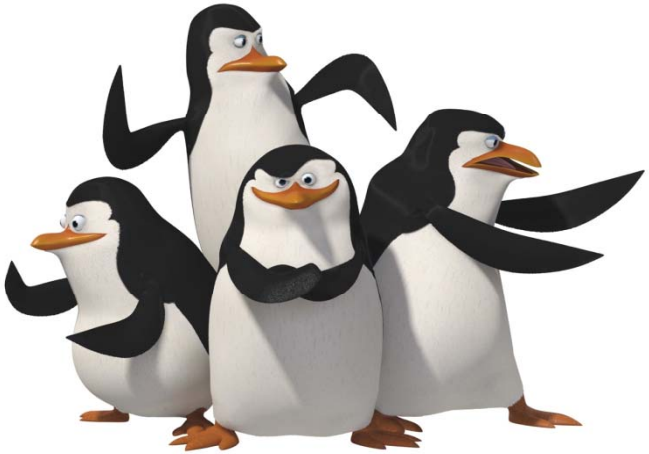
Drivers de Impresora

Jose Miguel Santana Núñez
Sebastián Eleazar Ortega Trujillo

Contenido

- Introducción
- Configuración
- Estructuras de datos
- Operaciones sobre puertos
- Operaciones de impresora



Introducción 1

- Linux dispone de drivers genéricos para impresoras.
- Éstos drivers son muy básicos y deben ser complementados con los propios del fabricante.
- El sistema de impresión de Linux (el sistema lp) es una adaptación del código escrito por los *Regents* de la Universidad de California para la versión *Berkeley Software Distribution* (*BSD*) del sistema operativo UNIX.

Introducción 2

- El sistema operativo accede mediante la relación puerto-interrupción en /dev/lpX , (enlaces simbólicos a las impresoras de la máquina).

- Posee tres características comunes a cualquier dispositivo:
  - Identificador del controlador.
  - Identificador del dispositivo.
  - Modo de funcionamiento: (Modo carácter o bloque)

- **Modo Carácter:** Las Impresoras pertenecen a este tipo de dispositivos no estructurados.
  - **Modo acceso:** Corriente de bytes, se lee y escribe carácter a carácter (byte) de forma secuencial.

Introducción

- Existen dos técnicas para comunicarse con el puerto:
  - **Interrupciones:** Para señalar un evento, el puerto envía una interrupción que es tratada por el gestor. Mediante las IRQ, este método libera trabajo del procesador.
  - **Exploración(polling):** En lugar de utilizar interrupciones, el gestor efectúa bucles de espera comprobando el registro de estado del puerto. Este método permite evitar la producción de numerosas interrupciones y puede resultar más eficaz.

Configuración

- La impresora debe estar conectada al **puerto paralelo.**
- Se puede configurar mediante un comando en el núcleo, que inserta el modulo en el sistema:
  - insmod lp.o parport = 1, none, 2 → Puerto paralelo.
  - insmod lp.o parport = auto → Autodetectar impresoras
  - insmod lp.o reset → Resetea la impresora durante la inicialización.
  - insmod lp.o off → Desactivar el driver de impresora completamente
- Estos comandos son los usados por **modprobe** para cargar módulos en el sistema.

Estructuras de Datos

- Las estructuras de datos que soportan el uso de la impresora se encuentran en */include/linux/lp.h*
- Los puertos paralelos gestionados se definen por descriptores del tipo **lp_struct** que poseen información sobre una impresora.
- Estos descriptores se encuentran localizados en la tabla **lp_table** (Struct lp_struct lp_table[LP_NO])**.**
  - Normalmente se utilizan tablas de tamaño 8. Si se dispone de más de 8 impresoras es necesario incrementar este valor en la macro **(#define LP_NO 8).**

Estructuras de datos – lp_struct

```
134 struct lp_struct {
135         struct pardevice *dev;
136         unsigned long flags;
137         unsigned int chars;
138         unsigned int time;
139         unsigned int wait;
140         char *lp_buffer;
141 #ifdef LP_STATS
142         unsigned int lastcall;
143         unsigned int runchars;
144         struct lp_stats stats;
145 #endif
146         wait_queue_head_t waitq;
147         unsigned int last_error;
148         struct semaphore port_mutex;
149         wait_queue_head_t dataq;
150         long timeout;
151         unsigned int best_mode;
152         unsigned int current_mode;
153         unsigned long bits;
154 };
```

