DEPARTAMENTO
DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico III

Informe

Organización del Computador II

Segundo Cuatrimestre de 2020

Integrante	LU	Correo electrónico
Tomas Curti	327/19	tomasacurti@gmail.com
Guido Rodriguez Celma	374/19	guido.rc98@gmail.com
Fermin Schlottmann	160/19	fermin.schlottmann@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. Ejercicios **3**

2.1. Ejercicio 1 3
2.2. Ejercicio 2 3
2.3. Ejercicio 3 3
2.4. Ejercicio 4 4
2.5. Ejercicio 5 4
2.6. Ejercicio 6 5
2.7. Estructuras del Juego 5
2.8. Ejercicio 7 5
2.9. Ejercicio 8 6

1. Introducción

El presente informe se enmarca dentro del tercer trabajo práctico de la materia Organización del Computador II. En el mismo vamos a describir las estructuras e implementaciones que realizamos a fin de implementar el sistema pedido.

2. Ejercicios

2.1. Ejercicio 1

Definimos cuatro segmentos en la Tabla Global de Descriptores (GDT), segmentos de código y de datos de nivel 0 y nivel 3. Para esto, los declaramos en la tabla de entradas \gdt_entry_t" con sus atributos correspondientes.

Los 4 segmentos están definidos para el mismo área de memoria (los primeros 201MB, sin considerar las primeras 10 posiciones que se consideran usadas), es decir, son segmentos \Flat", están solapados uno sobre el otro. Usaremos los mismos para acceder al modo protegido.

Luego, definimos un segmento adicional en la GDT, este será de datos y tendrá nivel de privilegio 0. Lo usaremos para acceder y escribir en la pantalla.

Para pasar a modo protegido, primero deshabilitamos las interrupciones de reloj y teclado con "cli". Luego llamamos a las funciones encargadas de habilitar y chequear el funcionamiento de la linea A20 del bus de direcciones del sistema. Después cargamos la GDT con los segmentos que definimos anteriormente, usando la instrucción "lgdt [GDT_DESC]", donde GDT_DESC es una estructura que contiene la longitud de la tabla y su dirección lineal. Para terminar seteamos el bit PE del registro CR0, saltamos al segmento de código de nivel 0 usando la instrucción JMP [CS_RING_0:modo_protegido] (el "gran salto") y cargamos los registros de segmento.

2.2. Ejercicio 2

Definimos 21 entradas en la IDT que van desde 0 hasta 20, una por cada excepción del procesador. Para esto, inicializamos, cada una, con la función "idt_init". Esta función se encarga de hacer las asignaciones correspondientes de offset y atributos para cada interrupción en la IDT.

Luego, en el archivo isr.asm, definimos una macro que aplique a cada una de ellas. La misma se encargará de llamar una función que imprima por pantalla una descripción de la excepción que se produjo en que linea, columna y en un color determinado. Además, a modo de interrumpir la ejecución, esta macro se quedará loopeando infinitamente.

Para hacer que el procesador utilice las entradas definidas anteriormente, en el archivo que contiene el código de nuestro Kernel, llamamos a "idt_init", luego cargamos la IDT con la instrucción "lidt [idt_desc]", donde idt_desc es una instancia de la estructura idt_descriptor_t, la cual contiene el tamaño de la tabla menos 1 y la posición de memoria donde se encuentra.

Finalmente llamamos a las funciones pic_reset y pic_enable que se encargan de activar el controlador de interrupciones del CPU. Así estamos en condiciones de habilitar las interrupciones con la instrucción "sti".

2.3. Ejercicio 3

Definimos 6 entradas en la IDT, una para la interrupción del reloj, una para la del teclado y 4 para las demás interrupciones pedidas, inicializamos las mismas con la función "idt_init" que usamos para las interrupciones excepcionales. Las mismas serán de nivel 0, pues el kernel se encargará de atenderlas.

Para la interrupción del reloj, primero activamos y configuramos PIC, la rutina se encargará de llamar a una función que le avisa al PIC que se recibió la interrupción y luego a otra que se encargue de imprimir en pantalla la animación del reloj(\, —, /), luego se quedara haciendo un loop infinito.

