TP 3: Orga 2 - Sin Pelotas

Integrantes

Integrante	Libreta Universitaria	Email
Ignacio Alonso Rehor	195/18	arehor.ignacio@gmail.com
Manuel Panichelli	72/18	panicmanu@gmail.com
Vladimir Pomsztein	364/18	blastervla@gmail.com

Ejercicio 1

gdt.c

- Llenamos las entradas de la GDT. Creamos 2 entradas para segmentos de Código (lvl 0 y 3) y 2 para segmentos de Datos (lvl 0 y 3).
- Para ello, configuramos los bits correspondientes en cada entrada:
  - Base del segmento (en 0, queremos los primeros 163 MB de memoria)
  - Límite del segmento (direccionamos los primeros 163 MB de memoria)
  - Bit S (en 1, no son descriptores de sistema)
  - Bits de tipo (Código execute only y datos read/write)
  - Bits de DPL (0 y 3, como ya dijimos)
  - Bit D/B en 1 (32 bits)
  - Bit L en 0 (estamos en 32 bits)
  - Bit G en 1 (podríamos no haberlo utilizado, pero queríamos probarlo)
  - Bit Presente prendido
- Agregamos un segmento para la pantalla de video:
  - Base en 0xB8000
  - Límite de 7999 bytes
  - Bit S en 1
  - Tipo datos R/W
  - DPL = 0
  - Bit D/B en 1
  - Bit L en 0
  - Bit G en 0 (esta vez nos portamos bien)
  - Bit Presente prendido

kernel.asm

- Escribimos el código para pasar a modo protegido y configurar la pila del Kernel en 0x2B000.
- Escribimos unas rutinas `limpiar_pantalla` y `draw_screen` que se encargan de limpiar la pantalla y dibujarla, respectivamente.

print.mac

- Hicimos una copia de la macro para imprimir en la pantalla en modo real (provista por la cátedra) y la modificamos para habilitarla a imprimir en modo protegido, utilizando el segmento de video creado previamente.

Ejercicios 2 y 3

idt.c

- Agregamos un par de defines para los atributos de las entradas de la IDT de Kernel y de Tarea (un define para cada una).
- Creamos una macro `IDT_ENTRY` para ayudarnos a armar la IDT. Esta macro toma dos parámetros, `numero` y `attrVal`, y configura la entrada de número `numero` de manera adecuada, utilizando los atributos `attrVal` para la entrada.
- En `idt_init` configuramos las entradas de las excepciones implementadas por Intel y de las interrupciones de Clock, Teclado y Syscall (32, 33 y 47, respectivamente). Configuramos todas como de nivel Kernel, salvo la Syscall (pues su propósito es poder ser utilizada por la tarea).

isr.asm

- Implementamos la subrutina `_isr%1` para que imprima el número de excepción que la llamó por pantalla.
- Utilizamos la macro `ISR 1` para implementar las rutinas de atención de todas las excepciones implementadas por Intel.
- Escribimos la rutina de atención del Clock para que printee la animación de reloj provista por la cátedra en la esquina inferior derecha de la pantalla.
- Implementamos la rutina de atención del Teclado para que, en caso de que el "make code" de la tecla presionada corresponda a algún dígito decimal, imprima en pantalla el dígito presiondo en la esquina superior derecha.
- Escribimos la rutina de atención de la Syscall para que reemplaze el valor de eax por el número 0x42.

isr.h

- Declaramos en isr.h todas las rutinas de atención de interrupción / ejecución.

kernel.asm

- Llamamos a `idt_init` para inicializar la IDT y luego cargamos el IDTR con la instrucción `lidt`.
- Resetteamos el PIC y lo habilitamos.
- Habilitamos interrupciones.
- Probamos que las excepciones se impriman en pantalla dividiendo por cero.

