DISEÑO DE SISTEMAS OPERATIVOS

ENTRADA / SALIDA

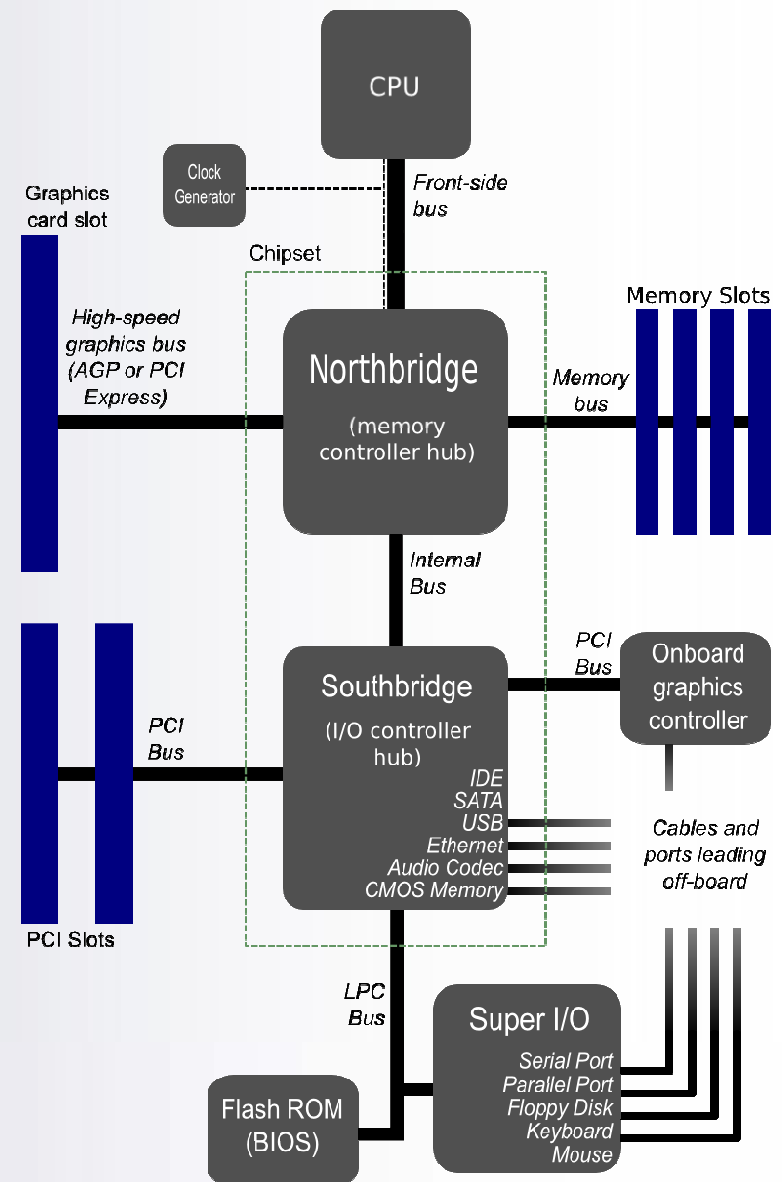
Víctor Olivares Guedes
Sergio Rosas Navarro

Entrada / Salida

- Arquitectura de Entrada/Salida
  - Puertos de Entrada/Salida
  - Recursos
  - Memoria Compartida de E/S
- Ficheros de Dispositivos
  - Ficheros de dispositivos en VFS
- Modelo de Drivers de Dispositivos
  - El sistema de ficheros sysfs
  - Estructuras
- Drivers de Dispositivos
  - Dispositivos de Caracteres
  - Dispositivos de Bloques

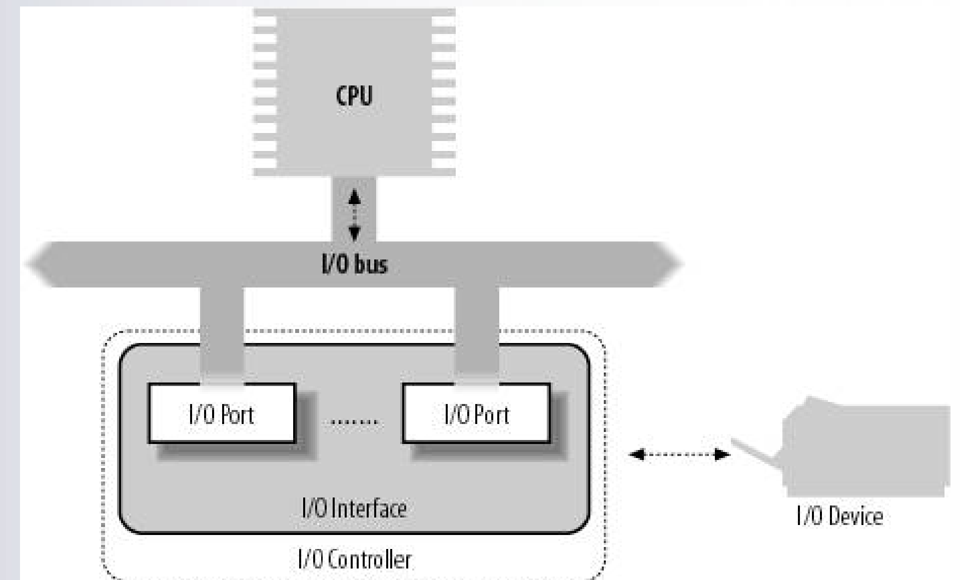
Arquitectura del sistema

- Varios buses. PCI, ISA, USB,etc.
- FSB, BSB, System bus (PCI).
- Controlados por puente norte y sur.
  - Norte maneja CPU, RAM y AGP.
  - Sur maneja periféricos más lentos.



