Arquitectura de las computadoras

Trabajo Práctico Especial

Integrantes del grupo:

- 49.254 - Ail, Brian
- 58.564 - Rojas, Cristobal
- 58.637 - Petrikovich, Florencia
- 58.717 - Tawara, Enrique

Observaciones:……
……
……

Fecha de aprobación:

Firma del docente:

Índice

Driver de Video

Para el driver de video, se definieron tres funciones que serán utilizadas por los syscalls:

- putPixel (dibuja un pixel)
- clearAll (limpia la pantalla)
- printChar (utilizada por el sys_write para imprimir)

Basado en código ejemplo de OSDev (https://wiki.osdev.org/Drawing_In_Protected_Mode), bajamos el bloque con la información del framebuffer de memoria, y el resto de los datos proporcionados por el VBEInfoBlock. Originalmente (y todavía están en el código, pero no fueron utilizados) programamos las mejoras de optimización recomendadas en OSDev, que permiten dibujar figuras o directamente strings, sin tener que recalcular las posiciones, pero no las terminamos usando (salvo para una función del kernel) ya que hubiese habido que implementar una syscall para cada una de estas optimizaciones. Si bien esto hace que dibujar figuras sea menos performante, decidimos imitar la decisión tomada por Linux, que solamente ofrece dibujar un pixel.

Decidimos entonces implementar un “cursor” al driver de video, que es el que dice donde se va a escribir el próximo carácter, que maneja lo que pasa cuando llega al borde derecho o inferior de la pantalla, y que es reseteado cuando se hace un clearAll. En este caso, al llegar al borde derecho se baja a la siguiente línea y se resetea la posición X del cursor, y si este salto hizo que se llegue al borde inferior, se llama a la funciń shiftUp. Que copia todo el framebuffer desde la segunda línea al principio del framebuffer, moviendo todo una línea hacia arriba, y luego pone el cursor al principio de donde estaría la última línea. No implementamos ningún tipo de “scroll” que vaya guardando las líneas pisadas y que permita ir hacia arriba o hacia abajo.

La mayoría de la funcionalidad extra (para dibujar rectángulos o círculos) fue implementada directamente en Userland, llamando cada vez a la sysCall de putPixel.

Teclado

Para el driver de teclado definimos un buffer de tamaño 200. El número fue decidido de forma arbitraria, considerando la posibilidad de que se puedan cargar muchas teclas al buffer si tener que ser obtenidas de forma inmediata con una syscall.

Decidimos usar dos flags para Shift, uno para el izquierdo y otro para el derecho, pues ambos tienen distinto Scan Code y temíamos que si apretamos el izquierdo y el derecho y soltabamos uno de los dos, el flag se volvería False por un milisegundo y podría cargar al buffer la versión normal de la tecla apretada. Sin embargo, como ctrl tiene el mismo Scan Code tanto para el derecho como el izquierdo no hicimos lo mismo. Decidimos ignorar la tecla Alt en esta implementación pues no servía ninguna funcionalidad para el propósito de este trabajo.

Driver de Sonido

Usando como referencia las acciones posibles del PIT disponible desde https://wiki.osdev.org/Programmable_Interval_Timer se armó un programa assembler que accede a los registros correspondientes para manejar el parlante de la pc.

Para el driver de sonido, se separaron las funciones playSound y stopSound, las cuales inician y detienen la emisión del sonido. Separamos play de stop así se tenía un control más flexible y de la mano del timer tick en conjunto a los demás comportamientos. La longitud del sonido podrá variar y ser controlado por el mismo programador con estas características. El lado negativo de este método es que puede causar que el código no apague el sonido si no se ejecuta en un ámbito controlado.

Para el caso de correr el programa en qemu se requirió de agregar las lineas “-soundhw pcspk”, que habilita el uso físico del parlante de la pc.

IDT

Para la carga del IDT nos basamos en el ejemplo proporcionado por la cátedra, cargamos en la posición 21 el teclado, en la 20 el time, en la 00 la excepción de división por 0 y en la 06 la excepción de operación inválido. Como fue solicitado, también usamos la 80 para las system calls.

Marcamos la máscara del PIC para habilitar teclado y timer tick únicamente, dado que son lo único que utilizamos en este TP que requería ser habilitado por el PIC.

Excepciones

Cargamos las excepciones de división por 0 y operación inválida en las posiciones 0x00 y 0x06 respectivamente dado que estas son las posiciones especificadas para dichas excepciones. Cuando se comete alguna de estas dos excepciones, en el stack se pushea el SS, RSP, FLAGS, CS, RIP, y el error code (el cual no se guarda para las excepciones mencionadas previamente) antes de ir a buscar la rutina de atención adecuada a la IDT y correrla.

