UNIVERSIDAD DE BUENOS AIRES

Facultad de Ciencias Exactas y Naturales
Departamento de Computación
Organización del Computador

RECUPERATORIO AL TRABAJO PRÁCTICO NÚMERO 3

Nombre de Grupo: Napolitana con Jamón y Morrones

Alumnos:

Izcovich, Sabrina | sizcovich@gmail.com | LU 550/11

López Veluscek, Matías | milopezv@gmail.com | 926/10

Índice

1. Introducción

El siguiente trabajo práctico consiste en un conjunto de ejercicios en los que se aplican de forma gradual los conceptos de System Programming vistos a lo largo de la materia. El objetivo de éste consiste en construir un sistema capaz de correr el juego *Infección*. Para la realización del mismo, debimos crear el soporte para que los jugadores (o sea las tareas) puedan ejecutar las reglas del juego. Dado que las tareas eran capaces de hacer cualquier cosa, debimos considerar que el sistema debía tener la capacidad de capturar cualquier problema y poder quitar a la tarea del juego.
A lo largo del trabajo práctico, se utilizaron los mecanismos que posee el procesador para la programación desde el punto de vista del sistema operativo enfocados sobre el sistema de protección. En lo que sigue, se detallan explícitamente los pasos realizados para lograr el objetivo mencionado.

2. Desarrollo

2.1. Ejercicio 1

En el primer ejercicio, nos limitamos a completar la GDT tomando como ejemplo la GDT_NULL provista por la cátedra. Para ello, debimos definir tres segmentos de códigos y de datos. El primero de ellos (el 0) correspondiente al kernel, el segundo (el 2) al árbitro y el tercero (el 3) a los jugadores.

Pregunta 1: ¿Qué ocurre si se intenta escribir en la fila 26, columna 1 de la matriz de video, utilizando el segmento de la GDT que direcciona a la memoria de video? ¿Por qué?

La consecuencia de intentar escribir en la fila 26, columna 1 de la matriz de video depende de cómo fue definido el segmento. Si el tamaño del segmento es igual al tamaño de la pantalla en bytes, entonces va a ocurrir un Segmentation Fault. En el caso en el que el segmento no sea del mismo tamaño de la pantalla, el caracter no va a ser visible.

Pregunta 2: ¿Qué ocurre si no se setean todos los registros de segmento al entrar en modo protegido? ¿Es necesario setearlos todos? ¿Por qué?
En el caso en el que los registros no se seteen se genera un General Protection.

2.2. Ejercicio 2

Pregunta 3: ¿Cómo se puede hacer para generar una excepción sin utilizar la instrucción int? Mencionar al menos 3 formas posibles.

Pregunta 4: ¿Cuáles son los valores del stack cuando se genera una interrupción? ¿Qué significan? (Indicar para el caso de operar en nivel 3 y nivel 0).

2.3. Ejercicio 3

Pregunta 5: ¿Puede el directorio de páginas estar en cualquier posición arbitraria de memoria?

Pregunta 6: ¿Es posible acceder a una página de nivel de kernel desde usuario?

Pregunta 7: ¿Se puede mapear una página física desde dos direcciones virtuales distintas, de manera tal que una esté mapeada con nivel de usuario y la otra a nivel de kernel? De ser posible, ¿Qué problemas puede traer?

2.4. Ejercicio 4

Pregunta 8: Qué permisos pueden tener las tablas y directorios de páginas? ¿Cuáles son los permisos efectivos de una dirección de memoria según los permisos del directorio y tabla de páginas?

Pregunta 9: ¿Es posible desde dos directorios de página, referenciar a una misma tabla de páginas? **Pregunta 10:** ¿qué es TLB (Translation Lookaside Buffer) y para qué sirve?

2.5. Ejercicio 5

Pregunta 11: ¿qué pasa si en la interrupción de teclado no se lee la tecla presionada? **Pregunta 12:** ¿qué pasa si no se resetea el PIC?

2.6. Ejercicio 6

Pregunta 13: Colocando un breakpoint luego de cargar una tarea, ¿Cómo se puede verificar, utilizando el debugger de Bochs, que la tarea se cargó correctamente? ¿Cómo se llega a esta conclusión?

Pregunta 14: ¿Cómo se puede verificar que la conmutación de tarea fue exitosa? **Pregunta 15:** Se sabe que las tareas llaman a la interrupción `0x80` y por `0x90`, ¿qué ocurre si esta no está implementada? ¿Por qué?

2.7. Ejercicio 7

3. Ejercicio 1: Modo Protegido y GDT

3.1. GDT

Al declararla completamos los 2 descriptores de código, los 2 de datos, y el de video pedidos. De las 10 posiciones nulas que menciona el enunciado, utilizamos 5 para los descriptores de TSS de las tareas, más los de la tarea inicial y la idle, por lo tanto quedan 3 vacías (nulas), entre ellas la requerida por Intel.