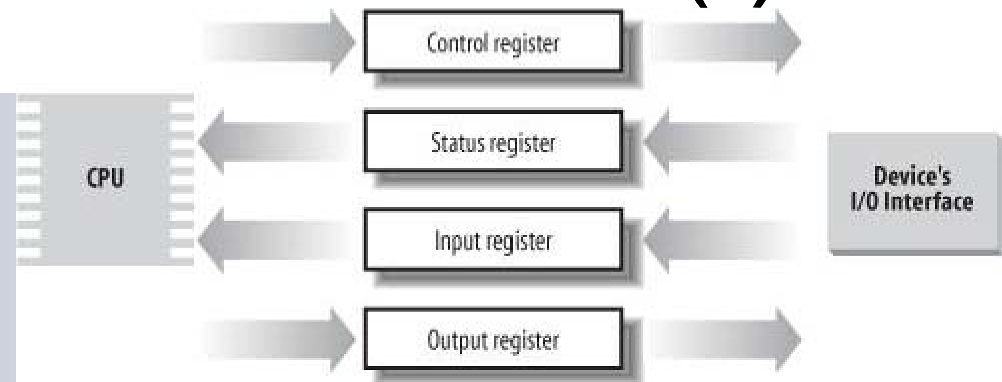
Arquitectura de Entrada/Salida

- El procesador se comunica con los dispositivos a través de los puertos de entrada/salida.
- En la arquitectura IBM PC, se disponen de 65536 puertos de entrada/salida.
- Los puertos son de 8-bits pero varios puertos consecutivos pueden ser tratados como puertos de 16 o 32 bits.
  - Todo dispositivo conectado al bus de entrada/salida tiene su propio conjunto de puertos.



Puertos de Entrada/Salida (I)

- El procesador dispone de cuatro instrucciones para acceder a los puertos:
  - **in**, **ins**
  - **out**, **outs**



- Los puertos acceden a registros del controlador del dispositivo.
- El kernel cuenta con las siguientes funciones auxiliares para acceder a los puertos:
  - **inb**, **inw**, **inl**: Leen 1, 2 o 4 bytes respectivamente de un puerto.
  - **outb**, **outw**, **outl**: Escriben 1, 2 o 4 bytes a un puerto.
- También hay variaciones de estas funciones para secuencias (insb, ..., outsl) o que añaden instrucciones para introducir una pausa (inb_p, ..., outl_p).

Puertos de Entrada/Salida (II)

- Los puertos de E/S también pueden ser mapeados en direcciones de memoria física (mov, and, or, ...).

- La E/S mapeada en memoria se usa ampliamente.
  - Acceso más rápido
  - Se puede combinar con DMA.

Recursos (I)

- Son partes de entidades que se asignan a un driver en exclusiva.
  - Puertos E/S, direcciones de memoria o números de interrupción.
- Todos los recursos del mismo tipo se agrupan en un árbol de recursos.
- Existen funciones para manejar los recursos:
  - **request_resource()**: asigna un recurso a un dispositivo de E/S.
  - **release_resource()**: libera un recurso previamente asignado.
- La estructura de datos de un recurso es la siguiente:

```
struct resource {
        resource_size_t start; //Dirección de inicio del recurso
        resource_size_t end; //Dirección final
        const char *name; //Nombre del dueño del recurso
        unsigned long flags; //Distintas opciones
     //Padre, hermano (siguiente nodo en la lista) y primer
     //hijo del nodo.
        struct resource *parent, *sibling, *child;

 };
```

Recursos (II)

- La implementación se encuentra en *kernel/resource.c*
- *request_resource recibe dos argumentos:*
  - ***root**: descriptor del recurso raíz*
  - ***new**: descriptor del recurso deseado*

```
int request_resource(struct resource *root, struct resource *new) {
      struct resource *conflict;

      write_lock(&resource_lock);
      conflict = __request_resource(root, new);
      write_unlock(&resource_lock);
      return conflict ? -EBUSY : 0;
}
```

- Devuelve 0 si se pudo solicitar el recurso o -EBUSY si ya estaba asignado.
- Funciona como reemplazo de *check_resource*.

Recursos (III)

```
static struct resource * _request_resource (struct resource *root, struct resource *new) {
    unsigned long start = new->start;
    unsigned long end = new->end;
    struct resource *tmp, **p;

    if (end < start) return root;
    if (start < root->start) return root;
    if (end > root->end) return root;
    p = &root->child;
    for (;;) {
        tmp = *p;
        if (!tmp || tmp->start > end) { // Si no hay recurso o tiene una dirección posterior
            new->sibling = tmp;
            *p = new; // añade el recurso a la lista de hermanos
            new->parent = root;
            return NULL;
        }
        p = &tmp->sibling; // si no ha encontrado un 'hueco' aún continua por el hermano
        if (tmp->end < start)
            continue;
        return tmp; // si no hay hueco para el recurso devuelve el recurso actual
```

Recursos (IV)

- release_resource recibe como argumento el recurso a librerar.
- Devuelve 0 si lo pudo liberar -EINVAL si no pudo encontrar el recurso.

```
int release_resource(struct resource *old) {
        int retval;

        write_lock(&resource_lock);
        retval = __release_resource(old);
        write_unlock(&resource_lock);
        return retval;
}
```

Recursos (V)

```
static int __release_resource(struct resource *old) {
        struct resource *tmp, **p;

        p = &old->parent->child; // p contiene la dirección del 1º hijo
        for (;;) {
                tmp = *p;  // tmp contiene el contenido de p
                if (!tmp)    // Si ya no quedan recursos, sale del bucle
                        break;
                if (tmp == old) {
                        *p = tmp->sibling; // Pasamos al siguiente recurso de la lista
                        old->parent = NULL; // Pone el padre a nulo
                        return 0;
                }
                p = &tmp->sibling;
        }
        return -EINVAL;
}
```

Memoria Compartida de E/S (I)

- Muchos dispositivos incluyen memoria propia a la que se accede mediante memoria compartida de E/S.

