LuccianOS’

Un sistema operativo con buen gusto 🍦

Informe - Trabajo Práctico Especial

Arquitectura de las Computadoras (72.08)

Profesores: Santiago Valles, Federico Ramos y Giuliano Scaglioni

Grupo: 8

Integrantes:

Oliveto, Lucia	loliveto@itba.edu.ar	64646
Pietravallo, Tomas	tpietravallo@itba.edu.ar	64288
Wehncke, Máximo	mwehncke@itba.edu.ar	64018

Introducción

LuccianOS' cuenta con un *Kernel* y programas de usuario. Al iniciar, el código comienza corriendo una base de código provista por la cátedra "PURE64" el cual preparará ciertos aspectos fundamentales del sistema operativo para que el kernel pueda comenzar a correr. Una vez que esto ocurre, el *Kernel* carga a memoria los módulos de usuario, la Tabla Descriptora de Interrupciones (*IDT*), y comenzara a correr una terminal de usuario.

Diseño

Interrupciones

Se habilitan en la *IDT* las respectivas entradas para cada interrupción: dos interrupciones de hardware (timer tick y teclado), y una interrupción de software. (0x80).

Excepciones

Se habilitan las respectivas entradas de *IDT*: la 0 para la interrupción de división por cero y la 6 para la interrupción por código inválido. Al generarse alguna de estas dos, automáticamente se corre la rutina de atención de excepción. Durante la cual se guarda el valor de los registros en un arreglo (se obtiene el valor de *RIP* y *RFLAGS* de la pila generada por la excepción).

Luego de guardar los registros, se procede a llamar a *exceptionDispatcher* con el código de excepción y el valor de los registros, el cual se encarga de manejar la excepción correspondiente. En ambos casos se imprime en pantalla el valor de los registros. Luego se restaura el stack pointer a su valor al comenzar la ejecución, y se retorna a la *shell* (de este modo restaurando el programa a su estado luego de que el *Kernel* termine de configurar el entorno).

System calls

Para no generar conflictos con *system calls* existentes en otros sistemas operativos o estándares – en caso de que en un futuro se quisieran portar –

implementamos las system calls específicas de nuestro sistema con códigos mayores a 0x8000000. Por ejemplo las *system calls read* y *write*, que imitan el comportamiento de las mismas en Linux, utilizan los mismos identificadores y argumentos.

Driver de teclado

Internamente mantiene un buffer de tamaño fijo, el cual se consume a medida que el código de usuario lee los caracteres. De superarse la capacidad del buffer interno este se reinicia, descartando cualquier entrada anterior.

Para obtener los *scancodes* correspondientes a cada tecla, utilizamos mayormente el código fuente de QEMU (*ver referencia 1*), y un recurso online detallando los diferentes códigos cuando se presiona y suelta una tecla (*ver referencia 2*).

Driver de video

El driver de video se construyó sobre el código provisto por la cátedra ("*videoDriver.c*" y archivos asociados). A través de *system calls* se expone la funcionalidad de imprimir rectángulos y círculos con las dimensiones deseadas, de modo que una sola llamada al *kernel* pueda pintar varios pixeles, reduciendo los llamados a función necesarios para pintar la totalidad de la pantalla.

Este driver, cuyas funciones se utilizan para cada píxel de la pantalla tiene el potencial de ser utilizado millones de veces por segundo, por ende se compila con la opción "-O3" de gcc. Así, potencialmente, miles de llamadas sin optimizar a *putPixel* pueden convertirse en un llamado a la optimizada *drawRectangle*.

Además, la función *scrollVideoMemoryUp*, que realiza repetidos accesos secuenciales a memoria, se optimizó para maximizar la posibilidad de utilizar la memoria caché al acceder a la memoria de video – notamos que esto mejoró notablemente la velocidad.

Driver de sonido

Se implementó utilizando el *PIT* (Programmable Interval Timer). Este se encarga de reproducir un sonido con una determinada frecuencia. El driver provee una función para iniciar un sonido y otra para detenerlo.