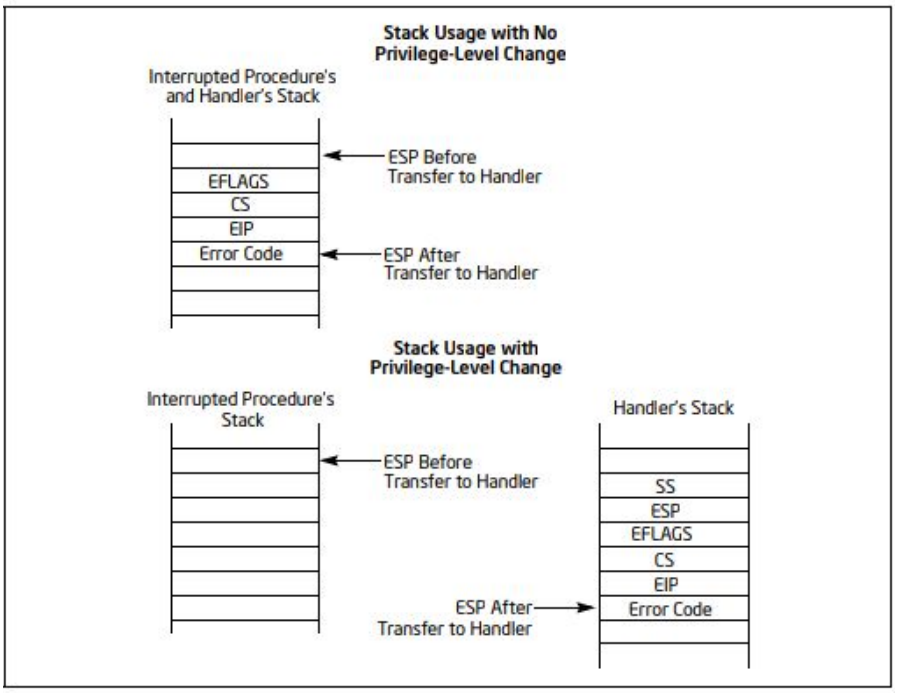

Figure 6-5. Stack Usage on Transfers to Interrupt and Exception Handling Routines

https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-manual-325462.pdf

Al entrar a la rutina de la excepción, se crea el stack frame y se pushean todos los registros para que los mismo mantengan sus valores cuando se llama a registerValueToString, a la cual se le pasa un buffer en donde se guardará el valor del registro RDI (primer argumento) como un string. Con esto, se imprime el mensaje de error adecuado y con [rbp + 8] se accede al valor del instruction pointer en el momento de la excepción, al igual que [rbp + 16] accede al valor del CS, [rbp + 32] a RSP y [rbp + 40] a SS, imprimiendo el valor de cada uno y luego poppeando el valor de los registros restantes para su impresión. Como ambas rutinas hacen lo mismo, se creo una macro con un parámetro, el string del error adecuado. Al final de la rutina, se retorna con un IRET, la cual restablece el valor que tenían CS y RIP antes de la excepción, restablece el EFLAGS, incrementa el stack pointer apropiadamente y resume la ejecución del proceso.

Sin embargo, antes de retornar reiniciamos el shell llamando al main nuevamente. Esto se hizo dado que luego de la excepción el shell no aceptaba nuevos comandos. En un SO que maneje el concepto de proceso se debería eliminar el proceso que provocó la excepción pero en este TPE no tenemos un verdadero administrador de procesos (Merovich).

System calls

La función syscall fue cargada en la posición 80h de la IDT y la misma se asemeja al comportamiento real de los system calls. Hay 9 system calls que definimos donde cada uno tiene un código que la representa: sys_ticks con código 0x01 (devuelve cantidad de ticks transcurridos), sys_sec con código 0x02 (devuelve cantidad de segundos transcurridos), sys_read con código 0x03 (lee del buffer una cantidad especificada), sys_write con código

0x04 (escribe del buffer una cantidad especificada), sys_time con código 0x05 (guarda en el buffer pasado la hora), sys_pixel con código 0x06 (imprime un pixel en pantalla en una posición especificada), sys_clear con código 0x07 (limpia la pantalla de la terminal), sys_beep con código 0x08 (empieza el sonido) y sys_unbeep con código 0x09 (termina el sonido).

Se creó en userland funciones en asm para llamar al system call que se desea, donde dentro de la función se carga en RDI (primer parámetro) el código correcto del system call. Las mismas son os_ticks, os_sec, read, write, os_time, os_pixel, os_clear, os_beep y os_unbeep. Luego de cargar en RDI el valor correcto y en los demás registros los parámetros adicionales en los casos que sean necesarios, con un int 80h se va a la IDT a buscar la función syscall cargada en la posición 80h. Dentro de la misma, se compara RDI con todos los posibles códigos, llamando a la system call adecuada. Si el código no se encuentra entre los enumerados previamente, el syscall devuelve el control al proceso sin hacer nada.

