Instituto Tecnológico de Buenos Aires

Trabajo Práctico Especial

Informe

Perez de Gracia, Mateo
Quian Blanco, Francisco
Stanfield, Theo
Ves Losada, Tobías

Arquitectura de Computadoras - 72.08

Primer cuatrimestre 2023

1 Introducción

El objetivo del presente informe es explicar y comentar las distintas decisiones y caminos tomados a la hora de realizar el TPE. La consigna consistía en desarrollar un Kernel booteable, tomando como base el Pure64 provisto por la cátedra.

El mismo debe administrar recursos de hardware y proveer una API para los usuarios basada en la de Linux.

El Kernel Space se encargará de interactuar de manera directa con el hardware y la memoria, implementando drivers que luego se utilizaran para manejar las excepciones e interrupciones.

El User Space podrá hacer uso de algunas rutinas del Kernel Space mediante la interrupción de software 69h, manteniendo los límites trazados. El User Space nunca tiene acceso directo a los recursos, sino mediante dichas interrupciones (system calls).

2 Metodología Empleada

2.1 Software utilizado

- **Docker:** Para compilar el proyecto desde un entorno común para todos.
- **Qemu:** Para correr la imágen compilada en una máquina virtual.
- **GDB (GNU Debugger):** Para debuguear el código línea por línea y solucionar errores que hemos encontrado.
- **Git + GitHub:** Para manejar el versionado del proyecto y contribuir sobre una misma base de código.

2.2 Reglas de Estilo

Para mantener un estilo acorde y consistente, decidimos preestablecer algunos estilos de código y utilizamos el programa `clang-format` que permite con un archivo de configuración (`.clang-format`) setear los estilos deseados y autoformatear el código para mantener la consistencia.

Algunos de los estilos utilizados son los siguientes:

- Para variables y funciones utilizamos `snake_case`.
- Para archivos utilizamos `camelCase`.
- Para carpetas principales (como `Kernel` o `Userland`) utilizamos `PascalCase`.
- Los `typedef` a tipos de datos como `struct` o `enum` utilizan `PascalCase`, simulando nomenclatura estándar de clases.
- Espacios entre todos los operadores.
- El tipo de retorno en la definición de las funciones se escribe en una línea y en la siguiente el nombre de la función.
- Las llaves van en una nueva línea única y exclusivamente si pertenecen a la definición de una función o de estructuras complejas como `struct` o `enum`, si no se escriben en la misma línea que el operador de control.
- Todo archivo `.c` debe tener su `.h` correspondiente, pero no necesariamente todo `.h` debe tener un `.c`. Similar para un archivo `.asm`.
- El largo máximo de caracteres por línea son 120 caracteres.

Esos son algunos de los estilos utilizados para mantener consistencia a lo largo de todo el proyecto.

3 Kernel Space

Todo el código perteneciente al *kernel space* se encuentra en el directorio `Kernel/`.

3.1 Modo Video

Luego de analizar la consigna y la documentación de **Pure64**, se decidió habilitar el modo gráfico.

Se habilitó mediante el bit `cfg_vesa` en `sysvar.asm`. A partir de la dirección donde se encontraba el `VBEModeInfoBlock` (`0x0000000000005C00`) definimos en **C** una estructura para manipular adecuadamente los datos.

Rápidamente notamos que todo lo implementado en modo texto, como `naive_console`, no funcionaba correctamente en modo video. Ahora teníamos, pixel por pixel, una pantalla de `1024x768`.

Para escribir texto, se utilizó un bitmap de 8x16 como fuente y se implementó un driver reconstruyendo y ampliando las funcionalidades de `naive_console`.

3.2 Interrupciones

Definimos implementar *dispatchers* para manejar tanto interrupciones como excepciones. Estas rutinas son cargadas a la IDT (***I**nterrupt **D**escriptor **T**able*). La IDT se encuentra en la dirección 0x0h y la cargamos declarando un arreglo de estructuras al iniciar el Kernel.

`irq_dispatcher` se encarga de manejar las interrupciones de hardware, que provienen del Timer Tick y Teclado. Se encuentran en las entradas `0x20` y `0x21` de la **IDT** respectivamente.

`syscall_dispatcher` se encarga de manejar las system calls, que permiten al usuario llamar a ciertas funciones de los drivers de texto, video, sonido, tiempo, teclado, entre otros.