- Dependiendo del dispositivo o el bus la memoria compartida puede tener diferentes rangos de direcciones:
  - ISA: Mapeado en el rango 0xa0000 a 0xfffff (direcciones de 16 bits).
  - PCI: Direcciones de 32-bit. Cercanas al límite de 4GB.

- Los drivers deben traducir las direcciones de la memoria física correspondientes a E/S en direcciones virtuales.

- En algunos casos, la tabla de páginas del núcleo debe ser modificada, para ello se utilizan las funciones:

  Ioremap(), ioremap_nocahce(), iounmap().

Memoria Compartida de E/S (II)

- En muchas arquitecturas, además del PC, la memoria de E/S puede ser accedida directamente.
- No obstante, el núcleo proporciona una interfaz común para todas las plataformas:
  - **readb, readw, readl:** Leen 1,2 o 4 bytes de la memoria de E/S.
  - **writeb**, **writew**, **writel**: Escriben 1,2 o 4 bytes en la memoria de E/S.
  - **memcpy_toIo**, **memcpy_fromIo**: Copian de/a la memoria de E/S.
  - **memset_Io**: Inicializa un segmento de la memoria de E/S a un valor fijo.

Ficheros de Dispositivos (I)

- Todo sistema Unix se basa en ficheros.
- Todo dispositivo tiene un fichero asociado mediante el cual podemos hacer operaciones de entrada/salida.
- Hay dos tipos de ficheros de dispositivos, atendiendo al hardware que representan:
  - **char**: Dispositivos de **caracteres.**
  - **block**: Dispositivos de **bloques.**
- Los ficheros de dispositivos son ficheros especiales que utilizan dos números (major y minor) para identificar un dispositivo.

- Éstos ficheros especiales normalmente se encuentran en la ruta /dev aunque pueden crearse en cualquier otra ruta.

Ficheros de Dispositivos (II)

- La llamada a sistema **mknod** permite crear ficheros de dispositivos especificando ruta, tipo y el identificador de dispositivo (major y minor).
- Pueden existir ficheros para dispositivos inexistentes y dispositivos para los que no hay fichero. P.e. la tarjeta de red o la salida o entrada estándar.
- Soluciones para el manejo de ficheros de dispositivos:
  - **mkdev:** Se crea un script al iniciar el sistema que se asegure de que existen todos los dispositivos necesarios (obsoleta).
  - **devfs**: Se genera un sistema de ficheros virtual que de acceso a todos los dispositivos (obsoleta).
  - **udev**: Se crean y destruyen los ficheros a medida que se conectan/desconectan dispositivos. Adicionalmente permite elegir nombres para esos dispositivos y programar acciones al cambiar el estado de los mismos.

Ficheros de dispositivos en VFS (I)

- "Esconde" las diferencias entre ficheros normales y de dispositivo
- Se añaden al inode las operaciones del dispositivo.
- Cada operación que se realice sobre el fichero llamará a la función del dispositivo correspondiente.
- Cuando se intenta abrir un fichero de dispositivo:
  - Si se detecta que es un fichero especial se llama a init_special_inode().
  - Se inicializa el campo i_rdev del inode con el major y minor numbers.
  - Inicializa el campo i_fop según el tipo de dispositivo (def_chr_fops o def_blk_fops)
- *def_chr_fops* y *def_blk_fops* son tablas que contienen punteros a las operaciones sobre ficheros (open, read, write..) para ficheros de caracteres y bloques respectivamente.

Ficheros de dispositivos en VFS (II)

```
void init_special_inode(struct inode *inode, umode_t mode, dev_t rdev)
{
        inode->i_mode = mode;
        if (S_ISCHR(mode)) { // Si es de tipo carácter establece la dirección
// de las operaciones genéricas para dispositivos de caracteres.
                inode->i_fop = &def_chr_fops;
                inode->i_rdev = rdev; // inicializa el campo i_rdev del inode con el
    major y minor
        } else if (S_ISBLK(mode)){ // Si es de tipo bloque, guarda la dirección de
//las operaciones genéricas para dispositivos de bloques.
                inode->i_fop = &def_blk_fops;
                inode->i_rdev = rdev; // inicializa el campo i_rdev del inode con el
    major y minor
        }
        ... // hay más ficheros especiales como sockets o tuberías que no vemos
            aquí
}
```

Ficheros de Dispositivos en VFS (III)

- Operaciones sobre ficheros de caracteres (*fs/char_dev.c*):

```
const struct file_operations def_chr_fops = {
        .open = chrdev_open,
};
//El resto de operaciones deben ser asignadas por la implementación del driver
```

- Operaciones sobre ficheros de bloques (*fs/block_dev.c*):

```
const struct file_operations def_blk_fops = {
        .open           = blkdev_open,
        .release        = blkdev_close,
        .llseek         = block_llseek,
        .read           = do_sync_read,
        .write          = do_sync_write,
        .mmap           = generic_file_mmap,
        .unlocked_ioctl = block_ioctl,
...
};
```

Modelo de Drivers de Dispositivos

- Para facilitar y organizar la construcción de drivers, Linux ofrece una serie de estructuras y funciones auxiliares que ofrecen una visión unificada de buses, dispositivos y drivers.

