Linux Aygıt Sürücüleri – Bölüm 4 –

Örnek Sanal LIFO-FIFO Aygıtı ve Sürücü Programlama

Bu bolumde, FIFO-LIFO (First in first out- İlk giren ilk çıkar, Last in first out, Son giren ilk çıkar) veri yapılarını kullanan örnek bir aygıtın, sürücüsünün nasıl yazıldığını inceleyeceğiz.

Öncelikle LIFO ve FIFO veri yapılarını hatırlayalım.

LIFO, son giren ilk çıkar, yapısıdır. En güzel örnek stack (yığın) dır.

Ekleme sırasına göre, LIFO veri yapısının içine baktığımızda aşağıdaki gibi bir tablo ile karşılaşırız. 1 numaralı bilgi ilk eklenen, 5 numaralı bilgi ise son eklenen bilgidir.

```
|    5    |
|    4    |
|    3    |
|    2    |
|____1____|
```

LIFO yapısından okuma yaptığımızda ilk önce en son yazılan bilgi okunacaktır. Bu da 5 numaralı bilgidir, sonra sırası ile 4,3,2 ve 1 numaralı bilgi okunacaktır.

LIFO yu aşağıdaki bağlı liste (linked list) yapısında tutmak istersek;

```
struct item_type {
    char      * string;
    item_type * next  ;
} ;
```

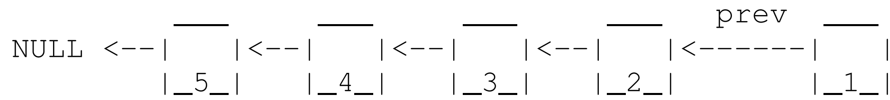
```
 ___  next   ___     ___     ___     ___
|   |------>|   |-->|   |-->|   |-->|   |--> NULL
|_5_|       |_4_|   |_3_|   |_2_|   |_1_|
```

Şeklinde olacaktır.

Görüldüğü gibi bilgi olarak sadece, yazılacak string'in pointerini ve bir sonraki elemanını gösteren pointeri tutuyoruz.

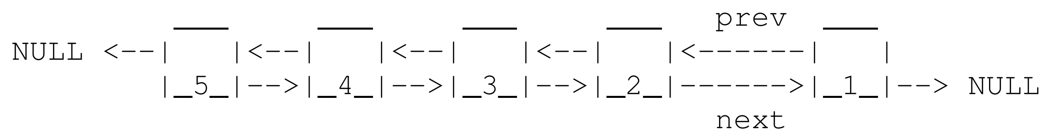
FIFO da ise, LIFO'nun tam tersi bir durum söz konusudur. Ilk eklenen eleman ilk olarak okunur. Kuyruk (queue) yapısı en iyi örnektir.

```
struct item_type {
    char      * string;
    item_type * prev  ;
} ;
```



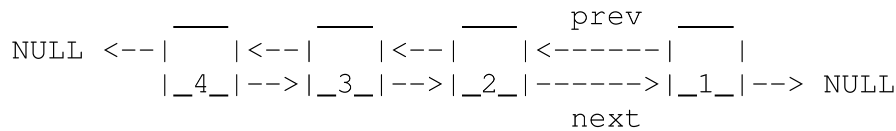
Iki yapıyı birleştirirsek, yani çift yönlü bir geçiş sağlarsak okuma yönümüze göre FIFO yada LIFO olarak kullanabiliriz.

```
struct item_type {
    char      * string;
    item_type * next  ;
    item_type * prev  ;
} ;
```

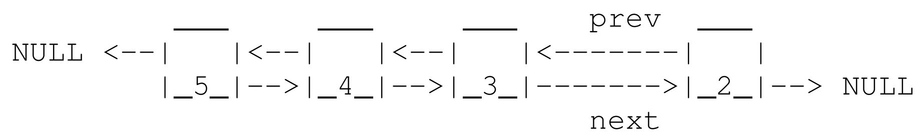


Ekleme işlemini her iki durumda da, soldan yapmamız gerekiyor. Çünkü FIFO da ilk eklenen elemanı (en sağdaki), LIFO da en son eklenen elemanı (en soldaki) okuyacağız. Okuma işlemi yapıldıktan sonra bu bilgiler silinmelidir.

