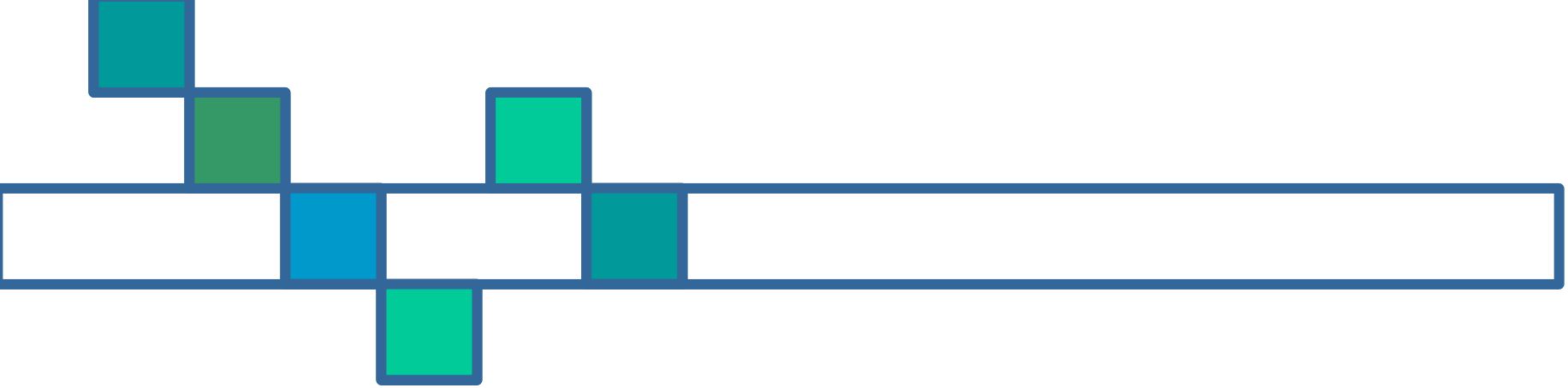

Entrada/Salida
RAMDISK

Oscar Alejandro Ferrer Bernal

Facultad de Informática – Universidad de Las Palmas de Gran Canaria

Ficheros de dispositivos

El acceso a dispositivos se realiza mediante ficheros especiales localizados en el directorio /dev

Cada archivo especial se caracteriza por 3 atributos. tipo (bloque o caracter), numero mayor (controlador que gestiona al dispositivo), numero menor (dispositivo fisico)

```
Zonzamas:/dev# ls -l hda1
brwxrwxr-x    1 root     disk       3,   1 Mar 14  2002 hda1
Zonzamas:/dev# ls -l hdb1
brw-rw----    1 root     disk       3,  65 Mar 14  2002 hdb1
Zonzamas:/dev# ls -l dri/card0
crw-rw-rw-    1 root     root     226,   0 Jun  2 16:39 dri/card0
```

Ficheros de dispositivo

Dispositivos de bloque

Corresponde a dispositivos estructurados en bloques (discos duros) a los que se accede proporcionando un numero de bloque a leer o escribir. Las entradas/salidas se efectuan mediante las funciones del bufer cache

Dispositivos de caracter

Son dispositivos no estructurados como los puertos serie o paralelo. Se puede leer o escribir datos byte a byte generalmente de forma secuencial

Otros conceptos

Bufer Cache

Memoria RAM usada como cache de disco

Virtual Filesystem Switch (VFS)

Nivel del kernel encargado de mantener los descriptores de ficheros y manejar la coexistencia de diversos tipos de sistemas de fichero.

¿Como se accede a los ficheros de dispositivo?

La entrada/salida sobre dispositivos se efectua por las mismas primitvas que las utilizadas para leer y escribir datos en archivos regulares. La apertura de un dispositivo se efectua con la primitiva open, el nucleo devuelve un descriptor de entrada/salida sobre el que se puede acceder utilizando las primitivas read,write y lseek. Tras la utilizacion puede cerrarse el dispositivo mediante la primitiva close.