Figura 1: Descriptor de segmento de la GDT

Los 2 descriptores de datos y los 2 de código tienen muchos campos en común, los únicos que varían son DPL y Tipo. Para los descriptores de código utilizamos tipo 0xA (lectura y ejecución), y para los de datos, 0x2 (lectura y escritura). Dentro de los de datos, hay uno con DPL 0, para el kernel, y otro de DPL 3, para las tareas. ídem con los de código. Utilizamos granularidad y segmentos de 32bits (D/B), L está en 0 ya que es código de 32bits, AVL es a gusto propio, están presentes, y no son de sistema.

Para abarcar los primeros 2GB de la memoria, claramente la base es 0x00000000, y ya que utilizamos granularidad:

$$2\text{GB} = 2048\text{MB} = 2097152\text{KB} = 524288 \text{ páginas de } 4\text{KB} = 0\text{x}80000.$$

Como el límite es la última posición accesible, en este caso es 0x7FFFF.

Figura 2: Campos comunes en nuestros 4 descriptores de segmento

El descriptor de video es igual al de datos de nivel 0, pero varían su base y tamaño, y no hace falta utilizar granularidad, lo que hace el límite claro.

Figura 3: Descriptor del segmento de video

Los descriptores de TSS son muy parecidos, pero llevan el tipo 0x9 (porque todos comienzan con el bit de busy en 0) y el límite 103d (0x67) ya que tienen un tamaño de 104 bytes.

índice	Descriptor
00 / 0x00	Nulo
01 / 0x08	Nulo
02 / 0x10	Nulo
03 / 0x18	TSS Tarea Inicial
04 / 0x20	TSS Tarea Idle
05 / 0x28	TSS Tarea 1
06 / 0x30	TSS Tarea 2
07 / 0x38	TSS Tarea 3
08 / 0x40	TSS Tarea 4
09 / 0x48	TSS Tarea 5
10 / 0x50	Código nivel 0
11 / 0x58	Código nivel 3
12 / 0x60	Datos nivel 0
13 / 0x68	Datos nivel 3
14 / 0x70	Video

Cuadro 1: La GDT resultante

3.2. Pasaje a Modo Protegido

Antes de cargar la GDT, habilitamos A20, utilizando el código provisto por la cátedra, para poder direccionar toda la memoria. Luego, cargamos la GDT con la instrucción lgdt, y pasándole base y límite de la GDT. Finalmente seteamos el bit PE del registro CR0, y saltamos a 0x50:mp, para así setear el CS en 0x50, el selector de nuestro código de nivel 0. Mp es simplemente una etiqueta donde comienza nuestro código para el modo protegido.

Apenas entramos a modo protegido seteamos todos los selectores (menos CS por supuesto), en 0x60, nuestros datos de nivel 0, y seteamos ebp y esp (la base de la pila) a 0x21000, ya que la pila empieza en 0x20000. El código es el siguiente:

```
BITS 16
call habilitar_A20
lgdt [GDT_DESC]

mov eax, cr0
or eax, 1
mov cr0, eax ;PE = 1
jmp 0x50:mp ;CS = 0x50

BITS 32
    mp:
xor eax, eax
mov ax, 0x60
mov ds, ax
mov ss, ax
mov es, ax
mov fs, ax
mov gs, ax

mov esp, 0x21000
mov ebp, 0x21000
```

Figura 4: Código para pasar a Modo Protegido, cargar los segmentos definidos anteriormente e inicializar la pila, tomado de `kernel.asm`.

3.3. Imprimir en pantalla

Una vez completada la GDT y ya en modo protegido, se pide limpiar la pantalla y pintar la primer y última línea de color de fondo negro y letras blancas, usando el segmento de video declarado anteriormente. Para esto implementamos una rutina (`clearScreen` en `kernel.asm`) que setea FS en `0x70` (índice del segmento de video en la GDT) y recorre la matriz de $80 * 25$, moviendo el caracter `0x77` (que corresponde a fondo negro y caracter negro) a la posición de memoria direccionada con el selector de video y el offset correspondiente.

```
clearScreen:
mov ax, 0x70
mov fs, ax ;video
mov ecx, 80*25*2
.limpiarPantalla:
mov byte [fs:ecx-1], 0x77
loop .limpiarPantalla
```

Figura 5: Código de la rutina para limpiar la pantalla, extraído de `kernel.asm`.

Para pintar la primer y última linea, se realiza un procedimiento análogo, solo que al caracter se le asigna color blanco (consultar `kernel.asm`).

3.4. Preguntas

3.4.1. Pregunta 1

> ¿Qué ocurre si se intenta escribir en la fila 26, columna 1 de la matriz de video, utilizando el usando el segmento de la GDT que direcciona a la memoria de video? ¿Por qué?

