Arquitectura de las Computadoras
Trabajo Práctico especial (TPE)

Informe final

Pampuri, Franco Manuel - fpampuri@itba.edu.ar
Glaubart, Jonas - jglaubart@itba.edu.ar
Migliaro, Eugenio - emigliaro@itba.edu.ar

GRUPO 5

2025 - 1C

Índice

Introducción

El desarrollo de sistemas operativos brinda una oportunidad única para aplicar conceptos fundamentales de arquitectura de computadoras, tales como administración de memoria, manejo de interrupciones, programación en bajo nivel y diseño modular. Este trabajo práctico consistió en la implementación de un sistema operativo propio, completamente funcional y booteable, utilizando como punto de partida el cargador **Pure64** en modo protegido de 64 bits (Long Mode).

El proyecto abarcó tanto el desarrollo del núcleo del sistema (kernel) como la implementación de una interfaz de usuario básica mediante una shell interactiva. Cada componente fue diseñado e implementado para correr sobre **QEMU** y posteriormente ser probado en hardware físico real, respetando una separación estricta entre kernel space y user space, y empleando llamadas al sistema (system calls) para la comunicación entre ambos entornos.

Objetivo del proyecto

El objetivo de este proyecto fue implementar una parte de un kernel, el cual es booteable mediante el software loader "Pure64", para que administre los recursos de hardware de una computadora. Además, se debe brindar a los potenciales usuarios una API para que sea posible hacer uso de dichos recursos desde el espacio de usuario (User Space).

Para lograr esto, se debió separar claramente la memoria en una sección destinada al kernel (Kernel Space) y otra al User Space. La primera interactúa directamente con el hardware mediante el uso de drivers que facilitan la comunicación con los periféricos (teclado, pantalla, etc.). Por otro lado, la segunda se comunica con estos utilizando las system calls brindadas por el kernel.

Esta implementación incluye además un intérprete de comandos, similar a una shell, que permite acceder a diversas funcionalidades del sistema. Entre estas se encuentran:

- Comando de ayuda *"help"* para listar las funcionalidades disponibles.
- Comandos para testear el manejo de excepciones como división por cero y código de operación inválido: *"invopcode" y "zerodiv"*.
- Comando *"time"* para mostrar la hora del sistema.
- Comando *"inforeg"* para obtener el estado de los registros del procesador.
- Comando *"golf"* para ejecutar el videojuego desarrollado, basado en el juego pongis-golf.
- Comando *"changeFontSize"* para modificar el tamaño del texto en pantalla.

Separación Kernel y User Space

Una de las bases fundamentales del sistema operativo desarrollado es la correcta separación entre el **Kernel Space** y el **User Space**. Esta división garantiza la seguridad, estabilidad y control del sistema, evitando que programas de usuario interfieran directamente con el hardware o con el funcionamiento interno del kernel.

El kernel tiene como finalidad administrar los recursos de la computadora, como la pantalla, las entradas del teclado y la memoria, y solo mostrar una interfaz controlada para que los programas de usuario puedan hacer uso de estos recursos. Para lograr esto, se diseñó e implementó una API basada en system calls, las cuales están completamente encapsuladas dentro del espacio del kernel.

El mecanismo elegido para invocar estas system calls fue la interrupción de software **INT 80h**. Cada system call está asociada a un identificador único, llamado *"**File descriptor**",* que se coloca en un registro específico, y puede recibir parámetros adicionales a través de otros registros. Estas interrupciones son manejadas por la tabla de descriptores de interrupción (**IDT**), que redirige correctamente el flujo de ejecución hacia la función que modela la rutina de interrupción correspondiente dentro del kernel.

Librería de Usuario

Para facilitar la interacción entre el espacio de usuario y el kernel, se desarrolló una librería propia "**usr_stdlib"**, que encapsula el uso de las system calls mediante funciones de alto nivel, permitiendo un desarrollo más cómodo y estructurado de programas de usuario. Para la misma se utilizó como base la implementación propia de la librería "stdlib" de C.

Esta librería proporciona funcionalidades agrupadas en los siguientes ejes:

Entrada/Salida estándar

- *puts, putChar, fdprintf, myprintf*: funciones para mostrar texto por pantalla, con soporte básico de formato (números, strings, caracteres, hexadecimal).
- *readChar, getChar, readLine*: funciones para lectura de caracteres y líneas desde entrada estándar.

