WRONLY)) < 0)
        err_sys("open");

    for (i = 0; i < 100; ++i)
        if (write(fd, &i, sizeof(int)) < 0)
            err_sys("write");

    close(fd);

    return 0;
}

void err_sys(const char *msg)
{
    perror(msg);
    exit(EXIT_FAILURE);
}

void err_usr(const char *msg)
{
    fprintf(stderr, "%s", msg);
    exit(EXIT_FAILURE);
}
```

Programı aşağıdaki gibi derleyip çalıştırabilirsiniz:

```
study@linux-sdhv:~>gcc -o pipewrite pipewrite.c
study@linux-sdhv:~>./pipewrite testpipe
```

Programı çalıştırdığınızda akış *open* fonksiyonunda bloke olacaktır. Blokeden ancak borudan okuma yapan bir programın çalıştırılmasıyla çıkılabilir. Borudan okuma yapan *piperead.c* isimli program da şöyle yazılabilir:

```
/* piperead.c */

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

void err_sys(const char *msg);
void err_usr(const char *msg);
```

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

    if (argc != 2)
        err_usr("yanlis sayida arguman girildi!\n");

    if ((fd = open(argv[1], O_RDONLY)) < 0)
        err_sys("open");

    while ((result = read(fd, &val, sizeof(int))) > 0)
        printf("%d\n", val);

    if (result < 0)
        err_sys("read");

    close(fd);

    return 0;
}

void err_sys(const char *msg)
{
    perror(msg);
    exit(EXIT_FAILURE);
}

void err_usr(const char *msg)
{
    fprintf(stderr, "%s", msg);
    exit(EXIT_FAILURE);
}
```

Programı aşağıdaki gibi derleyerek çalıştırabilirsiniz:

```
study@linux-sdhv:~>gcc -o piperead piperead.c
study@linux-sdhv:~>./piperead testpipe
```

Burada bir nokta üzerinde durmak istiyoruz. Denemeyi yaparken *piperead* ve *pipewrite* programlarını aynı terminalden çalıştıramayız. Çünkü ilk çalıştırdığımız program *blokeye* yol açarak bekleyeceğinden diğer programı çalıştırmamız mümkün olmayacaktır. Bu nedenle denemeyi programları ayrı terminallerden çalıştırarak yapmalıyız (*Linux* sistemlerinde *Ctrl+Alt+F* tuşlarıyla terminaller arasında geçiş yapılabildiğini anımsayınız.) Diğer bir seçenek de bu iki programı aynı terminalden fakat ilkini arka planda kalacak biçimde çalıştırmaktır:

```
study@linux-sdhv:~>./pipewrite testpipe &
[1] 5901
study@linux-sdhv:~>./piperead testpipe
```

Boruya yazan ya da borudan okuyan programın önce çalıştırılmasının bir soruna yol açmayacağına dikkat ediniz. İlk çalıştırılan program *open* fonksiyonu içerisinde bloke olarak diğerinin çalıştırılmasını bekleyecektir. Haberleşme bitince boruyu *pipeop* programıyla ya da *rm* komutuyla silebilirsiniz:

```
study@linux-sdhv:~>./pipeop -r testpipe
```

İsimli boruların çalışma mekanizmasını komut satırından da gözlemleyebilirsiniz. Örneğin:

```
study@linux-sdhv:~>mkfifo testpipe
study@linux-sdhv:~>ls > testpipe &
[1] 6684
study@linux-sdhv:~>cat testpipe
```

Burada önce *mkfifo* fonksiyonuyla isimli boru yaratılıyor, sonra boruya *ls* komutuyla bir şeyler yazılıyor ve nihayet *cat* işlemi ile de boruya yazılanlar okunuyor.

4.1. İsimli Borularla İstemci Sunucu Haberleşmeleri

İsimli borularla istemci sunucu (*client-server*) tarzı bir haberleşme modeli oluşturulabilir. Bildiğiniz gibi bu modelde istemci ve sunucu ayrı birer programdır. Sunucuya bir ya da birden fazla istemci bağlanabilir. Sunucu işlemi yapar ve sonuçları istemciye gönderir. İstemci sunucu haberleşmesinde istemciler isteklerini tek bir boruyla sunucuya iletebilirler. Fakat sunucu tüm istemcilerle tek bir boru kullanarak konuşamaz. Her istemci için sunucunun ayrı bir boru yaratması gerekir. (Bunun nedenini düşününüz). Haberleşme modeli şöyle olabilir:

- Sunucunun istek almakta kullanacağı sunucu borusu önceden belirlenmiştir. İstemciler bu boruyu kullanarak sunucuya bağlanırlar.

