LECCIÓN CONSOLA

LECCIÓN CONSOLA....1
INTRODUCCIÓN....2
Conceptos básicos....2
Terminal:....2
Tipos de terminales:....2
Organización y estructuras de datos 3
ESTRUCTURAS DE DATOS....5
tty_struct....5
tty_driver....6
tty operations....7
tty_ldisc....8
ktermio....9
vc_data....9
FUNCIONES....11
Inicialización de la Consola 11
Función de creación y configuración de la consola....12
Operaciones:....13
CON_OPEN....13
CON_CLOSE....14
CON_WRITE....15
Bibliografía....19

INTRODUCCIÓN

Conceptos básicos

Terminal:

Un terminal es un dispositivo que permite al usuario interactuar con una máquina. Tiene la capacidad de enviar y recibir datos mediante un canal de comunicación. Los terminales **se representan por archivos especialess** conocidos como **archivos de modo carácter,** en los que es posible leer cuando un usuario presiona una tecla del teclado, o escribir cuando el datos es enviado vía modem por el puerto serie.

Tipos de terminales:

Consolas virtuales: usadas normalmente cuando el usuario está conectado físicamente a la máquina.

Pseudo terminales: usadas en una conexión remota.

Puertos series: usados por módems y ratones.

Terminales particulares: usados por tarjetas de serie específicas y ciertos ratones.

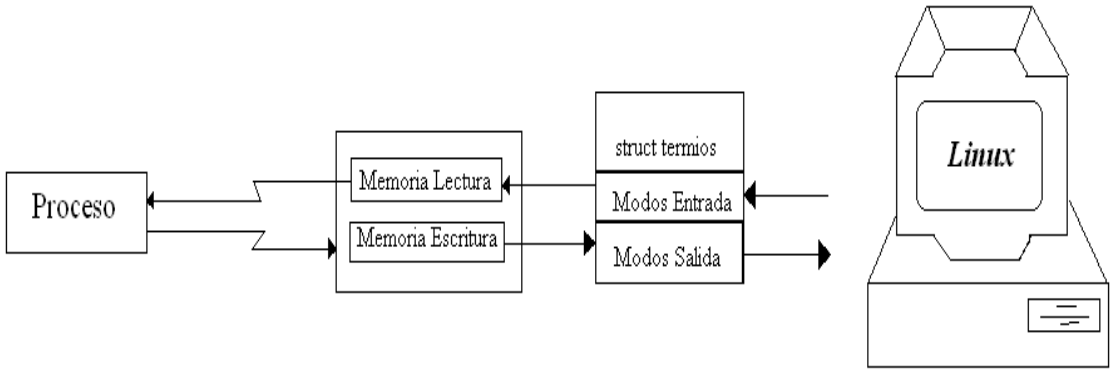
Se pueden distinguir dos modos de uso de un terminal:

- ✓ **Modo canónico**: Es el más simple. La entrada de un terminal se gestiona en forma de líneas. Esto significa que un programa que intenta leer una línea en un terminal debe esperar que una línea completa haya sido introducida antes de poder tratarla. Los caracteres son tratados, y ciertos caracteres como shift o control son interpretados.

- ✓ **Modo no canónico**: Los caracteres en entreda no se tratan en forma de línea. Son tratados e interpretados por el programa de usuario. Los valores MIN y TIME se utilizan para determinar la manera como se reciben los caracteres. MIN corresponde al número mínimo de caracteres que deben recibirse antes de que la lectura sea satisfecha. TIME corresponde a un timer en décimas de segundo que se utiliza para hacer accesibles los datos tras un cierto lapso de tiempo.

El intercambio de datos entre un proceso y un terminal se efectúa mediante dos memorias intermedias. Estas memorias permiten acelerar las transferencias entre el proceso y el terminal. Cuando se transfiere un carácter de la memoria al terminal, sufre una transformación en función de las características del terminal, y al revés cuando el carácter proviene del terminal, ya que, para poder dialogar correctamente con los parámetros concretos resulta necesario adaptarse al protocolo de comunicación del

terminal. La escritura y la lectura son transparentes porque la operación de conversión se encapsula y no es visible por el usuario.

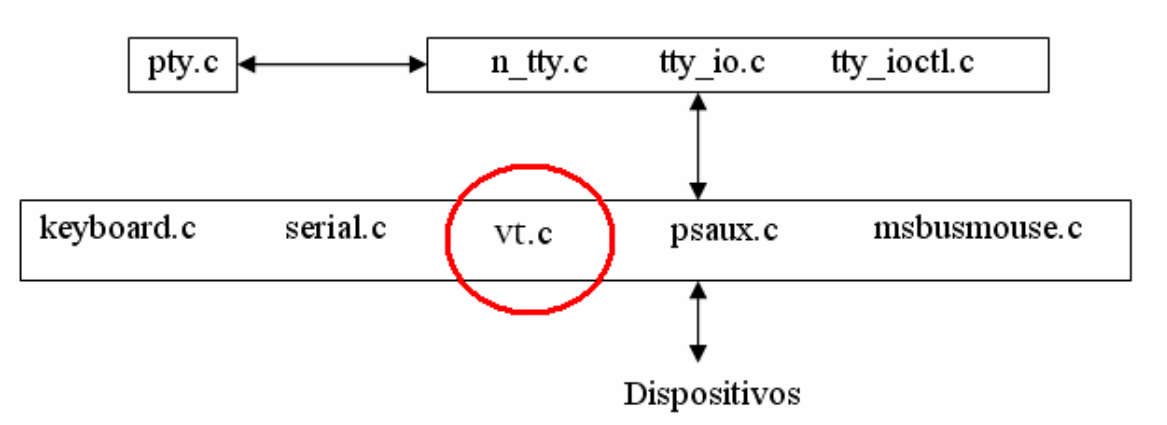


Comunicación entre proceso y terminal

Organización y estructuras de datos

Los terminales pueden considerarse como una interfaz lógica entre los datos y el material que debe transmitirse a través de un dispositivo cualquiera como una línea serie, un ratón, una impresora o incluso la consola de la máquina de un usuario.
Básicamente son importantes cuatro archivos que encapsulan las operaciones de alto nivel:

- ✓ tty_io.c: Gestiona todas las entradas/salidas de alto nivel sobre los terminales.
- ✓ tty_ioctl.c: Gestiona la llamada a ioctl sobre un terminal y se encarga de repercutir la llamada al gestor del dispositivo.
- ✓ n_tty.c: Se encarga de la disciplina de la línea.
- ✓ pty.c: Se encarga de la gestión de los pseudoterminales, que se basa realmente en los archivos anteriores.



Archivos que encapsulan las operaciones de alto nivel.

tty_io.c

Gestiona todas las entradas/salidas de alto nivel sobre los terminales.

tty_ioctl.c

Gestiona la llamada ioctl sobre un terminal y se encarga de repercutir la llamada, si es necesario, al gestor de dispositivo.

n_tty.c

Se encarga de la disciplina de la linea, es decir, procesa el flujo de entrada/salida al mismo tiempo que algunas funciones de control. Es la interfaz entre el usuario y el driver.

pty.c

Gestiona los pseudoterminales.

Fichero de cabecera <linux/tty.h>

struct tty_struct

Es la principal a partir de la cual se efectúan todas las operaciones realizadas en el núcleo. Se encarga del control de una terminal.

struct termios

Esta estructura contiene la configuración de un terminal peranto examinar la configuración del terminal como cambiar sus parámetros.

```
struct termios
{
 tcflag_t c_iflag;    /* modos de entrada     */
 tcflag_t c_oflag;    /* modos de salida      */
 tcflag_t c_cflag;    /* modos de control     */
 tcflag_t c_lflag;    /* modos locales        */
 cc_t c_line;         /* disciplina de linea  */
 cc_t c_cc[NCCS]; /* caracteres de control */
};
```

Fichero de cabecera <linux/tty_driver.h>

struct tty_driver

Es la estructura define el dispositivo que gestionará la capa de bajo nivel del terminal.

Fichero de cabecera <linux/tty_ldisc.h>

struct tty_ldisc

Es la estructura proporciona una interfaz de acceso a la disciplina de la linea.

ESTRUCTURAS DE DATOS

tty_struct

Se trata de la estructura principal a partir de la cual se efectúan todas las operaciones realizadas en el núcleo. Se encarga del control de un terminal. Se encuentra definida en el archivo cabecera linux/tty.h.

Nota: En las estructuras, lo que se especifica en rojo es lo más importante

```
struct tty_struct {
 221        int     magic;
 222        struct kref kref;
 223        struct tty_driver *driver; /*Interfaz de acceso al disp. Asociado al
terminal*/
 224         const struct tty_operations *ops; /*Operaciones de la consola. Se
inicializarán las operaciones de esta estructura con funciones como con_open.... */
 225        int index; /*Indice del terminal. Podemos tener varios*/
 226        /* The ldisc objects are protected by tty_ldisc_lock at the moment */
 227        struct tty_ldisc ldisc; /*Interfaz para la disciplina de linea*/
 228        struct mutex termios_mutex;
 229        spinlock_t ctrl_lock;
 230        /* Termios values are protected by the termios mutex */
 231        struct ktermios *termios, *termios_locked; /*Configuración del terminal*/
 232        struct termiox *termiox;        /* May be NULL for unsupported */
 233        char name[64];
 234        struct pid *pgrp;               /* Protected by ctrl lock */
 235        struct pid *session;
 236        unsigned long flags;  /*Estado del terminal*/
 237        int count;
 238        struct winsize winsize;         /* Tamaño de la ventana */
 239        unsigned char stopped:1, hw_stopped:1, flow_stopped:1, packet:1;
 240        unsigned char low_latency:1, warned:1;
 241        unsigned char ctrl_status;      /* ctrl_lock */
 242        unsigned int receive_room;      /* Bytes free for queue */
 243
 244        struct tty_struct *link;
 245        struct fasync_struct *fasync;
 246        struct tty_bufhead buf;         /* Locked internally */
 247        int alt_speed;          /* For magic substitution of 38400 bps */
 248        wait_queue_head_t write_wait; /*Lista de procesos en espera de escritura*/
 249        wait_queue_head_t read_wait; /*Lista de procesos en espera de lectura*/

 250        struct work_struct hangup_work;
 251        void *disc_data;
 252        void *driver_data; /*Puntero a la estructura de datos de la consola Si es
nulo es que no tiene asignado esta estructura*/
 253        struct list_head tty_files;
 254
 255#define N_TTY_BUF_SIZE 4096
 256
 257        /*
 258         * The following is data for the N_TTY line discipline.  For
 259         * historical reasons, this is included in the tty structure.
 260         * Mostly locked by the BKL.
 261         */
 262        unsigned int column;
 263        unsigned char lnext:1, erasing:1, raw:1, real_raw:1, icanon:1;
 264        unsigned char closing:1;
 265        unsigned char echo_overrun:1;
```

```
266       unsigned short minimum_to_wake;
267       unsigned long overrun_time;
268       int num_overrun;
269       unsigned long process_char_map[256/(8*sizeof(unsigned long))];
270       char *read_buf;
271       int read_head;
272       int read_tail;
273       int read_cnt;
274       unsigned long read_flags[N_TTY_BUF_SIZE/(8*sizeof(unsigned long))];
275       unsigned char *echo_buf;
276       unsigned int echo_pos;
277       unsigned int echo_cnt;
278       int canon_data;
279       unsigned long canon_head;
280       unsigned int canon_column;
281       struct mutex atomic_read_lock;
282       struct mutex atomic_write_lock;
283       struct mutex output_lock;
284       struct mutex echo_lock;
285       unsigned char *write_buf;
286       int write_cnt;
287       spinlock_t read_lock;
288       /* If the tty has a pending do_SAK, queue it here - akpm */
289       struct work_struct SAK_work;
290       struct tty_port *port;
291};
```