Las funciones os_ticks y os_sec son utilizadas para calcular la cantidad de ticks que transcurren por segundo y para tener noción de tiempo en el pong. Sabemos por como está definido el timer tick que ocurren 18 ticks por segundo, sin embargo, sería bueno hacer que la función que creamos, ticksPerSecond(), calcule ella misma cuantos transcurren para cuando no se sabe este valor de antemano (no pudimos hacer que funcione correctamente entonces hicimos que devuelva 18). De esta manera, el funcionamiento del pong sería independiente al timer tick.

Se trató de asemejar las funciones read y write con aquellas de la librería de C. os_time se usó para obtener la hora del RTC. os_pixel, os_beep, os_unbeep también son utilizadas en el pong.

Shell

Para el shell usamos un simple while exit pues era la versión más simple que encontramos de mantener al shell andando. Con scanf tomamos caracteres del teclado y los imprimimos al mismo tiempo. Al final quedan guardados en command que luego comparamos con nuestra implementación de strcmp con los comandos que definimos en shell_execute y finalmente llamamos a esos métodos que están definidos en modules.c.

En un principio definimos una función propia de Shell llamada shell_get que hacía lo mismo que hace el scanf que implementamos en stdio, pero decidimos hacer scanf genérico y dejarlo en stdio.Antes de llamar a la interrupción int 80h se cargan los parámetros indicados para el system call que se desee y dentro de syscall se compara el primer parámetro con ciertos códigos para saber a que system call se quiere llamar.

Stdio

Para nuestro stdio definimos getChar, putChar, printf, scanf y además getCharWithZero. Los primeros cuatro son estándar y hacen uso de las funciones read and write para leer del buffer del teclado o imprimir a pantalla, pero el último lo definimos para usarlo en Pong. A diferencia de getChar este método no espera al input del usuario, sino

que lee del buffer un char, de esta forma podemos obtener todos los input de teclado entre refresh del pong. Si usamos getChar, el pong cuelga esperando inputs y no funciona.

Pong

Se quiso representar una clase objeto dentro del funcionamiento de c y por ende se han utilizado variables static. A pesar de que el no uso de estructuras de datos facilitó el código en varias situaciones, también causó dificultades a la hora de diseñar las funciones debido a la poca flexibilidad de las variables (ej. no poder mandar una raqueta para el move y tener dicha función genérica en vez de moveDownL y sus parecidos).

Para representa la pelota se guardan 2 ints para su posición actual y otros 2 ints su velocidad, es decir cuánto se desplaza en el próximo tick.

Durante la actualización de la posición de la pelota, se estima si este colisiona con el techo o el piso en la próxima movida, y en dicho caso invierte su componente Y del vector a su valor se le multiplicó por -1 para invertir la dirección. También verifica en caso de que su posición X es suficientemente alto o bajo para colisionar con la raqueta y si ocurre compara si la posición de la pelota o algún punto de su circunferencia se encuentra dentro del rango “Posición de la raqueta” +/- su alcance. En dicho caso invierte la componente x para dirigirse hacia el lado del oponente.

Para el “rebote” de la pelota con la raqueta simulamos el movimiento aleatorio (no es completamente random) mediante una multiplicación de la variable “bRandom” con un número primo y tomando los últimos dígitos del resultado. Este comportamiento causa que los movimientos sigan cierto patrón pero se le dió poca importancia a este hecho dado que la existencia del patrón no pareció ser un factor deteriorante en la calidad el juego (como lo podría ser en un juego de azar como el blackjack o póker).

En un principio, para optimizar el movimiento de la raqueta, se trató de evitar dibujar la raqueta entera y reutilizar los segmentos de raquetas ya existentes que se mantienen a través del movimiento. Esto se lograría, si por ejemplo la raqueta se mueve para abajo, borrando la mitad superior y dibujando esa longitud en la parte inferior y de tal manera se ahorraría el dibujo de mitad de la raqueta. Sin embargo, esto se complicó mucho y no se logró su funcionamiento correcto, y como consecuencia, se usó el mecanismo que se trató de evitar, refrescar el dibujo de la raqueta entera.

También se tuvo que hacer un refresco constante de la línea punteada y del score debido a que cuando la bolita pasaba por encima, borra los píxeles de los mismos. Esto se debe a que la función moveBall() pinta de negro la posición actual de la bolita y luego pinta de blanco a la posición nueva. Una posible solución es analizar al mover la bolita si el pixel está pintado para luego no pintarlo de negro, sin embargo esto no sólo sería complejo, sino que se perdería mucho tiempo haciendo interrupciones para analizar el estado de cada pixel. Se perdería más tiempo que el refresco constante que se soporta actualmente.