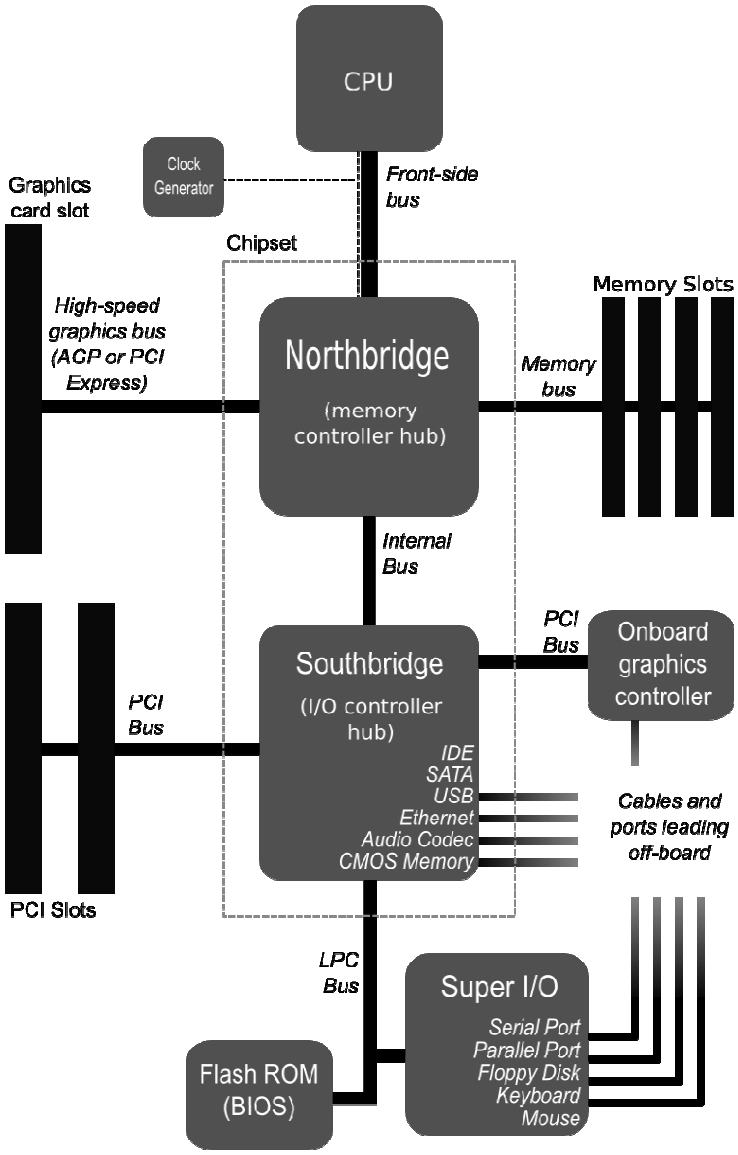
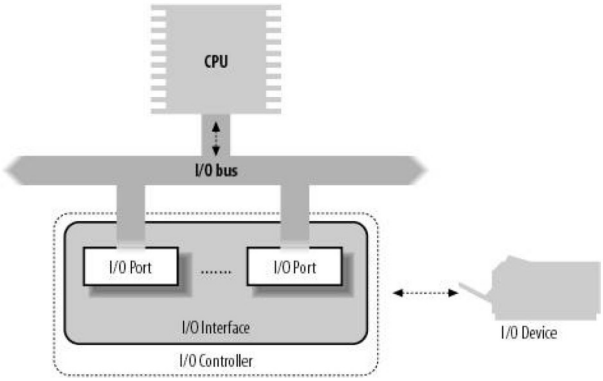
LECCIÓN I/O: ENTRADA SALIDA

I/O.1 Arquitectura de Entrada/Salida

Uno de los pilares básicos de un ordenador es el trasiego de información entre sus distintos dispositivos, como la CPU, la memoria RAM y los perfiéricos de E/S. Los circuitos que interconectan estos dispositivos y por los que fluyen los datos se llaman buses.
Existen distintos tipos de buses, tales como el PCI, EISA, ISA, USB o AGP, que interconectan los distintos periféricos del ordenador. En todo ordenador existen, además, otros buses, como son el *frontside bus* FSB y el *backside bus BSB*, que forman el nivel más bajo de la jerarquía de buses, y el *system bus*, que interconecta los distintos periféricos y está en el segundo nivel de la jerarquía. El BSB puede estar incluído dentro del chip de CPU y es usado para interconectar la CPU con la memoria cache. El FSB es usado para conectar el procesador con el controlador de memoria RAM. El *system bus*, que suele ser el PCI, tiene un propósito general de interconexión de distintos dispositivos.

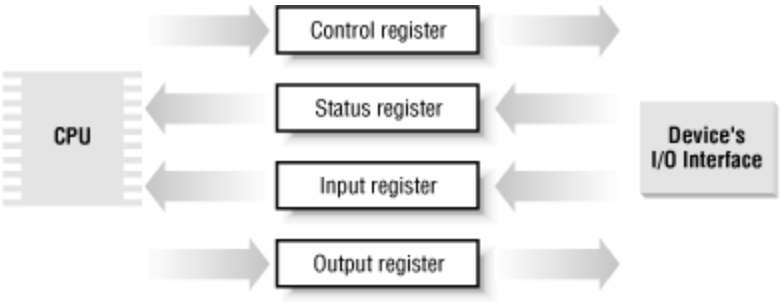


Los distintos buses son controlados a través de dos chips llamados *puente norte (northbridge)* y *puente sur (southbridge)*. Se solía usar como bus de interconexión un bus PCI, pero esto a la larga produjo un cuello de botella, por lo que los fabricantes optaron por implementar su propia interfaz de comunicaciones para sus chipsets. El puente norte maneja las comunicaciones entre la CPU, la RAM, los buses AGP y PCI-Express, mientras que el puente sur maneja las comunicaciones con periféricos más lentos. A este chip por tanto se conectan los periféricos de E/S. Cada dispositivo se conecta a uno y sólo a un bus, el cuál comprende de una serie de puertos de E/S, una interfaz y un controlador, como se puede ver en la siguiente figura:



I/O.1.1 Puertos de entrada salida

Cada dispositivo conectado a través del bus de E/S cuenta con una serie de direcciones para poder comunicarse. Estas direcciones son llamadas *puertos de E/S* y en el espacio de direcciones de una arquitectura PC contamos con 65.536 puertos de 8 bits, que pueden ser accedidos individualmente, en parejas (puertos consecutivos y en direcciones pares) como palabras de 16 bits o de 4 en 4 (consecutivos y en direcciones múltiplos de 4) como palabras de 32 bits.
La CPU cuenta con dos maneras de comunicarse con los dispositivos a través de los puertos: Una es tratando directamente con los puertos a través de instrucciones de ensamblador **in, ins, out** y **outs**. La otra es mapeando los puertos en memoria y usando instrucciones que puedan acceder a memoria directamente. Esta segunda manera es más rápida y es la que se suele usar.



Con el objetivo de facilitar la programación para dispositivos, los puertos de cada dispositivo están estructurados como se muestra en la figura. Las órdenes y los datos son enviados a través del *control register* y del *output register*, respectivamente. Asímismo, el estado del dispositivo y los datos de entrada son leídos del *status register* y del *input register.*

Como comentábamos, existen 4 instrucciones **in, ins, out** y **outs** para acceder a los puertos. Las siguientes macros del kernel simplifican dichos accesos. Están declaradas en **include/asm-*/io.h**.
Instrucciones de lectura:
• **inb(), inw(), inl():** Sirven para leer 1, 2 ó 4 bytes consecutivos de un puerto de E/S
• **inb_p(), inw_p (), inl_p ():** Igual que las anteriores, pero con el añadido de que tras la lectura realizan una pausa
• **insb(),insw(),insl():** Se usan para leer secuencias de 1,2 o 4 bytes contiguos, teniendo como parámetro la duración de la secuencia.

Instrucciones de escritura:
• **outb(), outw(), outl():** Sirven para escribir 1, 2 ó 4 bytes consecutivos de un puerto de E/S
• **outb_p(), outw_p(), outl_p():** Igual que las anteriores, pero con el añadido de que tras la escritura se hace una pausa.
• **outsb(),outsw(),outsb():** Se usan para escritura de secuencias de 1,2 o 4 bytes contiguos, teniendo como parámetro la duración de la secuencia.

I/O.1.2 Recursos

Aunque acceder a un puerto de E/S es bastante sencillo, saber qué puertos tiene asignado un determinado dispositivo puede no ser tan fácil. Con frecuencia, un driver debe escribir a ciegas en algunos puertos para sondear el dispositivo, lo cuál puede incurrir en un fallo del sistema si el puerto ya está siendo usado por otro dispositivo. Para evitar este tipo de situaciones, el kernel mantiene una lista de los puertos de E/S asignados, en términos de **recursos**.
Un **recurso** representa, en general, una parte de alguna entidad que puede ser asignada exclusivamente a un dispositivo. En este caso, es un rango de direcciones de puertos, aunque también pueden ser un rango de direcciones de memoria o números de interrupción.
Todos los recursos de un mismo tipo son almacenados en una estructura arbórea. En el caso de los puertos de E/S, la raíz se llama **ioport_resource.** En estos árboles, cada nodo representa un subrango del rango asociado al padre, teniendo la raíz todo el espacio de direcciones asociado. Este árbol es accesible a través de */proc/ioports.*

Funciones del manejo de recursos:

Los manejadores deben usar las siguientes funciones pasando como parámetros la raíz del árbol de recursos y la dirección del nuevo recurso.
• **request_resource():** Asigna un recurso a un dispositivo de E/S. Se puede usar para comprobar el estado
• **allocate_resource():** Encuentra un rango de direcciones en el árbol disponible de un determinado tamaño y alineamiento.
• **release_resource():** Libera un recurso o un subrango del mismo.

El núcleo aporta unas instrucciones que son atajos de las anteriores:
• **request_region():** Tiene como parámetro un rango de puertos, que es el que asigna al driver llamador.
• **release_region():** Libera un subrango de puertos pasados por parámetro.

La estructura de datos que utilizan los recursos se encuentra en /include/linux/ioport.h:

```
17 struct resource {
18      resource_size_t start; //Dirección de inicio del recurso
19      resource_size_t end; //Dirección final
20      const char *name; //Nombre del dueño del recurso
21      unsigned long flags; //Distintas opciones
22      struct resource *parent, *sibling, *child; //Padre, hermano (siguiente nodo en la lista) y primer
                                                   //hijo del nodo.
23 };
```

La función **request_resource** reserva un recurso para un driver:

request_resource recibe dos argumentos:
root: descriptor del recurso raíz
new: descriptor del recurso deseado

Devuelve 0 si se pudo solicitar el recurso o -EBUSY si ya estaba asignado.

La implementación es la siguente:

```
 203int request_resource(struct resource *root, struct resource *new)
 204{
 205        struct resource *conflict;
 206
 207        write_lock(&resource_lock);
 208        conflict = __request_resource(root, new);
 209        write_unlock(&resource_lock);
 210        return conflict ? -EBUSY : 0;
 211}
```

request_resource llama a la función __request_resource:

```
 149static struct resource * __request_resource(struct resource *root, struct resource *new)
 150{
 151        resource_size_t start = new->start;
 152        resource_size_t end = new->end;
 153        struct resource *tmp, **p;
 154
 155        if (end < start)
 156                return root;
 157        if (start < root->start)
 158                return root;
 159        if (end > root->end)
 160                return root;
 161        p = &root->child;
 162        for (;;) {
 163                tmp = *p;
 164                if (!tmp || tmp->start > end) {// Si no hay recurso o tiene una dirección posterior
 165                        new->sibling = tmp; // añade el recurso a la lista de hermanos
 166                        *p = new;
 167                        new->parent = root;
 168                        return NULL;
 169                }
 170                p = &tmp->sibling; ; // si no ha encontrado un 'hueco' aún, continua por el hermano
 171                if (tmp->end < start)
 172                        continue;
 173                return tmp; // si no hay hueco para el recurso devuelve el recurso actual (que causa el
problema)
 174        }
 175}
```

La función **release_resource** se encarga de liberar un recurso reservado. Recibe como argumento el recurso a librerar. Devuelve 0 si lo pudo liberar o -EINVAL si no pudo encontrar el recurso.

```
 219  int release_resource(struct resource *old)
 220 {
 221        int retval;
 222
 223        write_lock(&resource_lock);
 224        retval = __release_resource(old);
 225        write_unlock(&resource_lock);
 226        return retval;
 227 }
```

La función **release_resource** llama a __release_resource que es donde ser realiza la búsqueda y liberación del recurso:

```
177 static int __release_resource(struct resource *old)
178 {
179         struct resource *tmp, **p;
180
181         p = &old->parent->child; // p contiene la dirección del 1º hijo

182         for (;;) {
183                 tmp = *p;

184                 if (!tmp) // Si ya no quedan recursos, sale del bucle
185                         break;
186                 if (tmp == old) {

187                         *p = tmp->sibling;  // Si lo encontramos lo libera y devuelve 0
188                         old->parent = NULL;
189                         return 0;
190                 }
191                 p = &tmp->sibling; // Pasamos al siguiente recurso de la lista
192         }
193         return -EINVAL; // Si no se encuentra devuelve el código de error -EINVAL.
194 }
```

I/O.1.3 Interfases E/S

Una interfaz E/S es un circuito hardware insertado entre un grupo de puertos de E/S y los correspondientes controladores de dispositivo. No sólo actúa como si fuera un interprete, traduciendo los valores de los puertos de E/S a datos y comandos para los dispositivos, sino que además detecta cambios en el estado de los dispositivos y consecuentemente actualiza el puerto de E/S que juega el papel de Registro de Estado.
Existen dos tipos de interfaces:

- Las genéricas que se utilizan para conectar dispositivos de distinto tipo externo, como los puertos USB.
- Las específicas son para un solo tipo de dispositivo. Un ejemplo es el conector de teclado, o el de la disquetera.