tty_driver

Se define en el archivo de cabecera linux/tty_driver.h, y define el dispositivo que gestionará la capa de bajo nivel del terminal.

```
struct tty_driver {
270       int     magic;          /* numero mágico */
271       struct kref kref;       /* Reference management */
272       struct cdev cdev;
273       struct module   *owner;
274       const char      *driver_name; /*Nombre del driver*/
275       const char      *name;
276       int     name_base;      /* offset of printed name */
277       int     major;          /* major device number */
278       int     minor_start;    /* start of minor device number */
279       int     minor_num;      /* number of *possible* devices */
280       int     num;            /* number of devices allocated */
281       short   type;           /* type of tty driver */
282       short   subtype;        /* subtype of tty driver */
283       struct ktermios init_termios; /* Initial termios */
284       int     flags;          /* tty driver flags */
285       struct proc_dir_entry *proc_entry; /* /proc fs entry */
286       struct tty_driver *other; /* only used for the PTY driver */
287
288       /*
289        * Pointer to the tty data structures
290        */
291       struct tty_struct **ttys; /*Puntero a la estruc. De datos de los terminales*/
292       struct ktermios **termios;
293       struct ktermios **termios_locked;
294       void *driver_state;
295
296       /*
297        * Driver methods
298        */
299
```

```
 300        const struct tty_operations *ops; /*Puntero a los operaciones del
terminal*/
 301        struct list_head tty_drivers;
 302};
```

tty operations

```
struct tty_operations {
 227        struct tty_struct * (*lookup)(struct tty_driver *driver,
 228                    struct inode *inode, int idx);
 229        int  (*install)(struct tty_driver *driver, struct tty_struct *tty);
 230        void (*remove)(struct tty_driver *driver, struct tty_struct *tty);
 231        int  (*open)(struct tty_struct * tty, struct file * filp); /*Apertura del
terminal*/
 232        void (*close)(struct tty_struct * tty, struct file * filp); /*Cierre del terminal*/
 233        void (*shutdown)(struct tty_struct *tty);
 234        int  (*write)(struct tty_struct * tty,
 235                    const unsigned char *buf, int count); /*Escritura del terminal*/
 236        int  (*put_char)(struct tty_struct *tty, unsigned char ch);
 237        void (*flush_chars)(struct tty_struct *tty);
 238        int  (*write_room)(struct tty_struct *tty);
 239        int  (*chars_in_buffer)(struct tty_struct *tty);
 240        int  (*ioctl)(struct tty_struct *tty, struct file * file,
 241                    unsigned int cmd, unsigned long arg);
 242        long (*compat_ioctl)(struct tty_struct *tty, struct file * file,
 243                        unsigned int cmd, unsigned long arg);
 244        void (*set_termios)(struct tty_struct *tty, struct ktermios * old);
 245        void (*throttle)(struct tty_struct * tty);
 246        void (*unthrottle)(struct tty_struct * tty);
 247        void (*stop)(struct tty_struct *tty);
 248        void (*start)(struct tty_struct *tty);
 249        void (*hangup)(struct tty_struct *tty);
 250        int (*break_ctl)(struct tty_struct *tty, int state);
 251        void (*flush_buffer)(struct tty_struct *tty);
 252        void (*set_ldisc)(struct tty_struct *tty);
 253        void (*wait_until_sent)(struct tty_struct *tty, int timeout);
 254        void (*send_xchar)(struct tty_struct *tty, char ch);
 255        int (*read_proc)(char *page, char **start, off_t off,
 256                        int count, int *eof, void *data);
 257        int (*tiocmget)(struct tty_struct *tty, struct file *file);
 258        int (*tiocmset)(struct tty_struct *tty, struct file *file,
 259                    unsigned int set, unsigned int clear);
 260        int (*resize)(struct tty_struct *tty, struct winsize *ws);
 261        int (*set_termiox)(struct tty_struct *tty, struct termiox *tnew);
 262#ifdef CONFIG_CONSOLE_POLL
 263        int (*poll_init)(struct tty_driver *driver, int line, char *options);
 264        int (*poll_get_char)(struct tty_driver *driver, int line);
 265        void (*poll_put_char)(struct tty_driver *driver, int line, char ch);
 266#endif
 267};
```

tty_ldisc

Se define en el archivo de cabecera linux/tty_ldisc.h y proporciona una interfaz de acceso a la disciplina de la línea. En esta nueva versión aparece la estructura tty_ldisc_ops;

```
145struct tty_ldisc {
146       struct tty_ldisc_ops *ops; /*Puntero a las operaciones sobre la disciplina de linea*/
147       int refcount;
148};

struct tty_ldisc_ops {
108       int     magic;   /*Numero mágico*/
109       char    *name;
110       int     num;     /*Identificador de la línea*/
111       int     flags;   /*Tipo de linea*/
112
113       /*
114        * The following routines are called from above.
115        */
116       int     (*open)(struct tty_struct *);  /*Apertura de la línea*/
117        void    (*close)(struct tty_struct *);  /*Cierre de la linea*/
118       void    (*flush_buffer)(struct tty_struct *tty);
119       ssize_t (*chars_in_buffer)(struct tty_struct *tty);
120        ssize_t (*read)(struct tty_struct * tty, struct file * file,
121                        unsigned char __user * buf, size_t nr);
122       ssize_t (*write)(struct tty_struct * tty, struct file * file,
123                        const unsigned char * buf, size_t nr);
124       int     (*ioctl)(struct tty_struct * tty, struct file * file,
125                        unsigned int cmd, unsigned long arg);
126       long    (*compat_ioctl)(struct tty_struct * tty, struct file * file,
127                               unsigned int cmd, unsigned long arg);
128        void    (*set_termios)(struct tty_struct *tty, struct ktermios * old);
/*Configuración de la línea*/
129       unsigned int (*poll)(struct tty_struct *, struct file *,
130                            struct poll_table_struct *);
131       int     (*hangup)(struct tty_struct *tty);
132
133       /*
134        * The following routines are called from below.
135        */
136       void    (*receive_buf)(struct tty_struct *, const unsigned char *cp,
137                              char *fp, int count);
138       void    (*write_wakeup)(struct tty_struct *);
139
140       struct  module *owner;
141
142       int refcount;
143};
```