- Sunucu bağlantıyı kabul ettikten sonra istemciyle haberleşmek için istemci borusunu yaratır.

- İstemci isteğini sunucu borusuna iletir. Sunucu işlemini yaparak sonuçları istemci borusuna gönderir.

- İstemci haberleşmeye son vermek istediğini sunucuya iletir, sunucu da istemci için yarattığı boruyu siler.

5. Blokesiz Modda Boru İşlemleri

İsimli ve isimsiz borularda varsayılan çalışma modu blokeli moddur. Önceden de belirtildiği gibi blokeli modda yazan taraf yazmak isteği bilginin tamamı yazılana kadar, okuyan taraf da en az bir byte okunana kadar blokede bekler. Fakat bazı durumlarda blokeli çalışma istenmeyebilir. Örneğin, bir prosesin çok sayıda borudan okuma yapmak istediğini varsayalım. Bu işlemin blokeli modda yapılmasına olanak yoktur. Çünkü böylesi bir durumda borulardan birinde oluşan bloke diğer borulardan yapılabilecek okumayı engeller.

İsimsiz borular yaratıldıktan sonra *fcntl* fonksiyonu ile *O_NONBLOCK* bayrağı kullanılarak blokesiz moda geçirilebilirler. Örneğin:

```
int fdPipe[2];
int flags;

if (pipe(fdPipe) < 0) {
    perror("pipe");
    exit(EXIT_FAILURE);
}
```

```
/* okuma betimleyicisi blokesiz moda geçiriliyor */

if ((flags = fcntl(fdPipe[0], F_GETFL, 0)) < 0) {
    perror("fcntl");
    exit(EXIT_FAILURE);
}

flags |= O_NONBLOCK;

if (fcntl(fdPipe[0], F_SETFL, flags) < 0) {
    perror("fcntl");
    exit(EXIT_FAILURE);
}

/* yazma betimleyicisi blokesiz moda geçiriliyor */

if ((flags = fcntl(fdPipe[1], F_GETFL, 0)) < 0) {
    perror("fcntl");
    exit(EXIT_FAILURE);
}

flags |= O_NONBLOCK;

if (fcntl(fdPipe[1], F_SETFL, flags) < 0) {
    perror("fcntl");
    exit(EXIT_FAILURE);
}
```

İsimli borular ise doğrudan *open* fonksiyonunda *O_NONBLOCK* bayrağı kullanılarak blokesiz modda açılabilir. Örneğin:

```
int fdPipe;

if ((fdPipe = open("testpipe", O_RDONLY|O_NONBLOCK)) < 0) {
    perror("open");
    exit(EXIT_FAILURE);
}
```

Blokesiz modda *read* ve *write* fonksiyonları blokeye yol açmazlar. *read* fonksiyonu eğer boruda hiç bilgi yoksa başarısızlıkla geri döner ve *errno* değişkeni *EAGAIN* değeriyle doldurulur. Boruda en az 1 byte bilgi varsa *read* fonksiyonu -blokeli modda olduğu gibi- okuyabildiği kadar bilgiyi okur ve okuyabildiği *byte* sayısıyla geri döner. Blokesiz modda write fonksiyonunun davranışı yazılmak istenen miktarın *PIPE_BUF* değerinden büyük olup olmadığına göre değişmektedir:

1) Eğer yazılmak istenen miktar *PIPE_BUF* değerinden küçük ya da ona eşitse ve boruda tüm yazılmak istenen miktarın yazılmasına olanak sağlayacak kadar boş yer varsa *write* fonksiyonu tüm bilgileri boruya yazarak yazılan miktarla geri döner. Eğer boruda yazılmak istenen bilgilerin hepsinin yazılabileceği kadar yer yoksa bu durumda fonksiyon blokeye yol açmaz, hiç yazma işlemi yapmadan başarısızlıkla geri döner. Bu durumda *errno* değişkeni *EAGAIN* değeriyle doldurulur.