I/O.1.4 Memoria Compartida

Un dispositivo complejo necesita un controlador que lo maneje. Esencialmente, el controlador de dispositivos juega dos papeles importantes:

- Interpretar comandos de alto nivel enviados por la interfaz E/S, haciendo que el dispositivo realice tareas específicas, enviándole la apropiada secuencia de las señales eléctricas.
- Realizar el camino inverso: interpretar las señales eléctricas a lenguaje de alto nivel y modificar el valor del Registro de Estado.

Muchos dispositivos incluyen memoria propia llamada *memoria compartida de E/S*. Esta memoria también puede estar mapeada, y dependiendo a que bus se conecta se le pueden asignar unas direcciones físicas u otras.

Mapeo de direcciones de la memoria compartida:
Antiguamente existían tres rangos de direcciones físicas según el dispositivo y el tipo de bus:

- •Bus ISA: 0xa0000 a 0xfffff (640KB – 1MB)
- •Bus local VESA: 0xe0000 a 0xffffff (14MB – 16MB)
- •Bus PCI: Direcciones de 32 bits dan lugar a un límite de 4 GB.

Intel introdujo el bus AGP (Puerto Gráfico Acelerado) para tarjetas gráficas de alta calidad, a las cuales se les quedaba "corto" el bus PCI, con las siguientes características:

-El hardware usa la Graphics Adress Remapping Table (GART) para poder acceder a determinadas posiciones de la RAM directamente.
-El núcleo maneja de igual manera la memoria compartida del bus AGP que cualquier otra.
-Se consigue acelerar las transferencias tarjeta gráfica - memoria y memoria tarjeta gráfica.

Acceso a la memoria compartida:

Las direcciones de memoria que tiene asignadas la memoria compartida son las superiores al PAGE_OFFSET. Los dispositivos deben traducir las direcciones físicas de sus puertos E/S al espacio de direcciones del kernel.

En algunos casos, puede ocurrir que la dirección proporcionada se salga de rango, siendo necesario modificar la tabla de páginas del núcleo. Para ello se tienen las siguientes funciones:

- **ioremap()** e **ioremap_nocache()**: Modifica la tabla de páginas, mapeando un nuevo vm_struct en memoria virtual con el espacio requerido para el área de memoria compartida de E/S.
- **iounmap()**: Deshace los cambios del ioremap, quitando el mapeado y el vm_struct.

La estructura vm_struct está definida en /include/linux/vmalloc.h:

```
25struct vm_struct {
26      /* Mantener next, addr y size juntos, para acelerar las consultas. */
27      struct vm_struct       *next; //Siguiente estructura vm_struct
28      void                   *addr; //Dirección virtual
29      unsigned long          size; //Tamaño del área (más 4096, intervalo de seguridad entre áreas)
30      unsigned long          flags; //Distintas opciones
31      struct page            **pages; //Matriz de descriptores de páginas
32      unsigned int           nr_pages; //Número de descriptores
33      unsigned long          phys_addr; //Distinto de 0 si el área mapea la memoria compartida de
                                          //un dispositivo
34};
```

La función ioremap es la siguiente y se encuentra en /include/asm-i386/io.h:

```
107 static inline void __iomem * ioremap(unsigned long offset, unsigned long size)
108 {
109       return __ioremap(offset, size, 0); //Sólo llama a la función __ioremap
110 }
```

La implementación de la función __ioremap está en /arch/i386/mm/ioremap.c y es la siguiente:

```
/*
 29 * Remap an arbitrary physical address space into the kernel virtual
 30 * address space. Needed when the kernel wants to access high addresses
 31 * directly.
 32 *
 33 * NOTE! We need to allow non-page-aligned mappings too: we will obviously
 34 * have to convert them into an offset in a page-aligned mapping, but the
 35 * caller shouldn't need to know that small detail.
 36 */
 37void __iomem * __ioremap(unsigned long phys_addr, unsigned long size, unsigned long flags)
 38{
 39      void __iomem * addr;
 40      struct vm_struct * area;
 41      unsigned long offset, last_addr;
 42      pgprot_t prot;
 43
 44      /* No permite wraparound o tamaño 0*/
 45      last_addr = phys_addr + size - 1;
 46      if (!size || last_addr < phys_addr)
 47            return NULL;
 48
```

```
49      /*
50       * No remapear el bajo área de PCI/ISA, ya que siempre está mapeado…
51       */
52      if (phys_addr >= ISA_START_ADDRESS && last_addr < ISA_END_ADDRESS)
53              return (void __iomem *) phys_to_virt(phys_addr);
54
55      /*
56       * Impedir que alguien pueda remapear la RAM que estamos usando.
57       */
58      if (phys_addr <= virt_to_phys(high_memory - 1)) {
59              char *t_addr, *t_end;
60              struct page *page;
61
62              t_addr = __va(phys_addr);
63              t_end = t_addr + (size - 1);
64
65              for(page = virt_to_page(t_addr); page <= virt_to_page(t_end); page++)
66                      if(!PageReserved(page))
67                              return NULL;
68      }
69
70      prot = __pgprot(_PAGE_PRESENT | _PAGE_RW | _PAGE_DIRTY
71                      | _PAGE_ACCESSED | flags);
72
73      /*
74       * Los mapeos deben estar alineados a la página
75       */
76      offset = phys_addr & ~PAGE_MASK;
77      phys_addr &= PAGE_MASK;
78      size = PAGE_ALIGN(last_addr+1) - phys_addr;
79
80      /*
81       * Todo está correcto, se puede realizar la operación
82       */
83      area = get_vm_area(size, VM_IOREMAP | (flags << 20));
84      if (!area)
85              return NULL;
86      area->phys_addr = phys_addr;
87      addr = (void __iomem *) area->addr;
88      if (ioremap_page_range((unsigned long) addr,
89                      (unsigned long) addr + size, phys_addr, prot)) {
90              vunmap((void __force *) addr);
91              return NULL;
92      }
93      return (void __iomem *) (offset + (char __iomem *)addr);
94}
```

Puesto que en otras arquitecturas, el acceso a memoria compartida no es tan simple como en un PC, el núcleo ofrece unas macros para acceder a la memoria compartida de E/S:

• **readb, readw, readl**: Lee 1, 2 ó 4 bytes respectivamente de una posición de memoria compartida E/S

• **writeb, writew, writel**: Escribe 1, 2 ó 4 bytes respectivamente de una posición de memoria compartida E/S

• **memcpy_fromio, memcpy_toio**: Se encargan de copiar bloques de memoria de compartida a memoria dinámica y viceversa.

• **memset_io**: Fija un valor para una determinada zona de memoria compartida.

Se consigue una independencia entre la máquina y el software manejador.

Aquí mostramos la implementación de las funciones de lectura en /include/asm-i386/io.h:

```
146 static inline unsigned char readb(const volatile void __iomem *addr)
147 {
148         return *(volatile unsigned char __force *) addr;
149 }
150 static inline unsigned short readw(const volatile void __iomem *addr)
151 {
152         return *(volatile unsigned short __force *) addr;
153 }
154 static inline unsigned int readl(const volatile void __iomem *addr)
155 {
156         return *(volatile unsigned int __force *) addr;
157 }
```

I/O.1.5 DMA (Direct memory access)

Originalmente, el único componente que actuaba de "maestro" del bus de datos y direcciones para poder extraer y guardar información de la RAM era la CPU. En sistemas más modernos, los periféricos pueden actuar como maestros. Para ello, es necesario contar con una circuitería adicional, llamada **DMA**. La CPU indica al dispositivo que puede comenzar con la transacción y sigue con su trabajo. Mientras, el dispositivo se comunica de manera independiente con la memoria.
Los elementos fundamentales que aporta son:

- Un procesador llamado Controlador de acceso directo a la memoria (DMAC) que controla la transacción de datos entre la RAM y el dispositivo de E/S.
- Un circuito hardware llamado Arbitro de memoria que resuelve los conflictos que ocurren cuando tanto la CPU como el DMAC necesitan acceder a la misma zona de memoria en el mismo momento.

Poner el DMA en marcha

Existen 4 tipos de direcciones de memoria:

- Físicas: Utilizadas por la CPU para dirigir el bus de datos
- Lógicas y virtuales: Usadas internamente por la CPU
- Direcciones del bus: Utilizadas por los dispositivos para manejar el bus de datos

El núcleo ofrece dos macros de traducción de direcciones del bus a direcciones virtuales y viceversa en /include/asm-i386/io.h:

```
136 #define virt_to_bus virt_to_phys
137 #define bus_to_virt phys_to_virt

 93static inline void * phys_to_virt(unsigned long address)
 94{
 95        return __va(address);
 96}
 75static inline unsigned long virt_to_phys(volatile void * address)
 76{
 77        return __pa(address);
 78}
182#define __pa(x)             ((unsigned long)(x)-PAGE_OFFSET)
186#define __va(x)             ((void *)((unsigned long)(x)+PAGE_OFFSET))
```

Cuando un manejador de dispositivo comienza una operación DMA para algún dispositivo de E/S debe especificar el buffer de memoria correspondiente mediante direcciones del bus.
Pasos que sigue:

1. El driver del dispositivo escribe la dirección del bus en el buffer del DMA y el tamaño del dato comienzo de la transmisión en un puerto del dispositivo.
2. El driver suspende el proceso actual

3.Cuando la transferencia DMA termina el dispositivo lanza una interrupción que despierta a su driver.
4.Liberación de recursos

I/O.2 Ficheros de Dispositivos

Todo sistema *NIX se basa en la noción de fichero, que es simplemente un contenedor de información estructurado como una secuencia de bytes. Todo dispositivo tiene un fichero mediante el cual podemos hacer operaciones de entrada/salida. 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.

Estos ficheros de dispositivos se dividen en dos tipos de ficheros, atendiendo al hardware que representan:

Ficheros de bloques: Corresponde a dispositivos estructurados en bloque, como los discos a los que se accede proporcionando un número de bloque a leer o a escribir.
Ficheros de caractereres: Corresponden a dispositivos no estructurados como los puertos serie y paralelo, sobre los que se puede leer y escribir los datos byte a byte, generalmente de manera secuencial.

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.

Existen varias soluciones de cara al 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.

I/O.2.1 Manejo de ficheros de dispositivos en VFS

El VFS se encarga de esconder las diferencias entre ficheros normales y de dispositivo, creando una relación entre las llamadas al sistema y las funciones de acceso del dispositivo.

- Cada llamada al sistema en el fichero del dispositivo se traduce por la llamada a la función del dispositivo relacionada, en vez de llamar a la función que tiene el sistema por defecto.

