Organización del Computador II

Segundo Cuatrimestre de 2009

Departamento de Computación
Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires

Trabajo Practico N°3

Grupo RET

Integrante	LU	Correo electrónico
Gonzales Courtois Matias	453/07	curtu_infinito73@hotmail.com
Mancuso Emiliano	597/07	emiliano.mancuso@gmail.com
Mataloni Alejandro	706/07	amataloni@gmail.com

Reservado para la catedra

Instancia	Docente	Nota
Primera entrega		
Segunda entrega		

Índice

1. Ejercicio 1

1.1. Completar Tabla Descriptores Globales

Para realizar éste ejercicio no utilizamos los archivos gdt.h y gdt.c sino que completamos la tabla global de descriptores `GDT` en el `kernel.asm`. Ésta fue la solución al siguiente problema

```
relocation truncated to fit: R_386_16 against 'GDT_DESC'
```

Lo primero que tuvimos que hacer fue posicionarnos en la dirección de memoria en la cual debe estar la `GDT`. Como en el enunciado del TP nos pedía que la `GDT` esté en la dirección de memoria 0xE000, al final de nuestro archivo Kernel.asm y luego de la definición de las cosas que nos piden en ejercicios posteriores escribimos la siguiente linea:

```
TIMES 0xE000 - KORG - ($ - $$) db 0x00
```

`Instrucción para llenar de ceros hasta la dirección 0xE000.`
`TIMES:` Función del nasm que repite el código.

Como ya estabamos en la dirección correcta comenzamos a llenar la `GDT` de la siguiente manera:

```
ALIGN 0x10
gdt:
;descriptor nulo
        dd 0x00
        dd 0x00

; 0x8 (1) descriptor segmento de codigo
        dw 0xffff       ; segment limit
        dw 0x00         ; base 15
        db 0x00         ; base
        db 10011010b    ; p(1)|dpl(2)|s(1)|type(4)
        db 11001111b    ; g(1)|db(1)|l(1)|avl(1)|seglim(4)
        db 0x00         ; base 31:24

; 0x10(2) descriptor segmento de datos
        dw 0xffff       ; segment limit
        dw 0x00         ; base 15
        db 0x00         ; base
        db 10010010b    ; p(1)|dpl(2)|s(1)|type(4)
        db 11001111b    ; g(1)|db(1)|l(1)|avl(1)|seglim(4)
        db 0x00         ; base 31:24

; 0x18(3) descriptor segmento de video
        dw 0xffff       ; segment limit
        dw 0x8000       ; base 15
        db 0x0B         ; base
        db 10010010b    ; p(1)|dpl(2)|s(1)|type(4)
        db 11001111b    ; g(1)|db(1)|l(1)|avl(1)|seglim(4)
        db 0x00         ; base 31:24

        gdt_descriptor:
        dw gdt_end - gdt
        dd gdt
```

Primero definimos un segmento nulo, luego el segmento de codigo con los respectivos atributos que ocupa toda la memoria, luego el segmento de datos solapado con el de codigo, y por ulimo el segmento que direcciona a la memoria de video.

1.2. Pasar a modo protegido

Para pasar a modo protegido primero debemos setear algunas cosas. En principio debemos habilitar la `gate A20` la cual nos permite direccionar más de 1 Mb. Luego tenemos que setear la información (dirección y limite) de la GDT en el registro `gdtr`. Una vez realizados estos pasos debemos setear el bit que habilita la paginación en el registro `CR0`. Luego hacemos un `jump` con el descriptor de segmento de código.

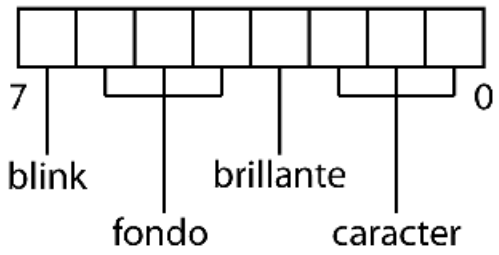
```
;Habilitar Gate 20
call enable_A20

;Cargar el registro GDTR
lgdt [gdt_descriptor]    ; cargamos la gdt

;Pasar a modo protegido
mov eax, cr0
or eax, 1
mov cr0, eax
jmp 0x08:modoprotegido
```

1.3. Pintar marco

Para pintar el marco, seteamos los registros de segmento con el número correspondiente (en nuestro caso 0x18) para utilizar el segmento de la memoria de video.
Una vez hecho esto utilizamos una serie de ciclos para escribir lo que nosotros queríamos, utilizando contadores y mensajes definidos.



`Cada elemento es de 2 bytes. Primero es el Modo, luego el caracter ASCII.`

2. Ejercicio 2

2.1. Inicializar directorios y tablas de páginas

Al igual que en el ejercicio `1.1`, para inicializar los directorios y tablas de páginas nos posicionamos en la dirección de memoria correspondiente según la tarea.

