Trabajo Práctico Especial

Arquitectura de Computadoras (72.08)

Profesores:

Santiago Raul Valles

Celeste Gabriela Guagliano

Federico Gabriel Ramos

Giuliano Scaglioni

Grupo 01

Buela Mateo - 64680

Nahuel Ignacio Prado - 64276

Fecha

5 de noviembre de 2024

Resumen:

En el presente informe se exponen el diseño, decisiones y aspectos técnicos del TPE del grupo 01 integrado por Buela Mateo y Nahuel Prado, de la materia Arquitectura de Computadoras (72.08) del ITBA. En el mismo se discutirán y explicarán los beneficios y desventajas de las decisiones tomadas, con sus respectivas justificaciones. Además de toda la información relevante sobre el desarrollo del trabajo.

Índice:

Introducción:

El propósito principal del trabajo práctico es implementar un kernel para comprender los procesos de gestión de los recursos de hardware de una computadora.

Este kernel es booteado por Pure64, suministrado por la cátedra. En él se pueden distinguir dos espacios bien diferenciados y separados: el kernel space y el user space. El kernel space abarca todo lo relacionado al funcionamiento interno del sistema. Es el responsable de gestionar los recursos y comunicarse con el hardware mediante drivers, mientras que proporciona las funciones necesarias al user space, que facilita la interacción con el usuario. En este último se encuentra el programa principal de la shell, que ofrece todas las funcionalidades al usuario.

Desarrollo del Kernel:

El kernel space se conforma principalmente de: los drivers, las interrupciones, las excepciones y syscalls.

Drivers:

Los drivers consisten en los módulos encargados de comunicarse directamente con el hardware. Que mediante interrupciones, syscalls y en ocasiones especiales excepciones puedan desplegar información al usuario.

Video Driver:

Encargado de dibujar en la pantalla (en modo gráfico) los pixeles necesarios según lo solicitado. La función más importante es putPixel() que dibuja un pixel de un color RGB hexa específico en una ubicación. Con esta base se desarrollaron las siguientes funcionalidades.

Está drawchar(), que dado un caracter, una posición en pixeles, un color de letra y fondo además de un factor de multiplicación para el zoom dibujan el caracter en pantalla. Justamente sobre el zoom, esta función se apoya en putMultPixel() que dibuja una grilla de $n \times n$ píxeles. Esto sin mencionar el arreglo font_bitmap[], donde se detalla píxel a píxel cómo se conforma cada letra/símbolo.

Luego se encuentra drawRectangle() que dibuja un rectángulo de algún color. He de notar que para facilitar el uso de esta y varias funciones siguientes se usó el struct Point, que informa una posición específica en la pantalla.

Para cerrar está drawSpray() que como menciona su nombre, ilustra un spray(dibujo) dado por una matriz que indica de qué color se pondrá cada píxel. El equipo decidió que en pos de poder dibujar de mejor manera, el color negro (0x000000) se toma como transparente, es decir no se dibuja.

Keyboard Driver:

Keyboard Driver administra lo relacionado al teclado. Al presionar una tecla, el procesador invoca la interrupción 01, que se detallará más adelante, esta llama al keyboar_handler() que obtiene la tecla, la procesa para saber si es alguna tecla especial (enter, alt, ctrl, capslock o shift.

En base a esta información la función guarda el carácter ascii que corresponde en un buffer de 64 lugares. Esto se ideó para que el usuario pueda escribir las teclas que desee, y luego cada programa lo vaya leyendo (utilizando la función nextKey()) a su ritmo, y utilice estas teclas a su gusto.

También hay que destacar que en caso de no tener teclas en el buffer, la función devuelve un -2. El por qué de esto se explicará más adelante.

PC Speaker Driver:

Y finalmente este driver se ocupa de proporcionar un sistema de sonidos para el kernel. Donde enviando frecuencias específicas al sistema de entrada y salida del hardware se reproduce un sonido.

Por cómo funciona el driver utilizado, se requieren dos llamadas, tanto para prender como para apagar la reproducción. Otra distinción es que al sólo permitir ondas cuadradas, no hay manejo de volumen.