Estructuras de datos – lp_struct

Tipo	Campo	Descripción
Struct pardevice*	**dev**	Estructura que guarda información asociada al dispositivo.
Int	**Flags**	Estado de la impresora conectada.
Unsigned int	Chars	Número de intentos a efectuar para imprimir un carácter.
Unsigned int	Time	Duración de la suspensión para una espera expresada en ciclos de reloj.
Unsigned int	Wait	Número de bucles de espera (de polling) a efectuar antes de que la impresora tenga en cuenta un carácter.
Char *	**Lp_buffer**	Puntero a una memoria intermedia que contiene los caracteres a imprimir.
Unsigned int	Lastcall	Fecha de la última escritura en la impresora.
Unsigned int	**Runchars**	Número de caracteres escritos en la impresora sin provocar suspensión.
Struct lp_stats	Stats	Estadísticas sobre el uso de la impresora.
Struct *wait_queue_head_t	**wait_q**	Cola de espera utilizada para esperar la llegada de una interrupción.
unsigned int	last_error;	Almacena el último error producido en la impresora.
Struct semaphore	**port_mutex**	Semáforo para bloquear el puerto.
wait_queue_head_t	dataq	Cola de espera utilizada para esperar la llegada de los datos.
long	timeout	Tiempo de espera.
unsigned int	best_mode	Código del mejor modo negociado.
unsigned int	**current_mode**	Código del modo actual negociado.
unsigned long	**bits**	Bits que definen el estado del puerto asociado (testeo y limpieza).

Estructuras de datos – lp_fops

- Contiene las operaciones realizables sobre la impresora.
- Sus campos se inicializan con punteros a las funciones de operación de la impresora.

```
662 static struct file_operations lp_fops = {
663         .owner          = THIS_MODULE,
664         .write          = lp_write,
665         .ioctl          = lp_ioctl,
666         .open           = lp_open,
667         .release        = lp_release,
668 #ifdef CONFIG_PARPORT_1284
669         .read           = lp_read,
670 #endif
671 };
```

Estructuras de datos – lp_fops

Campo	Descripción
.owner	Apunta a la estructura module.
.**write**	Operación que se encarga de realizar el envío de los datos a la impresora.
.**ioctl**	Operación que permite consultar y modificar parámetros relacionados con la impresora.
.**open**	Operación que establece la comunicación con la impresora.
.**release**	Operación que permite dejar la impresora desocupada y libre para usar.
.**read**	Operación que se encarga de realizar la obtención de los datos a la impresora.

Operaciones sobre los puertos

- **lp_claim_parport_or_block**
  - Este procedimiento se usa para demandar el puerto paralelo a menos que esté ocupado.

```
168 static void lp_claim_parport_or_block(struct lp_struct *this_lp)
169 {
            //Se testea y se marca el bit de reclamado de forma atómica
170         if (!test_and_set_bit(LP_PARPORT_CLAIMED, &this_lp->bits)) {
171             parport_claim_or_block (this_lp->dev);
172         }
173 }
```

- **Parport_claim_or_block**
  - Función definida en /linux/drivers/parport/share.c que pone a toma el puerto paralelo o pone a dormir el proceso en wait_q en espera de una señal de la impresora.

Operaciones sobre los puertos

- **lp_release_parport**
  - Este procedimiento se usa para liberar el puerto que ha sido previamente tomado por la función anterior.

```
176 static void lp_release_parport(struct lp_struct *this_lp)
177 {
            //Se testea y se borra el bit de forma atómica
178         if (test_and_clear_bit(LP_PARPORT_CLAIMED, &this_lp->bits)) {
179                 parport_release (this_lp->dev);
180         }
181 }
```

- **Parport_release**
  - Función definida en /linux/drivers/parport/share.c que termina el acceso a la impresora. Nunca falla.

Operaciones sobre los puertos

- **lp_preempt**
  - Función para dar prioridad a un dispositivo, activando el bit LP_PREEMPT_REQUEST.

```
185 static int lp_preempt(void *handle)
186 {
            //Se le pasa un punter al lp_struct del puerto
187         struct lp_struct *this_lp = (struct lp_struct *)handle;
188         set_bit(LP_PREEMPT_REQUEST, &this_lp->bits);
189         return (1);
190 }
```

Modos de Transferencia de Puerto Paralelo

- **Definidos en el estándar para puertos paralelos IEEE 1284:**
  - **Modo de Compatibilidad** (modo estándar o "Centronics")
  - **Modo Nibble**: 4 bits a la vez usando las líneas de estado (Status) para datos (Hewlett Packard Bi-tronics)
  - **Modo de Octeto (Byte Mode)**: 8 bits a la vez usando las líneas de datos, a veces nombrado como puerto bidireccional
  - **EPP (Enhanced Parallel Port)**: Puerto Paralelo Extendido, usado principalmente para periféricos que no son impresoras, como CD-ROM, Adaptadores de Red, etc.
  - **ECP (Extended Capability Port)**: Puerto de Capacidades Extendidas, usado principalmente por impresoras recientes y scanners.