Nota: Se requiere de un sleep entre medio de dichas llamadas para definir la duración del sonido.

Renderizado de texto

Se utilizó un mapa de bits de dominio público (*ver referencias 3, 4 y 5*). Si bien la opción de utilizar diferentes tipografías no fue implementada, se modularizó el código de modo que esto pueda ser implementado a futuro.

Al igual que el acceso directo a la memoria de video por parte del driver de video, el renderizado de texto se compila con "-O3" para mantener veloz la respuesta a las interacciones del usuario.

También se implementó el manejo de diferentes secuencias de escape (nueva línea, tabulador, etc) para facilitar la impresión de secuencias complejas de texto. La lista completa de secuencias válidas se puede consultar en el manual de usuario.

Shell

Se realizó un intérprete de comandos con comandos básicos como imprimir argumentos, forzar excepciones, imprimir últimos comandos que fueron ejecutados, imprimir el valor actual de los registros del procesador, imprimir la fecha, incrementar/decrementar el tamaño de fuente, entre otros.

Todos los comandos disponibles se encuentran referenciados al correr el comando *help*, que explica qué hace cada uno.

Se implementó que la pantalla de la *shell* tenga *scroll,* es decir que al llegar al final de la pantalla e intentar imprimir, lo que está impreso se mueva para arriba, dejando así lugar a la nueva impresión.

Se implementó un cursor, para que el usuario sepa dónde se escribirá a continuación.

Se añadió el uso de las flechas arriba y abajo, para iterar por los comandos recientemente ejecutados.

Para cualquier comando, si se le provisionan más argumentos de los que solicita, estos son ignorados.

Snake

El juego *snake* fue implementado en *Userland*, pues es un programa de usuario, no esencial para el sistema.

Al correr el comando *snake* en la consola se inicia el programa, el cual despliega un mensaje de bienvenida al usuario, dándole las opciones de elegir jugar de a 1 o 2 jugadores y el nivel de dificultad, que está relacionado a la velocidad en que se mueven las *snakes*. Luego muestra las indicaciones sobre qué teclas corresponden a cada jugador, el objetivo del juego y cómo salir en cualquier momento. Para esta última funcionalidad se eligió la tecla X, al presionarla se vuelve a la *shell*. El juego finaliza cuando uno de los jugadores colisiona contra un borde u otro jugador. Ante dicha situación, se muestran en pantalla los *scores* de cada jugador, quien/es perdieron o ganaron, y se dan las opciones de salir (X) o volver a jugar (ENTER). Si se elige esta última, se vuelve a jugar con la misma configuración (cantidad de jugadores y dificultad) de la partida anterior. Si se elige salir, se vuelve a la *shell*.

Se decidió crear un *struct* para cada *snake*, conteniendo su “cuerpo”, puntaje, dirección, color y estado (perdedor/ganador). De esta manera, cada *snake* guarda sus características. Para cada “parte del cuerpo” se creó otro *struct* con su posición en el tablero.

Se implementó una función que calcula pseudo aleatoriamente (*ver referencia 6*) las posiciones donde colocar la comida al iniciar el juego y cada vez que fuera consumida por una *snake*. Se creó un *struct* para la comida, teniendo este la información sobre su color y posición.

Cada vez que avanzan las *snakes*, se chequea si alguna sobrepasó algún límite del tablero, si chocó contra otra *snake* o contra ella misma y si “comió” la comida. Ante cualquiera de ellas, se alerta al usuario mediante un sonido (distintos para colisiones y comida).

Se modularizó el código de modo que, en un futuro, se puedan agregar más jugadores.

Problemas encontrados

- Duración fija de *timer tick*:
  Dado que el timer tick interrumpe en un intervalo de tiempo determinado, cuya duración no se puede cambiar, no fue posible implementar mayores

niveles de dificultad en el juego *snake*, es decir, hacer que las *snakes* se movieran más rápido, ni tampoco sonidos de menor duración.