LIFO olarak bir eleman okuduğumuzda;



FIFO olarak bir eleman okuduğumuzda



şeklinde olacaktır.

Hem LIFO hem de FIFO olarak kullanılabilen bir bağlı listeyi kullanan bir linux aygıt sürücüsü yazmak için, nelere ihtiyacımız olacağını ve hangi sistem çağrılarını cevaplayacağımızı tesbit edelim.

Kullanıcı programlarından gelen read ve write sistem çağrıları için bağlı listemizi kullanacağız. write ile kuyruğun soluna yeni bir eleman ekleyeceğiz, read ile de, FIFO yada LIFO durumuna göre, en sol yada en sağdaki elemanın string bilgisini kullanıcı programına gönderip, bu elemanı bellekten sileceğiz.

Aygıtımızın FIFO ya da LIFO olarak çalışmasını ise kullanıcı ayarlayabilmelidir. Bu durumda, kullanıcıların aygıtlar üzerinde kontrol işlemleri yapabilmelerini sağlayan ioctl sistem çağrısına ihtiyacımız olacak. Kullanicidan gelen ioctl isteğine göre, çalışma modunu FIFO yada LIFO olarak ayarlayıp, kullanıcıya, bellekte ne kadar eleman olduğunu dönderebilmeliyiz.

Kuyruk için kullanacağımız yapıyı, değişkenlerimizi, fonksiyonlarımızı ve file_operations yapısındaki fops değişkenimizi tanımlayalım.

```
// Kuyruğun yapısını tanımlayalım.
typedef struct item_type item_type;

struct item_type {
    char *string;
    item_type *next;
    item_type *prev;
    int no;
} ;
```

```
// Cevaplayacağımız sistem çağrıları için çalışacak fonksiyonları
tanımlayalım.
static int     bdriver_open  (struct inode *, struct file * );
static ssize_t bdriver_write (struct file  *, const char *, size_t,
loff_t *);
static ssize_t bdriver_read  (struct file  *, char * , size_t , loff_t
*);
static int     bdriver_ioctl (struct inode *, struct file *, unsigned
int,
                               unsigned long );
```

```
// Kuyruğa eleman ekleyen ve çıkaran fonksiyonlar.
void add_item    (char *);
void remove_item ();
```

```
// Kuyruğun başındaki, sonundaki eleman ve eleman sayısı.
item_type * ilk = NULL;                 // Listenin en solundaki
eleman.
item_type * son = NULL;                 // Listenin en sagindaki
eleman.
int item_count  = 0;
```

```
// Kuyruğun ön tanımlı çalışma modu.

int fifo_lifo = MOD_LIFO;               // defined in bdriver.h
```

```
static struct file_operations fops = {   /* define in linux/fs.h */
    NULL,                /* owner */
    NULL,                /* lseek   */
    bdriver_read,        /* read    */
    bdriver_write,       /* write   */
    NULL,                /* readdir */
    NULL,                /* select  */
    bdriver_ioctl,       /* ioctl   */
    NULL,                /* mmap    */
    bdriver_open,        /* open    */
    NULL,                /* release */
    NULL,                /* fsync   */
} ;
```

Bir header dosyası tanımlayalım. Bu dosyadan aygıtımızın özelliklerini ve ioctl için kullanacağımız kodlari tanimlayalim.

```
/*
 * Necati Ersen SISECI
 * siseci ~ enderunix org
 *
 * bdriver header file
 */

#include <linux/ioctl.h>

int device_major_number = 240 ;

#define BDRIVER_IOCTL_BASE 0xAE
#define BDRIGETFL _IOR(BDRIVER_IOCTL_BASE, 99,  int)   /* BDRIVER GET
FIFO-LIFO STATUS */
#define BDRISETFL _IOW(BDRIVER_IOCTL_BASE, 100, int)   /* BDRIVER SET
FIFO-LIFO STATUS */
#define BDRIGETIC _IOR(BDRIVER_IOCTL_BASE, 101, int)   /* BDRIVER GET
ITEM COUNT */

#define MOD_FIFO 1 /* Fifo Mode */
#define MOD_LIFO 2 /* Lifo Mode */


#define DEVICE_NAME "bdevice"
#define DEVICE_FNAME "/dev/"DEVICE_NAME
```

Görüldüğü gibi aygıtımız için major olarak 240'i seçtik.
(/usr/src/linux/Documentation/devices.txt)

ioctl kodlarimizi olusturmak için linux/ioctl.h daki _IOR ve _IOW tanımlamalarını kullanıyoruz.