Syscalls:

Mediante las syscalls, el user space es capaz de comunicarse y mandar comandos al kernel space. Su funcionamiento consiste en llamar a una función sys_call(...) en libc.c donde se pasan los parámetros necesarios para la llamada, su máxima cantidad de argumentos es de

seis (6), rax, rdi, rsi, rdx, r10, r8, en ese orden. Entre estos está el principal (rax) que indica a qué syscall se llamará.

Posteriormente en Assembler, se reordenan los registros para que coincidan con el estándar de linux y se realiza la interrupción 80h. Para luego llegar a sysCallDispatcher, que según los argumentos enviados decide qué syscall ejecutar. Las últimas mencionadas se detallan a continuación.

Read:

Registros: rax - 1; rdi - file descriptor; rsi - buffer; rdx - cantidad; r10- basura; r8 - basura.

Esta syscall llama cantidad de veces a la función nextKey() proporcionada por el Keyboard Driver que devuelve la primera tecla de su buffer, para luego guardarlas en el buffer enviado por user. En caso de cumplir con la cantidad, o de no obtener más teclas (recibir un -2 del teclado) la llamada termina, devolviendo la cantidad de caracteres leídos.

Como se explica, la syscall no se queda “esperando” nuevas teclas, esto es de vital importancia pues existen programas que necesitan seguir corriendo mientras no se presionen teclas, pero que al mismo tiempo tienen que estar verificando constantemente si se ingresaron nuevos caracteres en el buffer.

También se aclara que en el campo de file descriptor se espera únicamente un cero (0), pues por el momento no se implementó un sistema de manejo de archivos, por lo que sólo se puede leer de entrada estándar. En caso contrario no ocurre nada.

Write:

Registros: rax - 2; rdi - file descriptor; rsi - buffer; rdx - cantidad.

La idea del write es la siguiente, escribir en pantalla la cantidad deseada de letras del buffer. Esta utiliza la función drawchar() para imprimir los símbolos. Y se basa en dos variables estáticas que determinan el cursor, es decir, dónde se imprimirá.

La llamada contempla la posibilidad de “pasarse” del ancho de la pantalla o de encontrar se un enter (caracter ascii ‘\n’), en cuyo caso se continuará escribiendo una línea más abajo a la izquierda. Al terminar, se actualiza este mismo cursor para que a la próxima llamada se dibujen los símbolos a continuación.

Get Registers:

Registros: rax - 2.

Una de las opciones más importantes ofrecidas por el kernel space es el de poder guardar el estado de los registros en cualquier momento, simplemente presionando la tecla ‘enter’. Y usando esta syscall, estos mismos se devolverán en un arreglo.

El encargado de guardar los registros es la interrupción de teclado, que detallaremos más adelante.

Get Time:

Registros: rax - 5; .

Devuelve en forma de puntero a struct la fecha y hora actual en UTC. Usando instrucciones de assembler, es capaz de obtener el día, mes, año, horas, minutos y segundos. Según se leyó en la documentación.

Set Cursor:

Registros: rax - 4; rdi - pos_x; rsi - pos_y.

Se coloca el cursor, donde imprimirán las llamadas de write, draw Spray. Funciona simplemente teniendo dos variables estáticas que el usuario va a poder ir actualizando. Este diseño se eligió para simplificar el uso de estas llamadas, sin que el usuario tenga que pasar tantos parámetros.

Set Font Color:

Registros: rax - 7; rdi - color.

Setea el color de letra con el cual se escribirá en pantalla. En formato hexadecimal.

Set Zoom:

Registros: rax - 8; rdi - zoom.

Indica el zoom para imprimir en pantalla. De vuelta, esta syscall se implementó con la intención de que el usuario pueda especificar al kernel el formato del write.

Draw Rectangle:

Registros: rax - 9; rdi - point_1; rsi - point_2; rdx - color.

Usando el driver de video dibuja un rectángulo del punto uno al punto dos con el color dicho. Para simplificar el pasaje de parámetros, se ideó una estructura Point.

Set Background Color:

Registros: rax - 10; rdi - color.

Se selecciona el color de fondo con el cual se escribirá en pantalla, en formato hexadecimal. Con este, terminan todas las syscalls para customizar el write.

Draw Spray:

Registros: rax - 11; rdi - spray; rsi - size_x; rdx - size_y.

En base al driver de video dibuja, según una matriz de pixeles donde cada posición indica un color hexadecimal, con sus respectivas dimensiones un dibujo a elección del usuario.

