Trabajo Práctico
Arquitectura de Computadoras

Integrantes:
Barruffaldi, Carla (55421)
Bianchi, Luciano (56398)
Cerdá, Tomás (56281)

Diseño e implementación nanOS

Como se pidió en el enunciado, el S.O fue implementado separando claramente el espacio del kernel y el del usuario, con distintas responsabilidades para cada uno y System Calls para permitir el uso de partes del sistema que solo pueden ser accedidas con el S.O como intermediario, como la pantalla, el teclado, o el RTC, entre otros. Las system calls se realizan mediante interrupciones de software con el código 80h.

En primer lugar, se decidió realizar el sistema completamente en modo video (VESA SuperVGA), debido a la mayor versatilidad de este modo a la hora de dibujar los fractales (o algún otro elemento gráfico que se quiera implementar en el futuro), ya que no fue necesario moverse de modo texto a modo video y viceversa. El cambio se realizó en los parámetros del archivo isa.asm para que el Pure64 bootee en modo video con resolución 1024x768x24bits. Es el Pure el que se encarga de configurar el VESA e indicarle a la BIOS el modo de video deseado.

Para escribir los píxeles en el modo video se usó el método de Linear Frame Buffering, que permite escribir en memoria a partir de una determinada dirección (obtenida desde la información de VESA), 3 bytes (en 24 bits) por cada pixel, que luego se va a imprimir en pantalla.

Interrupciones

Para las interrupciones, se decidió implementar una IDT propia, que es una adaptación a 64 bits de la IDT de 32 bits estudiada en la práctica. No se usó la tabla ya proporcionada por Pure64 simplemente porque la IDT de la práctica resultaba más familiar y, como su funcionamiento ya estaba estudiado, los errores desconocidos producto de usar nuestra IDT iban a ser menos probables.

Los system calls se implementan a partir de la interrupción int 80h que lee los contenidos de los registros rax, rbx, rcx y rdx. En rax se encuentra el identificador de la system call a ejecutar. Luego se leen los valores de rbx, rcx y rdx que son interpretados como parámetros de la system call. Finalmente, las system call se encuentran dentro de un vector de funciones que devuelven un entero no signado de 64 bit y reciben 3. Para acceder a la

system call que se desea ejecutar basta ejecutar la función de índice rax dentro del vector. Dado que no todas las system calls reciben exactamente 3 enteros, se decidió realizar wrappers que sí lo hagan para poder incorporarlas al vector. Esto nos da la ventaja de que las system calls son claras en lo que hacen, no tienen parámetros “basura” extras pero lo malo es que para agregar una nueva system call puede que haga falta escribir su correspondiente wrapper para poder incorporarla al vector.

Driver de Video

Dentro del kernel se implementaron ciertas funciones en un driver de video, que permiten la impresión de caracteres de texto, o de píxeles individuales, con llamadas de sistema que permiten al user space hacer uso de estas funcionalidades.
Como decisión de diseño se optó por delegar a este driver de video la responsabilidad de saber como y donde imprimir los caracteres de texto, con un cursor que guarda la posición donde se está escribiendo. Para imprimir los caracteres se tiene un mapa de cada uno a un arreglo de bits de 16x8 que especifica (con 1) si en ese píxel corresponde escribir un pixel para la letra, o (con 0) si es parte del fondo de pantalla.
Otro aspecto a destacar de nuestra implementación del modo video, es que tanto user como kernel pueden funcionar con cualquier resolución de 24 bits. Esto se logró haciendo que el kernel consulte en las direcciones de memoria correspondientes, la información del modo VESA, en lugar de asumir una resolución predefinida. Además, se implementaron system calls que le permiten al usuario consultar dicha resolución, para poder, por ejemplo, imprimir los fractales en cualquier resolución en la que bootee el Pure64. Para cambiar la resolución con la que se bootea, basta con cambiar un parámetro dentro de isa.asm.