Si intentamos escribir la fila 26 de la memoria de video utilizando el descriptor previamente declarado, nos encontramos con un General Protection Fault, ya que esa memoria (fila 26, columna 1), es posterior al límite del segmento declarado, que abarca 25 filas.

Pregunta 2

> ¿Qué ocurre si no se setean todos los registros de segmento al entrar en modo protegido?, ¿Es necesario setearlos todos?, ¿Por qué?

Los segmentos que son necesarios setear son CS, DS y SS, si no se setean, se produce GPF. Los otros no son necesarios. Por convención de Intel.

4. Ejercicio 2: IDT

4.1. Implementación

En base al modelo provisto por la cátedra, se crearon dos macros de trap gates, uno con permiso de nivel 0, para las interrupciones de kernel, y otro con permiso de nivel 3, para las interrupciones de clock, teclado y de servicios. La configuración es la siguiente:
La diferencia en ambos modelos esta en los bits de DPL, el resto se configura de la misma manera. Para obtener el offset se hace uso de una etiqueta definida mediante un macro provisto por la cátedra en el archivo isr.asm, el cual define un comportamiento básico para el manejo de interrupciones; a ese macro se le agregó la llamada a IMPRIMIR_TEXTO, una función provista por la cátedra, la cual muestra en pantalla.
Una vez definidos los macros, la creación de la IDT es trivial. Para las interrupciones del kernel se invoca al macro IDT_ENTRY, el cual fabrica un gate con permiso de nivel 0, y para las interrupciones de teclado, clock y servicios se usa IDT_ENTRY3 que fabrica un gate con permiso de nivel 3.

Para que esto tenga sentido y funcionen las interrupciones se hace un call en la linea 83 de kernel.asm para generar la tabla y luego se usa la instrucción `lidt` para

cargar IDT_DESC, el descriptor de la idt, al registro IDTR.
Además fue necesario habilitar las interrupciones que se deshabilitaron en el primer ejercicio para poder entrar a modo protegido, por lo que en la linea 133 de kernel.asm, después de inicializar partes del kernel que necesitan tener las interrupciones deshabilitadas, se usa el comando `sti` para habilitar las interrupciones y que la idt tenga efecto.

4.2. Preguntas

Pregunta 3

> ¿Cómo se puede hacer para generar una excepción sin utilizar la instrucción `int`? Mencionar al menos 3 formas posibles.

Para generar una excepción sin utilizar la instrucción `int` se puede hacer uso de los siguientes métodos:

- Dividiendo por 0.
- Escribiendo a un sector de memoria que no esta mapeado en la unidad de segmentación y/o paginación.
- Escribiendo a un sector de la memoria para el cual no se tiene suficiente nivel de permiso.

Pregunta 4

> ¿Cuáles son los valores del stack cuando se genera una interrupción? ¿Qué significan? Indicar para el caso de operar en nivel 3 y nivel 0.

Cuando se genera una interrupción, se debe preservar el estado actual de los flags y todos los registros que son necesarios para retornar a la normal ejecución una vez resuelta la interrupción. Cuando se produce la interrupción puede que sucedan 2 cosas, la interrupción tiene el mismo nivel de privilegio que el proceso en el que se generó o bien tiene un privilegio diferente. Como nuestro kernel opera en nivel de privilegio 3 y solo cambia de privilegio al ejecutar un servicio o al generar una interrupción de kernel, siempre que suceda una interrupción de nivel 0 se produciría un cambio de privilegio con lo que el stack de la interrupción para una interrupción de nivel 0 siempre es el formato de cambio de privilegio y para una interrupción de nivel 3 siempre es el formato de mismo nivel.

Los datos que se almacenan por automático son entonces:

- SS: El selector de segmento donde se encuentra el stack del nivel anterior. Esto se debe hacer para no perder el stack anterior ya que se produce un cambio de stack al cambiar el privilegio.
- ESP: El puntero a la posición de memoria donde estaba el ESP antes del cambio de privilegio. Esto se debe hacer para no perder el stack anterior ya que se produce un cambio de stack al cambiar el privilegio.
- EFLAGS: El registro que contiene el estado de los flags del sistema. Como muchas operaciones afectan el estado de este registro, se debe conservar para asegurarse de que el estado del proceso que se interrumpió sea el mismo que al salir de la interrupción.
- CS: El selector de segmento del proceso interrumpido.
- EIP: El puntero que indica que dirección se debe ejecutar al retornar de la interrupción.
- Error Code: El código de error codifica información importante sobre el error.

El Error Code tiene 32 bits, de los cuales los bits 16 a 31 están reservados para mantener el stack alineado. El bit 0(EXT) indica si la interrupción es por un evento externo al procesador, el bit 1(IDT) indica si el índice, que va de los bits 3 a 15, hace referencia a un elemento de la IDT o a un descriptor en la GDT/LDT y el bit 2(TI) indica, solo en caso de que le bit 1 indique que se trata de un descriptor de GDT/LDT, en que tabla se encuentra el descriptor.