2) Eğer yazılmak istenen miktar *PIPE_BUF* değerinden büyükse ve bu durumda boruda en az 1 byte boş yer varsa *write* fonksiyonu yazabildiği kadar bilgiyi boruya yazar ve boruya yazılan byte

sayısı ile geri döner. Eğer boru tamamen doluysa ve hiçbir byte yazılamıyorsa *write* fonksiyonu başarısız olur ve *errno* değişkeni *EAGAIN* değeriyle doldurulur.

İsimli boruların blokesiz modda *open* fonksiyonuyla açılmasında da bloke oluşmaz. Eğer isimli boru O_*RDONLY*|*O_NONBLOCK* bayraklarıyla açılmaya çalışılırsa *open* fonksiyonu bloke olmadan -diğer koşullar da uygunsa- başarıyla geri döner. Fakat isimli boru O_*WRONLY*| *O_NONBLOCK* bayraklarıyla açılmaya çalışılırsa *open* fonksiyonu başka bir proses boruyu okuma yapmak amacıyla daha önce açmamışsa başarısızlıkla geri döner ve *errno* değişkeni *ENXIO* değeriyle doldurulur. Görüldüğü gibi blokesiz modda *read/write* ve *open* fonksiyonları blokeye yol açmamaktadır.

Şimdi blokesiz modda çalışmaya neden gereksinim duyulabileceğini açıklayalım. Elimizde bir grup boru betimleyicisi olsun. Biz bu borulardan gelen bilgileri işleme sokacak olalım. Aşağıdaki gibi bir döngü işimize yarar mıydı?

```
int pipes[NPIPES];
char buf[BUFSIZE];
int n;

for (;;) {
    for (i = 0; i < NPIPES; ++i) {
        if ((n = read(pipes[i], buf, BUFSIZE)) < 0) {
            perror("read");
            exit(EXIT_FAILURE);
        }
        /* okunan bilgi işleme sokuluyor */
    }
    /* ... */
}
```

Burada borulardan birine hiç bilgi gelmemişse o okuma işlemi blokeye yol açacak ve diğer borulardan okuma yapma olanağımız kalmayacaktır. Halbuki borular blokesiz modda açılsaydı bir boruda bilgi olmadığı durumda bloke oluşmayacağı için diğer borulardan okuma şansımız olabilecekti. Aynı kod parçası blokesiz çalışma için şöyle düzeltilebilir:

```
int pipes[NPIPES];
char buf[BUFSIZE];

for (;;) {
    for (i = 0; i < NPIPES; ++i) {
        if ((n = read(pipes[i], buf, BUFSIZE)) < 0) {
            if (errno == EAGAIN)
                continue;
            perror("read");
            exit(EXIT_FAILURE);
        }
        /* okunan bilgi işleme sokuluyor */
    }
    /* ... */
}
```

6. popen ve pclose Fonksiyonları

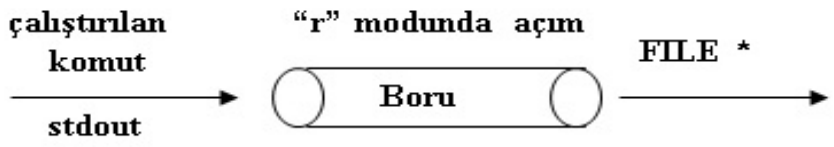
popen ve *pclose* bir programın *stdout* ve *stdin* dosyalarını boruya yönlendirebilmek için kullanılan standart *POSIX* fonksiyonlarıdır. *popen* fonksiyonunun prototipini inceleyiniz:

