Informe

Trabajo Práctico Especial
72.08 Arquitectura de Computadoras

Grupo 7

Arrinda, Maria Eugenia 65580
Cragno, Antonella Belén 65542
Gonzalez Nunez, Pedro 64467

Profesores
Santiago Valles
Federico Ramos
Giuliano Scaglioni

Fecha de entrega
Martes 10 de Junio, 2025

Índice

Objetivos

El objetivo principal de este proyecto fue implementar un kernel booteable por Pure64 que administre los recursos de una computadora a partir de una base provista por la cátedra. En dicha base existe una clara distinción entre kernel space y user space. El primero se encarga de la interacción directa con los componentes de hardware, mientras que el segundo se comunica con el primero y utiliza los datos brindados por éste.

Para permitir la comunicación entre ambas partes, se armó un archivo syscalls.h en user space que indica las system calls disponibles y los argumentos que recibe cada una. Además, para facilitar su uso particularmente en las operaciones de lectura y escritura se elaboraron dos librerías basadas en la librería estándar de C: stdio y strings.

El programa principal es el intérprete de comandos implementado desde el módulo de user space. Los comandos que facilita son: help, zoomin, zoomout, registers, clear, time, divzero, opcode, echo y pongis. Estos serán detallados en la sección Shell.

System Calls

Las system calls fueron cargadas en la IDT, y pueden ser accedidas desde el user space a través de la instrucción de Assembler “INT 80h”. Las funciones reciben argumentos variables que se especifican en la siguiente tabla:

Idx	Nombre	Primer argumento	Segundo argumento	Tercer argumento	Cuarto argumento	Quinto argumento
0	sys_write	int64_t fd	const void * buf	int64_t count		
1	sys_read	int64_t fd	void * buf	int64_t count		
2	sys_get_regs	register_set_t * registers				
3	sys_get_time	time_t * time				
4	sys_delete_char	-				
5	sys_get_key_state	uint8_t * keysState				
6	sys_zoom_in	-				
7	sys_zoom_out	-				
23	sys_sleep	int64_t ticks				
20	sys_start_beep	uint32_t frequency				
22	sys_beep	uint32_t frequency	int64_t ticks			
21	sys_stop_beep	-				
30	sys_clear_screen	-				

33	sys_fill_screen	uint32_t hexColor				
31	sys_draw_circle	uint64_t pos_x	uint64_t pos_y	uint64_t radius	uint32_t hexColor	
32	sys_draw_rec	uint64_t from_x	uint64_t fromy	uint64_t to_x	uint64_t to_y	uint32_t hexColor
34	sys_draw_pixel	-				
35	sys_chars_width	-				
36	sys_chars_height	-				
37	sys_get_screen_width	-				
38	sys_get_screen_height	-				

Las system calls descritas en la tabla anterior realizan las siguientes operaciones:

- sys_write: dado un file descriptor, que en este caso 1 indica la escritura por pantalla en el color default blanco y cualquier otro valor la escritura en rojo, escribe en pantalla hasta encontrarse con el caracter '\0' en *buf*, o escribir *count* caracteres. Retorna 0.
- sys_read: escribe en *buf* los caracteres imprimibles leídos por teclado. Es bloqueante: sólo termina cuando haya leído *count* caracteres o reciba el caracter '\0'. Retorna la cantidad de caracteres que fueron leídos.
- sys_get_regs: si en algún momento de la ejecución los registros se guardaron al presionar la tecla "F1", devuelve en la estructura pasada por parámetro los valores registrados y retorna 1. Si no, retorna 0.
- sys_get_time: devuelve en la estructura pasada por parámetro los valores leídos de rtc. Retorna 1.
- sys_delete_char: se borra el último caracter que se escribió en pantalla. Si no hay caracteres por borrar, es decir, está posicionado en la esquina superior izquierda, no hace nada. Retorna 1.
- sys_get_key_state: devuelve en el arreglo pasado por parámetro una copia del estado de cada tecla del teclado. El índice se corresponde al *scancode* de cada letra [1] y almacena un 1 si la tecla está presionada y un 0 si no. Retorna 0 si *keysState* es *NULL*, y 1 si no.
- sys_zoom_in: aumenta el tamaño de los caracteres que se escriben por pantalla. Hay tres niveles posibles de zoom. Por defecto se empieza en el nivel medio. Retorna 1.
- sys_zoom_out: disminuye el tamaño de los caracteres. También maneja tres niveles. Retorna 1.
- sys_sleep: espera que transcurran *ticks* unidades de tiempo. Retorna 1.
- sys_start_beep: emite la frecuencia indicada por parámetro indefinidamente. Retorna 1.
- sys_beep: emite la frecuencia *frequency* durante *ticks* unidades de tiempo. Retorna 1.
- sys_stop_beep: detiene la frecuencia emitida por la función sys_start_beep. Retorna 1.
- sys_clear_screen: limpia la pantalla. Retorna 1.
- sys_fill_screen: dado un color en formato hexadecimal, pinta la pantalla de *hexColor*. Retorna 1.
- sys_draw_circle: dibuja un círculo de centro (*pos_x*, *pos_y*), radio *radius* y color *hexColor*. Retorna 1.
- sys_draw_rec: dibuja un rectángulo desde (*from_x*, *from_y*) hasta (*to_x*, *to_y*) de color *hexColor*. Retorna 1.
- sys_draw_pixel: pinta un píxel en la posición (*pos_x*, *pos_y*) de color *hexColor*. Retorna 1.
- sys_chars_width: retorna la cantidad de caracteres que entran en una línea horizontal de la pantalla.