Ejercicio 4

Este ejercicio trata de la MMU.

defines.h

- Agregamos defines de utilidad para sumar descriptividad a los descriptores (valga la redundancia, y para que Ezequiel no nos pegue)

kernel.asm

- Llamamos a `mmu_initKernelDir`
- Habilitamos paginación (poniendo el bit 31) de CR0 en 1.
- Agregamos una subrutina `print_group` que se encarga de imprimir en pantalla las libretas universitarias de los integrantes del equipo.

mmu.h

- Agregamos las definiciones de los structs para las PDE's y PTE's

mmu.c

- Completamos `mmu_initKernelDir`, que se encarga de:
  - Inicializar el Page Directory del Kernel llenándolo de ceros.
  - Escribir la primera entrada del Directory, configurándola para acceso de supervisor.
  - Configuramos todas las entradas de la primera (y al menos por ahora única) tabla del Kernel, de manera tal que tengamos identity mapping para las direcciones 0x00000000 a 0x003FFFFF.

Ejercicio 5

Este ejercicio trata de la MMU también.

defines.h

- Continuamos agregando defines de utilidad para sumar descriptividad a los descriptores

kernel.asm

- Inicializamos la MMU llamando a `mmu_init`.
- A modo de prueba, llamamos a `init_taskDir` para configurar paginación para el jugador A tipo 1, y pisamos `CR3` con la dirección al directorio correspondiente.

mmu.h

- Agregamos defines para utilizar como tipo enumerado para los niveles de privilegios.
- Incluimos defines para los tipos de jugadores, cuyo valor será utilizado como índice para obtener la posición en la cual comienza el código para dicha tarea.
- Reemplazamos el parámetro `attrs` de la función `mmu_mapPage` por el nivel de privilegio de la tarea.

mmu.c

Definimos variables globales para saber cuál es la siguiente página libre para el kernel y la tarea.

Además, creamos la estructura auxiliar `address`, que usaremos para parsear las direcciones virtuales y obtener sus componentes. Implementamos las funciones:

- `mmu_init`: Inicializa las variables globales descriptas previamente
- `mmu_nextFreeKernelPage` y `mmu_nextFreeTaskPage`: Incrementan los punteros definidos como variables globales (cada uno tiene uno que utiliza exclusivamente) y devuelven la siguiente página libre.
- `mmu_mapPage`:
  - Descomponemos la dirección virtual haciendo uso de la estructura auxiliar mencionada.
  - Luego utilizamos el cr3 (completo, pues los campos bajos no se utilizan) y el índice obtenido de la dirección lineal para obtener la PDE (Page Directory Entry).
  - Si la tabla referenciada en la PDE no existe, la creamos con el privilegio correspondiente y como Read/Write.
  - Finalmente, obtenemos la PTE referenciada, la ponemos presente y le configuramos la base shifteando la dirección física provista, y la configuramos con el privilegio adecuado y como Read/Write.
  - Flusheamos (invalida la TLB)
- `mmu_createTable`: Se encarga de crear una tabla nueva desde 0, llenando todas las PTEs de la misma y configurándolas como "No presentes".
- `mmu_unmapPage`: Si el entry del directorio no está presente, no hace nada. Si lo está, simplemente obtiene la PTE adecuada y la configura como "No presente".
- `mmu_initTaskDir`:
  - Pide una página nueva del kernel para almacenar el Page Directory de la tarea. Configura cada PDE con 0s.

- Hace identity mapping con los primeros 4K de la memoria, para poder tener referenciado el Kernel. Esto es para poder resolver las excepciones que surjan durante la ejecución de la tarea y poder accionar desde el Kernel.
- Obtenemos la dirección física del código de la tarea pedida, y se piden dos nuevas páginas, puesto que el código de cada tarea puede ocupar dos páginas (por el enunciado).
- Mappeamos temporalmente las direcciones físicas donde queremos copiar el código en una dirección no utilizada.
- Copiamos el código en las dos páginas pedidas.
- Limpiamos los mappings temporales y devolvemos la dirección del nuevo directorio de páginas.