- 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. Se inicializa el campo i_fop según el tipo de dispositivo (def_chr_fops o def_blk_fops)

Implementación de la función **ext2_read_inode** en el fichero /fs/ext2/inode.c:

```
1025 void ext2_read_inode (struct inode * inode){
1027        struct ext2_inode_info *ei = EXT2_I(inode);//Saca un inode de su estructura y lo pone en ei
que  tiene la información del inode en memoria
1028        ino_t ino = inode->i_ino;
```

```
1029        struct buffer_head * bh;
1030        struct ext2_inode * raw_inode = ext2_get_inode(inode->i_sb, ino, &bh);
//inode  del disco
1031        int n;
1032
1033 #ifdef CONFIG_EXT2_FS_POSIX_ACL
1034        ei->i_acl = EXT2_ACL_NOT_CACHED; //ACL no ha sido almacenado en  cache
1035        ei->i_default_acl = EXT2_ACL_NOT_CACHED;
1036 #endif
1037        if (IS_ERR(raw_inode))  //Si error de E/S, vamos a la etiqueta bad_inode
1038                goto bad_inode;
1039          //Miramos los campos del inode
1040        inode->i_mode = le16_to_cpu(raw_inode->i_mode);
1041        inode->i_uid = (uid_t)le16_to_cpu(raw_inode->i_uid_low);
1042        inode->i_gid = (gid_t)le16_to_cpu(raw_inode->i_gid_low);
1043        if (!(test_opt (inode->i_sb, NO_UID32))) {
1044                inode->i_uid |= le16_to_cpu(raw_inode->i_uid_high) << 16;
1045                inode->i_gid |= le16_to_cpu(raw_inode->i_gid_high) << 16;
1046        }
1047        inode->i_nlink = le16_to_cpu(raw_inode->i_links_count);
1048        inode->i_size = le32_to_cpu(raw_inode->i_size);
1049        inode->i_atime.tv_sec = le32_to_cpu(raw_inode->i_atime);
1050        inode->i_ctime.tv_sec = le32_to_cpu(raw_inode->i_ctime);
1051        inode->i_mtime.tv_sec = le32_to_cpu(raw_inode->i_mtime);
1052        inode->i_atime.tv_nsec = inode->i_mtime.tv_nsec = inode->i_ctime.tv_nsec = 0;
1053        ei->i_dtime = le32_to_cpu(raw_inode->i_dtime);
1054        //Ahora tenemos suficientes campos para ver si el inode estaba activo o no.
1055        //Esto es necesario porque el nfsd tiene que acceder a inodes inactivos
1059        if (inode->i_nlink == 0 && (inode->i_mode == 0 || ei->i_dtime)) {
1060             //El nodo es borrado y se va a la etiqueta bad_inode
1061              brelse (bh);
1062              goto bad_inode;
1063        }
1064        inode->i_blksize = PAGE_SIZE;   //Este es el tamaño optimo de la E/S,
 no el tamaño del bloque de sistema de ficheros

            //Inicializa los campos del inode ei con los valores del inode del disco
1065        inode->i_blocks = le32_to_cpu(raw_inode->i_blocks);
1066        ei->i_flags = le32_to_cpu(raw_inode->i_flags);
1067        ei->i_faddr = le32_to_cpu(raw_inode->i_faddr);
1068        ei->i_frag_no = raw_inode->i_frag;
1069        ei->i_frag_size = raw_inode->i_fsize;
1070        ei->i_file_acl = le32_to_cpu(raw_inode->i_file_acl);
1071        ei->i_dir_acl = 0;
1072        if (S_ISREG(inode->i_mode))
1073                inode->i_size |= ((__u64)le32_to_cpu(raw_inode->i_size_high)) << 32;
1074        else
1075                ei->i_dir_acl = le32_to_cpu(raw_inode->i_dir_acl);
1076        ei->i_dtime = 0;
1077        inode->i_generation = le32_to_cpu(raw_inode->i_generation);
1078        ei->i_state = 0;
1079        ei->i_next_alloc_block = 0;
1080        ei->i_next_alloc_goal = 0;
1081        ei->i_prealloc_count = 0;
1082        ei->i_block_group = (ino - 1) / EXT2_INODES_PER_GROUP(inode->i_sb);
1083        ei->i_dir_start_lookup = 0;
1084
1085        /*NOTA: La memoria del i_data del inode está en little-endian
1089        for (n = 0; n < EXT2_N_BLOCKS; n++)
1090                ei->i_data[n] = raw_inode->i_block[n];
```

```
1091
1092        if (S_ISREG(inode->i_mode)) {
1093                inode->i_op = &ext2_file_inode_operations;
1094                inode->i_fop = &ext2_file_operations;
1095                if (test_opt(inode->i_sb, NOBH))
1096                        inode->i_mapping->a_ops = &ext2_nobh_aops;
1097                else
1098                        inode->i_mapping->a_ops = &ext2_aops;
1099        } else if (S_ISDIR(inode->i_mode)) {
1100                inode->i_op = &ext2_dir_inode_operations;
1101                inode->i_fop = &ext2_dir_operations;
1102                if (test_opt(inode->i_sb, NOBH))
1103                        inode->i_mapping->a_ops = &ext2_nobh_aops;
1104                else
1105                        inode->i_mapping->a_ops = &ext2_aops;
1106        } else if (S_ISLNK(inode->i_mode)) {
1107                if (ext2_inode_is_fast_symlink(inode))
1108                        inode->i_op = &ext2_fast_symlink_inode_operations;
1109                else {
1110                        inode->i_op = &ext2_symlink_inode_operations;
1111                        if (test_opt(inode->i_sb, NOBH))
1112                                inode->i_mapping->a_ops = &ext2_nobh_aops;
1113                        else
1114                                inode->i_mapping->a_ops = &ext2_aops;
1115                }
1116        } else {
1117                inode->i_op = &ext2_special_inode_operations;
1118                if (raw_inode->i_block[0])
1119                        init_special_inode(inode, inode->i_mode,
1120                           old_decode_dev(le32_to_cpu(raw_inode->i_block[0])));
1121                else
1122                        init_special_inode(inode, inode->i_mode,
1123                           new_decode_dev(le32_to_cpu(raw_inode->i_block[1])));
1124        }
1125        brelse (bh);
1126        ext2_set_inode_flags(inode);
1127        return;
1128
1129 bad_inode: //Marca un inode como malo debido a un error de E/S
1130        make_bad_inode(inode);
1131        return;
1132 }
```

Implementación de init_special_inode en el fichero /fs/inode.c:

```
1360 void init_special_inode(struct inode *inode, umode_t mode, dev_t rdev)
1361 {
1362        inode->i_mode = mode;
1363        if (S_ISCHR(mode)) { //Si es de tipo carácter
1364                inode->i_fop = &def_chr_fops; // coloca en el campo i_fop del inode la //dirección de
                                                        las operaciones del character
1365                  inode->i_rdev = rdev; // inicializa el campo i_rdev del inode con el major y //minor
                                                numbers.
1366        } else if (S_ISBLK(mode)) {//Si es de tipo bloque
1367                inode->i_fop = &def_blk_fops; // coloca en el campo i_fop del inode la //dirección de
                                                        las operaciones del bloque
```

```
1368                inode->i_rdev = rdev; // inicializa el campo i_rdev del inode con el major y minor
                                          numbers.
1369        } else if (S_ISFIFO(mode))
1370                inode->i_fop = &def_fifo_fops;
1371        else if (S_ISSOCK(mode))
1372                inode->i_fop = &bad_sock_fops;
1373        else
1374                printk(KERN_DEBUG "init_special_inode: bogus i_mode (%o)\n",
1375                        mode);
1376 }
```

Implementación de dentry_open en /fs/open.c. Esta función coloca un nuevo fichero activando su campo f_op con la dirección del i_fop:

```
struct file *dentry_open(struct dentry *dentry, struct vfsmount *mnt, int flags)
774 {
775         struct file * f;
776         struct inode *inode;
777         int error;
778
779         error = -ENFILE;
780         f = get_empty_filp(); //Puntero a una estructura de fichero libre
781         if (!f)  //Si es null, es que no hay espacios libres
782                 goto cleanup_dentry;
             //Inicaliza los campos del fichero f
783         f->f_flags = flags;
784                 f->f_mode = ((flags+1) & O_ACCMODE) | FMODE_LSEEK | FMODE_PREAD |
FMODE_PWRITE;
785         inode = dentry->d_inode;
786         if (f->f_mode & FMODE_WRITE) {
787                 error = get_write_access(inode); //Obtiene el permiso de escritura para el //fichero
788                 if (error)
789                         goto cleanup_file;
790         }
791
792         f->f_mapping = inode->i_mapping;
793         f->f_dentry = dentry;
794         f->f_vfsmnt = mnt;
795         f->f_pos = 0;
796         f->f_op = fops_get(inode->i_fop); //Coloca la dirección del i_fop en f_op
797         file_move(f, &inode->i_sb->s_files); //Coloca el fichero nuevo
798
799         if (f->f_op && f->f_op->open) {
800                 error = f->f_op->open(inode,f);
801                 if (error)
802                         goto cleanup_all;
803         }
804         f->f_flags &= ~(O_CREAT | O_EXCL | O_NOCTTY | O_TRUNC);
805
806         file_ra_state_init(&f->f_ra, f->f_mapping->host->i_mapping);
807
808         /* NB: we're sure to have correct a_ops only after f_op->open */
809         if (f->f_flags & O_DIRECT) {
810                 if (!f->f_mapping->a_ops || !f->f_mapping->a_ops->direct_IO) {
811                         fput(f);
812                         f = ERR_PTR(-EINVAL);
813                 }
814         }
816         return f;   //Devuelve el fichero
817
```

```
818 cleanup_all:
819         fops_put(f->f_op);
820         if (f->f_mode & FMODE_WRITE)
821                 put_write_access(inode);
822         file_kill(f);
823         f->f_dentry = NULL;
824         f->f_vfsmnt = NULL;
825 cleanup_file:
826         put_filp(f);
827 cleanup_dentry:
828         dput(dentry);
829         mntput(mnt);
830         return ERR_PTR(error);
831}
```

Las operaciones **def_chr_fops** o **def_blk_fops** a las que nos referimos antes corresponden a las tablas de punteros a funciones que asignan a cada operacion sobre los ficheros la rutina que la atiende:

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

```
431 const struct file_operations def_chr_fops = {
432         .open = chrdev_open,
433 };
```

Los dispositivos de caracteres solo asocian la opción open en un principio. La implementación del driver se encargará de asignar el resto de operaciones.

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

```
const struct file_operations def_blk_fops = {
1330        .open           = blkdev_open,
1331        .release        = blkdev_close,
1332        .llseek         = block_llseek,
1333        .read           = do_sync_read,
1334        .write          = do_sync_write,
1335        .aio_read       = generic_file_aio_read,
1336        .aio_write      = generic_file_aio_write_nolock,
1337        .mmap           = generic_file_mmap,
1338        .fsync          = block_fsync,
1339        .unlocked_ioctl = block_ioctl,
1340 #ifdef CONFIG_COMPAT
1341        .compat_ioctl   = compat_blkdev_ioctl,
1342 #endif
1343        .sendfile       = generic_file_sendfile,
1344        .splice_read    = generic_file_splice_read,
1345        .splice_write   = generic_file_splice_write,
1346 };
```

I/O.3 Modelo de Drivers de Dispositivos

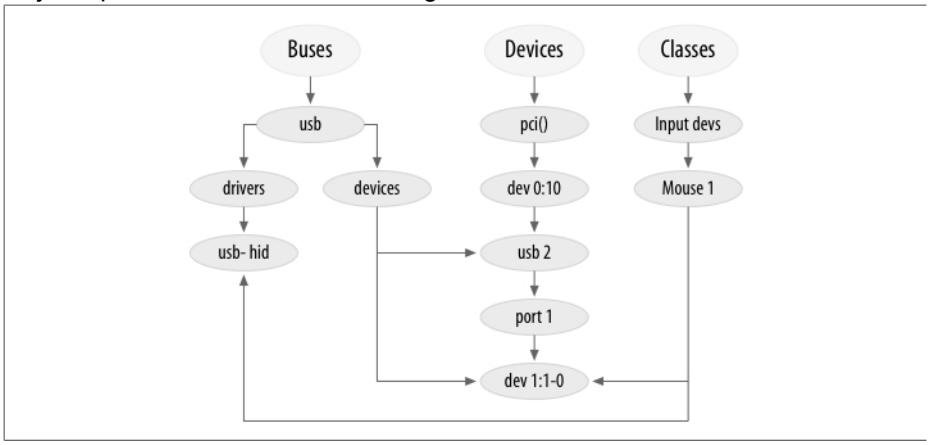
Para facilitar y organizar la construcción de drivers, Linux ofrece una serie de estructuras y funciones auxiliares que ofrecen una visión unificada de buses, dispositivos y drivers.

El Modelo de Drivers de Dispositivos tiene una interfaz unificada, accesible a través del sistema de ficheros virtual 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/**module**: 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.