Para la interrupción del teclado, la rutina mueve al registro "al" un Byte del puerto de I/O y pasa el mismo por parámetro a la función "printScanCode" que lo imprimirá por pantalla si es un make pero

no cuando sea un break. Finalmente le avisa al PIC que se recibió la interrupción y se queda haciendo el mismo loop que en el reloj.

Las demás interrupciones simplemente mueven el valor correspondiente al registro eax y le avisan al PIC que se recibió la interrupción.

2.4. Ejercicio 4

Definimos punteros a una entrada en la tabla de directorios (PD) en la posición 0x25000 y a una entrada en la tabla de páginas (PT) en la 0x26000. El directorio tiene 1024 entradas y cada pagina ocupa 4KB, esto nos da un total de 4MB de memoria utilizada para la PT.

Luego inicializamos en 0 las entradas de las nuevas tablas y definimos en la primer entrada de la PD, la dirección que apunta a la base de la tabla de páginas que definimos previamente.

Finalmente, definimos las entradas de la nueva tabla de páginas con sus atributos y direcciones correspondientes y devolvemos un puntero a la PD.

Para activar paginación, en nuestro kernel (luego de pasar a modo protegido) llamamos a la función "mmu_init" que se encarga de inicializar el manejador de memoria. Luego llamamos a "mmu_init_kernel_dir" que realiza la operatoria descripta anteriormente. Finalmente habilitamos paginación moviendo al registro CR3 la dirección física donde definimos nuestro directorio de páginas (base de la PD) junto con los atributos correspondientes y seteando el bit PG del registro CR0.

2.5. Ejercicio 5

Las siguientes rutinas se encuentran definidas en el archivo "mmu.c".

mmu_init: Declaramos un puntero a la próxima página libre en el área de memoria del Kernel, esta función simplemente inicializa el puntero con el valor de la primer página libre del Kernel (0x100000). Definimos además la función "mmu_next_free_kernel_page" que devuelve el valor del puntero y lo actualiza para que apunte a la siguiente página libre.

mmu_map_page: Esta rutina es invocada al inicializar los jugadores y cada vez que se quiera crear o mover un meeseek. Para esto primero obtiene, de la dirección virtual pasada por parámetro, los índices del directorio y tabla de página en la que haremos el mapeo. Luego, conseguimos la entrada en la PDT de la tarea actual, a partir del CR3 que recibimos por parámetro.

Si la tabla de páginas en la que queremos mapear no está presente, creamos una, pidiendo una pagina libre al kernel e inicializando las PTE en 0 y se la asignamos al PDT en la entrada correspondiente.

Para finalizar, llenamos la entrada de la PT que indica el índice que calculamos la principio, usando la dirección física y los atributos pasados por parámetro. Además limpiamos la TLB con la función "tlb_flush"

mmu_unmap_page: Análogamente a la rutina anterior, obtenemos los índices del PDT y PT de la página a borrar, luego usando el CR3 llegamos a la PT correspondiente y simplemente seteamos en 0 el flag presente de la página a desmapear. Finalmente, limpiamos la TLB.

mmu_init_task_dir: Esta rutina es invocada cuando se crean las tareas correspondientes a los jugadores. La misma pide una página libre al kernel para guardar el PD de la tarea y la inicializa en 0. Luego mapea las 4 páginas correspondientes al código de la tarea entre la posición virtual (0x1D00000) y la posición física pasada por parámetro. Este mapeo se hará por duplicado, para el kernel y para la tarea del jugador, con los atributos correspondientes a cada una. La razón de este doble mapeo es que el código de la tarea se encuentra en el área del kernel, la cual vive en nivel 0, mientras que las tareas viven en nivel 3.

Una vez mapeadas las posiciones necesarias, se copia el código desde la posición pasada por parámetro a la posición virtual de las tareas. Al terminar de copiar el código, borramos las páginas que le habíamos mapeado al kernel.

Finalmente devolvemos el puntero al PD que creamos para la tarea.

2.6. Ejercicio 6

En tiempo de compilación tendremos definida únicamente la entrada en la GDT correspondiente a el TSS inicial, que usaremos como blueprint para todos los demás descriptores de TSS en la GDT.

En tiempo de ejecución llamaremos a la función "init_tss_desc" la cual se encargará de crear las entradas en la GDT para los descriptores de las demás tareas y completar los campos de base de todas las mismas con su posición correspondiente en la página que pedimos al kernel para almacenar los TSS de todas las tareas.

