Instituto Tecnológico de Buenos Aires

Informe trabajo práctico especial

Arquitectura de Computadoras

1er Cuatrimestre 2025

Grupo 12

Integrantes:

- Valentino Esteves (64335)
- Augusto Felipe Ospal Madero (63669)
- Santiago Cibeira (64560)

Fecha de entrega: 10/06/25

Índice

Objetivo 2

Separación User Space - Kernel Space 2

System Calls 3

- Explicación de System Calls 4

Manejo de Interrupciones 5

Drivers 5

- Driver de teclado 5
- Driver de video 5
- Driver de sonido 6

Excepciones 6

Shell 6

Juego: pongis-golf 6

Compilación y ejecución del programa 7

Inconvenientes encontrados a lo largo del TPE 7

Objetivo

Este proyecto tiene como objetivo implementar una parte de un Kernel que sea booteable mediante el bootloader **Pure64**, con el fin de gestionar los recursos de hardware de una computadora. También, se proporciona una **API accesible desde el User Space** para que los usuarios puedan interactuar con dichos recursos.

Para lograr esto, se realizó una **separación clara de la memoria** en dos regiones: una destinada al **Kernel Space**, que interactúa directamente con el hardware a través de drivers, como por ejemplo driver de teclado, driver de pantalla, etc., y otra destinada al **User Space**, que accede a los recursos mediante **system calls** brindadas por el Kernel.

Con el fin de facilitar esta interacción, se desarrolló una librería que intenta asemejar a la librería estándar de C en Linux, implementando funciones comunes como **`printf`**, **`scanf`**, **`putChar`** y **`getc`**, entre otras.

Este sistema cuenta también con un intérprete de comandos similar a una Shell que soporta diversas funcionalidades. Algunas de ellas son las que se ven a continuación:

- ➔ Una función de ayuda (que muestre los distintos programas disponibles)
- ➔ Una función para verificar el funcionamiento de las rutinas de excepción como la instrucción inválida y la división por cero
- ➔ Un comando para desplegar la hora del sistema
- ➔ Un comando que permita obtener el valor de todos los registros del procesador en cualquier momento que se desee.
- ➔ Un comando que permita graficar en modo vídeo el juego pongis-golf. Es decir, que permita jugar el juego.
- ➔ Un Comando que permite agrandar o reducir el tamaño del texto de la pantalla
- ➔ Un comando para emitir sonido en momentos que considere oportunos del juego.

Separación User Space - Kernel Space

El **kernel** se encarga de **gestionar adecuadamente los recursos** de la computadora y de **regular el acceso al hardware**, impidiendo que el usuario interactúe directamente con los dispositivos. Para ello, se ofrece una API que permite a los usuarios comunicarse con el sistema de forma segura.

Esta API está conformada por distintas **system calls**, las cuales están programadas y almacenadas dentro del **Kernel Space**, específicamente en la **tabla IDT**. Los usuarios pueden invocar estas llamadas mediante la instrucción **"INT 80h"**, enviando varios

parámetros, entre los cuales se encuentra un **File Descriptor** que identifica la función específica a ejecutar.

Con el objetivo de facilitar el uso de estos recursos desde el **User Space**, se desarrollaron diversas librerías de apoyo. Entre ellas, se destacan funciones como **`printf`, `scanf`, `putChar` y `getc`**, entre otras. Algo a destacar es que estas funciones buscan emular el comportamiento de las más comunes en la librería estándar de C.

System Calls

Para llevar a cabo la implementación de las **System Calls**, en primer lugar se configuró la **entrada 80h de la tabla IDT**, la cual fue asociada a la **rutina de atención de interrupciones** responsable de gestionar y ejecutar la llamada al sistema correspondiente. En segundo lugar, se desarrollaron las **System Calls** que se describen en la siguiente tabla:

Syscall ID	Nombre	RDI (tipo)	RSI (tipo)	RDX (tipo)	RCX (tipo)	R8 (tipo)
0x01	sys_write	uint8_t fd	const char *str	uint64_t count	-	-
0x02	sys_read	uint8_t fd	char *buffer	uint64_t count	-	-
0x04	load_registers	uint64_t *regsArray	-	-	-	-
0x06	ZoomInFont	-	-	-	-	-
0x07	ZoomOutFont	-	-	-	-	-
0x10	clearScreen	-	-	-	-	-
0x11	putPixel	uint32_t color	uint64_t x	uint64_t y	-	-
0x12	drawChar	char c	uint32_t color	uint64_t x	uint64_t y	-
0x13	drawString	const char *str	uint32_t color	uint64_t x	uint64_t y	-
0x14	drawRectangle	uint64_t width	uint64_t height	uint32_t color	uint64_t x	uint64_t y
0x15	drawDecimal	uint64_t value	uint32_t color	uint64_t x	uint64_t y	-
0x16	drawHexa	uint64_t value	uint32_t color	uint64_t x	uint64_t y	-
0x17	drawBin	uint64_t value	uint32_t color	uint64_t x	uint64_t y	-
0x18	getScreenWidth	-	-	-	-	-
0x19	getScreenHeight	-	-	-	-	-
0x20	kbd_get_char	-	-	-	-	-
0x21	drawCircle	uint64_t radius	uint32_t color	uint64_t x	uint64_t y	-
0x30	playSoundForDuration	uint32_t freq	uint32_t duration	-	-	-
0x40	sleep	uint64_t millis	-	-	-	-