- Las principales estructuras son:
  - **struct device_driver** (drivers o manejadores)
  - **struct bus_type** (buses)
  - **struct device** (dispositivos)

    – El Modelo de Drivers de Dispositivos tiene una interfaz unificada accesible a través del sistema de ficheros virtual sysfs.

Sistema de ficheros sysfs

- El sistema de ficheros sysfs es similar a proc pero mucho más organizado y oriendado a los dispositivos y drivers del sistema.
- Normalmente está montado en /sys y contiene:
  - /sys/**block**: Dispositivos de bloques de cualquier bus.
  - /sys/**bus**: Buses del sistema, donde están los dispositivos
  - /sys/**devices**: Dispositivos del sistema organizados por buses.
  - /sys/**class**: Clases de dispositivos (audio, tarjetas gráficas, de red...)
  - /sys/**modules**: Drivers registrados en el núcleo
  - /sys/**power**: Ficheros para manejar estado de energía de distintos dispositivos.
  - /sys/**firmware**: Ficheros para manejar el firmware de algunos dispositivos.

Estructura device_driver

```
struct device_driver {
    const char              * name;
    struct bus_type          * bus; //Tipo de bus del dispositivo
    struct kobject          kobj; //Objeto que representa el dispositivo en la jerarquía
  del modelo.
    struct klist            klist_devices; //Dispositivos asociados a este driver.
    struct klist_node       knode_bus;
    struct module           * owner;
    const char              * mod_name;     /* used for built-in modules */
    struct module_kobject   * mkobj;
    int      (*probe)       (struct device * dev);//Comprueba si un dispositivo existe o
                                                  //puede ser manejado por el driver
    int      (*remove)      (struct device * dev); //Se llama al borrar el driver del
  sistema.
    void     (*shutdown)    (struct device * dev); //Se llama al apagar el sistema.
    int      (*suspend)     (struct device * dev, pm_message_t state);
    int      (*resume)      (struct device * dev);
};
```

Estructura bus_type

```
struct bus_type {
        const char              * name;
        struct module            * owner;

        struct kset             subsys; //Subsistema dentro de sysfs al que pertenece
        struct kset             drivers; //Drivers conocidos para ese bus
        struct kset             devices; //Dispositivos conectados al bus
        struct klist            klist_devices;
        struct klist            klist_drivers;
   // Funcion que comprueba si el bus puede manejar un cierto dispositivo o driver
    de dispositivo.
        int             (*match)(struct device * dev, struct device_driver * drv);
        int             (*probe)(struct device * dev);
        int             (*remove)(struct device * dev);
        void             (*shutdown)(struct device * dev);
       //Campos omitidos…
        unsigned int drivers_autoprobe:1;
 };
```

Estructura device

```
struct device {
    ...
    char    bus_id[BUS_ID_SIZE];    // identificador dentro del bus
    unsigned            is_registered:1;
    struct device_attribute uevent_attr;
    struct device_attribute *devt_attr;
    struct semaphore        sem;    // Semáforo para sicronizar llamadas de su driver
    struct bus_type * bus;          // Bus en el que se encuentra
    struct device_driver *driver;   // Driver que lo maneja
    void            *driver_data;   // Datos privados del driver
    void            *platform_data; // Datos del dispositivo específicos de la plataforma
    struct dev_pm_info      power;
    u64             *dma_mask;      // Máscara DMA
    struct list_head        dma_pools;      // Lista de DMA pools
    struct dev_archdata     archdata;
    struct class            *class;         // Clase del dispositivo
    dev_t                   devt;           // Identificador del dispositivo
    void    (*release)(struct device * dev);
};
```

Drivers de Dispositivos

- Los drivers de dispositivos además de comunicarse con el controlador implementan las operaciones del VFS (open, read, write, lseek, ioctl, ...)

- Como cada controlador de E/S tiene sus propios comandos, estados y características, prácticamente existe un driver para cada dispositivo.

Registro de Drivers (I)

- Cada operación sobre un fichero de dispositivo debe traducirse a una llamada a una rutina de su driver.
- El driver debe registrarse creando un *device_driver* y llamando a driver_register() para registrarlo.
- Si el driver esta compilado en el kernel, el registro se produce durante la fase de carga del núcleo. Si el driver está compilado como módulo, se cargará cuando sea necesario.
- Cuando se registra un driver, el kernel comprueba para cada dispositivo que quede sin manejar si el driver puede manejarlo mediante la función probe().
- Aunque esté registrado, el driver aún no tiene recursos asignados.

Registro de Drivers (II)

```
int driver_register(struct device_driver * drv) {
        if ((drv->bus->probe && drv->probe) ||
            (drv->bus->remove && drv->remove) ||
            (drv->bus->shutdown && drv->shutdown)) {
                printk(KERN_WARNING "Driver '%s' needs updating - please use
    bus_type methods\n", drv->name);
        }
        klist_init(&drv->klist_devices, NULL, NULL);
        init_completion(&drv->unloaded);
        return bus_add_driver(drv);
}

int bus_add_driver(struct device_driver *drv) {
        struct bus_type * bus = get_bus(drv->bus);
        int error = 0;
        if (!bus)
                return 0;
       pr_debug("bus %s: add driver %s\n", bus->name, drv->name);
```

Registro de Drivers (III)

```
        error = kobject_set_name(&drv->kobj, "%s", drv->name);
    if (error)
            goto out_put_bus;
    drv->kobj.kset = &bus->drivers;
    if ((error = kobject_register(&drv->kobj)))
            goto out_put_bus;

    error = driver_attach(drv);
    if (error)
            goto out_unregister;
    klist_add_tail(&drv->knode_bus, &bus->klist_drivers);
    module_add_driver(drv->owner, drv);

    error = driver_add_attrs(bus, drv);
    if (error) {
            printk(KERN_ERR "%s: driver_add_attrs(%s) failed\n", __FUNCTION__, drv-
>name);
    }

571 }
```