Tanto el directorio de páginas del `Pintor` (dirección 0xA000 a 0xAFFF) como el del `Traductor` (dirección 0xB000 a 0xBFFF) lo llenamos completamente (4kb), con descriptores de tablas de páginas caracterizados como:

supervisor, read/write, not present `(0x00000002)`

A su vez, para crear los descriptores de página en cada tabla, procedemos de la misma manera que los directorios de páginas.

Para el `Pintor` necesitamos hacer `identity mapping` de las direcciones:

- 0x0000 a 0x8FFF
- 0xE000 a 0xFFFF
- 0x15000 a 0x15FFF

```
;llenamos la primer tabla.
;pintorTable es la direccion de la primer tabla del pintor
mov     esi,pintorTable
mov     eax, 0
mov ecx,9
  ;Desde 0x0000 hasta 0x8FFF
llenarTablas:
        or      eax, 0x3
           ; supervisor, read/write, present
        mov     [esi], eax
        lea esi,[esi + 4]
           ;porque cada descriptor de tabla(eax) tiene 4 bytes
        lea eax,[eax + 4096]
           ;porque cada tabla ocupa 4k.
loop llenarTablas
```

`Identity mapping para el pintor, direcciones 0x0000 a 0x8FFF`

Además las páginas `0xB8000, 0x13000` deben ser mapeadas a las `0x10000, 0xB8000` respectivamente.

```
;pintorTable es la direccion de la primer tabla del pintor
mov     esi, pintorTable
mov     eax, 0xB8000
lea esi,[esi + 4 * 19]
or      eax, 0x3            ;supervisor, read/write, present
mov     [esi], eax
```

`Mapeo de la dirección 0x13000 a 0xB8000`

Para el `Traductor y Kernel` necesitamos hacer `identity mapping` de las direcciones:

- 0x0000 a 0x7FFF
- 0x9000 a 0x10FFF
- 0x16000 a 0x16FFF
- 0xA0000 a 0xBFFFF

Y la página `0x13000` debe ser mapeada a `0xB8000`.

Procedimos de la misma manera que con la tarea del `Pintor`.

2.2. Mostrar el nombre del grupo en la posición 2:2

```
mov ecx, mensajeEj_len
mov ah, 0x0c
mov esi, mensajeEj
xor edi, edi
        ; accedo a la memoria de video
add edi, (160) + 2 + 0x13000
.ciclo2:
        lodsb           ; lee desde ds:esi e incrementa esi en 1
        stosw           ; escribe en es:edi e incrementa edi en 2
loop .ciclo2

mensajeEj db "Orga 2   RET",0
mensajeEj_len equ $$-mensajeEj$
```

3. Ejercicio 3

3.1. Completar entradas necesarias en la IDT

Para este ejercicio utilizamos los archivos entregados por la materia:idt.c,idt.h,isr.h en los cuales llenamos la informacion para atender a todas las interrupciones. Tambien en el archivo isr.asm escribimos el codigo para atender a cada una de las interrupciones. Todas las interrupciones (menos la del timerTic y el teclado) nos muestran, en la parte superior derecha, el mensaje de que error se ha producido. Por otro lado el handler del timerTic lo que hace es dibujar el reloj en la parte inferior izquierda y el switcheo de tareas (solicitado en un ejercicio posterior). También inicializamos los pics de interrupciones con un codigo proporcionado por la materia, el cual mapea los pics a las direcciones de memoria correctas.
Para habilitar las interrupciones, y luego de llenar la `IDT` en `C`, se debe cargar el registro `IDTR`. Esto lo hacemos mediante la instruccion: `lidst [IDT_DESC]`, donde `IDT_DESC` es una variable en `C` la cual tiene la informacion de la `IDT`. Una vez realizados todos estos pasos podemos realizar la instrucción `sti`, la cual habilita las interrupciones.

Ejemplo de hanler de interrupción:

```
global _isr0
msgisr0: db 'EXCEPCION: Division por cero'
msgisr0_len equ $-msgisr0

_isr0:
        mov edx, msgisr0
        mov esi, msgisr0_len
        call IMPRIMIR_ERROR
        jmp $

IMPRIMIR_ERROR:
        pushad
        IMPRIMIR_TEXTO edx, esi, 0x0C, 0, 0, 0x13000
        popad
        ret
```

En este ejemplo vemos como se atienden las interrupciones: se define un mensaje, el cual se muestra por pantalla llamando a la función `IMPRIMIR_ERROR`. Luego se ejecuta la instrucción `jmp $` para colgar la ejecución del programa.