Una vez definidas las entradas en la GDT para las tareas, llamamos a la función "tss_init_idle" que consigue la dirección que definimos en la GDT para alojar el TSS de IDLE y le carga la estructura que inicializamos previamente con los datos correspondientes a la tarea IDLE.

Para ejecutar la tarea IDLE primero cargamos en el Task Register (tr) el selector de TSS Inicial, allí se guardará el contexto inicial de los registros antes de iniciar el juego. El mismo no volverá a ser accedido. Luego hacemos un salto con cambio de contexto (jmp far) usando el selector de TSS de la tarea IDLE.

Construimos la función "tss_init" la cual recibe la direccion donde está alojado el TSS, la dirección del Instruction Pointer, la dirección física donde estará mapeada la tarea y la direccion donde comienza el código de la misma.

Esta función se encarga de completar el TSS con los datos correspondientes. Para los selectores de segmento se usan los de datos y código de nivel 3 ya que este es el nivel en que correrán las tareas del juego. Para el campo cr3, que contiene el descriptor del DPT de la tarea se llama a la función "mmu_-init_task_dir" que hace los mapeos correspondientes a la tarea que estemos inicializando y devuelve la dirección física del directorio de páginas que utilizó para el mapeo. Para el stack de nivel 0 se pide una página del area libre del kernel y se lo apunta al final de la misma. Para el selector de segmento de nivel 0 se usa el de datos de nivel 0.

2.7. Estructuras del Juego

Para el desarrollo del juego definimos, en el archivo "player.h" una estructura principal "player_info" la cual contiene, para cada jugador (Rick o Morty): Su puntaje, el número de tareas Mr. Meeseeks que tiene actualmente vivas, el número de su tarea actual (10 = Si mismo, 0-9: Nro Meeseek) y un vector de instancias de la estructura "Meeseek", correspondientes a sus tareas Mr.Meeseek.

La estructura "Meeseek" contiene la información relevante para las tareas Mr. Meeseek, esto es: Su posición en el mapa, un bit de presente, un bit que indica si usó la portal gun y su edad (número de clocks que se le asignaron desde que fué creada).

Para las posiciones de los meeseeks y de las megasemillas usamos la estructura "map_pos" que tiene sus coordenadas en los ejes x e y.

La estructura "seed_info" contiene las coordenadas de las (40) megasemillas y un bit de presente.

2.8. Ejercicio 7

Usamos la función "sched_init" que se encarga de inicializar las estructuras que usaremos para implementar la lógica del scheduler. Estas estructuras constan de: Una variable que indica a qué jugador corresponde la tarea actual (0 = Rick, 1 = Morty) y de las estructuras descriptas en la sección 2.7.

La función "sched_next_task" se encarga de cambiar el jugador actual, actualizar las estructuras para que hagan referencia a la próxima tarea a ejecutarse y obtener el índice en la gdt de la misma. Para que se cambie de tarea por cada ciclo de reloj, modificamos la rutina de interrupción del reloj de forma que llame a la función "sched_next_task" para actualizar las estructuras, y luego realice el cambio de contexto tal que se pase a ejecutar la próxima tarea según corresponda.

Modificamos las rutinas de las interrupciones 88, 89, 100 y 123 de forma tal que efectúen un cambio de contexto mediante un "JMP FAR TSS_IDLE_SEL:0", donde TSS_IDLE_SEL es el selector de segmento correspondiente a la TSS de la tarea IDLE.

Para desalojar las tareas que generaron una excepción modificamos la macro que se encarga de manejar las excepciones. En la misma llamamos a la función "sched_dealloc_current_task" (definida en sched.c) que se encargará de realizar todo el proceso de desalojo:

quitar la tarea del vector de tareas vivas del jugador correspondiente, desmapear las páginas de dicha tarea, quitar la tarea de la pantalla del juego, remover el flag presente de la entrada de la GDT correspondiente a la TSS de la tarea desalojada y, en caso de que la excepción haya sido generada por la tarea Rick o Morty, terminar el juego dando por ganador al jugador contrario.