Registro de Drivers (IV)

```
        error = add_bind_files(drv);
          if (error) {
                printk(KERN_ERR "%s: add_bind_files(%s)
  failed\n",  __FUNCTION__, drv->name);
         }


         return error;
 out_unregister:
         kobject_unregister(&drv->kobj);
 out_put_bus:
         put_bus(bus);
         return error;
}
```

Inicialización del Driver

- Para controlar la asignación de recursos se sigue el siguiente esquema:
  - Un contador mantiene el número de procesos que operan o esperan para operar en fichero de dispositivo.
  - Al abrir el dispositivo, si este contador está a 0, se realiza la asignación de recursos al driver de dispositivo (IRQ, marcos de página, …)
  - Cada vez que un proceso cierra el fichero de dispositivo, se comprueba si el contador ha llegado a 0. Si es así se liberan todos los recursos hasta que se intente acceder de nuevo al dispositivo.

Supervisión de E/S

- Un manejador que comienza un operación E/S debe contar con unas técnicas de supervisión que le señale tanto la terminación de la operación como un time-out.
- Existen 2 Técnicas:
  - **polling**: La CPU comprueba periódicamente el registro de estado hasta que su valor señale que la operación ha terminado.
  - **interrupciones**: El controlador de E/S es capaz de avisar el final de la operación. El driver tendrá que crear una rutina de atención a la interrupción.

Dispositivos de Caracteres

- Los drivers de dispositivos de caracteres son relativamente simples porque no tienen que tener en cuenta buffers, caches, planificación de E/S, etc.
- Un driver de un dispositivo de caracteres viene descrito por la estructura cdev:

```
struct cdev {
    struct kobject kobj; // kobject correspondiente
    struct module *owner; // modulo en el que se encuentra el driver
    const struct file_operations *ops; // tabla de operaciones sobre el fichero
    struct list_head list; // lista de ficheros de dispositivos para este dispositivo
    dev_t dev; // números major y minor asignados al dispositivo
    unsigned int count; // número de dispositivos asignados al driver
};
```

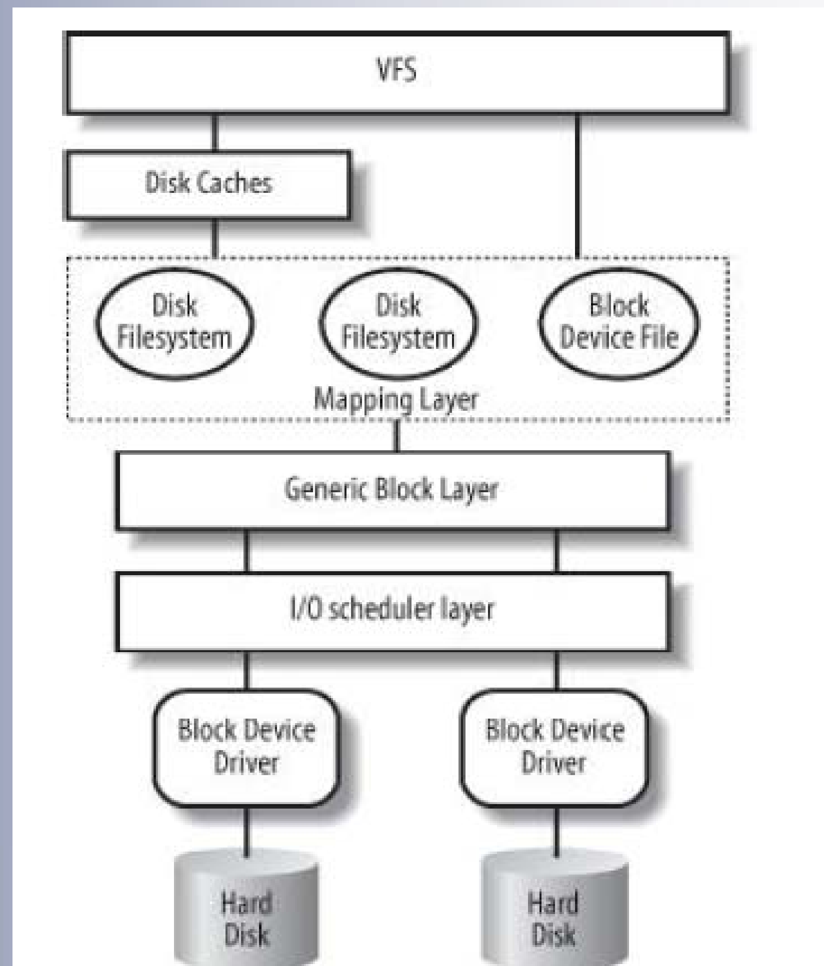
Manejadores de dispositivos de bloque

- Las operaciones de acceso a dispositivos de bloque requieren varios milisegundos para completar una transferencia. Esto es debido a que los discos deben mover las cabezas hasta alcanzar la posición del dato requerido.

- Para alcanzar una mejora aceptable, los dispositivos de bloque transfieren varios bytes adyacentes de una sola vez, éstos forman un sector.

Manejadores de dispositivos de bloque

- Una operación sobre un dispositivo de bloque afecta a varios componentes del Kernel.



Sectores, bloques y segmentos.

- Durante el proceso de I/O se trabaja con diferentes unidades según la capa en la que se encuentre el proceso en un momento dado, que pueden ser:

  - Sectores: es un grupo de bytes adyacentes, en la mayoría de discos son de 512 bytes. Los drivers de dispositivo de bloques transfieren sectores, siendo esta la unidad mínima de transferencia.

