1. Introducción

En este trabajo nos ocupamos de hacer un pseudokernel que se encargara de efectuar ciertas funciones básicas para el correcto booteo de un procesador. Dado que esto sería muy riesgoso de hacer en una PC propiamente dicha, usamos el emulador bochs para correrlo. El kernel se encarga de pasar a modo protegido, puesto que cuando el procesador se inicia lo hace en modo real, inicializa una GDT, crea una IDT que permita mostrar por pantalla las excepciones que se generen, habilita paginación, toma interrupciones del teclado y del clock, y finalmente crea tareas y permite simular un scheduler que alterna entre ellas.

Para hacer todo esto nos basamos en los talleres dados en clase que nos permitieron efectuar los diferentes cambios al kernel de una manera gradual; esto se ve reflejado en la manera de encarar los ejercicios, los cuales aumentan en complejidad. Es importante remarcar que algunos ejercicios son bastante similares a lo visto en clase y por tanto su solución es muy similar a la de aquellos.

Este trabajo fue corrido con los elementos provistos por la materia, usando el emulador Bochs para correrlo. Para probar este TP, uno debe situarse en donde se encuentren los archivos del diskette y de configuración del bochs y tipear $bochs - q$

2. Ejercicios

2.1. Ejercicio 1

En este ejercicio debemos pasar a modo protegido. Como hacer esto así nomás causaría un error, primero se deben crear entradas en la GDT, por lo menos dos (una para código y una para datos) de manera que los segmentos puedan tomar algun valor válido. Una vez creadas estas entradas, las cuales son de 4GB de tamaño y se solapan, también creamos un segmento de video al que asignamos la dirección 0xB8000 para poder escribir en pantalla fácilmente.

Una vez hecho esto podemos cambiar a modo protegido, esto lo hacemos colocando un 1 en el último bit de cr0. Ya en modo protegido, podemos asignar los segmentos a los valores correspondientes que creamos en la GDT, e imprimimos por pantalla un mensaje que confirme que todo está andado como se supone

2.2. Ejercicio 2

Este ejercicio consiste en crear una IDT que pueda mostrar por pantalla cuando se produce una excepción. si bien no podíamos atender de manera correcta dichas excepciones, no nos faltaron instancias en las cuales dichos mensajes fueran exhibidos.

Para esto debimos definir rutinas para las catorce excepciones, que simplemente se limitaban a imprimir por pantalla una leyenda que especifique qué tipo de excepción se produjo, luego lo único que fue necesario fue cargar dicha IDT. Para probar que ésta andaba correctamente contamos con varios métodos, aún cuando no todos las excepciones son reproducibles, la más fácil de observar es la que se produce al dividir por cero (se encuentra comentada en el código, de otra manera el kernel no andaría)

2.3. Ejercicio 3

En este ejercicio debemos activar paginación. Para esto, debemos crear primero el árbol de páginas del kernel con identity mapping sobre las primeras 0x4000000 direcciones, colocando el directorio y las tablas donde es pedido. Luego, podemos habilitar el bit de paginación, esto es, el primer bit de cr0, no sin antes haber colocado el directorio en cr3 para poder usar la paginación. Una vez hecho esto, podemos probar que la misma funciona, para ello cargamos un mensaje de nuestra película, en este caso tomado de la brillante cancion Çomfortably Numbτ lo imprimimos en pantalla.

2.4. Ejercicio 4

Para este ejercicio debemos profundizar la paginación, esta vez permitiendo que las páginas sean mapeadas no sólo con identity mapping sino de cualquier manera. Para esto, primero necesitamos crear diversas funciones que nos van a ayudar a paginar más adelante en el trabajo, ateniéndonos a las consignas pedidas en las mismas.

Para las funciones que devuelven una página libre, tanto de usuario como de kernel, consideramos que era mejor mantener dos variables globales que fuera inicializadas en las direcciones que queremos, y de tal manera cuando pedimos una página nueva tomamos esa variable y luego la incrementamos el tamaño de una página.

La función que inicializa un directorio de usuario se encarga de crear un nuevo árbol de páginas y devolver el nuevo cr3, pero en la consigna nos es pedido que se mapee con identity mappingo solo las primeras 0x200000 direcciones, por lo tanto, debemos asegurar que el resto está vacío, esto lo hacemos colocando un 0 en esas entradas.

En la función mapear página, dado un cr3 y una dirección virtual y una física se encarga de mapear la física a la virtual, esto lo logramos haciendo que las entradas en directorio y tabla de página que corresponden a dicha dirección virtual apunten a la física pasada por parámetro. En caso de que lleguemos a encontrar que la tabla de páginas que nos corresponde no está inicializada, para evitar un page fault debemos inicializarla, esto lo hacemos colocando en ceros la misma; una vez hecho esto se puede pasar a actualizar los punteros y a colocar los bits de presente en las entradas que necesitamos. Exactamente lo inverso hacemos en la función que unmapea la página.

Para testear que esto anda, inicializamos el directorio de usuario en kernel.asm, y luego mapeamos una dirección cualquiera a la de la memoria de video. Hecho esto, tomamos el nuevo cr3 que se creó al inicializar el directorio y escribimos en la posición virtual, lo que dado el mapeo nos lleva a escribir en la memoria de video. A modo de testeo, cambiamos el color de la letra W que es la primera impresa en nuestro mensaje, como esto funciona podemos unmapear y concluir esta parte.