Para implementar el mecanismo de Debugging agregamos en la macro que se encarga de manejar las excepciones del procesador una subrutina qué, en caso de qué estuvieramos corriendo en modo Debug al recibir la excepción, se encargará de guardar el contexto actual de ejecución en una estructura que definimos en "debug.h" para almacenar el contenido de los registros pertinentes. Luego, esta subrutina llama a la función "debug_excepction" que cambia el estado actual de DEBUG a EXCEPCION, guarda el contenido de la pantalla del juego e imprime en pantalla el contexto de ejecución al momento de recibir la excepción.

En el estado EXCEPCION el sistema queda a la espera de recibir una interrupción de teclado correspondiente a la tecla "y", momento en el cual volveremos la pantalla al estado en que se encontraba antes de recibir la excepción y reanudaremos la ejecución el juego.

2.9. Ejercicio 8

Las funciones relacionadas al desarrollo del juego, y por lo tanto mencionadas en este apartado, se encuentran definidas en el archivo "game.c", con funciones auxiliares en "player.c". Además, las estructuras que definimos para guardar información relevante al juego están en el archivo "game.h".

Para distribuir semillas en el mapa armamos una función propia "init_game_screen" que inicializa la estructura de las semillas, cada posición del vector "seed_info" representará una megasemilla, la función se encargará de asignarle una posicion random en el mapa a cada una (chequeando que no se superpongan) y setear su bit de presente en 1. Luego las imprime en pantalla.

Syscall Meeseeks: La rutina de interrupción de meeseeks llama a la función "init_meeseek", la cual se encarga primero de verificar que no haya ya 10 meeseeks activos para el jugador actual. Luego chequea si en la posición pasada por parámetro había una megasemilla, en cuyo caso no se crea el meeseek si no que directamente se le suman los puntos al jugador correspondiente y se borra la megasemilla del mapa.

En caso de que no hubiera una megasemilla en dicha posición, se inicializa la TSS del meeseeks a crear, se copia el codigo de la tarea en el area de mapa dada por los parámetros y se lo imprime en pantalla.

Syscall Move: La rutina de interrupción de move (123) llama a la función "move_meeseek" la cual sigue el siguiente algoritmo: Primero verifica si la tarea que llamo a la interrupción "move" fué Rick o Morty, de ser así la desaloja llamando a la función del scheduller "sched_dealloc_current_task", dando por finalizado el juego.

Luego se fija si en la posición final del movimiento hay una megasemilla, en tal caso se la asimila y se desaloja la tarea que llamó a la interrupción. Si luego de asimilarla no quedan semillas se finaliza el juego.

En caso de que no hubiera megasemilla en la posición final, se mueve al meeseek a la misma. Para esto se llama a la función "move_mee_code" que copiará el código del meeseek desde la posición actual hasta la nueva posición, mapeando y desmapeando las paginas correspondientes. Finalmente se actualiza la pantalla borrando al meeseek de la posición vieja e imprimiendolo en la nueva.

Syscall Look: La rutina de interrupción de look (100) reserva espacio en la pila antes de pushear el contexto actual para guardar los valores de retorno de la función "look". Esta función recibe como parámetro la posición de inicio del espacio que reservamos y guarda allí los valores del desplazamiento relativo en x e y entre la posicion del meeseek actual y su megasemilla más cercana.

En caso de que este servicio sea llamado por una tarea Rick o Morty, devuelve -1 en ambas coordenadas.

Para buscar la semilla más próxima, primero obtenemos la posición del meeseek actual. Luego, recorremos el vector de semillas calculando, para cada semilla presente, la distancia manhattan con nuestro meeseek, guardando la semilla más cercana que encontremos (sin considerar el mapa circular).

finalmente calculamos la diferencia entre la posicion de la semilla encontrada y nuestro meeseek y guardamos las coordenadas en la pila.

Syscall Use_Portal_gun: La rutina de interrupcion de este servicio (89) simplemente se encarga de llamar

a la función "teleport_meeseek" la cual primero verifica si el servicio fue llamado por un jugador o por un meeseek que ya usó la portal gun, en tal caso retorna sin hacer nada.

En otro caso, chequea si el jugador enemigo tiene algun meeseek vivo, en tal caso elige uno al azar y llama a la funcion "move_meseek" pasando como parámetros dos desplazamientos aleatorios en x e y, junto con el id del meeseek a mover.

finalmente actualiza la información del meeseek que llamó al servicio para que indique que ya usó la portal gun.