  - Bloques: es la unidad básica de transferencia para el VFS. Cada bloque necesita su propio buffer de bloque, que es un área de memoria RAM usada por el Kernel para almacenar el contenido del bloque.

  - Segmentos: se trata de un grupo de sectores adyacentes en disco. El driver de dispositivo de bloque debe ser capaz de manejar segmentos.

Generic Block Layer.

- La capa de bloque genérico es un componente que maneja las peticiones para todos los dispositivos de bloque del sistema. Para ello hace uso de dos elementos principales:

  -**La estructura Bio**: es un descriptor de una operación I/O sobre un dispositivo de bloque. Hay una estructura Bio por cada petición de entrada/salida. En cada petición están comprometidos uno o más bloques que son organizados en segmentos

  Cuando la capa de bloque genérico empieza una nueva operación I/O crea una nueva estructura bio invocando la función **bio_alloc()**

  - **gendisk:** Un disco es un dispositivo lógico de bloques que es manejado por la generic block layer . Para ello un disco es representado mediante el objeto gendisk. De esta manera se ocultan los detalles del disco a los otros componentes del sistema operativo, ya que un disco puede ser tanto dispositivo hardware de bloque como un dispositivo virtual (p.e particiones)

Generic Block Layer.

Estructura BIO (I/O Block)

Esta estructura representa una operación como una lista de SEGMENTOS. Un segmento es un conjunto de buffers (es decir, bloques de disco en memoria) que son contiguos en una página física de memoria. Por tanto un segmento puede ser un sólo buffer o varios Buffers en una misma página.

Generic Block Layer.

Una estructura Bio tiene la siguiente forma:

```
struct bio {
        sector_t bi_sector;              /* primer sector (512 bytes) a ser transferido
                                         para esta bio */
        struct bio *bi_next;              /* lista de peticiones */
        struct block_device *bi_bdev;/* dispositivo de bloque asociado */
        unsigned long bi_flags;           /* flags de status y comandos   */
        unsigned long bi_rw;             /* bits bajos READ/WRITE, bits altos
                                         prioridad */
        unsigned short bi_vcnt;          /* número de segmentos en el bio_vec
                                         array */
        unsigned short bi_idx;        /* índice actual sobre bi_io_vec */
        unsigned short bi_phys_segments;
        unsigned short bi_hw_segments;
        unsigned int bi_size;            /* Tamaño de los datos a ser transferidos,
                                         en bytes */
        unsigned int bi_max_vecs;      /* máximos bio_vecs posibles */
        struct bio_vec *bi_io_vec;      /* array de bio_vec */
        bio_end_io_t *bi_end_io;       / * método de terminación de I/O */
        atomic_t bi_cnt;                 /* contador de uso */
        void *bi_private;                /* método privado */
        bio_destructor_t *bi_destructor;  /* método destructor */
    };
```

Generic Block Layer.

```
struct bio_vec {
  struct page *bv_page;  /*puntero a la página física donde reside
el/los buffers */
  unsigned int bv_len;  /* longitud en bytes del segmento representado
*/
  unsigned int bv_offset;/* el byte offset dentro de la página donde
comienza el primer buffer */
}
```

Cada segmento es representado por una estructura *bio_vec*, que son agrupadas en la lista **bi_io_vec**. Estas estructuras actúan como vectores que describen cada situación del segmento en la memoria física, es decir, la situación en las páginas de memoria. El primer segmento involucrado en la operación de entrada/salida es apuntado por bi_io_vec. Cada segmento adicional sigue al primero, con un total de **bi_vcnt** segmentos en la lista (es decir, bi_vcnt elementos bi_vec en la lista). A medida que se completan las operaciones sobre cada segmento, el campo bi_idx es actualizado para que apunte el siguiente segmento pendiente del vector b_io_vec.

Generic Block Layer.

Objeto gendisk

Tipo	Campo	Descripción
Int	Major	Número mayor de el disco
Int	First_minor	Número menor asociado con el disco.
Int	Minor	Rango de números menores asociados con el disco
Char[32]	Disk_name	Nombre del convencional del disco
Struct hd_struct * *	Part	Array de descriptores de partición para el disco
Struct Block_device_operation *	Fops	Puntero a la tabla de métodos para dispositivos de bloque
Struct request _queue *	Queue	Puntero a la cola de peticiones de el disco

Generic Block Layer.

Tipo	Campo	Descripción
Sector_t	Capacity	Tamaño del área de almacenamiento del disco (en número de sectores)
int	Flags	Flags que describen el tipo de disco
int	Policy	Está a 1 si el disco es de solo lectura
Int	Int_flight	Número de operaciones I/O en curso

Generic Block Layer.

- **Major, first_minor:** debido a que los discos duros se encuentran eventualmente divididos en particiones lógicas, cada archivo de dispositivo de bloque puede representar tanto un disco entero como una partición de éste. Se usa el número mayor para representar a que disco "físico" pertenece la partición y el número menor para identificar cada una de las particiones. Ejemplo:

  /dev/hda (número mayor 3, número menor 0)

  /dev/hda1 (número mayor 3, número menor1)

  /dev/hda2 (número mayor 3, número menor 2)

- **Part:** si el disco está dividido en particiones sus distribuciones se mantienen en un array de **hd_struct** cuya dirección es almacenada en el campo part.

  Cuando el kernel detecta un nuevo disco en el sistema éste invoca la función **alloc_disk()** que crea e inicializa un nuevo objeto gendisk, y si el nuevo disco contiene particiones también se crea el array de descriptores hd_struct adecuados.

- **fops:** este campo apunta a una tabla block_device_operations que almacena algunos métodos para realizar las operaciones cruciales sobre el dispositivo de bloque.