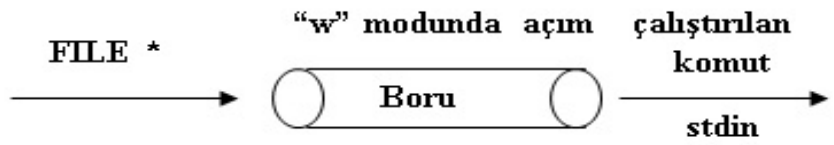
```
#include <stdio.h>

FILE *popen(const char *command, const char *mode);
```

popen fonksiyonunun birinci parametresi çalıştırılacak olan programı belirtir. *popen* bu programı kabuk yoluyla çalıştırmaktadır. Bu nedenle birinci parametre aslında bir programın çalışmasına yol açacak herhangi bir kabuk komutu olabilir. *POSIX* standartları birinci parametreyle belirtilen komutun aşağıdaki gibi çalıştırılacağını söylemektedir:

```
execl("/bin/sh", "sh", "-c", command, (char *) 0);
```

Fonksiyonun ikinci parametresi dosyanın açış modunu belirtir. Bu mod *"r"* ya da *"w"* olabilir. *fopen* fonksiyonunda kullanılan diğer modlar geçerli olarak kabul edilmemektedir. Fonksiyon başarı durumunda yaratılan boru üzerinde işlem yapmakta kullanılacak dosya bilgi göstericisine (*stream*), başarısızlık durumunda ise *NULL* adrese geri dönmektedir. İkinci parametredeki açış modu *"r"* biçiminde belirtildiyse fonksiyon boruyu yaratır ve birinci parametresiyle belirtilen programın 1 numaralı (*STDOUT_FILENO*) betimleyicisini boruya yönlendirir. Bu durumda fonksiyonun geri döndürdüğü dosya bilgi göstericisi kullanılarak okuma yapıldığında aslında borudan okuma yapılmış olacaktır. Başka bir deyişle biz *popen* fonksiyonunu *"r"* modunda kullandığımızda bize geri verilen dosya bilgi göstericisiyle okuma yaparsak, çalıştırılan programın *stdout* dosyasına yazdıklarını okumuş oluruz. İkinci parametredeki açış modunda *"w"* kullanılırsa fonksiyon boruyu yarattıktan sonra birinci parametreye belirtilen programın *0* numaralı (*STDIN_FILENO*) betimleyicisini boruya yönlendirir. Bu durumda fonksiyon boruya yazma yapmakta kullanılabilecek dosya bilgi göstericisi ile geri döner. Yani biz bu geri dönüş değeri ile verilen dosya bilgi göstericisini kullanarak dosyaya yazma yaptığımızda aslında çalıştırılan programın *stdin* dosyasından okuyacağı bilgileri oluşturmuş oluruz.





popen fonksiyonun çalışmasına ilişkin bir noktayı vurgulamak istiyoruz. Yukarıda da belirttiğimiz gibi aslında *popen* kabuk programını (yani */bin/sh programını*) çalıştırmaktadır. Kabuk programları *-c* seçeneğiyle çalıştırıldıklarında yalnızca tek bir komutu çalıştırıp geri dönerler. Bir proses başka bir prosesi yarattığında üst prosesin dosya betimleyicilerinin alt prosese aktarıldığını anımsayınız. Böylece biz bir programı çalıştırdığımızda eğer o programlar da başka programları çalıştırırsa aslında çalıştırılan tüm programların betimleyicileri üst prosesten aktarılmış olacaktır. Örneğin biz *popen* fonksiyonunu şöyle çağırmış olalım:

```
FILE *f;

f = popen("ls", "r");
```

Bu durumda aslında *popen* kabuk programını (*/bin/sh*) çalıştırır ve onun *stdout* dosyasını boruya yönlendirir. *ls* programı kabuk tarafından çalıştırıldığı için onun *stdout* dosyası da boruya yönlendirilmiş olacaktır.

pclose parametre olarak *popen* fonksiyonu ile elde edilen dosya bilgi göstericisini alır, yaratılmış olan boruyu ve dosyayı kapatır:

```
#include <stdio.h>

int pclose(FILE *stream);
```

