Keyboard.c

Autores:

Reyes Rodríguez Santana

Yeray Mendoza Quintana

Asignatura:

Ampliación de Sistemas Operativos

Curso:

2003 - 20004

Conceptos básicos

- Un terminal es cada dispositivo que permite a un usuario interactuar con una máquina.
- Los terminales se representan en forma de archivos especiales llamados “de modo carácter”.

Conceptos básicos

- **Linux tiene cinco tipos de terminales**:
  - Las consolas virtuales: que se utilizan principalmente cuando el usuario se conecta físicamente a la máquina.
  - Los pseudos terminales maestros/esclavos: que se utilizan en una conexión remota o en una ventana x-window.
  - Los puertos serie utilizados por los modems o ratón.
  - Terminales particulares (la consola, tarjetas, etc).

Conceptos básicos

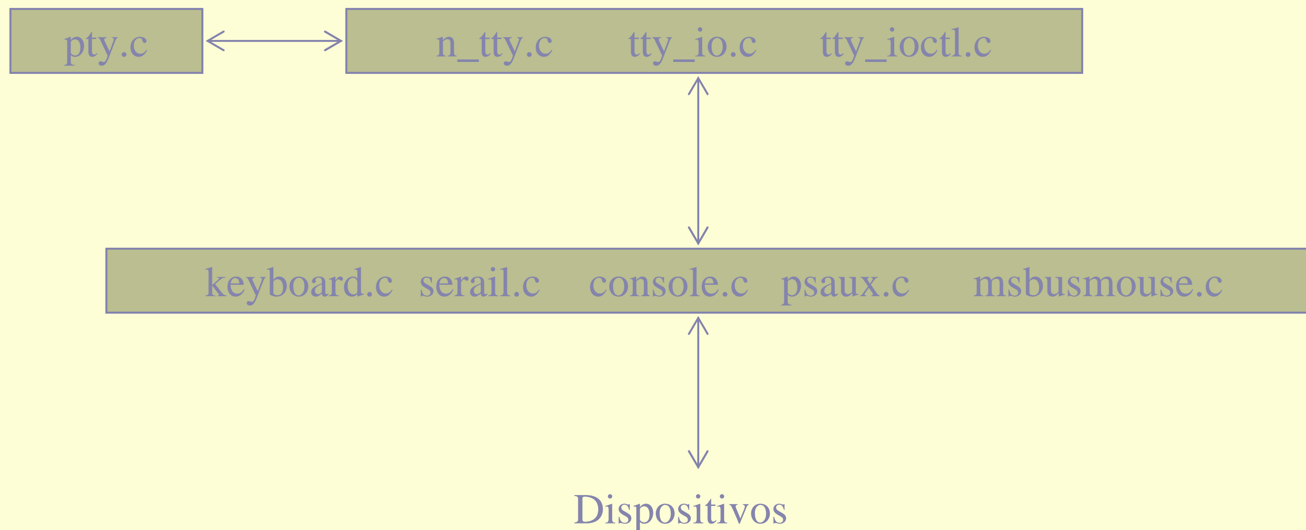
- Existen dos modos de uso de una terminal: canónico y no canónico.
- Modo canónico: la entrada de una terminal virtual se gestiona en forma de línea. Esto significa que el programa que intenta leer una línea en una terminal debe esperar que una línea completa haya sido introducida antes de poder tratarla.
- Modo no canónico: los caracteres en entrada no se tratan en forma de línea y los valores MIN y TIME se utilizan para determinar la manera como se recibe los caracteres.

Conceptos básicos

- 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.

Conceptos básicos

- 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.



