Instituto Tecnológico de Buenos Aires

Informe de Trabajo Práctico Especial

Arquitectura de Computadoras

2do Cuatrimestre 2025

Grupo 20

Integrantes:

- *Matias Holgado (64727)*
- *Santiago Cibeira (64560)*
- *Lucas Rodriguez Nahuel (63335)*

Fecha de entrega: 05/11/2025

Índice

Introducción

El presente informe documenta el diseño y la implementación de un kernel básico para x86-64, desarrollado como parte del trabajo práctico especial de Arquitectura de Computadoras. El objetivo fue construir un sistema operativo booteable bajo Pure64, capaz de administrar los recursos esenciales del hardware y de ofrecer una API para aplicaciones de usuario básicas, con una separación clara entre kernel space y user space.

La prioridad del proyecto fue demostrar, de forma práctica, los mecanismos de protección y separación entre el núcleo y los programas de usuario, implementando las funciones mínimas definidas en la consigna: control de drivers, sistema de interrupciones, manejo de excepciones, y una shell que permita demostrar todas las funcionalidades implementadas.

La mayor parte de las decisiones de diseño estuvieron orientadas a lograr un sistema funcional, modular y fácil de testear tanto en QEMU como booteado en nuestra PC, manteniendo la simplicidad, funcionalidad y escalabilidad.

Estructura

El sistema se compone de dos partes principales: el kernel y el espacio de usuario.

El kernel es responsable de manejar el hardware (teclado, video, sonido) a través de drivers propios, y también se encarga de recibir y procesar interrupciones, excepciones y las llamadas a sistema (syscalls) que hacen los programas de usuario. Toda la lógica para interactuar con los dispositivos está encapsulada ahí.

Por otro lado, en el espacio de usuario se encuentran tanto la shell como los programas que pueden lanzar los usuarios, como por ejemplo el juego Tron. Estos programas nunca interactúan directamente con el hardware, sino que lo hacen a través de una API que diseñamos, la cual está basada en llamadas a sistema al estilo de Linux (por ejemplo, usando funciones como read, write o drawRect). Esta API funciona utilizando una interrupción de software, que permite que el User space pida servicios al kernel de forma controlada y sin estar accediendo directamente al hardware.

Al arrancar el sistema se inicializan todos los drivers y estructuras necesarias, se deja listo el entorno gráfico y la shell. Desde ahí, el usuario puede ingresar

comandos para interactuar con el sistema, consultar el estado del hardware, realizar pruebas diseñadas para **medir el rendimiento del sistema** bajo distintos entornos, ejecutar pruebas de excepciones, cambiar el tamaño de la fuente, o lanzar programas como el Tron, todo usando las funciones que ofrece el kernel a través de su API.

Cada driver está en su propio archivo, la lógica de las syscalls se gestiona en un módulo separado, y los programas de usuario están organizados para que sea sencillo sumar nuevas aplicaciones en el futuro sin tener que modificar el núcleo del sistema, adicionando que contamos con una librería muy básica en C que permite usar funciones sin necesidad de saber que syscalls se están utilizando y cómo están implementadas.

Comunicación entre usuario y hardware

Una de las claves del sistema es que los programas de usuario nunca acceden directamente al hardware, sino que todo pasa sí o sí por el kernel. Para eso, armamos una API que funciona mediante llamadas al sistema (“syscalls”), parecidas a las de Linux. La comunicación se hace a través de la interrupción 80h. Cuando un programa quiere hacer algo (por ejemplo, mostrar algo en pantalla, leer una tecla, o dibujar un rectángulo), hace una syscall, el control pasa al kernel, y el kernel decide si permite o no la acción.

Cada syscall tiene un número y una lista de parámetros (por ejemplo, un puntero a un buffer, una cantidad de caracteres, un color, etc), y hay un protocolo para pasar esa información usando los registros de la CPU. El kernel se encarga de validar lo que pide el usuario y de interactuar con el correspondiente driver si hace falta.

De esta forma, permitimos que los programas de usuario simplemente usen funciones de biblioteca, sin saber el detalle del funcionamiento.