2.5. Ejercicio 5

Aquí debimos crear varias rutinas de atención de interrupciones, en primer lugar las que atienden al clock y al teclado, para la primera contábamos ya con la rutina provista, lo único que debimos hacer fue asegurar que cuando se producía la misma ésta era llamada; así con cada tick se produce una interrupción y el reloj imprime en pantalla un pseudomovimiento para demostrar que está en funcionamiento. Para el teclado, debimos definir la rutina 33, que lee la tecla presionada. Si es un break, no hace nada, luego se fija si el scancode está en el rango que nos interesa (o sea, si es un número de 0 a 9). En caso de que lo sea, procedemos a calcular su ASCII e imprimirlo en pantalla (como esta rutina se ve alterada por lo pedido en el ejercicio 7, la respuesta a la entrada de números de 1 a 8 no es la que pide este ejercicio).

Las otras dos interrupciones pedidas simplemente nos requerían devolver un número en eax, por lo que no causaron mayores inconvenientes.

2.6. Ejercicio 6

En este ejercicio debíamos crear las condiciones necesarias para agregar una nueva tarea (IDLE) y luego saltar a ella. Si solamente hubiésemos creado las entradas necesarias para IDLE, esto rompería ya que el procesador no sabría qué hacer al guardar la tarea actual, es por esto que debemos crear dos entradas en la GDT, una que guarde la información de la tarea actual y otra que guarde la de la IDLE. Para la tarea actual sólo creamos las condiciones necesarias para que el procesador guarde la información, no está pensado para que luego se pueda saltar a la misma, a diferencia de la IDLE.

Para hacer esto nos colocamos tanto en una entrada libre de la GDT como del arreglo de TSS, y colocamos la información necesaria, sabiendo que necesitamos pedir una página libre nueva para que haga las veces de pila de la tarea IDLE, la información la completamos secuencialmente con los datos exigidos por la consigna.

Una vez hecho esto, cargamos en el task register la tarea actual, y saltamos al selector de la IDLE con un offset cualquiera, el procesador interpreta que como es una entrada de tarea, debe ejecutarla.

2.7. Ejercicio 7

Para este ejercicio se nos pedía crear una nueva tarea asignada a una interrupción de teclado (o sea, que se active cuando se presiona una tecla) y además crear un scheduler que alterne entre las tareas que se encuentran en un arreglo de tareas que contiene las direcciones de la GDT a las cuales saltar. Para lograr esto, primero tuvimos que alterar la interrupción de teclado, ya que la rutina a ejecutar al presionar de 1 a 8 es diferente, sabiendo que los scancodes son consecutivos, podemos obtener la posición de memoria en la cual comienza el código que debe ejecutar la nueva tarea simplemente restando 2 (o sea, obteniendo el número verdadero) y shifteando 0x1000 (tamaño de una página), sumando ese resultado a una base de 0x13000 que es donde empiezan los códigos. Una vez encontrado el código de inicio, solo basta llamar a la función crearproceso con dicho dato como parámetro.

Esta función necesita crear una nueva entrada en la GDT para la tarea, una nueva TSS y además asegurarse que esta tarea tiene todos los recursos que necesita, para esto primero debimos crear un nuevo directorio de páginas de usuario, y crear páginas libres para la pila y el código, luego mapeamos estos mismos y copiamos el código a la posición pedida (o sea, 0 en dirección lineal). Una vez hecho esto, podemos setear los datos en la GDT y en una nueva TSS, esto lo hacemos en assembler de manera similar a como lo hicimos para IDLE.

Luego nos toca crear un scheduler que se encargue de alternar entre las tareas que se encuentran en el arreglo de tareas (esto es importante de remarcar, puesto que tanto la tarea inicial y la tarea IDLE no se encuentran en el mismo sino que están en sus TSS propios). Para hacer esto contamos con la función obtenerpid, que devuelve la posición en el arreglo de tareas de la actual, para esto lee el task register y busca en todo el arreglo un valor igual a éste que indica el segmento en la GDT. También tenemos la función proximoindice que devuelve el índice de la GDT de la

próxima tarea a ejecutar, pero además lo usamos para saber si es necesario saltar, en los casos límites en los que no hay que saltar esta función devuelve 0, y el scheduler compara el resultado con 0 y termina la interrupción si obtiene este número.

Nos dimos cuenta debuggeando que la interrupción 88 era llamada dentro del código de las tareas que creamos, y, dado que se nos pidió que asignáramos el resultado de esta interrupición al resultado de obtenerpid, concluimos que esto lo usaba para determinar en qué porción de la memoria de video iba a escribir, esto explica por qué presionando números se obtienen cambios en la imagen que son secuenciales.

Es importante aclarar que, a pesar de haber notado esto, y de haber consultado tanto con Cristian como con Jorge sobre las posibles causas, no conseguimos determinar por qué luego de cada interrupción, lo impreso en la memoria de video cambia el color del fondo, pero el ASCII que debería tener siempre se encuentra en 0, por lo tanto nunca aparece ningún texto. Esto solo lo notamos luego de cambiar el fondo negro por uno verde, el cual dejamos de esta manera para que el efecto causado por las tareas se note. Sin embargo, sabemos que se accede de manera correcta al código provisto por la cátedra, y que el scheduler funciona correctamente y como tal alterna una y otra vez entre las tareas del arreglo.

3. Conclusiones

A lo largo de este TP conseguimos entender mejor qué sucede cuando se inicia un procesador, pudimos realizar sin mayores problemas el salto a modo protegido y dentro del mismo conseguimos llevar a cabo ciertos procedimientos básicos que, a mucha mayor escala, son los aplicados por todos los kernels al bootear; esto obviamente es de suma importancia, porque ahora podemos entender mejor ciertos procesos de un kernel, además entender cuáles son las causas de las excepciones que se producen en un procesador, como por ejemplo saber por qué se produce un segmentation fault al codear.