Richard Stevens "*Advanced Programming In the Unix Environment*" kitabında *popen* ve *pclose* fonksiyonunun olası yazımını şöyle vermiş:

```
#include <errno.h>
#include <fcntl.h>
#include <sys/wait.h>

static pid_t *childpid = NULL;
static int maxfd;

FILE *popen(const char *cmdstring, const char *type)
{
    int i;
    int pfd[2];
    pid_t pid;
    FILE *fp;

    /* Açış modu kontrolü yapılıyor */

    if ((type[0] != 'r' && type[0] != 'w') || type[1] != 0) {
        errno = EINVAL
        return NULL;
    }

    if (childpid == NULL) {
        maxfd = open_max();
        if ((childpid = calloc(maxfd, sizeof(pid_t))) == NULL)
            return NULL;
    }

    if (pipe(pfd) < 0)
        return NULL;

    if ((pid = fork()) < 0)
        return NULL;
```

```
    if (pid == 0) {          /* Alt proses */
        if (*type == 'r') {
            close(pfd[0]);
            if (pfd[1] != STDOUT_FILENO) {
                dup2(pfd[1], STDOUT_FILENO);
                close(pfd[1]);
            }
        }
        else {
            close(pfd[1]);
            if (pfd[0] != STDIN_FILENO) {
                dup2(pfd[0], STDIN_FILENO);
                close(pfd[0]);
            }
        }

    /* childpid dizisindeki tüm betimleyiciler kapatılıyor */

        for (i = 0; i < maxfd; i++)
            if (childpid[i] > 0)
                close(i);

        execl("/bin/sh", "sh", "-c", cmdstring, (char *)0);
        _exit(127);
     }

    /* Üst proses devam ediyor... */

    if (*type == 'r') {
        close(pfd[1]);
        if ((fp = fdopen(pfd[0], type)) == NULL)
            return(NULL);
    }
    else {
        close(pfd[0]);
        if ((fp = fdopen(pfd[1], type)) == NULL)
            return(NULL);
    }

    childpid[fileno(fp)] = pid;

    return fp;
}

int pclose(FILE *fp)
{
    int fd, stat;
    pid_t pid;

    if (childpid == NULL) {
        errno = EINVAL;
        return -1;          /* popen fonksiyonu hiç çağrılmamış */
    }

    fd = fileno(fp);
```

```
    if ((pid = childpid[fd]) == 0) {
        errno = EINVAL;
        return -1;          /* fp popen tarafından açılmamış*/
    }

     childpid[fd] = 0;
     if (fclose(fp) == EOF)
        return -1;

    while (waitpid(pid, &stat, 0) < 0)
        if (errno != EINTR)
            return(-1);

    return stat;          /* return child's termination status */
}
```

Yukarıdaki kod üzerinde bazı açıklamalarda bulunalım. *popen* fonksiyonda önce borunun yaratıldığını görüyorsunuz:

```
if (pipe(pfd) < 0)
    return NULL;
```

Daha sonra alt proses yaratılarak yönlendirme yapılmıştır:

```
if (pid == 0) {          /* Alt proses */
    if (*type == 'r') {
        close(pfd[0]);
        if (pfd[1] != STDOUT_FILENO) {
            dup2(pfd[1], STDOUT_FILENO);
            close(pfd[1]);
        }
    }
    else {
        close(pfd[1]);
        if (pfd[0] != STDIN_FILENO) {
            dup2(pfd[0], STDIN_FILENO);
                close(pfd[0]);
        }
    }
    /*... */
```

dup2 fonksiyonundan önce kontrol yapıldığını görüyorsunuz. *popen* fonksiyonunu çağıran proses eğer zaten yaratılan borunun betimleyicileri STDIN_*FILENO* ve *STDOUT_FILENO* değerlerindeyse boşuna yönlendirme yapmak istememiştir. (Prosesin tüm betimleyicileri kapatarak pipe fonksiyonunu uygulaması durumunda böyle bir durumla karşılaşılabilir.) Bu işlemlerden sonra yaratılan alt proseste standartlarda belirtildiği gibi daha *önce popen* fonksiyonu ile yaratılmış olan betimleyiciler kapatılarak execl fonksiyonu uygulanmıştır:

```
     for (i = 0; i < maxfd; i++)
        if (childpid[i] > 0)
            close(i);

        execl("/bin/sh", "sh", "-c", cmdstring, (char *)0);
        _exit(127);
```