- read/write
- lseek
- ioctl
- open/close
- select

fs/devices.c

¿Como sabe el kernel que controlador debe usar para cada dispositivo?

Dentro de devices.c se mantienen dos tablas blkdevs y chrdevs.

Numero del controlador (numero mayor)

```
struct device_struct {
        const char *name;
         struct file_operations *fops;
};
```

fs/devices.c

Registrar un nuevo controlador

```
int register_blkdev(unsigned int major, const char * name,
                    struct file_operations *fops)
```

Obtener los punteros a las funciones del dispositivo

```
struct file_operations * get_blkfops(unsigned int major)
```

include/linux/fs.h

```
struct file_operations {
        loff_t (*llseek) (struct file *, loff_t, int);
        ssize_t (*read) (struct file *, char *, size_t, loff_t *);
        ssize_t (*write) (struct file *, const char *, size_t, loff_t *);
        int (*readdir) (struct file *, void *, filldir_t);
        unsigned int (*poll) (struct file *, struct poll_table_struct *);
        int (*ioctl) (struct inode *, struct file *, unsigned int, unsigned long);
        int (*mmap) (struct file *, struct vm_area_struct *);
        int (*open) (struct inode *, struct file *);
        int (*flush) (struct file *);
        int (*release) (struct inode *, struct file *);
        int (*fsync) (struct file *, struct dentry *);
        int (*fasync) (int, struct file *, int);
        int (*check_media_change) (kdev_t dev);
        int (*revalidate) (kdev_t dev);
        int (*lock) (struct file *, int, struct file_lock *);
};
```

driver/block/ll_rw_blk.c

Este fichero implementa las funciones de gestion de la lista de peticiones de entrada/salida .

Esta funcion es llamada por el buffer cache, los sitemas de archivosy el modulo de entradas/salidas de bloques a fin de efectuar una entrada/salida

void ll_rw_block(int rw, int nr, strcut buffer_head *bh[])

fs/block_dev.c

Este fichero implementa las funciones de lectura y escritura de datos sobre dispositivos accesibles en modo bloque. Estas funciones son las promitivas del bufer cache para acceder a las memorias intermedias asociadas a los dispositivos.

operación de escritura en dispositivos de bloque

```
ssize_t block_write(struct file * filp, const char * buf,
                    size_t count, loff_t *ppos)
```

operación de lectura en dispositivos de bloque

```
ssize_t block_read(struct file * filp, char * buf,
                   size_t count, loff_t *ppos)
```

fs/select.c

La funcion de select se implementa dentro de fs/select.c.

Para cada descriptor de entrada/salida especificado se efectua una comprobacion para determinar si es posible la entrada/salida. Si no es asi, el proceso actual se coloca en la cola de espera correspondiente al archivo. Cuando la entrada/salida es posible en uno de los descriptores el proceso suspendido puede ser despertado.

RAMDISK /drivers/block/rd.c

La implementacion del gestor de disco en memoria esta en el fichero /drivers/block/rd.c.

Este gestor permite reservar un area de memoria y utilizarla como un disco virtual.

Opcionalmente el driver ramdisk puedes estar compilado con la opcion initrd. Esto permite cargar un disco ram desde el boot loader permitiendo que el inicio del sistema ocurra en dos fases. Primero se carga un kernel con un conjunto de drivers minimo y a continuacion los modulos adicionales son cargados desde el initrd.