5. Ejercicio 3: Paginación

5.1. Implementación

Para esta parte utilizamos código assembler, ya que era bastante simple. En la dirección 0x21000, donde está la base del directorio, marcamos 1024 páginas como no presentes, y luego la primera entrada (posición 0x21000), la reescribimos como presente, con los permisos correspondientes y apuntando a 0x22000, que es donde debía estar la tabla de páginas. En otras palabras, en la posición 0x21000, se encuentra el valor 0x22003, que es la dirección de la base de la tabla de páginas más atributos, que son supervisor, lectura/escritura, y presente.

Luego nos encargamos de la tabla de páginas, en la posición 0x22000 volvemos a marcar 1024 entradas como no presentes, y tenemos que hacer identity mapping para el rango 0x00000000 a 0x1FFFFF, que es el equivalente a los primeros 2MB de memoria. Una tabla nos alcanza para 1024 entradas de 4KB, que son 4MB. Por lo tanto marcamos las primeras 512 entradas de la tabla con identity mapping, que implica poner en la

memoria que estoy escribiendo, 0x1000*índice*4 (multiplicamos por 4 por el tamaño de cada entrada de la tabla), más los permisos (3, mismos que antes).

5.2. Preguntas

Pregunta 5

> ¿Puede el directorio de página estar en cualquier posición arbitraria de memoria?

No, debe ser una posición de memoria accesible antes de activar paginación, que es la alcanzada por segmentación. Por lo tanto el directorio de páginas debe estar a lo sumo en la posición 0x7FFFF000, cualquier acceso a una memoria mayor va a resultar en GPF. Además, la dirección base estar alineada a 4KB.

Pregunta 6

> ¿Es posible acceder a una página de nivel de kernel desde usuario?

No, esto no es posible ya que para hacerlo se requieren por lo menos los mismos permisos, o mayores (menores numéricamente). Intentar acceder con privilegios menores va a resultar en una GPF.

Pregunta 7

> ¿Se puede mapear una página física desde dos direcciones virtuales distintas, tal que una está mapeada con nivel de usuario y la otra a nivel de kernel? De ser posible, ¿qué problemas podría traer?

Se puede, pero se corre el riesgo que un usuario (tarea) modifique datos o código que son del kernel.

6. Ejercicio 4: MMU

El presente sistema operativo cuenta con una unidad de manejo de memoria (o MMU), encargado de asignar páginas de memoria a la tareas y al mismo kernel. Sus funcionalidades consisten en mapear y desmapear direcciones físicas a virtuales e inicializar las estructuras de directorio y tablas de páginas para el kernel y para las tareas. Estas implementaciones se encuentran en el archivo `mmu.c`.

6.1. mapear_pagina y unmapear_pagina

La función `void mapear_pagina(unsigned int virtual, unsigned int cr3, unsigned int fisica, unsigned int supervisor)` se encarga de mapear una dirección física de memoria y una página determinada, para un CR3 particular, y asignarle los permisos deseados. El parámetro **supervisor** puede ser 0 o 1. En caso de ser uno, indica que esa página tendrá permisos de supervisor, en caso de ser cero, será de usuario. El código es el siguiente:

```
void mapear_pagina(unsigned int virtual, unsigned int cr3, unsigned int fisica, unsigned int sup) {
unsigned int indiceDir = virtual >>22; //los primeros 10 bits son el índice del directorio
unsigned int indiceTabla = (virtual << 10 ) >> 22; //los segundos, el índice de la tabla
//puntero a la tabla
unsigned int* tabla = (unsigned int*)(cr3+indiceDir*4);
//puntero a la base de la memoria
unsigned int* baseMemoria = (unsigned int*)((*tabla&0xFFFFF000)+indiceTabla*4);

unsigned int permisos;
if (supervisor == 1) permisos = 3;
else permisos = 7;
*baseMemoria = (fisica& 0xFFFFF000)+permisos;
tlbflush();
}
```

Figura 6: Código de mapear_pagina, extraído de `mmu.c`.

Al final de la función, ejecutamos `tlbflush()` para resetear el caché de traducción de direcciones, ya que se puede estar remapeando alguna página que estaba en la TLB. La función `void unmapear_pagina(unsigned int virtual, unsigned int cr3)`, es igual, pero le asigna ceros a esa dirección virtual. También es importante ejecutar `tlbflush()` en este caso (ver `mmu.c`).