Utilidades de sistema

- *clearScreen*: limpia la pantalla usando la system call correspondiente.
- *sleep*: detiene la ejecución del proceso por una cierta cantidad de ticks.
- *beep*: genera un sonido con frecuencia y duración configurables.
- *getTime*: obtiene y muestra la hora y fecha del sistema.

Gráficos y video

- *changeFontSize*: permite cambiar el tamaño del texto en pantalla.
- *getScreenWidth*, *getScreenHeight*: permiten consultar las dimensiones actuales de la pantalla.
- *setDrawBuffer*, *showBackBuffer*, *changeBackgroundColor*: funciones gráficas para trabajar con doble buffer y cambiar el fondo.

Excepciones y diagnóstico

- *InvalidOpCodeTest, zeroDivTest*: permiten forzar excepciones para testear su correcto manejo.
- *getRegisters*: muestra el estado de los registros del procesador (como *RAX, RIP, RFLAGS*, etc.), previamente respaldados desde el kernel.

Funciones matemáticas auxiliares

- *sin_taylor, cos_taylor, my_sqrt*: implementaciones de funciones trigonométricas y raíz cuadrada usando series de Taylor y Newton-Raphson, utilizadas por funcionalidades adicionales como el juego.

Utilidades varias

- *strcmp, strlen, strcpy, isDigit, isChar*: funciones auxiliares típicas de manipulación de cadenas y caracteres.

System Calls

Para implementar el mecanismo de llamadas al sistema (System Calls), se configuró la entrada 0x80 en la tabla de descriptores de interrupciones (IDT), redirigiendo su ejecución a una rutina del kernel encargada de manejar estas llamadas. Esta rutina identifica cada llamada a través del contenido del registro rax y, en base a este valor, ejecuta la función correspondiente dentro del kernel.
Se presenta un resumen de las system calls implementadas y sus respectivas funciones:

ID	Nombre	rdi	rsi	rdx
0	sys_read	FileDescriptor fd	char *buf	uint64_t count
1	sys_write	FileDescriptor fd	const char *buf	uint64_t count
2	sys_clear	-	-	-
3	sys_get_time	rtc_time_t *time	-	-
4	sys_sleep	uint64_t ticks	-	-
5	sys_beep	uint64_t freq	uint64_t ticks	-
10	sys_change_font_size	unsigned int size	-	-

11	sys_get_regs	uint64_t *array	-	-
12	sys_get_screen_width	-	-	-
13	sys_get_screen_height	-	-	-
14	sys_draw_circle	int[2][2] corners	uint32_t color	-
15	sys_draw_rectangle	int[4][2] corners	uint32_t color	-
16	sys_is_pressed	char c	-	-
17	sys_show_back_buffer	-	-	-
18	sys_set_draw_buffer	int buffer	-	-
19	sys_change_bg_color	uint32_t color	-	-
20	sys_move_cursor	uint64_t x	uint64_t y	

A continuación, se describen las llamadas al sistema implementadas:

- **sys_read (ID 0)**
  Permite leer caracteres desde la entrada estándar (STDIN). Recibe un FileDescriptor, un buffer (char *buf) y la cantidad máxima de caracteres a leer (count). Almacena en el buffer hasta count caracteres provenientes del teclado, y retorna la cantidad de caracteres efectivamente leídos.

- **sys_write (ID 1)**
  Escribe una cadena de texto en la salida estándar (STDOUT, STDERR). Recibe un FileDescriptor, un buffer con los caracteres a imprimir y su cantidad. Según el descriptor, se utiliza un color por defecto (blanco para salida normal, rojo para errores). Retorna la cantidad de caracteres escritos.

- **sys_clear (ID 2)**
  Limpia completamente la pantalla. No requiere argumentos y no retorna información significativa.

- **sys_get_time (ID 3)**
  Obtiene la hora actual desde el RTC (Real-Time Clock). Recibe un puntero a una estructura rtc_time_t, donde se guarda la información obtenida. Retorna un puntero a la misma estructura.

- **sys_sleep (ID 4)**
  Suspende la ejecución del proceso actual durante una cantidad determinada de ticks. Internamente habilita y deshabilita interrupciones (_sti, _cli) para permitir que otras tareas continúen su ejecución mientras el proceso duerme.

- **sys_beep (ID 5)**
  Emite un sonido mediante el speaker del sistema. Recibe como parámetros la frecuencia (freq) y duración (ticks) del sonido.

- **sys_change_font_size (ID 10)**
  Modifica el tamaño del texto que se muestra en pantalla. Recibe como parámetro el nuevo tamaño (valor entero). Este cambio afecta directamente la fuente utilizada por el driver de video.

