DEPARTAMENTO DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico 3: Jauría

Organización del Computador II

Grupo: Fuga Villera Nro. 2

Integrante	LU	Correo electrónico
Gabriel Matles	397/12	gabriel29m@gmail.com
Manuel Mena	313/14	manuelmena1993@gmail.com
Francisco Demartino	348/14	demartino.francisco@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

1. Ejercicios

1.1. Ejercicio 1 - Pasaje a modo protegido y segmentación

Lo primero que hicimos fue habilitar la línea A20. La línea A20 permite acceder a los primeros 16MB de memoria en lugar de acceder solo al primer megabyte (esto aparece en el Intel 286). Si bien esto no está en la consigna, es necesario para acceder a toda la memoria del kernel. La línea A20 viene deshabilitada por defecto para mantener soporte con el 8086. Mas info: $http://wiki.osdev.org/A20\_Line$, $http://www.independent-software.com/writing-your-own-toy-operating-system-enabling-the-a20-line/$, $https://en.wikipedia.org/wiki/A20\_line$

Antes de pasar a modo protegido, estamos en un modo compatible con 8086 de 16bits y pocas instrucciones. Para aprovechar al máximo el procesador y la arquitectura, pasamos a modo protegido. Para pasar a modo protegido hay que hacer algunas cosas:

1. Armar la GDT en memoria
2. Cargarla en el GDTR
3. Levantar el bit PE (Protected Mode Enable) de CR0
4. Hacer un salto largo a código de 32bits
5. Listo!

Para armar la GDT, dejamos en blanco (0, nula) los 8 primeros descriptores por lo que pide el enunciado. Después, completamos 4 descriptores para usar segmentación flat en los primeros 500MB de memoria, seteando los bits según si eran paginas de código/datos y usuario/kernel. Adicionalmente, pusimos un descriptor del segmento de memoria de video para el ejercicio c.

Una vez completa la GDT, se carga con la instrucción LGDT [GDT_DESC]. Luego de levantar el bit PE, hacemos un salto largo especificando el segmento de código de kernel. Este salto largo le permite al procesador limpiar el pipeline previo al cambio a modo protegido y empezar a trabajar con instrucciones de 32 bits. Hay que tener cuidado con decirle al ensamblador que use la codificación correspondiente a 32 bits con BITS 32. Apenas saltamos a este código, seteamos en todos los registros de segmento (DS, ES, FS, GS y SS) el segmento de código del kernel, asignamos la base de la pila a 0x27000 y se imprime el mensaje de bienvenida y el resto de la interfaz usando las funciones ya programadas en screen.h.

1.2. Ejercicio 2 - Interrupciones básicas

Aprovechamos la macro IDT_ENTRY para completar las entradas de la IDT, en lugar de tener que hacerlo a mano como para la GDT. Luego, con la macro ISR registramos los handlers de interrupciones para las interrupciones de la 0 a la 19. Similarmente a la GDT, usamos la instrucción LIDT [IDT_DESC], para cargar el registro IDTR.

La rutina de excepción que hicimos muestra un mensaje de error genérico en pantalla y cuelga la ejecución con un loop infinito. Para probarla, hicimos una función que usa el segmento de video y intenta escribir fuera de él. Con esto checkeamos que la excepción cae a nuestro handler y vimos el mensaje en la pantalla.

1.3. Ejercicio 3 - Paginación básica

Lo primero que hacemos es escribir la función mmu_mapear_pagina que se encarga de mapear una dirección de memoria virtual con una física. Los parametros que recibe son una dirección de memoria virtual, la dirección de memoria del directorio de paginas, una dirección física y los atributos de la página.

De la dirección de memoria virtual obtenemos los índices del directorio de páginas y de la tabla de páginas. De esta forma localizamos la entrada de la tabla de páginas correspondiente y escribimos la dirección física en los 20 bits más significativos. En los 12 bits restantes se escriben los atributos de

la página.

Luego, vamos a crear un directorio de páginas que mapee, usando identity mapping, las direcciones virtuales del rango 0x00000000-0x003FFFFF a las direcciones físicas en el 0x00000000-0x003FFFFF, o sea, el mismo rango.

El rango 0x00000000 a 0x003FFFFF consiste de 1024 páginas de 4KB cada una, por lo cual necesitamos una tabla con todas sus 1024 entradas configuradas a una página cada una, mapeando la misma direccion de memoria virtual a física. Entonces, la tabla va a ocupar exactamente 1 página de 4KB, ya que son 1024 entradas de 4bytes, y esta tabla va a ser apuntada por la primera entrada del directorio de paginas del kernel (y luego veremos que también va a estar en la primera entrada de los directorios de todas las tareas para manejar interrupciones correctamente).