Operaciones sobre los puertos

- **lp_negotiate**
  - Esta función se usa para negociar con el puerto el modo de transferencia. Si no se consigue, asigna el modo compat de forma predeterminada. La función **parport_negociate** establece el modo de transferencia que debe ser necesariamente uno de los establecidos por IEEE. Si consigue establecer el modo con éxito devolverá 0, si no cumple las especificaciones IEEE devuelve -1 y si el dispositivo rechaza la conexión devuelve 1.

```
197 static int lp_negotiate(struct parport * port, int mode)
198 {
        //Se negocia el modo de operación del puerto pasado por parámetro,
199     if (parport_negotiate (port, mode) != 0) {
         //si no lo consigue se  asigna el modo compat (transferencia bidireccional)
200         mode = IEEE1284_MODE_COMPAT;
201         parport_negotiate (port, mode);
202     }
203
204     return (mode);
205 }
```

Operaciones sobre los puertos

- **lp_reset**
  - Función para resetear la impresora. La macro **w_ctr** se encarga de enviar las señales a través del puerto para reiniciar la impresora. Con **r_str** leemos el estado actual del dispositivo.

```
207 static int lp_reset(int minor)
208 {
209         int retval;
            //Demanda el puerto paralelo o bloque a menos de que esté ocupado.
210         lp_claim_parport_or_block (&lp_table[minor]);
            //Inicializaciones:
211         w_ctr(minor, LP_PSELECP); //Macro de escritura de control a través del puerto
212         udelay (LP_DELAY);
213         w_ctr(minor, LP_PSELECP | LP_PINITP);
214         retval = r_str(minor);    //Macro para leer el macro a través del puerto
            //Libera el puerto:
215         lp_release_parport (&lp_table[minor]);
216         return retval;
217 }
```

Operaciones sobre los puertos

- **lp_error**
  - Función invocada en caso de error de la impresora. En caso de que el dispositivo esté llevando a cabo una tarea que no se puede interrumpir, pone la tarea en espera mientras el dispositivo se recupera del error.

```
static void lp_error (int minor)
{
        DEFINE_WAIT(wait);
        int polling;

        if (LP_F(minor) & LP_ABORT) return;

        //Si estamos en modo polling
        polling = lp_table[minor].dev->port->irq == PARPORT_IRQ_NONE;

        //Si estamos en modo polling se libera el puerto
        if (polling) lp_release_parport (&lp_table[minor]);
```

Operaciones sobre los puertos

```
        //Se prepara para dormir el proceso
        prepare_to_wait(&lp_table[minor].waitq, &wait, TASK_INTERRUPTIBLE);
        schedule_timeout(LP_TIMEOUT_POLLED);
        finish_wait(&lp_table[minor].waitq, &wait);

        //Al acabar si estamos en modo polling se vuelve a pedir el puerto
        if (polling) lp_claim_parport_or_block (&lp_table[minor]);
        //Si estamos en modo interrupt se cede el puerto temporalmente
        else parport_yield_blocking (lp_table[minor].dev);
}
```

Operaciones sobre los puertos

- **lp_check_status**
  - Comprueba el estado de la impresora a partir de una serie de posibilidades.

```
234 static int lp_check_status(int minor)
235 {       //Inicializaciones:
236         int error = 0;
237         unsigned int last = lp_table[minor].last_error;
238         unsigned char status = r_str(minor);
239         if ((status & LP_PERRORP) && !(LP_F(minor) & LP_CAREFUL))
240                  //No hay error
241                 last = 0;
                   //Si hay errores, se comprueba que tipo de error hay y se marca en error el
                   //tipo correspondiente.
                   //FALTA DE PAPEL:
242         else if ((status & LP_POUTPA)) {
243                 if (last != LP_POUTPA) {
244                         last = LP_POUTPA;
245                         printk(KERN_INFO "lp%d out of paper\n", minor);
246                 }
247                 error = -ENOSPC;
```

Operaciones sobre los puertos

```
                //DESCONECTADA:
248         } else if (!(status & LP_PSELECD)) {
249                 if (last != LP_PSELECD) {
250                         last = LP_PSELECD;
251                         printk(KERN_INFO "lp%d off-line\n", minor);
252                 }
253                 error = -EIO;
                //OTRO TIPO DE ERROR:
254         } else if (!(status & LP_PERRORP)) {
255                 if (last != LP_PERRORP) {
256                         last = LP_PERRORP;
257                         printk(KERN_INFO "lp%d on fire\n", minor);
258                 }
259                 error = -EIO;
260         } else {
                //Si LP_CAREFUL está activado y no se han encontrado errores:
261                 last = 0;
263         }
264          //Se almacena el tipo de error producido:
265         lp_table[minor].last_error = last;
266         //Se llama a  lp_error para que muestre el mensaje de error producido:
267         if (last != 0)
268                 lp_error(minor);
270         return error;
271 }
```

Operaciones de impresora

lp_write

Esta función se ocupa de enviar los datos a la impresora.