La jerarquía de este modelo es la siguiente:



Siendo los **buses** el nivel más bajo, el intermedio los **dispositivos** y el nivel más alto son las **clases**.
El modelo de drivers define varias estructuras en include/linux/device.h:

I/O.3.1 Tipo de bus

Cada bus del kernel deberá declara un objeto de tipo bus_type, inicializando al menos el campo *name*. Al inicializar un driver de bus, éste llamará a **bus_register**, que inicializará el resto de campos de la estructura.

```
52struct bus_type {
53      const char              * name;
54      struct module           * owner;
55
56      struct kset             subsys; //Subsistema dentro de sysfs al que pertenece
57      struct kset             drivers; //Drivers conocidos para ese bus
58      struct kset             devices; //Dispositivos conectados al bus
59      struct klist            klist_devices;
60      struct klist            klist_drivers;
61
62      struct blocking_notifier_head bus_notifier;
63
64      struct bus_attribute    * bus_attrs;
65      struct device_attribute * dev_attrs;
66      struct driver_attribute * drv_attrs;
67      struct bus_attribute drivers_autoprobe_attr;
68      struct bus_attribute drivers_probe_attr;
69
        // Funcion que comprueba si el bus puede manejar un cierto dispositivo o driver de dispositivo.
70      int             (*match)(struct device * dev, struct device_driver * drv);
71      int             (*uevent)(struct device *dev, char **envp,
```

```
 72                          int num_envp, char *buffer, int buffer_size);
 73      int         (*probe)(struct device * dev);
 74      int         (*remove)(struct device * dev);
 75      void         (*shutdown)(struct device * dev);
 76
 77      int (*suspend)(struct device * dev, pm_message_t state);
 78      int (*suspend_late)(struct device * dev, pm_message_t state);
 79      int (*resume_early)(struct device * dev);
 80      int (*resume)(struct device * dev);
 81
 82      unsigned int drivers_autoprobe:1;
 83};
```

I/O.3.2 Driver de dispositivo

Todo dispositivo en Linux está representado con una estructura de este tipo.

```
124struct device_driver {
125      const char            * name;
126      struct bus_type        * bus; //Tipo de bus del dispositivo
127
128      struct kobject         kobj; //Objeto que representa el dispositivo en la jerarquía del modelo.
129      struct klist          klist_devices; //Dispositivos asociados a este driver.
130      struct klist_node      knode_bus;
131
132      struct module          * owner;
133      const char            * mod_name;     /* used for built-in modules */
134      struct module_kobject   * mkobj;
135
136      int    (*probe)       (struct device * dev);//Comprueba si un dispositivo existe o puede ser
                                                    //manejado por el driver
137      int    (*remove)      (struct device * dev); //Se llama al borrar el driver del sistema.
138      void    (*shutdown)    (struct device * dev); //Se llama al apagar el sistema.
139      int    (*suspend)     (struct device * dev, pm_message_t state);
140      int    (*resume)      (struct device * dev);
141};
```

I/O.3.3 Clase de dispositivo

Este es el nivel de la jerarquía más alto y es el que hace posible que los procesos trabajen con los dispositivos según lo que hacen y no cómo lo hacen o cómo están conectados. P.e. si tenemos un disco SCSI o ATA, nos da igual, se verá simplemente como un disco.

```
177struct class {
178        const char              * name;
179        struct module           * owner;
180
181        struct kset             subsys;
182        struct list_head        children;
183        struct list_head        devices;
184        struct list_head        interfaces;
185        struct kset             class_dirs;
186        struct semaphore        sem;    /* locks both the children and interfaces lists */
187
188        struct class_attribute          * class_attrs; //Atributos genéricos de la clase
189        struct class_device_attribute   * class_dev_attrs; //Atributos de cada driver asociado
190        struct device_attribute         * dev_attrs;
191
192        int     (*uevent)(struct class_device *dev, char **envp,
193                          int num_envp, char *buffer, int buffer_size);
194        int     (*dev_uevent)(struct device *dev, char **envp, int num_envp,
195                              char *buffer, int buffer_size);
196
197        void    (*release)(struct class_device *dev);
198        void    (*class_release)(struct class *class);
199        void    (*dev_release)(struct device *dev);
200
201        int     (*suspend)(struct device *, pm_message_t state);
202        int     (*resume)(struct device *);
203};
```

I/O.3.4 Dispositivo

```
349 struct device {
350        struct klist            klist_children;
351        struct klist_node       knode_parent;
352        struct klist_node       knode_driver;
353        struct klist_node       knode_bus;
354        struct device   * parent;
355
356        struct kobject kobj;
357        char    bus_id[BUS_ID_SIZE];    /* posición en el bus */
358        unsigned                is_registered:1;
359        struct device_attribute uevent_attr;
360        struct device_attribute *devt_attr;
361
362        struct semaphore        sem;    /* semaforo para sincronizar llamadas de su driver */
363
364
365
366        struct bus_type * bus;          /* tipo de bus */
367        struct device_driver *driver;   /* driver que tiene asignado */
368
369        void            *driver_data;   /* datos privados del driver*/
```

```
370         void            *platform_data; /* datos dependientes de la plataforma */
371
372         struct dev_pm_info      power;
373
374 #ifdef CONFIG_NUMA
375         int             numa_node;
376 #endif
377         u64             *dma_mask;
378         u64             coherent_dma_mask;
384         struct list_head        dma_pools;
386         struct dma_coherent_mem *dma_mem;
389         struct dev_archdata     archdata;
390
392         struct list_head        node;
393         struct class            *class;
394         dev_t                   devt;           /* identificador del dispositivo (mayor y menor) */
395         struct attribute_group  **groups;
396
397         void    (*release)(struct device * dev);
398 };
```

I/O.4 Drivers de Dispositivos

Los drivers de dispositivos son un conjunto de rutinas del kernel que hacen que un dispositivo responda a las llamadas del VFS genéricas (open, read, ioctl,…). Puesto que cada dispositivo tiene un controlador E/S diferente, la implementación de dicho conjunto se realiza en el propio driver. La mayoría de los dispositivos cuentan con uno.

I/O.4.1 Registro de un driver

Cada operación sobre un fichero de dispositivo debe traducirse a una llamada a una rutina de su driver. Para ello, el driver debe registrarse y ser asociado a un dispositivo. Esto se hace creando un **device_driver** y llamando a **driver_register()** para registrarse.

Si el driver esta compilado en el núcleo, el registro se produce durante la fase de inicialización, al cargarse el kernel. Si no, se se compilará como módulo y será registrado al cargarse el módulo correspondiente.

Cuando se registra un driver el núcleo comprueba, para cada dispositivo que quede sin manejar, si el driver puede manejarlo mediante la función **probe()**. En el caso de dispositivos que se puedan conectar en “caliente” (hotplug), se consulta a los drivers conocidos. Si alguno puede manejar el dispositivo, el driver es registrado.

La implementación del registro de drivers se encuentre en drivers/base/driver.c y es la siguiente:

```
157 int driver_register(struct device_driver * drv)
158 {
159         if ((drv->bus->probe && drv->probe) || // Comprueba que el driver implementa las
operaciones básicas
160             (drv->bus->remove && drv->remove) ||
161             (drv->bus->shutdown && drv->shutdown)) {
162                 printk(KERN_WARNING "Driver '%s' needs updating - please use bus_type
methods\n", drv->name);
163         }
164         klist_init(&drv->klist_devices, NULL, NULL);
165         init_completion(&drv->unloaded); // Inicializa el contador. (inicialización de un manejador de
dispositivos).
166         return bus_add_driver(drv); // añade el driver a el bus en el que va a manejar dispositivos
167 }
```

bus_add_driver se implementa en drivers/base/bus.c:

```
530 int bus_add_driver(struct device_driver *drv)
531 {
532         struct bus_type * bus = get_bus(drv->bus); // obtiene el bus que maneja el driver
533         int error = 0;
534
535         if (!bus)
536                 return 0;
537
538         pr_debug("bus %s: add driver %s\n", bus->name, drv->name);
539         error = kobject_set_name(&drv->kobj, "%s", drv->name);
540         if (error)
541                 goto out_put_bus;
542         drv->kobj.kset = &bus->drivers;
543         if ((error = kobject_register(&drv->kobj)))
544                 goto out_put_bus;
545
```

```
546       error = driver_attach(drv); // se asocia el driver al bus (analizada a continuación)
547       if (error)
548             goto out_unregister;
549       klist_add_tail(&drv->knode_bus, &bus->klist_drivers);
550       module_add_driver(drv->owner, drv); // añadimos el driver en el modulo al que pertenece
551
552       error = driver_add_attrs(bus, drv);  // se añaden los atributos del driver
553       if (error) {
554
555             printk(KERN_ERR "%s: driver_add_attrs(%s) failed\n",
556                   __FUNCTION__, drv->name);
557       }
558       error = add_bind_files(drv);  // añade los ficheros bind a sysfs
559       if (error) {
560
561             printk(KERN_ERR "%s: add_bind_files(%s) failed\n",
562                   __FUNCTION__, drv->name);
563       }
564
565       return error; // en caso de error devuelve el error
566 out_unregister:
567       kobject_unregister(&drv->kobj);
568 out_put_bus:
569       put_bus(bus);
570       return error;
571 }
```

La asociación del driver al bus consiste en recorrer cada dispositivo llamando a la función __driver_attach. El objetivo de esto es comprobar si el driver puede manejar algún dispositivo de los contectados a el bus. En caso de que así sea se asocia el driver a dicho dispositivo.

```
291 int driver_attach(struct device_driver * drv)
292 {
293       return bus_for_each_dev(drv->bus, NULL, drv, __driver_attach);
294 }
```

```
256 static int __driver_attach(struct device * dev, void * data)
257 {
258       struct device_driver * drv = data;
259
270       if (dev->parent)
271             down(&dev->parent->sem);
272       down(&dev->sem);
273       if (!dev->driver)
274             driver_probe_device(drv, dev); // se comprueba si puede manejar el dispositivo
275       up(&dev->sem);
276       if (dev->parent)
277             up(&dev->parent->sem);
278
279       return 0;
280 }
```

```
190 int driver_probe_device(struct device_driver * drv, struct device * dev)
191 {
192       struct stupid_thread_structure *data;
193       struct task_struct *probe_task;
194       int ret = 0;
195
196       if (!device_is_registered(dev)) // falla si el dispositivo no existe
197             return -ENODEV;
```

```
198       if (drv->bus->match && !drv->bus->match(dev, drv)) // cancela la operación si el driver no
maneja este dispositivo
199             goto done;
200
201       pr_debug("%s: Matched Device %s with Driver %s\n",
202             drv->bus->name, dev->bus_id, drv->name);
203
204       data = kmalloc(sizeof(*data), GFP_NÚCLEO);
205       if (!data)
206             return -ENOMEM;
207       data->drv = drv;
208       data->dev = dev;
209
          // Asocia el driver al dispositivo
210       if (drv->multithread_probe) {
211             probe_task = kthread_run(really_probe, data,
212                                      "probe-%s", dev->bus_id);
213             if (IS_ERR(probe_task))
214                   ret = really_probe(data);
215       } else
216             ret = really_probe(data);
217
218 done:
219       return ret;
220 }
```

Inicialización de un manejador de dispositivo:

El registro de un driver y la asignación de recursos al mismo, son dos cosas diferentes. Mientras que lo primero se hace lo antes posible, para que las aplicaciones pudan usarlo a través de los ficheros de dispositivo, lo segundo se realiza lo más tarde posible, hasta la primera petición de acceso, ya que implica la reserva de recursos del sistema (IRQ, marcos de página….) que podrían no estar disponibles para otros dispositivos que los necesiten.

Existe un esquema a la hora de la asignación, para que se haga cuando sea necesario y no se haga de manera redundante:

- Existe un contador de los procesos que están accediendo al fichero.
  - Aumenta con los ***open*** del fichero del dispositivo.
  - Si el contador está a nulo, el manejador debe:
    - Asignar los recursos
    - Habilitar las interrupciones y el DMA al dispositivo
  - Decrementa con los ***release*** del fichero del dispositivo.
  El ***release*** examina el valor del contador antes del decremeto.
  Si el contador está a nulo, es que no hay procesos utlizando el dispositivo:
    - Deshabilitar las interrupciones y el DMA al dispositivo
    - Desasignar los recursos

Soporte de manejadores del núcleo de Linux:
El kernel de Linux no da un soporte completo a todos los posibles dispositivos E/S existentes. En general, puede dar 3 tipos de soporte:

1. SIN SOPORTE:
   - No reconoce el dispositivo
   - La aplicación interactúa directamente con el dispositivo con las funciones de ensamblador básicas in y out
2. SOPORTE MÍNIMO:
   - No reconoce el periférico pero reconoce su interfaz E/S.

- Los programas de usuario usan el dispositivo a través de esa interfaz como un dispositivo de acceso secuencial en el que se puede leer y escribir

3.SOPORTE EXTENDIDO:
- Lo reconoce perfectamente
- El núcleo es capaz de manejar la interfaz de E/S del periférico por si mismo, lo cual podría hacer innecesario el uso del fichero de dispositivo.

Al núcleo le conviene no dar soporte, ya que reduce el tamaño, pero esto rara vez es posible (dispositivos externos).
La función **ioctl(),** que controla la E/S del manejador, verifica los argumentos o realiza la acción deseada, también es utilizada cuando read y write no son suficientes, especialmente cuando es necesario el uso de funciones especiales del dispositivo.