Play Sound:

Registros: rax - 12; rdi - frequency.

Llama al driver de sonido para iniciar la reproducción de la frecuencia específica que el usuario indicó.

Stop Sound:

Registros: rax - 13.

Detiene la reproducción iniciada por Play Sound.

Get Ticks:

Registros: rax - 14.

Devuelve los ticks contabilizados por time handler llamado por el timer tick, cada tick del sistema. Se utiliza más que nada para el usuario poder tener una noción de cuánto tiempo pasó entre dos procesos.

Sleep:

Registros: rax - 35; rdi - ticks.

Suspende el sistema, utilizando la función hlt, contando la diferencia de ticks hasta cumplir lo requerido.

Interrupciones:

El funcionamiento general de las estas son, como indica el nombre, interrumpir al procesador para ejecutar una rutina específica guardada en la IDT. Esta misma tabla, según la documentación, se ubica en la posición 0x0000000000000000 de memoria con 32 primeros lugares para excepciones y posteriormente las interrupciones.

Al iniciar el kernel, se las setean con idt_loader() donde se colocan los diferentes punteros a función de las diferentes interrupciones requeridas por el sistema. Entre ellas están, el timer tick, la interrupción de teclado, y la int 80 que corresponde a las syscalls. Las dos primeras se denominan de hardware mientras que la última de software.

Luego en assembler, mediante una macro y el número de interrupción, se pushea todo el estado al stack, y llama a irqDispatcher() que se encarga de designar el qué hacer. Para luego devolver el estado del sistema. Esto es sólo si la interrupción es de hardware. En caso de ser una syscall se llama a syscallDispatcher(), que administra las syscalls.

El timer tick llama a time handler que es el encargado de contabilizar los ticks del sistema. En la interrupción del teclado también se encuentra una variable, en la zona de datos, que al ser activada por getKey guarda el estado de los registros en un buffer.

Excepciones:

Estas son interrupciones de software saltan en algún evento inesperado del sistema. Como se dijo, sus rutinas de atención se encuentran al inicio de la IDT. Este kernel dispone de soporte para dos excepciones, dividir por cero y operación inválida.

Ambas funcionan de manera similar y son cargadas junto a las interrupciones, al ser ejecutadas utilizan una macro que guarda los registros, informa acerca de la excepción, espera al usuario a presionar ‘enter’ y finalmente vuelve al kernel saltando al kernel space, no sin antes establecer bien el inicio del stack.

Librería Estándar:

Ya ubicado en el user space, se dispone de una librería con todas las funciones que podrían resultarles útiles al usuario. La mayor parte de estas se apoyan en las syscalls para cumplir su propósito.

Hay tres principales grupos en las que las podemos dividir.

Strings:

Las diferentes funciones que soportan el uso de strings son: itoa() para pasar enteros a una cadena, strlen(), strCpy(), strNCpy(), strCmp(), strCaseCmp(), isalpha(), toupper(), tolower(), timeToStr() que obtiene el tiempo y lo pasa a un string. Hay varias que fueron fuertemente inspiradas en funciones de tanto string.h como ctype.h.

IN/OUT:

Nuevamente también se ofrecen las funciones básicas de stdio.h, como print(), nprint(), putChar(), scan(), getChar(), que no ofrecen un sistema de formato pues el equipo no vio necesario implementarlo.

Luego se encuentran otros tipos de funciones que igualmente se comunican con la pantalla por ejemplo showRegisters() que muestra los registros guardados, setBackGroundColor(), drawRectangle(), drawSpray(), setFontColor(), setCursor(), setCharCursor() hace lo mismo que setCursor pero en caracteres, no pixeles.

Programas:

Existen ciertas funcionalidades de la shell que requieren de un “programa” aparte, es decir que la información se despliegue en pantalla y se suspenda el funcionamiento de la shell.

En la librería disponemos de varios programas: time, rectangle, help y registers. Todos esperan a que se presione la tecla ‘q’ para finalizar y volver a la shell.

Funcionamiento general:

Y finalmente se tienen de funciones varias que no entrarían en ninguna categoría específica como sleep(), getTicks(), getTime() y cleanFullScreen().