El motivo de hacer esto es, en primer lugar, que el sistema sea seguro ya que el kernel se encarga de validar todo lo que un programa de usuario pide antes de efectivamente realizarlo. Si el programa de usuario tuviera acceso directo podría generar problemas en el funcionamiento total del sistema operativo. Además de esto, si en algún momento se quisiera cambiar el hardware los programas de usuario no deberían modificarse para seguir funcionando.

Las funciones de las syscalls tienen como primer parámetro su número, para que cuando desde userland se hace la interrupción, y de ahí llega al handler de syscalls, los parámetros lleguen correctamente sin modificar ningún registro a mano.

- **syscall_write:** La implementación de esta syscall simplemente invoca a la función writeString, pasando el puntero al string y la longitud. En dicha función se hacen los chequeos que corresponden. Al escribir cada carácter, el driver de video se encarga de actualizar el framebuffer directamente y manejar el cursor y etc.

- **sys_read:** La syscall lee únicamente desde el teclado (STDIN). La función recorre la cantidad pedida, pidiendo caracteres uno por uno al driver, que a su vez saca datos de su buffer circular. Si el buffer está vacío (no se apretó ninguna tecla), se termina la lectura.

- **syscall_clear_screen:** Implementada como un wrapper directo a clearScreen(), que recorre el framebuffer poniendo todos los píxeles en cero (negro) y resetea las coordenadas del cursor. Se decidió hacerlo así para que cualquier syscall (desde shell o juegos) pueda dejar la pantalla limpia, independientemente del contexto.

- **get_registers:** Recibe un puntero a un buffer en userland y llama a la función save_registers, que copia los registros generales del procesador en ese espacio. Se implementó esta syscall para poder mostrar el estado en comandos de la shell o ante excepciones.

- **get_time:** Lee los registros del RTC directamente por puerto, usando las instrucciones outb e inb para seleccionar cada campo de la hora. El código espera activamente a que el RTC no esté actualizando (bit 7 en 0x0A), y lee campo por campo (segundos, minutos, horas, día, mes, año).

- **play_sound:** Llama a playBeep(frequency) para inicializar el PIT y habilitar el speaker. Para esperar la duración deseada, convierte los milisegundos recibidos en “ticks” de la función de sleep del kernel. Luego llama a stopBeep() para apagar el sonido. Hubo que ajustar los límites mínimos (por debajo de 25 ms se fuerza un solo tick) para evitar que no suene por tiempo muy corto.

- **change_font_size:** Llama a setScale(new_size) en el driver de video, que cambia la variable de escala de la fuente del driver de video. Como detalle, si se pasa un valor inválido (cero o negativo), no se hace nada.

Gestión de interrupciones y excepciones

El kernel está preparado para manejar tanto interrupciones de hardware (teclado y timer) como excepciones del procesador (división por cero e instrucciones

inválidas). Cada una de estas situaciones se maneja con una rutina específica que definimos y se encuentra cargada en la tabla de interrupciones IDT.

En el caso de las interrupciones, configuramos el PIC (controlador de interrupciones programable) para habilitar solo las que realmente necesitamos: el timer y el teclado. El timer se usa, por ejemplo, para poder implementar la función sleep y controlar el tiempo en juegos. La interrupción del teclado alimenta un buffer circular donde se almacenan las teclas presionadas, que después se pueden leer tanto desde la shell como desde otros programas.

Para las excepciones, se implementaron handlers que manejan las situaciones de división por cero o ejecución de una instrucción no permitida. Cuando ocurre una excepción, el kernel muestra en pantalla un mensaje informando qué pasó y el estado de los registros en ese momento. Después de eso, el sistema se recupera y devuelve el control a la shell, permitiendo seguir usando el sistema normalmente.

Esta estructura hace que cualquier evento inesperado quede bajo control del kernel, evitando que un error en userland pueda congelar o romper el sistema completo. Además, poder ver el estado de los registros en tiempo real puede ser útil para encontrar problemas y entender el comportamiento del sistema a bajo nivel.

Diseño de drivers: abstracción y control

Todos los dispositivos que usamos (teclado, video, sonido) están manejados por drivers que implementamos en el kernel. La idea fue siempre abstraer lo más posible la interacción con el hardware, para que desde userland todo se vea como funciones sencillas y no haya que preocuparse por los detalles internos de hardware.