Implementación de ioctl en /drivers/net/wan/sdlamain.c

```
886 static int ioctl(struct wan_device* wandev, unsigned cmd, unsigned long arg)
                        //arg es un puntero al espacio de direcciones del usuario
887 {
888       sdla_t* card;
889       int err;
890
891       /* CHEQUEOS */

892       if ((wandev == NULL) || (wandev->private == NULL))
893             return -EFAULT;
894       if (wandev->state == WAN_UNCONFIGURED)
895             return -ENODEV;
896
897       card = wandev->private;
898
899       if(card->hw.type != SDLA_S514){
900             disable_irq(card->hw.irq);
901       }
902
903       if (test_bit(SEND_CRIT, (void*)&wandev->critical)) {
904             return -EAGAIN;
905       }
906
907       switch (cmd) {
908       case WANPIPE_DUMP:
909             err = ioctl_dump(wandev->private, (void*)arg);
910             break;
911
912       case WANPIPE_EXEC:
913             err = ioctl_exec(wandev->private, (void*)arg, cmd);
914             break;
915       default:
916             err = -EINVAL;
917       }
918
919       return err;
920 }
```

Estrategias de buffering de los drivers:
Es una de las características de los manejadores de dispositivos que consiste en que el núcleo debe estar preparado para manejar grandes transferencias de datos eficientemente.
Para ello existen dos técnicas:
• Usar el DMAC para transferir bloques de datos.
• Usar buffer circular que es un buffer intermedio en el que se escriben bloques que se transfieren. Cuando el driver escribe uno de esos bloques en memoria de usuario, libera la posición. El objetivo de esta técnica es suavizar los picos de carga CPU.

Si la aplicación recibe datos más lentamente, pero el DMAC sigue funcionando y el manejador de interrupciones sigue almacenando en el buffer, con lo cual están trabajando para la aplicación.

I/O.4.2 Supervisión de las operaciones de entrada salida:

Un manejador que comienza una operación E/S debe contar con unas técnicas de supervisión, que le señalen tanto la terminación de la operación como un eventual time-out.

Existen 2 técnicas para ello:

1. Técnica por encuestas (Polling mode): La CPU chequea cada cierto tiempo el registro de estado del dispositivo, hasta que su valor señale que la operación ha terminado.
2. Técnica por interrupciones (Interrupt mode): El controlador de E/S es capaz de avisar, vía IRQ, el final de la operación.

Técnica por encuestas:
Ejemplo básico

```
for(;;){
   if(read_status(device) & DEVICE_END_OPERATION) break;
   if(--count==0)break;
}
```

La variable *count* se inicializa antes de entrar en el bucle y se decrementa en cada iteracion, por lo que se puede usar como un tosco mecanismo de time-out.
Si el tiempo de ejecución de la operación E/S es grande, esta técnica es ineficiente porque la CPU desperdicia ciclos de reloj esperando por el final. En este caso, es mejor liberar la CPU después de cada consulta insertando una llamada a *schedule*() en el bucle para dormir es proceso.

Técnica por interrupciones:
En el caso de que al acceder a través de las funciones genéricas read o write, la respuesta sea impredeciblemente larga, es preferible implementar el acceso con la técnica de las interrupciones.
Se basa en dos funciones del manejador (cada manejador tendrá las suyas propias):
1. foo_read() que lee el fichero del dispositivo.
2. foo_interrupt() que maneja la interrupción

Código foo_Read:

```
Ssize_t foo_read( struct file *filp, char *buf, size_t count, loff_t *ppos) {
        foo_dev_t * foo_dev = filp->private_data;
        if(down_interruptible(&foo_dev->sem) //Coge el semáforo para asegurar que no
                                                // habrá otro proceso que acceda
            return –ERESTARTSYS;
         foo_dev->intr =0; //Pone los flags a 0

         outb(DEV_FOO_READ,DEV_FOO_CONTROL_CONTROL_PORT); //Utiliza el
                                                          // comando read del dispositivo
         wait_event_interruptible(foo_dev->wait, (foo_dev->intr==1) //Suspende el proceso
                                                          //hasta que flags tome //el
                                                          valor 1
         if (put_user(foo_dev->data,buf)) //Copia el carácter de preparado en el espacio de //direcciones
                                          del usuario
            return –EFAULT;
         up(&foo_dev->sem); //Libera el semáforo
         return 1;
}
```

Código foo_interrupt():

```
void foo_interrupt(int irq, void *dev_id, struct pt_regs * regs) {
     foo->data= inb(DEV_FOO_DATA_PORT);//Lee el caracter del registro de entrada
      foo->intr =1;//Activa los flags
      wake_up_interruptible(&foo->wait); //Despierta al proceso quitandolo de la cola de espera
}
```

I/O.4.3 Drivers de dispositivos de caracteres:

Los drivers de dispositivos de caracteres son relativamente simples, ya que 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:

```
13 struct cdev {
14        struct kobject kobj;
15        struct module *owner;
16        const struct file_operations *ops;
17        struct list_head list;
18        dev_t dev;
19        unsigned int count;
20 };
```

Lo drivers de caracteres no tienen una implementación general ya que no trabajan con grandes flujos de datos sino con operaciones simples y concretas sobre dispositivos. Por ello, cada driver se encarga de implementar sus operaciones directamente y no existen, como en los dispositivos de bloques, operaciones genéricas de lectura/escritura que el driver debe traducir a su controlador.

Para saber que dispositivos están actualmente en uso, el kernel utiliza una tabla hash indexada por el número major. El vector de la tabla hash se almacena en la variable chrdevs, que incluye 255 punteros o descriptores a dispositivos de caracteres. Cada elemento de este vector es una estructura de dispositivo de caracteres que se muestra a continuación:
En el fichero ***/fs/char_dev.c*** encontramos las siguientes estructuras

```
static struct char_device_struct {
34        struct char_device_struct *next;
35        unsigned int major;
36        unsigned int baseminor;
37        int minorct;
38        const char *name;
39        struct file_operations *fops;
40        struct cdev *cdev;            /* will die */
41 } *chrdevs[MAX_PROBE_HASH]; /*tabla hash*/
```

Este es el indexado por el número mayor

```
43 /* index in the above */
44 static inline int major_to_index(int major)
45 {
46        return major % MAX_PROBE_HASH;
47 }
```

```
#define MAX_PROBE_HASH 255      /* Elementos de la tabla hash*/
```

Un descriptor de dispositivo se inserta en la tabla hash utilizando la función **init_special_inode(),** la cual es invocada por el sistema de fichero de bajo nivel. Esta función busca el descriptor de dispositivo en la tabla hash. Si no lo encuentra, la función crea el nuevo descriptor de dispositivo y lo inserta en la tabla hash. En el fichero ***/fs/inode.c***

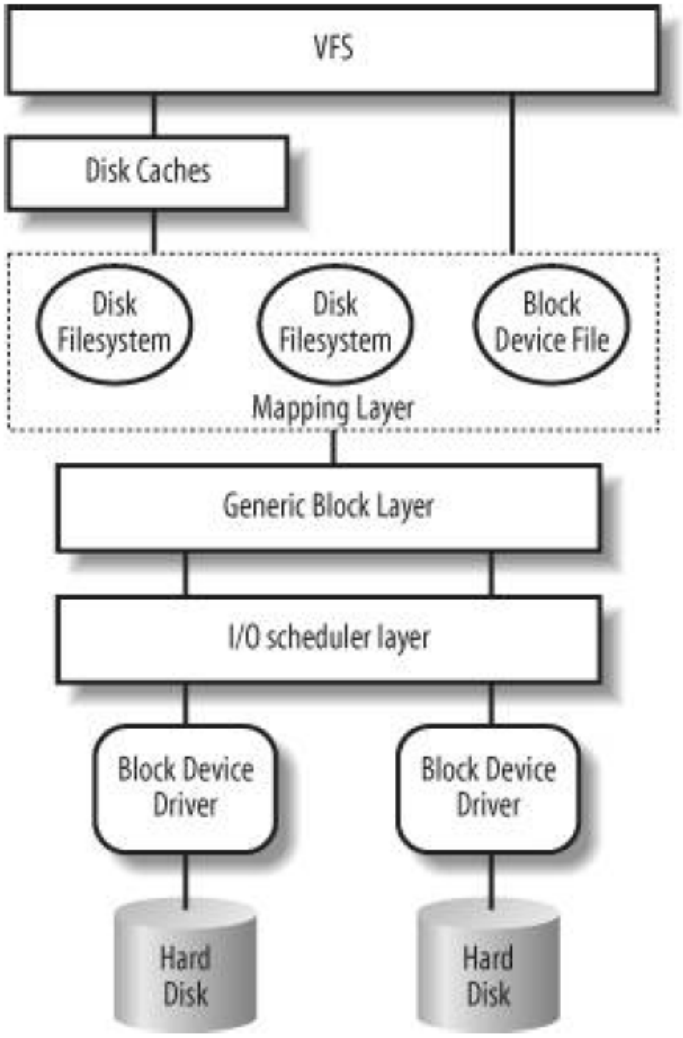
```
void init_special_inode(struct inode *inode, umode_t mode, dev_t rdev)
1361 {
1362        inode->i_mode = mode;
1363        if (S_ISCHR(mode)) {
1364              inode->i_fop = &def_chr_fops;
1365              inode->i_rdev = rdev;
1366        } else if (S_ISBLK(mode)) {
1367              inode->i_fop = &def_blk_fops;
1368              inode->i_rdev = rdev;
1369        } else if (S_ISFIFO(mode))
1370              inode->i_fop = &def_fifo_fops;
1371        else if (S_ISSOCK(mode))
1372              inode->i_fop = &bad_sock_fops;
1373        else
1374              printk(KERN_DEBUG "init_special_inode: bogus i_mode (%o)\n",
1375                    mode);
1376 }
1377 EXPORT_SYMBOL(init_special_inode);
```

I/O.5 Drivers 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.
Los núcleos que soportan dispositivos de bloques incluyen las siguientes características:

a. Una interfaz uniforme a través del VFS
b. Una eficiente lectura anticipada de los datos
c. Estrategias de caché para los datos

Cada operación de Entrada/Salida involucra a un conjunto de componentes del Núcleo, como se muestra en la siguiente figura.



Sectores, bloques y buffers

Sector.-unidad básica de datos transferida por un dispositivo hardware. El tamaño de un sector es de 512, 1.024, o 2.048 bytes

Hardsect_size.- Tabla donde el núcleo almacena el tamaño del sector de cada dispositivo indexado por el major y minor número de los dispositivos.

Ej: hardsect_size [3] [2] =tamaño del sector */dev/hda2*

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

Bloque.- Grupo de bytes adyacentes involucrados en una operación de entrada salida. 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 Núcleo para almacenar el contenido del bloque.

bllsize_size.- Tabla donde el núcleo almacena el tamaño del bloque de cada dispositivo indexada por el major y minor número de los dispositivos.

Cabeceras de los buffers

La cabeza de buffer es un descriptor de tipo buffer_head asociado con cada buffer. Este descriptor contiene toda la información que necesita el núcleo para manejar un buffer. Se encuentra en */include/linux/buffer_head.h*

```
struct buffer_head {
 61      unsigned long b_state;          /* buffer state bitmap (see above) */
 62      struct buffer_head *b_this_page;/* circular list of page's buffers */
 63      struct page *b_page;            /* the page this bh is mapped to */
 64
 65      sector_t b_blocknr;             /* start block number */
 66      size_t b_size;                  /* size of mapping */
 67      char *b_data;                   /* pointer to data within the page */
 68
 69      struct block_device *b_bdev;
 70      bh_end_io_t *b_end_io;          /* I/O completion */
 71      void *b_private;                /* reserved for b_end_io */
 72      struct list_head b_assoc_buffers; /* associated with another mapping */
 73      struct address_space *b_assoc_map;      /* mapping this buffer is
 74                                         associated with */
 75      atomic_t b_count;               /* users using this buffer_head */
 76};
```

En el fichero include/linux/buffer_head.h encontramos los siguientes flags que almacena el campo b_state de la estructura buffer_head :

```
 19enum bh_state_bits {
 20      BH_Uptodate,    /* Contains valid data */
 21      BH_Dirty,       /* Is dirty */
 22      BH_Lock,        /* Is locked */
 23      BH_Req,         /* Has been submitted for I/O */
 24      BH_Uptodate_Lock,/* Used by the first bh in a page, to serialise
 25                      * IO completion of other buffers in the page
 26                      */
 27
 28      BH_Mapped,      /* Has a disk mapping */
 29      BH_New,         /* Disk mapping was newly created by get_block */
 30      BH_Async_Read,  /* Is under end_buffer_async_read I/O */
 31      BH_Async_Write, /* Is under end_buffer_async_write I/O */
 32      BH_Delay,       /* Buffer is not yet allocated on disk */
 33      BH_Boundary,    /* Block is followed by a discontiguity */
```

```
 34      BH_Write_EIO,   /* I/O error on write */
 35      BH_Ordered,     /* ordered write */
 36      BH_Eopnotsupp,  /* operation not supported (barrier) */
 37      BH_Unwritten,   /* Buffer is allocated on disk but not written */
 38
 39      BH_PrivateStart,/* not a state bit, but the first bit available
 40                       * for private allocation by other entities
 41                       */
 42};
```

I/O.5.1The Generic Block Layer

La capa genérica de bloque se trata de un componente que facilita al kernel el manejo de los dispositivos de bloques.
Sus funciones son las siguientes:

- Mapea los datos en el espacio de direcciones del kernel unicamente cuando la cpu va a acceder a estos y desmontadolos cuando ya no sean necesarios.
- Implementa un esquema de “zero-copy”, cuando un dato se copia directamente en el espacio de direciones del usuario, el buffer utilizado por el kernel para la tranferencia reside en una página del espacio de direcciones del proceso del usuario.
- Maneja volumenes logico utilizado en en LVM y RAIDs: podemos ver particiones de diferentes dispositivos como una unica partición.
- Compatible con las caracteristicas de los nuevos discos.