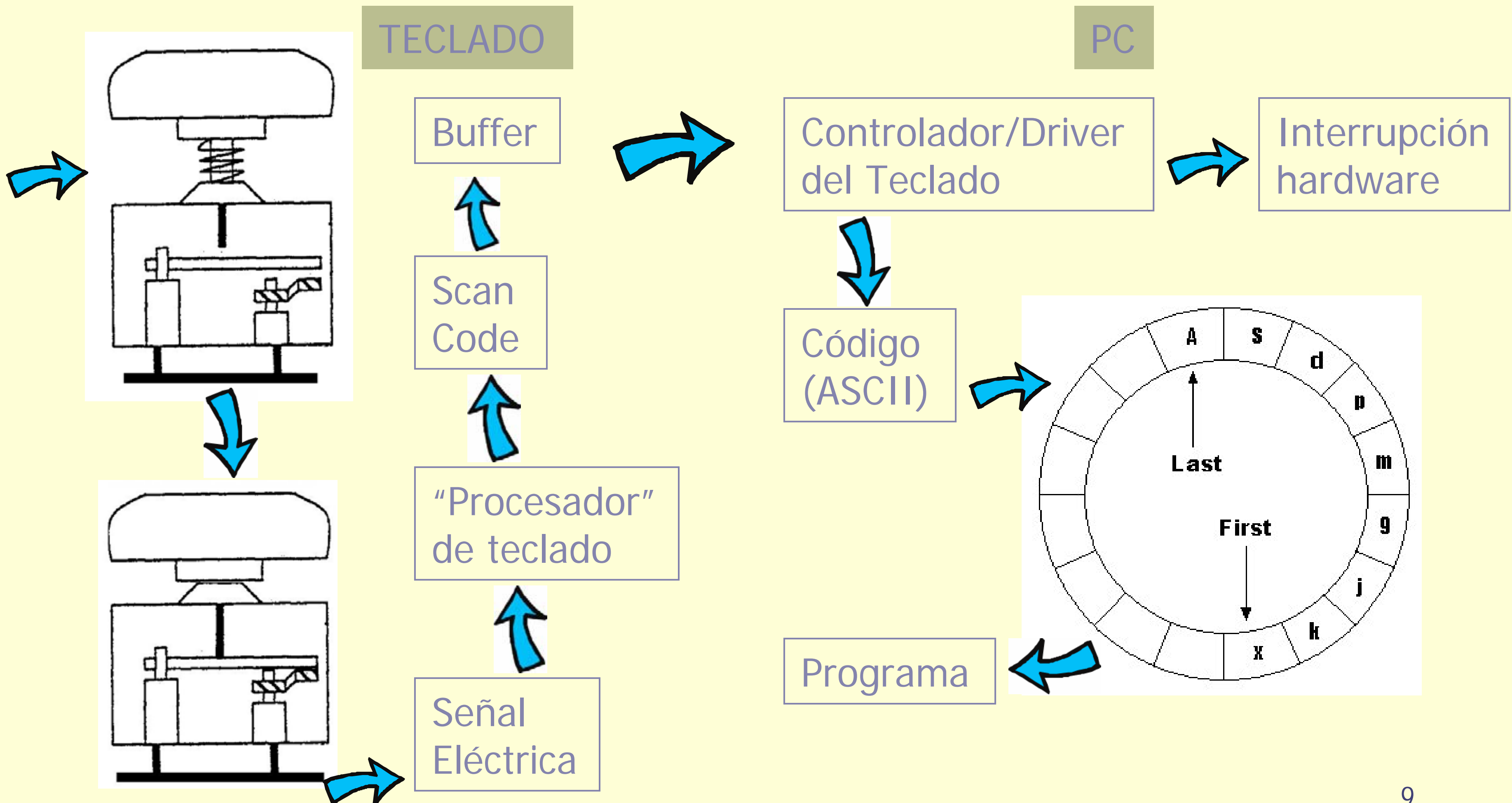
Conceptos básicos

- **Archivos básicos:**
  - tty_io.c: gestiona todas las entradas/salidas de alto nivel sobre las terminales.
  - tty_ioctl.c: gestiona la llamada a ioctl sobre una terminal y se encarga de repercutir la llamada, si es necesario, 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.

Definición de Teclado

- Interfaz de entrada estándar por excelencia.
- Formado por un conjunto de teclas.

Proceso de lectura de teclado



Scan Codes (Códigos de muestreo)

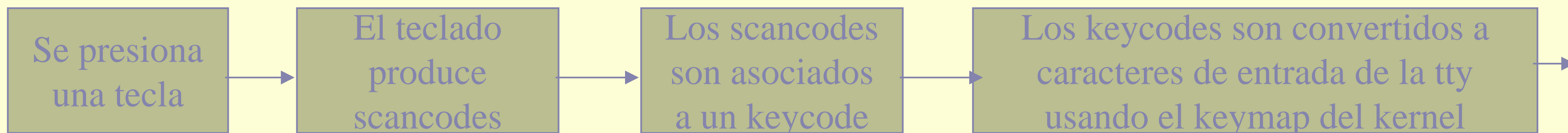
- Tipos:
  - Make Codes (pulsación).
  - Break Codes (liberación).
- Tamaño:
  - 8 bits.
  - El MSB identifica pulsación y liberación.
- Consecuencia:
  - 128 teclas distintas como máximo.
  - El Teclado detecta tanto pulsaciones como liberaciones.

Combinación de teclas

- ¿Cómo se controla la pulsación de Tecla1+Tecla2+...+TeclaN?
  - Considerando que estas teclas no sean modificadores, este control lo realiza el programa de usuario. Aunque también podría hacerlo el driver.
  - Si son modificadores combinadas con teclas no modificadoras, esto lo controla el driver del teclado.