Driver de Teclado

En cuanto al manejo de la entrada, se usa un buffer circular en el teclado para almacenar los códigos de las teclas que se tocan, y dicho buffer se "vacía" cuando se llama al system call Read. Devuelve el ASCII correspondiente al código de la tecla.

Entrada estándar

En Userland se implementa getchar(), que a su vez imprime en pantalla la letra que se lee del teclado y guarda lo que va leyendo en otro buffer. Cuando el usuario presiona 'enter' (\n), es cuando getchar() termina de llenar este otro buffer, y empieza a devolver, letra por letra, los caracteres ingresados.
Se eligió este diseño para que desde el lado del usuario se tenga mayor flexibilidad a la hora de pedir los caracteres al kernel. El usuario elige imprimirlos a medida que se escriben; tranquilamente se podría escribir una función que lea de entrada estándar pero no imprima a pantalla. Esto no sería posible si el kernel imprimiese a medida que se llama a la system call read, sin realizar otra system call análoga a read pero que no imprima.

Shell

La Shell aprovecha las funciones proporcionadas en nuestra versión de stdio.c, y posee un arreglo de strings a punteros de función, que nos permiten parsear el texto ingresado y ejecutar la función correspondiente. La shell proporciona funciones que permiten cambiar el color de texto y del fondo, lo que se hace mediante system calls que le indican al video driver el color RGB con el que se quieren imprimir los caracteres.

Comandos
Los comandos se guardan en un vector de commands, siendo un command una estructura con un puntero a char correspondiente al nombre del comando y un puntero a función que recibe un puntero a char como argumento y devuelve si los argumentos recibidos fueron válidos o no. Para agregar un nuevo comando solo basta escribirlo en commands.c y agregarlo al vector de comandos. Finalmente hay que actualizar la función help si se desea descripción del nuevo comando.

Módulo de Datos
Para crear el binario correspondiente al módulo de datos se usa un programa write cuyo código fuente corresponde al archivo write.c dentro de la carpeta SampleDataModule en el cual se escribe en el binario “0001-sampleDataModule.bin” los parámetros de los fractales. Simplemente hay que ejecutar el make dentro del SampleDataModule para que se ejecute el programa write.
Basta con modificar los fractales dentro write.c, re-compilar y crear el nuevo programa write y correrlo para que se escriba el binario con los nuevos fractales. Finalmente, estos parámetros son leídos apuntando a la dirección 0x500000. Esta dirección es obtenida por Userland a partir de un system call.

Otras aclaraciones
Las únicas partes de nuestro código que se tomaron de fuentes externas (además de la base proporcionada por la cátedra) fueron las tablas de datos para el mapeo del teclado, para dibujar el mapa de bits de cada carácter, y las funciones para dibujar los fractales. Están comentadas dentro del código la fuente de cada una.
La función 'clear', que limpia el texto de la pantalla en la shell, se implementó simplemente realizando '\n' por el doble de la cantidad de filas de texto que tenga el videoDriver. Se podría haber implementado simplemente sobreescribiendo toda la pantalla con el color de fondo de la shell, pero se decidió dejarlo de esta manera porque produce una especie de 'animación' al mover el contenido de la pantalla hacia arriba, y desde el punto de vista técnico es virtualmente lo mismo.

Consideraciones futuras
Como posibles mejoras para el S.O, se podría expandir el número de funciones provenientes de la librería estándar. En las acotadas funciones disponibles actualmente no fue muy necesario disponer de más de las funciones ya ofrecidas, pero puede serlo si se quieren implementar funciones más complejas.
Se podría también cargar distintos tipos de fuentes para los caracteres y diferentes mapas de scan codes para distintos tipos de teclados y que estos puedan ser configurados a gusto por el usuario.
Asimismo, puede ser útil una system call que devuelva no el ASCII del carácter correspondiente a la tecla presionada sino su scan code, para poder así manejar eventos que dependen de la presión de teclas no imprimibles. Implementar esto implica realizar pequeños cambios en el driver de video, como guardar todos los scan codes en su buffer, no solo los imprimibles y devolver carácter o scan code según la system call que se ejecute: la que devuelve caracteres o scan codes.