ktermio

Permite examinar la configuración del terminal y modificar los parámetros. Esta estructura es dependiente de la arquitectura. Definida en los archivo <arch/<arquitectura>/include/asm/termbits.h>.

```
 40struct ktermios {
 41      tcflag_t c_iflag;           /* Modo de entrada */
 42      tcflag_t c_oflag;           /* Modo de salida */
 43      tcflag_t c_cflag;           /* Modo de control*/
 44      tcflag_t c_lflag;           /* Modo locales*/
 45      cc_t c_line;                /* Caracteres de control*/
 46      cc_t c_cc[NCCS];            /* Disciplina de la línea*/
 47      speed_t c_ispeed;           /* Velocidad de entrada*/
 48      speed_t c_ospeed;           /* Velocidad de salida*/
 49};
```

vc_data

Se encuentra en el archivo de cabecera /include/linux/console_struct.h, y define a la consola virtual asociada a un Terminal determinado.

```
struct vc_data {
 24      unsigned short  vc_num;             /* Console number */
 25      unsigned int    vc_cols;            /* [#] Console size */
 26      unsigned int    vc_rows;
 27      unsigned int    vc_size_row;        /* Bytes per row */
 28      unsigned int    vc_scan_lines;      /* # of scan lines */
 29      unsigned long   vc_origin;          /* [!] Start of real screen */
 30      unsigned long   vc_scr_end;         /* [!] End of real screen */
 31      unsigned long   vc_visible_origin;  /* [!] Top of visible window */
 32      unsigned int    vc_top, vc_bottom;  /* Scrolling region */
 33      const struct consw *vc_sw;
 34      unsigned short  *vc_screenbuf;      /* In-memory character/attribute buffer */
 35      unsigned int    vc_screenbuf_size;
 36      unsigned char   vc_mode;            /* KD_TEXT, ... */
 37      /* attributes for all characters on screen */
 38      unsigned char   vc_attr;            /* Current attributes */
 39      unsigned char   vc_def_color;       /* Default colors */
 40      unsigned char   vc_color;           /* Foreground & background */
 41      unsigned char   vc_s_color;         /* Saved foreground & background */
 42      unsigned char   vc_ulcolor;         /* Color for underline mode */
 43      unsigned char   vc_itcolor;
 44      unsigned char   vc_halfcolor;       /* Color for half intensity mode */
 45      /* cursor */
 46      unsigned int    vc_cursor_type;
 47      unsigned short  vc_complement_mask;     /* [#] Xor mask for mouse pointer */
 48      unsigned short  vc_s_complement_mask;   /* Saved mouse pointer mask */
 49      unsigned int    vc_x, vc_y;         /* Cursor position */
 50      unsigned int    vc_saved_x, vc_saved_y;
 51      unsigned long   vc_pos;             /* Cursor address */
 52      /* fonts */
 53      unsigned short  vc_hi_font_mask;    /* [#] Attribute set for upper 256 chars of font
or 0 if not supported */
 54      struct console_font vc_font;        /* Current VC font set */
 55      unsigned short  vc_video_erase_char;    /* Background erase character */
 56      /* VT terminal data */
 57      unsigned int    vc_state;           /* Escape sequence parser state */
 58      unsigned int    vc_npar,vc_par[NPAR];   /* Parameters of current escape sequence */
 59      struct tty_struct *vc_tty;          /*Terminal en el que se encuetra la consola
*/ 60      /* data for manual vt switching */
 61      struct vt_mode  vt_mode;
 62      struct pid      *vt_pid;
```

```
 63      int           vt_newvt;
 64      wait_queue_head_t paste_wait;
 65      /* mode flags */
 66      unsigned int    vc_charset      : 1;    /* Character set G0 / G1 */
 67      unsigned int    vc_s_charset    : 1;    /* Saved character set */
 68      unsigned int    vc_disp_ctrl    : 1;    /* Display chars < 32? */
 69      unsigned int    vc_toggle_meta  : 1;    /* Toggle high bit? */
 70      unsigned int    vc_decscnm      : 1;    /* Screen Mode */
 71      unsigned int    vc_decom        : 1;    /* Origin Mode */
 72      unsigned int    vc_decawm       : 1;    /* Autowrap Mode */
 73      unsigned int    vc_deccm        : 1;    /* Cursor Visible */
 74      unsigned int    vc_decim        : 1;    /* Insert Mode */
 75      unsigned int    vc_deccolm      : 1;    /* 80/132 Column Mode */
 76      /* attribute flags */
 77      unsigned int    vc_intensity    : 2;    /* 0=half-bright, 1=normal, 2=bold */
 78      unsigned int    vc_italic:1;
 79      unsigned int    vc_underline    : 1;
 80      unsigned int    vc_blink        : 1;
 81      unsigned int    vc_reverse      : 1;
 82      unsigned int    vc_s_intensity  : 2;    /* saved rendition */
 83      unsigned int    vc_s_italic:1;
 84      unsigned int    vc_s_underline  : 1;
 85      unsigned int    vc_s_blink      : 1;
 86      unsigned int    vc_s_reverse    : 1;
 87      /* misc */
 88      unsigned int    vc_ques         : 1;
 89      unsigned int    vc_need_wrap    : 1;
 90      unsigned int    vc_can_do_color : 1;
 91      unsigned int    vc_report_mouse : 2;
 92      unsigned int    vc_kmalloced    : 1;
 93      unsigned char   vc_utf          : 1;    /* Unicode UTF-8 encoding */
 94      unsigned char   vc_utf_count;
 95              int     vc_utf_char;
 96      unsigned int    vc_tab_stop[8];         /* Tab stops. 256 columns. */
 97      unsigned char   vc_palette[16*3];       /* Colour palette for VGA+ */
 98      unsigned short * vc_translate;
 99      unsigned char   vc_G0_charset;
100       unsigned char   vc_G1_charset;
101       unsigned char   vc_saved_G0;
102       unsigned char   vc_saved_G1;
103       unsigned int    vc_resize_user;         /* resize request from user */
104       unsigned int    vc_bell_pitch;          /* Console bell pitch */
105       unsigned int    vc_bell_duration;       /* Console bell duration */
106       struct vc_data **vc_display_fg;         /* [!] Ptr to var holding fg console for this
display */
107       unsigned long   vc_uni_pagedir;
108       unsigned long   *vc_uni_pagedir_loc;  /* [!] Location of uni_pagedir variable for this
console */
109       /* additional information is in vt_kern.h */
110};
```