Los datos se dividen en bloques, cuyo tamaño no podrá exceder el del buffer.

La función toma el puerto y enviará los datos, mientras hayan datos para enviar y no hayan errores. Al acabar, liberará el puerto.

Parámetros usados por la función:

struct file *file -> El fichero para el cual se llamó esta rutina.

const char __user * buf -> Los datos en cuestión.

size_t count -> La longitud de los datos.

loff_t *ppos -> Sin uso aparente desde largo tiempo atrás.

Operaciones de impresora

```
292static ssize_t lp_write(struct file * file, const char __user * buf,
293                  size_t count, loff_t *ppos)
294{    /* Inicializando algunos datos para su uso más adelante */
295      unsigned int minor = iminor(file->f_path.dentry->d_inode);  //cogerá  el dispositivo de menor número
al que apunta el inodo del archivo.
296      struct parport *port = lp_table[minor].dev->port;
297      char *kbuf = lp_table[minor].lp_buffer;
298      ssize_t retv = 0;
299      ssize_t written;
300      size_t copy_size = count;
301      int nonblock = ((file->f_flags & O_NONBLOCK) ||
302                  (LP_F(minor) & LP_ABORT));
303
304#ifdef LP_STATS
305      if (time_after(jiffies, lp_table[minor].lastcall + LP_TIME(minor)))
306           lp_table[minor].runchars = 0;
307
308      lp_table[minor].lastcall = jiffies;
309#endif
```

Operaciones de impresora

```
311     /* Controlamos que sólo se copia un tamaño que quepa en el buffer */
312     if (copy_size > LP_BUFFER_SIZE)
313             copy_size = LP_BUFFER_SIZE;
314 /* Cerrojo */
315     if (mutex_lock_interruptible(&lp_table[minor].port_mutex))
316             return -EINTR;
317 /* Copiamos los datos del usuario */
318     if (copy_from_user (kbuf, buf, copy_size)) {
319             retv = -EFAULT;
320             goto out_unlock;
321     }
322
323     /* Claim Parport or sleep until it becomes available
324      */ /* Reclamar Parport o dormir hasta que esté disponible */
325     lp_claim_parport_or_block (&lp_table[minor]);
326     /* El modo actual ha de ser el más apropiado..... */
327     lp_table[minor].current_mode = lp_negotiate (port,
328                                     lp_table[minor].best_mode);
```

Operaciones de impresora

```
330     parport_set_timeout (lp_table[minor].dev,
331                          (nonblock ? PARPORT_INACTIVITY_O_NONBLOCK
332                           : lp_table[minor].timeout));
333     // Mientras el tamaño de los datos sea mayor que cero, vamos a escribir
334     if ((retv = lp_wait_ready (minor, nonblock)) == 0)
335     do {
336         /* Pasando  los datos a través del puerto */
337         written = parport_write (port, kbuf, copy_size);
338         if (written > 0) {
339             copy_size -= written;
340             count -= written;
341             buf  += written;
342             retv += written;
343         }
345         if (signal_pending (current)) {
346             if (retv == 0)
347                 retv = -EINTR;
349             break;
350         }
```

Operaciones de impresora

```
352         if (copy_size > 0) {
353             /* Si la escritura no se completó, algo ha fallado, buscamos el causante */
354             int error;
356             parport_negotiate (lp_table[minor].dev->port,
357                                IEEE1284_MODE_COMPAT);
358             lp_table[minor].current_mode = IEEE1284_MODE_COMPAT;
359
360             error = lp_wait_ready (minor, nonblock);
361             /* Caso de error salimos del bucle */
362             if (error) {
363                 if (retv == 0)
364                     retv = error;
365                 break;
366             } else if (nonblock) {
367                 if (retv == 0)
368                     retv = -EAGAIN;
369                 break;
370             }
```