RAMDISK

```
#include <linux/config.h>
/*   Includes
          .
          .
          .
*/
#include <asm/byteorder.h>

extern void wait_for_keypress(void);

#define MAJOR_NR RAMDISK_MAJOR
#include <linux/blk.h>
#define RDBLK_SIZE_BITS        9
#define RDBLK_SIZE             (1<<RDBLK_SIZE_BITS)


#define NUM_RAMDISKS 16
```

RAMDISK

```
#ifndef MODULE

#define RD_LOADER
#define BUILD_CRAMDISK

void rd_load(void);
static int crd_load(struct file *fp, struct file *outfp);

#ifdef CONFIG_BLK_DEV_INITRD
static int initrd_users = 0;
#endif
#endif

/* Various static variables go here.  Most are used only in the RAM disk code.
 */
static unsigned long rd_length[NUM_RAMDISKS];
static int rd_hardsec[NUM_RAMDISKS];
static int rd_blocksizes[NUM_RAMDISKS];
```

RAMDISK

```
static int rd_kbsize[NUM_RAMDISKS];
/


int rd_size = CONFIG_BLK_DEV_RAM_SIZE;

#ifndef MODULE
int rd_doload = 0;
int rd_prompt = 1;
int rd_image_start = 0;
#ifdef CONFIG_BLK_DEV_INITRD
unsigned long initrd_start,initrd_end;
int mount_initrd = 1;
int initrd_below_start_ok = 0;
#endif
#endif
```

RAMDISK

```
static void rd_request(void)
{
        unsigned int minor;
        unsigned long offset, len;

repeat:
        if (!CURRENT)
                return;

        INIT_REQUEST;

        minor = MINOR(CURRENT->rq_dev);

        if (minor >= NUM_RAMDISKS) {
                end_request(0);
                goto repeat;
        }

        offset = CURRENT->sector << RDBLK_SIZE_BITS;
        len = CURRENT->current_nr_sectors << RDBLK_SIZE_BITS;
```

RAMDISK

```
 if ((offset + len) > rd_length[minor]) {
        end_request(0);
        goto repeat;
    }

    if ((CURRENT->cmd != READ) && (CURRENT->cmd != WRITE)) {
        printk(KERN_INFO "RAMDISK: bad command: %d\n", CURRENT->cmd);
        end_request(0);
        goto repeat;
    }


    if (CURRENT->cmd == READ)
        memset(CURRENT->buffer, 0, len);
    else
        mark_buffer_protected(CURRENT->bh);

    end_request(1);
    goto repeat;
}
```

RAMDISK

```
static int rd_ioctl(struct inode *inode, struct file *file, unsigned int cmd, uns
igned long arg)
{
        unsigned int minor;

        if (!inode || !inode->i_rdev)
                return -EINVAL;

        minor = MINOR(inode->i_rdev);

        switch (cmd) {
                case BLKFLSBUF:
                        if (!capable(CAP_SYS_ADMIN)) return -EACCES;
                        destroy_buffers(inode->i_rdev);
                        break;
               case BLKGETSIZE:
                        if (!arg)  return -EINVAL;
                        return put_user(rd_length[minor] >> RDBLK_SIZE_BITS, (long *) arg);
```

RAMDISK

```
        case BLKSSZGET:
                if (!arg)  return -EINVAL;
                return put_user(rd_blocksizes[minor], (int *)arg);

        RO_IOCTLS(inode->i_rdev, arg);

        default:
                return -EINVAL;
        };

        return 0;
}
```

RAMDISK

```
#ifdef CONFIG_BLK_DEV_INITRD
static ssize_t initrd_read(struct file *file, char *buf,
                           size_t count, loff_t *ppos)
{
        int left;

        left = initrd_end - initrd_start - *ppos;
        if (count > left) count = left;
        if (count == 0) return 0;
        copy_to_user(buf, (char *)initrd_start + *ppos, count);
        *ppos += count;
        return count;
}
static int initrd_release(struct inode *inode,struct file *file)
{
        unsigned long i;

        if (--initrd_users) return 0;
        for (i = initrd_start; i < initrd_end; i += PAGE_SIZE)
                free_page(i);
        initrd_start = 0;
        return 0;
}
```

