Arquitecturas de las Computadoras

Trabajo Práctico Especial

2do Cuatrimestre 2014

Integrantes

- Cavo, María Victoria Legajo: 53202
- Rossi, Melisa Anabella Legajo: 54265
- Zannini, Franco Michel Legajo: 54080

INTRODUCCIÓN

El objetivo de este trabajo fue la implementación de un micro kernel que administre los recursos de hardware de la computadora y muestre características de modo protegido de Intel.

El kernel cuenta con intérprete de comandos que permite al usuario probar sus diferentes funcionalidades que se encuentran detalladas en el manual del usuario.

SEPARACIÓN USER SPACE Y KERNEL SPACE

El kernel implementado se encuentra dividido en dos sectores diferenciables: el *Kernel Space* y el *User Space.*

En el *User Space* se ubican todas las funciones y programas que se utilizan para interactuar con el *kernel*. Allí podremos encontrar el *Shell* y los comandos junto con las llamadas a función correspondientes. Además allí podemos encontrar el conjunto de librerías de c.

En el *Kernel Space* encontramos los drivers que controlan el hardware así como también las se encuentran las funciones que le brinda el *Kernel* al *User Space* para poder comunicarse entre sí. Estas funciones se comunican con el kernel mediante la interrupción 80h generando las llamadas *System Calls*.

En drivers encontramos los drivers del *teclado*, del *video*, del *timer tick* y del *speaker* y en *System* encontramos las funciones que controlan las *System calls* así como también las interrupciones y excepciones.

/USER SPACE

- *commands.c y commands.asm*

En commands.c podemos encontrar la función que se encarga de parsear todos los comando y se encarga de llamar a cada una de las funciones correspondientes.

En las funciones *printidt()*, *biosinfo()* y *escalerita()*, cuyo objetivo requiere la participación de un hardware determinado, se llama a una función en Assembler que carga los paramentos deseados y produce un *System Call* mediante la generación de una interrupción (INT 80). Esta interrupción llega al *Kernel Space* y este se encarga de comunicarle a los drivers las tareas a realizar. Se decidió implementar las funciones de esta manera ya que no deberían existir en el *User Space* funciones que interactúen directamente con el hardware sino que deben utilizar al kernel como intermediario.

En commands.asm sólo se encuentran 3 funciones cuyo objetivo es provocar excepciones. Una de ellas, *_debugtest()* fue obtenida de la página:

http://neilscomputerblog.blogspot.com.ar/2012/10/single-step-debugging-explained.html

- *shell.c*

En shell.c podemos encontrar la función *shell()* que es la función que interactúa todo el tiempo con el usuario. En esta función se cuenta con dos variables char * una llamada buffer y otra cmd.

El buffer se utiliza para guardar todo lo que el usuario escribe después de *user@fmv-->*. Se tomó un tamaño fijo de buffer tomando en cuenta cual es el máximo número de caracteres a ingresar, que en este caso es 70. Este valor se puede encontrar en shell.h bajo el nombre de CMD_SIZE. En el caso que el usuario llegue a este numero máximo se imprimirá un '\n' y se parseará lo ingresado hasta ese momento.

La variable cmd se utiliza para guardar el comando sin parámetros y es lo que después se enviará a la función *parsecommand()*.

/LIB

En la carpeta lib podremos encontrar los equivalentes a las librerías de C:

- *ctype.c*: Aquí se encuentran las funciones para clasificar caracteres.
- *stdio.c*: Aquí se encuentran las funciones con operaciones de entrada y salida.
- *stdlib.c*: Aquí se encuentran las funciones de conversión de tipos.
- *string.c*: Aquí se encuentran las funciones que manejan los string.
- *math.c*: Aquí se encuentran las funciones matemáticas básicas utilizadas

- *stdarg.h*: Aquí se encuentran las declaraciones para el manejo de argumentos de cantidad indefinida. Este código fue obtenido del trabajo realizado por Federico Ramundo y Conrado Mader Blanco con su previo consentimiento.

/KERNEL SPACE

- */drivers*

- *keyboard.c*

El keyboard se decidió implementarlo generando un buffer circular. Cuando se produce una interrupción del teclado se llama a una función llamada *scancode()* a la que se le pasa el número recibido por el teclado que machea con un carácter de la tabla de caracteres que se encuentra en la parte superior de este archivo. Esta tabla fue obtenida del proyecto de Federico Ramundo y Conrado Mader Blanco con su previo consentimiento.