- *scrollVideoMemory*:

  La implementación de *scrollVideoMemory* (utilizada al hacer un *scroll* de la pantalla) no preserva los pixeles que se "van" por el borde superior de la pantalla. Debido a esto, no se puede ver la actividad de la terminal hacia atrás. Además, si se ingresa una entrada lo suficientemente larga para generar un *scroll*, y luego se desea borrarla, esta efectivamente se borrara del buffer pero no se podrá mostrar la porción de la entrada que se encontraba en pantalla previo al *scroll*. Consideramos esto como un problema conocido y esperado ante este escenario.

- *SSE* y *stdarg*:

  Se observó que el uso de *stdarg* en conjunto con *doubles* utilizaba operaciones que requieren de *SSE* (*Streaming SIMD Extensions*) las cuales están deshabilitadas en compilación. Debido a esto, evitamos el uso de tipos de datos de punto flotante en funciones variadicas (como *printf* o *scanf*).

- Sobreescritura accidental de la *IDT*:

  Durante el desarrollo del trabajo práctico, se descubrió que desde *Userland* se podía pisar la *IDT* accidentalmente. Se corrigió la sección de código que la sobreescribía pero no se implementó protección contra esto dado que implicaría el uso de segmentación y/o paginación. Dado que la *IDT* es cargada en 0x00000000, y *NULL* es definido como (void *) 0x0, cualquier escritura en *NULL* (o las posiciones de memoria adyacentes) resultaría en una sobreescritura de la *IDT*.

- Fondo del juego *snake*:

  En la implementación del juego *snake*, más específicamente en la parte "dibujar" en pantalla, en un principio se optó por redibujar el fondo luego de cada "avance" de las *snakes*. Pero se vió que algunos objetos dibujados sobre este titilaban y era consecuencia de redibujar el fondo y luego los objetos. Entonces se solucionó dibujando por completo el fondo sólo al comienzo del juego y luego, en cada "avance" de las *snakes*, darle el color del fondo a las posiciones que las *snakes* dejaron de ocupar.

A implementar en una refactorización del código

Algunas cosas que nos gustaría haber hecho diferente, pero debido a tiempo y circunstancias no pudimos cambiar incluyen:

- Investigar el uso del *RTC* para generar interrupciones y obtener una mejor precisión.
- Mantener un historial del texto que es desplazado de la pantalla por *scrollVideoMemory* para que pueda ser restaurado si así se deseara.
- Agregar segmentación de la pantalla para poder mantener varias ventanas (por ejemplo un header con la hora actual)
- Configurar la paginación de memoria para que la escritura a zonas de memoria no deseadas generen una excepción.
- Habilitar el uso de *SSE* para utilizar stdarg con tipos de dato de punto flotante
- Mejorar el código del juego *snake* para desvincular el renderizado de los sonidos.

Apéndice de material referenciado

1. https://github.com/qemu/qemu/blob/master/pc-bios/keymaps/en-us
2. http://flint.cs.yale.edu/feng/cos/resources/BIOS/Resources/assembly/makecodes.html
3. https://github.com/dhepper/font8x8
4. https://wiki.osdev.org/PC_Screen_Font
5. https://wiki.osdev.org/Drawing_In_a_Linear_Framebuffer
6. https://wiki.osdev.org/Random_Number_Generator
7. https://github.com/hughsk/glsl-hsv2rgb/blob/master/index.glsl – utilizado para el color de las *snakes*.
8. https://wiki.osdev.org/VESA_Video_Modes – utilizado para el driver de video.
9. https://wiki.osdev.org/Interrupt_Descriptor_Table – utilizado para obtener información acerca de la *IDT*.
10. https://os.phil-opp.com/cpu-exceptions/#the-interrupt-stack-frame – utilizado para el *stack* de excepciones.
11. http://www.nacad.ufrj.br/online/intel/vtune/users_guide/mergedProjects/analyzer_ec/mergedProjects/reference_olh/mergedProjects/instructions/instruct32_hh/vc316.htm – utilizado para generar la excepción de código inválido.
12. https://wiki.osdev.org/Programmable_Interval_Timer – utilizado para el driver de sonido.