Fichero keyboard.c

- Cuando se presiona una tecla, el carácter correspondiente no es simplemente añadido al buffer de entrada de la tty (manejador genérico de terminal), es necesario un procesamiento previo antes de que el kernel sepa cual ha sido el carácter correspondiente a la tecla pulsada.
- El proceso sería:



Primero scancodes

- Existen tres modos de scancode:
  - En modo 1 al levantar una tecla en la mayoría de los casos se produce el mismo código que en el modo 2 pero los códigos de tecla presionada son totalmente diferentes.
  - El modo 2 se usa por defecto. Los rangos de valores al presionar una tecla van desde 0x01-0x5f y el correspondiente a soltar la tecla es s+0x80.
  - En el modo 3 las únicas teclas que producen scancode al levantar la tecla son Shift, Ctrl y Alt (izq).

Asociación entre scancodes y keycodes

- Al presionar o soltar una tecla se producen secuencias de 1 a 6 bytes conocidas como scancodes, que el kernel tiene que asociar a keycodes. Para conseguir esto cada tecla se asocia a un keycode único k en un rango de 1 hasta 127, y el presionar la tecla k produce el keycode k, mientras que al soltarla produce el keycode k+128.

- Los keymap son mapas de caracteres empleados para determinar el código de carácter que se le pasa a la aplicación basándose en la tecla que ha sido pulsada y los modificadores activos en ese momento.

keyboard.c

- void to_utf8(ushort c): dado un carácter lo traduce a UTF-8
- int setkeycode(unsigned int scancode, unsigned int keycode) : permite cambiar la asociación entre scancodes y keycodes.
- int getkeycode(unsigned int scancode): permite ver la asociación entre scancodes y keycodes.
- void handle_scancode(unsigned char scancode, int down): recibe los scancode y los convierte en keycode y luego los pasa al keymap.

Keyboard.c

- void put_queue(int ch): pone el carácter en un buffer de la terminal.
- static void puts_queue(char *cp): pone una secuencia de caracteres en un buffer de la terminal.
- static void applkey(int key, char mode): introduce el codigo de la tecla pulsada en el buffer.
- static void enter(void): pone en el buffer de la terminal un 13 correspondiente a CR y si tiene activado el modo VC_CRLF además pone un 10 que corresponde a LF.

keyboard.c

- static void caps_toggle(void) y static void caps_on(void) : permiten trabajar con el bloqueo de mayúsculas.
- static void show_ptregs(void): muestra el contenido de los registros de la CPU.
- static void hold(void): para o arranca la consola.
- static void num(void) y static void bare_num(void): sirven para trabajar con el teclado numérico.
- static void lastcons(void): cambia a la última consola.

keyboard.c

- static void decr_console(void): cambia a la consola que precede a la consola actual.
- static void incr_console(void): cambia a la consola que sigue a la consola actual.
- static void scroll_forw(void): scroll hacia debajo de la consola.
- static void scroll_back(void): scroll hacia arriba de la consola.
- static void boot_it(void): envía ctrl-alt-supr.

keyboard.c

- static void compose(void): hace una combinación de Ctrl-.><,><, produciendo los caracteres especiales como la cedilla.
- static void spawn_console(void): manda una señal específica a un proceso específico.
- static void SAK(void): Se supone que mata a todos los procesos de la consola actual y resetea la consola a un estado por defecto.
- static void do_ignore(unsigned char value, char up_flag): no hace nada.

keyboard.c

- static void do_spec(unsigned char value, char up_flag): se utiliza para realizar acciones especiales.
- static void do_lowercase(unsigned char value, char up_flag): se utiliza para manejar el bloqueo de mayúsculas.
- static void do_self(unsigned char value, char up_flag): comunmente usado para teclas ordinarias, devuelve el valor pasado, posiblemente después de tratar las teclas muertas pendientes.
- static void do_dead(unsigned char value, char up_flag): usado para las “teclas muertas”, que se pueden combinar para modificar la siguiente tecla.

keyboard.c

- static void do_dead2(unsigned char value, char up_flag): es igual que la anterior pero maneja varias teclas muertas a la vez.
- unsigned char handle_diacr(unsigned char ch): se utiliza para combinar las teclas especiales (`,´) con una tecla ordinaria para obtener caracteres especiales. Si se combina con el espacio se muestra la tecla especial pulsada.
- static void do_cons(unsigned char value, char up_flag): para cambiar de consola.
- static void do_fn(unsigned char value, char up_flag): comunmente usado para las teclas de función.

keyboard.c

- static void do_pad(unsigned char value, char up_flag): comunmente usado para el teclado numérico, si está bloqueado, éste funciona como los cursores.
- static void do_cur(unsigned char value, char up_flag): comunmente usado para los cursores.
- static void do_shift(unsigned char value, char up_flag): mantiene el estado shift.
- static void do_meta(unsigned char value, char up_flag): comunmente usada para teclas ordinarias combinadas con AltL.