3.2. Next Clock

Para llamar a la función next_clock lo que tuvimos que hacer fue en el handler de interrupción del timerTic pusimos el siguiente código:

```
global _isr32
_isr32:
     cli
     pushad
     call next_clock

     mov al, 0x20
     out 0x20, al
     popad

     ...     codigo para el switch de tareas
     sti
     iret
```

4. Ejercicio 4

4.1. Completar la TSS correspondiente a las dos tareas

Un descriptor de segmento de estado de la tarea (`TSS: Task State Segment`) contiene información sobre la ubicación, el tamaño y el nivel de privilegio de un `TSS`. Un `TSS` es un segmento especial con formato fijo que contiene toda la información sobre el estado de una tarea y un campo de enlace para permitir tareas anidadas.
El campo de tipo se usa para indicar si la tarea está ocupada (tipo = 3), es decir, en una cadena de tareas activas, o si el `TSS` está disponible (tipo = 1).
El registro de tarea (`TR: Task Register`) contiene el selector que apunta al `TSS` actual dentro de la `GDT`.

Los campos del `TSS` estan divididos en dos categorías principales: campos dinámicos y campos estáticos.
Campos Dinámicos

- Registros de propósito general: EAX, EBX, ECX, EDX, ESP, EBP, ESI, EDI
- Selectores de segmento: ES, ES, SS, DS, FS, GS
- EFLAGS
- EIP
- Selector de segmento de la TSS de la tarea anterior

Campos Estáticos

- LDT
- CR3
- Stack pointer de cada nivel de privilegio
- Debug trap flag

A pesar de que solo contemos con las tareas de `Pintor` y `Traductor`, necesitamos crear una tercer TSS, nula, para hacer el primer cambio de tarea.

La tarea `Traductor` también es utilizada por el kernel, y su TSS la definimos de la siguiente manera:

```
; Inicializar TSS para el traductor
        mov edi, tsss               ; usamos la tss[1] porque la cero es para volver.
        add edi, 104                ; tamano de la TSS

        add edi, 4 ; avanzamos a esp0
        mov [esi], esp
        add edi, 4 ; avanzamos a grabar ess
        mov [esi], ss

        add edi, 20  ; avanzamos al CR3
        mov eax, cr3
        mov dword [edi], eax

        add edi, 4  ; avanzamos al EIP
        mov dword [edi], 0x9000 ; tarea de traductor

        add edi, 4 ; avanzamos a los eflags
        mov dword [edi], 0x202 ; si hay interrupciones, poner 202

        add edi, 20 ; avanzamos a ESP
        mov dword [edi], 0x17000

        add edi, 4      ; ebp
        mov dword[edi], 0x17000

        add edi, 12     ; ES
        mov word [edi], 0x10    ;descriptor de datos del kernel

        add edi, 4      ; CS
        mov word [edi], 0x8

        mov cl, 4
        .ciclo:
                add edi, 4      ; el resto de los registros de segmentos SS DS FS GS
                mov word [edi], 0x10
        loop .ciclo

; inicializacion finalizada...
```

La `TSS` del `Pintor` la definimos de forma similar a la anterior.

4.2. Completar en la GDT las entradas de la TSS

En el siguiente código, podemos ver la continuación de la `GDT`, donde se agregan los descriptores de `TSS` para cada una de las tareas.

```
        ..
        ..

; 0x20(3) descriptor de TSS para la tarea Nula
        dw 0x67                 ; segment limit
        dw 0x00                 ; base 15
        db 0x00                 ; base
        db 10001001b    ; p|dpl|0|type
        db 0x00                   ; base|g|0|0|avl(0)|seglim(0)
        db 0x00                 ; base 31:24

; 0x28(3) descriptor de TSS para la tarea Traductor
        dw 0x67                 ; segment limit
        dw 0x00                 ; base 15
        db 0x00                 ; base
        db 10001001b    ; p|dpl|0|type
        db 0x00                   ; base|g|0|0|avl(0)|seglim(0)
        db 0x00                 ; base 31:24

; 0x30(3) descriptor de TSS para la tarea Pintor
        dw 0x67                 ; segment limit
        dw 0x00                 ; base 15
        db 0x00                 ; base
        db 10001001b    ; p|dpl|0|type
        db 0x00                   ; base|g|0|0|avl(0)|seglim(0)
        db 0x00                 ; base 31:24
```

4.3. Task switching

Para hacer el intercambio de tareas, y luego de haber preparado todas las entradas de la `TSS`, lo que hicimos fue escribir el codigo necesario en el handler de la interrupción del timerTic para que haga el switch entre las dos tareas. Lo que hacemos es leer el registro `TR` y comparar con los datos de las tareas. Si se está ejecutando una saltamos a la otra y viceversa. Para saltar a la otra tarea simplemente hacemos un `jmp indiceTareaGDT:0` y con esto el procesador cuando va a leer en la `GDT` y se encuentra con un descriptor de `TSS` se da cuenta de que estamos realizando un cambio de tareas y se encarga de cambiar los contextos.

Handler del timerTic:

```
global _isr32
_isr32:
        cli
        mov al, 0x20
        out 0x20, al
        push eax
        str ax
        cmp ax, 0x28
        je switchPintor
        pop eax
        jmp 0x28:0
        sti
        iret

  switchPintor:
    pop eax
        jmp 0x30:0
        sti
        iret
```