Implementación de Driver Mouse PS/2 en el Sistema Operativo xv6

Daniel Martín Gómez

12 de Octubre de 2021

Contents

1 Introducción

En las siguientes páginas abordamos el desarrollo de un driver de mouse PS/2 para el sistema operativo xv6, y posteriormente, un cursor que se apoya en este. La funcionalidad conseguida es un cursor que se mueve por la pantalla siguiendo los movimientos del mouse. Pulsar el botón izquierdo es el equivalente a pulsar la tecla *enter*, y pulsar el derecho a pulsar *retroceso*.

El driver proporciona dos APIs distintas (aunque equivalentes) para interactuar con él. Una para el kernel de xv6 y otra para el espacio de usuario, que funciona a través de un *device file*.

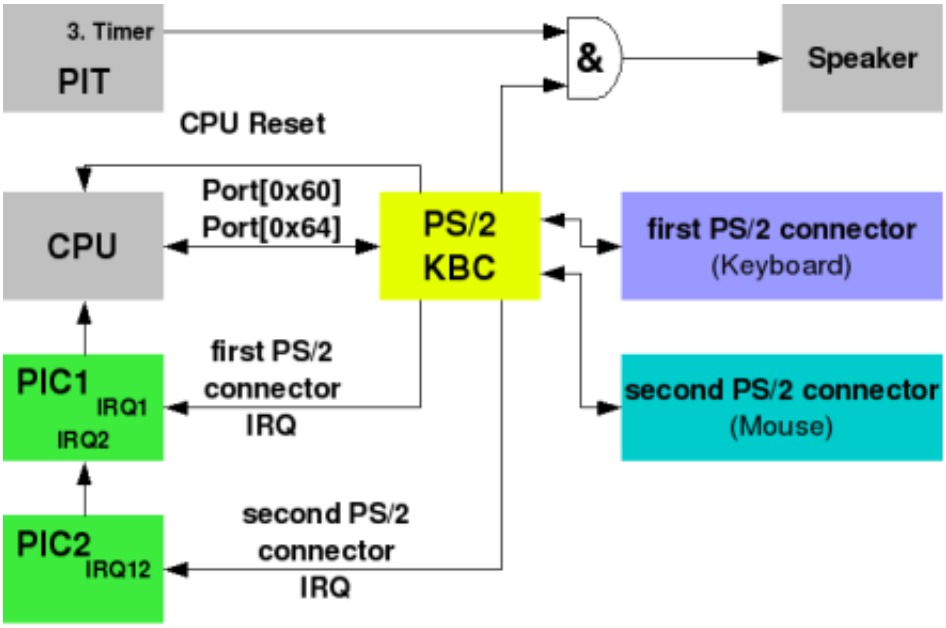
Todas las pruebas del driver se han hecho sobre la máquina virtual QEMU. Nótese además que por compatibilidad si un sistema operativo no inicializa el controlador USB (como es el caso de xv6) entonces un ratón USB simula (a ojos de xv6) ser un dispositivo PS/2 (y por tanto, funciona).

2 Introducción al sistema operativo xv6

xv6 es una reimplementación en C del sistema operativo Unix 6 (publicado en 1975), llevada a cabo por profesores del MIT. Actualmente soporta dos arquitecturas: x86 y RISC-V, estando la versión para x86 abandonada desde 2018. No obstante, será esta la que utilicemos. El sistema está compuesto por menos de 10 000 líneas de código, pero dispone de algunas características avanzadas (como multithreading y soporte multiprocesador). Desde el punto de vista de los drivers (que es con lo que vamos a trabajar) xv6 dispone únicamente de tres drivers: PS/2 Keyboard, IDE Hard Disk, Consola. El driver de consola se encarga de gestionar directamente la salida VGA, y es el único que puede ser accedido directamente desde el *userland* a través de un *device file* (*console* tiene major number 1). En el caso del IDE y del teclado solo pueden ser llamados directamente desde el kernel.

