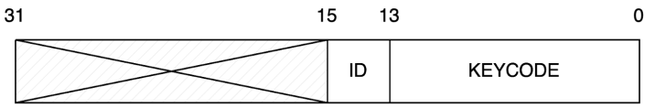
1. Sistema de E/S - Drivers

Se pide programar el driver para un teclado gamer. Este teclado es muy novedoso, ya que es mucho más grande y tiene muchas más teclas que los teclados normales, y puede ser utilizado al mismo tiempo en hasta tres juegos distintos. Esto permite que hasta tres personas a la vez puedan jugar con el mismo teclado, pero cada quién con su propia aplicación, y sin interferirse entre ellos. Además, el teclado cuenta con tres visores inteligentes que permiten mostrar información sobre el juego que se esté jugando (por ejemplo, la vida del jugador, o mensajes importantes). También permite activar, desactivar y remapear teclas, y configurar macros, para cada usuario por separado. Además de todas estas impresionantes características, tiene luces de todos los colores posibles (también configurables).

El funcionamiento del controlador del dispositivo es el siguiente: cada vez que el usuario presiona una tecla el teclado levanta la interrupción `IRQ_KEYB` y coloca en el registro `KEYB_REG_DATA` un entero de 32 bits que contiene en sus 14 bits menos significativos el código `KEYCODE` correspondiente a la tecla presionada, y en los 2 bits siguientes un identificador `P`, que vale `0` cuando la tecla debe ser recibida por todas las aplicaciones conectadas, y en caso contrario un número entre `1` y `3` que indica a qué aplicación está destinada esa tecla. Considerar que, al ser una interrupción, esta debe ser atendida en un tiempo acotado, es decir, no debe bloquearse ni quedarse esperando. Se le deberá informar al dispositivo que la tecla pudo ser almacenada escribiendo correctamente escribiendo el valor `READ_OK` en el registro `KEYB_REG_CONTROL`. En caso contrario se le deberá escribir el valor `READ_FAILED`.



Por otro lado, el dispositivo cuenta con tres direcciones de memoria (una por cada aplicación posible) `INPUT_MEM_0`, `INPUT_MEM_1`, `INPUT_MEM_2`, siendo cada una un arreglo de 100 bytes desde los que el dispositivo leerá el input de las aplicaciones. Estas direcciones serán mapeadas a memoria por el driver durante su carga en el Kernel, y se mapearán en el arreglo `input_mem`.

Cuando una aplicación se conecte al teclado, se le deberá informar al mismo escribiéndole el valor `APP_UP` en el registro `KEYB_REG_STATUS`, y el id de la aplicación correspondiente en el registro `KEYB_REG_AUX`. Cuando una aplicación se desconecte se deberá hacer lo mismo pero escribiendo el valor `APP_DOWN`.

El driver deberá ir almacenando los caracteres ASCII correspondientes a las distintas pulsaciones del teclado en un buffer, a la espera de ser leídas por las aplicaciones que se encuentren conectadas. Para ello, se cuenta con una función `keycode2ascii(int keycode)` que traduce los códigos de pulsación en su correspondiente caracter ascii.

Los procesos de usuario que deseen leer el input del teclado deberán hacerlo mediante una operación de `read()` bloqueante. Esta operación sólo puede retornar cuando haya leído la cantidad de bytes solicitados. En caso de haber más de un proceso conectado al dispositivo, cada proceso deberá poder consumir cada caracter leído de forma independiente. Ejemplo: suponer que hay tres procesos conectados, y desde el dispositivo se presionaron las teclas correspondientes a “sistemas”, en todos los casos con el identificador `P=0`. En tal caso si p0 hace un read de 2 bytes, obtendrá “si”, si p1 luego hace un read de 5 bytes, obtendrá “siste”, si luego p0 hace un read de 1 byte obtendrá “s”, y si luego p2 hace un read de 8 bytes obtendrá “sistemas”.

Además, los procesos conectados al teclado podrán utilizar la función `write()` para informarle al teclado el estado del jugador, comandar los colores del teclado, reconfigurar teclas, y varias otras funciones que provee el dispositivo.