3.3 Excepciones

El Kernel maneja dos excepciones, *Zero Division Error* e *Invalid Opcode* que se encuentran en el índice `0x0` y `0x6` de la **IDT** respectivamente. Cuando alguna de ellas ocurre, el procesador lanza una excepción que interrumpe la ejecución, la misma es atendida por `exception_dispatcher`.

En ambos casos se muestra un mensaje de error. Se decidió que a la hora de devolver el control al usuario, se restauren los registros, el puntero al stack y se reinicie el `shell`.

3.4 Drivers

- `text`: El driver de texto hace uso de funciones del driver de video para declarar funciones que permiten la impresión de texto en pantalla y funciones para el manejo de un cursor.
- `sound`: Provee una función que permite reproducir sonidos de frecuencias y duraciones específicas.
- `time`: Permite un cierto control sobre el tiempo como saber la cantidad o segundos que han pasado desde que corrió el programa y configurar sentencias de *sleep*, para demorar una ejecución por alguna razón.
- `video`: Su función es todo lo relacionado al modo vídeo, dibujar píxeles (compuestos de 3 bytes, red, green y blue) y renderizar caracteres o imágenes/wallpapers.
- `rtc`: Este driver se encarga de obtener el tiempo real del sistema y asignarle el formato correcto para ser llamado por la system call datetime.
- `keyboard`: Este driver es el encargado de recibir los códigos de las teclas presionadas, convertirlos en sus correspondientes caracteres ASCII y guardarlos hasta que sean pedidos por el programa.

4 User Space

Todo el código relacionado al User Space se encuentra en `Userland/SampleCodeModule/`.

4.1 Shell

Una implementación modesta de un shell interactivo, similar a los disponibles en Linux. La misma permite interactuar con un set de comandos preestablecidos que demuestran la funcionalidad del sistema completo.

4.1.1 Comandos disponibles:

- `help`: Imprime un mensaje de ayuda, mostrando todos los comandos disponibles y sus respectivas descripciones breves.
- `datetime`: Imprime fecha y hora del sistema actual (en formato GMT-3).
- `printreg`: Imprime los valores de los registros del procesador, capturados en algún momento de la ejecución al presionar la combinación de teclas `Ctrl+r`.
- `pong`: El clásico juego *pong*. Permite realizar partidas de dos jugadores a 5 puntos.
- `setcolor`: Permite setear los colores del shell en tiempo real.
- `switchcolors`: Invierte el color de foreground, con el color de background.
- `clear`: Limpia la pantalla completa.
- `testioe`: **Test Invalid Opcode Exception**. Al ejecutarlo genera una excepción de tipo *Invalid Opcode* asignando un valor al registro `cr6`.
- `testzde`: **Test Zero Division Error Exception**. Al ejecutarlo genera una excepción de tipo *Zero Division Error* simplemente intentando dividir por cero.
- `exit`: Finaliza la ejecución del shell. Para demostrarlo se decidió poner dos mensajes por fuera del shell una vez que finaliza su ejecución, uno en *Userland* al retornar del shell y otro en *Kernel* al retornar de *Userland*. Se soluciona de esta manera pues al finalizar el programa, `qemu` no finaliza su ejecución, por lo que no quedaba claro si el comando `exit` cumplía lo pedido.

4.2 Pong

El clásico juego Pong, permite jugar una partida de dos jugadores a 5 puntos.

No hay mucho que decir del mismo más que tuvimos que realizar muchas optimizaciones en cuanto al dibujado de las figuras para evitar imágenes flasheantes por redibujar la pantalla completa. Por lo que se optó por dibujar únicamente las partes necesarias y "borrar" en las que las imágenes no están más (que era simplemente dibujar el mismo área con el color de fondo).

Otro inconveniente más relevante se explica más adelante en la sección de "Problemas encontrados".