Generic Block Layer.

- **flags** : Almacena información sobre el disco. El flag más importante es **GENHD_FL_UP**, si está activo el disco está inicializado y funcionando. Otro flag relevante es **GENHD_FL_REMOVABLE**, el cual está activo si el disco es un soporte extraíble.

The I/O Scheduler

Cuando un componente del kernel ejecuta alguna operación de entrada/salida sobre algún dato de disco se crea una petición de dispositivo de bloque, que describe el sector solicitado y el tipo de operación (read o write) El objetivo del I/O scheduler es la planificación de peticiones de E/S pendientes para minimizar el tiempo gastado en mover las cabezas del disco. Se consigue este objetivo realizando dos operaciones principales, las operaciones **sorting** (ordenamiento) y **merging** (mezclado) que pertenecen al algoritmo de planificación por defecto (**Elevator I/O Scheduler**). El planificador de E/S mantiene una cola de peticiones de E/S pendientes (request queue).

- Sorting: las peticiones se ordenan por número de bloque en el disco. Cuando una nueva petición de E/S llega, es insertada de manera ordenada en la lista.

- **Mergin:** ocurre cuando una petición de E/S emitida es adyacente a una petición ya pendiente o es la misma, las dos peticiones pueden ser mezcladas y formar una sola petición.

The I/O Scheduler

Descriptor de la cola de peticiones:

En esencia una cola de peticiones es una lista doblemente enlazada cuyos elementos son descriptores de peticiones.

Cada dispositivo de bloque tiene su propia cola de peticiones, dicha cola se representa por la estructura **request_queue**. Esta estructura tiene multitud de campos de los que destacamos:

Struct list_ahead queue_head; almacena el primer elemento de la cola de peticiones.

Struct backing_dev_info backing_dev_info ; almacena información sobre el estado del tráfico entre el sistema y el dispositivo de bloque.

The I/O Scheduler

Descriptor de peticiones:

Cada petición para un dispositivo de bloque que esté pendiente de ejecutarse se representa con un descriptor de petición, el cual se almacena en una **request data structure**. Cada petición a su vez consiste en una o varias estructuras Bio.

Algunos campos de la estructura Request descriptors:

Tipo	Campo	Descripción
Struct list_ahead	Queuelist	Puntero para enlazar cada petición al anterior y siguiente elemento de la lista
Unsigned long	Flags	Flags para la petición
Sector_t	Sector	Numero del próximo sector que será transferido
Unsigned long	Nr_sectors	Número de sectores que quedan por ser transferidos
Struct nio*	Bio	Primera bio de la petición que no ha sido completamente transferida
Struct bio *	Biotail	Última bio en la lista de petición

The I/O Scheduler

El flag más importante es **REQ_RW** el cual detemina la dirección de la transferencia de datos READ(0) o WRITE (1)

OTROS ALGORITMOS DE PLANIFICACIÓN

Deadline: existen 3 colas; *"sorted queue"* (cola ordenada) , *"Write FIFO queue"* , "*Read FIFO queue*" + tiempo de expiración.

Anticipación: igual que el deadline + anticipación de lectura

CFQ(Complete Fairness Queueing):una cola por proceso (ordenada) + round-robin

Noop: Solo hace mezcla. Cuando una petición llega, es recogida en otra si existen adyacentes pero mantiene la cola en orden temporal de llegada, FIFO. Está destinado a usarse en dispositivos de bloques que son verdaderamente de acceso aleatorio,

Block Device Driver

El driver de dispositivo de bloques obtiene las peticiones desde el planificador de entrada salida y ejecuta las acciones que sean necesarias para procesarla. Un driver de dispositivos de bloque puede manejar varios dispositivos.

Cada dispositivo de bloque es representado mediante un **descriptor de dispositivo de bloque:**

Tipo	Campo	Descripción
Dev_t	Bd_dev	Número mayor y menor del dispositivo
Struct inode*	Bd_inode	Puntero al inode de el archivo asociado con el dispositivo de bloque en el sistema de ficheros
Struct semaphore	Bd_sem	Semáforo que protege la apertura y cierre de el dispositivo de bloque
Void*	Bd_holder	Actual titular de el descriptor de el dispositivo de bloque

Block Device Driver

Tipo	Campo	Descripción
Struct block_device *	Bd_contains	Si el dispositivo es un partición apunta al descriptor de dispositivo del disco entero, en otro caso, apunta a sí mismo
Struct hd_struct*	Bd_part	Puntero al descriptor de la partición (NULL si no es una partición)
Unsigned	Bd_part_count	Cuenta las veces que la partición incluida en este dispositivo ha sido abierta

Block Device Driver

Tipo	Campo	Descripción
Struct list_head*	Bd_list	Puntero para la lista de descriptores de dispositivo de bloques
Unsigned long	Bd_private	Puntero a datos privados de el dispositivo de bloque.
Int	Bd_openers	Cuenta cuantas veces el dispositivo de bloques ha sido abierto.
Struct gendisk *	Bd_disk	Puntero a la estructura gendisk de el disco relacionado por este dispositivo de bloque.

Block Device Driver

- Todos los descriptores de dispositivos de bloque son insertados en una lista global, cuya cabeza es representada por la variable global all_bdevs, los punteros para enlazar los descriptores se encuentran en el campo **bd_list** del descriptor de dispositivo.