Se cuenta con el siguiente código:

```
char input_mem[3][100];
char buffer_lectura[3][1000];
atomic_int buffer_start[3];
atomic_int buffer_end[3];
boolean procesos_activos[3];

void driver_load() {
    // Se corre al cargar el driver al kernel.
}

void driver_unload() {
    // Se corre al eliminar el driver del kernel.
}

int driver_open() {
    // Debe conectar un proceso, asignandole un ID y retornandolo,
    // o retornando -1 en caso de falla.
}

void driver_close(int id) {
    // Debe desconectar un proceso dado por parametro.
}

int driver_read(int id, char* buffer, int length) {
    // Debe leer los bytes solicitados por el proceso ''id''
}
```

```
int driver_write(char* input, int size, int proceso) {
    copy_from_user(input_mem[proceso], input, size);
    return size;
}
```

a) Implementar las funciones `driver_load`, `driver_unload`, `driver_open`, `driver_close` y `driver_read`. Implementar cualquier función o estructura adicional que considere necesaria (tener en cuenta que en el kernel no existe la *libc*). Se podrán utilizar las siguientes funciones vistas en la práctica:

```
unsigned long copy_from_user(char* to, char* from, uint size)
unsigned long copy_to_user(char* to, char* from, uint size)
void* kmalloc(uint size)
void kfree(void* buf)
void request_irq(int irqnum, void* handler)
void free_irq(int irqnum)
void sema_init(semaphore* sem, int value)
void sema_wait(semaphore* sem)
void sema_signal(semaphore* sem)
int IN(int regnum)
void OUT(int regnum, int value)
void mem_map(void* source, void* dest, int size)
void mem_unmap(void* source)
```

Para resolver la función `driver_read` es posible implementar una cola circular del modo descripto a continuación. Las variables `buffer_start[i]` y `buffer_end[i]` indican el inicio y el final del *buffer*. Al hacer una lectura, la variable `start` crece. Al hacer una escritura, la variable `end` crece. Si la variable `end` llega al final del *buffer*, debe comenzar por el principio, siempre teniendo cuidado de no pisarse con `start`. Si vale que `end+1` es igual a `start`, entonces se considera que el *buffer* está lleno. Para facilitar esta implementación, se cuenta con las funciones, que son todas atómicas:

- `int get_buffer_length(int i)`: dado un número de *buffer* devuelve la cantidad de caracteres ocupados en el mismo.
- `boolean write_to_buffer(int i, char src)`: intenta escribir un caracter en el *buffer* correspondiente, y retorna un booleano indicando si la escritura se pudo realizar.
- `boolean write_to_all_buffers(char src)`: intenta escribir un caracter en todos los *buffers*, y retorna un booleano indicando si la escritura se pudo realizar.
- `void copy_from_buffer(int i, char* dst, int size)`: lee la cantidad de caracteres indicada desde el *buffer* correspondiente, y los copia en un segundo *buffer* pasado como parámetro. Tener en cuenta que esta función no realiza ningún tipo de chequeo sobre los *buffers*, sino que simplemente copia.

Importante: El código entregado en este ejercicio debe ser correcto y no debe presentar fallas de seguridad ni de concurrencia.

2. Esquema de la solución

```
static semaphore mutex;

void driver_load(){

    // Obtener major y minor, crear device, registrarlo...
    sema_init(&mutex,1);
    for(int i = 0; i < 3; i++)
        mem_map(input_mem[i], INPUT_MEM[i], 100)
    request_irq(IRQ_KEYB, handler_irq_keyb)
}

void handler_irq_keyb(){
    int reg_data = INKEYB_REG_DATA
    int keycode = reg_data[0:14]
    int id = reg_data[14:16] // O bien hacer mascara, shiftear, etc
    bool res;
    if(id == 0)
        res = write_to_all_buffers(keycode2ascii(keycode))
    else
        res = write_to_buffer(id, keycode2ascii(keycode))

    OUT(KEYB_REG_CONTROL, res? READ_OK : READ_FAILED)
}


void driver_unload(){
    // liberar estructuras del kernel, remover device...

    for(int i = 0; i < 3; i++)
        mem_unmap(input_mem[i], INPUT_MEM[i], 100)
    free_irq(IRQ_KEYB)
}

int driver_open(){
    int id = -1;
    mutex.wait()
    for(int i = 0; i < 3; i++){
        if(!procesos_activos[i]){
            procesos_activos = true
            OUT(KEYB_REG_STATUS, id)
            OUT(KEYB_REG_AUX, APP_UP)
            return id;
        }
    }
    mutex.signal()
    return id;
}

void driver_close(int id){
    mutex.wait()
    procesos_activos[i] = false;
    // Vaciar el buffer con teclas no consumidas
    buffer_start[i] = buffer_end[i];
    OUT(KEYB_REG_STATUS, id)
    OUT(KEYB_REG_AUX, APP_DOWN)
    mutex.signal()
}

int driver_read(int id, char* buffer, int length){
    char* buf = kmalloc(length)
    // Busy waiting, pensar como evitarlo con semaforos
    while(get_buffer_length(id) < length);
    copy_from_buffer(id, buf, length);
    copy_to_user(buf, buffer, length)
    kfree(buf);
    return length;
}
```