Concretamente, primero pedimos una pagina para ubicar ahí el directorio, y luego dentro de ciclo llamamos a mmu_mapear_pagina, usando la misma dirección virtual que física para configurar la identidad. Esto va a crear las tablas de ser necesario y hacer el mapeo. Para asegurarnos que el identity mapping fue efectivo utilizamos el comando info tabs de bochs, que muestra la traducción de todas las páginas de memoria.

Para activar paginación ponemos el bit más significativo de cr0 en 1. Es necesario haber escrito previamente en cr3 la dirección de memoria del directorio de páginas.

Al hacer cualquier cambio relacionado con paginación, ya sea cambiar el directorio, cambiar una tabla, o cualquier mapeo, necesitamos refrescar la TLB, que es como un caché de paginación para evitar la traducción Virtual -¿Directorio -¿Tabla -¿Página y saltar directamente de Virtual a Página. Para refrescar la TLB el TP nos provee una función llamada tlbflush() que llamamos al final de las funciones mmu_mapear_pagina y mmu_unmapear_pagina.

Para desmapear una página, alcanza con poner el bit de Present en 0 en su entrada de la tabla correspondiente. Luego, como mencionamos anteriormente, llamamos a tlbflush().

Probamos mmu_unmapear_pagina desmapeando la última página del kernel (0x3FF000).

1.4. Ejercicio 4 - Paginación dinámica

El primer paso es inicializar el administrador de memoria del área libre. Usamos un contador que comienza en 0x100000 para saber cual es la dirección de la próxima página libre, e implementar la función mmu_proxima_pagina_fisica_libre, que devuelve el valor del contador y lo incrementa en 4k.

Acá asumimos que vamos a tener memoria suficiente para todas las paginas necesarias y no vamos a hacer ningun intento de reutilizar páginas que ya se desmapearon.

Entonces, las estructuras globales necesarias pasan a ser simplemente punteros para guardar las direcciones a páginas compartidas entre todos los perros del mismo jugador, e inicializarlas es pedir dos paginas consecutivas.

La rutina mmu_inicializar_memoria_perro es la encargada de inicializar un directorio de páginas y tablas de páginas para una tarea. La función copia el código de la tarea a su área asignada, es decir la posición indicada por el jugador dentro de el mapa y configura su directorio.

Para esto llamamos a la función mmu_proxima_pagina_fisica_libre la cual devuelve una página de memoria que utilizaremos como directorio del perro. Luego llamamos a la función mmu_inicializar_pagina con la página que acabamos de pedir para poner todos sus bits en 0.

Con el jugador y el tipo de perro, que nos pasan como parametros, obtenemos la direccion fisica de la pagina en la cual se encuentra el código del perro.

Luego mapeamos dicha página a la posición actual del perro en el mapa del juego, desde la base

virtual a su base física correspondiente en el directorio del kernel y luego copiarla para que el perro "viva" en esa posición del mapa. Además hacemos identity mapping de la memoria del kernel en el directorio del perro para que el kernel pueda, por ejemplo, atender una interrupción producida mientras que se está ejecutando la tarea perro. Ahora mapeamos las paginas de memoria donde se encuentra el codigo/memoria del perro y la memoria compartida entre todos los perros del jugador.

Mapeamos la página del mapa de juego (de la posición inicial del perro) en el directorio del perro. Completamos la pila al final de la página con la posición actual del perro asi sabe donde está al inicializarse. Finalmente, llamamos a mmu_inicializar_memoria_perro pasandole un puntero a una estructura perro previamente inicializada, y cargamos en CR3 la direccion al directorio del perro que nos devuelve la funcion. Una vez hecho esto chequeamos que podemos escribir en memoria de video usando el directorio del perro, y chequeamos en info tab que los rangos esperados se hayan mapeado correctamente.

1.5. Ejercicio 5 - Interrupciones de reloj, teclado y software

Definimos las entradas en la IDT que usaremos para asociarles las rutinas de interrupción de reloj, teclado y software 0x46. Las interrupciones de la 0 a la 31 están reservadas para el procesador y, en particular, de la 8 a la 15 ya están ocupadas por las excepciones del mismo, por lo que hay que remapearlas. Para esto usamos la funcion resetear_pic, que mapean la interrupciones externas a partir de la direccion 32, por lo que queda la del reloj mapeada a la interrupción 32 y la del teclado a la 33. Después, con habilitar_pic, habilitamos el handler. Estas dos funciones están definidas en pic.h. Luego habilitamos las interrupciones con la instrucción sti.

Ahora vamos a escribir las rutinas de interrupción. Para cada una vamos a salvar los registros y avisarle al pic que tomamos la interrupción mediante la función fin_intr_pic1 de pic.h, hacemos lo minimo indispensable en assembler y saltamos a un handler hecho en C.