3 El controlador PS/2

(diagrama original de `https://wiki.osdev.org/"8042"_PS/2_Controller`):



El diagrama presenta la pinta que más probablemente presentará nuestro controlador PS/2 (como veremos más adelante, podría ser distinto). El controlador se encarga de hacer de intermediario entre la CPU y los dispositivos PS/2. La comunicación con este controlador no se realiza mediante I/O mapeada en memoria (como hacíamos en la asignatura de Estructura de Computadores) sino a través de puertos (es decir, instrucciones explícitas para la CPU de lectura y escritura). Esto nos lleva al siguiente punto:

3.1 Puertos de E/S

El controlador del PS/2 utiliza dos puertos distintos: El 0x60 para leer y escribir datos al controlador, normalmente relacionados directamente con los dispositivos PS/2 (como por ejemplo, leer una pulsación del teclado, etc). El 0x64 se utiliza para leer el registro de estado y para enviar comandos al controlador (comandos para resetearlo, para iniciar tests sobre los periféricos PS/2, etc). Se tratan de dispositivos de caracteres, es decir, leemos y escribimos byte a byte (utilizando las instrucciones del repertorio de x86 *inb* y *outb*). El registro de estado es el siguiente:

Bit	Nombre	Descripción
7	Parity error	0=no-error, 1=parity error
6	Time-out error	0=no-error, 1=time-out error
5	Second PS/2 port output buffer full	0=empty, 1=full
4	Unknown	Chipset specific
3	Command/data	-
2	System Flag	-
1	Input buffer status	0=empty, 1=full Debe estar a 0 antes de escribir en el puerto 0x60 o 0x64
0	Output buffer status	0=empty, 1=full Debe estar a 1 antes de intentar leer del puerto 0x60

Estos registros ($0x60$ y $0x64$) acceden en la práctica a distintos registros internos. De la tabla anterior los únicos bits que van a ser relevantes para nosotros son el 0, 1 y 5.

Los nombres de *input buffer* y *output buffer* están dados desde el punto de vista del controlador, no de la CPU/software, por lo que el *input buffer* se refiere al buffer en el que escribimos, y el *output buffer* el que leemos.

3.2 Inicializar el controlador

Vamos a desarrollar nuestro driver partiendo de la base de que el SO ya se ha encargado de inicializar el controlador PS/2. Esta es una suposición razonable ya que ya existía en xv6 un driver de teclado PS/2. Tampoco estamos comprobando que el controlador PS/2 tenga soporte de dos puertos (es decir, que soporte mouse), ya que ni siquiera estamos haciendo las pruebas de nuestro driver sobre hardware real (y los controladores PS/2 con soporte sólo para teclado son muy poco comunes). Después de un reset, el estado del mouse es: Envío automático de paquetes deshabilitado, modo ratón PS/2 standard (sin ruleta, ni botones extra), y por tanto paquetes de 3 bytes, resolución de 4 píxeles por milímetro y velocidad de transmisión de 100 paquetes por segundo (para cuando activemos el envío automático de paquetes). Lo primero que notamos es que la velocidad de transmisión es demasiado alta, teniendo en cuenta que el ojo humano no distingue más de 30 cambios por

segundo, y que el tratamiento de excepciones (que además involucran I/O) es relativamente lento, por lo que nos interesará bajar esta tasa (valores válidos son 10, 20, 40, 60, 80, 100 o 200 paquetes por segundo). Otra opción que existe es no activar el envío automático de paquetes (stream mode) y utilizar polling. En nuestro caso usaremos stream mode e interrupciones.

3.2.1 Comandos para el controlador PS/2

A través del puerto $0x64$ podemos enviar distintos comandos al controlador PS/2 (para realizar tareas como habilitar el puerto secundario (mouse), realizar tests de diagnóstico del hardware, etc). Una lista exhaustiva de estos comandos puede encontrarse en `https://wiki.osdev.org/%228042%22_PS/2_Controller#PS.2F2_Controller_Commands`. Dado que la mayoría de estos comandos tienen que ver con la inicialización, nosotros solo nos centramos en tres de ellos:

1. $0xD4$: Este comando le indica al controlador PS/2 que el próximo byte que enviemos va dirigido al puerto secundario (mouse).
2. $0x20$: Indica que deseamos leer el contenido del PS/2 Controller Configuration Byte. Dicho byte se colocará en el puerto $0x60$.
3. $0x60$: Indica que queremos escribir el contenido del PS/2 Controller Configuration Byte. Debemos escribirlo a través del puerto $0x64$.

3.2.2 PS/2 Controller Configuration Byte

Este byte tiene el siguiente formato:

Bit	Nombre	Descripción
7	-	Always 0
6	First PS/2 port translation	1=enabled, 0=disabled
5	Second PS/2 port clock	1=disabled, 0=enabled (solo si el controlador soporta 2 puertos PS/2)
4	First PS/2 port clock	1=disabled, 0=enabled
3	-	Always 0
2	System Flag	-
1	Second PS/2 port interrupt	1=enabled, 0=disabled (solo si el controlador soporta 2 puertos PS/2)
0	First PS/2 port interrupt	1=enabled, 0=disabled

Aquí realmente lo único que nos interesa es el bit 1, ya que necesitamos colocarlo al valor 1 para habilitar las interrupciones del mouse. Además, necesitamos activar la línea de interrupciones del mouse PS/2, que es IRQ 12 (distinta de la del teclado, que es IRQ 1). Si no lo hacemos lo más probable es que el controlador decida enviar las interrupciones del mouse a través de la línea IRQ 1, con los problemas que eso puede acarrear.