RAMDISK

```
static struct file_operations initrd_fops = {
        NULL,           /* lseek */
        initrd_read,    /* read */
        NULL,           /* write */
        NULL,           /* readdir */
        NULL,           /* poll */
        NULL,           /* ioctl */
        NULL,           /* mmap */
        NULL,           /* open */
        NULL,           /* flush */
        initrd_release, /* release */
        NULL            /* fsync */
};

#endif
```

RAMDISK

```
static int rd_open(struct inode * inode, struct file * filp)
{
#ifdef CONFIG_BLK_DEV_INITRD
        if (DEVICE_NR(inode->i_rdev) == INITRD_MINOR) {
                if (!initrd_start) return -ENODEV;
                initrd_users++;
                filp->f_op = &initrd_fops;
                return 0;
        }
#endif

        if (DEVICE_NR(inode->i_rdev) >= NUM_RAMDISKS)
                return -ENXIO;

        MOD_INC_USE_COUNT;

        return 0;
}
```

RAMDISK

```
static int rd_release(struct inode * inode, struct file * filp)
{
        MOD_DEC_USE_COUNT;
        return 0;
}

static struct file_operations fd_fops = {
        NULL,           /* lseek - default */
        block_read,     /* read - block dev read */
        block_write,    /* write - block dev write */
        NULL,           /* readdir - not here! */
        NULL,           /* poll */
        rd_ioctl,       /* ioctl */
        NULL,           /* mmap */
        rd_open,        /* open */
        NULL,           /* flush */
        rd_release,     /* module needs to decrement use count */
        block_fsync     /* fsync */
};
```

RAMDISK

```
/* This is the registration and initialization section of the RAM disk driver */
__initfunc(int rd_init(void))
{
        int             i;

        if (register_blkdev(MAJOR_NR, "ramdisk", &fd_fops)) {
                printk("RAMDISK: Could not get major %d", MAJOR_NR);
                return -EIO;
        }

        blk_dev[MAJOR_NR].request_fn = &rd_request;

        for (i = 0; i < NUM_RAMDISKS; i++) {
                /* rd_size is given in kB */
                rd_length[i] = (rd_size << BLOCK_SIZE_BITS);
                rd_hardsec[i] = RDBLK_SIZE;
                rd_blocksizes[i] = BLOCK_SIZE;
                rd_kbsize[i] = (rd_length[i] >> BLOCK_SIZE_BITS);
        }
```

RAMDISK

```
        hardsect_size[MAJOR_NR] = rd_hardsec;          /* Size of the RAM disk b
locks */
        blksize_size[MAJOR_NR] = rd_blocksizes;        /* Avoid set_blocksize()
check */
        blk_size[MAJOR_NR] = rd_kbsize;               /* Size of the RAM disk i
n kB  */

        printk("RAM disk driver initialized:  %d RAM disks of %dK size\n",
                                                NUM_RAMDISKS, rd_size);

        return 0;
}
```

RAMDISK

```
#ifdef MODULE
MODULE_PARM     (rd_size, "1i");
MODULE_PARM_DESC(rd_size, "Size of each RAM disk.");
int init_module(void)
{
        int error = rd_init();
        if (!error)
                printk(KERN_INFO "RAMDISK: Loaded as module.\n");
        return error;
}
void cleanup_module(void)
{
        int i;

        for (i = 0 ; i < NUM_RAMDISKS; i++)
                destroy_buffers(MKDEV(MAJOR_NR, i));

        unregister_blkdev( MAJOR_NR, "ramdisk" );
        blk_dev[MAJOR_NR].request_fn = 0;
}
#endif  /* MODULE */
```

RAMDISK

```
initfunc(int
identify_ramdisk_image(kdev_t device, struct file *fp, int start_block))
{
  }
__initfunc(static void rd_load_image(kdev_t device, int offset, int unit)){
 }
__initfunc(static void rd_load_disk(int n)){
}
__initfunc(void rd_load(void))
{
}
__initfunc(void rd_load_secondary(void))
{
}
#ifdef CONFIG_BLK_DEV_INITRD
__initfunc(void initrd_load(void))
{
}
#endif
```

FIN