Operaciones de impresora

```
// Si no fue un error, restablecemos el modo más apropiado para el dispositivo.
372                 parport_yield_blocking (lp_table[minor].dev);
 373                 lp_table[minor].current_mode
 374                   = lp_negotiate (port,
 375                                   lp_table[minor].best_mode);
377             } else if (need_resched())
 378                 schedule ();
 379             // Si quedaran datos por escribir, se vuelve a rellenar el buffer de igual forma que antes
 380              if (count) {
 381                 copy_size = count;
 382                 if (copy_size > LP_BUFFER_SIZE)
 383                     copy_size = LP_BUFFER_SIZE;
385                 if (copy_from_user(kbuf, buf, copy_size)) {
 386                     if (retv == 0)
 387                         retv = -EFAULT;
 388                     break;
 389                 }
 390             }
 391         } while (count > 0);
```

Operaciones de impresora

```
/* Hora de salir: comprobamos el estado , volvemos a cambiar el modo del dispositivo y
liberamos el puerto paralelo */
393       if (test_and_clear_bit(LP_PREEMPT_REQUEST,
 394                              &lp_table[minor].bits)) {
 395               printk(KERN_INFO "lp%d releasing parport\n", minor);
 396               parport_negotiate (lp_table[minor].dev->port,
 397                                  IEEE1284_MODE_COMPAT);
 398               lp_table[minor].current_mode = IEEE1284_MODE_COMPAT;
 399               lp_release_parport (&lp_table[minor]);
 400       }
401out_unlock:
// y cómo no, liberar el cerrojo antes de salir...
 402       mutex_unlock(&lp_table[minor].port_mutex);
 403
 404       return retv;
 405}
```

Operaciones de impresora

lp_ioctl

Esta función nos da la posibilidad de consultar y modificar los parámetros relacionados con la impresora.

Parámetros usados por la función:

struct inode *inode -> El inodo correspondiente al dispositivo

struct file *file -> El fichero para el cual se llamó esta rutina.

unsigned int cmd -> Indica el comando deseado

unsigned long arg -> Permite indicar argumentos.

Operaciones de impresora

```
560static int lp_ioctl(struct inode *inode, struct file *file,
561                     unsigned int cmd, unsigned long arg)
562{
563     unsigned int minor = iminor(inode);
564     int status;
565     int retval = 0;
566     void __user *argp = (void __user *)arg;
567
568#ifdef LP_DEBUG
569     printk(KERN_DEBUG "lp%d ioctl, cmd: 0x%x, arg: 0x%lx\n", minor, cmd, arg);
570#endif
571     if (minor >= LP_NO)
572          return -ENODEV;
573     if ((LP_F(minor) & LP_EXIST) == 0)
574          return -ENODEV;
575     switch ( cmd ) {
576          struct timeval par_timeout;
577          long to_jiffies;
```

Operaciones de impresora

```
579         case LPTIME: // Fija cantidad de tiempo que el driver esperará por el dispositivo.
580             LP_TIME(minor) = arg * HZ/100;
581             break;
582         case LPCHAR: // Fija cantidad de tiempo máxima para cada carácter
583             LP_CHAR(minor) = arg;
584             break;
585         case LPABORT:
586             if (arg) // Si el argumento no es 0, entonces causa un error, si no, causa un reintento
587                 LP_F(minor) |= LP_ABORT;
588             else
589                 LP_F(minor) &= ~LP_ABORT;
590             break;
591         case LPABORTOPEN:
592             if (arg) // Si arg==0, se ignorarán errores en open(), si no, se abortará si hubieran errores.
593                 LP_F(minor) |= LP_ABORTOPEN;
594             else
595                 LP_F(minor) &= ~LP_ABORTOPEN;
596             break;
```

Operaciones de impresora

```
597         case LPCAREFUL:
598             if (arg) // Si no es 0, el driver se asegurará de que todo flag sea comprobado al escribir.
599                 LP_F(minor) |= LP_CAREFUL;
600             else // y si lo es, pues se ignora.
601                 LP_F(minor) &= ~LP_CAREFUL;
602             break;
603         case LPWAIT: // Cantidad de tiempo que el driver espera antes y después de un “strobe”
604             LP_WAIT(minor) = arg;
605             break;
606         case LPSETIRQ:
607             return -EINVAL;
608             break;
609         case LPGETIRQ: // Obtiene el número de IRQ.
610             if (copy_to_user(argp, &LP_IRQ(minor),
611                         sizeof(int)))
612                 return -EFAULT;
613             break;
```

Operaciones de impresora

```
614             case LPGETSTATUS:// Obtendrá el status de la impresora
615                     lp_claim_parport_or_block (&lp_table[minor]);
616                     status = r_str(minor);
617                     lp_release_parport (&lp_table[minor]);
619                     if (copy_to_user(argp, &status, sizeof(int)))
620                             return -EFAULT;
621                     break;
622             case LPRESET:  //Resetea la impresora
623                     lp_reset(minor);
624                     break;
625#ifdef LP_STATS
626             case LPGETSTATS: // Estadísticas actuales del driver
627                     if (copy_to_user(argp, &LP_STAT(minor),
628                                      sizeof(struct lp_stats)))
629                             return -EFAULT;
630                     if (capable(CAP_SYS_ADMIN))
631                             memset(&LP_STAT(minor), 0,
632                                             sizeof(struct lp_stats));
633                     break;
634#endif
```