Explicación de System Calls

- **sys_read:** recibe un File Descriptor fd, un buffer buf y una cantidad de caracteres count a leer. Guarda en buf a lo sumo count caracteres que estén disponibles desde la entrada estándar (STDIN). Retorna la cantidad de caracteres leídos.
- **sys_write:** recibe un File Descriptor fd, un buffer y una cantidad de caracteres count a escribir. Escribe en pantalla (STDOUT) a lo sumo count caracteres desde el buffer. Retorna la cantidad de caracteres escritos.
- **load_registers:** recibe un puntero a un arreglo donde se almacenarán los valores de los registros generales del CPU (como RAX, RBX, etc.). Llena ese arreglo con los valores actuales de los registros. No retorna valor útil.
- **zoomInFont:** Aumenta el tamaño de fuente y limpia la pantalla. Reinicia la posición del cursor.
- **zoomOutFont**: Disminuye el tamaño de fuente y limpia la pantalla. Reinicia la posicion del cursor.
- **clearScreen:** limpia completamente la pantalla. No recibe parámetros ni retorna valor.
- **putPixel:** recibe un color (uint32_t), una posición x y una posición y. Dibuja un solo píxel del color especificado en la coordenada (x, y).
- **drawChar:** recibe un carácter, un color (uint32_t) y una posición (x, y). Dibuja el carácter en pantalla en esa ubicación con el color indicado.
- **drawString:** recibe un puntero a una cadena (const char*), un color y una posición (x, y). Imprime la cadena completa en pantalla comenzando desde esa coordenada.
- **drawRectangle:** recibe el ancho, alto, color y coordenadas (x, y). Dibuja un rectángulo del tamaño y color indicados en la posición dada.
- **drawDecimal:** recibe un valor numérico, un color y coordenadas (x, y). Dibuja el número en notación decimal en pantalla en la posición especificada.
- **drawHexa:** Misma idea que drawDecimal, pero dibuja el número en formato hexadecimal.
- **drawBin:** igual que drawDecimal, pero dibuja el número en formato binario.
- **getScreenWidth:** no recibe parámetros. Retorna el ancho actual de la pantalla en px.
- **getScreenHeight:** no recibe parámetros. Retorna el alto actual de la pantalla en px.
- **kbd_get_char:** no recibe parámetros. Retorna el próximo carácter ingresado desde el teclado (STDIN), si hay alguno.
- **drawCircle:** recibe un radio, un color y una posición (x, y). Dibuja un círculo con el centro en esa coordenada.
- **playSoundForDuration:** recibe una frecuencia (Hz) y una duración (ms). Reproduce un tono con esas características.
- ***sleep*:** Detiene la ejecución del proceso durante una cantidad ***millis*** de milisegundos.

Manejo de Interrupciones

Para el manejo de interrupciones, se implementa soporte para la interrupción del **Timer Tick** y la de **Teclado**. Ambas fueron registradas en la **IDT** (Interrupt Descriptor Table), ocupando las entradas **0x20** y **0x21** respectivamente, cada una con su correspondiente **rutina de atención de interrupcion**.

El **Timer Tick** se encarga de incrementar un contador global (ticks) que permite medir el paso del tiempo en el sistema. Este mecanismo es fundamental para mantener una referencia temporal constante. A partir de este contador se implementó la función sleep(), que permite pausar la ejecución durante una cantidad específica de ticks, habilitando interrupciones durante la espera para no bloquear el sistema. Además, el Timer Tick fue configurado de forma adecuada para optimizar el rendimiento del juego, permitiendo pausas precisas y una mejor sincronización durante la ejecución.

El **keyboard_handler** se encarga de leer el scancode, actualizar el estado de las teclas modificadoras, convertir el scancode a un carácter ASCII (teniendo en cuenta las teclas modificadoras) y añadir el carácter al buffer.