- **sys_get_regs (ID 11)**
  Copia el estado de los registros del procesador en el momento en que ocurrió una excepción (si es que hay respaldo disponible). Recibe un puntero a un array donde se almacenan los registros. Retorna 1 si se pudo obtener el estado, 0 en caso contrario.

- **sys_get_screen_width (ID 12)**
  Devuelve el ancho actual de la pantalla. No requiere parámetros.

- **sys_get_screen_height (ID 13)**
  Devuelve la altura actual de la pantalla. No requiere parámetros.

- **sys_draw_circle (ID 14)**
  Dibuja un círculo en pantalla utilizando dos puntos (esquinas opuestas de un cuadro delimitador). Recibe un array de dos coordenadas int[2][2] y un color (uint32_t). Internamente se construyen puntos y se delega el dibujo al driver gráfico.

- **sys_draw_rectangle (ID 15)**
  Similar al anterior, dibuja un rectángulo usando cuatro puntos (esquinas). Recibe un array int[4][2] con las coordenadas y un color.

- **sys_is_pressed (ID 16)**
  Verifica si una tecla específica está siendo presionada en ese momento. Recibe un carácter y devuelve 1 si está presionado, 0 en caso contrario.

- **sys_show_back_buffer (ID 17)**
  Actualiza la pantalla mostrando el contenido del back buffer. Esto permite implementar gráficos sin parpadeo, utilizando doble buffer.

- **sys_set_draw_buffer (ID 18)**
  Cambia el buffer de dibujo activo (front o back buffer). Recibe un entero indicando el buffer deseado.

- **sys_change_bg_color (ID 19)**
  Modifica el color de fondo de la pantalla. Recibe un valor RGB empaquetado en un

- **sys_move_cursor (ID 20)**
  Modifica la ubicación del cursor.

Manejo de interrupciones

El sistema implementa interrupciones para gestionar eventos del hardware y del sistema. Se configuraron las siguientes entradas en la tabla IDT:

- **0x20**: Timer Tick
- **0x21**: Teclado
- **0x00**: Excepción por división por cero
- **0x06**: Excepción por código de operación inválido
- **0x80**: Llamadas al sistema (system calls)

La máscara del PIC maestro fue configurada en 0xFC, habilitando únicamente las interrupciones del timer y teclado. La del esclavo quedó en 0xFF.

El **Timer Tick** actualiza la noción de tiempo del sistema y permite implementar funciones como sleep. La interrupción del **teclado** guarda las teclas presionadas en un buffer, utilizado por funciones como readChar.

Las excepciones hacen backup del estado del procesador (incluyendo registros y RIP) y muestran información diagnóstica sin interrumpir la ejecución del sistema. Las **system calls** se manejan mediante la interrupción INT 80h, redirigida al "*syscall_handler"*.

Drivers

Driver de video

Se desarrolló un driver que permite imprimir texto y gráficos sobre el framebuffer en modo gráfico, partiendo del driver base provisto por la cátedra. Se integraron las fuentes Spleen 8x16 y 16x32, convertidas a C mediante *bdf2c*, lo que permitió representar cada carácter como una matriz de bits. Para el formato de la fuente utilizada se tomó como referencia la implementación propia de Linux, subida en GitHub.
El sistema mantiene la posición del cursor en pantalla, soporta caracteres especiales (**\n, \b, \t**) y cuenta con scroll automático al llegar al final. Se agregó una funcionalidad para cambiar el tamaño de fuente en tiempo de ejecución y un mecanismo de **doble buffer** para evitar parpadeos: el sistema puede dibujar en un buffer oculto y luego copiarlo a pantalla.
Además, se implementaron primitivas gráficas como *drawRectangle* y *drawCircle*, que fueron utilizadas tanto por el juego como por otras funciones de user space.

Driver de teclado

El teclado es manejado mediante interrupciones. Cada tecla presionada se traduce utilizando un keymap (ps2_make_keymap) que contempla versiones con y sin modificadores (Shift, AltGr, Caps Lock). Las teclas son almacenadas en un **buffer circular FIFO** para ser leídas por funciones de entrada estándar.
Se incorporaron funciones auxiliares como *isCharPressed*, usada en el juego para leer entradas en tiempo real, y una funcionalidad especial que guarda los registros cuando se presiona la tecla Ctrl, útil para testing y diagnóstico.