(/usr/src/linux/Documentation/ioctl-number.txt)

ioctl.h dosyası incelendiği zaman _IOR ile aygıttan okuma, _IOW ile aygıtın parametrelerinde değişiklik yapacağımızı belirtiyoruz. ioctl tanımlamaları için aygıtın major numarası da kullanılabilir. Burada ioctl-number.txt dosyasında boş olan ilk numarayı kullandım.

BDRIGETFL (Bdriver Get FIFO-LIFO status) tanımlaması ile aygıtın FIFO modunda mı LIFO modunda mı oldugunu öğreniyoruz.

BDRISETFL (Bdriver Set FIFO-LIFO status) tanımlaması ile aygıtın çalışma şeklini FIFO yada LIFO olarak değiştiriyoruz.

BDRIGETIC (Bdriver Get Item Count) tanımalaması ile aygıtta tanımlı kuyrukta kaç tane elemanın bulunduğunu öğreniyoruz.

init_module ile başlayalım.

```
int init_module (void)
{
    int i = 0;
    i = register_chrdev (device_major_number, DEVICE_NAME , &fops);

#if defined(DEBUG)
    printk(KERN_DEBUG "Register Device:%d\n", i);
#endif

    if (i < 0) return -i;
    if (device_major_number == 0) device_major_number = i;
    return 0;
}
```

device_major_number ve DEVICE_NAME 'i bdriver.h dosyasında tanımlamıştık. Önceki yazılarımızdan hatırlarsak, register_chrdev fonksiyonunda aygıt için major numarası belirtebiliyorduk. Eğer 0 seçersek, sistem ilk boş major numarasını seçip, register_chrdev le bize dönderiyordu.

file_operations yapısını tutan fops değişkenini de yukarıda tanımlamıştık.

cleanup_module fonksiyonunu yazalım.

```
void cleanup_module (void)
{
     fifo_lifo = 2;
     unregister_chrdev (device_major_number, DEVICE_NAME);
     /*
      * Eğer kuyrukta eleman varsa bu kısımda
      * bellekten atılmaları gerekiyor.
      */
     while (item_count > 0)
     {
#if defined(DEBUG)
          printk(KERN_DEBUG "Silinen: %s", ilk->string);
#endif
             remove_item ();
     }

}
```

remove_item, kuyruğun çalışma moduna göre en sol yada en sağdaki elemanı bellekten atar.

item_count ise, kuyrukta kaç eleman olduğunu tutar. BDRIGETIC, item_count global değişkenini kullanır.

Şimdi sistem çağrılarını karşılayacak olan fonksiyonları yazalım.

open sistem çağrısı için,

```
static int bdriver_open (struct inode *i, struct file *f)
{
     // Sanal device oldugu icin Open fonksiyonuna gerek yok.
     return 0;
}
```

write sistem çağrısı için,

```
static ssize_t
bdriver_write (struct file *f, const char *ch, size_t size, loff_t *
lto)
{
    char * mych;
    mych = kmalloc (size, GFP_KERNEL);
    strncpy(mych, ch, size);
    mych[size] = 0;
    add_item (mych);

#ifdef DEBUG
```

```
    printk(KERN_DEBUG "Eklenen: %d->:%d:%s", item_count, strlen(mych),
mych);
#endif

    return strlen (ch);
}
```

add_item fonksiyonu kuyruğa yeni bir eleman ekler.

read sistem çağrısı için,

```
static ssize_t
bdriver_read (struct file *f, char *cc, size_t t, loff_t * l)
{
    int str_size = 0;
    char *sstr;
    if ((item_count <= 0 ) || (t == 0)) return 0;
    if (fifo_lifo == 2)
        sstr = ilk->string;
    else
        sstr = son->string;
    str_size = strlen (sstr);

#if defined(DEBUG)
    printk(KERN_DEBUG "Okunan: %d->%s", str_size, sstr);
#endif

    copy_to_user (cc, sstr, str_size);
    remove_item ();
    return str_size;
}
```

fifo yada lifo moduna göre ilk yada son elemanın değerini kullanıcıya dönderip, bu elemanı kuyruktan siliyoruz.