4 El mouse PS/2

Existen tres tipos de mouse. El mouse PS/2 estándar es aquel que dispone de sensores de movimiento, y dos botones, llamados izquierdo y derecho. Adicionalmente existen ratones con una rueda que permite hacer scroll, y (aunque menos comunes) otro último tipo que dispone de todo lo anterior pero en lugar de dos botones dispone de hasta cinco. Estos otros tipos más exóticos de mouse se comportan como un mouse estándar PS/2 hasta que el driver le pida al dispositivo que active sus funcionalidades especiales (cosa que no haremos). Nosotros nos vamos a limitar a implementar soporte para un mouse estándar, ya que el soporte para los otros tipos de mouse no aporta ningún aprendizaje extra, pero sí aumenta la complejidad del código (con comprobaciones, distinciones de casos, etc). Así mismo tampoco soportaremos "hot plug" (conexión en caliente) ya que en teoría la tecnología PS/2 no está preparada para este comportamiento (si bien en la práctica algunos sistemas operativos sí la soportan).

4.1 Comandos del mouse PS/2

Aparte de los comandos ya mencionados para el controlador PS/2, existen otros que son específicos para el mouse, pues tratan aspectos específicos del mismo, como solicitar un paquete, configurar la escala, la tasa de paquetes, etc. En la documentación `https://wiki.osdev.org/Mouse_Input#Useful_Mouse_Command_Set` se puede encontrar la lista completa, nosotros solo utilizaremos $0xF4$ (habilita el envío automático de paquetes) y $0xF3$ (permite configurar la tasa de paquetes). Un detalle importante es que antes de enviar estos comandos al controlador debemos enviar el comando $0xD4$, para indicarle que deseamos hablar con el mouse (o mejor dicho, que lo que siga será una instrucción para el mouse, ya que nosotros siempre hablamos con el controlador y nunca directamente con el periférico). El comando $0xD4$ ha de ser enviado a través del puerto $0x64$, mientras que el comando para el mouse (y cualquier byte de data adicional) debe enviarse a través del puerto $0x60$. En el caso en que tengamos que enviar un byte de información (como es el caso del comando $0xF3$) debemos enviar de nuevo el comando $0xD4$ entre $0xF3$ y el byte de data, para indicar de nuevo que corresponde al mouse. Todos los comandos específicos del mouse provocan que el controlador envíe como respuesta al puerto $0x60$ el código $0xFA$ (acknowledged, o ACK), salvo que ocurra un error. Es importante recordar que en el caso de $0xD4$ esto no es así pues es un comando del controlador, y no del mouse. A este comando el controlador no envía ninguna respuesta.

4.2 Estructura de paquete

Los paquetes devueltos por el mouse estándar tienen una longitud de 3 bytes, que deben ser extraídos uno a uno a través del puerto $0x60$. Es MUY IMPORTANTE tener en cuenta que el controlador PS/2 no nos asegura que podamos leer el paquete completo. Es decir, si desde que el controlador coloca un byte (que no sea el último) de un paquete en el output buffer y antes de que lo saquemos se produce una pulsación de teclado (por ejemplo) puede ser que el siguiente byte que se coloque en el buffer sea esta pulsación, y los bytes que nos faltaban para completar el paquete de mouse se pierdan. Por esto es crítico comprobar antes de leer cada byte, de los 3 que conforman el paquete, que el byte en cuestión llega desde el ratón (en tal caso el sexto bit (bit 5) del registro de estado está a 1).

En un paquete de 3 bytes, lo que obtenemos es la variación de la posición del ratón con respecto a la última posición reportada (en las direcciones x e y, dados como enteros de 9 bits en complemento a 2), e información sobre si los botones izquierdo, derecho y central están pulsados (en ese caso el bit correspondiente vale 1):

Byte 1

Bit 7	Bit 6	Bit 5	Bit 4
Y-overflow	X-overflow	Y-sign bit	X-sign bit
Bit 3	Bit 2	Bit 1	Bit 0
Always 1	Middle Btn	Right Btn	Left Btn

Byte 2

X-movement

Byte 3

Y-movement

5 Implementación

Se incluyen aquí algunos detalles acerca de la implementación concreta que se ha realizado.