Drivers

Driver de teclado

Este driver de teclado utiliza un **mapa de scancodes** a ASCII para traducir las pulsaciones de teclas a caracteres imprimibles. Mantiene el estado de las teclas modificadoras (Shift, Ctrl, Alt, CapsLock) mediante una estructura **ModifierState_t** para determinar el carácter final. Los caracteres se almacenan en un buffer circular (keyboard_buffer) para su posterior lectura. El driver distingue entre "make codes" (cuando se presiona una tecla) y "break codes" (cuando se libera una tecla) para las teclas modificadoras. El manejo de CapsLock invierte su estado al presionar la tecla, y se considera al determinar el carácter final. El código incluye una función **kbd_get_char** para leer caracteres del buffer, con potencial para protección de acceso concurrente mediante deshabilitación de interrupciones.

Driver de video

Este driver de video proporciona funciones para manipular directamente el framebuffer en **modo video**. Se basa en la información del modo VBE (Video Electronics Standards Association) para acceder y modificar los píxeles de la pantalla. Incluye funciones básicas como putPixel para dibujar píxeles individuales, drawRectangle para dibujar rectángulos, drawChar y drawString para dibujar caracteres y cadenas de texto utilizando una fuente de mapa de bits, y drawCircle para dibujar círculos. También proporciona funciones para dibujar números en diferentes bases (decimal, hexadecimal, binario) y una función clearScreen para limpiar la pantalla. El driver incluye funciones

para validar coordenadas en pantalla y asegurar que las operaciones de dibujo se realicen dentro de los límites de la pantalla.

Driver de sonido

Este driver de sonido controla el altavoz de la PC conectándolo al PIT (Programmable Interval Timer). Permite reproducir sonidos a una frecuencia definida manipulando el Output 2 del PIT y controlando los estados del altavoz a través del puerto 0x61. El driver incluye funciones para reproducir un sonido a una frecuencia dada (play_sound), silenciar el altavoz (nosound), reproducir un sonido durante un tiempo determinado (playSoundForDuration), y generar un beep simple (beep). También incluye funciones para efectos de sonido específicos, como game_thud_sound y play_boot_sound.

Las funciones play_sound y nosound las obtuvimos de la siguiente pagina de internet: https://wiki.osdev.org/PC_Speaker

Excepciones

Para el correcto manejo de las excepciones se hizo un **exceptionDispatcher,** el mismo es llamado desde assembler por un **exceptionHandler** cuando salta la excepción.

El exceptionDispatcher maneja las excepciones mostrando un mensaje de error específico, acorde con la excepción que se logró identificar, y volcando el estado de los registros en la pantalla. Las excepciones que se pueden identificar son las implementadas, que son división por cero e instrucción inválida. Cuando se detectan las excepciones, las funciones zero_division() e invalid_operation_code() se encargan de mostrar (en cada caso) los mensajes de error para cada excepción, y printRegisters() muestra el estado de los registros hasta el momento.

Shell

Se diseñó una shell, que es un intérprete de comandos que procesa las entradas del usuario mediante un bucle infinito. Esta permite ejecutar comandos predefinidos. Es decir, cuando el usuario ingresa un comando, la shell verifica si el comando ingresado es válido y, de ser así, lo ejecuta. En cambio, si el usuario ingresa un comando invalido, la shell muestra un mensaje de error.

Juego: pongis-golf

En Pongis Golf, se implementa la lógica del juego utilizando **estructuras** para representar los Mips (jugadores), la bola y el hoyo, controlando sus posiciones, velocidades y colisiones

El juego se inicia con la creación de dos Mips (jugadores), una bola y un hoyo, cada uno con sus propias características iniciales. Un menú permite al usuario seleccionar el número de jugadores (1 o 2). El juego se desarrolla en un bucle principal donde se capturan las entradas del teclado para controlar el movimiento de los Mips. Si un Mip colisiona con la bola, esta se mueve en la dirección del Mip, y se reproduce un sonido. El objetivo es meter la bola en el hoyo.

Por último, se utilizaron ciertas syscalls para poder dibujar los elementos del juego en la pantalla y reproducir efectos de sonido al golpear la bola.

Compilacion y Ejecucion del Programa

Para compilar y ejecutar el programa se deben correr los siguientes comandos:

> ./compile.sh
> ./run.sh

Inconvenientes encontrados a lo largo del TPE

- Al intentar mantener una estructura de archivos ordenada, hubieron dificultades hasta entender el funcionamiento de los Makefile
- a implementación de múltiples mapas de bits para los distintos tamaños de la fuente impresa en pantalla trajo mucho problema
- cuando la bola entra el hoyo se espera obtener una pantalla mostrando el ganador, pero solo se espera, no ocurre y no se encontró el porqué. Lo que se supone es que en algunos de los cálculos se está haciendo una división por cero (visto gracias a la correcta implementación de la exception).