FUNCIONES

Inicialización de la Consola

Las inicializaciones se efectúan en el módulo drivers/char/tty_io.c encargado de gestionar las entradas/salidas sobre los terminales. La función de inicialización tty_init inicializa los terminales de la máquina y seguidamente, invoca las funciones de inicialización de los diferentes dispositivos. En nuestro caso nos centraremos en la función vty_init que inicializa los dispositivos vcs y que se encuentra en drivers/char/vt.c.

La inicialización con_init y las operaciones con_open con_close con_write usan un vector de punteros a vc_data denominado vc_cons [].

```
struct tty_driver *console_driver;

int __init vty_init(const struct file_operations *console_fops)
2931{
2932        cdev_init(&vc0_cdev, console_fops);
2933        if (cdev_add(&vc0_cdev, MKDEV(TTY_MAJOR, 0), 1) ||
2934            register_chrdev_region(MKDEV(TTY_MAJOR, 0), 1, "/dev/vc/0") < 0)
2935                panic("Couldn't register /dev/tty0 driver\n");
2936        device_create(tty_class, NULL, MKDEV(TTY_MAJOR, 0), NULL, "tty0");
2937
2938        vcs_init();
2939
2940        console_driver = alloc_tty_driver(MAX_NR_CONSOLES);/*Reserva memoria para
el dispositivo de la consola*/
2941        if (!console_driver)
2942                panic("Couldn't allocate console driver\n");
        //Se inicializa la consola
2943        console_driver->owner = THIS_MODULE;
2944        console_driver->name = "tty";
2945        console_driver->name_base = 1;
2946        console_driver->major = TTY_MAJOR;
2947        console_driver->minor_start = 1;
2948        console_driver->type = TTY_DRIVER_TYPE_CONSOLE;
2949        console_driver->init_termios = tty_std_termios;
2950        if (default_utf8)
2951                console_driver->init_termios.c_iflag |= IUTF8;
2952        console_driver->flags = TTY_DRIVER_REAL_RAW | TTY_DRIVER_RESET_TERMIOS;
        //Inicializa el conjunto de operaciones posibles
2953        tty_set_operations(console_driver, &con_ops);
        //Esta función inserta el dispositivo en la lista encadenada de dispositivos
(registro)

2954        if (tty_register_driver(console_driver))
2955                panic("Couldn't register console driver\n");
2956        kbd_init();
2957        console_map_init();
2958#ifdef CONFIG_PROM_CONSOLE
2959        prom_con_init();
2960#endif
2961#ifdef CONFIG_MDA_CONSOLE
2962        mda_console_init();
2963#endif
2964        return 0;
2965}
2966
```

Función de creación y configuración de la consola.

