Trabajo Práctico Especial (TPE)

Instituto Tecnológico de Buenos Aires - Arquitectura de las computadoras (72.08)

Grupo 21

Ignacio Searles
isearles@itba.edu.ar
64.536

Augusto Barthelemy Solá
abarthelemysola@itba.edu.ar
64.502

Lautaro Paletta
lapaletta@itba.edu.ar
64.449

Agustín Ronda
aronda@itba.edu.ar
64.507

4 de junio de 2024

Resumen

El presente informe trata sobre el desarrollo de un kernel que administra los recursos de hardware de una computadora y que tiene una API para interactuar con el espacio de usuario. En el espacio de usuario se desarrolló un shell que permite ejecutar diferentes módulos que tienen el objetivo de mostrar el funcionamiento del sistema.

1 Kernel

1.1 Video driver

El trabajo se desarrolló en modo video.

Además del buffer principal de pantalla, se dispuso de un buffer intermedio de pantalla. Este buffer sobreescribe el buffer principal de pantalla cuando se produce una interrupción timer tick. Todas las rutinas de video escriben a este buffer intermedio, en vez de escribir al buffer principal. Se tomó la decisión de usar este buffer intermedio para evitar efectos de "flickering" que se experimentaban.

Para el manejo de texto en la pantalla se decidió utilizar una matriz que almacena los caracteres en cada línea, junto con su color. Cuando se escribe texto a pantalla, el mismo es almacenado en está matriz, al almacenarlo en la matriz se manejan los salto de linea, tabulaciones y borrados. Se utiliza un puntero que apunta a la próxima posición en la matriz donde se debe escribir el texto. Luego de escribir el texto en la matriz se lo imprime en el buffer intermedio de pantalla, donde se maneja el scrolling del texto en la pantalla. La utilización de esta matriz es lo que permite tanto el scrolling del texto en pantalla, como el ajuste dinámico del tamaño del texto. Ambas operaciones requieren hacer un clear de la pantalla, que sin el buffer intermedio de pantalla generaba el efecto de "flickering".

La utilización de tanto el buffer intermedio de pantalla, como la matriz de caracteres, implicó que tuviesemos que mover donde empieza el código en userpace para que el kernel tenga más espacio en memoria.

Para la impresión de caracterés se utilizó una tipografía en formato bitmap. Cada caractér es de 8x16 cuadrados. Para el ajuste del tamaño de los caracterés cambiamos el tamaño

de cada cuadrado. Esto implica que podemos cambiar el tamaño del texto por múltiplos enteros. La tipografía utilizada se extrajo de `http://www.helenos.org/doc/refman/uspace-ia32/font-8x16_8c-source.html`.

1.2 Keyboard driver

Para manejar el teclado se dispuso de una queue de tecladas. Cuando se genera la interrupción de teclado (si se trata de un key press), se lo almacena en la queue. Luego mediante las diferentes routinas de teclado, cuando se solicita teclas, la queue se va consumiendo.

Se cuentan con routinas para obtener tanto las teclas como scan code o su respectivo código ASCII. Para las funciones que obtienen el valor ASCII se tiene en cuenta el CAPS LOCK y el SHIFT.

1.3 Sound driver

1.4 PIT

1.5 IDT

Para el manejo de interrupciones y excepciones se crearon diversas entradas en la IDT. Para preservar el estado de los registros, se almacena los mismos en el stack.

Debido a que varias de las interrupciones y excepciones requieren tener el estado de los registros al momento de que se dió la interrupción/excepción, se pasa como parametro a las funciones handler un puntero a struct que contiene el estado de los registros (se pasa la dirección al stack donde se almacenaron los registros para preservarlos).

1.5.1 Interrupciones de hardware

En cuanto a las interrupciones de hardware, se manejaron las interrupciones de timer tick y del teclado.

Cuando se produce una interrupción de timer tick, se reemplaza el buffer principal de pantalla por el buffer intermedio. Y cuando se produce una interrupción de teclado, se agrega la tecla presionada a la queue (los key pressed).