Operaciones de impresora

```
635         case LPGETFLAGS: // Obtiene los flags
636             status = LP_F(minor);
637             if (copy_to_user(argp, &status, sizeof(int)))
638                 return -EFAULT;
639             break;
641         case LPSETTIMEOUT: // Fijar el parámetro timeout en la estructura lpstruct del dispositivo actual
642             if (copy_from_user (&par_timeout, argp,
643                         sizeof (struct timeval))) {
644                 return -EFAULT;
645             }
646           /* Y ahora convertir  a jiffies (unidad de tiempo)  y emplazarlo en lp_table */
647             if ((par_timeout.tv_sec < 0) ||
648                (par_timeout.tv_usec < 0)) {
649                 return -EINVAL;
650             }
651             to_jiffies = DIV_ROUND_UP(par_timeout.tv_usec, 1000000/HZ);
652             to_jiffies += par_timeout.tv_sec * (long) HZ;
```

Operaciones de impresora

```
653                 if (to_jiffies <= 0) {
654                         return -EINVAL;
655                 }
656                 lp_table[minor].timeout = to_jiffies;
657                 break;
658
659         default: // Si el caso no está contemplado, devolvemos -EINVAL
660                 retval = -EINVAL;
661     }
662     return retval;
663}
```

Operaciones de impresora

lp_open

Esta operación abre una comunicación de datos con la impresora a través de un puerto paralelo.

Es imprescindible efectuar esta operación antes que cualquiera otra que se desee realizar.

Parámetros usados por la función:

struct inode *inode -> El inodo correspondiente al dispositivo

struct file *file -> El fichero para el cual se llamó esta rutina.

Operaciones de impresora

```
489static int lp_open(struct inode * inode, struct file * file)
 490{
 491       unsigned int minor = iminor(inode);
 492    /* Pilla el dispositivo menor y pregunta si está en el rango de impresoras, si la impresora existe y si la impresora no está
ocupada, devolviendo errores si fallan los test */
 493       if (minor >= LP_NO)
 494             return -ENXIO;
 495       if ((LP_F(minor) & LP_EXIST) == 0)
 496             return -ENXIO;
 497       if (test_and_set_bit(LP_BUSY_BIT_POS, &LP_F(minor)))
 498             return -EBUSY;
 499  // Caso de que ABORTOPEN activo y existan errores tipo falta de papel, los controla
505       if ((LP_F(minor) & LP_ABORTOPEN) && !(file->f_flags & O_NONBLOCK)) {
 506             int status;
 507             lp_claim_parport_or_block (&lp_table[minor]);
 508             status = r_str(minor);
 509             lp_release_parport (&lp_table[minor]);
 510             if (status & LP_POUTPA) {
 511                   printk(KERN_INFO "lp%d out of paper\n", minor);
 512                   LP_F(minor) &= ~LP_BUSY;
```

Operaciones de impresora

```
513                 return -ENOSPC;
514             } else if (!(status & LP_PSELECD)) {
515                 printk(KERN_INFO "lp%d off-line\n", minor);
516                 LP_F(minor) &= ~LP_BUSY;
517                 return -EIO;
518             } else if (!(status & LP_PERRORP)) {
519                 printk(KERN_ERR "lp%d printer error\n", minor);
520                 LP_F(minor) &= ~LP_BUSY;
521                 return -EIO;
522             }
523         }
524         lp_table[minor].lp_buffer = kmalloc(LP_BUFFER_SIZE, GFP_KERNEL); /* Crea memoria para el búfer. Si no hubiera
memoria disponible, canta error y cierra */
525         if (!lp_table[minor].lp_buffer) {
526             LP_F(minor) &= ~LP_BUSY;
527             return -ENOMEM;
528         }
529         /* Comprueba si el periférico soporta el modo ECP y en tal caso lo establece como mejor modo. Caso contrario, el
mejor modo será el modo de compatibilidad de IEEE1284 */
530         lp_claim_parport_or_block (&lp_table[minor]);
531         if ( (lp_table[minor].dev->port->modes & PARPORT_MODE_ECP) &&
532             !parport_negotiate (lp_table[minor].dev->port,
533                                 IEEE1284_MODE_ECP)) {
534             printk (KERN_INFO "lp%d: ECP mode\n", minor);
535             lp_table[minor].best_mode = IEEE1284_MODE_ECP;
536         } else {
537             lp_table[minor].best_mode = IEEE1284_MODE_COMPAT;
538         }
```

Operaciones de impresora

```
539     /* Deja el modo actual del periférico establecido en modo de compatibilidad y retorna*/
540     parport_negotiate (lp_table[minor].dev->port, IEEE1284_MODE_COMPAT);
541     lp_release_parport (&lp_table[minor]);
542     lp_table[minor].current_mode = IEEE1284_MODE_COMPAT;
543     return 0;
544}
```

Operaciones de impresora

lp_release

Esta función liberará la impresora para ser usada más tarde.