La función que se muestra a continuación inicializa las estructuras de las consolas de los terminales

```
static int __init con_init(void)
{
        const char *display_desc = NULL;
        struct vc_data *vc;
        unsigned int currcons = 0, i;

        acquire_console_sem();  //Adquirimos el cerrejo

        if (conswitchp)
                display_desc = conswitchp->con_startup();
        if (!display_desc) {
                fg_console = 0;
                release_console_sem();
                return 0;
        }

        for (i = 0; i < MAX_NR_CON_DRIVER; i++) {
                struct con_driver *con_driver = &registered_con_driver[i];

                if (con_driver->con == NULL) {
                        con_driver->con = conswitchp;
                        con_driver->desc = display_desc;
                        con_driver->flag = CON_DRIVER_FLAG_INIT;
                        con_driver->first = 0;
                        con_driver->last = MAX_NR_CONSOLES - 1;
                        break;
                }
        }

        for (i = 0; i < MAX_NR_CONSOLES; i++)
                con_driver_map[i] = conswitchp;

        if (blankinterval) {
                blank_state = blank_normal_wait;
                mod_timer(&console_timer, jiffies + blankinterval);
        }

        /*
         * kmalloc is not running yet - we use the bootmem allocator.
         */
      //Se crea y se configura la consola

        for (currcons = 0; currcons < MIN_NR_CONSOLES; currcons++) {
                    //Se reserve memoria para albergar la consola
                vc_cons[currcons].d = vc = alloc_bootmem(sizeof(struct vc_data));
                INIT_WORK(&vc_cons[currcons].SAK_work, vc_SAK);
                visual_init(vc, currcons, 1);
                vc->vc_screenbuf = (unsigned short *)alloc_bootmem(vc-
>vc_screenbuf_size);
                vc->vc_kmalloced = 0;
              /*Función de configuración de la consola (color...)*/
                vc_init(vc, vc->vc_rows, vc->vc_cols,
                        currcons || !vc->vc_sw->con_save_screen);
        }
        currcons = fg_console = 0;
        master_display_fg = vc = vc_cons[currcons].d;
        set_origin(vc);
        save_screen(vc);
        gotoxy(vc, vc->vc_x, vc->vc_y);
```

```
2894        csi_J(vc, 0);
        /*Actualiza la pantalla*/
2895        update_screen(vc);
2896        printk("Console: %s %s %dx%d",
2897             vc->vc_can_do_color ? "colour" : "mono",
2898             display_desc, vc->vc_cols, vc->vc_rows);
2899        printable = 1;
2900        printk("\n");
2901
2902        release_console_sem();
2903
2904#ifdef CONFIG_VT_CONSOLE
        /*Registra la consola e imprime los mensajes del kernel*/
2905        register_console(&vt_console_driver);
2906#endif
2907        return 0;
2908}
```

Operaciones:

Antes vimos que la estructura tty_driver tenía un campo struct tty_operations *ops que apuntaba a la estructura tty_operations. Es aquí donde se asignan dichas operaciones. Nos centraremos en las funciones con_open, con_close, con_write.

```
1static const struct tty_operations con_ops = {
2912        .open = con_open,
2913        .close = con_close,
2914        .write = con_write,
2915        .write_room = con_write_room,
2916        .put_char = con_put_char,
2917        .flush_chars = con_flush_chars,
2918        .chars_in_buffer = con_chars_in_buffer,
2919        .ioctl = vt_ioctl,
2920        .stop = con_stop,
2921        .start = con_start,
2922        .throttle = con_throttle,
2923        .unthrottle = con_unthrottle,
2924        .resize = vt_resize,
2925        .shutdown = con_shutdown
2926};
```

CON_OPEN

```
static int con_open(struct tty_struct *tty, struct file *filp)
2752{
        /*Index indica que numero de terminal queremos asociar a la consola a
abrir.*/
2753        unsigned int currcons = tty->index;
2754        int ret = 0;
2755
            //Adquirimos el cerrojo
2756        acquire_console_sem();
        /*Si no tiene asignado una estructura de datos de la consola*/
2757        if (tty->driver_data == NULL) {
        /* Esta función inicializa el entorno gráfico de la consola, comprueba que
hay memoria, la reserva si no hay más de un número máximo de terminales
abiertos, genera el terminal gráfico con los colores por defecto, etc. Si la llamada a
esta función se realiza con éxito, esta función devolverá un 0*/
```

```
2758            ret = vc_allocate(currcons);
2759            if (ret == 0) {
                        /*Inicializamos la consola virtual que corresponde al
terminal currcons*/
2760                    struct vc_data *vc = vc_cons[currcons].d;
2761
2762                    /* Still being freed */
2763                    if (vc->vc_tty) {
2764                            release_console_sem();
2765                            return -ERESTARTSYS;d
2766                    }
2767                    tty->driver_data = vc;
2768                    vc->vc_tty = tty; //Se le asigna el terminal a la consola
2769//Se le asigna el tamaño de la ventana
2770                    if (!tty->winsize.ws_row && !tty->winsize.ws_col) {
2771                            tty->winsize.ws_row = vc_cons[currcons].d->vc_rows;
2772                            tty->winsize.ws_col = vc_cons[currcons].d->vc_cols;
2773                    }
2774                    if (vc->vc_utf)
2775                            tty->termios->c_iflag |= IUTF8;
2776                    else
2777                            tty->termios->c_iflag &= ~IUTF8;
2778                    vcs_make_sysfs(tty); //Se libera el semaforo
2779                    release_console_sem();//Guardamos en un fichero el Terminal creado

2780                    return ret;
2781            }
2782    }
2783    release_console_sem();
2784    return ret;
2785}
```

CON_CLOSE

```
2787static void con_close(struct tty_struct *tty, struct file *filp)
2788{
2789     /* Nothing to do - we defer to shutdown */
2790}

2792static void con_shutdown(struct tty_struct *tty)
2793{
     //Obtenemos la 'consola' de ese terminal
2794     struct vc_data *vc = tty->driver_data;
2795     BUG_ON(vc == NULL);
     //Adquirimos el cerrojo
2796     acquire_console_sem();
     //Desvinculamos la 'consola' del terminal
2797     vc->vc_tty = NULL;
2798     vcs_remove_sysfs(tty);
2799     release_console_sem();
2800     tty_shutdown(tty);
2801}
```