1.5.2 Interrupciones de software/syscalls

Modelamos el manejo de syscalls a partir de la API de Linux de 64bit. En el caso de las interrupciones de software, no se preserva rax. Mediante este registro se pueden devolver datos al usuario.

Los syscalls son la intefaz entre el userspace y el kernel. Los syscalls implementados se volcaron en la tabla 1. En el manual de usuario se lista que parametros recibe en cada registro.

rax	Syscall	Descripción
0	sys_read	lee caracteres en formato ASCII hasta llegar a un newline.
1	sys_write	imprime texto por pantalla (en la matriz de texto).
2	sys_put_text	imprime texto por pantalla en una posición absoluta.
3	sys_set_font_size	cambia el tamaño de los caracteres.
4	sys_draw_square	imprime un cuadrado.
5	sys_get_screen_width	devuelve el ancho de la pantalla.
6	sys_get_screen_height	devuelve el alto de la pantalla.
7	sys_get_time	devulve la hora actual en formato "hh:mm:ss"
8	sys_get_key_pressed	obtiene el scan code de la última tecla presionada o 0 (si no hay teclas en la queue).
9	sys_get_character_pressed	obtiene el código ASCII de la última tecla presionada o 0 (si no hay teclas en la queue).
10	sys_clear_text_buffer	limpia la matriz de caracteres.
11	sys_get_cpu_vendor	obtiene el cpu vendor.
12	sys_beep	emite un ruido a cierta frecuencia usando una onda cuadrada.
13	sys_sleep	detiene la ejecución por cierta cantidad de tiempo.
14	sys_print_registers	imprime los registros capturados del sistema.
15	sys_clear_screen	limpia el buffer de video.

Tabla 1: syscalls implementadas para el llamado de funciones de kernel desde el userspace

1.5.3 Excepciones

Cuando se produce una excepción, se imprime la información relacionada a la excepción. Se imprime el estado de los registros al momento de efectuarse la excepción, y luego se vuelve a ejecutar la Shell.

Se capturaron las siguientes excepciones: división por cero y invalid opcode.

1.6 Ejecución de la Shell desde el Kernel

Para ejecutar la shell desde el Kernel, se decidió utilizar la instrucción iretq con el stack configurado de la siguiente manera:

ret address = SHELL_CODE_ADDRESS (dirección del comienzo del código de la shell)
Code Segment = 0x8
RFLAGS = 0x202
RSP = shell_start_rsp
Stack Segment = 0x0

donde shell_start_rsp es el valor de rsp en la primera invocación a la shell. Ejecutar la shell de esta manera permite que los stack frame no se apilen pues, en caso de reiniciar la shell, RSP se resetea. Y además, permite setear los flags de Userspace con los permisos adecuados.

2 Userspace

2.1 libc

A partir de los syscalls se implementó adaptadores a las syscalls para que puedan ser llamadas desde C. A partir de estos adaptadores se implementaron varias funciones similares a las de la librería estandar de C.

Para el manejo de strings se implementó: strcmp, strlen, strcpy, itoa y atoi. Para el manejo de IO se implementó: putchar, puts, printf, getchar y scanf (con un parametro de formato).

2.2 shell

A partir de las funciones estandar implementadas, se creó una shell.

Para manejar los módulos de la shell creamos un tipo de estructura que contiene el nombre del módulo, su descripción (usado por el módulo help) y un puntero a la función correspondiente al módulo. Estas estructuras son almacenadas en un array con los módulos disponibles.

2.3 registers

Cuando se produce una interrupción de teclado, se checkea si se presiono la tecla ESCAPE. En caso de que se haya presionado, se guarda los registros (el handler de teclado recibe un struct de registros) en un struct.

Esos registros capturados pueden ser impresos por pantalla, desde user space, mediante una syscall.

Hay que tener en cuenta que la mayoría del tiempo, el procesador está ejecutando código del kernel space, pues la rutinas de video y manejo de teclado ocupan una gran parte del tiempo de ejecución. Por ello, el rip suele estar en una posición de memoria correspondiente al kernel.

2.4 eliminator

Mediante las syscalls empleadas, se implemento el juego eliminator. El mismo se puede jugar de manera individual o con un contrincante.