- sys_chars_height: retorna la cantidad de líneas que entran en la pantalla.
- sys_get_screen_width: retorna el ancho de la pantalla.
- sys_get_screen_height: retorna la altura de la pantalla.

Drivers

Driver de video

Para comenzar a implementar el driver de video, se buscó replicar el funcionamiento de la naive console que era parte de la base del proyecto. Para lograrlo, se armaron los archivos font.c y font.h que contienen información sobre la impresión de cada carácter y se introdujeron dos variables currentX y currentY que determinan la posición actual de escritura en pantalla. Luego, se añadieron un par de funciones adicionales para la escritura de caracteres: *deleteChar*, que borra el contenido de la posición actual; el manejo de la impresión de caracteres especiales como '\n' y '\t'; y *scrollUp*, que al llegar por debajo de la última línea de la pantalla reimprime todas las líneas en una posición superior y escribe en la última línea.

A su vez, se agregaron funciones como *drawCircle*, *drawRectangle* para facilitar el dibujo de figuras geométricas métodos para acceder a las dimensiones de la pantalla, y un método *setMultiplier* que modifica una variable usada para agrandar o reducir el tamaño de los caracteres imprimidos.

Driver de Teclado

Para la implementación del driver de teclado, se decidió utilizar dos estructuras para almacenar los datos obtenidos a través de interrupción: un arreglo *buffer* de manejo circular que contiene todos los caracteres imprimibles que todavía no fueron leídos a través de la función bufferRead, y otro arreglo keysState que a través del scancode de una tecla permite saber si esta está presionada o no. Además, se incorporó el uso de dos variables: *shift_pressed* y *caps_lock_on* que indican el estado de las teclas *SHIFT* y *CAPS LOCK*, que alteran el funcionamiento del resto de las teclas.

El driver de teclado además tiene la responsabilidad de avisarle al handler de las interrupciones que la tecla "F1" fue presionada y por lo tanto debe guardar el estado de los registros previo a ser llamada la función. Esto lo hace al setear en 1 la variable externa toSave_registers_flag.

Por otra parte, se decidió que las teclas de las flechas se deben incluir en el buffer a pesar de no ser imprimible. Como no tienen representación en ascii, se les asignó los valores 17, 18, 19 y 20 ya que estos representan caracteres ascii no imprimibles que decidimos no procesar ya que no agregan a la funcionalidad del proyecto.

Driver de Sonido

Para el driver de sonido tomamos la implementación publicada en el sitio OSDev[2].

Excepciones

Para generar las excepciones de manera on demand desde la terminal de comandos se crearon dos funciones implementadas en assembler, que se invocan con sus comandos correspondientes, *throwZeroDivisionException* y *throwInvalidOpcodeException*, que utiliza la instrucción *ud2*.

Para el manejo de las excepciones en el Kernel se armó una función común a todas las excepciones que recibe por parámetro un mensaje de error que imprime junto al valor de los registros almacenados en el handler de las excepciones y espera a que el usuario ingrese *ENTER* para volver al módulo de UserSpace. Esto último consiste en guardar en el stack el nuevo rsp obtenido a través de

getStackBase, junto con el valor de la dirección de userland. Además, se desactiva *saved_registers_flag*, con la intención de que si se genera una excepción y se reinicia la ejecución del módulo de Userland, no se retengan los valores guardados de registros. Finalmente, para cargar el nuevo rsp y rip en los registros se ejecuta la instrucción “IRET”.

