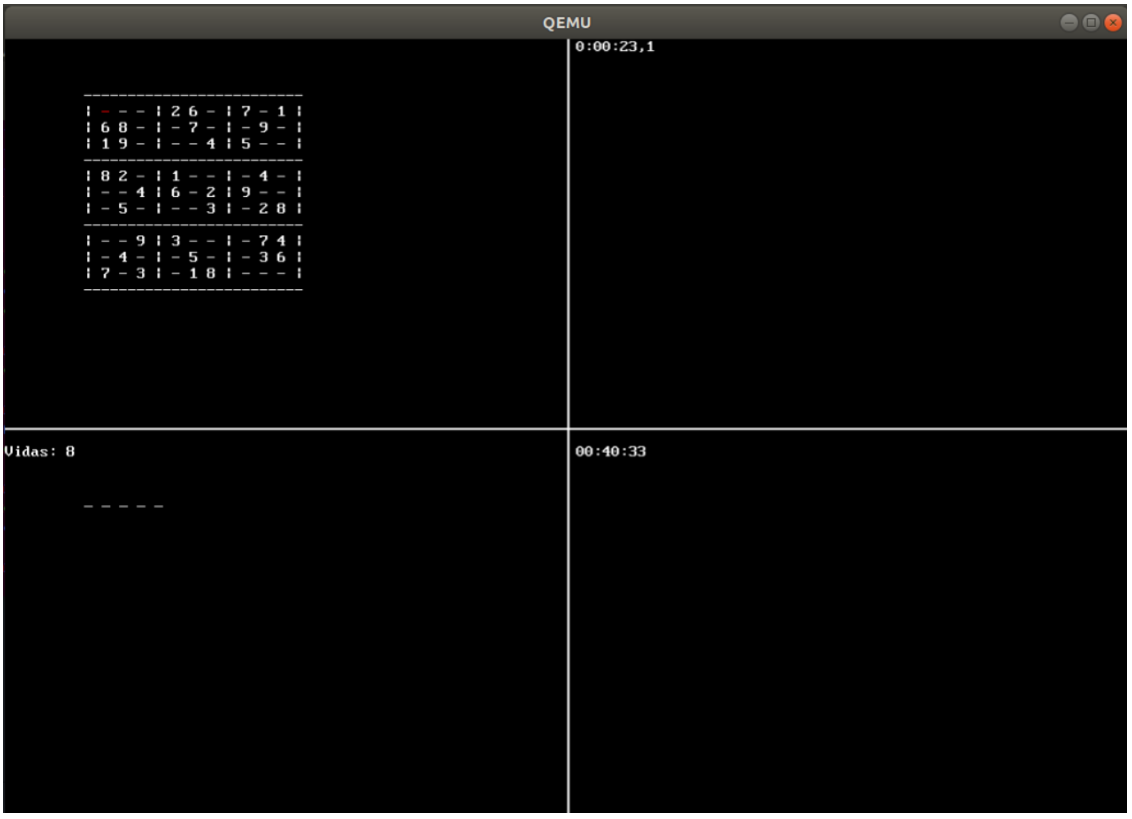
Informe - Trabajo Práctico Especial

Ferreiro, Lucas Agustín (61595)
lferreiro@itba.edu.ar

Di Toro, Camila (62576)
cditoro@itba.edu.ar

Chayer, Iván (61360)
ichayer@itba.edu.ar

Grupo 15 - Segundo cuatrimestre 2021



RESUMEN

El objetivo de este informe es explicar las distintas decisiones tomadas a la hora de realizar el trabajo práctico. El mismo consistía en realizar un Kernel booteable a partir de uno pre-estructurado provisto por la cátedra llamado Pure64.

Dicho Kernel administra recursos de hardware y provee una API para que las aplicaciones de usuarios puedan utilizarlas. Dicha API está basada en la API de Linux.

Se definieron dos espacios separados, un *Kernel Space* y un *User Space*. El *Kernel Space* interactúa directamente con el hardware mediante drivers, provee funciones al *User Space* y hace manejo de las excepciones e interrupciones. El *User Space* puede acceder ciertas funciones del *Kernel Space*, mediante la interrupción de software 80h. Esto se debe a la separación previamente mencionada y se refuerza con el hecho de que, bajo ninguna circunstancia, el usuario debe acceder directamente al hardware, sino que puede hacerlo a través del *Kernel Space*. Es por esto último que se definió en *User Space* un set de funciones para interactuar con dicha API y algunas funciones equivalentes a la biblioteca estándar de *C* (*printf*, *getChar*, *putChar*, etc).