CON_WRITE

La función con_write se invoca cuando se pretende escribir en la consola. Sin embargo, delega el tratamiento de los caracteres a la función do_con_write.

```
2658static int con_write(struct tty_struct *tty, const unsigned char *buf, int count)
2659{
2660        int     retval;
2661
2662        retval = do_con_write(tty, buf, count); //Se llama a do_con_write
2663        con_flush_chars(tty);
2664
2665        return retval;
2666}

//Especificamos cuanto queremos escribir con la variable count
static int do_con_write(struct tty_struct *tty, const unsigned char *buf, int count)
2094{
2095#ifdef VT_BUF_VRAM_ONLY
2096#define FLUSH do { } while(0);
2097#else
2098#define FLUSH if (draw_x >= 0) { \
2099        vc->vc_sw->con_putcs(vc, (u16 *)draw_from, (u16 *)draw_to - (u16 *)draw_from,
vc->vc_y, draw_x); \
2100        draw_x = -1; \
2101        }
2102#endif
2103
2104        int c, tc, ok, n = 0, draw_x = -1;
2105        unsigned int currcons;
2106        unsigned long draw_from = 0, draw_to = 0;
2107        struct vc_data *vc;
2108        unsigned char vc_attr;
2109        struct vt_notifier_param param;
2110        uint8_t rescan;
2111        uint8_t inverse;
2112        uint8_t width;
2113        u16 himask, charmask;
2114        const unsigned char *orig_buf = NULL;
2115        int orig_count;
2116
2117        if (in_interrupt())
2118                return count;
2119
2120        might_sleep();
2121
2122        acquire_console_sem();  //Adquiere el semáforo
2123        vc = tty->driver_data;  //Obtiene el la consola asociada

         //Si el terminal no tiene asociada ningúna consola error
2124        if (vc == NULL) {
2125                printk(KERN_ERR "vt: argh, driver_data is NULL !\n");
2126                release_console_sem();
2127                return 0;
2128        }
2129
2130        currcons = vc->vc_num;
2131        if (!vc_cons_allocated(currcons)) {
2132           /* could this happen? */
2133           static int error = 0;
2134           if (!error) {
2135              error = 1;
2136              printk("con_write: tty %d not allocated\n", currcons+1);
2137           }
```

```
2138            release_console_sem();
2139            return 0;
2140        }
2141        orig_buf = buf;
2142        orig_count = count;
2143
2144        himask = vc->vc_hi_font_mask;
2145        charmask = himask ? 0x1ff : 0xff;
2146
2147        /* undraw cursor first */
2148        if (IS_FG(vc))
2149                hide_cursor(vc);
2150
2151        param.vc = vc;
2152

                //Lee cada uno de los caracteres del buffer y los va tratando
2153        while (!tty->stopped && count) {
2154                int orig = *buf;
2155                c = orig;
2156                buf++;
2157                n++;
2158                count--;
2159                rescan = 0;
2160                inverse = 0;
2161                width = 1;
2162
2163                /* Do no translation at all in control states */
2164                if (vc->vc_state != ESnormal) {
2165                        tc = c; //Si está en modo normal se coge el carácter tal cual
2166                }
                 //Si la consola está en modo UTF, la secuencia de caracteres se
interpreta según el esquema de codificación UTF
else if (vc->vc_utf && !vc->vc_disp_ctrl) {
2167                    /* Combine UTF-8 into Unicode in vc_utf_char.
2168                     * vc_utf_count is the number of continuation bytes still
2169                     * expected to arrive.
2170                     * vc_npar is the number of continuation bytes arrived so
2171                     * far
2172                     */
2173rescan_last_byte:
2174                    if ((c & 0xc0) == 0x80) {
2175                        /* Continuation byte received */
2176                        static const uint32_t utf8_length_changes[] = { 0x0000007f, 0x000007ff,
0x0000ffff, 0x001fffff, 0x03ffffff, 0x7fffffff };
2177                        if (vc->vc_utf_count) {
2178                            vc->vc_utf_char = (vc->vc_utf_char << 6) | (c & 0x3f);
2179                            vc->vc_npar++;
2180                            if (--vc->vc_utf_count) {
2181                                /* Still need some bytes */
2182                                continue;
2183                            }
2184                            /* Got a whole character */
2185                            c = vc->vc_utf_char;
2186                            /* Reject overlong sequences */
2187                            if (c <= utf8_length_changes[vc->vc_npar - 1] ||
2188                                    c > utf8_length_changes[vc->vc_npar])
2189                                c = 0xfffd;
2190                        } else {
2191                            /* Unexpected continuation byte */
2192                            vc->vc_utf_count = 0;
2193                            c = 0xfffd;
2194                        }
2195                    } else {
2196                        /* Single ASCII byte or first byte of a sequence received */
2197                        if (vc->vc_utf_count) {
```

```
2198                    /* Continuation byte expected */
2199                    rescan = 1;
2200                    vc->vc_utf_count = 0;
2201                    c = 0xfffd;
2202                } else if (c > 0x7f) {
2203                    /* First byte of a multibyte sequence received */
2204                    vc->vc_npar = 0;
2205                    if ((c & 0xe0) == 0xc0) {
2206                        vc->vc_utf_count = 1;
2207                        vc->vc_utf_char = (c & 0x1f);
2208                    } else if ((c & 0xf0) == 0xe0) {
2209                        vc->vc_utf_count = 2;
2210                        vc->vc_utf_char = (c & 0x0f);
2211                    } else if ((c & 0xf8) == 0xf0) {
2212                        vc->vc_utf_count = 3;
2213                        vc->vc_utf_char = (c & 0x07);
2214                    } else if ((c & 0xfc) == 0xf8) {
2215                        vc->vc_utf_count = 4;
2216                        vc->vc_utf_char = (c & 0x03);
2217                    } else if ((c & 0xfe) == 0xfc) {
2218                        vc->vc_utf_count = 5;
2219                        vc->vc_utf_char = (c & 0x01);
2220                    } else {
2221                        /* 254 and 255 are invalid */
2222                        c = 0xfffd;
2223                    }
2224                    if (vc->vc_utf_count) {
2225                        /* Still need some bytes */
2226                        continue;
2227                    }
2228                }
2229                /* Nothing to do if an ASCII byte was received */