ioctl çağrısını karşılayacak fonksiyonumuz,

```
static int
bdriver_ioctl (struct inode *inode, struct file *file, \
                   unsigned int cmd, unsigned long arg)
{
    int status;
    switch (cmd)
    {
      case BDRIGETFL:
         if (put_user (fifo_lifo, (int *) arg)) return -EFAULT;
         return 0;
      case BDRISETFL:
```

```
        if (get_user (status, (int *) arg)) return -EFAULT;
        switch (status)
        {
           case MOD_LIFO:
               fifo_lifo = MOD_LIFO;
               break;
           case MOD_FIFO:
               fifo_lifo = MOD_FIFO;
               break;
           default:
               return -EINVAL;
        }
#ifdef DEBUG
        printk(KERN_DEBUG "bdevice mod changed: %d\n", fifo_lifo);
#endif
        break;
    case BDRIGETIC:
        if (put_user (item_count, (int *) arg)) return -EFAULT;
    default:
        return -ENOTTY;
    }
    return 0;
}
```

get_user ve put_user ile user belleğine ulaşıyoruz.

ioctl ile gelen paratmereye göre aygıtın çalışma şeklinde değişiklik yapıyoruz yada yukarıdaki koddan da görüldüğü üzere, BDRIGETIC gelmesi durumunda item_count değişkenin kullanıcı belleğindeki arg değişkenine yazıyoruz.Boylece o an driver in belleğindeki eleman sayısını döndermiş oluyoruz.

Sistem çağrılarını karşılayacak fonksiyonlar bu şekilde bitiyor.

Kuyruğa eleman ekleyen ve kuyruktan uygun elemani silen fonksiyonları yazalım.

```
void add_item(char * stri)
{
    item_type * new;
    new = (item_type * ) kmalloc( sizeof (item_type), GFP_KERNEL);
    new->string = stri;
    new->next = ilk;
    if (ilk != NULL) ilk->prev = new ;
    new->prev = NULL;
    if (item_count < 1)
     {
        item_count = 0;
        son = ilk;
    }
```

```
    new->no = ++item_count;
    ilk = new;
    if (item_count == 1) son = ilk;
}
```

```
void remove_item()
{
    item_type * new;
    item_count--;
    if (item_count > 0 )
    {
        if (fifo_lifo == 2)
        {
            new = ilk->next;
            new->prev = NULL;
            kfree(ilk);
            ilk = new;
        }
        else
        {
            new = son->prev;
            new->next = NULL;
            kfree(son);
            son = new;
        }
    }
}
```

Tum kodları tek bir dosyada toplayip test yapalim.

```
# gcc -Wall -DMODULE -D__KERNEL__ -DLINUX -c -O3 bdriver.c
# insmod ./bdriver.o
```

Modülümüzü yükledikten sonra /proc/devices dosyasında aygıtımızı görebiliyoruz.

```
# cat /proc/devices
...
240 bdevice
...
```

Sürücümüzün read ve write fonksiyonlarını yazmıştık.

Henuz herhangi bir bilgi yüklemediğimiz icin cat ile aygit a baktığımizda boş bir çıktı gelecektir.

```
# cat /dev/bdevice
#
```

Şimdi bir kaç satır bilgi yükleyelim.

```
# echo deneme > /dev/bdevice
# cat /dev/bdevice
deneme

#
```

Ufak bir program aracılığı ile aygıtımıza 10 kez yazalım. Program yazdıklarını aynı zamanda ekrana da basmaktadır.

```
# ./yaz
device a 10 kere yaziliyor
a
aa
aaa
aaaa
aaaaa
aaaaaa
aaaaaaa
aaaaaaaa
aaaaaaaaa
aaaaaaaaaa
```

Aygıtımız LIFO (Last in first out) modunda çalıştığı içn ilk olarak son eklenen elemani görecegiz.

```
# cat /dev/bdevice
aaaaaaaaaa
aaaaaaaaa
aaaaaaaa
aaaaaaa
aaaaaa
aaaaa
aaaa
aaa
aa
a
```

```
# ./yaz
device a 10 kere yaziliyor
a
aa
aaa
aaaa
aaaaa
aaaaaa
aaaaaaa
aaaaaaaa
```

```
aaaaaaaaa
aaaaaaaaaa
# ./yaz
device a 10 kere yaziliyor
a
aa
aaa
aaaa
aaaaa
aaaaaa
aaaaaaa
aaaaaaaa
aaaaaaaaa
aaaaaaaaaa
```