6.2. Inicialización de directorios y tablas de página

Implementamos una rutina (`incializar_dir_tarea`) que se encarga de inicializar un directorio de páginas y tablas de página para una tarea. Recibe como parámetro el número de tarea, y en base a este calcula el CR3 correspondiente. Luego genera las 1024 entradas del directorio de páginas con todos descriptores nulos, y apunta el CR3 a este directorio. Copia el código de la tarea a su área asignada dentro de la *arena* y mapea esta página a la dirección virtual `0x3A0000` (llamando a `mapear_pagina`). Además mapea la página correspondiente a la pila de la tarea a la dirección virtual `0x3B0000`. Finalmente, llama a `inicializar_tss_tarea`, función que se encarga de inicializar la TSS para la tarea y que describiremos en la sección TSS.

6.3. Preguntas

Pregunta 8

¿Qué permisos pueden tener las tablas y directorios de páginas? ¿Cuáles son los permisos efectivos de una dirección de memoria según los permisos del directorio y tabla de páginas?

Los permisos de las tablas y directorios de páginas se pueden definir como de Usuario o Supervisor. Las distintas combinaciones de esos permisos se definen en la siguiente tabla:

Figura 7: *: depende del flag `CR0.WP`. Resultados de la combinación de atributos entre el directorio y la tabla de páginas. Intel(R) 64 and IA-32 Architectures Software Developer's Manual, Volume 3, Chapter 5: Protection, p. 5-42

Pregunta 9

¿Es posible desde dos directorios de página, referenciar a una misma tabla de páginas?

Sí, se puede. Se debe escribir en la entrada correspondiente de cada uno de los directorios de tablas de páginas la dirección de la misma tabla que se quiere referenciar. Se puede hacer en el caso de que dos tareas con directorios propios quieran referenciar las mismas páginas con distintos privilegios en las mismas direcciones físicas.

Pregunta 10

¿qué es TLB (Translation Lookaside Buffer) y para qué sirve?

Es un buffer especial para guardar las traducciones de direcciones lineales a físicas. De esta manera, para un segundo acceso a la página, se evitan dos accesos a memoria para recorrer el directorio de tablas y la tabla apuntada. Sin embargo, cuando se remapea una página a una dirección virtual diferente (o se desmapea), se debe invalidar este caché para que sea consistente con el estado de las estructuras en memoria.

7. Ejercicio 5: Interrupciones (primera parte)

En esta parte implementamos la primera parte de las rutinas de atención de interrupciones del reloj, teclado y la `int 45`. Ã‰stas se encuentran implementadas en assembler en el archivo `isr.asm`. Antes de habilitar las interrupciones, es necesario (además de tener todas las estructuras definidas anteriormente, paginación habiltiada, etc) configurar controlador de interrupciones, o PIC. Para esto contamos con las funciones `deshabilitar_pic`, `resetear_pic` y `habilitar_pic`. Luego de llamar a esas funciones (en `kernel.asm`), queda la interrupción de reloj mapeada a la interrupción 32 y la de teclado a la 33.

7.1. Reloj

Por ahora, la única funcionalidad que se debe asociar con la interrupción de reloj, es hacer "girar" un reloj en la esquina inferior derecha de la pantalla. Para esto, se llama a las funciones provistas por la cátedra `fin_intr_pic1` para avisar al pic que atendimos la interrupción y `proximo_reloj` para mostrar la animación descripta anteriormente.

7.2. Teclado

La interrupción de teclado por el momento lo que va a hacer es, al leer que se presionó una tecla correspondiente a un número, imprimirlo en la esquina superior derecha de la pantalla. Con la instrucción `in al, 0x60` se lee el scan code de la tecla presionada, y se realizan comparaciones para saber de qué tecla se trata. En primer lugar se compara con 0x0b y si es mayor, se salta al final de la interrupción, ya que no corresponde a un número. Si es igual, es un cero y es tratado como un caso especial. Luego se compara con 0x2 y si es menor también se salta a fin, por no tratarse de un número. En el caso del cero, se imprime un cero, y en otro caso se decrementa el scan code y se le suma 0x30 para obtener el ascii deseado. Finalmente, se imprime.

7.3. interrupción 0x45 (a.k.a 69)

Esta interrupción será la encargada de proveerle a las tareas los servicios del sistema operativo de administración de memoria e impresión por pantalla. Por el momento, nos limitamos a utilizarla para colocar el valor 42 en `eax`.

7.4. Preguntas

Pregunta 11

¿qué pasa si en la interrupción de teclado no se lee la tecla presionada?

Si no se lee la tecla, su valor se encola en un buffer especial que tiene el controlador de teclado, y en la próxima lectura, se extrae de ahí.

Pregunta 12

¿qué pasa si no se resetea el PIC?

Si no se resetea el PIC se produce un conflicto ya que las IRQs de 0 a 7 están mapeadas a las interrupciones 0x8 a 0xF que son los números de interrupciones de las excepciones internas del procesador. Por esto, es necesario remepear las IRQs para que correspondan a interrupciones a partir de las 32 (de la 0 a la 31 están reservadas por el procesador).