I/O.5.2 La Estructura Bio

La estructura que conforma el núcleo de la capa generica de bloque es un descriptor a una operación de I/O sobre un dispositivo de bloque, llamada Bio. 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(). A continuación se muestra la estructura Bio.

```
struct bio {
 75      sector_t            bi_sector;     /* device address in 512 byte
 76                                            sectors */
 77      struct bio          *bi_next;      /* request queue link */
 78      struct block_device     *bi_bdev;
 79      unsigned long           bi_flags;      /* status, command, etc */
 80      unsigned long           bi_rw;         /* bottom bits READ/WRITE,
 81                                              * top bits priority
 82                                              */
 83
 84      unsigned short          bi_vcnt;       /* how many bio_vec's */
 85      unsigned short          bi_idx;        /* current index into bvl_vec */
 86
 87      /* Number of segments in this BIO after
 88       * physical address coalescing is performed.
 89       */
 90      unsigned short          bi_phys_segments;
 91
 92      /* Number of segments after physical and DMA remapping
 93       * hardware coalescing is performed.
 94       */
 95      unsigned short          bi_hw_segments;
 96
```

```
 97      unsigned int          bi_size;       /* residual I/O count */
 98
 99      /*
100       * To keep track of the max hw size, we account for the
101       * sizes of the first and last virtually mergeable segments
102       * in this bio
103       */
104      unsigned int          bi_hw_front_size;
105      unsigned int          bi_hw_back_size;
106
107      unsigned int          bi_max_vecs;    /* max bvl_vecs we can hold */
108
109      struct bio_vec        *bi_io_vec;     /* the actual vec list */
110
111      bio_end_io_t          *bi_end_io;
112      atomic_t              bi_cnt;        /* pin count */
113
114      void                  *bi_private;
115
116      bio_destructor_t      *bi_destructor; /* destructor */
117};
```

Cada segmento en una Bio es representado por una estructura *bio_vec* El campo *bi_io_vec* de la Bio apunta al primer elemento de un array de estructuras *bio_vec*, mientras que el campo bi_vcnt almacena el número actual de elementos en el array.

```
59struct bio_vec {
 60      struct page     *bv_page;
 61      unsigned int    bv_len;
 62      unsigned int    bv_offset;
 63};
```

I/O.5.3 Estructura gendisk

Un disco es un dispositivo lógico de bloques que es manejado por la generic block layer . Para ello un disco es representado mediante la estructura 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, RAID, LVM).

```
114struct gendisk {
115      int major;                  /* major number of driver */
116      int first_minor;
117      int minors;                 /* maximum number of minors, =1 for
118                                  * disks that can't be partitioned. */
119      char disk_name[32];         /* name of major driver */
120      struct hd_struct **part;    /* [indexed by minor] */
121      struct block_device_operations *fops;
122      struct request_queue *queue;
123      void *private_data;
124      sector_t capacity;
125
126      int flags;
127      struct device *driverfs_dev;  // FIXME: remove
128      struct device dev;
129      struct kobject *holder_dir;
130      struct kobject *slave_dir;
```

```
131
132       struct timer_rand_state *random;
133       int policy;
134
135       atomic_t sync_io;          /* RAID */
136       unsigned long stamp;
137       int in_flight;
138#ifdef  CONFIG_SMP
139       struct disk_stats *dkstats;
140#else
141       struct disk_stats dkstats;
142#endif
143       struct work_struct async_notify;
144};
145
```

Exites multitud de flags en esta estructura ahora comentaremos los mas importantes
GENHD_FL_UP dispositivos activado
GENHD_FL_REMOVABLE dispositivo extraible
GENHD_FL_CD activo si el dispositivo es un CD ROM

El campo fops de el objeto gendisk apunta a una tabla block_device_operations , la cual almacena las funciones cruciales sobre el dispositivo de bloque.

Métodos de dispositivos de bloques	
Método	**Disparador**
open	Abrir el archivo de dispositivo de bloque
release	Cerrar la última referencia a un archivo de dispositivo de bloque.
ioctl	Llamada al sistema ioctl() sobre el archivo de dispositivo de bloque(usa el cerrojo del núcleo)
compat_ioctl	Llamada al sistema ioctl() sobre el archivo de dispositivo de bloque(no usa el cerrojo del núcleo)
media_changed	Comprueba si el dispositivo extraíble ha sido cambiado (ej. Floppy disk)
revalidate_disk	Comprueba si el dispositivo de bloque mantiene datos válidos

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*

```
struct hd_struct {
 88       sector_t start_sect; //comienzo de la particion
 89       sector_t nr_sects; //longitud de la particion
 90       struct device dev; //struct device del dispositivo
 91       struct kobject *holder_dir; //kobject incrustado
 92       int policy, partno;
 93#ifdef CONFIG_FAIL_MAKE_REQUEST
 94       int make_it_fail;
 95#endif
 96       unsigned long stamp;
 97       int in_flight;
 98#ifdef  CONFIG_SMP
 99       struct disk_stats *dkstats; //estado del disco
100#else
101       struct disk_stats dkstats;
102#endif
103};
```

I/O.5.4 Descriptor de peticiones

Cuando un componente del núcleo 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)

Cada petición es representada por un descriptor de petición el cual es almacenado en una estructura de peticiones llamada **request**. La dirección del dato a trasmitir es almacenada en el campo *cmd*, este dato puede ser leído del dispositivo a la Ram o escrito de la RAM al dispositivo. Esta estructura se encuentra en ***include /linux/blkdev.***

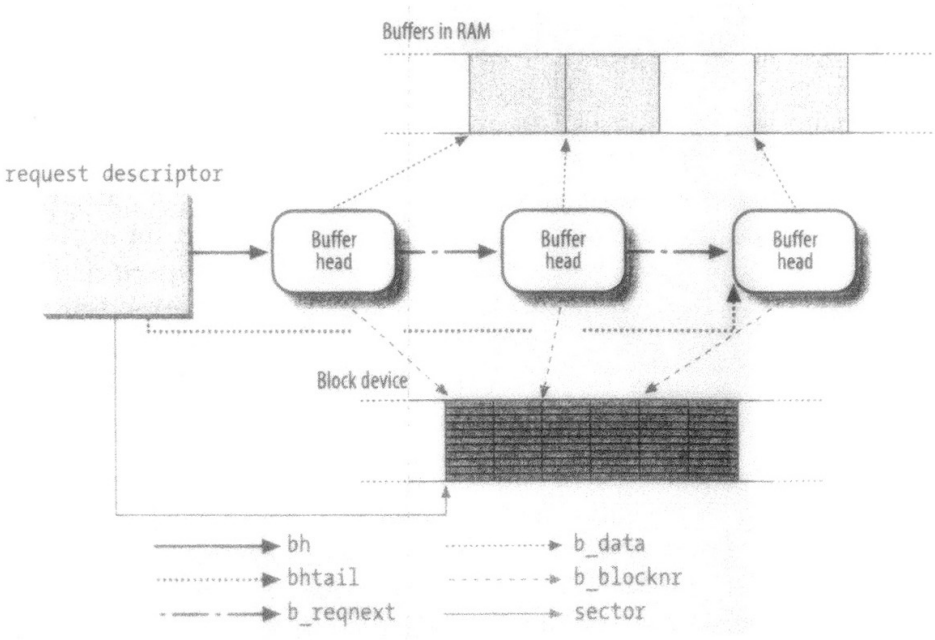
```
struct request {
145     struct list_head queuelist; /*lista de  la cola buffer head*/
146     struct list_head donelist; /*cola de terminados*/
147
148     struct request_queue q; /*cola q*/
149
150     unsigned int cmd_flags;/*flags de la peticion
151     enum rq_cmd_type_bits cmd_type;

157     sector_t sector;            /* siguente sector pedido */
158     sector_t hard_sector;       /* proximo sector a completar*/
159     unsigned long nr_sectors;   /* numero de sectores requeridos*/
160     unsigned long hard_nr_sectors;  /* numero de sectores restantes a para completar */
161     /*numero de sectores en el segmento actual*/
162     unsigned int current_nr_sectors;
163
164     /*numero de sectores que faltan para terminar en el segmento */
165     unsigned int hard_cur_sectors;
166
167     struct bio *bio;
168     struct bio *biotail;
169
170     struct hlist_node hash; /* merge hash */
171     /* El rb_node es utilizdo en el planificador de E/S */

176     union {
177         struct rb_node rb_node;
178         void *completion_data;
179     };
```

hard_nr_sectors, hard_sector. Estos campos se utilizan cuanto un driver maneja varios dispositivos de bloque de una sola vez. Un ejemplo típico es el MANEJADOR DE VOLÚMEN LÓGICO el cual es capaz de manejar varios discos en una partición de disco como una partición virtual. En este caso los campos se refieren al dispositivo virtual. Si no existen las particiones virtuales los valores son iguales que los del dispositivo físico.

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



Explicación del dibujo
El campo buffer del descriptor de petición apunta al área de memoria usada por la transferencia actual de datos. Si la petición involucra un bloque simple, el buffer es una copia del campo *b_data de la estructura de cabeza de buffer.* Sin embargo, si la petición tiene varios bloques cuyo buffer no son consecutivos en memoria, los buffer se unen a través del campo b_reqnext del buffer head.
En una lectura a bajo nivel, el driver puede elegir destinar una gran área de memoria apuntada por el buffer. Lee todo el sector de la petición de una sola vez y luego copia los datos en varios buffers. Similar para la escritura.

Cola de descriptores de petición

```
struct request_queue
279{
280       /*
281        * Together with queue_head for cacheline sharing
282        */
283       struct list_head        queue_head;
284       struct request          *last_merge;
285       elevator_t              *elevator;
286
287       /*
288        * the queue request freelist, one for reads and one for writes
289        */
290       struct request_list     rq;
291
292       request_fn_proc         *request_fn;
293       make_request_fn         *make_request_fn;
294       prep_rq_fn              *prep_rq_fn;
295       unplug_fn               *unplug_fn;
296       merge_bvec_fn           *merge_bvec_fn;
```

```
297     prepare_flush_fn        *prepare_flush_fn;
298     softirq_done_fn         *softirq_done_fn;
299     dma_drain_needed_fn     *dma_drain_needed;
300
301     /*
302      * Dispatch queue sorting
303      */
304     sector_t                end_sector;
305     struct request          *boundary_rq;
306
307     /*
308      * Auto-unplugging state
309      */
310     struct timer_list       unplug_timer;
311     int                     unplug_thresh;  /* After this many requests */
312     unsigned long           unplug_delay;   /* After this many jiffies */
313     struct work_struct      unplug_work;
314
315     struct backing_dev_info backing_dev_info;
316
317     /*
318      * The queue owner gets to use this for whatever they like.
319      * ll_rw_blk doesn't touch it.
320      */
321     void                    *queuedata;
322
323     /*
324      * queue needs bounce pages for pages above this limit
325      */
326     unsigned long           bounce_pfn;
327     gfp_t                   bounce_gfp;
328
329     /*
330      * various queue flags, see QUEUE_* below
331      */
332     unsigned long           queue_flags;
333
334     /*
335      * protects queue structures from reentrancy. ->__queue_lock should
336      * _never_ be used directly, it is queue private. always use
337      * ->queue_lock.
338      */
339     spinlock_t              __queue_lock;
340     spinlock_t              *queue_lock;
341
342     /*
343      * queue kobject
344      */
345     struct kobject kobj;
346
347     /*
348      * queue settings
349      */
350     unsigned long           nr_requests;    /* Max # of requests */
351     unsigned int            nr_congestion_on;
352     unsigned int            nr_congestion_off;
353     unsigned int            nr_batching;
354
355     unsigned int            max_sectors;
356     unsigned int            max_hw_sectors;
```

```
357     unsigned short          max_phys_segments;
358     unsigned short          max_hw_segments;
359     unsigned short          hardsect_size;
360     unsigned int            max_segment_size;
361
362     unsigned long           seg_boundary_mask;
363     void                    *dma_drain_buffer;
364     unsigned int            dma_drain_size;
365     unsigned int            dma_pad_mask;
366     unsigned int            dma_alignment;
367
368     struct blk_queue_tag    *queue_tags;
369     struct list_head        tag_busy_list;
370
371     unsigned int            nr_sorted;
372     unsigned int            in_flight;
373
374     /*
375      * sg stuff
376      */
377     unsigned int            sg_timeout;
378     unsigned int            sg_reserved_size;
379     int                     node;
380#ifdef CONFIG_BLK_DEV_IO_TRACE
381     struct blk_trace        *blk_trace;
382#endif
383     /*
384      * reserved for flush operations
385      */
386     unsigned int            ordered, next_ordered, ordseq;
387     int                     orderr, ordcolor;
388     struct request          pre_flush_rq, bar_rq, post_flush_rq;
389     struct request          *orig_bar_rq;
390
391     struct mutex            sysfs_lock;
392
393#if defined(CONFIG_BLK_DEV_BSG)
394     struct bsg_class_device bsg_dev;
395#endif
396};
```

Función LL_rw_block

Esta función crea una petición para los dispositivos de bloque. Esta función es invocada desde diferentes partes del núcleo para disparar transferencias de entrada salida de uno o más bloques. Se encuentra en /fs/buffer.c
Antes de acceder a esta función se comprueba si el tamaño del bloque de dispositivo es igual al tamaño de bloque de los buffer, además si la operación es de escritura se comprueba que el dispositivo no sea de sólo lectura.