Driver de sonido

Se implementó un driver para controlar el speaker de la PC utilizando el canal 2 del **PIT (Programmable Interval Timer)**. El sonido se genera alternando el estado del puerto `0x61` a una frecuencia determinada, lo que produce una

vibración audible. La función `playSound` configura el timer con la frecuencia deseada, y `mute` detiene el sonido. La syscall `beep` encapsula esta funcionalidad, recibiendo frecuencia y duración como parámetros.

Excepciones

El sistema operativo maneja dos excepciones del procesador: **división por cero** e **instrucción inválida**. Estas están configuradas en la IDT en las entradas 0x00 y 0x06 respectivamente, y redirigen la ejecución a rutinas específicas implementadas en Assembly, que luego llaman a una función común en C (exceptionDispatcher).

Cuando ocurre una excepción, se realiza un respaldo de 18 registros del procesador (registros generales, RIP y RFLAGS), los cuales se imprimen en pantalla en formato hexadecimal. Esta información es clave para diagnosticar el estado del sistema al momento del error.

Cada excepción muestra un mensaje identificando el tipo de error:

- Zero division exception occurred.
- Invalid opcode exception occurred.

Después de mostrar los registros, el sistema ejecuta una rutina de reinicio (reset) que lo devuelve a un estado funcional sin necesidad de reiniciar toda la máquina. Esto permite continuar usando el sistema luego de la excepción, cumpliendo con lo requerido por la consigna.

Shell

El intérprete de comandos se implementó como un loop infinito basado en funciones de la librería de usuario. Primero se lee una línea ingresada por el usuario, caracter por caracter, permitiendo mostrar en pantalla cada entrada a medida que se recibe.

Una vez completada la línea (detectando el **\n**), se separa el comando del posible parámetro y se compara con el conjunto de comandos disponibles. Si el comando es reconocido, se ejecuta la función correspondiente mediante un arreglo de punteros a función. En caso contrario, se informa al usuario que el comando es inválido.

La shell ofrece los siguientes comandos: *help, time, clear, golf, inforeg, zerodiv, invopcode y changeFontSize*. Estas funciones permiten, por ejemplo, probar excepciones, mostrar registros del procesador o iniciar el videojuego implementado.

Todas las funciones que representan comandos pertenecen a la librería de usuario, ya que requieren acceso a recursos del sistema mediante llamadas al kernel.

Juego: Pongis Golf

Se desarrolló un videojuego llamado **Pongis Golf**, el cual fue utilizado para demostrar el uso de gráficos, entrada de usuario en tiempo real, y funcionalidades físicas dentro del sistema operativo implementado. El juego se ejecuta mediante el comando **golf** desde la shell.

Objetivo y funcionamiento

El juego está inspirado en una mezcla entre Pong y minigolf. Participan uno o dos jugadores, cada uno controlando una entidad compuesta por un círculo y una flecha direccional. El objetivo principal es dirigir una pelota hacia un hoyo en pantalla y encestarla antes que el oponente. El primero en hacerlo gana el nivel. Los jugadores podrán colisionar entre ellos y con las pelotas del otro, por lo que se agrega otro objetivo: no solo será embocar, sino que impedir que el rival lo haga antes.

Controles

Cada jugador posee un conjunto de teclas para mover su entidad:

- **Jugador 1**: W, A, S, D
- **Jugador 2**: I, J, K, L
  Además, puede presionarse **Q** para salir del juego o **Espacio** para pasar al siguiente nivel.

Niveles

El juego contiene tres niveles:

- **Nivel 1**: un hoyo central con tamaño moderado.
- **Nivel 2**: un hoyo desplazado, más pequeño (mayor dificultad).
- **Nivel 2**: Mismo aumento de dificultad que el nivel 2. Los jugadores comienzan más alejados del hoyo.

Cada nivel se inicia con una posición predeterminada para los jugadores y las pelotas. Al anotar un gol, se muestra un mensaje con el jugador ganador y su puntaje.

Motor físico

Cada jugador y pelota es modelado como una **entidad física** (*physicsEntity*) con velocidad, aceleración, fricción y colisiones. Las entidades rebotan contra los bordes de la pantalla y entre sí. Se utiliza un sistema simple de detección y resolución de colisiones entre círculos, con restitución configurable.

El juego se ejecuta en modo **doble buffer** para evitar parpadeos, y utiliza primitivas gráficas como círculos y rectángulos, definidas mediante coordenadas proporcionales a la pantalla, lo cual permite adaptarse a distintas resoluciones.