DEPARTAMENTO
DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico III

System Programming

Organización del Computador II

Segundo Cuatrimestre de 2020

Integrante	LU	Correo electrónico
Ivo Pajor	460/19	ivo_pajor@hotmail.com
Laureano Muñiz	498/19	lau2000m@hotmail.com
Luciana Gorosito	577/18	lugorosito0@gmail.com

Facultad de Ciencias Exactas y Naturales

Universidad de Buenos Aires

Ciudad Universitaria - (Pabellón I/Planta Baja)

Intendente Güiraldes 2160 - C1428EGA

Ciudad Autónoma de Buenos Aires - Rep. Argentina

Tel/Fax: (54 11) 4576-3359

http://www.fcen.uba.ar

Índice

1. Introducción **3**

2. Desarrrollo **3**

2.1. Ejercicio 1 3
2.2. Ejercicio 2 4
2.3. Ejercicio 3 4
2.4. Ejercicio 4 5
2.5. Ejercicio 5 5
2.6. Ejercicio 6 5
2.7. Ejercicio 7 5
2.8. Ejercicio 8 5
2.9. Ejercicio 9 (opcional) 5

1. Introducción

El objetivo de este trabajo práctico es aplicar gradualmente los conceptos de *System Programming* vistos en las clases teóricas y prácticas, mediante la implementación de una serie de ejercicios que, inspirados en la serie *Rick y Morty*, en conjunto conformarán un kernel o un pequeño sistema operativo.

2. Desarrrollo

A continuación detallamos las implementaciones de los ejercicios del trabajo práctico.

2.1. Ejercicio 1

Para la realización de este ejercicio analizamos las estructuras **gdt_entry_t** y **gdt_descriptor_t** definidas por la cátedra. En esta implementación, la Tabla de Descriptores Globales(GDT) es un arreglo de **gdt_entry_t** y su descriptor, que luego cargaremos en GDTR, es del tipo **gdt_descriptor_t**.

Siguiendo lo indicado en el primer item, definimos a partir del índice 10, 4 descriptores de segmento en la GDT utilizando la estructura **gdt_entry_t**, atendiendo a las propiedades particulares de cada segmento. Así definimos las entradas de dos segmentos de código de nivel 0 y 3 y de dos segmentos de datos, también de nivel 0 y 3. Puesto que estos segmentos deben direccionar los primeros 201 MB de memoria establecimos en todos el bit de G en 1, y definimos su base en 0x00000000 y su limite en 0x00C8FF. Además, como son segmentos de código o datos de 32 bits, los bit S y D/B se encuentran seteados en 1 y el bit L en 0. Los bits de DPL de cada uno están seteados en 0 o en 3 de acuerdo con el nivel de privilegio que le corresponda. Por último, los bits de tipo están seteados como 0xA en caso de tratarse de un segmento de código y como 0x2 en caso de tratarse de un segmento de datos. En todos, el bit P se setea en 1 para reflejar que los segmentos están presentes. En el caso del bit de AVL, como es un bit reservado lo seteamos en 0.

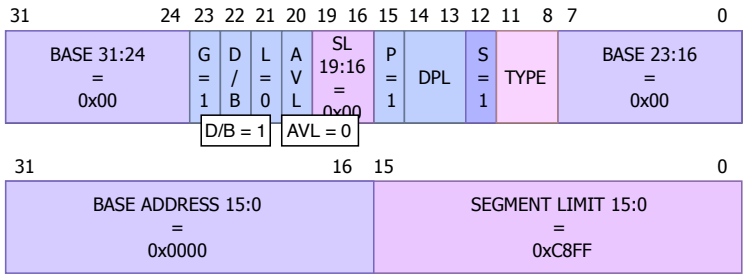

Figura 1: Esquema general de un descriptor de la GDT para los 4 segmentos.

Realizando lo pedido en el item b, pasamos a modo protegido. Para poder hacer esto modificamos el archivo kernel.asm, allí deshablitamos las interrupciones, cargamos en el registro GDTR la estructura **gdt_descriptor_t** y modificamos el ultimo bit del registro de control CR0, es decir, seteamos en 1 el bit de *Protection Enable*. Posteriormente, escribimos el código necesario para saltar efectivamente a modo protegido. Debido a que este salto se consigue haciendo un *far jump* a la próxima instrucción, designamos una etiqueta llamada **modo_protegido** a partir de la cual obtendremos el offset, mientras que como selector utilizamos el correspondiente al segmento de código de nivel 0. Además, seteamos la pila del kernel en la dirección 0x25000, es decir, en la base de la pila. Una vez que pasamos a modo protegido, cargamos correctamente los selectores de segmento, en los registros ds, es, fs, gs, ss, usando como registro auxiliar ax, donde se encontraba el selector de segmento de nivel 0 (RPL = 0).

Seguidamente, definimos otra entrada en la GDT destinada a un segmento de vídeo de nivel 0 (DPL = 0). A diferencia de los cuatro segmentos anteriores, la base de este segmento es distinta de 0, por lo que los campos de la base fueron ser completados con el valor 0x000B8000. Como la pantalla es de 80x50 y como cada pixel ocupa 2 bytes, el límite de este segmento es 0x01F3F. Como este límite entra en 20 bits,

el bit de granularidad de este descriptor queda en 0. Además, al ser un segmento de datos de lectura y escritura el tipo es 0x02 y el bit de s queda seteado en 1. Como en las entradas anteriores, los bits de AVL y L quedan en 0, mientras que los bits de present y D/B quedan en 1.