Sonidos:

Por fuera de libc.c se diseñaron otras funcionalidades de gran utilidad. Para el manejo de sonidos se encuentran playSound(), stopSound(), playSoundForTicks(), setSound() que agrega a un buffer un sonido a reproducir, actualizeSound() que reproduce el siguiente sonido a la lista o detiene el actual sonido sonando.

Random:

Hay otras dos funciones para generar números random, especialmente utilizadas para el snake.

He de aclarar que la utilización de cada una de estas funciones están desarrolladas en los respectivos .h de cada archivo.

Implementación del intérprete de comandos (Shell):

La parte principal del proyecto lo conforma el intérprete de comandos, es decir la shell. En la parte baja de la pantalla el usuario tiene una línea de comandos de dos líneas máximas. Al presionar ‘enter’ se ejecuta el comando dado, imprimiendose arriba junto a su respuesta. Si no está soportado por la shell imprime ‘Command not found’. Esta soporta los siguientes comandos.

Un detalle agregado al final fue una pantalla de inicio colorida y decorada con manzanas, con un breve tono para dar la bienvenida al usuario. Desplegando también a los creadores del kernel.

Color:

Cambia el color de la letra en forma cíclica. Se disponen un total de 21 colores en total. Hay ciertas funcionalidades que no se ven afectadas por esto, como por ejemplo los registros.

Date:

Inicia el programa date, que muestra la fecha y hora en tiempo real UTC en medio de la pantalla, se requiere presionar la ‘q’ para salir y volver a la shell.

Rec:

Dibuja un rectángulo en la pantalla del color que está seleccionado en ese momento. Se espera a que se presione la ‘q’ para salir.

Zoom in:

Aumenta el tamaño de la letra, esta instrucción se puede realizar hasta un máximo de tres veces cuando se le informa que ya no se puede agrandar más. El funcionamiento general es imprimir, por cada pixel de la matriz de la font, un cuadrado de n x n, así la letra termina siendo más grande. En base a todo esto, se actualiza, mediante una variable local zoom, la altura y ancho de un caracter y donde se va imprimiendo la línea de comandos.

Existen funcionalidades que utilizan su propio tamaño de letra, sin importar el zoom. Esto se hizo pues podría ocurrir que ciertos textos ya no entrasen con un zoom muy grande.

Zoom out:

Retrae la acción realizada por el zoom in, entre estos dos el valor del zoom puede variar entre 1 y 4.

Snake:

Se llama al juego snake, que se detalla más adelante. Al volver imprime ‘Snake exited’.

Div0:

Comando para testear la excepción de división por cero, únicamente se invoca una función de assembler que intenta la división.

InvOp:

Comando para testear la excepción de operando inválido, se llama a una función de assembler que aplica una operación no soportada por el lenguaje.

Registers:

Imprime en rojo el estado de los registros al apretar la tecla ‘enter’. Espera a la ‘q’ para salir.

Exit:

Finaliza la shell, al igual que al inicio, suena un breve tono y muestra la pantalla de salida. Para luego simplemente dejar que el programa vuelva al kernel para que termine en un cli hlt, donde ya no hará nada más hasta reiniciar por completo el sistema.

Help:

Invoca al programa de help, que imprime todos los comandos disponibles.

Snake:

El “programa” Snake es un videojuego que ofrece la posibilidad de jugar de a un jugador o dos jugadores, y también se puede activar un modo de juego especial, que se detalla más adelante. Este programa es aquel que demuestra todo el potencial de nuestro kernel, ya que hace uso de todos los drivers y usa la mayoría de las syscalls. El videojuego consta de una o dos serpientes, dependiendo de la cantidad de jugadores, las cuales deben comer unas manzanas para hacer el puntaje más grande o encerrar a la serpiente contraria para que choque. Incluimos una manzana especial (dorada) la cual aumenta el tamaño en 5 y tiene una aparición baja, mientras que la manzana normal (la roja) solo suma uno. Diseñamos un pequeño menú de opciones que permite cambiar entre “1P” ,”2P”, “NORMAL”, “EXIT”. Las opciones 1P y 2P hacen referencia a “One Player” y “Two Players” respectivamente. El modo especial ya mencionado lo único que hace es hacer que todas las manzanas sean doradas para esto hay que apretar el enter sobre la opción “NORMAL”, la cual cambiará a “GAPPLE” (golden apple). La última opción es “EXIT” que permite volver a la shell.

