Introduction

- character drivers,
- other drivers gonišniki?
- block drivers:
  - access to devices that allow random access to data,
  - randomly accessible data,
  - fixed-size blocks,
  - DISKS
  - can also be something else
Linux sees block devices differently than character:

- a different interface,
- new problems :)
- good drivers are important for the performance of the entire system,
- swap,
- indirectly are part of the basic memory.
the basic goal for architects is the speed,
most of the character devices can work slower than the ideal,
system will work well,
system will badly work if block I/O will not be optimized,
consequently, block drivers are more complex.
Example driver: ramdisk sbull

- **sbull** – Simple Block Utility for Loading Localities,
- **naprava:**
  - block-oriented,
  - memory-based,
  - ramdisk,
  - simplified.
Definitions

- **block**:
  - fixed-size chunk of data,
  - often 4096 bytes,

- **sector**:
  - small block,
  - depends on the hardware.
  - core expects 512 bytes large sectors,
  - if the device uses a different size, the kernel partially adjusts (problems),
  - the driver must properly modify the number passed by the kernel.
a set of registration interfaces,
similar to character drivers,
but different :(  
new structure,
new operations.
Driver registration

registration in kernel:

```
int register_blkdev(unsigned int major, const char *name);
```

- **name** – device name,
- **major** – number used by kernel:
  - same as character devices,
  - shown in `/proc/devices`,
  - if 0, kernel assigns new number,
- de-registration in kernel:

```c
int unregister_blkdev(unsigned int major, const char *name);
```
Disk registration

- `register_blkdev` registers a major number,
- disk is not yet available to the system,
- special disk registration interface,
- we need to know two other structures:
Block device operations

- equivalent *file_operations* from char,
- important fields:

```c
typedef int (*open)(struct inode *inode, struct file *filp);
typedef int (*release)(struct inode *inode, struct file *filp);

typedef int (*ioctl)(struct inode *inode, struct file *filp, 
                     unsigned int cmd, unsigned long arg);

typedef int (*media_changed) (struct gendisk *gd);

typedef int (*revalidate_disk) (struct gendisk *gd);

struct module *owner;
```
specialities:
- no read/write methods,
- this functionality is performed by *request* (later),
kernel represents a device (disk),

important fields:

```c
int major;
int first_minor;
int minors;
//številke naprave, za vsako particijo rezerviramo po eno
//minor število in za napravo posebno
char disk_name[32];
//ime naprave, izpiše se v sysfs in /proc/partitions
struct block_device_operations *fops;
//množica operacij, definirane so na prejšnji prosojnici
```
struct request_queue *queue;
//struktura, ki jo uporablja jedro za nadzor nad
//I/O zahtevki za to napravo
int flags;
//zastavice, ki opisujejo stanje naprave primer:
//GENHD_FL_REMOVABLE za removeble media
sector_t capacity;
kapaciteta diska, število 512 bajtnih sektorjev
void *private_data;
//gonilnik lahko uporabi za svoje namene
structure *gendisk* is dynamically allocated,

driver is not initialized directly,

we use:

```c
struct gendisk *alloc_disk(int minors);
```

*minors* – number of minor numbers,

released by:

```c
void del_gendisk(struct gendisk *gd);
```
gendisk structure

- Initialization structures:

```c
struct gendisk *alloc_disk(int minors);
```

- `minors` – number of minors used by device,
- released by:

```c
void add_disk(struct gendisk *gd);
```

- request start to arrive after this function call,
Initialization in sbull

- a set of in-memory virtual disk drives,
- for each disk allocates a field,
- access is enabled via block operations,
- can partition the disk,
- we make a file system,
- mount,
- ...,
Initialization in sbull

- register block device:

```c
sbull_major = register_blkdev(sbull_major, "sbull");
if (sbull_major <= 0) {
    printk(KERN_WARNING "sbull: unable to get major number\n");
    return -EBUSY;
}
```
device is represented by:

```c
struct sbull_dev {
    int size; /* Device size in sectors */
    u8 *data; /* The data array */
    short users; /* How many users */
    short media_change; /* Flag a media change? */
    spinlock_t lock; /* For mutual exclusion */
    struct request_queue *queue; /*device request queue */
    struct gendisk *gd; /* The gendisk structure */
    struct timer_list timer; /*simulated media changes */
};
```
Initialization in sbull

- structure initialization,
- memory allocation,
- initialization spinlock,

memset (dev, 0, sizeof (struct sbull_dev));
dev->size = nsectors*hardsect_size;
dev->data = vmalloc(dev->size);
if (dev->data == NULL) {
    printk (KERN_NOTICE "vmalloc failure.\n");
    return;
}
Initialization in sbull

- allocation of request queue:

\[
\text{dev->queue = blk_init_queue(sbull_request, \\&dev->lock);}
\]

- \textit{sbull\_request} – our request function,
- does the real read/write,
- \textit{lock} – spinlock, that controls queue access.
Initialization in sbull

- allocation, initialization and installation of `gendisk`:

```c
dev->gd = alloc_disk(SBULL_MINORS);
if (! dev->gd) {
    printk(KERN_NOTICE "alloc_disk failure\n");
    goto out_vfree;
}
dev->gd->major = sbull_major;
dev->gd->first_minor = which*SBULL_MINORS;
dev->gd->fops = &sbull_ops;
dev->gd->queue = dev->queue;
dev->gd->private_data = dev;
snprintf (dev->gd->disk_name, 32, "sbull%c", which + 'a');
set_capacity(dev->gd,  
    nsectors*(hardsect_size/KERNEL_SECTOR_SIZE));
add_disk(dev->gd);
```
allocation, initialization and installation of the *gendisk* structure:

- *SBULL_MINORS* - number of minor numbers,
- *name* – the name is compiled of sbulla, sbullb, ..., in the usersnamespace the partition number can also be added.
- example: `/dev/sbullb3,
- *add_disk* - the last call, after this call the disk requests can occur,
The block device operations

- osnovne metode za dostop do diska,
- sbull simulira "media removal",
- naprava čaka še 30 sekund po zadnjem close,
- tako lahko uporabnik naloži (mounta) disk,
- gonilnik šteje število uporabnikov,
- gonilnik šteje število open/close klicov.
Metoda open

- podobna kot pri znakovnem gonilniku,
- *inode* in *file* kazalca,
- inode ima posebno polje *i_bdev* → *bd_disk*,
- vsebuje kazalec na strukturo *gendisk*,
- ,
Metoda open

```c
static int sbull_open(struct inode *inode, struct file *filp)
{
    struct sbull_dev *dev = inode->i_bdev->bd_disk->private_data;
    del_timer_sync(&dev->timer);
    filp->private_data = dev;
    spin_lock(&dev->lock);
    if (! dev->users)
        check_disk_change(inode->i_bdev);
    dev->users++;  
    spin_unlock(&dev->lock);
    return 0;
}
```

- **del_timer_sync** – zbrisi "media removal" timer,
- zaklene napravo (spinlock),
- **check_disk_change** – preveri ali se je zamenjal medij,
- povečamo število uporabnikov,
static int sbull_release(struct inode *inode, struct file *filp) {
    struct sbull_dev *dev = inode->i_bdev->bd_disk->private_data;
    spin_lock(&dev->lock);
    dev->users--;
    if (!dev->users) {
        dev->timer.expires = jiffies + INVALIDATE_DELAY;
        add_timer(&dev->timer);
    }
    spin_unlock(&dev->lock);
    return 0;
}

- reduce number of users,
- start media removal timer.
"real" discs

- open/release methods,
- spin/stop disc,
- close door (CD-ROM),
- allocate DMA,
- ...,
we will look at a simple example,
usually, this part of the drivers is the most complex,
special emphasis is on speed.
void request(request_queue_t *queue);

- is called when the kernel finds that the driver must:
  - to read/write data or do something,
  - *queue* - request queue,
  - function does not fully meet all requirements,
  - on "serious" devices probably none,
  - method starts processing requests,
  - driver must perform them sometime.
disk needs a lot of time for the entire data transfer process:
  moves the head,
  starts reading to the buffer,
  ...
function is performed in an atomic context (spinlock),
returns as soon as possible,
most of the work is done by the driver later,
there can not be a new request in the queue in the meantime (it is locked).
static void sbull_request(request_queue_t *q)
{
    struct request *req;
    while ((req = elv_next_request(q)) != NULL) {
        struct sbull_dev *dev = req->rq_disk->private_data;
        if (! blk_fs_request(req)) {
            printk (KERN_NOTICE "Skip non-fs request\n");
            end_request(req, 0);
            continue;
        }
        sbull_transfer(dev, req->sector, req->current_nr_sectors, req->buffer, rq_data_dir(req));
        end_request(req, 1);
    }
}

- simple example,
- sbull uses this example,
structure *request* describes the request that the driver should perform,

*elv_next_request* - returns the first unmanaged request from the queue,

*block_fs_request* - returns true if the filesystem requires,

all other requests are only discarded,

start function:

```c
void end_request(struct request *req, int succeeded);
```

”good” requests are sent to *sbull_transfer*. 
- `sector_t sector;` – starting sector index,
- `unsigned long nr_sectors;` – number of sectors to transfer,
- `char *buffer;` – pointer to data buffer,
- `rq_data_dir(struct request *req);` – macro, returns data transfer direction.
we do not have a "real" disk,

simply copy from memory,

simple \texttt{memcpy} call:

```c
static void sbull_transfer(struct sbull_dev *dev, unsigned long sector,
unsigned long nsect, char *buffer, int write)
{
    unsigned long offset = sector*KERNEL_SECTOR_SIZE;
    unsigned long nbytes = nsect*KERNEL_SECTOR_SIZE;
    if ((offset + nbytes) > dev->size) {
        printk (KERN_NOTICE "Beyond-end write (%ld %ld)\n", offset, nbytes);
        return;
    }
    if (write)
        memcpy(dev->data + offset, buffer, nbytes);
    else
        memcpy(buffer, dev->data + offset, nbytes);
}
void end_request(struct request *req, int succeeded);
```
- most important stuff is omitted,
- optimization of transfers,
- multiple calls,
- transfers large amounts of consecutive data,
- simple driver.
mounting and unmounting

- block devices do not have their files in the file system,
- devices of this type are placed in the file system as a new partition,
- new file tree is located in the main file tree,
- new branch,
- when the kernel installs the file system, calls the open driver method,
- in the umount call the kernel calls the release function.
translate source code:
start make in sbull

do this as root:
insmod sbull.

ostart sbull_load:
./sbull_load
use mke2fs to create a file system on the new device:
mke2fs /dev/sbull

load the new file system to main file tree:
mkdir /mnt/jernej
mount -t ext2 /dev/sbull /mnt/jernej

Enjoy:
mkdir /mnt/jernej/lll
cd /mnt/jernej/lll
vi nekaDatoteka
...

Jernej Vičič

Linux device Drivers – Block drivers –