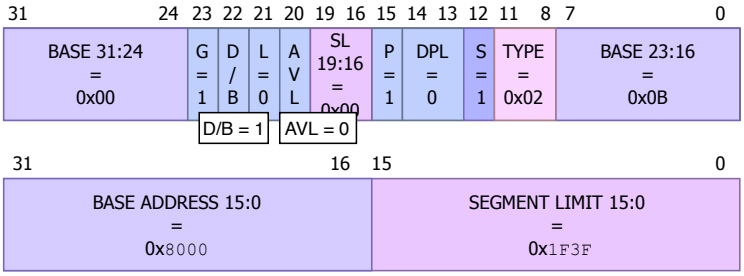

Figura 2: Esquema del descriptor de segmento de video.

Para finalizar este ejercicio, realizamos un rutina encargada de limpiar la pantalla, y pintar el área del mapa de color verde, junto con las barras de los jugadores Rick y Morty. Inicialmente, esta rutina fue implementada en ASM dentro del archivo *kernel.asm*, para corroborar el correcto funcionamiento del acceso al segmento de vídeo, pero luego fue reemplazada por la función en C ïnicializar_pantalla"que se encuentra en el archivo *screen.c*. La pantalla se ve de la siguiente manera:

2.2. Ejercicio 2

Inicialmente modificamos los archivos *idt.c* e *idt.h* para completar las entradas de la IDT. Definimos a la IDT como un arreglo de idt_entry_t de 255 posiciones, que primero inicializamos en cero, y el descriptor de la IDT (IDT_DESC), indicando el tamaño y la dirección de la IDT. Para inicializar las entradas de la IDT utilizamos un *DEFINE* que, dado el número de la interrupción, completa los campos del descriptor de la IDT correspondiente. Para el caso de las interrupciones de software el DPL es 3, dado que las mismas deben poder ser accedidas por las tareas y en los demás casos el DPL es 0, ya que no deben poder ser accedidas por las tareas. En todos los casos el selector de segmento es de código de nivel 0, porque las rutinas no pueden ser modificadas por las tareas. Como la dirección de la IDT no está definida en tiempos de compilación fue necesario definir una función llamada idt_init que inicializara las entradas de la IDT en tiempos de ejecución. Definimos 21 entradas en la IDT para las excepciones del procesador(de 0 a 20).

Para imprimir las excepciones en la esquina superior izquierda de la pantalla cuando estas ocurren, implementamos la función "print_exception", que se encuentra en el archivo *screen.c*. La misma es llamada inicialmente en el código de *isr.asm*, aunque luego esto será modificado como se ve en el código, ya que esta acción será realizada por el modo debug.

2.3. Ejercicio 3

En este ejercicio definimos dos entradas en la IDT para las interrupciones externas o de hardware (32 del clock y 33 del keyboard) y cuatro entradas para las interrupciones de software(88, 89, 100, 123).

De acuerdo a lo pedido en el segundo, tercer y cuarto item, dentro del archivo *isr.asm* escribimos el esqueleto de las rutinas de interrupciones del reloj, de teclado y de software, que luego irán siendo modificadas a lo largo del transcurso de los ejercicios.

Rutina de interrupción del reloj

Esta rutina comienza con un call a la función pic_finish1, para avisar que la interrupción fue atendida. Seguidamente se llama a la función "next_clock", provista por la cátedra. La misma se encarga de imprimir por pantalla una animación del un cursor rotando en la esquina inferior derecha de la pantalla.

Rutina de interrupción del teclado

Esta rutina comienza levantando del puerto 0x60 la tecla que fue presionada durante la interrupción. Para que esta tecla sea impresa por pantalla se implementó la función ”print_digito” en el archivo *screen.c*. Para finalizar, hacemos un call a pic_finish1, para avisar que la interrupción fue atendida.

Rutinas de interrupciones de software

Las rutinas de interrupciones de software solo mueven a eax los números indicados por el item d, esto luego será modificado.

2.4. Ejercicio 4

Para realizar este ejercicio modificamos los archivos *mmu.h* y *mmu.c*. Inicialmente completamos la función “mmu_init” con la dirección de la primera página libre del kernel (0x100000), y luego completamos la función “mmu_next_free_kernel_page” para que nos devuelva la dirección de la próxima página libre del área libre del kernel.

Para inicializar el directorio y las tablas de páginas del kernel, implementamos la función “mmu_init_kernel_dir”, que crea un directorio y una tabla de páginas (tabla_0) del tipo *page_directory_entry* y *page_table_entry*, respectivamente. Estas estructuras fueron definidas en el archivo *mmu.h* y siguen el formato de las entradas al directorio y a la tabla de páginas. Una vez creados los inicializamos en 0.

La primera entrada del directorio (directorio[0]) va a mapear a la tabla de páginas creada y esta mapeará a todo el kernel, por lo que completamos el campo de la base del directorio[0] con la dirección de la tabla de páginas. El privilegio de esta entrada es 0. Una vez mapeado el directorio a la primera tabla de páginas, mapeamos la misma al kernel, usando identity mapping. Nuevamente, el privilegio de cada entrada de la tabla de páginas es 0. Al final de la rutina devolvemos la dirección del directorio de tablas de páginas.

Ya creadas las funciones para inicializar el directorio y la tabla de páginas, escribimos en *kernel.asm* el código necesario para activar paginación: inicializamos el manejador de memoria y el directorio de páginas mediante un call a la función “mmu.init” y “mmu_init_kernel_dir”, respectivamente. Luego, cargamos en el registro cr3 la dirección del directorio y activamos paginación seteando en 1 el bit 31 de cr0.

2.5. Ejercicio 5

2.6. Ejercicio 6

2.7. Ejercicio 7

2.8. Ejercicio 8

2.9. Ejercicio 9 (opcional)