Ejercicio 6

defines.h

- Declaramos nuevas constantes para el uso en la GDT.

tss.c

Decidimos definir todas las tss's ahora, haciendo futurología (Facu Linari nos dijo que en Ingeniería I nos van a enseñar a no hacer esto, pero todavía no la cursamos :P)

- Implementamos `tss_init` para que inicialice todas las tss's de nuestro sistema (la de la tarea inicial, idle y de las pelotas con sus respectivos handlers).
- Decidimos guardarnos las tss's de las tareas y los handlers en un array para poder recorrerlo más fácilmente luego.
- Creamos un método `tss_new_ball` que recibe por parámetro el número de jugador (AlB + tipo) y si es handler o no, y se encarga de devolver una tss inicializada de manera adecuada.
- Creamos el método `tss_ball_reset`, que se encarga de reconfigurar el tss de una de las tareas de pelotas previo a su ejecución (en particular por el stack y un par de cosas más). Usaremos este método siempre que queramos lanzar una nueva pelota. Hicimos una función `tss_ball_handler_reset` análoga, específica para los handlers, puesto que hay algunas diferencias entre los stacks de las pelotas y los handlers. Estos métodos resettean también los registros selectores de segmento y código.

- En las TSS's, configuramos:
  - ESP0: Pedimos una nueva página de kernel para el stack lvl 0 y le asignamos el final de esta página.
  - SS0 = Segmento de datos Lvl. 0
  - CR3 = Dirección del Page Directory de la tarea
  - EIP = 0x8000000
  - EFLAGS = 0x202 (interrupciones habilitadas)
  - ESP: Stack de nivel 3 de la tarea (varía si es o no es handler)
  - Registros selectores de segmento: Segmento de datos Lvl. 3
  - Todo el resto de los registros en 0

gdt.c

- Escribimos las gdt entries de todas las TSS que necesitaremos. Estas son 6 para las tareas y otras 6 para sus respectivos handlers, una más para la tarea inicial y una última para la tarea *idle*. Para
- Colocamos el bit busy de la gdt entry de la tarea inicial en 1.

Ejercicio 7

defines.h y defines.mac

Metimos varias constantes afines al juego (keys, configs, etc.)

game.h y game.c

- Creamos todas las estructuras necesarias para el manejo del juego.
- Creamos la función `game_executeFrame` que se encarga de ejecutar todos los cálculos del juego y dibujado de pantalla.
- Agregamos una función `game_talk` que será llamada por la syscall `talk`.
- Agregamos una función `game_informAction` que será llamada por la syscall `informAction`.
- Agregamos una función `game_kbInput` que registrará el input del usuario. Junto con `game_executeFrame` y `game_tick`, estos inputs serán tomados en cuenta en el momento adecuado (la consigna decía cada 30 ticks, pero lo hicimos cada menos porque nos parecía injugable de otra forma, las paletas se movían muy lento en comparación con las pelotas).
- Creamos la función `game_showDebugInfo` que se encarga de mostrar el chart de debug.

i386.h

Agregamos inlines para conseguir los valores de los registros de propósito general

isr.asm

- Excepciones
  - En lugar de printear en pantalla como veníamos haciendo, comenzamos a llamar a la función `game_showDebugInfo` para mostrar la pantalla de debug.
- isr32 (clock)
  - Agregamos el comportamiento de salto de tarea y el llamado a `game_executeFrame` en el tick 7 para que se hagan los cálculos propios del juego y el dibujado de la pantalla.
- isr33 (teclado)
  - Llamamos a `game_kbInput` para que parsee el input.
- isr47 (syscalls)
  - Implementamos las syscalls pedidas por la cátedra, delegando a `game.c` siempre que podamos.

kernel.asm

- Llamamos a `sched_init` y `game_init` para inicializar todas las estructuras correspondientes.

sched.c

- Agregamos estructuras para almacenar los selectores de las TSS's de las tareas (pelotas y handlers correspondientes), así como las direcciones de los handlers (que serán configurados conforme se registren).
- Creamos funciones de utilidad (e.g: `sched_isHandler`, que indica si la tarea actualmente en ejecución es un handler).
- Definimos las funciones necesarias para proveer el comportamiento deseado del módulo.

screen.c

- Agregamos funciones para guardar y restaurar la pantalla en y desde un puntero a words de Text UI. Estas son utilizadas cuando queremos mostrar el mensaje del modo debug (y posteriormente reanudar el juego).