¿Cómo se dibujan las snakes? Para poder dibujar las snakes de la manera más eficiente se nos ocurrió solo borrar la cola y dibujar la cabeza. Para esto debemos saber cual es la cola y cual es la cabeza, la solución fue usar un vector circular que va guardando las posiciones actuales de toda la serpiente y luego con dos índices, un índice cabeza y otro cola tenemos toda la información que precisamos. Para no usar dos variables por cada lugar del vector de posiciones, hicimos que cada bloque del tablero tiene un número asignado y luego si fuera necesario mediante matemática, una función recibe ese número y lo transforma en un struct que contiene las coordenadas en x e y. Esta última función es utilizada principalmente para controlar las colisiones, ya que además de saber posiciones en el vector, usamos una matriz a modo de mapa para saber en todo momento dónde están la/s serpiente/s y si chocaron consigo mismas o con la otra serpiente. Además se controla que no se exceda del mapa en cuyo caso se considera un choque.

Resultados y conclusiones:

En conclusión, el equipo está muy satisfecho con el desarrollo del proyecto y los resultados obtenidos. Hubo una adecuada distribución del trabajo y una organización razonable del tiempo disponible, lo cual facilitó el avance continuo en cada etapa. Es cierto que algunas funcionalidades podrían haberse mejorado o implementado de manera distinta, pero debido a la limitación de tiempo y las prioridades principales del proyecto, no fue posible abordarlas en esta ocasión.

En definitiva, se cumplió el objetivo principal de aprender a implementar un kernel con sus respectivos drivers, syscalls, interrupciones, excepciones y shell, y a gestionar los recursos de hardware de manera efectiva separando correctamente el kernel y user space.

Bibliografía:

Apuntes del Campus-ITBA de IDT, modo video, Pure64 - Proporcionados por la cátedra.

Apunte página web OsDev PC Speaker - https://wiki.osdev.org/PC_Speaker

Apunte página web OsDev PC Keyboard - https://wiki.osdev.org/PS/2_Keyboard

Apunte página web OsDev Random Number Generator - https://wiki.osdev.org/Random_Number_Generator

Apunte página web OsDev Interrupt Descriptor Table - https://wiki.osdev.org/Interrupt_Descriptor_Table

Apunte página web OsDev VGA Fonts - https://wiki.osdev.org/VGA_Fonts

Apunte página web Mixbutton Pasaje de nota musical a frecuencia - https://mixbutton.com/mixing-articles/music-note-to-frequency-chart/

Anexo: explicación de error en guardado de registros

En pos de no imprimir basura al pedir los registros sin antes haber presionado la tecla ‘esc’, el equipo ideó una manera de verificar el estado del arreglo de registros antes de devolver su puntero. Su funcionamiento es muy simple, este verifica que una variable declarada de forma estática esté prendida, en caso contrario devuelve un puntero a null (ver Figura 1).

```
231     getRegisters:
232         mov rax, 0        ; ret null
233         cmp byte [registers_saved], 1
234         jne .not_saved
235         mov rax, regs_backup
236     .not_saved:
237         ret
```

Figura 1: fragmento de ce código de interrupts.asm donde se muestra la función getRegisters.

El problema surgió en cómo se prendía esta flag, es decir, en la variable se guardaba un uno sólamente al presionar el ‘esc’. Este fue un error puramente de implementación del código pensado, donde se pasó el caso de lanzar una excepción antes de guardar los registros con el ‘esc’.

El error fue fácilmente solucionado moviendo una única línea de código, donde se prendía la flag, a la macro donde se guardaban los registros. Así siempre que los registros se guarden, ya fuese por excepción o por interrupción de teclado, se pone la variable en uno, finalmente quedando lo mostrado por la Figura 2.

```
122         mov qword rdi, [rsp]      ; rip
123         mov qword [rax], rdi
124         add rax, 8
125         mov qword rdi, [rsp+8]    ; cs
126         mov qword [rax], rdi
127         add rax, 8
128         mov qword rdi, [rsp+16]   ; rflags
129         mov qword [rax], rdi
130         mov byte [registers_saved], 1   ; seteo flag de que se guardaron registros
131     %endmacro
```

Figura 2: final de la macro catchRegisters de interrupts.asm, donde se guarda un uno en la variable registers_saved.