Tekrar cat ettiğimizde aşağıdaki sonucu alıyoruz.

```
# cat /dev/bdevice
aaaaaaaaaa
aaaaaaaaa
aaaaaaaa
aaaaaaa
aaaaaa
aaaaa
aaaa
aaa
aa
a
aaaaaaaaaa
aaaaaaaaa
aaaaaaaa
aaaaaaa
aaaaaa
aaaaa
aaaa
aaa
aa
a
```

ioctl yardimi ile aygitimizin calisma modunu (Fifo - lifo) degistirebiliyor ve kac eleman oldugunu ogrenebiliyoruz.

```
# ./okuio
device tan okunuyor
BDRIGETFL Fifo-Lifo Status: Lifo Mode
BDRIGETIC Item Count: 0
```

```
# ./yaz
device a 10 kere yaziliyor
a
aa
aaa
aaaa
aaaaa
aaaaaa
aaaaaaa
aaaaaaaa
aaaaaaaaa
aaaaaaaaaa
# ./okuio
device tan okunuyor
BDRIGETFL Fifo-Lifo Status: Lifo Mode
BDRIGETIC Item Count: 10

# head -1 /dev/bdevice
aaaaaaaaaa
```

Görüldüğü gibi, aygıttan bir kayıt okuduk.
Aygıtın durumuna baktığımızda 9 eleman kaldığını görüyoruz.

```
# ./okuio
device tan okunuyor
BDRIGETFL Fifo-Lifo Status: Lifo Mode
BDRIGETIC Item Count: 9
```

okuio programı aynı zamanda aygıtın çalisma modunu da değiştirebilmektedir.

```
# ./okuio 1
device tan okunuyor
BDRIGETFL Fifo-Lifo Status: Lifo Mode
BDRIGETIC Item Count: 9
BDRISETFL Setting Fifo-Lifo status:1

# ./okuio
device tan okunuyor
BDRIGETFL Fifo-Lifo Status: Fifo Mode
BDRIGETIC Item Count: 9
```

Çalışma modunu değistirdiğimiz için kayitlar artık ters sırada okunacaklar.

```
# cat /dev/bdevice
a
aa
```

```
aaa
aaaa
aaaaa
aaaaaa
aaaaaaa
aaaaaaaa
aaaaaaaaa

# ./okuio
device tan okunuyor
BDRIGETFL Fifo-Lifo Status: Fifo Mode
BDRIGETIC Item Count: 0
```

Aygıt a tekrar 10 kayit yazdıktan sonra tekrar okuyalım.

```
# cat /dev/bdevice
a
aa
aaa
aaaa
aaaaa
aaaaaa
aaaaaaa
aaaaaaaa
aaaaaaaaa
aaaaaaaaaa
```

Goruldugu uzere kayitlar yazildigi sira ile okunmaktadir.

Yazilan kaynak kodlara

http://www.enderunix.org/siseci/bdriver/bdriver.c
http://www.enderunix.org/siseci/bdriver/bdriver.h
http://www.enderunix.org/siseci/bdriver/yaz.c
http://www.enderunix.org/siseci/bdriver/okuio.c
http://www.enderunix.org/siseci/bdriver/oku.c

adreslerinden ulaşabilirsiniz.

Tüm kodların ve bu dökümanın sıkıştırılmış haline
http://www.enderunix.org/siseci/bdriver/bdriver.tar.gz
adresinden ulaşabilirsiniz.

Yazılan kodlar Linux 2.4.26 çekirdeği üzerinde test edilmiştir.

Not: Bu dökümana neredeyse 2 sene once başlamıştım. Kod tarafı bitmiş olmasına rağmen döküman tarafını
bir türlü tamamlama fırsatım olmamıştı. En sonunda biraz zaman ayırıp tamamlamaya karar verdim.

N. Ersen SISECI
siseci ~ EnderUNIX Org
21 Kasim 2005