Arquitectura de Computadoras

Trabajo Práctico Especial

Juan Esteban Fraser
Legajo 53100

Nicolás Martín Castaño Gomez
Legajo 53384

Instituto Tecnológico de Buenos Aires

Índice

Introducción

El objetivo del presente informe consiste en detallar la implementación de un kernel booteable que administre los recursos de hardware de una computadora y muestre características del modo protegido de Intel. Es importante mencionar que la base del trabajo práctico parte de un kernel booteable por Pure64, modificado y provisto por la cátedra. A partir de allí, se implementó:

- la administración de los recursos de hadware
- una API para que aplicaciones de usuario puedan utilizar estos recursos

Además, en el informe se detallarán las decisiones de implementación del sistema. Recordar que el sistema cuenta con dos espacios claramente separados: ***Kernel space*** y ***User space***. En cada sección, se describirán pros y contras de diseño, así como también sus funcionalidades más destacadas.

Por último, se esboza una breve conclusión sobre el trabajo realizado.

Kernel Space

El kernel space es el espacio que interactúa directamente con los módulos de hardware mediantes drivers y, al mismo tiempo, provee de funciones al user space, las cuales serán utilizadas cuando el usuario ejecute comandos vía terminal. A continuación, se detallan los drivers para los módulos de hardware utilizados en el correspondiente trabajo práctico especial.

Teclado

Descripción:

Esta sección del código se encarga de todo el manejo del teclado, es decir de comunicar el dispositivo de hardware con el resto del sistema. Este driver se encuentra desarrollado en el archivo *keyboard.c* de la carpeta de kernel e incluye tanto a su correspondiente .h (*keyboard.h*) como a un archivo de definiciones llamado *defines.h* que incluye definiciones, enums y typedefs correspondientes a distintos archivos del kernel.

Diseño e implementación:

A la hora de implementar el driver del teclado, fue necesario leer bastante al respecto de cómo funcionaba el mismo internamente y de qué forma se comunicaba dicho dispositivo de hardware con el sistema en sí. Tras dicha lectura se aprendió que el teclado envía *scan codes* que representan las interrupciones de teclado,ya sea que se apretó una tecla(KEY PRESSED) o bien que se soltó la misma(KEY RELEASED).

Se decidió contar con un arreglo bidimensional de scan codes de soltado de una tecla (ya que salvo en el caso del shift, no se requiere saber si una tecla estaba apretada), de tamaño 2 por la cantidad de teclas elegidas De esta forma, para una tecla determinada, se cuenta con el arreglo de tal forma que el scan code corresponde al índice de dicha tecla en el arreglo. Y luego para la interpretación en caracter de dicha tecla, se cuenta en una fila su valor con shift y en la otra sin shift apretado. Cabe destacar que aquellos caracteres no imprimibles fueron con un valor definido en defines.h como NOT_PRINTABLE.

Además de este arreglo de datos, existen otras dos variables globales. Ambas son estructuras definidas en el archivo *defines.h:* *kKBSTatus* y *kKBBuffer*. La primera indica en todo momento si se ha presionado la tecla shift o si capsLock está encendido. Es de suma importancia diferenciar ambos controles ya que en caso de estar encendido capsLock y apretarse shift, se cancelan entre sí en caso de caracteres alfabéticos pero no para caracteres especiales. Dicha estructura fue definida de la siguiente manera:

```
typedef struct KB_STATUS {
bool shiftEnabled;
bool capsLockEnabled;
} kKBStatus;
```

Por otro lado, la otra estructura del keyboard consiste en un buffer circular, el cual es de suma importancia para manejar los caracteres ingresados por el usuario. Dicho buffer se implementó como un arreglo de chars y cuenta con un tamaño límite definido también en el archivo de definiciones. A su vez, se definieron dos índices para facilitar la lectura por parte del sistema de los caracteres introducidos y la escritura por parte del usuario a través

del hardware y pasada al buffer. Por lo tanto ambos índices fueron definidos como readIndex y writeIndex e incluidos en la estructura propia del buffer del teclado. Dicha estructura se implementó como se detalla a continuación:

```
typedef struct KB_BUFFER {
    word readIndex;
    word writeIndex;
    unsigned char buffer[KEYBOARD_BUFFER_SIZE];
} kKBBuffer;
```

En cuanto a las funciones del driver, se cuenta con una función de inicialización del mismo ***kKBInitialize***, la cual es llamada por el kernel al iniciar el sistema para así estar preparado el mismo para el correcto uso del teclado posteriormente. La misma únicamente limpia el buffer completo antes de comenzar a usarse.

Con respecto a la interpretación de los valores propios del teclado por medio de scan codes, se cuentan con varias constantes definidas: **KEY_PRESSED, LS_PRESSED, RS_PRESSED, LS_RELEASED, RS_RELEASED, CAPS_LOCK, SPACEBAR** y **EMPTY**. La primera se trata de una máscara para saber si se trata de un MAKE o un BREAK code. La última como lo indica es la referencia en el sistema de un lugar vacío, asignado arbitrariamente como 0xFF ya que dicho valor no es alcanzado por los caracteres utilizados. Luego se tienen definidos, como sus nombres indican, los scan codes del resto de las teclas especiales de interés.