- El campo *bd_holder* almacena una dirección lineal que representa al poseedor del dispositivo de bloque. Éste no es el driver de dispositivo que sirve las transferencias I/O de el dispositivo , sino que es un componente del kernel que hace uso del dispositivo y tiene privilegios exclusivos y especiales (campo bd_private)

Block Device Driver

Registro e inicialización del driver de dispositivo:

1.- Definir un descriptor de driver cliente: el driver necesita un descriptor *foo* de tipo *foo_dev_t* para mantener los datos necesarios para manejar el dispositivo hardware. Por cada dispositivo, el descriptor almacena información como puertos I/O usados, etc..

```
struct foo_dev_t{
        […]
        spinlock_t lock; //cerrojo para proteger los campos de el descriptor f
                         // foo
        struct gendisk *gd: //puntero al descriptor gendisk que representa el
                            //disco manejado por el driver
        […]
}foo
```

Block Device Driver

También se debe hacer la reserva del número mayor invocando a la función *register_blkdev()*

```
Err = register_blkdev (FOO_MAJOR,"foo");
If (err) goto error_major_is_busy;
```

Block Device Driver

2.- Inicializar el descriptor cliente.

Todos los campos del descriptor foo deben inicializarse antes de hacer uso del driver: Para inicializar los campos relacionados con el subsistema I/O se ejecutan las siguientes instrucciones:

```
Spin_lock_init(&foo.lock);

Foo.gd = aloc_disk(16)      // se asigna el array que almacena los
                            //descriptores de particiones del disco
                            //(hd_struct)

If (!foo.gd) goto error_no_gendisk;
```

Block Device Driver

3.- Inicializar el descriptor gendisk

Foo.gd -> private_data = &foo; //se almacena la dirección del descriptor foo (driver) en la estructura gendisk. ´Así las funciones de bajo nivel pueden encontrar rápidamente el descriptor del driver.

Foo.gd -> major = FOO_MAJOR;

Foo.gd ->first_minor = 16;

Set_capacity(foo.gd,foo_disk_capacity_in_sectors);

Strcpy(foo.gd->disk_name,"foo");

Foo.gd->fops = &foo_ops;

Block Device Driver

4.- Inicializar la tabla de métodos del dispositivo de bloque

El campo fops del descriptor gendisk es inicializado con las direcciones de los métodos almacenados en una tabla cliente del dispositivo de bloque. Con frecuencia, la tabla foo_ops del driver del dispositivo incluye funciones específicas para ese driver.

5.- Reservar e inicializar una cola de peticiones

//se genera un descriptor de cola y se inicializa la mayoría de campos con valores por defecto, también se define la rutina estratégica a utilizar que influye en la manera en que se ejecutan las peticiones desde la cola de peticiones.

```
foo.gd->rq = blk_init_queue(foo_strategy, &foo.lock);
if (!foo.gd->rq) goto error_no_request_queue;
blk_queue_hardsect_size(foo.gd->rd, foo_hard_sector_size);
blk_queue_max_sectors(foo.gd->rd, foo_max_sectors);
blk_queue_max_hw_segments(foo.gd->rd, foo_max_hw_segments);
blk_queue_max_phys_segments(foo.gd->rd, foo_max_phys_segments);
```

Block Device Driver

6.- Activar el manejador de interrupciones

Request_irq(foo_irq.foo_interrupt,

SA_INTERRUPT|SA_SHIRQ, “foo”, NULL);

Foo_interrupt() es la función manejadora para el dispositivo.

7.- Registrar el disco

El último paso consiste en registrar y activar el disco

Add_disk(foo.gd)

Block Device Driver

Se ejecutan las siguientes operaciones internas:

1.- ACTIVA el flag **GENHD_FL_UP** de gd->flags para indicar que el disco está inicializado y funcionando.

2.- Invoca la función kobj_map() para establecer el enlace entre el driver del dispositivo y el número mayor del dispositivo.

3.- Registra los objetos incluidos en el descriptor del gendisk en el driver de dispositivo.

4.- Escanea la tabla de particiones del disco, si la hubiera, y por cada partición inicializa el correspondiente descriptor hd_struct en el array foo.gd->part

5.- Se registran las colas de peticiones.

Una vez add_disk() retorna, el driver de dispositivo está activo y funcionando.

Realizando una petición

A continuación describiremos los pasos ejecutados por el kernel cuando se presenta una operación I/O a la capa de bloque genérica.

1.- Se ejecuta la función **bio_elloc()** para emplazar un nuevo descriptor de bio, el kenel inicializa el descriptor.

2.-Una vez el descriptor de bio ha sido inicializado el kernel ejecuta la función **generic_make_request(),** la función en esencia ejecuta los siguientes pasos:

- Comprueba que bio-> bi_sector no exceda de el número de sectores de el dispositivo, si fuera así la función activa el flag BIO_EOF de bio->bi_flags, imprime un mensaje de error, e invoca a la función bio_endoi() que actualiza los campos bi_size y bi_sector del descriptor bio, y por último invoca al método de la bio bi_end_io

-Obtiene la cola de peticiones **q** asociada con el dispositivo de bloque, que se puede encontrar en el campo bd_disk del descriptor de dispositivo, que a su vez es apuntado por bio-> bi_bdev

-Invoca block_wait_queue_running() para comprobar si el planificador I/O que se está usando está siendo reemplazado.

Realizando una petición

-Se ejecuta blk_partition_remap() para comprobar si el dispositivo de bloque se refiere a una partición

bio->bi_bdev no es igual a

bio->bi_bdev->bd_contains.

- Se invoca el método p->make_request_fn para insertar la petición bio en la cola de peticiones **q**

- Se retorna

Referencias

- http://lxr.linux.no/
- http://lwn.net/Kernel/LDD3/
- **O'Reilly-Understanding_The_Linux Kernel**
- **Lista de dispositivos de Linux (con sus pares major/minor):**
  - http://www.lanana.org/docs/device-list/devices-2.6