8. Ejercicio 6: Tareas

Para realizar la conmutación de tareas, debemos agregar nuevas entradas a la `GDT` y definir un arreglo de *TSS*s, debidamente inicializados. Estos últimos, decidimos definirlos como *structs* de `C` en el archivo `tss.c`.

8.1. Descriptores de TSS en la GDT

Como dijimos, para conmutar las tareas es necesario definir en la GDT las entradas correspondientes a los descriptores de TSS de la `tarea_inicial` (esta en particular está para que cuando se realize la primera conmutación se encuentre en la GDT un descriptor válido), la `IDLE` y las 5 tareas restantes provistas por la cátedra. En principio, en el archivo `gdt.c` agregamos las siguientes entradas, que luego vamos a inicializar:

Segmento		Atributos						
Índice	Offset	Base	DPL	G	Límite	P	S	Tipo
3	0x18	0	0	1	0x67	1	0	0x9 (TSS, available)
4	0x20	0	0	1	0x67	1	0	0x9 (TSS, available)
5	0x28	0	0	1	0x67	1	0	0x9 (TSS, available)
				...				
9	0x48	0	0	1	0x67	1	0	0x9 (TSS, available)

Cuadro 2: Descriptores de TSS definidos en GDT.c.

Como se observa en la tabla, en los descriptores lo único que se define es el límite (cada TSS va a medir 0x67 bytes), se los pone en presente, el tipo (TSS), se prende el bit de granularidad y se le asigna cero a DPL. El resto de los atributos van en principio en cero, para luego definirlos en `tss.c`.

8.2. Inicialización de tareas

Para la `tarea_inicial` definimos la función `inicializar_tss_inicial` que lo único que hace es asignarle la base correspondiente (ya que el resto de los atributos quedan en cero).

La `IDLE` la definimos mediante la función `inicializar_tss_idle` que hace lo siguiente:

- Le asigna como base la dirección 0x10000.
- EIP lo inicializa con la dirección 0x3A0000 que es la dirección virtual de código, esto vale para todas las demás tareas.
- Dado que la pila de la `IDLE` se encuentra en 0x3F000, asigna al ESP y EBP este valor más el tamaño de la página.
- Setea flags con interrupciones habilitadas.
- Dado que esta tarea corre en nivel cero, ESP0 va a ser igual a ESP.
- Setea los selectores de segmento (cs = 0x50, el resto en 0x60)
- Le asigna el mismo CR3 que el kernel.
- Finalmente, mapea la página de código a la dirección virtual 0x3A0000.

El resto de las tareas que corren en nivel 3 se van a inicializar a travÃ©s de un arreglo (`tsss`). La función encargada es `inicializar_tss_tarea` que recibe como parámetro el índice que cada tarea y hace lo siguiente:

- Le asigna como base la dirección de `tsss[indice]`.
- EIP lo inicializa con la dirección 0x3A0000 que es la dirección virtual de código, esto vale para todas las tareas.
- al EBP y ESP les asigna 0x3B0000, que es la dirección virtual de la pila para todas las tareas.
- Setea flags con interrupciones habilitadas.
- El ESP0 se calcula haciendo $0x3A000 + indice * TAMANO\_PAGINA + TAMANO\_PAGINA$ dado que en 0x3A000 comienza la pila de nivel cero para la primer tarea, y finalmente se suma TAMANO_PAGINA porque queremos apuntar al fondo de la pila.

- Setea los selectores de segmento (cs = 0x5b, el resto en 0x6b)
- Setea el CR3 como $0x30000 + (0x1000 * (indice))$.

8.3. Conmutación de tareas

Una vez definidas estas estructuras, para cambiar de tarea por primera vez, es decir, saltar de la tarea inicial a la IDLE, basta con hacer un `jmp 0x20:0`.

8.4. Preguntas

Pregunta 13

> Colocando un breakpoint luego de cargar una tarea, ¿Cómo se puede verificar, utilizando el debugger de Bochs, que la tarea se cargó correctamente? ¿Cómo se llega a esta conclusión?

Si el breakpoint se pone inmediatamente después del salto de tarea, sería un buen indicio que en principio no se detenga el flujo de ejecución, ya que después del jump far se debería estar ejecutando el código de la tarea que se acaba de cargar. Esto se puede verificar poniendo un breakpoint antes del salto, y luego haciendo next mediante el debuguer de Bochs. Luego de ejecutar el salto, se puede observar en la GDT el descriptor de TSS correspondiente a esa tarea, y corroborar que el bit de busy esté prendido. también se puede chequear que los registros TR y CR3 sean los correspondientes a la tarea que debió cargarse.

Pregunta 14

> ¿Cómo se puede verificar que la conmutación de tarea fue exitosa?