- **Video:**

  El driver de video es el encargado de dibujar tanto texto como gráficos en pantalla. Arranca inicializando la resolución usando VESA, y después ofrece funciones básicas como escribir caracteres (usando una fuente bitmap) y dibujar rectángulos de cualquier color, que son la base de los gráficos en los juegos. Se agregó una función para cambiar el tamaño de la fuente, dado que la información de la escala está guardada en dicho driver. Se agregaron otras funciones como escribir strings, limpiar pantalla, y otras funcionalidades útiles que se podrían requerir.

- **Teclado:**

Para el teclado, implementamos un driver que toma los datos crudos que llegan por interrupción y los traduce a caracteres ASCII, manejando casos especiales como shift, bloqueo de mayúsculas y teclas no imprimibles. También se ofrecen shortcuts (únicamente ctrl+l implementado). Se usó un buffer circular para las teclas, así cualquier programa puede leer de ahí y no se pierden datos aunque vengan muchas entradas seguidas.

- **Sonido:**

  El driver de sonido permite usar el speaker de la PC a través del PIT. Implementamos la función playBeep, que configura la frecuencia y habilita el speaker por un tiempo determinado, y la función stopBeep, que lo apaga. El código accede directamente a los puertos del PIT y del speaker usando instrucciones asm (outb e inb).

  Para que funcione bien en QEMU hubo que probar varios parámetros y agregar compatibilidad con la emulación de audio (por problemas de compatibilidad de nuestras pcs).

  En todos los casos, los drivers están separados en módulos distintos, con una interfaz clara hacia el resto del kernel y protegiendo el acceso directo a los registros de hardware. Esto hace que el sistema sea mucho más estable y más fácil de modificar si en algún momento hay que cambiar el manejo de algún dispositivo.

Interfaz de usuario: shell y utilidades

La forma de interactuar con el sistema es a través de la shell que implementamos. Al arrancar el sistema, la shell queda esperando comandos del usuario. El usuario puede escribir distintos comandos para probar o utilizar las funciones del kernel.

Entre los comandos disponibles está la función de ayuda (que muestra la lista de comandos), herramientas para probar las excepciones (división por cero e instrucción inválida), ver la hora del sistema, consultar el estado de los registros, cambiar el tamaño de la fuente de la pantalla, lanzar aplicaciones como el juego Tron. También existen funciones para testear los fps, mostrar información del estado del sistema, entre otras funcionalidades de benchMarking

La shell toma los comandos línea por línea, los compara con la lista de disponibles, y ejecuta la función correspondiente. Si el comando no existe, avisa al usuario.

Aplicaciones y juegos integrados

Para demostrar la API y las capacidades gráficas y de sonido, desarrollamos como aplicación principal el Tron, en una versión muy básica que ofrece la posibilidad de jugar de a 1 o 2 personas. Dando la posibilidad de elegir la velocidad con la que avanza cada jugador, como también seleccionando la cantidad de píxeles que hay por celda en el juego y además un mensaje para el usuario ganador. El programa corre en userland y hace uso intensivo de las syscalls del kernel: lee las teclas, dibuja los gráficos usando rectángulos de colores, actualiza el puntaje en pantalla y reproduce sonidos cuando un jugador colisionó con un obstáculo o con el otro jugador. Se implementó la detección de colisiones con obstáculos.

Durante el desarrollo del juego hubo que ajustar varias veces la lógica para que funcionara bien lo relacionado a la velocidad del juego, la latencia y la sincronización de los gráficos con el refresco de pantalla.

El Tron sirvió además como método de pruebas para todos los módulos del kernel: permitió detectar y corregir errores en los drivers, en el manejo de interrupciones y en la API de syscalls. Tener una aplicación real corriendo en userland fue clave para validar que el sistema fuera realmente funcional y estable.

Compilación y ejecución automatizada

Para facilitar el proceso de armado y prueba del sistema operativo, automatizamos los pasos de compilación y ejecución mediante scripts bash simples. Esto permite construir todo el proyecto y levantar el entorno de pruebas en QEMU evitando los comandos. Desde la raíz del proyecto se deben usar los comandos: ./build.sh y ./run.sh.