Parámetros usados por la función:

struct inode *inode -> El inodo correspondiente al dispositivo

struct file *file -> El fichero para el cual se llamó esta rutina.

Operaciones de impresora

```
546static int lp_release(struct inode * inode, struct file * file)
547{
548     unsigned int minor = iminor(inode);
549     /* Esta función pone el modo actual de la impresora en modo de compatibilidad, liberará el
espacio en memoria que fue adquirido para el búfer en la función lp_open() y finalmente cambia los flags
para marcar que la impresora ya no está ocupada */
550     lp_claim_parport_or_block (&lp_table[minor]);
551     parport_negotiate (lp_table[minor].dev->port, IEEE1284_MODE_COMPAT);
552     lp_table[minor].current_mode = IEEE1284_MODE_COMPAT;
553     lp_release_parport (&lp_table[minor]);
554     kfree(lp_table[minor].lp_buffer);
555     lp_table[minor].lp_buffer = NULL;
556     LP_F(minor) &= ~LP_BUSY;
557     return 0;
558}
```

Operaciones de impresora

lp_read

Esta función tendrá como objetivo obtener los datos de la impresora.

De igual forma que en la función lp_write, es necesario leer en bloques que no superen el tamaño del buffer, de hecho, la estructura de las dos funciones es muy parecida.

Parámetros usados por la función:

struct file *file -> El fichero para el cual se llamó esta rutina.

const char __user * buf ->Los datos en cuestión.

size_t count -> La longitud de los datos.

loff_t *ppos -> Sin uso aparente desde largo tiempo atrás.

Operaciones de impresora

```
410static ssize_t lp_read(struct file * file, char __user * buf,
411                     size_t count, loff_t *ppos)
412{   /* Detección del dispositivo de trabajo  y algunas inicializaciones */
413       DEFINE_WAIT(wait);
414       unsigned int minor=iminor(file->f_path.dentry->d_inode);
415       struct parport *port = lp_table[minor].dev->port;
416       ssize_t retval = 0;
417       char *kbuf = lp_table[minor].lp_buffer;
418       int nonblock = ((file->f_flags & O_NONBLOCK) ||
419                       (LP_F(minor) & LP_ABORT));
420 /* Nos aseguramos de copiar en trozos no mayores que el tamaño del buffer */
421       if (count > LP_BUFFER_SIZE)
422               count = LP_BUFFER_SIZE;
423 /* Tomamos el control del cerrojo y del puerto paralelo, o bien dormimos hasta que el puerto esté libre */
424       if (mutex_lock_interruptible(&lp_table[minor].port_mutex))
425               return -EINTR;
427       lp_claim_parport_or_block (&lp_table[minor]);
```

Operaciones de impresora

```
429     parport_set_timeout (lp_table[minor].dev,
430                          (nonblock ? PARPORT_INACTIVITY_O_NONBLOCK
431                           : lp_table[minor].timeout));
432    /* Negociamos el modo de operación del puerto a modo de compatibilidad. Si el modo disponible fuera el NIBBLE,
salimos */
433     parport_negotiate (lp_table[minor].dev->port, IEEE1284_MODE_COMPAT);
434     if (parport_negotiate (lp_table[minor].dev->port,
435                            IEEE1284_MODE_NIBBLE)) {
436         retval = -EIO;
437         goto out;
438     }
439    // Leemos los datos...
440     while (retval == 0) {
441         retval = parport_read (port, kbuf, count);
443         if (retval > 0)
444             break;
446        if (nonblock) {
447             retval = -EAGAIN;
448             break;
449         }
```