Una vez que tomamos el carácter representativo en esta tabla se verifica si alguna de las teclas especiales fue clickeada y a partir de esto se agrega al buffer el carácter deseado. Este buffer tiene un tamaño fijo de 15 y posee dos variables con las cuales se lo maneja. La variable head indica la posición del próximo carácter a leer y la variable tail indica cual fue el último carácter leído. Cada vez que se produce un *buffer_putchar()* se suma en 1 la variable de tail y cada vez que se realiza un *buffer_getchar()* se suma en 1 la variable de head. Se decidió realizar un buffer circular ya que consideramos que no se necesita un buffer muy extenso porque se están produciendo *buffer_getchar()* constantemente y a su vez este es modificado constantemente.

- *timerTick.c*

TimerTick.c ofrece una función *tick_wait(int)* que recibe por parámetro la cantidad de ciclos del timer Tick que debe esperar. Para ello setea en cero una variable global que cumple la función de contador y que, si el flag de que se está contando el tiempo está encendido, es incrementada con cada interrupción del timer Tick. La función se queda ciclando hasta que este contador alcance el valor esperado.

Para evitar situaciones en las que el usuario elija el mayor int posible y por alguna razón no se llegue a consultar a tiempo si se llegó al número deseado o no (se produciría overflow y se tendría que esperar el doble del tiempo como mínimo), se utilizó un flag de overflow, el cuál se apaga al inicio de la función y que en caso de detectarse que está encendido corta de inmediato el ciclo.

- *video.c*

En *video.c* podemos encontrar todas las funciones que manejan la placa de video. Cuenta con el puntero a la posición de la placa y dos variables *cursor_x* y *cursor_y*. Estas variables son modificadas constantemente por *kprint* moviendo *cursor_x* en 1 hasta que

llegue al tamaño máximo de renglón que hace que *cursor_x* vuelva a 0 y *cursor_y* se incremente en uno. Para obtener la posición exacta se utiliza la función *position()*. Video.c también ofrece otras funciones de impresión como *ksprint()* que imprime un string o *khprint()* que imprime un valor en hexadecimal. También podemos encontrar en *video.c* las funciones para imprimir el detalle de la IDT y de la BIOS.

- *speaker.c*

En *speaker.c* encontramos la función que maneja el speaker pasándole la frecuencia deseada y la cantidad de ciclos. Cuando nos comunicamos con el speaker el valor que hay que mandar para realizar una frecuencia deseada no es la frecuencia en sí, sino un 1193180/frecuencia . El resultado de esto se encuentra representado en la función por el *valor f*.

- */system*
- *bios.c*

En *bios.c* se halla el método *getBios* que recibe un puntero a un vector de punteros a char, donde dejará tres strings con la información solicitada de la BIOS.

Para hallar esta información se sabe que entre la posición 0xF0000 y 0xFFFFF está la entrada de la tabla de la SMBIOS (*System Management* BIOS) la cual tiene la siguiente estructura:

```
struct SMBIOSEntryPoint {
       char EntryPointString[4];    //This is _SM_
       uchar Checksum;              //This value summed with all the values of the table, should be 0 (overflow)
       uchar Length;                //Length of the Entry Point Table. Since version 2.1 of SMBIOS, this is 0x1F
       uchar MajorVersion;          //Major Version of SMBIOS
       uchar MinorVersion;          //Minor Version of SMBIOS
       ushort MaxStructureSize;     //Maximum size of a SMBIOS Structure (we will se later)
       uchar EntryPointRevision;    //...
       char FormattedArea[5];       //...
       char EntryPointString2[5];   //This is _DMI_
       uchar Checksum2;             //Checksum for values from EntryPointString2 to the end of table
       ushort TableLength;          //Length of the Table containing all the structures
       uint TableAddress;           //Address of the Table
       ushort NumberOfStructures;   //Number of structures in the table
       uchar BCDRevision;           //Unused
};
```

Por eso se recorre estas direcciones de memorias buscando que coincidan con "*_SM_*". Veinticuatro posiciones más adelante se podrá encontrar otra tabla que contiene la información de la BIOS y que consta de tres partes: *Header*, *Data* y *String*, localizándose en la sección String la información que se busca.

```
db 0 ; Indicates BIOS Structure Type            |
db 13h ; Length of information in bytes        | HEADER
dw ? ; Reserved for handle                     |

db 01h ; String 1 is the Vendor Name           |
db 02h ; String 2 is the BIOS version          |
dw 0E800h ; BIOS Starting Address              |
db 03h ; String 3 is the BIOS Build Date       | DATA
db 1 ; Size of BIOS ROM is 128K (64K * (1 + 1)) |
dq BIOS_Char ; BIOS Characteristics            |
db 0 ; BIOS Characteristics Extension Byte 1   |

db 'System BIOS Vendor Name',0 ;               |
db '4.04',0 ;                                  | STRINGS
db '00/00/0000',0 ;                            |

db 0 ; End of structure
```

En el segundo byte de la tabla se halla la longitud que tiene el header y la data, así que obteniendo ese dato y sumándose a la dirección de memoria en la que se estaba se llega a la sección String y se hace apuntar a cada puntero del vector a la información requerida.