5.1 API de eventos

Para nosotros un evento va a ser una pulsación de uno de los dos botones (izquierdo y derecho) del mouse. Además, vamos a almacenar la posición del ratón en el momento del evento, y un timestamp del mismo (que por limitaciones de xv6 tiene precisión máxima de segundos):

```
typedef struct {
        int button;
        int x;
        int y;
        struct rtcdate timestamp;
} mouse_event_t;
```

La idea será tener una lista de eventos, colocados en orden de llegada, que devolveremos cuando se nos solicite. Más específicamente:

```
typedef struct {
        // actual position of mouse
        int x;
        int y;
        // event list
        mouse_event_t events[MOUSE_MAX_EVENTS];
        int n_events;
} mouse_state_t;
```

Un detalle que hay que tener en cuenta es que tenemos que utilizar un array de tamaño fijo, ya que nuestro driver forma parte del kernel, en el que a diferencia del userland no disponemos de memoria dinámica. Todas las partes del kernel se compilan formando un único ejecutable, cuyo tamaño no debe superar los 4 MB, ya que ha de poder cargarse en una única página de memoria.

La API del driver permite obtener una estructura *mouse_state_t*, a través de las dos siguientes definiciones:

```
// returns a mouse_state_t with the current position of the mouse
// and an EMPTY list of events
mouse_state_t mouse_get_position(void);

// returns mouse current state (current state + list of unhandled
// mouse events)
mouse_state_t mouse_get_state(void);
```

Al utilizar *mouse_get_state()* el driver vacía su lista de eventos. Esto no supone ningún problema en la práctica ya que (en userland) si varios programas necesitan acceder al mouse lo normal es que lo hagan a través de alguna librería, y no accediendo directamente al driver, por lo que esta librería puede encargarse de duplicar la información, etc.

La API que acabamos de ver es a la que tiene acceso el kernel (a través de las funciones de arriba). Por otro lado, existe otra API idéntica para el espacio de usuario. Ambas son independientes, es decir, existen dos listas de eventos, de forma que vaciar la del kernel no vacía la del userland y viceversa.

El acceso desde userland se realiza a través de un device file.

5.2 Device file

Al igual que en Linux, en xv6 podemos utilizar device files para comunicarnos con los dispositivos. Disponemos de una estructura llamada *devsw* que es el análogo a la estructura *file_operations* del kernel de Linux, aunque es mucho menos potente ya que solo nos permite especificar las llamadas a *read* y *write*.

Nuestro *mouse* dispone del *major number* 2. Partimos de la base de que solo existe un ratón, así que ignoramos el *minor number* (es decir, todos los dispositivos mouse acceden al mismo).

Al leer del device file, obtenemos la posición actual del mouse, con el siguiente formato: una letra X, seguida de un entero que indica la coordenada X de la posición, y a continuación una letra Y seguida de un entero que indica dicha coordenada. Ejemplo: `X646Y-128`. Ya que no conocemos de antemano el número de bytes que va a ocupar la respuesta del device file, debemos ir haciendo lecturas hasta que el número de bytes leídos sea 0 (de esta forma el driver nos indica que ha terminado de transmitir el mensaje).

También podemos escribir al device file el comando *EVENT* (en mayúsculas), indicándole que deseamos extraer de la lista el evento más antiguo de la misma (si hay alguno, si no simplemente devolverá la posición del mouse). Es importante tener en cuenta que si enviamos este comando al driver mientras está gestionando una petición de posición, el driver no empezará a devolver información sobre el evento hasta que la petición de posición esté terminada (las peticiones no pueden interrumpirse unas a otras). El formato de un evento es: la posición del ratón en el momento del evento, seguida de una letra L o R (dependiendo de si el botón pulsado fue el *left* o el *right*), a continuación la fecha en formato año/mes/día, seguida de un @, y tras él la hora del evento (con precisión de hasta segundos). Ejemplo: `X734Y-456L2021/10/12@11:17:12`

5.3 Ejemplo de uso: Cursor

Para ilustrar la funcionalidad del driver implementaremos (utilizando la API) un cursor que se moverá por la pantalla siguiendo el movimiento del ratón,

y que capturará el botón izquierdo como una pulsación de la tecla Enter del teclado, y el derecho como la tecla retroceso (borrar).

En xv6 existe un driver de consola (con *major number* 1) que se encarga de gestionar la consola a todos los niveles: desde controlar directamente el VGA hasta gestionar el flujo de caracteres que llega desde el driver de teclado, pasando por controlar el scroll de la pantalla. Por tanto, va a ser aquí donde tengamos que colocar el código de manejo tanto del cursor como de las funcionalidades asociadas a los botones.