A su vez se tienen tres funciones de importancia vital para el driver:

-kKBKeyReceived: recibe un keycode o scan code y en caso de ser un caps lock o bien un shift, modifica el BufferStatus como corresponde; en caso contrario, se obtiene el valor ASCII correspondiente por medio del arreglo de scan codes de referencia. Para ir viendo en pantalla las teclas apretadas, en caso de ser un carácter imprimible, se muestran en pantalla las mismas y se las agrega al buffer mediante **kKBInsertKey.**

- kKBInsertKey: esta función se encarga de introducir el carácter en el buffer de teclado moviendo el writeIndex al lugar siguiente para la próxima escritura, implementando aquí el aspecto circular del buffer pasando de la última posición del arreglo a la primera.

- kKBGetKey: para poder realizar la correcta lectura del buffer por parte del sistema, se requiere de esta función. La misma levanta el carácter apuntado por el readIndex, limpia el contenido de la porción de buffer leída y mueve el readIndex contemplando que el buffer es circular.

Por último, se cuenta también con dos funciones menores en este driver pero que facilitan procesos para el resto del código: **isAlpha** que devuelve TRUE si se trata de un carácter alfabético y FALSE sino (notar que en defines.h se definieron elementos de tipo bool, donde TRUE = 1 y FALSE = 0 para facilitar la lectura del código escrito); y otra función **isEmpty** que verifica si se trata de un lugar vacío con el concepto de EMPTY definido en defines.h.

PC Speaker

Descripción

Esta sección del código, tanto del lado de user como de kernel, se encarga del manejo del sonido, comunicando las notas y duraciones almacenadas en el archivo binario, pasando por userspace y interrupciones de sistema para llegar a kernelspace y ser reproducido. Este driver se encuentra desarrollado en el archivo *sound.c* de la carpeta de kernel, el cual contiene únicamente la funcionalidad básica (recibir la frecuencia a reproducir). El manejo del archivo binario y la duración de cada nota recae en el archivo *sound.c* de la carpeta de user.

Diseño e implementación

La funcionalidad de sonido está dividida en dos partes. Por un lado, el lado user se encarga de levantar los datos del archivo binario, parsearlo a frecuencias y duraciones, y comunicarle dicha frecuencia a kernel mediante interrupciones de sistema. También usa la interrupción del timer tick 0 para actuar a medida que avanza el tiempo. Por otro lado, el lado kernel recibe la frecuencia a reproducir, con lo cual configura el timer tick 2 para oscilar a esa velocidad, y habilita (o deshabilita, según sea necesario) la conexión con los parlantes.

Se decidió manejar el sonido registrando un método de actualización de sonidos a la rutina de atención del timer tick 0. Al ingresar el comando play_song se carga a dos vectores las frecuencias y duraciones de cada nota de la canción, y se registra la funcion update al timer handler (el cual maneja las interrupciones del timer tick 0) a través de una interrupción de sistema. Luego, con cada llamada a update se checkea el paso del tiempo para cuando sea necesario lanzar una interrupción de sistema para cambiar la nota siendo reproducida. El comando stop_song elimina el puntero a la función update y deshabilita la conexión entre el timer tick 2 y los parlantes.

Nuestro objetivo con esta implementación fue lograr que la reproducción de sonido pase a ser un proceso asincrónico, y que el usuario pueda continuar usando el sistema. A su vez, utilizamos las interrupciones del timer tick 0 para calcular el paso del tiempo, con lo cual las canciones se reproducen a la misma velocidad sin importar la velocidad de cálculo del procesador.

Con respecto al comando beep, se decidió en su momento hacer toda su implementación en assembler. En este caso reconocemos que una mejor implementación habría sido que el beep sea una canción de una sola nota, pero por problemas en el desarrollo para hacer funcionar el sonido habíamos decidido simplificar el código lo más posible, lo cual facilitó en gran medida el debugging.

Tabla de descriptores de interrupción (IDT)

Descripción

Esta tabla contiene los descriptores que apuntan a la ubicación de hasta 256 rutinas de servicio de interrupción (interrupt handlers). Cada interrupción utilizada por el sistema debe tener una entrada propia en la IDT. Para este trabajo, las entradas de la IDT se referencian mediante instrucciones INT.