popen fonksiyonu ile yaratılmış olan tüm betimleyicilerin *childpid* isimli global bir dizide tutulduğunu belirtelim. Bu global dizinin uzunluğunu belirlemek için *open_max* isimli bir fonksiyon çağrılmıştır. *open_max* fonksiyonu çalışılan sistemde bir prosesin sahip olabileceği en büyük betimleyici sayısını vermektedir. Bu fonksiyonun kodları da *Stevens*'ın *Advanced Programming In the UNIX Environment* kitabında verilmiştir. Fakat deneme yapmak için bu fonksiyonun yerine *1024* gibi bir değer yazabilirsiniz. *popen* fonksiyonunda daha sonra *fdopen* fonksiyonu ile elde edilen betimleyiciden dosya bilgi göstericisi elde edilmiştir:

```
if (*type == 'r') {
    close(pfd[1]);
        if ((fp = fdopen(pfd[0], type)) == NULL)
            return NULL;
}
else {
    close(pfd[0]);
    if ((fp = fdopen(pfd[1], type)) == NULL)
        return NULL;
}
```

pclose fonksiyonunda yapılan işlem özet olarak boruyla ilişkili olan dosyanın kapatılmasıdır. Kapatılacak dosya *childpid* dizisinden de çıkartılmıştır.

Şimdi *popen* ve *pclose* fonksiyonlarının kullanımını açıklayan basit bir örnek verelim. Örneğimizde *ls* programını çalıştırıp programın çıktısını boruya yönlendirerek borudan okuma yapacağız:

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

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

    if ((f = popen("ls", "r")) == NULL) {
        perror("ls");
        exit(EXIT_FAILURE);
    }

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

    if (ferror(f)) {
        perror("fgetc");
        exit(EXIT_FAILURE);
    }

    pclose(f);

    return 0;
}
```

Şimdi -daha önce yaptığımız gibi- bir programın *stdout* dosyasına yazdıklarını boru yardımıyla diğer programın *stdin* dosyasına verelim:

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

int main(int argc, char *argv[])
{
    FILE *fout, *fin;
    char buf[PIPE_BUF];
    ssize_t n;

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

    if ((fin = popen(argv[1], "r")) == NULL) {
        perror("popen");
        exit(EXIT_FAILURE);
    }

    if ((fout = popen(argv[2], "w")) == NULL) {
        perror("popen");
        exit(EXIT_FAILURE);
    }

    if ((n = fread(buf, 1, PIPE_BUF, fin)) > 0)
        if (fwrite(buf, 1, n, fout) != PIPE_BUF) {
            perror("fwrite");
            exit(EXIT_FAILURE);
        }

    if (feof(fin)) {
        perror("fread");
        exit(EXIT_FAILURE);
    }

    pclose(fout);
    pclose(fin);

    return 0;
}
```

---

[1]: *C*'de fonksiyonların gösterici parametreleri dizisel biçimde belirtilebilmektedir. Bu gösterim biçiminde köşeli parantezin içi boş bırakılabilir ya da içerisine bir sabit ifadesi yazılabilir. Köşeli parantezler içerisine yazılmış olan değerin hiçbir önemi yoktur. Örneğin aşağıdaki tüm prototipler aynı anlama gelir ve özdeştir.

```
int pipe(int *filedes);
int pipe(int filedes[]);
int pipe(int filedes[2]);
```

Bu prototiplerin her biri aslında fonksiyonun bir gösterici parametresine sahip olduğunu belirtmektedir. Tabi bu gösterim biçimi yalnızca fonksiyon parametre değişkenleri için söz konusudur. Aşağıdaki gibi bir tanımlama geçerli değildir:

```
int p[]; /* Geçersiz tanımlama */
```

Fonksiyonların gösterici parametrelerinin dizisel biçimde gösterilmesi bazen okunabilirliği artırmaktadır. Örneğin *POSIX* standartlarında prototip olarak üçüncü biçim verilmiştir. Bu üçüncü biçimden biz fonksiyona iki elemanlı *int* türden bir dizi adresinin geçirilmesi gerektiğini anlarız.

Kaynaklar

1. Aslan, K. (2002). *UNIX/Linux Sistem Programlama Kurs Notları*. Istanbul: C ve Sistem Programcıları Derneği.

2. Maurice, B. (1986). *The Design Of The UNIX Operating System.* Prentice Hall: New Jersey.

3. Stevens, R., Rago, S. A. (2005). *Advanced Programming in the UNIX(R) Environment (2nd Edition)*. Addison-Wesley Professional.