4.3 Librería estándar

Se puede encontrar la API en el archivo `stdlib.c` que cuenta con las siguientes funciones:

```
uint8_t getchar(uint8_t* state);
uint32_t gets(char* buff, uint32_t size, uint32_t color);
void putchar(char c, uint32_t color);
void puts(char* str, uint32_t color);
uint64_t strlen(char* buff);
uint32_t strtok(char* buff, uint8_t token, char** args, uint32_t size);
uint32_t strcmp(char* s1, char* s2);
uint32_t uint_to_base(uint64_t value, char* buff, uint32_t base);
uint8_t is_hex_color_code(char* code);
uint32_t hex_to_uint(char* code);
```

5 Diseno y Problemas

5.1 Diseño elegido

Para elegir el diseño nos centramos en la simplicidad para el agregado de drivers y su implementación. Al dividir todo correctamente en carpetas para ser más organizado, el agregado de drivers puede ser más sencillo y esto lo comprobamos durante la realización del trabajo. Para el agregado de un driver se debe crear el archivo `.c` correspondiente en la carpeta drivers y luego su `.h` correspondiente en la carpeta `include/`. Este diseño también se separa en una carpeta `idt` (***I**nterrupt **D**escriptor **T**able*). Dentro de esta se encuentra todo lo relacionado a las llamadas al sistema que realiza nuestro trabajo. Por ende, para agregar una llamada al sistema se debe actualizar el archivo `syscalls.c` e introducir la nueva llamada al sistema e indicar que se debe ejecutar una vez obtenida esa llamada.

Este diseño del **Kernel** también facilita la implementación de las funciones assembler en **Userland**, ya que simplemente se realiza la interrupción y el idt se ocupa de llamar a las funciones desde el Kernel.

El lado negativo de este diseño es también la implementación de nuevos drivers. Aunque sea verdad que se puede volver más sencillo y organizado de esta manera, también se vuelve muy tedioso el agregado de un driver. Esto se debe a los diversos archivos a crear y los distintos archivos a los cual hay que aplicar el nombre del archivo para que pueda correr adecuadamente.

5.2 Problemas encontrados

- El primer problema encontrado fue el paso al modo video. Esto resultó en un problema ya que los ejercicios propuestos en el pre TP estaban todos basados en el modo texto. Por esta razón, tuvimos que investigar cómo funcionaba este nuevo modo y reiniciar el aprendizaje desde cero para aprender y poder entender este nuevo modo.
- Una vez arrancado el proyecto tuvimos que organizar todos los archivos provistos por la cátedra para adaptarlos a nuestro diseño. Esto concluye en un problema porque al modificar la carpeta donde se encontraban ciertos archivos estos ya no eran encontrados por el compilador. Por lo tanto luego de debugear, pudimos reconstruir el buildsystem para que pueda compilar todos los archivos, tanto actuales como futuros.
- Un error que tuvimos desarrollando el proyecto fue la errónea inclusión de un wallpaper en una primera instancia. Cuando quisimos incluir un wallpaper por primera vez, al ejecutar se rompía el código y no funcionaba. Concluimos que el tamaño de la imagen era lo que estaba causando el error. Al almacenar la imagen, el tamaño de esta misma era muy grande por lo cual terminaba pisando memoria del Userland. Para arreglar esto tuvimos que encontrar el tamaño máximo de imagen que podíamos almacenar, y luego encontrar una imagen que sea igual o menor al tamaño encontrado para que luego pueda ser utilizada correctamente.
- Otro contratiempo que tuvimos fue que no éramos capaces de verificar el orden en el que los registros se cargan al stack al momento de alguna excepción. Lo pudimos arreglar debugeando con `gdb` y corroborar un orden lógico que habíamos pensado posible.
- A la hora de implementar el movimiento de los jugadores en **Pong**, resultaba que una implementación "normal" donde se verifique que tecla se presionó durante cada tick generaría el problema de movimiento asincrónicos y trabados. Para solucionar esto declaramos un array de variables de estado que en el game loop verifica la tecla presionada y setea en el array el estado de la tecla presionada y cuando salta el tick ya se conoce el estado de movimiento de cada jugador para actualizar sus posiciones y dibujarlos. Sin embargo, para poder cambiar estos estados, no solo necesitábamos conocer la tecla presionada, sino también la tecla soltada para saber cuando un jugador se mueve o deja de moverse. Para eso hubo que reescribir el driver de teclado de manera tal que al pedir un caracter, devuelva también su estado.
- Otro problema que nos encontramos cerca de la finalización del proyecto, fue la implementación del sonido para el juego **pong**. Este problema ocurre a ciertos compañeros que no podían escuchar el sonido debido a lo que creíamos que era un problema con Ubuntu. Concluimos que Ubuntu y el pulseaudio no se comunican bien lo cual llevaba a un error. Aunque no hayamos podido encontrar la causa del problema, lo pudimos solucionar ajustando los flags del `run.sh`.