2230            }
2231            /* End of UTF-8 decoding. */
2232            /* c is the received character, or U+FFFD for invalid sequences. */
2233            /* Replace invalid Unicode code points with U+FFFD too */
2234            if ((c >= 0xd800 && c <= 0xdfff) || c == 0xfffe || c == 0xffff)
2235                c = 0xfffd;
2236            tc = c;
2237        } else {      /* no utf or alternate charset mode */            //Si no esta
en modo UTF se traduce
2238            tc = vc_translate(vc, c);
2239        }
2240
2241        param.c = tc;
2242        if (atomic_notifier_call_chain(&vt_notifier_list, VT_PREWRITE,
2243                        &param) == NOTIFY_STOP)
2244            continue;
2245
2246        /* If the original code was a control character we
2247         * only allow a glyph to be displayed if the code is
2248         * not normally used (such as for cursor movement) or
2249         * if the disp_ctrl mode has been explicitly enabled.
2250         * Certain characters (as given by the CTRL_ALWAYS
2251         * bitmap) are always displayed as control characters,
2252         * as the console would be pretty useless without
2253         * them; to display an arbitrary font position use the
2254         * direct-to-font zone in UTF-8 mode.
2255         */
           //Se comprueba que el carácter sea imprimible o no

2256        ok = tc && (c >= 32 ||
2257                !(vc->vc_disp_ctrl ? (CTRL_ALWAYS >> c) & 1 :
2258                    vc->vc_utf || ((CTRL_ACTION >> c) & 1)))
2259            && (c != 127 || vc->vc_disp_ctrl)
2260            && (c != 128+27);
```

```
2261
2262            if (vc->vc_state == ESnormal && ok) {
2263                    if (vc->vc_utf && !vc->vc_disp_ctrl) {
2264                            if (is_double_width(c))
2265                                    width = 2;
2266                    }
2267                    /* Now try to find out how to display it //En caso afirmativo, se llama
a la función conv_uni_to_c para                  // determinar las conversiones
necesarias a llevar a cabo para                                  //imprimir el
carácter

2268                    tc = conv_uni_to_pc(vc, tc);
2269                    if (tc & ~charmask) {
2270                            if (tc == -1 || tc == -2) {
2271                               continue; /* nothing to display */
2272                            }
2273                            /* Glyph not found */
2274                            if ((!(vc->vc_utf && !vc->vc_disp_ctrl) && c < 128) && !(c &
~charmask)) {
2275                               /* In legacy mode use the glyph we get by a 1:1 mapping.
2276                                  This would make absolutely no sense with Unicode in mind,
2277                                  but do this for ASCII characters since a font may lack
2278                                  Unicode mapping info and we don't want to end up with
2279                                  having question marks only. */
2280                               tc = c;
2281                            } else {
2282                               /* Display U+FFFD. If it's not found, display an inverse question
mark. */
2283                               tc = conv_uni_to_pc(vc, 0xfffd);
2284                               if (tc < 0) {
2285                                  inverse = 1;
2286                                  tc = conv_uni_to_pc(vc, '?');
2287                                  if (tc < 0) tc = '?';
2288                               }
2289                            }
2290                    }
2291
2292                    if (!inverse) {
2293                            vc_attr = vc->vc_attr;
2294                    } else {
2295                            /* invert vc_attr */
2296                            if (!vc->vc_can_do_color) {
2297                                    vc_attr = (vc->vc_attr) ^ 0x08;
2298                            } else if (vc->vc_hi_font_mask == 0x100) {
2299                                    vc_attr = ((vc->vc_attr) & 0x11) | (((vc->vc_attr) & 0xe0) >>
4) | (((vc->vc_attr) & 0x0e) << 4);
2300                            } else {
2301                                    vc_attr = ((vc->vc_attr) & 0x88) | (((vc->vc_attr) & 0x70) >>
4) | (((vc->vc_attr) & 0x07) << 4);
2302                            }
2303                            FLUSH
2304                    }
2305
2306                    while (1) {
2307                            if (vc->vc_need_wrap || vc->vc_decim)
2308                                    FLUSH
2309                            if (vc->vc_need_wrap) {
2310                                    cr(vc);
2311                                    lf(vc);
2312                            }
2313                            if (vc->vc_decim)
2314                                    insert_char(vc, 1);
2315                            scr_writew(himask ?
2316                                       ((vc_attr << 8) & ~himask) + ((tc & 0x100) ? himask : 0) +
(tc & 0xff) :
2317                                       (vc_attr << 8) + tc,
```

```
2318                            (u16 *) vc->vc_pos);
2319                    if (DO_UPDATE(vc) && draw_x < 0) {
2320                            draw_x = vc->vc_x;
2321                            draw_from = vc->vc_pos;
2322                    }
2323                    if (vc->vc_x == vc->vc_cols - 1) {
2324                            vc->vc_need_wrap = vc->vc_decawm;
2325                            draw_to = vc->vc_pos + 2;
2326                    } else {
2327                            vc->vc_x++;
2328                            draw_to = (vc->vc_pos += 2);
2329                    }
2330
2331                    if (!--width) break;
2332
2333                    tc = conv_uni_to_pc(vc, ' '); /* A space is printed in the second
column */
2334                    if (tc < 0) tc = ' ';
2335            }
2336            notify_write(vc, c);
2337
2338            if (inverse) {
2339                    FLUSH
2340            }
2341
2342            if (rescan) {
2343                    rescan = 0;
2344                    inverse = 0;
2345                    width = 1;
2346                    c = orig;
2347                    goto rescan_last_byte;
2348            }
2349            continue;
2350    }
2351    FLUSH
            //en caso negativo, se llama a la función do_con_trol para
tratar el carácter especial.

2352        do_con_trol(tty, vc, orig);
2353    }
2354    FLUSH
2355    console_conditional_schedule();
2356    release_console_sem();
2357    notify_update(vc);
2358    return n;
2359#undef FLUSH
2360}
```

Bibliografía

Linux cross references lxr.linux.no