Operaciones de impresora

```
452 /* Controlamos posibles errores y esperamos por el siguiente dato a leer */
453             if (lp_table[minor].dev->port->irq == PARPORT_IRQ_NONE) {
454                 parport_negotiate (lp_table[minor].dev->port,
455                                    IEEE1284_MODE_COMPAT);
456                 lp_error (minor);
457                 if (parport_negotiate (lp_table[minor].dev->port,
458                                        IEEE1284_MODE_NIBBLE)) {
459                     retval = -EIO;
460                     goto out;
461                 }
462             } else {
463                 prepare_to_wait(&lp_table[minor].waitq, &wait, TASK_INTERRUPTIBLE);
464                 schedule_timeout(LP_TIMEOUT_POLLED); //Esperando por el dato mediante polling.
465                 finish_wait(&lp_table[minor].waitq, &wait);
466             }
468             if (signal_pending (current)) {
469                 retval = -ERESTARTSYS;
470                 break;
471             }
```

Operaciones de impresora

```
473             cond_resched ();
474     }
/* Al terminar la lectura de datos, renegociamos el modo de operación y liberamos el puerto */
475     parport_negotiate (lp_table[minor].dev->port, IEEE1284_MODE_COMPAT);
476 out:
477     lp_release_parport (&lp_table[minor]);
478 /* Copiamos los datos leídos para que el usuario pueda disponer de ellos */
479     if (retval > 0 && copy_to_user (buf, kbuf, retval))
480             retval = -EFAULT;
481 /* y liberamos el cerrojo antes de retornar*/
482     mutex_unlock(&lp_table[minor].port_mutex);
483
484     return retval;
485}
```

Operaciones de impresora

lp_init

Esta función inicializa los parámetros del driver y es llamada únicamente cuando se inicia el sistema o bien cuando se cargue el gestor en forma de módulo.

A esta función no se le pasa ningún parámetro de entrada.

Operaciones de impresora

```
869static int __init lp_init (void)
870{
871       int i, err = 0;
872   /* Miramos que el puerto esté activo*/
873       if (parport_nr[0] == LP_PARPORT_OFF)
874               return 0;
875 /* Inicializamos una a una las LP_NO estructuras lp_struct que existan (por defecto 8) */
876       for (i = 0; i < LP_NO; i++) {
877               lp_table[i].dev = NULL;
878               lp_table[i].flags = 0;
879               lp_table[i].chars = LP_INIT_CHAR;
880               lp_table[i].time = LP_INIT_TIME;
881               lp_table[i].wait = LP_INIT_WAIT;
882               lp_table[i].lp_buffer = NULL;
883#ifdef LP_STATS
884               lp_table[i].lastcall = 0;
885               lp_table[i].runchars = 0;
886               memset (&lp_table[i].stats, 0, sizeof (struct lp_stats));
887#endif
```

Operaciones de impresora

```
888             lp_table[i].last_error = 0;
889             init_waitqueue_head (&lp_table[i].waitq);
890             init_waitqueue_head (&lp_table[i].dataq);
891             mutex_init(&lp_table[i].port_mutex);
892             lp_table[i].timeout = 10 * HZ;
893         }
894 /* Registramos el gestor con los puntos de llamada. Se observa que le pasamos un parámetro lp_fops que es el que
indicará qué operaciones podemos hacer con el archivo */
895         if (register_chrdev (LP_MAJOR, "lp", &lp_fops)) {
896             printk (KERN_ERR "lp: unable to get major %d\n", LP_MAJOR);
897             return -EIO;
898         }
899 /* Creamos la clase. Si hubiera algún problema salimos */
900         lp_class = class_create(THIS_MODULE, "printer");
901         if (IS_ERR(lp_class)) {
902             err = PTR_ERR(lp_class);
903             goto out_reg;
904         }
```

Operaciones de impresora

```
/* Registramos el driver. Si no fuera posible, saltará un error y destruiríamos la clase creada antes */
•       if (parport_register_driver (&lp_driver)) {
907             printk (KERN_ERR "lp: unable to register with parport\n");
908             err = -EIO;
909             goto out_class;
910     }
912     if (!lp_count) { /* Si no se encontró el dispositivo, saltará un error */
913             printk (KERN_INFO "lp: driver loaded but no devices found\n");
914#ifndef CONFIG_PARPORT_1284
915             if (parport_nr[0] == LP_PARPORT_AUTO)
916                     printk (KERN_INFO "lp: (is IEEE 1284 support enabled?)\n");
917#endif
918     }
920     return 0;
922out_class:
923     class_destroy(lp_class);
924out_reg:
925     unregister_chrdev(LP_MAJOR, "lp");
926     return err;
927}
```

Bibliografía utilizada

Archivos fuente disponibles en:

http://lxr.linux.no/linux+v2.6.25.4/drivers/char/lp.c

http://lxr.linux.no/linux/include/linux/lp.h

Libro recomendado:

Programación Linux 2.0 – API del sistema y funcionamiento del núcleo (Remy Card) → **Disponible en la carpeta de la asignatura**

Algo más de información en:

http://www.leapster.org/linux/kernel/lp/