Escribimos la rutina asociada a la interrupción del reloj. Esta se encarga de mostrar la animación de un cursor rotando. Para ello utilizamos la función screen_actualizar_reloj_global de screen.h. Más adelante vamos a llamar al scheduler, y éste va a hacer un par de cosas antes de actualizar los relojes (por ejemplo, chequear a que tarea saltar luego del tick). Escribimos la rutina asociada a la interrupción del teclado, cuya función es, por ahora, imprimir la tecla presionada en la esquina superior derecha de la pantalla. Para esto obtenemos un scan code a través del puerto 0x60, y usamos la función print_hex de screen.h para mostrar la tecla.

Por último escribimos la rutina asociada a la interrupción de software 0x46, que lo único que hace es escribir 0x42 en el registro eax. Podemos verificar poniendo breakpoints con bochs que el valor efectivamente se carga.

1.6. Ejercicio 6 - Tareas

Definimos las entradas necesarias en la GDT para usar como descriptores de TSS. Una para la tarea inicial y otra para la tarea Idle. Completamos el descriptor de la tarea Idle con la siguiente información:

1. Segmento de código definido en la GDT
2. Segmento de datos definido en la GDT
3. La dirección física de la base de la pila del kernel como stack pointer, que será mapeada con identity mapping
4. La dirección 0x00016000 como instruction pointer, que es donde se encuentra la tarea
5. Mismo cr3 que el kernel
6. Mismo segmento de datos como stack segment de nivel 0
7. Misma dirección como stack pointer de nivel 0

Hicimos una función que completa un descriptor de TSS libre con la información correspondiente a una área:

1. Segmento de código definido en la GDT
2. Segmento de datos definido en la GDT
3. Base del espacio de tarea como pila (teniendo en cuenta que se habran apilado sus 2 argumentos y una dirección de retorno)
4. Dirección de inicio de código de tareas
5. Como cr3 usaremos la dirección que nos devuelve mmu_inicializar_memoria_perro.
6. Mismo segmento de datos como stack segment de nivel 0
7. Se define una nueva página libre y usamos la base como stack pointer de nivel 0

Completamos las entradas de la GDT correspondientes a la tarea inicial y a la tarea Idle llamando a la funcion anteriormente descripta. Cargamos la tarea inicial moviendo el índice de la entrada de la GDT, correspondiente a la TSS de la tarea inicial, al task register. Luego hacemos el salto a la tarea Idle con un jmp far.

La interrupción 0x46 guarda los parametros en la pila y llama a game_syscall_manejar que se encarga de manejar que tipo de syscall se hizo. Luego llama a la rutina que atiende la syscall pedida y devuelve el resultado. Las rutinas game_perro_mover, game_perro_cavar y game_perro_olfatear son las encargadas de resolver las distintas tipos de syscall.

Creamos una tarea perro en la TSS y la entrada en la GDT para dicha tarea. Luego saltamos al comienzo de nuestra tarea. Como por ahora no teniamos implementadas las syscalls ni nada de eso, no pudimos ver demasiado, pero festejamos el hecho de que no hubiera page faults ni general protection faults!

1.7. Ejercicio 7 - Scheduler

La función sched_inicializar inicializa las estructuras del scheduler con NULL y otros valores iniciales.

La funcion sched_proxima_a_ejecutar busca una tarea del jugador opuesto al que está ejecutándose actualmente. En caso de encontrarla devuelve el índice de la tarea. En caso de no poder encontrarla, busca una tarea del mismo jugador que actualmente tiene una tarea en ejecución. En caso de no encontrar ninguna tarea devuelve el indice de la tarea IDLE.

La función sched_atender_tick llama a game_atender_tick pasandole el puntero al perro de la tarea actual. game_atender_tick se encarga llamar a screen_actualizar_reloj_perro, game_perro_ver_si_en_cucha en caso de que el perro pasado sea distinto de Null y a game_terminar_si_es_hora.

Ademas reemplazamos el llamado a game_atender_tick por uno a sched_atender_tick en el handler de la interrupcion de reloj. El handler de interrupcion de reloj, luego de llamar a sched_atender_tick, recibira el valor que retorne sched_proxima_a_ejecutar. Ese valor es un selector de segmento de código correpsondiente a la siguiente tarea a ejecutarse. Cargando oportunamente este valor en memoria y haciendo un jmp far tenemos task switching! Se modifican las rutinas de atención de las excepciones del procesador para desalojar la tarea actual llamando a la función call sched_desalojame_esta y luego se pasa a ejecutar la tarea IDLE haciendo un jmp GDT_SELECTOR_TSS_IDLE:0, en lugar de mostrar un mensaje en pantalla y colgar la ejecucion.

Finalmente, como mecanismo de debugging, se implementó la ventanita de debugging: Si el modo debug esta activado, en lugar de desalojar directamente la tarea al momento de producirse la excepcion, se muestra en pantalla un “snapshot” del estado del procesador. En el momento en que se desactiva el debug, se procede con matar la tarea y pisarla con la IDLE como en el caso anterior.