Esto se puede hacer utilizando lo explicado en el item anterior para ver que se cargó la nueva tarea correctamente. Para corroborar que el contexto de la anterior se guardó correctamente, habría que volver a saltar a ella.

Pregunta 15

> Se sabe que las tareas llaman a la interrupción `0x45`. ¿qué ocurre si esta no está implementada? ¿Por qué?

Si no está implementada la interrupción, se produce una excepción de protección general.

9. Ejercicio 7: Scheduler, interrupciones (segunda parte) y backtrace

9.1. Scheduler

El funcionamiento del scheduler es muy básico, simplemente debe decirle a la interrupción de reloj, qué tarea es la próxima que debe ejecutar. Inicialmente todas son candidatas, y apenas alguna genera una excepción de algún tipo, debe ser desalojada,

esto significa que deja de ejecutarse en ese momento, y no vuelve a ser considerada como próxima a ejecutar.

La idea es manejarnos con los índices de las tareas, entonces decidimos implementarlo simplemente como un arreglo de unsigned shorts (que funcionan como bools), llamado *tareas*. El arreglo tiene longitud CANT_TAREAS, e inicializamos todas las posiciones en 1. Entonces para saber si una tarea i es candidata o no a ejecutarse, hay que ver si $tareas[i]$ es igual a 1[1]. también tenemos una variable llamada *tareaActal* para guardar la tarea actual, inicializada en 4 (así al ejecutarse por primera vez la función de próxima tarea a ejecutar, resulta en 0, la primera). De esta forma tenemos la información de qué tarea se está ejecutando actualmente, y cuáles son candidatas a ejecutarse (a estas las vamos a llamar "vivas", las que no son candidatas, "muertas"), y la función de proximoIndice, que nos devuelve el índice de la próxima tarea a ejecutar, es directa. La función hace dos cosas: Actualiza el valor de *tareaActal*, y devuelve el índice de la próxima tarea que debe ejecutarse, o un valor especial, que es 6, y nos dice que no debemos saltar de tarea. ¿Cuándo puede suceder esto? En dos casos: Cuando el próximo índice a ejecutar es una tarea que no está viva (por la implementación de la función, puede suceder, e implica que todas las tareas están muertas), o cuando estoy en la misma tarea que debo ejecutar, sin estar idle, esto implicaría saltar a la misma tarea, que esta busy, y resultaría en un GPF.

Figura 8: Diagrama de flujo del algoritmo que decide el próximo indice de tarea a ejecutar

El código para primero calcular el próximo índice es simple:

```
unsigned short nuevoIndice=(tareaActual+1) % 5;
unsigned int i;
for(i=0;i<5;i++)
{
if(tareas[(nuevoIndice+i) % 5]==1)
{
nuevoIndice=nuevoIndice+i;
break;
}
}
```

Y debido a esta implementación es que esta parte puede devolver el índice de una tarea que está muerta. Como se ve, en los casos donde no hay que saltar de tarea, se devuelve un índice especial, 6, y no se modifica la tareaActual, ya que no va a saltar de tarea. En los casos donde sí salta, se actualiza tareaActual al valor que se va a saltar, para tenerlo actualizado la próxima vez que se entre a este código.

Para ver si estamos en el estado idle (por ejemplo si una tarea cede su quantum restante a la tarea idle por pedir una página, en la próxima interrupción de reloj

[1] Indizamos las tareas desde el 0.

entramos a esta rutina estando idle), miramos el CR3 actual, que se lo pasamos como parámetro desde la rutina anterior, y simplemente lo compara con 0x20.

9.2. Interrupciones (segunda parte)

Las interrupciones que faltan terminar son la 0x45, para que las tareas puedan llamar a los servicios set_page y get_code_stack, la de reloj, para que integre el scheduler y efectivamente manejen el intercambio de tareas, y las de las teclas P y R.

interrupción 0x45

Para esto primero implementamos las funciones set_page y get_code_stack. Para set_page, como ya tenemos implementada mapear_pagina, el procedimiento es muy simple:

```
void map_compartida(unsigned short indice)
{
unsigned int dirFisica = indice*TAMANO_PAGINA+INIT_LUCHA;
unsigned int cr3actual = rcr3();
mapear_pagina(TASK_SHARE, cr3actual, dirFisica, 0);
}
```

Figura 9: Versión reducida del código de set_page

Primero se calcula la dirección física que queremos mapear, y luego se mapea a TASK_SHARE (0x3C0000), utilizando el cr3 actual, que es el de la tarea que pidió el servicio set_page.

Como debíamos mostrar la página compartida en pantalla, y eventualmente sacarla de la pantalla si la tarea muere, decidimos guardar el índice de la página compartida de cada tarea, si la tuviese. Para esto utilizamos un arreglo, donde se guarda el índice de la página compartida para cada tarea. Cada vez que una tarea mapea una página nueva, nos fijamos si ya tenía una, si la tiene, cambiamos el índice en el arreglo, y cambiamos en la pantalla la página que se muestra como la compartida de esa tarea[2].