5.3.1 Text buffer VGA

En el modo texto de VGA (que es en el que se ejecuta xv6), la pantalla está compuesta por 25 líneas de 80 caracteres cada una (es decir, 25*80 caracteres). Estos son accesibles directamente desde el kernel (en el caso de *console.c* a través del array crt), por orden, comenzando desde la esquina superior izquierda de la pantalla, y recorriendo línea a línea (es decir, la posición 80 del array se corresponde con el primer carácter de la segunda línea). Cada posición del array tiene 2 bytes (es un *short int*). Los 8 bits menos significativos contienen el carácter ASCII que debe escribirse en esa posición. Los bits 8-11 contienen el color de frente (es decir, el color del carácter), mientras que los bits 12-15 almacenan el color de fondo (es decir, el color para la parte del cuadro donde no esté pintado el carácter). Los códigos para los 16 colores soportados pueden encontrarse en `https://github.com/dalmemail/AmayaOS/blob/master/srv/video/vga/VGA.h`.

Nosotros usaremos el color de código 7 (lightgrey), tanto para el fondo como para el frente, de forma que para situar el cursor solo tenemos que cambiar el color de la casilla en cuestión, dejando inalterado el carácter que contiene. Cuando el cursor se mueve de posición solo tenemos que devolver a la casilla los colores que tenía antes de situar el cursor (por ello deberemos guardar el color que tiene una casilla antes de modificarlo). Esto explica el siguiente código, que se encarga de colocar o quitar el cursor de una posición:

```
static void
draw_mouse_cursor(void)
{
  old_mouse_cursor_pos_color = crt[mouse_cursor_pos] & 0xFF00;
```

```
  crt[mouse_cursor_pos] = (crt[mouse_cursor_pos] & 0x00FF) | 0x7700;
}

static void
remove_mouse_cursor(void)
{
  crt[mouse_cursor_pos] = (crt[mouse_cursor_pos] & 0x00FF) |
  old_mouse_cursor_pos_color;
}
```

La función *cursorintr()* se encarga de llamar a estos procedimientos, así como gestionar la funcionalidad de los botones (si se ha producido un evento). Como el nombre sugiere es llamada cada vez que se produce una interrupción de mouse (después de que el driver haya gestionado la interrupción).

5.4 Código y resumen de cambios

El código completo está disponible en el siguiente repositorio de GitHub: `https://github.com/dalmemail/xv6-public`. Una parte del código está comentado, y la parte que no puede entenderse con los comentarios hechos en este documento a tal efecto. Los cambios realizados en el código de xv6 han sido:

1. *Makefile*: Modificado para compilar *mouse.c* como parte del kernel, y *write.c* como aplicación de usuario
2. *console.c*: Añadido el código para controlar el cursor y la funcionalidad de los botones del mouse
3. *defs.h*: Se han añadido las definiciones de las funciones de inicialización y tratamiento de excepciones del driver
4. *file.h*: Registrado el *major number* para el mouse
5. *init.c*: Se crea un *device file* para el mouse
6. *kbd.c*: Corregido un bug que provocaba que el teclado tratase interrupciones y/o paquetes procedentes del mouse
7. *main.c*: Añadida llamada a la rutina de inicialización del driver

8. *mouse.c*: Implementación del driver *per-se*
9. *mouse.h*: Definición de la API para el kernel
10. *trap.c*: Tratamiento de las interrupciones de la línea IRQ 12 (mouse)
11. *traps.h*: Definición de `IRQ_MOUSE`
12. *write.c*: Pequeño programa de usuario que permite escribir en ficheros (utilizado para poder hacer pruebas de escritura en el *device file* del mouse)

References

[1] Russ Cox et Al. *xv6: a simple, Unix-like teaching operating system*. Rev. 7, 2012. `https://pdos.csail.mit.edu/6.828/2012/xv6/book-rev7.pdf`

[2] *8042 PS/2 Controller*. `https://wiki.osdev.org/%228042%22_PS/2_Controller`, accedido a 12/10/2021.

[3] *Mouse Input*. `https://wiki.osdev.org/Mouse_Input`, accedido a 12/10/2021.

[4] *PS/2 Mouse*. `https://wiki.osdev.org/PS/2_Mouse`, accedido a 12/10/2021.

[5] *VGA Text Mode*. `https://en.wikipedia.org/wiki/VGA_text_mode`, accedido a 12/10/2021.

[6] *VGA Text Mode Colors*. `https://github.com/dalmemail/AmayaOS/blob/master/srv/video/vga/VGA.h`, accedido a 12/10/2021.

[7] *Interrupts*. `https://wiki.osdev.org/IRQ`, accedido a 12/10/2021.