METODOLOGÍA DE TRABAJO

Las herramientas utilizadas por los integrantes del equipo fueron las siguientes:

- Un repositorio en Github donde se trabajaron con distintas ramas.
- Visual Studio Code junto con la extensión de LiveShare que permite que más de una persona pueda trabajar sobre el mismo código en simultáneo.
- GDB para debbugear los codigo.

Con estas herramientas logramos dividirnos las tareas más superficiales y también trabajar todos juntos en los momentos más complicados del código.

DIFICULTADES

Si bien todos los conceptos fueron dados en la teoría, el salto a la práctica fue un proceso que nos costó en todo momento. Ninguno de los 3 participantes había participado de un proyecto tan grande como este donde tuvimos que manejar varios archivos distintos que no fueron programados por nosotros.

Haremos referencia a las dificultades encontradas en la discusión de cada implementación.

Kernel Space

Dentro del directorio **Kernel-Pure64/Kernel** podemos encontrar el código encargado de manejar tanto los drivers de video, como los de teclado, las excepciones y las interrupciones.

Interrupciones y Excepciones

Como se mencionó, nuestro objetivo fue imitar el comportamiento de Linux. Para ello se definieron *handlers* para manejar las distintas interrupciones y excepciones, las cuáles son cargadas en la *Interrupt Descriptor Table (IDT)*.

El Kernel maneja dos excepciones que son mencionadas en el *Manual de Usuario*, *zero division* e *invalid opcode*. Al lanzar alguna de ellas mediante un comando en la *Shell* nos pareció correcto aguardar a que el usuario pueda visualizar el mensaje de error, ver los registros al momento de la excepción y que, luego de presionar la tecla Enter, se le devuelva el control al usuario, reiniciándose la *Shell*

Las interrupciones de hardware manejadas son: *timer tick* y *keyboard*. Para el manejo del driver del *keyboard*, se creó una representación del teclado en *Us International QWERTY*. Se puede escribir tanto en minúscula como en mayúscula. Para la mayúscula se debe mantener pulsada la tecla Shift. A medida que se escriben los caracteres, estos se guardan en un arreglo que actúa como buffer circular, el cual elegimos para evitar overflows.

Las *syscalls* fueron implementadas bajo el puerto 80h de la IDT para mantener el estilo que decidió adoptar Linux. Estas reciben por registro ciertos valores para ser invocadas siguiendo la convención de registros en 64 bits para adaptarlo a nuestra librería.

Modo Gráfico

Como grupo decidimos implementar el modo gráfico. Luego de haber encontrado su dirección en el archivo *sysvar.asm*, definir una estructura adecuada para acceder a los datos de la misma y setear esta última en la dirección 0x0000000000005C00, prendimos el bit cfg_vesa para habilitar el modo gráfico.

Cuando pasamos a modo gráfico, tuvimos que ingeniárnosla para hacer un buen manejo de los pixeles. Este manejo no había que tenerlo en *naiveConsole.h* y al principio nos confundió un poco. Siguiendo el link que se encuentra en *fonts.h* el grupo pudo pasar de manejo de pixeles a manejo de caracteres para facilitar luego, el re-armado de las funciones de *naiveConsole.h*, que habían dejado de funcionar luego de habilitar esta característica. Para esto, dividimos el bitMap en bloques de 8x16.

Múltiples Ventanas

Una de las dificultades que tuvimos fue pensar como dividir la pantalla en 4 más pequeñas e independientes entre sí. En este punto, tuvimos que decidir como imprimir ciertas cosas en una pantalla y en otra no. Una vez implementada esta parte, se nos dificultó el pasaje de modo default (una única pantalla) a 4 pantallas y viceversa. Notar que para salir de pantalla dividida se debe presionar la tecla SPACE.

Decidimos dividir la pantalla con unas líneas centradas de manera horizontal y vertical. Siempre se imprime/borra mediante una estructura que contiene información del ancho, alto y posición del cursor de la respectiva ventana. De esta forma, el usuario no tiene por qué preocuparse de la posición que tiene en cada pantalla.

Al tener esta división de pantallas, nos pareció apropiado que el usuario pueda interrumpir para setear en la pantalla donde quiere escribir a través de la función *sys_setScreen(uint8_t id)*, para dividir la pantalla usando *sys_divide()* o para pasar a modo default usando *sys_uniqueWindow()*.