El comienzo de la función *getBios* y la información necesaria para entender la localización de los strings fue obtenida de :

http://wiki.osdev.org/System_Management_BIOS

y complementada con:

http://www.dmtf.org/sites/default/files/standards/documents/DSP0134_2.8.0.pdf

de la página 21 a 31.

- *exceptions.c*

En *exceptions.c* podemos encontrar la función que se encarga de mostrarle al usuario que tipo de excepción se produjo, imprimiendo por salida estándar el nombre de la excepción y el tipo así como también generando el sonido configurado para tal excepción.

Además se decidió que por convención las excepciones de tipo *Abort*, que son las más graves, produzcan sonido dos veces.

También podemos encontrar un array de *veinte* posiciones donde cada posición representa el número de excepción y como valor encontramos la frecuencia con la que se produce el sonido. Este valor es configurable por el usuario mediante el comando *setsound* que llama a *changeFrec*.

- *idt.c*

En idt.c encontramos todas las funciones que controlan la IDT. Se encuentran la incorporación a la IDT de las excepciones con sus respectivos *handlers* asi como tambien se incorpora el *InterruptGate* de la interrupción del *teclado*, *timerTick* y *SystemCalls*. Se decidió poner las funciones en un archivo aparte por un tema de orden de código.

- *interruptions.c*

En *interruptions.c* encontramos el handler de las interrupciones. La función de *int_80*() maneja las interrupciones de *SystemCalls*. En este kernel contamos con 5 posibles *SystemCalls* representados por el argumento task.

Luego tenemos el *int_08()* que controla la interupción del timer Tick y el *int_09* que controla la interrupción del teclado.

- */asm*

En la carpeta asm podemos encontrar las funciones en assembler que se utilizan para controlar los *systemcalls*.

- *syscall.asm*

En syscall.asm encontramos las funciones de *syscall_read*, *syscall_write*, *syscall_printidt*, *syscall_makesound* y *_int_80_hand*.

Las primeras se encargan de cargar los parametros segun el tipo de tarea a realizar y generar un *SystemCall* mediante la llamada int 80h. Luego esto produce que se entre a la funcion *_int_80_hand* que se encarga de llamar a la función *int_80*() en interruptions.c con los parámetros deseados.

- *exceptions.asm*

En *exceptions.asm* encontramos los handlers de las excepciones.Lo que hacen es armar el *stack frame*, pushear el número de excepción, llamar a una función de c genérica que atiende la excepciones, imprimiendo por pantalla y generando el ruido correspondiente a cada una. Cuando termina la ejecución, desarma el *stack frame* dos veces para que al retornar no vuelva a la línea que produjo la excepción y entre en un *loop* “infinito” sino que vuelva a la función que llamó a la función que produjo la excepción.

- *pic.asm*

En *pic.asm* encontramos los *handlers* de las interrupciones así como también la inicialización de la máscara del PIC y la función *_remapPIC* que se encarga de redireccionar las interrupciones del PIC master a partir de la posición 20h y a las del PIC slave a partir de la posición 28h debido a que el PIC *master* por default las envía a partir de la posición 8h que es de uso reservado para excepciones. El PIC *slave* las envía a partir del 70h que si bien no “molesta”, nos pareció más prolijo que estuviesen a continuación de las de PIC *master*.

El proceso de redireccionar consta de cinco partes, la primera es salvar las máscaras, la segunda es avisarle a ambos PIC que se le va a agregar un offset a las interrupciones enviando un 10h a sus puertos de comandos (21h y A1h respectivamente), luego se les envía el offset deseado a estos mismos puertos, a continuación se les indica en qué IRQ se halla conectado el otro PIC y por último se restablecen las máscaras.

La información necesaria para entender este procedimiento se obtuvo de:
http://wiki.osdev.org/PIC#Initialisation

CONCLUSIÓN

A lo largo del informe se discutió la forma en que se implementaron puntos claves del trabajo. Estos puntos constituyen la aplicación de fundamentos teóricos tratados a lo largo del cuatrimestre en la materia.

Un punto importante tratado en el presente trabajo es la correcta división entre User Space y Kernel Space. Se cree que se logró, con éxito, la correcta divisón previamente mencionada.

Para concluir, se cree que se logró cumplir el objetivo del trabajo. El producto final es un micro kernel, completamente funcional que queda abierto a una posible ampliación en futuras materias.