Para get_code_stack, simplemente hacemos la cuenta inversa que en set_page para conseguir los índices (la hacemos dos veces, una con la posición de memoria del código y otra con la del stack), y devolvemos el índice. A continuación puede verse el código para getCode, el de getStack es análogo pero utiliza el arreglo que contiene las posiciones de memoria de los stacks en vez del de las de código.

Las mencionadas previamente son las funciones en C que llama la interrupción 0x45, cuyo funcionamiento es el siguiente:

Figura 10: Diagrama de flujo del handler de la interrupción 0x45

[2]Este fragmento de código es el que no se muestra en la figura 9, puede verse en sched.c

```
unsigned short getCode()
{
unsigned int memoriaTareaActual = pc[tareaActual];
memoriaTareaActual = memoriaTareaActual - INIT_LUCHA;
memoriaTareaActual = memoriaTareaActual >> 12
return (unsigned short) memoriaTareaActual;
}
```

La función desalojarActual se llama es muy simple en sí, pero también borra de la pantalla las páginas de la tarea, lo cual alarga el código un poco. La parte pertinente a desalojar la tarea es simplemente poner un cero en el arreglo del scheduler. Antes de desalojarla, desde ASM se llama a la función (también en ASM), que se encarga de mostrar por pantalla el problema que causó, el backtrace, los registros, etc. también puede apreciarse que luego de llamar a set_page, se salta a la tarea idle, para cederle el resto del tiempo restante. Esto no pasa luego de llamar a get_code_stack, donde se sale de la interrupción normalmente y la tarea sigue ejecutándose.

interrupción de reloj

Una gran parte de la interrupción de reloj está explicada en la parte del scheduler, donde se muestra el funcionamiento del procedimiento para elejir el próximo índice de tarea a ejecutar. Es importante tener en cuenta que esta función puede devolver el índice especial 6, que indica que no hay que saltar de tarea, por las razones ya explicadas. Acá también puede apreciarse la funcionalidad de pausar y reanudar (las teclas P y R respectivamente), que será explicado en más detalle próximamente, pero son las encargadas de ver si hay que pausar o no. Entonces, cuando surje la interrupción de reloj, el handler realiza lo siguiente:

Figura 11: Diagrama de flujo del handler de la interrupción de reloj

Lo que en este caso es llamado proximoIndice, es la función previamente mencionada que o bien devuelve el índice de la próxima tarea a ejecutar, o el índice 6. Cuando el handler del reloj decide que efectivamente hay que saltar a una tarea determinada, se debe calcular el índice de la GDT al cual hay que hacer el jmp far. Como vimos en el ejercicio 1, los descriptores de las tareas quedaron de la siguiente forma:

05 / 0x28	TSS Tarea 1
06 / 0x30	TSS Tarea 2
07 / 0x38	TSS Tarea 3
08 / 0x40	TSS Tarea 4
09 / 0x48	TSS Tarea 5

Como tenemos en AX el índice de la tarea a la cual tenemos que saltar, hacemos lo siguiente: Primero incrementamos el valor, ya que lo tenemos como índice, y lo vamos

```
inc ax
shl ax, 3
add ax, 0x20
mov [selector], ax
jmp far [offset]
```

a utilizar como si fuese el número de la tarea. Luego lo multiplicamos por 8, que nos da el índice en la GDT, y le sumamos el offset desde donde empiezan las tareas.

interrupción de teclado (letras P y R)

Para agregar la funcionalidad de pausar y reanudar, simplemente tuvimos que agregar las teclas a la interrupción de teclado, y agregamos una variable al scheduler que dice si en el próximo tick de reloj debe ejecutarse la tarea correspondiente o no (pausar). Esta variable es inicializada en falso, y cuando la tecla P es presionada, se cambia a 1, entonces en la próxima interrupción de reloj, el sistema es puesto en pausa. Si en cualquier momento la tecla R es presionada, esta variable es seteada en 0, y en la próxima interrupción de reloj, ”Hay que pausar?"va a dar negativo.

9.3. Backtrace

Justo antes de que una tarea sea desalojada, se llama a la función en ASM que muestra el backtrace, registros, etc. Esta función primero revisa el stack (a lo sumo 6 valores o el final de la pila, lo que suceda antes), leyendo valores, convirtiéndolos a hexa con la función provista por la cátedra, y luego los imprime en la pantalla en la posición correspondiente. Luego lee los selectores de segmento y los registros (teniendo cuidado con EBX, ya que es utilizado por la función de la cátedra para convertir a hexa), y los convierte a hexa y los imprime, y finalmente hace lo mismo con los EFLAGS, recuperándolos de la pila.

10. Conclusión