Diseño e implementación

Se decidió implementar una estructura establecida con el tipo ***IDTR***, que simula el registro de la IDT, la cual contiene una dirección base y el tamaño. Inicialmente, en el archivo *idt.c,* se inicializa la IDT mediante la función ***kIDTInitialize,*** la cual se encarga de:

- Almacenar la estructura IDT en el registro que se pasa como parámetro (rdi), el cual es la dirección de memoria de la estructura definida anteriormente.
- Deshabilitar las interrupciones
- Setear las rutinas de atención o de servicio (interrupt handlers)
- Habilitar las interrupciones de la IDT
- Permitir que el PIC se encargue de ejecutar las interrupciones

Otro archivo de gran importancia es ***interruptsHandlers.c***, el cual contiene toda la lógica asociada al manejo de las rutinas de atención, como por ejemplo:

- En caso de que se escriba, se debe llamar a la rutina de atención de teclado para que se muestre en pantalla lo escrito.
- En caso de que se ejecute la terminal, cuando se ingresa por teclado los comandos, se debe llamar a la rutina de atención de teclado para que lea del buffer lo ingresado.
- Cuando se está reproduciendo una canción, se utiliza la rutina de atención al timer tick para ir actualizando la nota siento tocada a medida que sea necesario.

Funcionalidades más destacadas

- Permitir el manejo de las interrupciones del sistema

User Space

Descripción

Es el espacio destinado a aquellas funciones que permitan que el usuario interactúe con el sistema, sin poder acceder directamente al hardware. Es decir, el usuario se va a manejar con los comandos de la consola y, al indicar un comando soportado por el sistema, internamente se genera una interrupción de sistema (int 80h), la cual invoca los servicios/rutinas de atención de las distintas interrupciones/servicios del sistema.

Diseño e implementación

Básicamente, en el archivo ***sampleCodeModule.c*** se ejecuta la función ***terminal.c,*** la cual cicla infinitamente, y se encarga de mostrar en pantalla la leyenda: "user@pixel_os:> ". De manera que, la terminal espera a recibir caracteres ingresados por el usuario cuando utiliza el teclado. Estos caracteres se irán almacenando en un buffer. Los caracteres almacenados son aquellos caracteres distintos del '\n'(salto de línea). Luego, esos caracteres se toman juntos y se los considera un posible 'nombre de comando'. Para verificar que ese nombre se trata de un comando válido, se compara este nombre contra aquellos nombres de comandos almacenados en un vector. En el caso de coincidir, se ejecutará la acción correspondiente a ese comando.

Con respecto al parseo del texto ingresado por el usuario, en el archivo *defines.h,* se define una estructura establecida con el tipo ***CommandDescriptor***, la cual contiene:

- Puntero a char que sería el texto ingresado por el usuario.
- Un puntero a función que va a permitir ejecutar la acción correspondiente a la rutina de atención de la interrupción en particular.
- Un puntero a char que sería el texto que se muestra en caso de que el usuario invoque el comando –help, el cual indica la descripción de cada comando.
- Un boolean que indica si el comando está visible o no.

La ventaja de utilizar esta estructura es que en la función ***executeCommand*** del archivo *terminal.c* facilita:

- la lógica y la operatoria de chequear si el comando ingresado es válido o no,
- ejecutar la acción correspondiente al comando utilizando el puntero a función,
- chequear en caso de ingresar un comando que acepta varios comandos, que los "subcomandos" sean los correctos. Esto ocurre en el caso del comando echo expresion_a_imprimir_en_pantalla.

Vale la pena mencionar que se desarrollaron todas las funciones vinculadas con la entrada y salida de la consola de comandos y se las puede encontrar en la biblioteca estándar: "print", "read", "strlen", entre otras.

Funcionalidades más destacadas

- Le permite al usuario interactuar desde la consola, con las distintos servicios de hardware, como por ejemplo:
    - escribir desde el teclado

- mostrar en pantalla lo escrito
- reproducir sonidos
- limpiar pantalla

Otras Consideraciones

Si bien durante el informe se señalaron aspectos positivos que sirvieron para mejorar o facilitar la implementación del sistema, hay ciertas desventajas que se mencionan a continuación:

- Se utilizaron dos cursores para poder moverse en el display, lo cual no era necesario, ya que con un cursor era suficiente, debido a que la memoria es lineal. La presencia de dos cursores complica aún más el código.
- Otro aspecto a considerar es el hecho de la validación de comandos. Si bien se parsean correctamente los comandos y, en caso de ser un comando inválido se muestra el mensaje de error correspondiente, puede ser que haya casos en los que la validación no sea tan perfecta, por ejemplo, en el caso del comando echo, el cual no se consideraron todas las variantes que puede haber al utilizarlo, ya sea teniendo en cuenta espacios o tabs('\n').

Conclusión

La realización de este trabajo práctico especial fue una tarea ardua, debido a que para la realización del mismo, fue necesario del estudio/manejo de conceptos asociados a:

- Manejo de interrupciones
- Pasaje de parámetros en assembler, utilizando una arquitectura de 64 bits
- Sistema de comunicación con los distintos módulos de hardware utilizados

Si bien implementó un sistema que garantiza el manejo de módulos de hardware, tales como el display, keyboard y parlantes, muchos otros módulos quedaron fuera de la implementación, tales como el manejo del mouse o del rtc para implementar un driver para el screensaver. Todas cuestiones asociadas al factor tiempo a la hora de desarrollar el sistema.