El proceso que el usuario desde *User Space* debería seguir para utilizar la pantalla dividida es el siguiente:

- Primero, debe utilizar el método *sys_divide()*, que inicializa las pantallas divididas.
- Luego, mediante el método *sys_setScreen(uint8_t id)*, puede elegir entre los ids 0, 1, 2 y 3 para seleccionar la pantalla que desea manipular. Luego de ejecutar esta función, todas las funciones que se utilicen relacionadas con el manejo de la pantalla, realizarán cambios sobre la pantalla que corresponda con el id seleccionado.
- Por útlimo, el método *sys_uniqueWindow()* permite regresar a una única pantalla.

El modo de múltiples pantallas puede verse ejecutando el comando *play* en la *Shell*, el cual desplegará las siguentes funcinalidades, una en cada pantalla:

- Un cronómetro
- Un reloj
- Un sudoku
- Un ahorcado

A continuación, se describren brevemente las alternativas posibles para manipular cada uno de los ítems mencionados anteriormente.

Cronómetro

El cronómetro desplegado en la pantalla superior derecha comienza a correr cuando se ejecuta el commando play con una precisión de décima de segundo. Para manipularlo, se cuenta con las siguientes alternativas:

- Presionando +, el cronómetro puede pausarse o despausarse
- Presionando 0, el cronómetro se detiene y se reinicia (vuelve a estar en 00:00:00). Basta presionar + para que vuelva a comenzar.

Reloj

El reloj no cuenta con ninguna funcionalidad extra, puede observarse constantemente la evolución de la hora actual en la pantalla inferior derecha.

Sudoku

Para manipular el sudoku, el usuario puede utilizar las flechas del teclado para moverse entre los espacios libres. Puede utilizar los números del teclado para agregarlos al sudoku. Al agregar un número, se desplegará un mensaje que indicará si el número ingresado es válido o no, o bien si el sudoku fue completado.

Mas en detalle, el funcionamiento es el siguiente: si el buffer de teclado recibe un número, este se insertará donde esté el cursor rojo (posición acutual del usuario). Ademas se puede notar que el cursor rojo no puede pasar sobre ciertas posiciones que están **fijas**.

La actual implementación del sudoku cuenta con un único sudoku inicial, el cual sabemos que contiene una solución válida. Generar un sudoku de manera pseudoaleatoria hubiera sido difícil, pues deberíamos estar chequeando constantemente si generamos un sudoku válido para mostrar en pantalla y que el usuario pueda ser capaz de resolverlo.

Ahorcado

Por ultimo, cada vez que el buffer del teclado recibe una letra, este se utilizará para jugar al ahorcado en la última ventana. Si la letra insertada pertenece a la palabra a adivinar, esta aparecerá en todos los lugares correspondientes. Caso contrario, el jugador irá perdiendo vidas.

En caso de que se quede sin vidas o que adivine la palabra, el usuario podrá volver a iniciar un nuevo ahorcado presionando la letra `p`

User Space

Dentro del directorio **Kernel-Pure64/Userland/SampleCodeModule** podemos encontrar los códigos encargados de manejar la librería con las funciones provistas para el usuario y los comandos disponibles en la *Shell*.

(1) Comandos

Dentro del archivo *shell.c* podemos encontrar las implementaciones de los comandos disponibles en la *Shell*:

- help
- time
- inforeg
- printmem
- play
- dividezero
- invalidop

Las funcionalidades de cada comando se encuentran detallas en el *Manual de Usuario*.

(2) Libreria

Dentro del archivo *userstdlib.c* podemos encontrar las implementaciones de las distintas funciones a disposición del usuario. Dichas funciones se tomaron como base de la biblioteca estándar de C.

- int **put_char** (uint8_t fd, char c)
- int **get_char** ()
- int **read_char** ()
- void **get_time** (char * buffer)
- void **get_date** (char * buffer)
- int **_strlen** (const char * str)
- int **strcmp** (char * s1, char * s2)
- char * **my_strcpy** (char * destination, char * source)
- char * **my_strncpy** (char * destination, char * source, int size)
- int **sprint** (uint8_t fd, char * str)
- void **my_printf** (const char * frmt, ...)
- char * **convert** (unsigned int num, int base)
- int **atoi** (char *str)
- int **tick** ()

- void **clearScreen** ()
- void **restartCursor** ()
- void **divideWindow** ()
- void **uniqueWindow** ()
- void **divideByZero** ()
- void **invalidOp** ()
- void **setScreen** (uint8_t id)
- void **printMem** ()