keyboard.c

- static void do_ascii(unsigned char value, char up_flag): combina la tecla AltL y un código numérico para producir el correspondiente carácter.
- static void do_lock(unsigned char value, char up_flag): establece el estado del correspondiente modificador de bloqueo de teclas.
- void setledstate(struct kbd_struct *kbd, unsigned int led), void register_leds(int console, unsigned int led, 852 unsigned int *addr, unsigned int mask) y static inline unsigned char getleds(void): sirven para trabajar con los leds.

keyboard.c

- static void kbd_bh(unsigned long dummy): esta rutina trabaja con la rutina de interrupción del teclado.
- int __init kbd_init(void): rutina de inicialización del teclado.

keyboard.c

```
void handle_scancode(unsigned char scancode, int down)
{
unsigned char keycode;
char up_flag = down ? 0 : 0200;
char raw_mode;
pm_access(pm_kbd);
add_keyboard_randomness(scancode | up_flag);

tty = ttytab? ttytab[fg_console]: NULL;
if (tty && (!tty->driver_data)) {
	/*Si dirver_data es falso es porque no hay ninguna consola abierta*/
	tty = NULL;
}
```

keyboard.c

```
kbd = kbd_table + fg_console;
if ((raw_mode = (kbd->kbdmode == VC_RAW))) {
        put_queue(scancode | up_flag);
/*no volvemos todavíz para mantener el vector de teclas
pulsadas y así tener los valores correctos al finalizar el modo
RAW o cuando cambiamos de consola*/
}
/*Convertimos el scancode a keyocde*/
if (!kbd_translate(scancode, &keycode, raw_mode))
        goto out;
```

keyboard.c

/*En este punto la variable ‘keycode’ contiene el keycode. Mantenemos el estado up/down de la tecla y devolvemos el keycode si estamos en modo MEDIUMRAW*/

```
if (up_flag) {
        rep = 0;
        if(!test_and_clear_bit(keycode, key_down))
                up_flag = kbd_unexpected_up(keycode);
} else
        rep = test_and_set_bit(keycode, key_down);
```

keyboard.c

```
#ifdef CONFIG_MAGIC_SYSRQ        /* Handle the SysRq Hack */
    if (keycode == SYSRQ_KEY) {
            sysrq_pressed = !up_flag;
            goto out;
    } else if (sysrq_pressed) {
            if (!up_flag) {
                    handle_sysrq(kbd_sysrq_xlate[keycode],
    kbd_pt_regs, kbd, tty);
                    goto out;
            }
    }
#endif
```

keyboard.c

```
if (kbd->kbdmode == VC_MEDIUMRAW) {
/* pronto los keycodes requerirás más de un byte */
        put_queue(keycode + up_flag);
 /* la mayoría de las clases de teclas serán ignoradas */
        raw_mode = 1;
}
```

keyboard.c

/*Repetir la tecla solo si los búfers de entrada están vacíos o los caracteres serán repetidos localmente. Esto hace usable la repetición de teclas con aplicaciones lentar y bajo cargas pesadas*/

```
if (!rep || (vc_kbd_mode(kbd,VC_REPEAT) && tty &&
    (L_ECHO(tty) || (tty->driver.chars_in_buffer(tty) == 0)))) {
        u_short keysym;
        u_char type;

        int shift_final = (shift_state | kbd->slockstate) ^
        kbd->lockstate;
        ushort *key_map = key_maps[shift_final];
```

keyboard.c

```
if (key_map != NULL) {
    keysym = key_map[keycode];
    type = KTYP(keysym);

    if (type >= 0xf0) {
        type -= 0xf0;
        if (raw_mode && !
(TYPES_ALLOWED_IN_RAW_MODE & (1 << type)))
            goto out;
        if (type == KT_LETTER) {
            type = KT_LATIN;
            if (vc_kbd_led(kbd, VC_CAPSLOCK)) {
                key_map = key_maps[shift_final ^
(1<<KG_SHIFT)];
```

keyboard.c

```
            if (key_map)
                keysym = key_map[keycode];
        }
    }
    (*key_handler[type])(keysym & 0xff, up_flag);
    if (type != KT_SLOCK)
        kbd->slockstate = 0;
    } else {
/* maybe only if (kbd->kbdmode == VC_UNICODE) ? */
        if (!up_flag && !raw_mode)
            to_utf8(keysym);
    }
} else {
```

keyboard.c

```
/*tenemos que actualizar el shift_state, existen dos posibilidades*/
#if 1
                compute_shiftstate();
                kbd->slockstate = 0;
#else
                keysym = U(key_maps[0][keycode]);
                type = KTYP(keysym);
                if (type == KT_SHIFT)
                (*key_handler[type])(keysym & 0xff, up_flag);
#endif
             }
     }
out:
     do_poke_blanked_console = 1; schedule_console_callback();
}
```