Parámetros:
rw indica la operación a realizar que puede ser **Read**,**Write o Reada**
nr indica el número de bloques a transferir
bhs vector de punteros nr a buffer heads de bloques. Todos deben tener el mismo tamaño y referirse al mismo bloque.

La función realiza las siguientes acciones:
Para cada buffer head en el vector bh realiza los siguientes pasos:

a. Mira si el flag BH_Lock del buffer head está activo.Si ya fue activado por otro hilos del sistema salta ese buffer.
b. Asigna al campo bd_end_io la función end_buffer_write_sync, o end_buffer_read_sync según sea una escritura o lectura que maneja el descriptor del buffer cuando la transferencia se completa.
c. Si es una escritura mira el flag BH_Dirty, según su valor puede ocurrir lo siguiente:
   Si está activa,llama a submit_bh***(***WRITE***,*** bh***);***
d. Como regla general la función ll_rw_block debe activar el flag BH_Dirty para cada bloque que se va a escribir. Sin embargo, si el ll_rw_block detecta que el flags está desbloqueado, entonces el flags está ya involucrado en una operación de escritura por lo que no puede hacer nada.
e. Si es una lectura mira el flag BH_Uptodatedel buffer head, si está activo invoca ***a*** submit_bh***(***rw***,*** bh***);***

```
2948void ll_rw_block(int rw, int nr, struct buffer_head *bhs[])
2949{
2950        int i;
2951
2952        for (i = 0; i < nr; i++) {
2953                struct buffer_head *bh = bhs[i];
2954
2955                if (rw == SWRITE)
2956                        lock_buffer(bh);
2957                else if (test_set_buffer_locked(bh))
2958                        continue;
2959
2960                if (rw == WRITE || rw == SWRITE) {
2961                        if (test_clear_buffer_dirty(bh)) {
2962                                bh->b_end_io = end_buffer_write_sync;
2963                                get_bh(bh);
2964                                submit_bh(WRITE, bh);
2965                                continue;
2966                        }
2967                } else {
2968                        if (!buffer_uptodate(bh)) {
2969                                bh->b_end_io = end_buffer_read_sync;
2970                                get_bh(bh);
2971                                submit_bh(rw, bh);
2972                                continue;
2973                        }
2974                }
2975                unlock_buffer(bh);
2976        }
2977}
2978
```

Función submit_bh

El objetivo de esta función es rellenar el contenido de la bio, que será utilizada para realizar las operaciones de entrada salida a bajo nivel

```
int submit_bh(int rw, struct buffer_head * bh)
2875{
2876        struct bio *bio;
2877        int ret = 0;
2878
2879        BUG_ON(!buffer_locked(bh));
2880        BUG_ON(!buffer_mapped(bh));
```

```
2881        BUG_ON(!bh->b_end_io);
2882
2883        if (buffer_ordered(bh) && (rw == WRITE))
2884                rw = WRITE_BARRIER;
2885
2886        /*
2887         * Only clear out a write error when rewriting, should this
2888         * include WRITE_SYNC as well?
2889         */
2890        if (test_set_buffer_req(bh) && (rw == WRITE || rw == WRITE_BARRIER))
2891                clear_buffer_write_io_error(bh);
2892
2893        /*
2894         * from here on down, it's all bio -- do the initial mapping,
2895         * submit_bio -> generic_make_request may further map this bio around
2896         */
2897        bio = bio_alloc(GFP_NOIO, 1);
2898
2899        bio->bi_sector = bh->b_blocknr * (bh->b_size >> 9);
2900        bio->bi_bdev = bh->b_bdev;
2901        bio->bi_io_vec[0].bv_page = bh->b_page;
2902        bio->bi_io_vec[0].bv_len = bh->b_size;
2903        bio->bi_io_vec[0].bv_offset = bh_offset(bh);
2904
2905        bio->bi_vcnt = 1;
2906        bio->bi_idx = 0;
2907        bio->bi_size = bh->b_size;
2908
2909        bio->bi_end_io = end_bio_bh_io_sync;
2910        bio->bi_private = bh;
2911
2912        bio_get(bio);
2913        submit_bio(rw, bio);
2914
2915        if (bio_flagged(bio, BIO_EOPNOTSUPP))
2916                ret = -EOPNOTSUPP;
2917
2918        bio_put(bio);
2919        return ret;
2920}
```

generic_make_request

Esta función hace una serie de comprobaciones sobre la estructura bio antes de enviar los datos a los dispositivos de bajo nivel.

```
void generic_make_request(struct bio *bio)
1424{
1425        if (current->bio_tail) {
1426                /* make_request is active */
1427                *(current->bio_tail) = bio;
1428                bio->bi_next = NULL;
1429                current->bio_tail = &bio->bi_next;
1430                return;
1431        }
1432        /* following loop may be a bit non-obvious, and so deserves some
1433         * explanation.
1434         * Before entering the loop, bio->bi_next is NULL (as all callers
```

```
1435        * ensure that) so we have a list with a single bio.
1436        * We pretend that we have just taken it off a longer list, so
1437        * we assign bio_list to the next (which is NULL) and bio_tail
1438        * to &bio_list, thus initialising the bio_list of new bios to be
1439        * added.  __generic_make_request may indeed add some more bios
1440        * through a recursive call to generic_make_request.  If it
1441        * did, we find a non-NULL value in bio_list and re-enter the loop
1442        * from the top.  In this case we really did just take the bio
1443        * of the top of the list (no pretending) and so fixup bio_list and
1444        * bio_tail or bi_next, and call into __generic_make_request again.
1445        *
1446        * The loop was structured like this to make only one call to
1447        * __generic_make_request (which is important as it is large and
1448        * inlined) and to keep the structure simple.
1449        */
1450       BUG_ON(bio->bi_next);
1451       do {
1452             current->bio_list = bio->bi_next;
1453             if (bio->bi_next == NULL)
1454                   current->bio_tail = &current->bio_list;
1455             else
1456                   bio->bi_next = NULL;
1457             __generic_make_request(bio);
1458             bio = current->bio_list;
1459       } while (bio);
1460       current->bio_tail = NULL; /* deactivate */
1461}
```

I/O.5.5 Planificador de Entrada/Salida

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.

Existen otros algoritmos de planificación:

- Deadline
- Anticipación
- CFQ (Complete Fairness Queueing)
- Noop

Linus Elevator I/O Scheduler

Era el algoritmo por defecto (y único) para la serie 2.4. Este algoritmo funciona exactamente como se ha descrito en la sección anterior, proporciona sorting y merging. Pero obtiene poca eficiencia en determinadas situaciones como más tarde veremos.

En el Linus Elevator, cuando una petición es añadida a la cola de peticiones, primero se coteja con el resto de peticiones en la cola para ver si es candidata de merging. Es posible un merging por delante o un merging por detrás, es decir, si a la nueva petición le precede inmediatamente una petición

adyacente, se produce un front merging, si por el contrario la nueva petición es adyacente inmediatamente después de una ya existente, se produce merging por detrás. Si el intento de merging falla, se intenta insertar en la cola de manera ordenada, sino se añade al final de la cola.

Un problema inherente a este sistema se ilustra en el siguiente ejemplo. Consideremos un flujo de peticiones de e/s de los siguientes bloques 20, 30, 700 y 25, en este orden. El algoritmo Linus Elevator tras realizar el sorting procesaría la cola en el siguiente orden, 20, 25, 30 y 700. Sin embargo, supongamos que cuando va a servir la petición del bloque 25, llega otra petición en la misma parte del disco (bloque 26 por ejemplo), y luego otra (el 27 ....), y luego otra, vemos que es totalmente posible que la petición del bloque 700 no sea servida en bastante tiempo. La situación puede ser peor aún. Las peticiones de lectura suelen ser síncronas, es decir, cuando una aplicación emite una petición de lectura de algún dato, típicamente se bloquea hasta que el núcleo devuelve el dato. La aplicación debe esperar, hasta que nuestro bloque 700 fuera servido por el núcleo, si la petición hubiera sido una lectura sobre el mismo.

El algoritmo Linus Elevator intenta eliminar la inanición de la siguiente manera, si una petición de e/s ya existente en la cola es más vieja que un predefinido umbral, la nueva petición es añadida al final de la cola, aunque pudiera haberse ordenado. Esto previene que cuando hay muchas peticiones en zonas cercanas del disco provoquen inanición de otras más lejanas. El problema es que el uso de este umbral no es muy eficiente en este algoritmo. En realidad el algoritmo es únicamente para la inserción ordenada en la cola cuando se ha cumplido dicho umbral para cierta petición. El resultado puede ser poco rentable en el cómputo global.

Deadline I/O Scheduler

Como hemos visto el algoritmo Linus Elevator mantiene una única cola ordenada de peticiones de e/s. La petición en la cabeza de la cola, es la siguiente petición a ser servida. El algoritmo Deadline usa tres colas en la forma que explicamos a continuación.

Cada petición de e/s es asociada con un tiempo de expiración, 500 milisegundos para peticiones de lectura y 5 segundos para peticiones de escritura. Se mantiene la cola del algoritmo Linus Elevator, es decir, una lista ordenada por adyacencia en el disco de las peticiones, en este ámbito la cola es denominada la “sorted queue” (cola ordenada), o cola estándar. Cuando una nueva petición de e/s llega es mezclada y ordenada en la sorted queue, al estilo del algoritmo tradicional (hay que indicar que en el Deadline, el merging por delante es opcional). El planificador Deadline también inserta la petición en una segunda cola que depende del tipo de petición. Las peticiones de lectura son ordenadas en una cola especial conocida como “Read FIFO queue” y las peticiones de escritura en una cola especial denominada “Write FIFO queue”. Si bien las peticiones son ordenadas en la sorted queue por sector del disco, estas dos colas especiales son ordenadas por tiempo de llegada, es decir, FIFO, nuevas peticiones son añadidas a la cola de la lista. Bajo operación normal el algoritmo Deadline recoge las peticiones de la cabeza de la sorted queue y las coloca para su emisión en la “dispatch queue” (es el paso último antes del disco). No obstante, si una petición de la cabeza de la Write FIFO queue o de la Read FIFO queue expira, entoces el Deadline deja de servir las peticiones de la cola estándar y da servicio a la petición de la cola FIFO que expiró, también sirve un par más de esa cola, como prevención (serán próximas a expirar).

Anticipatory I/O Scheduler

Resumiendo el planificador Anticipatory se comporta exactamente igual al Deadline, implementa las tres colas anteriores (mas la dispatch queue) y expiración para las peticiones, exáctamente igual al Deadline. El mayor cambio es la adición de una heurística de anticipación. El planificador Anticipatory intenta minimizar los posicionamientos del cabezal (los seeks) de las peticiones de lecturas mientras otra actividad se está produciendo en el disco. Cuando una petición de lectura es emitida, es tratada de la manera usual, dentro de su tiempo de expiración (deadline). Después de que la petición es servida, el planificador Anticipatory no vuelve a posicionarse sobre las peticiones que dejó pendientes, sino que se para, no hace nada, en 6 milisegundos (es un valor configurable), en espera de que la aplicación emita otra petición de lectura en la misma zona. Y así hasta que expiran ese tiempo por no llegar más peticiones de lectura en esa zona.

Es importante advertir que merece la pena ese desperdicio de tiempo intentando anticiparse a las lecturas, pero en el cómputo global es más rentable en rendimiento que el desperdiciado en los dobles posicionamientos del cabezal.

Complete Fair Queueing I/O Scheduler

El planificador CFQ fue diseñado para cargas de trabajo especializadas, pero en la práctica ha proporcionado un buen rendimiento en todo tipo de cargas de trabajo. Es en esencia totalmente distinto a los planificadores que hemos estudiado. El planificador CFQ mantiene una cola por cada proceso que emite peticiones de e/s. Por ejemplo si el proceso P1 emite peticiones de e/s se encolarán en una cola específica para dicho proceso. Las peticiones emitidas por un proceso P2 serán coleccionadas en una cola del proceso P2.

Dentro de cada cola las peticiones son mezcladas según sean adyacentes y ordenadas (merging y sorting). El planificador entonces sirve un número de peticiones configurable (por defecto, cuatro) por cada cola según round robin. Esto proporciona imparcialidad a nivel de los procesos, asegurando que cada proceso del sistema recibe una porción de ancho de banda del disco justa.