Shell

El intérprete de comandos lee de a caracteres por entrada estándar y los escribe en pantalla. Se considera que se terminó de ingresar un comando cuando se recibe un ‘\n’ o se alcanza el límite de 100 caracteres. Cuenta con un arreglo circular que funciona como historial en donde se almacenan todas las líneas mostradas en pantalla e indica si cada una es un comando. El historial cumple una doble función: por un lado, permite cambiar el tamaño de la fuente de todo el contenido en pantalla; por el otro, permite navegar por el historial de comandos utilizando las flechas del teclado (↓↑) mientras se ingresa un comando.

Para procesar los comandos implementamos la función *processCommands* que mediante un switch-case se encarga de identificar y ejecutar las instrucciones correspondientes a cada comando. Para ello primero se eliminan los espacios innecesarios de la cadena leía y se parsea la línea. Luego, para cada uno de ellos, en caso de ser necesario, se consulta el formato de los argumentos ingresados para que en caso de que no sean los correctos imprima en pantalla el mensaje correspondiente. Los comandos implementados son los siguientes:

1. clear: llama a la system call que limpia todos los píxeles dibujados en pantalla y retorna al ciclo de la terminal que espera que ingresen comandos.
2. divzero: genera una excepción por división por cero al llamar a *throwZeroDivisionException* (ver sección Excepciones).
3. echo: es el único comando que no parsea sus argumentos en base a las comas e interpreta todo lo ingresado luego del comando y una coma como un solo argumento que luego imprime en pantalla.
4. help: Este comando imprime en pantalla, utilizando los métodos implementados en *stdio.c,* una lista con todos los comandos disponibles y cómo utilizarlos en caso de que acepten argumentos.
5. opcode: genera una excepción por división por cero al llamar a *throwInvalidOpcodeException* (ver sección Excepciones).
6. pongis: inicializa el juego pongis (ver sección Juego Pongis Golf).
7. registers: este comando se comunica con el kernel para guardar el valor de los registros a través de una system call que indica si ya fue presionada la tecla que indica el momento en el cual guardar estos valores. Para asegurarnos de que se guarden los registros en el momento solicitado y no luego de la interrupción de teclado, en la macro *irqHandleMaster* restauramos los registros pusheados al stack mediante un *pop* para luego cargar los valores en la estructura correspondiente, y luego volvemos al estado original mediante un *push* para continuar con el flujo de atención a la interrupción como se realizaría normalmente.
8. time: este comando imprime en pantalla el tiempo local (se ejecuta en QEMU con el flag -rtc base=localtime) utilizando las funciones implementadas en las librerías auxiliares mencionadas anteriormente. Puede llamarse sin argumentos, caso en el cual imprime la fecha y la hora y su respectivo formato. También puede ser llamado con uno de los siguientes tres argumentos: con *h* (hour) imprime solamente la hora, con *d* (date) imprime solamente la fecha, y con *y* (year) imprime solamente el año.

9. zoomin: este comando permite achicar el tamaño de la letra en la pantalla llamando a la system call que cambia el valor de *multiplier* en el kernel space. Este hace que cuando se consulte el tamaño de la fuente de texto se retorna el valor real multiplicado por el *multiplier*. Como es zoom in, lo incrementa.
10. zoomout: al igual que zoomin, este comando permite cambiar el tamaño de la letra, solo que este lo reduce.

Es importante aclarar que estos últimos dos comandos sólo manejan tres niveles de zoom ya que el kernel solo permite 3 valores de *multiplier*; 0,9, 1 y 1,25. La terminal comienza con el nivel de zoom medio y los comandos no permiten agrandar o reducir el tamaño de la letra en pantalla por fuera de estos niveles.

Juego Pongis Golf

Para el juego se planteó un campo de juego del tamaño completo de la pantalla. Dentro de la parte de backend, se implementan estructuras de datos para los jugadores, la bola, y el hoyo. También se tienen varias funciones para manejar los movimientos, las colisiones, aceleraciones, etc. Por otro lado, para el frontend se hicieron funciones para dibujar las escenas, los niveles, y todos los objetos.

Finalmente, por temas de compatibilidad se puede editar gran parte de las especificaciones de los jugadores, bola, hoyo, entre otras. Agregar niveles de forma sencilla, cambiar las dimensiones de los objetos, así como cambiar tonalidades de colores.

Bibliografía

[1] https://wiki.osdev.org/PS/2_Keyboard
[2] https://wiki.osdev.org/PC_Speaker#Sample_Code.