El escenario de carga de trabajo sobre el que se intencionó el diseño era en los sistemas multimedia, en los que con el algoritmo CFQ se garantizaba que un reproductor de audio, por ejemplo, fuera capaz de rellenar sus buffers de audio del disco a tiempo, sin saltos. En la práctica se ha observado que dicho algoritmo se comporta de manera muy aceptable en todo tipo de escenarios.

Noop I/O Sheduler

Este planificador es algo especial, su función es no hacer casi nada (no-operation). No realiza ordenamiento de las peticiones, no se preocupa de los posicionamientos de los cabezales (seeks), pero si realiza merging. 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, sin movimientos de cabezales, como los dispositivos flash USB o PCMCIA.

I/O.5.6 Manejadores de dispositivos de bloque

Cuando se va a abrir el fichero de un dispositivo el núcleo debe determinar si ya está abierto. Para saber que dispositivo está en uso el kerner utiliza una tabla hash indexada por el número mayor y el menor.
Por lo tanto si el driver del dispositivo está en la tabla hash, es que ya está en uso. Si por el contrario al chequear no lo encuentra, el núcleo inserta el nuevo elemento en la tabla.
El vector asociado a la tabla hash está almacenado en la variable: bdev_hashtable (incluye 64 listas de descriptores de dispositivos bloque)

```
struct block_device {
360     dev_t               bd_dev;  /* mayor y menor  número del dispositivo*/
361     struct inode *      bd_inode;     /* puntero al inode principal del dispositivo */
362     int                 bd_openers;   /*Cuenta el nº de veces que se ha abierto el dispositivo*/

363     struct mutex        bd_mutex;  /* Semáforo que protege el driver del dispositivo */

364     struct semaphore    bd_mount_sem;   /* mount mutex */
365     struct list_head    bd_inodes;
366    void *          bd_holder; /*Actual titular de el descriptor de el dispositivo de bloque*/
367     int                 bd_holders;
368     struct block_device *   bd_contains; /* Si el dispositivo es una partición apunta al descriptor de dispositivo del disco entero, en otro caso, apunta a sí mismo*/

369     unsigned            bd_block_size;
```

370 struct hd_struct * bd_part; ***/*Puntero al descriptor de la partición (NULL si no es una partición)*/***

372 unsigned bd_part_count***; /*número de veces que las particiones del disco han sido abiertas*/***

373 int bd_invalidated;
374 struct gendisk * bd_disk; ***/*Puntero a la estructura gendisk del disco relacionado con este dispositivo de bloque.*/***

375 struct list_head bd_list; ***/*lista de inodes de fichero de dispositivos de bloque abiertos*/***
376 struct backing_dev_info *bd_inode_backing_dev_info;

383 unsigned long bd_private;
384 };

I/O.5.7 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 foo
        struct gendisk *gd: //puntero al descriptor gendisk que representa el
                            //disco manejado por el driver
        […]
}foo
```

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;
```

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;
```

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;
```

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);

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)

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.

I/O.5.8 Realizando una petición en la capa de bloque generica

A continuación describiremos los pasos ejecutados por el núcleo 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 núcleo 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.

- 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

I/O.5.9 Manejadores de dispositivos de bloque

El Núcleo abre un fichero de dispositivo de bloque cada vez que un sistema de ficheros es montado sobre un disco o partición, cada vez que la partición swap es activada y cada vez que un usuario ejecuta una llamada al sistema open() sobre un archivo de dispositivo de bloque. En todos los casos, el núcleo ejecuta en esencia las mismas operaciones: busca el descriptor de dispositivo de bloque (posiblemente crea un nuevo descriptor si el dispositivo no está en uso aún), e inicializa los métodos para la transferencia de datos.

Cuando un fichero de dispositivo de bloque se abre por la petición de una operación, su f_op es redireccionado a la variable def_blk_fops. Esta variable apunta a una tabla que contiene todas las funciones asociadas a un dispositivo de bloque. Esa tabla se encuentra en el fichero *fs/block_dev*.c y su estructura es la siguiente.

```
1171struct file_operations {
1172        struct module *owner;
1173        loff_t (*llseek) (struct file *, loff_t, int);
1174        ssize_t (*read) (struct file *, char __user *, size_t, loff_t *);
1175        ssize_t (*write) (struct file *, const char __user *, size_t, loff_t *);
1176        ssize_t (*aio_read) (struct kiocb *, const struct iovec *, unsigned long, loff_t);
1177        ssize_t (*aio_write) (struct kiocb *, const struct iovec *, unsigned long, loff_t);
1178        int (*readdir) (struct file *, void *, filldir_t);
1179        unsigned int (*poll) (struct file *, struct poll_table_struct *);
1180        int (*ioctl) (struct inode *, struct file *, unsigned int, unsigned long);
1181        long (*unlocked_ioctl) (struct file *, unsigned int, unsigned long);
1182        long (*compat_ioctl) (struct file *, unsigned int, unsigned long);
1183        int (*mmap) (struct file *, struct vm_area_struct *);
1184        int (*open) (struct inode *, struct file *);
1185        int (*flush) (struct file *, fl_owner_t id);
1186        int (*release) (struct inode *, struct file *);
1187        int (*fsync) (struct file *, struct dentry *, int datasync);
1188        int (*aio_fsync) (struct kiocb *, int datasync);
1189        int (*fasync) (int, struct file *, int);
1190        int (*lock) (struct file *, int, struct file_lock *);
1191        ssize_t (*sendpage) (struct file *, struct page *, int, size_t, loff_t *, int);
1192        unsigned long (*get_unmapped_area)(struct file *, unsigned long, unsigned long, unsigned
long, unsigned long);
1193        int (*check_flags)(int);
1194        int (*dir_notify)(struct file *filp, unsigned long arg);
1195        int (*flock) (struct file *, int, struct file_lock *);
1196        ssize_t (*splice_write)(struct pipe_inode_info *, struct file *, loff_t *, size_t, unsigned int);
1197        ssize_t (*splice_read)(struct file *, loff_t *, struct pipe_inode_info *, size_t, unsigned int);
1198        int (*setlease)(struct file *, long, struct file_lock **);
1199};
```

Existe unas operaciones cuya función es chequear si el driver de un dispositivo de bloque ya está en uso, estas funciones es: bd_acquire

bd_acquire***(operaciones):***
Comprueba que el fichero del dispositivo de bloque ya está abierto, si es así el campo inode -> i_bdev apunta al descriptor de dispositivo de bloque.). Si el fichero ya está abierto se aumenta el contador del descriptor del dispositivo inode ->i_bdev-> bd_count y retorna
Busca el driver del dispositivo en la tabla hash usando el minor y el major número (inode_rdev). De no estar, creará un nuevo block_device y un nuevo inode para el dispositivo y luego inserta el nuevo descriptor en la tabla hash.
Guardará la dirección del driver del dispositivo en inode->i_bdev
Añade el inode a la lista de inodes del descriptor del driver.

```
static struct block_device *bd_acquire(struct inode *inode)
 425{
 426        struct block_device *bdev; ; /*bdev apunta al descriptor del
                                                    dispositivo*/
 427
 428        spin_lock(&bdev_lock);
 429        bdev = inode->i_bdev;
 430        if (bdev) {
 431                atomic_inc(&bdev->bd_inode->i_count);
 432                spin_unlock(&bdev_lock);
 433                return bdev;
 434        }
 435        spin_unlock(&bdev_lock);
 436
 437        bdev = bdget(inode->i_rdev); );/*Busca el driver en la lista
                                          utilizando el número major y minor*/
 438        if (bdev) {
 439                spin_lock(&bdev_lock);
 440                if (!inode->i_bdev) {

 447                        atomic_inc(&bdev->bd_inode->i_count);
 448                        inode->i_bdev = bdev; ;/*almacena la dirección del
                                          descriptor de dispositivo de bloque*/

 449                        inode->i_mapping = bdev->bd_inode->i_mapping;
 450                        list_add(&inode->i_devices, &bdev->bd_inodes);
                                          /*Añade el inode a la lista inodes de
                                          descriptores de dispositivos*/
 451                }
 452                spin_unlock(&bdev_lock);
 453        }
 454        return bdev;
 455}
```

Luego la función blkdev_open***()*** invoca a ***do_open ()*** que realiza los pasos que veremos a continuación:

```
static int blkdev_open(struct inode * inode, struct file * filp)
1049{
1050        struct block_device *bdev;
1051        int res;
1052
1053        /*
1054         * Preserve backwards compatibility and allow large file access
1055         * even if userspace doesn't ask for it explicitly. Some mkfs
1056         * binary needs it. We might want to drop this workaround
```

```
1057        * during an unstable branch.
1058        */
1059       filp->f_flags |= O_LARGEFILE;
1060
1061       bdev = bd_acquire(inode);
1062       if (bdev == NULL)
1063            return -ENOMEM;
1064
1065       res = do_open(bdev, filp, 0);//invocacion de do_open
1066       if (res)
1067            return res;
1068
1069       if (!(filp->f_flags & O_EXCL) )
1070            return 0;
1071
1072       if (!(res = bd_claim(bdev, filp)))
1073            return 0;
1074
1075       blkdev_put(bdev);
1076       return res;
1077}
1078
```

Do_open() (operaciones):

1. Si el campo bd_op del descriptor del driver esté a NULL, lo inicializa del elemento de la tabla blkdevs correspondiente al major número del fichero del dispositivo.
2. Llamar al método bd->open del descriptor del driver bd_op->open si existe
3. Incrementa el contador de los descritores de drivers de dispositivos abiertos
4. Inicializa los campos i_size , i_blkbits del inode (bd_inode)
5. Se determina el tamaño del dispositivo de bloque correspondiente al fichero de dispositivos y se almacena en blk_size, que es un vector global

I/O.5.10 Manejadores de peticiones a bajo nivel

Este nivel está implementado por una estrategia de rutina la cual interactúa con el dispositivo de bloque físico para satisfacer las peticiones almacenadas en la cola. Los driver de bajo nivel deben manejar todas las peticiones en la cola y terminar cuando la cola está vacía.

Muchos de los dispositivos de bajo nivel siguen la siguiente estrategia:

- Se maneja la primera petición de la cola creando el controlador del bloque hasta aterrizar en una interrupción cuando la transferencia de datos se completa. En este momento la rutina de estrategia termina.
- Cuando se aterriza en la función manejadora de la interrupción, dentro del manejador del dispositivo de bloque, se activa un bottom half. El manejador del bottom half quita la petición de la cola y ejecuta la rutina de estrategia para servir la siguiente petición en la cola.

Los drivers de bajo nivel se pueden clasificarse en:

- Driver que atienden cada bloque de la cola por separado
- Drivers que atienden varios bloque juntos en una petición

Operación de bloque y páginas de entrada salida

En este apartado describiremos dos clases fundamentales de transferencia de datos de E/S para dispositivos de bloques.

- **Operaciones de E/S para bloque**.- las operaciones de E/S transfieren un bloque de datos simple
- **Operaciones de E/S para página.-** las operaciones de E/S transfieren tantos bloques de datos como se necesitan para rellenar una pagina

Las operaciones de entrada salida para bloque son usadas cuando el núcleo lee o escribe un bloque simple en un sistema de archivos mientras que las de página se usan principalmente para leer y escribir ficheros, para acceder a ficheros a través del mapeo de memoria y para el swapping.

Operaciones de E/S para bloque
La función bread() lee un bloque simple de un dispositivo de bloque y lo almacena en un buffer. Esta función recibe como parámetros el identificador de dispositivo, el número de bloque y el tamaño del bloque devolviendo un puntero a la cabeza del buffer que contiene el bloque. Esta estructura la encontramos en el fichero ***inode.c***

```
914 struct buffer_head *ext3_bread(handle_t *handle, struct inode * inode,
915                                 int block, int create, int *err)
916 {
917         struct buffer_head * bh;
918
919         bh = ext3_getblk(handle, inode, block, create, err); /*Función que busca el bloque en un software cache llamado buffer cache. Si no existe esta función almacena un nuevo buffer para el bloque.*/

920         if (!bh)
921                 return bh;
922         if (buffer_uptodate(bh))
923                 return bh;
924         ll_rw_block(READ, 1, &bh); /*Empieza la operación de lectura*/
925          wait_on_buffer(bh);/*espera hasta que la transferencia de datos se complete.Inserta el proceso actual en la b_wait (cola de espera) y suspende el proceso hasta que el buffer es desbloqueado*/

926          if (buffer_uptodate(bh))/*comprueba que el buffer contenga datos válidos, si es así devuelve la dirección de la cabeza del buffer sino devuelve nulo*/
927                 return bh;
928         put_bh(bh);
929         *err = -EIO;
930         return NULL;
931 }
```

I/O.6 Bibliografía

- **Understanding Linux Kernel 3ª edicion.** Daniel P. Bovet, Marco Cesati. O'Reilly Associates INC 2005

- Linux 2.6. The Linux Cross-Reference. **lxr**.linux.no